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/6.828/2016/xv6.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 (entryother.S, mp.h, mp.c, lapic.c)
    FreeBSD (ioapic.c)
    NetBSD (console.c)

The following people have made contributions: Russ Cox (context switching, locking), Cliff Frey (MP), Xiao Yu (MP), Nickolai Zeldovich, and Austin Clements.

We are also grateful for the bug reports and patches contributed by Silas Boyd-Wickizer, Cody Cutler, Mike CAT, Nelson Elhage, Nathaniel Filardo, Peter Froehlich, Yakir Goaron, Shivam Handa, Bryan Henry, Jim Huang, Anders Kaseorg, kehao95, Wolfgang Keller, Eddie Kohler, Imbar Marinescu, Yandong Mao, Hitoshi Mitake, Carmi Merimovich, Joel Nider, Greg Price, Ayan Shafqat, Eldar Sehayek, Yongming Shen, Cam Tenny, Rafael Ubal, Warren Toomey, Stephen Tu, Pablo Ventura, Xi Wang, Keiichi Watanabe, Nicolas Wolovick, Jindong Zhang, and Zou Chang Wei.

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). If you have suggestions for improvements, please keep in mind that the main purpose of xv6 is as a teaching operating system for MIT's 6.828. For example, we are in particular interested in simplifications and clarifications, instead of suggestions for new systems calls, more portability, etc.

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/6.828/2016/tools.html. Then run "make TOOLPREFIX=i386-jos-elf-".

To run xv6, install the QEMU PC simulators. To run in QEMU, run "make qemu".

To create a typeset version of the code, run "make xv6.pdf". This requires the "mpage" 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 2658
        0374 2428 2466 2657 2658
```

indicates that swtch is defined on line 2658 and is mentioned on five lines
on sheets 03, 24, and 26.

```
0100 typedef unsigned int   uint;
0101 typedef unsigned short ushort;
0102 typedef unsigned char  uchar;
0103 typedef uint pde_t;
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 KSTACKSIZE 4096  // size of per-process kernel stack
0152 #define NCPU          8  // maximum number of CPUs
0153 #define NOFILE       16  // open files per process
0154 #define NFILE       100  // open files per system
0155 #define NINODE       50  // maximum number of active i-nodes
0156 #define NDEV         10  // maximum major device number
0157 #define ROOTDEV       1  // device number of file system root disk
0158 #define MAXARG       32  // max exec arguments
0159 #define MAXOPBLOCKS  10  // max # of blocks any FS op writes
0160 #define LOGSIZE      (MAXOPBLOCKS*3)  // max data blocks in on-disk log
0161 #define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
0162 #define FSSIZE       1000  // size of file system in blocks
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 // Memory layout
0201
0202 #define EXTMEM  0x100000            // Start of extended memory
0203 #define PHYSTOP 0xE000000           // Top physical memory
0204 #define DEVSPACE 0xFE000000         // Other devices are at high addresses
0205
0206 // Key addresses for address space layout (see kmap in vm.c for layout)
0207 #define KERNBASE 0x80000000         // First kernel virtual address
0208 #define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked
0209
0210 #define V2P(a) (((uint) (a)) - KERNBASE)
0211 #define P2V(a) (((void *) (a)) + KERNBASE)
0212
0213 #define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
0214 #define P2V_WO(x) ((x) + KERNBASE)    // same as P2V, but without casts
0215
0216
0217
0218
0219
0220
0221
0222
0223
0224
0225
0226
0227
0228
0229
0230
0231
0232
0233
0234
0235
0236
0237
0238
0239
0240
0241
0242
0243
0244
0245
0246
0247
0248
0249
```

```
0250 struct buf;
0251 struct context;
0252 struct file;
0253 struct inode;
0254 struct pipe;
0255 struct proc;
0256 struct rtcdate;
0257 struct spinlock;
0258 struct sleeplock;
0259 struct stat;
0260 struct superblock;
0261
0262 // bio.c
0263 void            binit(void);
0264 struct buf*     bread(uint, uint);
0265 void            brelse(struct buf*);
0266 void            bwrite(struct buf*);
0267
0268 // console.c
0269 void            consoleinit(void);
0270 void            cprintf(char*, ...);
0271 void            consoleintr(int(*)(void));
0272 void            panic(char*) __attribute__((noreturn));
0273
0274 // exec.c
0275 int             exec(char*, char**);
0276
0277 // file.c
0278 struct file*    filealloc(void);
0279 void            fileclose(struct file*);
0280 struct file*    filedup(struct file*);
0281 void            fileinit(void);
0282 int             fileread(struct file*, char*, int n);
0283 int             filestat(struct file*, struct stat*);
0284 int             filewrite(struct file*, char*, int n);
0285
0286 // fs.c
0287 void            readsb(int dev, struct superblock *sb);
0288 int             dirlink(struct inode*, char*, uint);
0289 struct inode*   dirlookup(struct inode*, char*, uint*);
0290 struct inode*   ialloc(uint, short);
0291 struct inode*   idup(struct inode*);
0292 void            iinit(int dev);
0293 void            ilock(struct inode*);
0294 void            iput(struct inode*);
0295 void            iunlock(struct inode*);
0296 void            iunlockput(struct inode*);
0297 void            iupdate(struct inode*);
0298 int             namecmp(const char*, const char*);
0299 struct inode*   namei(char*);
```



```
0300 struct inode*   nameiparent(char*, char*);
0301 int             readi(struct inode*, char*, uint, uint);
0302 void            stati(struct inode*, struct stat*);
0303 int             writei(struct inode*, char*, uint, uint);
0304
0305 // ide.c
0306 void            ideinit(void);
0307 void            ideintr(void);
0308 void            iderw(struct buf*);
0309
0310 // ioapic.c
0311 void            ioapicenable(int irq, int cpu);
0312 extern uchar    ioapicid;
0313 void            ioapicinit(void);
0314
0315 // kalloc.c
0316 char*           kalloc(void);
0317 void            kfree(char*);
0318 void            kinit1(void*, void*);
0319 void            kinit2(void*, void*);
0320
0321 // kbd.c
0322 void            kbdintr(void);
0323
0324 // lapic.c
0325 void            cmostime(struct rtcdate *r);
0326 int             cpunum(void);
0327 extern volatile uint*    lapic;
0328 void            lapiceoi(void);
0329 void            lapicinit(void);
0330 void            lapicstartap(uchar, uint);
0331 void            microdelay(int);
0332
0333 // log.c
0334 void            initlog(int dev);
0335 void            log_write(struct buf*);
0336 void            begin_op();
0337 void            end_op();
0338
0339 // mp.c
0340 extern int      ismp;
0341 void            mpinit(void);
0342
0343 // picirq.c
0344 void            picenable(int);
0345 void            picinit(void);
0346
0347
0348
0349
```

```
0350 // pipe.c
0351 int             pipealloc(struct file**, struct file**);
0352 void            pipeclose(struct pipe*, int);
0353 int             piperead(struct pipe*, char*, int);
0354 int             pipewrite(struct pipe*, char*, int);
0355
0356
0357 // proc.c
0358 void            exit(void);
0359 int             fork(void);
0360 int             growproc(int);
0361 int             kill(int);
0362 void            pinit(void);
0363 void            procdump(void);
0364 void            scheduler(void) __attribute__((noreturn));
0365 void            sched(void);
0366 void            sleep(void*, struct spinlock*);
0367 void            userinit(void);
0368 int             wait(void);
0369 void            wakeup(void*);
0370 void            yield(void);
0371
0372 // swtch.S
0373 void            swtch(struct context**, struct context*);
0374
0375 // spinlock.c
0376 void            acquire(struct spinlock*);
0377 void            getcallerpcs(void*, uint*);
0378 int             holding(struct spinlock*);
0379 void            initlock(struct spinlock*, char*);
0380 void            release(struct spinlock*);
0381 void            pushcli(void);
0382 void            popcli(void);
0383
0384 // sleeplock.c
0385 void            acquiresleep(struct sleeplock*);
0386 void            releasesleep(struct sleeplock*);
0387 int             holdingsleep(struct sleeplock*);
0388 void            initsleeplock(struct sleeplock*, char*);
0389
0390 // string.c
0391 int             memcmp(const void*, const void*, uint);
0392 void*           memmove(void*, const void*, uint);
0393 void*           memset(void*, int, uint);
0394 char*           safestrcpy(char*, const char*, int);
0395 int             strlen(const char*);
0396 int             strncmp(const char*, const char*, uint);
0397 char*           strncpy(char*, const char*, int);
0398
0399
```



