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.

```
    4774 4809 4813 4825
    4867 4937 5020 6444
    6513 6533
inb 0353
    0353 0928 0936 1154
    3336 3363 5646 6214
    6217 6282 6307 6309
INDIRECT 3168
    3168 4027 4030 4065
    4066 4073
initlock 1363
    0314 1363 1620 2231
    2524 3355 3542 3791
    4414 5184 6553 6554
inode 3252
    0203 0234 0235 0236
    0237 0239 0240 0241
    0242 0243 0245 0246
    0247 0248 0249 1539
    2951 2952 3106 3252
    3675 3785 3802 3805
    3811 3837 3838 3852
    3884 3902 3924 3951
    3977 4010 4052 4082
    4102 4152 4211 4212
    4252 4256 4353 4356
    4388 4395 4666 4702
    4753 4800 4804 4856
    4903 4921 4933 5015
    6435 6501
INPUT_BUF 6450
    6450 6453 6474 6486
    6487 6489 6518
insl 0362
    0362 1173 3414
INT_DISABLED 5869
    5869 5917
IOAPIC 5858
    5858 5908
ioapic_enable 5923
    0256 3357 5923 6561
ioapic_id 5516
    0257 5516 5628 5911
    5912
ioapic_init 5901
    0258 1226 5901 5912
ioapic_read 5884
    5884 5909 5910
ioapic_write 5891
    5891 5917 5918 5931
    5932
IO_PIC1 5957
    5957 5970 5985 5994
    5997 6002 6012 6026
    6027
IO_PIC2 5958
    5958 5971 5986 6015
    6016 6017 6020 6029
    6030
IO_RTC 5800
    5800 5813 5814
IO_TIMER1 6609
    6609 6618 6628 6629
IPB 3187
    3187 3190 3196 3868
    3962 3983
iput 3902
    0240 2020 3902 3908
    3927 4260 4382 4471
    4687 4943
IRQ_ERROR 2384
    2384 5727
IRQ_IDE 2383
    2383 2556 3356 3357
IRQ_KBD 2382
    2382 2560 6560 6561
IRQ_OFFSET 2379
    2379 2547 2556 2560
    2564 2591 5707 5714
    5727 5917 5931 5997
    6016
IRQ_SLAVE 5960
    5960 5964 6002 6017
IRQ_SPURIOUS 2385
    2385 2564 5707
IRQ_TIMER 2381
    2381 2547 2591 5714
    6630
isdirempty 4702
    4702 4709 4778
ismp 5514
    0276 1233 5514 5613
    5905 5925
itrunc 4052
    3675 3911 4052
iunlock 3884
    0241 3884 3887 3926
    4371 4483 4514 4534
    4679 4880 4942 6439
    6506
```



```
iunlockput 3924
    0242 3924 4366 4375
    4378 4674 4686 4692
    4696 4766 4771 4779
    4780 4787 4791 4812
    4815 4822 4833 4834
    4845 4869 4877 4913
    4925 4939 5069 5112
iupdate 3977
    0243 3913 3977 4077
    4178 4678 4695 4790
    4829 4840
I_BUSY 3266
    3266 3861 3863 3886
    3890 3907 3909 3915
I_VALID 3267
    3267 3866 3876 3905
kalloc 2304
    0261 1283 1657 1719
    1730 1764 2231 2304
    2310 2328 5052 5178
kalloc_lock 2212
    2212 2231 2265 2293
    2312 2316 2322 2326
KBDATAP 6054
    6054 6217
kbd_getc 6206
    6206 6248
kbd_intr 6246
    0266 2561 6246
KBSTATP 6052
    6052 6214
KBS_DIB 6053
    6053 6215
KEY_DEL 6078
    6078 6119 6141 6165
KEY_DN 6072
    6072 6115 6137 6161
KEY_END 6070
    6070 6118 6140 6164
KEY_HOME 6069
    6069 6118 6140 6164
KEY_INS 6077
    6077 6119 6141 6165
KEY_LF 6073
    6073 6117 6139 6163
KEY_PGDN 6076
    6076 6116 6138 6162
KEY_PGUP 6075
    6075 6116 6138 6162
KEY_RT 6074
    6074 6117 6139 6163
KEY_UP 6071
    6071 6115 6137 6161
kfree 2255
    0262 1662 1731 2069
    2070 2236 2255 2260
    5101 5111 5202 5228
kill 1976
    0296 1976 2578 2839
    6817
kinit 2225
    0263 1227 2225
KSTACKSIZE 0152
    0152 1283 1284 1680
    1719 1723 1731 2070
lapicw 5690
    5690 5707 5713 5714
    5715 5718 5719 5724
    5727 5730 5731 5734
    5737 5738 5743 5775
    5821 5822 5824 5833
    5834
lapic_eoi 5772
    0271 2554 2558 2562
    2567 5772
lapic_init 5701
    0272 1220 1258 5701
lapic_startap 5805
    0273 1286 5805
lgdt 0403
    0403 0411 0954 1054
    1700
lidt 0417
    0417 0425 2530
LINT0 5679
    5679 5718
LINT1 5680
    5680 5719
LIST 6860
    6860 6940 7107 7383
listcmd 6890 7101
    6890 6911 6941 7101
    7103 7246 7357 7384
LPTPORT 6265
    6265 6282 6286 6287
    6288
lpt_putc 6278
    6278 6341
ltr 0429
```

```
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
0689
0690
0691
0692
0693
0694
0695
0696
0697
0698
0699
```

```
0700 // Task state segment format
0701 struct taskstate {
0702   uint link;         // Old ts selector
0703   uint esp0;         // Stack pointers and segment selectors
0704   ushort ss0;        //   after an increase in privilege level
0705   ushort padding1;
0706   uint *esp1;
0707   ushort ss1;
0708   ushort padding2;
0709   uint *esp2;
0710   ushort ss2;
0711   ushort padding3;
0712   void *cr3;         // Page directory base
0713   uint *eip;         // Saved state from last task switch
0714   uint eflags;
0715   uint eax;          // More saved state (registers)
0716   uint ecx;
0717   uint edx;
0718   uint ebx;
0719   uint *esp;
0720   uint *ebp;
0721   uint esi;
0722   uint edi;
0723   ushort es;         // Even more saved state (segment selectors)
0724   ushort padding4;
0725   ushort cs;
0726   ushort padding5;
0727   ushort ss;
0728   ushort padding6;
0729   ushort ds;
0730   ushort padding7;
0731   ushort fs;
0732   ushort padding8;
0733   ushort gs;
0734   ushort padding9;
0735   ushort ldt;
0736   ushort padding10;
0737   ushort t;          // Trap on task switch
0738   ushort iomb;       // I/O map base address
0739 };
0740
0741
0742
0743
0744
0745
0746
0747
0748
0749
```



```
0750 // Gate descriptors for interrupts and traps
0751 struct gatedesc {
0752   uint off_15_0 : 16;   // low 16 bits of offset in segment
0753   uint cs : 16;         // code segment selector
0754   uint args : 5;        // # args, 0 for interrupt/trap gates
0755   uint rsv1 : 3;        // reserved(should be zero I guess)
0756   uint type : 4;        // type(STS_{TG,IG32,TG32})
0757   uint s : 1;           // must be 0 (system)
0758   uint dpl : 2;         // descriptor(meaning new) privilege level
0759   uint p : 1;           // Present
0760   uint off_31_16 : 16;  // high bits of offset in segment
0761 };
0762
0763 // Set up a normal interrupt/trap gate descriptor.
0764 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
0765 //   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
0766 // - sel: Code segment selector for interrupt/trap handler
0767 // - off: Offset in code segment for interrupt/trap handler
0768 // - dpl: Descriptor Privilege Level -
0769 //        the privilege level required for software to invoke
0770 //        this interrupt/trap gate explicitly using an int instruction.
0771 #define SETGATE(gate, istrap, sel, off, d)                \
0772 {                                                         \
0773   (gate).off_15_0 = (uint) (off) & 0xffff;                \
0774   (gate).cs = (sel);                                      \
0775   (gate).args = 0;                                        \
0776   (gate).rsv1 = 0;                                        \
0777   (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
0778   (gate).s = 0;                                           \
0779   (gate).dpl = (d);                                       \
0780   (gate).p = 1;                                           \
0781   (gate).off_31_16 = (uint) (off) >> 16;                  \
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
0944
0945
0946
0947
0948
0949
```



```
0950   # Switch from real to protected mode, using a bootstrap GDT
0951   # and segment translation that makes virtual addresses
0952   # identical to physical addresses, so that the
0953   # effective memory map does not change during the switch.
0954   lgdt    gdtdesc
0955   movl    %cr0, %eax
0956   orl     $CR0_PE_ON, %eax
0957   movl    %eax, %cr0
0958
0959   # Jump to next instruction, but in 32-bit code segment.
0960   # Switches processor into 32-bit mode.
0961   ljmp    $PROT_MODE_CSEG, $protcseg
0962
0963   .code32                     # Assemble for 32-bit mode
0964 protcseg:
0965   # Set up the protected-mode data segment registers
0966   movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
0967   movw    %ax, %ds                # -> DS: Data Segment
0968   movw    %ax, %es                # -> ES: Extra Segment
0969   movw    %ax, %fs                # -> FS
0970   movw    %ax, %gs                # -> GS
0971   movw    %ax, %ss                # -> SS: Stack Segment
0972
0973   # Set up the stack pointer and call into C.
0974   movl    $start, %esp
0975   call    bootmain
0976
0977   # If bootmain returns (it shouldn't), loop.
0978 spin:
0979   jmp     spin
0980
0981 # Bootstrap GDT
0982 .p2align 2                                 # force 4 byte alignment
0983 gdt:
0984   SEG_NULLASM                              # null seg
0985   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)    # code seg
0986   SEG_ASM(STA_W, 0x0, 0xffffffff)          # data seg
0987
0988 gdtdesc:
0989   .word   0x17                             # sizeof(gdt) - 1
0990   .long   gdt                              # address gdt
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
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
```



```
1050   # Switch from real to protected mode, using a bootstrap GDT
1051   # and segment translation that makes virtual addresses
1052   # identical to their physical addresses, so that the
1053   # effective memory map does not change during the switch.
1054   lgdt    gdtdesc
1055   movl    %cr0, %eax
1056   orl     $CR0_PE_ON, %eax
1057   movl    %eax, %cr0
1058
1059   # Jump to next instruction, but in 32-bit code segment.
1060   # Switches processor into 32-bit mode.
1061   ljmp    $PROT_MODE_CSEG, $protcseg
1062
1063   .code32                     # Assemble for 32-bit mode
1064 protcseg:
1065   # Set up the protected-mode data segment registers
1066   movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
1067   movw    %ax, %ds                # -> DS: Data Segment
1068   movw    %ax, %es                # -> ES: Extra Segment
1069   movw    %ax, %fs                # -> FS
1070   movw    %ax, %gs                # -> GS
1071   movw    %ax, %ss                # -> SS: Stack Segment
1072
1073   movl    start-4, %esp
1074   movl    start-8, %eax
1075   jmp     *%eax
1076
1077 # Bootstrap GDT
1078 .p2align 2                                # force 4 byte alignment
1079 gdt:
1080   SEG_NULLASM                             # null seg
1081   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
1082   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
1083
1084 gdtdesc:
1085   .word   0x17                            # sizeof(gdt) - 1
1086   .long   gdt                             # address gdt
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       swtch(&c->context, &p->context);
1833
1834       // Process is done running for now.
1835       // It should have changed its p->state before coming back.
1836       c->curproc = 0;
1837       setupsegs(0);
1838     }
1839     release(&proc_table_lock);
1840
1841   }
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   swtch(&cp->context, &cpus[cpu()].context);
1865 }
1866
1867 // Give up the CPU for one scheduling round.
1868 void
1869 yield(void)
1870 {
1871   acquire(&proc_table_lock);
1872   cp->state = RUNNABLE;
1873   sched();
1874   release(&proc_table_lock);
1875 }
1876
1877 // A fork child's very first scheduling by scheduler()
1878 // will swtch here.  "Return" to user space.
1879 void
1880 forkret(void)
1881 {
1882   // Still holding proc_table_lock from scheduler.
1883   release(&proc_table_lock);
1884
1885   // Jump into assembly, never to return.
1886   forkret1(cp->tf);
1887 }
1888
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
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
```



```
1950 // Wake up all processes sleeping on chan.
1951 // Proc_table_lock must be held.
1952 static void
1953 wakeup1(void *chan)
1954 {
1955   struct proc *p;
1956
1957   for(p = proc; p < &proc[NPROC]; p++)
