Multithreaded Programming in C:
POSIX.1c

Stephen P. Spackman, Chris Provenzano and Nancy Gilman

DRAFT of December 6, 1995

Preface

the need

Most programs today are still written in sequential programming languages with a single flow of control. But times are changing: technical and economic forces continue to favor computers with multiple physical processors; user interfaces are becoming more sophisticated, less transactional, and more important; and while hardware becomes ever faster, client/server technology and wide area networking makes waiting for responses an increasingly dominant determinant of application performance. The common factor is an increasing need to deal effectively with asynchrony.

At the same time, C remains the preferred low-level programming language in many communities, and in concert with this, Unix and Unix emulation have become the most important tools for application portability offered by system vendors today. The ongoing POSIX standardization effort seeks to provide a stable basis for this practise. It may be a decade or more before a coherent alternative emerges.

The challenge for the C community in general and the POSIX committee in particular, then, has been to address the triple need for more responsive, better structured user interfaces; for the ability to use multiple processors as and when they are available; and for practical software techniques that are vastly more tolerant of variation in service response times; and to do all this within the confines of the now standard C-and-Unix paradigm.

POSIX threads—pthreads—are the result.

threads

The basic notion behind threads is very simple. The concept of multiple processes supported by a single processor is familiar to every Unix programmer: by switching attention rapidly between several programs, the illusion can be maintained that they all progress simultaneously. And of course, arranging for different programs to run simultaneously on separate processors is conceptually no trouble at all. The insight behind multithreading is that it is not substantially harder (indeed, it is often easier) to arrange for more than one 'thread' of sequential control to exist within a single copy of a program: rather than requiring each operation to complete before the next can begin, processing is switched between several execution sequences so that many different functions can be executing at the 'same' time, with each of these 'threads' having access to memory and the other resources of the program as usual.

The advantage is threefold. First, while one thread is 'blocked', awaiting a command or a response from the outside world, other threads can continue about their business within the shared environment. Second, since the programmer can introduce threads of control at will, the actual number of physical processors available has greatly diminished impact on software structure: if several processors are free, multiple threads will really be able to progress at one time, while if only a single processor is provided the same behavior is transparently simulated. Third, threads are an important software structuring tool, because when several tasks need to be taken care of at once, each can be assigned its own thread, executing code structured according to the needs of the particular subtask, without 'warping' one sub-problem to fit the needs of another, and without sacrificing common access to common data.

At the same time, programming in a new paradigm requires acquiring new habits, developing new skills, and finding new perspectives on algorithmic problem-solving. Using threads portably and correctly is not trivial, and using them effectively demands mastery of a number of interrelated facilities provided by the system.

standardization

Many different systems for multithreaded programming in C have been developed over the years. While these systems each have strengths, they share a common weakness: their very diversity inhibits software portability. Fortunately, in 1995, a *standard* for multithreaded C programming facilities was added to POSIX, the IEEE's Unix-like *P*ortable *O*perating *S*ystem *I*nterface. While vendors are not obliged to support this system, preliminary versions were already widespread before the final text of the standard was approved, and broad support can safely be anticipated. POSIX threads, pthreads, is already *the* system you need to know for the writing of portable multithreaded C software.

this volume

The purpose of this book is to provide the programmer with everything needed to make good use of multithreading, both in general and more particularly as specified by the POSIX standard, and to do so reliably, portably, effectively and in good style.

The book is organized so you can read it from cover to cover or use it as a quick reference. Code examples exist throughout, showing how you would use each thread function, along with comparisons to how similar code would be implemented in a non-threaded environment[[**sps: ???**]]. Short, but complete, code examples are used to demonstrate thread functions. Longer, real-world application, examples are given in later chapters. There are also chapters on code tuning, performance analysis, and debugging techniques.

[[(Description here of the gross structure of the book.)]]

In order to help you find what you need quickly and easily, and save unnecessary trips to the index, there is an extensive system of cross-references, both from uses of C identifiers to their descriptions in the text (written with subscripts: `pthread_create()`$_{33}$) and from one section of the book to another (with an arrow:$\Rightarrow^{4.3}_{33}$). Don't feel obliged to follow them, though, especially on a first reading: they are designed for the advanced reader trying to track down all aspects of a single issue, or to tell you where you need to turn if your memory needs jogging about some particular function or constant.

Other material that you should feel especially free to skip on a first reading (though we hope that some readers will find that it tells them just what they wanted to know!) is placed in footnotes with bracketed call-outs,[1] or gathered together in appendices to the various chapters: this includes discussions of a representative *implementation* of the pthreads standard, descriptions of earlier threads packages, and various technical issues, gee-whizzes, and pet peeves.

[1] Like that....

Contents

Part I

Multithreaded Programming

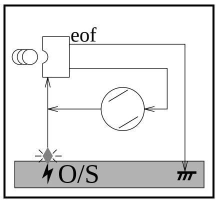
Chapter 1

Threads

[[Add more forward references at the chapter level.]]

Multithreading allows programs in imperative languages like C to do more than one thing at a time in a straightforward and intuitive way. Threads packages have existed for a long time, but now there is a standard threads interface, pthreads, for Unix-like and POSIX-compliant systems, making it an important new technique for portable software. Multithreaded programs can be simpler, clearer, more efficient and more responsive than their traditional counterparts. Multithreading is especially important for modern applications that focus on high-quality user interfaces and efficient network communication, and for fully exploiting contemporary multiprocessor hardware. In this chapter we discuss the *why* of multithreaded programming and give a first taste of the *how*.

1.1 The changing nature of software



The traditional view of software is that the typical program reads some input, processes it, and yields an output. Having done this, it may exit; or it may loop back for another piece of input and so keep going. In the picture at the left, the diamond-shaped processor runs around a loop carrying wafers one at a time from the input stream, through the buffing mill, to the output stream. Whether the wafers appear at predictable times or not is (of course) irrelevant: each wafer is processed as it appears, whenever it appears.

While a lot of the power and the popularity Unix enjoys among programmers is due to the effectiveness with which its 'pipelines' implement this model, such one input—one computation—one output programs are no longer so typical. Today's average computer user spends most of their time with software that maintains a graphical interface (taking input from keyboard and mouse, routing output to display and perhaps speakers), a second interface with the file system, simultaneously performing commands the user has requested and often background tasks as well (like reformatting documents, collecting external data, or maintaining various kinds of logs or checkpoints). Many exciting new computer applications of the last few years have involved communication, often with multiple remote computers, any one of which might be fast or slow depending on factors outside anyone's control.

Likewise in the area of scientific computation, which often continues the grand old tradition of batch processing, times have changed: large computers now almost invariably sport multiple processors. To exploit them, scientific programs must be written to do multiple things at the same time, even if these tasks all contribute to a single final result.

Nor is the middle ground between these trends empty: interactive visualization applications continue to gain ground in scientific computing, while more and more network server machines are being built to multiprocessor designs.

Why should these changes in the way software is used require a change in the way software is written?

In the case of the exploitation of multiprocessors, the answer is simple: the problem of breaking a program up for the different CPUs is in general still too hard to automate. If computations are very simple and homogeneous (like basic matrix manipulation) it is easy enough to write special libraries to exploit parallel hardware and then build user software in terms of these higher-level operations. But if the basic operations that need parallelizing are complex, irregular or unique to your application, you need to be able to program the parallelism yourself; and a notation is needed for that.

In the case of communication-oriented software (whether that communication is with remote computers or with humans) the problem is different. The difficulty here is that too many things are going on at one time, both externally and internally to the program, and it is not predictable *which* of these things will happen first. Of course, a basic methodology has long existed for dealing with this: it is the so-called **event loop** structure, in which your program constantly evaluates a single big loop that checks for all the different things that *might* happen, and then takes action on those that *did* happen on each iteration. That's simple as far as it goes: so long as the set of possible events stays more or less constant, so long as they are all reported in a compatible manner, so long as the complexity of the relationships between the inputs and the outputs is reasonable, and so long as timing is not critical, the approach should work—in essence we've just reduced the multiple input/multiple output case to the classical structure of a single logical stream of input **events** and a single stream of output **actions**.

What goes wrong with the event-loop methodology, of course, is that the whole structure of the application has to be twisted around to fit the loop itself—which is doubly odd, since the top-level event loop is often extraneous to the abstract specification of the problem being solved. Each path from input to output, considered in isolation, probably has a 'natural' logic of its own, with its own loops and calls and internal state and so on, but to fit the event loop model the code has to be structured so that all the relevant state is held in global storage between events, in case one incoming event and the next happen to be unrelated.

Even then things can go wrong: sometimes timing is important, and if one incoming event takes too long to process, the next may not be handled properly at all—especially if 'properly' means that the user does not have to sit staring at an hourglass icon while

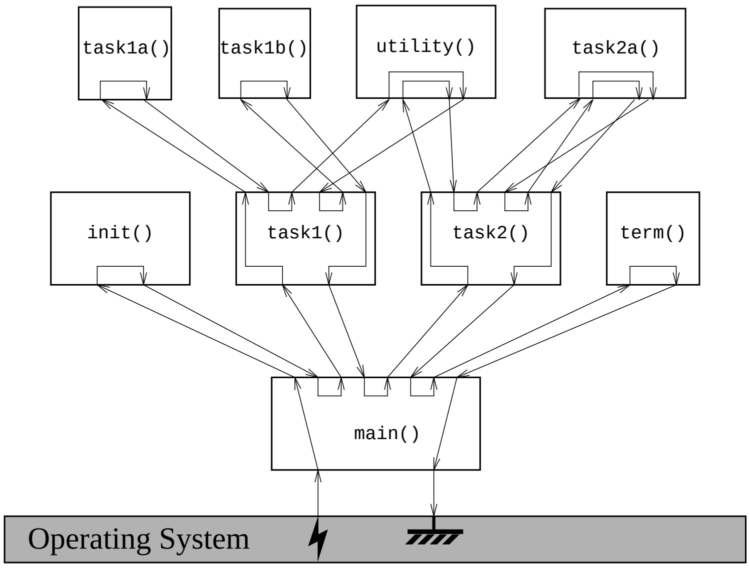


Figure 1.1: 'Single-threaded' execution

some internal part of the program is busy, or a robot arm should be told to stop before it collides with the wall.

How much better it would be if programs that keep track of many asynchronous things could be structured in the same way as programs that work with only one input stream: with functions and loops and variables that track the state of the external subproblem they concern, interacting only in the places where interaction is a real part of the program's mandate.

1.2 Multithreading

Multithreading does just this. Normally when you think about a C program executing, you imagine a processor working along evaluating each statement in turn. It can execute the various statements of a function while still in middle of the statement that called it, of course, but the calling statement is kept 'on hold' while it happens, with its state squirreled away on a stack maintained by the system. If you think of the processor as a needle passing through each statement and function of a program as it is executed, the thread that it draws then represents the history of the computation,

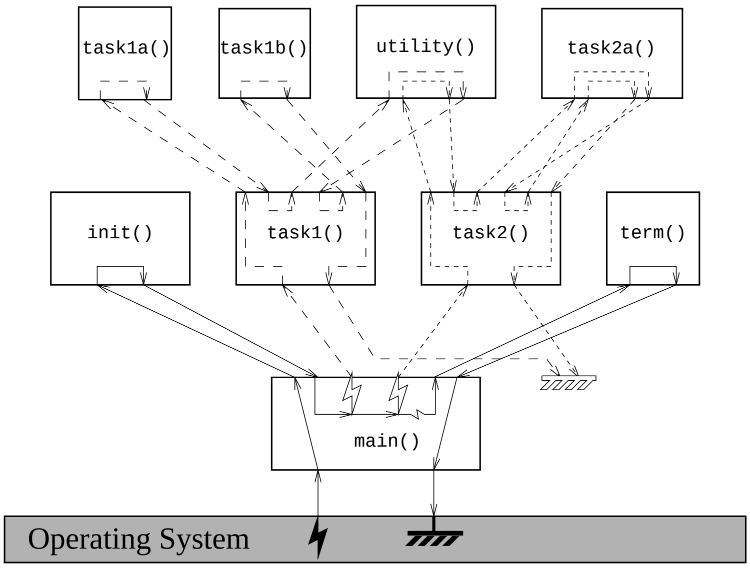


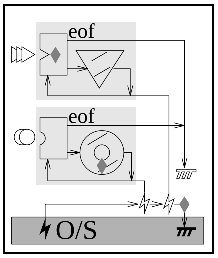
Figure 1.2: Multithreaded execution of a similar program

at least as regards the control decisions that were made. This imaginary thread, this unfolding history, is what is known as the program's **thread of control** (see figure 1.1).

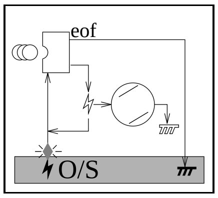
Now imagine what would happen if somewhere you could get a second processor, a second needle, to draw another thread of control through your code. And unlike the traditional Unix model of multiple 'processes' executing different programs, or different *copies* of a single program, imagine that this new processor is working on the *same* copy of the *same* program at the *same* time. At any given moment there are now *two* statements that are being executed; each processor has its own stack, so there are two *separate* sets of function calls that are temporarily 'on hold', with no confusion between them; and (since C stores local variables as well as control information on the stack) there are separate sets of local state. Because they're both working on the same program, though, these two processors see the same global variables and can communicate with each other when they need to—even when they are working on very different parts of the program text.

This is virtually what multithreading gives you: the ability to have more than one part of a single program being executed at the same time. (We'll come back to that word, *virtually,* in a minute.) In the case of pthreads, the POSIX-specified multithreading system that we'll be looking at most closely, you can effectively add a new logical processor to your program at the point that you call a function[1]: the new processor executes the function you called, letting the calling thread go immediately ahead with the next sequential statement, not waiting for the function to return (see figure 1.2).

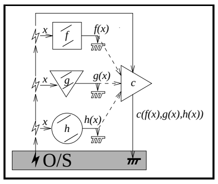
What does this buy you? Here are some examples:



The ability to have each subtask that a program pursues be coded according to its natural logic, without (much) regard for what other subproblems must be solved at the same time. In the picture at left, instead of needing a top-level event-loop that recognizes both circular and triangular wafers and ships each one off to the appropriate mill for processing, the application has been structured with an independent loop (in a grey box) for each sub-task, each one of which is structured as if it were wholly independent. These loops are serviced by threads that are 'sparked off' by the primal thread, execute in parallel, and complete in their own good time.



The ability to start up a thread of control when a new task comes along, reducing the extent to which old tasks and new tasks interfere with each other's performance. In this picture the primal thread just picks up the input wafers and hands them off to newly created 'worker' threads: it is the new threads that do the actual processing. Now the primal thread need not risk missing an incoming wafer because the processing of a preceding wafer is going slowly; uncompleted processing and the acquisition of new input continue in parallel, with each thread exiting when its own job is done. Because all the threads are part of the same process, it may not even be necessary to spend time copying the contents of wafers as they are handed off. (Your threads package may help even further by providing 'priorities' so you can control how resources are allocated.) Best of all, while one thread is waiting for its next input, the others can keep running at full speed, reducing the amount of computer time that gets wasted.



The ability to exploit multiple processors to pursue multiple subproblems simultaneously *so that the the computation as a whole gets finished sooner.* Here the primal thread has sparked off three worker threads to compute $f(x)$, $g(x)$ and $h(x)$ *at the same time,* perhaps on different physical processors. It then waits for the three results to arrive so it can combine them with the function c. If there really are enough processors to do this all at once, the elapsed time will depend on the slowest subproblem ($|c| + \max\{|f|, |g|, |h|\}$) instead of on the sum of all subproblems ($|c| + |f| + |g| + |h|$) as is normally the case.

The true beauty of the virtual processor based multithreaded paradigm, though, is that while it can *use* multiple processors to speed up a computation, it doesn't *need* any more than one; having more only accelerates the computation. The other

[1] Ok, it's not *quite* that elegant.

benefits—the ones to do with doing several things at once, with finding useful work to do even while subcomputations are waiting for external events, and with disentangling the control structure of your software—apply even when there's really only CPU. Everything else can be—and is—faked up by system software in the threads package you use.

1.3 Conflicts between threads

It can't, of course, be that simple. What goes wrong?

The main difficulty is in synchronizing access to shared resources. As we've already noted, since all threads actually execute within the same copy of the program, they all have access to the same global variables (and anything else they can reach by following shared pointers, for example). What happens if two of them try to make use of the same variable—or other resource—*at the same moment?* If all they want to do is look at it, there's no real problem; they both see the result you'd expect. But if one of them changes it, something unpleasant is likely to occur: the other thread may end up seeing a half-updated version of the value, with potentially catastrophic consequences! Something is needed to mediate access to shared resources so that this doesn't happen.

There are a number of such somethings that are possible. The simplest method is just to have global **critical sections** with the property that when one thread enters a critical section all the other threads are immediately stopped; it can then do whatever it needs, safe from external interference until it leaves the section. But this can be immensely inefficient, especially when there are really multiple processors! After all, though another thread *might* be about to interfere with the critical operation on shared data, the odds (in most programs) that any given thread *will* do so in any single instance are quite slim. It would be much better to find a method that only blocks those threads that are really in danger of interfering with each other—to use a **finer grain** of locking.

The basic idea behind most fine-grained resource locking schemes is to make sure that only one thread at a time can use a given shared resource by associating some kind of 'token' (or **lock**) with it that a thread must **hold** before using the resource; other threads have to wait until the current holder of the token gives it up. (Such tokens themselves are often implemented with further shared variables—but the chicken-and-egg problem is resolved by having the implementation employ extraordinary measures for the locks, specially tailored to the needs and abilities of the hardware being used.[2]

[2]There's also an interesting class of so-called '**optimistic**' techniques in which a thread that wants to update a shared variable is implicitly given a *copy* of the object to work on; when it is finished making changes, the 'real' version of the variable is 'instantaneously' updated—providing no other thread has modified it in the meantime. Otherwise (depending on the system) either the thread is alerted to the problem and left to recover as it will, or the entire update operation is invisibly restarted from the point where the copy was made. There are even versions of this idea where the independent updates of various threads are *merged* to compute the correct overall effect directly—though this

semaphores The most basic method of synchronizing access to individual shared resources is perhaps the **semaphore**. This is like a numeric variable that can (through special operations) be incremented and decremented, except that instead of going negative, 'wrapping around' or raising an error when you try to decrement past zero, the offending thread of control is just **blocked**—made to sleep—until the value of the semaphore is greater than zero and the decrement becomes legal. (The increment and decrement operations on semaphores were traditionally known as V and P, for 'signal' and 'wait'[3]) In a single threaded program this would be nonsense—you'd wait forever—but because only the offending thread waits, any other thread can easily increment the semaphore and satisfy the condition. Semaphores trivially meet our needs: we use them as Boolean variables (in the typical C encoding of `0` for false, `1` for true) that specify when a shared resource is available. If you want to use the resource, you mark it as *un*available by decrementing the semaphore, do with it what you need to do, then increment the semaphore again when you are finished. Assuming the resource was initially free, the semaphore will be in the 'false' state while it is being used, and true otherwise. If you try to acquire the resource while it is being used, though, your thread will automatically block trying to decrement zero until the thread using it is done. (If you use values other than zero and one, of course, you can encode more complex information like how many slots in a buffer are free. But this may be less useful than you expect; generally you also need to know which ones are which, and that information must also be protected from access conflicts.)[4] Semaphores have been used in many systems, including some Unices, and are an optional part of POSIX—though POSIX semaphores are vastly more complex than the basic version we've just described.

object locks Semaphores are all very well, but a trifle chaotic in practise: since they can be incremented and decremented by any thread at any time, they encourage a rather unstructured style and can be hard to debug. Another approach that has been tried in object-oriented languages is to implicitly associate locks with objects: arrange that any thread executing a method on a given object waits until all other methods invoked on the same object complete. That is, the system automatically **serializes** all operations on a given object. This is nice for the programmer because it is automatic, reducing the amount of extra code that must be written, and suitably fine-grained because—at some cost in time and space—a separate lock can fairly easily be associated with each shared data object. Unfortunately, it has the usual problem of object-oriented techniques: *sometimes* the object is a good organizational focus for the application, but just as often it isn't: you see this as soon as you need to compute a function of *two* shared objects—as we shall see later, it is very important in this

---

depends on special properties of the shared data structure if it is to work.

If there are a *lot* of threads trying to use this resource at a single time, optimistic approaches may perform very poorly. But if contention is low, the whole sequence can often be made to complete without slowing down for any kind of synchronization at all—the extra copies and so on can be made almost free by various arcane hardware tricks.

[3] In Dutch, the language known by the Dutch as 'the secret language of computer science'.

[4] This is all usually explained the other way round: Boolean-valued '**binary semaphores**' are generally introduced first and then generalized to many-valued '**counting semaphores**'. But building Booleans out of integral types is how C does things, and this is a book about C.

situation to have control of the *order* in which multiple locks are acquired, and it may not match the nesting order of method invocations. And, of course, it just won't work at all for non-object-oriented languages. For one reason or another, it looks as if *explicit* synchronization facilities are necessary.

monitors Another solution is the **monitor**, which associates exclusion with areas in the program text (perhaps module-like syntactic constructs containing several functions; or perhaps named collections of code blocks). This is the 'structured' counterpart to the 'object-oriented' object locks: mutual exclusion is again automatic (though more explicitly visible), but it prevents two threads from entering the same function or family of functions at once. This may be a little more versatile than object locks—the programmer receives more control over how the monitor relates to the data it protects, which in turn improves control over locking order and allows nontrivial relationships between logical structure and physical representation (which object oriented programming tends to assume are one and the same thing). By making a global variable local to a monitor you can even ensure statically that the locking protocol is never violated for it. This comes, however, at the price of a loss in grain: since (classical) monitors are associated with code and not data, a thread entering a monitor will keep all other threads out, even if they want to manipulate a different *instance* of a given structure than the one the monitor's current owner is working on. Furthermore, there is no obvious way to enforce the locking protocol for dynamic data structures. The ability to associate locks directly with data structures—not just named variables—seems essential.

One other important thing the monitor model *does* provide—in common with the *mutexes* we look at next—is the *condition variable*; we'll get back to these in the next section.

mutexes How can we combine the strengths of monitors and object locks? If we have the freedom to manipulate the source language, a reasonable solution might be to treat locks like unions in type-safe languages[5]. That is, we can introduce a type-building construct into the language that has syntax like C's `struct`, and then only allow programs to access its internal fields from inside a special statement that locks the whole substructure.

Of course, if we are stuck with an existing programming language—and when working with C this is certainly the case[6]—we probably can't add this kind of notation. But we may be able to provide the *semantics,* wrapped up in new data structures and procedure calls. What do we need? A data type ('mutex', for ***mut*ual *ex*clusion variable**) that represents the locks themselves (it would be nice if we could retain the explicit association with the controlled data, but we can't; we'll have to rely on careful programming conventions for that); an operation to lock a mutex; and another one to unlock it when we are done (again, we'd like that to be automatic but we're forced to rely on the programmer).

[5]Unlike C. C is as type-safe as a cabbage.

[6]Note that the C preprocessor is not really powerful enough to simulate C control or data constructs—it is blind to the use of braces.

The astute reader will notice an irony here: we've come full circle. The minimal version of mutexes we've described (which is in fact the system used by many 'light-weight' threads packages, and is the standard technique used with POSIX threads) is actually a *restriction* of the semaphore model we started from: a mutex is just like a binary (Boolean) semaphore *except* that it may only be incremented—unlocked—by the last thread to decrement it, its current '**owner**'. That's a significant piece of functionality we've given up; we'll need condition variables—we're coming to these, remember—to get it back. We'll see later on in the book, though, that the fact that a mutex has an identifiable owner (which semaphores do not, since any thread can increment or decrement a semaphore at any time) is very important, and opens the way to new generalizations of the mutex, some of which are very useful in practical applications.

1.4 Communication between threads

We've just discussed how to prevent threads from interfering with each other when data are shared, but this only goes part way towards allowing them to communicate. Given any of the above synchronization techniques we can pass information from one thread to another without it getting scrambled in transit, but that won't necessarily tell a thread when it has new data waiting. It's easy enough to keep looking until something changes, of course, but that's inefficient: it uses processor cycles for checking if a calculation is done that could have been used on the calculation itself! Once again there are several good solutions.

message queues One method is to provide 'message queues' directly. These are a little bit like traditional Unix pipes, in that threads can write messages to them and read messages from them; if there's no message waiting when you read, you block until one becomes available. Once again, these appear in many systems, including (though only as an option) POSIX.

The nice thing about message queues is that they work even if the communicating threads don't share any memory: the system can arrange to move the content of the messages as necessary. Then again, given that the threads within a program *do* share memory, that may be overkill. What about solutions that provide *only* the synchronizing function, and leave the rest to the application?

semaphores Another, very simple, method of signaling is to use use semaphores—the same semaphores we just discussed for locking—in reverse. Now we interpret the semaphore value as meaning 'a message is waiting for you'. In this application we initialize the semaphores to *false*—no waiting messages—and a thread that wishes to wait for a communication just tries to decrement it. This won't work, of course; the thread will go to sleep until some other thread writes a message to some shared variable and increments the count, setting the semaphore true. The waiting thread now wakes up (setting the semaphore back to false in the process—that was what it was waiting to do, remember) and reads the waiting message. This also works properly if the message sender hits the synchronization point first, of course: it writes the message, increments the semaphore from false to true, and proceeds. When the receiver tries to decrement the semaphore it now succeeds immediately, and knows it can process the message. In this case it doesn't have to wait for anything.

But beware: with this approach a second semaphore, used in the 'forwards' direction, may also be needed to ensure that two threads don't inadvertently write to the shared buffer at once!

condition variables What's a good way of doing synchronization if we are working with monitors or mutexes? The usual solution (and again the one in POSIX threads) is the **condition variable**. A condition variable is akin to a semaphore used in its signaling mode, as just described, with one very important modification to the operations that manipulate it. The crucial insight for understanding condition variables is that when a thread waits for a condition to be signaled, it does so because it is waiting for some other thread or threads to finish operating on a shared resource. That resource must be protected by a mutex; and that mutex must be have been held by the waiting thread when it decided that waiting was necessary in the first place (since in general it has no way of knowing that the thing it is about to wait for hasn't already happened; to wait for something *after* it has happened is always stupid and typically catastrophic—you could wait for a *long time*). Furthermore, whatever thread is going to signal the condition and wake up the waiter is *also* going to need the mutex so that it can update the shared structure. It would be possible just to have the waiting thread release the mutex and then start waiting on the condition, but this would have the disadvantage that the programmer would have to be careful only to signal the condition when there really was a thread waiting for it (otherwise the count on the condition-semaphore would creep up and it would stop working properly). Instead, condition variables *don't* keep any state—they perform only the blocking and awakening functions of semaphores (which is the part we wanted)—and the operations of waiting for the condition variable and releasing the associated mutex are done **atomically**, *i.e.*, as an indivisible operation, thereby preventing any signals getting lost between the point at which the mutex is unlocked and the time the thread really does start waiting. Correspondingly, when the thread is woken up, it is given the mutex back—since we know that it will need it in order to receive its message.[7]

1.5 Scheduling work

All of this talk of how to synchronize multiple threads of control is fine, but what if there aren't really as many processors in your computer as you ask for threads in your program? What really runs? Or what if there are enough processors but there is some other resource—such as a shared variable—that more than one thread of control needs at the same time—which one gets it? Questions like these are resolved by the **scheduler**. Let's return now, as promised, to the 'virtual' availability of processors.

[7] There is some variation between systems as to whether the signaling thread or the awakening thread gets to go first when a condition is signaled.

1.5.1 Virtual CPUs

Virtualization is, of course, the technical term for *Smoke* and *Mirrors.* The Smoke is the collection of hardware features that prevent your software from ever finding out what is *really* going on; the Mirrors re-route your attempts to see through the smoke to code that returns answers more useful and interesting than reality.[8] Familiar examples of this idea are *virtual memory,* in which the hardware fakes up memory references so as to switch addresses around or make disk-resident or remote or computed data appear as if they were in main memory, and *processes,* which are lifelike cardboard replicas of pretty much the whole computer. Since the main components of a computer are the memory and the CPU, and since the memory can be virtualized either as part of a whole virtual computer (in a process) or alone (as in virtual memory), you might ask whether the CPU can be virtualized separately, too. In fact, it can: and that is the basic idea behind the implementation of threads on a machine where processors are in short supply.

The amount of information stored in a conventional CPU is fairly small (at least if we restrict our attention to explicitly-managed state; for present purposes we can treat caches and so on as part of the *memory* system, not the CPU): the contents of a few registers, and not much else. These registers determine what the computer is doing at any given moment; even the instructions that it executes are found by looking up the target of the instruction pointer, itself a register. If we change the contents of the registers, instruction pointer and stack pointer included, we completely change what the CPU is doing. And if we save the old values somewhere (typically in a data structure called a 'thread descriptor'), we can switch back to the old task whenever we need. Because there are relatively few registers to save and restore, the switch can be made quite fast (unlike context switches between processes, where changing the memory mappings turns out to be costly—suddenly we *do* have to worry about managing the various caches and so forth).

But how and when do the registers get switched? Obviously if we want it to appear as if there are multiple processors, it has to happen quite often,[9] but multithreaded code (unlike code written with **coroutines**) doesn't have explicit calls to functions that would do this.

Part of the answer is just that appearances can be deceiving (more smoke, more mirrors): threads systems often add context switching code to a lot of innocuous-seeming system functions. The idea here is that if a thread is going to have to wait for something—if the system function 'blocks'—you might as well switch to another thread that has work to do in the meantime, and keep the CPU busy. Another

---

[8] Of course this is *always* possible with an interpreter—that's a consequence of the theoretical notion of *Turing equivalence* (which basically states that all computers beyond a certain minimal power are equally good for computing—so long as we ignore questions of time and memory space!). The real trick to virtualization, and the thing that matters if we *do* care about performance, is that it be fast—ideally, it should be very, very nearly as fast as the real thing. That's possible; it just relies on finding out what the usual cases are and optimizing the hell out of them, often in hardware—while making sure that the other cases can still be handled in software.

[9] 'Often' here is ten, a hundred or even a thousand times a second. With special hardware (such as multiple sets of CPU registers that can be exchanged with each other instantly) machines have been built that can switch contexts on every clock cycle, if necessary—that's large numbers of megahertz. At rates like these, the question of whether there is one processor or many starts to blur!

important trick is 'time-slicing'—the system periodically interrupts your program and forces it to execute a fixed piece of code (it probably does this anyway to keep the system clock up to date). After a certain number of ticks—after a thread has had its allocated slice of time—the system software flips the register contents around and effectively returns control to a different thread than it came from. That way, even if a thread doesn't explicitly call *any* system functions, it's still possible to wrest control from it and let another thread have its turn.

(Interestingly enough, this technique amounts to re-introducing the *event loop,* but at a level that is invisible to the programmer. There's now a loop hidden deep within the thread system itself, whose input events are clock ticks and changes in the availability of mutexes, and whose output actions are allowing this thread or that thread to run for a while. The programmer doesn't see it because the programmer's view is defined by the processor state, and we've arranged for each thread to have a local state of its own.)

1.5.2 Priorities

Not everything that a computer does is equally critical. Some things have to be done immediately, some things have to be done eventually, and some things are nice but don't *have* to be done at all (but why let cycles go to waste if the computer isn't busy anyway?). By associating **priorities** with threads and using them to decide, when there's a choice, which thread should go next, you can get better performance than by choosing them at random, say, or by going in strict rotation.

Priorities can be used in many places, not just when selecting which of the running threads should go next; wherever threads contend for a resource (like a lock or the chance to use some kernel facility) there is potentially a question of which thread should go first. It might be good if these were *always* decided by priority, but to paraphrase the sage this is the realest of all possible worlds: sometimes it won't work out that way.

As we shall see later, even the theory of priority scheduling is not as intuitive as you might hope; in particular, if mutual exclusion is to continue to do its job you can't just say that higher priority threads *always* take precedence over lower priority ones, and it takes some reworking of the synchronization primitives to ensure that the priority system doesn't get messed up completely. But we don't need to worry about this for several chapters yet.

1.6 Safety in a threaded world

When doing multithreaded programming there are a number of new things that you need to look out for. These come in two rough categories: stepping on your own feet, and the fact that *someone is watching you.*

As to stepping on your own feet, we've already seen one problem: when there are multiple threads around, global variables start to look like not such a brilliant idea. The fact is that, like it or not, global variables communicate between threads, and as such *must* be protected by some kind of lock if they are to be changed at all; when communication is not the *purpose* of the variable, that's a lot of effort to save the typing of one more parameter.[10] The opposite thing can also go wrong: too *much* synchronization—or rather, synchronization with its lines crossed—can result in **deadlock**. Consider the plight of four cautious drivers at an intersection with four-way stop signs, or of two threads each of which holds one lock but needs the other. You get the picture: the threads just sit there, while the computer rusts!

The other class of problems has to do with the bad habits we all get into when no one is looking: we don't check the return codes, we don't close the files, we don't free the memory, because the system will clean up after us when the program exits and who's to know anyway? When working with threads, it's a different story. *Those other threads are watching you:* other threads are going to need the resources your thread is finished with, and because the program is still running even after your thread is finished, nobody else is going to do your cleanup for you—not, at least, until *all* the threads are done. This is especially important because threads make it easy to write programs that run for a long time, and over an extended period even very small resource leaks add up.

All in all, programming with threads requires just a little more care than most of us are used to (though most of the things that you need to pay attention to are things the stylists and academics have been talking about for years—threads just give us a practical reason to care). Fortunately, as is often the case in programming, software itself can be made to help: threads packages may provide some added assistance for the kind of bookkeeping that multithreaded programming makes so necessary. We'll see a number of examples, most especially in chapter 6.

1.7 Summary

There are two main motivations for multithreaded programming. One is simplify the exploitation of multiprocessor hardware. The other is to make asynchronously occurring events appear synchronous to software—by allowing subcomputations to progress at the same rate and in a parallel manner to the external phenomena they interact with. When software can be structured like the hardware it can be made more efficient; when software can be structured like the systems it models it can be made clearer, simpler, faster and easier to maintain.[11]

Threads, like any other programming technique, are not a panacea: working with multiple threads involves some run-time overhead in creating and manipulating the threads themselves and in synchronizing access to shared resources; and it requires

[10] It seems to be a popular conception that global variables are much faster than parameters. For a number of reasons having to do with caching, memory addressing, register allocation, instruction processing and the visibility rules for variables this is less true than you might think—even without synchronization overhead. Try some benchmarks!

[11] If you'll pardon a philosophical aside, you might notice that these two motivations are very closely allied: both the *host* environment, the hardware, and the *task* environment, the application, are normally heavily parallel and becoming more so. Software, mediating between the two, needs to follow suit if the mapping costs between various levels are to be kept under control. At least, this is true for low-level imperative languages like C, which provide both a sequential abstract model and a machine-oriented implementation.

more attention to detail in some places than classical methods. Nonetheless, for a wide and growing class of applications, where the user—or a hardware device, or a remote computer—does not wish to sit staring at an hourglass while the computer is busy doing something else, it is the technique of choice.

[[Bibliography: semaphores, monitors, threads, ...?]]

Chapter 2

The World of Threads

[[Chapter not yet written]]

2.1 Early history

[[The concept of threads predates UNIX. Early multiuser systems had one master program which allocated one thread per terminal. Users could login and attach one program to the master program. When the attached program was done another could be attached in it's place. These early threads developed into what UNIX calles processes, and the master program developed into the kernel.]]

2.2 Ancestors of pthreads

2.2.1 MACH Cthreads

2.2.2 SunOS LWP

2.2.3 DCE threads (OSF/1, HP-UX, Windows NT, ...)

2.3 Pthreads and the posix standard

2.3.1 Implementations

Capthreads (NetBSD, FreeBSD, Linux, ...)

2.4 Standard C

2.5 Related posix concepts

2.6 Other contemporary thread systems

2.6.1 GATech Cthreads

2.6.2 OS/2

2.6.3 Plan9

[[Doesn't have threads as a separate concept, but....]]

Part II

POSIX Threads

Chapter 3

Threads in Posix

In this chapter we begin our exploration of the POSIX multithreaded programming facility, **pthreads**.

3.1 The Posix standard

The bulk of this book concerns multithreaded programming under POSIX. The most important thing to bear in mind when reading is that pthreads (and POSIX of which it forms a small part) is a **standard** and not itself an actual operating system (or other piece of software). Many very different implementations of the POSIX specification exist—they arise whenever vendors wish to allow standard Unix-based software to run on their systems—and many of them will provide implementations of pthreads.

The idea behind POSIX is to raise the level of the basic facilities available from a "Unix-like" operating system to the point that *non-trivial* software can interact with the host environment in a *portable* way. At the same time, as little as possible is said about *how* this is to be accomplished, to allow the greatest number of systems to achieve POSIX compatibility, and (as much as possible) to avoid stifling further research and development.

Indeed, POSIX itself is a moving target: as new ideas arise and gain currency in the operating systems community, POSIX may adopt and standardize them. In the current round of revisions and extensions, real-time programming facilities and multithreading—as presented in this book—have been added. Later versions of the standard will probably add further features.

At the same time, Unix itself is a legacy system: it is more than twenty years old, and it shows its age. While extensions—like pthreads—constantly provide new ways of doing things, it takes a long time for the effects of old decisions to fade; much of what POSIX standardizes is not, by contemporary standards, very pretty.

One such piece of heritage is that the main programming language of POSIX is C, a language that lacks features now considered essential by many researchers, including type safety and parallel processing.[1] It is to address the latter shortfall that pthreads

[1]Indeed, many researchers considered such features indispensable long *before* the design of C.

was introduced; as we shall see, the result of doing this—without, it should be noted, modification to the language—is also not entirely beautiful. But, like the rest of POSIX, it works.

3.1.1 About posix compliance

POSIX (for "*P*ortable *O*perating *S*ystem *I*nterface"[2] [ISO/IEC 9945-1: 199x]) is typical of a class of standards that are developed by committees of representatives from industry and academia, both producers and consumers of the systems being standardized. In these the standards organizations themselves only oversee the development, dictate the form, and arrange for publication of the finished standards documents. POSIX thus represents a compromise that all parties officially involved feel they can live with, for a common ground that suppliers then offer to their customers as 'the standard.' But these suppliers are still in competition; they would not support a standard that was so detailed in its prescriptions that their proprietary techniques and marketable innovations could not co-exist with a claim of compliance.

POSIX in particular is a fairly loose specification, drawing together a number of threads of 'Unix compatibility' that have existed in the industry for many years. Much of it attempts to set down only a minimum common basis for the support of portable software, which vendors should provide and on which programmers can rely. The threads package itself is actually a new development, designed especially for the standard (albeit on the basis of much earlier work); but it still leaves implementors a lot of latitude in choosing how to make it function.

Compliance with the POSIX standard, furthermore, is not mandatory (unless indirectly through the stipulations of a contract you might have with a supplier, for example). Finally, pthreads is only an option within POSIX, and it contains several sub-options of its own.

3.1.2 Implications for the programmer

What this all means to you, the programmer, is:

- Just because it is described in this book does not mean that your POSIX-compliant system supports it. You must pay attention to the options your system supports. Fortunately, preprocessor symbols are provided to help you write software that can configure itself as needed.

- Just because it is *not* described in this book does not mean that it can't be done. Material in your system's documentation that extends the standard should also work—but it won't be portable. In fact, when in the text we say 'you must not

At that time the question was whether the additional overhead of such advanced capabilities could be tolerated in a practical and widely-used programming language. From today's perspective this seems almost backwards: while better implementation techniques constantly reduce the overhead of high-level features, the cost of supporting the 'efficient' *unsafe* features of C-like languages is actually increasing as hardware evolves.

[2]The final -*X* has been used widely in the names of Unix-like operating systems, and should perhaps be seen as a linguistic joke.

do such-and-such a thing' this does *not* imply that it necessarily won't work—but it does mean you will be violating a restriction *in the standard* and quite likely compromising your program's robustness and/or portability. Once again the real question is whether it violates a restriction in the documentation (or code...) of the system(s) it runs on.

- This book is no substitute for your own system's documentation. The manpages that come with your implementation will (with luck) describe
  - the extensions your vendor provides (which you may be able to exploit if portability is not a big concern),
  - the restrictions it imposes (which in real life may contradict guarantees that the standard tries to make),
  - known bugs in the release, and
  - important performance-related information that falls outside the scope of the standard itself (though in fact the standard does prescribe how some of this should be measured and described).

  Of course, even if you follow the documentation to the letter, and your program is otherwise perfect, it still may not work. But at least in that case we know whose bug it is, and who to ask to fix it!

What the standard actually provides for the programmer of multithreaded systems is an optional (but hopefully widely available) package of functions and types that supply a basic but flexible footing for the development of software. There are objects, **pthreads**, that represent separate flows of control within a single process; **mutual exclusion locks** and **condition variables** providing safe synchronization between them; facilities for threads to manage private data; and a full specification of how the *other* parts of the POSIX application program interface interact in the face of multiple threads—along with replacements for the few standard functions that are inherently unsuited for use with multithreaded software.

The calling conventions for the thread-related features have been made very systematic (even at the expense of breaking with tradition and the conventions of other parts of the POSIX specification), and the major new objects introduced—the threads and the synchronization objects—have been given an extensible initialization protocol that allows vendors to add arbitrary new options in a simple and orthogonal manner.

In the next section we introduce the POSIX thread itself.

3.2 The posix thread

What is a POSIX thread, a **pthread**, and how does it fit in with the rest of the operating system?

First of all, pthreads provides an alternative to using multiple processes for applications needing multiple threads of control. Threads are light-weight, provide relatively easy sharing of data, and work smoothly in the case where all the required

	per host	**per process**	**per thread**
identifier	IP #, name	process ID	thread ID
family	domain	process group session	process
authority		(e)uid, (e)gid, ...	
name spaces	file tree mountpoints IP ports	memory map environment comm. handles file handles	frame pointer thread-specific data
storage	RAM, disk	virtual memory	registers, stack
state		fs root, pwd file creation mask	pc, sp
exceptions	interrupt hdlrs exception hdlrs int. enable	asynch sig hdlrs synch sig hdlrs	`errno` signal mask
mode	single/multiuser	process state	thread state
termination		exit handlers	term. handlers
scheduling		priority, policy	priority, policy
arguments	rc files	process args	initial argument
software	kernel	program	entry point
context		locale	

every host has at least one process;
every process has at least one thread

Figure 3.1: Loci of State

logic falls, conceptually, within a single 'program'. With pthreads, POSIX allows each process to contain tens, perhaps as many as thousands, of threads of control. Like processes themselves, these threads are made independent in their blocking behavior: one thread is not forced to wait indefinitely because another thread (even one within the same process) is delayed inside a system call—unless, of course, its purpose is such synchronization.

Second, pthreads are used from within otherwise conventional programs; indeed, while bindings for other languages—including specialized parallel languages—are possible, the system has been designed for use with and specified in terms of an unmodified C compiler. Using pthreads may require the use of special 'thread-safe' software libraries and new techniques for ensuring data synchronization, but it falls basically within the space of traditional Unix-and-C programming.

Third, in their function of rendering the asynchronous synchronous, pthreads provide a coherent, efficient and legible alternative to the various asynchronous and non-blocking I/O facilities that have been developed over the years. Except perhaps in how it deals with timeouts, pthreads provides better solutions to the 'select problem' (of how to monitor multiple sources of work at one time) than `select()` itself (or System V `poll()`, for that matter). Most importantly it works for *arbitrary* sources of synchronization constraint, while `select()` is a solution specifically applicable only to file descriptors (and `poll()` is even more restricted).

Fourth, pthreads offer the beginnings of a solution to the problem of exploiting physical multiprocessors to meet the needs of 'ordinary' programs in the POSIX environment. Without threads, multiple processors can be exploited only by specialized hand-coded routines or on heavily-loaded machines running several processes at once. Proprietary threads packages have existed for some time, but these only solve the problem at the expense of portability. Pthreads addresses the reality that multiprocessor hardware running Unix-like (or otherwise POSIX-compliant) software is increasingly the norm, and it must become possible to use it effectively without sacrificing portability.

At the same time, the need to provide hooks for experimental or vendor-specific extensions has been recognized; pthreads specifically provides ways of specifying and controlling such extensions, while limiting their impact on portable code.

Finally, since the threads of a process all operate within a single address space, making thread operations much cheaper than process operations, pthreads succeeds in addressing many of the needs of real-time programming better than the new real-time POSIX extensions can alone.

3.2.1 Threads and processes

What is the relationship between threads and processes? For most purposes, a (POSIX) thread exists entirely within a single process, in much the same way that a POSIX process is permanently bound to a single physical host computer. From the outside, there is no real way of telling whether a process is multithreaded, or not. The one obvious exception to this layering is that an implementation may arrange for threads to compete directly for use of physical processors, regardless of which processes they reside in. It is, however, equally valid for thread scheduling to take place entirely *within* a process, meaning that a thread cannot possibly run unless its containing process is scheduled first, and it is only eligible for a sub-portion of the resources allocated to the process by the kernel.

How the various resources and pieces of state associated with a program have been divided up between the host computer, the process and the thread is summarized in figure 3.1. The process remains the main locus of dynamic resource control—it provides automatic cleanup of many resources, which pthreads don't—and resources have remained associated with the process wherever this would not compromise the usability of the threads themselves. The consistent objective has been to keep threads cheap enough to build and discard as needed: everything focuses on providing independent threads of control with independent synchronization behavior, with as little overhead as possible. There are no 'firewalls' between threads; where by default processes contend for resources, threads share them.

The only really subtle decisions are found in the row marked 'exceptions': signal handling and error reporting have been distributed between the thread and process levels in accordance with a number of conflicting goals; we will have much to say about the results later in the book.

3.3 Example program

3.4 Compiling for pthreads

3.5 Standard C

POSIX is moving towards compliance with Standard C [[**sps: cite!**]] in its use of C. Pthreads, as a recent addition to POSIX, is designed for use with standard compilers, and depends on a number of subtleties of this dialect. This section is a brief overview of the most important changes and additions in Standard C relative to older C dialects, as they relate to pthreads. (The seasoned ANSI warrior may wish to skip over it.)

- Standard C allows (and in some cases *requires*) **function prototypes**, declarations of functions that include information about the types of their arguments—and whether they take a variable argument list. Here are some typical examples:

  - `char *strcpy(char *dst, const char *src);`
  - `int printf(const char *format, ...);`

  (An abbreviated form is also permitted in which the argument names are omitted: `char *strcpy(char *, const char *);`. The names are only there for documentary purposes, anyway.)

  These have been designed (and this is an important and insufficiently appreciated fact) so that carefully and correctly written code need almost never contain a cast. This is important, since casts explicitly defeat C's type system, and any use of one should now stand out like a red flag.

- The `const` modifier marks a value as not being changed through the name being introduced. Because of the way C argument passing works (the argument itself is copied, though what it may point to is not) this isn't a very interesting declaration in an argument list *unless* it applies to the target of a pointer type (occasionally you can catch some errors by using it on ordinary values in local variable declarations, however, marking variables that shouldn't change after they are initialized). By the way, the position of the `const` modifier matters: '`const char *`' (or '`char const *`') is a (modifiable!) pointer to a character that doesn't change, while '`char * const`' is a pointer that doesn't change to a character that can. Like everything else in C declarators, this is consistent, but backwards!

- There's a new 'type' (or semi-random collection of hacks, on one view), `void`. This shows up as the return type of functions that don't actually return anything interesting, and in declarations like `char *memcpy(void *dst, const void *src);`. When used as the base type for a pointer it means that it doesn't *matter* what types `src` and `dst` point to—and this saves you a cast that would otherwise have been unavoidable now the compiler knows the types of all function arguments. The various uses of `void` are tricky and inconsistent; more on this below.

- Header files can now be expected to include function prototype declarations, and it is important to use them—otherwise the unexpected may result.

- In Standard C, `errno` is not required to be a variable, though it is still an **lvalue**—you can still assign to it. The pthreads system takes advantage of this to make it private to each thread (which a real global variable cannot be). It's still close enough to being a global variable to be a bad piece of design, however (as we work with multiple threads we will begin to see just *how* bad), and the pthreads functions themselves avoid using it.

- For some obscure reason (perhaps having to do with implementation on segmented memory architectures, though this doesn't seem a strong enough motivation) Standard C 'breaks' pointer comparison: within the standard, pointers can only be compared if they refer to objects allocated within a single array! Pthreads thus goes to some effort to make pointer comparison unnecessary (or at least conceal it within a library function if it happens).

There are a few regards in which pthreads does *not* exploit the full capacities of Standard C:

- Although the standard now guarantees that structures can be passed by value and returned as results (a feature good compilers have supported for many years), POSIX still avoids this usage, consistently passing structures by reference.

- Standard C provides a modifier `volatile`, used in much the same way as `const`. It's tempting to think of `volatile` as warning that another thread of control might modify a variable unexpectedly, just as `const` says that the present thread of control won't, and asking that the compiler generate code 'literally'. Unfortunately, if you read the C standard carefully, you'll discover that *this is not what it does*. It *does* constrain the actions of the compiler in such a way as to limit how accesses to the variable can be moved during optimization, but the requirements are sufficiently weak that the result is not very useful for synchronization unless great care is exercised. Pthreads provides a very different model in which specialized functions manipulate data structures called **mutexes** to guarantee correct synchronization; if appropriate mutex discipline is followed, other shared data do not have to be specially marked at all.

3.5.1 The many meanings of '`void`'

One of the most confusing things about Standard C is the multiplicity of uses to which it puts the reserved word '`void`'.

`void` A function with return type `void` is one that does not explicitly return a value. Thus '`void`' is often—and misleadingly—described as being a type with no elements. Mathematically speaking it has *one* element (if a function doesn't return *anything* it follows that it doesn't return at all...) but since this element is always the same it is never interesting—and can be stored in zero bits!—which is what we wanted. As a 'type' it is very restricted; it can only be used as a function return value or in a cast (to discard a result); there are no void variables or void structure fields. The idea of `void` is to formalize the distinction between true `int` functions (those that return with `return x;`) and those returning no useful result at all (`return;`).

`void *` Confusingly, the type `void *` is defined to be a pointer to almost anything (the opposite of what you might expect). Any kind of pointer can be assigned, without casting, to a void pointer, and any pointer that has been converted into a void pointer can be cast or assigned back again without harm. There is an exception and a disclaimer: the exception is that pointers to *functions* cannot be stored in void pointers (presumably for the benefit of compilers that use different sizes of pointers for code and data); and the disclaimer is that you can't use void pointers as a trick to get around the restrictions on when casting pointers is *meaningful*. The idea behind `void *` is to provide a more explicit replacement for the old practise of using `char *` as a universal pointer type, one that avoids casts that (wrongly) imply that a change in the object's *intended* type is wanted. (The idea has one big flaw: `sizeof(char)` is `1`, but `sizeof(void)` is defined to be an error.[3] Doing arithmetic on `void *` variables thus no longer works. For this reason buffer management software still ends up using `char *` internally, anyway.)

`f(void)` A third and even more confusing use of the keyword `void` is as the only member of the argument list of a function being declared. Here—even though it looks like an argument declaration—it means that the function takes no arguments at all. This and the `...` convention for describing functions with variable argument lists together resolve the three-way ambiguity between old-style function declarations that give no argument type information, functions that allow variable argument lists (and which therefore can't be pre-specified with accuracy), and functions that truly take no arguments at all.[4]

3.6 Important posix concepts

[[

- *shared memory [maybe discussed in more detail under synch, when it comes up?*
- *other posix.4 things?*
- *fork/exec?*
- *signals?*

[3]This is a natural consequence of the overloading of the `void` keyword for several types: as the type of abstract storage ('`void *`') it should refer to the smallest addressable object, which is, on most machines, the byte; as the uninformative type (with only one element that are all the same, as it were) it refers to an object of zero length. This is *not* useful as the basic unit of storage, because you have to build a *big* array of them to hold an item of any other type!

[4]As a party trick you can 'prove' that this is really the same use of the word `void` as in the return type of a function, on the assumption that you can have void variables with the null string as their name—except that then you have to explain why `,` and `?:` don't allow null arguments.

]]

Chapter 4

Basic Thread Operations

4.1 Introduction

As we saw in chapter 1, a thread can be understood as an abstraction of a CPU. In traditional systems there is only one CPU, and programs are written more or less as if they had full ownership of it. Behind the scenes it is really shared between processes, but the operating system is designed to conceal this fact. Such a system organization provides exactly one thread of control per process, and since that thread is unique within its context, it needs no name of its own.

Multithreaded programming provides a different view of the world. The basic model is that there are multiple virtual CPUs—perhaps even multiple physical CPUs—working together to execute each process. This way, if one of them is waiting for an external event to occur, the others can still do useful work, without contorting the whole program. In the particular style of multithreading chosen for POSIX, threads are dynamic objects that can be created and destroyed as needed, recruiting processing resources to an ongoing task or freeing them for use elsewhere.

In order to maximize flexibility for the programmer, POSIX threads are represented by explicit data objects that can be stored in variables, passed as arguments to functions, and returned as results. To give system implementors more freedom, their type is opaque and the standard specifies only how the functions that operate on them should behave. They are intended to be sufficiently cheap to create that programs can make them whenever they are needed, and throw them away when they have served their purpose (there's a comment in the standard that implementors should pay attention to this even if memory-intensive caching techniques are required).

In this chapter we present the thread data type itself, and the most basic operations over it: those that create threads, compare them, and destroy them. At least some of these operations will be used by every multithreaded application.

4.2 The thread type: `pthread_t`

In POSIX, threads are represented by **thread IDs**, objects of type `pthread_t`, as defined in the standard header file `pthread.h`$_{273}$. They provide access to all of the operations that one thread can perform upon another, though as with processes there are many more functions that operate implicitly on the thread that calls them.

Like most other 'handles' designed for internal use by software, thread IDs are not eternally unique; they are defined only for the lifetime of the thread they name $\Rightarrow^{\text{F.c}}_{293}$. Each process starts out with a single thread executing the `main()` function, and new thread objects are allocated dynamically as threads are introduced with `pthread_create()`$_{33}$. After a thread's life has ended—which can happen in any of a number of ways—its ID is deallocated, and passing it as an argument to a thread function becomes an error. Many of the thread functions are *permitted* to report this error (typically by returning the error number `ESRCH`$_{271}$), but none of them is *required* to—they might just as easily signal an access violation or do something even more 'random'. This laxity lets the implementation free all a thread's resources as soon as the thread expires.

Thread objects, like members of `struct` types, can be copied without coming to harm: they can be stored in variables, passed as arguments to functions, and returned from functions as results. The standard says nothing about their internal structure, however, specifying only the operations that manipulate them. In particular, although many implementations will choose to represent threads as integer handles or pointers to control structures, it's not safe to assume that this is always the case—casting a thread value to an `int` or a `void *` might work on one machine, but will fail on another. Similarly, to compare them you must use the special function `pthread_equal()`$_{43}$; there's no reason the C `==` operator should work. Most important of all, thread values are only meaningful within their home process: not only are there no input/output operations that will work on them, but even if you should manage to get them into another process (perhaps via shared memory) they will not be usable there. This differs from process IDs, which are valid throughout the system for as long as the process survives (but of course even process IDs are not unique between hosts).

Joinability Threads come in two flavors, depending on how they behave when they terminate. The default, *joinable* threads, produce final values that are delivered to some other thread; they must wait for this delivery operation to complete before they can finally be deallocated. The thread accepting delivery will be delayed as necessary, which may be useful in its own right. Threads that do not behave in this way are called *detached*: detached threads do not expect to hand off a result, and any final value they produce is ignored. Thus the system can destroy them completely as soon as they terminate. The difference between the two flavors is a little like the difference between functions that do and don't return values—you use a thread for its value or for its effects—but, because of the resource release issue, much more important to program correctness. Using a lot of *joinable* threads and just ignoring their results will definitely lead to problems.

PTHREAD_THREADS_MAX Each system will have a limit to the number of threads that it is willing to guarantee can exist at one time. This is made known to software through the integer constant `PTHREAD_THREADS_MAX`, defined in `limits.h`, and is required to be at least `64` in any conforming implementation. Implementations are permitted to allow additional threads to be constructed on a best-effort basis; and on the other hand, independent resource limitations, such as shortage of memory, may prevent a thread being built before this limit is reached. But a request for a new thread will not be refused arbitrarily before `PTHREAD_THREADS_MAX` threads have been allocated.

4.3 Creating a new thread

In programming models where the number of executing threads can be changed dynamically, we need a method for the program to start a new thread and specify what it should do. Under POSIX, the job of thread creation is performed by `pthread_create()`. This function takes as arguments a pointer to a structure describing the desired initial attributes for the new thread, a function for the new thread to run, and an argument to pass to the function. It returns either zero or an error code, and when successful stores the ID of a newly created thread in a reference parameter that you provide.

(The initial thread attributes allow the programmer to control various qualities of the new thread, such as scheduling priority, stack allocation and initial detach state. Many of these must be set immediately because there is no real way to change them once thread execution has started. Thread attributes are discussed in detail later $\Rightarrow^{8.2,\,9.2}_{138,\,158}$.

Passing `NULL` as the attributes object instructs the system to use its default settings, which are usually acceptable.)

`pthread_create()` **16.1.2◇**

```
#include <pthread.h>

int /*errno*/ pthread_create
  ( pthread_t *new_thread_id
  , const pthread_attr_t *new_thread_attributes
  , void *(*new_thread_function)(void *argument)
  , void *new_thread_function_argument
  );
```

`pthread_t`$_{32}$ `*new_thread_id`

Pointer to the location in which pthread_create should store the ID of the new thread.

`pthread_attr_t`$_{138}$ `*new_thread_attributes`

Pointer to the attribute structure giving the properties wanted for the new thread $\Rightarrow^{8.2}_{138}$. If this argument is `NULL`, reasonable defaults are supplied$\Rightarrow^{??}_{??}$.

`void *(*new_thread_function)(void *argument)`
: Pointer to the function that the new thread should execute. This function should accept a single `void *` argument which can be used to distinguish between different invocations of the same function. The function is also declared to return a value of type `void *` so the thread can return an arbitrary data pointer when it terminates, just as `main()` can return an `int` process exit status. If the new thread is *joinable*$_{32}$, the default, the final value returned by ***new_thread_function*** can and should be retrieved using `pthread_join()`$_{40}$.

`void *new_thread_function_argument`
: The actual argument to be passed to the ***new_thread_function*** when it is invoked within the new thread. Since it is declared to be of type `void *` you can use it to pass a pointer to any information you like[1]. However, the called function will receive it in a different thread than the one calling `pthread_create()`, so you must take care that its lifetime is sufficient; in particular, if it is stack-allocated (as `auto` storage or by a stack-allocating function like `alloca()`), the thread that allocated it must not leave the block in which it was allocated until it has ascertained, by explicit synchronization, that all threads are finished using it $\Rightarrow^{5.2}_{64}$.

The return value is an integer containing an error number$\Rightarrow^{A}_{271}$.

On success `pthread_create()` returns `0` and places the (`pthread_t`) thread id of a newly created thread in the location pointed to by its first argument; when it starts, the new thread will evaluate
`(*new_thread_function)(new_thread_function_argument)`

On failure it returns an error code, and the location pointed to by the first argument is unusable. The following codes may be returned:

- `EAGAIN`$_{271}$ The process lacks the resources to create another thread, or the total number of threads in the process would exceed `PTHREAD_THREADS_MAX`$_{33}$.
- `EINVAL`$_{271}$ A value specified by *new_thread_attributes* is invalid.

The function `pthread_create()` is a **synchronizing function**$\Rightarrow^{5.2}_{64}$: the new thread starts with a view of memory that is consistent with that of the caller at the moment it invokes `pthread_create()`.

If `pthread_create()` is successful, it constructs a new thread and immediately starts it executing the specified function; no additional operation is required to set it running.

Each thread created within a process has its own:–

- virtual CPU;
- stack—though each thread can address variables in other threads' stacks indirectly, via pointers;

[1] Except for a function; Standard C does not guarantee that a function pointer can be used as a value of type `void *`$\Rightarrow^{3.5}_{26}$.

- signal mask and list of pending signals$\Rightarrow^{10}_{183}$;
- the `errno`$_{271}$ 'variable';$\Rightarrow^{12}_{219}$[[**sps: ?**]][2]
- thread-specific data mappings$\Rightarrow^{6.2}_{114}$;
- cleanup handler stack$\Rightarrow^{6.3}_{121}$; and
- thread control information, including scheduling settings$\Rightarrow^{9.3}_{160}$, cancelation state $\Rightarrow^{11}_{203}$, and system state.

As you can see from the cross-references, most of this stuff will be described in detail later on. The new thread's private data structures are initialized; all but the signal mask are zeroed or emptied as appropriate:–

- The **stack** is set up so that it will call into the new thread's initial function with the specified argument, and return its ultimate result to an implicit call on `pthread_exit()`$_{37}$ (described next).
- The **signal mask** is copied from its creator. Be careful: this is only going to be what you want if you adopt a programming style in which most signals are masked most of the time.$\Rightarrow^{10}_{183}$ The list of signals pending for the thread is made empty.
- The **errno** 'variable' is set to `0`, indicating that no errors have yet occurred in the new thread.
- All its **thread specific data** are set to `NULL`, its stack of pending **cleanup handlers** is made empty, and its cancelation state is made *enabled* and *defered*.

Note in particular that the new stack is *completely* new; it shares nothing with the stack of the creating thread, not even the part active when `pthread_create()` is called; thus the specified initial function becomes the 'outermost' (user-visible) function invocation of the new thread, just as `main()` is the outermost (user-visible) invocation in a newly `exec()`ed process. Not only does this simplify the relationship between the thread abstraction and function call by eliminating the possibility that the same call will be returned from more than once, but it is important to performance: no block-copying of data is required to support the basic thread operations, and stack space is conserved.

From the C programmer's perspective, when `pthread_create()` is called, a new thread of control comes into being, starting at the specified function (the calling thread, of course, continues to execute `pthread_create()` and will eventually return from it normally). This new thread has access to all the code and data of the program, including `static`, `auto`, `extern`al and heap-allocated objects such as those obtained with `malloc()`. In particular, all global identifiers and all pointers will refer to the

[2] This is possible because Standard C does *not* require `errno` to be an actual variable, so long as it is some kind of lvalue after preprocessing$\Rightarrow^{3.5}_{26}$. An implementation can now legitimately use a macro expanding to something like `pthread_self()->_errno`, though user code may not depend on this, either.

same objects in every thread. This doesn't hold for local variables that aren't declared `static` or `extern`, but then local identifiers *always* get new bindings when their scope is entered. As usual, you *can* get at the locations they name from outside their original scopes, but only indirectly, using pointers—and only for as long as they exist.

As we discuss elsewhere, simply having access to an object is not enough to be able to use it safely; there are still problems of synchronizing data access by the various threads$\Rightarrow^{1.3}_{8},^{5.2}_{64}$. Such problems quite often lurk within library functions$\Rightarrow^{12.2}_{220}$, particularly those using static variables.

The rule for pthreads is that as many threads as want may share read access to an object; but if a thread *modifies* a value it mut be the only thread that uses that value for any purpose. Synchronization is required to transfer ownership between threads.

It is guaranteed, though, that a new thread starts off synchronized with its creator; in particular, this means (1) that either the old thread or the new one can assume read-write use of any object that the old thread previously had read-write use of, providing the other stops using it altogether; and (2) that both threads can maintain read-only use of any object that the old thread previously had use of, whether read-only or read-write.[3] These rules imply that there are no problems with constant objects, objects that are only used by only a single thread, or, of course, with the code itself.

Since we don't cover the POSIX inter-thread synchronization facilities until later, we should issue **one warning** here: attempts to share unsynchronized objects or call non-thread-safe library functions normally result in unpredictable or non-portable behavior, *not* well-reported errors; programs that work fine in one context and bomb in others are quite common. Don't assume that sharing something officially unshareable will work out fine on the basis of experimentation!

From the operating system's perspective, the only substantive difference between the threads in a process is that they execute distinct instruction streams and have different CPU contexts, including registers, stack pointer, signal masks, `errno`$_{271}$ 'variable' and caches[4]$\Rightarrow^{3.1}_{24}$. Indeed, the original proposal for POSIX threads allowed them to be implemented as a user-level abstraction, without special assistance from either the compiler or the kernel.[5]

The important structures that threads share include:–

- the process ID, process group and session membership;
- real and effective user and group IDs;

[3]The base cases are, as you'd expect, that the primal thread initially has read-write use of all `static` and `extern` objects, and each thread initially has read-write use of any objects that it allocates.

[4]The fact that multiple *physical* CPUs may have their own data caches is the biggest single reason that data synchronization has to be performed explicitly, even when there are apparently reasonable arguments from the program structure that memory updates take place in the right order. The way caches are structured and implemented, they can happen to re-order main memory accesses quite arbitrarily, and the operation of resynchronizing them is too expensive to perform frequently or unnecessarily$\Rightarrow^{C.2}_{277}$.

[5]The requirement, added later, that no blocked thread prevent a runnable thread from eventually making progress has made a strictly conforming user library implementation impossible on many of today's systems. POSIX systems are not required to provide a mechanism for user processes to avoid indefinite blocking on file accesses, and such accesses to remote file systems could quite possibly block indefinitely.

- file handles;
- file creation mask;
- file system root and working directory;
- signal handlers;
- memory map, including code, data and stack segments; and
- process arguments and environment.

Ancestry In the POSIX thread model there is no particular concept of ancestry: threads can enter into a number of different kinds of relationship with each other, but which thread originally created the current thread is not recorded anywhere. This is unlike the situation with processes, whose parent process ID is saved and used to determine where to send their exit status. Even the primal thread that dates from when the process was started is not distinguished in other than that if it returns from `main()` it will perform an implicit call to `exit()`$\Rightarrow^{4.4.1}_{38}$.

4.4 Self-termination

Once a thread has served its purpose, we have to be able to dispose of it, both to recover the static resources that it consumes, and to prevent it from using further processing time. The obvious, and most 'structured', way for a thread to terminate itself is for it to return from its initial function. In accordance with C tradition, POSIX also provides a non-local method: just as a process can end itself *(and all its threads)* by calling `exit()`, an individual thread can terminate itself at any time by calling `pthread_exit()` with its final value as argument.

`pthread_exit()` **16.1.4**

```
void /*No Return*/ pthread_exit(void *final_value);
```

`void *final_value`
: The final value returned by the thread, analogous to the status code accepted by `exit()`. Since it is of type `void *`, it can be used to return a value of any C pointer type other than a pointer to a function. The object pointed to should not be allocated on the caller's stack, because the stack may be freed by this operation.[6]

The function `pthread_exit()` never returns to its caller, since it terminates the thread that calls it and (if it is detached$\Rightarrow^{4.6}_{41},^{8.3.1}_{142}$ or when it is joined$\Rightarrow^{4.5}_{40}$) frees its resources.

When `pthread_exit()` is invoked, whether directly or implicitly by returning from a thread's initial function, it does the following:–

[6] It isn't clear whether this caution would necessarily apply in the case that the stack was user-supplied$\Rightarrow^{8.3.2}_{146}$, but unless the final version of the standard specifically guarantees that this will work, we strongly recommend against trying it.[[**sps: ***]]

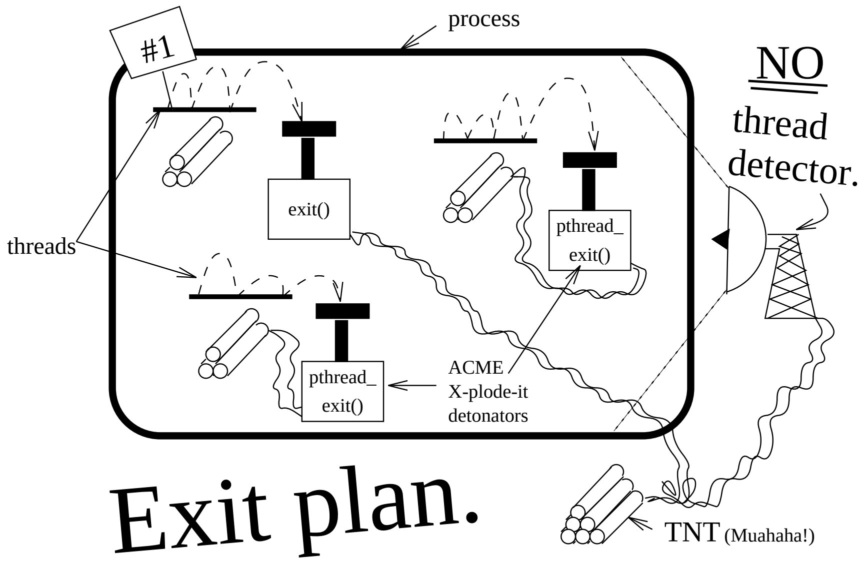


Figure 4.1: Implicit termination of threads and processes

1. If the thread has any pending (unpopped) **cleanup handlers**, they are invoked in the reverse of the order in which they were pushed$\Rightarrow^{6.3.1}_{122}$; and the registered destructors are called on any thread-specific data$\Rightarrow^{6.2}_{114}$.