```
0400 // syscall.c
0401 int             argint(int, int*);
0402 int             argptr(int, char**, int);
0403 int             argstr(int, char**);
0404 int             fetchint(uint, int*);
0405 int             fetchstr(uint, char**);
0406 void            syscall(void);
0407
0408 // timer.c
0409 void            timerinit(void);
0410
0411 // trap.c
0412 void            idtinit(void);
0413 extern uint     ticks;
0414 void            tvinit(void);
0415 extern struct spinlock tickslock;
0416
0417 // uart.c
0418 void            uartinit(void);
0419 void            uartintr(void);
0420 void            uartputc(int);
0421
0422 // vm.c
0423 void            seginit(void);
0424 void            kvmalloc(void);
0425 pde_t*          setupkvm(void);
0426 char*           uva2ka(pde_t*, char*);
0427 int             allocuvm(pde_t*, uint, uint);
0428 int             deallocuvm(pde_t*, uint, uint);
0429 void            freevm(pde_t*);
0430 void            inituvm(pde_t*, char*, uint);
0431 int             loaduvm(pde_t*, char*, struct inode*, uint, uint);
0432 pde_t*          copyuvm(pde_t*, uint);
0433 void            switchuvm(struct proc*);
0434 void            switchkvm(void);
0435 int             copyout(pde_t*, uint, void*, uint);
0436 void            clearpteu(pde_t *pgdir, char *uva);
0437
0438 // number of elements in fixed-size array
0439 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
0440
0441
0442
0443
0444
0445
0446
0447
0448
0449
```

```
0450 // Routines to let C code use special x86 instructions.
0451
0452 static inline uchar
0453 inb(ushort port)
0454 {
0455   uchar data;
0456
0457   asm volatile("in %1,%0" : "=a" (data) : "d" (port));
0458   return data;
0459 }
0460
0461 static inline void
0462 insl(int port, void *addr, int cnt)
0463 {
0464   asm volatile("cld; rep insl" :
0465                "=D" (addr), "=c" (cnt) :
0466                "d" (port), "0" (addr), "1" (cnt) :
0467                "memory", "cc");
0468 }
0469
0470 static inline void
0471 outb(ushort port, uchar data)
0472 {
0473   asm volatile("out %0,%1" : : "a" (data), "d" (port));
0474 }
0475
0476 static inline void
0477 outw(ushort port, ushort data)
0478 {
0479   asm volatile("out %0,%1" : : "a" (data), "d" (port));
0480 }
0481
0482 static inline void
0483 outsl(int port, const void *addr, int cnt)
0484 {
0485   asm volatile("cld; rep outsl" :
0486                "=S" (addr), "=c" (cnt) :
0487                "d" (port), "0" (addr), "1" (cnt) :
0488                "cc");
0489 }
0490
0491 static inline void
0492 stosb(void *addr, int data, int cnt)
0493 {
0494   asm volatile("cld; rep stosb" :
0495                "=D" (addr), "=c" (cnt) :
0496                "0" (addr), "1" (cnt), "a" (data) :
0497                "memory", "cc");
0498 }
0499
```



```
0500 static inline void
0501 stosl(void *addr, int data, int cnt)
0502 {
0503   asm volatile("cld; rep stosl" :
0504                "=D" (addr), "=c" (cnt) :
0505                "0" (addr), "1" (cnt), "a" (data) :
0506                "memory", "cc");
0507 }
0508
0509 struct segdesc;
0510
0511 static inline void
0512 lgdt(struct segdesc *p, int size)
0513 {
0514   volatile ushort pd[3];
0515
0516   pd[0] = size-1;
0517   pd[1] = (uint)p;
0518   pd[2] = (uint)p >> 16;
0519
0520   asm volatile("lgdt (%0)" : : "r" (pd));
0521 }
0522
0523 struct gatedesc;
0524
0525 static inline void
0526 lidt(struct gatedesc *p, int size)
0527 {
0528   volatile ushort pd[3];
0529
0530   pd[0] = size-1;
0531   pd[1] = (uint)p;
0532   pd[2] = (uint)p >> 16;
0533
0534   asm volatile("lidt (%0)" : : "r" (pd));
0535 }
0536
0537 static inline void
0538 ltr(ushort sel)
0539 {
0540   asm volatile("ltr %0" : : "r" (sel));
0541 }
0542
0543 static inline uint
0544 readeflags(void)
0545 {
0546   uint eflags;
0547   asm volatile("pushfl; popl %0" : "=r" (eflags));
0548   return eflags;
0549 }
```

```
0550 static inline void
0551 loadgs(ushort v)
0552 {
0553   asm volatile("movw %0, %%gs" : : "r" (v));
0554 }
0555
0556 static inline void
0557 cli(void)
0558 {
0559   asm volatile("cli");
0560 }
0561
0562 static inline void
0563 sti(void)
0564 {
0565   asm volatile("sti");
0566 }
0567
0568 static inline uint
0569 xchg(volatile uint *addr, uint newval)
0570 {
0571   uint result;
0572
0573   // The + in "+m" denotes a read-modify-write operand.
0574   asm volatile("lock; xchgl %0, %1" :
0575                "+m" (*addr), "=a" (result) :
0576                "1" (newval) :
0577                "cc");
0578   return result;
0579 }
0580
0581 static inline uint
0582 rcr2(void)
0583 {
0584   uint val;
0585   asm volatile("movl %%cr2,%0" : "=r" (val));
0586   return val;
0587 }
0588
0589 static inline void
0590 lcr3(uint val)
0591 {
0592   asm volatile("movl %0,%%cr3" : : "r" (val));
0593 }
0594
0595
0596
0597
0598
0599
```



```
0600 // Layout of the trap frame built on the stack by the
0601 // hardware and by trapasm.S, and passed to trap().
0602 struct trapframe {
0603   // registers as pushed by pusha
0604   uint edi;
0605   uint esi;
0606   uint ebp;
0607   uint oesp;      // useless & ignored
0608   uint ebx;
0609   uint edx;
0610   uint ecx;
0611   uint eax;
0612
0613   // rest of trap frame
0614   ushort gs;
0615   ushort padding1;
0616   ushort fs;
0617   ushort padding2;
0618   ushort es;
0619   ushort padding3;
0620   ushort ds;
0621   ushort padding4;
0622   uint trapno;
0623
0624   // below here defined by x86 hardware
0625   uint err;
0626   uint eip;
0627   ushort cs;
0628   ushort padding5;
0629   uint eflags;
0630
0631   // below here only when crossing rings, such as from user to kernel
0632   uint esp;
0633   ushort ss;
0634   ushort padding6;
0635 };
0636
0637
0638
0639
0640
0641
0642
0643
0644
0645
0646
0647
0648
0649
```

```
0650 //
0651 // assembler macros to create x86 segments
0652 //
0653
0654 #define SEG_NULLASM                                             \
0655         .word 0, 0;                                             \
0656         .byte 0, 0, 0, 0
0657
0658 // The 0xC0 means the limit is in 4096-byte units
0659 // and (for executable segments) 32-bit mode.
0660 #define SEG_ASM(type,base,lim)                                  \
0661         .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
0662         .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
0663                 (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
0664
0665 #define STA_X     0x8       // Executable segment
0666 #define STA_E     0x4       // Expand down (non-executable segments)
0667 #define STA_C     0x4       // Conforming code segment (executable only)
0668 #define STA_W     0x2       // Writeable (non-executable segments)
0669 #define STA_R     0x2       // Readable (executable segments)
0670 #define STA_A     0x1       // Accessed
0671
0672
0673
0674
0675
0676
0677
0678
0679
0680
0681
0682
0683
0684
0685
0686
0687
0688
0689
0690
0691
0692
0693
0694
0695
0696
0697
0698
0699
```