1958     if(p->state == SLEEPING && p->chan == chan)
1959       p->state = RUNNABLE;
1960 }
1961
1962 // Wake up all processes sleeping on chan.
1963 // Proc_table_lock is acquired and released.
1964 void
1965 wakeup(void *chan)
1966 {
1967   acquire(&proc_table_lock);
1968   wakeup1(chan);
1969   release(&proc_table_lock);
1970 }
1971
1972 // Kill the process with the given pid.
1973 // Process won't actually exit until it returns
1974 // to user space (see trap in trap.c).
1975 int
1976 kill(int pid)
1977 {
1978   struct proc *p;
1979
1980   acquire(&proc_table_lock);
1981   for(p = proc; p < &proc[NPROC]; p++){
1982     if(p->pid == pid){
1983       p->killed = 1;
1984       // Wake process from sleep if necessary.
1985       if(p->state == SLEEPING)
1986         p->state = RUNNABLE;
1987       release(&proc_table_lock);
1988       return 0;
1989     }
1990   }
1991   release(&proc_table_lock);
1992   return -1;
1993 }
1994
1995
1996
1997
1998
1999
```

```
2000 // Exit the current process.  Does not return.
2001 // Exited processes remain in the zombie state
2002 // until their parent calls wait() to find out they exited.
2003 void
2004 exit(void)
2005 {
2006   struct proc *p;
2007   int fd;
2008
2009   if(cp == initproc)
2010     panic("init exiting");
2011
2012   // Close all open files.
2013   for(fd = 0; fd < NOFILE; fd++){
2014     if(cp->ofile[fd]){
2015       fileclose(cp->ofile[fd]);
2016       cp->ofile[fd] = 0;
2017     }
2018   }
2019
2020   iput(cp->cwd);
2021   cp->cwd = 0;
2022
2023   acquire(&proc_table_lock);
2024
2025   // Parent might be sleeping in wait().
2026   wakeup1(cp->parent);
2027
2028   // Pass abandoned children to init.
2029   for(p = proc; p < &proc[NPROC]; p++){
2030     if(p->parent == cp){
2031       p->parent = initproc;
2032       if(p->state == ZOMBIE)
2033         wakeup1(initproc);
2034     }
2035   }
2036
2037   // Jump into the scheduler, never to return.
2038   cp->killed = 0;
2039   cp->state = ZOMBIE;
2040   sched();
2041   panic("zombie exit");
2042 }
2043
2044
2045
2046
2047
2048
2049
```



```
2050 // Wait for a child process to exit and return its pid.
2051 // Return -1 if this process has no children.
2052 int
2053 wait(void)
2054 {
2055   struct proc *p;
2056   int i, havekids, pid;
2057
2058   acquire(&proc_table_lock);
2059   for(;;){
2060     // Scan through table looking for zombie children.
2061     havekids = 0;
2062     for(i = 0; i < NPROC; i++){
2063       p = &proc[i];
2064       if(p->state == UNUSED)
2065         continue;
2066       if(p->parent == cp){
2067         if(p->state == ZOMBIE){
2068           // Found one.
2069           kfree(p->mem, p->sz);
2070           kfree(p->kstack, KSTACKSIZE);
2071           pid = p->pid;
2072           p->state = UNUSED;
2073           p->pid = 0;
2074           p->parent = 0;
2075           p->name[0] = 0;
2076           release(&proc_table_lock);
2077           return pid;
2078         }
2079         havekids = 1;
2080       }
2081     }
2082
2083     // No point waiting if we don't have any children.
2084     if(!havekids || cp->killed){
2085       release(&proc_table_lock);
2086       return -1;
2087     }
2088
2089     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2090     sleep(cp, &proc_table_lock);
2091   }
2092 }
2093
2094
2095
2096
2097
2098
2099
```

```
2100 // Print a process listing to console.  For debugging.
2101 // Runs when user types ^P on console.
2102 // No lock to avoid wedging a stuck machine further.
2103 void
2104 procdump(void)
2105 {
2106   static char *states[] = {
2107   [UNUSED]    "unused",
2108   [EMBRYO]    "embryo",
2109   [SLEEPING]  "sleep ",
2110   [RUNNABLE]  "runble",
2111   [RUNNING]   "run   ",
2112   [ZOMBIE]    "zombie"
2113   };
2114   int i, j;
2115   struct proc *p;
2116   char *state;
2117   uint pc[10];
2118
2119   for(i = 0; i < NPROC; i++){
2120     p = &proc[i];
2121     if(p->state == UNUSED)
2122       continue;
2123     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
2124       state = states[p->state];
2125     else
2126       state = "???";
2127     cprintf("%d %s %s", p->pid, state, p->name);
2128     if(p->state == SLEEPING){
2129       getcallerpcs((uint*)p->context.ebp+2, pc);
2130       for(j=0; j<10 && pc[j] != 0; j++)
2131         cprintf(" %p", pc[j]);
2132     }
2133     cprintf("\n");
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
3199
```

```
3200 // Directory is a file containing a sequence of dirent structures.
3201 #define DIRSIZ 14
3202
3203 struct dirent {
3204   ushort inum;
3205   char name[DIRSIZ];
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
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
```



```
3450 // Sync buf with disk.
3451 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
3452 // Else if B_VALID is not set, read buf from disk, set B_VALID.
3453 void
3454 ide_rw(struct buf *b)
3455 {
3456   struct buf **pp;
3457
3458   if(!(b->flags & B_BUSY))
3459     panic("ide_rw: buf not busy");
3460   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
3461     panic("ide_rw: nothing to do");
3462   if(b->dev != 0 && !disk_1_present)
3463     panic("ide disk 1 not present");
3464
3465   acquire(&ide_lock);
3466
3467   // Append b to ide_queue.
3468   b->qnext = 0;
3469   for(pp=&ide_queue; *pp; pp=&(*pp)->qnext)
3470     ;
3471   *pp = b;
3472
3473   // Start disk if necessary.
3474   if(ide_queue == b)
3475     ide_start_request(b);
3476
3477   // Wait for request to finish.
3478   // Assuming will not sleep too long: ignore cp->killed.
3479   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
3480     sleep(b, &ide_lock);
3481
3482   release(&ide_lock);
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
3545
3546
3547
3548
3549
```



```
3550   // Create linked list of buffers
3551   bufhead.prev = &bufhead;
3552   bufhead.next = &bufhead;
3553   for(b = buf; b < buf+NBUF; b++){
3554     b->next = bufhead.next;
3555     b->prev = &bufhead;
3556     bufhead.next->prev = b;
3557     bufhead.next = b;
3558   }
3559 }
3560
3561 // Look through buffer cache for sector on device dev.
3562 // If not found, allocate fresh block.
3563 // In either case, return locked buffer.
3564 static struct buf*
3565 bget(uint dev, uint sector)
3566 {
3567   struct buf *b;
3568
3569   acquire(&buf_table_lock);
3570
3571  loop:
3572   // Try for cached block.
3573   for(b = bufhead.next; b != &bufhead; b = b->next){
3574     if((b->flags & (B_BUSY|B_VALID)) &&
3575        b->dev == dev && b->sector == sector){
3576       if(b->flags & B_BUSY){
3577         sleep(buf, &buf_table_lock);
3578         goto loop;
3579       }
3580       b->flags |= B_BUSY;
3581       release(&buf_table_lock);
3582       return b;
3583     }
3584   }
3585
3586   // Allocate fresh block.
3587   for(b = bufhead.prev; b != &bufhead; b = b->prev){
3588     if((b->flags & B_BUSY) == 0){
3589       b->flags = B_BUSY;
3590       b->dev = dev;
3591       b->sector = sector;
3592       release(&buf_table_lock);
3593       return b;
3594     }
3595   }
3596   panic("bget: no buffers");
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
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
```



```
3950 // Allocate a new inode with the given type on device dev.
3951 struct inode*
3952 ialloc(uint dev, short type)
3953 {
3954   int inum;
3955   struct buf *bp;
3956   struct dinode *dip;
3957   struct superblock sb;
3958
3959   readsb(dev, &sb);
3960   for(inum = 1; inum < sb.ninodes; inum++){  // loop over inode blocks
3961     bp = bread(dev, IBLOCK(inum));
3962     dip = (struct dinode*)bp->data + inum%IPB;
3963     if(dip->type == 0){  // a free inode
3964       memset(dip, 0, sizeof(*dip));
3965       dip->type = type;
3966       bwrite(bp);   // mark it allocated on the disk
3967       brelse(bp);
3968       return iget(dev, inum);
3969     }
3970     brelse(bp);
3971   }
3972   panic("ialloc: no inodes");
3973 }
3974
3975 // Copy inode, which has changed, from memory to disk.
3976 void
3977 iupdate(struct inode *ip)
3978 {
3979   struct buf *bp;
3980   struct dinode *dip;
3981
3982   bp = bread(ip->dev, IBLOCK(ip->inum));
3983   dip = (struct dinode*)bp->data + ip->inum%IPB;
3984   dip->type = ip->type;
3985   dip->major = ip->major;
3986   dip->minor = ip->minor;
3987   dip->nlink = ip->nlink;
3988   dip->size = ip->size;
3989   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
3990   bwrite(bp);
3991   brelse(bp);
3992 }
3993
3994
3995
3996
3997
3998
3999
```

```
4000 // Inode contents
4001 //
4002 // The contents (data) associated with each inode is stored
4003 // in a sequence of blocks on the disk.  The first NDIRECT blocks
4004 // are listed in ip->addrs[].  The next NINDIRECT blocks are
4005 // listed in the block ip->addrs[INDIRECT].
4006
4007 // Return the disk block address of the nth block in inode ip.
4008 // If there is no such block, alloc controls whether one is allocated.
4009 static uint
4010 bmap(struct inode *ip, uint bn, int alloc)
4011 {
4012   uint addr, *a;
4013   struct buf *bp;
4014
4015   if(bn < NDIRECT){
4016     if((addr = ip->addrs[bn]) == 0){
4017       if(!alloc)
4018         return -1;
4019       ip->addrs[bn] = addr = balloc(ip->dev);
4020     }
4021     return addr;
4022   }
4023   bn -= NDIRECT;
4024
4025   if(bn < NINDIRECT){
4026     // Load indirect block, allocating if necessary.
4027     if((addr = ip->addrs[INDIRECT]) == 0){
4028       if(!alloc)
4029         return -1;
4030       ip->addrs[INDIRECT] = addr = balloc(ip->dev);
4031     }
4032     bp = bread(ip->dev, addr);
4033     a = (uint*)bp->data;
4034
4035     if((addr = a[bn]) == 0){
4036       if(!alloc){
4037         brelse(bp);
4038         return -1;
4039       }
4040       a[bn] = addr = balloc(ip->dev);
4041       bwrite(bp);
4042     }
4043     brelse(bp);
4044     return addr;
4045   }
4046
4047   panic("bmap: out of range");
4048 }
4049
```



```
4050 // Truncate inode (discard contents).
4051 static void
4052 itrunc(struct inode *ip)
4053 {
4054   int i, j;
4055   struct buf *bp;
4056   uint *a;
4057
4058   for(i = 0; i < NDIRECT; i++){
4059     if(ip->addrs[i]){
4060       bfree(ip->dev, ip->addrs[i]);
4061       ip->addrs[i] = 0;
4062     }
4063   }
4064
4065   if(ip->addrs[INDIRECT]){
4066     bp = bread(ip->dev, ip->addrs[INDIRECT]);
4067     a = (uint*)bp->data;
4068     for(j = 0; j < NINDIRECT; j++){
4069       if(a[j])
4070         bfree(ip->dev, a[j]);
4071     }
4072     brelse(bp);
4073     ip->addrs[INDIRECT] = 0;
4074   }
4075
4076   ip->size = 0;
4077   iupdate(ip);
4078 }
4079
4080 // Copy stat information from inode.
4081 void
4082 stati(struct inode *ip, struct stat *st)
4083 {
4084   st->dev = ip->dev;
4085   st->ino = ip->inum;
4086   st->type = ip->type;
4087   st->nlink = ip->nlink;
4088   st->size = ip->size;
4089 }
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
```

```
4100 // Read data from inode.
4101 int
4102 readi(struct inode *ip, char *dst, 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].read)
4109       return -1;
4110     return devsw[ip->major].read(ip, dst, n);
4111   }
4112
4113   if(off > ip->size || off + n < off)
4114     return -1;
4115   if(off + n > ip->size)
4116     n = ip->size - off;
4117
4118   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
4119     bp = bread(ip->dev, bmap(ip, off/BSIZE, 0));
4120     m = min(n - tot, BSIZE - off%BSIZE);
4121     memmove(dst, bp->data + off%BSIZE, m);
4122     brelse(bp);
4123   }
4124   return n;
4125 }
4126
4127
4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```



