xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix
Version 6 (v6). xv6 loosely follows the structure and style of v6,
but is implemented for a modern x86-based multiprocessor using ANSI C.

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14,
2000)). See also http://pdos.csail.mit.edu/6.828/2007/v6.html, which
provides pointers to on-line resources for v6.

xv6 borrows code from the following sources:
    JOS (asm.h, elf.h, mmu.h, bootasm.S, ide.c, console.c, and others)
    Plan 9 (bootother.S, mp.h, mp.c, lapic.c)
    FreeBSD (ioapic.c)
    NetBSD (console.c)

The following people made contributions:
    Russ Cox (context switching, locking)
    Cliff Frey (MP)
    Xiao Yu (MP)

ERROR REPORTS

If you spot errors or have suggestions for improvement, please send
email to Frans Kaashoek and Robert Morris (kaashoek,rtm@csail.mit.edu).

BUILDING AND RUNNING XV6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run "make".
On non-x86 or non-ELF machines (like OS X, even on x86), you will
need to install a cross-compiler gcc suite capable of producing x86 ELF
binaries. See http://pdos.csail.mit.edu/6.828/2007/tools.html.
Then run "make TOOLPREFIX=i386-jos-elf-".

To run xv6, you can use Bochs or QEMU, both PC simulators.
Bochs makes debugging easier, but QEMU is much faster.
To run in Bochs, run "make bochs" and then type "c" at the bochs prompt.
To run in QEMU, run "make qemu". Both log the xv6 screen output to
standard output.

To create a typeset version of the code, run "make xv6.pdf".
This requires the "mpage" text formatting utility.
See http://www.mesa.nl/pub/mpage/.

The numbers to the left of the file names in the table are sheet numbers.
The source code has been printed in a double column format with fifty
lines per column, giving one hundred lines per sheet (or page).
Thus there is a convenient relationship between line numbers and sheet numbers.

The source listing is preceded by a cross-reference that lists every defined
constant, struct, global variable, and function in xv6. Each entry gives,
on the same line as the name, the line number (or, in a few cases, numbers)
where the name is defined. Successive lines in an entry list the line
numbers where the name is used. For example, this entry:

```
    swtch 2256
        0311 1928 1962 2255
        2256
```