```
0700 // This file contains definitions for the
0701 // x86 memory management unit (MMU).
0702
0703 // Eflags register
0704 #define FL_CF           0x00000001      // Carry Flag
0705 #define FL_PF           0x00000004      // Parity Flag
0706 #define FL_AF           0x00000010      // Auxiliary carry Flag
0707 #define FL_ZF           0x00000040      // Zero Flag
0708 #define FL_SF           0x00000080      // Sign Flag
0709 #define FL_TF           0x00000100      // Trap Flag
0710 #define FL_IF           0x00000200      // Interrupt Enable
0711 #define FL_DF           0x00000400      // Direction Flag
0712 #define FL_OF           0x00000800      // Overflow Flag
0713 #define FL_IOPL_MASK    0x00003000      // I/O Privilege Level bitmask
0714 #define FL_IOPL_0       0x00000000      //   IOPL == 0
0715 #define FL_IOPL_1       0x00001000      //   IOPL == 1
0716 #define FL_IOPL_2       0x00002000      //   IOPL == 2
0717 #define FL_IOPL_3       0x00003000      //   IOPL == 3
0718 #define FL_NT           0x00004000      // Nested Task
0719 #define FL_RF           0x00010000      // Resume Flag
0720 #define FL_VM           0x00020000      // Virtual 8086 mode
0721 #define FL_AC           0x00040000      // Alignment Check
0722 #define FL_VIF          0x00080000      // Virtual Interrupt Flag
0723 #define FL_VIP          0x00100000      // Virtual Interrupt Pending
0724 #define FL_ID           0x00200000      // ID flag
0725
0726 // Control Register flags
0727 #define CR0_PE          0x00000001      // Protection Enable
0728 #define CR0_MP          0x00000002      // Monitor coProcessor
0729 #define CR0_EM          0x00000004      // Emulation
0730 #define CR0_TS          0x00000008      // Task Switched
0731 #define CR0_ET          0x00000010      // Extension Type
0732 #define CR0_NE          0x00000020      // Numeric Errror
0733 #define CR0_WP          0x00010000      // Write Protect
0734 #define CR0_AM          0x00040000      // Alignment Mask
0735 #define CR0_NW          0x20000000      // Not Writethrough
0736 #define CR0_CD          0x40000000      // Cache Disable
0737 #define CR0_PG          0x80000000      // Paging
0738
0739 #define CR4_PSE         0x00000010      // Page size extension
0740
0741 // various segment selectors.
0742 #define SEG_KCODE 1  // kernel code
0743 #define SEG_KDATA 2  // kernel data+stack
0744 #define SEG_KCPU  3  // kernel per-cpu data
0745 #define SEG_UCODE 4  // user code
0746 #define SEG_UDATA 5  // user data+stack
0747 #define SEG_TSS   6  // this process's task state
0748
0749
```

```
0750 // cpu->gdt[NSEGS] holds the above segments.
0751 #define NSEGS     7
0752
0753
0754
0755
0756
0757
0758
0759
0760
0761
0762
0763
0764
0765
0766
0767
0768
0769
0770
0771
0772
0773
0774
0775
0776
0777
0778
0779
0780
0781
0782
0783
0784
0785
0786
0787
0788
0789
0790
0791
0792
0793
0794
0795
0796
0797
0798
0799
```



```
0800 #ifndef __ASSEMBLER__
0801 // Segment Descriptor
0802 struct segdesc {
0803   uint lim_15_0 : 16;  // Low bits of segment limit
0804   uint base_15_0 : 16; // Low bits of segment base address
0805   uint base_23_16 : 8; // Middle bits of segment base address
0806   uint type : 4;       // Segment type (see STS_ constants)
0807   uint s : 1;          // 0 = system, 1 = application
0808   uint dpl : 2;        // Descriptor Privilege Level
0809   uint p : 1;          // Present
0810   uint lim_19_16 : 4;  // High bits of segment limit
0811   uint avl : 1;        // Unused (available for software use)
0812   uint rsv1 : 1;       // Reserved
0813   uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
0814   uint g : 1;          // Granularity: limit scaled by 4K when set
0815   uint base_31_24 : 8; // High bits of segment base address
0816 };
0817
0818 // Normal segment
0819 #define SEG(type, base, lim, dpl) (struct segdesc)    \
0820 { ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff,      \
0821   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
0822   (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
0823 #define SEG16(type, base, lim, dpl) (struct segdesc)  \
0824 { (lim) & 0xffff, (uint)(base) & 0xffff,              \
0825   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
0826   (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
0827 #endif
0828
0829 #define DPL_USER    0x3     // User DPL
0830
0831 // Application segment type bits
0832 #define STA_X       0x8     // Executable segment
0833 #define STA_E       0x4     // Expand down (non-executable segments)
0834 #define STA_C       0x4     // Conforming code segment (executable only)
0835 #define STA_W       0x2     // Writeable (non-executable segments)
0836 #define STA_R       0x2     // Readable (executable segments)
0837 #define STA_A       0x1     // Accessed
0838
0839 // System segment type bits
0840 #define STS_T16A    0x1     // Available 16-bit TSS
0841 #define STS_LDT     0x2     // Local Descriptor Table
0842 #define STS_T16B    0x3     // Busy 16-bit TSS
0843 #define STS_CG16    0x4     // 16-bit Call Gate
0844 #define STS_TG      0x5     // Task Gate / Coum Transmitions
0845 #define STS_IG16    0x6     // 16-bit Interrupt Gate
0846 #define STS_TG16    0x7     // 16-bit Trap Gate
0847 #define STS_T32A    0x9     // Available 32-bit TSS
0848 #define STS_T32B    0xB     // Busy 32-bit TSS
0849 #define STS_CG32    0xC     // 32-bit Call Gate
```

```
0850 #define STS_IG32    0xE     // 32-bit Interrupt Gate
0851 #define STS_TG32    0xF     // 32-bit Trap Gate
0852
0853 // A virtual address 'la' has a three-part structure as follows:
0854 //
0855 // +--------10------+-------10-------+---------12----------+
0856 // | Page Directory |   Page Table   | Offset within Page  |
0857 // |      Index     |      Index     |                     |
0858 // +----------------+----------------+---------------------+
0859 //  \--- PDX(va) --/ \--- PTX(va) --/
0860
0861 // page directory index
0862 #define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)
0863
0864 // page table index
0865 #define PTX(va)         (((uint)(va) >> PTXSHIFT) & 0x3FF)
0866
0867 // construct virtual address from indexes and offset
0868 #define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
0869
0870 // Page directory and page table constants.
0871 #define NPDENTRIES      1024    // # directory entries per page directory
0872 #define NPTENTRIES      1024    // # PTEs per page table
0873 #define PGSIZE          4096    // bytes mapped by a page
0874
0875 #define PGSHIFT         12      // log2(PGSIZE)
0876 #define PTXSHIFT        12      // offset of PTX in a linear address
0877 #define PDXSHIFT        22      // offset of PDX in a linear address
0878
0879 #define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
0880 #define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
0881
0882 // Page table/directory entry flags.
0883 #define PTE_P           0x001   // Present
0884 #define PTE_W           0x002   // Writeable
0885 #define PTE_U           0x004   // User
0886 #define PTE_PWT         0x008   // Write-Through
0887 #define PTE_PCD         0x010   // Cache-Disable
0888 #define PTE_A           0x020   // Accessed
0889 #define PTE_D           0x040   // Dirty
0890 #define PTE_PS          0x080   // Page Size
0891 #define PTE_MBZ         0x180   // Bits must be zero
0892
0893 // Address in page table or page directory entry
0894 #define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
0895 #define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF)
0896
0897 #ifndef __ASSEMBLER__
0898 typedef uint pte_t;
0899
```



```
0900 // Task state segment format
0901 struct taskstate {
0902   uint link;         // Old ts selector
0903   uint esp0;         // Stack pointers and segment selectors
0904   ushort ss0;        //   after an increase in privilege level
0905   ushort padding1;
0906   uint *esp1;
0907   ushort ss1;
0908   ushort padding2;
0909   uint *esp2;
0910   ushort ss2;
0911   ushort padding3;
0912   void *cr3;         // Page directory base
0913   uint *eip;         // Saved state from last task switch
0914   uint eflags;
0915   uint eax;          // More saved state (registers)
0916   uint ecx;
0917   uint edx;
0918   uint ebx;
0919   uint *esp;
0920   uint *ebp;
0921   uint esi;
0922   uint edi;
0923   ushort es;         // Even more saved state (segment selectors)
0924   ushort padding4;
0925   ushort cs;
0926   ushort padding5;
0927   ushort ss;
0928   ushort padding6;
0929   ushort ds;
0930   ushort padding7;
0931   ushort fs;
0932   ushort padding8;
0933   ushort gs;
0934   ushort padding9;
0935   ushort ldt;
0936   ushort padding10;
0937   ushort t;          // Trap on task switch
0938   ushort iomb;       // I/O map base address
0939 };
0940
0941
0942
0943
0944
0945
0946
0947
0948
0949
```