```
4150 // Write data to inode.
4151 int
4152 writei(struct inode *ip, char *src, uint off, uint n)
4153 {
4154   uint tot, m;
4155   struct buf *bp;
4156
4157   if(ip->type == T_DEV){
4158     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
4159       return -1;
4160     return devsw[ip->major].write(ip, src, n);
4161   }
4162
4163   if(off + n < off)
4164     return -1;
4165   if(off + n > MAXFILE*BSIZE)
4166     n = MAXFILE*BSIZE - off;
4167
4168   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
4169     bp = bread(ip->dev, bmap(ip, off/BSIZE, 1));
4170     m = min(n - tot, BSIZE - off%BSIZE);
4171     memmove(bp->data + off%BSIZE, src, m);
4172     bwrite(bp);
4173     brelse(bp);
4174   }
4175
4176   if(n > 0 && off > ip->size){
4177     ip->size = off;
4178     iupdate(ip);
4179   }
4180   return n;
4181 }
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
```

```
4200 // Directories
4201
4202 int
4203 namecmp(const char *s, const char *t)
4204 {
4205   return strncmp(s, t, DIRSIZ);
4206 }
4207
4208 // Look for a directory entry in a directory.
4209 // If found, set *poff to byte offset of entry.
4210 // Caller must have already locked dp.
4211 struct inode*
4212 dirlookup(struct inode *dp, char *name, uint *poff)
4213 {
4214   uint off, inum;
4215   struct buf *bp;
4216   struct dirent *de;
4217
4218   if(dp->type != T_DIR)
4219     panic("dirlookup not DIR");
4220
4221   for(off = 0; off < dp->size; off += BSIZE){
4222     bp = bread(dp->dev, bmap(dp, off / BSIZE, 0));
4223     for(de = (struct dirent*)bp->data;
4224         de < (struct dirent*)(bp->data + BSIZE);
4225         de++){
4226       if(de->inum == 0)
4227         continue;
4228       if(namecmp(name, de->name) == 0){
4229         // entry matches path element
4230         if(poff)
4231           *poff = off + (uchar*)de - bp->data;
4232         inum = de->inum;
4233         brelse(bp);
4234         return iget(dp->dev, inum);
4235       }
4236     }
4237     brelse(bp);
4238   }
4239   return 0;
4240 }
4241
4242
4243
4244
4245
4246
4247
4248
4249
```



```
4250 // Write a new directory entry (name, ino) into the directory dp.
4251 int
4252 dirlink(struct inode *dp, char *name, uint ino)
4253 {
4254   int off;
4255   struct dirent de;
4256   struct inode *ip;
4257
4258   // Check that name is not present.
4259   if((ip = dirlookup(dp, name, 0)) != 0){
4260     iput(ip);
4261     return -1;
4262   }
4263
4264   // Look for an empty dirent.
4265   for(off = 0; off < dp->size; off += sizeof(de)){
4266     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4267       panic("dirlink read");
4268     if(de.inum == 0)
4269       break;
4270   }
4271
4272   strncpy(de.name, name, DIRSIZ);
4273   de.inum = ino;
4274   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4275     panic("dirlink");
4276
4277   return 0;
4278 }
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
```

```
4300 // Paths
4301
4302 // Copy the next path element from path into name.
4303 // Return a pointer to the element following the copied one.
4304 // The returned path has no leading slashes,
4305 // so the caller can check *path=='\0' to see if the name is the last one.
4306 // If no name to remove, return 0.
4307 //
4308 // Examples:
4309 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
4310 //   skipelem("///a//bb", name) = "bb", setting name = "a"
4311 //   skipelem("", name) = skipelem("////", name) = 0
4312 //
4313 static char*
4314 skipelem(char *path, char *name)
4315 {
4316   char *s;
4317   int len;
4318
4319   while(*path == '/')
4320     path++;
4321   if(*path == 0)
4322     return 0;
4323   s = path;
4324   while(*path != '/' && *path != 0)
4325     path++;
4326   len = path - s;
4327   if(len >= DIRSIZ)
4328     memmove(name, s, DIRSIZ);
4329   else {
4330     memmove(name, s, len);
4331     name[len] = 0;
4332   }
4333   while(*path == '/')
4334     path++;
4335   return path;
4336 }
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
```



```
4350 // Look up and return the inode for a path name.
4351 // If parent != 0, return the inode for the parent and copy the final
4352 // path element into name, which must have room for DIRSIZ bytes.
4353 static struct inode*
4354 _namei(char *path, int parent, char *name)
4355 {
4356   struct inode *ip, *next;
4357
4358   if(*path == '/')
4359     ip = iget(ROOTDEV, 1);
4360   else
4361     ip = idup(cp->cwd);
4362
4363   while((path = skipelem(path, name)) != 0){
4364     ilock(ip);
4365     if(ip->type != T_DIR){
4366       iunlockput(ip);
4367       return 0;
4368     }
4369     if(parent && *path == '\0'){
4370       // Stop one level early.
4371       iunlock(ip);
4372       return ip;
4373     }
4374     if((next = dirlookup(ip, name, 0)) == 0){
4375       iunlockput(ip);
4376       return 0;
4377     }
4378     iunlockput(ip);
4379     ip = next;
4380   }
4381   if(parent){
4382     iput(ip);
4383     return 0;
4384   }
4385   return ip;
4386 }
4387
4388 struct inode*
4389 namei(char *path)
4390 {
4391   char name[DIRSIZ];
4392   return _namei(path, 0, name);
4393 }
4394
4395 struct inode*
4396 nameiparent(char *path, char *name)
4397 {
4398   return _namei(path, 1, name);
4399 }
```

```
4400 #include "types.h"
4401 #include "defs.h"
4402 #include "param.h"
4403 #include "file.h"
4404 #include "spinlock.h"
4405 #include "dev.h"
4406
4407 struct devsw devsw[NDEV];
4408 struct spinlock file_table_lock;
4409 struct file file[NFILE];
4410
4411 void
4412 fileinit(void)
4413 {
4414   initlock(&file_table_lock, "file_table");
4415 }
4416
4417 // Allocate a file structure.
4418 struct file*
4419 filealloc(void)
4420 {
4421   int i;
4422
4423   acquire(&file_table_lock);
4424   for(i = 0; i < NFILE; i++){
4425     if(file[i].type == FD_CLOSED){
4426       file[i].type = FD_NONE;
4427       file[i].ref = 1;
4428       release(&file_table_lock);
4429       return file + i;
4430     }
4431   }
4432   release(&file_table_lock);
4433   return 0;
4434 }
4435
4436 // Increment ref count for file f.
4437 struct file*
4438 filedup(struct file *f)
4439 {
4440   acquire(&file_table_lock);
4441   if(f->ref < 1 || f->type == FD_CLOSED)
4442     panic("filedup");
4443   f->ref++;
4444   release(&file_table_lock);
4445   return f;
4446 }
4447
4448
4449
```



```
4450 // Close file f.  (Decrement ref count, close when reaches 0.)
4451 void
4452 fileclose(struct file *f)
4453 {
4454   struct file ff;
4455
4456   acquire(&file_table_lock);
4457   if(f->ref < 1 || f->type == FD_CLOSED)
4458     panic("fileclose");
4459   if(--f->ref > 0){
4460     release(&file_table_lock);
4461     return;
4462   }
4463   ff = *f;
4464   f->ref = 0;
4465   f->type = FD_CLOSED;
4466   release(&file_table_lock);
4467
4468   if(ff.type == FD_PIPE)
4469     pipeclose(ff.pipe, ff.writable);
4470   else if(ff.type == FD_INODE)
4471     iput(ff.ip);
4472   else
4473     panic("fileclose");
4474 }
4475
4476 // Get metadata about file f.
4477 int
4478 filestat(struct file *f, struct stat *st)
4479 {
4480   if(f->type == FD_INODE){
4481     ilock(f->ip);
4482     stati(f->ip, st);
4483     iunlock(f->ip);
4484     return 0;
4485   }
4486   return -1;
4487 }
4488
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499
```

```
4500 // Read from file f.  Addr is kernel address.
4501 int
4502 fileread(struct file *f, char *addr, int n)
4503 {
4504   int r;
4505
4506   if(f->readable == 0)
4507     return -1;
4508   if(f->type == FD_PIPE)
4509     return piperead(f->pipe, addr, n);
4510   if(f->type == FD_INODE){
4511     ilock(f->ip);
4512     if((r = readi(f->ip, addr, f->off, n)) > 0)
4513       f->off += r;
4514     iunlock(f->ip);
4515     return r;
4516   }
4517   panic("fileread");
4518 }
4519
4520 // Write to file f.  Addr is kernel address.
4521 int
4522 filewrite(struct file *f, char *addr, int n)
4523 {
4524   int r;
4525
4526   if(f->writable == 0)
4527     return -1;
4528   if(f->type == FD_PIPE)
4529     return pipewrite(f->pipe, addr, n);
4530   if(f->type == FD_INODE){
4531     ilock(f->ip);
4532     if((r = writei(f->ip, addr, f->off, n)) > 0)
4533       f->off += r;
4534     iunlock(f->ip);
4535     return r;
4536   }
4537   panic("filewrite");
4538 }
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4549
```



```
4550 #include "types.h"
4551 #include "defs.h"
4552 #include "param.h"
4553 #include "stat.h"
4554 #include "mmu.h"
4555 #include "proc.h"
4556 #include "fs.h"
4557 #include "fsvar.h"
4558 #include "file.h"
4559 #include "fcntl.h"
4560
4561 // Fetch the nth word-sized system call argument as a file descriptor
4562 // and return both the descriptor and the corresponding struct file.
4563 static int
4564 argfd(int n, int *pfd, struct file **pf)
4565 {
4566   int fd;
4567   struct file *f;
4568
4569   if(argint(n, &fd) < 0)
4570     return -1;
4571   if(fd < 0 || fd >= NOFILE || (f=cp->ofile[fd]) == 0)
4572     return -1;
4573   if(pfd)
4574     *pfd = fd;
4575   if(pf)
4576     *pf = f;
4577   return 0;
4578 }
4579
4580 // Allocate a file descriptor for the given file.
4581 // Takes over file reference from caller on success.
4582 static int
4583 fdalloc(struct file *f)
4584 {
4585   int fd;
4586
4587   for(fd = 0; fd < NOFILE; fd++){
4588     if(cp->ofile[fd] == 0){
4589       cp->ofile[fd] = f;
4590       return fd;
4591     }
4592   }
4593   return -1;
4594 }
4595
4596
4597
4598
4599
```

```
4600 int
4601 sys_read(void)
4602 {
4603   struct file *f;
4604   int n;
4605   char *p;
4606
4607   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
4608     return -1;
4609   return fileread(f, p, n);
4610 }
4611
4612 int
4613 sys_write(void)
4614 {
4615   struct file *f;
4616   int n;
4617   char *p;
4618
4619   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
4620     return -1;
4621   return filewrite(f, p, n);
4622 }
4623
4624 int
4625 sys_dup(void)
4626 {
4627   struct file *f;
4628   int fd;
4629
4630   if(argfd(0, 0, &f) < 0)
4631     return -1;
4632   if((fd=fdalloc(f)) < 0)
4633     return -1;
4634   filedup(f);
4635   return fd;
4636 }
4637
4638 int
4639 sys_close(void)
4640 {
4641   int fd;
4642   struct file *f;
4643
4644   if(argfd(0, &fd, &f) < 0)
4645     return -1;
4646   cp->ofile[fd] = 0;
4647   fileclose(f);
4648   return 0;
4649 }
```



```
4650 int
4651 sys_fstat(void)
4652 {
4653   struct file *f;
4654   struct stat *st;
4655
4656   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
4657     return -1;
4658   return filestat(f, st);
4659 }
4660
4661 // Create the path new as a link to the same inode as old.
4662 int
4663 sys_link(void)
4664 {
4665   char name[DIRSIZ], *new, *old;
4666   struct inode *dp, *ip;
4667
4668   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
4669     return -1;
4670   if((ip = namei(old)) == 0)
4671     return -1;
4672   ilock(ip);
4673   if(ip->type == T_DIR){
4674     iunlockput(ip);
4675     return -1;
4676   }
4677   ip->nlink++;
4678   iupdate(ip);
4679   iunlock(ip);
4680
4681   if((dp = nameiparent(new, name)) == 0)
4682     goto  bad;
4683   ilock(dp);
4684   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0)
4685     goto bad;
4686   iunlockput(dp);
4687   iput(ip);
4688   return 0;
4689
4690 bad:
4691   if(dp)
4692     iunlockput(dp);
4693   ilock(ip);
4694   ip->nlink--;
4695   iupdate(ip);
4696   iunlockput(ip);
4697   return -1;
4698 }
4699
```

```
4700 // Is the directory dp empty except for "." and ".." ?
4701 static int
4702 isdirempty(struct inode *dp)
4703 {
4704   int off;
4705   struct dirent de;
4706
4707   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
4708     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4709       panic("isdirempty: readi");
4710     if(de.inum != 0)
4711       return 0;
4712   }
4713   return 1;
4714 }
4715
4716
4717
4718
4719
4720
4721
4722
4723
4724
4725
4726
4727
4728
4729
4730
4731
4732
4733
4734
4735
4736
4737
4738
4739
4740
4741
4742
4743
4744
4745
4746
4747
4748
4749
```



```
4750 int
4751 sys_unlink(void)
4752 {
4753   struct inode *ip, *dp;
4754   struct dirent de;
4755   char name[DIRSIZ], *path;
4756   uint off;
4757
4758   if(argstr(0, &path) < 0)
4759     return -1;
4760   if((dp = nameiparent(path, name)) == 0)
4761     return -1;
4762   ilock(dp);
4763
4764   // Cannot unlink "." or "..".
4765   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0){
4766     iunlockput(dp);
4767     return -1;
4768   }
4769
4770   if((ip = dirlookup(dp, name, &off)) == 0){
4771     iunlockput(dp);
4772     return -1;
4773   }
4774   ilock(ip);
4775
4776   if(ip->nlink < 1)
4777     panic("unlink: nlink < 1");
4778   if(ip->type == T_DIR && !isdirempty(ip)){
4779     iunlockput(ip);
4780     iunlockput(dp);
4781     return -1;
4782   }
4783
4784   memset(&de, 0, sizeof(de));
4785   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4786     panic("unlink: writei");
4787   iunlockput(dp);
4788
4789   ip->nlink--;
4790   iupdate(ip);
4791   iunlockput(ip);
4792   return 0;
4793 }
4794
4795
4796
4797
4798
4799
```