indicates that swtch is defined on line 2256 and is mentioned on five lines
on sheets 03, 19, and 22.

```
0100 typedef unsigned int   uint;
0101 typedef unsigned short ushort;
0102 typedef unsigned char  uchar;
0103
0104
0105
0106
0107
0108
0109
0110
0111
0112
0113
0114
0115
0116
0117
0118
0119
0120
0121
0122
0123
0124
0125
0126
0127
0128
0129
0130
0131
0132
0133
0134
0135
0136
0137
0138
0139
0140
0141
0142
0143
0144
0145
0146
0147
0148
0149
```



```
0150 #define NPROC        64  // maximum number of processes
0151 #define PAGE       4096  // granularity of user-space memory allocation
0152 #define KSTACKSIZE PAGE  // size of per-process kernel stack
0153 #define NCPU          8  // maximum number of CPUs
0154 #define NOFILE       16  // open files per process
0155 #define NFILE       100  // open files per system
0156 #define NBUF         10  // size of disk block cache
0157 #define NINODE       50  // maximum number of active i-nodes
0158 #define NDEV         10  // maximum major device number
0159 #define ROOTDEV       1  // device number of file system root disk
0160
0161
0162
0163
0164
0165
0166
0167
0168
0169
0170
0171
0172
0173
0174
0175
0176
0177
0178
0179
0180
0181
0182
0183
0184
0185
0186
0187
0188
0189
0190
0191
0192
0193
0194
0195
0196
0197
0198
0199
```

```
0200 struct buf;
0201 struct context;
0202 struct file;
0203 struct inode;
0204 struct pipe;
0205 struct proc;
0206 struct spinlock;
0207 struct stat;
0208
0209 // bio.c
0210 void            binit(void);
0211 struct buf*     bread(uint, uint);
0212 void            brelse(struct buf*);
0213 void            bwrite(struct buf*);
0214
0215 // console.c
0216 void            console_init(void);
0217 void            cprintf(char*, ...);
0218 void            console_intr(int(*)(void));
0219 void            panic(char*) __attribute__((noreturn));
0220
0221 // exec.c
0222 int             exec(char*, char**);
0223
0224 // file.c
0225 struct file*    filealloc(void);
0226 void            fileclose(struct file*);
0227 struct file*    filedup(struct file*);
0228 void            fileinit(void);
0229 int             fileread(struct file*, char*, int n);
0230 int             filestat(struct file*, struct stat*);
0231 int             filewrite(struct file*, char*, int n);
0232
0233 // fs.c
0234 int             dirlink(struct inode*, char*, uint);
0235 struct inode*   dirlookup(struct inode*, char*, uint*);
0236 struct inode*   ialloc(uint, short);
0237 struct inode*   idup(struct inode*);
0238 void            iinit(void);
0239 void            ilock(struct inode*);
0240 void            iput(struct inode*);
0241 void            iunlock(struct inode*);
0242 void            iunlockput(struct inode*);
0243 void            iupdate(struct inode*);
0244 int             namecmp(const char*, const char*);
0245 struct inode*   namei(char*);
0246 struct inode*   nameiparent(char*, char*);
0247 int             readi(struct inode*, char*, uint, uint);
0248 void            stati(struct inode*, struct stat*);
0249 int             writei(struct inode*, char*, uint, uint);
```



```
0250 // ide.c
0251 void            ide_init(void);
0252 void            ide_intr(void);
0253 void            ide_rw(struct buf *);
0254
0255 // ioapic.c
0256 void            ioapic_enable(int irq, int cpu);
0257 extern uchar    ioapic_id;
0258 void            ioapic_init(void);
0259
0260 // kalloc.c
0261 char*           kalloc(int);
0262 void            kfree(char*, int);
0263 void            kinit(void);
0264
0265 // kbd.c
0266 void            kbd_intr(void);
0267
0268 // lapic.c
0269 int             cpu(void);
0270 extern volatile uint*    lapic;
0271 void            lapic_eoi(void);
0272 void            lapic_init(int);
0273 void            lapic_startap(uchar, uint);
0274
0275 // mp.c
0276 extern int      ismp;
0277 int             mp_bcpu(void);
0278 void            mp_init(void);
0279 void            mp_startthem(void);
0280
0281 // picirq.c
0282 void            pic_enable(int);
0283 void            pic_init(void);
0284
0285 // pipe.c
0286 int             pipealloc(struct file**, struct file**);
0287 void            pipeclose(struct pipe*, int);
0288 int             piperead(struct pipe*, char*, int);
0289 int             pipewrite(struct pipe*, char*, int);
0290
0291 // proc.c
0292 struct proc*    copyproc(struct proc*);
0293 struct proc*    curproc(void);
0294 void            exit(void);
0295 int             growproc(int);
0296 int             kill(int);
0297 void            pinit(void);
0298 void            procdump(void);
0299 void            scheduler(void) __attribute__((noreturn));
```

```
0300 void            setupsegs(struct proc*);
0301 void            sleep(void*, struct spinlock*);
0302 void            userinit(void);
0303 int             wait(void);
0304 void            wakeup(void*);
0305 void            yield(void);
0306
0307 // swtch.S
0308 void            swtch(struct context*, struct context*);
0309
0310 // spinlock.c
0311 void            acquire(struct spinlock*);
0312 void            getcallerpcs(void*, uint*);
0313 int             holding(struct spinlock*);
0314 void            initlock(struct spinlock*, char*);
0315 void            release(struct spinlock*);
0316 void            pushcli();
0317 void            popcli();
0318
0319 // string.c
0320 int             memcmp(const void*, const void*, uint);
0321 void*           memmove(void*, const void*, uint);
0322 void*           memset(void*, int, uint);
0323 char*           safestrcpy(char*, const char*, int);
0324 int             strlen(const char*);
0325 int             strncmp(const char*, const char*, uint);
0326 char*           strncpy(char*, const char*, int);
0327
0328 // syscall.c
0329 int             argint(int, int*);
0330 int             argptr(int, char**, int);
0331 int             argstr(int, char**);
0332 int             fetchint(struct proc*, uint, int*);
0333 int             fetchstr(struct proc*, uint, char**);
0334 void            syscall(void);
0335
0336 // timer.c
0337 void            timer_init(void);
0338
0339 // trap.c
0340 void            idtinit(void);
0341 extern int      ticks;
0342 void            tvinit(void);
0343 extern struct spinlock tickslock;
0344
0345 // number of elements in fixed-size array
0346 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
0347
0348
0349
```



```
0350 // Routines to let C code use special x86 instructions.
0351
0352 static inline uchar
0353 inb(ushort port)
0354 {
0355   uchar data;
0356
0357   asm volatile("in %1,%0" : "=a" (data) : "d" (port));
0358   return data;
0359 }
0360
0361 static inline void
0362 insl(int port, void *addr, int cnt)
0363 {
0364   asm volatile("cld\n\trepne\n\tinsl"     :
0365                    "=D" (addr), "=c" (cnt)    :
0366                    "d" (port), "0" (addr), "1" (cnt)  :
0367                    "memory", "cc");
0368 }
0369
0370 static inline void
0371 outb(ushort port, uchar data)
0372 {
0373   asm volatile("out %0,%1" : : "a" (data), "d" (port));
0374 }
0375
0376 static inline void
0377 outw(ushort port, ushort data)
0378 {
0379   asm volatile("out %0,%1" : : "a" (data), "d" (port));
0380 }
0381
0382 static inline void
0383 outsl(int port, const void *addr, int cnt)
0384 {
0385   asm volatile("cld\n\trepne\n\toutsl"    :
0386                    "=S" (addr), "=c" (cnt)    :
0387                    "d" (port), "0" (addr), "1" (cnt)  :
0388                    "cc");
0389 }
0390
0391 static inline uint
0392 read_ebp(void)
0393 {
0394   uint ebp;
0395
0396   asm volatile("movl %%ebp, %0" : "=a" (ebp));
0397   return ebp;
0398 }
0399
```

```
0400 struct segdesc;
0401
0402 static inline void
0403 lgdt(struct segdesc *p, int size)
0404 {
0405   volatile ushort pd[3];
0406
0407   pd[0] = size-1;
0408   pd[1] = (uint)p;
0409   pd[2] = (uint)p >> 16;
0410
0411   asm volatile("lgdt (%0)" : : "r" (pd));
0412 }
0413
0414 struct gatedesc;
0415
0416 static inline void
0417 lidt(struct gatedesc *p, int size)
0418 {
0419   volatile ushort pd[3];
0420
0421   pd[0] = size-1;
0422   pd[1] = (uint)p;
0423   pd[2] = (uint)p >> 16;
0424
0425   asm volatile("lidt (%0)" : : "r" (pd));
0426 }
0427
0428 static inline void
0429 ltr(ushort sel)
0430 {
0431   asm volatile("ltr %0" : : "r" (sel));
0432 }
0433
0434 static inline uint
0435 read_eflags(void)
0436 {
0437   uint eflags;
0438   asm volatile("pushfl; popl %0" : "=r" (eflags));
0439   return eflags;
0440 }
0441
0442 static inline void
0443 write_eflags(uint eflags)
0444 {
0445   asm volatile("pushl %0; popfl" : : "r" (eflags));
0446 }
0447
0448
0449
```



```
0450 static inline uint
0451 xchg(volatile uint *addr, uint newval)
0452 {
0453   uint result;
0454
0455   // The + in "+m" denotes a read-modify-write operand.
0456   asm volatile("lock; xchgl %0, %1" :
0457                "+m" (*addr), "=a" (result) :
0458                "1" (newval) :
0459                "cc");
0460   return result;
0461 }
0462
0463 static inline void
0464 cli(void)
0465 {
0466   asm volatile("cli");
0467 }
0468
0469 static inline void
0470 sti(void)
0471 {
0472   asm volatile("sti");
0473 }
0474
0475 // Layout of the trap frame built on the stack by the
0476 // hardware and by trapasm.S, and passed to trap().
0477 struct trapframe {
0478   // registers as pushed by pusha
0479   uint edi;
0480   uint esi;
0481   uint ebp;
0482   uint oesp;      // useless & ignored
0483   uint ebx;
0484   uint edx;
0485   uint ecx;
0486   uint eax;
0487
0488   // rest of trap frame
0489   ushort es;
0490   ushort padding1;
0491   ushort ds;
0492   ushort padding2;
0493   uint trapno;
0494
0495   // below here defined by x86 hardware
0496   uint err;
0497   uint eip;
0498   ushort cs;
0499   ushort padding3;
```

```
0500   uint eflags;
0501
0502   // below here only when crossing rings, such as from user to kernel
0503   uint esp;
0504   ushort ss;
0505   ushort padding4;
0506 };
0507
0508
0509
0510
0511
0512
0513
0514
0515
0516
0517
0518
0519
0520
0521
0522
0523
0524
0525
0526
0527
0528
0529
0530
0531
0532
0533
0534
0535
0536
0537
0538
0539
0540
0541
0542
0543
0544
0545
0546
0547
0548
0549
```



```
0550 //
0551 // assembler macros to create x86 segments
0552 //
0553
0554 #define SEG_NULLASM                                             \
0555         .word 0, 0;                                             \
0556         .byte 0, 0, 0, 0
0557
0558 #define SEG_ASM(type,base,lim)                                  \
0559         .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
0560         .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
0561                 (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
0562
0563 #define STA_X     0x8       // Executable segment
0564 #define STA_E     0x4       // Expand down (non-executable segments)
0565 #define STA_C     0x4       // Conforming code segment (executable only)
0566 #define STA_W     0x2       // Writeable (non-executable segments)
0567 #define STA_R     0x2       // Readable (executable segments)
0568 #define STA_A     0x1       // Accessed
0569
0570
0571
0572
0573
0574
0575
0576
0577
0578
0579
0580
0581
0582
0583
0584
0585
0586
0587
0588
0589
0590
0591
0592
0593
0594
0595
0596
0597
0598
0599
```

```
0600 // This file contains definitions for the
0601 // x86 memory management unit (MMU).
0602
0603 // Eflags register
0604 #define FL_CF           0x00000001      // Carry Flag
0605 #define FL_PF           0x00000004      // Parity Flag
0606 #define FL_AF           0x00000010      // Auxiliary carry Flag
0607 #define FL_ZF           0x00000040      // Zero Flag
0608 #define FL_SF           0x00000080      // Sign Flag
0609 #define FL_TF           0x00000100      // Trap Flag
0610 #define FL_IF           0x00000200      // Interrupt Enable
0611 #define FL_DF           0x00000400      // Direction Flag
0612 #define FL_OF           0x00000800      // Overflow Flag
0613 #define FL_IOPL_MASK    0x00003000      // I/O Privilege Level bitmask
0614 #define FL_IOPL_0       0x00000000      //   IOPL == 0
0615 #define FL_IOPL_1       0x00001000      //   IOPL == 1
0616 #define FL_IOPL_2       0x00002000      //   IOPL == 2
0617 #define FL_IOPL_3       0x00003000      //   IOPL == 3
0618 #define FL_NT           0x00004000      // Nested Task
0619 #define FL_RF           0x00010000      // Resume Flag
0620 #define FL_VM           0x00020000      // Virtual 8086 mode
0621 #define FL_AC           0x00040000      // Alignment Check
0622 #define FL_VIF          0x00080000      // Virtual Interrupt Flag
0623 #define FL_VIP          0x00100000      // Virtual Interrupt Pending
0624 #define FL_ID           0x00200000      // ID flag
0625
0626 // Segment Descriptor
0627 struct segdesc {
0628   uint lim_15_0 : 16;  // Low bits of segment limit
0629   uint base_15_0 : 16; // Low bits of segment base address
0630   uint base_23_16 : 8; // Middle bits of segment base address
0631   uint type : 4;       // Segment type (see STS_ constants)
0632   uint s : 1;          // 0 = system, 1 = application
0633   uint dpl : 2;        // Descriptor Privilege Level
0634   uint p : 1;          // Present
0635   uint lim_19_16 : 4;  // High bits of segment limit
0636   uint avl : 1;        // Unused (available for software use)
0637   uint rsv1 : 1;       // Reserved
0638   uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
0639   uint g : 1;          // Granularity: limit scaled by 4K when set
0640   uint base_31_24 : 8; // High bits of segment base address
0641 };
0642
0643
0644
0645
0646
0647
0648
0649
```



```
0650 // Null segment
0651 #define SEG_NULL        (struct segdesc){ 0,0,0,0,0,0,0,0,0,0,0,0,0,0 }
0652
0653 // Normal segment
0654 #define SEG(type, base, lim, dpl) (struct segdesc)                    \
0655 { ((lim) >> 12) & 0xffff, (base) & 0xffff, ((base) >> 16) & 0xff,       \
0656     type, 1, dpl, 1, (uint) (lim) >> 28, 0, 0, 1, 1,                    \
0657     (uint) (base) >> 24 }
0658
0659 #define SEG16(type, base, lim, dpl) (struct segdesc)                    \
0660 { (lim) & 0xffff, (base) & 0xffff, ((base) >> 16) & 0xff,               \
0661     type, 1, dpl, 1, (uint) (lim) >> 16, 0, 0, 1, 0,                    \
0662     (uint) (base) >> 24 }
0663
0664 #define DPL_USER    0x3     // User DPL
0665
0666 // Application segment type bits
0667 #define STA_X       0x8     // Executable segment
0668 #define STA_E       0x4     // Expand down (non-executable segments)
0669 #define STA_C       0x4     // Conforming code segment (executable only)
0670 #define STA_W       0x2     // Writeable (non-executable segments)
0671 #define STA_R       0x2     // Readable (executable segments)
0672 #define STA_A       0x1     // Accessed
0673
0674 // System segment type bits
0675 #define STS_T16A    0x1     // Available 16-bit TSS
0676 #define STS_LDT     0x2     // Local Descriptor Table
0677 #define STS_T16B    0x3     // Busy 16-bit TSS
0678 #define STS_CG16    0x4     // 16-bit Call Gate
0679 #define STS_TG      0x5     // Task Gate / Coum Transmitions
0680 #define STS_IG16    0x6     // 16-bit Interrupt Gate
0681 #define STS_TG16    0x7     // 16-bit Trap Gate
0682 #define STS_T32A    0x9     // Available 32-bit TSS
0683 #define STS_T32B    0xB     // Busy 32-bit TSS
0684 #define STS_CG32    0xC     // 32-bit Call Gate
0685 #define STS_IG32    0xE     // 32-bit Interrupt Gate
0686 #define STS_TG32    0xF     // 32-bit Trap Gate
0687
0688 // Task state segment format
0689 struct taskstate {
0690   uint link;         // Old ts selector
0691   uint esp0;         // Stack pointers and segment selectors
0692   ushort ss0;        //   after an increase in privilege level
0693   ushort padding1;
0694   uint *esp1;
0695   ushort ss1;
0696   ushort padding2;
0697   uint *esp2;
0698   ushort ss2;
0699   ushort padding3;
```

```
0700   void *cr3;         // Page directory base
0701   uint *eip;         // Saved state from last task switch
0702   uint eflags;
0703   uint eax;          // More saved state (registers)
0704   uint ecx;
0705   uint edx;
0706   uint ebx;
0707   uint *esp;
0708   uint *ebp;
0709   uint esi;
0710   uint edi;
0711   ushort es;         // Even more saved state (segment selectors)
0712   ushort padding4;
0713   ushort cs;
0714   ushort padding5;
0715   ushort ss;
0716   ushort padding6;
0717   ushort ds;
0718   ushort padding7;
0719   ushort fs;
0720   ushort padding8;
0721   ushort gs;
0722   ushort padding9;
0723   ushort ldt;
0724   ushort padding10;
0725   ushort t;          // Trap on task switch
0726   ushort iomb;       // I/O map base address
0727 };
0728
0729 // Gate descriptors for interrupts and traps
0730 struct gatedesc {
0731   uint off_15_0 : 16;   // low 16 bits of offset in segment
0732   uint cs : 16;         // code segment selector
0733   uint args : 5;        // # args, 0 for interrupt/trap gates
0734   uint rsv1 : 3;        // reserved(should be zero I guess)
0735   uint type : 4;        // type(STS_{TG,IG32,TG32})
0736   uint s : 1;           // must be 0 (system)
0737   uint dpl : 2;         // descriptor(meaning new) privilege level
0738   uint p : 1;           // Present
0739   uint off_31_16 : 16;  // high bits of offset in segment
0740 };
0741
0742
0743
0744
0745
0746
0747
0748
0749
```



```
0750 // Set up a normal interrupt/trap gate descriptor.
0751 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
0752 //   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
0753 // - sel: Code segment selector for interrupt/trap handler
0754 // - off: Offset in code segment for interrupt/trap handler
0755 // - dpl: Descriptor Privilege Level -
0756 //        the privilege level required for software to invoke
0757 //        this interrupt/trap gate explicitly using an int instruction.
0758 #define SETGATE(gate, istrap, sel, off, d)                \
0759 {                                                         \
0760   (gate).off_15_0 = (uint) (off) & 0xffff;                \
0761   (gate).cs = (sel);                                      \
0762   (gate).args = 0;                                        \
0763   (gate).rsv1 = 0;                                        \
0764   (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
0765   (gate).s = 0;                                           \
0766   (gate).dpl = (d);                                       \
0767   (gate).p = 1;                                           \
0768   (gate).off_31_16 = (uint) (off) >> 16;                  \
0769 }
0770
0771
0772
0773
0774
0775
0776
0777
0778
0779
0780
0781
0782
0783
0784
0785
0786
0787
0788
0789
0790
0791
0792
0793
0794
0795
0796
0797
0798
0799
```

```
0800 // Format of an ELF executable file
0801
0802 #define ELF_MAGIC 0x464C457FU  // "\x7FELF" in little endian
0803
0804 // File header
0805 struct elfhdr {
0806   uint magic;  // must equal ELF_MAGIC
0807   uchar elf[12];
0808   ushort type;
0809   ushort machine;
0810   uint version;
0811   uint entry;
0812   uint phoff;
0813   uint shoff;
0814   uint flags;
0815   ushort ehsize;
0816   ushort phentsize;
0817   ushort phnum;
0818   ushort shentsize;
0819   ushort shnum;
0820   ushort shstrndx;
0821 };
0822
0823 // Program section header
0824 struct proghdr {
0825   uint type;
0826   uint offset;
0827   uint va;
0828   uint pa;
0829   uint filesz;
0830   uint memsz;
0831   uint flags;
0832   uint align;
0833 };
0834
0835 // Values for Proghdr type
0836 #define ELF_PROG_LOAD           1
0837
0838 // Flag bits for Proghdr flags
0839 #define ELF_PROG_FLAG_EXEC      1
0840 #define ELF_PROG_FLAG_WRITE     2
0841 #define ELF_PROG_FLAG_READ      4
0842
0843
0844
0845
0846
0847
0848
0849
```



```
0850 // Blank page.
0851
0852
0853
0854
0855
0856
0857
0858
0859
0860
0861
0862
0863
0864
0865
0866
0867
0868
0869
0870
0871
0872
0873
0874
0875
0876
0877
0878
0879
0880
0881
0882
0883
0884
0885
0886
0887
0888
0889
0890
0891
0892
0893
0894
0895
0896
0897
0898
0899
```

```
0900 #include "asm.h"
0901
0902 # Start the first CPU: switch to 32-bit protected mode, jump into C.
0903 # The BIOS loads this code from the first sector of the hard disk into
0904 # memory at physical address 0x7c00 and starts executing in real mode
0905 # with %cs=0 %ip=7c00.
0906
0907 .set PROT_MODE_CSEG, 0x8         # kernel code segment selector
0908 .set PROT_MODE_DSEG, 0x10        # kernel data segment selector
0909 .set CR0_PE_ON,      0x1         # protected mode enable flag
0910
0911 .globl start
0912 start:
0913   .code16                     # Assemble for 16-bit mode
0914   cli                         # Disable interrupts
0915   cld                         # String operations increment
0916
0917   # Set up the important data segment registers (DS, ES, SS).
0918   xorw    %ax,%ax             # Segment number zero
0919   movw    %ax,%ds             # -> Data Segment
0920   movw    %ax,%es             # -> Extra Segment
0921   movw    %ax,%ss             # -> Stack Segment
0922
0923   # Enable A20:
0924   #   For backwards compatibility with the earliest PCs, physical
0925   #   address line 20 is tied low, so that addresses higher than
0926   #   1MB wrap around to zero by default.  This code undoes this.
0927 seta20.1:
0928   inb     $0x64,%al               # Wait for not busy
0929   testb   $0x2,%al
0930   jnz     seta20.1
0931
0932   movb    $0xd1,%al               # 0xd1 -> port 0x64
0933   outb    %al,$0x64
0934
0935 seta20.2:
0936   inb     $0x64,%al               # Wait for not busy
0937   testb   $0x2,%al
0938   jnz     seta20.2
0939
0940   movb    $0xdf,%al               # 0xdf -> port 0x60
0941   outb    %al,$0x60
0942
0943   # Switch from real to protected mode, using a bootstrap GDT
0944   # and segment translation that makes virtual addresses
0945   # identical to physical addresses, so that the
0946   # effective memory map does not change during the switch.
0947   lgdt    gdtdesc
0948   movl    %cr0, %eax
0949   orl     $CR0_PE_ON, %eax
```



```
0950   movl    %eax, %cr0
0951
0952   # Jump to next instruction, but in 32-bit code segment.
0953   # Switches processor into 32-bit mode.
0954   ljmp    $PROT_MODE_CSEG, $protcseg
0955
0956   .code32                     # Assemble for 32-bit mode
0957 protcseg:
0958   # Set up the protected-mode data segment registers
0959   movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
0960   movw    %ax, %ds                # -> DS: Data Segment
0961   movw    %ax, %es                # -> ES: Extra Segment
0962   movw    %ax, %fs                # -> FS
0963   movw    %ax, %gs                # -> GS
0964   movw    %ax, %ss                # -> SS: Stack Segment
0965
0966   # Set up the stack pointer and call into C.
0967   movl    $start, %esp
0968   call    bootmain
0969
0970   # If bootmain returns (it shouldn't), loop.
0971 spin:
0972   jmp     spin
0973
0974 # Bootstrap GDT
0975 .p2align 2                                # force 4 byte alignment
0976 gdt:
0977   SEG_NULLASM                             # null seg
0978   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
0979   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
0980
0981 gdtdesc:
0982   .word   0x17                            # sizeof(gdt) - 1
0983   .long   gdt                             # address gdt
0984
0985
0986
0987
0988
0989
0990
0991
0992
0993
0994
0995
0996
0997
0998
0999
```

```
1000 #include "asm.h"
1001
1002 # Each non-boot CPU ("AP") is started up in response to a STARTUP
1003 # IPI from the boot CPU.  Section B.4.2 of the Multi-Processor
1004 # Specification says that the AP will start in real mode with CS:IP
1005 # set to XY00:0000, where XY is an 8-bit value sent with the
1006 # STARTUP. Thus this code must start at a 4096-byte boundary.
1007 #
1008 # Because this code sets DS to zero, it must sit
1009 # at an address in the low 2^16 bytes.
1010 #
1011 # Bootothers (in main.c) sends the STARTUPs, one at a time.
1012 # It puts this code (start) at 0x7000.
1013 # It puts the correct %esp in start-4,
1014 # and the place to jump to in start-8.
1015 #
1016 # This code is identical to bootasm.S except:
1017 #   - it does not need to enable A20
1018 #   - it uses the address at start-4 for the %esp
1019 #   - it jumps to the address at start-8 instead of calling bootmain
1020
1021 .set PROT_MODE_CSEG, 0x8         # kernel code segment selector
1022 .set PROT_MODE_DSEG, 0x10        # kernel data segment selector
1023 .set CR0_PE_ON,      0x1         # protected mode enable flag
1024
1025 .globl start
1026 start:
1027   .code16                     # Assemble for 16-bit mode
1028   cli                         # Disable interrupts
1029   cld                         # String operations increment
1030
1031   # Set up the important data segment registers (DS, ES, SS).
1032   xorw    %ax,%ax             # Segment number zero
1033   movw    %ax,%ds             # -> Data Segment
1034   movw    %ax,%es             # -> Extra Segment
1035   movw    %ax,%ss             # -> Stack Segment
1036
1037   # Switch from real to protected mode, using a bootstrap GDT
1038   # and segment translation that makes virtual addresses
1039   # identical to their physical addresses, so that the
1040   # effective memory map does not change during the switch.
1041   lgdt    gdtdesc
1042   movl    %cr0, %eax
1043   orl     $CR0_PE_ON, %eax
1044   movl    %eax, %cr0
1045
1046   # Jump to next instruction, but in 32-bit code segment.
1047   # Switches processor into 32-bit mode.
1048   ljmp    $PROT_MODE_CSEG, $protcseg
1049
```



```
1050   .code32                     # Assemble for 32-bit mode
1051 protcseg:
1052   # Set up the protected-mode data segment registers
1053   movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
1054   movw    %ax, %ds                # -> DS: Data Segment
1055   movw    %ax, %es                # -> ES: Extra Segment
1056   movw    %ax, %fs                # -> FS
1057   movw    %ax, %gs                # -> GS
1058   movw    %ax, %ss                # -> SS: Stack Segment
1059
1060   movl    start-4, %esp
1061   movl    start-8, %eax
1062   jmp     *%eax
1063
1064 # Bootstrap GDT
1065 .p2align 2                                # force 4 byte alignment
1066 gdt:
1067   SEG_NULLASM                             # null seg
1068   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
1069   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
1070
1071 gdtdesc:
1072   .word   0x17                            # sizeof(gdt) - 1
1073   .long   gdt                             # address gdt
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
```

```
1100 // Boot loader.
1101 //
1102 // Part of the boot sector, along with bootasm.S, which calls bootmain().
1103 // bootasm.S has put the processor into protected 32-bit mode.
1104 // bootmain() loads an ELF kernel image from the disk starting at
1105 // sector 1 and then jumps to the kernel entry routine.
1106
1107 #include "types.h"
1108 #include "elf.h"
1109 #include "x86.h"
1110
1111 #define SECTSIZE  512
1112
1113 void readseg(uint, uint, uint);
1114
1115 void
1116 bootmain(void)
1117 {
1118   struct elfhdr *elf;
1119   struct proghdr *ph, *eph;
1120   void (*entry)(void);
1121
1122   elf = (struct elfhdr*)0x10000;  // scratch space
1123
1124   // Read 1st page off disk
1125   readseg((uint)elf, SECTSIZE*8, 0);
1126
1127   // Is this an ELF executable?
1128   if(elf->magic != ELF_MAGIC)
1129     goto bad;
1130
1131   // Load each program segment (ignores ph flags).
1132   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
1133   eph = ph + elf->phnum;
1134   for(; ph < eph; ph++)
1135     readseg(ph->va & 0xFFFFFF, ph->memsz, ph->offset);
1136
1137   // Call the entry point from the ELF header.
1138   // Does not return!
1139   entry = (void(*)(void))(elf->entry & 0xFFFFFF);
1140   entry();
1141
1142 bad:
1143   outw(0x8A00, 0x8A00);
1144   outw(0x8A00, 0x8E00);
1145   for(;;)
1146     ;
1147 }
1148
1149
```



```
1150 void
1151 waitdisk(void)
1152 {
1153   // Wait for disk ready.
1154   while((inb(0x1F7) & 0xC0) != 0x40)
1155     ;
1156 }
1157
1158 // Read a single sector at offset into dst.
1159 void
1160 readsect(void *dst, uint offset)
1161 {
1162   // Issue command.
1163   waitdisk();
1164   outb(0x1F2, 1);   // count = 1
1165   outb(0x1F3, offset);
1166   outb(0x1F4, offset >> 8);
1167   outb(0x1F5, offset >> 16);
1168   outb(0x1F6, (offset >> 24) | 0xE0);
1169   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
1170
1171   // Read data.
1172   waitdisk();
1173   insl(0x1F0, dst, SECTSIZE/4);
1174 }
1175
1176 // Read 'count' bytes at 'offset' from kernel into virtual address 'va'.
1177 // Might copy more than asked.
1178 void
1179 readseg(uint va, uint count, uint offset)
1180 {
1181   uint eva;
1182
1183   eva = va + count;
1184
1185   // Round down to sector boundary.
1186   va &= ~(SECTSIZE - 1);
1187
1188   // Translate from bytes to sectors; kernel starts at sector 1.
1189   offset = (offset / SECTSIZE) + 1;
1190
1191   // If this is too slow, we could read lots of sectors at a time.
1192   // We'd write more to memory than asked, but it doesn't matter --
1193   // we load in increasing order.
1194   for(; va < eva; va += SECTSIZE, offset++)
1195     readsect((uchar*)va, offset);
1196 }
1197
1198
1199
```

```
1200 #include "types.h"
1201 #include "defs.h"
1202 #include "param.h"
1203 #include "mmu.h"
1204 #include "proc.h"
1205 #include "x86.h"
1206
1207 static void bootothers(void);
1208 static void mpmain(void) __attribute__((noreturn));
1209
1210 // Bootstrap processor starts running C code here.
1211 int
1212 main(void)
1213 {
1214   extern char edata[], end[];
1215
1216   // clear BSS
1217   memset(edata, 0, end - edata);
1218
1219   mp_init(); // collect info about this machine
1220   lapic_init(mp_bcpu());
1221   cprintf("\ncpu%d: starting xv6\n\n", cpu());
1222
1223   pinit();         // process table
1224   binit();         // buffer cache
1225   pic_init();      // interrupt controller
1226   ioapic_init();   // another interrupt controller
1227   kinit();         // physical memory allocator
1228   tvinit();        // trap vectors
1229   fileinit();      // file table
1230   iinit();         // inode cache
1231   console_init();  // I/O devices & their interrupts
1232   ide_init();      // disk
1233   if(!ismp)
1234     timer_init();  // uniprocessor timer
1235   userinit();      // first user process
1236   bootothers();    // start other processors
1237
1238   // Finish setting up this processor in mpmain.
1239   mpmain();
1240 }
1241
1242
1243
1244
1245
1246
1247
1248
1249
```



```
1250 // Bootstrap processor gets here after setting up the hardware.
1251 // Additional processors start here.
1252 static void
1253 mpmain(void)
1254 {
1255   cprintf("cpu%d: mpmain\n", cpu());
1256   idtinit();
1257   if(cpu() != mp_bcpu())
1258     lapic_init(cpu());
1259   setupsegs(0);
1260   xchg(&cpus[cpu()].booted, 1);
1261
1262   cprintf("cpu%d: scheduling\n", cpu());
1263   scheduler();
1264 }
1265
1266 static void
1267 bootothers(void)
1268 {
1269   extern uchar _binary_bootother_start[], _binary_bootother_size[];
1270   uchar *code;
1271   struct cpu *c;
1272   char *stack;
1273
1274   // Write bootstrap code to unused memory at 0x7000.
1275   code = (uchar*)0x7000;
1276   memmove(code, _binary_bootother_start, (uint)_binary_bootother_size);
1277
1278   for(c = cpus; c < cpus+ncpu; c++){
1279     if(c == cpus+cpu())  // We've started already.
1280       continue;
1281
1282     // Fill in %esp, %eip and start code on cpu.
1283     stack = kalloc(KSTACKSIZE);
1284     *(void**)(code-4) = stack + KSTACKSIZE;
1285     *(void**)(code-8) = mpmain;
1286     lapic_startap(c->apicid, (uint)code);
1287
1288     // Wait for cpu to get through bootstrap.
1289     while(c->booted == 0)
1290       ;
1291   }
1292 }
1293
1294
1295
1296
1297
1298
1299
```

```
1300 // Mutual exclusion lock.
1301 struct spinlock {
1302   uint locked;   // Is the lock held?
1303
1304   // For debugging:
1305   char *name;    // Name of lock.
1306   int  cpu;      // The number of the cpu holding the lock.
1307   uint pcs[10];  // The call stack (an array of program counters)
1308                  // that locked the lock.
1309 };
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
```



```
1350 // Mutual exclusion spin locks.
1351
1352 #include "types.h"
1353 #include "defs.h"
1354 #include "param.h"
1355 #include "x86.h"
1356 #include "mmu.h"
1357 #include "proc.h"
1358 #include "spinlock.h"
1359
1360 extern int use_console_lock;
1361
1362 void
1363 initlock(struct spinlock *lock, char *name)
1364 {
1365   lock->name = name;
1366   lock->locked = 0;
1367   lock->cpu = 0xffffffff;
1368 }
1369
1370 // Acquire the lock.
1371 // Loops (spins) until the lock is acquired.
1372 // Holding a lock for a long time may cause
1373 // other CPUs to waste time spinning to acquire it.
1374 void
1375 acquire(struct spinlock *lock)
1376 {
1377   pushcli();
1378   if(holding(lock))
1379     panic("acquire");
1380
1381   // The xchg is atomic.
1382   // It also serializes, so that reads after acquire are not
1383   // reordered before it.
1384   while(xchg(&lock->locked, 1) == 1)
1385     ;
1386
1387   // Record info about lock acquisition for debugging.
1388   // The +10 is only so that we can tell the difference
1389   // between forgetting to initialize lock->cpu
1390   // and holding a lock on cpu 0.
1391   lock->cpu = cpu() + 10;
1392   getcallerpcs(&lock, lock->pcs);
1393 }
1394
1395
1396
1397
1398
1399
```

```
1400 // Release the lock.
1401 void
1402 release(struct spinlock *lock)
1403 {
1404   if(!holding(lock))
1405     panic("release");
1406
1407   lock->pcs[0] = 0;
1408   lock->cpu = 0xffffffff;
1409
1410   // The xchg serializes, so that reads before release are
1411   // not reordered after it.  (This reordering would be allowed
1412   // by the Intel manuals, but does not happen on current
1413   // Intel processors.  The xchg being asm volatile also keeps
1414   // gcc from delaying the above assignments.)
1415   xchg(&lock->locked, 0);
1416
1417   popcli();
1418 }
1419
1420 // Record the current call stack in pcs[] by following the %ebp chain.
1421 void
1422 getcallerpcs(void *v, uint pcs[])
1423 {
1424   uint *ebp;
1425   int i;
1426
1427   ebp = (uint*)v - 2;
1428   for(i = 0; i < 10; i++){
1429     if(ebp == 0 || ebp == (uint*)0xffffffff)
1430       break;
1431     pcs[i] = ebp[1];     // saved %eip
1432     ebp = (uint*)ebp[0]; // saved %ebp
1433   }
1434   for(; i < 10; i++)
1435     pcs[i] = 0;
1436 }
1437
1438 // Check whether this cpu is holding the lock.
1439 int
1440 holding(struct spinlock *lock)
1441 {
1442   return lock->locked && lock->cpu == cpu() + 10;
1443 }
1444
1445
1446
1447
1448
1449
```



```
1450 // Pushcli/popcli are like cli/sti except that they are matched:
1451 // it takes two popcli to undo two pushcli.  Also, if interrupts
1452 // are off, then pushcli, popcli leaves them off.
1453
1454 void
1455 pushcli(void)
1456 {
1457   int eflags;
1458
1459   eflags = read_eflags();
1460   cli();
1461   if(cpus[cpu()].ncli++ == 0)
1462     cpus[cpu()].intena = eflags & FL_IF;
1463 }
1464
1465 void
1466 popcli(void)
1467 {
1468   if(read_eflags()&FL_IF)
1469     panic("popcli - interruptible");
1470   if(--cpus[cpu()].ncli < 0)
1471     panic("popcli");
1472   if(cpus[cpu()].ncli == 0 && cpus[cpu()].intena)
1473     sti();
1474 }
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
```

```
1500 // Segments in proc->gdt
1501 #define SEG_KCODE 1  // kernel code
1502 #define SEG_KDATA 2  // kernel data+stack
1503 #define SEG_UCODE 3
1504 #define SEG_UDATA 4
1505 #define SEG_TSS   5  // this process's task state
1506 #define NSEGS     6
1507
1508 // Saved registers for kernel context switches.
1509 // Don't need to save all the %fs etc. segment registers,
1510 // because they are constant across kernel contexts.
1511 // Save all the regular registers so we don't need to care
1512 // which are caller save, but not the return register %eax.
1513 // (Not saving %eax just simplifies the switching code.)
1514 // The layout of context must match code in swtch.S.
1515 struct context {
1516   int eip;
1517   int esp;
1518   int ebx;
1519   int ecx;
1520   int edx;
1521   int esi;
1522   int edi;
1523   int ebp;
1524 };
1525
1526 enum proc_state { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
1527
1528 // Per-process state
1529 struct proc {
1530   char *mem;                // Start of process memory (kernel address)
1531   uint sz;                  // Size of process memory (bytes)
1532   char *kstack;             // Bottom of kernel stack for this process
1533   enum proc_state state;    // Process state
1534   int pid;                  // Process ID
1535   struct proc *parent;      // Parent process
1536   void *chan;               // If non-zero, sleeping on chan
1537   int killed;               // If non-zero, have been killed
1538   struct file *ofile[NOFILE];  // Open files
1539   struct inode *cwd;        // Current directory
1540   struct context context;   // Switch here to run process
1541   struct trapframe *tf;     // Trap frame for current interrupt
1542   char name[16];            // Process name (debugging)
1543 };
1544
1545
1546
1547
1548
1549
```



```
1550 // Process memory is laid out contiguously, low addresses first:
1551 //   text
1552 //   original data and bss
1553 //   fixed-size stack
1554 //   expandable heap
1555
1556 // Per-CPU state
1557 struct cpu {
1558   uchar apicid;               // Local APIC ID
1559   struct proc *curproc;       // Process currently running.
1560   struct context context;     // Switch here to enter scheduler
1561   struct taskstate ts;        // Used by x86 to find stack for interrupt
1562   struct segdesc gdt[NSEGS];  // x86 global descriptor table
1563   volatile uint booted;        // Has the CPU started?
1564   int ncli;                   // Depth of pushcli nesting.
1565   int intena;                 // Were interrupts enabled before pushcli?
1566 };
1567
1568 extern struct cpu cpus[NCPU];
1569 extern int ncpu;
1570
1571 // "cp" is a short alias for curproc().
1572 // It gets used enough to make this worthwhile.
1573 #define cp curproc()
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
```

```
1600 #include "types.h"
1601 #include "defs.h"
1602 #include "param.h"
1603 #include "mmu.h"
1604 #include "x86.h"
1605 #include "proc.h"
1606 #include "spinlock.h"
1607
1608 struct spinlock proc_table_lock;
1609
1610 struct proc proc[NPROC];
1611 static struct proc *initproc;
1612
1613 int nextpid = 1;
1614 extern void forkret(void);
1615 extern void forkret1(struct trapframe*);
1616
1617 void
1618 pinit(void)
1619 {
1620   initlock(&proc_table_lock, "proc_table");
1621 }
1622
1623 // Look in the process table for an UNUSED proc.
1624 // If found, change state to EMBRYO and return it.
1625 // Otherwise return 0.
1626 static struct proc*
1627 allocproc(void)
1628 {
1629   int i;
1630   struct proc *p;
1631
1632   acquire(&proc_table_lock);
1633   for(i = 0; i < NPROC; i++){
1634     p = &proc[i];
1635     if(p->state == UNUSED){
1636       p->state = EMBRYO;
1637       p->pid = nextpid++;
1638       release(&proc_table_lock);
1639       return p;
1640     }
1641   }
1642   release(&proc_table_lock);
1643   return 0;
1644 }
1645
1646
1647
1648
1649
```



```
1650 // Grow current process's memory by n bytes.
1651 // Return old size on success, -1 on failure.
1652 int
1653 growproc(int n)
1654 {
1655   char *newmem;
1656
1657   newmem = kalloc(cp->sz + n);
1658   if(newmem == 0)
1659     return -1;
1660   memmove(newmem, cp->mem, cp->sz);
1661   memset(newmem + cp->sz, 0, n);
1662   kfree(cp->mem, cp->sz);
1663   cp->mem = newmem;
1664   cp->sz += n;
1665   setupsegs(cp);
1666   return cp->sz - n;
1667 }
1668
1669 // Set up CPU's segment descriptors and task state for a given process.
1670 // If p==0, set up for "idle" state for when scheduler() is running.
1671 void
1672 setupsegs(struct proc *p)
1673 {
1674   struct cpu *c;
1675
1676   pushcli();
1677   c = &cpus[cpu()];
1678   c->ts.ss0 = SEG_KDATA << 3;
1679   if(p)
1680     c->ts.esp0 = (uint)(p->kstack + KSTACKSIZE);
1681   else
1682     c->ts.esp0 = 0xffffffff;
1683
1684   c->gdt[0] = SEG_NULL;
1685   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0x100000 + 64*1024-1, 0);
1686   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1687   c->gdt[SEG_TSS] = SEG16(STS_T32A, (uint)&c->ts, sizeof(c->ts)-1, 0);
1688   c->gdt[SEG_TSS].s = 0;
1689   if(p){
1690     c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, (uint)p->mem, p->sz-1, DPL_USER);
1691     c->gdt[SEG_UDATA] = SEG(STA_W, (uint)p->mem, p->sz-1, DPL_USER);
1692   } else {
1693     c->gdt[SEG_UCODE] = SEG_NULL;
1694     c->gdt[SEG_UDATA] = SEG_NULL;
1695   }
1696
1697
1698
1699
```

```
1700   lgdt(c->gdt, sizeof(c->gdt));
1701   ltr(SEG_TSS << 3);
1702   popcli();
1703 }
1704
1705 // Create a new process copying p as the parent.
1706 // Sets up stack to return as if from system call.
1707 // Caller must set state of returned proc to RUNNABLE.
1708 struct proc*
1709 copyproc(struct proc *p)
1710 {
1711   int i;
1712   struct proc *np;
1713
1714   // Allocate process.
1715   if((np = allocproc()) == 0)
1716     return 0;
1717
1718   // Allocate kernel stack.
1719   if((np->kstack = kalloc(KSTACKSIZE)) == 0){
1720     np->state = UNUSED;
1721     return 0;
1722   }
1723   np->tf = (struct trapframe*)(np->kstack + KSTACKSIZE) - 1;
1724
1725   if(p){  // Copy process state from p.
1726     np->parent = p;
1727     memmove(np->tf, p->tf, sizeof(*np->tf));
1728
1729     np->sz = p->sz;
1730     if((np->mem = kalloc(np->sz)) == 0){
1731       kfree(np->kstack, KSTACKSIZE);
1732       np->kstack = 0;
1733       np->state = UNUSED;
1734       np->parent = 0;
1735       return 0;
1736     }
1737     memmove(np->mem, p->mem, np->sz);
1738
1739     for(i = 0; i < NOFILE; i++)
1740       if(p->ofile[i])
1741         np->ofile[i] = filedup(p->ofile[i]);
1742     np->cwd = idup(p->cwd);
1743   }
1744
1745   // Set up new context to start executing at forkret (see below).
1746   memset(&np->context, 0, sizeof(np->context));
1747   np->context.eip = (uint)forkret;
1748   np->context.esp = (uint)np->tf;
1749
```



```
1750   // Clear %eax so that fork system call returns 0 in child.
1751   np->tf->eax = 0;
1752   return np;
1753 }
1754
1755 // Set up first user process.
1756 void
1757 userinit(void)
1758 {
1759   struct proc *p;
1760   extern uchar _binary_initcode_start[], _binary_initcode_size[];
1761
1762   p = copyproc(0);
1763   p->sz = PAGE;
1764   p->mem = kalloc(p->sz);
1765   p->cwd = namei("/");
1766   memset(p->tf, 0, sizeof(*p->tf));
1767   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
1768   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
1769   p->tf->es = p->tf->ds;
1770   p->tf->ss = p->tf->ds;
1771   p->tf->eflags = FL_IF;
1772   p->tf->esp = p->sz;
1773
1774   // Make return address readable; needed for some gcc.
1775   p->tf->esp -= 4;
1776   *(uint*)(p->mem + p->tf->esp) = 0xefefefef;
1777
1778   // On entry to user space, start executing at beginning of initcode.S.
1779   p->tf->eip = 0;
1780   memmove(p->mem, _binary_initcode_start, (int)_binary_initcode_size);
1781   safestrcpy(p->name, "initcode", sizeof(p->name));
1782   p->state = RUNNABLE;
1783
1784   initproc = p;
1785 }
1786
1787 // Return currently running process.
1788 struct proc*
1789 curproc(void)
1790 {
1791   struct proc *p;
1792
1793   pushcli();
1794   p = cpus[cpu()].curproc;
1795   popcli();
1796   return p;
1797 }
1798
1799
```

```
1800 // Per-CPU process scheduler.
1801 // Each CPU calls scheduler() after setting itself up.
1802 // Scheduler never returns.  It loops, doing:
1803 //  - choose a process to run
1804 //  - swtch to start running that process
1805 //  - eventually that process transfers control
1806 //      via swtch back to the scheduler.
1807 void
1808 scheduler(void)
1809 {
1810   struct proc *p;
1811   struct cpu *c;
1812   int i;
1813
1814   c = &cpus[cpu()];
1815   for(;;){
1816     // Enable interrupts on this processor.
1817     sti();
1818
1819     // Loop over process table looking for process to run.
1820     acquire(&proc_table_lock);
1821     for(i = 0; i < NPROC; i++){
1822       p = &proc[i];
1823       if(p->state != RUNNABLE)
1824         continue;
1825
1826       // Switch to chosen process.  It is the process's job
1827       // to release proc_table_lock and then reacquire it
1828       // before jumping back to us.
1829       c->curproc = p;
1830       setupsegs(p);
1831       p->state = RUNNING;
1832       cons_putc('a');
1833       swtch(&c->context, &p->context);
1834       cons_putc('b');
1835
1836       // Process is done running for now.
1837       // It should have changed its p->state before coming back.
1838       c->curproc = 0;
1839       setupsegs(0);
1840     }
1841     release(&proc_table_lock);
1842
1843   }
1844 }
1845
1846
1847
1848
1849
```



```
1850 // Enter scheduler.  Must already hold proc_table_lock
1851 // and have changed curproc[cpu()]->state.
1852 void
1853 sched(void)
1854 {
1855   if(read_eflags()&FL_IF)
1856     panic("sched interruptible");
1857   if(cp->state == RUNNING)
1858     panic("sched running");
1859   if(!holding(&proc_table_lock))
1860     panic("sched proc_table_lock");
1861   if(cpus[cpu()].ncli != 1)
1862     panic("sched locks");
1863
1864   cons_putc('c');
1865   swtch(&cp->context, &cpus[cpu()].context);
1866   cons_putc('d');
1867 }
1868
1869 // Give up the CPU for one scheduling round.
1870 void
1871 yield(void)
1872 {
1873   acquire(&proc_table_lock);
1874   cp->state = RUNNABLE;
1875   sched();
1876   release(&proc_table_lock);
1877 }
1878
1879 // A fork child's very first scheduling by scheduler()
1880 // will swtch here.  "Return" to user space.
1881 void
1882 forkret(void)
1883 {
1884   // Still holding proc_table_lock from scheduler.
1885   release(&proc_table_lock);
1886
1887   // Jump into assembly, never to return.
1888   forkret1(cp->tf);
1889 }
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
```

```
1900 // Atomically release lock and sleep on chan.
1901 // Reacquires lock when reawakened.
1902 void
1903 sleep(void *chan, struct spinlock *lk)
1904 {
1905   if(cp == 0)
1906     panic("sleep");
1907
1908   if(lk == 0)
1909     panic("sleep without lk");
1910
1911   // Must acquire proc_table_lock in order to
1912   // change p->state and then call sched.
1913   // Once we hold proc_table_lock, we can be
1914   // guaranteed that we won't miss any wakeup
1915   // (wakeup runs with proc_table_lock locked),
1916   // so it's okay to release lk.
1917   if(lk != &proc_table_lock){
1918     acquire(&proc_table_lock);
1919     release(lk);
1920   }
1921
1922   // Go to sleep.
1923   cp->chan = chan;
1924   cp->state = SLEEPING;
1925   sched();
1926
1927   // Tidy up.
1928   cp->chan = 0;
1929
1930   // Reacquire original lock.
1931   if(lk != &proc_table_lock){
1932     release(&proc_table_lock);
1933     acquire(lk);
1934   }
1935 }
1936
1937 // Wake up all processes sleeping on chan.
1938 // Proc_table_lock must be held.
1939 static void
1940 wakeup1(void *chan)
1941 {
1942   struct proc *p;
1943
1944   for(p = proc; p < &proc[NPROC]; p++)
1945     if(p->state == SLEEPING && p->chan == chan)
1946       p->state = RUNNABLE;
1947 }
1948
1949
```



```
1950 // Wake up all processes sleeping on chan.
1951 void
1952 wakeup(void *chan)
1953 {
1954   acquire(&proc_table_lock);
1955   wakeup1(chan);
1956   release(&proc_table_lock);
1957 }
1958
1959 // Kill the process with the given pid.
1960 // Process won't actually exit until it returns
1961 // to user space (see trap in trap.c).
1962 int
1963 kill(int pid)
1964 {
1965   struct proc *p;
1966
1967   acquire(&proc_table_lock);
1968   for(p = proc; p < &proc[NPROC]; p++){
1969     if(p->pid == pid){
1970       p->killed = 1;
1971       // Wake process from sleep if necessary.
1972       if(p->state == SLEEPING)
1973         p->state = RUNNABLE;
1974       release(&proc_table_lock);
1975       return 0;
1976     }
1977   }
1978   release(&proc_table_lock);
1979   return -1;
1980 }
1981
1982 // Exit the current process.  Does not return.
1983 // Exited processes remain in the zombie state
1984 // until their parent calls wait() to find out they exited.
1985 void
1986 exit(void)
1987 {
1988   struct proc *p;
1989   int fd;
1990
1991   if(cp == initproc)
1992     panic("init exiting");
1993
1994   // Close all open files.
1995   for(fd = 0; fd < NOFILE; fd++){
1996     if(cp->ofile[fd]){
1997       fileclose(cp->ofile[fd]);
1998       cp->ofile[fd] = 0;
1999     }
```

```
2000   }
2001
2002   iput(cp->cwd);
2003   cp->cwd = 0;
2004
2005   acquire(&proc_table_lock);
2006
2007   // Parent might be sleeping in wait().
2008   wakeup1(cp->parent);
2009
2010   // Pass abandoned children to init.
2011   for(p = proc; p < &proc[NPROC]; p++){
2012     if(p->parent == cp){
2013       p->parent = initproc;
2014       if(p->state == ZOMBIE)
2015         wakeup1(initproc);
2016     }
2017   }
2018
2019   // Jump into the scheduler, never to return.
2020   cp->killed = 0;
2021   cp->state = ZOMBIE;
2022   sched();
2023   panic("zombie exit");
2024 }
2025
2026 // Wait for a child process to exit and return its pid.
2027 // Return -1 if this process has no children.
2028 int
2029 wait(void)
2030 {
2031   struct proc *p;
2032   int i, havekids, pid;
2033
2034   acquire(&proc_table_lock);
2035   for(;;){
2036     // Scan through table looking for zombie children.
2037     havekids = 0;
2038     for(i = 0; i < NPROC; i++){
2039       p = &proc[i];
2040       if(p->state == UNUSED)
2041         continue;
2042       if(p->parent == cp){
2043         if(p->state == ZOMBIE){
2044           // Found one.
2045           kfree(p->mem, p->sz);
2046           kfree(p->kstack, KSTACKSIZE);
2047           pid = p->pid;
2048           p->state = UNUSED;
2049           p->pid = 0;
```



```
2050           p->parent = 0;
2051           p->name[0] = 0;
2052           release(&proc_table_lock);
2053           return pid;
2054         }
2055         havekids = 1;
2056       }
2057     }
2058
2059     // No point waiting if we don't have any children.
2060     if(!havekids || cp->killed){
2061       release(&proc_table_lock);
2062       return -1;
2063     }
2064
2065     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2066     sleep(cp, &proc_table_lock);
2067   }
2068 }
2069
2070 // Print a process listing to console.  For debugging.
2071 // Runs when user types ^P on console.
2072 // No lock to avoid wedging a stuck machine further.
2073 void
2074 procdump(void)
2075 {
2076   static char *states[] = {
2077   [UNUSED]    "unused",
2078   [EMBRYO]    "embryo",
2079   [SLEEPING]  "sleep ",
2080   [RUNNABLE]  "runble",
2081   [RUNNING]   "run   ",
2082   [ZOMBIE]    "zombie"
2083   };
2084   int i, j;
2085   struct proc *p;
2086   char *state;
2087   uint pc[10];
2088
2089   for(i = 0; i < NPROC; i++){
2090     p = &proc[i];
2091     if(p->state == UNUSED)
2092       continue;
2093     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
2094       state = states[p->state];
2095     else
2096       state = "???";
2097     cprintf("%d %s %s", p->pid, state, p->name);
2098     if(p->state == SLEEPING){
2099       getcallerpcs((uint*)p->context.ebp+2, pc);
```

```
2100       for(j=0; j<10 && pc[j] != 0; j++)
2101         cprintf(" %p", pc[j]);
2102     }
2103     cprintf("\n");
2104   }
2105 }
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
```



```
2150 #   void swtch(struct context *old, struct context *new);
2151 #
2152 # Save current register context in old
2153 # and then load register context from new.
2154
2155 .globl swtch
2156 swtch:
2157   # Save old registers
2158   movl 4(%esp), %eax
2159
2160   popl 0(%eax)  # %eip
2161   movl %esp, 4(%eax)
2162   movl %ebx, 8(%eax)
2163   movl %ecx, 12(%eax)
2164   movl %edx, 16(%eax)
2165   movl %esi, 20(%eax)
2166   movl %edi, 24(%eax)
2167   movl %ebp, 28(%eax)
2168
2169   # Load new registers
2170   movl 4(%esp), %eax  # not 8(%esp) - popped return address above
2171
2172   movl 28(%eax), %ebp
2173   movl 24(%eax), %edi
2174   movl 20(%eax), %esi
2175   movl 16(%eax), %edx
2176   movl 12(%eax), %ecx
2177   movl 8(%eax), %ebx
2178   movl 4(%eax), %esp
2179   pushl 0(%eax)  # %eip
2180
2181   ret
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
```

```
2200 // Physical memory allocator, intended to allocate
2201 // memory for user processes. Allocates in 4096-byte "pages".
2202 // Free list is kept sorted and combines adjacent pages into
2203 // long runs, to make it easier to allocate big segments.
2204 // One reason the page size is 4k is that the x86 segment size
2205 // granularity is 4k.
2206
2207 #include "types.h"
2208 #include "defs.h"
2209 #include "param.h"
2210 #include "spinlock.h"
2211
2212 struct spinlock kalloc_lock;
2213
2214 struct run {
2215   struct run *next;
2216   int len; // bytes
2217 };
2218 struct run *freelist;
2219
2220 // Initialize free list of physical pages.
2221 // This code cheats by just considering one megabyte of
2222 // pages after _end.  Real systems would determine the
2223 // amount of memory available in the system and use it all.
2224 void
2225 kinit(void)
2226 {
2227   extern int end;
2228   uint mem;
2229   char *start;
2230
2231   initlock(&kalloc_lock, "kalloc");
2232   start = (char*) &end;
2233   start = (char*) (((uint)start + PAGE) & ~(PAGE-1));
2234   mem = 256; // assume computer has 256 pages of RAM
2235   cprintf("mem = %d\n", mem * PAGE);
2236   kfree(start, mem * PAGE);
2237 }
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
```



```
2250 // Free the len bytes of memory pointed at by v,
2251 // which normally should have been returned by a
2252 // call to kalloc(len).  (The exception is when
2253 // initializing the allocator; see kinit above.)
2254 void
2255 kfree(char *v, int len)
2256 {
2257   struct run *r, *rend, **rp, *p, *pend;
2258
2259   if(len <= 0 || len % PAGE)
2260     panic("kfree");
2261
2262   // Fill with junk to catch dangling refs.
2263   memset(v, 1, len);
2264
2265   acquire(&kalloc_lock);
2266   p = (struct run*)v;
2267   pend = (struct run*)(v + len);
2268   for(rp=&freelist; (r=*rp) != 0 && r <= pend; rp=&r->next){
2269     rend = (struct run*)((char*)r + r->len);
2270     if(r <= p && p < rend)
2271       panic("freeing free page");
2272     if(pend == r){  // p next to r: replace r with p
2273       p->len = len + r->len;
2274       p->next = r->next;
2275       *rp = p;
2276       goto out;
2277     }
2278     if(rend == p){  // r next to p: replace p with r
2279       r->len += len;
2280       if(r->next && r->next == pend){  // r now next to r->next?
2281         r->len += r->next->len;
2282         r->next = r->next->next;
2283       }
2284       goto out;
2285     }
2286   }
2287   // Insert p before r in list.
2288   p->len = len;
2289   p->next = r;
2290   *rp = p;
2291
2292  out:
2293   release(&kalloc_lock);
2294 }
2295
2296
2297
2298
2299
```

```
2300 // Allocate n bytes of physical memory.
2301 // Returns a kernel-segment pointer.
2302 // Returns 0 if the memory cannot be allocated.
2303 char*
2304 kalloc(int n)
2305 {
2306   char *p;
2307   struct run *r, **rp;
2308
2309   if(n % PAGE || n <= 0)
2310     panic("kalloc");
2311
2312   acquire(&kalloc_lock);
2313   for(rp=&freelist; (r=*rp) != 0; rp=&r->next){
2314     if(r->len == n){
2315       *rp = r->next;
2316       release(&kalloc_lock);
2317       return (char*)r;
2318     }
2319     if(r->len > n){
2320       r->len -= n;
2321       p = (char*)r + r->len;
2322       release(&kalloc_lock);
2323       return p;
2324     }
2325   }
2326   release(&kalloc_lock);
2327
2328   cprintf("kalloc: out of memory\n");
2329   return 0;
2330 }
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
```



```
2350 // x86 trap and interrupt constants.
2351
2352 // Processor-defined:
2353 #define T_DIVIDE         0      // divide error
2354 #define T_DEBUG          1      // debug exception
2355 #define T_NMI            2      // non-maskable interrupt
2356 #define T_BRKPT          3      // breakpoint
2357 #define T_OFLOW          4      // overflow
2358 #define T_BOUND          5      // bounds check
2359 #define T_ILLOP          6      // illegal opcode
2360 #define T_DEVICE         7      // device not available
2361 #define T_DBLFLT         8      // double fault
2362 // #define T_COPROC      9      // reserved (not used since 486)
2363 #define T_TSS           10      // invalid task switch segment
2364 #define T_SEGNP         11      // segment not present
2365 #define T_STACK         12      // stack exception
2366 #define T_GPFLT         13      // general protection fault
2367 #define T_PGFLT         14      // page fault
2368 // #define T_RES        15      // reserved
2369 #define T_FPERR         16      // floating point error
2370 #define T_ALIGN         17      // aligment check
2371 #define T_MCHK          18      // machine check
2372 #define T_SIMDERR       19      // SIMD floating point error
2373
2374 // These are arbitrarily chosen, but with care not to overlap
2375 // processor defined exceptions or interrupt vectors.
2376 #define T_SYSCALL       48      // system call
2377 #define T_DEFAULT      500      // catchall
2378
2379 #define IRQ_OFFSET      32      // IRQ 0 corresponds to int IRQ_OFFSET
2380
2381 #define IRQ_TIMER        0
2382 #define IRQ_KBD          1
2383 #define IRQ_IDE         14
2384 #define IRQ_ERROR       19
2385 #define IRQ_SPURIOUS    31
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
```

```
2400 #!/usr/bin/perl -w
2401
2402 # Generate vectors.S, the trap/interrupt entry points.
2403 # There has to be one entry point per interrupt number
2404 # since otherwise there's no way for trap() to discover
2405 # the interrupt number.
2406
2407 print "# generated by vectors.pl - do not edit\n";
2408 print "# handlers\n";
2409 print ".text\n";
2410 print ".globl alltraps\n";
2411 for(my $i = 0; $i < 256; $i++){
2412     print ".globl vector$i\n";
2413     print "vector$i:\n";
2414     if(($i < 8 || $i > 14) && $i != 17){
2415         print "  pushl \$0\n";
2416     }
2417     print "  pushl \$$i\n";
2418     print "  jmp alltraps\n";
2419 }
2420
2421 print "\n# vector table\n";
2422 print ".data\n";
2423 print ".globl vectors\n";
2424 print "vectors:\n";
2425 for(my $i = 0; $i < 256; $i++){
2426     print "  .long vector$i\n";
2427 }
2428
2429 # sample output:
2430 #   # handlers
2431 #   .text
2432 #   .globl alltraps
2433 #   .globl vector0
2434 #   vector0:
2435 #     pushl $0
2436 #     pushl $0
2437 #     jmp alltraps
2438 #   ...
2439 #
2440 #   # vector table
2441 #   .data
2442 #   .globl vectors
2443 #   vectors:
2444 #     .long vector0
2445 #     .long vector1
2446 #     .long vector2
2447 #   ...
2448
2449
```



```
2450 .text
2451
2452 .set SEG_KDATA_SEL, 0x10   # selector for SEG_KDATA
2453
2454   # vectors.S sends all traps here.
2455 .globl alltraps
2456 alltraps:
2457   # Build trap frame.
2458   pushl %ds
2459   pushl %es
2460   pushal
2461
2462   # Set up data segments.
2463   movl $SEG_KDATA_SEL, %eax
2464   movw %ax,%ds
2465   movw %ax,%es
2466
2467   # Call trap(tf), where tf=%esp
2468   pushl %esp
2469   call trap
2470   addl $4, %esp
2471
2472   # Return falls through to trapret...
2473 .globl trapret
2474 trapret:
2475   popal
2476   popl %es
2477   popl %ds
2478   addl $0x8, %esp  # trapno and errcode
2479   iret
2480
2481   # A forked process switches to user mode by calling
2482   # forkret1(tf), where tf is the trap frame to use.
2483 .globl forkret1
2484 forkret1:
2485   movl 4(%esp), %esp
2486   jmp trapret
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
```

```
2500 #include "types.h"
2501 #include "defs.h"
2502 #include "param.h"
2503 #include "mmu.h"
2504 #include "proc.h"
2505 #include "x86.h"
2506 #include "traps.h"
2507 #include "spinlock.h"
2508
2509 // Interrupt descriptor table (shared by all CPUs).
2510 struct gatedesc idt[256];
2511 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
2512 struct spinlock tickslock;
2513 int ticks;
2514
2515 void
2516 tvinit(void)
2517 {
2518   int i;
2519
2520   for(i = 0; i < 256; i++)
2521     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
2522   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
2523
2524   initlock(&tickslock, "time");
2525 }
2526
2527 void
2528 idtinit(void)
2529 {
2530   lidt(idt, sizeof(idt));
2531 }
2532
2533 void
2534 trap(struct trapframe *tf)
2535 {
2536   if(tf->trapno == T_SYSCALL){
2537     if(cp->killed)
2538       exit();
2539     cp->tf = tf;
2540     syscall();
2541     if(cp->killed)
2542       exit();
2543     return;
2544   }
2545
2546   switch(tf->trapno){
2547   case IRQ_OFFSET + IRQ_TIMER:
2548     if(cpu() == 0){
2549       acquire(&tickslock);
```



```
2550       ticks++;
2551       wakeup(&ticks);
2552       release(&tickslock);
2553     }
2554     lapic_eoi();
2555     break;
2556   case IRQ_OFFSET + IRQ_IDE:
2557     ide_intr();
2558     lapic_eoi();
2559     break;
2560   case IRQ_OFFSET + IRQ_KBD:
2561     kbd_intr();
2562     lapic_eoi();
2563     break;
2564   case IRQ_OFFSET + IRQ_SPURIOUS:
2565     cprintf("cpu%d: spurious interrupt at %x:%x\n",
2566             cpu(), tf->cs, tf->eip);
2567     lapic_eoi();
2568     break;
2569
2570   default:
2571     if(cp == 0 || (tf->cs&3) == 0){
2572       // In kernel, it must be our mistake.
2573       cprintf("unexpected trap %d from cpu %d eip %x\n",
2574               tf->trapno, cpu(), tf->eip);
2575       panic("trap");
2576     }
2577     // In user space, assume process misbehaved.
2578     cprintf("pid %d %s: trap %d err %d on cpu %d eip %x -- kill proc\n",
2579             cp->pid, cp->name, tf->trapno, tf->err, cpu(), tf->eip);
2580     cp->killed = 1;
2581   }
2582
2583   // Force process exit if it has been killed and is in user space.
2584   // (If it is still executing in the kernel, let it keep running
2585   // until it gets to the regular system call return.)
2586   if(cp && cp->killed && (tf->cs&3) == DPL_USER)
2587     exit();
2588
2589   // Force process to give up CPU on clock tick.
2590   // If interrupts were on while locks held, would need to check nlock.
2591   if(cp && cp->state == RUNNING && tf->trapno == IRQ_OFFSET+IRQ_TIMER)
2592     yield();
2593 }
2594
2595
2596
2597
2598
2599
```

```
2600 // System call numbers
2601 #define SYS_fork    1
2602 #define SYS_exit    2
2603 #define SYS_wait    3
2604 #define SYS_pipe    4
2605 #define SYS_write   5
2606 #define SYS_read    6
2607 #define SYS_close   7
2608 #define SYS_kill    8
2609 #define SYS_exec    9
2610 #define SYS_open   10
2611 #define SYS_mknod  11
2612 #define SYS_unlink 12
2613 #define SYS_fstat  13
2614 #define SYS_link   14
2615 #define SYS_mkdir  15
2616 #define SYS_chdir  16
2617 #define SYS_dup    17
2618 #define SYS_getpid 18
2619 #define SYS_sbrk   19
2620 #define SYS_sleep  20
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
```



```
2650 #include "types.h"
2651 #include "defs.h"
2652 #include "param.h"
2653 #include "mmu.h"
2654 #include "proc.h"
2655 #include "x86.h"
2656 #include "syscall.h"
2657
2658 // User code makes a system call with INT T_SYSCALL.
2659 // System call number in %eax.
2660 // Arguments on the stack, from the user call to the C
2661 // library system call function. The saved user %esp points
2662 // to a saved program counter, and then the first argument.
2663
2664 // Fetch the int at addr from process p.
2665 int
2666 fetchint(struct proc *p, uint addr, int *ip)
2667 {
2668   if(addr >= p->sz || addr+4 > p->sz)
2669     return -1;
2670   *ip = *(int*)(p->mem + addr);
2671   return 0;
2672 }
2673
2674 // Fetch the nul-terminated string at addr from process p.
2675 // Doesn't actually copy the string - just sets *pp to point at it.
2676 // Returns length of string, not including nul.
2677 int
2678 fetchstr(struct proc *p, uint addr, char **pp)
2679 {
2680   char *s, *ep;
2681
2682   if(addr >= p->sz)
2683     return -1;
2684   *pp = p->mem + addr;
2685   ep = p->mem + p->sz;
2686   for(s = *pp; s < ep; s++)
2687     if(*s == 0)
2688       return s - *pp;
2689   return -1;
2690 }
2691
2692 // Fetch the nth 32-bit system call argument.
2693 int
2694 argint(int n, int *ip)
2695 {
2696   return fetchint(cp, cp->tf->esp + 4 + 4*n, ip);
2697 }
2698
2699
```

```
2700 // Fetch the nth word-sized system call argument as a pointer
2701 // to a block of memory of size n bytes.  Check that the pointer
2702 // lies within the process address space.
2703 int
2704 argptr(int n, char **pp, int size)
2705 {
2706   int i;
2707
2708   if(argint(n, &i) < 0)
2709     return -1;
2710   if((uint)i >= cp->sz || (uint)i+size >= cp->sz)
2711     return -1;
2712   *pp = cp->mem + i;
2713   return 0;
2714 }
2715
2716 // Fetch the nth word-sized system call argument as a string pointer.
2717 // Check that the pointer is valid and the string is nul-terminated.
2718 // (There is no shared writable memory, so the string can't change
2719 // between this check and being used by the kernel.)
2720 int
2721 argstr(int n, char **pp)
2722 {
2723   int addr;
2724   if(argint(n, &addr) < 0)
2725     return -1;
2726   return fetchstr(cp, addr, pp);
2727 }
2728
2729 extern int sys_chdir(void);
2730 extern int sys_close(void);
2731 extern int sys_dup(void);
2732 extern int sys_exec(void);
2733 extern int sys_exit(void);
2734 extern int sys_fork(void);
2735 extern int sys_fstat(void);
2736 extern int sys_getpid(void);
2737 extern int sys_kill(void);
2738 extern int sys_link(void);
2739 extern int sys_mkdir(void);
2740 extern int sys_mknod(void);
2741 extern int sys_open(void);
2742 extern int sys_pipe(void);
2743 extern int sys_read(void);
2744 extern int sys_sbrk(void);
2745 extern int sys_sleep(void);
2746 extern int sys_unlink(void);
2747 extern int sys_wait(void);
2748 extern int sys_write(void);
2749
```



```
2750 static int (*syscalls[])(void) = {
2751 [SYS_chdir]   sys_chdir,
2752 [SYS_close]   sys_close,
2753 [SYS_dup]     sys_dup,
2754 [SYS_exec]    sys_exec,
2755 [SYS_exit]    sys_exit,
2756 [SYS_fork]    sys_fork,
2757 [SYS_fstat]   sys_fstat,
2758 [SYS_getpid]  sys_getpid,
2759 [SYS_kill]    sys_kill,
2760 [SYS_link]    sys_link,
2761 [SYS_mkdir]   sys_mkdir,
2762 [SYS_mknod]   sys_mknod,
2763 [SYS_open]    sys_open,
2764 [SYS_pipe]    sys_pipe,
2765 [SYS_read]    sys_read,
2766 [SYS_sbrk]    sys_sbrk,
2767 [SYS_sleep]   sys_sleep,
2768 [SYS_unlink]  sys_unlink,
2769 [SYS_wait]    sys_wait,
2770 [SYS_write]   sys_write,
2771 };
2772
2773 void
2774 syscall(void)
2775 {
2776   int num;
2777
2778   num = cp->tf->eax;
2779   if(num >= 0 && num < NELEM(syscalls) && syscalls[num])
2780     cp->tf->eax = syscalls[num]();
2781   else {
2782     cprintf("%d %s: unknown sys call %d\n",
2783             cp->pid, cp->name, num);
2784     cp->tf->eax = -1;
2785   }
2786 }
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
```

```
2800 #include "types.h"
2801 #include "defs.h"
2802 #include "param.h"
2803 #include "mmu.h"
2804 #include "proc.h"
2805
2806 int
2807 sys_fork(void)
2808 {
2809   int pid;
2810   struct proc *np;
2811
2812   if((np = copyproc(cp)) == 0)
2813     return -1;
2814   pid = np->pid;
2815   np->state = RUNNABLE;
2816   return pid;
2817 }
2818
2819 int
2820 sys_exit(void)
2821 {
2822   exit();
2823   return 0;  // not reached
2824 }
2825
2826 int
2827 sys_wait(void)
2828 {
2829   return wait();
2830 }
2831
2832 int
2833 sys_kill(void)
2834 {
2835   int pid;
2836
2837   if(argint(0, &pid) < 0)
2838     return -1;
2839   return kill(pid);
2840 }
2841
2842 int
2843 sys_getpid(void)
2844 {
2845   return cp->pid;
2846 }
2847
2848
2849
```



```
2850 int
2851 sys_sbrk(void)
2852 {
2853   int addr;
2854   int n;
2855
2856   if(argint(0, &n) < 0)
2857     return -1;
2858   if((addr = growproc(n)) < 0)
2859     return -1;
2860   return addr;
2861 }
2862
2863 int
2864 sys_sleep(void)
2865 {
2866   int n, ticks0;
2867
2868   if(argint(0, &n) < 0)
2869     return -1;
2870   acquire(&tickslock);
2871   ticks0 = ticks;
2872   while(ticks - ticks0 < n){
2873     if(cp->killed){
2874       release(&tickslock);
2875       return -1;
2876     }
2877     sleep(&ticks, &tickslock);
2878   }
2879   release(&tickslock);
2880   return 0;
2881 }
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
```

```
2900 struct buf {
2901   int flags;
2902   uint dev;
2903   uint sector;
2904   struct buf *prev; // LRU cache list
2905   struct buf *next;
2906   struct buf *qnext; // disk queue
2907   uchar data[512];
2908 };
2909 #define B_BUSY  0x1  // buffer is locked by some process
2910 #define B_VALID 0x2  // buffer has been read from disk
2911 #define B_DIRTY 0x4  // buffer needs to be written to disk
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
```



```
2950 struct devsw {
2951   int (*read)(struct inode*, char*, int);
2952   int (*write)(struct inode*, char*, int);
2953 };
2954
2955 extern struct devsw devsw[];
2956
2957 #define CONSOLE 1
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
```

```
3000 #define O_RDONLY  0x000
3001 #define O_WRONLY  0x001
3002 #define O_RDWR    0x002
3003 #define O_CREATE  0x200
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
```



```
3050 struct stat {
3051   int dev;     // Device number
3052   uint ino;    // Inode number on device
3053   short type;  // Type of file
3054   short nlink; // Number of links to file
3055   uint size;   // Size of file in bytes
3056 };
3057
3058
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
```

```
3100 struct file {
3101   enum { FD_CLOSED, FD_NONE, FD_PIPE, FD_INODE } type;
3102   int ref; // reference count
3103   char readable;
3104   char writable;
3105   struct pipe *pipe;
3106   struct inode *ip;
3107   uint off;
3108 };
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
```



```
3150 // On-disk file system format.
3151 // Both the kernel and user programs use this header file.
3152
3153 // Block 0 is unused.
3154 // Block 1 is super block.
3155 // Inodes start at block 2.
3156
3157 #define BSIZE 512  // block size
3158
3159 // File system super block
3160 struct superblock {
3161   uint size;         // Size of file system image (blocks)
3162   uint nblocks;      // Number of data blocks
3163   uint ninodes;      // Number of inodes.
3164 };
3165
3166 #define NADDRS (NDIRECT+1)
3167 #define NDIRECT 12
3168 #define INDIRECT 12
3169 #define NINDIRECT (BSIZE / sizeof(uint))
3170 #define MAXFILE (NDIRECT  + NINDIRECT)
3171
3172 // On-disk inode structure
3173 struct dinode {
3174   short type;           // File type
3175   short major;          // Major device number (T_DEV only)
3176   short minor;          // Minor device number (T_DEV only)
3177   short nlink;          // Number of links to inode in file system
3178   uint size;            // Size of file (bytes)
3179   uint addrs[NADDRS];   // Data block addresses
3180 };
3181
3182 #define T_DIR  1   // Directory
3183 #define T_FILE 2   // File
3184 #define T_DEV  3   // Special device
3185
3186 // Inodes per block.
3187 #define IPB           (BSIZE / sizeof(struct dinode))
3188
3189 // Block containing inode i
3190 #define IBLOCK(i)     ((i) / IPB + 2)
3191
3192 // Bitmap bits per block
3193 #define BPB           (BSIZE*8)
3194
3195 // Block containing bit for block b
3196 #define BBLOCK(b, ninodes) (b/BPB + (ninodes)/IPB + 3)
3197
3198 // Directory is a file containing a sequence of dirent structures.
3199 #define DIRSIZ 14
```

```
3200 struct dirent {
3201   ushort inum;
3202   char name[DIRSIZ];
3203 };
3204
3205
3206
3207
3208
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
```



```
3250 // in-core file system types
3251
3252 struct inode {
3253   uint dev;           // Device number
3254   uint inum;          // Inode number
3255   int ref;            // Reference count
3256   int flags;          // I_BUSY, I_VALID
3257
3258   short type;         // copy of disk inode
3259   short major;
3260   short minor;
3261   short nlink;
3262   uint size;
3263   uint addrs[NADDRS];
3264 };
3265
3266 #define I_BUSY 0x1
3267 #define I_VALID 0x2
3268
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
```

```
3300 // Simple PIO-based (non-DMA) IDE driver code.
3301
3302 #include "types.h"
3303 #include "defs.h"
3304 #include "param.h"
3305 #include "mmu.h"
3306 #include "proc.h"
3307 #include "x86.h"
3308 #include "traps.h"
3309 #include "spinlock.h"
3310 #include "buf.h"
3311
3312 #define IDE_BSY       0x80
3313 #define IDE_DRDY      0x40
3314 #define IDE_DF        0x20
3315 #define IDE_ERR       0x01
3316
3317 #define IDE_CMD_READ  0x20
3318 #define IDE_CMD_WRITE 0x30
3319
3320 // ide_queue points to the buf now being read/written to the disk.
3321 // ide_queue->qnext points to the next buf to be processed.
3322 // You must hold ide_lock while manipulating queue.
3323
3324 static struct spinlock ide_lock;
3325 static struct buf *ide_queue;
3326
3327 static int disk_1_present;
3328 static void ide_start_request();
3329
3330 // Wait for IDE disk to become ready.
3331 static int
3332 ide_wait_ready(int check_error)
3333 {
3334   int r;
3335
3336   while(((r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
3337     ;
3338   if(check_error && (r & (IDE_DF|IDE_ERR)) != 0)
3339     return -1;
3340   return 0;
3341 }
3342
3343
3344
3345
3346
3347
3348
3349
```



```
3350 void
3351 ide_init(void)
3352 {
3353   int i;
3354
3355   initlock(&ide_lock, "ide");
3356   pic_enable(IRQ_IDE);
3357   ioapic_enable(IRQ_IDE, ncpu - 1);
3358   ide_wait_ready(0);
3359
3360   // Check if disk 1 is present
3361   outb(0x1f6, 0xe0 | (1<<4));
3362   for(i=0; i<1000; i++){
3363     if(inb(0x1f7) != 0){
3364       disk_1_present = 1;
3365       break;
3366     }
3367   }
3368
3369   // Switch back to disk 0.
3370   outb(0x1f6, 0xe0 | (0<<4));
3371 }
3372
3373 // Start the request for b.  Caller must hold ide_lock.
3374 static void
3375 ide_start_request(struct buf *b)
3376 {
3377   if(b == 0)
3378     panic("ide_start_request");
3379
3380   ide_wait_ready(0);
3381   outb(0x3f6, 0);  // generate interrupt
3382   outb(0x1f2, 1);  // number of sectors
3383   outb(0x1f3, b->sector & 0xff);
3384   outb(0x1f4, (b->sector >> 8) & 0xff);
3385   outb(0x1f5, (b->sector >> 16) & 0xff);
3386   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
3387   if(b->flags & B_DIRTY){
3388     outb(0x1f7, IDE_CMD_WRITE);
3389     outsl(0x1f0, b->data, 512/4);
3390   } else {
3391     outb(0x1f7, IDE_CMD_READ);
3392   }
3393 }
3394
3395
3396
3397
3398
3399
```

```
3400 // Interrupt handler.
3401 void
3402 ide_intr(void)
3403 {
3404   struct buf *b;
3405
3406   acquire(&ide_lock);
3407   if((b = ide_queue) == 0){
3408     release(&ide_lock);
3409     return;
3410   }
3411
3412   // Read data if needed.
3413   if(!(b->flags & B_DIRTY) && ide_wait_ready(1) >= 0)
3414     insl(0x1f0, b->data, 512/4);
3415
3416   // Wake process waiting for this buf.
3417   b->flags |= B_VALID;
3418   b->flags &= ~B_DIRTY;
3419   wakeup(b);
3420
3421   // Start disk on next buf in queue.
3422   if((ide_queue = b->qnext) != 0)
3423     ide_start_request(ide_queue);
3424
3425   release(&ide_lock);
3426 }
3427
3428 // Sync buf with disk.
3429 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
3430 // Else if B_VALID is not set, read buf from disk, set B_VALID.
3431 void
3432 ide_rw(struct buf *b)
3433 {
3434   struct buf **pp;
3435
3436   if(!(b->flags & B_BUSY))
3437     panic("ide_rw: buf not busy");
3438   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
3439     panic("ide_rw: nothing to do");
3440   if(b->dev != 0 && !disk_1_present)
3441     panic("ide disk 1 not present");
3442
3443   acquire(&ide_lock);
3444
3445   // Append b to ide_queue.
3446   b->qnext = 0;
3447   for(pp=&ide_queue; *pp; pp=&(*pp)->qnext)
3448     ;
3449   *pp = b;
```



```
3450   // Start disk if necessary.
3451   if(ide_queue == b)
3452     ide_start_request(b);
3453
3454   // Wait for request to finish.
3455   // Assuming will not sleep too long: ignore cp->killed.
3456   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
3457     sleep(b, &ide_lock);
3458
3459   release(&ide_lock);
3460 }
3461
3462
3463
3464
3465
3466
3467
3468
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3479
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
```

```
3500 // Buffer cache.
3501 //
3502 // The buffer cache is a linked list of buf structures holding
3503 // cached copies of disk block contents.  Caching disk blocks
3504 // in memory reduces the number of disk reads and also provides
3505 // a synchronization point for disk blocks used by multiple processes.
3506 //
3507 // Interface:
3508 // * To get a buffer for a particular disk block, call bread.
3509 // * After changing buffer data, call bwrite to flush it to disk.
3510 // * When done with the buffer, call brelse.
3511 // * Do not use the buffer after calling brelse.
3512 // * Only one process at a time can use a buffer,
3513 //     so do not keep them longer than necessary.
3514 //
3515 // The implementation uses three state flags internally:
3516 // * B_BUSY: the block has been returned from bread
3517 //     and has not been passed back to brelse.
3518 // * B_VALID: the buffer data has been initialized
3519 //     with the associated disk block contents.
3520 // * B_DIRTY: the buffer data has been modified
3521 //     and needs to be written to disk.
3522
3523 #include "types.h"
3524 #include "defs.h"
3525 #include "param.h"
3526 #include "spinlock.h"
3527 #include "buf.h"
3528
3529 struct buf buf[NBUF];
3530 struct spinlock buf_table_lock;
3531
3532 // Linked list of all buffers, through prev/next.
3533 // bufhead->next is most recently used.
3534 // bufhead->tail is least recently used.
3535 struct buf bufhead;
3536
3537 void
3538 binit(void)
3539 {
3540   struct buf *b;
3541
3542   initlock(&buf_table_lock, "buf_table");
3543
3544   // Create linked list of buffers
3545   bufhead.prev = &bufhead;
3546   bufhead.next = &bufhead;
3547   for(b = buf; b < buf+NBUF; b++){
3548     b->next = bufhead.next;
3549     b->prev = &bufhead;
```



```
3550     bufhead.next->prev = b;
3551     bufhead.next = b;
3552   }
3553 }
3554
3555 // Look through buffer cache for sector on device dev.
3556 // If not found, allocate fresh block.
3557 // In either case, return locked buffer.
3558 static struct buf*
3559 bget(uint dev, uint sector)
3560 {
3561   struct buf *b;
3562
3563   acquire(&buf_table_lock);
3564
3565  loop:
3566   // Try for cached block.
3567   for(b = bufhead.next; b != &bufhead; b = b->next){
3568     if((b->flags & (B_BUSY|B_VALID)) &&
3569        b->dev == dev && b->sector == sector){
3570       if(b->flags & B_BUSY){
3571         sleep(buf, &buf_table_lock);
3572         goto loop;
3573       }
3574       b->flags |= B_BUSY;
3575       release(&buf_table_lock);
3576       return b;
3577     }
3578   }
3579
3580   // Allocate fresh block.
3581   for(b = bufhead.prev; b != &bufhead; b = b->prev){
3582     if((b->flags & B_BUSY) == 0){
3583       b->flags = B_BUSY;
3584       b->dev = dev;
3585       b->sector = sector;
3586       release(&buf_table_lock);
3587       return b;
3588     }
3589   }
3590   panic("bget: no buffers");
3591 }
3592
3593
3594
3595
3596
3597
3598
3599
```

```
3600 // Return a B_BUSY buf with the contents of the indicated disk sector.
3601 struct buf*
3602 bread(uint dev, uint sector)
3603 {
3604   struct buf *b;
3605
3606   b = bget(dev, sector);
3607   if(!(b->flags & B_VALID))
3608     ide_rw(b);
3609   return b;
3610 }
3611
3612 // Write buf's contents to disk.  Must be locked.
3613 void
3614 bwrite(struct buf *b)
3615 {
3616   if((b->flags & B_BUSY) == 0)
3617     panic("bwrite");
3618   b->flags |= B_DIRTY;
3619   ide_rw(b);
3620 }
3621
3622 // Release the buffer buf.
3623 void
3624 brelse(struct buf *b)
3625 {
3626   if((b->flags & B_BUSY) == 0)
3627     panic("brelse");
3628
3629   acquire(&buf_table_lock);
3630
3631   b->next->prev = b->prev;
3632   b->prev->next = b->next;
3633   b->next = bufhead.next;
3634   b->prev = &bufhead;
3635   bufhead.next->prev = b;
3636   bufhead.next = b;
3637
3638   b->flags &= ~B_BUSY;
3639   wakeup(buf);
3640
3641   release(&buf_table_lock);
3642 }
3643
3644
3645
3646
3647
3648
3649
```



```
3650 // File system implementation.  Four layers:
3651 //   + Blocks: allocator for raw disk blocks.
3652 //   + Files: inode allocator, reading, writing, metadata.
3653 //   + Directories: inode with special contents (list of other inodes!)
3654 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
3655 //
3656 // Disk layout is: superblock, inodes, block in-use bitmap, data blocks.
3657 //
3658 // This file contains the low-level file system manipulation
3659 // routines.  The (higher-level) system call implementations
3660 // are in sysfile.c.
3661
3662 #include "types.h"
3663 #include "defs.h"
3664 #include "param.h"
3665 #include "stat.h"
3666 #include "mmu.h"
3667 #include "proc.h"
3668 #include "spinlock.h"
3669 #include "buf.h"
3670 #include "fs.h"
3671 #include "fsvar.h"
3672 #include "dev.h"
3673
3674 #define min(a, b) ((a) < (b) ? (a) : (b))
3675 static void itrunc(struct inode*);
3676
3677 // Read the super block.
3678 static void
3679 readsb(int dev, struct superblock *sb)
3680 {
3681   struct buf *bp;
3682
3683   bp = bread(dev, 1);
3684   memmove(sb, bp->data, sizeof(*sb));
3685   brelse(bp);
3686 }
3687
3688 // Zero a block.
3689 static void
3690 bzero(int dev, int bno)
3691 {
3692   struct buf *bp;
3693
3694   bp = bread(dev, bno);
3695   memset(bp->data, 0, BSIZE);
3696   bwrite(bp);
3697   brelse(bp);
3698 }
3699
```

```
3700 // Blocks.
3701
3702 // Allocate a disk block.
3703 static uint
3704 balloc(uint dev)
3705 {
3706   int b, bi, m;
3707   struct buf *bp;
3708   struct superblock sb;
3709
3710   bp = 0;
3711   readsb(dev, &sb);
3712   for(b = 0; b < sb.size; b += BPB){
3713     bp = bread(dev, BBLOCK(b, sb.ninodes));
3714     for(bi = 0; bi < BPB; bi++){
3715       m = 1 << (bi % 8);
3716       if((bp->data[bi/8] & m) == 0){  // Is block free?
3717         bp->data[bi/8] |= m;  // Mark block in use on disk.
3718         bwrite(bp);
3719         brelse(bp);
3720         return b + bi;
3721       }
3722     }
3723     brelse(bp);
3724   }
3725   panic("balloc: out of blocks");
3726 }
3727
3728 // Free a disk block.
3729 static void
3730 bfree(int dev, uint b)
3731 {
3732   struct buf *bp;
3733   struct superblock sb;
3734   int bi, m;
3735
3736   bzero(dev, b);
3737
3738   readsb(dev, &sb);
3739   bp = bread(dev, BBLOCK(b, sb.ninodes));
3740   bi = b % BPB;
3741   m = 1 << (bi % 8);
3742   if((bp->data[bi/8] & m) == 0)
3743     panic("freeing free block");
3744   bp->data[bi/8] &= ~m;  // Mark block free on disk.
3745   bwrite(bp);
3746   brelse(bp);
3747 }
3748
3749
```



```
3750 // Inodes.
3751 //
3752 // An inode is a single, unnamed file in the file system.
3753 // The inode disk structure holds metadata (the type, device numbers,
3754 // and data size) along with a list of blocks where the associated
3755 // data can be found.
3756 //
3757 // The inodes are laid out sequentially on disk immediately after
3758 // the superblock.  The kernel keeps a cache of the in-use
3759 // on-disk structures to provide a place for synchronizing access
3760 // to inodes shared between multiple processes.
3761 //
3762 // ip->ref counts the number of pointer references to this cached
3763 // inode; references are typically kept in struct file and in cp->cwd.
3764 // When ip->ref falls to zero, the inode is no longer cached.
3765 // It is an error to use an inode without holding a reference to it.
3766 //
3767 // Processes are only allowed to read and write inode
3768 // metadata and contents when holding the inode's lock,
3769 // represented by the I_BUSY flag in the in-memory copy.
3770 // Because inode locks are held during disk accesses,
3771 // they are implemented using a flag rather than with
3772 // spin locks.  Callers are responsible for locking
3773 // inodes before passing them to routines in this file; leaving
3774 // this responsibility with the caller makes it possible for them
3775 // to create arbitrarily-sized atomic operations.
3776 //
3777 // To give maximum control over locking to the callers,
3778 // the routines in this file that return inode pointers
3779 // return pointers to *unlocked* inodes.  It is the callers'
3780 // responsibility to lock them before using them.  A non-zero
3781 // ip->ref keeps these unlocked inodes in the cache.
3782
3783 struct {
3784   struct spinlock lock;
3785   struct inode inode[NINODE];
3786 } icache;
3787
3788 void
3789 iinit(void)
3790 {
3791   initlock(&icache.lock, "icache.lock");
3792 }
3793
3794
3795
3796
3797
3798
3799
```

```
3800 // Find the inode with number inum on device dev
3801 // and return the in-memory copy.
3802 static struct inode*
3803 iget(uint dev, uint inum)
3804 {
3805   struct inode *ip, *empty;
3806
3807   acquire(&icache.lock);
3808
3809   // Try for cached inode.
3810   empty = 0;
3811   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
3812     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
3813       ip->ref++;
3814       release(&icache.lock);
3815       return ip;
3816     }
3817     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
3818       empty = ip;
3819   }
3820
3821   // Allocate fresh inode.
3822   if(empty == 0)
3823     panic("iget: no inodes");
3824
3825   ip = empty;
3826   ip->dev = dev;
3827   ip->inum = inum;
3828   ip->ref = 1;
3829   ip->flags = 0;
3830   release(&icache.lock);
3831
3832   return ip;
3833 }
3834
3835 // Increment reference count for ip.
3836 // Returns ip to enable ip = idup(ip1) idiom.
3837 struct inode*
3838 idup(struct inode *ip)
3839 {
3840   acquire(&icache.lock);
3841   ip->ref++;
3842   release(&icache.lock);
3843   return ip;
3844 }
3845
3846
3847
3848
3849
```



```
3850 // Lock the given inode.
3851 void
3852 ilock(struct inode *ip)
3853 {
3854   struct buf *bp;
3855   struct dinode *dip;
3856
3857   if(ip == 0 || ip->ref < 1)
3858     panic("ilock");
3859
3860   acquire(&icache.lock);
3861   while(ip->flags & I_BUSY)
3862     sleep(ip, &icache.lock);
3863   ip->flags |= I_BUSY;
3864   release(&icache.lock);
3865
3866   if(!(ip->flags & I_VALID)){
3867     bp = bread(ip->dev, IBLOCK(ip->inum));
3868     dip = (struct dinode*)bp->data + ip->inum%IPB;
3869     ip->type = dip->type;
3870     ip->major = dip->major;
3871     ip->minor = dip->minor;
3872     ip->nlink = dip->nlink;
3873     ip->size = dip->size;
3874     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
3875     brelse(bp);
3876     ip->flags |= I_VALID;
3877     if(ip->type == 0)
3878       panic("ilock: no type");
3879   }
3880 }
3881
3882 // Unlock the given inode.
3883 void
3884 iunlock(struct inode *ip)
3885 {
3886   if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
3887     panic("iunlock");
3888
3889   acquire(&icache.lock);
3890   ip->flags &= ~I_BUSY;
3891   wakeup(ip);
3892   release(&icache.lock);
3893 }
3894
3895
3896
3897
3898
3899
```

```
3900 // Caller holds reference to unlocked ip.  Drop reference.
3901 void
3902 iput(struct inode *ip)
3903 {
3904   acquire(&icache.lock);
3905   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
3906     // inode is no longer used: truncate and free inode.
3907     if(ip->flags & I_BUSY)
3908       panic("iput busy");
3909     ip->flags |= I_BUSY;
3910     release(&icache.lock);
3911     itrunc(ip);
3912     ip->type = 0;
3913     iupdate(ip);
3914     acquire(&icache.lock);
3915     ip->flags &= ~I_BUSY;
3916     wakeup(ip);
3917   }
3918   ip->ref--;
3919   release(&icache.lock);
3920 }
3921
3922 // Common idiom: unlock, then put.
3923 void
3924 iunlockput(struct inode *ip)
3925 {
3926   iunlock(ip);
3927   iput(ip);
3928 }
3929
3930 // Allocate a new inode with the given type on device dev.
3931 struct inode*
3932 ialloc(uint dev, short type)
3933 {
3934   int inum;
3935   struct buf *bp;
3936   struct dinode *dip;
3937   struct superblock sb;
3938
3939   readsb(dev, &sb);
3940   for(inum = 1; inum < sb.ninodes; inum++){  // loop over inode blocks
3941     bp = bread(dev, IBLOCK(inum));
3942     dip = (struct dinode*)bp->data + inum%IPB;
3943     if(dip->type == 0){  // a free inode
3944       memset(dip, 0, sizeof(*dip));
3945       dip->type = type;
3946       bwrite(bp);   // mark it allocated on the disk
3947       brelse(bp);
3948       return iget(dev, inum);
3949     }
```



```
3950     brelse(bp);
3951   }
3952   panic("ialloc: no inodes");
3953 }
3954
3955 // Copy inode, which has changed, from memory to disk.
3956 void
3957 iupdate(struct inode *ip)
3958 {
3959   struct buf *bp;
3960   struct dinode *dip;
3961
3962   bp = bread(ip->dev, IBLOCK(ip->inum));
3963   dip = (struct dinode*)bp->data + ip->inum%IPB;
3964   dip->type = ip->type;
3965   dip->major = ip->major;
3966   dip->minor = ip->minor;
3967   dip->nlink = ip->nlink;
3968   dip->size = ip->size;
3969   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
3970   bwrite(bp);
3971   brelse(bp);
3972 }
3973
3974 // Inode contents
3975 //
3976 // The contents (data) associated with each inode is stored
3977 // in a sequence of blocks on the disk.  The first NDIRECT blocks
3978 // are listed in ip->addrs[].  The next NINDIRECT blocks are
3979 // listed in the block ip->addrs[INDIRECT].
3980
3981 // Return the disk block address of the nth block in inode ip.
3982 // If there is no such block, alloc controls whether one is allocated.
3983 static uint
3984 bmap(struct inode *ip, uint bn, int alloc)
3985 {
3986   uint addr, *a;
3987   struct buf *bp;
3988
3989   if(bn < NDIRECT){
3990     if((addr = ip->addrs[bn]) == 0){
3991       if(!alloc)
3992         return -1;
3993       ip->addrs[bn] = addr = balloc(ip->dev);
3994     }
3995     return addr;
3996   }
3997   bn -= NDIRECT;
3998
3999
```

```
4000   if(bn < NINDIRECT){
4001     // Load indirect block, allocating if necessary.
4002     if((addr = ip->addrs[INDIRECT]) == 0){
4003       if(!alloc)
4004         return -1;
4005       ip->addrs[INDIRECT] = addr = balloc(ip->dev);
4006     }
4007     bp = bread(ip->dev, addr);
4008     a = (uint*)bp->data;
4009
4010     if((addr = a[bn]) == 0){
4011       if(!alloc){
4012         brelse(bp);
4013         return -1;
4014       }
4015       a[bn] = addr = balloc(ip->dev);
4016       bwrite(bp);
4017     }
4018     brelse(bp);
4019     return addr;
4020   }
4021
4022   panic("bmap: out of range");
4023 }
4024
4025 // Truncate inode (discard contents).
4026 static void
4027 itrunc(struct inode *ip)
4028 {
4029   int i, j;
4030   struct buf *bp;
4031   uint *a;
4032
4033   for(i = 0; i < NDIRECT; i++){
4034     if(ip->addrs[i]){
4035       bfree(ip->dev, ip->addrs[i]);
4036       ip->addrs[i] = 0;
4037     }
4038   }
4039
4040   if(ip->addrs[INDIRECT]){
4041     bp = bread(ip->dev, ip->addrs[INDIRECT]);
4042     a = (uint*)bp->data;
4043     for(j = 0; j < NINDIRECT; j++){
4044       if(a[j])
4045         bfree(ip->dev, a[j]);
4046     }
4047     brelse(bp);
4048     ip->addrs[INDIRECT] = 0;
4049   }
```



```
4050   ip->size = 0;
4051   iupdate(ip);
4052 }
4053
4054 // Copy stat information from inode.
4055 void
4056 stati(struct inode *ip, struct stat *st)
4057 {
4058   st->dev = ip->dev;
4059   st->ino = ip->inum;
4060   st->type = ip->type;
4061   st->nlink = ip->nlink;
4062   st->size = ip->size;
4063 }
4064
4065 // Read data from inode.
4066 int
4067 readi(struct inode *ip, char *dst, uint off, uint n)
4068 {
4069   uint tot, m;
4070   struct buf *bp;
4071
4072   if(ip->type == T_DEV){
4073     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
4074       return -1;
4075     return devsw[ip->major].read(ip, dst, n);
4076   }
4077
4078   if(off > ip->size || off + n < off)
4079     return -1;
4080   if(off + n > ip->size)
4081     n = ip->size - off;
4082
4083   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
4084     bp = bread(ip->dev, bmap(ip, off/BSIZE, 0));
4085     m = min(n - tot, BSIZE - off%BSIZE);
4086     memmove(dst, bp->data + off%BSIZE, m);
4087     brelse(bp);
4088   }
4089   return n;
4090 }
4091
4092
4093
4094
4095
4096
4097
4098
4099
```

```
4100 // Write data to inode.
4101 int
4102 writei(struct inode *ip, char *src, uint off, uint n)
4103 {
4104   uint tot, m;
4105   struct buf *bp;
4106
4107   if(ip->type == T_DEV){
4108     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
4109       return -1;
4110     return devsw[ip->major].write(ip, src, n);
4111   }
4112
4113   if(off + n < off)
4114     return -1;
4115   if(off + n > MAXFILE*BSIZE)
4116     n = MAXFILE*BSIZE - off;
4117
4118   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
4119     bp = bread(ip->dev, bmap(ip, off/BSIZE, 1));
4120     m = min(n - tot, BSIZE - off%BSIZE);
4121     memmove(bp->data + off%BSIZE, src, m);
4122     bwrite(bp);
4123     brelse(bp);
4124   }
4125
4126   if(n > 0 && off > ip->size){
4127     ip->size = off;
4128     iupdate(ip);
4129   }
4130   return n;
4131 }
4132
4133 // Directories
4134
4135 int
4136 namecmp(const char *s, const char *t)
4137 {
4138   return strncmp(s, t, DIRSIZ);
4139 }
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```