```
0950 // Gate descriptors for interrupts and traps
0951 struct gatedesc {
0952   uint off_15_0 : 16;   // low 16 bits of offset in segment
0953   uint cs : 16;         // code segment selector
0954   uint args : 5;        // # args, 0 for interrupt/trap gates
0955   uint rsv1 : 3;        // reserved(should be zero I guess)
0956   uint type : 4;        // type(STS_{TG,IG32,TG32})
0957   uint s : 1;           // must be 0 (system)
0958   uint dpl : 2;         // descriptor(meaning new) privilege level
0959   uint p : 1;           // Present
0960   uint off_31_16 : 16;  // high bits of offset in segment
0961 };
0962
0963 // Set up a normal interrupt/trap gate descriptor.
0964 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
0965 //   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
0966 // - sel: Code segment selector for interrupt/trap handler
0967 // - off: Offset in code segment for interrupt/trap handler
0968 // - dpl: Descriptor Privilege Level -
0969 //        the privilege level required for software to invoke
0970 //        this interrupt/trap gate explicitly using an int instruction.
0971 #define SETGATE(gate, istrap, sel, off, d)                \
0972 {                                                         \
0973   (gate).off_15_0 = (uint)(off) & 0xffff;                \
0974   (gate).cs = (sel);                                      \
0975   (gate).args = 0;                                        \
0976   (gate).rsv1 = 0;                                        \
0977   (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
0978   (gate).s = 0;                                           \
0979   (gate).dpl = (d);                                       \
0980   (gate).p = 1;                                           \
0981   (gate).off_31_16 = (uint)(off) >> 16;                  \
0982 }
0983
0984 #endif
0985
0986
0987
0988
0989
0990
0991
0992
0993
0994
0995
0996
0997
0998
0999
```



```
1000 // Format of an ELF executable file
1001
1002 #define ELF_MAGIC 0x464C457FU  // "\x7FELF" in little endian
1003
1004 // File header
1005 struct elfhdr {
1006   uint magic;  // must equal ELF_MAGIC
1007   uchar elf[12];
1008   ushort type;
1009   ushort machine;
1010   uint version;
1011   uint entry;
1012   uint phoff;
1013   uint shoff;
1014   uint flags;
1015   ushort ehsize;
1016   ushort phentsize;
1017   ushort phnum;
1018   ushort shentsize;
1019   ushort shnum;
1020   ushort shstrndx;
1021 };
1022
1023 // Program section header
1024 struct proghdr {
1025   uint type;
1026   uint off;
1027   uint vaddr;
1028   uint paddr;
1029   uint filesz;
1030   uint memsz;
1031   uint flags;
1032   uint align;
1033 };
1034
1035 // Values for Proghdr type
1036 #define ELF_PROG_LOAD           1
1037
1038 // Flag bits for Proghdr flags
1039 #define ELF_PROG_FLAG_EXEC      1
1040 #define ELF_PROG_FLAG_WRITE     2
1041 #define ELF_PROG_FLAG_READ      4
1042
1043
1044
1045
1046
1047
1048
1049
```

```
1050 // Blank page.
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
```



```
1100 # The xv6 kernel starts executing in this file. This file is linked with
1101 # the kernel C code, so it can refer to kernel symbols such as main().
1102 # The boot block (bootasm.S and bootmain.c) jumps to entry below.
1103
1104 # Multiboot header, for multiboot boot loaders like GNU Grub.
1105 # http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
1106 #
1107 # Using GRUB 2, you can boot xv6 from a file stored in a
1108 # Linux file system by copying kernel or kernelmemfs to /boot
1109 # and then adding this menu entry:
1110 #
1111 # menuentry "xv6" {
1112 #  insmod ext2
1113 #  set root='(hd0,msdos1)'
1114 #  set kernel='/boot/kernel'
1115 #  echo "Loading ${kernel}..."
1116 #  multiboot ${kernel} ${kernel}
1117 #  boot
1118 # }
1119
1120 #include "asm.h"
1121 #include "memlayout.h"
1122 #include "mmu.h"
1123 #include "param.h"
1124
1125 # Multiboot header.  Data to direct multiboot loader.
1126 .p2align 2
1127 .text
1128 .globl multiboot_header
1129 multiboot_header:
1130   #define magic 0x1badb002
1131   #define flags 0
1132   .long magic
1133   .long flags
1134   .long (-magic-flags)
1135
1136 # By convention, the _start symbol specifies the ELF entry point.
1137 # Since we haven't set up virtual memory yet, our entry point is
1138 # the physical address of 'entry'.
1139 .globl _start
1140 _start = V2P_WO(entry)
1141
1142 # Entering xv6 on boot processor, with paging off.
1143 .globl entry
1144 entry:
1145   # Turn on page size extension for 4Mbyte pages
1146   movl    %cr4, %eax
1147   orl     $(CR4_PSE), %eax
1148   movl    %eax, %cr4
1149   # Set page directory
```

```
1150   movl    $(V2P_WO(entrypgdir)), %eax
1151   movl    %eax, %cr3
1152   # Turn on paging.
1153   movl    %cr0, %eax
1154   orl     $(CR0_PG|CR0_WP), %eax
1155   movl    %eax, %cr0
1156
1157   # Set up the stack pointer.
1158   movl $(stack + KSTACKSIZE), %esp
1159
1160   # Jump to main(), and switch to executing at
1161   # high addresses. The indirect call is needed because
1162   # the assembler produces a PC-relative instruction
1163   # for a direct jump.
1164   mov $main, %eax
1165   jmp *%eax
1166
1167 .comm stack, KSTACKSIZE
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
```



```
1200 #include "asm.h"
1201 #include "memlayout.h"
1202 #include "mmu.h"
1203
1204 # Each non-boot CPU ("AP") is started up in response to a STARTUP
1205 # IPI from the boot CPU.  Section B.4.2 of the Multi-Processor
1206 # Specification says that the AP will start in real mode with CS:IP
1207 # set to XY00:0000, where XY is an 8-bit value sent with the
1208 # STARTUP. Thus this code must start at a 4096-byte boundary.
1209 #
1210 # Because this code sets DS to zero, it must sit
1211 # at an address in the low 2^16 bytes.
1212 #
1213 # Startothers (in main.c) sends the STARTUPs one at a time.
1214 # It copies this code (start) at 0x7000.  It puts the address of
1215 # a newly allocated per-core stack in start-4,the address of the
1216 # place to jump to (mpenter) in start-8, and the physical address
1217 # of entrypgdir in start-12.
1218 #
1219 # This code combines elements of bootasm.S and entry.S.
1220
1221 .code16
1222 .globl start
1223 start:
1224   cli
1225
1226   # Zero data segment registers DS, ES, and SS.
1227   xorw    %ax,%ax
1228   movw    %ax,%ds
1229   movw    %ax,%es
1230   movw    %ax,%ss
1231
1232   # Switch from real to protected mode.  Use a bootstrap GDT that makes
1233   # virtual addresses map directly to physical addresses so that the
1234   # effective memory map doesn't change during the transition.
1235   lgdt    gdtdesc
1236   movl    %cr0, %eax
1237   orl     $CR0_PE, %eax
1238   movl    %eax, %cr0
1239
1240   # Complete the transition to 32-bit protected mode by using a long jmp
1241   # to reload %cs and %eip.  The segment descriptors are set up with no
1242   # translation, so that the mapping is still the identity mapping.
1243   ljmpl    $(SEG_KCODE<<3), $(start32)
1244
1245
1246
1247
1248
1249
```

```
1250 .code32  # Tell assembler to generate 32-bit code now.
1251 start32:
1252   # Set up the protected-mode data segment registers
1253   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
1254   movw    %ax, %ds                # -> DS: Data Segment
1255   movw    %ax, %es                # -> ES: Extra Segment
1256   movw    %ax, %ss                # -> SS: Stack Segment
1257   movw    $0, %ax                 # Zero segments not ready for use
1258   movw    %ax, %fs                # -> FS
1259   movw    %ax, %gs                # -> GS
1260
1261   # Turn on page size extension for 4Mbyte pages
1262   movl    %cr4, %eax
1263   orl     $(CR4_PSE), %eax
1264   movl    %eax, %cr4
1265   # Use entrypgdir as our initial page table
1266   movl    (start-12), %eax
1267   movl    %eax, %cr3
1268   # Turn on paging.
1269   movl    %cr0, %eax
1270   orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
1271   movl    %eax, %cr0
1272
1273   # Switch to the stack allocated by startothers()
1274   movl    (start-4), %esp
1275   # Call mpenter()
1276   call      *(start-8)
1277
1278   movw    $0x8a00, %ax
1279   movw    %ax, %dx
1280   outw    %ax, %dx
1281   movw    $0x8ae0, %ax
1282   outw    %ax, %dx
1283 spin:
1284   jmp     spin
1285
1286 .p2align 2
1287 gdt:
1288   SEG_NULLASM
1289   SEG_ASM(STA_X|STA_R, 0, 0xffffffff)
1290   SEG_ASM(STA_W, 0, 0xffffffff)
1291
1292
1293 gdtdesc:
1294   .word   (gdtdesc - gdt - 1)
1295   .long   gdt
1296
1297
1298
1299
```