2. If this is the last thread remaining in the process, the whole process exits with an exit status of `0`. (To get a nonzero exit status you must call exit explicitly or equivalently arrange for the primal thread to return.)

3. Otherwise, if this thread is *joinable*$_{32}$, which is the default, the ***`final_value`*** is transmitted to the thread that calls `pthread_join()`$_{40}$ with the ID of this thread as argument. One of the two threads involved in the transfer may be delayed until the other reaches this point, but since `pthread_exit()` never returns this is not observable from the exiting thread.

4. Finally, the private resources that the thread uses—such as stack space—are released.

4.4.1 Implicit thread termination

If *any* thread calls `exit()` it immediately terminates the *whole* process including *all* of the threads running within it *without* doing any thread-specific cleanup. If, on the other hand, a thread only wants to terminate *itself,* there are two ways to do it. The first is for it to call `pthread_exit()` explicitly; the second, and more elegant, is for it to return a final value from the function it was originally given to evaluate—but this only works for threads other than the primal thread. In either case, only the one thread terminates, and thread-specific cleanup *is* performed. The whole process may then exit too, but only if the thread that just exited was the last thread remaining within it.

One way of thinking about how C programs usually exit is to imagine `main()` being called from another function that contains the sequence

```
int status = main(argc, argv); exit(status);
```

(in fact, this is how most implementations actually work). In the same way, you can imagine the initial function parameter for a new thread being used like this:

```
void *status = (*function)(argument); pthread_exit(status);
```

This way of looking at things helps explain an irregularity in the POSIX thread specification: while other threads can return from their top-level functions and have only themselves terminate, if the primal thread returns from—or 'falls off the end of'—`main()`, the whole process exits. There's nothing magic about the primal thread or how it returns, it's just that the (internal) function that called `main()` also calls `exit()`, and `exit()` specifically blows away the whole process (see figure 4.1).

Note that this one small difference between the process' primal thread and any others that may be created later makes an explicitly callable `pthread_exit()` essential: any other thread can terminate itself by returning from its initial function, but the primal thread must have some means of private termination, one which allows other threads to continue their work.

The relationship between thread termination and process exit in the POSIX thread standard makes sense in light of the notion that threads are entities that exist *within* a single C program, which in turn resides *within* a process. Since the threads are part of a single program execution, and C specifies that `exit()` terminates a program, it follows that all threads should die when `exit()` is called. Other systems have interpreted threads as being simply processes that share all their memory, in which case it can be argued that each thread is a logically distinct program invocation. In this view, `exit()` should terminate only the calling thread. Various other conventions have also been used.

4.5 Retrieving a final value

When one or more threads are started to perform services or subcomputations, the initiating thread must often gather their results before proceeding. This common pattern of thread use led to the introduction of the `pthread_join()` function, roughly analogous to the `wait()` system call that retrieves the exit status of a child process. There are other means whereby threads can synchronize to exchange data, but the existence of `pthread_join()` allows programs to synchronize precisely on thread termination, and without having to allocate any additional explicit structures.

It has been commented that this function is singularly poorly named, since the English word 'join' usually implies a symmetric relationship between two joined things.

Perhaps better images would have been 'collecting' a thread after it exits (as of a cinema), 'picking it up' after it falls down, or 'burying' it after it dies.

`pthread_join()` **16.1.3†◇**

```
int /*errno*/ pthread_join
  (pthread_t pthread, void **final_value);
```

`pthread_t`$_{32}$ *`pthread`*
: Thread whose termination is awaited.

`void **`*`final_value`*
: Place to store `pthread`'s final value, as passed to `pthread_exit()`$_{37}$; or `NULL` if this value is not needed and you are simply awaiting termination.

The calling thread is suspended until the thread identified by *`pthread`*, which must be *joinable*$_{32}$ and not joined by any other thread, terminates$\Rightarrow^{4.4}_{37}$. Joinable threads will wait indefinitely to be joined, and their resources cannot be recovered until this happens, or the process exits. The order in which one thread terminates and the other joins with it is immaterial.

On success the return value is `0` and, by the time `pthread_join()` returns, *`pthread`* has already terminated and its final value has been stored in the location pointer to by *`final_value`*, if any. In the case that *`pthread`* did not terminate itself, but was canceled by another thread$\Rightarrow^{11.2}_{204}$, the final value will be the special constant `PTHREAD_CANCELED`$_{204}$ (which is neither `NULL` nor the address of *any* object but can be tested for with `==`[7]).

On failure An error code encoded as for `errno`$_{271}$ is returned, and the location pointed to by *`final_value`* is not updated. In particular:

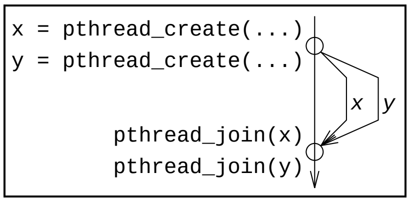
- `EINVAL`$_{271}$ The implementation was able to detect that the argument *`pthread`* is not a valid thread object.
- `ESRCH`$_{271}$ The argument *`pthread`* cannot be joined with. This may be because it is *detached*$_{32}\Rightarrow^{8.3.1}_{143},^{4.6}_{41}$, or because some other thread has already joined with it.
- `EDEADLK`$_{271}$ The join would result in a deadlock; perhaps *`pthread`* was the calling thread itself.

What happens if you attempt to join a thread that is detached is undefined. More surprisingly, what happens if more than one thread tries to join the same thread is also undefined. The reason is that joining with a thread allows it to be deallocated as soon as it exits, so any other convention would invite synchronization errors.

This is a **synchronizing function**$\Rightarrow^{5.2}_{64}$: after returning from `pthread_join()`, the calling thread's view of memory is consistent with that of the thread being joined, providing all of the usual restrictions have been obeyed.

[7]Language lawyers will be titillated to note that this constitutes a substantial extension to Standard C pointer semantics. Everyone else will rightly assume that it will be defined as something implementation-specific along the lines of `((void *)1)` and wonder what this footnote is for.

The synchronizing effect of this function is that if the exiting thread has read-write use of an object, the joining thread acquires it. If the exiting thread has read-only use of an object it surrenders it to the joining thread; if the joining thread is now the *only* thread with use of the object, it obtains read-write access.



The `pthread_join()` convention nicely provides for 'structured parallelism': using it, a computation that has fanned out into several parallel threads can readily pool the results before proceeding to its next step, in a classical fork/join structure. In other situations, however, it suffers from a significant drawback. Unlike `wait()`, the caller of `pthread_join()` must specify *which* thread to join with next. This makes it a poor tool for the case where a supervisory thread creates subordinates that run for varying lengths of time, generating new work for the supervisor whenever any of them terminates, since the supervisor has no way of knowing which worker thread to wait for first. Such a situation often arises when threads are created to perform I/O over independent channels, for example. In the next chapter we present a fairly complete workaround for this problem.$\Rightarrow^{5.7.1}_{95}$

In the next section we see an alternate attack on the problem: if it is not necessary to detect exactly when a thread completes and to retrieve its final result, that thread can simply be **detached**.

4.6 Detaching a thread (making it non-joinable)

In general, it can be inconvenient to have to know at creation time whether a thread will be joined, or if it should be detached so that it can be freed as soon as it exits—or you might want detached threads anyway but not want to mess around with thread attributes (which we haven't introduced yet anyway!). POSIX provides a simple way to **detach** an existing thread—providing it is still *joinable*$_{32}$, that is, it has not yet been joined or detached—by passing it to `pthread_detach()`.

pthread_detach()

```
int /*errno*/ pthread_detach(pthread_t pthread);
```

`pthread_t`$_{32}$ *`pthread`*
: The thread to detach.

By **detaching** a thread you declare that no thread will ever join it. The advantage is that once it has exited its physical resources may be reclaimed immediately$\Rightarrow^{4.4}_{37}$. You can detach a *joinable*$_{32}$ thread at any time.[[**sps: Check this against the final standard.**]]

On success `pthread_detach()` returns 0; *`pthread`* is now detached.

On failure returns an error code as for `errno`; specifically, it may return:

`ESRCH`$_{271}$ The thread *`pthread`* could not be located. Perhaps it has already terminated.

`EINVAL`$_{271}$ The argument *`pthread`* is not joinable. This may be because it is already *detached*$_{32}$.

Note that this function must not be used with threads with explicitly allocated stack storage$\Rightarrow^{8.3.2}_{146}$ if that storage is to be freed or re-used, since only by joining with a thread can you reliably determine when it has finished terminating and its stack is finally free.

Detaching a thread differs from joining with it in two important ways. The first is that the caller of `pthread_detach()` does not wait until the argument thread terminates; detachment takes effect immediately, even if the thread in question is still running. The second and related difference is that `pthread_detach()` performs no data synchronization. Any transfer of data between a detached thread and other threads must be synchronized explicitly$\Rightarrow^{5.2}_{64}$.

4.7 Other basic operations

The basic vocabulary of POSIX thread operations also provides for thread assignment and comparison, and allows a thread to obtain its own thread object. Assignment is accomplished directly with the usual `=` operator of C; the remaining two operations are provided explicitly by the thread library.

It is sometimes useful for a thread to be able to obtain its own thread ID; it might want to store it in a variable, give it to another thread, or pass it as an explicit argument to a function. This can be done by calling `pthread_self()`.

`pthread_self()` **16.1.5**

```
#include <pthread.h>

pthread_t pthread_self(void);
```

The function `pthread_self()` takes no parameters, and returns its caller's thread object. It cannot fail.

This is the only way to obtain the thread object of a process' primal thread, since unlike other threads the primal thread is not built explicitly. It is also convenient in other circumstances, since when a thread is first created, its ID is reported to the old thread, but not to the new one.

Thread objects can be compared for equality with `pthread_equal()`.[8]

[8] It might be nice if there was an order defined on threads, even an arbitrary one, but there isn't; if you want that, you'll have to do it yourself—and that will involve either linear search or finding a way to make threads identify themselves. There isn't a 'null thread' value either.

`pthread_equal()` **16.1.6**

```
#include <pthread.h>

int /*boolean*/ pthread_equal
  (pthread_t thread1, pthread_t thread2);
```

`pthread_t`$_{32}$ ***`thread1, thread2`***
Threads to be compared for equality.

Given two pthread objects ***`thread1`*** and ***`thread2`***, `pthread_equal()` returns nonzero if they represent the same actual thread, and `0` otherwise. We can't just use the `==` operator because `pthread_t` is specified as an opaque type, in order to allow the implementation free choice of physical representation.[9]

The effect of calling this function is undefined if either of the threads is already terminated⇒$^{4.4}_{37}$.

Because it is a predicate and may be used very heavily, `pthread_equal()` has been designed to return its boolean result directly, rather than returning an error number and updating a reference parameter as most other POSIX thread functions do. If either of the arguments is not valid at the time of call, the result will be unpredictable, and perhaps even fatal to the process.

Although the two functions we presented in this section are very simple, their utility only becomes apparent in programs that are sufficiently complex that they compute *about* threads and not just *with* them. For this reason they do not appear in examples until later in the book⇒$^{12.a.1}_{237}$. The next section provides examples of the use of the other functions introduced in this chapter.

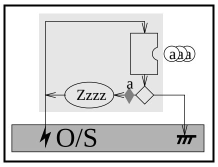
4.8 Example program: race.c

We now present a simple example program that illustrates basic thread creation and termination. It runs a simple 'race' of between one and twenty-six threads, each of which is identified by a different letter of the alphabet. Every second, each thread writes its name to standard output, until it has done so a predetermined number of times. The interesting question, of course, is not which thread wins, but what the program looks like.

The computation of ray-traced images is a practical example of a program that, like `race.c`, performs a large number of independent calculations; in ray-tracing, each pixel may be computed separately. The example below could form the basis of a program of this kind, if the `racer()` function that is run by each thread were modified

[9] In fact, it is mostly likely to use a pointer of some kind, but even then the function would be needed in order to avoid putting a small upper bound on the number of threads available. Standard C does not define the result of pointer comparison between objects not residing in the same array—a strange limitation that is irrelevant on many machines, but is supposed to make pointer comparison more efficient on machines with segmented memory architectures.

to perform the repeated calculation. This would certainly perform better than using multiple processes, and on a machine with multiple physical processors would be faster than using a loop whenever the parallel computations take a significant length of time.



Looking at the trivial case of a 'race' with just one participant, the most natural way to structure the program is probably with a main routine that, as usual, reads and interprets the command line, and one other function that actually does the work of producing the output. That function will need to do nothing but sit in a loop with a print statement and a call to `sleep()`?? to introduce the desired delays. Working with threads, we find that this same simple structure that works for one racer will work for twenty-six, almost without change.

First, though, the declarations:

race.c — declarations

```
#include <stddef.h>                                          /* for NULL */
#include <stdlib.h>                                        /* for atoi() */
#include <unistd.h>                                       /* for write() */
#include <pthread.h>
static char const id_letters[] = "abcdefghijklmnopqrstuvwxyz";
static int const fd = 1;                              /* standard output */
static int count = 60;                /* iterations within each thread */
```

The header file for POSIX threads is `pthread.h`, and it must be included wherever we want to use thread types or operations. The other includes are just the usual ones for the facilities we will be using.

We next declare three variables, of which only one actually varies: `count` holds the number of times we want each 'racer' to print its identifying letter; we initialize it to a default value, but we'll let the main program override it for races of different lengths. The different threads will all use this value, but we stop changing it before they are created; that way there will be no synchronization problems when the various threads try to use it. The array of `id_letters` for the threads to print and the variable holding the file descriptor number they are to write to will also be shared between threads, but in this program they are never changed and we happen to have declared them constant. This is purely a matter of style—it might enable the compiler to produce slightly better code, and it documents our intent not to change them.

The function `racer()` that loops to produce the output will actually be used as the function executed by each new thread. If we weren't using multithreading we could write it something like this:

```
static void racer(char id)
  { int i;

    for (i = 0; i < count; i++) {write(fd, &id, 1); sleep(1);}
  }
```

Adapting this idea to the threaded case requires only one really visible change: because it will be passed as an argument to `pthread_create()`$_{33}$, its declared type must be `void *(void *)`—a function from a generic pointer to a generic pointer. We use the argument to pass a pointer to the identifying letter of the thread; as a matter of style we assign the generic pointer to a variable of the value's true type to minimize the window of opportunity for type errors.[10]

race.c (continued) `racer()`

```
static void *racer(void *arg)
  { int i;
    char *id = arg;

    for (i = 0; i < count; i++)
      {write(fd, id, 1); sleep(1);}
    return NULL;                      /* or equivalently, pthread_exit(NULL); */
  }
```

Just as might be expected, once it has picked up its argument, the function goes into a simple loop bounded by the globally shared static variable `count`, writing its identifying letter to standard output.

To actually perform the output, we use `write()` (at race.c(2)). Since `write()` is actually a system call, not just a library function, it is effectively atomic—it is a basic property of Unix that system calls are effectively serialized. If we used higher-level I/O facilities, particularly ones like stdio that employ buffering, we would have to be sure that we were using a thread-safe library (pthreads does provide one, as we shall see later), lest competing threads perform inconsistent updates on the library's data structures$\Rightarrow^{12.3}_{223}$. Of course, one could choose to use `write()` in the single thread case, too.

The use of `sleep()` at race.c(3) is more worrisome, and will serve as an introductory example of another kind of problem that can arise when programming with threads. Our idea in calling `sleep()` is that after writing its letter, our thread should pause for a while, giving other threads a chance to run and slowing down the output so that we can see it accumulate. As it happens, the POSIX standard specifies that `sleep(n)` delays its caller's execution for approximately `n` seconds *and acts only on the calling thread;* according to POSIX our example is correct. But imagine a Unix variant in which `sleep()` is not thread-aware (or is specified to operate on the calling *process* rather than thread). The effect would now be that each time *any* thread invokes `sleep()`, *all* threads are suspended for a second. Our 'race' program would now take sixty seconds *per thread,* not just sixty seconds in total—and for most of that time, all the threads will be suspended and incapable of processing.

The important thing to notice through all of this is that the function `racer()` does *exactly what it appears to do.* We have had to follow coding conventions that are

[10] Using this technique also allows us to reserve cast notation for unsafe situations. Far from forcing a value of one type to be reinterpreted as another, this instance is a safe, pro-forma, conversion.

a little more stringent than usual, but each thread, considered in isolation, enters the function normally, prints its character through `count` iterations of the loop, and returns. The effect of interleaving the different racers will be accomplished for us by the thread package.

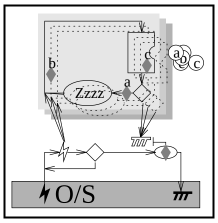
We turn now to the main program.

race.c (continued) `main()`

```
int /*No Return*/ main(int argc, char **argv)
  { pthread_t thread;
    int i;
    int num_pthreads = 2;

    if (argc > 1) num_pthreads = atoi(argv[1]);
    if (argc > 2) count = atoi(argv[2]);
    for (i = 0; i < num_pthreads; i++)
      {
        if ( pthread_create
               (&thread, NULL, racer, id_letters + i)  5
           )
          { static char const message[] =
              "error creating a new thread\n";

            write(1, message, sizeof(message));
            exit(1);  6                         /* or equivalently, return 1; */
          }
      }
    /* Don't exit the process until all threads are done. */
    pthread_exit(NULL);  7
  }
```



After decoding its arguments, `main()` drops into a loop from which the output is indirectly printed. Rather than doing the work itself, however, it spawns, at race.c(5), a number of threads, which proceed to call `racer()` *in parallel.* Each one receives a pointer to a different letter of the alphabet as an argument—recall that the `NULL` second argument says that we accept the default thread attributes. The effect when we run this program will be an arbitrary interleaving of the various assigned id letters.[11]

It shouldn't happen, of course, but if at some point the `pthread_create()` call should fail, the main thread will proceed to output an error message and exit abnormally at race.c(6). This terminates the whole process and immediately shuts down any other threads that may already have been created.

[11] Note that the behavior is arbitrary, but not random: under some circumstances you will find that it is repeatable, under others you will find that it is not. If you are running a simulation or have some other need for randomness *per se*, you will still need to use a good random number generator to get it.

Since returning from the first invocation of `main()` also has the effect of invoking `exit()`, and since invoking `exit()` terminates all threads, it is important that we find something else for the primal thread to do when we have finished creating all `count` threads, something that will not interfere with those threads finishing their jobs. In the absence of specific work, an appropriate choice is to call `pthread_exit()`, as at race.c(7); this will terminate the current thread, in this case the initial thread, but it will not interfere with the others; the process as a whole will continue to exist until the very last thread terminates, or something terminates it directly.

Returning to race.c(4) in `racer()`, above, we note that the function returns `NULL` when its job is done. In this program, it might equally well have made an explicit call of `pthread_exit(NULL)`; the effect, in each case, is to terminate the then-current thread. Eventually, the last thread will finish looping and return; then, and only then, will the execution of the program be over.

At this point we should admit that there's a minor flaw in our example, though not one that will cause any difficulty in this case. The problem is that we have accepted the default thread attributes, which specify, among other things, that the threads be *joinable*. This results in a **resource leak:** although all of the various racer threads terminate, they are never joined, and so the threads package can never free them; each thread keeps waiting for a join that never comes. In this particular case it isn't a serious shortcoming, since all the threads exit at about the same time, and when the last one does the entire process is torn down. It's like when using `malloc()` to allocate heap memory: you can sometimes rely on process exit to do your cleanup, but in general it's important to remember to call `free()` on each object.

In a case like this, where we do not have any interest in a rendez-vous with outgoing threads, what we really want do is to use *detached* threads; to see why, let's first try doing our cleanup without.

In order to clean up after a *joinable* thread we need to have some other thread call `pthread_join()` for it. The best thread to have do this is clearly the primal thread (after all, it's not busy running a race, and it is the only thread that is told each racer's ID in the first place). But unless we're going to wait for each thread before the next one starts—fairly futile if we're trying to exploit parallelism—we're going to have to store the thread IDs during the race. In `race-clean.c` we present a revised version of `main()` that does this with an array. No change to the actual racer function is required; using this approach the racing threads themselves can be oblivious to the issue.

race-clean.c `main()`

```
int /*No Return*/ main(int argc, char **argv)
  { pthread_t thread[26];                         /* Save the thread IDs. */
    int i;
    int num_pthreads = 2;

    /* ... */
    for (i = 0; i < num_pthreads; i++)
      {
```

```
        if ( pthread_create
               (&thread[i], NULL, racer, id_letters + i)
           ) {/* ... */}
      }
    /* Join with all the threads so that they can be deallocated. */
    for (i = 0; i < num_pthreads; i++)
      pthread_join(thread[i], NULL /* Discard final value. */);
    exit(0);                             /* All threads must be done by now. */
  }
```

This last example clearly exhibits one difficulty with using `pthread_join()`. At race-clean.c(1) we iterate over the racing threads in a fixed order, arranging to join with each one as it terminates. In this case it doesn't make much difference, because all of the threads finish at about the same time (it is a very evenly matched race), but it's easy to imagine a situation where one thread happens to terminate well before another, and it would be best to join first with the thread that finishes first, whether it started first or not. Unfortunately it is difficult for one thread to determine whether another has yet terminated. But `pthread_join()` still requires that the caller name the specific thread that is to be joined with; you can't just ask for 'whichever thread terminates next'. With the tools we've seen so far, the above loop is about the best we can do.

Alright, now let's do it properly. We still haven't seen how to create a thread that is *detached* from birth, but we do know about the function `pthread_detach()`. This function also takes a thread ID as an argument, but unlike `pthread_join()` it doesn't introduce any unwanted synchronization; we can use it immediately and the caller won't block:

race-detached.c | `main()`

```
int /*No Return*/ main(int argc, char **argv)
  { pthread_t thread;
    int i;
    int num_pthreads = 2;

    /* ... */
    pthread_detach(pthread_self());
    for (i = 0; i < num_pthreads; i++)
      {
        if ( pthread_create
               (&thread, NULL, racer, id_letters + i)
           ) {/* ... */}
        pthread_detach(thread);
      }
    pthread_exit(NULL);
  }
```

In this version of the main function, instead of joining with the racing threads, we detach them. The primal thread does this as they are created, at race-detached.c(2). Calling `pthread_detach()` indicates to the system that no thread has an interest in joining with its argument, and allows the system to deallocate it as soon as it has finished its execution. Since no thread will ever call `pthread_join()`$_{40}$ with the primal thread as argument, we detach it, too: at race-detached.c(1) it calls `pthread_self()`$_{42}$ to obtain it own thread ID and then uses the result to detach itself. Thus, as soon as it calls `pthread_exit()`$_{37}$ at race-detached.c(3)—probably long before the 'race' is over—its resources can be freed.

This suggests another approach to the problem: have every thread call `pthread_detach(pthread_self())` at some point before it terminates. This, too, will work, though it suffers from the slight inelegance of requiring special coding of the worker thread functions. Later on we'll see examples where this is unavoidable: many kinds of advanced thread behaviour require the thread's initial function to perform some special setup before going about the thread's main business.

4.9 Summary

Considered as virtual CPUs, POSIX threads share everything within a process except `errno`, their signal masks, the thread-specific data and cleanup handlers that were introduced to support them, and their identities as flows of control. Considered as data, they are represented to the programmer as opaque binary values that have meaning only within the process they belong to.

Multithreaded code requires extra care to write, but those parts of it that rely on traditional features read like traditional code. Indeed, the function `pthread_create()`$_{33}$ takes an ordinary function as an argument, and calls it just as usual—merely in parallel with the original computation. When a thread wishes to exit, it can return as it would from any other function call or, by analogy with processes, call `pthread_exit()`$_{37}$ to have the job done immediately, perhaps from within an arbitrary nest of calls.

The main wrinkles are these: first, that because the creating thread is running independently from its child, it has to make a second call, to `pthread_join()`$_{40}$, to pick up the return value; and second, that because POSIX doesn't record the ancestry of threads, the parent thread might just as well return to the child as the converse(!).

Communicating between threads at their moments of birth and death is easy: the functions that accomplish these tasks provide `void *` arguments for the purpose, and all the synchronization necessary for data that are either read-only or have full ownership transferred at this moment happens implicitly. The one caveat is that passing a pointer to another thread is a bit like storing it in a global variable: you must be sure that the lifetime of the object it points to is sufficient that there is no chance of dereferencing a pointer to deallocated storage.

In this chapter we have presented the most basic functions on POSIX threads. By now we have the tools to create new threads of control, to have them run independently, to terminate them, and to gather their final results. We can assign them, compare them, and ensure that they are freed when we are finished with them.

Things are more complicated when one thread must read what another thread writes, in the middle of their respective lifetimes. The facilities for doing this safely are the subject of the next chapter.

Appendices

4.a A look inside

In this section we take a look at what might be found inside an implementation of `pthread_t`$_{32}$. Our hope is that this will help you develop your intuitions for how multithreaded software might perform, and suggest strategies for structuring your own code. It's relatively technical, though, and we've had no choice but to use concepts that aren't properly introduced until later in the book; so on a first reading you might want to skip this section.

Our discussion is loosely based on the source code of the CAPthreads implementation, and assumes that the threads package has been implemented primarily as a library—not in the kernel—on a single-processor machine.[12] The code is in some ways more interesting when there are multiple CPUs to worry about, but many of the additional complexities have to do with low-level memory synchronization details and their solutions are not in general portable.

The presentation abstracts away from the details of the original code and simplifies it somewhat. Even if you happen to be using the CAPthreads implementation of threads you should not rely on the literal truth of the information in this section. The POSIX standard specifies *only* the external behavior of the threads package, and particular implementations—or different releases of the same implementation—might be very different from each other. To give an example, you might conclude from a close reading of section 4.a.2 that `pthread_create()` will never modify the reference parameter that receives the ID of the newly constructed thread unless it returns success; but if you look at the description of the function on page 33 you will find a caution to the contrary.[13] Indeed, while this book was in the process of being written, the CAPthreads package went through stages in its evolution that modified this parameter on failure, and ones that politely left it alone.

4.a.1 pthread_t

[[sps: christoph asks, "Where's the scheduler?"]]

In CAPthreads, the opaque type `pthread_t` happens to be implemented as a pointer to a structure. Using an integral 'handle' would have permitted better dynamic validity checking, and using a structure containing a pointer would have given stricter static type checking, but a direct pointer provides the best performance for almost all architectures and compilers.

Threads are kept on a number of global queues, foremost of which is the `allqueue`, the list of all pthread objects that presently exist. Because these queues are used both by operations that threads invoke themselves and by the internal mechanisms of the threads library (which relies heavily on signals for its operation), something must be done to prevent them becoming scrambled if a thread-related interrupt causes them to be examined or changed while a user program (for example) is in the middle of modifying them. As we shall see in the next chapter, this is a general problem with a general solution at the user level; inside the CAPthreads code,

[12] A small amount of kernel support is actually required in order to avoid pathological blocking patterns while doing file I/O, and in order to get signal semantics just right.

[13] Of course, the final arbiter is really the standard itself, and it's there, not here, that you should check before deciding whether or not a given implementation is technically correct.

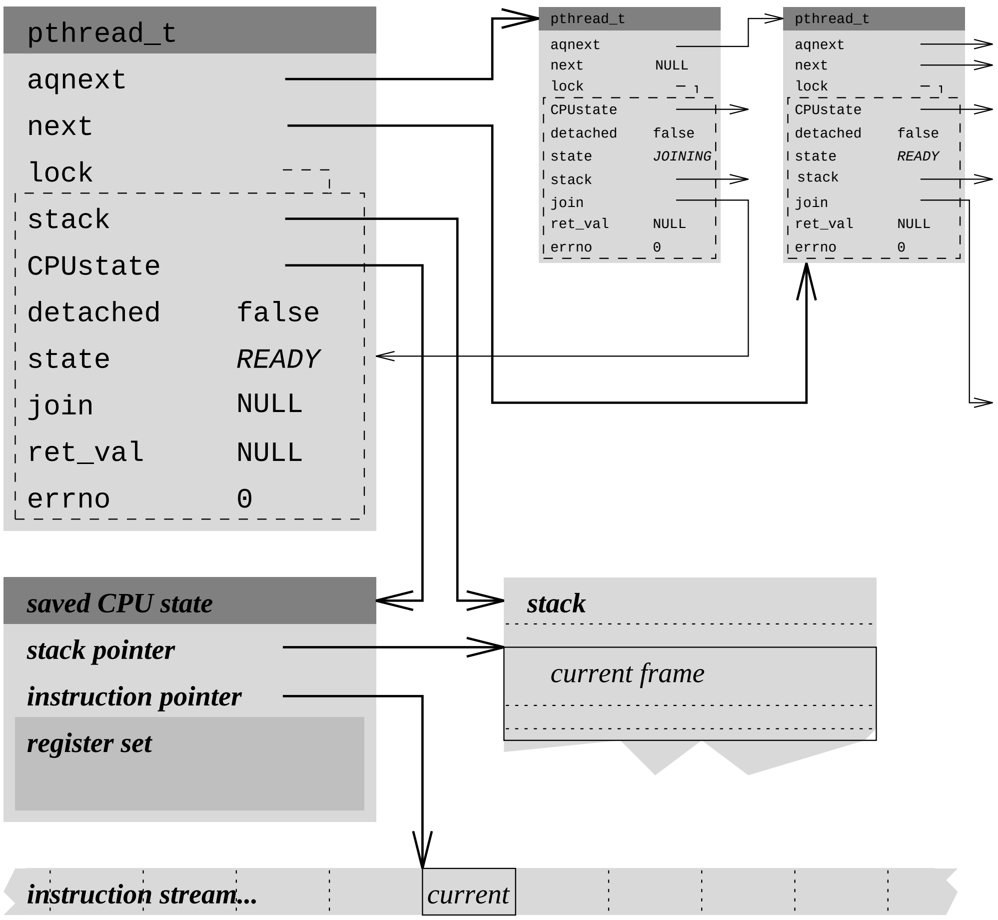


Figure 4.2: The `pthread_t` structure (simplified)

below the level of the user synchronization facilities, it is solved by the **global lock**, a variable that is set whenever the global queues must be updated to let other code know that it needs to wait for the current operation to complete (which it can do just by repeatedly switching to another thread until the lock is finally free).

Many similar locks are also used to protect individual data structures in the system. Associating some locks with particular objects—rather than always using the global lock—increases the complexity of the system slightly but greatly reduces the amount of waiting that is required to keep everything running smoothly.

Inside the pthread structure we find a fairly large number of fields with a variety of functions. Of these, the fields that support functionality presented in this chapter are:–

aqnext Pointer to the next item in the `allqueue`. This field is protected by the global lock.

next Pointer to the next thread in the **ready queue**. This holds all the threads that are currently ready to run. There are actually a number of other queues of threads in the system, but they are used to implement facilities we haven't yet introduced. Some of them share the use of this field. Protected by the global lock.

lock Used to synchronize access to the remaining fields of this structure.

CPU state buffer Place to store the (machine-dependent) CPU state for this thread while it is not physically running.

stack Pointer to the root of this thread's private stack area.

state The thread's scheduling state. This is an enumeration whose values include *running*, *joining* and *dead*. Other states will be described in technical sections of later chapters.

join pointer Pointer to the thread which has joined with this one, if any.

detached A boolean value recording whether the thread is *detached*$_{32}$.

return value Temporary storage for the final value yielded by the thread when it exits.

errno The thread's private `errno`$_{271}$ 'variable'.

See figure 4.2.

Of these, it is the machine-specific CPU state buffer that represents the 'soul' of the thread. Since there are often more threads than there are physical processors in the system, a typical threads package is ultimately implemented in terms of a low-level, machine-specific function that can save the whole CPU state in one buffer and reload it from another. By associating such a buffer with each thread object we can then guarantee each thread its own identity, and arrange for it to alternate between actually running on a CPU and existing in 'limbo' with its state safely buffered in the pthread object, while other threads chosen from the ready queue run.

The other truly crucial component of the pthread object is the lock field. By convention, whenever a function wants to manipulate the pthread object, it must 'own' this lock, meaning that it has a guarantee that no other thread will attempt to manipulate the same object at the same time. Low-level, machine-specific techniques may have to be used to maintain the invariant that only one thread can own a given lock at a time.

4.a.2 `pthread_create()`

When `pthread_create()`$_{33}$ is first called to start a new thread, the sequence of events goes something like this:–

1. Construct a new thread descriptor. In our implementation this will involve allocating memory to hold the pthread structure. If we can't get the memory and this step fails, return `EAGAIN`$_{271}$, because in principle the failure is recoverable.[14]
2. Examine the `new_thread_attributes` argument, supplying a default value if it is `NULL` or returning `EINVAL`$_{271}$ if it is specified but obviously ill-formed.
3. Initialize the new pthread object according to the requested thread attributes.
4. Obtain a stack for the new thread. This may have been supplied by the caller in the attribute structure$\Rightarrow^{\S 3.2}_{146}$, but otherwise will again require memory allocation, and again might fail with an `EAGAIN` return. [This is the last point at which `pthread_create()` is supposed to be able to fail.]
5. Initialize the machine-specific CPU state buffer in the new pthread object.

[14] Earlier versions of the specification allowed `ENOMEM`$_{271}$ in this situation, which violated the abstraction slightly but was potentially more useful to the programmer, since some algorithms can easily be structured so as to recover from memory shortages intelligently.

6. Initialize the newly allocated stack to a state in which `new_thread_function` is about to be called, with whatever value it returns being passed to `pthread_exit()`$_{37}$.
7. Zero the thread-specific `errno`$_{271}$ variable.
8. Set the thread descriptor lock to *free* and its state to *running*.
9. Update the location pointed to by the `new_thread_id` argument to refer to the new thread descriptor, which is now consistent and can be made visible to other threads.
10. Acquire the global lock so that we can update the thread system's global administrative structures in safety.
11. Put the new thread on both the `allqueue` and the ready queue. The new thread is now a full citizen and will be able to run independently when it is scheduled.
12. Unlock the thread system. (This may cause us to be pre-empted and another thread to run if a timeslice boundary passed while we owned the lock.)
13. Return `0` to indicate success.

The reader may have noticed that despite our claim that the pthread object lock must be acquired whenever the pthread object is manipulated, all we did in the above was to set it to *free*. The reason is that before it was actually inserted into the `allqueue`, it was only accessible from the local variables of the thread that created it; it had no global existence. But inserting it into that queue was the last step in its initialization; the lock is not subsequently needed. Finally, the global queues themselves are protected from simultaneous access by the global lock.[15] In essence, using the private lock was unnecessary because the object it protects did not (visibly) exist until all modifications were complete.

It is important to note that all of this concern about locks is warranted because `pthread_create()`, as we have described it here, is just an ordinary library function. Unlike a system call, it has no guarantee that another thread will not be chosen to run while it is in the middle of its execution. (Typical implementations of system calls on single-processor hardware automatically disable interruption and a system call that needs to be interruptible must take explicit action to ensure that it is possible. This is not true of the usual function call mechanism.)

4.a.3 pthread_exit()

A process can terminate itself by calling `pthread_exit()`$_{37}$ explicitly or by returning from its initial function; but as we have just seen, `pthread_create()` arranges that `pthread_exit()` is called in the latter case, too, by making or faking the appropriate stack entry just below the initial function's stack frame. In either event, execution of `pthread_exit()` proceeds as follows:–

1. Acquire the lock on the current pthread object.
2. Store the final value for the current thread, which we received as an argument, in the return value field of the pthread object. This will be needed by the thread joining this one.
3. Execute any cleanup handlers that calls to `pthread_cleanup_push()`$_{122}$ have registered for this thread$\Rightarrow^{6.3}_{121}$.

[15] Note that this informal argument would not be quite correct on many multiprocessors, because of cache coherence problems$\Rightarrow^{C.2}_{277}$. The difficulty is that without additional precautions, different processors might still see different cached *copies* of the memory holding the pthread object—or any objects that are supposed to be visible in both the old and new threads.

4. Call the destructors for any thread-specific data belonging to this thread$\Rightarrow^{6.2}_{114}$.
5. If the current thread's join pointer is `NULL`, no other thread has (yet) joined on this one. Set its state to *dead* and yield the CPU to any other thread that wants it.
6. Otherwise, the join pointer points to a thread that wishes to join with us.
   (a) Acquire the global lock.
   (b) Set the state of the joining thread from *joining* to *running*.
   (c) Set the current thread's state to from *running* to *dead*.
   (d) Yield the CPU to any thread that wants it, releasing the global lock.

   The effect of this will be to awaken the joining thread, which is already blocked inside a call to `pthread_join()`. It will set our thread's `detached` flag as soon as it has completed its side of the synchronization operation$\Rightarrow^{4.a.4}_{54}$.
7. By this point our thread is dead, and will never run again. If it is not already detached, another thread should eventually join with it and set its detached flag. Once the scheduler notices a thread that is both *dead* and *detached*, it removes it from the various global queues and frees all the memory it consumes, and the dead thread is now finally buried.

There are two reasons why the final cleanup is done in the scheduler and not within `pthread_exit()` itself. The first is that the thread cannot be completely freed until any thread that is going to join with it has done so, and in general the joiner may not have arrived at its call on `pthread_join()` before our thread tries to terminate itself. The second and far more profound reason is that since a thread calls `pthread_exit()` to terminate *itself* it is actually *running* while the function is executing. This is quite different from the case of `pthread_create()`, which is called by another thread on the new thread's behalf. The important consequence is that the context that the physical CPU is using at this moment, including its stack, is the context of the thread being deallocated. Thus, if we freed the thread's memory, the process would be left with its stack pointer pointing into unallocated memory, and disaster would rapidly ensue. Thus, we must defer deallocation until some other thread—any other thread—is already running. Because our thread has at this point already been marked *dead*, there is no danger that the scheduler will reschedule it any more.

4.a.4 pthread_join()

The process of joining with a thread is fairly straightforward if the thread has already terminated; all that is necessary is to retrieve the final value and mark it detached so that cleanup can occur. If it's still running, though, some careful synchronization is needed. The function `pthread_join()`$_{40}$ works as follows:–

1. Acquire the lock on the target thread.[16]
2. Check if it is either detached, or already has a non-`NULL` join pointer. In either case it is not legal to join with this thread; release the lock and return `ESRCH`$_{271}$.
3. Check the thread's state. If it is not yet *dead*, we must wait for it to terminate before proceeding, as follows:–

[16] Note that there is a potential deadlock here: if two threads attempt to join each other, they will each attempt to acquire the same two locks in different orders. This operation itself is meaningless, however, for obvious reasons.

(a) Acquire the lock on our *own* pthread object.

(b) Make the target thread's join pointer point to our own pthread object so that it knows to reawaken us when it finally exits.

(c) Release the lock on the target thread.

(d) Set our own state from *running* to *joining*.

(e) Release the lock on our thread and yield the CPU to any thread that wants it.

(f) While our state is set to *joining*, the scheduler will ignore us. Eventually, the target thread should exit, and in the process set our state back to *running*. At that point the scheduler will once again allow our thread to run. By then the target thread will have finished executing `pthread_exit()`, and will have left its own pthread object locked, as described in the previous section.

(g) If target thread is already marked as detached, we were not supposed to join with it after all. Unlock it and return `ESRCH`$_{271}$.

4. Store the thread's saved final value in the location provided by our caller, if any.
5. Set its detached flag so that the scheduler will know it is safe to deallocate it.
6. Unlock the target thread.
7. Return 0 to indicate success.

4.b Non-posix analogs

Methods of thread creation and termination must naturally be provided by any threads package. In this section we give a brief comparative synopsis of the facilities provided by each of a number of other systems.

4.b.1 DCE threads (OSF/1, HP-UX, Windows NT, ...)

The DCE threads package, available on a number of platforms, is an implementation of an early version (draft 4) of the POSIX thread standard. As such, it is almost identical in its basic functionality with what we have already described in this chapter, even to the names of the functions and the order of their arguments. The main differences we know of are:–

1. Error reporting is handled more in accordance with Unix tradition: on failure, status-returning thread functions return `-1` and deposit an error code in the (thread-specific) `errno` 'variable'.
2. The attribute argument of `pthread_create()` may *not* be left `NULL`, though a default attribute object is available under the name `pthread_attr_default`.
3. The initial thread may *not* call `pthread_exit()` in order to terminate itself while its descendents survive. Instead it must somehow keep itself occupied in the interim, perhaps by joining with other threads.
4. Any number of threads may call `pthread_join()` with the same argument. This is tricky, because they must be certain that the thread in question has not already terminated, lest they end up attempting to join a thread that has been physically deallocated.
5. The function `pthread_equal()` is not available.
6. The error codes returned are different in some circumstances.

4.b.2 LWP (SunOS)

The SunOS LWP (Light Weight Processes) package, while it had goals similar to (but somewhat more ambitious than) those of pthreads, took a slightly different angle on at least three points: the specification of thread attributes, the passing of initial arguments, and the manner of thread termination.

Thread creation is accomplished by a call to `lwp_create()`, which is conceptually very similar to `pthread_create()`:

```
int /*lwp error code*/ lwp_create
  ( thread_t *new_thread_id
  , void (*new_thread_function)(int arg1, ..., int argn)
  , int priority
  , int flags
  , stkalign_t *stack
  , int nargs
  , int arg1, ..., int argn
  );
```

Just as with its pthread homologue, this function takes a reference parameter in which to store the new thread id, a function for the new thread to execute, arguments for the function, and various thread attributes. Unlike pthreads, the thread attributes are unpacked into several arguments: a priority (higher priorities get preference), various flag bits, and a stack (which must be provided). The argument passing conventions are also different: zero or more `int` parameters are provided in a varargs style, along with a count; the parameters are then passed to the thread function as its actual parameters. The function can thus receive as many parameters as it wants, subject to the (nominal) restriction that they all be `ints`.[17]

Thread termination in lwp reflects a rather more significant difference in philosophy. As the name suggests, at least some of the developers of the Light Weight Processes package thought of a thread as being a kind of process. Consequently, under SunOS, the C program termination function `exit()` terminates not the current *process* but only the current *thread*. To terminate the whole process abruptly, a different function is used:

```
void pod_exit(int status);
```

The term **pod**, in lwp parlance, refers to the set of all threads belonging to a single conventional process.

As with pthreads, the whole process also exits if all its threads terminate. Actually, the situation is slightly more complex in that one of the flags you can set on thread creation is `LWPSERVER`, which specifies that for purposes of deciding termination this thread doesn't count if it is not runnable. A shade more generality than in POSIX is also provided in that the default final status code is not fixed at `0`, but can be examined and changed with the functions

```
void pod_setexit(int status);
int pod_getexit(int status);
```

[17] POSIX could not have adopted this strategy, since Standard C permits variable-length argument lists only through the use of the full varargs mechanism. This is fairly pretty on the calling side but is very ugly for the receiver. Furthermore, even if the LWP solution were legal, it is still host-specific in that it assumes that the programmer knows the correspondence between the sizes of various data as they are actually passed on the stack—few programmers will really restrict themselves to `int` parameters—and this is not readily determinable.

It is a question for philosophers why `pod_getexit()` takes an argument, or what it is for; perhaps this is a typo in the man(1)page.

4.b.3 MACH Cthreads

The MACH operating system actually provides three threads packages with compatible interfaces. You select which package you want by linking with the corresponding library. Of the three packages, one (obtained with `-lco_threads`) uses simple coroutines—without preemption—to provide an implementation at the application level; one (`-lthreads`) uses MACH system threads directly; and one (`-ltask_threads`) simulates threads by using a full process for each thread with resources shared by manipulating the virtual memory mappings—with the consequent limitation that file descriptors and stack-allocated variables are *not* mutually accessible. The second of these packages is the most similar to the POSIX thread facility.

Overall, MACH Cthreads are much simpler than POSIX threads, while providing the same core functionality. The basic functions are:–

```
cthread_t cthread_fork(any_t (*func)(any_t arg), any_t arg);
cthread_exit(any_t result);
any_t cthread_join(cthread_t thread);
void cthread_detach(cthread_t thread);
cthread_t cthread_self(void);
```

These functions correspond closely to the POSIX and DCE functions with similar names; `cthread_fork()` corresponds to `pthread_create()` and, as with DCE threads, there is no equality function. As you can see, the main difference is that the calling conventions are more streamlined—or less sophisticated, depending on how you look at it.

A MACH Cthread must be joined or detached exactly once, and as in POSIX the primal thread may call `cthread_exit()` to allow its descendents to outlive it. In earlier versions you also had to call `void cthread_init(void)` at least once before using any other Cthreads functions; later versions do this automatically.

Another thing to know about MACH is that how memory is allocated in general depends on the thread implementation in use. In particular, `-ltask_threads`, which makes each thread a MACH task (approximately a process) makes stack segments *private* to each thread, meaning that `auto` storage is *not* shared, and threads cannot pass the addresses of local variables between each other (they won't point to the same things)—though they still share a common heap. The other two implementations behave like pthreads and share the whole address space between all threads.

4.b.4 GATech Cthreads

The GATech Cthreads package, which traces its lineage to the same roots as MACH Cthreads, has interpreted `cthread_fork()` as a macro that captures the *func* argument textually as a name for the thread (to be used in debugging output, for instance). The underlying functionality is split into two operations, `cthread_thread_alloc()`, which creates a suspended thread, and `cthread_thread_schedule()` which makes it eligible to run; internally, `cthread_fork()` just calls

```
cthread_thread_schedule(cthread_thread_alloc(func, arg, node))
```

Splitting things this way allows various thread parameters to be set *before* the thread is dispatched (and provides an alternative solution to the problem addressed by the attributes objects of pthreads).⇒??.

GATech Cthreads also adds one more parameter to `cthread_fork()`, specifying the **node** (typically a physical processor or a host operating system process) on which the thread should run, as an integer or one of the magic constants `N_CURRENT` (the calling node) or `N_ANYWHERE` (any node available).

GATech Cthreads specifies that when the primal thread terminates, the process exits. Thus it too faces the problem of how to keep the primal thread busy until all other threads are finished. A direct solution is provided in the form of the function `void barrier(void);` when invoked by the primal thread it suspends until no other threads exist. (If called from another thread it just blocks forever, which isn't especially useful.)

Memory allocation under GATech Cthreads is very different from the other threads packages we describe, since it does not assume that *any* storage is shared between nodes unless it is allocated by the thread-specific routine `memory_alloc()`??. Stack-allocated storage is in particular never shared between threads.

The storage of global variables is thus not predictably shared in GATech Cthreads, *but* each thread's copy of a global (or `static`) variable is guaranteed to appear at the same address. The function `int cthread_publish(void **)` can be used to propagate a `void *` value to all *copies* of a given address, which will have the right effect if this operation is considered to be an integral part of assigning to a shared global variable.

4.b.5 OS/2

Threads under OS/2 are created with the function `DosCreateThread()`:

```
DosCreateThread
  ( PTID id
  , PFNTHREAD  addr
  , ULONG arg
  , ULONG flags
  , ULONG stackSize
  )
```

This corresponds to `pthread_create()`, the main difference being that the thread attribute argument has been unpacked into an unsigned long full of flags and an explicit specification of the necessary stack size. Just as with the POSIX function, *`id`* will be made to point to a thread executing `(*addr)(arg)`; OS/2 has the luxury of knowing that an unsigned long and a pointer are the same size, so having *`arg`* be a `ULONG` and not a `void *` results in no loss of generality. Again like POSIX, the return value is an error indication. It can be one of `NO_ERROR`, `ERROR_NOT_ENOUGH_MEMORY`, `ERROR_INTERRUPT`, `ERROR_PROTECTION_VIOLATION` and `ERROR_MAX_THRDS_REACHED`.

Where POSIX mandates separate `exit()` and `pthread_exit()` functions, OS/2 folds them together into a single exit function, `DosExit()`:

```
VOID DosExit(ULONG actionCode, ULONG resultCode)
```

where the requested action is thread termination if *`actionCode`* is `0`, but process exit if *`actionCode`* is `1`. The parameter `resultCode` is used to specify a final value, but here the symmetry between the two uses of the call falls apart: the final value is only recoverable when the whole process is exiting. As in DCE threads, there is a restriction that the primal thread not itself terminate except by full process exit.

If, analogously to joining in POSIX, one thread wishes to await the termination of another, it calls `DosWaitThread()`:

`DosWaitThread(PTID id, ULONG waitOption)`

The *waitOption* parameter specifies, if it is `0`, that the caller wishes to be delayed until the thread exits; or if it is `1`, that the caller should return with the 'error' `ERROR_THREAD_NOT_TERMINATED` if it is still running. In either case, return is immediate if the thread has already exited. The argument *id* is a pointer to the identifier of the thread to wait for; if it points to the value `0` instead, then whichever thread exits next is accepted, and the location pointed to is updated to reflect the actual thread found. The function returns one of `NO_ERROR` or `ERROR_INVALID_THREADID`; the latter might imply that the current thread is trying to wait on itself or the primal thread, or that the supplied thread value is mutant binary garbage from outer space.

No functions equivalent to `pthread_detach()`, `pthread_equal()` or `pthread_self()` are provided.

4.b.6 Plan 9

Plan 9 is the back-to-basics redesign of Unix from Bell Labs. Its greatest strength relative to its ancestor (aside from being an engineered system rather than one that has, by now, accreted over a long period of time) is that it takes virtual memory technology as a basic resource.

Presuming that memory mapping is cheap, Plan 9 provides no independent notion of a thread at all, relying on its process model to do the trick. Plan 9 processes are a little cheaper than Unix ones, however, and the Plan 9 `rfork()` system call provides class-by-class control over resource sharing between parent and child processes:

`int rfork(int flags)`

This system call creates a new process or changes the attributes of an existing one. The *flags* argument is the bitwise **or** of any of a large number of constants that jointly specify the operation required. If `RFPROC` is among the flags, `rfork()` creates a new process, rather than just modifying the existing one; if `RFMEM` is present then the `data` and `bss` segments are *not* copied—though the stack always is—when this occurs. Setting both then has an effect rather like creating a new thread, except—as with the process-simulation option of MACH Cthreads—variables allocated on the stack are not made shareable.

Flags also control whether other process resources are shared, copied, or zeroed. The resources so controlled include the name space (analogous to the Unix file tree, but process-local), the environment, and the file descriptor table. Flags also control whether notices of process termination are delivered—a distant cousin of detach state control—and support a kind of process group notion.

Chapter 5

Synchronization Facilities

*[[Use the phrase **serialize access** more often! Maybe just "conditions" for "condition variables"? Text for examples. Chappendices. "Mutual exclusion locks" best terminology. We talk about spurious wakeups too much now! Mention volatile - it seems like it should be relevant even if it isn't.*

Check how the standard has signal manipulate the mutex.

Add sections about the other synchronization mechanisms from posix.4?]]

5.1 Introduction

In order for multithreaded programming to improve on the traditional Unix process model, threads must be able to share data safely and synchronize with each other efficiently. As we saw in section 1.3, shared memory alone is not enough; special techniques and sometimes special hardware support are needed to allow even basic synchronization. Partly this is because processors—and programs—can be made to run much faster if it is *not* assumed that most operations are synchronization points; partly it is because much of the programmer's craft lies in the construction of *abstractions,* and it would take a much higher level language than C to give the compiler enough information about the ideas behind a program, for synchronization interlocks to be determined automatically.

To recap, the basic synchronization problem arises when two threads, sharing a variable, try to use it at the same time. A number of things can go wrong. If the variable is non-atomic—a typical example is a C `struct`, though on some processors you can see this effect even with an `int`[1]—one thread might read or write it while another thread is in the midst of changing its value. In the former, read/write, case, the reading thread may see a completely inconsistent value for the variable (see figure 5.1), while in the latter case of a write/write conflict, the variable itself may become inconsistent (5.2). Even with an atomic value things can go wrong; consider two

[1] Standard C does actually mandate an integral type `sig_atomic_t` which is required to be exempt from the problem. It was introduced to deal with asynchrony between programs and exception handlers; as we shall see, however, pthreads provides much better solutions to synchronizing both data access and signal handling.$\Rightarrow^{10.5.2}_{195}$

`static struct {int a, b;} s = {0, 0}, v = {1, 1}, x;`

thread A	*thread B*	*effects*
`...s = v;...`	`...x = s;...`	
`ld r0,[v.a]`		
`st r0,[s.a]`		
	`ld r0,[s.a]`	
	`st r0,[x.a]`	
	`ld r0,[s.b]`	
	`st r0,[x.b]`	→ `x` is now `{1, 0}` !!!
`ld r0,[v.b]`		
`st r0,[s.b]`		`s` is now `{1, 1}`

Figure 5.1: Read/write conflict

`static struct {int a, b;} s = {0, 0}, v = {1, 1}, x;`

thread A	*thread B*	*effects*
`...x = s;...`	`...x = v;...`	
`ld r0,[s.a]`		
`st r0,[x.a]`		
	`ld r0,[v.a]`	
	`st r0,[x.a]`	
	`ld r0,[v.b]`	
	`st r0,[x.b]`	`x` is now `{1, 1}`
`ld r0,[s.b]`		
`st r0,[x.b]`		→ `x` is now `{1, 0}` !!!

Figure 5.2: Write/write conflict

threads which try to increment a shared `int`. It may happen that each of them copies the variable into a private register before either of them updates or saves the result: the net effect is that the variable only gets updated once (5.3). Few modern processors increment a value in memory with a single, atomic[2] instruction, even if you use the "`++`" operator. Worst of all, because it defeats even the most clever programming, hardware caches and compiler optimizations can completely change the numbers and positions of loads and stores to main memory by keeping copies of variables or parts of variables when it happens to be convenient—a vital performance optimization that

[2]**Atomic**: having, or appearing to have, no separable 'steps', as seen from a given perspective. The perspective we are generally interested in here is that of user-mode software; we consider something 'atomic' if there is no experiment that you can perform using the usual software development tools on your system that can demonstrate otherwise. Of course if you observed the hardware by physical means you would see no atomic instructions at all! Even the operating system sees more internal structure to program execution than user-mode code does. In general, 'atomicity' is a constructed thing, and not the basic property of digital computing that we often, for the sake of simplicity, assume.

`int x = 0;`		
thread A	*thread B*	*effects*
`...x++;...`	`...x++;...`	
`ld r0,[x]`		`x` is 0
`add r0,1`		
	`ld r0,[x]`	
	`add r0,1`	`x` has not changed
	`st r0,[x]`	`x` is now `1`
`st r0,[x]`		→ `x` is still `1`, not 2!!!

Figure 5.3: Update/update conflict

on some machines could slow programs down by a factor of a hundred if disabled—and according to the C standard, quite legal. The easiest way of solving these basic synchronization problems is to provide a mechanism to mark the points in the program where memory synchronization is necessary.

Of course, such synchronization *does* involve a performance cost. If this operation simply suspended all threads but the one in the 'critical section', a great deal of performance would be lost—on a physical multiprocessor, the speed reduction would be catastrophic. Thus pthreads provides a special **mutex** *data type* which is used to indicate when a shared data structure is in use. You can use as many mutexes as you like; only threads trying to use the same mutex lock each other out. Where there is little contention or a lot of subtly inter-related global variables to be protected, a single mutex may be appropriate; but typically a better style, especially for new code, is to associate mutexes with particular data structures. Note that unlike systems that provide **global** exclusion (in an appendix to this chapter you'll see that this includes DCE threads and OS/2$\Rightarrow^{5.a}_{102}$), this arrangement means that owning a mutex does *not* prevent a thread's pre-emption: threads *not* contending for the particular mutex in question are completely unaffected.[3]

Another thing that happens from time to time when programming with threads is that a thread needs to *wait* for some internal event to occur: for a resource to become free, for example, or for another thread to reach a particular stage of its processing. Of course, once the problem of synchronizing access to variables has been solved, it is possible to designate a variable to represent the condition, and have the waiting thread just sit in a loop, constantly checking for the right value to appear in it. But this may be immensely wasteful—CPU cycles that could be doing useful work are being used to evaluate an empty loop. Evidently some mechanism is needed to allow a thread to block on an abstract, programmer-defined event, much as it is conventionally possible to block waiting for a system-defined one, such as the arrival of input.

This chapter presents the main facilities POSIX provides for synchronization between threads: **mutual exclusion locks,** which are normally used to protect access to data structures; **condition variables,** which provide an efficient means for threads

[3]Except possibly as regards effective thread priority.$\Rightarrow^{9.5.1}_{173}$

to await abstract events; and the **once** facility, a (perhaps redundant) mechanism designed especially for the safe initialization of modules shared between threads. Other synchronization mechanisms, including **signals**$\Rightarrow^{10}_{183}$, **semaphores**$\Rightarrow^{??}_{??}$ and **file locking**$\Rightarrow^{12.3.1}_{224}$ are available for situations involving multiple *processes;* in practical terms this is a separate topic, and the interactions of these mechanisms with pthreads are discussed later in the book.

5.2 The posix synchronization model

Logically, any inter-thread synchronization error must have two ingredients: multiple threads, and changing memory contents. The basic idea behind the pthread memory access model, then, is that each location in memory may safely have, at any one time, *either* one thread modifying it *or* any number of threads that read it only. Either of these situations alone is conceptually safe.

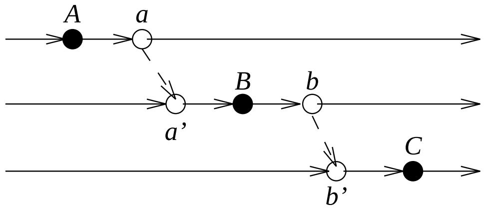
The phrase *at any one time* requires some attention, however. There are two problems, one local—that the hardware or the compiler may re-order memory operations, altering the temporal relationships of points in the source program—and one global—that threads execute independently and often at different rates. Consequently, a 'single moment' in one thread may be a long execution sequence in another, and a single 'sequence' of events may even have a different order as seen from two different threads.

POSIX deals with this by using a notion of events *definitely* preceding one another. Within a single thread this is defined just by the usual semantics of C programs, but *between* threads only a small number of **synchronizing functions** have the property of ensuring that an event in one definitely precedes an event in another. Only at these well-defined points, and only between the threads explicitly involved, is the transfer of 'ownership'—of which thread, if any, is allowed to modify a memory location—guaranteed. And in consequence, only at these points does the implementation have to use special synchronizing procedures. The hand-off process is called **serialization of access** to the location, since it takes accesses that would not otherwise have had a defined sequence and *makes* them appear serial—and mutually atomic.

Of course, a thread can have the *right* to modify memory without actually doing so; this is important, because it means that our notion of definite precedence can be extended to a partial order (that is to say, if A definitely precedes B and B definitely precedes C then A definitely precedes C—see figure 5.4), which is certainly what we want for reasoning about the order in which things (might) happen.

The complete list of synchronizing functions, according to the POSIX standard,[4] is given in figure 5.5, which is arranged so that calls to the functions and other events in the left column *definitely precede* returns from those in the right (assuming, of course, that their arguments match—that we're talking about the same process, the same thread, the same mutex and so on).

[4]It is probably an oversight of the standard that `pthread_once()`$_{89}$ is not listed; it is certainly the intention that any initializations it protects be universally visible.



Here $A \ll a\text{–}a' \ll B \ll b\text{–}b' \ll C$. Thus $A \ll C$ and 'ownership' of an lvalue can pass from A to C.

Figure 5.4: Transitivity of synchronization operations.

Process-related		
life cycle	`fork()`??	*(process startup)*
	(process exit)	`wait()`?? `waitpid()`??
synch.	`sem_post()`??	`sem_trywait()`?? `sem_wait()`??
Thread-related		
life cycle	`pthread_create()`$_{33}$	*(thread startup)*
	(thread exit)	`pthread_join()`$_{40}$
exclusion	`pthread_mutex_trylock()`$_{72}$ `pthread_mutex_unlock()`$_{75}$	`pthread_mutex_lock()`$_{72}$
synch.	`pthread_cond_signal()`$_{86}$ `pthread_cond_broadcast()`$_{86}$	`pthread_cond_wait()`$_{83}$ `pthread_cond_timedwait()`$_{83}$

Figure 5.5: Data synchronizing functions

Thread-specific synchronization facilities The functions in the upper panel of figure 5.5 pertain largely to processes; the topic of the remainder of this chapter is the functions in the lower panel, those that are specifically thread-related.

Because pthreads is a library rather than an extension to the C language itself, synchronization is—like threads—implemented with opaque data types manipulated by functions rather than with new syntactic constructs (as is done in many language systems that were designed with multiple threads from the start).

In fact, as we mentioned in the previous section, there are two main types used in thread synchronization: **mutexes**, of type `pthread_mutex_t`$_{68}$, used to provide *mut*ual *ex*clusion; and **condition variables**, of type `pthread_cond_t`$_{80}$, which represent conditions that threads can wait on. The ideas behind these objects are simple: mutexes are supplied with **lock** and **unlock** operations arranged so that at most one thread can have a given mutex locked at a time. A condition variable represents an event that can be awaited and signaled to hold, and that is associated closely with a mutex that can limit the chance that another thread negates the awaited situation before the waiting thread does its job.

Figure 5.6: The 'small room' model of the mutex

One traditional way of visualizing mutexes and condition variables is in terms of, um, small rooms (see figure 5.6). Imagine the mutex itself as a lockable room containing the Resource that needs to be protected against conflicts arising from multiple simultaneous access (1). When you 'lock' the mutex you, in effect, take the key down off the hook and lock yourself into the room: you have the key with you, so no one else can come into the room and disturb your use of the Resource. Late-comers are just forced to queue outside the door (2). The 'unlock' operation involves letting yourself out and replacing the key so that others can take their turn.

(In view of the fact that this is C, and in practice you can do what you like, readers of fantasy novels may instead want to think of a mutex as a particular kind of chalk circle drawn on the floor. Nothing *stops* you from stepping over the line. But—trust me—you really, *really* don't want to....)

A condition variable is a room with a couch in a conveniently adjoining location. Suppose, for instance, that while using the Resource and doing a crossword puzzle, you have got stuck because you can't get 23 Across—the obvious solution is to take a nap until someone else provides the answer; but it would be dreadfully antisocial to fall asleep in the mutex! So you use the condition variable instead, leaving the mutex unlocked behind you (3).

Once someone else using the mutex has done a little more work on the crossword puzzle, they can, if they are polite, pass by the couch and rouse a sleeper, who can then drop by the small room to see if 23 Across was among the newly solved clues. Less polite but more amusing is to **broadcast** the signal: this amounts to setting off the large alarm bell on the condition variable wall (4), and will result in all the sleepers for this condition variable awakening and rejoining the queue at once—if there are any, of course.

In the POSIX model there is also assumed to be a cat, which wanders around making noises and sitting on people and generally causing spurious wakeups.

Now for a small caveat: the operations involving both a mutex and a condition variable are *atomic*. In particular, when you wait for a condition, no other thread can get the mutex before you fall asleep; when you signal a condition, no other thread can get the mutex before the signal is delivered; and when you awaken from waiting on a condition you have no choice but to go re-enter the small room, though someone else may slip in ahead of you, leaving you to queue.

5.3 Mutexes

Exploiting mutexes is exceptionally simple. In figure 5.7 we show how the program fragment in figure 5.3 should have been written.

```
int x = 0;
static pthread_mutex_t x_m = PTHREAD_MUTEX_INITIALIZER;
```

thread A	*thread B*	*effects*

`...`
`pthread_mutex_lock`
`  (&x_m);`
`x++;`
`pthread_mutex_unlock`
`  (&x_m);`
`...`
	`...`
	`pthread_mutex_lock`
	`  (&x_m);`
	`x++;`
	`pthread_mutex_unlock`
	`  (&x_m);`
	`...`	

lock `x_m` *–ok*		`x` is initially `0`
`ld  r0,[x]`		
`add r0,1`		
	lock `x_m`	thread B waits...
`st  r0,[x]`	`...`	
unlock `x_m` ⇒	...*–ok*	...until here
	`ld  r0,[x]`	`x` is 1
	`add r0,1`	
	`st  r0,[x]`	`x` is now, correctly, 2
	unlock `x_m`	

Figure 5.7: Correctly synchronized increment

When this code is run, there is no chance for `x` to end up with the wrong value. Since changes to `x` are only performed by threads that **own** the mutex `x_m` (*i.e.*, have it *locked*), and since the system ensures that this can be true of at most one thread at a time, it follows that the updates cannot overlap.[5] In the event that two threads arrive simultaneously at the calls on `pthread_mutex_lock()`$_{72}$, one of them will be arbitrarily selected to proceed, while the other is blocked until the first is finished and the mutex once more becomes free (if it is a problem for your thread to block, `pthread_mutex_lock()` has a variant, `pthread_mutex_trylock()`$_{72}$, that returns an error in this case instead).

[5] If you're wondering what happens if the compiler copies or re-orders things so that this reasoning doesn't hold, it's not allowed; the purpose of these functions is to require serialization, and the compiler must, at least in effect, abide by this. Technically speaking, like thread creation and termination, mutexes provide **synchronization points**⇒??. Conventional optimizing compilers are too stupid and/or cautious ever to move anything past a function call, anyway, though the situation is changing.

Note that because there is no special syntactic construct associated with mutual exclusion, there is no explicit record in the source program of what data structure(s) the mutex protects—either for the compiler or the programmer. Indeed, strictly speaking, it is the programmer's convention of acquiring a mutex before updating a variable that protects it, not the mutex itself. Nothing actually *stops* you from going ahead and using a variable without acquiring its associated mutex; your program may just quietly give the wrong answers (or then again, it may crash and burn). This makes it vitally important to document the relationships between mutexes and data in your code and follow your locking conventions scrupulously.

Of course, such flexibility can be exploited by the careful programmer—for instance, mutexes can be used to protect multi-part dynamic data structures as easily as unitary static ones—but synchronization is notoriously hard to reason about, and in general it is best to err on the side of caution, using stylized code that is sure to be correct.

Something that can go wrong when you *do* use correctly-associated mutexes is that your threads can **deadlock**—that is, they can get into a state where they are waiting for something that can logically never happen. The simplest example, and the easiest to avoid, is when a thread tries to lock a mutex that it already owns, and ends up waiting for itself. More complicated examples involve several threads and several mutexes—A waits for B which waits for C,..., which waits for A. The simplest rule to avoid trouble is that if any thread tries to lock mutex a while it owns mutex b, then no thread must try to lock b while it holds a. That is, you want to have some order of locking, over *all* mutexes, that you obey throughout your program. We cover this topic in more detail below⇒$^{5.6.2}_{93}$.

5.3.1 Mutual exclusion type: `pthread_mutex_t`

Mutual exclusion variables, **mutexes,** are represented in POSIX by members of the type `pthread_mutex_t`, defined in `pthread.h`$_{273}$. The standard says nothing about the structure of this type, which is normally manipulated through a pointer, and the only operations you can safely perform on mutexes are the ones provided by the pthreads library. In particular, unlike thread IDs, you cannot even assign them meaningfully (though of course you may freely manipulate pointers to them). This can be inconvenient when initializing data structures containing mutex fields, but they are the most performance-critical user-visible part of the thread system, and this decision allows each implementation to choose whether mutexes should be pointers or values or even special hardware resources according to the capabilities of the target machine, while still allowing for static initialization. Mutexes are expected to see heavy use providing short-term protection of an application's crucial shared data structures.

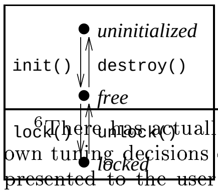
Warning: Note that strictly speaking the fact that assignment is undefined for the `pthread_mutex_t` *type*—and not just for the initialized *values* of that type—makes it illegal to initialize a dynamically allocated structure containing a mutex field by assignment to the whole structure, even if the source of the assignment is itself uninitialized (or statically initialized and as yet unused). Only on machines offering specialized hardware or compiler support is this likely to be a problem when the source of the initialization is an initializer, an *unused* statically initialized structure, or an uninitialized one; but portable—or even reasonably cautious—code cannot assume that this superficially harmless and labor-saving practice is safe.

Attributes Mutual exclusion variables all provide the same basic service, but they are not all the same. On a multiprocessor, where wasting a little CPU time may be less costly than switching between threads, the simplest exclusion method may really be an empty loop that constantly retests a field in the mutex; but for those mutexes that stay locked for a long time it may still be best to reschedule a waiting thread. Or an implementation providing **shared memory** may allow for mutexes that can operate between the threads of different processes; these may be more complicated than is needed for a process-local mutex. Or debugging mutexes might be provided that check for more error conditions than is consistent with peak performance.

In order to allow for this kind of variation, additional information about a mutex is specified when it is initialized, using an **attributes object**. Like the other attributes objects in pthreads, these are designed for extensibility: each implementation can specify which attributes its mutexes support, whether standard and portable or implementation-specific. The details are given in a later chapter$\Rightarrow^{\S 8.4}_{147}$; for now we discuss only those things that are common to all mutexes. For many applications the default (once again specified with a `NULL` attributes object pointer) will be all you need. Things have been arranged so that the default mutexes can be lean, mean ones, and there is a clever technique that good implementations may use that allows heavy-duty cases to co-exist with fast ones without performance problems; the interested reader will find a description at the end of this chapter$\Rightarrow^{\S 5.5.2}_{151}$.

Note in particular that unless attributes are used to specify otherwise, a mutex cannot be shared between threads belonging to multiple processes, even if it is allocated in memory shared between them.