```
4800 static struct inode*
4801 create(char *path, int canexist, short type, short major, short minor)
4802 {
4803   uint off;
4804   struct inode *ip, *dp;
4805   char name[DIRSIZ];
4806
4807   if((dp = nameiparent(path, name)) == 0)
4808     return 0;
4809   ilock(dp);
4810
4811   if(canexist && (ip = dirlookup(dp, name, &off)) != 0){
4812     iunlockput(dp);
4813     ilock(ip);
4814     if(ip->type != type || ip->major != major || ip->minor != minor){
4815       iunlockput(ip);
4816       return 0;
4817     }
4818     return ip;
4819   }
4820
4821   if((ip = ialloc(dp->dev, type)) == 0){
4822     iunlockput(dp);
4823     return 0;
4824   }
4825   ilock(ip);
4826   ip->major = major;
4827   ip->minor = minor;
4828   ip->nlink = 1;
4829   iupdate(ip);
4830
4831   if(dirlink(dp, name, ip->inum) < 0){
4832     ip->nlink = 0;
4833     iunlockput(ip);
4834     iunlockput(dp);
4835     return 0;
4836   }
4837
4838   if(type == T_DIR){  // Create . and .. entries.
4839     dp->nlink++;  // for ".."
4840     iupdate(dp);
4841     // No ip->nlink++ for ".": avoid cyclic ref count.
4842     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
4843       panic("create dots");
4844   }
4845   iunlockput(dp);
4846   return ip;
4847 }
4848
4849
```



```
4850 int
4851 sys_open(void)
4852 {
4853   char *path;
4854   int fd, omode;
4855   struct file *f;
4856   struct inode *ip;
4857
4858   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
4859     return -1;
4860
4861   if(omode & O_CREATE){
4862     if((ip = create(path, 1, T_FILE, 0, 0)) == 0)
4863       return -1;
4864   } else {
4865     if((ip = namei(path)) == 0)
4866       return -1;
4867     ilock(ip);
4868     if(ip->type == T_DIR && (omode & (O_RDWR|O_WRONLY))){
4869       iunlockput(ip);
4870       return -1;
4871     }
4872   }
4873
4874   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
4875     if(f)
4876       fileclose(f);
4877     iunlockput(ip);
4878     return -1;
4879   }
4880   iunlock(ip);
4881
4882   f->type = FD_INODE;
4883   f->ip = ip;
4884   f->off = 0;
4885   f->readable = !(omode & O_WRONLY);
4886   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
4887
4888   return fd;
4889 }
4890
4891
4892
4893
4894
4895
4896
4897
4898
4899
```

```
4900 int
4901 sys_mknod(void)
4902 {
4903   struct inode *ip;
4904   char *path;
4905   int len;
4906   int major, minor;
4907
4908   if((len=argstr(0, &path)) < 0 ||
4909      argint(1, &major) < 0 ||
4910      argint(2, &minor) < 0 ||
4911      (ip = create(path, 0, T_DEV, major, minor)) == 0)
4912     return -1;
4913   iunlockput(ip);
4914   return 0;
4915 }
4916
4917 int
4918 sys_mkdir(void)
4919 {
4920   char *path;
4921   struct inode *ip;
4922
4923   if(argstr(0, &path) < 0 || (ip = create(path, 0, T_DIR, 0, 0)) == 0)
4924     return -1;
4925   iunlockput(ip);
4926   return 0;
4927 }
4928
4929 int
4930 sys_chdir(void)
4931 {
4932   char *path;
4933   struct inode *ip;
4934
4935   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0)
4936     return -1;
4937   ilock(ip);
4938   if(ip->type != T_DIR){
4939     iunlockput(ip);
4940     return -1;
4941   }
4942   iunlock(ip);
4943   iput(cp->cwd);
4944   cp->cwd = ip;
4945   return 0;
4946 }
4947
4948
4949
```



```
4950 int
4951 sys_exec(void)
4952 {
4953   char *path, *argv[20];
4954   int i;
4955   uint uargv, uarg;
4956
4957   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0)
4958     return -1;
4959   memset(argv, 0, sizeof(argv));
4960   for(i=0;; i++){
4961     if(i >= NELEM(argv))
4962       return -1;
4963     if(fetchint(cp, uargv+4*i, (int*)&uarg) < 0)
4964       return -1;
4965     if(uarg == 0){
4966       argv[i] = 0;
4967       break;
4968     }
4969     if(fetchstr(cp, uarg, &argv[i]) < 0)
4970       return -1;
4971   }
4972   return exec(path, argv);
4973 }
4974
4975 int
4976 sys_pipe(void)
4977 {
4978   int *fd;
4979   struct file *rf, *wf;
4980   int fd0, fd1;
4981
4982   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
4983     return -1;
4984   if(pipealloc(&rf, &wf) < 0)
4985     return -1;
4986   fd0 = -1;
4987   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
4988     if(fd0 >= 0)
4989       cp->ofile[fd0] = 0;
4990     fileclose(rf);
4991     fileclose(wf);
4992     return -1;
4993   }
4994   fd[0] = fd0;
4995   fd[1] = fd1;
4996   return 0;
4997 }
4998
4999
```

```
5000 #include "types.h"
5001 #include "param.h"
5002 #include "mmu.h"
5003 #include "proc.h"
5004 #include "defs.h"
5005 #include "x86.h"
5006 #include "elf.h"
5007
5008 int
5009 exec(char *path, char **argv)
5010 {
5011   char *mem, *s, *last;
5012   int i, argc, arglen, len, off;
5013   uint sz, sp, argp;
5014   struct elfhdr elf;
5015   struct inode *ip;
5016   struct proghdr ph;
5017
5018   if((ip = namei(path)) == 0)
5019     return -1;
5020   ilock(ip);
5021
5022   // Compute memory size of new process.
5023   mem = 0;
5024   sz = 0;
5025
5026   // Program segments.
5027   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
5028     goto bad;
5029   if(elf.magic != ELF_MAGIC)
5030     goto bad;
5031   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
5032     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
5033       goto bad;
5034     if(ph.type != ELF_PROG_LOAD)
5035       continue;
5036     if(ph.memsz < ph.filesz)
5037       goto bad;
5038     sz += ph.memsz;
5039   }
5040
5041   // Arguments.
5042   arglen = 0;
5043   for(argc=0; argv[argc]; argc++)
5044     arglen += strlen(argv[argc]) + 1;
5045   arglen = (arglen+3) & ~3;
5046   sz += arglen + 4*(argc+1);
5047
5048   // Stack.
5049   sz += PAGE;
```



```
5050   // Allocate program memory.
5051   sz = (sz+PAGE-1) & ~(PAGE-1);
5052   mem = kalloc(sz);
5053   if(mem == 0)
5054     goto bad;
5055   memset(mem, 0, sz);
5056
5057   // Load program into memory.
5058   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
5059     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
5060       goto bad;
5061     if(ph.type != ELF_PROG_LOAD)
5062       continue;
5063     if(ph.va + ph.memsz > sz)
5064       goto bad;
5065     if(readi(ip, mem + ph.va, ph.offset, ph.filesz) != ph.filesz)
5066       goto bad;
5067     memset(mem + ph.va + ph.filesz, 0, ph.memsz - ph.filesz);
5068   }
5069   iunlockput(ip);
5070
5071   // Initialize stack.
5072   sp = sz;
5073   argp = sz - arglen - 4*(argc+1);
5074
5075   // Copy argv strings and pointers to stack.
5076   *(uint*)(mem+argp + 4*argc) = 0;  // argv[argc]
5077   for(i=argc-1; i>=0; i--){
5078     len = strlen(argv[i]) + 1;
5079     sp -= len;
5080     memmove(mem+sp, argv[i], len);
5081     *(uint*)(mem+argp + 4*i) = sp;  // argv[i]
5082   }
5083
5084   // Stack frame for main(argc, argv), below arguments.
5085   sp = argp;
5086   sp -= 4;
5087   *(uint*)(mem+sp) = argp;
5088   sp -= 4;
5089   *(uint*)(mem+sp) = argc;
5090   sp -= 4;
5091   *(uint*)(mem+sp) = 0xffffffff;   // fake return pc
5092
5093   // Save program name for debugging.
5094   for(last=s=path; *s; s++)
5095     if(*s == '/')
5096       last = s+1;
5097   safestrcpy(cp->name, last, sizeof(cp->name));
5098
5099
```

```
5100   // Commit to the new image.
5101   kfree(cp->mem, cp->sz);
5102   cp->mem = mem;
5103   cp->sz = sz;
5104   cp->tf->eip = elf.entry;  // main
5105   cp->tf->esp = sp;
5106   setupsegs(cp);
5107   return 0;
5108
5109  bad:
5110   if(mem)
5111     kfree(mem, sz);
5112   iunlockput(ip);
5113   return -1;
5114 }
5115
5116
5117
5118
5119
5120
5121
5122
5123
5124
5125
5126
5127
5128
5129
5130
5131
5132
5133
5134
5135
5136
5137
5138
5139
5140
5141
5142
5143
5144
5145
5146
5147
5148
5149
```



```
5150 #include "types.h"
5151 #include "defs.h"
5152 #include "param.h"
5153 #include "mmu.h"
5154 #include "proc.h"
5155 #include "file.h"
5156 #include "spinlock.h"
5157
5158 #define PIPESIZE 512
5159
5160 struct pipe {
5161   int readopen;   // read fd is still open
5162   int writeopen;  // write fd is still open
5163   uint writep;    // next index to write
5164   uint readp;     // next index to read
5165   struct spinlock lock;
5166   char data[PIPESIZE];
5167 };
5168
5169 int
5170 pipealloc(struct file **f0, struct file **f1)
5171 {
5172   struct pipe *p;
5173
5174   p = 0;
5175   *f0 = *f1 = 0;
5176   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
5177     goto bad;
5178   if((p = (struct pipe*)kalloc(PAGE)) == 0)
5179     goto bad;
5180   p->readopen = 1;
5181   p->writeopen = 1;
5182   p->writep = 0;
5183   p->readp = 0;
5184   initlock(&p->lock, "pipe");
5185   (*f0)->type = FD_PIPE;
5186   (*f0)->readable = 1;
5187   (*f0)->writable = 0;
5188   (*f0)->pipe = p;
5189   (*f1)->type = FD_PIPE;
5190   (*f1)->readable = 0;
5191   (*f1)->writable = 1;
5192   (*f1)->pipe = p;
5193   return 0;
5194
5195
5196
5197
5198
5199
```

```
 bad:
  if(p)
    kfree((char*)p, PAGE);
  if(*f0){
    (*f0)->type = FD_NONE;
    fileclose(*f0);
  }
  if(*f1){
    (*f1)->type = FD_NONE;
    fileclose(*f1);
  }
  return -1;
}

void
pipeclose(struct pipe *p, int writable)
{
  acquire(&p->lock);
  if(writable){
    p->writeopen = 0;
    wakeup(&p->readp);
  } else {
    p->readopen = 0;
    wakeup(&p->writep);
  }
  release(&p->lock);

  if(p->readopen == 0 && p->writeopen == 0)
    kfree((char*)p, PAGE);
}
```



```
int
pipewrite(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  for(i = 0; i < n; i++){
    while(p->writep == p->readp + PIPESIZE) {
      if(p->readopen == 0 || cp->killed){
        release(&p->lock);
        return -1;
      }
      wakeup(&p->readp);
      sleep(&p->writep, &p->lock);
    }
    p->data[p->writep++ % PIPESIZE] = addr[i];
  }
  wakeup(&p->readp);
  release(&p->lock);
  return i;
}

int
piperead(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  while(p->readp == p->writep && p->writeopen){
    if(cp->killed){
      release(&p->lock);
      return -1;
    }
    sleep(&p->readp, &p->lock);
  }
  for(i = 0; i < n; i++){
    if(p->readp == p->writep)
      break;
    addr[i] = p->data[p->readp++ % PIPESIZE];
  }
  wakeup(&p->writep);
  release(&p->lock);
  return i;
}
```

```
5300 #include "types.h"
5301
5302 void*
5303 memset(void *dst, int c, uint n)
5304 {
5305   char *d;
5306
5307   d = (char*)dst;
5308   while(n-- > 0)
5309     *d++ = c;
5310
5311   return dst;
5312 }
5313
5314 int
5315 memcmp(const void *v1, const void *v2, uint n)
5316 {
5317   const uchar *s1, *s2;
5318
5319   s1 = v1;
5320   s2 = v2;
5321   while(n-- > 0){
5322     if(*s1 != *s2)
5323       return *s1 - *s2;
5324     s1++, s2++;
5325   }
5326
5327   return 0;
5328 }
5329
5330 void*
5331 memmove(void *dst, const void *src, uint n)
5332 {
5333   const char *s;
5334   char *d;
5335
5336   s = src;
5337   d = dst;
5338   if(s < d && s + n > d){
5339     s += n;
5340     d += n;
5341     while(n-- > 0)
5342       *--d = *--s;
5343   } else
5344     while(n-- > 0)
5345       *d++ = *s++;
5346
5347   return dst;
5348 }
5349
```