```
4150 // Look for a directory entry in a directory.
4151 // If found, set *poff to byte offset of entry.
4152 // Caller must have already locked dp.
4153 struct inode*
4154 dirlookup(struct inode *dp, char *name, uint *poff)
4155 {
4156   uint off, inum;
4157   struct buf *bp;
4158   struct dirent *de;
4159
4160   if(dp->type != T_DIR)
4161     panic("dirlookup not DIR");
4162
4163   for(off = 0; off < dp->size; off += BSIZE){
4164     bp = bread(dp->dev, bmap(dp, off / BSIZE, 0));
4165     for(de = (struct dirent*)bp->data;
4166         de < (struct dirent*)(bp->data + BSIZE);
4167         de++){
4168       if(de->inum == 0)
4169         continue;
4170       if(namecmp(name, de->name) == 0){
4171         // entry matches path element
4172         if(poff)
4173           *poff = off + (uchar*)de - bp->data;
4174         inum = de->inum;
4175         brelse(bp);
4176         return iget(dp->dev, inum);
4177       }
4178     }
4179     brelse(bp);
4180   }
4181   return 0;
4182 }
4183
4184 // Write a new directory entry (name, ino) into the directory dp.
4185 int
4186 dirlink(struct inode *dp, char *name, uint ino)
4187 {
4188   int off;
4189   struct dirent de;
4190   struct inode *ip;
4191
4192   // Check that name is not present.
4193   if((ip = dirlookup(dp, name, 0)) != 0){
4194     iput(ip);
4195     return -1;
4196   }
4197
4198
4199
```