Initialization Just declaring a mutex is not enough; before it can be used, it must be initialized. This allows the implementation to do any necessary setup, as well as providing an opportunity for the mutex's attributes to be specified. Because mutexes can, if you wish, be allocated dynamically, there is also a **destruction** operation that un-initializes a mutex, restoring it to its original, unusable state—and ensuring that no resources will be lost if it is `free()`d. The fact that a mutex that has been destroyed can later be re-initialized from the beginning may also be useful occasionally, either because you want to recycle a data structure, or so that you can *change* the attributes of a mutex, something which for fairly obvious reasons cannot be done while the mutex is in use.[6]



The rules about initialization are naturally quite strict: you must not initialize a mutex while it is initialized, use or destroy a mutex while it is not initialized, or destroy one while it is locked.

[6] There has actually been some research done recently on **adaptive** mutexes, which change their own tuning decisions over time—and, yes, while they are in active use. Of course these are typically presented to the user as *default* mutexes that achieve higher performance than they could using a single strategy, rather than as objects allowing or requiring additional dynamic configuration.

5.3.2 Initializing a mutex

Mutual exclusion variables have the special property (and purpose) that at most one thread can have them *locked* at a given time—assuming they have been initialized correctly. If a mutex is statically allocated and needs only the default attributes, it can be pre-initialized with the magic constant `PTHREAD_MUTEX_INITIALIZER`; dynamically allocated mutexes (whether on the stack or the heap), and ones with non-default attributes, however, must be initialized explicitly by calling `pthread_mutex_init()` before their first use. This function takes a pointer to the mutex to initialize and, a pointer to its attributes object$\Rightarrow^{8.4}_{147}$, which may be `NULL` if the default attributes are acceptable. The function returns `0` if it succeeds, or an error code if it detects a problem.

Once again the attributes object specifies details of how the mutex should behave; notably you can ask for a mutex that works between process, in shared memory$\Rightarrow^{8.4}_{148}$; and you can control how it interacts with thread priority scheduling$\Rightarrow^{9.5.1}_{173}$.

`pthread_mutex_init()`, `PTHREAD_MUTEX_INITIALIZER` **11.3.2**

```
#include <pthread.h>

pthread_mutex_t static_mutex = PTHREAD_MUTEX_INITIALIZER;

int /*errno*/ pthread_mutex_init
  ( pthread_mutex_t *new_mutex
  , const pthread_mutexattr_t *new_mutex_attributes
  );
```

`pthread_mutex_t`$_{68}$ *`static_mutex`*
: A *static* mutex variable to be initialized with the default attributes.

`pthread_mutex_t`$_{68}$ `*`*`new_mutex`*
: Pointer to a mutex variable to be initialized dynamically.

`const pthread_mutexattr_t`$_{138}$ `*`*`new_mutex_attributes`*
: Pointer to the attribute structure giving the properties wanted for the mutex $\Rightarrow^{8.4}_{147}$. If this argument is `NULL`, default attributes are used.

The function `pthread_mutex_init()` initializes a mutex and makes it available for use. It is undefined what happens if you try to initialize a mutex that is already initialized, or to use one that is not; if you need to change the attributes of a mutex you can do it by passing it to `pthread_mutex_destroy()`$_{72}$ and then re-initializing it with `pthread_mutex_init()`.

A statically allocated mutex (one declared `static` or `extern`) may be initialized at compile time, according to the default attributes, with the static initializer `PTHREAD_MUTEX_INITIALIZER`. The effect is the same as that produced by `pthread_mutex_init()` except that no error checking is done. For dynamically allocated mutexes, however, you must call `pthread_mutex_init()` explicitly.

In either case the newly initialized mutex object will be *unlocked*$_{??}$.

The value returned by `pthread_mutex_init()` is an integer containing an error number⇒$^{A}_{271}$:

On success `pthread_mutex_init()` returns `0` after initializing the mutex object pointed to by the first argument.

On failure it returns an error code. The following codes may be returned:

- `EINVAL`$_{271}$ A value specified by *`new_mutex_attributes`* is invalid.
- `EBUSY`$_{271}$ It was detected that the mutex is already initialized.
- `ENOMEM`$_{271}$ The process lacks the memory to initialize another mutex.
- `EAGAIN`$_{271}$ The process lacks some other necessary resource.

In the case that `EINVAL` or `EBUSY` is returned it is guaranteed that the mutex is unchanged from its original state. *[[Of the other two cases of failure, the standard says nothing!!!]]*

The reason that two separate systems are provided to initialize mutexes is that the limitations of the C preprocessor and the rules for initializing structures greatly restrict the flexibility of macro-based initialization. In particular there is no obvious way within the confines of Standard C to build static structure initializers that nonetheless allow vendor-specific extensions to the range of attributes available. Breaking attributes out into a separate data structure manipulated by functions solves this problem, at the cost of making static initialization of objects with complex attributes impossible.

An alternative mechanism is provided for simple statically initialized mutexes because there must be a good way of ensuring that data structures—including *non*-default mutexes—are initialized only once. Other solutions to this problem are possible, including use of the **once only**⇒$^{5.5}_{89}$ mechanism described later in this chapter, or doing all initialization from the primal thread before any calls are made to `pthread_create()`; but statically initialized mutexes provide a solution that is simple, uniform, powerful, and appropriate for use in library functions that must be ready whenever they are called. A dynamically initialized mutex cannot be used to control access to initialization code, because it too would need to be initialized exactly once—a classic chicken-and-egg problem.

5.3.3 De-initializing a mutex

The most important reason there is an operation to destroy a mutex is so that any resources it consumes can be returned to the system. This may be doubly important in the case of a mutex, because, depending on how they are implemented, mutexes may contain data structures belonging to the threads implementation that would cause mayhem if unexpectedly overwritten—as might happen if they were freed without informing the system. The other reason for destroying mutexes is to make them uninitialized, so that they can be re-initialized by `pthread_mutex_init()`, perhaps with different attributes.

It is not defined what happens if you try to destroy an uninitialized mutex, or one that is still locked by any thread.

pthread_mutex_destroy() **11.3.2**

```
#include <pthread.h>

int /*errno*/ pthread_mutex_destroy(pthread_mutex_t *mutex);
```

pthread_mutex_t$_{68}$ *mutex
Pointer to the mutex to be 'destroyed' (*i.e.* de-initialized).

Returns the mutex to its uninitialized state. It is undefined what happens if the mutex is currently *locked*??, or not initialized. It is also undefined what happens if you later try to use a destroyed mutex for anything other than reinitializing it.

Returns an integer containing an error number.$\Rightarrow^{A}_{271}$

On success pthread_mutex_destroy() returns 0.

On failure it returns an error code and leaves the mutex unchanged:

EINVAL$_{271}$ The mutex argument was invalid.

EBUSY$_{271}$ The mutex is either locked or is being used by another thread.

5.3.4 Locking a mutex

The primary purpose of mutexes is to support local mutual exclusion through **locking**.[7] Two functions are provided to lock a mutex (which must, of course, already be initialized$\Rightarrow^{5.3.2}_{70}$): pthread_mutex_lock() and pthread_mutex_trylock(). They differ only in what happens if their only argument, the mutex in question, is locked already; pthread_mutex_lock() will block the calling thread until the mutex becomes available, while pthread_mutex_trylock() will return a (benign) error. In either case, a successful return—one that returns the status code 0—indicates that the calling thread now owns the mutex in demand.

pthread_mutex_lock(), pthread_mutex_trylock() **11.3.3◇**

```
#include <pthread.h>

int /*errno*/ pthread_mutex_lock(pthread_mutex_t *mutex);
```

[7] We shall see later that they can also be used to affect[8] thread scheduling, but perhaps it is a bit of a hack to use them specifically for this purpose.$\Rightarrow^{??}_{??}$

[8] That's *a*ffect, 'impact', not *e*ffect, 'cause'. :-)

```
int /*errno*/ pthread_mutex_trylock(pthread_mutex_t *mutex);
```

`pthread_mutex_t`$_{68}$ `*mutex`
Pointer to the mutex to be locked by the calling thread.

Both these functions lock the specified `mutex`. If they return successfully, the mutex is now owned by the calling thread. If `mutex` is already locked, `pthread_mutex_lock()` will block until it becomes available, while `pthread_mutex_trylock()` will return immediately with a (reliable, benign) error.

It is undefined what happens if a thread that already owns a given mutex tries to call `pthread_mutex_lock()` on it (the thread might block forever; it might report an error; or something even more awful might happen); but `pthread_mutex_trylock()` will simply report that the mutex is unavailable. It is undefined what happens if `mutex` is uninitialized.$\Rightarrow^{5.3.2}_{70}$

Each function returns an integer containing an error number$\Rightarrow^{A}_{271}$:

On success they return `0`. The mutex is now locked by the current thread.

On failure they returns an error code and leave `mutex` as it was:

- `EBUSY`$_{271}$ *(`pthread_mutex_trylock()` only)* The mutex is already locked by some thread. This may be a normal occurrence.
- `EINVAL`$_{271}$ Either the mutex argument was invalid, or the calling thread's priority is higher than the mutex's current priority ceiling and the mutex was initialized with the **protocol** attribute `PTHREAD_PRIO_PROTECT`$_{175}$.$\Rightarrow^{??}_{??}$
- `EDEADLK`$_{271}$ *(`pthread_mutex_lock()` only)* It was detected that the *current* thread must perform some action before the mutex can become available. It would therefore cause a deadlock if the thread waited for it.$\Rightarrow^{5.6.2}_{93}$ Perhaps the calling thread already owns this mutex, or a mutex that blocks the thread that does.

The functions `pthread_mutex_lock()` and `pthread_mutex_trylock()` are **synchronizing functions**$\Rightarrow^{5.2}_{64}$: the successfully locking thread's view of memory is consistent with that of the last thread to unlock this mutex.

The important thing to realize about mutexes is that they are, like threads, a lightweight construct designed to be cheap enough to use heavily, even at the expense of flexibility. Although they can be used for other tasks, their main intended use is to protect data structures from being accessed while they are being updated, something that is assumed to take a short time but to happen often. This explains why a number of possible features are *missing* from the basic locking operation:

- There is not a variant of `pthread_mutex_lock()` that takes a timeout bounding the time waited: such an option would increase the complexity of the mutex mechanism itself, and if waiting time is going to be a problem, you should be using another method—perhaps the condition variables described in the next section.
- Although an implementation is allowed to report when a **deadlock** occurs—when you are trying to lock a mutex owned by a thread that is in some way

waiting for you to do something—it is not obliged to do so. Detecting deadlocks is possible, but relatively expensive.

- No provision is made to find out which thread presently has an unavailable mutex locked.

- No attention is paid to the special case of a thread trying to re-lock a mutex it already owns. Other ('**counting**' or '**recursive**') versions of mutexes count the number of times a mutex has been locked by a thread and require that it be unlocked as many times before it is actually released, but this again is an unnecessary complexity.

- A mutex can only be unlocked by the thread that locked it. This means that mutexes cannot be used as **binary semaphores**$\Rightarrow^{1.3}_{9}$ for inter-thread signaling—for instance, a thread cannot attempt to re-lock a mutex it already holds, awaiting its unlocking by another thread. (This restriction has the unfortunate side-effect that a thread cannot pass ownership of a (locked) mutex to a new thread that it creates—or another thread that joins it when it exits—even though the old and new threads are otherwise fully synchronized.)

Of course, an implementation might provide these facilities if it wanted to, and the mutex attributes mechanism is there to provide a uniform way to request support for any such variants as might exist locally.

The method actually used to implement mutual exclusion will vary from system to system, but it is likely to take one of two basic forms. If there are as many physical CPUs as threads, the best implementation of `pthread_mutex_lock()` may be based on wasting time in a loop until `pthread_mutex_trylock()` finally succeeds. If there is really only one CPU shared among all threads, on the other hand, this would be stupid, because the computational effort being wasted in the loop could have been used by the thread holding the lock (or whatever is, in turn, holding it up). Thus instead of 'spinning' in an empty loop, the frustrated thread might invoke `sched_yield()`$_{172}$ to give up its share of the CPU to a thread with something useful to do, possibly even removing itself from the queue of threads waiting to run in the meantime.

In either case, the formal rule that the implementation must follow is that just because one thread is blocked, other threads should not be indefinitely prevented from making progress.

How locking itself is performed—how the synchronization primitives themselves synchronize—depends on intimate details of the hardware and is discussed in the appendix to this chapter.$\Rightarrow^{5.c}_{107}$

5.3.5 Releasing a mutex

Once a thread has finished using the data structure (or other shared resource) protected by a mutex, it must release its lock so that other threads get their chance. This is done by calling `pthread_mutex_unlock()`. It is undefined what happens if the calling thread doesn't own the mutex when it attempts to do this.

pthread_mutex_unlock() **11.3.3◇**

```
#include <pthread.h>

int /*errno*/ pthread_mutex_unlock(pthread_mutex_t *mutex);
```

`pthread_mutex_t`$_{68}$ `*mutex`
Pointer to the mutex to be released.

Assuming that the mutex is presently locked by the current thread, the lock is released and becomes available to other threads; if there are several threads requesting the mutex the one with the highest priority will get it (which is not necessarily the one that has been waiting longest)$\Rightarrow^{9.2}_{158}$. It is the programmer's responsibility to ensure that the calling thread really does have the mutex locked; otherwise, the effect of this function is undefined.

Returns an integer containing an error number$\Rightarrow^{A}_{271}$.

On success `pthread_mutex_unlock()` returns `0`. The mutex is no longer locked by the current thread; another thread may already have acquired it.

On failure it returns an error code:

- `EINVAL`$_{271}$ The mutex argument was invalid.
- `EPERM`$_{271}$ The mutex is not owned by the current thread.

The function `pthread_mutex_unlock()` is a **synchronizing function**$\Rightarrow^{5.2}_{64}$: the next thread to lock ***mutex*** will have a view of memory consistent with that left by the caller at the moment it invokes `pthread_mutex_unlock()`.

If more than one thread is awaiting the release of a mutex when it is unlocked, which one it is granted to depends on the scheduling policy.$\Rightarrow^{9}_{157},^{9.5}_{172}$

5.3.6 Example: mutexes

As an example of how to use mutexes and why they are important, let's look at the traditional case of a circular buffer that mediates between two threads, one a producer of data and one a consumer. The idea is to allow the producer and the consumer to work at their own rates; the producer can write into the buffer as fast as it likes, so long as the buffer is not completely full; likewise, the consumer can read from the buffer whenever it is ready, providing there are data there for it.

circular_buffer declarations

```
#include <stdlib.h>
#include <pthread.h>

typedef void *item_ty;
```

```
typedef struct
  { pthread_mutex_t mutex;       /* protects read_pos, write_pos and *buffer */
    unsigned read_pos;                       /* first FULL slot, modulo size */
    unsigned write_pos;                        first EMPTY slot, modulo size */
    unsigned size;                              /* number of slots in buffer */
    item_ty *buffer;                         /* pointer to the actual buffer */
  } *circular_buffer_ty;

extern circular_buffer_ty circular_buffer_new(unsigned n_slots);
extern void circular_buffer_old(circular_buffer_ty circular_buffer);
extern item_ty circular_buffer_read(circular_buffer_ty cb);
extern void circular_buffer_write(circular_buffer_ty cb, item_ty item);
```

circular_buffer (continued) — allocation

```
circular_buffer_ty circular_buffer_new(unsigned size)
  { circular_buffer_ty cb = malloc(sizeof(*cb));

    if (cb)
      { cb->buffer = malloc(size * sizeof(item_ty));
        if (cb->buffer)
          { pthread_mutex_init(&cb->mutex, NULL);
            cb->read_pos = cb->write_pos = 0;
            cb->size = size;
          }
        else
          { free(cb);
            cb = NULL;
          }
      }
    return cb;
  }

void circular_buffer_old(circular_buffer_ty cb)
  { pthread_mutex_destroy(cb->mutex);
    free(cb->buffer);
    free(cb);
  }
```

A circular buffer of size n consists of an array of n slots, along with two markers, the read position r and the write position w, both integers modulo n. The idea is that the slots from r to w are full and waiting to be read, while those from w to r (counting around the end) are empty and waiting to be filled. Thus we can write to the buffer with `b[w++] = x; w %= n;` and read from it with `x = b[r++]; r %= n;`. The nice thing about this arrangement is that the only interaction between the reader and the writer is in checking for the full and empty conditions: the reader has to wait for more data if $r = w$, while the writer has to wait for more space if $w = r - 1 \pmod{n}$.

Because the read and write positions change *and* are shared between reading and writing threads (and, for that matter, between multiple reading threads and multiple writing threads), they must be protected by a mutex. Instead of having all buffers in the system share a single mutex, which might result in a lot of false synchronization, we place a separate mutex in each buffer structure, in the `mutex` field. The read and write functions lock this mutex before they access the other fields of the structure:

circular_buffer (continued) access functions (inefficient version)

```
void circular_buffer_write
  (circular_buffer_ty cb, item_ty item)
  { pthread_mutex_lock(&cb->mutex);
    while
      (  (cb->write_pos + 1) % cb->size
      == cb->read_pos
      )
      { pthread_mutex_unlock(&cb->mutex);
        /* give other threads a chance to read */
        pthread_mutex_lock(&cb->mutex);
      }
    cb->buffer[cb->write_pos++] = item;
    cb->write_pos %= cb->size;
    pthread_mutex_unlock(&cb->mutex);
  }

item_ty circular_buffer_read(circular_buffer_ty cb)
  { item_ty item;

    pthread_mutex_lock(&cb->mutex);
    while (cb->read_pos == cb->write_pos)
      { pthread_mutex_unlock(&cb->mutex);
        /* give other threads a chance to write */
        pthread_mutex_lock(&cb->mutex);
      }
    item = cb->buffer[cb->read_pos++];
    cb->read_pos %= cb->size;
    pthread_mutex_unlock(&cb->mutex);
    return item;
  }
```

The mutexes ensure that no thread looks at the structure while another thread is changing it.

There's a severe efficiency problem with this code, however. If there are no data available in `circular_buffer_read()`, for example, we have to wait until some have been written by another thread. Here we've done it with a while loop whose body unlocks the mutex (to give the writing thread a chance to use it), then immediately locks it and tries again. The problem is that this loop is very tight, and the desired data will almost never be available yet. Worse yet, if there is only physical processor, while we are busy testing for new data the CPU is not available to compute it.

How can we address this shortcoming? One solution would be to insert a call to `sleep()`, for example, just to slow the loop down without wasting processing cycles;

but how long should we sleep for? Another method (as we shall see later) is to call `sched_yield()`$_{172}$ to at least make sure that the other thread *does* run while the mutex is unlocked. The best answer, however, involves condition variables, and is described next.

(For an example of `PTHREAD_MUTEX_INITIALIZER`, please see section 5.5.)

5.4 Condition Variables

As we have now seen, mutexes perform an essential function in regulating safe shared access to data structures. But as became clear in the discussion of the last example, they are not quite enough; a means is needed for one thread to wait for others to establish conditions it needs in order to proceed. (In the example, a reading thread may need a writing thread to put some data into the buffer, while a writing thread may need a reading thread to remove something when the buffer is completely full.) This gap is filled by condition variables.

A **condition variable** is an object in your program that represents a **condition**—an arbitrary, user-definable event[9] in the co-operative world of the threads. The idea is to make a condition something that can conveniently and efficiently be awaited.

When waiting on a condition variable—that is, when waiting for a condition—it is presumed that the current thread has already obtained some mutex that protects the information necessary to check that the condition does not already hold. What is needed is a way to temporarily return the use of the locked data to other threads so that they can go ahead and try to cause the condition to occur. Then at the moment that the thread starts to wait it automatically and atomically 'steps aside' from this mutex, leaving it unlocked and available to other threads during its wait.

When another thread finally signals the condition—an explicit action, for maximum flexibility—the waiting thread is notified, awoken, and automatically starts trying to re-acquire the lock so that it can check that everything is indeed in order and go about its work. When the signaling thread next releases the lock, the thread that waited will get its fair chance to resume, competing in the usual way for the mutex.

All of this has the very straightforward effect that by adding a couple of condition variables to our `circular_buffer_ty` type of the previous example, we can employ read and write functions like this:

circular buffer — read (improved version)

[9] You might think that the choice of the word *condition* suggests a state of affairs rather than an event. Indeed, that's how the idea started out and how the terminology was chosen; but the term has been transferred to the signal that is propagated between threads when a *change into* the awaited state occurs. This actually allows better performance and more flexibility for the programmer—for instance a condition in the current sense can be shared among multiple interesting states of affiars—but perhaps at some cost in code transparency.

```
typedef struct
  { pthread_mutex_t mutex;       /* protects read_pos, write_pos and *buffer */
    pthread_cond_t read_ok, write_ok;               /* associated with mutex */
    unsigned read_pos;                        /* first FULL slot, modulo size */
    unsigned write_pos;                         first EMPTY slot, modulo size */
    unsigned size;                               /* number of slots in buffer */
    item_ty *buffer;                          /* pointer to the actual buffer */
  } *circular_buffer_ty;

item_ty circular_buffer_read(circular_buffer_ty cb)
  { item_ty item;

    pthread_mutex_lock(&cb->mutex);
    while (cb->read_pos == cb->write_pos)
      pthread_cond_wait(&cb->read_ok, &cb->mutex);
    item = cb->buffer[cb->read_pos++];
    cb->read_pos %= cb->size;
    pthread_cond_signal(&cb->write_ok);
    pthread_mutex_unlock(&cb->mutex);
    return item;
  }
```

which are somewhat clearer (the lock/unlock sequences now come in well-balanced pairs, for instance), and much more efficient: waiting threads now really sleep, and are only awoken when there is good reason for it. (Not shown here, by the way, are the initialization and destruction code for the new condition variables. They are handled much like the mutex itself.)

In fact things are a little more complicated, for three reasons. The first is that multiple threads can easily end up waiting on a single condition variable. In that case, which thread is awoken depends on which thread is deemed to have the highest priority. $\Rightarrow^{9}_{157}$ The second is that there are two methods of awakening threads waiting on a condition: by **signaling** it as we have described, or by **broadcast,** which simultaneously makes *all* the threads waiting on a given mutex ready to run and allows them to contend for it. The third is that (for technical reasons to do with synchronization theory and high preformance parallel hardware) there is a possibility of **spurious wakeups,** in which more than one thread is accidentally awoken by a signaling operation.

This last phenomenon has a curious effect on your code. Because it is not always the case that a thread's being awoken is due to the condition 'represented' by the condition variable, uses of `pthread_cond_wait()` must be embedded in a loop that checks the condition explicitly. This is odd, because the program would have the same effect—however much less efficiently—if the call to `pthread_cond_wait()` were replaced by the `pthread_mutex_unlock()/pthread_mutex_lock()` sequence that we resorted to in the orignal mutex example. Thus there's a sense in which condition variables have no formal semantics, though they perform an essential function in allocating processing resources sensibly (the price you pay for this is that you must remember to signal the condition as appropriate, of course; if your implementation *doesn't* issue a spurious wakeup and your other threads forget to signal, the 'better' version will wait forever). On the other hand, there are independent motivations for an explicitly checking coding style: first, that infrequent events, or ones whose exact details are known only to waiting threads, can be made to share their condition variables to good effect; and second, waiters are protected from the behaviour of other threads contending for the mutex and receiving it before the waiting thread.[10]

We have already seen that POSIX mutexes are not syntactic constructs or an implicit mechanism (as they are in some other programming languages), but rely on the explicit manipulation of the members of a specialised data type. Similarly, the association between a POSIX **condition variable** and its mutex is not syntactic—or even fixed. A condition variable becomes associated with a mutex when they are passed as the arguments of a call to `pthread_cond_wait()`, and although that association must be respected by other threads, it is dissolved whenever no more threads are waiting on the condition (admittedly it is hard to imagine a situation in which this fact is exploitable by reasonably transparent code, but that's what the standard tells us). The rule here is that a given condition variable can be used with at most one mutex at a time; a mutex, on the other hand, can have as many condition variables associated with it as you want.

One final technical point is that there is no specific requirement that a thread own a mutex before signaling one of its associated conditions (though since no prior synchronisation has occurred the results as regards scheduling are nondeterministic). This *might* be useful in the odd case that the condition awaited controls whether an operation can be performed on the protected data, but is not determined by those data, though in general the necessity of checking that the wakeup was non-spurious $\Rightarrow^{5.4}_{79}$ makes it hard to see how this would work.

5.4.1 Condition variable type: `pthread_cond_t`

The POSIX condition variable type is `pthread_cond_t`, defined in `pthread.h`$_{273}$. Condition variables are closely related to mutual exclusion locks in formal properties as well as use; most of the restrictions we noted on the type `pthread_mutex_t`$_{68}$ apply, *mutatis mutandis,* to condition variables.$\Rightarrow^{5.3.1}_{68}$ To summarize:

- The condition variable type is *utterly* opaque, and cannot be assigned to, not even with an uninitialized value to get an uninitialized value.

- Different kinds of condition variables can co-exist within the `pthread_cond_t` type, as specified by an **attributes object** when the condition variable is initialized. In particular, attributes must be used to request shared memory compatibility (if available); details are provided in a later chapter.$\Rightarrow^{8.4}_{147}$

- Condition variables must be explicitly initialized before use; this can be done either dynamically or with a static initializer macro, but the latter only works for variables that are actually statically allocated (*i.e.* are `static` or `extern`), and can only yield a condition variable with default attributes.

[10] Other designs avoid this problem by giving awakened waiters priority of access to the mutex, but this would interfere with pthreads' treatment of thread priority scheduling.$\Rightarrow^{9.5}_{172}$

- In order to change the attributes of a condition variable you must first ‘destroy’ it, making it uninitialized again, and then reinitialize it.
- It is undefined what happens if you attempt to wait on or destroy an uninitialized variable, initialize an initialized variable, or destroy a condition variable on which one or more threads are waiting.

5.4.2 Initializing a condition variable

Condition variables represent abstract events in the interactions between threads; typically ones that may be awaited for relatively long periods of time. Variables of type `pthread_cond_t`$_{80}$ require explicit initialization. If they are declared `static` it suffices to initialize them with the magic constant `PTHREAD_COND_INITIALIZER`, but the more general technique is to pass their address, along with an **attributes object**$\Rightarrow^{8.2}_{138}$, to `pthread_cond_init()` before their first use. If the static initialization technique is used, or if the attributes object passed to `pthread_cond_init()` is `NULL`, default attributes are assumed. The function `pthread_cond_init()` returns `0` if it succeeds, and an error code otherwise; errors with statically initialized condition variables are not detected.

`pthread_cond_init()`, `PTHREAD_COND_INITIALIZER` **11.4.2**

```
#include <pthread.h>

pthread_cond_t static_cond = PTHREAD_COND_INITIALIZER;

int /*errno*/ pthread_cond_init
  ( pthread_cond_t *new_cond
  , const pthread_condattr_t *new_cond_attributes
  );
```

`pthread_cond_t`$_{80}$ *`static_cond`*
: A *static* condition variable to be initialized with the default attributes.

`pthread_cond_t`$_{80}$ *`*new_cond`*
: Pointer to the condition variable to be initialized.

`const pthread_condattr_t`$_{138}$ *`*new_cond_attributes`*
: Pointer to an attribute structure giving the properties wanted for the newly initialized condition variable. The defaults are used if you pass `NULL` for this argument.

The function `pthread_cond_init()` initializes a condition variable and makes it available for use; it is undefined what happens if you try to initialize a condition variable that is already initialized, or to use one that is not. If you need to change the attributes of a condition variable, you must first call `pthread_cond_destroy()`$_{82}$ on it, and then re-initialize it with `pthread_cond_init()`.

A statically allocated condition variable (one declared `static` or `extern`) can be initialized with the default attributes at compile time with `PTHREAD_COND_INITIALIZER`. The effect is the same as that produced by `pthread_cond_init()` except that no error checking is done. For dynamically allocated variables, however, you must call `pthread_cond_init()` explicitly.

The value returned by `pthread_cond_init()` is an integer containing an error number:$\Rightarrow^{A}_{271}$

On success `pthread_cond_init()` returns `0` after initializing *new_cond* according to the specifications in *new_cond_attributes*.

On failure it returns an error code:

- `EINVAL`$_{271}$ Something is wrong with *new_cond_attributes*.
- `ENOMEM`$_{271}$ The process lacks the memory to initialize another condition variable.
- `EAGAIN`$_{271}$ The process lacks some other resource needed to initialize the condition variable.

5.4.3 De-initializing a condition variable

Just occasionally it may be necessary to de-initialize a condition variable so that it can be re-initialized with new attributes. Far more likely is the possibility that the variable will need to be destroyed so that it can then be deallocated. Before it can be destroyed, a condition variable must first have been initialized, and must not have any threads waiting on it. It is undefined what happens if any use (other than re-initialization) is made of the variable after its destruction.

`pthread_cond_destroy()` **11.4.2**

```
#include <pthread.h>
int /*errno*/ pthread_cond_destroy(pthread_cond_t *cond);
```

`pthread_cond_t`$_{80}$ **cond*
Pointer to the condition variable to be destroyed (*i.e.* de-initialized).

Returns the condition variable to its uninitialized state, suitable for freeing or rebuilding with `pthread_cond_init()`. It's undefined what happens if it is already uninitialized or if other threads are *blocked*$_{294}$ on it.

The function `pthread_cond_destroy()` returns an integer containing an error number:$\Rightarrow^{A}_{271}$

On success it returns `0`.

On failure it returns one of the following:

- `EINVAL`$_{271}$ The *cond* argument was invalid.
- `EBUSY`$_{271}$ The condition variable is being used by another thread.

5.4.4 Awaiting a condition

Condition variables exist to be waited on, and waiting is performed with `pthread_cond_wait()` and its variant `pthread_cond_timedwait()`, used if a thread does not wish to wait for the condition indefinitely. Condition variables are always awaited from within a mutex—that is, they must be called by the owner of a mutex; they cause the calling thread to release its lock on the mutex atomically with going to sleep; and at that moment the condition variable becomes associated with the mutex for as long as any thread is waiting for it.

Threads awoken from their sleep, for any reason other than an error, will find that, when they return, the mutex has been restored to them (they may therefore be implicitly blocked for some time waiting for it to become available). Although the reason for (deliberate) awakening may be very specific, awakening threads should check the state of the computation explicitly to see if the intended circumstances indeed hold, since 'spurious' (*i.e.* unasked) awakening is a documented possibility.

A thread, of course, typically blocks within a call to one of the `cond_wait` functions for some time. The value eventually returned will be one of `0`, `ETIMEDOUT`, or an error code.

`pthread_cond_wait()`, `pthread_cond_timedwait()` **11.4.4†◇**

```
#include <pthread.h>

int /*errno*/ pthread_cond_wait
  (pthread_cond_t *cond, pthread_mutex_t *mutex);

int /*errno*/ pthread_cond_timedwait
  ( pthread_cond_t *cond
  , pthread_mutex_t *mutex
  , const struct timespec *absolute_time
  )
```

`pthread_cond_t`$_{80}$ `*cond`
: Pointer to the condition variable representing the condition awaited.

`pthread_mutex_t`$_{68}$ `*mutex`
: Pointer to the mutex within which you are waiting for the condition.

`const struct timespec *absolute_time`
: Pointer to the specification of an absolute time at (or after) which the calling thread should abandon its wait. (Remember that some additional delay may be occasioned by the need to re-acquire the *`mutex`* before returning.)

The purpose of these two functions is to wait until some condition is satisfied, which is suggested (though not guaranteed) by its condition variable being signaled.

In either case the *`mutex`* must be locked by the calling thread at the moment the function is invoked; that thread will own the mutex once again when the function returns, if the return is successful (if, that is, the return value is either `0` or `ETIMEDOUT`$_{271}$—for this purpose we don't consider a timeout to be an error).

After one of these functions has been called, and before it returns, the calling thread will block and, simultaneously, ***`mutex`*** will be released for possible use by other threads. Threads blocked on a condition variable in this way can be unblocked by another thread if it invokes `pthread_cond_signal()`$_{86}$ or `pthread_cond_broadcast()`$_{86}$ on the condition variable ***`cond`***; *but there is no guarantee that they will not be unblocked for other, unspecified reasons.* Once a thread has been released from its wait on the condition variable it may still be blocked waiting for its turn with the mutex (which it must obtain before the wait function can return). The order in which threads get the mutex is decided in the usual way (whether a thread is calling a lock function or trying to return from a condition wait is not important).$\Rightarrow^{9.2}_{158}$ After returning from a wait, a thread should check that the condition it awaited indeed holds, and typically re-issue the wait call identically if it does not.

In the case of `pthread_cond_timedwait()`, a third parameter, `absolute_time`, specifies a time beyond which the calling thread does not wish to wait; if that moment passes without ***`cond`*** being signaled, or if it has already passed by the time of call, the thread awakes, (eventually) re-acquires the mutex, and returns from the function with a benign timeout error. See the discussion below for the implications of this wording.

It is undefined what happens if you try to wait on a condition variable that is uninitialized.$\Rightarrow^{5.4.2}_{81}$ It is undefined what happens if two threads attempt to use the same condition variable in association with two different mutexes during overlapping intervals.

The value returned by either function is an integer holding an error number:$\Rightarrow^{A}_{271}$

On success `pthread_cond_wait()` returns `0` and the current thread owns `mutex`; it is probable *(but not certain)* that `cond` has been signaled.

On failure it returns an error code:

- `ETIMEDOUT`$_{271}$ *(`pthread_cond_timedwait()` only)* Signifies that `absolute_time` has been reached or exceeded before any thread signaled ***`cond`***. This may be a perfectly normal occurrence. The current thread owns the `mutex`.
- `EINVAL`$_{271}$ It was determined that one of the arguments is invalid, or that ***`mutex`*** is not locked by the calling thread.

The functions `pthread_cond_wait()` and `pthread_cond_timedwait()` are **synchronizing functions**$\Rightarrow^{5.2}_{64}$ on two counts: (1) the next thread to acquire `mutex` will see a view of memory consistent with that of the calling thread at the moment of call; and (2) the (successful or timed out) caller will return to a view of memory consistent with that of the last thread to own ***`mutex`***, normally a thread signaling ***`cond`***.

One obvious question about the design of `pthread_cond_timedwait()` is why the timeout argument should be an absolute time-point, rather than a (sometimes more convenient) delay. After all, this introduces the peculiar special case of 'instant time-out' into the behavior of the function. In fact, this choice is the result of careful design.

The most obvious advantage of using a fixed endpoint is that if (as is recommended) condition variables are waited for in a loop, to take spurious wakeups and overloaded (and even erroneous) use of condition variables into account, the need to recompute an ever-diminishing remaining delay is removed. Significantly, this means that the real time can be checked once, *outside* the loop, rather than repeatedly, within it.

Perhaps more important is the fact that the presence of other threads and, indeed, other processes, on a physical CPU means that a thread can be delayed at any moment. In particular, this can happen in the critical span between the moment it checks the real time clock and the moment it puts that information to use. If the target time for a wait is an absolute one, such an error is effectively subtracted from the time that the thread waits, and it will wake up earlier (at around the time that would have been expected if the delay hadn't occurred). If the delay is expressed relatively, however, the sign is reversed, and such a delay is *added* to the time it takes the thread to reach the code on the other side of its wait. This is more likely to be a damaging error mode even than not waiting at all.[11]

Wakeup failure Now for the bad news: the wording of the standard[12] implies—no, states—that if a condition is signaled while your thread is `timedwait()`ing for it but some *other* thread is the one that is awoken and gets the mutex, your thread will *not* be awoken when the timeout expires. The implication is that, unless perhaps the implementation you are using takes it upon itself to deliver a *spurious* 'spurious wakeup'[13] under these circumstances, `pthread_cond_timedwait()` can generally only be used effectively by *single* waiters or with *broadcast* awakening. (This is sufficiently bizarre that we anticipate that many implementations will not go to the trouble of getting it 'right' and will awaken waiting threads with an `ETIMEDOUT` whenever their specified deadline passes; but of course, this is not to be relied upon.)

5.4.5 Signaling conditions

The function `pthread_cond_wait()`$_{83}$ is used to allow a thread holding a mutex to 'step aside' and sleep until some condition is fulfilled, while granting other threads access to the mutex in the meantime. Watching for an arbitrary condition, of course, is not part of the model of a conventional programming language like C; so detecting and reporting the occurrence of the condition falls within the responsibilities of the programmer. This is fine, since usually conditions are made true by the explicit action of some thread, and that thread can signal the condition whenever it has—or might have—made it come true.

This signaling is done by calling one of the functions `pthread_cond_signal()` or `pthread_cond_broadcast()` with the condition variable as argument—the latter if *all* waiting threads should be awoken. The signaled threads (if any) then start to contend for use of the mutex in the usual manner.

[11] Fans of Douglas Adams' *Hitch-Hiker's Guide to the Galaxy* may recall Eddie the shipboard computer explaining that "... by the time I finish saying what I'm saying now, it will be, hmm, seven seconds..." before the missiles impact the ship.

[12] "... an error [ETIMEDOUT] is returned if the absolute time specified by *abstime* passes (...) before the condition *cond* is signaled or broadcasted [*sic*],...". 11.4.4.2

[13] *I.e.*, since the standard allows you to be awoken 'by mistake,' the implementation could arrange to make just such a 'mistake' at exactly the expected moment; the return value would, however, properly have to be `0` and not `ETIMEDOUT`$_{271}$.

`pthread_cond_signal()`, `pthread_cond_broadcast()` **11.4.3◇**

```
#include <pthread.h>

int /*errno*/ pthread_cond_signal(pthread_cond_t *cond);

int /*errno*/ pthread_cond_broadcast(pthread_cond_t *cond);
```

`pthread_cond_t`$_{80}$ `*`*`cond`*
Pointer to the condition variable representing the condition to be signaled.

Both of these functions awaken threads waiting on the condition variable pointed to by *`cond`* so that they resume, contending for the mutex associated with *`cond`* at wait time. $\Rightarrow^{5.4.4}_{83}$ If there are any threads waiting on *`cond`*, `pthread_cond_signal()` will awaken *at least one* of them, as chosen by the scheduler$\Rightarrow^{??}_{??}$, while `pthread_cond_broadcast()` will awaken *all* of them. (The intention is that `pthread_cond_signal()` will attempt to awaken *exactly* one thread, but this condition is relaxed for the benefit of some hardware on which this guarantee would make the operation much more expensive.)

These functions return an integer containing an error number$\Rightarrow^{A}_{271}$:

On success they each return `0`.

On failure they return:

`EINVAL`$_{271}$ It was detected that the *`cond`* argument was invalid.

`pthread_cond_signal()` and `pthread_cond_broadcast()` are **synchronizing functions**$\Rightarrow^{5.2}_{64}$ on two counts: (1) the next thread to acquire the mutex associated with *`cond`*, typically as a result of being signaled, will see a view of memory consistent with that of the calling thread at the moment of call; and (2) the caller will return to a view of memory consistent with that of the last thread to own the mutex.

5.4.6 Example: condition variables

For our second example of the use of condition variables we take the construction of a useful higher-level synchronization tool: the multiple-reader, single-writer lock. As we have already discussed, POSIX rules allow at most one thread to have write access to a variable at a time, but any number of threads can share read-only access to the same area of memory without trouble—provided that proper synchronization is observed when changing 'ownership'.$\Rightarrow^{5.2}_{64}$ Clearly, it would be useful to have a kind of lock which enforces just this condition; one that allows as many threads as want to lock it in 'read' mode, but ensures that at most one thread can lock it in 'write' mode—and this only when no read locks are outstanding.

This is an ideal example for condition variables, since in the usual case we hope that a thread will be able to go ahead and grab the lock it needs, checking only that this is legal; that's a typical task for a mutex. Sometimes, however, a writer will need to wait for all the readers to leave the mutex, and this is where condition variables come into play.

For the sake of style, we will model our **mrsw locks** after mutexes (though we shall omit the business about attributes because we haven't covered that yet). We can't give them names beginning `pthread_`, though—those are reserved to the system for pthread functions—so we will use `spg_` names (after the authors) instead.

mrsw declarations

```
#include <pthread.h>

typedef struct
  { pthread_mutex_t mutex;
    pthread_cond_t  cond;
    unsigned readers, writers, grabbers;
  } spgmrsw_ty;

#define SPG_MRSW_INITIALIZER                                             \
  { PTHREAD_MUTEX_INITIALIZER                                            \
  , PTHREAD_COND_INITIALIZER                                             \
  , 0, 0, 0                                                              \
  }

enum spg_mrsw_mode
  { SPG_MRSW_RO                                 /* Request read access only */
  , SPG_MRSW_RW                                /* Request read/write access */
  , SPG_MRSW_GRAB           /* R/W access, inhibit new readers while waiting */
  };

extern int /*errno*/ spg_mrsw_init(spgmrsw_ty *mrsw);
extern int /*errno*/ spg_mrsw_destroy(spgmrsw_ty *mrsw);
extern int /*errno*/ spg_mrsw_lock
  (spgmrsw_ty *mrsw, enum spg_mrsw_mode mode);
extern int /*errno*/ spg_mrsw_unlock(spgmrsw_ty *mrsw);
```

The lock structure itself contains a mutex that protects the remaining contents, a condition variable that threads can use to wait for the mrsw lock to become available, and three count fields: one of threads that currently have the lock for read, one of writers, and one of 'grabbers.' The invariant we are trying to maintain is that at most one of `readers` and `writers` is nonzero, and `writers` is never more than `1`. What of grabbers? This field is intended to solve the problem that if there are enough readers coming and going, threads wishing to use the lock may 'starve' because the lock is never completely free of reading threads. Sometimes this isn't a problem, in which case we call `spg_mrsw_lock()` with a `mode` of `SPG_MRSW_RW`. But we also provide a mode `SPG_MRSW_GRAB` which actively prevents new readers from acquiring the lock while we are waiting for write access. The number of threads so waiting for the lock to 'drain' is stored in `grabbers`.

mrsw (continued) implementation

```
int /*errno*/ spg_mrsw_init(spgmrsw_ty *mrsw)
  { int x = pthread_mutex_init(&mrsw->mutex);

    if (x) return x;
    x = pthread_cond_init(&mrsw->cond);
    if (x) pthread_mutex_destroy(&mrsw->mutex);
    mrsw->readers = mrsw->writers = mrsw->grabbers = 0;
    return x;
  }

int /*errno*/ spg_mrsw_destroy(spgmrsw_ty *mrsw)
  { int x = pthread_cond_destroy(&mrsw->cond);

    return x ? x : pthread_mutex_destroy(&mrsw->mutex);
  }

int /*errno*/ spg_mrsw_lock(spgmrsw_ty *mrsw, int mode)
  { int x = pthread_mutex_lock(&mrsw->mutex);

    if (x) return x;
    if (mode == SPG_MRSW_GRAB) mrsw->grabbers++;
    while (  mrsw->writers                 /* if there's a writer, no entry */
          || ( mode == SPG_MRSW_RO
             ? mrsw->grabbers                  /* readers wait for grabbers */
             : mrsw->readers                    /* writers wait for readers */
             )
          )
      { x = pthread_cond_wait(&mrsw->cond, &mrsw->mutex);
        if (x) return x;
      }
    if (mode == SPG_MRSW_GRAB) mrsw->grabbers--;
    if (mode == SPG_MRSW_RO) mrsw->readers++;
    else mrsw->writers = 1;
    return pthread_mutex_unlock(&mrsw->mutex);
  }

int /*errno*/ spg_mrsw_unlock(spgmrsw_ty *mrsw)
  { int x = pthread_mutex_lock(&mrsw->mutex);

    if (x) return x;
    if (mrsw->writers) mrsw->writers = 0;
    else mrsw->readers--;
    if (!mrsw->readers) pthread_cond_broadcast(&mrsw->cond);
    return pthread_mutex_unlock(&pthread->mutex);
  }
```

This example differs from the one that introduced the section in that shares the condition variable between three different classes of waiters (readers, writers and grabbers), simply broadcasting it and causing a free-for-all whenever the conditions are ripe for the lock to change mode—whenever the lock has neither readers nor writers. The while loop in the locking function then becomes responsible for arbitrating whch thread gets access next. This will result in poorer performance under high contention than using a separate condition variable for each circumstance, but it simplifies the code by eliminating the need for the unlocker to be concerned about what kinds of requests are outstanding.

Note that using `pthread_cond_signal()` in place of `pthread_cond_broadcast()` wouldn't work with our single condition method, because it could awaken *any* waiting thread. If there is a grabber waiting for the lock, though, a reader would not be allowed to proceed even if it was chosen, and the system might deadlock.

You might wonder why (at least in the read/write case) we don't make direct use of the mutex field to provide exclusion, just leaving it locked. The answer is that we would then need either two different unlock operations, or an additional mutex in the mrsw structure in order to protect the field that tells us whether the mutex is in use at unlock time!

5.5 Ensuring a function is called only once

Imagine there was no such thing as `PTHREAD_MUTEX_INITIALIZER`$_{70}$. Getting an application's mutexes initialized would now be quite a challenge, since there would no longer be any simple way of ensuring mutual exclusion around the initialization code itself. For the main program, this is hardly insurmountable, since it's simple (if not always pretty) to call various initialization functions before the first call to `pthread_create()`$_{33}$. But when writing library routines, where one does not wish to burden the caller with explicit initialization, things are much more difficult. In earlier versions of the pthreads standard, before the various `_INITIALIZER` constants were introduced, this was a real concern.

The original solution was the introduction of another, more specialized facility: the **once** system. This consists of a new type of static, 'one-use' mutual exclusion variables, `pthread_once_t`. Its instances *must* be allocated statically and initialized with a constant named (anomalously) `PTHREAD_ONCE_INIT`. There is then a function, `pthread_once()` whose sole purpose is to protect a single call to a single user function from happening more than once in the lifetime of the computation. A lot of work for a simple effect, and the whole arrangement has been rendered obsolete by static mutex initialization, which is almost as simple to use and a lot more flexible. Nonetheless, the **once** system remains part of POSIX.

`pthread_once()` **16.1.7.5**

```
#include <pthread.t>

pthread_once_t static_once = PTHREAD_ONCE_INIT;

int /*errno*/ pthread_once
  (pthread_once_t *once, void (*function)(void));
```

`pthread_once_t` *`static_once`*
: A *statically allocated* control object being initialized.

`pthread_once_t *`*`static_once`*
: Pointer to a `pthread_once_t` object that records whether this 'once' operation has been (or is being) performed.

`void`$_{27}$ `(*`*`function`*`)(void)`
: Pointer to the function that actually performs the operation to be done only once.

The first thread to invoke `pthread_once()` with a given `pthread_once_t` object *`once`* evaluates *`function`*`()`. Other threads calling `pthread_once()` on the same once object are blocked until this evaluation is complete. Subsequent calls with the same *`once`* object do nothing and return immediately.

It is undefined what happens if *`static_once`* is not statically allocated (*i.e.* does not have `static` or `extern` storage class), or if it was not initialized as shown, with the constant `PTHREAD_ONCE_INIT`.

The function `pthread_once()` theoretically returns an error number, but in fact there are no errors specified for it. It should normally return `0`, indicating success.

Since this facility is likely to be used (if at all) to protect initialization routines for library functions, it is important to note that it has potential for **deadlock**: if the protected initialization code itself calls a function that does a similar conditional initialization under the same once lock, the second call will block indefinitely waiting for the first to return, even if the initialization function itself is written (as it would normally be) to ignore recursive calls to itself.

In fact, as we've noted, `pthread_once()` is trivial to code for yourself, given statically initialized mutexes as a building block:[14]

spg_once (basic version) definition

```
#include <pthread.t>

typedef struct {pthread_mutex_t mutex; int state;} spg_once_ty;
#define SPG_ONCE_INIT {PTHREAD_MUTEX_INITIALIZER, 0}

void *spg_once(spg_once_ty *once, void (*function)(void))
  { pthread_mutex_lock(&once->mutex);
    if (!once->state)
      { function();
        once->state = 1;
      }
    pthread_mutex_unlock(&once->mutex);
    return 0;
  }
```

[14]Actually the version given here is slightly buggy. For reasons that will become clear in chapter 11, we should really be using the cleanup facility we will introduce in section 6.3 to handle the unlocking of the mutex. But for simplicity's sake we will ignore the matter for now.

The advantage of writing your own initialization routine, of course, is that you can extend the basic paradigm in a number of useful ways. For example, `pthread_once()` does not give you any direct method of passing arguments to or recovering status information from the client function, and provides no method of associating the control object with the function, which is unsafe and clutters up the notation. The following function is rather nicer (and just as simple!):

spg_once (improved version) definition

```
#include <pthread.h>

typedef struct
  { pthread_mutex_t mutex;                          /* Serialization control */
    void *(*function)(void *arg);               /* Function to call, or NULL */
    void *value;                      /* Argument for function(), or result */
  } spg_once_ty;

#define SPG_ONCE_INIT(f,a) {PTHREAD_MUTEX_INITIALIZER, f, a}

void *spg_once(spg_once_ty once)
  { pthread_mutex_lock(&once->mutex);
    if (once->function)
      { once->value = once->function(once->value);
        once->function = NULL;
      }
    pthread_mutex_unlock(&once->mutex);
    return once->value;
  }
```

Many other enhancements are possible, of course, and might prove useful in one application or another. For example, you might want to allow retrying the initialization function in case of failure; or you might take precautions (as we'll see later) against the calling thread exiting while inside the user's function.

5.6 Things to look out for

Other systems have been designed with parallelism and synchronization built in from the ground up. POSIX synchronization facilities are at a lower, more programmer-visible level than those provided by most such systems, appearing as library routines rather than primitive language constructs. One of the consequences of this is that mutexes are not explicitly associated with they data they protect, and condition variables are are not explicitly connected with the mutexes they are used with. Of course, this results in a great deal of added flexibility, and in one sense encourages *higher*-level programming than the syntactically-driven approaches, since it becomes possible to associate mutexes and conditions with *abstractions* (of which the most obvious are various kinds of collections and linked data structures) and not just things that are actually encoded as single, contiguous items of data.

5.6.1 Race conditions

Unfortunately, the consequences of making an error in associating synchronization primitives with their charges are the worst: very often you get a program which seems sensible to the human reader, which compiles cleanly, which may run perfectly on the machine you develop it on, but which fails catastrophically in the field—or after a compiler upgrade. C programmers may become resigned to tracking down the occasional wild pointer bug, but these **race conditions** (so called because in older, simpler times what happened depended unpredictably on which thread got to the data first; on modern hardware it may depend unpredictably on other factors as well) are even worse because they are nondeterministic and subject to such intangibles as system load.

There are techniques for debugging multithreaded programs (chapter 18 is devoted to the subject), but the most important technique is to code defensively from the start. Fortunately, pthreads mutexes are designed to be very efficient—taking only a couple of instructions on a single processor machine, if the implementor was sufficiently clever and gets a little co-operation from the compiler—and so the temptation to skimp on synchronization is minimized.

Not all races, by the way, are symmetric; two other common problems are *timing races* and *initialization races*. We've already mentioned a typical case of timing race: had `pthread_cond_timedwait()` taken a *relative* timeout value instead of an absolute one, the calling thread would in effect be racing against the clock between the computation of the timeout and the initiation of the wait.

Initialization races on the other hand are races between resources' initialization and their first use. For instance, it is sometimes tempting (but incorrect) to code something like: `pthread_create(&p->thread, NULL, f, p)`. The problem here is that there is no guarantee that the new thread ID has been stored in `*p` before the new thread starts running. (This problem is basically dual to to the deallocation races that can occur when passing pointers to stack allocated storage between threads.)

In any event, here's our advice on how to avoid races in general:

Ritualistic programming Since the only thing that makes a multithreaded program run reliably is strict adherence to the discipline of locking data structures before you use them and unlocking them when you are done, it makes more than a little sense to develop strict conventions about how this should be done. While your fingers will occasionally betray you by not typing what you intended, they can also be trained to remember things that your brain loses track of from time to time, and locking the data structure before use is at about the right level to form a habit—if your naming conventions and calling conventions are rigid enough.

Internal documentation is also very helpful, of course; in the case of a mutex, this should include a description of exactly what it protects, and in the case of a condition variable, just what it represents—and when to signal it.

But these are local phenomena, and the biggest synchronization problems are the global ones.

Defensive publishing There's an old adage of C programming—or at least, there should be—that goes "Don't Put Anything Interesting In A Header File."[15] Unfortunately, in this realest of all possible worlds, this is not always an option, but "Try To Forget What You Saw In The Header" comes in a close second.

The point is that the only reliable way to prevent people (including yourself) from forgetting to perform the required synchronization steps is to make it impossible. There's a style of programming, popular in some circles, that never, *never* uses global variables. When programming with threads this becomes more than an æsthetic principle; it is a basic survival skill.

A global variable, of course, is something to which access must be controlled; but there is no guarantee that everyone will observe the conventions correctly. There are only two solutions: either learn to live without globals completely (this generally involves adding "handle" arguments to a lot of your functions, which—say the anti-global-variable-ists—makes them clearer, more re-usable, and more self-documenting anyway), or convert globals into file-level `static` variables to which you export only **access functions**. The theory is that those access functions can all be kept in the same file, where examples of the locking discipline are right in front of you all the time you are working on them; when you're working in another file and they're out of sight, you can (to a first approximation—see the next section) also put them out of your mind.

Even among purists, publishing the contents of a `struct` in a header file is sometimes considered a minor sin by comparison with committing a global variable—to this day, macros are usually faster than calls to separately compiled files—but when working with threads it becomes clear that it is actually just as unsafe. All the synchronization problems that apply to global variables apply as strongly to any datum at all. In fact, since the locking protocols for complex data structures are often more involved than those for simple variables, the problems may be worse. Once again, the only real defense you have against someone not following your locking conventions is to give them no means to access the data but through access functions that implement the protocol themselves. At the very least, and if you *must* publish the details of these structures, continue to provide such accessors as macros.[16]

5.6.2 Deadlocks

As with many things in this life, synchronization is something of which one can get too little—or too much. Too little synchronization leads to races, while too much can result in **deadlock**.

[15] As opposed to anything *useful,* of course; header files should declare *everything* useful—they should just conceal as much of the 'how' as possible.

[16] One simple information-hiding technique that deserves to be employed more heavily is the systematic use of *two* header files per module, one of which declares all and only the public interface to the module, and one of which declares the common private structures required by the individual source files. The latter includes the former at the top—not vice-versa (and if this causes errors, your declarations are inconsistent or the public interface is incomplete!).

A **deadlock** is simply what happens when one or more threads are all waiting for each other; in this situation, unless something times out or is otherwise forcibly unblocked, nothing more will ever get done. The most common way of achieving deadlock with the facilities presented in this chapter is through **cyclic locking**. In its simplest form this involves no more than a single thread trying to obtain a lock it already holds (and note that there exist useful synchronization primitives for which this is perfectly legal and safe—with POSIX mutexes it is *not*), but in general any number of threads might be involved. For example, the following program deadlocks whenever control reaches the two marked points at about the same time:

deadlocking DON'T DO THIS!

```
#include <pthread.h>

pthread_mutex_t a = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t b = PTHREAD_MUTEX_INITIALIZER;

void *f1(void *)
  { pthread_mutex_lock(&a);
⇒   pthread_mutex_lock(&b);  /* Try to lock b while holding a */
    pthread_mutex_unlock(&b);
    pthread_mutex_unlock(&a);
  }

void *f2(void *)
  { pthread_mutex_lock(&b);
⇒   pthread_mutex_lock(&a);  /* Try to lock a while holding b */
    pthread_mutex_unlock(&a);
    pthread_mutex_unlock(&b);
  }

void main(void)
  { pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, f1, NULL);
    pthread_create(&thread2, NULL, f1, NULL);
    pthread_join(&thread1);
    pthread_join(&thread2);
  }
```

Fortunately, there exists a simple solution that is also usually practical, which is always to acquire mutexes in a consistent order. Observe that two copies of `f1()` could *not* deadlock when run in parallel; this is because a thread will be prevented from acquiring mutex `b` unless it *already* holds mutex `a`, and only one thread at a time can do that. Thus, in our example, had both threads been running functions with locking order patterned after `f1()`—and this would almost always be consistent with whatever processing involving mutexes `a` and `b` `f2()` might perform—then the pathological situation simply could not arise—one function would always wait for the other to complete before starting execution. There is, by the way, no reason to acquire any mutex that you *don't* want to use, and no restriction on the order in which mutexes are released; *not* holding a lock—or rather, not *trying* to hold a lock—always makes the deadlock situation better (of course, failing to acquire a mutex that protects data that you use is no fix—that sacrifices consistency, which was the purpose of using mutexes in the first place).

There is one thing to be careful about, though: the order you choose must encompass *all* of the mutexes in the program, including ones that might be manipulated internally, for example, in library functions. For example, the thread-safe version of stdio uses a mutex internally, as we shall see in a later chapter.⇒[12.3][223]

Why is deadlock not detected automatically and reported as an error? There are good techniques for doing this, but they are far too slow for building into the lowest level synchronization primitives provided to the user. The mutex is more important to the performance of a pthreads implementation than any other feature; the implementor of a highly tuned version will be counting instructions on the common paths through the code—with luck, on the fingers of one hand. And there are so many other, indirect, ways of deadlocking your program that catching this instance would not make the need for care vanish. Still, deadlock detection is useful both for debugging and for code that must be extremely robust, and for this reason the standard recognizes that an implementation *might* choose to detect deadlock and has allocated an error return, `EDEADLK`[271], for reporting it. Your implementation *may* make use of this provision, either by default, or, as likely, as an option that can be requested through the attribute structure at mutex initialization time.

5.7 Example programs

5.7.1 `join_any()`

In the last chapter we promised a solution to the problem of allowing a thread to join with *any* other terminating thread, rather than having to name a specific target thread. The trick here is to have threads *tell* us when they are about to die by publishing their identities in a well-known place. So long as they actually proceed to exit, this means that it is a good moment for another thread to join on them: the joining thread will only have to wait for as long as it takes the exiting thread to finish up.

Of course, the variable in which dying threads store these notices needs to be protected by a mutex. We also provide two condition variables: one of them signals the arrival of a death notice; the other is raised whenever the notice-board becomes empty. Doing it this way allows us to avoid explicitly managing queue structures: instead of storing a list of all the threads that are trying to exit at a given time, we provide a single place where notices are transferred and let the condition variable handle the queuing of threads waiting to use it.

join_any — join_any.c—static data

```
static int exiting = 0;
static pthread_t thread;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
static pthread_cond_t empty = PTHREAD_COND_INITIALIZER;
static pthread_cond_t full = PTHREAD_COND_INITIALIZER;
```

The code for `join_any()` itself works like this:

join_any (continued) join_any.c—join_any()

```
void *join_any(void)
  { void *r;
    pthread_mutex_lock(&mutex);
    while (!exiting) pthread_cond_wait(&full, &mutex);
    r = pthread_join(thread);
    exiting = 0;
    pthread_signal(&empty);
    pthread_mutex_unlock(&mutex);
    return r;
  }
```

Finally, here's the code that we must ensure is called as the last thing done by any thread that wishes to be eligible for pickup by `join_any()`:

join_any (continued) join_any.c—join_me()

```
void join_me(void)
  { pthread_mutex_lock(&mutex);
    while (exiting) pthread_cond_wait(&empty, &mutex);
    thread = pthread_self();
    exiting = 1;
    pthread_cond_signal(&full);
    pthread_mutex_unlock(&mutex);
  }
```

Note the slight ugliness in needing a separate variable, `exiting`, to record whether `thread` contains a meaningful value; normally, we would expect to be able to store `NULL` in `thread` and let that serve as a sentinel. The type `pthread_t`$_{32}$ is, however, opaque, and no constant `PTHREAD_NULL` is provided that will do the required trick. Thus the un-C-like redundancy.

5.7.2 accept_buf

As a second example of how mutexes and condition variables fit together in real life, we consider the problem of providing a network service that bounds the number of connections it is willing to serve at one time, in order to conserve memory and processing resources.

We start by building a reusable abstraction to represent reusable resources in general.

resource resource.h

```
typedef struct resource *resource_ty;

extern resource resource_init(void);
extern void resource_destroy(resource_ty);
extern void *resource_alloc(resource_ty);
extern void *resource_alloc_nowait(resource_ty);
extern int resource_free(resource_ty, void *);
```

A `resource` is a collection of interchangeable but not necessarily identical items, represented by generic pointers, that can be allocated and freed individually. As with mutex locking, allocation normally blocks until the requested resource becomes available; but an alternative non-blocking allocation function is also provided.

When a new resource object is initialized it is not directly given any initial items; instead new items are just constructed 'freed' into the pool. Similarly, you can permanently delete an item just be allocating it and omitting to free it again.

resource (continued) resource.c

```
#include <pthread.h>
#include <stdlib.h>
#include "resource.h"

#define increment 16

struct resource
  { pthread_mutex_t mutex;
    pthread_cond_t free_c;
    unsigned count;
    void **free;
    void **p;
  };

resource resource_init(void)
  { resource r = malloc(sizeof(*r));

    if (r)
      { pthread_mutex_init(&r->mutex);
        pthread_cond_init(&r->free_c);
        r->count = 0;
        r->free = r->p = NULL;
      }
    return r;
  }

void resource_destroy(resource r)
  { if (r->free) free(r->free);
    pthread_cond_destroy(&r->free_c);
    pthread_mutex_destroy(&r->mutex);
    free(r);
  }
```

```
void *resource_alloc_nowait(resource r)
  { void *x;

    pthread_mutex_lock(&r->mutex);
    x = r->p == r->free ? NULL : *--r->p;
    pthread_mutex_unlock(&r->mutex);
    return x;
  }

void *resource_alloc(resource r)
  { void *x;

    pthread_mutex_lock(&r->mutex);
    while (r->p == r->free)
      pthread_cond_wait(&r->free_c, &r->mutex);
    x = *--r->p;
    pthread_mutex_unlock(&r->mutex);
    return x;
  }

int resource_free(resource r, void *x)
  { pthread_mutex_lock(&r->mutex);
    if (r->p - r->free >= r->count)
      { void **f = realloc
          ( r->free
          , (r->p - r->free + increment) * sizeof(void **)
          );

        if (!f) return -1;
        r->free = f;
        r->count += increment;
      }
    *r->p++ = x;
    pthread_cond_signal(&r->free_c);
    pthread_mutex_unlock(&r->mutex);
    return 0;
  }
```

As it happens, this implementation of the abstraction works with a mutex-protected stack of item pointers. Waiting for a resource to become available is done with a condition variable that is signalled by `resource_free()`; this function is also responsible for extending the allocated stack storage as necessary. That's done inside the mutex so no synchronizing trickery of any sort is required.