```
1300 #include "types.h"
1301 #include "defs.h"
1302 #include "param.h"
1303 #include "memlayout.h"
1304 #include "mmu.h"
1305 #include "proc.h"
1306 #include "x86.h"
1307
1308 static void startothers(void);
1309 static void mpmain(void)  __attribute__((noreturn));
1310 extern pde_t *kpgdir;
1311 extern char end[]; // first address after kernel loaded from ELF file
1312
1313 // Bootstrap processor starts running C code here.
1314 // Allocate a real stack and switch to it, first
1315 // doing some setup required for memory allocator to work.
1316 int
1317 main(void)
1318 {
1319   kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320   kvmalloc();      // kernel page table
1321   mpinit();        // detect other processors
1322   lapicinit();     // interrupt controller
1323   seginit();       // segment descriptors
1324   cprintf("\ncpu%d: starting xv6\n\n", cpunum());
1325   picinit();       // another interrupt controller
1326   ioapicinit();    // another interrupt controller
1327   consoleinit();   // console hardware
1328   uartinit();      // serial port
1329   pinit();         // process table
1330   tvinit();        // trap vectors
1331   binit();         // buffer cache
1332   fileinit();      // file table
1333   ideinit();       // disk
1334   if(!ismp)
1335     timerinit();   // uniprocessor timer
1336   startothers();   // start other processors
1337   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1338   userinit();      // first user process
1339   mpmain();        // finish this processor's setup
1340 }
1341
1342
1343
1344
1345
1346
1347
1348
1349
```

```
1350 // Other CPUs jump here from entryother.S.
1351 static void
1352 mpenter(void)
1353 {
1354   switchkvm();
1355   seginit();
1356   lapicinit();
1357   mpmain();
1358 }
1359
1360 // Common CPU setup code.
1361 static void
1362 mpmain(void)
1363 {
1364   cprintf("cpu%d: starting\n", cpunum());
1365   idtinit();       // load idt register
1366   xchg(&cpu->started, 1); // tell startothers() we're up
1367   scheduler();     // start running processes
1368 }
1369
1370 pde_t entrypgdir[];  // For entry.S
1371
1372 // Start the non-boot (AP) processors.
1373 static void
1374 startothers(void)
1375 {
1376   extern uchar _binary_entryother_start[], _binary_entryother_size[];
1377   uchar *code;
1378   struct cpu *c;
1379   char *stack;
1380
1381   // Write entry code to unused memory at 0x7000.
1382   // The linker has placed the image of entryother.S in
1383   // _binary_entryother_start.
1384   code = P2V(0x7000);
1385   memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
1386
1387   for(c = cpus; c < cpus+ncpu; c++){
1388     if(c == cpus+cpunum())  // We've started already.
1389       continue;
1390
1391     // Tell entryother.S what stack to use, where to enter, and what
1392     // pgdir to use. We cannot use kpgdir yet, because the AP processor
1393     // is running in low  memory, so we use entrypgdir for the APs too.
1394     stack = kalloc();
1395     *(void**)(code-4) = stack + KSTACKSIZE;
1396     *(void**)(code-8) = mpenter;
1397     *(int**)(code-12) = (void *) V2P(entrypgdir);
1398
1399     lapicstartap(c->apicid, V2P(code));
```



```
1400     // wait for cpu to finish mpmain()
1401     while(c->started == 0)
1402       ;
1403   }
1404 }
1405
1406 // The boot page table used in entry.S and entryother.S.
1407 // Page directories (and page tables) must start on page boundaries,
1408 // hence the __aligned__ attribute.
1409 // PTE_PS in a page directory entry enables 4Mbyte pages.
1410
1411 __attribute__((__aligned__(PGSIZE)))
1412 pde_t entrypgdir[NPDENTRIES] = {
1413   // Map VA's [0, 4MB) to PA's [0, 4MB)
1414   [0] = (0) | PTE_P | PTE_W | PTE_PS,
1415   // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1416   [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1417 };
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
```

```
1450 // Blank page.
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
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 // Mutual exclusion lock.
1501 struct spinlock {
1502   uint locked;       // Is the lock held?
1503
1504   // For debugging:
1505   char *name;        // Name of lock.
1506   struct cpu *cpu;   // The cpu holding the lock.
1507   uint pcs[10];      // The call stack (an array of program counters)
1508                      // that locked the lock.
1509 };
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
```

```
1550 // Mutual exclusion spin locks.
1551
1552 #include "types.h"
1553 #include "defs.h"
1554 #include "param.h"
1555 #include "x86.h"
1556 #include "memlayout.h"
1557 #include "mmu.h"
1558 #include "proc.h"
1559 #include "spinlock.h"
1560
1561 void
1562 initlock(struct spinlock *lk, char *name)
1563 {
1564   lk->name = name;
1565   lk->locked = 0;
1566   lk->cpu = 0;
1567 }
1568
1569 // Acquire the lock.
1570 // Loops (spins) until the lock is acquired.
1571 // Holding a lock for a long time may cause
1572 // other CPUs to waste time spinning to acquire it.
1573 void
1574 acquire(struct spinlock *lk)
1575 {
1576   pushcli(); // disable interrupts to avoid deadlock.
1577   if(holding(lk))
1578     panic("acquire");
1579
1580   // The xchg is atomic.
1581   while(xchg(&lk->locked, 1) != 0)
1582     ;
1583
1584   // Tell the C compiler and the processor to not move loads or stores
1585   // past this point, to ensure that the critical section's memory
1586   // references happen after the lock is acquired.
1587   __sync_synchronize();
1588
1589   // Record info about lock acquisition for debugging.
1590   lk->cpu = cpu;
1591   getcallerpcs(&lk, lk->pcs);
1592 }
1593
1594
1595
1596
1597
1598
1599
```



```
1600 // Release the lock.
1601 void
1602 release(struct spinlock *lk)
1603 {
1604   if(!holding(lk))
1605     panic("release");
1606
1607   lk->pcs[0] = 0;
1608   lk->cpu = 0;
1609
1610   // Tell the C compiler and the processor to not move loads or stores
1611   // past this point, to ensure that all the stores in the critical
1612   // section are visible to other cores before the lock is released.
1613   // Both the C compiler and the hardware may re-order loads and
1614   // stores; __sync_synchronize() tells them both not to.
1615   __sync_synchronize();
1616
1617   // Release the lock, equivalent to lk->locked = 0.
1618   // This code can't use a C assignment, since it might
1619   // not be atomic. A real OS would use C atomics here.
1620   asm volatile("movl $0, %0" : "+m" (lk->locked) : );
1621
1622   popcli();
1623 }
1624
1625 // Record the current call stack in pcs[] by following the %ebp chain.
1626 void
1627 getcallerpcs(void *v, uint pcs[])
1628 {
1629   uint *ebp;
1630   int i;
1631
1632   ebp = (uint*)v - 2;
1633   for(i = 0; i < 10; i++){
1634     if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
1635       break;
1636     pcs[i] = ebp[1];     // saved %eip
1637     ebp = (uint*)ebp[0]; // saved %ebp
1638   }
1639   for(; i < 10; i++)
1640     pcs[i] = 0;
1641 }
1642
1643 // Check whether this cpu is holding the lock.
1644 int
1645 holding(struct spinlock *lock)
1646 {
1647   return lock->locked && lock->cpu == cpu;
1648 }
1649
```

```
1650 // Pushcli/popcli are like cli/sti except that they are matched:
1651 // it takes two popcli to undo two pushcli.  Also, if interrupts
1652 // are off, then pushcli, popcli leaves them off.
1653
1654 void
1655 pushcli(void)
1656 {
1657   int eflags;
1658
1659   eflags = readeflags();
1660   cli();
1661   if(cpu->ncli == 0)
1662     cpu->intena = eflags & FL_IF;
1663   cpu->ncli += 1;
1664 }
1665
1666 void
1667 popcli(void)
1668 {
1669   if(readeflags()&FL_IF)
1670     panic("popcli - interruptible");
1671   if(--cpu->ncli < 0)
1672     panic("popcli");
1673   if(cpu->ncli == 0 && cpu->intena)
1674     sti();
1675 }
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
```