```
4200   // Look for an empty dirent.
4201   for(off = 0; off < dp->size; off += sizeof(de)){
4202     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4203       panic("dirlink read");
4204     if(de.inum == 0)
4205       break;
4206   }
4207
4208   strncpy(de.name, name, DIRSIZ);
4209   de.inum = ino;
4210   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4211     panic("dirlink");
4212
4213   return 0;
4214 }
4215
4216 // Paths
4217
4218 // Copy the next path element from path into name.
4219 // Return a pointer to the element following the copied one.
4220 // The returned path has no leading slashes,
4221 // so the caller can check *path=='\0' to see if the name is the last one.
4222 // If no name to remove, return 0.
4223 //
4224 // Examples:
4225 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
4226 //   skipelem("///a//bb", name) = "bb", setting name = "a"
4227 //   skipelem("", name) = skipelem("////", name) = 0
4228 //
4229 static char*
4230 skipelem(char *path, char *name)
4231 {
4232   char *s;
4233   int len;
4234
4235   while(*path == '/')
4236     path++;
4237   if(*path == 0)
4238     return 0;
4239   s = path;
4240   while(*path != '/' && *path != 0)
4241     path++;
4242   len = path - s;
4243   if(len >= DIRSIZ)
4244     memmove(name, s, DIRSIZ);
4245   else {
4246     memmove(name, s, len);
4247     name[len] = 0;
4248   }
4249   while(*path == '/')
```