With this tool in hand, we now proceed to define the `accept_buf`, a data structure that records an accepted connection.

accept_buf — accept_buf.h

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
typedef struct accept_buf
  { int             fd;
    struct sockaddr addr;
  } *accept_buf;

extern int accept_buf_init(unsigned max_connections);
extern accept_buf accept_buf_alloc(void);
extern void accept_buf_free(accept_buf buf);
```

These buffers are in fact a trivial application of the resource abstraction: the only vaguely interesting part is allocation the initial buffer pool, which we do just by parcelling out a suitably `malloc()`ed array.

accept_buf (continued) accept_buf.c

```
#include <stdio.h>
#include "resource.h"
#include "accept_buf.h"

unsigned nbufs;
struct accept_buf *bufs;
resource res;

int accept_buf_init(unsigned count)
  { if (count == 0)
      { fprintf(stderr, "Zero count in accept_buf_init()\n");
        return -1;
      }
    bufs = malloc(count * sizeof(struct accept_buf));
    res = resource_init();
    if (!bufs || !res)
      { printf("Not enough memory to init %d buffers\n", count);
        return(-1);
      }
    while (count--)
      if (resource_free(res, &bufs[count]))
        { printf("Not enough memory to init %d buffers\n", count);
          return(-1);
        }
    return 0;
  }

accept_buf accept_buf_alloc(void) {return resource_alloc(res);}

void accept_buf_free(accept_buf buf) {resource_free(res, buf);}
```

Now we come to the meat of the matter: our server. But this, too, is now trivial: if connections are recorded in properly allocated buffers, the resource mechanism itself takes care of any necessary synchronization, and bounds the number of outstanding threads as we desired. All we need do is fire up threads as connections come in and detach them so they can process and complete in their own time.

accept_buf (continued) main.c

```
#include <pthread.h>
#include <stdio.h>                      /* For NULL */
#include <fcntl.h>                      /* For O_RDONLY */
#include "accept_buf.h"

static void *process_connection (void *connection_)
  { accept_buf connection = connection_;
    int count, fd, i;
    char fname[1024];

    /* Do some real work here */
    count = read(my_connection->fd, fname, sizeof(fname) - 1);
    if (count > 0)
      { fname[count] = '\0';
        write(1, fname, count);
        for (i = 0; fname[i] && !isspace(fname[i]); i++)
          ;
        fname[i] = '\0';
        fd = open(fname, O_RDONLY);
        if (fd >= 0)
          while ((count = read(fd, fname, sizeof(fname))) > 0)
            write(connection->fd, fname, count);
      }

    /* Close connection and free buffer */
    close(connection->fd);
    accept_buf_free(connection);
    pthread_exit(NULL);
  }

int accept_count_max = 10;
short port = 8002;

int main(int argc, char **argv)
  { struct sockaddr_in connect_addr;
    struct accept_buf *buf = NULL;
    int connect_port, len;
    pthread_t thread;

    accept_buf_init(accept_count_max);

    /* Setup server */
    connect_port = socket(AF_INET, SOCK_STREAM, 0);
    if (connect_port < 0)
      { perror("socket");
        exit(1);
      }
    connect_addr.sin_family = AF_INET;
    connect_addr.sin_port = htons(port);
    connect_addr.sin_addr.s_addr = INADDR_ANY;
```

```
  if ( bind
         ( connect_port
         , (struct sockaddr *)&connect_addr
         , sizeof(connect_addr)
         )
      )
     { perror("bind");
       exit(1);
     }
  listen(connect_port, 8);
  while
     ( buf || (buf = accept_buf_alloc())
     , buf->fd = accept(connect_port, &buf->addr, &len)
     , buf->fd >= 0
     )
     { if (pthread_create(&thread, NULL, process_connection, buf))
         { /* Send an error back to the connector */
           /* Close connection and recycle buffer */
           close(buf->fd);
         }
       else
         { buf = NULL;
           pthread_detach(thread);                     /* Fly! Be free!!!! */
         }
     }
  /* At least let all the current connections finish */
  pthread_exit(NULL);
}
```

5.8 Summary

Data that are being updated cannot be shared between threads, but as many threads as you like can read from the same unchanging variable. In order to live with this restriction, there must be a way to 'hand off' use of data from one thread to another. This happens automatically on thread creation$\Rightarrow^{4.3}_{33}$ and termination$\Rightarrow^{4.5}_{39}$, but during the normal lifetime of a thread special synchronization facilities are needed.$\Rightarrow^{5.2}_{64}$

The basic tool for serializing access to a data structure is the **mutex**, represented by values of type `pthread_mutex_t`$_{68}$. These can be locked and unlocked, with the system guaranteeing that at most one thread may hold the lock at any given time. It also ensures that handing off works properly when controlled by a mutex; that is, if all threads use mutexes properly, data that were left when one thread released a mutex will be available and consistent when another thread acquires the same mutex.

Some discipline is necessary when working with mutexes: it is important to use mutexes to protect *all* modifications of shared data and *all* read accesses that could possibly, in any sequence of events, coincide with a modification; and it is essential to avoid **deadlock**$\Rightarrow^{5.6.2}_{93}$, which is most easily accomplished by always acquiring mutexes in the same order. The necessary level of care is hard to sustain; near-fanatic code structuring habits—in particular, hiding *all* data behind accessor functions—are a great help.

Another important kind of synchronization involves waiting for an event. This is done by waiting on a **condition variable** of type `pthread_cond_t`$_{80}$, either indefinitely or with a timeout; the condition can be signaled once with `pthread_cond_signal()`$_{86}$ or sent to all waiters on the same condition with `pthread_cond_broadcast()`$_{86}$. A condition variable is (dynamically) associated with a mutex, which is presumed to guard the data that the condition depends upon. Before waiting for a condition you must hold the relevant mutex; while waiting it is implicitly unlocked so that other threads can work on making the condition come to pass; and when the condition is finally signaled (or your wait times out), the ownership of the mutex is returned to you so that you can check the details—and reality—of the condition, and either wait again or finish processing. *This checking step is vital* because POSIX also allows threads to be awoken unexpectedly.

Finally, POSIX contains an archæological relic in the form of the `pthread_once()`$_{89}$ facility. This is designed to allow function libraries to initialize themselves automatically without getting confused by multiple simultaneous invocations. It is already obsolete because using regular mutexes statically initialized with `PTHREAD_MUTEX_INITIALIZER`$_{70}$ seems to be just as simple, more flexible, and, probably, faster.

Appendices

5.a Global exclusion

One facility that POSIX does *not* provide—perhaps because it is efficient *only* on single-processor machines—is **global exclusion**, by which a thread is guaranteed that no other thread (at least within its contention scope$\Rightarrow^{9.3.2}_{163}$) is running at all. This is useful as a synchronization tool of last resort (as, for example, when trying to use library functions that were not designed with multithreaded use in mind) or, as we shall see in the next section, internally, as part of the implementation of user-level mutual exclusion.

Because it prevents all other threads from making progress—even those uninvolved with the resource being protected—global exclusion is a drastic measure. Above all else, it is vital that it be used only for tightly bounded periods. It is nonetheless both efficient easy to implement on a single-processor machine, since all that is necessary is to prevent context switches. At the user level this involves no more than disabling the scheduling mechanism; at the hardware level, disabling interrupts. In either case this can often be accomplished with a single write.[17] A number of systems other than POSIX pthreads do provide it as a user-level facility.

---

[17] On a multiprocessor, the situation is quite different. The fact that there are multiple physical processors potentially at play in the program means that just disabling thread-switching will not suffice; the other processors will have to be switched to other processes or perhaps even stalled completely. In general this is not feasible (and it is certainly not good use of resources), and different solutions are employed, as we shall see in our discussion of hardware issues.$\Rightarrow^{5.c}_{107}$

5.a.1 DCE

DCE threads, which, we recall, are based on an early draft of POSIX threads (and so follow the POSIX naming conventions even to the point of using the `_np` suffix on functions that were not part of the standard at the time DCE was frozen), provide functions `pthread_lock_global_np()` and `pthread_unlock_global_np()`.

[[What I don't know is whether this protects you against all other callers of `pthread_lock_global_np()` *or against all other threads,* simple.*]]*

5.a.2 OS/2

Under OS/2 functions are also provided to enter and leave critical sections at the global level. These functions are `DOSEnterCritSec()` and `DOSExitCritSec()`. [[**sps: Check this:**]] They are even more drastic than the DCE threads global exclusion operations, since they disable interrupt handling at the system level [[**sps: as far as I can figure out—then again, PGM says I've got this wrong. . . .**]].

5.b A look inside

In this section we once again take the cover off the CAPthreads implementation to see one way in which the synchronization facilities we have just presented have been realized. The usual caveats apply: there is more than one way to do terminal wrong to a feline, the author of CAPthreads reserves the right to change his mind about the best way to do things at any time, and CAPthreads is a *portable* implementation of pthreads that assumes the very simplest case: a completely conventional workstation with a single processor and minimal kernel support.

5.b.1 Implementing mutual exclusion

There are a large number of different ways of implementing a mutual exclusion lock.

Technically, on a single-processor computer, it suffices just to disable interrupts altogether so that nothing short of a hardware error (including time-slicing, which fundamentally relies on the delivery of timer interrupts to end each time slice⇒[1.5.1][13]) will cause the current thread to lose full use of the processor: if nothing else can run, it can't interfere with the current thread's access to data. This is, however, extremely drastic and unsuitable for user code, since it locks out *all* other processing, even processing unrelated to the resources requiring exclusion protection, including basic operating system functionality. (Nonetheless, as we've just seen, some operating systems do extend this kind of service to the user, at least within a contention scope.⇒[5.a][102]) With this implementation, all locks are the same lock—and this isn't quite correct in its dynamic semantics, since logically uninvolved threads should be free to make progress.

A 'real' mutex—one that doesn't interfere with other mutexes—is basically a boolean variable that (a) can be cleared without interference from other threads—that's the **unlock** operation—and (b) can be set if it is not already set, atomically returning the *old* value, providing the basic functionality of **trylock**:

```
typedef volatile int mutex_t;

int idea_for_trylock(mutex_t *mutex)
  { int oldValue;
```

```
    indivisibly
      { oldValue = *mutex;
        if (!oldValue) *mutex = 1;
      }
    return oldValue;
  }
void idea_for_unlock(mutex_t *mutex)
  { indivisibly *mutex = 0; }
```

The **lock** operation can be built out of **trylock** just by using it in a loop (followed, on a uniprocessor, by a call to `sched_yield()` to make sure that the thread now holding the lock gets a chance to release it$\Rightarrow^{9.4}_{172}$).[18] This still leaves us with a chicken-and-egg problem, though: the obvious implementation of the '*indivisibly*' construct involves a mutex.

Resolving this difficulty on real multiprocessors is sufficiently difficult that hardware engineers nowadays provide something like `idea_for_trylock()` and `idea_for_unlock()` *directly*, as specialized **synchronization instructions**; this is the topic of the next section.$\Rightarrow^{5.c}_{107}$

If there's only one processor, however, we already have the solution in hand: the global lock idea we proposed and dismissed at the start of this section will actually work to provide the atomicity we need, since each of the 'indivisible' sections above takes at most a handful of instructions.[19]

5.b.2 pthread_mutex_t

In CAPthreads the type `pthread_mutex_t`$_{68}$ is implemented as a structure with (to simplify slightly) the following fields:

type Used to distinguish the different kinds of mutex supported by CAPthreads$\Rightarrow^{8}_{137}$; in this section we describe only basic mutexes.

owner Points to the `pthread_t`$_{32}$ object representing the current owner of the mutex; or `NULL` if currently un-owned.

queue Queue of threads waiting for this mutex to become available. Linked through the pthread objects' **next** fields.

data Generic data field for the use of non-default mutex types.

In this structure it is the `owner` field that serves to record whether the mutex is currently locked: the mutex is free if and only if the **owner** field is `NULL`.

As you can see, this implementation is somewhat more complex than the basic method described above. Rather than building `pthread_mutex_lock()`$_{72}$ out of `pthread_mutex_trylock()`$_{72}$ and a loop, each mutex contains a queue of threads that are blocked upon it. Threads unlocking a mutex wake up any thread they find at the head of this queue as soon as they have reset the owner to `NULL`. Under heavy contention this approach reduces the number of context switches that are needed, and it can simplify the implementation of priority scheduling.

[18] Actually even this doesn't suffice in the presence of priority scheduling; but the implementation we present in a moment solves this problem *and* avoids the loop in **lock**.

[19] We're glossing over something here. Even if there is only a single processor but it supports multiple processes *with shared memory*, then thread packages implemented as user-level libraries will still suffer from synchronization problems: locking the local scheduler in one process does not, after all, exclude threads in other processes. In this case the solution to the problem of how to lock a shared structure is basically identical to the multi*processor* case. The question of how to wait for an unavailable lock is stickier; in terms of efficiency, the single-processor solutions are better, but this would require co-operation between the different per-process thread schedulers; it might be preferable complexity-wise just to treat the system as a virtual shared-memory multiprocessor.

5.b.3 `pthread_mutex_lock()`

If the function `pthread_mutex_lock()`$_{72}$ finds the mutex free, it grabs it; otherwise the calling thread must place itself on the queue of those waiting for the mutex. The way CAPthreads is coded the responsibility of changing ownership of the mutex to a newly awoken thread lies with the previous owner; thus when `pthread_mutex_lock()` wakes up from its wait it has nothing more to do but return success. The sequence of events is like this:

1. Disable scheduling, making this operation effectively indivisible.
2. Check the **owner** field. If the mutex is currently free:

   (a) Set the **owner** to `pthread_self()`$_{42}$.

   (b) Re-enable scheduling.

   Otherwise:

   (a) Add the calling thread, `pthread_self()`, onto the mutex's **queue**.

   (b) Set the current thread's state to *mutex wait*, relinquish the processor, and re-enable scheduling. We will not run again until we are at the front of the queue and the mutex has been released.

   (c) By this point we have *returned* from waiting for the mutex. By convention, this only happens if we have already been given ownership of the mutex.
3. By now we certainly own the mutex. Return zero for success.

The function `pthread_mutex_trylock()`$_{72}$ is much like `pthread_mutex_lock()`, only simpler: if the mutex is owned it can immediately return `EBUSY`$_{271}$, without any need to manipulate the queue.

One interesting thing to notice about this implementation is that `pthread_lock()` obtains mutual exclusion by briefly disabling the scheduler. This gives it access to the *globally* protected queue fields even when the mutex's *local* lock is owned by another thread. That is, ownership of the lock that protects user data and ownership of the lock protecting the metadata of the mutex are crucially distinct. (It is lucky that the user of mutexes doesn't need to worry about this!)

5.b.4 `pthread_mutex_unlock()`

Even with the additional complication of manipulating the queue of waiters, `pthread_mutex_unlock()`$_{75}$ is still straightforward:

1. Disable scheduling.
2. Check the mutex's **queue**. If it is non-empty:

   (a) Move the first thread on **queue** into the **owner** field (replacing the current value, which is always the current thread's ID).

   (b) Make the new owner runnable.

   (c) Resign the processor in favor of the new thread, re-enabling scheduling.

   Otherwise set the **owner** to `NULL` and re-enable scheduling.
3. Return success.

5.b.5 Condition variables

Interestingly enough, in CAPthreads the physical structure of **`pthread_cond_t`**$_{80}$ is almost identical to that of **`pthread_mutex_t`**$_{68}$, except in that it lacks an **owner** field. Though POSIX mandates that each condition variable in use be associated with a single mutex, CAPthreads does not represent this association explicitly:

type Again used to differentiate different classes of condition variable$\Rightarrow^{8}_{137}$.

queue Queue of threads waiting for this condition to be signaled, again linked through the threads' **next** field.

data Generic data field for the use of non-default condition variable types.

Waiting for a condition is accomplished simply by having a thread place itself on the condition's queue. **`pthread_cond_wait()`**$_{83}$ takes both a (locked) mutex and a condition variable as arguments, and proceeds as follows:

1. Disable scheduling.
2. Enqueue the current thread on the condition variable's **queue**.
3. Unlock the mutex [scheduling is still disabled].
4. Set the current thread's state to *condition wait*.
5. Relinquish the processor, re-enabling scheduling. We will not return from this call until the condition has been signaled.
6. Re-lock the mutex (for the signaler to do this we would have to record the relevant mutex in the condition variable itself).
7. Return.

The **`pthread_cond_signal()`**$_{86}$ operation is identical to **`pthread_mutex_unlock()`**$_{75}$, except for the absence of ownership manipulation:

1. Disable scheduling.
2. Check the condition variable's **queue**. If it is non-empty:
   (a) Pop the queue into a local variable.
   (b) Make the thread (if any) so obtained runnable.
   (c) Resign the processor in favor of the new thread, re-enabling scheduling.

   Otherwise just re-enable scheduling.
3. Return success.

5.b.6 Multiprocessor issues

[[Discuss Sparc v8 memory models?]]

A lot of research has gone into synchronization problems, especially for physical multiprocessors, both at the theoretical level and in terms of improving practical performance. Among the important practical issues are providing good basic synchronization primitives (discussed in the next section) and reducing the cost of contending for a mutex.

The queue-based method adopted in the implementation just described is a good choice if the queue structure itself distributes well on the hardware in question, and if the mutexes are relatively heavily contended or kept locked for fairly long periods. A second, and much simpler technique of implementing `pthread_lock()` is the **spin lock**, which works like this:

```
while (EBUSY == pthread_trylock(mutex))
  ;
```

and requires no special data structure for remembering waiters at all—that information is effectively kept in the processors waiting to get the lock.

This method can be used for high-speed, low-contention locks on multiprocessors, a case in which wasting the time of a thread (now on just one processor among many) has relatively minor impact on the performance of the system as a whole. The condition for it being a sensible choice is that the average time spent spinning in the `while`-loop is less than the time required for a processor to switch to a new thread (or that there are enough processors that switching is totally unnecessary).

At the other end of the spectrum, locks that see *very* high contention rates and large numbers of processors will see not only the primary mutex lock but also the queue of waiting threads as hot-spots. These benefit from having more parallelizable data structures than a simple queue in which to record the waiters.

One approach to addressing these different requirements is to provide more than one distinct class of mutex variable, selectable by the user through, say, the attribute mechanism. $\Rightarrow^{\S}_{137}$ Another promising approach is dynamic, '*reactive*' selection of strategy, whereby a mutex might convert from a test-and-set spin-lock to a queue to a tree as contention rises, and conversely; the interested reader is directed to [**?**] and references therein.

5.c Hardware synchronization instructions

In a computer with only a single processor, memory synchronization is in general unproblematic; since every thread runs on the same processor and (shared memory aside) in the same address space, they will all see the same cache state. This limits the possible consistency problems to the ones caused by the compiler moving non-private values into registers, and only very advanced compilers will do this across function calls (such compilers are probably sufficiently sophisticated to be taught synchronization routines as special cases in any event). Similarly, global locking is very efficient on these machines: its overhead is on the order of a machine instruction or two.

As soon as multiple processors with shared memory enter the picture, however, synchronization becomes a real concern and the cost of global locking is that all processors but one must be suspended! This is clearly unacceptable.

The usual solution to this problem is to provide one or more synchronization primitives *in hardware*. Different vendors have provided different primitives, varying (it turns out) in formal power. Some of them are quite interesting.

Here are a few typical examples [[**sps: These are from memory; check against good references**]]:

VAX Digital Equipment Corporation's venerable VAX CISC architecture provided direct support for atomic insertion and deletion into doubly-linked lists (in both absolute and relative flavors—the latter for use in variably-based shared memory). These instructions favor a lock design, efficient under contention, in which a lock is a queue containing both the owner and the waiters, and the thread at the queue head is the current owner.

Intel x86 The processors descended from the 8086 provide an external signal that is intended to be used to lock processor bus transactions and provide instruction indivisibility. This signal can be raised for the duration of one (arbitrary) instruction by prefixing the instruction (in the object code) with the `LOCK` modifier. In conjunction with an

`XCHG` swapping register contents and memory (or, starting with the 80386, `BTS`, setting a bit while loading its old value into the flags register), this provides for an indivisible test-and-set operation.

MC680x0 The Motorola 68020 and following provide three purpose-built synchronizing instructions: `tas`, an indivisible test-and-set; `cas`, a compare-and-swap that exchanges the contents of a register with those of a memory address providing the memory location is unequal to a third value; and `cas2`, which performs two compare-and-swap operations at once (and is thus a simultaneous generalization of test-and-set, atomic register/memory swap, and the 'interlocked' queue instructions of the VAX!).

HPPA The Hewlett-Packard Precision Architecture, by stark contrast with the 68020, provides a very understated test-byte-and-clear instruction.

Sparc *[[Look this up.* [[**cap: load memory and store byte**]]*.]]*

Alpha [[**sps: Check this description. Recent MIPS are maybe similar?**]] The VAX's replacement, the Alpha RISC, also provides a very aggressive set of synchronization primitives, in a very different, very low-level style:

load linked performs a normal read from memory, marking the cache line that was read from as being actively involved in a synchronization operation;

store conditional performs a write, but *only* if the target cache line is marked; and it clears any marks appearing on other processors' cached copies of the same line. It reports whether it succeeded.

Note that this arrangement is remarkable in that it can support protocols that deal fairly directly with high-level data structures (at least through pointers) and incur no overhead at the processor level in the absence of contention. In some sense it is thus both the simplest and the most powerful of the primitives on this list. It works by exposing a level of the architecture—the data cache—which is normally concealed from the programmer. Even the normal memory semantics of the Alpha expose caching, however, since the synchronization of memory contents is normally only guaranteed at explicit 'memory barriers'—synchronization points that, for reasons of efficiency, it is best to avoid too many of.

5.d Non-posix analogs

5.d.1 DCE

Synchronization in DCE threads is basically identical to that in POSIX—not surprisingly, since it is based on an earlier version of the same design. One thing that is noticeable is the absence of static initializers for mutexes and condition variables, since these were added only after DCE had branched off from POSIX; thus in DCE threads, unlike POSIX, `pthread_once()`$_{89}$ is nearly indispensable. A similar minor inconvenience is that instead of `NULL` being used to request the default attributes for an object, the predefined constants `pthread_attr_default`, `pthread_mutexattr_default` and `pthread_condattr_default` must be used.

5.d.2 MACH Cthreads

MACH's synchronization facilities are also historically related to those of pthreads, but more distantly than those of DCE are. MACH has types `mutex_t` (which is defined to be exactly `struct mutex *`, though `struct mutex` is itself private) and `condition_t` (similarly equivalent to `struct condition *`) that represent mutual exclusion locks and condition variables, respectively.

The usual method of creating and destroying mutexes and condition variables is with their own 'alloc' and 'free' routines:

```
mutex_t mutex_alloc(void);
void mutex_free(mutex_t);
condition_t condition_alloc(void);
void condition_free(condition_t);
```

Note that in keeping with Cthreads' minimalist style the allocation routines take no parameters and provide the programmer with no special control over the objects created.

There is also a lower level interface provided for initializing and finalizing synchronization objects whose storage is already allocated (using the appropriate `struct` types):

```
void mutex_init(struct mutex *);
void mutex_clear(struct mutex *);
void condition_init(struct condition *);
void condition_clear(struct condition *);
```

The only semantic difference between these two interfaces is that the higher-level functions call `malloc()` and `free()` to manipulate the storage holdoing the synchronizing structures, while the lower-level ones leave that up to the programmer.

As in POSIX there are three basic functions that manipulate mutual exclusion locks:

```
void mutex_lock(mutex_t);
int /*boolean*/ mutex_try_lock(mutex_t);
void mutex_unlock(mutex_t);
```

and condition variables can be awaited, signaled and broadcast:

```
void condition_wait(condition_t, mutex_t);
void condition_signal(condition_t);
void condition_broadcast(condition_t);
```

Just as in pthreads, wakeup may be spurious: `condition_wait()` returns at a moment when it is merely *likely* that the condition has been signaled.

One of the interesting consequences of the difference in semantics between the *non-preemptive* coroutine threads library (`-lco_threads`) and the MACH thread- and task- based implementations is that with the coroutine implementation and only that implementation, the use of mutexes may be unnecessary. The guarantees on when processing may switch between threads, and MACH's memory semantics, are sufficiently strong that it is fairly straightforward to write code that relies on known scheduling behavior to guarantee necessary exclusion (this is never true for portable pthreads programs).

5.d.3 GATech Cthreads

Under GATech Cthreads, `mutex_alloc()` has been revised to provide for failure reporting and so that the programmer can specify which of the available physical processing nodes the mutex object should be allocated on. The interface is now:

```
RESULT mutex_alloc(mutex_t mutex, int node);
```

where *node* is one of a processor number, `N_CURRENT` (the processor on which the call is being executed) or `N_ANYWHERE`. (This parameter is ignored on hosts for which it is meaningless.)

In reality, GATech Cthreads' `mutex_alloc()` is a macro, and captures the textual *mutex* argument as the mutex's *name*, for debugging purposes. Two further functions allow explicit manipulation of mutex names:

```
void mutex_set_name(mutex_t mutex, char *name);
char *mutex_name(mutex_t mutex);
```

Similar changes have appeared in the interface to condition variables: `condition_alloc()` has been made into a name-capturing macro with an extra *node* argument, and functions `condition_set_name()` and `condition_name()` have been added. There is one other substantive (and interesting) change in interface; the signature of `condition_signal()` has been changed to

```
RESULT condition_signal(condition_t condition);
```

The return value is either `T_SUCCEED` if a thread was signaled or `T_NO_THREAD_QUEUED` if no thread was actually awaiting the condition in question.

Bear in mind when setting up a synchronization variable that it may be necessary to call `cthread_publish()`58 to propagate anything stored in non-`memory_alloc()`ed storage. Thus the typical idiom for building a new mutual exclusion lock is:

```
static mutex_t m;

mutex_alloc(&m, N_ANYWHERE);
cthread_publish(&m);
```

5.d.4 OS/2

OS/2 provides three varieties of synchronization object (or 'semaphore'): **mutex semaphores**, akin to pthreads' mutexes; **event semaphores**, which correspond to condition variables; and **multiplex semaphores**, a 'data concentration' construct vaguely reminiscent of BSD Unix `select()` that allows a thread to wait for any or all of several occurrences.

OS/2 semaphore objects can be globally named at creation time (by strings, which for some reason must begin '\SEM32\' [*cf.* POSIX 1003.1b message queues]); this simplifies, but is not strictly required for, sharing between processes. They are created, according to kind, by the following calls:

```
DosCreateMutexSem
  ( PSZ pszName          /* Optional name */
  , PHMTX phmtx          /* Pointer to mutex handle */
  , ULONG flAttr         /* 0 = private; 1 = shared */
  , BOOL32 fState        /* Initial state; TRUE = owned */
  );
DosCreateEventSem
  ( PSZ pszName          /* Optional name */
  , PHEV phev            /* Pointer to event sem. handle */
  , ULONG flAttr         /* 0 = private; 1 = shared */
  , BOOL32 fState        /* Initial state; TRUE = posted */
  );
```

```
DosCreateMuxWaitSem
  ( PSZ pszName          /* Optional name */
  , PHMUX phmux          /* Pointer to mux. sem. handle */
  , ULONG cSemRec        /* Size of SemRec array */
  , PSEMRECORD pSemRec   /* Array of {hSemCur, ulUser} */
  , ULONG flAttr         /* Bit 0 set = shared
                            DCMW_WAIT_ANY set = wait for any
                            DCMW_WAIT_ALL set = wait for all */
  );
```

The `SEMRECORD` array passed to `DosCreateMuxWaitSem()` consists of pairs of sub-semaphores and user data fields which are returned to identify which sub-semaphore was the one to cause a multiplex wait to complete.

When a shared semaphore has been created, threads in other processes may **open** it with `DosOpen*Sem()`; it is usable and will not be destroyed until all the openers (including its original creator) have called the appropriate `DosClose*Sem()` routine to release it. Inconveniently enough, multiplex wait semaphores can only be opened by a process that has *already* opened each of the sub-semaphores indivdually.

The operation of a mutex semaphore is in general much like that of a POSIX mutex; it is locked by calling

```
DosRequestMutexSem
  ( HMTX hmtx
  , ULONG ulTimeout /* ms or SEM_INDEFINITE_WAIT */
  );
```

and unlocked with

```
DosReleaseMutexSem(HMTX);
```

OS/2 mutexes are, however, *counting* mutexes, which is to say that the owning thread may legitimately re-lock a mutex that it already owns; the mutex will not be released until it has been unlocked as many times as it was originally locked. This requires a little more effort in the implementation, but is convenient for protecting data structures that are manipulated at multiple levels of abstraction.$\Rightarrow^{8.c.1}_{152}, ^{12.a.1}_{236}$ OS/2 has also unified the `lock()` and `trylock()` notions into a single function with an explicit timeout that can be set to zero (`trylock()`), infinity (`lock()`), or millisecond values in between.

Unlike POSIX mutexes, it is also possible to ask a mutex semaphore about its current state:

```
DosQueryMutexSem(HTMX htmx, PPID ppid, PTID ptid, PULONG pCount);
```

takes a mutex semaphore and returns the identity of the currently locking process and thread (if any) and the number of times it is currently locked (which, recall, can be greater than one). One more useful piece of information that can be obtained from this call is the error status `ERROR_SEM_OWNER_DIED`, which indicates that the holder of the mutex terminated without first releasing the semaphore; presumably the resource the semaphore protected is now in an inconsistent state.

Event semaphores, as we have noted, correspond to pthread condition variables. They can be awaited with

```
DosWaitEventSem(HEV, ULONG);
```

and signaled by calling

```
DosPostEventSem(HEV);
```

Like OS/2 mutex semaphores they are also *counting,* which in the case of event semaphores means that they remember how *many* times they were posted. This count is atomically retrieved and reset to zero *independently* of thread wakeup, by the function

```
DosResetEventSem(HEV, PULONG);
```

Once again, there is a query function to retrieve the event semaphore's current state.

MuxWait—multiplex wait—semaphores are unlike anything found in pthreads. They allow you to combine several mutex or event semaphores (though a mixture is not allowed) into one, waiting for either any or all of them to become available. The list of semaphores awaited can be varied dynamically by calling the functions `DosAddMuxWaitSem()` and `DosDeleteMuxWaitSem()`. When you wait on a multiplex semaphore, calling

```
DosWaitMuxWaitSem
  ( HMUX hmux
  , ULONG ulTimeout
  , PULONG pulUser    /* From SEMRECORD */
  );
```

the location pointed to by the ***pulUser*** is set to contain the user identifier associated, at initialization or addition time, with the last sub-semaphore to contribute to the wait's returning.

Finally, because of a danger of deadlocks arising from window messages arriving from the OS/2 Presentation Manager while the receiving thread is blocked, the Presentation Manager provides replacement versions of the wait routines, `WinRequestMutexSem()`, `WinWaitEventSem()` and `WinWaitMuxWaitSem()`, that should be coded in place of the corresponding `Dos*Sem()` routines when the Presentation Manager is in use (would it be too cruel to call this an eensy disgustingXFde hack?).

Chapter 6

User Data

Chris's Corn for Chapter 6:

"Do THE BORG *ever have private thoughts?"*

6.1 Introduction

In this chapter we look at the special facilities that POSIX provides to manage threads' private data structures, and to help ensure data consistency *within* a thread.

The basic motivation for multithreaded programming is to allow many distinct threads of control to share access to a single data space; but one can have too much of a good thing. Using the tools we've presented so far, there is no way for a thread to have pervasive private data: a variable must either be global to the entire *process,* or local to a block; to reach called functions a thread-specific datum must be passed through the entire call chain. Particularly in the case of library functions that must maintain hidden state, or code that is being rebored for threads from a global-variable-heavy style, this can pose huge problems.

Threads face other data-related challenges; the most significant may be consistency maintenance. Most programs are designed with attention to the usual course of processing, with error handling something of an afterthought. This may not be unreasonable, since if an error is not of the sort that prompts immediate retry or a quick switch to some plan B, the response is usually to back all the way out; if a few things are left in inconsistent states along the way, they will be tidied up on exit or rebuilt on a later iteration, and no lasting harm is done. When other threads enter the picture, however, such transient inconsistencies are a recipe for disaster: unreleased mutexes, unsignaled conditions and illegal values in disregarded data structures will trigger serious errors elsewhere, even when the perpetrating thread gets off scot-free.

One unglamorous but especially important subclass of consistency maintenance problems is final cleanup and deallocation of data structures. In the multi*process* paradigm, of course, this is hardly an issue; aside from the occasional file left in `/tmp`, the protection barriers around a process allow the operating system to provide cleanup both effectively and implicitly. Individual threads are quite another matter. Since the many threads of a process share a single protection domain, every undeallocated resource counts against them all. In a long-running system even the smallest resource leaks will eventually accumulate to crash the process.

In the following sections we examine the features pthreads provides to address these issues: thread-specific data, and user cleanup handlers.

6.2 Thread-specific data; `pthread_key_t`

Thread-specific data provide something very much like variables that are *global* to the various functions of a program but *local* to a thread—and thus able to assume a different value in each thread. In order to get this effect without having to extend the C language (with a new storage class or whatever), pthreads introduces an opaque type of **keys**, `pthread_key_t`, which "name" some local storage in each thread (or, if you prefer, identify a *family* of local variables, one per thread). Like the rest of the symbols of the pthreads system, this type is defined in the standard header file **`pthread.h`**$_{273}$.

Such keys must be created with `pthread_key_create()`$_{115}$ before they can be used, and may be destroyed with `pthread_key_delete()`$_{117}$; as usual, double initialization and use of uninitialized values is forbidden. It would be very convenient if keys could be statically initialized like some of the other pthreads structures, but this would be very difficult to implement efficiently,[1] since the C preprocessor provides no effective way of generating unique values.

From the argument types of the thread-specific data manipulation functions it appears that keys are actually values and might be assigned and passed around rather like pointers, but apparently this is not the intent.[2]

One way in which thread-specific data improve over C's native variables is in the provision made for *destructors*, functions that are run on thread exit to release any thread-specific resources that might remain under still-allocated keys (the details of this process are discussed in a later section$\Rightarrow^{6.2.5}_{120}$). This is especially important in light of the fact that a thread might abruptly exit while deeply nested in function calls. $\Rightarrow^{4.4}_{37}, ^{11}_{203}$ The price paid is that, despite the declared `void *` type of thread-specific data, the values stored under a single key must all actually be of the same type—the same type anticipated by the destructor registered for the key at the time it was created. And beware: the destructor is *not* run when a new value is stored over an old one, or when the key itself is destroyed, though both of these events will cause the old binding to be lost.

A slight inconvenience of POSIX thread-specific data is that they are truly private to a thread; there is no way to read or write another thread's stored values. In particular,

---

[1] Hashing on the address of the key would work, since keys can apparently not be assigned, but would not be very fast. Storing the data *in* the keys requires locking and might prove even worse. See the discussion of implementation in the appendix to this chapter.$\Rightarrow^{6.a.1}_{127}$

[2] We infer this from an example in the (non-binding) rationale associated with the standard, which illustrates a possible implementation of `pthread_getspecific()`$_{119}$ for certain architectures in which it takes advantage of being a macro to assign to its `key` argument; something which would be impossible if it were passed, or could be passed, by value.

there is no way to pre-initialize them (unlike the private data of processes, say, which are initially copied form the processes' parents). This can be a nuisance, but there is a payoff in performance—it makes it possible to implement these facilities so as to read and write thread-specific data without inter-thread synchronization delays.

`PTHREAD_KEYS_MAX` Some implementations of pthreads may trade off flexibility for efficiency and predictability, and allow only a fixed number of thread-specific data keys to exist within a process. This number is available as the value of the macro `PTHREAD_KEYS_MAX`, which is guaranteed to be no less than 128.

6.2.1 Creating a data key

Thread-specific variables are not named directly by C identifiers, but indirectly through **keys.** Before being used, such a key key must be initialized by calling `pthread_key_create()`. This function takes an additional argument (which may be `NULL` if it isn't needed): the address of a function to be called when any thread is about to terminate with a non-`NULL` value stored under the new key.$\Rightarrow^{6.2.5}_{120}$ This function receives the stored value as an argument and is responsible for cleaning it up as appropriate. The fundamental assumption behind this idea is that the key will be treated like a variable and always hold the same *actual* type of data (even though the data accepted are just declared as `void *`).

`pthread_key_create()` **17.1.1**

```
#include <pthread.h>

int /*errno*/ pthread_key_create
  (pthread_key_t *key, void (*destructor)(void *));
```

`pthread_key_t`$_{114}$ `*key`
: Pointer to the location in which to store the newly created thread-specific data key.

`void`$_{27}$ `(*destructor)(void *)`
: Function to be called on thread exit to deallocate non-`NULL` thread-specific data stored under the new key. May be `NULL` if no cleanup is required.

A new thread-specific data key is created and the given destructor is associated with it. This destructor is applied to data stored under the key whenever a thread terminates. This can help in avoiding memory (or other resource) leaks. Be aware that use of `pthread_setspecific()`$_{118}$ or `pthread_key_delete()`$_{117}$, and *process* exit, never result in the destructor being invoked.$\Rightarrow^{6.2.5}_{120}$

This function returns an integer containing an error number.$\Rightarrow^{A}_{271}$

On success `pthread_key_create()` returrns `0` and places a newly allocted key in the location pointed to by *`key`*.

On failure it returns an error code, and no promises are made about the value of **key*:

ENOMEM$_{271}$ The process lacks the memory to create another key.

EAGAIN$_{271}$ The process lacks other resources needed to create another key. Perhaps PTHREAD_KEYS_MAX$_{115}$ keys have already been allocated.

Warning: Although C allows pointers to functions to be freely cast to each other, the actual and declared types of a function that is accessed through a pointer[3] *must actually match* at the moment that the function is called. This is because if a change of representation is needed for an argument or result it will be done on the basis of the type known at the call-point—the type associated with the pointer—not the 'real' type of the function. For example, this code will typically break, because the compiler thinks `(*d)()` (*i.e.* `f()`) takes and returns `float`s:

```
double f(double x) {return x;}
float (*d)(float) = (float (*)(float))f;
...
    double g = d(1.0);
```

For most compilers on byte-addressed hardware (in 1996 almost all workstations are byte-addressed), type mismatch will not be a problem in practise for arguments that are data pointers (since they choose the same representation for all pointer types) so long as they don't have variable argument lists; but there are no official guarantees. Thus, technically speaking, passing a pointer to a function of type `void ()(char *)` for the *destructor* argument—which expects a `void *` parameter—is incorrect. Instead, you should write and pass a **wrapper** function. Here are a couple of frequently-used destructor wrappers:

```
#include <stdio.h>

void fflush_destructor(void *fp) {fflush(fp);}
void fclose_destructor(void *fp) {fclose(fp);}
```

(No cast is necessary in the calls of `fflush()` and so on because C *does* provide automatic conversion between `void *` and other data pointer types in contexts where it's clear what needs doing—as it is here, from the definition in `stdio.h` of `fflush()`'s `FILE *` argument type.)

In the future this may matter more, since byte-addressability is a fashion, not a necessity.[4]

The good news is that, according to Standard C (and hence POSIX), the type of `free()` is exactly `void ()(void *)`—thus `free()` *is* a valid thread-specific storage

[3] After applying the Miranda prototypes and so on.

[4] A machine with 64-bit words and designed for the 16-bit Unicode character set rather than 8-bit 'ASCII' [really ASCII is 7-bit and the various 8-bit codes are all something else] might sensible use word addressing by the same reasoning that current machines use byte addressing; or then again it might decide that there are plenty of address bits now and provide full bit addressability for the benefit of—increasingly important—cryptography and compression code [wider codes for both characters and instructions make compression even more attractive].

destructor; there is no need for you to write a wrapper function to fix the types up in this important special case.

6.2.2 Destroying a data key

After a thread-specific data key has served its purpose, it can be discarded by calling `pthread_key_delete()`. This does *not* invoke the destructor for the key in threads that still have a value stored for it, though it does make the key unavailable for further use; you should therefore make sure that each thread has done all necessary cleanup before discarding the key.

`pthread_key_delete()` **17.1.3**

```
#include <pthread.h>

int /*errno*/ pthread_key_delete(pthread_key_t key);
```

`pthread_key_t`$_{114}$ *`key`*
The key (*not* a pointer to it) to be destroyed.

Deletes the thread-specific data `key`. It is then no longer available for use. This function does *not* call the associated destructor⇒$^{6.2.5}_{120}$; the thread-specific data bindings for this key are simply forgotten. It is legitimate to call this function from within the destructor associated with a thread-specific data key.

On success returns `0`.

On failure returns an error code:

`EINVAL`$_{271}$ It was noticed that the *`key`* was invalid.

The reason for the decision not to make `pthread_key_delete()` invoke the registered destructor is that in order to make any sense, a destructor presumably has to be invoked in the context of each thread which has a value to be destroyed; but at the moment that one thread deletes a key, nothing can be said about states of the other threads that might have a value for it. Invoking destructors asynchronously in this manner could not be made safe. On the other hand, in order for it to be safe to destroy a key, you must know that no other thread is going use it in the future. Other threads that use the key can thus do their own finalization as part of the process of providing notification that they have finished with it.

On the other hand, if the appropriate synchronization is done, destructors *may* invoke `pthread_key_delete()`, even for the key with which they are associated.

6.2.3 Storing private data

Because thread-local variables are, like everything else to do with threads, not built in to C, accessing them is not quite trivial. To store a value in the thread-specific data area, one calls `pthread_setspecific()` with a key of type `pthread_key_t`$_{114}$ identifying the private variable to use, and the value to be stored (which must be a `void *`—that is, a pointer to a non-function datum). The real type of the datum should match the actual type expected by any destructor routine associated with the key. The datum can be retrieved later using `pthread_getspecific()`$_{119}$.

`pthread_setspecific()` **17.1.2**○

```
#include <pthread.h>

int /*errno*/ pthread_setspecific
  (pthread_key_t key, const void *value);
```

`pthread_key_t`$_{114}$ *`key`*
  Key under which to store a datum.

`const void`$_{27}$ *`*value`*
  The datum to store.

Associates the pointer *`value`* with the *`key`* in the private data area of the calling thread so that it can later be retrieved with `pthread_getspecific()`$_{119}$. The key must have been previously allocated with `pthread_key_create()`$_{115}$, and must not have been deleted. If `value` is non-`NULL` and the thread exits with the datum still associated with this key, the key's destructor will be called on it.

On success returns `0`; the pointer has been stored.

On failure returns an error code:

- `ENOMEM`$_{271}$ The process lacks the memory to record the association between the key and the datum.
- `EINVAL`$_{271}$ It was detected that the key is invalid.

This function may be implemented as a macro.

Unlike storing into a conventional variable, it is possible for this operation to fail because there is not enough memory to build necessary internal data structures; to avoid this the implementation would have to allocate enough space for all keys in every thread, which is wasteful.

6.2.4 Retrieving private data

The macro that retrieves thread-specific data from the current thread's private storage, `pthread_getspecific()`, has been optimized for speed and convenience; unlike almost all other pthreads functions it returns the value retrieved directly, and does not report an error status in any way. Its sole argument is a key; if no value is stored under that key in the current thread, it returns `NULL`, and this is indistinguishable from an explicitly stored `NULL`.

`pthread_getspecific()` **17.1.2**○

```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);
```

`pthread_key_t`$_{114}$ *key*
  Key from which to retrieve data.

Retrieves the value associated with *key* in the calling thread's private data area. If no value has yet been stored, returns `NULL`. The key must previously have been allocated with `pthread_key_create()`$_{115}$, and must not have been deleted.

Returns the value retrieved.

This function may be implemented as a macro. Note that it does no error checking; in particular, it is the programmer's responsibility to ensure that the *key* is valid.

One might ask why it is that an exception has been made to the pthreads calling conventions for `pthread_getspecific()`; after all, there are many other functions that rarely if ever fail and have a 'natural' return value, but they have all been made to return an error number in conformity to the thread package conventions. The answer is returning the retrieved value as the function's main result permits a coding hack that is essential to making old code threadable: it allows for the semblance of true thread-local variables. It's done like this:

Suppose that some global variable `foo` of type `foo_ty` is to be made thread-specific. Allocate a new thread-specific data key:

```
pthread_key_t foo_k;
```

The key can usually be declared in the same place that the original global (or static) variable was. Ensure that `foo_k` id properly initialized during the startup of the overall program, giving `free()` as its destructor:

```
    pthread_key_create(&foo_k, free);
```

and that during the initialization of each thread that needs to use `foo` there is some storage backing it:

```
    pthread_setspecific(foo_k, malloc(sizeof foo_ty));
```

Now a simulation of the original variable, `foo`, can be installed with the following macro definition:

```
#define foo (*(foo_ty *)pthread_getspecific(foo_k))
```

Note that this macro now expands to an **lvalue** that can be legally assigned to–or have its address taken—as well as read; it really does behave just like a variable.

Of course, the simulation of a C variable is not perfect. Because it is a macro, `foo` can now capture things other than the original variable having the same name: `struct` and `union` fields, formal parameters, local variables and so on. Furthermore, because it relies on compile-time substitution of the definition for the name it requires *recompiling* of all modules that refer to it. And it will be a little slower than a global variable would be. This situation is actually identical to that of `errno`$_{271}$: with `int` substituted for `foo_ty` and `errno` for `foo`, this might be roughly[5] how your implementation provides each thread with its own `errno` variable, as the pthreads standard requires.

6.2.5 Destroying thread-specific data

When a POSIX process terminates, it is completely, but not systematically, dismantled—many of its resources, notably memory, can (in the usual case) simply be discarded. The situation is different with threads, whose protection domain typically outlives them: other, longer-lived threads still need those resources. Nor does C provide the kind of garbage-collected environment that would be necessary to selectively reclaim just those resources made inaccessible by thread termination; any pointers, file handles, or other data left dangling by thread termination will continue to burden the process for the rest of its existence.

One of the two important facilities provided by pthreads to help deal with this problem—the other is the cleanup system described in of the second half of this chapter—is the thread-specific data **destructor** function that is passed as the second argument to `pthread_key_create()`$_{115}$ when a key is first generated. The idea is that each key is associated with a function that encapsulates the resource release, storage deallocation and general cleanup appropriate to the type of value that the key will store. This way the programmer need not take (much) special care when terminating a thread; whether the thread falls off its own end (as it were), calls `pthread_exit()`$_{37}$ or is **canceled**$\Rightarrow^{11}_{203}$, this cleanup code will run.

The destructor is *not* run if the stored value is `NULL`, and uninitialized thread-specific variables are implicitly set to this value; this is convenient, since it ensures that threads 'bailing out' from an error with partially initialized data structures are relatively likely to do the right thing without extra programmer effort.

There are, however, a number of caveats. Destructors are run in an unpredictable order—which is nasty if there are interdependencies between the data stored. And although they are always run if a single thread terminates, they are *not* run in the following circumstances:

- when a key is deleted (with `pthread_key_delete()`$_{117}$);
- when a value is *overwritten* (with `pthread_setspecific()`$_{118}$); or
- when the entire *process* is exiting.

[5] Not exactly, since we must be sure that `errno` continues to work until *after* all the other destructors have been called.

In each of these cases the requisite cleanup is the programmer's responsibility. Only in the first case is this likely to be a burden; but since keys cannot be used after deletion, every thread concerned must presumably have already performed an appropriate synchronization in which it agreed to the key's decommissioning, and at that time it could have done any cleanup that it needed to.

Iterated destruction 'The observant reader,' as they used to say, may have noticed that there is no rule against a destructor calling `pthread_setspecific()`118, thereby creating a new thread-specific data binding during the cleanup process. Obviously this too should be cleaned up, but equally obviously this could result in an infinite loop. There's no clear answer to the question of what should be done when this happens.

What the standard specifies is that the implementation should try at least `PTHREAD_DESTRUCTOR_ITERATIONS` times to remove all the thread-specific data for which there are destructors, and this number should be at least four; it is, however, allowed to loop forever if the destruction process does not converge—so do be careful.

6.3 General-purpose cleanup handlers

In the last section we saw the facilities POSIX provides for thread-specific data and the way in which destructors can be used to ensure that the locally stored values are freed on thread termination. In general, however, other effects than the allocation of variables may have to be undone when a thread exits: open files may have to be closed, locked mutexes may have to be released, the thread itself may have to be removed from software 'directory' structures and so on.

Furthermore, the objects subject to this kind of general cleanup processing are, unlike thread-specific data—which are specifically designed to be quasi-static in nature—often of dynamic extent: in general, cleanup may have to be done on exiting from any scope. Finally, and most significantly, it is possible to distinguish two distinct flavors of cleanup. First, there is routine **teardown** that undoes setup local to a scope whenever it is exited, normally or not; closing a locally-opened file might fall in this class. But second, there is **abort processing**, processing that is invoked specifically when leaving a scope precipitately, as the thread exits directly from somewhere inside it.

POSIX provides a pair of macros that simplify both kinds of cleanup: `pthread_cleanup_push()` and `pthread_cleanup_pop()` bracket an area of code, associating with it a function (and its argument) to be called when control leaves.

Although the distinction between routine cleanup and abort processing is conceptually clear enough, it is not always statically determined; for example, it might be the case that an opened network socket should be closed *unless* it has already been handed off to another thread. For this reason, rather than providing distinct macros for the two cases, a boolean parameter to `pthread_cleanup_pop()`122 is used to specify whether the cleanup function should be called on normal scope exit normally (the cleanup function is *always* called if the scope is left precipitately; that's the point).

As noted, these macros delimit a *scope;* they are a bracketing pair and not normal function-like macros at all (though the syntax supported by the C preprocessor imposes a function-like appearance on them). They must (like the built-in `while...do` keywords) always appear in balanced, well-bracketed pairs. They are also subject to some rather heavy constraints on flow of control: it is forbidden to jump in or out of such a pair by any means—including `goto`, `break`, `continue`, `return`, `longjmp()`, `siglongjmp()`—that does not terminate the thread. The reason for this Draconian restriction is that the macros can then be implemented without special support from the compiler.

6.3.1 Managing cleanup handlers

Cleanup handlers are (or rather may be) managed on the stack like all other scope-specific data in C. A handler, consisting of a pointer to a `void` function and an argument for that function, is pushed onto the stack with `pthread_cleanup_push()`, and remains in force until it is popped with `pthread_cleanup_pop()`. The latter has a boolean argument which, if true, signifies that the function is a teardown function and should be called as it is popped; otherwise the entry is simply discarded.

When a thread exits (other than because its process terminates), its active cleanup handlers are popped and executed in stack order before the corresponding stack frames are destroyed. The two macros form a bracketing pair and control may not jump in or out of the area they protect.

`pthread_cleanup_push()`...`pthread_cleanup_pop()` **18.2.3**○

> **Note that these are not functions:**
> The macros `pthread_cleanup_push()` and `pthread_cleanup_pop()` are the two ends of a bracketing construct (à la `do`...`while ()`).

```
#include <pthread.h>

pthread_cleanup_push
  ( void (*function)(void *argument)
  , void *argument
  );

pthread_cleanup_pop(int /*boolean*/ execute);
```

`void`$_{27}$ `(*`*`function`*`)(void *`*`argument`*`)`
: Cleanup function to be registered.

`void`$_{27}$ `*`*`argument`*
: Programmer-supplied argument to *`function`*`()`.

`int` *`execute`*
: If true, execute *`function`*`(`*`argument`*`)` as the entry is popped.

The *`function`* is registered by `pthread_cleanup_push()` as a cleanup handler until it is deregistered by the corresponding call of `pthread_cleanup_pop()`. If the calling thread exits⇒$^{4.4}_{37}$, is canceled⇒$^{11}_{203}$, or otherwise terminates, *`function`* will be called with the *`argument`* supplied. At the time the *`function`* is invoked, the scope in which it was pushed is still active.

Otherwise, if *`execute`* is nonzero when `pthread_cleanup_pop()` is called, the registered function is called after it is popped from the cleanup stack.

You may not `longjmp()` or `siglongjmp()` out of a cleanup scope or a cleanup function. You may not `goto`, `break`, `continue`, or `return` into or out of a cleanup scope.

These routines report no error conditions. This may be problematic for systems that employ heap storage in their implementation of this feature (but it is not necessary for them to do so).

It is instructive to compare the pthread cleanup facility with the traditional `at_exit()` mechanism for tidying up on process exit. They perform similar functions, but pthread cleanup is a much more flexible and modern design (though also far from perfect).

The single most obvious difference is that the cleanup facility is *block structured*—cleanup functions have scopes which are entered and left, and the handlers are discarded on exit. On the assumption that there are specific pieces of setup that are of local importance, this is a huge improvement. In particular, it makes it safe to use the mechanism to clean up after uses of stack-allocated variables; `at_exit()` enforces the use of a global variable oriented programming style, which some stylists consider to be inexcusable (and is certainly problematic when used heavily in conjunction with threads). It also means that the internal storage of the cleanup facility can be allocated on the stack, which by conventional wisdom frees it from reporting memory shortages (though in fact it just means that such a shortfall is unrecoverable and unpredictable in its effects, instead of being at least theoretically handlable. With multiple threads, the fiction of an unlimited stack is much, much harder for the implementation to maintain $\Rightarrow^{8.3.2}_{145}$). At very least it removes the temptation for an implementation to impose a fixed limit on the number of cleanup handlers available.

The ingredient needed to make block-structured cleanup work[6] is for the cleanup function to take an argument, to allow it to see *into* the local scope it is supposed to be cleaning up. In accordance with post-standardisation C style, of course, `pthread_cleanup_push()` uses a generic pointer to do this. Even for cleaning up global structures this is an improvement, since it means that the cleanup functions can be reusable, instead of being bound to specific global locations. Notably, `free()` is an appropriate argument.

Finally, thread cleanup is integrated with normal processing: use of the *`execute`* argument of `pthread_cleanup_pop()` allows cleanup to be invoked on *normal* scope exit without redundant expression of the procedure to be followed. Once cleanup has been encapsulated in a function this saves only a single statement, but its positive effect on the maintainability of code is fairly great, since it reduces both code replication and the chances of **race conditions**$\Rightarrow^{5.6.1}_{92}$ in which the same cleanup is performed twice (once explicitly and once by the handlers) in a thread that exits unexpectedly.

The main price paid for these enhancements is that the integration with the block structure of the program is very strong—stronger even than that of the built-in control

[6] At least in a language that lacks local functions!

structures (which at least have the benefit of compiler co-operation). This can be less than convenient, especially for programmers who make heavy use of the `goto` and `longjmp()` mechanisms, with which it is incompatible. At the same time, experience with languages that use this kind of construct heavily (COMMON LISP, for instance) is that it more than pays for itself in terms of increased simplicity and safety.[7]

The cleanup functions are sometimes convenient, but they are *essential* under two sets of circumstances: when writing code for a thread that is subject to **cancelation** $\Rightarrow^{11}_{203}$, and when a thread holds resources across the invocation of an externally specified function—typically passed as a parameter—that might call `pthread_exit()`$_{37}$. Each of these cases introduces the possibility of thread exit not requested by, and thus not compensated for by, local code.

Cautious cleanup is especially important in the case of library code, since it must assume the worst about its context: that thread cancelation will be attempted, and that functional arguments will indeed sometimes fail to return.

Sometimes cleanup of structures global to the processing carried out by a thread is also required (notably for things like caches or tables that are initialized once and then endure indefinitely), but thread-specific data and their destructors are provided for exactly this case.

6.4 Example

For our example of thread-specific data and cleanup handlers, we look at providing a solution to the related problem of thread *exit* handlers. These, too, provide finalization handling, but unlike cleanup handlers they can be deleted in any order, on presentation of a handle that is optionally returned by the `_add` operation that installs them. We have borrowed from cleanup handlers the idea of optional execution as part of the deletion operation.

spg_atexit — spg_atexit.h

```
typedef struct spg_atexit *spg_atexit_ty;

extern int /*errno*/ spg_atexit_add
  ( spg_atexit_ty *atexit_id
  , void (*function)(void *argument)
  , void *argument
  );
extern int /*errno*/ spg_atexit_del
  ( spg_atexit_ty *atexit_id
  , int /*boolean*/ execute
```

[7] There is still some question of how well the idea has been captured, however: these other languages all either specify the cleanup action as a *block* or allow local functions, thus giving the cleanup code direct access to local variables. Meanwhile, C has no choice but to use a separate top-level function. Worse yet, because the language cannot store an argument *list*, that function must have a fixed type in the generic-pointer style. In practise this means that it must always be specially coded: even if the ideal cleanup function already exists, it will have a more specific type than `void ()(void *)` and will have to be called through a wrapper. Perhaps more than any other in pthreads the cleanup facility cries out for a more intelligent preprocessor.

```
);
```

The list of currently active thread exit handlers is thread-specific, and stored under a key allocated by the exit handler system upon its first execution. We've chosen to protect the allocation operation with a statically initialized mutex rather than the over-specialized `pthread_once()`$_{89}$ system.

spg_atexit (continued) spg_atexit.c

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "spg_atexit.h"

struct spg_atexit
  { spg_atexit_ty next, prev;
    void (*function)(void *argument);
    void *argument;
  };

static int inited = 0;
static pthread_key_t key;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

static void done(void *argument)
  { spg_atexit_ty id, id_next;

    for (id = argument; id; id = id_next)
      { id_next = id->next;
        id->function(id->argument);
        free(id);
      }
  }

int /*errno*/ spg_atexit_add(spg_atexit_ty *id, void (*f)(void *a), void *a)
  { int x;
    spg_atexit_ty new;

    x = pthread_mutex_lock(&mutex);
    if (x) return x;
    if (!inited)
      { x = pthread_key_create(&key, done);
        if (!x) inited++;
      }
    pthread_mutex_unlock(&mutex);
    if (x) return x;
    new = malloc(sizeof(*new));
    if (id) *id = new;
    if (!new) return ENOMEM;
    new->next = pthread_getspecific(key);
    if (new->next)
```

```
    new->next->prev = new;
  pthread_setspecific(key, new);
  new->prev = NULL;
  new->function = f;
  new->argument = a;
  return 0;
}

int /*errno*/ spg_atexit_del(spg_atexit_ty id, int /*boolean*/ execute)
{ spg_atexit_ty head = pthread_getspecific(key);

  if (!head) return EINVAL;
  if (id == head)
    { head = head->next;
      head->prev = NULL;
      pthread_setspecific(key, head);
    }
  else
    { if (id->next)
        id->next->prev = id->prev;
      id->prev->next = id->next;
    }
  pthread_cleanup_push(free, id);
  if (execute)
    id->function(id->argument);
  pthread_cleanup_pop(1);
  return 0;
}
```

Maintaining the list of outstanding handlers is straightforward, but suffers from a single wrinkle: we must make sure that the internal record that records the handler is actually freed even if the thread exits during the execution of the user's handler function. This—ensuring the cleanup happens after execution of externally specified code—is a classic application of the cleanup handler: `free(id)` will be called by the `spg_atexit_del()` even if control does not return to it normally (assuming that the user is not so rude as to `longjmp()` out, at any rate).[8]

6.5 Summary

In this chapter we've seen how a POSIX thread can have private variables; just as `auto` and `register` variables receive separate bindings in each function invocation, thread-specific storage can assume an independent value in each thread. Not being built in to the C compiler they don't behave quite like ordinary variables: in particular, they are named not by identifiers but indirectly through the vaguely pointer-like objects of

[8]In fact, it would have been possible—and slightly more efficient—to code this example in such a way that the free is performed *before* the user code is invoked, with the contents of `*id` being stored until use in an auto variable. In the general case—had the example involved a file handle, for instance—no such workaround is possible.

type `pthread_key_t`$_{114}$. Their values are always of the generic pointer type `void *`, and they are accessed through the macros `pthread_getspecific()`$_{119}$ and `pthread_setspecific()`. Nonetheless it is possible to use the macro processor to simulate the syntax of a static or global variable—a useful trick when rehabilitating old code.

In order to help with resource management when a thread exits precipitately, thread-specific variables are provided with **destructor** functions that handle cleanup automatically. You can also associate cleanup functions with local scopes by using `pthread_cleanup_push()`$_{122}$ and `pthread_cleanup_pop()`$_{122}$. All active cleanup functions are automatically invoked if the thread is canceled or exits, and `pthread_cleanup_pop()` can optionally do cleanup on normal exit, too. No cleanup is done, however, if the whole *process* exits, whether because of a signal, a call to `exit()`, or some other cause.

In the next chapter we examine the optional **attributes** of pthreads, mutexes and condition variables, in particular those that control thread *detach state* (which controls how cleanup of the thread object itself takes place), *stack allocation* (which enables the programmer to control the allocation of the storage for automatic variables and the thread's internal control information), and *shared memory* by which—on implementations that support it—threads in different *processes* can share synchronization variables and thus provide consistent access to other memory objects.

Appendices

6.a A look inside

6.a.1 Thread-specific data

A thread specific datum is visible only within a single thread, and then only on presentation of the appropriate **key** object. Thus, from the global perspective, each datum is identified by a ⟨thread, key⟩ pair. Thread specific data might therefore be stored in a single global table; or the association could be factored in either of two ways. First, each thread could be given a table mapping keys to values; or second, each key could contain a table of values indexed by thread IDs. Quite simple.

In practise, storing the information in the thread object is preferable since (if we are very careful, and in particular do not put any modifiable data inside a key) it renders locking unnecessary: one thread will never examine a structure while another is changing it. All the modifiable parts of the structure are accessed only by the owning thread. Unlike the end user, the implementor of pthreads has control of the contents of `pthread_t`$_{32}$ and can easily add a new field to its internal representation if necessary; if keys are made to be integers, the implementation of `pthread_getspecific()`$_{119}$ can be almost as simple as `pthread_self()->specific[key]`. This does not even commit the implementation to a prior bound on the number of keys issued, since `realloc()` (say) can be used to extend the array of thread-specific values as new stores make it necessary. Unfortunately, as we shall see in a moment, this extension of the thread's private storage *does* require a lock (even though `pthread_getspecific()` and the usual, non-extending case of `pthread_setspecific()` still do not).

The semantics of key creation and deletion—that the user is responsible for synchronizing between the thread allocating or destroying the key and the threads that use it—are important here; if, for example, destroying a key were made to block until the key was no longer in use (semantically a very reasonable idea), keys would now be shared resources with locks requiring frequent manipulation. But as it is, no such synchronization is required.

There is, however, one wrinkle: the POSIX committee has stipulated that the value associated with a newly-created key is always `NULL`. This means that *either* keys cannot be re-used, *or* `pthread_key_delete()`$_{117}$ must reset existing mappings for the key being 'deleted'—*i.e.* put up for re-use—to `NULL`, in all threads that use it (this one operation would have been much simpler if the key, and not the thread, had been the physical locus of storage).[9] The problem cannot simply be ignored, since a long-running program might repeatedly allocate and deallocate keys and it is unacceptable, if the number of keys in use stays under a reasonable bound, either for the threads using those keys to run out of memory or for the supply of new keys to dry up.

The restrictions on key manipulation guarantee that there is no race between the thread deleting the key and the thread owning the corresponding field, for the latter has already relinquished the association. *However,* the key-deleting thread must still traverse the associative data structure in the thread to reach the field it is nulling, and *this* requires mutual exclusion with any computation that might modify the path it follows: notably, the local-storage-extending case of `pthread_setspecific()`.

Fortunately, both growing local storage and deallocating keys are relatively rare events (at least by comparison to calling `pthread_getspecific()`); if synchronization overhead must be paid somewhere in the system, this is a good place to pay it.

For an implementation in this style, the relevant part of `pthread.h` will look like this:

```
typedef struct
  {
    ...
    pthread_mutex_t _specific_m;                  /* Lock for the following */
    unsigned _nspecific;                /* #slots allocated in this thread */
    void *_specific;                             /* thread-specific storage */
    ...
  } *pthread_t;
typedef unsigned pthread_key_t;

extern int /*errno*/ pthread_key_create
  (pthread_key_t *key, void (*destructor)(void *value));
extern int /*errno*/ pthread_key_delete(pthread_key_t key);

extern void *pthread_getspecific(pthread_key_t key);
extern int /*errno*/ pthread_setspecific(pthread_key_t key, void *value);

#define pthread_getspecific(key)                                           \
  ( (key) < pthread_self()->_nspecific                                     \
  ? pthread_self()->_specific[key]                                         \
  : NULL                                                                   \
  )
#define pthread_setspecific(key, value)                                    \
  ( (key) < pthread_self()->_nspecific                                     \
```

[9] Actually there is another possibility: each valid key could contain a 64-bit or so **generation number** telling how many times it had been used, and this value would be stored along with the user's data by `pthread_setspecific()`. If `pthread_getspecific()` is used on a value in an old generation, it now returns `NULL` instead of the stored value. This is not, however, very attractive in terms of efficiency!

```
  ? (pthread_self()->_specific[key] = (value), 0)                          \
  : pthread_setspecific(key, value)                                        \
  )
```

Note that we provide both macro and function implementations of `pthread_getspecific()` and `pthread_getspecific()`. In fact, the macro definition of `pthread_setspecific()` relies on a subtlety of the Standard C macro processor which guarantees that a macro is never expanded inside its own expansion to delegate the hard case in which storage for the key has not been allocated yet to the function version.

```
#include <malloc.h>
#include <pthread.h>

#define none (~0UL)

static unsigned nspecific = 0;
static pthread_key_t free = none;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static struct key
  { void (*destructor)(void *argument);
    pthread_key_t next_free;
  } *table = NULL;

int /*errno*/ pthread_key_create(pthread_key_t *key, void (*d)(void *))
  { int x = 0;

    pthread_mutex_lock(&mutex);
    if (free == none)
      { unsigned newn = nspecific + 1;
        struct key *new_table =
          realloc(table, newn * sizeof(struct key));

        if (new)
          { unsigned i;

            for (i = nspecific; i < newn; i++)
              new_table[i].next_free = i + 1;
            new_table[newn - 1].next_free = none;
            free = nspecific;
            table = new_table;
            nspecific = newn;
          }
        else x = ENOMEM;
      }
    if (free != none)
      { *key = free;
        table[free].destructor = d;
        free = table[free].next_free;
      }
    pthread_mutex_unlock(&mutex);
    return x;
  }

int /*errno*/ pthread_key_delete(pthread_key_t key)
  { pthread_t thread;
```

```
    pthread_mutex_lock(&mutex);
    table[key].next_free = free;
    free = key;
    _pthread_kernel_lock();
    for (thread = allqueue; thread; thread = thread->aqnext)
      if (key < thread->_nspecific)
        thread->specific[key] = NULL;
    _pthread_kernel_unlock();
    pthread_mutex_unlock(&mutex);
    return 0;
  }

/* This is a hack to prevent unwanted macro expansion. */
#define FUNCTION

void *pthread_getspecific FUNCTION(pthread_key_t key)
  {return pthread_getspecific(key);}

int /*errno*/ pthread_setspecific FUNCTION(pthread_key_t key, void *value)
  { pthread_t self = pthread_self();
    int x = 0;

    if (key < self->_nspecific)
      { void **s;
        const unsigned newn = key + 1;

        pthread_mutex_lock(&self->_specific_m);
        s = realloc(self->_specific, newn * sizeof(void *));
        if (s)
          { unsigned i;

            for (i = self->_nspecific; i < newn; i++)
              s[i] = NULL;
            self->_specific = s;
            self->_nspecific = newn;
          }
        else x = ENOMEM;
        pthread_mutex_unlock(&self->_specific_m);
      }
    self->_specific[key] = value;
    return x;
  }
```

Note that while we've defined `pthread_getspecific()` entirely in terms of the macro definition (using another preprocessor rule that forbids moving 'backwards' in the source file to locate an expandable macro to prevent spurious expansion of the function header), `pthread_setspecific()`, as promised, handles the general case in which the vector of local values needs to be expanded.

Finally, there is an internal function that is responsible for making sure that the destructors are invoked; it must be called late in the execution of `pthread_exit()`, before the thread finally terminates.

```
void _pthread_run_destructors(void)
  { unsigned i, j;
    void **specific = pthread_self()->_specific;
```

```
    pthread_mutex_lock(&mutex);
    for (i = 0; i < PTHREAD_DESTRUCTOR_ITERATIONS; i++)
      for (j = 0; j < nspecific; j++)
        { void *t = specific[j];

          if (t)
            { specific[j] = NULL;
              table[j].destructor(t);
            }
        }
    pthread_mutex_unlock(&mutex);
  }
```

6.a.2 Cleanup handlers

The simplest implementation of cleanup handlers is with two macros that expand to the start and end of a block (that is, we give them expansions containing individually unmatched braces) and a cell of thread-specific data:

```
struct __cleanup
  { struct __cleanup *prev;
    void (*function)(void *argument);
    void *argument;
  };
typedef struct
  {
    ...
    struct __cleanup *_cleanup;
    ...
  } *pthread_t;

#define pthread_cleanup_push(f, a)                                  \
  { struct __cleanup __cleanup_link;                                \
    __cleanup_link.function = f;                                    \
    __cleanup_link.argument = a;                                    \
    __cleanup_link.prev =                                           \
      pthread_self()->_cleanup;                                     \
    pthread_self()->_cleanup = &__cleanup_link;

#define pthread_cleanup_pop(x)                                      \
    pthread_self()->_cleanup = __cleanup_link.prev;                 \
    if (x) __cleanup_link.function(__cleanup_link.argument);        \
  }
```

The thread exit logic will just need to look at the terminating thread's `_cleanup` slot and unwind the linked list if finds there to implement the rest of the cleanup semantics, like this:

```
void _cleanup_unwind(void)
  { struct __cleanup *c;
```

```
    for (c = pthread_self()->_cleanup; c; c = c->prev)
      c->function(c->argument);
  }
```

An even cleverer idea would be to use thread-specific storage to hold the cleanup chain and install something like `_cleanup_unwind()` as the associated destructor, letting this automatic mechanism take care of everything for us. Unfortunately this wouldn't be correct, since the standard requires that cleanup handlers execute *before* the destructors for thread-specific data, while this admittedly elegant solution would interleave them.

In fact, as an extension of the behaviour required by the standard, CAPthreads implements both `pthread_cleanup_push()`$_{122}$ and `pthread_cleanup_pop()`$_{122}$ as *functions*, keeping the cleanup links in heap-allocated storage. This has the advantage that pushes and pops can be managed programmatically, making, *e.g.*, returns from interior blocks manageable. Unfortunately, it also introduces the remote possiblity that a push will fail for want of memory. This is reported in CAPthreads by having `pthread_cleanup_push()` return an integer error code (`ENOMEM`$_{271}$ when the push failed), but portable code is forbidden to check this (since—to allow definitions like the above—the standard allows it to be a macro that does not even expand to an expression).

Note, by the way, that the apparent race in `pthread_cleanup_pop()` above (imagine an interruption between the first and second lines of the expansion) is not pernicious: as we will see in a later chapter, cancelation cannot normally occur at this point.

6.b Non-posix analogs

6.b.1 DCE

The thread-specific data facility of DCE pthreads is similar to that of POSIX except that there is no means to deallocate a key once `pthread_keycreate()` has built it. The interface to `pthread_getspecific()` is slightly different as well; rather than using the streamlined calling convention that returns the value retrieved as the value of the function, DCE has it follow the usual pthreads calling convention in parallel with `pthread_setspecific()`: it assigns the thread-specific value to a reference parameter and returns a status code as result. This allows it to return `EINVAL`$_{271}$ if the key is invalid, but tends to clutter up user code and makes direct macro substitution for global variables impossible.

The absence of a key deallocation routine slightly simplifies the implementation internally; see the discussion in the previous section.

The DCE cleanup facility is essentially the same as we described for POSIX.

6.b.2 MACH Cthreads

MACH's thread-specific data facilities are very much simpler than pthreads'. They consist of two functions, `void cthread_set_data(cthread_t thread, any_t data)` and `any_t cthread_data(cthread_t thread)` (`any_t` is essentially `void *`) and can store only a *single* pointer for each thread; there is no notion of a 'key' as there is in POSIX. The thread whose private data are to be manipulated is passed as an explicit parameter.

On the one hand, these functions provide a method for the programmer to associate user data with a thread that can be manipulated even from without; this is very convenient at times. On the other hand, no means is provided for co-ordinating use of this single pointer by different facilities. Thus, although more sophisticated systems can be built upon it, in the real world more than one programmer is likely to attempt this, resulting in conflict and incompatibility.

6.b.3 GATech Cthreads

[[@@@@ we need to discuss the whole memory model situation as well as the cthread_data call itself.]]

6.b.4 True thread-private variables

Many other thread systems end up with all stack-allocated storage visible *only* to one thread; the same address in the stack segment refers to different physical storage from one thread to another. This can be accomplished through the manipulation of virtual memory address mappings, but more or less requires a kernel-level thread implementation. This is all very different from the pthreads model in which *pointers* are invariably valid throughout all threads of a process.

Systems which allocate completely private stacks have still not provided an automatic solution to the problem of thread-specific data, however. The problem is that although there are memory addresses that inherently have the desired property of providing thread-specific *storage,* there is still no mechanism within the C language to name it—the only variables that are normally allocated within the stack are locals.

Given a suitable language extension, the physical implementation would be straightforward: since stacks are *not* shared, thread-specific variables could be allocated at fixed addresses within the stack area. Of course, such a language extension could equally well be implemented in a common-address-space system with a scheme like that we described for pthreads, perhaps simplified to allocate per-thread variables directly in the thread descriptor. Performance might suffer very slightly from the need to load `pthread_self()` into a register and compute thread-local variables' addresses as an offset from it.

Chapter 7

Posix Inter-Process Communication

[[sps: This tentatively projected chapter explores the posix IPC tools, which provide (a) alternative synchronization facilities which are specified to work correctly for pthreads when available; and (b) shared memory, which has to be understood to appreciate the need for the pshared attributes described in the next chapter. None of this material falls within the `pthread_` namespace, though.

Introducing it will let us streamline material elsewhere, and its location seems to be consistent.]]

7.1 Shared Memory

[[sps: A picture might be good?

Not a very good picture: no threads appear!]]

7.2 Semaphores

7.3 Message Queues

Chapter 8

Pthread Object Creation Attributes

8.1 Introduction

Threads and synchronization objects may have a number of options and parameters in some implementations, making them fairly complex data structures. Rather than adding a lot of parameters to the creation calls for the types `pthread_t`, `pthread_mutex_t` and `pthread_cond_t`, pthreads uses a system of **attributes objects**, which are a kind of template from which other objects are constructed. After an object has been constructed from a template some of its attributes may be changed dynamically; others are permanently fixed throughout the lifetime of the object.

This chapter and the next explain how to use attributes and attributes objects to tailor threads and synchronization variables to your needs. The present chapter deals with the attributes mechanism itself and the basic attributes of each of the three major pthreads types. The next is devoted to scheduling, and the various scheduling-related attributes are covered there.⇒157

POSIX—because it is intended to cater to the needs of a wide range of present and future computing environments—has attempted to define a thread system that is both flexible and open-ended. Nonetheless, it is obviously desirable that simple needs should result in simple programs and—even more important—that hardware-independent ideas should be capturable in portable programs.

To meet these somewhat conflicting goals, the operation that constructs (or initializes) a new member of each the three main pthreads types has been given a single, *opaque* **attributes object** type that is itself created, manipulated and destroyed purely through a function interface. The standard can thus define the basic parameters that apply to an object, without over-constraining the implementation or interfering with the addition of new attributes for use by specific systems or later revisions of the standard.

At the moment, the standard provides ways to specify a thread's scheduling priority and contention scope (both described in the next chapter⇒157), stack allocation and detach state; and synchronization objects' *inter-process* shareability. But any number of extensions can be imagined, for debugging, for tuning, for compatibility modes, or to enable extensions to the basic semantics, for example.