```
1700 #include "param.h"
1701 #include "types.h"
1702 #include "defs.h"
1703 #include "x86.h"
1704 #include "memlayout.h"
1705 #include "mmu.h"
1706 #include "proc.h"
1707 #include "elf.h"
1708
1709 extern char data[];  // defined by kernel.ld
1710 pde_t *kpgdir;  // for use in scheduler()
1711
1712 // Set up CPU's kernel segment descriptors.
1713 // Run once on entry on each CPU.
1714 void
1715 seginit(void)
1716 {
1717   struct cpu *c;
1718
1719   // Map "logical" addresses to virtual addresses using identity map.
1720   // Cannot share a CODE descriptor for both kernel and user
1721   // because it would have to have DPL_USR, but the CPU forbids
1722   // an interrupt from CPL=0 to DPL=3.
1723   c = &cpus[cpunum()];
1724   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
1725   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1726   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
1727   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1728
1729   // Map cpu and proc -- these are private per cpu.
1730   c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
1731
1732   lgdt(c->gdt, sizeof(c->gdt));
1733   loadgs(SEG_KCPU << 3);
1734
1735   // Initialize cpu-local storage.
1736   cpu = c;
1737   proc = 0;
1738 }
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
```

```
1750 // Return the address of the PTE in page table pgdir
1751 // that corresponds to virtual address va.  If alloc!=0,
1752 // create any required page table pages.
1753 static pte_t *
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756   pde_t *pde;
1757   pte_t *pgtab;
1758
1759   pde = &pgdir[PDX(va)];
1760   if(*pde & PTE_P){
1761     pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1762   } else {
1763     if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1764       return 0;
1765     // Make sure all those PTE_P bits are zero.
1766     memset(pgtab, 0, PGSIZE);
1767     // The permissions here are overly generous, but they can
1768     // be further restricted by the permissions in the page table
1769     // entries, if necessary.
1770     *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1771   }
1772   return &pgtab[PTX(va)];
1773 }
1774
1775 // Create PTEs for virtual addresses starting at va that refer to
1776 // physical addresses starting at pa. va and size might not
1777 // be page-aligned.
1778 static int
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return -1;
1789     if(*pte & PTE_P)
1790       panic("remap");
1791     *pte = pa | perm | PTE_P;
1792     if(a == last)
1793       break;
1794     a += PGSIZE;
1795     pa += PGSIZE;
1796   }
1797   return 0;
1798 }
1799
```



```
1800 // There is one page table per process, plus one that's used when
1801 // a CPU is not running any process (kpgdir). The kernel uses the
1802 // current process's page table during system calls and interrupts;
1803 // page protection bits prevent user code from using the kernel's
1804 // mappings.
1805 //
1806 // setupkvm() and exec() set up every page table like this:
1807 //
1808 //   0..KERNBASE: user memory (text+data+stack+heap), mapped to
1809 //                phys memory allocated by the kernel
1810 //   KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
1811 //   KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
1812 //                for the kernel's instructions and r/o data
1813 //   data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
1814 //                                     rw data + free physical memory
1815 //   0xfe000000..0: mapped direct (devices such as ioapic)
1816 //
1817 // The kernel allocates physical memory for its heap and for user memory
1818 // between V2P(end) and the end of physical memory (PHYSTOP)
1819 // (directly addressable from end..P2V(PHYSTOP)).
1820
1821 // This table defines the kernel's mappings, which are present in
1822 // every process's page table.
1823 static struct kmap {
1824   void *virt;
1825   uint phys_start;
1826   uint phys_end;
1827   int perm;
1828 } kmap[] = {
1829  { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
1830  { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
1831  { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
1832  { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
1833 };
1834
1835 // Set up kernel part of a page table.
1836 pde_t*
1837 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
1845   if (P2V(PHYSTOP) > (void*)DEVSPACE)
1846     panic("PHYSTOP too high");
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                 (uint)k->phys_start, k->perm) < 0)
```

```
1850       return 0;
1851   return pgdir;
1852 }
1853
1854 // Allocate one page table for the machine for the kernel address
1855 // space for scheduler processes.
1856 void
1857 kvmalloc(void)
1858 {
1859   kpgdir = setupkvm();
1860   switchkvm();
1861 }
1862
1863 // Switch h/w page table register to the kernel-only page table,
1864 // for when no process is running.
1865 void
1866 switchkvm(void)
1867 {
1868   lcr3(V2P(kpgdir));   // switch to the kernel page table
1869 }
1870
1871 // Switch TSS and h/w page table to correspond to process p.
1872 void
1873 switchuvm(struct proc *p)
1874 {
1875   if(p == 0)
1876     panic("switchuvm: no process");
1877   if(p->kstack == 0)
1878     panic("switchuvm: no kstack");
1879   if(p->pgdir == 0)
1880     panic("switchuvm: no pgdir");
1881
1882   pushcli();
1883   cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
1884   cpu->gdt[SEG_TSS].s = 0;
1885   cpu->ts.ss0 = SEG_KDATA << 3;
1886   cpu->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
1887   // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
1888   // forbids I/O instructions (e.g., inb and outb) from user space
1889   cpu->ts.iomb = (ushort) 0xFFFF;
1890   ltr(SEG_TSS << 3);
1891   lcr3(V2P(p->pgdir));  // switch to process's address space
1892   popcli();
1893 }
1894
1895
1896
1897
1898
1899
```



```
1900 // Load the initcode into address 0 of pgdir.
1901 // sz must be less than a page.
1902 void
1903 inituvm(pde_t *pgdir, char *init, uint sz)
1904 {
1905   char *mem;
1906
1907   if(sz >= PGSIZE)
1908     panic("inituvm: more than a page");
1909   mem = kalloc();
1910   memset(mem, 0, PGSIZE);
1911   mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W|PTE_U);
1912   memmove(mem, init, sz);
1913 }
1914
1915 // Load a program segment into pgdir.  addr must be page-aligned
1916 // and the pages from addr to addr+sz must already be mapped.
1917 int
1918 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
1919 {
1920   uint i, pa, n;
1921   pte_t *pte;
1922
1923   if((uint) addr % PGSIZE != 0)
1924     panic("loaduvm: addr must be page aligned");
1925   for(i = 0; i < sz; i += PGSIZE){
1926     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1927       panic("loaduvm: address should exist");
1928     pa = PTE_ADDR(*pte);
1929     if(sz - i < PGSIZE)
1930       n = sz - i;
1931     else
1932       n = PGSIZE;
1933     if(readi(ip, P2V(pa), offset+i, n) != n)
1934       return -1;
1935   }
1936   return 0;
1937 }
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
```

```
1950 // Allocate page tables and physical memory to grow process from oldsz to
1951 // newsz, which need not be page aligned.  Returns new size or 0 on error.
1952 int
1953 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1954 {
1955   char *mem;
1956   uint a;
1957
1958   if(newsz >= KERNBASE)
1959     return 0;
1960   if(newsz < oldsz)
1961     return oldsz;
1962
1963   a = PGROUNDUP(oldsz);
1964   for(; a < newsz; a += PGSIZE){
1965     mem = kalloc();
1966     if(mem == 0){
1967       cprintf("allocuvm out of memory\n");
1968       deallocuvm(pgdir, newsz, oldsz);
1969       return 0;
1970     }
1971     memset(mem, 0, PGSIZE);
1972     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
1973       cprintf("allocuvm out of memory (2)\n");
1974       deallocuvm(pgdir, newsz, oldsz);
1975       kfree(mem);
1976       return 0;
1977     }
1978   }
1979   return newsz;
1980 }
1981
1982 // Deallocate user pages to bring the process size from oldsz to
1983 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
1984 // need to be less than oldsz.  oldsz can be larger than the actual
1985 // process size.  Returns the new process size.
1986 int
1987 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1988 {
1989   pte_t *pte;
1990   uint a, pa;
1991
1992   if(newsz >= oldsz)
1993     return oldsz;
1994
1995   a = PGROUNDUP(newsz);
1996   for(; a  < oldsz; a += PGSIZE){
1997     pte = walkpgdir(pgdir, (char*)a, 0);
1998     if(!pte)
1999       a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
```