```
5350 int
5351 strncmp(const char *p, const char *q, uint n)
5352 {
5353   while(n > 0 && *p && *p == *q)
5354     n--, p++, q++;
5355   if(n == 0)
5356     return 0;
5357   return (uchar)*p - (uchar)*q;
5358 }
5359
5360 char*
5361 strncpy(char *s, const char *t, int n)
5362 {
5363   char *os;
5364
5365   os = s;
5366   while(n-- > 0 && (*s++ = *t++) != 0)
5367     ;
5368   while(n-- > 0)
5369     *s++ = 0;
5370   return os;
5371 }
5372
5373 // Like strncpy but guaranteed to NUL-terminate.
5374 char*
5375 safestrcpy(char *s, const char *t, int n)
5376 {
5377   char *os;
5378
5379   os = s;
5380   if(n <= 0)
5381     return os;
5382   while(--n > 0 && (*s++ = *t++) != 0)
5383     ;
5384   *s = 0;
5385   return os;
5386 }
5387
5388 int
5389 strlen(const char *s)
5390 {
5391   int n;
5392
5393   for(n = 0; s[n]; n++)
5394     ;
5395   return n;
5396 }
5397
5398
5399
```

```
5400 // See MultiProcessor Specification Version 1.[14]
5401
5402 struct mp {             // floating pointer
5403   uchar signature[4];           // "_MP_"
5404   void *physaddr;               // phys addr of MP config table
5405   uchar length;                 // 1
5406   uchar specrev;                // [14]
5407   uchar checksum;               // all bytes must add up to 0
5408   uchar type;                   // MP system config type
5409   uchar imcrp;
5410   uchar reserved[3];
5411 };
5412
5413 struct mpconf {         // configuration table header
5414   uchar signature[4];           // "PCMP"
5415   ushort length;                // total table length
5416   uchar version;                // [14]
5417   uchar checksum;               // all bytes must add up to 0
5418   uchar product[20];            // product id
5419   uint *oemtable;               // OEM table pointer
5420   ushort oemlength;             // OEM table length
5421   ushort entry;                 // entry count
5422   uint *lapicaddr;              // address of local APIC
5423   ushort xlength;               // extended table length
5424   uchar xchecksum;              // extended table checksum
5425   uchar reserved;
5426 };
5427
5428 struct mpproc {         // processor table entry
5429   uchar type;                   // entry type (0)
5430   uchar apicid;                 // local APIC id
5431   uchar version;                // local APIC verison
5432   uchar flags;                  // CPU flags
5433     #define MPBOOT 0x02           // This proc is the bootstrap processor.
5434   uchar signature[4];           // CPU signature
5435   uint feature;                 // feature flags from CPUID instruction
5436   uchar reserved[8];
5437 };
5438
5439 struct mpioapic {       // I/O APIC table entry
5440   uchar type;                   // entry type (2)
5441   uchar apicno;                 // I/O APIC id
5442   uchar version;                // I/O APIC version
5443   uchar flags;                  // I/O APIC flags
5444   uint *addr;                  // I/O APIC address
5445 };
5446
5447
5448
5449
```



```
5450 // Table entry types
5451 #define MPPROC    0x00  // One per processor
5452 #define MPBUS     0x01  // One per bus
5453 #define MPIOAPIC  0x02  // One per I/O APIC
5454 #define MPIOINTR  0x03  // One per bus interrupt source
5455 #define MPLINTR   0x04  // One per system interrupt source
5456
5457
5458
5459
5460
5461
5462
5463
5464
5465
5466
5467
5468
5469
5470
5471
5472
5473
5474
5475
5476
5477
5478
5479
5480
5481
5482
5483
5484
5485
5486
5487
5488
5489
5490
5491
5492
5493
5494
5495
5496
5497
5498
5499
```

```
5500 // Multiprocessor bootstrap.
5501 // Search memory for MP description structures.
5502 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
5503
5504 #include "types.h"
5505 #include "defs.h"
5506 #include "param.h"
5507 #include "mp.h"
5508 #include "x86.h"
5509 #include "mmu.h"
5510 #include "proc.h"
5511
5512 struct cpu cpus[NCPU];
5513 static struct cpu *bcpu;
5514 int ismp;
5515 int ncpu;
5516 uchar ioapic_id;
5517
5518 int
5519 mp_bcpu(void)
5520 {
5521   return bcpu-cpus;
5522 }
5523
5524 static uchar
5525 sum(uchar *addr, int len)
5526 {
5527   int i, sum;
5528
5529   sum = 0;
5530   for(i=0; i<len; i++)
5531     sum += addr[i];
5532   return sum;
5533 }
5534
5535 // Look for an MP structure in the len bytes at addr.
5536 static struct mp*
5537 mp_search1(uchar *addr, int len)
5538 {
5539   uchar *e, *p;
5540
5541   e = addr+len;
5542   for(p = addr; p < e; p += sizeof(struct mp))
5543     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
5544       return (struct mp*)p;
5545   return 0;
5546 }
5547
5548
5549
```



```
5550 // Search for the MP Floating Pointer Structure, which according to the
5551 // spec is in one of the following three locations:
5552 // 1) in the first KB of the EBDA;
5553 // 2) in the last KB of system base memory;
5554 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
5555 static struct mp*
5556 mp_search(void)
5557 {
5558   uchar *bda;
5559   uint p;
5560   struct mp *mp;
5561
5562   bda = (uchar*)0x400;
5563   if((p = ((bda[0x0F]<<8)|bda[0x0E]) << 4)){
5564     if((mp = mp_search1((uchar*)p, 1024)))
5565       return mp;
5566   } else {
5567     p = ((bda[0x14]<<8)|bda[0x13])*1024;
5568     if((mp = mp_search1((uchar*)p-1024, 1024)))
5569       return mp;
5570   }
5571   return mp_search1((uchar*)0xF0000, 0x10000);
5572 }
5573
5574 // Search for an MP configuration table.  For now,
5575 // don't accept the default configurations (physaddr == 0).
5576 // Check for correct signature, calculate the checksum and,
5577 // if correct, check the version.
5578 // To do: check extended table checksum.
5579 static struct mpconf*
5580 mp_config(struct mp **pmp)
5581 {
5582   struct mpconf *conf;
5583   struct mp *mp;
5584
5585   if((mp = mp_search()) == 0 || mp->physaddr == 0)
5586     return 0;
5587   conf = (struct mpconf*)mp->physaddr;
5588   if(memcmp(conf, "PCMP", 4) != 0)
5589     return 0;
5590   if(conf->version != 1 && conf->version != 4)
5591     return 0;
5592   if(sum((uchar*)conf, conf->length) != 0)
5593     return 0;
5594   *pmp = mp;
5595   return conf;
5596 }
5597
5598
5599
```

```
5600 void
5601 mp_init(void)
5602 {
5603   uchar *p, *e;
5604   struct mp *mp;
5605   struct mpconf *conf;
5606   struct mpproc *proc;
5607   struct mpioapic *ioapic;
5608
5609   bcpu = &cpus[ncpu];
5610   if((conf = mp_config(&mp)) == 0)
5611     return;
5612
5613   ismp = 1;
5614   lapic = (uint*)conf->lapicaddr;
5615
5616   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
5617     switch(*p){
5618     case MPPROC:
5619       proc = (struct mpproc*)p;
5620       cpus[ncpu].apicid = proc->apicid;
5621       if(proc->flags & MPBOOT)
5622         bcpu = &cpus[ncpu];
5623       ncpu++;
5624       p += sizeof(struct mpproc);
5625       continue;
5626     case MPIOAPIC:
5627       ioapic = (struct mpioapic*)p;
5628       ioapic_id = ioapic->apicno;
5629       p += sizeof(struct mpioapic);
5630       continue;
5631     case MPBUS:
5632     case MPIOINTR:
5633     case MPLINTR:
5634       p += 8;
5635       continue;
5636     default:
5637       cprintf("mp_init: unknown config type %x\n", *p);
5638       panic("mp_init");
5639     }
5640   }
5641
5642   if(mp->imcrp){
5643     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
5644     // But it would on real hardware.
5645     outb(0x22, 0x70);   // Select IMCR
5646     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
5647   }
5648 }
5649
```



```
5650 // The local APIC manages internal (non-I/O) interrupts.
5651 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
5652
5653 #include "types.h"
5654 #include "defs.h"
5655 #include "traps.h"
5656 #include "mmu.h"
5657 #include "x86.h"
5658
5659 // Local APIC registers, divided by 4 for use as uint[] indices.
5660 #define ID      (0x0020/4)   // ID
5661 #define VER     (0x0030/4)   // Version
5662 #define TPR     (0x0080/4)   // Task Priority
5663 #define EOI     (0x00B0/4)   // EOI
5664 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
5665   #define ENABLE     0x00000100   // Unit Enable
5666 #define ESR     (0x0280/4)   // Error Status
5667 #define ICRLO   (0x0300/4)   // Interrupt Command
5668   #define INIT       0x00000500   // INIT/RESET
5669   #define STARTUP    0x00000600   // Startup IPI
5670   #define DELIVS     0x00001000   // Delivery status
5671   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
5672   #define LEVEL      0x00008000   // Level triggered
5673   #define BCAST      0x00080000   // Send to all APICs, including self.
5674 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
5675 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
5676   #define X1         0x0000000B   // divide counts by 1
5677   #define PERIODIC   0x00020000   // Periodic
5678 #define PCINT   (0x0340/4)   // Performance Counter LVT
5679 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
5680 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
5681 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
5682   #define MASKED     0x00010000   // Interrupt masked
5683 #define TICR    (0x0380/4)   // Timer Initial Count
5684 #define TCCR    (0x0390/4)   // Timer Current Count
5685 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
5686
5687 volatile uint *lapic;  // Initialized in mp.c
5688
5689 static void
5690 lapicw(int index, int value)
5691 {
5692   lapic[index] = value;
5693   lapic[ID];  // wait for write to finish, by reading
5694 }
5695
5696
5697
5698
5699
```

```
5700 void
5701 lapic_init(int c)
5702 {
5703   if(!lapic)
5704     return;
5705
5706   // Enable local APIC; set spurious interrupt vector.
5707   lapicw(SVR, ENABLE | (IRQ_OFFSET+IRQ_SPURIOUS));
5708
5709   // The timer repeatedly counts down at bus frequency
5710   // from lapic[TICR] and then issues an interrupt.
5711   // If xv6 cared more about precise timekeeping,
5712   // TICR would be calibrated using an external time source.
5713   lapicw(TDCR, X1);
5714   lapicw(TIMER, PERIODIC | (IRQ_OFFSET + IRQ_TIMER));
5715   lapicw(TICR, 10000000);
5716
5717   // Disable logical interrupt lines.
5718   lapicw(LINT0, MASKED);
5719   lapicw(LINT1, MASKED);
5720
5721   // Disable performance counter overflow interrupts
5722   // on machines that provide that interrupt entry.
5723   if(((lapic[VER]>>16) & 0xFF) >= 4)
5724     lapicw(PCINT, MASKED);
5725
5726   // Map error interrupt to IRQ_ERROR.
5727   lapicw(ERROR, IRQ_OFFSET+IRQ_ERROR);
5728
5729   // Clear error status register (requires back-to-back writes).
5730   lapicw(ESR, 0);
5731   lapicw(ESR, 0);
5732
5733   // Ack any outstanding interrupts.
5734   lapicw(EOI, 0);
5735
5736   // Send an Init Level De-Assert to synchronise arbitration ID's.
5737   lapicw(ICRHI, 0);
5738   lapicw(ICRLO, BCAST | INIT | LEVEL);
5739   while(lapic[ICRLO] & DELIVS)
5740     ;
5741
5742   // Enable interrupts on the APIC (but not on the processor).
5743   lapicw(TPR, 0);
5744 }
5745
5746
5747
5748
5749
```



```
5750 int
5751 cpu(void)
5752 {
5753   // Cannot call cpu when interrupts are enabled:
5754   // result not guaranteed to last long enough to be used!
5755   // Would prefer to panic but even printing is chancy here:
5756   // everything, including cprintf, calls cpu, at least indirectly
5757   // through acquire and release.
5758   if(read_eflags()&FL_IF){
5759     static int n;
5760     if(n++ == 0)
5761       cprintf("cpu called from %x with interrupts enabled\n",
5762         ((uint*)read_ebp())[1]);
5763   }
5764
5765   if(lapic)
5766     return lapic[ID]>>24;
5767   return 0;
5768 }
5769
5770 // Acknowledge interrupt.
5771 void
5772 lapic_eoi(void)
5773 {
5774   if(lapic)
5775     lapicw(EOI, 0);
5776 }
5777
5778 // Spin for a given number of microseconds.
5779 // On real hardware would want to tune this dynamically.
5780 static void
5781 microdelay(int us)
5782 {
5783   volatile int j = 0;
5784
5785   while(us-- > 0)
5786     for(j=0; j<10000; j++);
5787 }
5788
5789
5790
5791
5792
5793
5794
5795
5796
5797
5798
5799
```