To keep things simple, the system will use suitable defaults if you pass `NULL` as the value of the attributes argument, as we have been doing in our examples to this point. These defaults should be 'sane', in the sense that they will provide the expected basic semantics on any machine and in most circumstances.

As we shall see, the main disadvantages of this system are that it is verbose and (due essentially to limitations of the C preprocessor) incompatible with static initialization. Still, the scheme is sufficiently clean and practical that it is a pity that it was not invented sooner and used more widely in POSIX. Its advantages—beyond meeting the portability and generality goals outlined above—include providing a simple way to find out (in most cases) what the default attribute settings *are,* and the provision of an explicit object that encapsulates a single 'kind' of thread, mutex or condition variable so that many similar instances of a type can be manufactured simply.

8.2 The attribute protocol

Each of the three major types in the POSIX threads system has its own type for storing attributes objects:

object type	**attributes type**
`pthread_t`$_{32}$	`pthread_attr_t`
`pthread_mutex_t`$_{68}$	`pthread_mutexattr_t`
`pthread_cond_t`$_{80}$	`pthread_condattr_t`

The attributes are **opaque**: they can be neither assigned nor compared. (Actually your compiler will probably let you assign them, but don't do it! The standard contains a couple of paragraphs of interesting ideas for optimizations that could go seriously awry if you ignored this restriction. In any case, since an implementation has free choice over whether an attributes type is a pointer or an actual value, you can never guess what the real effect of such an assignment will be.)

As we've just noted, the fact that POSIX has made attribute specifications true objects (rather than relying on additional parameters to creation operations as many previous threads packages have done) is both a great strength and a great weakness: a strength because it enhances abstraction and portability, and a weakness because these objects must be explicitly created and destroyed by your program. Unlike synchronization objects, there isn't even a method provided for static initialization (but that's not usually a problem given that `NULL` has the same effect as a default attributes object). Instead, each object must be created and initialized by calling an `_init()`$_{140}$ routine, and may be destroyed with a `_destroy()`$_{141}$ function. Each of the attributes types has its own init/destroy pair, with names formed on the pattern of **`pthread_attr_init()`** and **`pthread_mutexattr_destroy()`**.

In each case the initialization function takes a pointer to a variable of the attributes type and initializes it to the implementation's default attributes for the corresponding kind of object; the exact settings used are determined partly by the standard and

attribute function	**values or type**
pthread_attr_t$_{138}$	
pthread_attr_getdetachstate()$_{142}$ pthread_attr_setdetachstate()$_{142}$	PTHREAD_CREATE_DETACHED PTHREAD_CREATE_JOINABLE
pthread_attr_getinheritsched()$_{161}$ pthread_attr_setinheritsched()$_{161}$	PTHREAD_EXPLICIT_SCHED PTHREAD_INHERIT_SCHED
pthread_attr_getschedparam()$_{168}$ pthread_attr_setschedparam()$_{168}$	struct schedparam
pthread_attr_getschedpolicy()$_{165}$ pthread_attr_setschedpolicy()$_{165}$	SCHED_FIFO SCHED_RR SCHED_OTHER
pthread_attr_getscope()$_{163}$ pthread_attr_setscope()$_{163}$	PTHREAD_SCOPE_PROCESS PTHREAD_SCOPE_SYSTEM
pthread_attr_getstackaddr()$_{146}$ pthread_attr_setstackaddr()$_{146}$	void *
pthread_attr_getstacksize()$_{145}$ pthread_attr_setstacksize()$_{145}$	size_t
pthread_mutexattr_t$_{138}$	
pthread_mutexattr_getprioceiling()$_{176}$ pthread_mutexattr_setprioceiling()$_{176}$	int /*thread priority*/
pthread_mutexattr_getprotocol()$_{174}$ pthread_mutexattr_setprotocol()$_{174}$	PTHREAD_PRIO_NONE PTHREAD_PRIO_INHERIT PTHREAD_PRIO_PROTECT
pthread_mutexattr_getpshared()$_{148}$ pthread_mutexattr_setpshared()$_{148}$	PTHREAD_PROCESS_PRIVATE PTHREAD_PROCESS_SHARED
pthread_condattr_t$_{138}$	
pthread_condattr_getpshared()$_{148}$ pthread_condattr_setpshared()$_{148}$	PTHREAD_PROCESS_PRIVATE PTHREAD_PROCESS_SHARED

Figure 8.1: Standard attribute functions

partly by the implementation itself. Conversely, the destruction function takes a pointer to an attributes variable and releases any resources it consumes. All these functions return zero if all goes according to plan, and an error number otherwise.

As always, you may not use or destroy an attributes object that is uninitialized or initialize one that *is* initialized; and destroying it makes it become uninitialized.

Of course, the default value of an attributes object is very rarely what you need (you could always use `NULL` to get those settings, remember). Generally you will want to change, and sometimes examine, the various individual attributes recorded in them. This is done with a large collection of `set` and `get` functions, one for each attribute of each attribute type; for instance, the initial detach state of a thread is controlled using the functions **`pthread_attr_getdetachstate()`**$_{142}$ and **`pthread_attr_setdetachstate()`**$_{142}$. The various standard attribute functions, listed in figure 8.1, are described in later sections of this and the next chapter; your implementation may provide others (which should be described in detail in your documentation).

In general, the various `set` functions take an attributes object and a new value to store in that object for the attribute they control, while the `get` functions receive the attributes and a pointer to a location in which they store the current value. They both return error numbers. These error numbers may be important to portable software, since implementations are not required to implement all of the various attributes or all of the possible values of each attribute. Returns of `ENOSYS`$_{271}$ or `EINVAL`$_{271}$ report on these conditions.

One final, important thing to remember is that an attributes object is like a cookie-cutter and not like a radio transmitter! Changes (including destruction) that you make to an attributes object after it has been used do *not* affect objects that were built using it. The attribute information is used to build a new thread, mutex or condition variable, and then the association ends. In some cases (such as detach state$\Rightarrow^{4.6}_{41}$ or priority$\Rightarrow^{9.3.6}_{170},{}^{9.5.4}_{178}$) there are ways of changing the new object's settings dynamically, but attributes objects are no longer involved.

8.2.1 Initializing attributes objects

Before an attributes object can be used, it must be initialized. This cannot be done statically; you must pass the object's address to an initialization function. There is a separate type of attributes object for each type of pthreads object, and there is a separate initialization function for each type of attributes object. In each case, when the function returns successfully, the new object is set up to contain the implementation's default attributes for the corresponding kinds of objects. You may now proceed to modify those attributes of which you want to take control using the functions listed in figure 8.1.$\Rightarrow^{8.2}_{139}$

`pthread_{attr,mutexattr,condattr}_init()` **16.1.1,11.3.1,11.4.1**

```
#include <pthread.h>

int /*errno*/ pthread_attr_init
  (pthread_attr_t *new_thread_attributes);

int /*errno*/ pthread_mutexattr_init
  (pthread_mutexattr_t *new_mutex_attributes);

int /*errno*/ pthread_condattr_init
  (pthread_condattr_t *new_condition_attributes);
```

`pthread_attr_t`$_{138}$ *`*new_thread_attributes`*
: Pointer to thread attributes object to be initialized.

`pthread_mutexattr_t`$_{138}$ *`*new_mutex_attributes`*
: Pointer to mutex attributes object to be initialized.

`pthread_condattr_t`$_{138}$ *`*new_condition_attributes`*
: Pointer to condition variable attributes object to be initialized.

After calling one of these functions successfully, the attributes object passed as an argument is initialized and can be modified by resetting one or more of its attributes, used in the creation of a thread, mutex or condition variable (as appropriate), or de-initialized with the corresponding `_destroy()`$_{141}$ function. It is an error to initialize an object that is already initialized.

Note that if the new attributes object is dynamically allocated (in `auto` storage or by `malloc()`, *e.g.*), it *must* be destroyed before it is discarded, or a resource leak (or worse) may occur.

These functions each return an integer containing an error number.⇒A 271

On success they return zero and the new attributes argument is initialized with the default attributes for the corresponding type.

On failure they return:

`ENOMEM` 271 Insufficient memory was available to build the new attributes object.

8.2.2 Destroying attributes objects

Before a dynamically allocated attributes object is freed, or before any attributes object is re-initialized, it must be de-initialized so that the resources it consumes are released. This is done by calling the appropriate `_destroy()` function on it. When the function returns, the attributes object is no longer useful.

pthread_{attr,mutexattr,condattr}_destroy() **16.1.1,11.3.1,11.4.1**

```
#include <pthread.h>

int /*errno*/ pthread_attr_destroy
  (pthread_attr_t *old_thread_attributes);

int /*errno*/ pthread_mutexattr_destroy
  (pthread_mutexattr_t *old_mutex_attributes);

int /*errno*/ pthread_condattr_destroy
  (pthread_condattr_t *old_condition_attributes);
```

`pthread_attr_t` 138 `*old_thread_attributes`
: Pointer to a location holding a thread attributes object to be deallocated.

`pthread_mutexattr_t` 138 `*old_mutex_attributes`
: Pointer to a location holding a mutex attributes object to be deallocated.

`pthread_condattr_t` 138 `*old_condition_attributes`
: Pointer to a location holding a condition variable attributes object to be deallocated.

Each of these functions releases the resources consumed by its argument and returns it to the 'uninitialized' state so that it can be discarded or initialized anew. The implementation may (or may not) change the value physically stored in the variable whose address is passed when it does this, depending on whether it is optimized for safety or for speed. In either case, the variable becomes unusable for any purpose other than re-initialization.

They each return an integer error code$\Rightarrow^{A}_{271}$ that is zero on success; no particular failure codes are defined for these functions since the only circumstance under which they should fail is when they are passed an incorrect argument, and this cannot, in general, be successfully detected; indeed, your program may just crash.

8.3 Static thread attributes

Aside from the scheduling-related parameters discussed in the next chapter, POSIX presently specifies only three attributes that apply to thread creation; they control a thread's initial detach state and the way in which its stack is allocated. Many implementations will define additional thread attributes in order to provide access to vendor- or hardware- specific functionality.

8.3.1 Detach state

By default, threads are created *joinable*$_{32}$, which is to say that they supply a final value and/or some final state to be retrieved using the synchronizing function `pthread_join()`$_{40}$, and their resources are not fully freed until this has happened. A thread that wishes to become *detached*$_{32}$—one that wishes to die silently and unmourned, leaving its resources immediately freeable—can always call `pthread_detach()`$_{41}$. The function `pthread_attr_setdetachstate()`$_{142}$, however, allows you to build an attributes object that makes threads that are detached from birth. This may be stylistically preferable to having a thread detach itself, and is more reliable in the case that the new thread may somehow die unexpectedly before it can invoke `pthread_detach()`.[1]

`pthread_attr_{set,get}detachstate()` **16.1.1**

```
#include <pthread.h>

int /*errno*/ pthread_attr_setdetachstate
  ( pthread_attr_t *attributes
  , int /*detachstate*/ new_detach
  );

int /*errno*/ pthread_attr_getdetachstate
  ( const pthread_attr_t *attributes
  , int /*detachstate*/ *old_detach
  );
```

`pthread_attr_t`$_{138}$ `*attributes`

Pointer to initialized$\Rightarrow^{8.2.1}_{140}$ thread attributes object to be examined or changed.

[1] For instance, if a higher-priority thread (perhaps acting on behalf of an interactive interface, *e.g.*) **cancels**$\Rightarrow^{11}_{203}$ the new thread during its initialization sequence.

`int /*detachstate*/ new_detach`
: New detach state for objects created with `attributes`, chosen from the constants listed below.

`int /*detachstate*/ *old_detach`
: Area to receive current detach state specified by `attributes`.

The possible values for the **detach state** attribute are the integers:

`PTHREAD_CREATE_DETACHED` The newly created thread is *detached*$_{32}$—it is deallocated automatically on exit and does not implicitly synchronize with any other thread.

> **Warning:** If a thread has a stack explicitly allocated with `pthread_attr_setstackaddr()`$_{146}$, and if that stack is to be reused or freed, then the thread *must* be joinable or there is no way to synchronize with it for that purpose on termination. (See the next section.)

`PTHREAD_CREATE_JOINABLE` *(Default)* The newly created thread is *joinable*$_{32}$ and will wait (if necessary) to synchronize with one other thread before it completes its termination sequence.$\Rightarrow^{4.5}_{40}$

Returns an integer error number.$\Rightarrow^{A}_{271}$

On success returns zero; the detach state has been successfully read or written.

On failure returns an error code:

`EINVAL`$_{271}$ *(*`pthread_attr_setdetachstate()` *only)* the argument *`new_detach`* was invalid.

8.3.2 Stack allocation

In order to implement multiple threads of control, it is generally necessary to provide each thread with its own stack on which to store `auto` variables, arguments, results, return addresses, and (if implemented) `alloca()`d storage; some processors put other information there as well (during interrupt handling, *e.g.*). In fact, once one has abstracted away from the details of the hardware architecture, it is the stack (in conjunction with the instruction pointer) that provides a thread with its distinct identity.

The physical details of how a thread's stack works are highly hardware-dependent. The amount of space required to run a given function, the addresses where the stack can legally reside, the presence or absence of a mechanism to transparently grow the stack and the presence or absence of a mechanism to trap a thread if it should suffer stack overrun all vary from configuration to configuration. For this reason, stack allocation is tricky, and by default completely automatic.

In the early days of computing, automatic stack allocation was problematic, since the amount of stack space that is needed, particularly by a recursive program, is quite unpredictable by the operating system (and is sometimes a surprise to the programmer, too). Allocating too much memory for the stack wasted then-valuable memory space; allocating too little would cause the program to crash. This problem was basically solved for the single-thread process by the advent of 32-bit addresses and **virtual memory**, which allowed (through various kinds of smoke, mirrors and trap-handling) the operating system to allocate a lot of *space* for the stack without having to allocate any *storage* until it was actually used: the system could allow for many megabytes of *virtual* stack in each process without overextending itself in the least.

Unfortunately, the use of multiple threads has re-introduced the old problem in a new form: since (under pthreads and many other systems) threads are allowed to exchange pointers to their stack-allocated storage freely, all stacks must effectively be allocated from the same address space. And since stacks must typically be contiguous and can't be moved once they are in use, they have to be allocated fixed chunks of that space.

Once upon a time, the 32-bit address spaces that have now become standard appeared immense, but designs of both hardware and software have become increasingly memory-hungry. Sixty-four kilobytes, which was once considered to be a reasonable architectural limit on stack size, is now quite easily transgressed by software that makes heavy—but not abnormal—use of the stack; some RISCs, in particular, are very stack-hungry. And while applications involving tens of thousands of threads are not the norm, they should not be impossible. Simple arithmetic suggests that address space is in short supply, the more so since the thread system must compete with other address-space-intensive practices, such as memory-mapping of files.[2] Fortunately not every thread has the same memory requirements.

POSIX, then, provides the following somewhat uneasy compromise: stack allocation is, as we've noted, completely automatic by default. A method is specified, however, for the programmer to tell the system how much stack space a given thread is going to need; or even to go one step further and allocate the needed memory manually. Implementations are not *required* to provide these options since they may not even be possible on every machine, so you rely on them being available to portable software. Worse yet, no general advice can be given on how much stack space a program really *will* need! Coming up with a suitable stack size requires knowledge of not only the program's worst-case behavior, but of the intimate details of both the compiler and the hardware architecture. It's all a bit like being handed a gun and encouraged to do something 'creative':-).

In any event, there are two (optional) standard attributes that control stack allocation: stack size and stack address—the latter implying that the caller has pre-allocated the necessary storage.[3] As attributes they are clearly anomalous, because the former

[2]Indeed, research has shown that it is already feasible to completely exhaust a 64-bit address space with suitably aggressive software engineering techniques!

[3]The reader completely comfortable with the idea of virtual memory will observe that there's a slight confusion going on in POSIX between the allocation of *address space*—which is the motivation for this call—and the use of *memory*—which is the form in which it manifests. Higher performance could probably be achieved if the former could be performed without the latter, since stack memory has distinctive properties that the virtual memory system can exploit (most notably in that storage beyond the pointer does not need to be preserved, and in terms of the access patterns induced by stack frame discipline). But POSIX lacks the conceptual vocabulary in which to say this.

(as just explained) is extremely dangerous and non-portable, while the latter is the only attribute that is non-re-usable—assigning the same stack address to two different threads would be catastrophic, and the stack address attribute (if used at all) must be reset each time a new thread is created.

Finally note that, unlike detach state, there is no practical way to change how a thread's stack is allocated once the thread is running; by the time the thread starts, its stack is already in use. Thus the attribute mechanism is the only possible way of providing the desired control.[4]

Stack size Since some implementations will be unable to vary thread stack size and on others there may be no advantage in doing so, implementations declare their willingness to honor the stack size attribute by defining the symbol `_POSIX_THREAD_ATTR_STACKSIZE`. Each implementation will also have an absolute minimum stack size that a thread can use; this is provided to the programmer as the value of the symbol `PTHREAD_STACK_MIN`.

`pthread_attr_{set,get}stacksize()` **16.1.1⋆**

```
#include <pthread.h>
#include <unistd.h>              /* For _POSIX_THREAD_ATTR_STACKSIZE */
#ifdef _POSIX_THREAD_ATTR_STACKSIZE

int /*errno*/ pthread_attr_setstacksize
  (pthread_attr_t *attributes, size_t new_stacksize);

int /*errno*/ pthread_attr_getstacksize
  (const pthread_attr_t *attributes, size_t *old_stacksize);
```

`pthread_attr_t`$_{138}$ `*`*`attributes`*

Pointer to initialized$\Rightarrow^{8.2.1}_{140}$ thread attributes object to be examined or changed.

`size_t` *`new_stacksize`*

Desired stack size for threads created using *`attributes`*.

`size_t *`*`old_stacksize`*

Pointer to variable to receive the stack size currently specified by *`attributes`*.

The value of the stack size attribute of threads created with *`attributes`* is read or updated.

The actual size of the stack storage allocated to a new thread may be somewhat larger than requested, but will not be smaller. Excessively large values of *`new_stacksize`* will result in either `pthread_attr_setstacksize()` failing with `EINVAL`$_{271}$ or `pthread_create()`$_{33}$ failing with `ENOMEM`$_{271}$.

These functions return an integer error number.$\Rightarrow^{A}_{271}$

[4] Actually, we'll see in the appendices to this chapter that there *is* a theoretical alternative: some systems can create threads in a 'suspended' state in which they do no processing. This provides an opportunity to modify them after creation but before they go to work. The real difference between POSIX attributes objects and such 'uninflated' objects is that attributes objects are *copied* each time they are used, while uninflated objects must be redescribed and rebuilt at each instantiation.

On success these functions return zero and the attributes object or *`*old_stacksize`* location has been updated.

On failure one of the following is returned:

`ENOSYS`$_{271}$ This implementation does not provide control of stack size; `_POSIX_THREAD_ATTR_STACKSIZE` was not defined.

`EINVAL`$_{271}$ *(*`pthread_attr_setstacksize()` *only)* The value of *`new_stacksize`* is either too large or less than `PTHREAD_STACK_MIN`$_{145}$.

As we have already noted, this attribute is very difficult to use properly, since on the one hand the amount of stack space required by a thread is system-dependent and typically very hard to determine, and on the other hand stack overruns are catastrophic.

At the same time, some applications may have recourse to these functions, either because they use a very large number of simple threads animating non-recursive functions (which can use quite small stacks), or because they have some threads with exceptionally heavy stack use that succeed in overrunning the default stacks provided implicitly by the implementation.

Stack address Some implementations of pthreads further allow the application to specify at creation time the *address* of a thread's stack. If so, the symbol `_POSIX_THREAD_ATTR_STACKADDR` will be defined. The size of the programmer-supplied stack must, as always, be at least `PTHREAD_STACK_MIN`$_{145}$.

`pthread_attr_{set,get}stackaddr()` **16.1.1⋆**

```
#include <pthread.h>
#include <unistd.h>              /* For _POSIX_THREAD_ATTR_STACKADDR */
#ifdef _POSIX_THREAD_ATTR_STACKADDR

int /*errno*/ pthread_attr_setstackaddr
  (pthread_attr_t *attributes, void *new_stackaddr);

int /*errno*/ pthread_attr_getstackaddr
  (const pthread_attr_t *attributes, void **old_stackaddr);
```

`pthread_attr_t`$_{138}$ *`*attributes`*
Pointer to initialized$\Rightarrow^{8.2.1}_{140}$ thread attributes object to be examined or changed.

`void` *`*new_stackaddr`*
Address of storage of at least `PTHREAD_STACK_MIN`$_{145}$ bytes to be used for the stack of threads created with *`attributes`*.

`void` *`**old_stackaddr`*
Pointer to location receiving the currently specified stack address attribute.

Obtain or change the value of the stack address entry in ***attributes***, the value to be used as the address of any thread built with this attributes object. The standard apparently does not specify what result to expect from reading this field if it has never been set, nor yet does it specify a value you can assign to it to return stack allocation to its default behavior of automatically allocating a new stack for each thread.

Note that it is catastrophic for two threads to attempt to use the same stack area at the same time. Thus stack addresses can only be freed or re-used after you are *certain* that the previous user has terminated, which in turn is only possible if that thread was *joinable*$_{32}$ and has in fact been joined with `pthread_join()`$_{40}$.

No validation of the ***new_stackaddr*** argument is required of the implementation; if it is ill-formed this will result in failure of thread creation or undefined program behavior thereafter.

These functions return an integer error number:$\Rightarrow^{A}_{271}$

On success they return zero.

On failure they return an error code:

> `ENOSYS`$_{271}$ This implementation does not provide control of stack address; the symbol `_POSIX_THREAD_ATTR_STACKADDR` was not defined.

Note that using this option on a detached thread unavoidably results in either a resource leak or a race condition: there will be no way to determine when the stack storage again becomes free.

Using this option shifts the responsibility for allocating stack storage from the threads package to the application itself. The storage pointed to by `new_stackaddr` must be valid from the moment `pthread_create()`$_{33}$ is invoked until the new thread has finally (and fully) terminated. Note in particular that *unless the thread is joinable*$_{32}$ *there is no way to determine that the stack storage is free*—any synchronizing action that a *detached*$_{32}$ thread takes to report that it is terminating must *ipso facto* take place while it is still executing, and thus while its stack is still actively being used.

In particular, it is vital that two threads not end up using the same stack! Thus, unless the `attributes` object is used only to build threads whose lifetimes do not overlap (as seen by `pthread_create()`$_{33}$ and `pthread_join()`$_{40}$), the specified stack address must be *changed* to point to new storage each time it is used. Combined with the fact that the standard provides no way to reset the stack address attribute to its default (automatic) behavior, this implies that use of `pthread_attr_setstackaddr()` converts the attributes object into something less convenient than an arbitrarily re-usable 'cookie-cutter' for threads; it now typically requires that an appropriate stack address be set in the attributes object *each* time it is used.

8.4 Synchronization variables: shareability

All synchronization variables are of course shared between threads. The only standard attribute they have that is not related to thread scheduling is common to both mutexes and condition variables: it controls their shareability between *processes.* [Scheduling-related attributes are discussed in the next chapter.⇒9.5 172]

Unlike threads which—under the POSIX model—are permanently bound to particular processes, synchronization variables may meaningfully appear in shared memory (if provided). This is only reasonable: when a data structure is shared between processes, it is essential that the structures synchronizing access to them be shared also. Otherwise, all of the difficulties arising from unsynchronized access between threads of the same process—problems that are solved by mutexes and condition variables—would reappear in the multi-process/shared memory case.

Ensuring that mutexes and condition variables work properly in shared memory may, however, be fairly expensive; and since for many purposes, especially ones where performance is important, multiple threads provide a better solution than multiple processes, process shareability has been made an *option* in pthreads. That is: by default, synchronization variables are local to a process and they cannot be used by the threads of another process even if shared memory makes their addresses visible. If, on the other hand, a mutex or condition variable is initialized with attributes that specify that process-shareability is required, then they can legitimately and successfully be used by threads of any process which can directly address them.

Implementations are not *required* to support this facility even if they provide both threads and shared memory; if they do they will so signify by defining the symbol `_POSIX_THREAD_PROCESS_SHARED`.

`pthread_{mutex,cond}attr_{set,get}pshared()` **11.3.1,11.4.1⋆**

```
#include <pthread.h>
#include <unistd.h>             /* For _POSIX_THREAD_PROCESS_SHARED */
#ifdef _POSIX_THREAD_PROCESS_SHARED

int /*errno*/ pthread_mutexattr_setpshared
  ( pthread_mutexattr_t *attributes
  , int /*shareability*/ new_shareability
  );

int /*errno*/ pthread_mutexattr_getpshared
  ( const pthread_mutexattr_t *attributes
  , int /*shareability*/ *old_shareability
  );

int /*errno*/ pthread_condattr_setpshared
  ( pthread_condattr_t *attributes
  , int /*shareability*/ new_shareability
  );

int /*errno*/ pthread_condattr_getpshared
  ( const pthread_condattr_t *attributes
  , int /*shareability*/ *old_shareability
  );
```

`pthread_mutexattr_t`$_{138}$ `*attributes`
: Pointer to initialized$\Rightarrow^{\S 8.2.1}_{140}$ mutex attributes object to be examined or changed.

`pthread_condattr_t`$_{138}$ `*attributes`
: Pointer to initialized$\Rightarrow^{\S 8.2.1}_{140}$ condition variable attributes object to be examined or changed.

`int /*shareability*/ new_shareability`
: New value for the shareability attribute, chosen from the constants below.

`int /*shareability*/ *old_shareability`
: Pointer to location receiving the old value of the shareability attribute.

The standard values for process shareability are:

`PTHREAD_PROCESS_PRIVATE` *(Default)* Objects created using ***attributes*** are private to the creating process.

`PTHREAD_PROCESS_SHARED` Objects created using ***attributes*** are available for use by any process that can address them.

These functions return an integer error code:$\Rightarrow^{A}_{271}$

On success they return zero.

On failure they return the code:

`EINVAL`$_{271}$ Either ***attr*** or ***new_shareability*** was invalid.

These functions are only supported in the case that `_POSIX_THREAD_PROCESS_SHARED`$_{148}$ is defined; thus `ENOSYS`$_{271}$ is never returned (unlike some other attribute functions that are defined even when the corresponding attributes are not implemented).

The way the standard is written, it would appear that there is no requirement that both a mutex and a condition variable be shared in order for them to be used together; a shared mutex might be associated with a process-local condition variable providing that only threads of the given process wait on or signal the condition. Similarly, a process-shared condition variable might usefully be associated with a process-local mutex by making use of the ability to signal a condition without holding the associated mutex. (It would not be surprising, however, if an implementation failed to get these hybrid possibilities right.)[[**sps: ??**]]

8.5 Example

8.6 Summary

[[. . .]]

In the next chapter we look (among other things) at the remaining attributes of threads and mutexes, which relate to scheduling.

Appendices

8.a Common implementation-defined attributes

Extensions to the set of attributes required by the POSIX standard are likely to be common. One obvious candidate is the provision of additional semantic classes for mutexes. The mutexes required by POSIX are designed for high performance, with safety an option: in particular, double-locking a mutex is allowed to deadlock.$\Rightarrow^{5.2}_{64}$

Often, the performance gains from allowing this behavior justify the inconvenience, but at other times (especially when debugging) it would be pleasant to have a guarantee that such an auto-deadlock will be detected and reported as an error—indeed, it is convenient if other, indirect sources of deadlock are reported, too. On the other hand, it is also possible (and often helpful) to assign a completely different semantics to multiple locking of a single mutex: the **counting** semantics. Under this regime, a single thread can lock a mutex as many times as it wants; count is kept, and the mutex is not ultimately released until it has been unlocked as many times as it was locked.[5]

Such variation is readily supported through the addition of an attribute that selects the mutex class desired. As we shall see below, both DCE threads and the CAPthreads POSIX implementation provide just this behavior.

Further likely candidates for implementation-defined attributes include a number of matters of policy lying right on the edge of what might normally be considered "semantics". These include how threads are distributed among physical processors in a multiprocessor, what (and whether) guarantees are made about wakeup *order* for mutexes and condition variables (obvious options being random, FIFO and arbitrary), and tuning information (such as the waiting mechanism for a mutex, the best choice for which, as mentioned earlier, might depend on the average length of time a mutex is held and the number of threads expected to contend for it$\Rightarrow^{5.b.6}_{106}$).

8.b Implementation considerations

Attributes objects are conventional data structures that are not performance critical; as such, there is little interesting to say directly about their implementation. Two things stand out as worthy of note, however.

8.b.1 Upward binary compatibility

In the text of the pthreads standard there is a note to the effect that if not only source but *binary* compatibility is required between different releases of a threads package, it is important to put a version number—or at least a length field—in each (initialized) attributes object. Otherwise, if you link software compiled for an early version of the threads package against a later release of the pthreads library that provides additional attributes (and thus expects a larger attributes structure), data may be read or even written out of bounds. For this version-stamping technique to work, however, either the attributes structure needs to be indirected (so that its storage is actually allocated by the `_init()` routine—but in this case the precaution is hardly necessary), or the initialization routines must themselves be *macros* rather than true functions—the code that sets the length field must be compiled into the client module!

[5] Counting mutexes are used for `FILE` object locking in the pthreads standard library$\Rightarrow^{12.3}_{223}$ and are standard in some other systems, such as OS/2.$\Rightarrow^{5.d.4}_{110}$

8.b.2 Performance of multi-class mutexes

In implementations where more than one class of mutex exists, it is a real question how to dispatch between them without seriously impacting performance on the 'fast path' of locking an uncontested basic mutex, given that all mutex classes share a *single* type and the *same* operational interface. Obviously it is appropriate to dispatch through a function pointer; this introduces only one or two instructions of delay into an inline-compiled lock operation. But those two functions (a load and an indirect call) are among the slowest on modern hardware, and may even dominate the fast path logic itself (typically an atomic lock instruction and a correctly predicted branch).[6]

The solution to this problem relies on the observation that the 'fast path' for a mutex is the case where it is not in contention; if a mutex is already locked at the time it is reached, a certain amount of additional overhead (whether spinning, queuing, or some other waiting technique) is unavoidable. Is it possible to put all of the overhead onto the slow path?

Almost—but not quite. When *locking* a mutex, it is possible to arrange for all non-default mutexes to have (what in fast mutexes is) the **lock** field *set* from the moment they are created. This will immediately switch processing to the slow track (where fast mutexes wait anyway) witotu changing a single instruction of the sequence that locks an unowned fast mutex. Unfortunately, efficient *un*locking of non-default mutexes is trickier. Somehow we must switch unlocking processing between fast and slow paths, and we must make sure that the "fast" lock of a non-default mutex never gets unlocked at all. To the best of our knowledge, this cannot be achieved without an overhead of at least one instruction, typically rather more.

8.c Capthreads extensions

In this section we describe a number of nonstandard attributes for mutexes and threads that the CAPthreads implementation supports. This serves directly as documentation for the CAPthreads implementation and indirectly as an example of how vendor-specific extensions are integrated into the standard pthreads framework.

---

[6]Here's the deal. When processing a conditional jump, a typical modern processor 'guesses' whether the branch will be taken, and continues to fetch instructions along the predicted path; to do this, the branch instruction itself doesn't even have to have been executed yet. When the branch is finally evaluated—and its condition is checked—either the guess was correct, in which case the next instruction is already fetched and ready to go, or it wasn't, and the extra pre-fetched instructions need to be discarded and the alternate instruction stream fetched instead (there are several truly amusing ways to speed this up, but that's beside the present point). Whatever algorithm the hardware uses to arrive at its prediction of how the branch will go, the compiler will know in advance, and can arrange that the 'guess' will be right in the (presumably usual) uncontended case (which of course is the only case that *can* be made fast). The conditional branch will now execute as fast as any other instruction.

The indirect call is another matter, however. Although this call is *always* 'taken'—it's not conditional—its *target* is not (in general) known until the preceeding fetch completes; there's no opportunity to prefetch the target instructions. It's worse than that, of course; a call also needs to store the address it came from in order to be able to return subsequently—and when that return eventually does happen, it amounts to *another* indirect—and thus unpredicatable—branch (contemporary processors don't seem to take much advantage of the fact that return addresses are known well in advance; perhaps this is just because they have been optimized for numerical code [where conditional branches are very common] rather than functional or object-oriented software [with its high proportion of calls]).

8.c.1 Selecting a mutex type

Like many implementations of threads (both according to the pthreads standard and otherwise), CAPthreads provides several classes of mutexes, optimized for different purposes. In addition to basic, **fast** mutexes, these include '**recursive**' (or **counting**) mutexes—which can be repeatedly relocked by their owning thread, requiring as many unlocks as lock to release them; and **debugging** mutexes—which take extra care to detect possible error conditions, at the cost of performance. The programmer selects between them through an extension to the mutex creation attribute protocol.

---

pthread_mutexattr_{set,get}type_np() — **capthreads**

```
#include <pthread.h>

int /*errno*/ pthread_mutexattr_settype_np
  ( pthread_attr_t *attributes
  , int /*mutex type*/ new_type
  );

int /*errno*/ pthread_mutexattr_gettype_np
  ( pthread_attr_t *attributes
  , int /*mutex type*/ *old_type
  );
```

`pthread_attr_t`$_{138}$ `*attributes`
: Pointer to initialized⇒$^{8.2.1}_{140}$ attributes object to be examined or changed.

`int /*mutex type*/ new_type`
: New type for mutexes created with `attributes`, chosen from the constants below.

`int /*mutex type*/ *old_type`
: Current value of the mutex type attribute.

The supported mutex types are:

`PTHREAD_MUTEXTYPE_FAST_NP` *(Default)* Optimize mutex configuration for speed, otherwise meeting only the minimum requirements of the standard.

`PTHREAD_MUTEXTYPE_RECURSIVE_NP` Modify mutex behavior so that multiple locking by the same thread does not result in deadlock; a mutex will not be released until it has been unlocked as many times as it was locked.

`PTHREAD_MUTEXTYPE_DEBUG_NP` Perform additional checks when the mutex is manipulated, detecting as many deviations from correct, portable use of the mutex as possible.

These functions allow the programmer to select one of the variant mutex types provided by the CAPthreads implementation. Many implementations of pthreads can be expected to provide a similar extension.⇒$^{8.a}_{150}$

On success returns zero.

On failure returns an error code (no error returns are currently specified, however).

8.c.2 Freeing a thread's initial arguments

One minor annoyance with the design of `pthread_create()`$_{33}$ is that while a thread's initial argument must be allocated by the parent thread it must be freed by the child. It would be more convenient if the creating thread could specify a destructor for it instead; this destructor would still be run in the context of the new thread, but the thread's code would no longer need to contain any explicit dependencies on how the allocation was done. To this end CAPthreads provides the nonstandard call `pthread_attr_setcleanup_np()`. The function has been designed so that it can also be used for other kinds of termination processing.

`pthread_attr_setcleanup_np()` **capthreads**

```
#include <pthread.h>

int /*errno*/ pthread_attr_setcleanup_np
  ( pthread_attr_t *attributes
  , void (*function)(void *argument)
  , void *argument
  );
```

`pthread_attr_t`$_{138}$ `*attributes`
: Thread attributes object to be modified.

`void (*function)(void *argument)`
: Function to be called on exit from each thread created with *`attributes`*.

`void *argument`
: Argument to be passed to *`function`*.

The *`function`* and its *`argument`* are stored in the thread *`attributes`* object and transferred to the thread object when it is created. When the thread finally terminates the argument is passed to the function so that it can perform any necessary termination processing. Typically this might include deallocation of resources associated with the thread, including its initial argument.

On success returns zero.

On failure returns an error code (no error returns are currently specified, however).

8.c.3 Declaring a thread's use of floating point

Not all software makes use of floating point operations, but modern processors devote large amounts of hardware context to its support.[7] Saving and restoring this floating point context adds significantly to the cost of switching the processor between threads on some processors, and is unnecessary for threads that are floating-point-free. The CAPthreads system therefore allows the programmer to declare whether or not a thread wishes to use floating point facilities.

---

[7] Our suspicion is that this is because the hardware engineers who design processors make heavy use of floating point arithmetic themselves, for circuit simulation. Whether they naïvely assume that everyone else uses the same software or whether the general public is being deliberately exploited to drive down the cost of hardware engineers' workstations is a matter for conspiracy theorists to debate!

pthread_attr_{set,get}floatstate_np() **capthreads**

```
#include <pthread.h>

int /*errno*/ pthread_attr_setfloatstate_np
  ( pthread_attr_t *attributes
  , int /*float state*/ new_floatstate
  );

int /*errno*/ pthread_attr_getfloatstate_np
  ( pthread_attr_t *attributes
  , int /*float state*/ *old_floatstate
  );
```

`pthread_attr_t`138 `*attributes`
: Thread attributes object to be modified or examined.

`int /*float state*/ new_floatstate`
: Float state for threads created from the `attributes` object, chosen from the list below.

`int /*float state*/ *old_floatstate`
: Location receiving old float state from `attributes`.

`PTHREAD_FLOAT_NP` *(Default)* The thread may make use of floating point facilities, including `float` and `double` constants, variables, and operations upon them.

`PTHREAD_NOFLOAT_NP` The thread makes *no* use of the processor's floating point capacity.

The floatstate attribute allows the programmer to inform the implementation of threads that make no use of floating point values or operations, permitting certain optimizations in the thread-handling code to be enabled.

Warning: The consequences of using any floating point operations at all (including simple assignment) in a thread with floatstate `PTHREAD_NOFLOAT_NP` are likely to be extremely unpleasant, and include the arbitrary trashing of floating point computations in other threads.

On success returns zero.

On failure returns an error code (no error returns are currently specified, however).

Declaring a thread to be `PTHREAD_NOFLOAT_NP` is simply an optimization; it makes no positive change to the program semantics. The burden of ensuring that such threads do not use any floating point facilities—even in library code—rests with the programmer.

8.d Other systems

8.d.1 Attributes in DCE threads

The concept of attributes objects is basically present only in the POSIX world. DCE threads, sharing a common heritage with POSIX, provide them. Attributes in DCE threads are almost identical to those we have just described, aside from the fact that default behavior is requested with constants having names of the form `pthread_*attr_default`.

DCE is a more stringent standard than POSIX, with more detailed requirements, being intended for a narrow range of applications (essentially distributed computing using uniprocessors); it is therefore not surprising that it mandates attributes beyond those required by POSIX. Relevant to this chapter is that it (like the CAPthreads implementation of the POSIX standard) provides a **class** attribute for mutexes [[**sps: look up the actual identifiers for the accessors and the constants**]]. The three standard DCE mutex classes are **fast**—the default, with the same semantics as POSIX; **recursive**, with counting semantics; and **nonrecursive**, which disallows multiple locking but guarantees to report it as an error (this last may slightly reduce performance by requiring each mutex to record its current owner).

8.d.2 GATech Cthreads

GATech Cthreads lacks the notion of an attributes object as found in POSIX, but its threads and synchronization variables nonetheless have attributes: all major objects can be *named* to assist in producing useful debugging output when the variable `cthread_debug` is true [the naming is normally done implicitly by allocation macros which capture the textual form of the variables their result is stored is stored in], and threads also have both a field for thread-specific user data and one for scheduler-related data (though the standard scheduler doesn't make use of it).

These attributes are set and read directly from the objects they relate to; that is, they provide a dynamic interface but no specialized creation-time interface. The relevant functions are {`cthread,mutex,condition`}`_set_name()`, `cthread_set_`{`data,sched_info`}`()`, {`cthread,mutex,condition`}`_name()` and `cthread_`{`data,sched_info`}`()`.

There's a conceptual difficulty with this approach: since the scheduling information is set by direct manipulation of the Cthread object, there is no obvious way to set the *initial* value of the scheduler data. The solution to this problem is that GATech Cthreads splits the `cthread_fork()`$_{??}$ operation into two parts: an *allocation* phase in which the thread object is constructed, and a *scheduling* operation which makes it eligible to run.$\Rightarrow^{4.b.4}_{57}$ Between allocation and scheduling it is in a suspended state: it can be manipulated with the attribute functions just described, but it has not yet commenced execution. Thus, in effect, the initial conditions of the thread have been set without requiring an extra type of object to hold them. On the downside, this setup sequence must be repeated in its entirety for each thread that is created.

GATech Cthreads also provides some control of parameters such as stack size and the size of the shared memory arena, but since the system relies on each thread being provided with an identical memory map (its solution to the problem that it supports platforms on which most memory is node-specific) these are set on a per-*program* basis rather than per thread. Selection of these parameters is achieved by calling `cthread_configure()` *before* the threads package is initialized with `cthread_configure()`$_{155}$.

Chapter 9

Scheduling

Chris's Corn for Chapter 9:

"The world is...
...excuse me a moment...
interrupt driven."

[[Switch to ***initial priority/effective priority/base priority*** *terminology throughout.]]*

[[message queues—assuming we add some mention of them!—have their OWN priority scheme.]]

9.1 Introduction

The purpose of multithreading is to increase the computer's ability to do several things at once. But not everything is of equal importance; very often a program will have primary duties that it performs for a user (like controlling an airplane, editing text, or handling the CANCEL button) and various background tasks (such as updating the passengers' Dynamic Trivia Display, reformatting parts of a document that are currently off the screen, or spooling a print job). To reflect such differences, threads packages (like process scheduling systems) may provide a notion of thread **priority**.

Actually, since even very low priority threads will run if nothing else is going on (or if there are enough physical processors available), it might be better to think of priority as reflecting *urgency* rather than importance. User-feedback tasks, for example, have traditionally been seen as having low conceptual significance, but their timeliness makes the difference between systems that are confidence-inspiring and ones that are unusable. Pen-based input was an idea that languished for decades because experimental mainframe implementations were unable to draw "ink" to the display fast enough that users could write normally—resulting in extremely negative assessments of their usability. Similarly, threads accepting even very unimportant input may need to respond rapidly, lest that input be 'dropped' and lost forever. Thus response-critical and often-blocked threads may need a higher priority than threads that are purely compute-bound.

Computational tasks also differ in what *kind* of attention they require. In some applications, it is important that processing be done as efficiently as possible: it is the final completion time that matters, not the details of when each thread runs. Other tasks demand that threads receive moment-to-moment fair treatment: no thread should go too long without receiving a chance to run. These goals conflict, because switching between threads has a real expense, and different scheduling *policies* have been invented that have different overall effects on how often and for how long each thread runs. As we shall see below, POSIX in fact permits a scheduling policy to be selected separately for each thread—though of course the policies will always interact.

From the perspective of the system, of course, processor time is just another resource—like storage or communication bandwidth—to be allocated. Specifically, it is the resource that a thread *represents.* Part of its interest lies in the fact that for some tasks, time is effectively a *renewable* resource—there's always more of it, later. For other tasks—so-called 'real-time' tasks—it is *non*-renewable: a moment missed is gone forever.

9.2 Scheduling in posix

The POSIX threads standard places no specific requirements on the way threads are scheduled beyond specifying a standard interface for setting initial scheduling parameters, based on the **attributes** model described in the last chapter.⇒8.2 138 There is, however, an option (available if the implementation defines the symbol `_POSIX_THREAD_PRIORITY_SCHEDULING`) which provides a priority scheduling model derived from the POSIX real-time scheduling facilities;[1] those familiar with POSIX real-time scheduling will find the material in this chapter very familiar. This optional facility is the main topic of the present chapter.

If your implementation of pthreads does *not* provide the priority scheduling option, the only directly relevant material in this chapter is the functions `pthread_attr_setschedparam()`168 and `pthread_setschedparam()`170, and you will have to consult your own documentation to learn exactly how they are used in your implementation. We also recommend that you read your system's documentation carefully in the case that you have multiple physical processors: the current revision of pthreads has left the hard task of developing of a good portable interface to multiprocessor scheduling for future work.

9.2.1 The scheduling model

The POSIX thread priority scheduling model divides the scheduling-related settings for a thread into three attributes: the threads' **contention scope**, its **scheduling policy** and its **scheduling parameters**. There is actually a fourth concept defined, important for multiprocessors, called the **allocation domain**; but no attribute related

[1] POSIX 1003.1b, previously POSIX.4. One good reference for this is Bill O. Gallmeister's book *Programming for the Real World: POSIX.4*.

to it has yet been put into the standard. The standard scheduling attributes can all be specified at thread creation time through the attributes objects (of type `pthread_attr_t`[138]) introduced in the last chapter; most of them can also be changed dynamically while the thread is running. These attributes are described in more detail later; but briefly, they are as follows:

scope A thread's contention scope, standardly either the whole system or just the thread's owning process, determines the 'pond' in which it competes for processing resources; is its allocation taken from the overall system budget—relying on the system scheduler, almost as if the thread were a process unto itself—or from that already parceled out to its owning process? Since the two methods require rather different techniques,⇒9.a 179 an implementation may provide both kinds of scope but is not required to.

Unlike the other scheduling attributes, a thread's contention scope is permanent; there is no way to change it after the thread has started running.

policy A scheduling policy is an algorithm that determines when a thread should run, and when it should yield to another. Three policies are named by the standard ('**FIFO**', '**round-robin**' and 'other'), but the meanings and interactions of only the first two are specified [the third is intended to be the system's 'normal' scheduling algorithm for non-real-time computations, whatever it may be].

parameters Each policy defines its own set of scheduling parameters, though they are accessed through a common interface. The most commonly used parameter (and the only one required by the standard FIFO and round-robin policies) is a **priority** which, if you will, determines *how big a fish* the thread is in its pond: threads with higher priorities effectively get first pick of the processing time resources.

allocation domain The allocation domain of a thread is the set of *physical* processors on which it is eligible to run. For some multiprocessors this may be a very important parameter, but because there is so much variation in multiprocessing hardware and software, it has not yet been standardized.

The interaction between these concepts is a little subtle. First, the scheduling parameters attribute is the only one that is present in every implementation of pthreads; the other attributes are all part of the *priority scheduling option* (and so effectively receive implementation-defined default values otherwise). However, the *meaning* of the scheduling parameters is completely determined by the scheduling policy in effect for the thread—and the definitions of the policies determine how they interact.

How threads interact also depends on their contention scope: the policy and parameter settings only determine how threads with the *same* scope interact. That is, they specify which of the system-scope threads in the system will run, or which of the process-scope threads in a process are selected, but they have no direct effect on when process-scope threads run relative to system-scope threads, or even threads whose scopes are other processes.

Finally, when a thread runs may be further restricted by its allocation domain; presumably threads with the same scope but non-overlapping allocation domains do not directly pre-empt each other, even if all other factors suggest that they should.

9.3 Scheduling threads

The scheduling attributes that a thread carries are determined by combining information from at least four different sources (aside from any interactions there might be between the scheduling of a thread with process scope and the scheduling of the process that contains it, about which the standard has little to say other than that the policy and parameters of the *initial* thread of a process are set from those of the process itself—which may or may not be helpful). These are:

creation attributes As might be expected, the main method of setting up a thread to have particular scheduling behavior is through the attributes object that is passed to `pthread_create()`$_{33}$ when the thread is first constructed.⇒$^{8.2}_{138}$ There are attributes provided to determine the scope, policy and parameters of the new thread (and implementations are permitted to add others, to handle, for instance, allocation domains).

inheritance Each thread also possesses an overriding attribute, **scheduling inheritance**. If a thread *inherits,* the other scheduling attributes are in fact ignored, and the necessary information is taken from the creating thread instead.

dynamic settings Although it is not possible to modify the contention scope of an existing thread, it may be necessary to modify its policy or parameters from time to time, notably (a) when the whole program undergoes a mode change or (b) when a particular thread starts or finishes a time-critical task (though the latter case may also be handled through the next item:).

implicit adjustment While a thread is executing, its scheduling parameters may be further influenced by its circumstances. Policies other than FIFO and round-robin may adjust them heuristically, based on thread, process and system performance. Or, as we shall see towards the end of this chapter, the priority-scheduling policies sometimes revise a thread's **effective priority** upwards while it holds a mutex with certain attributes.⇒$^{9.5.1}_{173}$

Unlike the preceding factors, this does not impact a thread's **base priority**; the unadjusted value is remembered so that it can be restored when the modifying circumstance is removed. (One caveat: the standard does not specify what happens if a thread uses explicit dynamic adjustment while it is already under the influence of an implicit adjustment originating with a mutex. Things might get confused.)

In the following sections we look at the POSIX thread scheduling interface in detail.

9.3.1 Scheduling inheritance

When the first thread of a process is started by the system, its contention scope is set to the system's default and its scheduling policy and parameters are copied from those of the process itself. The scheduling data for other threads are set *either* by inheritance from the creating thread *or* according to the attributes that are used when the thread is created.

The **scheduling inheritance** thread creation attribute$\Rightarrow^{\S 8.2}_{138}$ determines which: its legal values are `PTHREAD_INHERIT_SCHED` and `PTHREAD_EXPLICIT_SCHED`. The standard does not specify whether inheritance or explicit scheduling is the default, and so a defensive program must set this attribute explicitly if it manipulates scheduling information at all.

Warning: Unless the explicit option is used, the other scheduling attributes (policy and parameters, and possibly scope—though the standard is unclear in this case) are *ignored*.

`pthread_attr_{set,get}inheritsched()` **13.5.1⋆**

```
#include <pthread.h>
#include <unistd.h>         /* For _POSIX_THREAD_PRIORITY_SCHEDULING */
#ifdef _POSIX_THREAD_PRIORITY_SCHEDULING

int /*errno*/ pthread_attr_setinheritsched
  ( pthread_attr_t *attributes
  , int /*inherit*/ new_inherit
  );

int /*errno*/ pthread_attr_getinheritsched
  ( pthread_attr_t *attributes
  , int /*inherit*/ *old_inherit
  );
```

`pthread_attr_t`$_{138}$ `*attributes`
: Pointer to initialized$\Rightarrow^{\S 8.2.1}_{140}$ thread attributes object to be examined or changed.

`int /*inherit*/ new_inherit`
: New value for the scheduling setup inheritance attribute, chosen from the constants below.

`int /*inherit*/ *old_inherit`
: Pointer to location receiving the old value of the scheduling setup inheritance attribute.

Legal inheritance values include:

`PTHREAD_EXPLICIT_SCHED` The 'scheduling attributes', including the **scheduling policy** and **scheduling parameters**, of a thread newly created with these *`attributes`*, are determined by the relevant fields of the thread attributes object (but see the warning below).

`PTHREAD_INHERIT_SCHED` The 'scheduling attributes' of a new thread are to be 'inherited' (once again, see the warning) from the creating thread. *The other scheduling attributes in the attributes object are then ignored by* `pthread_create()`$_{33}$.

On success these functions return `0`.

On failure they return one of the following status codes:

`ENOSYS`$_{271}$ This implementation does not support priority scheduling.

`EINVAL`$_{271}$ *(*`pthread_attr_setinheritsched()` *only)* the value of *`new_inherit`* is invalid.

`ENOTSUP`$_{271}$ *(*`pthread_attr_setinheritsched()` *only)* the value of *`new_inherit`* is unsupported. (Though documented, it is not clear whether this error can ever occur.)

Warning: The text of the standard is unclear on several points regarding this attribute:

1. it determines the heritability of scheduling policy and any scheduling parameters, but does it allow for inheritance of scheduling **contention scope**?
2. when attributes are inherited, are they *copied* from the creating thread to the created thread, or do they share the *same* attributes (that is, does a `pthread_setschedparam()` applied to one thread affect others that have 'inherited' its attributes)?
3. again, are the attributes inherited the *initial,* creation-time, attributes of the creating thread, its *base* attributes (as seen by `pthread_getschedparam()`), or its effective ones (as modified, *e.g.*, by any mutexes that it might hold or scheduling heuristics that its policy might apply?)
4. what is the default value of the inheritance attribute?

While we guess that the intended answers to these questions are that scope *is* covered, that attributes are copied rather than being permanently associated, that the base attributes are the ones inherited and that inheritance is the intended default, we can only recommend that you read your local documentation closely and write portable software so that both the scheduling inheritance attribute and the contention scope are always set explicitly whenever they are significant.

In view of these problems, the best advice of all might be for portable, scheduling-sensitive code always to specify explicit scheduling, explicitly.

We advise against reliance on inherited scheduling only because of its poor specification in the standard. It is intended to provide a convenient means of arranging for new threads to to acquire the scheduling characteristics of their parents. When a thread is created to perform a subcomputation this is almost always the right behavior, since the model in which each level of priority absolutely dominates all lower ones means that the parts of a computation must be done at a priority no lower than that of their parent, and usually no higher. Similarly, copying other scheduling attributes from the parent thread leads to more modular and predictable behavior relative to uninvolved threads. In sum, discontinuities of scheduling attributes are generally useful in 'configurational' thread creation while inheritance is typically better for 'decompositional' applications.

Inherited scheduling is, by the way, one of the few cases in which there is any kind of parent/child relationship between POSIX threads.

9.3.2 Scheduling scope

The **contention scope** of a thread determines what it competes with for processing resources. A 'system scope' thread is one that is scheduled by the system scheduler: it relies on kernel support to compete directly with conventional processes for processor time. This contrasts with 'process scope' threads which are managed within a single process, often by user-level code in the threads library.

Contention scope is the only standard attribute of a POSIX thread that cannot be modified after the thread is created: different scopes produce truly different flavors of objects. Many systems, in fact, will allow objects of only one of the two scopes: whether a thread is a system-level or a process-level object is a design parameter of many threads packages. Nonetheless an interface is provided for setting the scope desired in the event that an implementation does provide both.

`pthread_attr_{set,get}scope()` **13.5.1⋆**

```
#include <pthread.h>
#include <unistd.h>         /* For _POSIX_THREAD_PRIORITY_SCHEDULING */
#ifdef _POSIX_THREAD_PRIORITY_SCHEDULING

int /*errno*/ pthread_attr_setscope
  ( pthread_attr_t *attributes
  , int /*scope*/ new_scope
  );

int /*errno*/ pthread_attr_getscope
  ( pthread_attr_t *attributes
  , int /*scope*/ *old_scope
  );
```

`pthread_attr_t`$_{138}$ `*`*`attributes`*

Pointer to initialized$\Rightarrow^{8.2.1}_{140}$ thread attributes object to be examined or changed.

`int /*scope*/` *`new_scope`*

New value for the scheduling contention scope attribute, chosen from the constants below.

`int /*scope*/ *`*`old_scope`*

Pointer to location receiving the old value of the scheduling contention scope attribute.

The possible (standard) scopes are:

`PTHREAD_SCOPE_PROCESS` The thread directly competes for processing time only with other threads of this process; the process effectively has its own sub-scheduler. The thread's actual performance thus depends on the *process* priority and scheduling parameters as well as those of the thread itself.

`PTHREAD_SCOPE_SYSTEM` The thread competes for processing time with other threads of system scope, regardless of the processes in which they reside; the thread is directly visible to the system scheduler. The thread's performance thus does not directly depend on the scheduling parameters of its owning process.

An implementation is only required to support *one* of the two contention scope values, process and system; if it implements both, it may choose which of them is the default. Check your system's documentation.

On success these functions return zero.

On failure an integer error indication is returned:

- `ENOSYS`$_{271}$ this implementation does not support the priority scheduling mechanism.
- `EINVAL`$_{271}$ *(*`pthread_attr_setscope()` *only)* the value of *`new_scope`* was invalid.
- `ENOTSUP`$_{271}$ *(*`pthread_attr_setscope()` *only)* the value of *`new_scope`* is not supported by this implementation.

In some sense a thread's contention scope dominates all other scheduling attributes, since the interaction of threads with different scopes is not specified.

From the system's perspective a system-scope thread is very much like a new process that happens to share *all* its memory with its parent (though in truth it receives its own stack and shares its filesystem context and so on, as for any thread). A thread with process scope, on the other hand, hardly exists; *to the kernel* it looks rather like a very complicated kind of function call, part of the normal execution of a a process. Creating a process-scope thread may involve no system calls at all.

This will often have the consequence that switching between running threads is *much* cheaper for process-scope threads than for system-scope threads; it takes a more than usually sophisticated system design to propagate system-level policy decisions into a process without incurring at least the expense of a system call. Process-scope threads will also consume less resources than system-scope threads do, allowing larger numbers of them to co-exist effectively within the system.

From the user's point of view this may have profound impact on performance. If a thread has process scope, it is at the mercy of the process scheduling algorithm: it can only receive processing cycles when the process that contains it is selected to run. By their very nature, furthermore, multithreaded *processes* are relatively rarely blocked; usually one or another of their threads has work to do. Thus, their scheduling is largely determined by the process scheduling policy, rather than the behavior of the threads themselves.

The consequence of this two-level scheduling behavior is that it's very difficult to predict the absolute performance of process-level threads, even though their scheduling policy may completely determine how the threads within a single process behave relative to each other. Nonetheless it is much more efficient, since much less of the work is done by the operating system kernel.

In fact the situation may be even more complicated: on systems that provide *both* user and kernel level threads, *hybrid* systems are possible in which a pool of kernel threads is shared among a number of user threads. In such a system, there may be additional operations to create and delete kernel threads in the pool; these differ from normal thread operations in that they don't change the number of threads of control in the program, but affect the way they map onto the hardware.

Systems with multiple physical processors are quite likely to follow this model; with pool thread creation and deletion allocating or releasing real CPUs.

The interested reader should see also the discussion of user and kernel threads in the appendices to chapter.$\Rightarrow^{9.a}_{179}$

9.3.3 The policy attribute

POSIX thread scheduling is based on the new POSIX real-time process scheduling system, and uses the same three scheduling policies: FIFO (first in—first out), round-robin, and 'other' (which is each system's 'normal' scheduling policy and is otherwise unspecified by the standard). All three will be present in an implementation that fully supports the thread priority scheduling option, and a policy can be chosen separately for each thread by setting the policy attribute at thread creation (and may be modified later using `pthread_setschedparam()`[170]).

`pthread_attr_{set,get}schedpolicy()` **13.5.1⋆**

```
#include <sched.h>                                  /* for SCHED_* */
#include <pthread.h>
#include <unistd.h>        /* For _POSIX_THREAD_PRIORITY_SCHEDULING */
#ifdef _POSIX_THREAD_PRIORITY_SCHEDULING

int /*errno*/ pthread_attr_setschedpolicy
  ( pthread_attr_t *attributes
  , int /*policy*/ new_policy
  );

int /*errno*/ pthread_attr_getschedpolicy
  ( pthread_attr_t *attributes
  , int /*policy*/ *old_policy
  );
```

`pthread_attr_t`$_{138}$ ***attributes***
: Pointer to initialized$\Rightarrow^{8.2.1}_{140}$ thread attributes object to be examined or changed.

`int /*scope*/` ***new_policy***
: New value for the scheduling policy attribute, chosen from the constants below. $\Rightarrow^{9.3.4}_{167}$

`int /*scope*/` ****old_policy***
: Pointer to location receiving the old value of the scheduling policy attribute.

The available scheduling policies may include:

`SCHED_FIFO` The **first in, first out** standard priority scheduling policy selects the least recently run thread of the highest priority that is not blocked waiting for a resource, and allows it to run until (a) it calls a function that blocks, (b) a higher-priority thread is unblocked, or (c) it yields$\Rightarrow^{9.4}_{172}$—however long this takes.

`SCHED_RR` The **round-robin** standard priority scheduling policy is identical to the FIFO policy except that the current thread is periodically de-scheduled to allow other threads at the same priority to run. The frequency with which this happens is implementation-defined.[2]

`SCHED_OTHER` The uninformatively named 'other' scheduling policy is the standard system-wide scheduling policy; POSIX systems of different heritage may implement different policies of varying degrees of heuristic sophistication.

Implementations may define further scheduling policies; they should define how they interact and specify any necessary control parameters as fields in `struct sched_param`$_{169}$. The interaction between FIFO and round robin scheduling is fully specified by the standard (see the discussion following) and depends on a single priority parameter that they both respect. The parameters and interactions of the 'other' policy are implementation-defined.

On success these functions return zero.

On failure they return one of the following error codes:

- `ENOSYS`$_{271}$ priority scheduling is not supported by this implementation.
- `EINVAL`$_{271}$ *(*`pthread_attr_setschedpolicy()` *only)* the value of ***new_policy*** is invalid.
- `ENOTSUP`$_{271}$ *(*`pthread_attr_setschedpolicy()` *only)* the value of ***new_policy*** is unsupported by this implementation.

Note that this attribute is ignored unless the inheritance attribute of ***attributes*** is `PTHREAD_EXPLICIT_SCHED`$_{161}$.

The policy chosen in general completely determines the meaning of the scheduling parameters described in the next section; but FIFO and round robin have been defined each to take only one parameter, a **priority**, which they interpret compatibly.

[2] The function `sched_rr_get_interval(pid_t, struct timespec *)` is provided to obtain the timeslice length for a real time *process*, but no corresponding operation seems to exist for threads, even if they have system contention scope.

9.3.4 The scheduling policies

The basic priority scheduling model is very straightforward. Assume that the number of threads in a given contention scope that can *physically* run at one time is fixed (on a uniprocessor, of course, this number is always either one or—for process-scope threads within a de-scheduled process—zero. In general it is determined in some complex manner by the applicable **allocation domains**, and may not really be constant). At any given moment, all the highest-priority threads that are not actually blocked waiting for a mutex, resource or event are running, up to that maximum. If this means that some, but not all, of the threads at some priority level are running, it is the least recently unblocked threads that are chosen to run first, so that—so long as every thread blocks eventually—each thread gets a turn.[3]

One way of thinking about this is to imagine that the scheduler stores the ready-to-run threads in an array of queues—the array is indexed by priority, and the queues by order of arrival. When selecting a new thread to run, the scheduler effectively looks through the priority array until it finds a non-empty queue; it then takes the first thread on that queue.

The FIFO and round-robin share this approach to selecting threads to run; they differ in how threads are *de*-scheduled and moved to the back of the queue for their priority level.

Note that in consequence of the strategy just described, a thread can assume that no thread at a higher priority is waiting to run—but it does not in general follow that no thread at a lower priority is actually running (unless you can prove it indirectly on the basis of the number of processors available—or more generally the details of the allocation domains in effect. For instance, you might be *certain* that your software is running on only one physical CPU[4]). It follows that there is no way for a thread to deliberately defer to another thread of lower priority, other than by blocking itself, perhaps on a condition variable.

FIFO (`SCHED_FIFO`$_{166}$) threads are simply managed: once scheduled, a FIFO thread runs (subject to the availability of the processor at its priority level) until it either blocks or (as we shall see later in this chapter) it calls `sched_yield()`$_{172}$ to deliberately relinquish its turn. This is the 'run until stopped' policy.

Whether it yields or is blocked, the thread next to run is chosen according to the basic rules—this may even result in the same thread running again.

It is important to note that so long as there is only one CPU the FIFO scheduling policy is *deterministic*—the same external sequence of events will result in the same interleaving of thread execution, and the possible interleavings are relatively straightforwardly derivable from the software structure. Software that uses FIFO scheduling

[3] There's a slight unclarity in the standard here. It may also be interpreted as saying that a thread that loses its processor because a higher-priority thread unblocks also loses its place in the queue. The interpretation given here is both simpler and more consistent, however.

[4] An increasing number of multiprocessor architectures are appearing on the market, however; the days are probably past when one could assume that multiprocessors were rare and specialized equipment. Our advice: don't rely on reasoning about the number of processors for correctness, unless your system provides an extension that lets you *specify* it explicitly.

(but does not deliberately rely on it for correctness!) may be more robust and more reliable in the field, since the nature of the policy greatly limits the *number* of possible interleavings of execution and in effect improves the testability and debuggability of the code. Direct reasoning about correctness is also facilitated.

On the other hand, a FIFO thread may run indefinitely, starving its siblings—*even if they themselves use other policies.*

Round-robin (`SCHED_RR`$_{166}$) scheduling is only slightly more complicated. In this scheme, a thread is allocated a **time slice**, and it is only allowed to run until that slice has expired. At that point—if none of the other things that would cause it to be de-scheduled has already occurred—it is made to relinquish the processor, as if it had called `sched_yield()` explicitly. This is the 'run until bored' policy.

Once again, this will not always prevent the thread just interrupted from being the next to run, if there are enough processors to support all of the runnable threads at its priority level and above.

At the expense of reproducibility, the round-robin policy buys some fairness: a thread running under this policy cannot starve another thread at the same priority level just by performing lengthy calculations without calling any blocking functions.

Other (`SCHED_OTHER`$_{166}$) The unspecified 'other' policy is supported to allow systems to implement a scheduling policy for non-real-time processes and threads like they have in the past. Berkeley-type systems, for instance, have a scheduler that tries to allocate the CPU fairly between all processes by adjusting the effective priority of the process on the basis of its recent demands. Note that such a policy also needs to use the scheduling parameter mechanism, but—for example—its notion of 'priority' is not quite the same as that of FIFO or round-robin scheduling. This—forgive me—is the 'run until little green men jump out from behind the refrigerator' policy.

9.3.5 Scheduling parameters

A scheduling policy may associate one or more **scheduling parameters** with a thread in order to guide its behavior. The standard priority-scheduling policies FIFO and round-robin both rely on a **priority**, an integer falling between the values reported by `sched_get_priority_min()`$_{293}$ and `sched_get_priority_max()`$_{293}$ for the policy in question.$\Rightarrow^{F}_{293}$. All the parameters of the available policies are represented as fields in `struct sched_param`, and this whole structure can be specified as a *single* thread creation attribute.

`pthread_attr_{set,get}schedparam()` **13.5.1**

```
#include <sched.h>                                    /* for SCHED_* */
#include <pthread.h>
```

```
int /*errno*/ pthread_attr_setschedparam
  ( pthread_attr_t *attributes
  , const struct sched_param *new_param
  );

int /*errno*/ pthread_attr_getschedparam
  ( pthread_attr_t *attributes
  , struct sched_param *old_param
  );

struct sched_param
  {
    ...
    int sched_priority;
    ...
  };
```

`pthread_attr_t`$_{138}$ ***attributes***
: Pointer to initialized$\Rightarrow^{8.2.1}_{140}$ thread attributes object to be examined or changed.

`const struct sched_param` ***new_param***
: *Pointer* to a new scheduling parameters structure to be used for threads created with these ***attributes***.

`struct sched_param`$_{169}$ ***old_param***
: Pointer to a structure to receive the scheduling parameters currently specified by ***attributes***.

These functions provide a method of specifying the initial value of any parameters that may be associated with a thread's scheduling policy. The relevant fields of the parameter structure are determined by the policy; the standard priority scheduling policies `SCHED_FIFO`$_{166}$ and `SCHED_RR`$_{166}$ take a single integer parameter, `sched_priority`, and interpret it consistently and compatibly within each scope—higher numbers denote higher priorities. At any moment, at least one of the (non-blocked) threads having the highest priority will be running; depending on the details of the allocation domain model in effect other threads may also run.$\Rightarrow^{9.3.4}_{167}$ Other policies may define other parameters; see your implementation's documentation for details.

On success these functions return zero.

On failure they return one of the following integral error codes:

- `ENOSYS`$_{271}$ This implementation does not support this function. This actually contravenes the standard. [[**sps: CHECK**]]
- `EINVAL`$_{271}$ *(*`pthread_attr_setschedparam()` *only)* *`*new_param`* is invalid.
- `ENOTSUP`$_{271}$ *(*`pthread_attr_setschedparam()` *only)* one of the fields of *`*new_param`* is unsupported.

Note that this attribute is ignored unless the inheritance attribute of ***attributes*** is `PTHREAD_EXPLICIT_SCHED`$_{161}$.
See also `pthread_setschedparam()`$_{170}$.

You may notice that the style of this interface is very different from that of most of the pthreads facility: rather than an opaque data type with an extensible vocabulary of accessors, the scheduling parameters are represented in an explicit `struct` object to which different policies contribute whatever fields they might need. The type `struct sched_param` is in fact shared with the POSIX real-time extensions (where it is used to specify *process* scheduling behavior) and was originally defined there.

Next we present functions that can be used to read and modify a thread's scheduling parameters *dynamically,* while the thread is running.

9.3.6 Dynamic scheduling change

After a thread is started, its parameters and policy can be examined with the inconsistently named function `pthread_getschedparam()`, and changed with `pthread_setschedparam()`. These functions take a thread ID as an argument, and can thus be used to affect threads other than the current one (this is an unavoidable necessity, since the functions need to be able to operate on threads with low priorities which might otherwise have no chance of running). Since these functions affect an already created thread, they don't use a thread creation attributes object.

`pthread_{set,get}schedparam()` **13.5.2⋆**

```
#include <sched.h>                                   /* for SCHED_* */
#include <pthread.h>
#include <unistd.h>        /* For _POSIX_THREAD_PRIORITY_SCHEDULING */
#ifdef _POSIX_THREAD_PRIORITY_SCHEDULING

int /*errno*/ pthread_setschedparam
  ( pthread_t thread
  , int /*policy*/ new_policy
  , const struct sched_param *new_param
  );

int /*errno*/ pthread_getschedparam
  ( pthread_t thread
  , int /*policy*/ *old_policy
  , struct sched_param *old_param
  );
```

`pthread_t`$_{32}$ *`thread`*
: The thread object whose policy and parameters are to be manipulated.