```
2000     else if((*pte & PTE_P) != 0){
2001       pa = PTE_ADDR(*pte);
2002       if(pa == 0)
2003         panic("kfree");
2004       char *v = P2V(pa);
2005       kfree(v);
2006       *pte = 0;
2007     }
2008   }
2009   return newsz;
2010 }
2011
2012 // Free a page table and all the physical memory pages
2013 // in the user part.
2014 void
2015 freevm(pde_t *pgdir)
2016 {
2017   uint i;
2018
2019   if(pgdir == 0)
2020     panic("freevm: no pgdir");
2021   deallocuvm(pgdir, KERNBASE, 0);
2022   for(i = 0; i < NPDENTRIES; i++){
2023     if(pgdir[i] & PTE_P){
2024       char * v = P2V(PTE_ADDR(pgdir[i]));
2025       kfree(v);
2026     }
2027   }
2028   kfree((char*)pgdir);
2029 }
2030
2031 // Clear PTE_U on a page. Used to create an inaccessible
2032 // page beneath the user stack.
2033 void
2034 clearpteu(pde_t *pgdir, char *uva)
2035 {
2036   pte_t *pte;
2037
2038   pte = walkpgdir(pgdir, uva, 0);
2039   if(pte == 0)
2040     panic("clearpteu");
2041   *pte &= ~PTE_U;
2042 }
2043
2044
2045
2046
2047
2048
2049
```

```
2050 // Given a parent process's page table, create a copy
2051 // of it for a child.
2052 pde_t*
2053 copyuvm(pde_t *pgdir, uint sz)
2054 {
2055   pde_t *d;
2056   pte_t *pte;
2057   uint pa, i, flags;
2058   char *mem;
2059
2060   if((d = setupkvm()) == 0)
2061     return 0;
2062   for(i = 0; i < sz; i += PGSIZE){
2063     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
2064       panic("copyuvm: pte should exist");
2065     if(!(*pte & PTE_P))
2066       panic("copyuvm: page not present");
2067     pa = PTE_ADDR(*pte);
2068     flags = PTE_FLAGS(*pte);
2069     if((mem = kalloc()) == 0)
2070       goto bad;
2071     memmove(mem, (char*)P2V(pa), PGSIZE);
2072     if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
2073       goto bad;
2074   }
2075   return d;
2076
2077 bad:
2078   freevm(d);
2079   return 0;
2080 }
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
```



```
2100 // Map user virtual address to kernel address.
2101 char*
2102 uva2ka(pde_t *pgdir, char *uva)
2103 {
2104   pte_t *pte;
2105
2106   pte = walkpgdir(pgdir, uva, 0);
2107   if((*pte & PTE_P) == 0)
2108     return 0;
2109   if((*pte & PTE_U) == 0)
2110     return 0;
2111   return (char*)P2V(PTE_ADDR(*pte));
2112 }
2113
2114 // Copy len bytes from p to user address va in page table pgdir.
2115 // Most useful when pgdir is not the current page table.
2116 // uva2ka ensures this only works for PTE_U pages.
2117 int
2118 copyout(pde_t *pgdir, uint va, void *p, uint len)
2119 {
2120   char *buf, *pa0;
2121   uint n, va0;
2122
2123   buf = (char*)p;
2124   while(len > 0){
2125     va0 = (uint)PGROUNDDOWN(va);
2126     pa0 = uva2ka(pgdir, (char*)va0);
2127     if(pa0 == 0)
2128       return -1;
2129     n = PGSIZE - (va - va0);
2130     if(n > len)
2131       n = len;
2132     memmove(pa0 + (va - va0), buf, n);
2133     len -= n;
2134     buf += n;
2135     va = va0 + PGSIZE;
2136   }
2137   return 0;
2138 }
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
```

```
2150 // Blank page.
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
```



```
2200 // Blank page.
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
```

```
2250 // Blank page.
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
```



```
2300 // Per-CPU state
2301 struct cpu {
2302   uchar apicid;                // Local APIC ID
2303   struct context *scheduler;   // swtch() here to enter scheduler
2304   struct taskstate ts;         // Used by x86 to find stack for interrupt
2305   struct segdesc gdt[NSEGS];   // x86 global descriptor table
2306   volatile uint started;       // Has the CPU started?
2307   int ncli;                    // Depth of pushcli nesting.
2308   int intena;                  // Were interrupts enabled before pushcli?
2309
2310   // Cpu-local storage variables; see below
2311   struct cpu *cpu;
2312   struct proc *proc;           // The currently-running process.
2313 };
2314
2315 extern struct cpu cpus[NCPU];
2316 extern int ncpu;
2317
2318 // Per-CPU variables, holding pointers to the
2319 // current cpu and to the current process.
2320 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2321 // and "%gs:4" to refer to proc.  seginit sets up the
2322 // %gs segment register so that %gs refers to the memory
2323 // holding those two variables in the local cpu's struct cpu.
2324 // This is similar to how thread-local variables are implemented
2325 // in thread libraries such as Linux pthreads.
2326 extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
2327 extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc
2328
2329
2330 // Saved registers for kernel context switches.
2331 // Don't need to save all the segment registers (%cs, etc),
2332 // because they are constant across kernel contexts.
2333 // Don't need to save %eax, %ecx, %edx, because the
2334 // x86 convention is that the caller has saved them.
2335 // Contexts are stored at the bottom of the stack they
2336 // describe; the stack pointer is the address of the context.
2337 // The layout of the context matches the layout of the stack in swtch.S
2338 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2339 // but it is on the stack and allocproc() manipulates it.
2340 struct context {
2341   uint edi;
2342   uint esi;
2343   uint ebx;
2344   uint ebp;
2345   uint eip;
2346 };
2347
2348
2349
```

```
2350 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2351
2352 // Per-process state
2353 struct proc {
2354   uint sz;                     // Size of process memory (bytes)
2355   pde_t* pgdir;                // Page table
2356   char *kstack;                // Bottom of kernel stack for this process
2357   enum procstate state;        // Process state
2358   int pid;                     // Process ID
2359   struct proc *parent;         // Parent process
2360   struct trapframe *tf;        // Trap frame for current syscall
2361   struct context *context;     // swtch() here to run process
2362   void *chan;                  // If non-zero, sleeping on chan
2363   int killed;                  // If non-zero, have been killed
2364   struct file *ofile[NOFILE];  // Open files
2365   struct inode *cwd;           // Current directory
2366   char name[16];               // Process name (debugging)
2367 };
2368
2369 // Process memory is laid out contiguously, low addresses first:
2370 //   text
2371 //   original data and bss
2372 //   fixed-size stack
2373 //   expandable heap
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
```



```
2400 #include "types.h"
2401 #include "defs.h"
2402 #include "param.h"
2403 #include "memlayout.h"
2404 #include "mmu.h"
2405 #include "x86.h"
2406 #include "proc.h"
2407 #include "spinlock.h"
2408
2409 struct {
2410   struct spinlock lock;
2411   struct proc proc[NPROC];
2412 } ptable;
2413
2414 static struct proc *initproc;
2415
2416 int nextpid = 1;
2417 extern void forkret(void);
2418 extern void trapret(void);
2419
2420 static void wakeup1(void *chan);
2421
2422 void
2423 pinit(void)
2424 {
2425   initlock(&ptable.lock, "ptable");
2426 }
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
```

```
2450 // Look in the process table for an UNUSED proc.
2451 // If found, change state to EMBRYO and initialize
2452 // state required to run in the kernel.
2453 // Otherwise return 0.
2454 static struct proc*
2455 allocproc(void)
2456 {
2457   struct proc *p;
2458   char *sp;
2459
2460   acquire(&ptable.lock);
2461
2462   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2463     if(p->state == UNUSED)
2464       goto found;
2465
2466   release(&ptable.lock);
2467   return 0;
2468
2469 found:
2470   p->state = EMBRYO;
2471   p->pid = nextpid++;
2472
2473   release(&ptable.lock);
2474
2475   // Allocate kernel stack.
2476   if((p->kstack = kalloc()) == 0){
2477     p->state = UNUSED;
2478     return 0;
2479   }
2480   sp = p->kstack + KSTACKSIZE;
2481
2482   // Leave room for trap frame.
2483   sp -= sizeof *p->tf;
2484   p->tf = (struct trapframe*)sp;
2485
2486   // Set up new context to start executing at forkret,
2487   // which returns to trapret.
2488   sp -= 4;
2489   *(uint*)sp = (uint)trapret;
2490
2491   sp -= sizeof *p->context;
2492   p->context = (struct context*)sp;
2493   memset(p->context, 0, sizeof *p->context);
2494   p->context->eip = (uint)forkret;
2495
2496   return p;
2497 }
2498
2499
```