```
5800 #define IO_RTC  0x70
5801
5802 // Start additional processor running bootstrap code at addr.
5803 // See Appendix B of MultiProcessor Specification.
5804 void
5805 lapic_startap(uchar apicid, uint addr)
5806 {
5807   int i;
5808   ushort *wrv;
5809
5810   // "The BSP must initialize CMOS shutdown code to 0AH
5811   // and the warm reset vector (DWORD based at 40:67) to point at
5812   // the AP startup code prior to the [universal startup algorithm]."
5813   outb(IO_RTC, 0xF);  // offset 0xF is shutdown code
5814   outb(IO_RTC+1, 0x0A);
5815   wrv = (ushort*)(0x40<<4 | 0x67);  // Warm reset vector
5816   wrv[0] = 0;
5817   wrv[1] = addr >> 4;
5818
5819   // "Universal startup algorithm."
5820   // Send INIT (level-triggered) interrupt to reset other CPU.
5821   lapicw(ICRHI, apicid<<24);
5822   lapicw(ICRLO, INIT | LEVEL | ASSERT);
5823   microdelay(200);
5824   lapicw(ICRLO, INIT | LEVEL);
5825   microdelay(100); // should be 10ms, but too slow in Bochs!
5826
5827   // Send startup IPI (twice!) to enter bootstrap code.
5828   // Regular hardware is supposed to only accept a STARTUP
5829   // when it is in the halted state due to an INIT.  So the second
5830   // should be ignored, but it is part of the official Intel algorithm.
5831   // Bochs complains about the second one.  Too bad for Bochs.
5832   for(i = 0; i < 2; i++){
5833     lapicw(ICRHI, apicid<<24);
5834     lapicw(ICRLO, STARTUP | (addr>>12));
5835     microdelay(200);
5836   }
5837 }
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```



```
5850 // The I/O APIC manages hardware interrupts for an SMP system.
5851 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
5852 // See also picirq.c.
5853
5854 #include "types.h"
5855 #include "defs.h"
5856 #include "traps.h"
5857
5858 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
5859
5860 #define REG_ID     0x00  // Register index: ID
5861 #define REG_VER    0x01  // Register index: version
5862 #define REG_TABLE  0x10  // Redirection table base
5863
5864 // The redirection table starts at REG_TABLE and uses
5865 // two registers to configure each interrupt.
5866 // The first (low) register in a pair contains configuration bits.
5867 // The second (high) register contains a bitmask telling which
5868 // CPUs can serve that interrupt.
5869 #define INT_DISABLED   0x00010000  // Interrupt disabled
5870 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
5871 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
5872 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
5873
5874 volatile struct ioapic *ioapic;
5875
5876 // IO APIC MMIO structure: write reg, then read or write data.
5877 struct ioapic {
5878   uint reg;
5879   uint pad[3];
5880   uint data;
5881 };
5882
5883 static uint
5884 ioapic_read(int reg)
5885 {
5886   ioapic->reg = reg;
5887   return ioapic->data;
5888 }
5889
5890 static void
5891 ioapic_write(int reg, uint data)
5892 {
5893   ioapic->reg = reg;
5894   ioapic->data = data;
5895 }
5896
5897
5898
5899
```

```
5900 void
5901 ioapic_init(void)
5902 {
5903   int i, id, maxintr;
5904
5905   if(!ismp)
5906     return;
5907
5908   ioapic = (volatile struct ioapic*)IOAPIC;
5909   maxintr = (ioapic_read(REG_VER) >> 16) & 0xFF;
5910   id = ioapic_read(REG_ID) >> 24;
5911   if(id != ioapic_id)
5912     cprintf("ioapic_init: id isn't equal to ioapic_id; not a MP\n");
5913
5914   // Mark all interrupts edge-triggered, active high, disabled,
5915   // and not routed to any CPUs.
5916   for(i = 0; i <= maxintr; i++){
5917     ioapic_write(REG_TABLE+2*i, INT_DISABLED | (IRQ_OFFSET + i));
5918     ioapic_write(REG_TABLE+2*i+1, 0);
5919   }
5920 }
5921
5922 void
5923 ioapic_enable(int irq, int cpunum)
5924 {
5925   if(!ismp)
5926     return;
5927
5928   // Mark interrupt edge-triggered, active high,
5929   // enabled, and routed to the given cpunum,
5930   // which happens to be that cpu's APIC ID.
5931   ioapic_write(REG_TABLE+2*irq, IRQ_OFFSET + irq);
5932   ioapic_write(REG_TABLE+2*irq+1, cpunum << 24);
5933 }
5934
5935
5936
5937
5938
5939
5940
5941
5942
5943
5944
5945
5946
5947
5948
5949
```



```
5950 // Intel 8259A programmable interrupt controllers.
5951
5952 #include "types.h"
5953 #include "x86.h"
5954 #include "traps.h"
5955
5956 // I/O Addresses of the two programmable interrupt controllers
5957 #define IO_PIC1         0x20    // Master (IRQs 0-7)
5958 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
5959
5960 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
5961
5962 // Current IRQ mask.
5963 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
5964 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
5965
5966 static void
5967 pic_setmask(ushort mask)
5968 {
5969   irqmask = mask;
5970   outb(IO_PIC1+1, mask);
5971   outb(IO_PIC2+1, mask >> 8);
5972 }
5973
5974 void
5975 pic_enable(int irq)
5976 {
5977   pic_setmask(irqmask & ~(1<<irq));
5978 }
5979
5980 // Initialize the 8259A interrupt controllers.
5981 void
5982 pic_init(void)
5983 {
5984   // mask all interrupts
5985   outb(IO_PIC1+1, 0xFF);
5986   outb(IO_PIC2+1, 0xFF);
5987
5988   // Set up master (8259A-1)
5989
5990   // ICW1:  0001g0hi
5991   //    g:  0 = edge triggering, 1 = level triggering
5992   //    h:  0 = cascaded PICs, 1 = master only
5993   //    i:  0 = no ICW4, 1 = ICW4 required
5994   outb(IO_PIC1, 0x11);
5995
5996   // ICW2:  Vector offset
5997   outb(IO_PIC1+1, IRQ_OFFSET);
5998
5999
```

```
6000  // ICW3:  (master PIC) bit mask of IR lines connected to slaves
6001  //        (slave PIC) 3-bit # of slave's connection to master
6002  outb(IO_PIC1+1, 1<<IRQ_SLAVE);
6003
6004  // ICW4:  000nbmap
6005  //    n:  1 = special fully nested mode
6006  //    b:  1 = buffered mode
6007  //    m:  0 = slave PIC, 1 = master PIC
6008  //      (ignored when b is 0, as the master/slave role
6009  //      can be hardwired).
6010  //    a:  1 = Automatic EOI mode
6011  //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
6012  outb(IO_PIC1+1, 0x3);
6013
6014  // Set up slave (8259A-2)
6015  outb(IO_PIC2, 0x11);                  // ICW1
6016  outb(IO_PIC2+1, IRQ_OFFSET + 8);      // ICW2
6017  outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
6018  // NB Automatic EOI mode doesn't tend to work on the slave.
6019  // Linux source code says it's "to be investigated".
6020  outb(IO_PIC2+1, 0x3);                 // ICW4
6021
6022  // OCW3:  0ef01prs
6023  //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
6024  //    p:  0 = no polling, 1 = polling mode
6025  //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
6026  outb(IO_PIC1, 0x68);             // clear specific mask
6027  outb(IO_PIC1, 0x0a);             // read IRR by default
6028
6029  outb(IO_PIC2, 0x68);             // OCW3
6030  outb(IO_PIC2, 0x0a);             // OCW3
6031
6032  if(irqmask != 0xFFFF)
6033    pic_setmask(irqmask);
6034 }
6035
6036
6037
6038
6039
6040
6041
6042
6043
6044
6045
6046
6047
6048
6049
```



```
6050 // PC keyboard interface constants
6051
6052 #define KBSTATP         0x64    // kbd controller status port(I)
6053 #define KBS_DIB         0x01    // kbd data in buffer
6054 #define KBDATAP         0x60    // kbd data port(I)
6055
6056 #define NO              0
6057
6058 #define SHIFT           (1<<0)
6059 #define CTL             (1<<1)
6060 #define ALT             (1<<2)
6061
6062 #define CAPSLOCK        (1<<3)
6063 #define NUMLOCK         (1<<4)
6064 #define SCROLLLOCK      (1<<5)
6065
6066 #define E0ESC           (1<<6)
6067
6068 // Special keycodes
6069 #define KEY_HOME        0xE0
6070 #define KEY_END         0xE1
6071 #define KEY_UP          0xE2
6072 #define KEY_DN          0xE3
6073 #define KEY_LF          0xE4
6074 #define KEY_RT          0xE5
6075 #define KEY_PGUP        0xE6
6076 #define KEY_PGDN        0xE7
6077 #define KEY_INS         0xE8
6078 #define KEY_DEL         0xE9
6079
6080 // C('A') == Control-A
6081 #define C(x) (x - '@')
6082
6083 static uchar shiftcode[256] =
6084 {
6085   [0x1D] CTL,
6086   [0x2A] SHIFT,
6087   [0x36] SHIFT,
6088   [0x38] ALT,
6089   [0x9D] CTL,
6090   [0xB8] ALT
6091 };
6092
6093 static uchar togglecode[256] =
6094 {
6095   [0x3A] CAPSLOCK,
6096   [0x45] NUMLOCK,
6097   [0x46] SCROLLLOCK
6098 };
6099
```

```
6100 static uchar normalmap[256] =
6101 {
6102   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
6103   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
6104   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
6105   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
6106   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
6107   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
6108   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
6109   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
6110   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
6111   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
6112   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
6113   [0x9C] '\n',      // KP_Enter
6114   [0xB5] '/',       // KP_Div
6115   [0xC8] KEY_UP,    [0xD0] KEY_DN,
6116   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
6117   [0xCB] KEY_LF,    [0xCD] KEY_RT,
6118   [0x97] KEY_HOME,  [0xCF] KEY_END,
6119   [0xD2] KEY_INS,   [0xD3] KEY_DEL
6120 };
6121
6122 static uchar shiftmap[256] =
6123 {
6124   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
6125   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
6126   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
6127   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
6128   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
6129   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
6130   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
6131   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
6132   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
6133   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
6134   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
6135   [0x9C] '\n',      // KP_Enter
6136   [0xB5] '/',       // KP_Div
6137   [0xC8] KEY_UP,    [0xD0] KEY_DN,
6138   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
6139   [0xCB] KEY_LF,    [0xCD] KEY_RT,
6140   [0x97] KEY_HOME,  [0xCF] KEY_END,
6141   [0xD2] KEY_INS,   [0xD3] KEY_DEL
6142 };
6143
6144
6145
6146
6147
6148
6149
```



```
6150 static uchar ctlmap[256] =
6151 {
6152   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
6153   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
6154   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
6155   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
6156   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
6157   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
6158   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
6159   [0x9C] '\r',      // KP_Enter
6160   [0xB5] C('/'),    // KP_Div
6161   [0xC8] KEY_UP,    [0xD0] KEY_DN,
6162   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
6163   [0xCB] KEY_LF,    [0xCD] KEY_RT,
6164   [0x97] KEY_HOME,  [0xCF] KEY_END,
6165   [0xD2] KEY_INS,   [0xD3] KEY_DEL
6166 };
6167
6168
6169
6170
6171
6172
6173
6174
6175
6176
6177
6178
6179
6180
6181
6182
6183
6184
6185
6186
6187
6188
6189
6190
6191
6192
6193
6194
6195
6196
6197
6198
6199
```

```
6200 #include "types.h"
6201 #include "x86.h"
6202 #include "defs.h"
6203 #include "kbd.h"
6204
6205 int
6206 kbd_getc(void)
6207 {
6208   static uint shift;
6209   static uchar *charcode[4] = {