`int /*policy*/` *`new_policy`*
: An integer representing the new scheduling policy to install for ***thread***.$\Rightarrow^{9.3.4}_{167}$

`const struct sched_param`$_{169}$ `*`*`new_param`*
: *Pointer* to a structure containing the corresponding new scheduling parameters.

`int /*policy*/ *`*`old_policy`*
: Pointer to an integer receiving the policy currently in effect for *`thread`*.

`struct sched_param`$_{169}$ `*`*`old_param`*
: Pointer to a structure receiving the corresponding parameters.

Valid policies include at least `SCHED_FIFO`$_{166}$, `SCHED_RR`$_{166}$ and `SCHED_OTHER`$_{166}$. The relevant fields of the parameter structures are determined by the policy; in particular, `SCHED_FIFO` and `SCHED_RR` use an integral field *`sched_priority`*.

The values returned by `pthread_getschedparam()` are the *base* values from the thread's creation and updated by `pthread_setschedparam()`; they are unaffected by any temporary change in priority resulting (for example) from the thread's state.

On success zero is returned.

On failure the result is an error code; and in the case of `pthread_setschedparam()` no change has been made to the thread's scheduling. The codes returned are as follows:

`ENOSYS`$_{271}$ `_POSIX_THREAD_PRIORITY_SCHEDULING` is not defined and these functions are not supported.

`ESRCH`$_{271}$ The *`thread`* seems not to be valid.

`EINVAL`$_{271}$ *(`pthread_attr_setschedparam()` only)* Either the *`new_policy`* or one of the parameters it uses from `*`*`new_param`* is invalid.

`ENOTSUP`$_{271}$ *(`pthread_attr_setschedparam()` only)* Either the *`new_policy`* or one of the parameters it uses from `*`*`new_param`* is unsupported by this implementation.

`EPERM`$_{271}$ *(`pthread_attr_setschedparam()` only)* The calling thread doesn't have the privileges necessary to perform this operation (at least with the parameters specified).

Note that the standard does not give any conditions under which `EPERM` would be returned, but it does permit an implementation to define privileges that are needed to change thread priorities. In particular, changing the scheduling parameters of system scope threads might be restricted in similar ways and for the same reasons as changing process priorities.

The definitions of the policies and the corresponding parameters are described in an earlier section.$\Rightarrow^{9.3.4}_{167}$

Be aware that when a thread dynamically reduces its own priority or increases that of another thread may lose the processor *immediately,* before it even returns from this function.

9.4 Yielding

It is sometimes desirable for a running thread to give up the processor voluntarily, to ensure that other threads have a chance. There is a function, originally introduced as part of the real-time extensions, that can be used for this purpose: `sched_yield()`.

Because of the way the priority scheduling algorithms work (and this may not apply to other scheduling policies), it is only possible for a thread to yield to another thread of the same priority as the caller; and there is no direct way of controlling *which* thread will run next. If there are no other threads of the current priority, calling `sched_yield()` will have no effect—the caller will immediately be scheduled to run again.

`sched_yield()` **13.3.5**

```
#include <sched.h>

void sched_yield(void);
```

By calling `sched_yield()`, a thread signals its willingness to have some other thread start running immediately, in its place. It takes no arguments, returns no result, and has no semantics in the traditional sense—if a process contains only a single thread, `sched_yield()` is a no-op. It is nonetheless an important component of almost any synchronization scheme, since it enables a thread to indicate that until something external happens, it is wasting processor time by running.

Warning: This is only useful when there are other threads of the same scope and priority also in a runnable state!

In earlier drafts of the standard there was a distinct function `pthread_yield()`, with `sched_yield()` applying only to *processes,* but the two functions have now been collapsed together.[5]

9.5 Scheduling and synchronization

The concepts of mutex variables and priority thread scheduling are individually quite simple, but their combination is subject to some subtle and unexpected interactions. The problem is that because mutual exclusion must, in order to guarantee consistency, be able to block *any* thread regardless of priority, a low-priority thread can indefinitely prevent even a higher-priority thread from running—a situation known as a **priority inversion**. The net effect of an inversion is to *lower* the priority of the high-priority thread: if there exists a third thread of intermediate priority, it will run in preference to the thread holding the lock—which can prevent the high-priority thread from running indefinitely! This problem arises whenever threads at different priority levels share mutex-protected data.

[5] This apparently means that there is no longer a way for a process-scope thread to cause its parent process to relinquish the processor.

One theoretical escape would be to allow high-priority threads to 'steal' a mutex, but this is not an acceptable solution in practice: recall that the purpose of the mutex is primarily the protection of data structures from asynchronous manipulation, and blocking is just a byproduct of the technique employed. Another solution, the one adopted in POSIX, observes that in the event of a priority inversion, the thread holding the mutex is actually on the **critical path** of the blocked thread; since in getting out of the way of another thread it is working on the blocked thread's behalf as well as its own, it 'deserves' to have its priority raised. (This phenomenon also occurs in real life; in some countries, for example, private cars are temporarily exempted from usual traffic laws when they need to clear the path of an emergency vehicle.) This section describes the additional varieties of mutex POSIX provides to get this more priority-safe effect, and how they are controlled.[6]

Hackery note: It may occasionally be useful to put such a mutex around library functions that already provide exclusion internally, simply in order to get the priority-management behavior you need.

9.5.1 The mutex priority model

The basic idea behind the POSIX mutex priority model is that a thread may have its **effective priority** (the priority used for scheduling decisions) temporarily raised above its **base priority** (the priority initially set from the thread attributes and accessible to `pthread_getschedparam()`$_{170}$/`pthread_setschedparam()`$_{170}$) by the mutexes that it owns. Mutexes, too, have priorities, and at any point in time, a thread's effective priority is the *maximum* of its base priority and the priorities of any mutexes it has locked. This works best when the priorities of both threads and mutexes are chosen to be just high enough to get the job done; maximum (rather than, say, sum) is the right operation for combining them since, as you will recall, under standard priority scheduling, a higher effective thread priority *completely dominates* a lower one.

This simple model for the combination of scheduling effects allows the system to determine the consequences of owning several mutexes, even if they use different algorithms for their individual effective priorities.

Under the priority scheduling option, POSIX in fact describes three different kinds of mutexes with different priority protocols, though implementations are only required to provide the basic variety. These are:

Basic mutexes which have no effect on the effective priorities of threads holding them; in terms of the uniform model of priority combination just outlined, their

[6] Actually, successful systems *have* been built in which, at least optionally, a high-priority thread may 'steal' a mutex from a low priority thread that holds it; the crucial idea is to let the low priority thread execute *recovery code* at the higher priority, so that the protected data are eventually handed over in a consistent state. Having the low-priority thread go ahead and complete the update—the POSIX solution—is of course a trivial special case of this strategy. Other variants include somehow aborting and retrying when a conflict is detected; this works well in large systems, since conflict detection can then be overlapped with normal processing. None of these more complex strategies yet provides a robust high-level solution, however, which is why they have not been adopted in pthreads.

priority is fixed at the minimum thread priority. They are simple and efficient, but as already noted they may be dangerous when shared by threads of different priorities.

Priority ceiling mutexes which have a *programmable* priority that is set when they are initialized and can be changed, if required, with the function `pthread_mutex_setprioceiling()`$_{178}$.

Priority inheritance mutexes have a dynamically varying priority. The priority of one of these mutexes is the maximum of the effective priorities of all threads that are blocked on it. Note that this effect is transitive, since the effective priorities of the blocked threads may in turn be affected by other mutexes that they own.

When a mutex holds a priority ceiling or priority inheritance mutex it is also protected from being moved to the end of the scheduling queue at its priority level as a result of changes to its *base* priority, even when the mutex is finally released.

9.5.2 Specifying mutex scheduling protocol

`pthread_mutexattr_{set,get}protocol()` **13.6.1**

```
#include <pthread.h>
#include <unistd.h>                    /* For configuration constants */
#ifdef _POSIX_THREAD_PRIORITY_SCHEDULING
#if defined(_POSIX_THREADS_PRIO_PROTECT)                              \
 || defined(_POSIX_THREADS_PRIO_INHERIT)

int /*errno*/ pthread_mutexattr_setprotocol
  ( pthread_mutexattr_t *attr
  , int /*mutex scheduling protocol*/ new_protocol
  );

int /*errno*/ pthread_mutexattr_getprotocol
  ( pthread_mutexattr_t *attr
  , int /*mutex scheduling protocol*/ *old_protocol
  );
```

`pthread_mutexattr_t`$_{138}$ `*attr`
: Pointer to the initialized$\Rightarrow^{8.2.1}_{140}$ mutex attributes object to be examined or modified.

`int /*mutex scheduling protocol*/ new_protocol`
: Integer specifying the new scheduling protocol for mutexes initialized with these *`attributes`*, chosen from the list below.

`int /*mutex scheduling protocol*/ *old_protocol`
: Pointer to an integer receiving the protocol currently specified by *`attributes`*.

Valid mutex scheduling policies include:

`PTHREAD_PRIO_NONE` *(default)* This mutex has no effect on the effective priority of the thread holding it; great care should be exercised in sharing it between threads of different priority, since priority inversions are a danger.⇒$^{9.5}_{172}$

`PTHREAD_PRIO_PROTECT` *(if* `_POSIX_THREAD_PRIO_PROTECT` *defined)* The mutex obeys the priority protection protocol; the priority of a thread holding the mutex is raised to that specified by the **priority ceiling** attribute of ***`attributes`***, which, to avoid inversions, must be at least as high as the priority of any thread that locks it. Although it is a poor piece of design and the description of priority protection mutexes in the standard suggests otherwise 13.6.1.2, the specification of `pthread_lock()`?? explicitly states that an attempt to lock a mutex that violates this constraint will fail with a return code of `EINVAL`$_{271}$ 11.3.3.4.

`PTHREAD_PRIO_INHERIT` *(if* `_POSIX_THREAD_PRIO_INHERIT` *defined)* The mutex supports priority inheritance; the effective priority of a thread holding the mutex is dynamically raised to match the effective priority of the highest-priority thread that is blocked on it.

The protocol of a mutex determines the effect of owning that mutex on the effective priority of the owning thread. If a thread holds multiple mutexes, their effects are combined (along with the base priority of the thread itself) and the effective priority is the maximum of all such contributions.

On success these functions return zero.

On failure they return one of the following error codes:

`EINVAL`$_{271}$ One of the arguments is invalid.

`ENOSYS`$_{271}$ Neither `_POSIX_THREAD_PRIO_PROTECT` nor `_POSIX_THREAD_PRIO_INHERIT` is defined, and the operation is not supported.

`ENOTSUP`$_{271}$ *(`pthread_mutexattr_setprotocol()` only)* The value of ***`new_protocol`*** is not supported by this implementation.

`EPERM`$_{271}$ The caller lacks the privileges to perform this operation. (Since it can be used as a back-door method of changing a thread's priority.)

The choice between the different kinds of mutexes depends on the problem being solved. If priority inversions are not an issue—if thread priorities are not being used at all, or if the code sequences protected by the mutex are very short and there are plenty of opportunities for higher-priority threads to block—then basic mutexes are the best solution, because they are the most efficient. Otherwise, the decision depends on whether the mutex is *primarily* used at a low or a high priority.

If it is mostly used by low-priority threads and contention with high-priority threads is rare, the priority inheritance protocol is appropriate. The reasoning is that it does not disrupt the normal priority scheme of the application unless it is actually necessary.

If, on the other hand, the mutex is often needed by high-priority threads, it may make sense to use the priority ceiling protocol, since raising the priority of any low-priority thread that acquires the mutex will enable it to complete its critical tasks more rapidly and reduce the overall level of contention between threads. In order for this to work correctly it is important that the priority chosen for the mutex to be at least as high as that of the highest-priority thread that uses the mutex; otherwise the priority inversion effect is still possible.

Taking this kind of reasoning a little further, the possibility also exists of using priority mutexes simply for their scheduling effects, but we will not explore that here.

9.5.3 Specifying mutex priority ceiling

Mutexes that employ the priority ceiling protocol (`PTHREAD_PRIO_PROTECT`$_{175}$) have an inherent priority to which they boost threads that acquire them. This is set at mutex creation from the priority ceiling initialization attribute though it can also be altered dynamically$\Rightarrow^{9.5.4}_{178}$. Its value is a thread priority (such as might be assigned to a thread with FIFO or round-robin scheduling).$\Rightarrow^{9.3.5}_{168},^{9.3.6}_{170}$

`pthread_mutexattr_{set,get}prioceiling()` **13.6.1⋆**

```
#include <pthread.h>
#include <unistd.h>                /* For _POSIX_THREAD_PRIO_PROTECT */
#ifdef _POSIX_THREAD_PRIO_PROTECT

int /*errno*/ pthread_mutexattr_setprioceiling
  ( pthread_mutexattr_t *attr
  , int /*thread priority*/ new_ceiling
  );

int /*errno*/ pthread_mutexattr_getprioceiling
  ( pthread_mutexattr_t *attr
  , int /*thread priority*/ *old_ceiling
  );
```

`pthread_mutexattr_t`$_{138}$ `*attr`
: Pointer to the initialized$\Rightarrow^{8.2.1}_{140}$ mutex attribute object whose priority ceiling is to be examined or changed.

`int /*thread priority*/ new_ceiling`
: New ceiling priority for mutexes initialized with *`attributes`*.

`int /*thread priority*/ *old_ceiling`
: Pointer to integer receiving old priority ceiling associated with *`attributes`*.

When a mutex has priority protocol `PTHREAD_PRIO_PROTECT`$_{175}$, the **priority ceiling** mutex attribute specifies the minimum priority to which threads owning the mutex are boosted. This value can subsequently be modified, if necessary, with `pthread_mutex_setprioceiling()`$_{178}$.

On success these functions return zero.

On failure they return an error number:

- `EINVAL`$_{271}$ One of the arguments is invalid.
- `ENOSYS`$_{271}$ `_POSIX_THREAD_PRIO_PROTECT` is not defined, and the operation is not supported.
- `ENOTSUP`$_{271}$ *(`pthread_mutexattr_setprioceiling()` only)* The value of *`new_ceiling`* is not supported by this implementation.
- `EPERM`$_{271}$ The caller lacks the privileges to perform this operation. (Note that it can be used as a back-door method of changing a thread's priority.)

In order to prevent priority inversions, the priority ceiling chosen for a mutex should meet or exceed the highest priority of any thread that locks the mutex. Indeed, an error (`EINVAL`$_{271}$) is reported if a higher priority thread attempts to lock such a mutex.

Warning: This has the potential to cause severe bugs if you forget to check the return code from `pthread_lock()`??, since your program is then likely to go ahead and make an unsynchronized access, a problem that is probably far more serious—and harder to diagnose—than the inversion itself. When locking a priority ceiling mutex, *always* check the return code.

Priority inversion is normally a fatal problem only because of **processor starvation**: if there is only one physical processor, low priority threads can be descheduled indefinitely, preventing them from releasing resources they are using. What happens if there are multiple processors?

If we assume that any thread can run on any processor, the situation clearly improves: if there are k processors, the top k threads can be allowed to run, even if they are not all at the same priority level, and only the $k+1^{\text{st}}$ thread is subject to starvation (and can therefore provoke inversions). But not all harddware is so flexible.

If threads are bound to particular physical processors[7] things are more complex. Now priority inversions are again possible between the threads running on a single processor, but there is a further problem: **remote blocking**. Even when priority ceilings are chosen correctly, a thread that holds a mutex can still be descheduled (and possibly starved) to accomodate a higher priority thread that does not use the mutex in question. The problem comes when that mutex is used by threads running on other processors: if they request the shared mutex they will naturally block until it becomes free, potentially making the entire remote processor sit idle until an unrelated high-priority thread on the processor owning the mutex completes its task. Your entire multiprocessor could be reduced to running just a single thread by this surfeit of serialization!

To avoid this effect you still need to follow the usual rules for choosing the ceilings of 'local' mutexes (that are used only within a single processor) but set the priorities of mutexes that provide synchronization between processors even higher—to a value strictly *greater* than the priority of *any* of the threads on *any* of the processors that use it (whether those threads are directly involved or not). This will ensure that

[7] Or where there are threads in multiple processes communicating by shared memory.

any thread that holds a resource on which remote *processors* might block has a high enough priority to compete with its local cohort.

9.5.4 Dynamic mutex priority ceiling change

Although the scheduling protocol of a mutex is set when it is initialized, mutexes using the priority ceiling protocol can have their inherent priority modified later on.

pthread_mutex_{set,get}prioceiling() **13.6.2⋆**

```
#include <pthread.h>
#include <unistd.h>                /* For _POSIX_THREAD_PRIO_PROTECT */
#ifdef _POSIX_THREAD_PRIO_PROTECT

int /*errno*/ pthread_mutex_setprioceiling
  ( pthread_mutex_t *mutex
  , int /*thread priority*/ new_ceiling
  , int /*thread priority*/ *old_ceiling
  );

int /*errno*/ pthread_mutex_getprioceiling
  ( const pthread_mutex_t *mutex
  , int /*thread priority*/ *old_ceiling
  );
```

`pthread_mutex_t`$_{68}$ `*mutex`
: Pointer to the mutex variable to be operated upon.

`int /*thread priority*/ new_ceiling`
: New priority to install as the 'ceiling' of `mutex`.

`int /*thread priority*/ *old_ceiling`
: Pointer to a variable that receives the previous/current priority value.

These functions read the current priority associated with `mutex`; `pthread_mutex_setprioceiling()` also atomically installs a new priority. This priority is the value to which the priority of a thread owning the mutex is temporarily boosted.

These operations are legal whenever `_POSIX_THREAD_PRIO_PROTECT` is defined by the implementation, but are not meaningful for mutexes other than those that use the `PTHREAD_PRIO_PROTECT`$_{175}$ protocol.

On success both functions return zero.

On failure they return one of the following error codes:

`ENOSYS`$_{271}$ `_POSIX_THREAD_PRIO_PROTECT` is not defined, and the operation is not supported.

`EINVAL`$_{271}$ *`new_ceiling`* is out of range or *`mutex`* is invalid.

`EPERM`$_{271}$ The caller lacks the privileges to perform this operation. (Note that it can be used as a back-door method of changing a thread's priority.)

Warning: According to the wording of the standard, `pthread_mutex_setprioceiling()` blocks until *`mutex`* becomes available, and thus will deadlock if used by a thread that already owns the mutex.

The choice of a mutex's priority may be important to correctness, and the correct value is a global property of the program; so modifying it on the fly should not be taken lightly. Nonetheless, since it is a function of the priorities of the threads that will be using the mutex, its modification is likely to be useful or necessary in some of the circumstances in which dynamic modification of a thread's basic priority is performed with `pthread_setschedparam()`$_{170}$.

The reason for the 'atomic swap' design of `pthread_mutex_setprioceiling()` is that it is intended for use precisely by threads that do *not* have the mutex locked and are not therefore assumed to be synchronized.[8]

9.6 Example

9.7 Discussion

[[cap: **A user once asked "how do I specify the next thread to be run" Well there is no call for that, but it can be built using the pthread_setschedparam() routine.**

This assumes that the specified thread isn't blocked waiting for an event and that the current thread isn't running at the highest priority possible.]]

Appendices

9.a Implementation considerations

[[sps: **We might want to put a complete description of how thread states are implemented somewhere. Is that "somewhere" here, or does it need a section/chapter of its own elsewhere in the book? Certainly this section can be expanded, made a lot more detailed and interesting, by tracing how priority propagates around, and maybe doing a priority-queue algorithm.**]]

Compared to other priority-based scheduling schemes, the standard POSIX priority scheduling scheme is very simple. The choice of which (ready) thread to run next depends only on its priority; the mechanism for determining when to de-schedule it depends only on its scheduling policy; and any temporary adjustments in priority are synchronous: they derive from actions taken by either the thread itself or a thread that defers to it. Thus, of the scheduling features (so far[9]) mandated by the standard, the implementationally interesting one is contention scope.

---

[8] This is admittedly an inconsistency with the calling conventions of `pthread_setschedparam()`$_{170}$.

[9] The concept of allocation domain in multiprocessors is a major field of research and has been for many years. If this, too, converges to something readily standardizable, it will be fascinating!

CAPthreads, the implementation we have been basing our discussion of the implementation around, is a user-space thread implementation; that is, it provides **PTHREAD_SCOPE_PROCESS**$_{164}$ as its only option for contention scope. This means that a thread can be a data structure existing entirely within the process$\Rightarrow^{4.a}_{50}$, and synchronization facilities are normal user-level library functions (albeit ones that occasionally use low-level processor features). When a thread blocks, control is transferred to a thread scheduler within the process; the scheduler attempts to find another runnable thread and 'returns' to that, instead of to its caller. Only if no threads are runnable does it deliberately resign the processor to the kernel. The consequences are (1) that mutual exclusion between threads can be implemented purely in software, by providing an atomically-writable variable that inhibits thread rescheduling; and (2) that it absolutely relies on the kernel's normal process scheduling policy to ensure fairness across the system: the thread scheduler operates completely without regard to the outside world.

(In fact, observation (1) applies only to process-local operations. Synchronization through shared memory, if provided, still requires "real" low-level synchronization instructions, since, in consequence of (2), scheduling of a thread relative to threads in another process is not controllable.)

Kernel threads, on a uniprocessor, are even simpler,[10] since all synchronization operations will be implemented as kernel calls: there is only one scheduler and it can have a conventional architecture. As already noted, the only real difference between a kernel thread and a conventional process is that it does *not* have an address space of its own. Actually, this is the insight behind an optimization that is sometimes used: switching between address spaces is a relatively expensive operation, because a modern processor contains a large amount of hidden state (in hardware caches of various kinds) specific to an address map. Context switch overhead can be reduced if the sequence in which threads are dispatched can be manipulated in such a way that threads within the same process are scheduled back-to-back[11]; and when multiple processors are available, threads from the same process should be scheduled on the same processor and/or on a processor on which they have run recently. In general this cannot be done in total freedom within the POSIX priority scheduling model, but for threads with **SCHED_OTHER**$_{166}$ and those within processes not using shared memory, it is feasible.

[[And 'scheduling threads onto kernel entities'?]]

9.a.1 Why neither scope is required

As we noted earlier, both contention scopes are clearly useful: process-scope threads provide high overall performance by minimizing the number of system calls (which are expensive, at least in the context of a traditional Unix—though POSIX' more relaxed requirements sometimes allows faster implementations), while system-scope threads provide reliable performance and predictable latency for its threads by ensuring that their parent processes'

[10] On the assumption that there is already a functioning operating system kernel!

[11] Co-operative hybrid implementations of this idea are, of course, theoretically possible; this would require the user process and the kernel to share an explicitly represented scheduling model. Thread dispatch would be performed within the process using processor allocations *delegated* by the kernel on the basis of the aggregate behavior of the threads. For example, when a process containing round-robin threads was dispatched it would be assigned a time-slice equal to the *sum* of the time-slices of the highest-priority threads within it; it would be trusted to subdivide that time-slice appropriately. The kernel's notion of a process's priority would now be the highest of the priorities of the runnable threads within it; this can be tracked fairly efficiently by the kernel if the process resigns whenever its priority changes according to this measure, reporting the new priority value. The main additional work the kernel needs to do is to maintain the *priority* at which each system call is issued, so that responses can be used to awaken process-local schedulers as necessary.

descheduling does not cause them to be passed over. Why, then, are both optional? Why not require every implementation to provide both scopes?

The argument against requiring system scope—'kernel threads'—is simple: it requires significant modification to the kernel, while (with the possible exception of one small extension to allow true nonblocking file I/O) process-scope threads can be implemented as a simple library that user code is linked with. Conversely, implementations providing only system scope are allowed because system scope is independently useful, is often adequate and sometimes ideal, and many systems already provide it. Allowing such systems to support the pthreads interface and claim POSIX compliance greatly increases the portability of basic threaded applications.

9.b Non-posix analogs

9.b.1 DCE

As with static attributes of objects, DCE scheduling diverges from POSIX principally in providing additional options. In particular, DCE threads provides two further scheduling policies:

`SCHED_FG_NP` Foreground ('`FG`') scheduling is similar to round-robin (`SCHED_RR`⇒$^{9.3.3}_{166}$), except that the dominance between different priority levels is not absolute. A foreground thread defers to other threads at lower priorities, allowing them to run also, but with reduced processor allocations. (Round robin and FIFO threads, if they are also used, continue to dominate lower priority threads completely, however).

`SCHED_BG_NP` Background ('`BG`') threads have behavior similar to foreground threads except that they are *not* scheduled if any non-background thread is ready to run at the same priority.

(The `_NP` names were assigned in deference to the POSIX draft standard on which DCE was based, and relative to which they are an extension.)

DCE also specifies particular semantics for `SCHED_OTHER`; it is, in DCE, always a synonym for `SCHED_FG_NP`.

9.b.2 MACH Cthreads

In MACH Cthreads scheduling is at the mercy of the objects underlying the threads. If you are using the coroutine implementation (`-lcothreads`), *no* pre-emption takes place: threads that do not otherwise synchronize with each other *must* call `cthread_yield()`, the MACH analog of `sched_yield()`$_{172}$, or other threads in the task will not run. In other implementations calling this function is not strictly necessary, but it may be invoked as "a hint to the scheduler".

In order to avoid running out of physical tasks when using the `-ltask_threads` implementation—in MACH tasks are a bounded resource—it may be necessary to call `void cthread_set_limit(int)` to bound the total number of threads active; new threads over this limit must wait until another thread dies before receiving any CPU time. This function is apparently specific to the `-ltask_threads` implementation.

9.b.3 OS/2

Thread scheduling in OS/2 is uniformly round-robin and (like the POSIX priority scheduling options) obeys priorities strictly. Explicit control of priority is via the function `DosSetPriority()`, which has the following arguments:

```
DosSetPriority
  ( ULONG Scope
  , ULONG PriorityClass
  , ULONG PriorityDelta
  , ULONG ID
  );
```

Priorities are divided into four **priority classes** as set by the *PriorityClass* argument; these are `PRTYC_TIMECRITICAL`, `PRTYC_FOREGROUNDSERVER`, `PRTYC_REGULAR`, and `PRTYC_IDLETIME`. Each class dominates those that follow; a thread in a lower priority class will never run if a thread in a higher class is ready.

Thread priorities, which vary from 0 to 31 within each class, can be modified with the *PriorityDelta* argument, which represents a priority *change* of between -31 and 31. If a change in priority but not of class is desired, you can pass the constant `PRTYC_NOCHANGE` as the value of the *PriorityClass* argument.

The priorities assigned by the programmer are not, in OS/2, immutable; various factors will cause the system to 'boost' them, including:

- being in the 'foreground' session, the session having the user-interface focus;
- having recently been blocked in I/O; and
- not having received the processor at all for several seconds (the exact value of 'several' being a user-settable system configuration parameter).

The 'scope' argument of this function refers not to contention scope, but to the domain over which the requested adjustment in priority applies. It should be chosen from among:

`PRTYS_THREAD` Only a single thread is affected, as identified by *ID* (which must be a thread ID).

`PRTYS_PROCESS` The target of the operation is a process; this will affect all threads within it. In this case *ID* must be a *process* ID.

`PRTYS_PROCESSTREE` The target of the operation is a process and *ID* is a process ID, but the modification in priority also applies to all *descendents* of the identified process.

Thread suspension

In addition to the ability to change the priority of a thread, OS/2 provides calls to **suspend** (make unschedulable; block) a thread and to unblock it, allowing it to resume processing. These are the functions

```
DosSuspendThread(TID thread_id);
```

and

```
DosResumeThread(TID thread_id);
```

each of which takes a single argument, the thread to be operated upon.

Chapter 10

Signals

[[But posix.4 RT signals provide queueing and argument passing, and have a defined order of delivery. They provide info about how they were raised. `SIGRTMIN-SIGRTMAX`*]]*

The traditional Unix way of passing synchronizing information from the kernel to a process and from one process to another is the **signal**. In pthreads, signals sent to a process can be received by any thread, while signals raised automatically by the kernel or sent from one thread to another can be handled by a particular thread. Nevertheless, to keep matters simple, all threads that accept a given signal must handle it in the same way.

An extension originating (like priority scheduling) with the POSIX real time extensions optionally introduces user-definable signals with more predictable delivery behaviour than basic signals and the ability to carry a data field along with the signal itself. These might be useful if they are available on your system, though for most purposes pthreads' own synchronization facilties provide both more flexibility and much higher performance, both within and—if you have shared memory—between processes.

10.1 Introduction

If a process is a virtualization of a computer and a thread is a virtualization of a CPU, a signal is a virtualization of an **exception**. At the hardware level, the exception is the de facto standard method for dealing with events of interest to software without needing to insert a lot of explicit checks. At its simplest, exception processing amounts to the hardware implicitly inserting a procedure call at the first legal moment after the exception condition is detected; control transfers to the target of this call, which can then react to the situation being reported. If this **exception handler** was careful about saving the processor state at the beginning of the routine, that state can be restored and the original processing sequence resumed, if desired, as if nothing had happened.[1]

[1]Typical hardware disables externally generated exceptions on entering an exception handler, to avoid recursion problems. Software re-enables them at the point of return from the handler, if not

(In the case that the condition is sufficiently serious that the processor can't continue effectively, the implicitly inserted instruction may be one that shuts the processor down completely, rather than a call.)

This kind of hardware exception handling mechanism was originally developed as a way of dealing with external events—like input arriving from an I/O device—that happen infrequently but require rapid attention. (If they occurred constantly or could afford a longer—perhaps arbitrary—delay before being serviced, it would be more efficient to have software check for them explicitly.) The existence of such an *interrupt* system allows a lot of tasks that would otherwise need specialized hardware to be handled by the main processor, and was in part responsible for the success of the microprocessor.

But the process of taking an interrupt is typically triggered by raising a line on the processor chip; any status signal in the computer can be routed to an interrupt, not just ones involving 'external' events. In particular, it's easy to arrange for things like wild addresses (that don't correspond to physical memory installed in the computer) to trigger interrupt processing.

This results in an interesting situation: the branch to the interrupt handler is now not actually the result of an external event *interrupting* the flow of the program. Rather, the semantics of the instructions being executed has been augmented so that in *exceptional* conditions—such as using an illegal address or, depending on the hardware setup, dividing by zero, a **trap** is taken to a previously arranged function that can deal with the situation.

This is wonderful, because it allows many instructions to behave almost as if they they contain conditional branches that check for cases not fully handled in hardware, even allowing a sensible semantics for these harder cases to be specified in software, without using *any* valuable bits in the instruction encoding to do it.[2]

There remains, however, the possibility of one completely weird effect: what if an external signal should be wired to the *same* hardware interrupt line as is used for such an internally generated trap? If this external signal is asserted, the subroutine that handles the instruction's exceptional case will be invoked, but in a context that is completely meaningless to it. This is almost certainly a bad idea! Oddly enough, though, it is a problem that has been inherited directly by the POSIX signal model.[3],[4]

---

before.

[2] At the time that this development arose, microprocessors were still more or less executing only one instruction at a time—they were not heavily **pipelined**—with the result that address traps and so on could be delivered soon enough to be handled before the next instruction began. Modern processors may be working a dozen instructions ahead of themselves by the time a memory reference has external effects and nothing so naïve would work so well today. But the feature is so useful that hardware designers have gone to a great deal of trouble to get the same effect by other means.

[3] You'd expect that with real hardware this wouldn't happen—who would be stupid enough to use the same line for such distinct uses? It appears, however, that the programmers at Microsoft did not bother to read the note in the documentation of the x86 series of processors that reserves the first 32 interrupt vectors to Intel use, and built an operating system around them. MS-DOS thus uses interrupt vectors that are also employed directly by later versions of the processor hardware; a fair amount of confusion has resulted.

[4] It may be worth noting in passing that exceptions are the basic technology underlying traditional POSIX implementations: system calls (which need to execute at a high level of *privilege*—one aspect of exception handling we skipped in the discussion above is that exception handlers, which after all

Unix translates this hardware concept of a single system covering both asynchronous interrupts and synchronous execution conditions directly into a software facility. Processes, the kernel (acting on behalf of the CPU, as if were) and device drivers send **signals** that can: be handled by immediate context-independent execution of pre-registered functions; be ignored; or cause process exit.

In the presence of threads, a number of new and better options become available for signal handling. Dedicating a thread to handling a given signal, for instance, gives an effect rather like adding a specialized (virtual) coprocessor to the process. We might even arrange for a new thread to 'pop up' to handle an event without going through the traditional signalling mechanism at all—as if a new virtual CPU were being installed just to deal with it.

But POSIX has sought to ensure that the external interface of a piece of software need not change because it exploits multithreading. In particular, it preserves the use of signals for process termination. And unlike other designs (OS/2$\Rightarrow^{10.b.2}_{200}$, *e.g.*) it has not chosen to adapt them to the asynchronous termination of *threads;* that function is performed by the separate pthread cancelation facility.$\Rightarrow^{11}_{203}$

Thus the biggest challenge faced by the POSIX committee in adapting signals to the multithreaded environment has been designing a scheme that routes signals to the threads that handle them, even though some signals are generated for particular threads while others are sent to the process as a whole, and despite consequences that range from calling a function in the local context of a thread to instant process termination.

10.2 The basic posix signal model

The basic signal model that POSIX has adopted for single thread processes is a synthesis of the approaches developed in the Berkeley and AT&T worlds. A small, fixed number of signals is provided (no longer assumed to be less than the number of bits in a word) and assigned names reflecting their conventional uses, including the circumstances under which they are generated by the kernel itself. These uses include both **synchronous** exceptions (corresponding to traps generated directly by the execution of the instruction stream, such as `SIGFPE`—instruction-level semantic error ['floating point exception'] and `SIGSEGV`—invalid address reference ['segmentation violation']) and **asynchronous** ones (which, like hardware interrupts, are due to external causes—either hardware *or* software—like `SIGINT`—interrupt key pressed, `SIGCHLD`—child process stoppage or termination, and `SIGALRM`—expiry of a programmed timeout[5]). But

execute known code on behalf of the hardware, are generally given less restricted access to hardware facilities than normal user code, including the ability to tweak processor modes, switch tasks, and so on) are typically implemented through *trap instructions*—instructions with no purpose other than to raise an exception. Timeslicing—periodic entry into the kernel independent of basic software activity, used to switch between processes or threads—is likewise implemented with a hardware timer that delivers periodic interrupts.

[5]Actually `SIGALRM` was the subject of some discussion. Since a process requests that an alarm be delivered to *itself*—and indeed at a pre-planned time—there is some case to be made that it isn't *as* 'asynchronous' as other cases. Furthermore, there is a well-established programming style in which `alarm()` is used to interrupt possibly long-running system calls, which is the business of only a single

user software is free to generate *any* signal asynchronously, using `kill()` (though there are security restrictions on which processes can send signals to each other).

What happens when a signal is delivered to a process depends on a number of factors. In two cases, the effect is completely determined by the signal; `SIGKILL` immediately terminates the receiving process, and `SIGSTOP` unconditionally suspends it. The remaining signals are more flexible. Depending on the signal number and its conventional meaning, their default effect is to make the receiving process to terminate abruptly; to dump core; to suspend; or to resume; or else they will just be ignored. With `signal()` or (preferably) `sigaction()`, the response can be changed to totally ignoring the signal, or to calling a user-specified signal-handling routine. The functions controlling signal disposition manipulate the entries of the so-called **signal vector** table.[6]

A process that may receive signals can (except in the case of `SIGKILL` and `SIGSTOP`, and except for 'ignored' signals, which are discarded immediately) also arrange for their delivery to be postponed. This can be used to prevent low-priority signals from interrupting important operations; but far more importantly it allows a process that is already handling one signal to avoid being interrupted by another (or worse yet, a recurrence of the first). This is accomplished by providing the process with a **signal mask**, indexed by signal number, which temporarily blocks signal delivery; unless you are using pthreads (which as we shall shortly see exclusively uses a new function, `pthread_sigmask()`193), this mask can be manipulated with `sigprocmask()`. When a non-default signal handler is registered using `sigaction()`, the programmer specifies (in the `sa_mask` field of the `struct sigaction` argument) a further mask of signals that should be blocked *just while the signal handler is executing*. A process can check which signals are *pending*—generated but, due to blocking, not yet delivered—by calling `sigpending()`. Alternatively, it can use `sigsuspend()` to go to sleep with a temporarily modified signal mask; an incoming signal will then awaken it.

(Normally a signal handler either returns when it has done its job, or forces the process to exit. Using `siglongjmp()` it is also possible for a signal handler to return to a saved computational state other than that in effect when the signal arrived, analogously to using `longjmp()` to exit several levels of function invocation at once.)

As we shall see, the importance of the notion of a signal mask has been extended even further in the multithreaded model.

---

thread. Nonetheless, since the `alarm()` timer has been made a process-global resource, since there are other, more tightly controllable methods of awakening (or terminating) a long-blocked thread available in the multithreaded environment, and since alarm delivery *is* technically asynchronous to the undefined rate of progress of a given thread of control, it was decided to classify the delivery of `SIGALRM` by `alarm()` as asynchronous.

[6] The terminology is once again borrowed from the hardware level: hardware engineers have long used the term *vectored interrupts* for a system where individual interrupt sources can transfer control ('vector') to different addresses, as opposed to an arrangement where all interrupts are directed to a single handler and software must do the dispatch.

10.3 The posix compromise for thread signals

How should this model be adapted to a situation with multiple threads of control? The POSIX committee considered several options:

1. Switch to a completely new model in which the threads of a process are *never* 'interrupted' by incoming signals. Instead, either require the programmer to designate a thread that *waits* for incoming signals, or have the system generate *new* threads to handle signals as they come in. This exploits the ability of multithreading to synchronize with the asynchronous, and would certainly be a fine idea for a new design. Of course, it is not remotely compatible with existing code or current practise.

2. Keep the system as it is. The whole signal mechanism is still global to the process. Presumably signals are delivered to only one thread, perhaps the process's primal thread. Some kind of thread-related mechanism might be built on top of this. In some ways this is simple, but in practise it would probably require a number of weird restrictions to keep things consistent.

3. Keep the current system, but interpret signaling as a per-*thread*, not a per-process, phenomenon. This raises further questions about, for example, whether signals associated with the process should be sent to one arbitrary thread or to all threads that are not blocking them.

The compromise finally selected maintains all the individual elements of the original single thread design, organizing them in a way that hopefully maximizes efficiency. It was decided that there would be, in pthreads, a *single* table of signal vectors shared by all threads in the process, and that termination, suspension and resumption would continue to apply to the whole process. Each thread, however, gets a *separate* signal mask (manipulated with the new function `pthread_sigmask()`$_{193}$). Each signal is delivered only *once:* if it is associated with a particular thread, it is delivered to that thread; otherwise, a target is selected arbitrarily from the threads not blocking the signal. In particular, *synchronous* exceptions are always delivered to the same threads that caused them to be generated. The following signals are considered to be synchronous *when they are generated by the system:*

```
SIGILL SIGSEGV SIGFPE SIGBUS SIGPIPE SIGABRT
```

A new function `pthread_kill()`$_{190}$ is also introduced to allow one thread to target a signal at another programmatically; recall that thread IDs are private to a process, so no privileges are checked. Signals sent by `pthread_kill()` or by the old `kill()` function are always considered asynchronous, regardless of signal number.

One important change partially supports the radically synchronous style of the first proposal above: the addition of a function `sigwait()`$_{195}$ that allows a thread to block until a signal is delivered and then continue processing normally.

In the following sections we examine the implications of this model.

10.4 Signal generation

Signals in POSIX arise from a number of sources. They may be:

- Terminal-generated, when the user strikes an interrupt key;
- Generated explicitly by a call of kill(2) [or kill(1)];
- Sent by a process to itself, with `raise()` or `abort()`;
- Hardware exceptions (seen by software after reflection back from the kernel); or
- Produced within the kernel in response to a software event. The signals `SIGALRM` and `SIGPIPE` are of this sort.

With pthreads these signal sources are now be classified according to whether they send a signal to the process in general (in which case any of the threads in the process is potentially eligible to receive it), or to one thread in particular.

- Signals coming from outside the process and directed at the whole process[7]
  - Terminal-generated signals
  - Signals generated explicitly by `kill()`
  - Signals generated by the kernel for the whole process
- Signals coming from within the process and directed at a particular thread
  - Hardware exceptions caused by the thread
  - Signals sent by the thread to itself
  - Signals generated for a given thread by the new function `pthread_kill()`$_{190}$.
  - Signals generated by the kernel for a particular thread

As you can see, signals generated by the kernel have been split between the two categories. For example, when a timer times out the system sends `SIGALRM` to the entire process, but when an attempt is made to write to a pipe that no longer has a reader, the ensuing `SIGPIPE` is sent directly to the writing thread.

The complete list of (standard) POSIX signals is given in figure 10.1; the ones that the kernel generates for a particular thread are marked with an asterisk.

[7]These are handled this way as a consequence of the fact that threads have no names that are meaningful outside of their containing processes; thus, any signal that does not come from either the kernel or the process itself cannot name a target that is a thread, anyway.

signal	default action	automatically raised by	when
SIGQUIT	coredump	terminal driver	Quit key pressed
SIGINT	terminate	terminal driver	Interrupt key pressed
SIGTERM	terminate	(software only)	Process termination requested
SIGKILL	† terminate	(software only)	Process termination required
* SIGILL	coredump	processor	Invalid instruction executed
* SIGSEGV	coredump	processor	Invalid address referenced
* SIGFPE	coredump	processor	Invalid (floating point) operation
* SIGBUS	coredump	processor	Implementation defined hardware fault
* SIGABRT	coredump	`abort()`	*(Requested by software)*
SIGTSTP	stop	terminal driver	Process stop requested
SIGSTOP	† stop	job control	Process stop required
SIGCONT	continue	job control	Process should be continued
SIGTTIN	stop	terminal driver	Read from terminal while backgrounded
SIGTTOU	stop	terminal driver	Write to terminal while backgrounded
SIGALRM	terminate	`alarm()`	Timer expires
* SIGPIPE	terminate	kernel	Receiver of write has terminated
SIGCHLD	ignore	kernel	A child process stops or terminates
SIGHUP	terminate	kernel	Line hangup ⇒ session leader; Session leader termination ⇒ members. Conventionally used to 'kick' daemons
SIGUSR1	terminate	(software only)	*(Requested by software)*
SIGUSR2	terminate	(software only)	*(Requested by software)*

* Synchronous (hence thread-specific) when raised by the system.
† This action cannot be overridden.
Italicized signals are required by POSIX only when job control is provided.

Figure 10.1: Standard POSIX signals

When interpreting this table it is important to realize that whether a signal is generated for a thread or for the process ultimately depends on whether it was generated (explicitly or implicitly) with `kill()` or with `pthread_kill()`. If the signal originated in the kernel, this in turn depends on whether that signal is deemed to be 'synchronous' with respect to some thread, the result of the execution of the current instruction; otherwise it depends only on software structure—not on the signal number at all. This is a charming silliness that POSIX has inherited from its Unix forebears and from hardware before that. Its consequence is that although a floating point exception is transparently a private event resulting from the (mis-)execution of a program, one process is quite free to explicitly send a `SIGFPE` to another—though mayhem will typically result. Setting aside questions of access control, the problem arises from an earlier conflation of synchronous and asynchronous phenomena that the POSIX committee was not free to repair, though pthreads has at least forced it to be documented properly.[8] In the presence of multiple threads—which are above all a tool for providing synchronous treatment of asynchronous events—the consequences of this traditional confusion verge on the ludicrous.

10.4.1 Sending signals explicitly

As always in Unix, software can send any signal explicitly. With pthreads a signal can be sent either to any process on the computer (subject to the sender having appropriate privileges), or to any thread within the current process—including, in each case, the sender. This is done by calling `kill()` or `pthread_kill()` (respectively), passing it the identifier of the object to be signaled and the number of the signal to be raised.

(There is, of course, no shell command for signaling threads, since threads are identified only by handles private to a process.)

`kill()`, `pthread_kill()` **3.3.11**

```
#include <sys/types.h>                          /* kill() only */
#include <signal.h>

int kill(pid_t process, int /*signal*/ signal);

int /*errno*/ pthread_kill
  (pthread_t thread, int /*signal*/ signal);
```

`pid_t` *process*
  Process (if > 0) or process group (if ≤ 0) to receive signal.

`pthread_t`$_{32}$ *thread*
  Thread to receive signal.

`int /*signal*/` *signal*
  Signal to raise in recipient.

[8] The division into synchronous and asynchronous signals is not in fact perfect. Many architectures, for example, perform their floating point calculations in a coprocessor that is only loosely synchronized with main processor—almost as if it were another thread! This means that floating point exceptions, while *arising* from synchronous program execution, may nonetheless be *delivered* at an unrelated later moment. This is typically dealt with by having exceptions record and report two sets of state, one for the main processor at whatever point it has reached in the integer instruction stream, and one relating just to the state of the floating point unit itself. This allows for recovery from floating point traps, but, for example, makes it impossible to use floating point exceptions to extend the semantics of the instruction set beyond tweaking the instruction's normal results. *Truly* synchronous exceptions (in the sense that the processor takes trouble to relate the exception to a single point in the instruction stream, regardless of how physical processing might really be sequenced), on the other hand, can be used to emulate significant new functionality, including things like branching behavior or the manipulation of extended data types with different physical representations from those normally processed by the instructions.

Send a signal to a process or a thread, respectively. A signal sent to a process having non-default handling for that signal will be received by exactly *one* thread, arbitrarily selected from those that are (or failing that, become) eligible. The default behaviors all affect the process as a whole, even if they are received by a particular thread.

In the case of `kill()`, a signal can be sent to all members of a given process group for which the caller has permission. If *`process`* < 0 the target process group is $|$*`process`*$|$; if it is 0, then it is the caller's process group.

On success both functions return zero.

On failure `kill()` returns `-1` and sets `errno`, while `pthread_kill()` returns an error code:

`ESRCH`$_{271}$ The *`thread`* or *`process`* argument is incorrect.

`EINVAL`$_{271}$ The *`signal`* number is invalid or unsupported by this implementation.

`EPERM`$_{271}$ *(`kill()` only)* The calling process lacks the privileges necessary to deliver the signal.

10.4.2 Other directly-signaling functions

POSIX and Standard C provide a number of other methods to send signals that are (in POSIX) logically built on top of the `kill()`$_{190}$ interface. One of these, `raise()`, was intended to allow a program to send a signal to itself (indeed, this is the only place signals show up in Standard C itself). This function has been interpreted by the committee to behave in the following way in a multithreaded environment:

```
int raise(int signal)
  {return pthread_kill(pthread_self(), signal);}
```

The `abort()` function, typically used by a process that wishes to commit suicide, depends in turn on this interpretation; it is now sent to the calling thread itself, though its handling will typically affect the whole process. It now behaves approximately as if it executed `raise(SIGABORT)` (though it also has other duties including, for instance, explicitly ensuring that it is not blocked or ignored).

10.5 Handling signals

If nothing is done to specify how a signal is to be processed within a program, the default action for that signal will apply. Examination of figure 10.1 shows that in most cases this is process termination, often with a core file; this is true even if the signal is directed to a particular thread. There are basically four things that can be done to modify this behavior.

1. The signal can be *ignored*, by using `signal()` or `sigaction()` to set the action associated with the signal to `SIG_IGN`.[9]

[9]This differs from just installing a user signal handler that does nothing, since ignored signals may be discarded even when they are masked, while at least one instance of a handled signal will be remembered ('held pending') until the signal is re-un-masked.

2. A user-supplied **handler** can be installed for the signal, again with `signal()` or `sigaction()`. This is a function that is called when the signal is received, allowing the programmer to take any action that may be necessary to recover, and typically returning to the original point of interruption.[10]

3. The signal can be temporarily **blocked** by setting the appropriate bit in the thread's **signal mask** with `pthread_sigmask()`$_{193}$.

4. The signal can be handled *synchronously* using `sigwait()`$_{195}$ to arrange for a thread to await its delivery.

(The behavior on signal reception can of course be reset to the original default by giving an action of `SIG_DFL`.)

Signal handlers have not changed at all with the advent of pthreads; the process still has a single set of signal vectors, and any thread that receives a (handled, unmasked) signal will abruptly find itself executing the signal handling function registered there.[11] Indeed, so long as all threads but the primal one mask out all of the asynchronous signals, applications may not detect any change from the single-threaded POSIX semantics at all (actually there *is* a difficulty, since the committee has chosen to make `sigprocmask()` undefined for multithreaded programs. We can hope that most vendors will cause it *either* to be identical to `pthread_sigmask()`$_{193}$ *or* to manipulate a process-level signal mask, either of which would have the necessary effect). The use of signal masks has changed somewhat, however, since each thread has been given its own mask; and the option of synchronizing a signal to a thread is, of course, entirely new.

Finally, since it makes no sense to transfer control *between* threads, `siglongjmp()` can only transfer control to a state that was saved *by the calling thread*. This may require a little additional care.

---

[10] Unlike some other versions of Unix, POSIX signal handlers remain installed even after the signal they handle has been received. Recursive invocation of the same handler is instead avoided by masking the current signal until the handler returns.

[11] Since signal handlers are essentially executed asynchronously from the main text of the thread that is running them, they are subject to constraints about manipulation of shared data and the calling of library functions similar to those placed upon threads. The restriction on available functions is (naturally enough) that they be re-entrant.$\Rightarrow^{12}_{219}$ The restriction imposed on signal handlers for data synchronization is, however, rather different from that imposed on threads: they may only manipulate data shared with the rest of the program if it belongs to the (integral) type `sig_atomic_t`. This is an older idea for synchronization originating with Standard C; objects of this type are supposed to be manipulated by the hardware in such a way that they are *implicitly* locked against interference by other threads of control.

One might think that it would be logical to move over to using mutexes for this function as well, but there would be a problem: there would still be no way for a signal handler to communicate with the thread on whose behalf it was running! The difficulty is that in order to synchronize with its non-signal-handling alter-ego, it would have to acquire the mutex protecting any shared data; but if the other 'identity'—the normal flow of execution—already had it locked, immediate deadlock would ensue: a single thread may not lock the same mutex twice.

10.5.1 Per-thread signal masks

In order to provide some measure of control over how threads receive signals, POSIX has given each thread its own signal mask. This allows each thread to determine independently whether it wishes to receive a given signal, regardless of whether that signal was generated for the process or for the thread alone. The fact that signals sent to the process can be received by *any* available thread means that signal masking now goes beyond its original brief of delaying signal delivery until a convenient time: by masking a signal in one thread that is unmasked in another, you can ensure that the second thread effectively handles all such signals incoming at the process level without the first being interrupted. As we shall see in the next section, an even more elegant way of handling process signals is to mask them in *all* threads and then have a signal-server thread use the `sigwait()`$_{195}$ call to field incoming signals synchronously.

As with single-thread programs, masked signals are not dropped; they are stored for later delivery. If they were generated for a thread they will be delivered if and when that thread unmasks them; if for a process, they will be delivered once *any* thread does so.

When a thread is first created, its signal mask is copied from its parent (one of the rare cases where POSIX threads show any signs of ancestry) and its set of (private) pending signals is cleared. To modify its signal mask it can call `pthread_sigmask()`, which, aside from applying to only the calling thread and obeying the pthread-style error reporting mechanism, is just like the `sigprocmask()` function that it replaces.

`pthread_sigmask()` **3.3.10**

```
#include <signal.h>

int /*errno*/ pthread_sigmask
  ( int /*sigmask_mode*/ how
  , const sigset_t *new_sigset
  , sigset_t *old_sigset
  );
```

`int /*sigmask_mode*/ how`
: A value chosen from among those listed below, specifying how the signal mask of the calling thread is to be modified. Ignored if *`new_sigset`* is `NULL`.

`const sigset_t *new_sigset`
: A set of signals to be combined with the thread's current mask as specified by *`how`*.

`sigset_t *old_sigset`
: A pointer to a variable receiving the old value of the thread's signal mask. If `NULL` this information is discarded.

The *`how`* argument must be one of the following:

`SIG_BLOCK` The signals in ***new_sigset*** are to be added to those blocked for the current thread.

`SIG_UNBLOCK` The signals in ***new_sigset*** are to be unblocked for the calling thread.

`SIG_SETMASK` This thread's signal mask should be set equal to ***new_sigset***.

If ***old_sigset*** is non-`NULL`, the area it points to is updated with the current value of the calling thread's signal mask. Then, if ***new_sigset*** is non-`NULL`, the mask is updated with the information pointed to by ***new_sigset***, according to the rule ***how***. If this unblocks any signals *pending*$_{186}$ for either the thread or the process, at least one of them will be delivered before `pthread_sigmask()` returns.

On success returns zero.

On failure returns a status code:

`EINVAL`$_{271}$ The value of ***how*** was invalid.

(Portable) multithreaded programs should not call `sigprocmask()`; its behavior is not defined for them (presumably to allow implementations the freedom to decide whether or not to implement a per-process signal mask in addition to the per-thread masks).

Because signal masks are inherited from a thread's creator, there is a risk that signals intended for the parent thread will be delivered to the child. Therefore the following protocol should be followed: before creating a thread, set the creator's signal mask to no less than the *union* of its current mask and the intended mask of the child—masking out any signals for which there is competition. After the call to `pthread_create()`$_{33}$ returns, the parent can restore its mask to the original value. The child can adopt its own permanent signal mask value in its startup code:

```
sigset_t childMask, tempMask;
pthread_t child;

void *childCode(void *data)
  { pthread_sigmask(SIG_SETMASK, &childMask, NULL);
    ...
  }
```

inside parent code

```
    ...
    /* set up childMask */
    ...
    pthread_sigmask(SIG_BLOCK, &childMask, &tempMask);
    pthread_create(&child, attributes, childCode, arguments);
    pthread_sigmask(SIG_SETMASK, &tempMask, NULL);
    ...
```

(though this code can't share the `childMask` between multiple new threads without adding synchronization code). Matters would have been greatly simplified if a thread creation attribute had been provided to set a thread's initial signal mask, but it wasn't.

In fact, though, this technique is rarely needed in new code, due to the superior techniques (in which most signals are masked most of the time) made possible by the *synchronous* signal-handling method described in the next section.

10.5.2 Handling signals synchronously

One of the two main motivations for multithreading (the other being the exploitation of available parallelism) is to render the asynchronous synchronous by allowing subcomputations to progress at the same rate and in a parallel manner to the external phenomena they relate to. This applies as much to signals as it does to other, higher-bandwidth communications channels.

Actually, POSIX has always allowed a process to sleep until a signal arrives, with the functions `pause()` (which puts a thread to sleep until a signal is caught) and `sigsuspend()` (which additionally allows the temporary installation of a modified—typically more liberal—signal mask). Both of these functions, however, leave the actual processing of the incoming signal to the installed signal handler, which is still considered ‘asynchronous’ to the handling thread’s everyday life. What is wanted for synchronization with the threads world is a function that suspends a thread until a signal comes in, and then allows the thread to process it normally, in its primary flow of control.

The new function `sigwait()` does this.

The thread calling `sigwait()` registers a set of (currently masked) signals that it awaits, and is suspended. When one of those signals is raised, the thread is reawakened, informed of which signal has arrived, and allowed to proceed normally. The signal is then deemed to have been handled—without the use of the usual asynchronous signal handlers. Indeed, from the perspective of the traditional signal handling mechanisms, the signal can be thought of as having been ‘masked’ the whole time, and delivered only through a ‘loophole.’

Note that in order to leave the implementation the fullest possible latitude in implementing the signal system, you cannot use both `sigwait()` and `sigaction()` on the same signal at the same time.

`sigwait()` **3.3.10**

```
#include <signal.h>

int /*errno*/ sigwait
  (const sigset_t *sigset, int /*signal*/ *signal);
```

`sigset_t *sigset`
: A set of signals the calling thread is willing to handle synchronously. These signals must already be blocked [[**sps: check**]] in all threads.⇒10.2 186

`int /*signal*/ *signal`
: Pointer to a location that will be filled in with the identity of the signal that was actually delivered.

If any of the signals specified in *`sigset`* is *pending*, it is cleared and its number returned in *`*signal`*. Otherwise, the calling thread blocks until one of the signals in *`sigset`* is raised, and proceeds similarly. If several threads are `sigwait()`ing for the same signal at the same time, only one of them, chosen arbitrarily, will receive it.

Note that the signals to be received in this way must first be blocked, but that `sigwait()` provides a way to receive them regardless.

The effect `sigwait()` has on a handler installed by `sigaction()` is undefined.

On success returns zero.

On failure returns an error code:

> `EINVAL`$_{271}$ *`sigset`* contained an invalid or unsupported signal.

Note that although this function was introduced as part of the work on multithreading, it is not a pthreads function. It is available in POSIX regardless of whether threads are supported or being used. Generalised versions, `sigwaitinfo()` and `sigtimedwait()` are available under the realtime signals extension.

The function `sigpending()`, which checks to see which signals are outstanding, still exists and is useful under pthreads, but its result is now (naturally enough) relativized to the calling thread. That is, it returns the list of signals that are pending *either* for the calling thread, *or* for the process as a whole.

This does introduce a new complication: in a traditional, single thread process, signals could not leave the pending set unless they were either ignored or delivered. Thus, calling `sigpending()` would result in a mask that could safely be used in a subsequent call to `sigsuspend()`, without danger of the process blocking forever. With pthreads, however, another thread might unblock and accept delivery of a signal pending for the process, between the moment `sigpending()` returns and the time the signal set it returns is used. From the perspective of the first thread, a signal reported pending has just vanished without trace.

In exploring the relationship between `sigsuspend()` and `sigwait()` it is instructive to try to code the former in terms of the latter. For simplicity, we will assume an implementation in which—though this extends standard behaviour—signal handlers may be installed on awaited signals provided those signals are masked.

Assuming we have a function `sigmask_invert()` that negates the bits of a `sigset_t`, we can write:

```
#include <signal.h>
#include <pthread.h>

int sigsuspend(const sigset_t *sigmask)
  { int sig, ret;
    sigset_t saved_mask, temp_mask;
```

```
  /* If we mask out all signals and then let sigwait()
     pick up those we anticipate, the effect will be like
     setting the mask to sigmask, as required.
  */
  pthread_sigmask(SIG_BLOCK, sigmask_invert(sigmask), &saved_mask);

  /* Wait for something in sigmask */
  ret = sigwait(sigmask, &sig);

  if (ret)
    errno = ret;
  else
    { /* Re-enqueue the signal */
      sigemptyset(&temp_mask);
      sigaddset(&temp_mask, sig);
      pthread_sigmask(SIG_UNBLOCK, &temp_mask, NULL);
      pthread_kill(pthread_self(), sig);

      /* Set the mask back to its "official" value.
         This will let sig (or possibly some other signal
         in sigmask) through.
      */
      pthread_sigmask(SIG_SETMASK, temp_mask, NULL);
      errno = EINTR;
    }
  pthread_sigmask(SIG_SETMASK, &saved_mask, NULL);
  return -1;
}
```

Actually this implementation has two bugs (aside from the assumption mentioned above): first, if multiple signals listed in `sigmask` arrive in succession, more than one of them may be delivered even though they were not in the signal mask in effect at the time our version of `sigsuspend()` was called. That is, to be strictly correct, all signals should be masked between the last two occurrences of `pthread_sigmask()`$_{193}$. Second, `sigsuspend()` is supposed to ignore non-fatal, non-caught exceptions: that is, our function should remove from `sigmask` any signals whose disposition is currently set to `SIG_IGN`, and those signals listed in figure 10.1 as job-control related (and non-fatal by default) whose dispositions are set to `SIG_DFL`.

The latter problem is easy (if tedious) to address; the former, a race condition, much harder.

10.6 Important points

This section contains a short list of signal-related phenomena to bear in mind.

- In pthreads there is both a set of pending signals for each thread, *and* a pending set for the process as a whole. The status of a thread with respect to the signal system is determined by the union of the two sets.

- When a signal is sent to a process, it is forwarded to the first thread able to accept it (with ties resolved in a system-dependent manner). At the moment it is delivered to a thread it ceases to be pending for the process.

- No matter how delivery takes place, and no matter how many threads might be eligible to receive it, a signal is delivered at most once. If, however, a single signal is raised more than once while it is masked (causing the pending instances of the signal to 'pile up'), the only guarantee is that it will be raised *at least* once when (through unmasking or a call to `sigwait()`$_{195}$) it becomes deliverable. That is, occurrences of masked signals after the first may be 'dropped'*[[unless you're working with queued signals]].*

- Signals with a given number may be delivered either to the process as a whole or to particular threads, depending on how they were generated. Conversely, even if a signal is earmarked for a single thread, its *default* action (termination, stoppage, *etc.*) typically affects the process *as a whole.*

- As noted in the last section, synchronization problems make `sigpending()` an unreliable source of information about signals delivered to the process but handleable by multiple threads.

- Low priority threads may be put in the position of receiving signals that they never receive enough cycles to process. If a higher priority thread subsequently cancels the receiver$\Rightarrow^{11}_{203}$, the effect may be that the signal is permanently lost.

10.7 Popup Threads—avoiding signals with `SIGEV_THREAD`

A number of recently introduced functions—the `aio_` family of asynchronous I/O operations (present if `_POSIX_ASYNCHRONOUS_IO`$_{238}$ is defined), and `mq_notify()` (which depends on `_POSIX_MESSAGE_PASSING`) use the type `struct sigevent` from `signal.h` to specify how the caller should be notified when an asynchronous event comes to pass; the events in question are the arrival of new data or the completion of an aio operation. The original specification of this structure allowed the controlling field `sigev_notify` to have the values `SIGEV_NONE`, meaning that no special action should be taken, and `SIGEV_SIGNAL`, which requests a `sigqueue()` using the structure's `sigev_signo` and `sigev_value` fields for the details.

Pthreads adds the further option of completely avoiding asynchrony by requesting a new 'pop-up' thread to be created to receive the notification. This is done by giving a `sigev_notify` of `SIGEV_THREAD`. The initial function of this new thread comes from the (new) `sigevent` field `sigev_notify_function` and its attributes are set from `sigev_notify_attributes`. The initial function's argument is `sigev_value` (and in fact has type `union sigval` instead of the usual `void *`).

The new notification mechanism may be convenient because it is in effect completely synchronous and does not require pre-allocation of the handling thread; it's a little bit sad that a similar mechanism has not been made available to handle signals in general.

One small detail to bear in mind when designing a program that uses this feature is that for the process to stay alive it must have at least one thread. Thus a style in which the consequences of an I/O operation are 'handed off' to the pop-up thread receiving completion notification will not work properly if there is any chance that all other threads will exit before it is actually created. You may have to find some busy-work for some other thread to do in the mean time (though of course there is no reason that this 'work' should involve more than blocking).

10.8 Examples

`sigwait_multiple()`?

10.9 Summary

In adapting the Unix signal system to multiple threads, pthreads as chosen to interpret the signal *mask* as a thread-specific resource, but to keep the signal *vectors* global to the process. This means that a thread can dictate which signals it is willing to accept (and when), but, while the traditional method of handling signals continues to work, all threads must use the same handling function.

This is an inconvenience when handling *synchronous* signals—ones that are an immediate consequence of the receiving thread's execution—but for *asynchronous* signals there is a better solution available with the `sigwait()`[195] functions; these allow a thread to wait for a signal to be raised, synchronizing with it, and handle it in their normal flow of control. This is a vast improvement, since the thread then has full access to shared state without fear of the auto-deadlock that would arise if a signal handler tried to acquire a mutex that the handling thread already owned at the moment the signal was delivered.

Finally, where notification uses `struct sigevent` to specify delivery details, as with asynchronous I/O and POSIX message queues, pthreads adds the possiblity of handling asynchronous events *directly,* using `SIGEV_THREAD`[198] mode. This creates a new handling thread automatically when the condition is detected, potentially bypassing the whole signal mechanism.

Appendices

10.a Under the hood

In a user-level pthreads implementation like CAPthreads, the signal handling logic is probably the most complex part of the system. This is not only because of the need to demultiplex incoming signals between the various threads, but because the thread library itself needs to use the signal facilities itself to implement features like round-robin scheduling⇒9.3.4 [167] and `pthread_cond_timedwait()`[83].

[[It's also necessary for the kernel to provide a siginfo structure so that the implementation can distinguish between synchronous and asynchronous signals! This is another of the small number of cases where the threads package may require direct kernel support.]]

[[@@@@extensive description goes here@@@@]]

10.a.1 Competing handlers

The standard does not specify the interaction between thread-signal synchronization with `sigwait()`$_{195}$ and conventional delivery for the same signal. There is no reason in principle why such redundant handling should be outlawed, but there is a real conflict between the principles of behavior for priority-scheduled threads if it is allowed. The problem is that when a signal is delivered to the process, any willing thread can receive it. In principle, then, any one thread waiting for it becomes runnable; since,under standard priority scheduling, the highest priority runnable thread always runs, we would expect that the highest priority `sigwait()`er will get delivery.

At the same time, it also seems logical (according to the traditional view of signal delivery) that the *current* thread should accept the signal, asynchronously, if it has it unmasked and a handler is installed. This approach is also considerably more efficient.

The simplest solution for the programmer is probably always to resolve such contests in favor of the highest priority thread that can handle the incoming signal by any method. The exact converse is true for the implementor. Thus the standard is silent.

10.b Other Systems

10.b.1 DCE

[[`sigwait()` *allows a thread to wait for a signal.* `sigaction()` *"can specify the handling of synchronous signals within a particular thread" [Lockhart,214 - what does this MEAN?]. Otherwise, signal delivery is only to whole processes (and perhaps no mechanisms other than the above should be used?)]]*

10.b.2 OS/2

OS/2 distinguishes 'system exceptions' from 'signal exceptions'; these correspond approximately to our classification into synchronous and asynchronous exceptions. As in POSIX, system exceptions are delivered directly to the thread that triggered them, though if they are uncaught the default action is typically to terminate only the excepting thread.

Signal exceptions, on the other hand, are always reported to the primal thread of a process. *Which* process they are reported to can, however, be controlled. Signal exceptions are potentially deliverable to any of the children of the signaled process, and `DosSetSignalExceptionFocus()` can be used to determine where they are actually sent.

OS/2 does not provide vectored exceptions; instead, each thread has a chain of registered exception handlers which are offered an incoming exception in succession. Each one may either handle the exception or return it for the next handler in the chain to try. Similarly, there is no fine-grained notion of signal masking. A single pair of functions, `DosEnterMustComplete()` and `DosExitMustComplete()` allow the reception of signal exceptions to be disabled temporarily.

10.b.3 Plan 9

In Plan 9, the entire Unix signal mechanism has been replaced by the "note" system. This new facility, while greatly simplified, maintains the confusion between synchronous, often hardware-specific traps and asynchronous, application-specific interrupts. The system works like this:

Each process p (recall that in Plan 9 there is not much to choose between a process and a thread) is associated with a note file, `/proc/`p`/note`. A note is sent to a process just by writing its text into this 'file'. The function `postnote()` provides a convenient method for doing this; it takes as arguments a target process ID (or a negated group ID) and a string holding the note text. If a process triggers a synchronous exception, the operating system uses the same means to send it a note beginning with the characters '`sys: `'.

A process also has a list (initially empty) of **notification handlers** that it has registered by calling `atnotify()`. These handlers are functions that take the text of a note and a machine-dependent processor-state object as arguments. When a process receives a note, it is interrupted and made to execute each of its notification handlers in turn until one of them returns non-zero—at which point normal processing resumes (except that a system call in progress will be interrupted)—or contrives to continue processing directly by using `notejmp()` and the machine-specific state to recover from the exception. If all the handlers return zero, the process is either terminated (if the note was posted from outside) or suspended in the '*Broken*' state (if the exception was synchronous) for your debugging pleasure.

There is also a note file `/proc/`m`/notepg` provided for each process group that allows a program to broadcast a note to all members of the group; this is actually how `postnote()` handles the case of a negative argument.

This scheme appears to provide everything that is necessary of a signal handling mechanism without the complexity of the Unix model—at the expense of having the programmer explicitly handle all conditions that should not be fatal—including ignoring and (the simulation of) masking—explicitly. Of course, things are radically simplified by the fact that Plan 9 merges processes with threads: there is no decision to be made about *who* receives an incoming note.

[[

---

*Made heavy use of [**?**] while writing this.]]*

Chapter 11

Cancelation

11.1 Introduction

The ability to actually **kill** a process when its purpose is served (as opposed to merely signaling to it) allows programs to be written without explicit checks for user termination requests, and typically works even when the process is blocked within a system call. Of course, the ability to restrict the times at which signals are received—to avoid leaving shared resources in an inconsistent state—is also essential. While slightly at odds with the multithreaded philosophy (which after all seeks to eliminate asynchrony wherever possible), many applications of multithreading also benefit from the ability to terminate threads from without; typical examples include the interruption of lengthy data-retrieval operations, the cancelation of speculative computation, or termination of a branch of a parallel search that is known to have been bettered.

The designers of the POSIX threads system thus provided a mechanism whereby threads can **cancel** one another—can, that is, force them to exit. Internally cancelation works much like normal, 'voluntary' thread exit, except that the canceled thread has no opportunity to specify a final value. A special token, `PTHREAD_CANCELED`$_{204}$, is automatically supplied instead.