```
4250     path++;
4251   return path;
4252 }
4253
4254 // Look up and return the inode for a path name.
4255 // If parent != 0, return the inode for the parent and copy the final
4256 // path element into name, which must have room for DIRSIZ bytes.
4257 static struct inode*
4258 _namei(char *path, int parent, char *name)
4259 {
4260   struct inode *ip, *next;
4261
4262   if(*path == '/')
4263     ip = iget(ROOTDEV, 1);
4264   else
4265     ip = idup(cp->cwd);
4266
4267   while((path = skipelem(path, name)) != 0){
4268     ilock(ip);
4269     if(ip->type != T_DIR){
4270       iunlockput(ip);
4271       return 0;
4272     }
4273     if(parent && *path == '\0'){
4274       // Stop one level early.
4275       iunlock(ip);
4276       return ip;
4277     }
4278     if((next = dirlookup(ip, name, 0)) == 0){
4279       iunlockput(ip);
4280       return 0;
4281     }
4282     iunlockput(ip);
4283     ip = next;
4284   }
4285   if(parent){
4286     iput(ip);
4287     return 0;
4288   }
4289   return ip;
4290 }
4291
4292 struct inode*
4293 namei(char *path)
4294 {
4295   char name[DIRSIZ];
4296   return _namei(path, 0, name);
4297 }
4298
4299
```

```
4300 struct inode*
4301 nameiparent(char *path, char *name)
4302 {
4303   return _namei(path, 1, name);
4304 }
4305
4306
4307
4308
4309
4310
4311
4312
4313
4314
4315
4316
4317
4318
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4330
4331
4332
4333
4334
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
```



```
4350 #include "types.h"
4351 #include "defs.h"
4352 #include "param.h"
4353 #include "file.h"
4354 #include "spinlock.h"
4355 #include "dev.h"
4356
4357 struct devsw devsw[NDEV];
4358 struct spinlock file_table_lock;
4359 struct file file[NFILE];
4360
4361 void
4362 fileinit(void)
4363 {
4364   initlock(&file_table_lock, "file_table");
4365 }
4366
4367 // Allocate a file structure.
4368 struct file*
4369 filealloc(void)
4370 {
4371   int i;
4372
4373   acquire(&file_table_lock);
4374   for(i = 0; i < NFILE; i++){
4375     if(file[i].type == FD_CLOSED){
4376       file[i].type = FD_NONE;
4377       file[i].ref = 1;
4378       release(&file_table_lock);
4379       return file + i;
4380     }
4381   }
4382   release(&file_table_lock);
4383   return 0;
4384 }
4385
4386 // Increment ref count for file f.
4387 struct file*
4388 filedup(struct file *f)
4389 {
4390   acquire(&file_table_lock);
4391   if(f->ref < 1 || f->type == FD_CLOSED)
4392     panic("filedup");
4393   f->ref++;
4394   release(&file_table_lock);
4395   return f;
4396 }
4397
4398
4399
```

```
4400 // Close file f.  (Decrement ref count, close when reaches 0.)
4401 void
4402 fileclose(struct file *f)
4403 {
4404   struct file ff;
4405
4406   acquire(&file_table_lock);
4407   if(f->ref < 1 || f->type == FD_CLOSED)
4408     panic("fileclose");
4409   if(--f->ref > 0){
4410     release(&file_table_lock);
4411     return;
4412   }
4413   ff = *f;
4414   f->ref = 0;
4415   f->type = FD_CLOSED;
4416   release(&file_table_lock);
4417
4418   if(ff.type == FD_PIPE)
4419     pipeclose(ff.pipe, ff.writable);
4420   else if(ff.type == FD_INODE)
4421     iput(ff.ip);
4422   else
4423     panic("fileclose");
4424 }
4425
4426 // Get metadata about file f.
4427 int
4428 filestat(struct file *f, struct stat *st)
4429 {
4430   if(f->type == FD_INODE){
4431     ilock(f->ip);
4432     stati(f->ip, st);
4433     iunlock(f->ip);
4434     return 0;
4435   }
4436   return -1;
4437 }
4438
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
```



```
4450 // Read from file f.  Addr is kernel address.
4451 int
4452 fileread(struct file *f, char *addr, int n)
4453 {
4454   int r;
4455
4456   if(f->readable == 0)
4457     return -1;
4458   if(f->type == FD_PIPE)
4459     return piperead(f->pipe, addr, n);
4460   if(f->type == FD_INODE){
4461     ilock(f->ip);
4462     if((r = readi(f->ip, addr, f->off, n)) > 0)
4463       f->off += r;
4464     iunlock(f->ip);
4465     return r;
4466   }
4467   panic("fileread");
4468 }
4469
4470 // Write to file f.  Addr is kernel address.
4471 int
4472 filewrite(struct file *f, char *addr, int n)
4473 {
4474   int r;
4475
4476   if(f->writable == 0)
4477     return -1;
4478   if(f->type == FD_PIPE)
4479     return pipewrite(f->pipe, addr, n);
4480   if(f->type == FD_INODE){
4481     ilock(f->ip);
4482     if((r = writei(f->ip, addr, f->off, n)) > 0)
4483       f->off += r;
4484     iunlock(f->ip);
4485     return r;
4486   }
4487   panic("filewrite");
4488 }
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499
```

```
4500 #include "types.h"
4501 #include "defs.h"
4502 #include "param.h"
4503 #include "stat.h"
4504 #include "mmu.h"
4505 #include "proc.h"
4506 #include "fs.h"
4507 #include "fsvar.h"
4508 #include "file.h"
4509 #include "fcntl.h"
4510
4511 // Fetch the nth word-sized system call argument as a file descriptor
4512 // and return both the descriptor and the corresponding struct file.
4513 static int
4514 argfd(int n, int *pfd, struct file **pf)
4515 {
4516   int fd;
4517   struct file *f;
4518
4519   if(argint(n, &fd) < 0)
4520     return -1;
4521   if(fd < 0 || fd >= NOFILE || (f=cp->ofile[fd]) == 0)
4522     return -1;
4523   if(pfd)
4524     *pfd = fd;
4525   if(pf)
4526     *pf = f;
4527   return 0;
4528 }
4529
4530 // Allocate a file descriptor for the given file.
4531 // Takes over file reference from caller on success.
4532 static int
4533 fdalloc(struct file *f)
4534 {
4535   int fd;
4536
4537   for(fd = 0; fd < NOFILE; fd++){
4538     if(cp->ofile[fd] == 0){
4539       cp->ofile[fd] = f;
4540       return fd;
4541     }
4542   }
4543   return -1;
4544 }
4545
4546
4547
4548
4549
```