6210     normalmap, shiftmap, ctlmap, ctlmap
6211   };
6212   uint st, data, c;
6213
6214   st = inb(KBSTATP);
6215   if((st & KBS_DIB) == 0)
6216     return -1;
6217   data = inb(KBDATAP);
6218
6219   if(data == 0xE0){
6220     shift |= E0ESC;
6221     return 0;
6222   } else if(data & 0x80){
6223     // Key released
6224     data = (shift & E0ESC ? data : data & 0x7F);
6225     shift &= ~(shiftcode[data] | E0ESC);
6226     return 0;
6227   } else if(shift & E0ESC){
6228     // Last character was an E0 escape; or with 0x80
6229     data |= 0x80;
6230     shift &= ~E0ESC;
6231   }
6232
6233   shift |= shiftcode[data];
6234   shift ^= togglecode[data];
6235   c = charcode[shift & (CTL | SHIFT)][data];
6236   if(shift & CAPSLOCK){
6237     if('a' <= c && c <= 'z')
6238       c += 'A' - 'a';
6239     else if('A' <= c && c <= 'Z')
6240       c += 'a' - 'A';
6241   }
6242   return c;
6243 }
6244
6245 void
6246 kbd_intr(void)
6247 {
6248   console_intr(kbd_getc);
6249 }
```



```
6250 // Console input and output.
6251 // Input is from the keyboard only.
6252 // Output is written to the screen and the printer port.
6253
6254 #include "types.h"
6255 #include "defs.h"
6256 #include "param.h"
6257 #include "traps.h"
6258 #include "spinlock.h"
6259 #include "dev.h"
6260 #include "mmu.h"
6261 #include "proc.h"
6262 #include "x86.h"
6263
6264 #define CRTPORT 0x3d4
6265 #define LPTPORT 0x378
6266 #define BACKSPACE 0x100
6267
6268 static ushort *crt = (ushort*)0xb8000;  // CGA memory
6269
6270 static struct spinlock console_lock;
6271 int panicked = 0;
6272 int use_console_lock = 0;
6273
6274 // Copy console output to parallel port, which you can tell
6275 // .bochsrc to copy to the stdout:
6276 //   parport1: enabled=1, file="/dev/stdout"
6277 static void
6278 lpt_putc(int c)
6279 {
6280   int i;
6281
6282   for(i = 0; !(inb(LPTPORT+1) & 0x80) && i < 12800; i++)
6283     ;
6284   if(c == BACKSPACE)
6285     c = '\b';
6286   outb(LPTPORT+0, c);
6287   outb(LPTPORT+2, 0x08|0x04|0x01);
6288   outb(LPTPORT+2, 0x08);
6289 }
6290
6291
6292
6293
6294
6295
6296
6297
6298
6299
```

```
6300 static void
6301 cga_putc(int c)
6302 {
6303   int pos;
6304
6305   // Cursor position: col + 80*row.
6306   outb(CRTPORT, 14);
6307   pos = inb(CRTPORT+1) << 8;
6308   outb(CRTPORT, 15);
6309   pos |= inb(CRTPORT+1);
6310
6311   if(c == '\n')
6312     pos += 80 - pos%80;
6313   else if(c == BACKSPACE){
6314     if(pos > 0)
6315       crt[--pos] = ' ' | 0x0700;
6316   } else
6317     crt[pos++] = (c&0xff) | 0x0700;  // black on white
6318
6319   if((pos/80) >= 24){  // Scroll up.
6320     memmove(crt, crt+80, sizeof(crt[0])*23*80);
6321     pos -= 80;
6322     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
6323   }
6324
6325   outb(CRTPORT, 14);
6326   outb(CRTPORT+1, pos>>8);
6327   outb(CRTPORT, 15);
6328   outb(CRTPORT+1, pos);
6329   crt[pos] = ' ' | 0x0700;
6330 }
6331
6332 void
6333 cons_putc(int c)
6334 {
6335   if(panicked){
6336     cli();
6337     for(;;)
6338       ;
6339   }
6340
6341   lpt_putc(c);
6342   cga_putc(c);
6343 }
6344
6345
6346
6347
6348
6349
```



```
6350 void
6351 printint(int xx, int base, int sgn)
6352 {
6353   static char digits[] = "0123456789ABCDEF";
6354   char buf[16];
6355   int i = 0, neg = 0;
6356   uint x;
6357
6358   if(sgn && xx < 0){
6359     neg = 1;
6360     x = 0 - xx;
6361   } else {
6362     x = xx;
6363   }
6364
6365   do{
6366     buf[i++] = digits[x % base];
6367   }while((x /= base) != 0);
6368   if(neg)
6369     buf[i++] = '-';
6370
6371   while(--i >= 0)
6372     cons_putc(buf[i]);
6373 }
6374
6375 // Print to the console. only understands %d, %x, %p, %s.
6376 void
6377 cprintf(char *fmt, ...)
6378 {
6379   int i, c, state, locking;
6380   uint *argp;
6381   char *s;
6382
6383   locking = use_console_lock;
6384   if(locking)
6385     acquire(&console_lock);
6386
6387   argp = (uint*)(void*)&fmt + 1;
6388   state = 0;
6389   for(i = 0; fmt[i]; i++){
6390     c = fmt[i] & 0xff;
6391     switch(state){
6392     case 0:
6393       if(c == '%')
6394         state = '%';
6395       else
6396         cons_putc(c);
6397       break;
6398
6399
```

```
6400     case '%':
6401       switch(c){
6402       case 'd':
6403         printint(*argp++, 10, 1);
6404         break;
6405       case 'x':
6406       case 'p':
6407         printint(*argp++, 16, 0);
6408         break;
6409       case 's':
6410         s = (char*)*argp++;
6411         if(s == 0)
6412           s = "(null)";
6413         for(; *s; s++)
6414           cons_putc(*s);
6415         break;
6416       case '%':
6417         cons_putc('%');
6418         break;
6419       default:
6420         // Print unknown % sequence to draw attention.
6421         cons_putc('%');
6422         cons_putc(c);
6423         break;
6424       }
6425       state = 0;
6426       break;
6427     }
6428   }
6429
6430   if(locking)
6431     release(&console_lock);
6432 }
6433
6434 int
6435 console_write(struct inode *ip, char *buf, int n)
6436 {
6437   int i;
6438
6439   iunlock(ip);
6440   acquire(&console_lock);
6441   for(i = 0; i < n; i++)
6442     cons_putc(buf[i] & 0xff);
6443   release(&console_lock);
6444   ilock(ip);
6445
6446   return n;
6447 }
6448
6449
```



```
6450 #define INPUT_BUF 128
6451 struct {
6452   struct spinlock lock;
6453   char buf[INPUT_BUF];
6454   uint r;  // Read index
6455   uint w;  // Write index
6456   uint e;  // Edit index
6457 } input;
6458
6459 #define C(x)  ((x)-'@')  // Control-x
6460
6461 void
6462 console_intr(int (*getc)(void))
6463 {
6464   int c;
6465
6466   acquire(&input.lock);
6467   while((c = getc()) >= 0){
6468     switch(c){
6469     case C('P'):  // Process listing.
6470       procdump();
6471       break;
6472     case C('U'):  // Kill line.
6473       while(input.e != input.w &&
6474             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
6475         input.e--;
6476         cons_putc(BACKSPACE);
6477       }
6478       break;
6479     case C('H'):  // Backspace
6480       if(input.e != input.w){
6481         input.e--;
6482         cons_putc(BACKSPACE);
6483       }
6484       break;
6485     default:
6486       if(c != 0 && input.e-input.r < INPUT_BUF){
6487         input.buf[input.e++ % INPUT_BUF] = c;
6488         cons_putc(c);
6489         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
6490           input.w = input.e;
6491           wakeup(&input.r);
6492         }
6493       }
6494       break;
6495     }
6496   }
6497   release(&input.lock);
6498 }
6499
```

```
6500 int
6501 console_read(struct inode *ip, char *dst, int n)
6502 {
6503   uint target;
6504   int c;
6505
6506   iunlock(ip);
6507   target = n;
6508   acquire(&input.lock);
6509   while(n > 0){
6510     while(input.r == input.w){
6511       if(cp->killed){
6512         release(&input.lock);
6513         ilock(ip);
6514         return -1;
6515       }
6516       sleep(&input.r, &input.lock);
6517     }
6518     c = input.buf[input.r++ % INPUT_BUF];
6519     if(c == C('D')){  // EOF
6520       if(n < target){
6521         // Save ^D for next time, to make sure
6522         // caller gets a 0-byte result.
6523         input.r--;
6524       }
6525       break;
6526     }
6527     *dst++ = c;
6528     --n;
6529     if(c == '\n')
6530       break;
6531   }
6532   release(&input.lock);
6533   ilock(ip);
6534
6535   return target - n;
6536 }
6537
6538
6539
6540
6541
6542
6543
6544
6545
6546
6547
6548
6549
```



```
6550 void
6551 console_init(void)
6552 {
6553   initlock(&console_lock, "console");
6554   initlock(&input.lock, "console input");
6555
6556   devsw[CONSOLE].write = console_write;
6557   devsw[CONSOLE].read = console_read;
6558   use_console_lock = 1;
6559
6560   pic_enable(IRQ_KBD);
6561   ioapic_enable(IRQ_KBD, 0);
6562 }
6563
6564 void
6565 panic(char *s)
6566 {
6567   int i;
6568   uint pcs[10];
6569
6570   __asm __volatile("cli");
6571   use_console_lock = 0;
6572   cprintf("cpu%d: panic: ", cpu());
6573   cprintf(s);
6574   cprintf("\n");
6575   getcallerpcs(&s, pcs);
6576   for(i=0; i<10; i++)
6577     cprintf(" %p", pcs[i]);
6578   panicked = 1; // freeze other CPU
6579   for(;;)
6580     ;
6581 }
6582
6583
6584
6585
6586
6587
6588
6589
6590
6591
6592
6593
6594
6595
6596
6597
6598
6599
```

```
6600 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
6601 // Only used on uniprocessors;
6602 // SMP machines use the local APIC timer.
6603
6604 #include "types.h"
6605 #include "defs.h"
6606 #include "traps.h"
6607 #include "x86.h"
6608
6609 #define IO_TIMER1       0x040           // 8253 Timer #1
6610
6611 // Frequency of all three count-down timers;
6612 // (TIMER_FREQ/freq) is the appropriate count
6613 // to generate a frequency of freq Hz.
6614
6615 #define TIMER_FREQ      1193182
6616 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
6617
6618 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
6619 #define TIMER_SEL0      0x00    // select counter 0
6620 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
6621 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
6622
6623 void
6624 timer_init(void)
6625 {
6626   // Interrupt 100 times/sec.
6627   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
6628   outb(IO_TIMER1, TIMER_DIV(100) % 256);
6629   outb(IO_TIMER1, TIMER_DIV(100) / 256);
6630   pic_enable(IRQ_TIMER);
6631 }
6632
6633
6634
6635
6636
6637
6638
6639
6640
6641
6642
6643
6644
6645
6646
6647
6648
6649
```



```
6650 // Blank page
6651
6652
6653
6654
6655
6656
6657
6658
6659
6660
6661
6662
6663
6664
6665
6666
6667
6668
6669
6670
6671
6672
6673
6674
6675
6676
6677
6678
6679
6680
6681
6682
6683
6684
6685
6686
6687
6688
6689
6690
6691
6692
6693
6694
6695
6696
6697
6698
6699
```

```
6700 # Initial process execs /init.
6701
6702 #include "syscall.h"
6703 #include "traps.h"
6704
6705 # exec(init, argv)
6706 .globl start
6707 start:
6708   pushl $argv
6709   pushl $init
6710   pushl $0
6711   movl $SYS_exec, %eax
6712   int $T_SYSCALL
6713
6714 # for(;;) exit();
6715 exit:
6716   movl $SYS_exit, %eax
6717   int $T_SYSCALL
6718   jmp exit
6719
6720 # char init[] = "/init\0";
6721 init:
6722   .string "/init\0"
6723
6724 # char *argv[] = { init, 0 };
6725 .p2align 2
6726 argv:
6727   .long init
6728   .long 0
6729
6730
6731
6732
6733
6734
6735
6736
6737
6738
6739
6740
6741
6742
6743
6744
6745
6746
6747
6748
6749
```



```
6750 // init: The initial user-level program
6751
6752 #include "types.h"
6753 #include "stat.h"
6754 #include "user.h"
6755 #include "fcntl.h"
6756
6757 char *sh_args[] = { "sh", 0 };
6758
6759 int
6760 main(void)
6761 {
6762   int pid, wpid;
6763
6764   if(open("console", O_RDWR) < 0){
6765     mknod("console", 1, 1);
6766     open("console", O_RDWR);
6767   }
6768   dup(0);  // stdout
6769   dup(0);  // stderr
6770
6771   for(;;){
6772     printf(1, "init: starting sh\n");
6773     pid = fork();
6774     if(pid < 0){
6775       printf(1, "init: fork failed\n");
6776       exit();
6777     }
6778     if(pid == 0){
6779       exec("sh", sh_args);
6780       printf(1, "init: exec sh failed\n");
6781       exit();
6782     }
6783     while((wpid=wait()) >= 0 && wpid != pid)
6784       printf(1, "zombie!\n");
6785   }
6786 }
6787
6788
6789
6790
6791
6792
6793
6794
6795
6796
6797
6798
6799
```

```
6800 #include "syscall.h"
6801 #include "traps.h"
6802
6803 #define STUB(name) \
6804   .globl name; \
6805   name: \
6806     movl $SYS_ ## name, %eax; \
6807     int $T_SYSCALL; \
6808     ret
6809
6810 STUB(fork)