The problem of tidying up after a terminated thread—handled for processes by the same protection mechanisms that prevent processes from interfering with each other—is addressed, at least if the programmer is careful, by the cleanup mechanisms described in chapter 6. A more difficult problem is posed by the mismatch between cancelation, whose utility lies in its asynchrony, and multithreading, whose power derives from multiple flows of control synchronizing individually with whatever phenomena they model or manage. That is, by moving the asynchrony in the system into the gaps between the threads, as it were, and performing explicit synchronization where necessary, each thread can proceed locally as if asynchronous events were something that only happened to other people. Meanwhile, cancelation has the opposite character: one thread uses it to zap another with a bolt from out of the synchronous blue sky.

This produces two basic classes of difficulty; the first, that of defining and controlling the points at which cancelation may take place; and the second, that of eliminating race conditions between a canceling thread and a target that may terminate on its own. Indeed, except in the special cases of threads that *only* terminate through cancelation and threads that join⇒$^{4.5}_{40}$ with their own canceler, safe cancelation generally requires additional synchronization, greatly increasing the complexity of some of its superficially simple applications. We discuss these two problems later in the chapter, after we have presented the cancelation facilities themselves.⇒$^{11.3.2}_{206}$, $^{11.4.2}_{212}$

That said, the cancelation mechanism has one significant benefit shared with many built-in facilities of bounded wonderfulness: since it is standard it should operate reasonably cleanly with library code of diverse origins, something that is almost impossible to guarantee for home-brew solutions.

11.2 Cancelling a thread

The **cancelation** mechanism is provided to allow one thread to force another to exit. Although it is possible for the target thread to evade death indefinitely, either by disabling cancelation⇒$^{11.3.3}_{208}$ or by looping forever in its (cancelation-immune) cleanup phase, there is no means provided for a thread to 'catch' and 'handle' a cancelation attempt. Once issued, a cancelation remains pending until the target finally dies.

Cancelation is invoked by calling `pthread_cancel()` with the target thread's ID as argument. As soon as cancelation of that thread is permitted (and some system calls may be interrupted for the purpose) it exits; *cf.* `pthread_exit()`$_{37}$.

`pthread_cancel()` **18.2.1**

```
#include <pthread.h>

int /*errno*/ pthread_cancel(pthread_t thread);
```

`pthread_t`$_{32}$ *`thread`*
    The thread to be canceled.

This function requests that *`thread`* be terminated. The next time *`thread`* has cancelation enabled and is at a point where cancelation takes effect⇒$^{11.3}_{205}$, it will exit with the distinctive final value `PTHREAD_CANCELED` (which has type `void *` but is neither `NULL` nor the address of any value), as if it had called `pthread_exit()`$_{37}$ with this argument. In particular, the usual thread-termination cleanup is performed.⇒$^{6.2.5}_{120}$, $^{6.3}_{121}$

On success the function `pthread_cancel()` returns `0`.

On failure it returns an error number:

`ESRCH`$_{271}$ The thread ID *`thread`* was invalid.

11.3 Cancelability

Allowing cancelation to take effect anywhere during the execution of a thread would cause immense problems. While facilities are provided to clean up after an exiting thread$\Rightarrow^{6.3}_{121}$, and these apply to a thread that has been canceled, they are all predicated on the assumptions that it can be determined when they are necessary, and that the values on which they operate can be located. As we have already noted, however, the action of an optimizing compiler can cause operations to be reordered and values to be copied unpredictably; the basic synchronization model of the pthreads system guarantees only a very limited number of points at which the appearance of a code sequence matches the global reality.[1]

The needs of cancelation are slightly different; since it provides no data transfer between the canceling and canceled threads, there is no need to ensure global *data* consistency between them. All that is needed is a kind of control consistency: that cancelation should occur at a definite point in the canceled thread's execution. To this end, the standard defines a list of functions that are **cancelation points**—locations where a thread internally checks to see if it has been canceled. The most basic of these, `pthread_testcancel()`$_{205}$, has no other purpose; it is simply a means for the programmer to **poll** the thread's cancelation state. The fact that there are also *implicit* cancelation points inside other functions$\Rightarrow^{11.3.2}_{206}$ allows cancelation to take effect when the thread is blocked.

In later sections we shall see that a thread can (temporarily or even permanently) disable its cancelability$\Rightarrow^{11.3.3}_{208}$; and that there is a special mode in which, notwithstanding the above discussion, the limitations on where cancelation can occur are lifted, at the cost of tight restrictions on what the thread can legally do$\Rightarrow^{11.3.4}_{209}$.

11.3.1 Volunteering for cancelation

The most basic means by which a thread whose cancelation has been requested can come to be terminated is by *volunteering* for cancelation; it does this by calling `pthread_testcancel()`. In a sense this is an assertion that the calling thread's state is logically consistent with respect to other threads and the various cleanup handlers that may be in effect (and the compiler is expected to respect this when optimizing). $\Rightarrow^{6.2.5}_{120},^{6.3}_{121}$

`pthread_testcancel()` **18.2.2†**

```
#include <pthread.h>

void pthread_testcancel(void);
```

If the calling thread has been canceled by `pthread_cancel()`$_{204}$ and its cancelability is not presently disabled$\Rightarrow^{11.3.4}_{209}$, this function will cause the cancelation to take effect;

[1]More than one computer has actually had the *hardware* limitation that it only takes interrupts at branches so that it doesn't have to synchronize elsewhere. Of course, this makes things look much tidier for software—the hardware does much of the necessary ignoring of asynchronous events for you!

that is, it will be as if `pthread_exit(PTHREAD_CANCELED)` had been called. (`PTHREAD_CANCELED`$_{204}$ is a distinctive constant of type `void *`, neither `NULL` nor the address of any object.) Otherwise it does nothing.

All normal cleanup processing will be performed.

The effect of this function is rather like that of testing a (per-thread) global flag and exiting when it is set; with this arrangement a thread wishing to cancel another would just need to set its flag and wait for it to pay attention. The differences are that:

- the complexity of arranging for a flag variable for each thread is handled by the system;
- the overhead of a full data synchronization between the canceling and canceled threads may be avoided;
- the mechanism is built-in and thus requires no special programming conventions to work; and
- a number of other POSIX functions implicitly check for cancelation even while their calling thread is blocked.

In the next section we give the complete list of functions that provide cancelation points.

11.3.2 Cancelation points

The real value of the cancelation mechanism, as compared to something user-coded, is its integration with the functions and system calls of POSIX and standard C. In particular, the following functions contain (or may contain) **cancelation points**, points at which (unless cancelation is explicitly disabled) a pending cancelation request *must* be recognized[18.1.2]:

Functions containing (mandatory) cancelation points
`pthread_testcancel()`$_{205}$
`pthread_cond_wait()`$_{83}$ `pthread_cond_timedwait()`$_{83}$ `pthread_join()`$_{40}$

`creat()` `open()` `close()` `remove()` `rename()` `fcntl()` `fsync()`
`read()` `write()` `aio_suspend()`
`mq_send()` `mq_receive()`
`sleep()` `nanosleep()`
`pause()`
`sem_wait()`
`sigwait()`$_{195}$ `sigtimedwait()`$_{196}$ `sigwaitinfo()`$_{196}$ `sigsuspend()`
`wait()` `waitpid()`
`system()`
`msync()`
`tcdrain()`

Functions that may recognize cancelation
`pthread_setcancelstate()`$_{208}\Rightarrow^{11.3.3}_{208}$

`lseek()`
`fopen()` `freopen()` `fclose()` `fflush()` `fseek()` `ftell()` `rewind()`
`fread()` `fwrite()`
`fputc()` `fputs()` `putc()` `putchar()` `puts()`
`fgetc()` `fgets()` `getc()` `getchar()` `gets()` `ungetc()`
`getc_unlocked()`$_{226}$ `getchar_unlocked()`$_{226}$
`putc_unlocked()`$_{226}$ `putchar_unlocked()`$_{226}$
`fprintf()` `fscanf()` `printf()` `scanf()`
`perror()`
`tmpfile()` `tmpname()`
`opendir()` `closedir()` `rewinddir()` `readdir()`
`ctermid()` `getcwd()`
`getpwnam()` `getpwnam_r()` `getpwuid()` `getpwuid_r()`
`getgrnam()` `getgrnam_r()` `getgrgid()` `getgrgid_r()`
`getlogin()` `getlogin_r()`
`ttyname()` `ttyname_r()`

If cancelation is recognized during a POSIX system call, that call is 'interrupted' in the same sense, and with the same effects, as if it had received a signal that would have caused it to return `EINTR`$_{271}$—except that cancelation processing, rather then normal processing, ensues. The standard allows *no* other cancelation points in POSIX or C functions.

With the exception of the function `pthread_testcancel()`$_{205}$, which has no purpose other than to introduce an explicit cancelation point, these functions are all ones that could block indefinitely; defining cancelation points to occur within them allows threads to be canceled effectively even when they are blocked. This would be a very difficult effect for a polled implementation to accomplish. Not all blocking functions are in the list, however; the function `pthread_mutex_lock()`$_{72}$ is deliberately absent so that threads can share state easily without becoming subject to cancelation. This reflects the intention that mutexes be used for short-term synchronization and condition variables for longer-term blocking.

There are other times when it is necessary to ensure that cancelation does *not* occur, even though one of the functions listed above be called.

11.3.3 Temporarily disabling cancelation

Since it is vital that cancelation *not* occur while data structures are inconsistent as seen by other threads or the target thread's active cleanup handlers, a function `pthread_setcancelstate()` is provided to inhibit cancelation. While cancelation is disabled for a thread, attempts to cancel it with `pthread_cancel()`204 will be recorded, but will not take effect unless and until cancelation is re-enabled. Even cancelation resulting from explicit calls to `pthread_testcancel()` is suppressed, since cancelation is only safe when *all* active functions (including not just the current function but all of its callers) are in a consistent state.

`pthread_setcancelstate()` **18.2.2**

```
#include <pthread.h>

int /*errno*/ pthread_setcancelstate
  ( int /*cancelstate*/ new_cancelstate
  , int /*cancelstate*/ *old_cancelstate
  );
```

`int /*cancelstate*/ new_cancelstate`
  The new cancelation enable state for the calling thread.

`int /*cancelstate*/ *old_cancelstate`
  Pointer to a location receiving the previous enable state.

The values for these parameters are chosen from among the following constants:

`PTHREAD_CANCEL_DISABLE` Cancelation of this thread is suspended for as long as this state is in effect.

`PTHREAD_CANCEL_ENABLE` *(Default)* Cancelability is enabled. *Note: a program using this constant explicitly is probably buggy! See the discussion below for an explanation.*

Cancelability is enabled or disabled for the calling thread according to the value passed as *`new_cancelstate`*. If called with `PTHREAD_CANCEL_ENABLE` there is a cancelation point just before the function returns; thus any pending cancelation request will immediately take effect. In either case, the previous setting is returned in *`old_cancelstate`*.

On success returns `0`.

On failure returns an error code:

EINVAL[271] The value of *new_cancelstate* was not among the valid options.

This function should normally be used in balanced pairs around a protected sequence, the first call locally disabling cancelation and saving the previous setting, the second restoring it, like this:

```
{ int cs;
  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &cs);
  ...
  pthread_setcancelstate(cs, &cs);
}
```

Thus the constant PTHREAD_CANCEL_ENABLE will not normally appear in user code. The reason is that a calling function that disables cancelation for reasons of safety will have its purpose defeated if any function it calls even temporarily re-enables cancelation. A function diverging from this practise should certainly document it.

Note that there is no way of getting the cancelability state without setting it (or vice-versa); a better name for this function might have been pthread_swapcancelstate(). This doesn't cause any particular difficulty, however, since it is always safe to *disable* cancelation for a short period.

11.3.4 Permitting asynchronous cancelation

Threads that do not call any of the functions listed in section 11.3.2 don't contain any natural cancelation points; even if they run indefinitely they will normally never respond to a cancelation attempt. A function that does not make any bocking system calls can always support cancelation by periodically invoking pthread_testcancel()[205], but this may be unacceptably inefficient: adding even one instruction to a tight inner loop might nbe an unacceptable overhead. For this reason the function pthread_setcanceltype() is provided: it can be used to modify the conditions under which a cancelation request is acted upon so that the calling thread can be canceled (potentially) between any two machine instructions. But note that the notion of 'machine instruction' is *not* part of the C programming model, so the behavior of software that employs this feature is not very well defined. See the next section for a list of the caveats that apply to this '**asynchronous cancelation mode**'.

pthread_setcanceltype() **18.2.2**

```
#include <pthread.h>

int /*errno*/ pthread_setcanceltype
  ( int /*canceltype*/ new_canceltype
  , int /*canceltype*/ *old_canceltype
  );
```

int /*canceltype*/ *new_canceltype*
The new value for the cancelation-recognition method of the calling thread.

`int /*canceltype*/ *old_canceltype`
: Location to receive the previous value of the cancelation-recognition method.

The values for these parameters are chosen from among the following constants:

`PTHREAD_CANCEL_DEFERRED` *(Initial value on thread creation.)* Cancelation may only occur at designated cancelation points.$\Rightarrow^{11.3.2}_{206}$

`PTHREAD_CANCEL_ASYNCHRONOUS` Cancelation may occur between any two machine instructions (which does not necessarily map onto source statements at all). This does not however imply that cancelation *must* apply at any point other than a cancelation point. This mode *greatly* restricts the functions that the thread can safely call. Of the standard POSIX and C functions, only `pthread_cancel()`$_{204}$, `pthread_setcanceltype()` and `pthread_setcancelstate()`$_{208}$ may be called. $\Rightarrow^{11.4.1}_{211}$

The cancelation recognition method of the thread is set to `new_canceltype`, and the old value is returned `old_canceltype`.

On success returns `0`.

On failure returns an error code:

`EINVAL`$_{271}$ The value of *`new_canceltype`* was not valid.

When adjusting cancel type locally within a function, it is *strongly* recommended that the setting originally in effect when the function was entered be saved, and then restored before the function exits. This is because which cancelation type should be used is a local (and crucial) property of a piece of code, and it is not practical to work with a coding style in which functions do not leave the state as they found it. For technical reasons explored in the next section, deferred mode must furthermore be the default; that is, functions that can (or must) be called in asynchronous cancelation mode are the exception and should be documented as such.

Warning: The fact that the cancelation type has been set to `PTHREAD_CANCEL_ASYNCHRONOUS` does not imply that cancelation *must* be detected between any two machine instructions; only that it *may* be. In some sense this is a flaw in the specification, but it does reflect a possible hardware limitation: it would permit an implementation on a machine with multiple physical processors and no interrupt mechanism at all—that is, one in which all scheduling-relevant behavior is due to explicit action on the part of the affected thread.

The astute reader may now have noticed the following bizarre fact: that (according to the standard) the official only way to be *sure* that cancelation takes place in asynchronous cancelation mode is to introduce a cancelation point by calling `pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &ocs)`, taking advantage of its side effect of immediate processing of pending cancelation requests. You'd think you could call `pthread_testcancel()` as you can in deferred cancelation mode, but it is not defined to be cancelation safe. Of course, any implementation that requires you to do this in order to have asynchronous cancelation recognized may be implementing the letter of the standard but is severely lacking in the area of the spirit.

Note that there is no way of getting the cancelation recognition type without setting it (or vice-versa); a better name for this function might have been `pthread_swapcanceltype()`.

11.4 Safety considerations

11.4.1 Asynchronous cancelation safety

Both deferred cancelation and asynchronous cancelation modes are useful: deferred mode because it restricts the effect of cancelation to places where the thread is blocked, making it easier to reason about what can happen; and asynchronous mode because it works without any special provisions in the program. Why is deferred mode, the more complex of the two, the default?

One reason is the severe restrictions on what functions a thread can call while in asynchronous cancelation mode; of the whole panoply of POSIX functions, there are only three that are safe: `pthread_cancel()`$_{204}$, `pthread_setcancelstate()`$_{208}$ and `pthread_setcanceltype()`$_{209}$. In particular even `malloc()` and `pthread_mutex_lock()`$_{72}$ are unavailable. This is the result of a substantial technical difficulty with the asynchronous cancelation model itself.

The problem is that the idea that cancelation can take place between any two *instructions* does not map well onto source-level notions. As we saw in chapter 5, the pthreads synchronization model goes to a lot of trouble to allow each thread to do its processing in the most efficient way, forcing the shared memory state to be consistent with the sequence of events in the source code only at explicit synchronization points. Asynchronous cancelation by its very nature conflicts with this idea, with the result that it can easily cause variables to be half-updated or left in an 'impossible' state with respect to each other.

Indeed, the problem goes even deeper: merely assigning to a value that has a cleanup handler registered for it with `pthread_cleanup_push()`$_{122}$ is unsafe. The problem here is that if the assignment is not implemented as a single instruction—and this is surprisingly likely—cancelation may strike in the *middle* of an update, leaving the value in an inconsistent state. The cleanup handler will then be invoked as part of thread termination, and will be presented with inconsistent data. If the value in question is a pointer, or a value with range restrictions, this is almost certain to be catastrophic. Even if the value happens to be assigned atomically, but represents a shared resource, there is a problem, since cancelation may take effect between the point at which the resource is marked as allocated and the point at which is stored where the cleanup routine expects to find it.

Note that this problem does not even involve interaction with other threads, except as the source of the cancelation request itself. The only solution is to protect the assignment operation by temporarily disabling cancelation or going into deferred mode (though a resource allocation function that is designed to be called from asynchronous cancelation mode might do this internally around an assignment to a reference parameter; if it returns an allocated value *as its result*, the bulletproofing burden necessarily still falls on the caller).

(Using some other implicit cancelability point, such as between statements or at branches, was not an option for the designers of pthreads; partly because some of the most important optimizations performed by a good compiler involve, for example, moving code out of loops whenever possible, thus blurring all such lines in the compiled code; and partly because there is no hardware support for other grains of interruptibility on most architectures, meaning that a compiler would be forced to insert implicit calls to `pthread_testcancel()`205 at all possible 'asynchronous cancelation points', making asynchronous cancelation mode much *less* efficient than deferred mode.)

After that long list of warnings, a very sensible next question is, what *can* be done in asynchronous cancelation mode? The answer is that all the basic mechanisms of the C language are available, including programmer-defined functions (though they must use deferred mode or disable cancelation when they need to synchronize or call library routines). The fundamental limitation is just that if a thread is canceled, none of the variables it altered can be expected to contain a meaningful value, even during the thread's own cleanup phase.

11.4.2 Cancelation races

Even when working with deferred cancelation, synchronization problems arise. Consider the common need to create a thread with a fixed lifetime. Intuitively, one might expect to write something like this:

thread_a (watchdog)	*thread_b (worker)*
`sleep(duration);` `pthread_cancel(thread_b);`	`process();` `pthread_cancel(thread_a);`

Unfortunately, there is a race condition here: if both threads reach their respective `pthread_cancel()`204 calls at around the same time, *both* calls will be executed; subsequently one or the other cancelation may be delivered to a thread that has already exited (which is an error). The problem is that `pthread_cancel()` does *not* contain a cancelation point, so neither thread will 'get there first' in any useful sense. To get the intended effect you must code something that tests for cancelation before issuing its own cancel, and does it indivisibly:

thread_a (watchdog)	*thread_b (worker)*

`sleep(duration);`
`pthread_cleanup_push`
`  ( mutex_unlock_destructor`
`  , &mutex_ab`
`  );`
`pthread_mutex_lock(&mutex_ab);`
`pthread_testcancel();`
`pthread_cancel(thread_b);`
`pthread_cleanup_pop(1);`
	`process();`
	`pthread_cleanup_push`
	`  ( mutex_unlock_destructor`
	`  , &mutex_ab`
	`  );`
	`pthread_mutex_lock(&mutex_ab);`
	`pthread_testcancel();`
	`pthread_cancel(thread_a);`
	`pthread_cleanup_pop(1);`

Note the further complication that unlocking the mutex must be handled by a cleanup handler because we may—indeed must—leave the critical section precipitately, by volunteering to be canceled.

The motivation for this inconvenience is, as usual, avoiding excess (implicit) synchronization. On a computer with multiple physical processors, hardware synchronization must be invoked to avoid the pathological case, synchronization that may be unnecessary for threads that do not indulge in mutual assassination. Rather than having all uses of cancelation pay for this special case, the programmer is forced to make the necessary operations explicit.

[[cap: **We should break up the type of problems into classes and show the best method for each. Such classes are:**

- **unbounded number crunching problems with no I/O or syscalls.**
  - **encryption/decryption**
  - **compression**
  - **matrix manipulation**
- **user specified canceling of an I/O request**
- **timeout of an I/O (or blocking request)**
- **(there should be more)**

We should mention that the standard doesn't specify whether a canceled thread that was waiting in pthread_cond_wait() or pthread_cond_timedwait() reacquires the mutex or not. This seems to be a serious oversight because I can see cases for either.]]

11.5 Example

[[sps: **Check all expired code in the following:**]]

tbounded-thread

```
#include <pthread.h>
#include <errno.h>

extern int /*errno*/ spg_pthread_create_detached
  (pthread_t *pthread, void *(*function)(*argument), void *argument);

extern int /*errno*/ spg_pthread_create_expiring
  ( pthread_t pthread
  , pthread_attr_t *attributes
  , void *(*function)(*argument)
  , void *argument
  , struct timespec *abstime
  );

#ifdef SPG_PTHREAD_EXPIRED
extern int /*boolean*/ spg_pthread_expired(void);
#endif

#include <pthread.h>

struct data
  { pthread_mutex_t mutex;
    pthread_cond_t cond;
    void *(*function)(void *argument);
    void *argument;
    struct timespec expiry;
    unsigned state;
    pthread_t worker;
  };

#define state_running 0x01
#define state_exiting 0x02
#ifdef SPG_PTHREAD_EXPIRED
#define state_expired 0x04
#endif

static void *assassin(void *data);
static void *cleanup(void *data);
static void *worker(void *data);

#ifdef SPG_PTHREAD_EXPIRED
static pthread_key_t expired;
#endif

int /*errno*/ spg_pthread_create_detached
  ( pthread_t *pthread
  , void *(*function)(void *argument)
  , void *argument
  )
  { pthread_attr_t attr;
    int x;
```

```
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    x = pthread_create(pthread, &attr, function, argument);
    pthread_attr_destroy(&attr);
    return x;
  }

int /*errno*/ spg_pthread_create_expiring
  ( pthread_t *thread
  , const pthread_attr_t attributes
  , void *(*function)(void *argument)
  , void *argument
  , const struct timespec *abstime
  )
  { struct data *data = malloc(sizeof(struct data));
    int x = 0;
    pthread_t a;

    #ifdef SPG_PTHREAD_EXPIRED
    static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
    static int inited = 0;

    /* Because there's no KEY_INITIALIZER! */
    pthread_mutex_lock(&mutex);
    if (!inited)
      { inited = 1;
        pthread_key_create(&key, NULL);
      }
    #endif

    if (!data) return ENOMEM;
    pthread_mutex_create(&data->mutex);
    pthread_cond_create(&data->cond);
    data->function = function;
    data->argument = argument;
    data->expiry = abstime;
    data->state = 0;
    spg_pthread_create(&a, assassin, data);
    x = pthread_create(&data->worker, attributes, worker, data);
    *thread = data->worker;
    return x;
  }

static void *assassin(void *data_);
  { struct data *data = data_;

    pthread_mutex_lock(data->lock);
    while
      (!( data->state & state_exiting
        ||data->state
        &&ETIMEDOUT == pthread_cond_timedwait
            (&data->cond, &data->mutex, &data->expiry)
      ) ) ;
```

```
    if (!(data->state & state_exiting))
      {
        #ifdef SPG_PTHREAD_EXPIRED
        data->state |= state_expired;
        #endif
        pthread_cancel(data->worker);
      }
    pthread_mutex_unlock(data->mutex);
    return NULL;
  }

static void exiting(void *data_);
  { struct data *data = data_;

    pthread_mutex_lock(data->mutex);
    data->state |= state_exiting;
    /* Note that we unlock *before* signaling! */
    pthread_mutex_unlock(data->mutex);
    pthread_cond_signal(data->cond);
    pthread_cond_destroy(data->cond);
    pthread_mutex_destroy(data->mutex);
    free(data);
  }

static void *worker(void *data_)
  { struct data *data = data_;
    void *x;

    #ifdef SPG_PTHREAD_EXPIRED
    pthread_setspecific(expired, &data->expired);
    #endif
    pthread_cancel_push(exiting, data);
    mutex_pthread_lock(data->mutex);
    data->state |= state_running;
    mutex_pthread_unlock(data->mutex);
    x = data->function(data->argument);
    pthread_cancel_pop(1);
    return x;
  }

#ifdef SPG_PTHREAD_EXPIRED
int /*boolean*/ spg_pthread_expired(void)
  { int *expired = pthread_getspecific(expired);

    return expired ? *expired : 0;
  }
#endif
```

11.6 Discussion

[[

- *Shows the strain of asynchronous notion in a synchronous system!*
- *Also evidence that hardware models are not as uniform as standardization committees might like. . . .*

]]

Appendices

11.a Under the Hood

Internally, cancelation works a lot like signals; the designers of pthreads even discussed using a single system to satisfy both sets of goals. To see the relationship, imagine that we add a pseudo-signal, `SIGCAN`, *just within the per-thread structures;* we do nothing to inform the operating system or the 'real' per-process signal system about it (you can't cancel a thread from outside the process anyway).

Cancelling a thread now corresponds to sending the signal:

```
int pthread_cancel(pthread_t thread)
  {return pthread_kill(thread, SIGCAN);}
```

If the thread were normally unmasked, this would result in cancelation processing being started *immediately;* this is not what we want. Cancelation only has immediate effect if cancelation is both **enabled** and **asynchronous**:

```
int pthread_setcancelstate(int new, int *old)
  { pthread_sigmask(SIG_BLOCK, cancel_mask, NULL);
    *old =
        pthread_self()->cancel & CANCEL_ENABLE
      ? PTHREAD_CANCEL_ENABLE
      : PTHREAD_CANCEL_DISABLE;
    pthread_self()->cancel = *old & ~CANCEL_ENABLE;
    if (new == PTHREAD_CANCEL_ENABLE)
      pthread_self()->cancel |= CANCEL_ENABLE;
    if ( (pthread_self()->cancel & (CANCEL_ENABLE | CANCEL_TYPE))
       == (CANCEL_ENABLE | CANCEL_TYPE)
       ) pthread_sigmask(SIG_UNBLOCK, cancel_mask, NULL);
    return 0;
  }

int pthread_setcanceltype(int new, int *old)
  { pthread_sigmask(SIG_BLOCK, cancel_mask, NULL);
    *old =
        pthread_self()->cancel & CANCEL_TYPE
      ? PTHREAD_CANCEL_ASYNCHRONOUS
      : PTHREAD_CANCEL_DEFERRED;
    pthread_self()->cancel = *old & ~CANCEL_TYPE;
    if (new == PTHREAD_CANCEL_ASYNCHRONOUS)
      pthread_self()->cancel |= CANCEL_TYPE;
    if ( (pthread_self()->cancel & (CANCEL_ENABLE | CANCEL_TYPE))
       == (CANCEL_ENABLE | CANCEL_TYPE)
```

```
    ) pthread_sigmask(SIG_UNBLOCK, cancel_mask, NULL);
  return 0;
}
```

(Here `cancel_mask` is a `sigset_t` containing only `SIGCAN` and we assume that each thread has a `cancel` field—a thread-specific variable might do instead.)

That explains how asynchronous cancelation is initiated: the new 'signal' is delivered and the thread starts executing the signal handler immediately. What about deferred cancelation? Obviously the first step is to mask `SIGCAN`.

In order to *volunteer* for outstanding (deferred) cancelation, we now need only 'blink' the thread's signal mask:

```
int pthread_testcancel(void)
  { sigset_t old;

    if (pthread_self()->cancel & CANCEL_ENABLE)
      { pthread_sigmask(SIG_UNBLOCK, cancel_mask, &old);
        pthread_sigmask(SIG_SETMASK, &old, NULL);
      }
  }
```

This ensures that if the cancelation 'signal' is pending it is serviced immediately; otherwise, of course, nothing happens. In order to make the implict cancelation points work we now need to insert calls to `pthread_testcancel()`$_{205}$ within each cancelable function. There's one wrinkle: cancelation is supposed to be recognized within a number of *blocking* calls. But since these are situations in which signals are also recognized, the signal-handling logic provides us with the solution we need: we need to enable `SIGCAN` while the thread is asleep, by placing the blocking part of the call between the two calls to `pthread_sigmask()`$_{193}$ in a copy of the testcancel code.

The scheme we've just presented accounts for how cancelation requests are generated and delivered; but how are they processed? On being canceled, a thread is supposed to unwind the stack of cleanup handlers; to call the destructors on the thread-specific data; and to make sure that a subsequent join of the canceled thread will return the magic constant `PTHREAD_CANCELED`$_{204}$. This turns out to be easy, because the requirements are almost identical to those for normal thread exit; all we need to do is install the following handler for `SIGCAN`:

```
void cancel_myself() {pthread_exit(PTHREAD_CANCELED);}
```

Existing logic will do the rest.

11.b Other Systems

11.b.1 OS/2

OS/2 lacks a notion of thread cancelation that is distinct from process termination. Instead, *thread* termination is the default consequence of receiving an uncaught signal, and general purpose signal exceptions raised with `DosKillThread()` serve the function of cancelation. This works, since under OS/2 *process* termination results from termination of the process's primal thread, while the primal thread receives all signals originating outside the process. Thus the OS/2 design trades a separate notion of cancelation for a permanent distinguished status for the primal thread.

Chapter 12

Libc for Threads

[[WHY NO SELECT()/POLL()!!!!]]

[[May want to add "other functions not to be used in a multithreaded program"—`sigprocmask()`*, for one.]]* [[**sps: Might talk about how good a job can be done with thread-specific.**]] [[**sps: Discuss that `select()` is non-posix, if popular?**]]

12.1 Introduction

A great deal of the power of the C programming language arises from the fact that it has been kept relatively small[1] with much of the functionality "built into" other languages provided through libraries. This reduces the inherent discrimination *against* user-programmed functions by placing them on an equal footing with vendor-supplied ones; it allows more functionality to be provided without taxing the programmer's memory of the base language; and it allows the standard library of basic functions to be overridden if necessary.

When introducing multiple threads into the language, this last attribute is a blessing! As we have seen, multithreaded programming, while conceptually fairly simple, requires paying attention to synchronization issues just to get what you would normally expect, to happen. In particular, the standard C libraries are, for one reason or another, largely incompatible with multithreaded programs.

Fortunately, most of the incompatibilities between the standard library and the needs of threaded software are at the library implementation level and *don't* require a change in the function-call interface (though they may cause some headaches for your compiler supplier). What this means for the programmer is that, for the most part, all you need to do is link with a **thread-safe** version of the standard library and all of the usual functions become available as before: everything again works as expected. This is essential, or migrating software to a threaded environment would be extremely hard. There are, however, a few things in the library that unavoidably change; and these form the main topic of this chapter. In discussing them we place some stress on *why* the various changes needed to be made; our hope is that this will help you see

[1]This, of course, is why C++ is a dog's breakfast by comparison :-).

what changes in your own programming style may be necessary when writing library code for multithreaded use.

The exceptions to business as usual (for the application programmer, that is) fall into four categories:

1. input/output functions requiring a new, explicit locking protocol;
2. functions whose interfaces are inherently inconsistent with multithreading and have had to be altered;
3. functions that continue to work only under certain restrictions; and
4. functions now having a more detailed specification that distinguishes between per-thread and per-process effects.

Items three and four of the list above, while important, are really just revisions to the documentation. Item two involves the addition of new, thread-safe versions of several functions; POSIX has named them systematically by adding the suffix `_r` (for '**re-entrant**') to the existing function names. Only item one, which relates to the `stdio` facility, involves the introduction of genuinely new functionality.

[[Here we want a table of user-visible changes.]] [[2.3.9 lists the few officially non-re-entrant libfns.]]

In the next section we examine the general issues behind the changes in the C library, both internal and external, since this is relevant to writing your own code as well as understanding the `libc` changes themselves. Subsequent sections go through the above four categories of interface change in sequence, and describe the POSIX functions that are subject to them. Finally, in an appendix to this chapter we discuss the changes that have to be made under the hood of a C library to make it thread-safe. As you will see there, they are not all trivial.

Warning: The adjustments in the standard library described in this chapter apply to all programs using the pthreads functions, whether or not they are actually multithreaded! This allows the pthreads library to use threads of its own internally.

12.2 Basic Issues

Before turning our attention to the specific changes pthreads makes to `libc`, let's take a look at the issues involved and the kinds of solutions that are appropriate.

When writing multithreaded code it is unfortunately not usually possible to use pre-existing libraries that are not thread-aware. Even if use of the library's facilities is limited to a single thread problems may still arise through 'third party' interactions—that is, the non-thread-aware library may invoke the services of another library (like, as we shall see, the standard library) which, while nominally thread-safe, imposes additional conditions on use in a multithreaded environment—conditions which may not be met.

What, then, are the problems?

Some of them are general problems of parallel programming, familiar to us from our discussion of synchronization facilities:

statics Centralized data structures used by a library are inherently subject to data races between threads. Since nothing stops multiple client threads from entering the same library and even the same function at the same time, static data structures, even if they are not shared between functions, must be protected by mutexes if they are not to be corrupted. A number of standard library functions suffer from this difficulty (not least of them `malloc()`), but the solution is generally transparent to the caller.

stateful interfaces Particularly in older coding styles and (for some reason) in graphics libraries, functions often maintain explicit state on the user's behalf. Making the individual *functions* thread-safe through the addition of locks then doesn't suffice to make the *library* thread-safe; the user state must also be moved to thread-private storage. The problem is that sequences of interleaved operations from different threads can result in one thread unexpectedly using the state set up by another. In general this kind of interaction is completely unpredictable. The pthreads standard has generally chosen to replace the standard functions with such stateful behavior by new ones in which the caller manages the state; another approach is to use the thread-specific data facility.$\Rightarrow^{6.2}_{114}$

globals Perhaps the worst kind of stateful interface is the direct exposure (as an `extern`) of a global variable. The problem is that if access to a shared object is direct, there is no function to which synchronization code can be transparently added. Other than revising the interface, the only solution is to use the C preprocessor to replace references to the global variable by an expression that is acceptable to C as both a value and an **lvalue**[2] but which refers to thread-specific storage (the syntax of C doesn't let a macro wrap the use of an lvalue in a locking sequence). This is the solution POSIX has assumed for `errno`$_{271}$, at least for those functions that have not adopted the newer and more thread-friendly convention of returning their error status directly.

structures Where a library presents a data structure abstraction, sharing objects between threads may (again through data races) violate its integrity. This means that either data structures need to be protected by mutexes, or this burden needs to be placed on the caller, so that they can only be used by one thread at a time. Even in the latter case, mutexes will be required if the library chains the structures together in any way, to handle links that cross between structures 'owned' by different threads.

In the case where single conceptual operations on a structure require multiple calls on the library, it may be necessary not only to provide locks within the structures, but to export new functions to allow the client to group the calls into atomic sequences—essentially exposing the internal locking to the user.

[2] Recall that an **lvalue** is an expression that is acceptable as the *left-hand side* of an assignment operation or the argument of the `&` operator: this includes variables, fields of structures and arrays, and dereferenced pointers—no matter how those pointers may be produced. The most likely form for an expansion of a pseudo-variable such as we describe is thus `*foo(bar)`, where `foo()` is a function (or macro) that returns the address of the 'real' variable that is to be read or written.

This is the approach adopted for the thread-safe version of the C `FILE *` I/O interface.

static returns Finally, one somewhat artificial synchronization problem arises from the use of functions that return references to their own private storage (often the intent is merely to return a `struct` value but the interface design dates to before the days of `struct` returns in C—or at least before most compilers were able to generate efficient code for them). These have the property that the actions of other threads can cause the returned values to be unexpectedly overwritten, even when there is no obvious semantic interaction between computations. The POSIX thread standard has chosen to replace these functions with equivalent ones where the caller provides the storage for results.

Fortunately, most contemporary C stylists would agree that even for single-thread code, good style requires that data structures be accessed only through functions provided for the purpose and that state be captured in data structures held by the user rather than held in the local storage of the library. Unfortunately(?), the ideals of code stylists rarely hold total sway in the real world.

Libraries also suffer from a number of problems that are not normally major concerns of application code. These include:

initialization A common problem with writing libraries is ensuring that data structures (other than those that can be handled with static initialization forms) are correctly set up. The traditional solution—whether or not initialization is made an explicit operation—is to use a static flag to record whether initialization has been performed, and to have library functions check this flag (and call the initialization routine) if it is not set.[3] Unfortunately there is a race condition here: if two threads arrive at the test simultaneously, initialization may occur more than once. The solution to this problem is again the introduction of explicit synchronization within the library; fortunately, this can be done in a way that is invisible to the caller.⇒$^{5.5}_{89}$

private data When state *must* be stored on behalf of a client thread or when interthread synchronization overhead is unacceptable it will be necessary to use thread-specific data in place of static variables.⇒$^{6.2}_{114}$ This is more often necessary in library code than in application code, because the library writer has no knowledge of how or when functions will be called; the worst case of arbitrary, unsynchronized interleaving of calls must be assumed.

transitive calls Finally, as already noted, there is a liability to the opacity of libraries. It should ideally *never* be necessary to read another programmer's source code[4], but the fact that the source of a library routine is generally not available for consultation means that the library writer is responsible for the

[3] The trivial solution of making it the library user's responsibility to ensure that an initialization routine is called only once is inadequate, since it does not allow for libraries that employ each other's services.

[4] I wish!

transitive behavior of their code to an even greater extent than usual. In particular, a library must function correctly in conjunction with any functions it calls from *other* libraries, and this may involve upgrading parts of those other libraries to thread-safety, inserting many additional synchronizing operations, or completely changing the way in which they are used. If you read the appendices to this chapter, you will see that this problem even exists between the thread-safe version of the standard C library and the kernel itself.

12.3 Stdio

Most C programs use the language's standard input-output package, `stdio`, in some way. The `FILE` objects that form `stdio`'s main data structure are shared resources in two distinct senses: first, in that three of them (`stdin`, `stdout` and `stderr`) have global names—and indeed are often accessed implicitly—and require synchronization to arbitrate between accesses from different threads; and second, in that the resources *named* by `FILE` objects are truly and visibly shared from the user's perspective. Thus, in the absence of some mutual exclusion protocol, there is a risk that the internal structure of `FILE` objects will become garbled; and even if this problem were solved, the output from different threads would still become arbitrarily mixed together—severely compromising the usefulness of the software!

These two issues are intimately related—and share a common solution—but are not quite identical.

To see this, note that the internal consistency problem can be solved simply—and invisibly to the applications programmer—by placing a mutex⇒$^{5.2}_{64}$ in each `FILE` object and only allowing access to the file by threads that hold the mutex, protecting the consistency of other fields. This is essentially the solution pthreads adopts (though the details—like the rest of the contents of the `FILE` structure—are private): all of the standard input/output functions implicitly lock their `FILE *` argument around their entire execution. This preserves consistency and furthermore prevents output from *single* calls on `stdio` functions becoming interleaved. (There's one small complication, which is that many of the standard I/O functions call each other behind the scenes, and the library programmer will have to be careful not to try to lock the same mutex twice!)

Adding this mutex field solves the problem of maintaining internal consistency and helps with the external interleaving problem: with the mutex in place, two calls to `printf()`, for example, will not get their output intermingled even if they are executed by different threads at the same time. But what if a thread needs to write its output character-by-character, or if a single message is constructed by more than one call to `printf()`? The solution is to expose the internal locking mechanism to the user.

Of course, this makes the precautions against multiple locking of the synchronization variable doubly important: it would be catastrophic if, having explicitly locked a file to obtain (temporary) exclusive use, the standard I/O functions were to attempt to lock the same mutex *again;* the thread would deadlock, waiting forever for someone to release the lock so it can acquire it (which can never happen since it is held by the very thread that is waiting). The solution is that the file object must contain not a raw mutex but some *recursively lockable* structure built using the basic synchronization tools (minimally, a mutex and a counter). That way a thread that already holds the file lock can re-lock it repeatedly; and not until it has been unlocked as many times as it was originally locked does it become available for other threads' use.

The presence of an explicit interface to this often implicit locking protocol has one benefit beyond the avoidance of garbling in lengthy output: it permits an optimization in which lightweight versions of the smallest I/O functions are provided that do *not* follow the normal locking protocol, relying on their caller to provide exclusion. This allows output to be built character-by-character without the cost of manipulating locks at each step. Of course, this is a somewhat risky practise and should only be followed in places where performance is critical.

Note, however, that the locks provided by the I/O library do not provide for exclusive access to other data structures like mutexes do; their synchronization applies only to the `FILE` objects they are associated with (this may seem a little counter-intuitive, but it allows for important optimizations on multiprocessors).

Warning: Under POSIX, C `stdio`-style I/O and Unix-style file descriptor I/O cannot be intermixed reliably. This being the case, *no* attempt has been made in pthreads to make `FILE` locking interact cleanly with the file descriptor layer. In a multithreaded program you probably shouldn't even consider trying to combine the two styles of operation on the same file or stream.

12.3.1 Locking a `FILE` explicitly

As we have just noted, the problem of unwontedly interdigitating output is solved by allowing the programmer to lock `FILE` objects explicitly, forcing other threads to wait before accessing them. This is done with the functions `flockfile()` and `funlockfile()`; these take a `FILE *` as argument and return nothing—and must, of course, occur in properly balanced pairs. A function `ftrylockfile()` is also provided, which a thread can use without fear of blocking to obtain a file lock if it is available; if the file is already locked it immediately returns a non-zero value (instead of just waiting like `flockfile()` does).

The only time these functions *must* be used is around calls to the optimized `_unlocked()`$_{226}$ I/O functions described below.

`flockfile(), ftrylockfile()` **8.2.6**

```
#include <stdio.h>

void flockfile(FILE *file);
int /*boolean*/ ftrylockfile(FILE *file);
```

`FILE *file`
: Standard I/O library file object to lock.

These functions both attempt to obtain exclusive use of *`file`* by acquiring its internal lock (in the same manner that the standard I/O functions do under pthreads). The function `flockfile()` waits until the lock becomes available, while `ftrylockfile()` returns a boolean status telling whether locking *failed:* that is, if `ftrylockfile()` returns zero then *`file`* is now owned by the calling thread; otherwise, the lock was unavailable.

If the file was *successfully* locked, the calling thread should remember to unlock it again when access by other threads is to be permitted. (Unlike the functions manipulating basic mutexes) a thread that holds a lock on a given file *may* attempt to lock that file again, and will succeed; the file maintains an internal count of how many times it has been locked by the owning thread.

Note that these are *not* synchronizing functions ($\Rightarrow^{5.2}_{64}$) in the usual sense; they cannot be safely used to mediate access to data structures other than *`file`* itself.

Warning: In order to avoid deadlocks within the standard I/O library's input echoing logic, threads must already hold any locks they need on input streams before acquiring locks on related output streams.

The ability to relock a locked file simplifies the coding of layered input/output systems, but it does not eliminate the danger of deadlock. As with any kind of blocking mutual exclusion mehchanism, two (or more) threads can still deadlock if they attempt to acquire locks that are (in some cyclic fashion) already held.$\Rightarrow^{5.6.2}_{93}$ But beware that there is also a hidden dependency *inside* the input/output library, since some input streams *echo* to output streams. This means that an input operation (other than `flockfile()` or `ftrylockfile()`) may implicitly wait for an output lock to become available.

Thus the usual advice that threads should avoid deadlock by acquring their resources in a globally fixed order must be extended: the order chosen must be consistent with that used internally by the `stdio` library—which locks input streams before output streams. Locking *all* input streams before locking *any* output streams is the safest course, since since less conservative policies require reasoning about the modes and relationships of the actual streams that are open.

Once a thread is finished using a file it has locked, it should unlock it to grant access to other threads (this is a good idea even in the case where there is only one thread that normally uses a given I/O stream, since software errors may cause other threads to try to write to the file; debugging may be simplified if this output is allowed, rather than just leaving the failing thread to wait forever).

If a file has been locked more than once, `funlockfile()` must be called the same number of times before the file is actually released.

`funlockfile()` **8.2.6**

```
#include <stdio.h>

void funlockfile(FILE *file);
```

`FILE *file`
: File to unlock.

This function decrements the file's internal count of the number of times the calling thread has locked it. When this count reaches zero, the caller no longer owns the file and other threads are once more permitted to use it.

This function may only be called by the thread that actually owns the file; otherwise something unpleasant may happen.

Note that this is *not* a synchronizing function ($\Rightarrow^{5.2}_{64}$) in the usual sense; it cannot be safely used to mediate access to data structures other than `file` itself.

12.3.2 Character I/O assuming explicit locking

Specialized versions of the standard character-by-character input/output macros like `getchar()` have been provided that do *not* perform any implicit locking; instead, they rely on the calling thread to have already obtained a lock on the appropriate `FILE` object. This eliminates the overhead of performing repeated locking and unlocking, at the expense of safety: if two threads *should* invoke these macros on the same file at the same time, the `FILE` structures may be corrupted.

{get,put}{c,char}_unlocked() **8.2.7†○**

```
#include <stdio.h>

int /*uchar or eof*/ getc_unlocked(FILE *file);
int /*uchar or eof*/ getchar_unlocked(void);
int /*uchar or eof*/ putc_unlocked(int c, FILE *file);
int /*uchar or eof*/ putchar_unlocked(int c);
```

These are identical to the standard macros `getc()`, `getchar()`, `putc()` and `putchar()`, except that (a) they take no precautions of their own to avoid interference between threads and (b) they contain cancelation points$\Rightarrow^{11.3}_{205}$. They may *only* be used by a thread that already owns a lock on the relevant stream—`stdin`, `stdout` or `file`.

Their results are the same as their normal, locking, counterparts:

On success they return the character read or written, as an unsigned character widened to `int`.

On failure they return the constant `EOF`, and set the per-thread `errno`$_{271}$ 'variable'.

In fact, these `_unlocked` versions of the character I/O macros will typically have identical implementations to the 'normal', single-thread versions; it is the macros that do implicit locking that have changed. Why was this done?

It is because you don't want to be norked[5]. The convention that the version of the macro with the standard name is the one with the seat belts ensures that programs written in standard C continue to work as expected. The 'original' versions—while admittedly more efficient—have different and more dangerous semantics in the presence of multiple threads, and so receive the new name.

[5]The verb **to nork** is, as far as we are aware, invented. Nonetheless, we are confident that it designates something fairly unpleasant and generally to be avoided.

Only the *macro* versions of the character I/O operations have been provided with non-locking versions—there is a `getc_unlocked()` but no '`fgetc_unlocked()`'. The reason is that the cost of locking is expected to be comparable to that of a function call. Thus, applications that are willing to pay the expense of invoking an actual library *function* can be assumed to be able to tolerate the locking overhead as well. Only in the character-by-character case is the overhead sufficiently great (proportionately speaking) that a new function becomes necessary.

12.4 Replacements for static data functions

The second category of functions requiring attention in a thread-safe C library are the only ones whose *interface* has actually changed (rather than just being extended or specified in further detail). These are the functions whose design is incompatible with multithreading.

How can this be? They all share variants of the same problem: they all make use of static storage, either to maintain internal state or to provide space to hold result data—to which they then return a pointer. In either case they are designed so that this local storage can be reused the next time they are called. The idea was that this lifts the burden of allocating a (possibly unknown) amount of storage from the caller, on condition that results be copied if their use continues beyond the next call of the function. By modern standards this is poor design (it confuses *values*, which should not change, with *objects*, which might; and it mixes object lifetimes into control flow in non-transparent ways), but the functions have already been inherited by the C and POSIX standards.

With multiple threads what was once merely poor code/data structuring becomes a disaster: when multiple threads call such functions, they will conflict for access to the static storage in which they keep their state or build their results. *Semantically* the functionality they provide does not involve synchronization, but their traditional interfaces apparently introduce a synchronization requirement.

To get around this problem, revised, re-entrant versions of these functions have been provided (with new names ending in `_r`) which move the responsibility for allocation of the contended storage to the caller. (Actually a further, less necessary change was made: a number of the new `_r`-interface functions also return error codes in the pthreads style, rather than setting `errno`.)

In fact, the problem could have been solved *without* change of interface (at some cost in performance), either by specifying that the caller is responsible for ensuring serialization (perhaps even by limiting their use to the initial thread of a process), or by having each thread implicitly allocate a thread-specific data area$\Rightarrow^{6.2}_{114}$ to transparently replace the static storage. Indeed, just for the sake of backward compatibility, we can hope that most implementations will adopt (and document!) one or the other solution. *The standard, however, makes no such guarantee:* in the view of strict POSIX, any multithreaded program (or even any program that makes use of thread-related or re-entrant facilities, such as `pthread_atfork()`$_{241}$) is *incorrect* if it calls any of the original versions of the functions in this section.

Note that we have not provided full descriptions of the new re-entrant versions of the functions in this section, because their semantics are unchanged from the traditional versions; only the details of argument passing are changed. They are therefore described relative to their traditional definitions.

12.4.1 Time functions

Four of the standard time-related functions, `asctime()`, `ctime()`, `gmtime()` and `localtime()`, are specified as returning a pointer to a (re-usable) static buffer. In each case this buffer is of a fixed size and type; the revised re-entrant interface just requires the caller to provide the address of a buffer to receive the result as a new final argument. This also becomes the value returned, once it is filled in.

```
#include <time.h>
#include <unistd.h>                /* For _POSIX_REENTRANT_FUNCTIONS */
#ifdef _POSIX_REENTRANT_FUNCTIONS

char *asctime_r(const struct tm *time, char buf[26]);
char *ctime_r(const time_t *clock, char buf[26]);
struct tm *gmtime_r(const time_t *clock, struct tm *buf);
struct tm *localtime_r(const time_t *clock, struct tm *buf);
```

Note that the results of the original `asctime()` and `ctime()` functions are specified by the standard to be exactly 26 bytes long.

12.4.2 Process environment queries

Two functions that retrieve information about the current process also suffer from the problem of using a single buffer to return all their results. These are `getlogin()` (which returns the name under which the the user of the control terminal of the current process logged in) and `ttyname()` (which returns the pathname of the terminal associated with a given file descriptor, if any). Unlike the time functions described in the previous section, the values returned do not have a pre-specified size. Instead, the new interface also has the caller specify the size of the buffer being passed, and an error is returned if it is too small.

These functions have been modified to use the new convention of returning an error number (rather than returning `NULL` on error and setting the `errno`$_{271}$ variable). In particular, they return zero on success and the new status `ERANGE`$_{271}$ if the buffer supplied was too small to hold the result (including its terminating null).

```
#include <unistd.h>                /* For _POSIX_REENTRANT_FUNCTIONS */
#ifdef _POSIX_REENTRANT_FUNCTIONS

int /*errno*/ getlogin_r(char *name, size_t name_size);
```

```
int /*errno*/ ttyname_r
  (int /*fd*/ file_descriptor, char *name, size_t name_size);
```

So that you don't have to guess blindly at a suitable size for these buffers, a `sysconf()`242 query is provided for each of these functions to obtain the maximum size necessary in the current environment: these are `_SC_GETLOGIN_R_SIZE_MAX` and `_SC_TTYNAME_R_SIZE_MAX`.

12.4.3 File/directory access functions

The original function that scans directories, `readdir()`, and those that retrieve entries from the `/etc/passwd` and `/etc/group` files also share the problem of returning statically allocated structures. The new re-entrant interfaces have been complicated slightly by the (slightly strange) decision to retain the convention that a `NULL` result indicates failure while introducing pthread-style `errno` returns. As can be seen from the prototype of `readdir_r()`, this has resulted in the addition of *two* new parameters, one a pointer to the entry to be filled in, and one a pointer to a variable that is made `NULL` on failure and set to the address of the result structure on success:

```
#include <sys/types.h>
#include <dirent.h>
#include <unistd.h>              /* For _POSIX_REENTRANT_FUNCTIONS */
#ifdef _POSIX_REENTRANT_FUNCTIONS

int /*errno*/ readdir_r
  (DIR *dirptr, struct dirent *entry, struct dirent **result);
```

(One wonders why it was deemed preferable to add an eccentric last parameter rather than allowing the function to return a new status code to the caller, indicating end of directory; or perhaps to use a single argument of type `struct dirent **` for both input and output.)

The group and password database access functions have been modified similarly, but they suffer from an additional complication: the structures they return contain strings, which must themselves be allocated and freed correctly. Rather than requiring the caller to pass in a large number of separate buffers, a **scratch buffer** has been added—a single contiguous area of storage that the query carves up into strings as necessary. The minimum size that is guaranteed to be sufficient for the function to succeed is available from `sysconf()`242 with the parameters `_SC_GETPW_R_SIZE_MAX` (for the scratch storage associated with `struct passwd`) and `_SC_GETGR_R_SIZE_MAX` (for `struct group`).

```
#include <pwd.h>
#include <sys/types.h>
#include <unistd.h>              /* For _POSIX_REENTRANT_FUNCTIONS */
#ifdef _POSIX_REENTRANT_FUNCTIONS
```

```
int /*errno*/ getpwuid_r
  ( uid_t uid
  , struct passwd *passwd
  , char *scratch
  , size_t scratch_size
  , struct passwd **result
  );
int /*errno*/ getpwnam_r
  ( const char *name
  , struct passwd *passwd
  , char *scratch
  , size_t scratch_size
  , struct passwd **result
  );
```

```
#include <grp.h>
#include <sys/types.h>
#include <unistd.h>               /* For _POSIX_REENTRANT_FUNCTIONS */
#ifdef _POSIX_REENTRANT_FUNCTIONS

int /*errno*/ getgrid_r
  ( gid_t gid
  , struct group *group
  , char *scratch
  , size_t scratch_size
  , struct group **result
  );
int /*errno*/ getgrnam_r
  ( const char *name
  , struct group *group
  , char *scratch
  , size_t scratch_size
  , struct group **result
  );
```

The use of `sysconf()`$_{242}$ as the repository for information about suitable sizes for the scratch area unfortunately means that sizes are not *manifestly* determined, and cannot be used as array dimensions—making stack allocation of the scratch area infeasible.[6] It probably simplifies allocation and freeing of the necessary scratch storage more often than it complicates it, though—and keeps the argument lists of these functions a reasonable length.

[6]Unless your environment provides `alloca()` or, like the GNU Project's `gcc` compiler, allows dynamic computation of array bounds.

12.4.4 Exposing state: rand() and strtok()

The final two functions whose interfaces have been revised to provide for re-entrancy were originally designed specifically to retain hidden internal state. These functions are `rand()`, which generates a sequence of pseudorandom numbers, and `strtok()`, which destructively segments strings on delimiters. The revised versions alter the interface only insofar as they expose this state to the caller.

random numbers

The new re-entrant function `rand_r()` actually combines the old functionality of `srand()` and `rand()` by exposing the current state (which serves also as the initial seed) as a parameter:

```
#include <stdlib.h>
#include <unistd.h>                 /* For _POSIX_REENTRANT_FUNCTIONS */
#ifdef _POSIX_REENTRANT_FUNCTIONS

int rand_r(unsigned int *seed);
```

This new interface has the advantage, by the way, that it is simple to maintain multiple independently reproducible streams of pseudorandom numbers, which can be a blessing when debugging; this would not have been possible if the simpler solution of just placing a mutex around the function body within the library code had been adopted.[7]

tokenization

Tokenzation of strings is an unusually specialized operation to build into the standard library of a small language like C, and the `strtok()` function is even more restricted in its application by virtue of its destructive design. Nonetheless, the designers of pthreads have provided a re-entrant version that makes its hidden state explicit.

The original version of `strtok()` takes, on its *first* invocation, a string and a list of delimiters, and returns the string itself if one of the delimiters occurs in it, and `NULL` otherwise. The string, however, is truncated in place by overwriting the first delimiter found with a null. The address following this character is saved in an internal variable and used as the new starting point if a `NULL` is passed as the first argument of the next call. This allows a string to be divided into several tokens without the caller having to maintain any explicit state.

Of course, to make it usable by several threads at the same time, the state variable needs to become private to the tokenizing computation. This is once again accom-

[7] It's worth noting that the fact that the internal state of the `rand()` family is now exposed specifically as an `unsigned int` *guarantees* that it will provide a very poor source of randomness: since an `unsigned int` provides only one more bit of representation than the positive-valued `ints` in the sequence, and since the standard guarantees that `rand_r()` is deterministic relative to the values of the seed, it follows that each number in the sequence generated by `rand_r()` has (on average, and probably in practise) only two possible successors. Your C library probably provides a better alternative—such as BSD's `random()`—though it may be neither thread-safe nor POSIXly portable.

plished by adding an extra parameter that points to a user-allocated buffer—this time a single `char *`:

```
#include <string.h>
#include <unistd.h>              /* For _POSIX_REENTRANT_FUNCTIONS */
#ifdef _POSIX_REENTRANT_FUNCTIONS

char *strtok_r
  (char *string, const char *separators, char **state);
```

Note, however, that the standard does *not* specify what exactly is stored in *`state`*—it need not actually be the next value to be returned—and that in practise it is simpler just to use `strpbrk()` anyway.

12.4.5 Functions with argument restrictions

There remain two final functions whose calling conventions pose problems for multithreaded software. The functions `ctermid()` and `tmpnam()` are normally re-entrant, but as a convenience to the programmer traditionally allow the passing of a `NULL` pointer for the buffer argument—in which case a statically allocated area is used instead. The change for pthreads is simply that passing `NULL` is forbidden: only explicitly passed buffers are allowed.

This fix is simple but a trifle unfortunate for people converting old code, because the new restriction is incompatible with existing practise and there is no general method of checking a program for conformity—certainly not at link time. This contrasts with the I/O library, which retains a reasonable interpretation of its original functionality, and the functions receiving new re-entrant versions which can be validated just by omitting the original, non-re-entrant versions completely from the thread-safe version of the system library.

12.5 Effects limited to individual threads

The final respect in which multithreading impacts the user of the standard library is that some functions require interpretation as to whether their effects apply to the whole process or just to the calling thread. In general terms, blocking phenomena (waiting for something to happen), changes to `errno`$_{271}$, and the `setjmp()/longjmp()` mechanism apply to single threads, while most pieces of explicit state (such as the working directory and the '**locale**'—which determines currency and time formats, for example) continue to be associated with processes. In the following paragraphs we list the functions—other than those that we have already discussed—that are now interpreted as operating at the thread level.

12.5.1 Blocking

A number of functions cause their caller to wait for something to happen, either because that's what they are for (like `sleep()` or `wait()`), or because (as with `read()`) there is sometimes a delay before they can complete. One of the main purposes of threads, of course, is to allow computations in other threads to proceed, even while something is being awaited. Thus, blocking is specifically interpreted as applying only to the calling thread; if one thread calls a function that waits for the occurrence of some event, the other threads continue to compute.[8]

12.5.2 Nonlocal jumps

One odd mechanism (in terms of its effect on the flow of control) provided by C is the nonlocal jump through `setjmp()`/`longjmp()` and (POSIX) their signal-manipulating counterparts, `sigsetjmp()`/`siglongjmp()`. These are specified to work only *within* threads: a thread may only `longjmp()` to a `jmp_buf` that it set itself. This restriction is quite reasonable when you think that the only reason `longjmp()` works is that the current stack frame is abandoned when the new one is adopted; if `longjmp()`ing between threads were allowed, two frames of the *same* stack could end up active—and that could spell nothing but disaster.

Note that this means that if you are using `siglongjmp()` you have to pay very careful attention to which thread services each signal!$\Rightarrow^{10.5}_{192}$

12.6 Effects still global to the process

A number of functions still have effect only at the process level; a few of them may be somewhat surprising, if only because of the idioms they break. Foremost among these, as we've already seen, is `sigaction()`??, which sets the entries in the table of signal handlers.

Another function that works at the process level and will break familiar idioms because of it is `alarm()`: rather than making it the only system function to deliver an implicit asynchronous signal to a particular thread, it has been interpreted to signal the entire process when the timer expires. Thus it can't be used (or at least, it can't be used by more than one thread at a time) to bound the time spent in a system call. Instead of using `alarm()` it is now necessary to use a separate watchdog thread to do your timing for you; it can then deliver a signal (or send a cancelation request) later; the alarm can be reset by canceling the watchdog.$\Rightarrow^{11}_{203}$

spg_alarm.h

```
extern void spg_alarm(unsigned seconds);
```

spg_alarm.c

```
#include <pthread.h>
#include <unistd.h>
#include <signal.h>
```

[8]Technically, the POSIX standard makes no such guarantee—it specifies only that threads are not suspended *indefinitely* just because other threads are suspended. The legalistically inclined might argue that the heat death of the universe provides an upper bound on any suspension. Of course, such lawyers are unlikely to sell many threads implementations!

```
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static int inited = 0;
static pthread_key_t watchdog;

struct dogspec
  { unsigned time;
    pthread_t victim;
  };

static void *dogcode(void *argument)
  { struct dogspec d = *argument;

    free(argument);
    sleep(d.time);
    pthread_kill(d.victim, SIGALRM);
    return NULL;
  }

static void canicide(void *argument)
  { pthread_t *t = argument;

    pthread_cancel(*t);
    free(t);
  }

void spg_alarm(unsigned t)
  { pthread_t *thread;
    struct dogspec *arg;

    pthread_mutex_lock(&mutex);
    if (!inited)
      { pthread_key_create(&watchdog, canicide);
        inited = 1;
      }
    pthread_mutex_unlock(&mutex);
    if (pthread_getspecific(watchdog)
      canicide(pthread_getspecific(watchdog));
    arg = malloc(sizeof unsigned);
    arg->time = t;
    arg->victim = pthread_self();
    thread = malloc(sizeof pthread_t);
    pthread_setspecific(watchdog, thread);
    pthread_create(thread, NULL, dogcode, arg);
  }
```

(This implementation isn't perfect; it doesn't return the number of seconds remaining on the previous `alarm()` request like the standard version does.)

A similar problem that may break familiar idioms is the fact that the working directory is shared between all threads of a process; this means that thread-safe file-system-scanning code typically has to be written with full pathnames rather than by adjusting the working directory and using unqualified filenames.

Perhaps less bothersome, the **locale** (which controls time and number formatting, among other things) is also shared between all threads of the process.

In general, you will need to pay careful attention to problems arising from process-shared settings when updating old code or converting libraries for thread-safety.

12.7 Example

[[This should use `FILE` locking and an `_r` function or two.]]

12.8 Summary

When working with threads, you need to use thread-safe libraries. The pthreads standard requires a thread-safe `libc` to be provided, and it is almost identical to the standard system library (though of course there are some new requirements on how it behaves). The differences are:

- the standard I/O functions now perform automatic locking so that each call behaves atomically;
- functions `flockfile()`$_{224}$, `ftrylockfile()`$_{224}$ and `funlockfile()`$_{225}$ are provided so that the programmer can make longer sequences of I/O operations behave atomically;
- `_unlocked` versions of the character I/O macros are provided for maximum performance in places where the file is already explicitly locked;
- a few functions have been given new re-entrant calling sequences through new names ending in `_r`; these just allow the caller to pass in a locally allocated buffer to replace what used to be statically-allocated storage that might have been clobbered by competing threads;
- functions that generally accept a buffer argument but used to allow you to pass `NULL` and get a statically allocated default buffer now *require* a non-`NULL` argument.

The main surprises are that you must obey the new conventions whenever you are using any of the threads-related functionality—not just when there is obvious potential for conflict—in case the system wants to use multiple threads of its own. You should also pay careful attention to which resources are now per-process and which per-thread, since in a few cases the choice may be surprising and break a lot of familiar coding techniques. This applies especially to signals, `longjmp()`, the working directory, and the timer and signal used by `alarm()`.

Appendices

12.a A look inside

On the whole, there is nothing remarkably difficult about the changes necessary to make the C library thread-safe; at the worst, it is a matter of recoding everything in a thread-aware fashion. In this section we look at a few highlights.

12.a.1 Stdio

As we have seen, the main thing that has been added to the C standard input-output library is a mutual exclusion system that guarantees that (a) the behavior of the basic operations is reasonably intuitive and (b) it is straightforward to ensure that this intuitiveness can be extended to non-primitive I/O functions supplied by the programmer. Why—and how—this should work is made clearer by contemplation of `printf()`, which is not actually very 'primitive' in a typical implementation, being coded in terms of a number of other library functions.

What we need to get the guarantee that different standard library operations will not interleave incorrectly is, of course, to add a lock—a mutex—to each `FILE` structure. (Using a single global lock wouldn't work because then a thread blocking for input on one file would prevent other threads from producing output on another, for instance.) The functions `flockfile()`$_{224}$, `ftrylockfile()`$_{224}$ and `funlockfile()`$_{225}$ can now just manipulate this lock; and the other functions of the standard I/O library need to be bracketed internally by `flockfile()/funlockfile()` pairs.

Unfortunately, a straight mutex isn't quite good enough for this purpose. Since I/O operations are typically layered and may call each other, we need a **recursive** or **counting mutex**—it's alright if `printf()`, having acquired the lock on the stdout structure, wants to call `putchar_unlocked()`$_{226}$ as an *optimization*, but *forbidding* it to call `putchar()`—or `puts()`—would complicate matters unnecessarily. Worse yet, using a simple mutex would make it impossible for an application programmer writing an output function with proper locking to call `printf()`!

Here's how to implement counting mutexes in standard pthreads (though your implementation may already provide them directly as an extended mutex class$\Rightarrow^{8.c}_{151}$):

```
typedef struct
  { pthread_mutex_t lock;
    pthread_cond_t free;
    pthread_t owner;
    unsigned count;
  } counting_mutex;

int counting_mutex_init(counting_mutex *mutex)
  { int x = 0;

    mutex->owner = NULL;
    mutex->count = 0;
    x = pthread_mutex_init(&mutex->lock, NULL);
    if (!x) x = pthread_cond_init(&mutex->cond, NULL);
    return x;
  }

int counting_mutex_lock(counting_mutex *mutex)
  { pthread_mutex_lock(&mutex->lock);
    if ( mutex->count
       &&!pthread_equal(pthread_self(), mutex->owner)
       )
      { while (mutex->count)
          pthread_cond_wait(&mutex->cond, &mutex->lock);
        mutex->owner = pthread_self();
      }
```

```
    mutex->count++;
    pthread_mutex_unlock(&mutex->lock);
    return 0;
  }

int counting_mutex_unlock(counting_mutex *mutex)
  { pthread_mutex_lock(&mutex->lock);
    if (!--mutex->count)
      { mutex->owner = NULL;
        pthread_cond_signal(&mutex->cond);
      }
    pthread_mutex_unlock(&mutex->lock);
    return 0;
  }
```

The function `counting_mutex_lock()`, by the way, provides a long-awaited example of the use of `pthread_equal()`$_{43}$; here we use it to determine if the current owner of the lock is indeed the calling thread—if so, we re-lock the counting mutex, but otherwise we must block until it becomes free.

12.a.2 Exclusion

Another case where additional locking is required for thread-safety arises in functions like `malloc()`. The heap storage functions between them manage a *global* storage pool, which, because it is shared between all threads of a process, must be protected by a mutex (in this case a conventional mutex is entirely sufficient); the lock and unlock operations can just go around the outside of the standard storage management logic (indeed, if the linker permits, the new version might actually call the single-thread version). For applications with many threads and a high allocation bandwidth, this new mutex may be something of a bottleneck—most especially on multiprocessors. What might a highly optimized implementation do it ameliorate the situation?

The way the input-output functions avoid such a global bottleneck is by associating locks not with the process, but with the file, thereby spreading the load. In the case of `malloc()` there is no obvious way of grouping allocation requests (a fast `malloc()` may have different cases for different sizes, but a distributed application with a high allocation bandwidth is quite likely to have all threads hammering on the *Same* size objects), but there is no reason not to invent one! One potentially effective idea is to maintain a separate list of free storage in each thread, so that re-allocation of freed storage does not require further external synchronization. A global lock is still needed to protect first-time allocation of memory, however, and to allow a dying thread to give up its local pool.

In order to prevent pathological cases where free storage is “pumped” from one thread to another, either each thread needs to keep track of the amount of storage it has in its local pool and dump some of it back to the global area when it becomes excessive, or threads need to be able to ‘steal’ from each other in some way. The former is probably more efficient, since rebalancing by theft would require each thread to have a lock of its own: although this lock would rarely be contended and locking an uncontended mutex is fast, it would still need to be used around every allocation operation.

One variant on this approach that is very important on multiprocessors is to maintain a separate free storage pool *per physical processor*. Doing things this way (instead of basing it on user-level threads as we have just described) potentially exploits a number of caching properties of typical multiprocessor hardware, especially if the scheduler tends to repeatedly run each thread on the same physical processor (one of the so-called **affinity-scheduling** techniques).

12.a.3 Blocking

The main area in which the standard library requires work to make it thread-safe is one that isn't even visible to the user—at least, not when seen from the perspective of an individual thread. This is that the many blocking library calls all need to be modified so that they delay only the calling thread.