```
4550 int
4551 sys_read(void)
4552 {
4553   struct file *f;
4554   int n;
4555   char *p;
4556
4557   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
4558     return -1;
4559   return fileread(f, p, n);
4560 }
4561
4562 int
4563 sys_write(void)
4564 {
4565   struct file *f;
4566   int n;
4567   char *p;
4568
4569   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
4570     return -1;
4571   return filewrite(f, p, n);
4572 }
4573
4574 int
4575 sys_dup(void)
4576 {
4577   struct file *f;
4578   int fd;
4579
4580   if(argfd(0, 0, &f) < 0)
4581     return -1;
4582   if((fd=fdalloc(f)) < 0)
4583     return -1;
4584   filedup(f);
4585   return fd;
4586 }
4587
4588 int
4589 sys_close(void)
4590 {
4591   int fd;
4592   struct file *f;
4593
4594   if(argfd(0, &fd, &f) < 0)
4595     return -1;
4596   cp->ofile[fd] = 0;
4597   fileclose(f);
4598   return 0;
4599 }
```

```
4600 int
4601 sys_fstat(void)
4602 {
4603   struct file *f;
4604   struct stat *st;
4605
4606   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
4607     return -1;
4608   return filestat(f, st);
4609 }
4610
4611 // Create the path new as a link to the same inode as old.
4612 int
4613 sys_link(void)
4614 {
4615   char name[DIRSIZ], *new, *old;
4616   struct inode *dp, *ip;
4617
4618   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
4619     return -1;
4620   if((ip = namei(old)) == 0)
4621     return -1;
4622   ilock(ip);
4623   if(ip->type == T_DIR){
4624     iunlockput(ip);
4625     return -1;
4626   }
4627   ip->nlink++;
4628   iupdate(ip);
4629   iunlock(ip);
4630
4631   if((dp = nameiparent(new, name)) == 0)
4632     goto  bad;
4633   ilock(dp);
4634   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0)
4635     goto bad;
4636   iunlockput(dp);
4637   iput(ip);
4638   return 0;
4639
4640 bad:
4641   if(dp)
4642     iunlockput(dp);
4643   ilock(ip);
4644   ip->nlink--;
4645   iupdate(ip);
4646   iunlockput(ip);
4647   return -1;
4648 }
4649
```



```
4650 // Is the directory dp empty except for "." and ".." ?
4651 static int
4652 isdirempty(struct inode *dp)
4653 {
4654   int off;
4655   struct dirent de;
4656
4657   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
4658     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4659       panic("isdirempty: readi");
4660     if(de.inum != 0)
4661       return 0;
4662   }
4663   return 1;
4664 }
4665
4666 int
4667 sys_unlink(void)
4668 {
4669   struct inode *ip, *dp;
4670   struct dirent de;
4671   char name[DIRSIZ], *path;
4672   uint off;
4673
4674   if(argstr(0, &path) < 0)
4675     return -1;
4676   if((dp = nameiparent(path, name)) == 0)
4677     return -1;
4678   ilock(dp);
4679
4680   // Cannot unlink "." or "..".
4681   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0){
4682     iunlockput(dp);
4683     return -1;
4684   }
4685
4686   if((ip = dirlookup(dp, name, &off)) == 0){
4687     iunlockput(dp);
4688     return -1;
4689   }
4690   ilock(ip);
4691
4692   if(ip->nlink < 1)
4693     panic("unlink: nlink < 1");
4694   if(ip->type == T_DIR && !isdirempty(ip)){
4695     iunlockput(ip);
4696     iunlockput(dp);
4697     return -1;
4698   }
4699
```

```
4700   memset(&de, 0, sizeof(de));
4701   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4702     panic("unlink: writei");
4703   iunlockput(dp);
4704
4705   ip->nlink--;
4706   iupdate(ip);
4707   iunlockput(ip);
4708   return 0;
4709 }
4710
4711 static struct inode*
4712 create(char *path, int canexist, short type, short major, short minor)
4713 {
4714   uint off;
4715   struct inode *ip, *dp;
4716   char name[DIRSIZ];
4717
4718   if((dp = nameiparent(path, name)) == 0)
4719     return 0;
4720   ilock(dp);
4721
4722   if(canexist && (ip = dirlookup(dp, name, &off)) != 0){
4723     iunlockput(dp);
4724     ilock(ip);
4725     if(ip->type != type || ip->major != major || ip->minor != minor){
4726       iunlockput(ip);
4727       return 0;
4728     }
4729     return ip;
4730   }
4731
4732   if((ip = ialloc(dp->dev, type)) == 0){
4733     iunlockput(dp);
4734     return 0;
4735   }
4736   ilock(ip);
4737   ip->major = major;
4738   ip->minor = minor;
4739   ip->nlink = 1;
4740   iupdate(ip);
4741
4742   if(dirlink(dp, name, ip->inum) < 0){
4743     ip->nlink = 0;
4744     iunlockput(ip);
4745     iunlockput(dp);
4746     return 0;
4747   }
4748
4749
```



```
4750   if(type == T_DIR){  // Create . and .. entries.
4751     dp->nlink++;  // for ".."
4752     iupdate(dp);
4753     // No ip->nlink++ for ".": avoid cyclic ref count.
4754     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
4755       panic("create dots");
4756   }
4757   iunlockput(dp);
4758   return ip;
4759 }
4760
4761 int
4762 sys_open(void)
4763 {
4764   char *path;
4765   int fd, omode;
4766   struct file *f;
4767   struct inode *ip;
4768
4769   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
4770     return -1;
4771
4772   if(omode & O_CREATE){
4773     if((ip = create(path, 1, T_FILE, 0, 0)) == 0)
4774       return -1;
4775   } else {
4776     if((ip = namei(path)) == 0)
4777       return -1;
4778     ilock(ip);
4779     if(ip->type == T_DIR && (omode & (O_RDWR|O_WRONLY))){
4780       iunlockput(ip);
4781       return -1;
4782     }
4783   }
4784
4785   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
4786     if(f)
4787       fileclose(f);
4788     iunlockput(ip);
4789     return -1;
4790   }
4791   iunlock(ip);
4792
4793   f->type = FD_INODE;
4794   f->ip = ip;
4795   f->off = 0;
4796   f->readable = !(omode & O_WRONLY);
4797   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
4798
4799
```

```
4800   return fd;
4801 }
4802
4803 int
4804 sys_mknod(void)
4805 {
4806   struct inode *ip;
4807   char *path;
4808   int len;
4809   int major, minor;
4810
4811   if((len=argstr(0, &path)) < 0 ||
4812      argint(1, &major) < 0 ||
4813      argint(2, &minor) < 0 ||
4814      (ip = create(path, 0, T_DEV, major, minor)) == 0)
4815     return -1;
4816   iunlockput(ip);
4817   return 0;
4818 }
4819
4820 int
4821 sys_mkdir(void)
4822 {
4823   char *path;
4824   struct inode *ip;
4825
4826   if(argstr(0, &path) < 0 || (ip = create(path, 0, T_DIR, 0, 0)) == 0)
4827     return -1;
4828   iunlockput(ip);
4829   return 0;
4830 }
4831
4832 int
4833 sys_chdir(void)
4834 {
4835   char *path;
4836   struct inode *ip;
4837
4838   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0)
4839     return -1;
4840   ilock(ip);
4841   if(ip->type != T_DIR){
4842     iunlockput(ip);
4843     return -1;
4844   }
4845   iunlock(ip);
4846   iput(cp->cwd);
4847   cp->cwd = ip;
4848   return 0;
4849 }
```



```
4850 int
4851 sys_exec(void)
4852 {
4853   char *path, *argv[20];
4854   int i;
4855   uint uargv, uarg;
4856
4857   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0)
4858     return -1;
4859   memset(argv, 0, sizeof(argv));
4860   for(i=0;; i++){
4861     if(i >= NELEM(argv))
4862       return -1;
4863     if(fetchint(cp, uargv+4*i, (int*)&uarg) < 0)
4864       return -1;
4865     if(uarg == 0){
4866       argv[i] = 0;
4867       break;
4868     }
4869     if(fetchstr(cp, uarg, &argv[i]) < 0)
4870       return -1;
4871   }
4872   return exec(path, argv);
4873 }
4874
4875 int
4876 sys_pipe(void)
4877 {
4878   int *fd;
4879   struct file *rf, *wf;
4880   int fd0, fd1;
4881
4882   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
4883     return -1;
4884   if(pipealloc(&rf, &wf) < 0)
4885     return -1;
4886   fd0 = -1;
4887   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
4888     if(fd0 >= 0)
4889       cp->ofile[fd0] = 0;
4890     fileclose(rf);
4891     fileclose(wf);
4892     return -1;
4893   }
4894   fd[0] = fd0;
4895   fd[1] = fd1;
4896   return 0;
4897 }
4898
4899
```

```
4900 #include "types.h"
4901 #include "param.h"
4902 #include "mmu.h"
4903 #include "proc.h"
4904 #include "defs.h"
4905 #include "x86.h"
4906 #include "elf.h"
4907
4908 int
4909 exec(char *path, char **argv)
4910 {
4911   char *mem, *s, *last;
4912   int i, argc, arglen, len, off;
4913   uint sz, sp, argp;
4914   struct elfhdr elf;
4915   struct inode *ip;
4916   struct proghdr ph;
4917
4918   if((ip = namei(path)) == 0)
4919     return -1;
4920   ilock(ip);
4921
4922   // Compute memory size of new process.
4923   mem = 0;
4924   sz = 0;
4925
4926   // Program segments.
4927   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
4928     goto bad;
4929   if(elf.magic != ELF_MAGIC)
4930     goto bad;
4931   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
4932     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
4933       goto bad;
4934     if(ph.type != ELF_PROG_LOAD)
4935       continue;
4936     if(ph.memsz < ph.filesz)
4937       goto bad;
4938     sz += ph.memsz;
4939   }
4940
4941   // Arguments.
4942   arglen = 0;
4943   for(argc=0; argv[argc]; argc++)
4944     arglen += strlen(argv[argc]) + 1;
4945   arglen = (arglen+3) & ~3;
4946   sz += arglen + 4*(argc+1);
4947
4948   // Stack.
4949   sz += PAGE;
```



```
4950   // Allocate program memory.
4951   sz = (sz+PAGE-1) & ~(PAGE-1);
4952   mem = kalloc(sz);
4953   if(mem == 0)
4954     goto bad;
4955   memset(mem, 0, sz);
4956
4957   // Load program into memory.
4958   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
4959     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
4960       goto bad;
4961     if(ph.type != ELF_PROG_LOAD)
4962       continue;
4963     if(ph.va + ph.memsz > sz)
4964       goto bad;
4965     if(readi(ip, mem + ph.va, ph.offset, ph.filesz) != ph.filesz)
4966       goto bad;
4967     memset(mem + ph.va + ph.filesz, 0, ph.memsz - ph.filesz);
4968   }
4969   iunlockput(ip);
4970
4971   // Initialize stack.
4972   sp = sz;
4973   argp = sz - arglen - 4*(argc+1);
4974
4975   // Copy argv strings and pointers to stack.
4976   *(uint*)(mem+argp + 4*argc) = 0;  // argv[argc]
4977   for(i=argc-1; i>=0; i--){
4978     len = strlen(argv[i]) + 1;
4979     sp -= len;
4980     memmove(mem+sp, argv[i], len);
4981     *(uint*)(mem+argp + 4*i) = sp;  // argv[i]
4982   }
4983
4984   // Stack frame for main(argc, argv), below arguments.
4985   sp = argp;
4986   sp -= 4;
4987   *(uint*)(mem+sp) = argp;
4988   sp -= 4;
4989   *(uint*)(mem+sp) = argc;
4990   sp -= 4;
4991   *(uint*)(mem+sp) = 0xffffffff;   // fake return pc
4992
4993   // Save program name for debugging.
4994   for(last=s=path; *s; s++)
4995     if(*s == '/')
4996       last = s+1;
4997   safestrcpy(cp->name, last, sizeof(cp->name));
4998
4999
```

```
5000   // Commit to the new image.
5001   kfree(cp->mem, cp->sz);
5002   cp->mem = mem;
5003   cp->sz = sz;
5004   cp->tf->eip = elf.entry;  // main
5005   cp->tf->esp = sp;
5006   setupsegs(cp);
5007   return 0;
5008
5009  bad:
5010   if(mem)
5011     kfree(mem, sz);
5012   iunlockput(ip);
5013   return -1;
5014 }
5015
5016
5017
5018
5019
5020
5021
5022
5023
5024
5025
5026
5027
5028
5029
5030
5031
5032
5033
5034
5035
5036
5037
5038
5039
5040
5041
5042
5043
5044
5045
5046
5047
5048
5049
```



```
5050 #include "types.h"
5051 #include "defs.h"
5052 #include "param.h"
5053 #include "mmu.h"
5054 #include "proc.h"
5055 #include "file.h"
5056 #include "spinlock.h"
5057
5058 #define PIPESIZE 512
5059
5060 struct pipe {
5061   int readopen;   // read fd is still open
5062   int writeopen;  // write fd is still open
5063   uint writep;    // next index to write
5064   uint readp;     // next index to read
5065   struct spinlock lock;
5066   char data[PIPESIZE];
5067 };
5068
5069 int
5070 pipealloc(struct file **f0, struct file **f1)
5071 {
5072   struct pipe *p;
5073
5074   p = 0;
5075   *f0 = *f1 = 0;
5076   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
5077     goto bad;
5078   if((p = (struct pipe*)kalloc(PAGE)) == 0)
5079     goto bad;
5080   p->readopen = 1;
5081   p->writeopen = 1;
5082   p->writep = 0;
5083   p->readp = 0;
5084   initlock(&p->lock, "pipe");
5085   (*f0)->type = FD_PIPE;
5086   (*f0)->readable = 1;
5087   (*f0)->writable = 0;
5088   (*f0)->pipe = p;
5089   (*f1)->type = FD_PIPE;
5090   (*f1)->readable = 0;
5091   (*f1)->writable = 1;
5092   (*f1)->pipe = p;
5093   return 0;
5094
5095  bad:
5096   if(p)
5097     kfree((char*)p, PAGE);
5098   if(*f0){
5099     (*f0)->type = FD_NONE;
```

```
5100     fileclose(*f0);
5101   }
5102   if(*f1){
5103     (*f1)->type = FD_NONE;
5104     fileclose(*f1);
5105   }
5106   return -1;
5107 }
5108
5109 void
5110 pipeclose(struct pipe *p, int writable)
5111 {
5112   acquire(&p->lock);
5113   if(writable){
5114     p->writeopen = 0;
5115     wakeup(&p->readp);
5116   } else {
5117     p->readopen = 0;
5118     wakeup(&p->writep);
5119   }
5120   release(&p->lock);
5121
5122   if(p->readopen == 0 && p->writeopen == 0)
5123     kfree((char*)p, PAGE);
5124 }
5125
5126 int
5127 pipewrite(struct pipe *p, char *addr, int n)
5128 {
5129   int i;
5130
5131   acquire(&p->lock);
5132   for(i = 0; i < n; i++){
5133     while(p->writep == p->readp + PIPESIZE) {
5134       if(p->readopen == 0 || cp->killed){
5135         release(&p->lock);
5136         return -1;
5137       }
5138       wakeup(&p->readp);
5139       sleep(&p->writep, &p->lock);
5140     }
5141     p->data[p->writep++ % PIPESIZE] = addr[i];
5142   }
5143   wakeup(&p->readp);
5144   release(&p->lock);
5145   return i;
5146 }
5147
5148
5149
```



```
5150 int
5151 piperead(struct pipe *p, char *addr, int n)
5152 {
5153   int i;
5154
5155   acquire(&p->lock);
5156   while(p->readp == p->writep && p->writeopen){
5157     if(cp->killed){
5158       release(&p->lock);
5159       return -1;
5160     }
5161     sleep(&p->readp, &p->lock);
5162   }
5163   for(i = 0; i < n; i++){
5164     if(p->readp == p->writep)
5165       break;
5166     addr[i] = p->data[p->readp++ % PIPESIZE];
5167   }
5168   wakeup(&p->writep);
5169   release(&p->lock);
5170   return i;
5171 }
5172
5173
5174
5175
5176
5177
5178
5179
5180
5181
5182
5183
5184
5185
5186
5187
5188
5189
5190
5191
5192
5193
5194
5195
5196
5197
5198
5199
```

```
5200 #include "types.h"
5201
5202 void*
5203 memset(void *dst, int c, uint n)
5204 {
5205   char *d;
5206
5207   d = (char*)dst;
5208   while(n-- > 0)
5209     *d++ = c;
5210
5211   return dst;
5212 }
5213
5214 int
5215 memcmp(const void *v1, const void *v2, uint n)
5216 {
5217   const uchar *s1, *s2;
5218
5219   s1 = v1;
5220   s2 = v2;
5221   while(n-- > 0){
5222     if(*s1 != *s2)
5223       return *s1 - *s2;
5224     s1++, s2++;
5225   }
5226
5227   return 0;
5228 }
5229
5230 void*
5231 memmove(void *dst, const void *src, uint n)
5232 {
5233   const char *s;
5234   char *d;
5235
5236   s = src;
5237   d = dst;
5238   if(s < d && s + n > d){
5239     s += n;
5240     d += n;
5241     while(n-- > 0)
5242       *--d = *--s;
5243   } else
5244     while(n-- > 0)
5245       *d++ = *s++;
5246
5247   return dst;
5248 }
5249
```



```
5250 int
5251 strncmp(const char *p, const char *q, uint n)
5252 {
5253   while(n > 0 && *p && *p == *q)
5254     n--, p++, q++;
5255   if(n == 0)
5256     return 0;
5257   return (uchar)*p - (uchar)*q;
5258 }
5259
5260 char*
5261 strncpy(char *s, const char *t, int n)
5262 {
5263   char *os;
5264
5265   os = s;
5266   while(n-- > 0 && (*s++ = *t++) != 0)
5267     ;
5268   while(n-- > 0)
5269     *s++ = 0;
5270   return os;
5271 }
5272
5273 // Like strncpy but guaranteed to NUL-terminate.
5274 char*
5275 safestrcpy(char *s, const char *t, int n)
5276 {
5277   char *os;
5278
5279   os = s;
5280   if(n <= 0)
5281     return os;
5282   while(--n > 0 && (*s++ = *t++) != 0)
5283     ;
5284   *s = 0;
5285   return os;
5286 }
5287
5288 int
5289 strlen(const char *s)
5290 {
5291   int n;
5292
5293   for(n = 0; s[n]; n++)
5294     ;
5295   return n;
5296 }
5297
5298
5299
```

```
5300 // See MultiProcessor Specification Version 1.[14]
5301
5302 struct mp {             // floating pointer
5303   uchar signature[4];           // "_MP_"
5304   void *physaddr;               // phys addr of MP config table
5305   uchar length;                 // 1
5306   uchar specrev;                // [14]
5307   uchar checksum;               // all bytes must add up to 0
5308   uchar type;                   // MP system config type
5309   uchar imcrp;
5310   uchar reserved[3];
5311 };
5312
5313 struct mpconf {         // configuration table header
5314   uchar signature[4];           // "PCMP"
5315   ushort length;                // total table length
5316   uchar version;                // [14]
5317   uchar checksum;               // all bytes must add up to 0
5318   uchar product[20];            // product id
5319   uint *oemtable;               // OEM table pointer
5320   ushort oemlength;             // OEM table length
5321   ushort entry;                 // entry count
5322   uint *lapicaddr;              // address of local APIC
5323   ushort xlength;               // extended table length
5324   uchar xchecksum;              // extended table checksum
5325   uchar reserved;
5326 };
5327
5328 struct mpproc {         // processor table entry
5329   uchar type;                   // entry type (0)
5330   uchar apicid;                 // local APIC id
5331   uchar version;                // local APIC verison
5332   uchar flags;                  // CPU flags
5333     #define MPBOOT 0x02           // This proc is the bootstrap processor.
5334   uchar signature[4];           // CPU signature
5335   uint feature;                 // feature flags from CPUID instruction
5336   uchar reserved[8];
5337 };
5338
5339 struct mpioapic {       // I/O APIC table entry
5340   uchar type;                   // entry type (2)
5341   uchar apicno;                 // I/O APIC id
5342   uchar version;                // I/O APIC version
5343   uchar flags;                  // I/O APIC flags
5344   uint *addr;                  // I/O APIC address
5345 };
5346
5347
5348
5349
```



```
5350 // Table entry types
5351 #define MPPROC    0x00  // One per processor
5352 #define MPBUS     0x01  // One per bus
5353 #define MPIOAPIC  0x02  // One per I/O APIC
5354 #define MPIOINTR  0x03  // One per bus interrupt source
5355 #define MPLINTR   0x04  // One per system interrupt source
5356
5357
5358
5359
5360
5361
5362
5363
5364
5365
5366
5367
5368
5369
5370
5371
5372
5373
5374
5375
5376
5377
5378
5379
5380
5381
5382
5383
5384
5385
5386
5387
5388
5389
5390
5391
5392
5393
5394
5395
5396
5397
5398
5399
```

```
5400 // Multiprocessor bootstrap.
5401 // Search memory for MP description structures.
5402 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
5403
5404 #include "types.h"
5405 #include "defs.h"
5406 #include "param.h"
5407 #include "mp.h"
5408 #include "x86.h"
5409 #include "mmu.h"
5410 #include "proc.h"
5411
5412 struct cpu cpus[NCPU];
5413 static struct cpu *bcpu;
5414 int ismp;
5415 int ncpu;
5416 uchar ioapic_id;
5417
5418 int
5419 mp_bcpu(void)
5420 {
5421   return bcpu-cpus;
5422 }
5423
5424 static uchar
5425 sum(uchar *addr, int len)
5426 {
5427   int i, sum;
5428
5429   sum = 0;
5430   for(i=0; i<len; i++)
5431     sum += addr[i];
5432   return sum;
5433 }
5434
5435 // Look for an MP structure in the len bytes at addr.
5436 static struct mp*
5437 mp_search1(uchar *addr, int len)
5438 {
5439   uchar *e, *p;
5440
5441   e = addr+len;
5442   for(p = addr; p < e; p += sizeof(struct mp))
5443     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
5444       return (struct mp*)p;
5445   return 0;
5446 }
5447
5448
5449
```