6811 STUB(exit)
6812 STUB(wait)
6813 STUB(pipe)
6814 STUB(read)
6815 STUB(write)
6816 STUB(close)
6817 STUB(kill)
6818 STUB(exec)
6819 STUB(open)
6820 STUB(mknod)
6821 STUB(unlink)
6822 STUB(fstat)
6823 STUB(link)
6824 STUB(mkdir)
6825 STUB(chdir)
6826 STUB(dup)
6827 STUB(getpid)
6828 STUB(sbrk)
6829 STUB(sleep)
6830
6831
6832
6833
6834
6835
6836
6837
6838
6839
6840
6841
6842
6843
6844
6845
6846
6847
6848
6849
```



```
6850 // Shell.
6851
6852 #include "types.h"
6853 #include "user.h"
6854 #include "fcntl.h"
6855
6856 // Parsed command representation
6857 #define EXEC  1
6858 #define REDIR 2
6859 #define PIPE  3
6860 #define LIST  4
6861 #define BACK  5
6862
6863 #define MAXARGS 10
6864
6865 struct cmd {
6866   int type;
6867 };
6868
6869 struct execcmd {
6870   int type;
6871   char *argv[MAXARGS];
6872   char *eargv[MAXARGS];
6873 };
6874
6875 struct redircmd {
6876   int type;
6877   struct cmd *cmd;
6878   char *file;
6879   char *efile;
6880   int mode;
6881   int fd;
6882 };
6883
6884 struct pipecmd {
6885   int type;
6886   struct cmd *left;
6887   struct cmd *right;
6888 };
6889
6890 struct listcmd {
6891   int type;
6892   struct cmd *left;
6893   struct cmd *right;
6894 };
6895
6896 struct backcmd {
6897   int type;
6898   struct cmd *cmd;
6899 };
```

```
6900 int fork1(void);  // Fork but panics on failure.
6901 void panic(char*);
6902 struct cmd *parsecmd(char*);
6903
6904 // Execute cmd.  Never returns.
6905 void
6906 runcmd(struct cmd *cmd)
6907 {
6908   int p[2];
6909   struct backcmd *bcmd;
6910   struct execcmd *ecmd;
6911   struct listcmd *lcmd;
6912   struct pipecmd *pcmd;
6913   struct redircmd *rcmd;
6914
6915   if(cmd == 0)
6916     exit();
6917
6918   switch(cmd->type){
6919   default:
6920     panic("runcmd");
6921
6922   case EXEC:
6923     ecmd = (struct execcmd*)cmd;
6924     if(ecmd->argv[0] == 0)
6925       exit();
6926     exec(ecmd->argv[0], ecmd->argv);
6927     printf(2, "exec %s failed\n", ecmd->argv[0]);
6928     break;
6929
6930   case REDIR:
6931     rcmd = (struct redircmd*)cmd;
6932     close(rcmd->fd);
6933     if(open(rcmd->file, rcmd->mode) < 0){
6934       printf(2, "open %s failed\n", rcmd->file);
6935       exit();
6936     }
6937     runcmd(rcmd->cmd);
6938     break;
6939
6940   case LIST:
6941     lcmd = (struct listcmd*)cmd;
6942     if(fork1() == 0)
6943       runcmd(lcmd->left);
6944     wait();
6945     runcmd(lcmd->right);
6946     break;
6947
6948
6949
```



```
6950   case PIPE:
6951     pcmd = (struct pipecmd*)cmd;
6952     if(pipe(p) < 0)
6953       panic("pipe");
6954     if(fork1() == 0){
6955       close(1);
6956       dup(p[1]);
6957       close(p[0]);
6958       close(p[1]);
6959       runcmd(pcmd->left);
6960     }
6961     if(fork1() == 0){
6962       close(0);
6963       dup(p[0]);
6964       close(p[0]);
6965       close(p[1]);
6966       runcmd(pcmd->right);
6967     }
6968     close(p[0]);
6969     close(p[1]);
6970     wait();
6971     wait();
6972     break;
6973
6974   case BACK:
6975     bcmd = (struct backcmd*)cmd;
6976     if(fork1() == 0)
6977       runcmd(bcmd->cmd);
6978     break;
6979   }
6980   exit();
6981 }
6982
6983 int
6984 getcmd(char *buf, int nbuf)
6985 {
6986   printf(2, "$ ");
6987   memset(buf, 0, nbuf);
6988   gets(buf, nbuf);
6989   if(buf[0] == 0) // EOF
6990     return -1;
6991   return 0;
6992 }
6993
6994
6995
6996
6997
6998
6999
```

```
7000 int
7001 main(void)
7002 {
7003   static char buf[100];
7004   int fd;
7005
7006   // Assumes three file descriptors open.
7007   while((fd = open("console", O_RDWR)) >= 0){
7008     if(fd >= 3){
7009       close(fd);
7010       break;
7011     }
7012   }
7013
7014   // Read and run input commands.
7015   while(getcmd(buf, sizeof(buf)) >= 0){
7016     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
7017       // Clumsy but will have to do for now.
7018       // Chdir has no effect on the parent if run in the child.
7019       buf[strlen(buf)-1] = 0;  // chop \n
7020       if(chdir(buf+3) < 0)
7021         printf(2, "cannot cd %s\n", buf+3);
7022       continue;
7023     }
7024     if(fork1() == 0)
7025       runcmd(parsecmd(buf));
7026     wait();
7027   }
7028   exit();
7029 }
7030
7031 void
7032 panic(char *s)
7033 {
7034   printf(2, "%s\n", s);
7035   exit();
7036 }
7037
7038 int
7039 fork1(void)
7040 {
7041   int pid;
7042
7043   pid = fork();
7044   if(pid == -1)
7045     panic("fork");
7046   return pid;
7047 }
7048
7049
```



```
7050 // Constructors
7051
7052 struct cmd*
7053 execcmd(void)
7054 {
7055   struct execcmd *cmd;
7056
7057   cmd = malloc(sizeof(*cmd));
7058   memset(cmd, 0, sizeof(*cmd));
7059   cmd->type = EXEC;
7060   return (struct cmd*)cmd;
7061 }
7062
7063 struct cmd*
7064 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
7065 {
7066   struct redircmd *cmd;
7067
7068   cmd = malloc(sizeof(*cmd));
7069   memset(cmd, 0, sizeof(*cmd));
7070   cmd->type = REDIR;
7071   cmd->cmd = subcmd;
7072   cmd->file = file;
7073   cmd->efile = efile;
7074   cmd->mode = mode;
7075   cmd->fd = fd;
7076   return (struct cmd*)cmd;
7077 }
7078
7079 struct cmd*
7080 pipecmd(struct cmd *left, struct cmd *right)
7081 {
7082   struct pipecmd *cmd;
7083
7084   cmd = malloc(sizeof(*cmd));
7085   memset(cmd, 0, sizeof(*cmd));
7086   cmd->type = PIPE;
7087   cmd->left = left;
7088   cmd->right = right;
7089   return (struct cmd*)cmd;
7090 }
7091
7092
7093
7094
7095
7096
7097
7098
7099
```

```
7100 struct cmd*
7101 listcmd(struct cmd *left, struct cmd *right)
7102 {
7103   struct listcmd *cmd;
7104
7105   cmd = malloc(sizeof(*cmd));
7106   memset(cmd, 0, sizeof(*cmd));
7107   cmd->type = LIST;
7108   cmd->left = left;
7109   cmd->right = right;
7110   return (struct cmd*)cmd;
7111 }
7112
7113 struct cmd*
7114 backcmd(struct cmd *subcmd)
7115 {
7116   struct backcmd *cmd;
7117
7118   cmd = malloc(sizeof(*cmd));
7119   memset(cmd, 0, sizeof(*cmd));
7120   cmd->type = BACK;
7121   cmd->cmd = subcmd;
7122   return (struct cmd*)cmd;
7123 }
7124
7125
7126
7127
7128
7129
7130
7131
7132
7133
7134
7135
7136
7137
7138
7139
7140
7141
7142
7143
7144
7145
7146
7147
7148
7149
```



```
7150 // Parsing
7151
7152 char whitespace[] = " \t\r\n\v";
7153 char symbols[] = "<|>&;()";
7154
7155 int
7156 gettoken(char **ps, char *es, char **q, char **eq)
7157 {
7158   char *s;
7159   int ret;
7160
7161   s = *ps;
7162   while(s < es && strchr(whitespace, *s))
7163     s++;
7164   if(q)
7165     *q = s;
7166   ret = *s;
7167   switch(*s){
7168   case 0:
7169     break;
7170   case '|':
7171   case '(':
7172   case ')':
7173   case ';':
7174   case '&':
7175   case '<':
7176     s++;
7177     break;
7178   case '>':
7179     s++;
7180     if(*s == '>'){
7181       ret = '+';
7182       s++;
7183     }
7184     break;
7185   default:
7186     ret = 'a';
7187     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
7188       s++;
7189     break;
7190   }
7191   if(eq)
7192     *eq = s;
7193
7194   while(s < es && strchr(whitespace, *s))
7195     s++;
7196   *ps = s;
7197   return ret;
7198 }
7199
```

```
7200 int
7201 peek(char **ps, char *es, char *toks)
7202 {
7203   char *s;
7204
7205   s = *ps;
7206   while(s < es && strchr(whitespace, *s))
7207     s++;
7208   *ps = s;
7209   return *s && strchr(toks, *s);
7210 }
7211
7212 struct cmd *parseline(char**, char*);
7213 struct cmd *parsepipe(char**, char*);
7214 struct cmd *parseexec(char**, char*);
7215 struct cmd *nulterminate(struct cmd*);
7216
7217 struct cmd*
7218 parsecmd(char *s)
7219 {
7220   char *es;
7221   struct cmd *cmd;
7222
7223   es = s + strlen(s);
7224   cmd = parseline(&s, es);
7225   peek(&s, es, "");
7226   if(s != es){
7227     printf(2, "leftovers: %s\n", s);
7228     panic("syntax");
7229   }
7230   nulterminate(cmd);
7231   return cmd;
7232 }
7233
7234 struct cmd*
7235 parseline(char **ps, char *es)
7236 {
7237   struct cmd *cmd;
7238
7239   cmd = parsepipe(ps, es);
7240   while(peek(ps, es, "&")){
7241     gettoken(ps, es, 0, 0);
7242     cmd = backcmd(cmd);
7243   }
7244   if(peek(ps, es, ";")){
7245     gettoken(ps, es, 0, 0);
7246     cmd = listcmd(cmd, parseline(ps, es));
7247   }
7248   return cmd;
7249 }
```



```
7250 struct cmd*
7251 parsepipe(char **ps, char *es)
7252 {
7253   struct cmd *cmd;
7254
7255   cmd = parseexec(ps, es);
7256   if(peek(ps, es, "|")){
7257     gettoken(ps, es, 0, 0);
7258     cmd = pipecmd(cmd, parsepipe(ps, es));
7259   }
7260   return cmd;
7261 }
7262
7263 struct cmd*
7264 parseredirs(struct cmd *cmd, char **ps, char *es)
7265 {
7266   int tok;
7267   char *q, *eq;
7268
7269   while(peek(ps, es, "<>")){
7270     tok = gettoken(ps, es, 0, 0);
7271     if(gettoken(ps, es, &q, &eq) != 'a')
7272       panic("missing file for redirection");
7273     switch(tok){
7274     case '<':
7275       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
7276       break;
7277     case '>':
7278       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
7279       break;
7280     case '+':  // >>
7281       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
7282       break;
7283     }
7284   }
7285   return cmd;
7286 }
7287
7288
7289
7290
7291
7292
7293
7294
7295
7296
7297
7298
7299
```

```
7300 struct cmd*
7301 parseblock(char **ps, char *es)
7302 {
7303   struct cmd *cmd;
7304
7305   if(!peek(ps, es, "("))
7306     panic("parseblock");
7307   gettoken(ps, es, 0, 0);
7308   cmd = parseline(ps, es);
7309   if(!peek(ps, es, ")"))
7310     panic("syntax - missing )");
7311   gettoken(ps, es, 0, 0);
7312   cmd = parseredirs(cmd, ps, es);
7313   return cmd;
7314 }
7315
7316 struct cmd*
7317 parseexec(char **ps, char *es)
7318 {
7319   char *q, *eq;
7320   int tok, argc;
7321   struct execcmd *cmd;
7322   struct cmd *ret;
7323
7324   if(peek(ps, es, "("))
7325     return parseblock(ps, es);
7326
7327   ret = execcmd();
7328   cmd = (struct execcmd*)ret;
7329
7330   argc = 0;
7331   ret = parseredirs(ret, ps, es);
7332   while(!peek(ps, es, "|)&;")){
7333     if((tok=gettoken(ps, es, &q, &eq)) == 0)
7334       break;
7335     if(tok != 'a')
7336       panic("syntax");
7337     cmd->argv[argc] = q;
7338     cmd->eargv[argc] = eq;
7339     argc++;
7340     if(argc >= MAXARGS)
7341       panic("too many args");
7342     ret = parseredirs(ret, ps, es);
7343   }
7344   cmd->argv[argc] = 0;
7345   cmd->eargv[argc] = 0;
7346   return ret;
7347 }
7348
7349
```



```
7350 // NUL-terminate all the counted strings.
7351 struct cmd*
7352 nulterminate(struct cmd *cmd)
7353 {
7354   int i;
7355   struct backcmd *bcmd;
7356   struct execcmd *ecmd;
7357   struct listcmd *lcmd;
7358   struct pipecmd *pcmd;
7359   struct redircmd *rcmd;
7360
7361   if(cmd == 0)
7362     return 0;
7363
7364   switch(cmd->type){
7365   case EXEC:
7366     ecmd = (struct execcmd*)cmd;
7367     for(i=0; ecmd->argv[i]; i++)
7368       *ecmd->eargv[i] = 0;
7369     break;
7370
7371   case REDIR:
7372     rcmd = (struct redircmd*)cmd;
7373     nulterminate(rcmd->cmd);
7374     *rcmd->efile = 0;
7375     break;
7376
7377   case PIPE:
7378     pcmd = (struct pipecmd*)cmd;
7379     nulterminate(pcmd->left);
7380     nulterminate(pcmd->right);
7381     break;
7382
7383   case LIST:
7384     lcmd = (struct listcmd*)cmd;
7385     nulterminate(lcmd->left);
7386     nulterminate(lcmd->right);
7387     break;
7388
7389   case BACK:
7390     bcmd = (struct backcmd*)cmd;
7391     nulterminate(bcmd->cmd);
7392     break;
7393   }
7394   return cmd;
7395 }
7396
7397
7398
7399
```