Unless threads are implemented in the kernel, this does not come for free. Because in a user-level threads implementation there is only one 'real' thread of control in a process (as seen by the kernel), which must be multiplexed between the various user-level threads, any blocking call that is executed will actually block the entire process and, *a fortiori,* every thread in it.

The workaround—which is far from straightforward to implement—is to re-implement every blocking call in terms of nonblocking calls with the same functionality. A blocking read, for example, must be replaced by code that checks whether enough input is yet available to satisfy the operation. If so, the read is performed and the call returns; otherwise the calling thread is suspended, and another thread is run. When the original thread is finally resumed, it retries the nonblocking test, and so on. (This is actually quite problematic in the case of a read from a file, since (single-threaded) POSIX until recently provided no non-blocking read operation on actual *files:* if you read the specification of the non-blocking flag, you will discover that regular files are a specific exception in its behavior. This can be circumvented using the newly introduced **asynchronous I/O functions**, but these are still optional (they are present if `_POSIX_ASYNCHRONOUS_IO` is defined). This is one of a small number of cases where a user-level threads library may *require* an extension to a standard POSIX kernel in order to handle the specified semantics perfectly.[9])

One class of function that is particularly tricky to multiplex is those that provide timeout processing. The problem is that POSIX has not been especially generous in providing timer functions; or rather, a number of different timing facilities have been provided, but each one allows only a single outstanding request. In order to provide the appearance of separate timers for each thread, it's necessary to sort outstanding requests together into a single time-line for each kernel timer service, physically issuing only the most imminent request, then issuing an additional request with the remaining time on the *next* deadline before returning to the blocked thread when that expires—and so on. This works (though in some cases its overhead and resynchronization load may introduce timing errors)—but it is a mess to implement.

Although providing the correct blocking semantics for the system primitives guarantees that user programs will not be subject to hidden blocking effects within the system libraries, the application programmer must still beware of introducing similar problems at a higher level. The difficulty is that if a thread blocks for an extended period while holding some protected resource, all other threads needing that resource will block, resulting in a 'log jam' (and possibly even deadlock, if there are problems with the locking order). In general the system libraries provide 'enough' parallelism for applications to make good progress, but it pays to think carefully about each point where you reduce this parallelism by introducing serializing

[9] The reasoning behind the exception is apparently that local file operations do not block indefinitely anyway. Network file systems are now common, however, so the assumption is no longer realistic. Indeed, with common processors approaching one GIPS, even a local disk is starting to look extremely slow!

code: can a thread bring the whole system to its knees by blocking within a protected region? And if so, is there an alternate way of coding things that allows the other threads to continue with their work?

12.b Implementations

12.b.1 NetBSD

12.b.2 Frank Mueller's[RENAME]

12.b.3 Solaris

12.b.4 OSF/1

functions	*header*
`readdir_r()` $\boxed{5.1.2.1}$	`dirent.h`
`flockfile()`$_{224}$ `funlockfile()`$_{225}$ $\boxed{8.2.6}$ `getc_unlocked()`$_{226}$ `getchar_unlocked()`$_{226}$ $\boxed{8.2.7}$ `putc_unlocked()`$_{226}$ `putchar_unlocked()`$_{226}$ $\boxed{8.2.7}$	`stdio.h`
`rand_r()` $\boxed{8.3.9}$	`stdlib.h`
`strtok_r()` $\boxed{8.3.4}$	`string.h`
`asctime_r()` $\boxed{8.3.5}$ `ctime_r()` $\boxed{8.3.6}$ `gmtime_r()` $\boxed{8.3.7}$ `localtime_r()` $\boxed{8.3.8}$	`time.h`
Functions restricted to non-`NULL` *arguments*	
`ctermid()` `tmpnam()`	`stdio.h`

12.c Non-posix analogs

12.c.1 MACH Cthreads

The MACH coroutine implementation provides a much reduced thread-safe library, including the standard memory allocation functions `malloc()` and `free()`, the (non-POSIX) Unix function `select()`, and thread-safe versions of MACH's `msg_send()`, `msg_receive()` and `msg_rpc()` primitives. Other I/O calls may be made from multithreaded programs but if they block all other threads may block with them. The other MACH thread implementations suffer from no such difficulty, since MACH itself is basically thread-aware at the kernel level.

12.c.2 GATech Cthreads

Rather than replacing `libc`, GATech Cthreads has gone the route of introducing a new layer of explicitly thread-safe library functions. Threaded code may still call the original library code, but must provide its own exclusion.

stdio

Atomic output in GATech Cthreads is provided by the functions `aprintf()` and `afprintf()`. These calls are semantically identical to `printf()` and `fprintf()`, respectively, except that each is bracketed internally by calls to `aprintf_lock()` and `aprintf_unlock()`. The effect is rather like the pthreads interpretation of how `printf()` and `fprintf()` should work in a multithreaded program, in that this both ensures that the output from a single formatted output appears contiguously in the output stream, and provides a method for users to program their own output functions with similar 'atomic' behavior.

However, the signatures of the locking functions are `void aprintf_lock(void)` and `void aprintf_unlock(void)`. Note that these operations do *not* take a `FILE *` argument; they (and the `aprintf()` functions) manipulate a single, *global* lock. This means that only one thread may use the atomic output facility at a time, regardless of what stream it may be writing to. This slightly reduces the chance of deadlock over the POSIX lock-per-stream approach, but greatly increases lock contention in software that writes to multiple output streams.

The manual page does not specify whether `aprintf_lock()` can safely be called by a thread that already holds the lock, and consequently whether a thread that has called `aprintf_lock()` may safely call `aprintf()`. Presumably it cannot, and only traditional calls may be used between `aprintf_lock()/aprintf_unlock()` pairs.

Lower level output routines are not provided in GATech Cthreads; compare this decision, only to provide synchronizing variants of the most expensive operations, with that of POSIX in only providing *unlocked* versions of the *cheapest* operations. GATech Cthreads was designed for environments in which synchronization is relatively expensive.

There are no implicitly synchronized input operations in GATech Cthreads; this is not so unreasonable, since most programming methodologies will assign a single thread to each input channel at any given moment, even if output (especially of logging and debugging data) originates from several places.

memory allocation

Recall that under GATech Cthreads, most memory (including stack and all named variables) may be local to a node. Shared memory is allocated with the function

```
RESULT memory_alloc(memory_t *mem_ptr, int size, int node);
```

where the *node* is the 'home' node of the memory whose address is stored in **pointer*. Functions are provided to 'resize' memory areas (as with `realloc()` this may result in the object being copied to a new address), to free it, and to enquire as to its home node:

```
RESULT memory_realloc(memory_t *mem_ptr, int size);
void memory_free(memory_t mem);
int memory_node(memory_t mem);
```

The function `memory_node()` may only be called on memory objects allocated with `memory_alloc()` and returns either a node number or `-1` if the memory is not associated with a particular node (if, for instance, a given host treats all memory equally).

Chapter 13

Miscellanea

[[Odd interactions. . . ?]]

13.1 Interaction with `fork()/exec()`

[[Move this to the libc chapter?]]

[[`fork()` *is a synchronizing function, according to the standard. Not sure what this means, though. . . .]]*

13.1.1 Registering fork handlers

`_POSIX_THREAD_ATFORK`

`pthread_atfork()` **3.1.3⋆**

```
#include <sys/types.h> #include <unistd.h>  /* For _POSIX_THREAD_ATFORK
*/
#ifdef _POSIX_THREAD_ATFORK

int /*errno*/ pthread_atfork
  ( void (*pre)(void)
  , void (*parent_post)(void)
  , void (*child_post)(void)
  );
```

`void (*pre)(void)`

`void (*parent_post)(void)`

`void (*child_post)(void)`

On success

On failure `ENOMEM`$_{271}$ *[[@]]*

[[Under DCE, you have to make sure that there's only one thread live at `fork()` *time, manually!]]*

[[locale is per-process 8.1.2*]]*

13.2 The flavours of atomicity

*[[*POSIX *gotcha classification [x-safety] (does this go so early?—maybe it belongs in the thread-safety § of part III?)*

*Answer: it *does* belong here, but just briefly; then it should point forward to a more detailed explanation later on.]]*

[[`sysconf()`*?]]*

[[`fork()` only *clones the calling thread!!!* 3.1.1.2 *Note also that uses of* `fork()` *other than to* `exec()` *are* no longer supported *in pthread-providing implementations. See also* `pthread_atfork()`$_{241}$ 3.1.3.1*.]]*

Part III

Applications

Chapter 14

When to Use Threads

Chapter 15

Making Your Applications Threadable

[[2.3.9 lists problematic libfns.]]
[[Some problems arise from deadlock. Most arise from poor coding practises!]]
[[fpops not asynch-safe - 2.2.2.165.2/310]]
[[interface vs. implementation probs]]

15.1 Basics: How to compile for threads

[[Really a reprise of an earlier section!]]

15.2 Making you Libraries Threadable

15.3 Dealing with Dusty Decks

15.4 Summary of Gotchas and Solutions

Chapter 16

Example: Finger

16.1 Introduction

16.2 Finger program

net.h

```
extern void *netfinger(char *name);
```

net.c

```
#include <pthreadutil.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/param.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <stdio.h>
#include <ctype.h>

#ifndef MAXHOSTNAMELEN
#define MAXHOSTNAMELEN 64
#endif
/*
 * These globals are set initialy and then are only read.
 * They do not need mutexes.
 */
extern int lflag;
char myhostname[MAXHOSTNAMELEN];
```

```
/*
 * These globals change and therefore do need mutexes
 */
pthread_mutex_t spmutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t spcond = PTHREAD_COND_INITIALIZER;
struct servent *sp = NULL;

void netsetup(void)
  { pthread_mutex_lock(&spmutex);
    if (sp)
      { fprintf
          ( stderr
          , "finger: service pointer already initialized.\n"
          );
        exit(2);
      }
    if ((sp = malloc(sizeof(struct servent_answer))) == NULL)
      { fprintf
          ( stderr
          , "finger: Couldn't allocate service pointer.\n"
          );
        exit(2);
      }
    if
      ( getservbyname_r
          ( "finger"
          , "tcp"
          , (struct servent_answer *)sp
          ) == NULL
      )
      { fprintf(stderr, "finger: tcp/finger: unknown service\n");
        exit(2);
      }
    if (gethostname(myhostname, MAXHOSTNAMELEN))
      { fprintf(stderr, "finger: couldn't get my hostname.\n");
        exit(2);
      }
    pthread_cond_broadcast(&spcond);
    pthread_mutex_unlock(&spmutex);
  }

void netsetupwait(void)
  { pthread_mutex_lock(&spmutex);
    while(sp == NULL)
      pthread_cond_wait(&spcond, &spmutex);
    pthread_mutex_unlock(&spmutex);
  }

void *netfinger(char *name)
  { spg_atexit_ty atexit_id;
    register int c, lastc;
    struct in_addr defaddr;
```

```
struct hostent *hp;
struct sockaddr_in sin;
int s, i, readbuflen;
char readbuf[1024];
char *host;

netsetupwait();
spg_atexit_add(&atexit_id, fflush_nrv, NULL);

if (!(host = strrchr(name, '@')))
  host = myhostname;
else
  *host++ = '\0';
if (!(hp = gethostbyname(host)))
  { if ((defaddr.s_addr = inet_addr(host)) < 0)
      { fprintf
          ( stderr
          , "[%s] gethostbyname: Unknown host\n"
          , host
          );
        return;
      }
  }
sin.sin_family = hp->h_addrtype;
memcpy((char *)&sin.sin_addr, hp->h_addr, hp->h_length);
sin.sin_port = sp->s_port;

if ((s = socket(sin.sin_family, SOCK_STREAM, 0)) < 0)
  { sprintf(readbuf, "[%s]: socket", hp->h_name);
    perror(readbuf);
    return;
  }

/* have network connection; identify the remote host */
if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
  { sprintf(readbuf, "[%s]: connect", hp->h_name);
    perror(readbuf);
    close(s);
    return;
  }

/* -l flag for remote fingerd  */
if (lflag) write(s, "/W ", 3);
/* send the name followed by <CR><LF> */
write(s, name, strlen(name));
write(s, "\r\n", 2);

/*
 * Read from the remote system; once we're connected, we
 * assume some data.  If none arrives, we hang until the
 * user interrupts, or until the thread timeout expires.
 *
 * If we see a <CR> or a <CR> with the high bit set,
```

```
     * treat it as a newline; if followed by a newline
     * character, only output one newline.
     *
     * Otherwise, all high bits are stripped; if it isn't
     * printable and it isn't a space, we can simply set the
     * 7th bit.  Every ASCII character with bit 7 set is
     * printable.
     */
    flockfile(stdout);
    for
      ( readbuflen = read(s, readbuf, 1024)
      , lastc = '\n'
      , printf("[%s]\n", hp->h_name)
      ; readbuflen > 0
      ; readbuflen = read(s, readbuf, 1024)
      )
      { for (i = 0; i < readbuflen; i++)
          { c = readbuf[i] & 0x7f;
            if (c == 0x0d)
              { c = '\n';
                lastc = '\r';
              }
            else
              { if (!isprint(c) && !isspace(c))
                  c |= 0x40;
                if (lastc != '\r' || c != '\n')
                  lastc = c;
                else
                  { lastc = '\n';
                    continue;
                  }
              }
            putchar_unlocked(c);
          }
      }
    if (lastc != '\n')
      putchar_unlocked('\n');
    funlockfile(stdout);
  }
```

finger.c

```
#include <pthreadutil.h>
#include <sys/param.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
```

```
void usage(int eval)
  { fprintf
      ( stderr
      , "usage: finger [-lps] [-c <net_count>] [-t|T <timeout>]"
        " [-f <filename>] [login ...]\n"
      );
    exit(eval);
  }

/*
 * These globals are set initialy and then are only read.
 * They do not need mutexes.
 */
int thread_time = 0, program_timeout = 0, lflag = 0;
spg_tad_ty parse_file_tad;
spg_tad_ty netfinger_tad;

void *timeout_thread(void *arg)
  { sleep(program_timeout);
    exit(0);
  }

void *signal_thread(void *arg)
  { int sig;
    sigset_t  program_signals;

    sigemptyset(&program_signals);
    sigaddset(&program_signals, SIGINT);
    sigwait(&program_signals, &sig);
    exit(0);
  }

#ifndef MAXHOSTNAMELEN
#define MAXHOSTNAMELEN 64
#endif

void *parse_file(void *arg)
  { char hostname[MAXHOSTNAMELEN];
    char *filename = arg;
    spg_atexit_ty atexit_id;
    pthread_attr_t attr;
    pthread_t thread_id;
    char *thread_arg;
    FILE *fp;
    int len;

    netsetupwait();

    /* Parse the file and create a thread per connection */
    if ((fp = fopen(filename, "r")) == NULL)
      { fprintf(stderr, "Can't open file %s\n", filename);
        pthread_exit(NULL);
      }
    spg_atexit_add(&atexit_id, fclose_nrv, fp);
```

```
    if (pthread_attr_init(&attr))
      { fprintf
          ( stderr
          , "Error: Can't initialize thread attributes\n"
          );
        exit(2);
      }
    spg_atexit_add
      (&atexit_id, spg_attr_destroy_nrv, &attr);

    while (fgets(hostname, MAXHOSTNAMELEN, fp))
      { if
          ( (thread_arg = malloc(len = strlen(hostname)))
          ==NULL
          )
          { fprintf(stderr, "Error: out of memory\n");
            exit(2);
          }
        hostname[len - 1] = '\0';
        strcpy(thread_arg, hostname);
        pthread_attr_setcleanup(&attr, free, thread_arg);
        if
          ( spg_tad_create
              ( &netfinger_tad
              , &thread_id
              , NULL
              , netfinger
              , thread_arg
              )
          )
          { fprintf
              ( stderr
              , "Error: spg_tad_create() netfinger_tad.\n"
              );
            exit(2);
          }
      }
    pthread_exit(NULL);
  }

main(int argc, char **argv)
  { spg_atexit_ty atexit_id;
    pthread_t thread_id;
    int max_count = 0;
    char ch;

    /* getopt variables */
    extern char *optarg;
    extern int optind;

    /* Setup tad for parse_file() threads */
    if (spg_tad_init(&parse_file_tad, max_count))
```

```
    { fprintf
        ( stderr
        , "Error: couldn't create parse_file() TAD.\n"
        );
      exit(1);
    }
  while ((ch = getopt(argc, argv, "c:f:t:T:ls")) != (char)EOF)
    switch(ch)
      {
    case 't':        /* Time to let each thread run */
        if ((thread_time = atoi(optarg)) <= 0)
          usage(1);
        break;
    case 'T':        /* Time to let entire program run */
        if ((program_timeout = atoi(optarg)) <= 0)
          usage(1);
        break;
    case 'f': /* Parse file for list of places to finger */
        if
          ( spg_tad_create
              ( &parse_file_tad
              , &thread_id
              , NULL
              , parse_file
              , optarg
              )
          )
          { fprintf
              ( stderr
              , "Error: spg_tad_create() parse_file_tad.\n"
              );
            exit(1);
          }
        break;
    case 'c':
        max_count = atoi(optarg);
        break;
    case 'l':         /* long format */
        lflag = 1;
        break;
    case 's':         /* short format */
        lflag = 0;
        break;
    case '?':
        usage(0);
    default:
        usage(1);
      }

  /* The rest of the argumants are hosts */
```

```
argc -= optind;
argv += optind;



/* Setup timeout thread, if there is one */
if (program_timeout)
  { if
      ( pthread_create
          (&thread_id, NULL, timeout_thread, NULL)
      )
      { fprintf
          ( stderr
          , "Error: couldn't create"
            " program_timeout() thread\n"
          );
        exit(1);
      }
  }



/* Setup cleanup thread for signals */
if (pthread_create(&thread_id, NULL, signal_thread, NULL))
  { fprintf
      ( stderr
      , "Error: couldn't create signal_timeout() thread\n"
      );
    exit(1);
  }



/* Setup tad for netfinger() threads */
if (spg_tad_init(&netfinger_tad, max_count))
  { fprintf
      ( stderr
      , "Error: couldn't create netfinger() TAD.\n"
      );
    exit(1);
  }



/* Setup the net and let everyone run */
netsetup();



while (*argv)
```

```
      { if
          ( spg_tad_create
              ( &netfinger_tad
              , &thread_id
              , NULL
              , netfinger
              , *argv
              )
          )
        { fprintf
            ( stderr
            , "Error: spg_tad_create() netfinger_tad.\n"
            );
          exit(2);
        }
        argv++;
      }
    spg_tad_wait(&parse_file_tad, 0);
    spg_tad_wait(&netfinger_tad, 0);
    exit(0);
  }
```

16.3 Finger daemon

16.4 Discussion

Chapter 17

(Multiprocessor Example)

Chapter 18

Debugging Techniques

"Deadlocks are your Friend" - Nawaf Bitar

18.1 Introduction

18.2 [...]

18.3 Summary

Appendices

[[sps: MACH Cthreadsprovides three implementations, one of which is user-level *and non-pre-emptive*, one of which provides a MACH thread per user thread, and one of which runs each thread in a separate 'task' [process]. The MACH developers advise you to debug your application *first* in the user-level (`-lco_threads`) environment, which is repeatable, stdio-compatible, and more compatible with traditional debuggers, and only switch to one of the other two implementations after that seems to be running smoothly.]]

Chapter 19

Optimization Hints

19.1 Introduction

19.2 Locks

19.3 Priorities

19.4 Scheduling Attribute

19.5 [...]

19.6 Summary

Part IV

Appendices

Appendix A

Conventions of the Pthreads Package

The names introduced by the pthreads standard for types, functions and constants follow a systematic naming scheme [2.1.2]. They are built up from the facility name (typically '`pthread`'), an optional sub-facility (a type, other than the `pthread` itself, around which a number of functions are organized), an entity or operation name, and an optional modifying suffix marking a special property that the named object has—in particular, non-portable extensions to the pthreads standard can take `pthread_` names, but only if they end in `_np`.

name	*[_]facility[_subfacility]_thing[_modifier]* *[_]facility[_setting]_value[_modifier]*	
facility	`pthread`	The pthreads facility
facility	`sched`	Scheduling
subfacility	`attr`	Attributes descriptor
	`mutex`	*Mut*ual *ex*clusion lock
	`mutexattr`	Mutex attributes descriptor
	`cond`	Condition variable
	`condattr`	Condition attributes descriptor
	`cleanup`	Cleanup handler
	`key`	Thread-local storage key
	`once`	Once-only initialization object
thing	`t`	Type
(or function)	`initializer`	Static initializer
	`init`	Initialization routine
	`destroy`	Termination routine
	`get`*attribute*	Read setting
	`set`*attribute*	Update setting
	...	...other function names
setting	`cancel`	Cancelation behavior
	`create`	Initial thread state
	`process`	Interprocess shareability
	`sched`	Scheduling policy
	`scope`	Scheduling 'scope'
modifier	`r`	Re-entrant variant
	`unlocked`	Fast, non-interlocked variant
	`np`	Non-portable extension
value	...	...specific values for settings

Following (subtly incomprehensible) C tradition, the names of preprocessor constants—primarily those identifiers following the second format—are written in `UPPERCASE` throughout.

For example:

`pthread_t`	The type of pthreads
`pthread_create`	Make a new thread
`pthread_mutex_init`	Initialize a mutual exclusion lock
`pthread_attr_setdetachstate`	Set thread attributes to 'detached'
`PTHREAD_MUTEX_INITIALIZER`	Static initializer for mutexes
`SCHED_FG_NP`	'Foreground' scheduling (DCE-specific, but POSIX-compatible)

The POSIX threads facility provides a number of opaque data types catering to different aspects of multithreaded programming. As we saw earlier, threads (of type `pthread_t`$_{32}$) themselves are called into being with the `pthread_create()`$_{33}$ operation, live until they exit (or are **canceled**), and have their resources freed when either immediately (if they are *detached*$_{32}$) or when another thread picks up their exit status by calling `pthread_join()`$_{40}$.

The other, more conventional, types (such as `pthread_mutex_t`$_{68}$ and `pthread_condattr_t`$_{138}$) follow a related, but simpler, protocol.

These objects are stored in variables that must be initialized before use. In only two cases (`PTHREAD_MUTEX_INITIALIZER`$_{70}$ and `PTHREAD_COND_INITIALIZER`$_{81}$) are static initializers provided, and even these allow only basic 'default' objects to result. In general, an `_init()` function must be called dynamically to initialize the variable before it can be used. Then, once the object has served its purpose, it can be returned to its uninitialized state by calling the corresponding `_destroy()` function. It is undefined what happens if you attempt to initialize an initialized object, or uninitialize an uninitialized object.

This protocol allows the implementation maximum freedom to choose an appropriate representation for each type.

Part of the motivation for requiring explicit dynamic initialization of all (non-default) objects is to give implementors a good, extensible method for specifying the attributes of objects that are built. This is accomplished through the use of **attributes objects**.

Attributes objects

[[**sps: Christoph says that the intro to chapter 8 is clearer**]]

Attribute objects themselves follow the above `_init()`/`_destroy()` protocol. They group together all of the various parameters of other thread-related types in a single object that you can store in a variable.

Their initialization function takes no extra arguments, and their initial state corresponds to the implementation's default attributes for the object type they control. Set (and get) operations can then be used on the attributes objects to tailor them to the application's needs—objects built using the specialized attributes objects will all have the chosen properties.

This method is used (rather than the traditional approach of passing a number of specialized flags and parameters directly to the object creation functions) to make it easier for vendors (or indeed subsequent revisions of the standard) to extend the set of available attributes without forcing modifications to existing code. Their main problems are that they are fairly verbose and (largely due to the limitations of the C preprocessor) there is no effective way to allow static initialization of objects under this regime (except, as we have said, for the absolute default[a]).

They have one important indirect advantage: because the initialization operation provides a default attributes object, and because `_get()` operations are provided to read the current values stored in one, it is possible to determine the defaults an implementation applies simply by examining a fresh attributes object.

By the way, one other way of obtaining an object with default attributes—useful especially for threads themselves, which cannot be constructed statically anyway—is to pass a `NULL` pointer in place of an attributes object. This saves a little effort in the case where dynamic initialization is required but no special control need be exerted over the characteristics of the objects built. It helps keep the simple cases simple: in fact, many programs you write will contain no attribute objects at all.

Attributes are discussed in more detail later on.$\Rightarrow^{8.2}_{138}$

[a] Even this may require special processing by the implementation, depending on how things are managed. The standard contains a couple of suggestions for 'dirty tricks' that the pthreads implementor can use to meet the requirement for static initializers without excessive cramping of implementational style.

The calling conventions of the POSIX thread functions have been modified from the usual conventions of the Unix world because a fundamental difficulty with global variables in multithreaded programs (of which more later⇒$^{5.2}_{64}$) makes the traditional method of reporting errors by returning a bogus value and setting `errno` inefficient. Instead, most pthreads operations return an error status (or zero on success) *as their result*; any 'outputs' (in the conventional sense) are assigned to reference parameters. Thus the prototypical argument list consists of three parts: (1) a 'target' object; (2) zero or more parameters; and (3) zero or more pointers to 'result' variables. The return value encodes only status.

There are a few exceptions; the macros **`pthread_cleanup_push()`**$_{122}$ and **`pthread_cleanup_pop()`**$_{122}$ do not behave like functions at all, and a few 'lightweight' operations such as **`pthread_getspecific()`**$_{119}$ and **`getchar_unlocked()`**$_{226}$ have limited error reporting and return their results directly as an optimization.

In addition to the completely new functions that provide functionality for multithreaded programming, pthreads provides a few **re-entrant** replacements for existing POSIX functions, those that are particularly ill-suited for use with multiple threads of control. These are named after their originals by the addition of the suffix `_r`, but have been given calling conventions more in line with the new model and return a status rather than setting `errno`.

One final thing that should be noted about the design of pthreads is that each function has been designed to do only *one* thing (there are very few 'mode' parameters) and that the Standard C type system has been carefully exploited. Most arguments receive their 'true' types (rather than, say, an arbitrarily selected type of `int` or `char *`); those that manipulate data on behalf of the user, on the other hand, specify a type of `void *` in order to impose the least possible constraints consistent with portability. (There's a short summary of the relevant points of Standard C in the appendix to this chapter.⇒$^{3.5}_{26}$)

As we've just noted, the pthreads functions never set `errno`, but directly return an error code instead. The codes returned, however, are the same ones as are used by older, `errno`-setting routines. Pthreads in fact adds no new error codes; here is a list of the codes that it does use, with a brief description of each:

`EAGAIN` The operation cannot be completed at this time, possibly because needed resources were unavailable. If you try again later, however, it may succeed.

`EBUSY` The operation could not be completed because a resource was busy.

`EDEADLK` Performing this operation would result in deadlock.

`EINTR` The operation was interrupted before it could complete.

`EINVAL` One of the arguments was invalid.

`ENOMEM` The operation could not be completed because there was not enough memory available.

`ENOSYS` This operation is not supported by the current implementation. If you call a function that POSIX defines (perhaps as an option) but your implementation does not provide, what *should* happen is that `ENOSYS` is returned. [[**sps: Seems like a damned nuisance to me; I'd rather know statically that there was a problem. Or have I misunderstood this?**]]

`ENOTSUP` One of the arguments specifies has a value not supported by this implementation.

`EPERM` You do not have the privileges necessary to perform this operation.

`ERANGE` Traditionally, a numerical function was supplied with an inappropriate argument. In pthreads, a re-entrant library function was passed a buffer whose size was inadequate to the data to be returned. Retry with a larger buffer.

`ESRCH` The thread or process specified does not exist.

`ETIMEDOUT` The operation timed out.

One thing to remember when working with the *non*-pthreads functions in a multithreaded program is that Standard C (and lately POSIX) *requires* that you `#include <pthread.h>` before referring to `errno`. If you do not, your multithreaded program may not compile and will not work correctly.

The reason for this is that (for reasons we will see later) each thread needs a private copy of `errno`, and the only simple way to do this is to exploit some of the flexibility allowed to the implementor by the C standard and make `errno` into a macro.

Under pthreads `errno` is no longer a variable.

If you do forget to include `pthread.h`273—which contains the correct definitions—your program will not work.

Appendix B

What's in `pthread.h`

In this section we provide a lightening tour of the contents of the pthreads header file, `pthread.h`. First off, it is important to note that not all of the things that *might* be present in a standard threads implementation will necessarily be present in the one you are using; indeed, support for pthreads itself is an option. In order to help write portable code, the header file `unistd.h` defines symbols naming options: if a preprocessor symbol in the following table is defined, the corresponding feature is (supposedly) available. [2.9.3]

	Feature test symbol	Description
	`_POSIX_REENTRANT_FUNCTIONS`$_{??}$	Re-entrant functions
	`_POSIX_THREADS`$_{??}$	Multithreading facilities
$\triangle$	`_POSIX_THREAD_ATTR_STACKADDR`$_{146}$	User-specifiable thread stack addr.
$\diamond$	`_POSIX_THREAD_ATTR_STACKSIZE`$_{145}$	User-specifiable thread stack sizes
$\natural$	`_POSIX_THREAD_ATFORK`$_{241}$	`pthread_atfork()`$_{241}$ supported
$*$	`_POSIX_THREAD_PRIORITY_SCHEDULING`$_{158}$	Priority scheduling for threads
$\dagger$	`_POSIX_THREAD_PRIO_INHERIT`$_{175}$	Inherited mutex priority
$\ddagger$	`_POSIX_THREAD_PRIO_PROTECT`$_{175}$	Protected mutex priority
$\circ$	`_POSIX_THREAD_PROCESS_SHARED`$_{148}$	Synchronization in shared memory

The indentation reflects the logical arrangement of these symbols: for instance, every POSIX implementation that supports the pthreads functions should also provide re-entrant library functions (including the newly revised functions whose names end in `_r`$\Rightarrow^{12}_{219}$). Similarly, if an implementation does *not* support pthreads, it follows that it will not let you choose the size of a thread's stack—and `_POSIX_THREAD_ATTR_STACKSIZE` will not be defined.

Each of the optional features within the threads package has been given a symbol—displayed in the first column of the figure—used in the tables below to indicate that some function or constant depends on the presence of that option for its existence.

Now we proceed to the symbols defined within `pthread.h` itself (assuming at least `_POSIX_THREADS` is defined).

Threads themselves are represented by objects of type `pthread_t`. The functions operating directly on this type are listed in the following table (though *many* other functions operate on threads indirectly by taking effect specifically in the calling thread). The right-hand column lists symbolic constants related to the functions on the left.

pthread_t$_{32}$	
pthread_create()$_{33}$	
pthread_exit()$_{37}$	
pthread_join()$_{40}$	PTHREAD_CANCELED$_{204}$
pthread_equal()$_{43}$	
pthread_self()$_{42}$	
pthread_cancel()$_{204}$	
pthread_setcancelstate()$_{208}$	PTHREAD_CANCEL_DISABLE$_{208}$ PTHREAD_CANCEL_ENABLE$_{208}$
pthread_setcanceltype()$_{209}$	PTHREAD_CANCEL_DEFERRED$_{210}$ PTHREAD_CANCEL_ASYNCHRONOUS$_{210}$
pthread_testcancel()$_{205}$	
pthread_kill()$_{190}$	
pthread_sigmask()$_{193}$	
* pthread_getschedparam()$_{170}$ * pthread_setschedparam()$_{170}$	
pthread_getspecific()$_{119}$ pthread_setspecific()$_{118}$	
pthread_cleanup_push()$_{122}$ pthread_cleanup_pop()$_{122}$	

These functions provide for basic thread manipulation$\Rightarrow^{4}_{31}$, **cancelation** (*i.e.* termination of a thread from without) and control thereof$\Rightarrow^{11}_{203}$, handling of POSIX signals $\Rightarrow^{10}_{183}$, dynamic changes to thread scheduling priority$\Rightarrow^{9}_{157}$, and manipulation of the data pertaining to a thread$\Rightarrow^{6}_{113}$.

The next major opaque type defined in pthread.h describes the **attributes** with which threads are created.

pthread_attr_t$_{138}$	
pthread_create()$_{33}$	
pthread_attr_init()$_{140}$ pthread_attr_destroy()$_{141}$	
pthread_attr_getdetachstate()$_{142}$ pthread_attr_setdetachstate()$_{142}$	PTHREAD_CREATE_DETACHED$_{143}$ PTHREAD_CREATE_JOINABLE$_{143}$
△ pthread_attr_getstackaddr()$_{146}$ △ pthread_attr_setstackaddr()$_{146}$	
◇ pthread_attr_getstacksize()$_{145}$ ◇ pthread_attr_setstacksize()$_{145}$	
* pthread_attr_getinheritsched()$_{161}$ * pthread_attr_setinheritsched()$_{161}$	* PTHREAD_EXPLICIT_SCHED$_{161}$ * PTHREAD_INHERIT_SCHED$_{162}$
* pthread_attr_getscope()$_{163}$ * pthread_attr_setscope()$_{163}$	* PTHREAD_SCOPE_PROCESS$_{164}$ * PTHREAD_SCOPE_SYSTEM$_{164}$
* pthread_attr_getschedpolicy()$_{165}$ * pthread_attr_setschedpolicy()$_{165}$	* SCHED_FIFO$_{166}$ * SCHED_RR$_{166}$ * SCHED_OTHER$_{166}$
pthread_attr_getschedparam()$_{168}$ pthread_attr_setschedparam()$_{168}$	struct schedparam??

The functions operating on thread attribute objects provide for initialization and finalization, manipulation of **detach state** (determining whether synchronization with the terminated thread is desired) and custom stack allocation$\Rightarrow^{8}_{137}$; and various parameters relating to thread scheduling$\Rightarrow^{9}_{157}$.

Mutual exclusion locks are the main means by which POSIX threads synchronize for the exchange of data. They are a special data structure which only one thread may “own” at a time. They are represented by objects of type `pthread_mutex_t`.

pthread_mutex_t$_{68}$	PTHREAD_MUTEX_INITIALIZER$_{70}$
pthread_mutex_init()$_{70}$ pthread_mutex_destroy()$_{72}$	
pthread_mutex_lock()$_{72}$ pthread_mutex_trylock()$_{72}$ pthread_mutex_unlock()$_{75}$	
‡ pthread_mutex_getprioceiling()$_{178}$ ‡ pthread_mutex_setprioceiling()$_{178}$ pthread_cond_wait()$_{83}$ pthread_cond_timedwait()$_{83}$	

The functions on mutexes allow them to be created and destroyed, and locked and unlocked$\Rightarrow^{5.2}_{64}$; and provide for dynamic manipulation of their interactions with the scheduling system$\Rightarrow^{9.5}_{172}$.

The condition-wait functions (which allow threads to wait indefinitely for some user-defined event) are listed here as well as under `pthread_cond_t` since they involve both a mutual exclusion lock and a condition variable.

Mutexes, too, are first set up using attributes objects:

pthread_mutexattr_t$_{138}$	
pthread_mutex_init()$_{70}$	
pthread_mutexattr_init()$_{140}$ pthread_mutexattr_destroy()$_{141}$	
∘ pthread_mutexattr_getpshared()$_{148}$ ∘ pthread_mutexattr_setpshared()$_{148}$	∘ PTHREAD_PROCESS_PRIVATE$_{149}$ ∘ PTHREAD_PROCESS_SHARED$_{149}$
pthread_mutexattr_getprioceiling()$_{176}$ pthread_mutexattr_setprioceiling()$_{176}$	
pthread_mutexattr_getprotocol()$_{174}$ pthread_mutexattr_setprotocol()$_{174}$	PTHREAD_PRIO_NONE$_{175}$ † PTHREAD_PRIO_INHERIT$_{175}$ ‡ PTHREAD_PRIO_PROTECT$_{175}$

Mutex attributes objects can be created and destroyed, and have functions that allow you to request mutexes that are shareable between processes using shared memory $\Rightarrow^{8.4}_{147}$, and to set the initial values of the parameters relating to scheduling$\Rightarrow^{9.5}_{172}$.

Condition variables also provide synchronization between threads, but unlike mutexes (which ensure exclusive short-term access to some resource), condition variables are used to wait, possibly for a long time, for some condition to become true as determined by other threads in the computation.

pthread_cond_t$_{80}$	PTHREAD_COND_INITIALIZER$_{81}$
pthread_cond_init()$_{81}$ pthread_cond_destroy()$_{82}$	
pthread_cond_wait()$_{83}$ pthread_cond_timedwait()$_{83}$ pthread_cond_signal()$_{86}$ pthread_cond_broadcast()$_{86}$	

The operations on condition variables permit waiting for them to become true (with or without a timeout), and **signaling** to either one or all (‘broadcasting’) threads waiting for a condition that it now holds.$\Rightarrow^{5.4}_{78}$

Condition variables, like mutexes, are created using attributes objects, but only one attribute is currently defined by the standard:

pthread_condattr_t$_{138}$	
pthread_cond_init()$_{81}$	
pthread_condattr_init()$_{140}$ pthread_condattr_destroy()$_{141}$	
○ pthread_condattr_getpshared()$_{148}$ ○ pthread_condattr_setpshared()$_{148}$	○ PTHREAD_PROCESS_PRIVATE$_{149}$ ○ PTHREAD_PROCESS_SHARED$_{149}$

The **pshared** attribute of a condition variable, like that of a mutex, specifies whether it is desirable for this condition variable to be able to work between processes, through shared memory.

POSIX provides each thread with a means of storing data locally to the thread but globally to all code that is called by that thread. Because C variables can't be made to perform this function (C has no notion of a 'thread', as such), another opaque data type, `pthread_key_t`, is involved in providing this service. A **key** fills a rôle rather like the name of a conventional variable: it designates a place, private to each thread, where data can be stored.

pthread_key_t$_{114}$	
pthread_key_create()$_{115}$ pthread_key_delete()$_{117}$	
pthread_setspecific()$_{118}$ pthread_getspecific()$_{119}$	

Thread-specific data keys can be created and deleted, and values can be stored under them and later retrieved.$\Rightarrow^{6.2}_{114}$

The type `pthread_once_t` provides a small facility designed to help in initializing multithreaded library code.

pthread_once_t$_{90}$	PTHREAD_ONCE_INIT$_{89}$
pthread_once()$_{89}$	

This initialization code can be invoked with `pthread_once()`.$\Rightarrow^{5.5}_{89}$

Finally, the function `pthread_atfork()`, while part of the pthreads facility, is not associated with any data type in particular. Rather, it helps control the behavior of a multithreaded program during a call to the operating system `fork()` operation.

♮ pthread_atfork()$_{241}$	

Appendix C

Varieties of Host Environment

C.1 Kernel-Level versus User-Level Threads

C.2 Multiprocessor Considerations

[[The problem is basically that caches are divided into ***cache lines****, small, fixed-sized blocks of fast memory that hold copies of recently used areas of main memory. It is to these that most of the CPU's memory accesses really go. Of course, there is less cache than there is main memory—otherwise you'd use it for everything—and so cache lines have to be re-used to hold data from different addresses. One method of cache management* ***writes back*** *any changes in the old contents of a cache line before loading in the new data, but the effect of that is that changes are made in main memory only when a cache line is recycled, not when the processor executes the original store instruction, and this is completely unpredictable in practise. To make synchronisation possible—and so that a process can be put in a stable state before switching processes—a means is provided to write back* all *of the cache lines, but that amounts to doing all of the main memory accesses that the cache hardware has been trying to defer, and is sufficiently expensive that you wouldn't want it to happen when it wasn't absolutely necessary.*

Another approach to cache management, used in so-called ***write-through*** *caches, only caches memory reads; write instructions go right ahead and update main memory, which isn't as slow as it sounds since the processor needn't wait for a write to complete. Unless the hardware provides sophisticated interlocks, though, this still results in a problem for systems with multiple physical processors, since when one processor updates a memory location, others may be left with* ***stale*** *copies in their caches—ones that date from before the change. This might be tolerable as a kind of 'communication delay' between the processors, except that the unpredictable way in which cache lines are re-used means that different processors won't always see changes in main memory in the same order.*

There are fairly good hardware solutions to these problems of ***cache coherence****, but the fast ones are complex and expensive the number of processors is large, so not all systems employ them. At any rate,* POSIX *threads are designed on the assumption that the hardware can perform data synchronisation on request, but that it doesn't necessarily happen automatically.]]*

C.3 Advanced Hardware Support

Appendix D

Bindings in Other Languages

D.1 Ada

D.2 Concurrent-C

Appendix E

Implementation Availability

E.1 Introduction

E.2 capthreads (NetBSD, FreeBSD, Linux, . . .)

E.3 DCE Threads (Windows NT, HP-UX, OSF/1, . . .)

E.4 Cthreads (Mach)

E.5 Threads (OS/2)

E.6 [...]

E.7 Summary

Appendix F

Aspects of the IEEE POSIX 1003.4a Specification

Annotated Bibliography

[with short comments about most entries – what's useful in each, etc.]

[[Stevens:Unix: Stevens, Richard W. Advanced Programming in the UNIX Environment Addison-Wesley 1992 ISBN 0-201-56317-7

Stevens, Richard W. UNIX Network Programming

Standard C

IEEE Std 1003.4a-199X Portable Operating System Interface(POSIX) - Draft 8 Part 1: System Application Program Interface(API) - Amendment 2:Threads Extension [C Language]

[[sps: Unlike many standards documents, this can be recommended as a source to real programmers—and not just ones in search of a purer masochistic thrill. It even contains useful advice on how the pthreads system might be used or implemented. What it gains in readability it loses in formality, of course—but it probably comes out even in the end.]]

IEEE Std 1003.1-1990 Portable Operating System Interface(POSIX) - Part 1: System Application Program Interface(API) - [C Language]

"Programming for the Real World POSIX.4 – Bill O Gallmeister

]]

[[Lim:Agarwal:ReactiveSynch Lim, Beng-Hong & Agarwal, Anant. Reactive Synchronization Algorithms for Multiprocessors. *In ASPLOS-IV (SIGPLAN Notices 209:11 Nov94, SIGOPS Operating Systems Review 28:5 Dec94, SIGARCH Computer Architecture News 22:Special Issue Oct94)p25]]*

Index of Identifiers

Index

Authors' Notes

[["Eligible for Processing Resources"]]

- NetBDS pthreads ⇒ ???
- Things to work in:
  - Note that C *locale* is per-process.
  - crossrefs (in manpages)
  - config constants (in manpages): add #ifdefs throughout
  - the *real* need for joinable is to find out that it is safe to free user-allocated stacks!
  - Find somewhere logical to put sections summarising the default attributes of eachkind of object. Maybe this should be an appendix.
  - Factor general introduction from posix-intro better in some of the chapters of section II.
- Trivia
  - Dorado 0-cycle context switch
  - load-linked/store-conditional (Alpha, new MIPS?)
  - Memory hierarchy issues
  - Other execution models (Oz soup)
  - Other communication models (lazy term transit)
  - Type-based synchronisation
  - Language integration issues
  - Paritioning (manual/automatic - FORTRAN XX!)
  - Synch signals (foll. POSIX): ABRT BUS ILL FPE PIPE SEGV
- Unix hatage
  - [[**cap: The whole signal concept is very basic because it tries to maintain backward compatibility with standard POSIX signal semantics [Which suck rocks if you ask me, and since you didn't I'll, just add that**

UNIX should never have grouped asycronous signals such as SIGHUP, SIGSUSPEND, with syncronous signals such as SIGBUS, SIGILL, and SIGALRM. It's totally ludicrous for me to do kill(pid, SIGILL).]]]

- File Descriptors

  File descriptors are inherited accross fork()/exec() invocations which is fine, but almost all the attributes are shared between the different processes, including the flags FASYNC, O_FSYNC, and O_NONBLOCK. The last one is the one that has really thrown a monkey wrench into the thread implementation. I have no idea why these flags are shared between processes (It can't be that much more efficiency because the close on exec flag is not shared, so what planet were these people on when they thought that sharing O_NONBLOCK was a good idea.) Too make matters worse, there is no call that gets you a new fd which doesn't share these attributes. Some versions of UNIX have gone out of there way to add /dev/fd/ directory which contains files 0 through n and when opened do open the same file as the corresponding fd does in you're process but the newfd opened doesn't share any attribute from the old fds. (This is one of the grossest hacks I've seen to compensate for a moronic design in the first place, what's worse is POSIX standardized on the first have of this problem in .1)

- The second bogus thing about the fd flags is the flag O_NONBLOCK on a regular file is ignored. The kernel will block the process until the file read or write was completed regardless of the state of the flag. (When I say do not block I mean DO NOT BLOCK. Which word of "DO NOT BLOCK" do you not understand.) [[**sps: Is this behaviour specced, or simply permitted?**]]

[[**cap: Gerneral Structure Notes**

a Reduce the number of chapters in Section I b Do we want a "UNIX-HATERS" section at the end of chapters or incorpoate notes as foot notes.
]]

An important thing to remember about POSIX is that it is a *standard,* and not a particular piece of software. Different software developers will interpret the standard differently (which is bad, because it means that the standard was too vague), and they will make use of the lattitude deliberately built into the standard to produce very different implementations that may not behave the same at all (which is good, if done well, because it can greatly improve performance and give you access to the idiosyncratic abilities of different machines). And of course, what you read in this book is its authors' own interpretation of the standard too. If it's important, there are two places you should be checking besides this volume: for the official story, the text of standard itself [**?**]; and for real information about how your implementation works, the relevant manpages.

Systematic cross-references in the manpages would be good! But best to do that in a big pass.

Check all examples to make sure that we follow our advice and document what each mutex and cv is for, and which variables are protected by each mutex!!!!

CHECK ALL CHAPTER SUMMARIES!!!!

TIGHTEN CHAPTER INTROS!

We may need to talk about shared memory somewhere; it's another new concept and it comes up two or three times.

What does the standard want us to do if we try to create a thread at a priority we don't have privileges for? DYnamic change can return EPERM but neither create nor attr_setchedparam has this as a listed error.

Check that the final standard has not introduced cancelation points into the lock I/O functions.

Check that pthread_yield() and sched_yield() haven't been made separate again; with two-level scheduling they have different meanings!!!!

Check whether PRIO_PROTECT now MUST be at least as high as a locking thread's prio, or if this is only advice.

Weirdness in the standard: is an ignored, masked, sigwaited signal delivered or not? I can't tell.

More weirdness: mq's persist but are not ls'able bc of their parallel "filesystem".

`sched_get_priority_min()sched_get_priority_max() f(SCHED_xxx).`

We might want to look at Summary vs. Discussion in the various chapters. Most chapters need a Summary, even if they have a Discussion as well.

Appendices

F.a Displaced material

F.b Package conventions

[[**sps: Abbreviate this or move some to chappendix**]]

By now you may already have noticed some of the conventions according to which pthreads has been designed, especially as regards the choice of names for functions. In this section we outline the full naming scheme, describe the general calling conventions of the thread-related functions, and sketch the **attributes** mechanism that is used to specify the parameters of the major pthreads objects. While a full understanding of this material is certainly not necessary for programming with threads under POSIX—one can always look in the manpage for the usage notes—knowing the (unusually consistent) conventions may help you remember the details more easily. Then again, the reader unimpressed by excessive systematicity may wish to skip to the next section.

F.c Life cycle of the pthread

We now examine the life cycle of the pthread in more detail. A thread is represented by an explicit, manipulable type of object, `pthread_t`$_{32}$. Threads, the members of this type, are called into existence by invoking `pthread_create()`$_{33}$, passing the **attributes** of the new thread and a function for it to execute (see figure F.1). Subject to the requirements of the scheduling policy and so on, it will immediately start running (this differs from some other thread systems that provide a separate startup operation); it will be made either *joinable*$_{32}$ or *detached*$_{32}$ depending on its attributes. The thread then survives as an independent flow of control until it is **canceled** from the outside with `pthread_cancel()`$_{204}$ or **exits** either implicitly by returning from its initial function or explicitly by calling `pthread_exit()`$_{37}$ (it also ceases to exist if the process exits or `fork()`s, but that's another matter).

The difference between a *joinable* and a *detached* thread is that a joinable thread, when it exits, may return a value to be picked up by another thread with `pthread_join()`$_{40}$.[1] Not until this final synchronization has taken place is the thread actually deleted. Detached threads, on the other hand, are deleted as soon as they complete their termination processing. A joinable thread may become detached by calling `pthread_detach()`$_{41}$; the reverse is not possible within the pthread model.

F.c.1 Execution states

At a finer level of detail, throughout its active life—that is, while it is *joinable* or *detached*—a thread moves between the three scheduling states *runnable*$_{294}$, *running*$_{294}$ and *blocked*$_{294}$:

running A thread that is *running* is actually using the CPU; it is the currently active thread. It can stop running either because it **blocks**, waiting for some event to occur or data to arrive—in which case it becomes *blocked*; or because it **resigns** the processor or is **pre-empted** by the thread scheduler, in which case it remains *runnable* (but is no longer *running*).

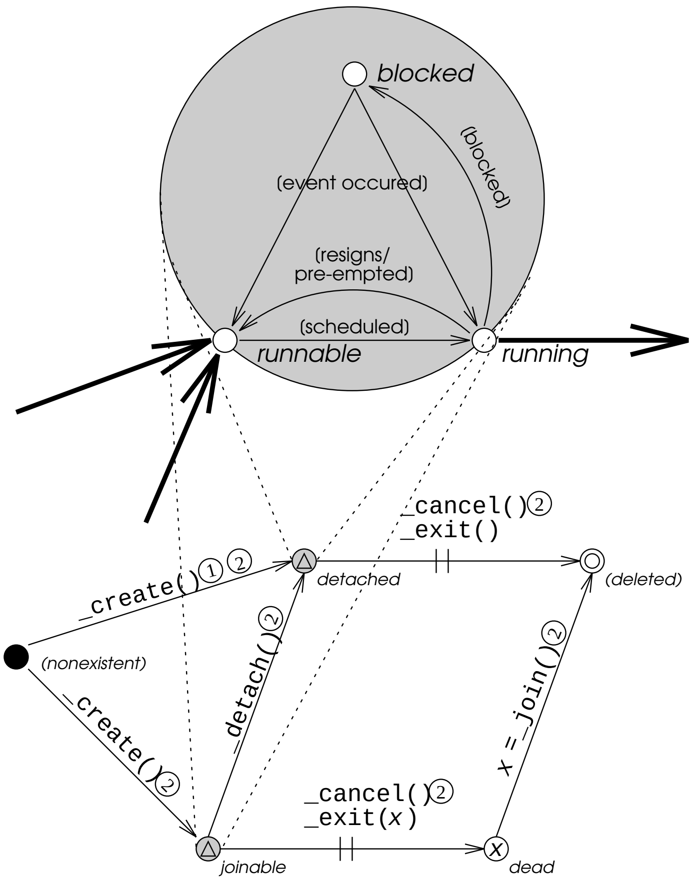
runnable A thread is *runnable* if it has everything it needs to continue processing but is awaiting its turn for the processor. If other runnable (or running) threads have higher priority, and depending on the scheduling policy, it may have to wait indefinitely. It can only leave the *runnable* state by being scheduled and becoming *running* (or possibly by being canceled when requiring no cleanup; this is not shown in figure F.1).

blocked A thread is *blocked* if it is waiting for something. This could be the completion of an i/o operation, notification that some event has occurred, a signal, the completion of an explicit delay, cancelation (in which case it will resume with cleanup processing), or simply permission to proceed. Sometimes when a thread ceases to be blocked it will immediately enter the *running* state (forcing the previously running thread to drop back to *runnable*); otherwise it will become *runnable* and wait its turn with the other runnable threads. Which happens depends upon the scheduling policy in effect, the details of thread priorities, and so on.

Why would a running thread be pre-empted? We have seen that this can happen because another thread of higher priority becomes unblocked and steals the processor from it; similarly, it can happen if it itself does something that unblocks a thread of higher *or equal* priority. A thread can also be descheduled because it is subject to a scheduling policy that periodically (and automatically) pre-empts it just to ensure that other threads get use of the processor, or by explicitly resigning the processor by calling `sched_yield()`$_{172}$.

Note that although we have been discussing the 'life cycle' of a thread, *threads lack ancestry:* there is no requirement that the thread that eventually joins with a given thread be the one that created it, for example, and although a few settings are copied across from the creating thread to the created, no permanent association between them results. In the POSIX model, threads, once spawned, are fully equal partners in the execution of a process.

[1] If the thread did not exit explicitly but was canceled, the joining thread will receive the special value `PTHREAD_CANCELED`$_{204}$.



① Specifying the `CREATE_DETACHED` option

② Performed by another thread

Figure F.1: The states and stages of a thread

processes	**threads**
are expensive to make and destroy	are cheap
consume many system resources	require only a few K of memory
are protected from each other's actions	are vulnerable to each other
virtualize many resources	provide *only* a thread of control
must take action to share data	must protect their data explicitly
can almost ignore each other	must consider each other's actions
interact only clumsily	co-operate with ease

Figure F.2: Relative characteristics of processes and threads: two sides of a single coin

F.d Threads and processes

In a sense, a new thread is like a new process: a new **virtual CPU**[2] is made, complete with a private processor state and a part share of a physical processor with which to carry out computations; it is given a stack to hold variables and return addresses and an initial function to run; and it is set going.

The difference—and it is a very important difference—is that a process is generally started to run a whole new program. The operating system does what it can to ensure processes' independence, allowing them to interact only on explicit request. Threads, on the other hand, are started to run individual *functions,* and like any other function execution a thread shares everything with its creator that is not specifically made local. In many cases the programmer's task is to *prevent* threads interacting in unwanted ways. Of course, this way of doing things makes close co-operation between threads simple—and makes threads *much* cheaper than processes to create.

Another way of looking at the relationships between processes and threads is that a thread is to a CPU as a process is to the whole machine. That is, the kernel takes all the resources afforded by the computer and multiplexes them between all the processes in the system, making each process in effect a virtual computer. The thread system takes just the basic CPU resources and multiplexes these between the threads in a process. Other resources that processes can pretend are their own—memory, device access and so on—remain shared between threads. This is summarized in figure F.2.

The differences in purpose between processes and threads (and two decades of divergent evolution and practical experience) are reflected in differences between the `fork()/exec()` protocol by which new processes are initiated, and thread construction with `pthread_create()`$_{33}$.

F.d.1 The creation process

When creating a process, the effect of the traditional `fork()` call is that an exact clone (in all but the return value of the `fork()` call itself and a few administrative details) of the calling process is created, with, in effect, new private copies of all the memory and data structures belonging to the original, including the stack.[3] If it did not examine the value

[2]We use the term **virtual CPU** rather than the more conventional **virtual processor** because the POSIX committee uses the latter term to refer specifically to those aspects of the thread of control that are virtualized *by the kernel,* and this may or may not co-incide with pthreads themselves. See C.1 277 for a discussion of various implementation strategies for POSIX-compliant threads.

[3]At the time of writing, the POSIX thread draft standard (presently version 8) specifies that only *one* thread is copied by `fork()`, but this whole area of the standard is likely to undergo further

returned by `fork()`, the child process would continue to execute the same program as the parent; since there is almost never a point to this, new processes typically use the return value as a boolean switch, with the child calling a task-specific subroutine or more often invoking `exec()` to overlay itself with an entirely new program.[4] While this sequence serves to protect the parent and the child from each other and does possess a certain elegance, it turns out to be, in most applications, a separation of concerns along the wrong axis: a new thread of control is typically needed to execute new code relating to the same computation, while `fork()` is optimized for executing the same code within a new walled-off computation.

In any event, because of the administrative overhead and the amount of data that may have to be copied (though modern implementations reduce this overhead with **copy on write** technology[5]), process creation is one of the most expensive operations provided by the Unix kernel.[6]

For all of these reasons, new threads are constructed *within* a process and share the vast majority of its resources with any other threads inhabiting it, retaining private copies of only a few data structures that really are logically specific to a single thread of control: the virtual CPU itself, the stack, termination handlers for doing final cleanup, and the internal structures behind thread control, signal masking, and error reporting. Everything else is—and remains—shared.

Thread creation is thus, almost invariably, cheaper than starting a new process.

F.d.2 Initial arguments, final results

Another small but important difference between threads and processes is the way in which they receive their input and yield their output: while Unix processes typically accept control input in the form of an array ('`argv`') of strings, interact with the external world via file handles, and finally return a numeric status code, pthreads receive a single generic pointer

---

revision before it is finalized.

[4]If the program in question is multithreaded the POSIX standard now *requires* a `fork()` to be followed by an `exec()`.

[5]**Copy on Write:** The idea is to take a page that is supposed to be copied, and simply share it between the parent and child processes, informing the virtual memory support hardware that it is a *read-only* page. The next time one of the sharing processes attempts to write to the page, the ensuing error is trapped, a 'real' copy of the page is made, write permission is restored to the faulting process, and it is resumed at the point of the error as if nothing had happened. The beauty of this scheme is threefold: First, it is quite likely that any given page never *will* be written to, because it often represents static or mostly-static data. Second, since a `fork()` operation is almost always followed closely by an `exec()`, the page mappings usually change so as to remove the sharing before very many pages have had to be copied at all. Third, it may well happen that even if a page is modified while it is still shared, it has dropped out of the working set of the non-faulting processes, in which case the process of 'copying' the page involves only checking that it has been written back to disk and adjusting its swap address.

[6]`vfork()`: The `fork()/exec()` sequence is so inefficient, indeed, that it has spawned one of the most amazingly misguided hacks ever to find its way into a system call. Some versions of Unix provide a call, `vfork()`, whose conceptual description is, very sensibly, that it is a variant of `fork()` advising the system that an `exec()` is shortly to follow. The implementation, however, is that the the parent is suspended and the child 'process' starts execution *completely within the parent's context;* construction of the new context doesn't even begin until a subsequent `exec()` call, at which point the parent is resumed. In the interim, the child process has not even been given a new stack! In effect, `vfork()` performed a `setjmp()`, and set a flag modifying the semantics of `exec()` to do a process spawn and a `longjmp()` back to the parent. If the child is so rash as to perform a `return` in the meantime, for example, when it finally calls `exec()` the parent process will be restarted somewhere in the twilight zone....

as their control input, communicate via shared memory, and return another pointer upon completion of their task.

Historically, Unix processes were designed primarily to run programs interactively or under the control of simple shell scripts, and as has already been noted they are relatively expensive objects to set up and tear down. Receiving a human-readable command line[7] and returning a status code that is easily used to control batch processing is thus entirely natural.

Conversely, pthreads are intended to process relatively small pieces of a single logical program, and need to avoid the overhead of parsing textual arguments; they must similarly provide a relatively rich return mechanism. Ideally, arguments and results would obey the same rules as normal function calls, but C doesn't support this—it doesn't have polymorphic typing and every function must have a single, specific type.[8] Fortunately, threads share a common address space; a pointer will mean the same thing to any of the threads of a process. Thus `void *` mechanism provided by Standard C for unchecked pointers, despite its reduced safety and convenience, provides a suitable compromise. A `void *` can, after all, point to (almost) anything, including an arbitrary `struct`; it also has the virtue that in machine terms it is very simple and fast.

In sum, compared to `argv` and exit codes, the mechanism used by pthreads is more regular, faster, more general and better integrated with the language.

F.d.3 Resource Release

Like processes, POSIX threads enter a zombie state on termination, and their final status must usually be collected before their resources can be released. (The exception to this is for so called *detached* threads, which release their resources immediately and implicitly, at the cost of not being able to deliver a result at all.) While with processes this is done with `wait()` (or more generally `waitpid()`), which allows you to await the termination of *any* child process, pthreads provides only `pthread_join()`, which always waits for a *specific* thread.

F.e more stuff

Someplace we need to talk about namespace control. In partciular, we're not supposed to use _t names!

Do we want to talk about `sigwaitinfo()`$_{196}$ and `sigtimedwait()`$_{196}$?

---

[7] Well, sort of almost.

[8] **Polymorphism** in the sense that I here intend is the ability of a function to manipulate data of *varying* type *without* abandoning type safety (this contrasts with overloading, whereby multiple functions with the same name are resolved according to the types of their arguments). In this case, we might be allowed to write a thread creation function with a type like

$\forall_{\texttt{t1, t2}}$ `xthread_t[t1] xthread_create(t1 task(... t2 args),... t2 args)`

which builds a thread with specific—but abstract—guarantees about its argument and result types. It happens (but is not necessarily obvious) that such polymorphism can be implemented without 'tagging' data or losing much in the way of performance: essentially, you first introduce *type-valued* parameters (these could be implicit or explicit, there are advantages each way), and letting one parameter take another as its type. Together with inlining, normal constant folding will almost always eliminate the extra information anyway.

What is the *content* of a type value? Just a code segment that can copy elements of the type on and off the stack—indeed, for C, the only real attribute a type has at runtime is its size—just saving the value `sizeof` should return might suffice.

Needless to say, no such facility exists in C (though it is present in more modern languages like ML or Haskell), nor is likely to be introduced in the foreseeable future!—but this is a nice example of a place where an abstract concept would make a practical problem much easier to solve.

Appendix G

Tables

Types, functions and magic constants

pthread_t$_{32}$	
pthread_create()$_{33}$	
pthread_exit()$_{37}$	
pthread_join()$_{40}$	PTHREAD_CANCELED$_{204}$
pthread_equal()$_{43}$	
pthread_self()$_{42}$	
pthread_cancel()$_{204}$	
pthread_setcancelstate()$_{208}$	PTHREAD_CANCEL_DISABLE$_{208}$ PTHREAD_CANCEL_ENABLE$_{208}$
pthread_setcanceltype()$_{209}$	PTHREAD_CANCEL_DEFERRED$_{210}$ PTHREAD_CANCEL_ASYNCHRONOUS$_{210}$
pthread_testcancel()$_{205}$	
pthread_cleanup_push()$_{122}$ pthread_cleanup_pop()$_{122}$	
pthread_getschedparam()$_{170}$ pthread_setschedparam()$_{170}$	
pthread_getspecific()$_{119}$ pthread_setspecific()$_{118}$	
pthread_kill()$_{190}$	
pthread_sigmask()$_{193}$	
pthread_attr_t$_{138}$	
pthread_create()$_{33}$	
pthread_attr_init()$_{140}$ pthread_attr_destroy()$_{141}$	
pthread_attr_getdetachstate()$_{142}$ pthread_attr_setdetachstate()$_{142}$	PTHREAD_CREATE_DETACHED$_{143}$ PTHREAD_CREATE_JOINABLE$_{143}$
pthread_attr_getinheritsched()$_{161}$ pthread_attr_setinheritsched()$_{161}$	PTHREAD_EXPLICIT_SCHED$_{161}$ PTHREAD_INHERIT_SCHED$_{162}$
pthread_attr_getschedparam()$_{168}$ pthread_attr_setschedparam()$_{168}$	struct schedparam??
pthread_attr_getschedpolicy()$_{165}$ pthread_attr_setschedpolicy()$_{165}$	SCHED_FIFO$_{166}$ SCHED_RR$_{166}$ SCHED_OTHER$_{166}$
pthread_attr_getscope()$_{163}$ pthread_attr_setscope()$_{163}$	PTHREAD_SCOPE_PROCESS$_{164}$ PTHREAD_SCOPE_SYSTEM$_{164}$
pthread_attr_getstackaddr()$_{146}$ pthread_attr_setstackaddr()$_{146}$	
pthread_attr_getstacksize()$_{145}$ pthread_attr_setstacksize()$_{145}$	
pthread_mutex_t$_{68}$	PTHREAD_MUTEX_INITIALIZER$_{70}$
pthread_mutex_init()$_{70}$ pthread_mutex_destroy()$_{72}$	
pthread_mutex_lock()$_{72}$ pthread_mutex_trylock()$_{72}$ pthread_mutex_unlock()$_{75}$	
pthread_mutex_getprioceiling()$_{178}$ pthread_mutex_setprioceiling()$_{178}$	
pthread_cond_wait()$_{83}$ pthread_cond_timedwait()$_{83}$	
pthread_mutexattr_t$_{138}$	
pthread_mutex_init()$_{70}$	
pthread_mutexattr_init()$_{140}$ pthread_mutexattr_destroy()$_{141}$	
pthread_mutexattr_getprioceiling()$_{176}$ pthread_mutexattr_setprioceiling()$_{176}$	
pthread_mutexattr_getprotocol()$_{174}$ pthread_mutexattr_setprotocol()$_{174}$	PTHREAD_PRIO_NONE$_{175}$ PTHREAD_PRIO_INHERIT$_{175}$ PTHREAD_PRIO_PROTECT$_{175}$
pthread_mutexattr_getpshared()$_{148}$ pthread_mutexattr_setpshared()$_{148}$	PTHREAD_PROCESS_PRIVATE$_{149}$ PTHREAD_PROCESS_SHARED$_{149}$
pthread_cond_t$_{80}$	PTHREAD_COND_INITIALIZER$_{81}$
pthread_cond_init()$_{81}$ pthread_cond_destroy()$_{82}$	
pthread_cond_signal()$_{86}$ pthread_cond_broadcast()$_{86}$ pthread_cond_wait()$_{83}$ pthread_cond_timedwait()$_{83}$	
pthread_condattr_t$_{138}$	
pthread_cond_init()$_{81}$	
pthread_condattr_init()$_{140}$ pthread_condattr_destroy()$_{141}$	
pthread_condattr_getpshared()$_{148}$ pthread_condattr_setpshared()$_{148}$	PTHREAD_PROCESS_PRIVATE$_{149}$ PTHREAD_PROCESS_SHARED$_{149}$
pthread_key_t$_{114}$	
pthread_key_create()$_{115}$ pthread_key_delete()$_{117}$	
pthread_getspecific()$_{119}$ pthread_setspecific()$_{118}$	
pthread_once_t$_{90}$	PTHREAD_ONCE_INIT$_{89}$
pthread_once()$_{89}$	

`pthread_atfork()`241

Synchronizing functions 2.3.8

`fork()`??
`pthread_create()`33
`pthread_join()`40
`pthread_mutex_lock()`72
`pthread_mutex_trylock()`72
`pthread_mutex_unlock()`75
`pthread_cond_wait()`83
`pthread_cond_timedwait()`83
`pthread_cond_signal()`86
`pthread_cond_broadcast()`86
`sem_post()`??
`sem_trywait()`??
`sem_wait()`??
`wait()`??
`waitpid()`??

Non-re-entrant functions 2.3.8

`getc_unlocked()`226 `getchar_unlocked()`226
`putc_unlocked()`226 `putchar_unlocked()`226
`asctime()` `ctime()` `gmtime()` `localtime()`
`getgrgid()` `getgrnam()` `getpwnam()` `getpwuid()`
`getlogin()`
`rand()`
`readdir()`
`strok()` *[[(*`strtok?()`*)]]*
`ttyname()`
`ctermid(NULL)` `tmpnam(NULL)`

Feature lists (showing dependencies) 2.9.3

`_POSIX_REENTRANT_FUNCTIONS`??
`_POSIX_THREADS`??
`_POSIX_THREAD_ATTR_STACKADDR`146*
`_POSIX_THREAD_ATTR_STACKSIZE`145*
`_POSIX_THREAD_ATFORK`241*
`_POSIX_THREAD_PRIORITY_SCHEDULING`158
`_POSIX_THREAD_PRIO_INHERIT`175
`_POSIX_THREAD_PRIO_PROTECT`175
`_POSIX_THREAD_PROCESS_SHARED`148
* Dependency missing from draft 8 text.

Sysconf keys 4.8.1

parameter	*Buffer size for:*
_SC_GETGR_R_SIZE_MAX$_{229}$	getgrid_r()?? getgrname_r()??
_SC_GETLOGIN_R_SIZE_MAX$_{229}$	getlogin_r()??
_SC_GETPW_R_SIZE_MAX$_{229}$	getpwuid_r()?? getpwnam_r()??
_SC_TTYNAME_R??	ttyname_r()??

parameter	*Corresponding limit constant*
_SC_THREAD_DESTRUCTOR_ITERATIONS??	PTHREAD_DESTRUCTOR_ITERATIONS$_{121}$
_SC_THREAD_KEYS_MAX??	PTHREAD_KEYS_MAX$_{115}$
_SC_THREAD_STACK_MIN??	PTHREAD_STACK_MIN$_{145}$
_SC_THREAD_THREADS_MAX??	PTHREAD_THREADS_MAX$_{33}$

parameter	*Corresponding configuration constant*
_SC_REENTRANT_FUNCTIONS??	_POSIX_REENTRANT_FUNCTIONS??
_SC_THREADS??	_POSIX_THREADS??
_SC_THREAD_ATTR_STACKADDR??	_POSIX_THREAD_ATTR_STACKADDR$_{146}$
_SC_THREAD_ATTR_STACKSIZE??	_POSIX_THREAD_ATTR_STACKSIZE$_{145}$
_SC_THREAD_ATFORK??	_POSIX_THREAD_ATFORK$_{241}$
_SC_THREAD_PRIORITY_SCHEDULING??	_POSIX_THREAD_PRIORITY_SCHEDULING$_{158}$
_SC_THREAD_PRIO_INHERIT??	_POSIX_THREAD_PRIO_INHERIT$_{175}$
_SC_THREAD_PRIO_PROTECT??	_POSIX_THREAD_PRIO_PROTECT$_{175}$
_SC_THREAD_PROCESS_SHARED??	_POSIX_THREAD_PROCESS_SHARED$_{148}$