```
5450 // Search for the MP Floating Pointer Structure, which according to the
5451 // spec is in one of the following three locations:
5452 // 1) in the first KB of the EBDA;
5453 // 2) in the last KB of system base memory;
5454 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
5455 static struct mp*
5456 mp_search(void)
5457 {
5458   uchar *bda;
5459   uint p;
5460   struct mp *mp;
5461
5462   bda = (uchar*)0x400;
5463   if((p = ((bda[0x0F]<<8)|bda[0x0E]) << 4)){
5464     if((mp = mp_search1((uchar*)p, 1024)))
5465       return mp;
5466   } else {
5467     p = ((bda[0x14]<<8)|bda[0x13])*1024;
5468     if((mp = mp_search1((uchar*)p-1024, 1024)))
5469       return mp;
5470   }
5471   return mp_search1((uchar*)0xF0000, 0x10000);
5472 }
5473
5474 // Search for an MP configuration table.  For now,
5475 // don't accept the default configurations (physaddr == 0).
5476 // Check for correct signature, calculate the checksum and,
5477 // if correct, check the version.
5478 // To do: check extended table checksum.
5479 static struct mpconf*
5480 mp_config(struct mp **pmp)
5481 {
5482   struct mpconf *conf;
5483   struct mp *mp;
5484
5485   if((mp = mp_search()) == 0 || mp->physaddr == 0)
5486     return 0;
5487   conf = (struct mpconf*)mp->physaddr;
5488   if(memcmp(conf, "PCMP", 4) != 0)
5489     return 0;
5490   if(conf->version != 1 && conf->version != 4)
5491     return 0;
5492   if(sum((uchar*)conf, conf->length) != 0)
5493     return 0;
5494   *pmp = mp;
5495   return conf;
5496 }
5497
5498
5499
```

```
5500 void
5501 mp_init(void)
5502 {
5503   uchar *p, *e;
5504   struct mp *mp;
5505   struct mpconf *conf;
5506   struct mpproc *proc;
5507   struct mpioapic *ioapic;
5508
5509   bcpu = &cpus[ncpu];
5510   if((conf = mp_config(&mp)) == 0)
5511     return;
5512
5513   ismp = 1;
5514   lapic = (uint*)conf->lapicaddr;
5515
5516   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
5517     switch(*p){
5518     case MPPROC:
5519       proc = (struct mpproc*)p;
5520       cpus[ncpu].apicid = proc->apicid;
5521       if(proc->flags & MPBOOT)
5522         bcpu = &cpus[ncpu];
5523       ncpu++;
5524       p += sizeof(struct mpproc);
5525       continue;
5526     case MPIOAPIC:
5527       ioapic = (struct mpioapic*)p;
5528       ioapic_id = ioapic->apicno;
5529       p += sizeof(struct mpioapic);
5530       continue;
5531     case MPBUS:
5532     case MPIOINTR:
5533     case MPLINTR:
5534       p += 8;
5535       continue;
5536     default:
5537       cprintf("mp_init: unknown config type %x\n", *p);
5538       panic("mp_init");
5539     }
5540   }
5541
5542   if(mp->imcrp){
5543     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
5544     // But it would on real hardware.
5545     outb(0x22, 0x70);   // Select IMCR
5546     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
5547   }
5548 }
5549
```



```
5550 // The local APIC manages internal (non-I/O) interrupts.
5551 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
5552
5553 #include "types.h"
5554 #include "defs.h"
5555 #include "traps.h"
5556 #include "mmu.h"
5557 #include "x86.h"
5558
5559 // Local APIC registers, divided by 4 for use as uint[] indices.
5560 #define ID      (0x0020/4)   // ID
5561 #define VER     (0x0030/4)   // Version
5562 #define TPR     (0x0080/4)   // Task Priority
5563 #define EOI     (0x00B0/4)   // EOI
5564 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
5565   #define ENABLE     0x00000100   // Unit Enable
5566 #define ESR     (0x0280/4)   // Error Status
5567 #define ICRLO   (0x0300/4)   // Interrupt Command
5568   #define INIT       0x00000500   // INIT/RESET
5569   #define STARTUP    0x00000600   // Startup IPI
5570   #define DELIVS     0x00001000   // Delivery status
5571   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
5572   #define LEVEL      0x00008000   // Level triggered
5573   #define BCAST      0x00080000   // Send to all APICs, including self.
5574 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
5575 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
5576   #define X1         0x0000000B   // divide counts by 1
5577   #define PERIODIC   0x00020000   // Periodic
5578 #define PCINT   (0x0340/4)   // Performance Counter LVT
5579 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
5580 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
5581 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
5582   #define MASKED     0x00010000   // Interrupt masked
5583 #define TICR    (0x0380/4)   // Timer Initial Count
5584 #define TCCR    (0x0390/4)   // Timer Current Count
5585 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
5586
5587 volatile uint *lapic;  // Initialized in mp.c
5588
5589 static void
5590 lapicw(int index, int value)
5591 {
5592   lapic[index] = value;
5593   lapic[ID];  // wait for write to finish, by reading
5594 }
5595
5596
5597
5598
5599
```

```
5600 void
5601 lapic_init(int c)
5602 {
5603   if(!lapic)
5604     return;
5605
5606   // Enable local APIC; set spurious interrupt vector.
5607   lapicw(SVR, ENABLE | (IRQ_OFFSET+IRQ_SPURIOUS));
5608
5609   // The timer repeatedly counts down at bus frequency
5610   // from lapic[TICR] and then issues an interrupt.
5611   // If xv6 cared more about precise timekeeping,
5612   // TICR would be calibrated using an external time source.
5613   lapicw(TDCR, X1);
5614   lapicw(TIMER, PERIODIC | (IRQ_OFFSET + IRQ_TIMER));
5615   lapicw(TICR, 10000000);
5616
5617   // Disable logical interrupt lines.
5618   lapicw(LINT0, MASKED);
5619   lapicw(LINT1, MASKED);
5620
5621   // Disable performance counter overflow interrupts
5622   // on machines that provide that interrupt entry.
5623   if(((lapic[VER]>>16) & 0xFF) >= 4)
5624     lapicw(PCINT, MASKED);
5625
5626   // Map error interrupt to IRQ_ERROR.
5627   lapicw(ERROR, IRQ_OFFSET+IRQ_ERROR);
5628
5629   // Clear error status register (requires back-to-back writes).
5630   lapicw(ESR, 0);
5631   lapicw(ESR, 0);
5632
5633   // Ack any outstanding interrupts.
5634   lapicw(EOI, 0);
5635
5636   // Send an Init Level De-Assert to synchronise arbitration ID's.
5637   lapicw(ICRHI, 0);
5638   lapicw(ICRLO, BCAST | INIT | LEVEL);
5639   while(lapic[ICRLO] & DELIVS)
5640     ;
5641
5642   // Enable interrupts on the APIC (but not on the processor).
5643   lapicw(TPR, 0);
5644 }
5645
5646
5647
5648
5649
```



```
5650 int
5651 cpu(void)
5652 {
5653   // Cannot call cpu when interrupts are enabled:
5654   // result not guaranteed to last long enough to be used!
5655   // Would prefer to panic but even printing is chancy here:
5656   // everything, including cprintf, calls cpu, at least indirectly
5657   // through acquire and release.
5658   if(read_eflags()&FL_IF){
5659     static int n;
5660     if(n++ == 0)
5661       cprintf("cpu called from %x with interrupts enabled\n",
5662         ((uint*)read_ebp())[1]);
5663   }
5664
5665   if(lapic)
5666     return lapic[ID]>>24;
5667   return 0;
5668 }
5669
5670 // Acknowledge interrupt.
5671 void
5672 lapic_eoi(void)
5673 {
5674   if(lapic)
5675     lapicw(EOI, 0);
5676 }
5677
5678 // Spin for a given number of microseconds.
5679 // On real hardware would want to tune this dynamically.
5680 static void
5681 microdelay(int us)
5682 {
5683   volatile int j = 0;
5684
5685   while(us-- > 0)
5686     for(j=0; j<10000; j++);
5687 }
5688
5689
5690
5691
5692
5693
5694
5695
5696
5697
5698
5699
```

```
5700 #define IO_RTC  0x70
5701
5702 // Start additional processor running bootstrap code at addr.
5703 // See Appendix B of MultiProcessor Specification.
5704 void
5705 lapic_startap(uchar apicid, uint addr)
5706 {
5707   int i;
5708   ushort *wrv;
5709
5710   // "The BSP must initialize CMOS shutdown code to 0AH
5711   // and the warm reset vector (DWORD based at 40:67) to point at
5712   // the AP startup code prior to the [universal startup algorithm]."
5713   outb(IO_RTC, 0xF);  // offset 0xF is shutdown code
5714   outb(IO_RTC+1, 0x0A);
5715   wrv = (ushort*)(0x40<<4 | 0x67);  // Warm reset vector
5716   wrv[0] = 0;
5717   wrv[1] = addr >> 4;
5718
5719   // "Universal startup algorithm."
5720   // Send INIT (level-triggered) interrupt to reset other CPU.
5721   lapicw(ICRHI, apicid<<24);
5722   lapicw(ICRLO, INIT | LEVEL | ASSERT);
5723   microdelay(200);
5724   lapicw(ICRLO, INIT | LEVEL);
5725   microdelay(100); // should be 10ms, but too slow in Bochs!
5726
5727   // Send startup IPI (twice!) to enter bootstrap code.
5728   // Regular hardware is supposed to only accept a STARTUP
5729   // when it is in the halted state due to an INIT.  So the second
5730   // should be ignored, but it is part of the official Intel algorithm.
5731   // Bochs complains about the second one.  Too bad for Bochs.
5732   for(i = 0; i < 2; i++){
5733     lapicw(ICRHI, apicid<<24);
5734     lapicw(ICRLO, STARTUP | (addr>>12));
5735     microdelay(200);
5736   }
5737 }
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749
```



```
5750 // The I/O APIC manages hardware interrupts for an SMP system.
5751 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
5752 // See also picirq.c.
5753
5754 #include "types.h"
5755 #include "defs.h"
5756 #include "traps.h"
5757
5758 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
5759
5760 #define REG_ID     0x00  // Register index: ID
5761 #define REG_VER    0x01  // Register index: version
5762 #define REG_TABLE  0x10  // Redirection table base
5763
5764 // The redirection table starts at REG_TABLE and uses
5765 // two registers to configure each interrupt.
5766 // The first (low) register in a pair contains configuration bits.
5767 // The second (high) register contains a bitmask telling which
5768 // CPUs can serve that interrupt.
5769 #define INT_DISABLED   0x00010000  // Interrupt disabled
5770 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
5771 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
5772 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
5773
5774 volatile struct ioapic *ioapic;
5775
5776 // IO APIC MMIO structure: write reg, then read or write data.
5777 struct ioapic {
5778   uint reg;
5779   uint pad[3];
5780   uint data;
5781 };
5782
5783 static uint
5784 ioapic_read(int reg)
5785 {
5786   ioapic->reg = reg;
5787   return ioapic->data;
5788 }
5789
5790 static void
5791 ioapic_write(int reg, uint data)
5792 {
5793   ioapic->reg = reg;
5794   ioapic->data = data;
5795 }
5796
5797
5798
5799
```

```
5800 void
5801 ioapic_init(void)
5802 {
5803   int i, id, maxintr;
5804
5805   if(!ismp)
5806     return;
5807
5808   ioapic = (volatile struct ioapic*)IOAPIC;
5809   maxintr = (ioapic_read(REG_VER) >> 16) & 0xFF;
5810   id = ioapic_read(REG_ID) >> 24;
5811   if(id != ioapic_id)
5812     cprintf("ioapic_init: id isn't equal to ioapic_id; not a MP\n");
5813
5814   // Mark all interrupts edge-triggered, active high, disabled,
5815   // and not routed to any CPUs.
5816   for(i = 0; i <= maxintr; i++){
5817     ioapic_write(REG_TABLE+2*i, INT_DISABLED | (IRQ_OFFSET + i));
5818     ioapic_write(REG_TABLE+2*i+1, 0);
5819   }
5820 }
5821
5822 void
5823 ioapic_enable(int irq, int cpunum)
5824 {
5825   if(!ismp)
5826     return;
5827
5828   // Mark interrupt edge-triggered, active high,
5829   // enabled, and routed to the given cpunum,
5830   // which happens to be that cpu's APIC ID.
5831   ioapic_write(REG_TABLE+2*irq, IRQ_OFFSET + irq);
5832   ioapic_write(REG_TABLE+2*irq+1, cpunum << 24);
5833 }
5834
5835
5836
5837
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```



```
5850 // Intel 8259A programmable interrupt controllers.
5851
5852 #include "types.h"
5853 #include "x86.h"
5854 #include "traps.h"
5855
5856 // I/O Addresses of the two programmable interrupt controllers
5857 #define IO_PIC1         0x20    // Master (IRQs 0-7)
5858 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
5859
5860 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
5861
5862 // Current IRQ mask.
5863 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
5864 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
5865
5866 static void
5867 pic_setmask(ushort mask)
5868 {
5869   irqmask = mask;
5870   outb(IO_PIC1+1, mask);
5871   outb(IO_PIC2+1, mask >> 8);
5872 }
5873
5874 void
5875 pic_enable(int irq)
5876 {
5877   pic_setmask(irqmask & ~(1<<irq));
5878 }
5879
5880 // Initialize the 8259A interrupt controllers.
5881 void
5882 pic_init(void)
5883 {
5884   // mask all interrupts
5885   outb(IO_PIC1+1, 0xFF);
5886   outb(IO_PIC2+1, 0xFF);
5887
5888   // Set up master (8259A-1)
5889
5890   // ICW1:  0001g0hi
5891   //    g:  0 = edge triggering, 1 = level triggering
5892   //    h:  0 = cascaded PICs, 1 = master only
5893   //    i:  0 = no ICW4, 1 = ICW4 required
5894   outb(IO_PIC1, 0x11);
5895
5896   // ICW2:  Vector offset
5897   outb(IO_PIC1+1, IRQ_OFFSET);
5898
5899
```

```
5900   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
5901   //        (slave PIC) 3-bit # of slave's connection to master
5902   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
5903
5904   // ICW4:  000nbmap
5905   //    n:  1 = special fully nested mode
5906   //    b:  1 = buffered mode
5907   //    m:  0 = slave PIC, 1 = master PIC
5908   //      (ignored when b is 0, as the master/slave role
5909   //      can be hardwired).
5910   //    a:  1 = Automatic EOI mode
5911   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
5912   outb(IO_PIC1+1, 0x3);
5913
5914   // Set up slave (8259A-2)
5915   outb(IO_PIC2, 0x11);                  // ICW1
5916   outb(IO_PIC2+1, IRQ_OFFSET + 8);      // ICW2
5917   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
5918   // NB Automatic EOI mode doesn't tend to work on the slave.
5919   // Linux source code says it's "to be investigated".
5920   outb(IO_PIC2+1, 0x3);                 // ICW4
5921
5922   // OCW3:  0ef01prs
5923   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
5924   //    p:  0 = no polling, 1 = polling mode
5925   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
5926   outb(IO_PIC1, 0x68);             // clear specific mask
5927   outb(IO_PIC1, 0x0a);             // read IRR by default
5928
5929   outb(IO_PIC2, 0x68);             // OCW3
5930   outb(IO_PIC2, 0x0a);             // OCW3
5931
5932   if(irqmask != 0xFFFF)
5933     pic_setmask(irqmask);
5934 }
5935
5936
5937
5938
5939
5940
5941
5942
5943
5944
5945
5946
5947
5948
5949
```



```
5950 // PC keyboard interface constants
5951
5952 #define KBSTATP         0x64    // kbd controller status port(I)
5953 #define KBS_DIB         0x01    // kbd data in buffer
5954 #define KBDATAP         0x60    // kbd data port(I)
5955
5956 #define NO              0
5957
5958 #define SHIFT           (1<<0)
5959 #define CTL             (1<<1)
5960 #define ALT             (1<<2)
5961
5962 #define CAPSLOCK        (1<<3)
5963 #define NUMLOCK         (1<<4)
5964 #define SCROLLLOCK      (1<<5)
5965
5966 #define E0ESC           (1<<6)
5967
5968 // Special keycodes
5969 #define KEY_HOME        0xE0
5970 #define KEY_END         0xE1
5971 #define KEY_UP          0xE2
5972 #define KEY_DN          0xE3
5973 #define KEY_LF          0xE4
5974 #define KEY_RT          0xE5
5975 #define KEY_PGUP        0xE6
5976 #define KEY_PGDN        0xE7
5977 #define KEY_INS         0xE8
5978 #define KEY_DEL         0xE9
5979
5980 // C('A') == Control-A
5981 #define C(x) (x - '@')
5982
5983 static uchar shiftcode[256] =
5984 {
5985   [0x1D] CTL,
5986   [0x2A] SHIFT,
5987   [0x36] SHIFT,
5988   [0x38] ALT,
5989   [0x9D] CTL,
5990   [0xB8] ALT
5991 };
5992
5993 static uchar togglecode[256] =
5994 {
5995   [0x3A] CAPSLOCK,
5996   [0x45] NUMLOCK,
5997   [0x46] SCROLLLOCK
5998 };
5999
```

```
6000 static uchar normalmap[256] =
6001 {
6002   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
6003   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
6004   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
6005   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
6006   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
6007   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
6008   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
6009   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
6010   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
6011   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
6012   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
6013   [0x9C] '\n',      // KP_Enter
6014   [0xB5] '/',       // KP_Div
6015   [0xC8] KEY_UP,    [0xD0] KEY_DN,
6016   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
6017   [0xCB] KEY_LF,    [0xCD] KEY_RT,
6018   [0x97] KEY_HOME,  [0xCF] KEY_END,
6019   [0xD2] KEY_INS,   [0xD3] KEY_DEL
6020 };
6021
6022 static uchar shiftmap[256] =
6023 {
6024   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
6025   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
6026   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
6027   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
6028   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
6029   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
6030   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
6031   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
6032   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
6033   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
6034   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
6035   [0x9C] '\n',      // KP_Enter
6036   [0xB5] '/',       // KP_Div
6037   [0xC8] KEY_UP,    [0xD0] KEY_DN,
6038   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
6039   [0xCB] KEY_LF,    [0xCD] KEY_RT,
6040   [0x97] KEY_HOME,  [0xCF] KEY_END,
6041   [0xD2] KEY_INS,   [0xD3] KEY_DEL
6042 };
6043
6044
6045
6046
6047
6048
6049
```



```
6050 static uchar ctlmap[256] =
6051 {
6052   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
6053   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
6054   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
6055   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
6056   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
6057   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
6058   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
6059   [0x9C] '\r',      // KP_Enter
6060   [0xB5] C('/'),    // KP_Div
6061   [0xC8] KEY_UP,    [0xD0] KEY_DN,
6062   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
6063   [0xCB] KEY_LF,    [0xCD] KEY_RT,
6064   [0x97] KEY_HOME,  [0xCF] KEY_END,
6065   [0xD2] KEY_INS,   [0xD3] KEY_DEL
6066 };
6067
6068
6069
6070
6071
6072
6073
6074
6075
6076
6077
6078
6079
6080
6081
6082
6083
6084
6085
6086
6087
6088
6089
6090
6091
6092
6093
6094
6095
6096
6097
6098
6099
```

```
6100 #include "types.h"
6101 #include "x86.h"
6102 #include "defs.h"
6103 #include "kbd.h"
6104
6105 int
6106 kbd_getc(void)
6107 {
6108   static uint shift;
6109   static uchar *charcode[4] = {
6110     normalmap, shiftmap, ctlmap, ctlmap
6111   };
6112   uint st, data, c;
6113
6114   st = inb(KBSTATP);
6115   if((st & KBS_DIB) == 0)
6116     return -1;
6117   data = inb(KBDATAP);
6118
6119   if(data == 0xE0){
6120     shift |= E0ESC;
6121     return 0;
6122   } else if(data & 0x80){
6123     // Key released
6124     data = (shift & E0ESC ? data : data & 0x7F);
6125     shift &= ~(shiftcode[data] | E0ESC);
6126     return 0;
6127   } else if(shift & E0ESC){
6128     // Last character was an E0 escape; or with 0x80
6129     data |= 0x80;
6130     shift &= ~E0ESC;
6131   }
6132
6133   shift |= shiftcode[data];
6134   shift ^= togglecode[data];
6135   c = charcode[shift & (CTL | SHIFT)][data];
6136   if(shift & CAPSLOCK){
6137     if('a' <= c && c <= 'z')
6138       c += 'A' - 'a';
6139     else if('A' <= c && c <= 'Z')
6140       c += 'a' - 'A';
6141   }
6142   return c;
6143 }
6144
6145 void
6146 kbd_intr(void)
6147 {
6148   console_intr(kbd_getc);
6149 }
```



```
6150 // Console input and output.
6151 // Input is from the keyboard only.
6152 // Output is written to the screen and the printer port.
6153
6154 #include "types.h"
6155 #include "defs.h"
6156 #include "param.h"
6157 #include "traps.h"
6158 #include "spinlock.h"
6159 #include "dev.h"
6160 #include "mmu.h"
6161 #include "proc.h"
6162 #include "x86.h"
6163
6164 #define CRTPORT 0x3d4
6165 #define LPTPORT 0x378
6166 #define BACKSPACE 0x100
6167
6168 static ushort *crt = (ushort*)0xb8000;  // CGA memory
6169
6170 static struct spinlock console_lock;
6171 int panicked = 0;
6172 int use_console_lock = 0;
6173
6174 // Copy console output to parallel port, which you can tell
6175 // .bochsrc to copy to the stdout:
6176 //   parport1: enabled=1, file="/dev/stdout"
6177 static void
6178 lpt_putc(int c)
6179 {
6180   int i;
6181
6182   for(i = 0; !(inb(LPTPORT+1) & 0x80) && i < 12800; i++)
6183     ;
6184   if(c == BACKSPACE)
6185     c = '\b';
6186   outb(LPTPORT+0, c);
6187   outb(LPTPORT+2, 0x08|0x04|0x01);
6188   outb(LPTPORT+2, 0x08);
6189 }
6190
6191
6192
6193
6194
6195
6196
6197
6198
6199
```

```
6200 static void
6201 cga_putc(int c)
6202 {
6203   int pos;
6204
6205   // Cursor position: col + 80*row.
6206   outb(CRTPORT, 14);
6207   pos = inb(CRTPORT+1) << 8;
6208   outb(CRTPORT, 15);
6209   pos |= inb(CRTPORT+1);
6210
6211   if(c == '\n')
6212     pos += 80 - pos%80;
6213   else if(c == BACKSPACE){
6214     if(pos > 0)
6215       crt[--pos] = ' ' | 0x0700;
6216   } else
6217     crt[pos++] = (c&0xff) | 0x0700;  // black on white
6218
6219   if((pos/80) >= 24){  // Scroll up.
6220     memmove(crt, crt+80, sizeof(crt[0])*23*80);
6221     pos -= 80;
6222     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
6223   }
6224
6225   outb(CRTPORT, 14);
6226   outb(CRTPORT+1, pos>>8);
6227   outb(CRTPORT, 15);
6228   outb(CRTPORT+1, pos);
6229   crt[pos] = ' ' | 0x0700;
6230 }
6231
6232 void
6233 cons_putc(int c)
6234 {
6235   if(panicked){
6236     cli();
6237     for(;;)
6238       ;
6239   }
6240
6241   lpt_putc(c);
6242   cga_putc(c);
6243 }
6244
6245
6246
6247
6248
6249
```



```
6250 void
6251 printint(int xx, int base, int sgn)
6252 {
6253   static char digits[] = "0123456789ABCDEF";
6254   char buf[16];
6255   int i = 0, neg = 0;
6256   uint x;
6257
6258   if(sgn && xx < 0){
6259     neg = 1;
6260     x = 0 - xx;
6261   } else {
6262     x = xx;
6263   }
6264
6265   do{
6266     buf[i++] = digits[x % base];
6267   }while((x /= base) != 0);
6268   if(neg)
6269     buf[i++] = '-';
6270
6271   while(--i >= 0)
6272     cons_putc(buf[i]);
6273 }
6274
6275 // Print to the console. only understands %d, %x, %p, %s.
6276 void
6277 cprintf(char *fmt, ...)
6278 {
6279   int i, c, state, locking;
6280   uint *argp;
6281   char *s;
6282
6283   locking = use_console_lock;
6284   if(locking)
6285     acquire(&console_lock);
6286
6287   argp = (uint*)(void*)&fmt + 1;
6288   state = 0;
6289   for(i = 0; fmt[i]; i++){
6290     c = fmt[i] & 0xff;
6291     switch(state){
6292     case 0:
6293       if(c == '%')
6294         state = '%';
6295       else
6296         cons_putc(c);
6297       break;
6298
6299
```

```
6300     case '%':
6301       switch(c){
6302       case 'd':
6303         printint(*argp++, 10, 1);
6304         break;
6305       case 'x':
6306       case 'p':
6307         printint(*argp++, 16, 0);
6308         break;
6309       case 's':
6310         s = (char*)*argp++;
6311         if(s == 0)
6312           s = "(null)";
6313         for(; *s; s++)
6314           cons_putc(*s);
6315         break;
6316       case '%':
6317         cons_putc('%');
6318         break;
6319       default:
6320         // Print unknown % sequence to draw attention.
6321         cons_putc('%');
6322         cons_putc(c);
6323         break;
6324       }
6325       state = 0;
6326       break;
6327     }
6328   }
6329
6330   if(locking)
6331     release(&console_lock);
6332 }
6333
6334 int
6335 console_write(struct inode *ip, char *buf, int n)
6336 {
6337   int i;
6338
6339   iunlock(ip);
6340   acquire(&console_lock);
6341   for(i = 0; i < n; i++)
6342     cons_putc(buf[i] & 0xff);
6343   release(&console_lock);
6344   ilock(ip);
6345
6346   return n;
6347 }
6348
6349
```



```
6350 #define INPUT_BUF 128
6351 struct {
6352   struct spinlock lock;
6353   char buf[INPUT_BUF];
6354   uint r;  // Read index
6355   uint w;  // Write index
6356   uint e;  // Edit index
6357 } input;
6358
6359 #define C(x)  ((x)-'@')  // Control-x
6360
6361 void
6362 console_intr(int (*getc)(void))
6363 {
6364   int c;
6365
6366   acquire(&input.lock);
6367   while((c = getc()) >= 0){
6368     switch(c){
6369     case C('P'):  // Process listing.
6370       procdump();
6371       break;
6372     case C('U'):  // Kill line.
6373       while(input.e != input.w &&
6374             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
6375         input.e--;
6376         cons_putc(BACKSPACE);
6377       }
6378       break;
6379     case C('H'):  // Backspace
6380       if(input.e != input.w){
6381         input.e--;
6382         cons_putc(BACKSPACE);
6383       }
6384       break;
6385     default:
6386       if(c != 0 && input.e-input.r < INPUT_BUF){
6387         input.buf[input.e++ % INPUT_BUF] = c;
6388         cons_putc(c);
6389         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
6390           input.w = input.e;
6391           wakeup(&input.r);
6392         }
6393       }
6394       break;
6395     }
6396   }
6397   release(&input.lock);
6398 }
6399
```

```
6400 int
6401 console_read(struct inode *ip, char *dst, int n)
6402 {
6403   uint target;
6404   int c;
6405
6406   iunlock(ip);
6407   target = n;
6408   acquire(&input.lock);
6409   while(n > 0){
6410     while(input.r == input.w){
6411       if(cp->killed){
6412         release(&input.lock);
6413         ilock(ip);
6414         return -1;
6415       }
6416       sleep(&input.r, &input.lock);
6417     }
6418     c = input.buf[input.r++ % INPUT_BUF];
6419     if(c == C('D')){  // EOF
6420       if(n < target){
6421         // Save ^D for next time, to make sure
6422         // caller gets a 0-byte result.
6423         input.r--;
6424       }
6425       break;
6426     }
6427     *dst++ = c;
6428     --n;
6429     if(c == '\n')
6430       break;
6431   }
6432   release(&input.lock);
6433   ilock(ip);
6434
6435   return target - n;
6436 }
6437
6438
6439
6440
6441
6442
6443
6444
6445
6446
6447
6448
6449
```



```
6450 void
6451 console_init(void)
6452 {
6453   initlock(&console_lock, "console");
6454   initlock(&input.lock, "console input");
6455
6456   devsw[CONSOLE].write = console_write;
6457   devsw[CONSOLE].read = console_read;
6458   use_console_lock = 1;
6459
6460   pic_enable(IRQ_KBD);
6461   ioapic_enable(IRQ_KBD, 0);
6462 }
6463
6464 void
6465 panic(char *s)
6466 {
6467   int i;
6468   uint pcs[10];
6469
6470   __asm __volatile("cli");
6471   use_console_lock = 0;
6472   cprintf("cpu%d: panic: ", cpu());
6473   cprintf(s);
6474   cprintf("\n");
6475   getcallerpcs(&s, pcs);
6476   for(i=0; i<10; i++)
6477     cprintf(" %p", pcs[i]);
6478   panicked = 1; // freeze other CPU
6479   for(;;)
6480     ;
6481 }
6482
6483
6484
6485
6486
6487
6488
6489
6490
6491
6492
6493
6494
6495
6496
6497
6498
6499
```

```
6500 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
6501 // Only used on uniprocessors;
6502 // SMP machines use the local APIC timer.
6503
6504 #include "types.h"
6505 #include "defs.h"
6506 #include "traps.h"
6507 #include "x86.h"
6508
6509 #define IO_TIMER1       0x040           // 8253 Timer #1
6510
6511 // Frequency of all three count-down timers;
6512 // (TIMER_FREQ/freq) is the appropriate count
6513 // to generate a frequency of freq Hz.
6514
6515 #define TIMER_FREQ      1193182
6516 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
6517
6518 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
6519 #define TIMER_SEL0      0x00    // select counter 0
6520 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
6521 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
6522
6523 void
6524 timer_init(void)
6525 {
6526   // Interrupt 100 times/sec.
6527   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
6528   outb(IO_TIMER1, TIMER_DIV(100) % 256);
6529   outb(IO_TIMER1, TIMER_DIV(100) / 256);
6530   pic_enable(IRQ_TIMER);
6531 }
6532
6533
6534
6535
6536
6537
6538
6539
6540
6541
6542
6543
6544
6545
6546
6547
6548
6549
```



```
6550 // Blank page
6551
6552
6553
6554
6555
6556
6557
6558
6559
6560
6561
6562
6563
6564
6565
6566
6567
6568
6569
6570
6571
6572
6573
6574
6575
6576
6577
6578
6579
6580
6581
6582
6583
6584
6585
6586
6587
6588
6589
6590
6591
6592
6593
6594
6595
6596
6597
6598
6599
```

```
# Initial process execs /init.

#include "syscall.h"
#include "traps.h"

# exec(init, argv)
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0
  movl $SYS_exec, %eax
  int $T_SYSCALL

# for(;;) exit();
exit:
  movl $SYS_exit, %eax
  int $T_SYSCALL
  jmp exit

# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .long init
  .long 0
```



```
// init: The initial user-level program

#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

char *sh_args[] = { "sh", 0 };

int
main(void)
{
  int pid, wpid;

  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr

  for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", sh_args);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

```
6700 #include "syscall.h"
6701 #include "traps.h"
6702
6703 #define STUB(name) \
6704   .globl name; \
6705   name: \
6706     movl $SYS_ ## name, %eax; \
6707     int $T_SYSCALL; \
6708     ret
6709
6710 STUB(fork)
6711 STUB(exit)
6712 STUB(wait)
6713 STUB(pipe)
6714 STUB(read)
6715 STUB(write)
6716 STUB(close)
6717 STUB(kill)
6718 STUB(exec)
6719 STUB(open)
6720 STUB(mknod)
6721 STUB(unlink)
6722 STUB(fstat)
6723 STUB(link)
6724 STUB(mkdir)
6725 STUB(chdir)
6726 STUB(dup)
6727 STUB(getpid)
6728 STUB(sbrk)
6729 STUB(sleep)
6730
6731
6732
6733
6734
6735
6736
6737
6738
6739
6740
6741
6742
6743
6744
6745
6746
6747
6748
6749
```



```
6750 // Shell.
6751
6752 #include "types.h"
6753 #include "user.h"
6754 #include "fcntl.h"
6755
6756 // Parsed command representation
6757 #define EXEC  1
6758 #define REDIR 2
6759 #define PIPE  3
6760 #define LIST  4
6761 #define BACK  5
6762
6763 #define MAXARGS 10
6764
6765 struct cmd {
6766   int type;
6767 };
6768
6769 struct execcmd {
6770   int type;
6771   char *argv[MAXARGS];
6772   char *eargv[MAXARGS];
6773 };
6774
6775 struct redircmd {
6776   int type;
6777   struct cmd *cmd;
6778   char *file;
6779   char *efile;
6780   int mode;
6781   int fd;
6782 };
6783
6784 struct pipecmd {
6785   int type;
6786   struct cmd *left;
6787   struct cmd *right;
6788 };
6789
6790 struct listcmd {
6791   int type;
6792   struct cmd *left;
6793   struct cmd *right;
6794 };
6795
6796 struct backcmd {
6797   int type;
6798   struct cmd *cmd;
6799 };
```

```
6800 int fork1(void);  // Fork but panics on failure.
6801 void panic(char*);
6802 struct cmd *parsecmd(char*);
6803
6804 // Execute cmd.  Never returns.
6805 void
6806 runcmd(struct cmd *cmd)
6807 {
6808   int p[2];
6809   struct backcmd *bcmd;
6810   struct execcmd *ecmd;
6811   struct listcmd *lcmd;
6812   struct pipecmd *pcmd;
6813   struct redircmd *rcmd;
6814
6815   if(cmd == 0)
6816     exit();
6817
6818   switch(cmd->type){
6819   default:
6820     panic("runcmd");
6821
6822   case EXEC:
6823     ecmd = (struct execcmd*)cmd;
6824     if(ecmd->argv[0] == 0)
6825       exit();
6826     exec(ecmd->argv[0], ecmd->argv);
6827     printf(2, "exec %s failed\n", ecmd->argv[0]);
6828     break;
6829
6830   case REDIR:
6831     rcmd = (struct redircmd*)cmd;
6832     close(rcmd->fd);
6833     if(open(rcmd->file, rcmd->mode) < 0){
6834       printf(2, "open %s failed\n", rcmd->file);
6835       exit();
6836     }
6837     runcmd(rcmd->cmd);
6838     break;
6839
6840   case LIST:
6841     lcmd = (struct listcmd*)cmd;
6842     if(fork1() == 0)
6843       runcmd(lcmd->left);
6844     wait();
6845     runcmd(lcmd->right);
6846     break;
6847
6848
6849
```



```
6850   case PIPE:
6851     pcmd = (struct pipecmd*)cmd;
6852     if(pipe(p) < 0)
6853       panic("pipe");
6854     if(fork1() == 0){
6855       close(1);
6856       dup(p[1]);
6857       close(p[0]);
6858       close(p[1]);
6859       runcmd(pcmd->left);
6860     }
6861     if(fork1() == 0){
6862       close(0);
6863       dup(p[0]);
6864       close(p[0]);
6865       close(p[1]);
6866       runcmd(pcmd->right);
6867     }
6868     close(p[0]);
6869     close(p[1]);
6870     wait();
6871     wait();
6872     break;
6873
6874   case BACK:
6875     bcmd = (struct backcmd*)cmd;
6876     if(fork1() == 0)
6877       runcmd(bcmd->cmd);
6878     break;
6879   }
6880   exit();
6881 }
6882
6883 int
6884 getcmd(char *buf, int nbuf)
6885 {
6886   printf(2, "$ ");
6887   memset(buf, 0, nbuf);
6888   gets(buf, nbuf);
6889   if(buf[0] == 0) // EOF
6890     return -1;
6891   return 0;
6892 }
6893
6894
6895
6896
6897
6898
6899
```

```
6900 int
6901 main(void)
6902 {
6903   static char buf[100];
6904   int fd;
6905
6906   // Assumes three file descriptors open.
6907   while((fd = open("console", O_RDWR)) >= 0){
6908     if(fd >= 3){
6909       close(fd);
6910       break;
6911     }
6912   }
6913
6914   // Read and run input commands.
6915   while(getcmd(buf, sizeof(buf)) >= 0){
6916     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
6917       // Clumsy but will have to do for now.
6918       // Chdir has no effect on the parent if run in the child.
6919       buf[strlen(buf)-1] = 0;  // chop \n
6920       if(chdir(buf+3) < 0)
6921         printf(2, "cannot cd %s\n", buf+3);
6922       continue;
6923     }
6924     if(fork1() == 0)
6925       runcmd(parsecmd(buf));
6926     wait();
6927   }
6928   exit();
6929 }
6930
6931 void
6932 panic(char *s)
6933 {
6934   printf(2, "%s\n", s);
6935   exit();
6936 }
6937
6938 int
6939 fork1(void)
6940 {
6941   int pid;
6942
6943   pid = fork();
6944   if(pid == -1)
6945     panic("fork");
6946   return pid;
6947 }
6948
6949
```



```
6950 // Constructors
6951
6952 struct cmd*
6953 execcmd(void)
6954 {
6955   struct execcmd *cmd;
6956
6957   cmd = malloc(sizeof(*cmd));
6958   memset(cmd, 0, sizeof(*cmd));
6959   cmd->type = EXEC;
6960   return (struct cmd*)cmd;
6961 }
6962
6963 struct cmd*
6964 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
6965 {
6966   struct redircmd *cmd;
6967
6968   cmd = malloc(sizeof(*cmd));
6969   memset(cmd, 0, sizeof(*cmd));
6970   cmd->type = REDIR;
6971   cmd->cmd = subcmd;
6972   cmd->file = file;
6973   cmd->efile = efile;
6974   cmd->mode = mode;
6975   cmd->fd = fd;
6976   return (struct cmd*)cmd;
6977 }
6978
6979 struct cmd*
6980 pipecmd(struct cmd *left, struct cmd *right)
6981 {
6982   struct pipecmd *cmd;
6983
6984   cmd = malloc(sizeof(*cmd));
6985   memset(cmd, 0, sizeof(*cmd));
6986   cmd->type = PIPE;
6987   cmd->left = left;
6988   cmd->right = right;
6989   return (struct cmd*)cmd;
6990 }
6991
6992
6993
6994
6995
6996
6997
6998
6999
```

```
7000 struct cmd*
7001 listcmd(struct cmd *left, struct cmd *right)
7002 {
7003   struct listcmd *cmd;
7004
7005   cmd = malloc(sizeof(*cmd));
7006   memset(cmd, 0, sizeof(*cmd));
7007   cmd->type = LIST;
7008   cmd->left = left;
7009   cmd->right = right;
7010   return (struct cmd*)cmd;
7011 }
7012
7013 struct cmd*
7014 backcmd(struct cmd *subcmd)
7015 {
7016   struct backcmd *cmd;
7017
7018   cmd = malloc(sizeof(*cmd));
7019   memset(cmd, 0, sizeof(*cmd));
7020   cmd->type = BACK;
7021   cmd->cmd = subcmd;
7022   return (struct cmd*)cmd;
7023 }
7024 // Parsing
7025
7026 char whitespace[] = " \t\r\n\v";
7027 char symbols[] = "<|>&;()";
7028
7029 int
7030 gettoken(char **ps, char *es, char **q, char **eq)
7031 {
7032   char *s;
7033   int ret;
7034
7035   s = *ps;
7036   while(s < es && strchr(whitespace, *s))
7037     s++;
7038   if(q)
7039     *q = s;
7040   ret = *s;
7041   switch(*s){
7042   case 0:
7043     break;
7044   case '|':
7045   case '(':
7046   case ')':
7047   case ';':
7048   case '&':
7049   case '<':
```



```
7050     s++;
7051     break;
7052   case '>':
7053     s++;
7054     if(*s == '>'){
7055       ret = '+';
7056       s++;
7057     }
7058     break;
7059   default:
7060     ret = 'a';
7061     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
7062       s++;
7063     break;
7064   }
7065   if(eq)
7066     *eq = s;
7067
7068   while(s < es && strchr(whitespace, *s))
7069     s++;
7070   *ps = s;
7071   return ret;
7072 }
7073
7074 int
7075 peek(char **ps, char *es, char *toks)
7076 {
7077   char *s;
7078
7079   s = *ps;
7080   while(s < es && strchr(whitespace, *s))
7081     s++;
7082   *ps = s;
7083   return *s && strchr(toks, *s);
7084 }
7085
7086
7087
7088
7089
7090
7091
7092
7093
7094
7095
7096
7097
7098
7099
```

```
7100 struct cmd *parseline(char**, char*);
7101 struct cmd *parsepipe(char**, char*);
7102 struct cmd *parseexec(char**, char*);
7103 struct cmd *nulterminate(struct cmd*);
7104
7105 struct cmd*
7106 parsecmd(char *s)
7107 {
7108   char *es;
7109   struct cmd *cmd;
7110
7111   es = s + strlen(s);
7112   cmd = parseline(&s, es);
7113   peek(&s, es, "");
7114   if(s != es){
7115     printf(2, "leftovers: %s\n", s);
7116     panic("syntax");
7117   }
7118   nulterminate(cmd);
7119   return cmd;
7120 }
7121
7122 struct cmd*
7123 parseline(char **ps, char *es)
7124 {
7125   struct cmd *cmd;
7126
7127   cmd = parsepipe(ps, es);
7128   while(peek(ps, es, "&")){
7129     gettoken(ps, es, 0, 0);
7130     cmd = backcmd(cmd);
7131   }
7132   if(peek(ps, es, ";")){
7133     gettoken(ps, es, 0, 0);
7134     cmd = listcmd(cmd, parseline(ps, es));
7135   }
7136   return cmd;
7137 }
7138
7139
7140
7141
7142
7143
7144
7145
7146
7147
7148
7149
```



```
7150 struct cmd*
7151 parsepipe(char **ps, char *es)
7152 {
7153   struct cmd *cmd;
7154
7155   cmd = parseexec(ps, es);
7156   if(peek(ps, es, "|")){
7157     gettoken(ps, es, 0, 0);
7158     cmd = pipecmd(cmd, parsepipe(ps, es));
7159   }
7160   return cmd;
7161 }
7162
7163 struct cmd*
7164 parseredirs(struct cmd *cmd, char **ps, char *es)
7165 {
7166   int tok;
7167   char *q, *eq;
7168
7169   while(peek(ps, es, "<>")){
7170     tok = gettoken(ps, es, 0, 0);
7171     if(gettoken(ps, es, &q, &eq) != 'a')
7172       panic("missing file for redirection");
7173     switch(tok){
7174     case '<':
7175       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
7176       break;
7177     case '>':
7178       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
7179       break;
7180     case '+':  // >>
7181       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
7182       break;
7183     }
7184   }
7185   return cmd;
7186 }
7187
7188
7189
7190
7191
7192
7193
7194
7195
7196
7197
7198
7199
```

```
7200 struct cmd*
7201 parseblock(char **ps, char *es)
7202 {
7203   struct cmd *cmd;
7204
7205   if(!peek(ps, es, "("))
7206     panic("parseblock");
7207   gettoken(ps, es, 0, 0);
7208   cmd = parseline(ps, es);
7209   if(!peek(ps, es, ")"))
7210     panic("syntax - missing )");
7211   gettoken(ps, es, 0, 0);
7212   cmd = parseredirs(cmd, ps, es);
7213   return cmd;
7214 }
7215
7216 struct cmd*
7217 parseexec(char **ps, char *es)
7218 {
7219   char *q, *eq;
7220   int tok, argc;
7221   struct execcmd *cmd;
7222   struct cmd *ret;
7223
7224   if(peek(ps, es, "("))
7225     return parseblock(ps, es);
7226
7227   ret = execcmd();
7228   cmd = (struct execcmd*)ret;
7229
7230   argc = 0;
7231   ret = parseredirs(ret, ps, es);
7232   while(!peek(ps, es, "|)&;")){
7233     if((tok=gettoken(ps, es, &q, &eq)) == 0)
7234       break;
7235     if(tok != 'a')
7236       panic("syntax");
7237     cmd->argv[argc] = q;
7238     cmd->eargv[argc] = eq;
7239     argc++;
7240     if(argc >= MAXARGS)
7241       panic("too many args");
7242     ret = parseredirs(ret, ps, es);
7243   }
7244   cmd->argv[argc] = 0;
7245   cmd->eargv[argc] = 0;
7246   return ret;
7247 }
7248
7249
```



```
7250 // NUL-terminate all the counted strings.
7251 struct cmd*
7252 nulterminate(struct cmd *cmd)
7253 {
7254   int i;
7255   struct backcmd *bcmd;
7256   struct execcmd *ecmd;
7257   struct listcmd *lcmd;
7258   struct pipecmd *pcmd;
7259   struct redircmd *rcmd;
7260
7261   if(cmd == 0)
7262     return 0;
7263
7264   switch(cmd->type){
7265   case EXEC:
7266     ecmd = (struct execcmd*)cmd;
7267     for(i=0; ecmd->argv[i]; i++)
7268       *ecmd->eargv[i] = 0;
7269     break;
7270
7271   case REDIR:
7272     rcmd = (struct redircmd*)cmd;
7273     nulterminate(rcmd->cmd);
7274     *rcmd->efile = 0;
7275     break;
7276
7277   case PIPE:
7278     pcmd = (struct pipecmd*)cmd;
7279     nulterminate(pcmd->left);
7280     nulterminate(pcmd->right);
7281     break;
7282
7283   case LIST:
7284     lcmd = (struct listcmd*)cmd;
7285     nulterminate(lcmd->left);
7286     nulterminate(lcmd->right);
7287     break;
7288
7289   case BACK:
7290     bcmd = (struct backcmd*)cmd;
7291     nulterminate(bcmd->cmd);
7292     break;
7293   }
7294   return cmd;
7295 }
7296
7297
7298
7299
```