```
2500 // Set up first user process.
2501 void
2502 userinit(void)
2503 {
2504   struct proc *p;
2505   extern char _binary_initcode_start[], _binary_initcode_size[];
2506
2507   p = allocproc();
2508
2509   initproc = p;
2510   if((p->pgdir = setupkvm()) == 0)
2511     panic("userinit: out of memory?");
2512   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2513   p->sz = PGSIZE;
2514   memset(p->tf, 0, sizeof(*p->tf));
2515   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2516   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2517   p->tf->es = p->tf->ds;
2518   p->tf->ss = p->tf->ds;
2519   p->tf->eflags = FL_IF;
2520   p->tf->esp = PGSIZE;
2521   p->tf->eip = 0;  // beginning of initcode.S
2522
2523   safestrcpy(p->name, "initcode", sizeof(p->name));
2524   p->cwd = namei("/");
2525
2526   // this assignment to p->state lets other cores
2527   // run this process. the acquire forces the above
2528   // writes to be visible, and the lock is also needed
2529   // because the assignment might not be atomic.
2530   acquire(&ptable.lock);
2531
2532   p->state = RUNNABLE;
2533
2534   release(&ptable.lock);
2535 }
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
```

```
2550 // Grow current process's memory by n bytes.
2551 // Return 0 on success, -1 on failure.
2552 int
2553 growproc(int n)
2554 {
2555   uint sz;
2556
2557   sz = proc->sz;
2558   if(n > 0){
2559     if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2560       return -1;
2561   } else if(n < 0){
2562     if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2563       return -1;
2564   }
2565   proc->sz = sz;
2566   switchuvm(proc);
2567   return 0;
2568 }
2569
2570 // Create a new process copying p as the parent.
2571 // Sets up stack to return as if from system call.
2572 // Caller must set state of returned proc to RUNNABLE.
2573 int
2574 fork(void)
2575 {
2576   int i, pid;
2577   struct proc *np;
2578
2579   // Allocate process.
2580   if((np = allocproc()) == 0){
2581     return -1;
2582   }
2583
2584   // Copy process state from p.
2585   if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2586     kfree(np->kstack);
2587     np->kstack = 0;
2588     np->state = UNUSED;
2589     return -1;
2590   }
2591   np->sz = proc->sz;
2592   np->parent = proc;
2593   *np->tf = *proc->tf;
2594
2595   // Clear %eax so that fork returns 0 in the child.
2596   np->tf->eax = 0;
2597
2598
2599
```



```
2600   for(i = 0; i < NOFILE; i++)
2601     if(proc->ofile[i])
2602       np->ofile[i] = filedup(proc->ofile[i]);
2603   np->cwd = idup(proc->cwd);
2604
2605   safestrcpy(np->name, proc->name, sizeof(proc->name));
2606
2607   pid = np->pid;
2608
2609   acquire(&ptable.lock);
2610
2611   np->state = RUNNABLE;
2612
2613   release(&ptable.lock);
2614
2615   return pid;
2616 }
2617
2618 // Exit the current process.  Does not return.
2619 // An exited process remains in the zombie state
2620 // until its parent calls wait() to find out it exited.
2621 void
2622 exit(void)
2623 {
2624   struct proc *p;
2625   int fd;
2626
2627   if(proc == initproc)
2628     panic("init exiting");
2629
2630   // Close all open files.
2631   for(fd = 0; fd < NOFILE; fd++){
2632     if(proc->ofile[fd]){
2633       fileclose(proc->ofile[fd]);
2634       proc->ofile[fd] = 0;
2635     }
2636   }
2637
2638   begin_op();
2639   iput(proc->cwd);
2640   end_op();
2641   proc->cwd = 0;
2642
2643   acquire(&ptable.lock);
2644
2645   // Parent might be sleeping in wait().
2646   wakeup1(proc->parent);
2647
2648
2649
```

```
2650   // Pass abandoned children to init.
2651   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2652     if(p->parent == proc){
2653       p->parent = initproc;
2654       if(p->state == ZOMBIE)
2655         wakeup1(initproc);
2656     }
2657   }
2658
2659   // Jump into the scheduler, never to return.
2660   proc->state = ZOMBIE;
2661   sched();
2662   panic("zombie exit");
2663 }
2664
2665 // Wait for a child process to exit and return its pid.
2666 // Return -1 if this process has no children.
2667 int
2668 wait(void)
2669 {
2670   struct proc *p;
2671   int havekids, pid;
2672
2673   acquire(&ptable.lock);
2674   for(;;){
2675     // Scan through table looking for exited children.
2676     havekids = 0;
2677     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2678       if(p->parent != proc)
2679         continue;
2680       havekids = 1;
2681       if(p->state == ZOMBIE){
2682         // Found one.
2683         pid = p->pid;
2684         kfree(p->kstack);
2685         p->kstack = 0;
2686         freevm(p->pgdir);
2687         p->pid = 0;
2688         p->parent = 0;
2689         p->name[0] = 0;
2690         p->killed = 0;
2691         p->state = UNUSED;
2692         release(&ptable.lock);
2693         return pid;
2694       }
2695     }
2696
2697
2698
2699
```



```
2700     // No point waiting if we don't have any children.
2701     if(!havekids || proc->killed){
2702       release(&ptable.lock);
2703       return -1;
2704     }
2705
2706     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2707     sleep(proc, &ptable.lock);
2708   }
2709 }
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
```

```
2750 // Per-CPU process scheduler.
2751 // Each CPU calls scheduler() after setting itself up.
2752 // Scheduler never returns.  It loops, doing:
2753 //  - choose a process to run
2754 //  - swtch to start running that process
2755 //  - eventually that process transfers control
2756 //      via swtch back to the scheduler.
2757 void
2758 scheduler(void)
2759 {
2760   struct proc *p;
2761
2762   for(;;){
2763     // Enable interrupts on this processor.
2764     sti();
2765
2766     // Loop over process table looking for process to run.
2767     acquire(&ptable.lock);
2768     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2769       if(p->state != RUNNABLE)
2770         continue;
2771
2772       // Switch to chosen process.  It is the process's job
2773       // to release ptable.lock and then reacquire it
2774       // before jumping back to us.
2775       proc = p;
2776       switchuvm(p);
2777       p->state = RUNNING;
2778       swtch(&cpu->scheduler, p->context);
2779       switchkvm();
2780
2781       // Process is done running for now.
2782       // It should have changed its p->state before coming back.
2783       proc = 0;
2784     }
2785     release(&ptable.lock);
2786
2787   }
2788 }
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
```



```
2800 // Enter scheduler.  Must hold only ptable.lock
2801 // and have changed proc->state. Saves and restores
2802 // intena because intena is a property of this
2803 // kernel thread, not this CPU. It should
2804 // be proc->intena and proc->ncli, but that would
2805 // break in the few places where a lock is held but
2806 // there's no process.
2807 void
2808 sched(void)
2809 {
2810   int intena;
2811
2812   if(!holding(&ptable.lock))
2813     panic("sched ptable.lock");
2814   if(cpu->ncli != 1)
2815     panic("sched locks");
2816   if(proc->state == RUNNING)
2817     panic("sched running");
2818   if(readeflags()&FL_IF)
2819     panic("sched interruptible");
2820   intena = cpu->intena;
2821   swtch(&proc->context, cpu->scheduler);
2822   cpu->intena = intena;
2823 }
2824
2825 // Give up the CPU for one scheduling round.
2826 void
2827 yield(void)
2828 {
2829   acquire(&ptable.lock);
2830   proc->state = RUNNABLE;
2831   sched();
2832   release(&ptable.lock);
2833 }
2834
2835 // A fork child's very first scheduling by scheduler()
2836 // will swtch here.  "Return" to user space.
2837 void
2838 forkret(void)
2839 {
2840   static int first = 1;
2841   // Still holding ptable.lock from scheduler.
2842   release(&ptable.lock);
2843
2844   if (first) {
2845     // Some initialization functions must be run in the context
2846     // of a regular process (e.g., they call sleep), and thus cannot
2847     // be run from main().
2848     first = 0;
2849     iinit(ROOTDEV);
```

```
2850     initlog(ROOTDEV);
2851   }
2852
2853   // Return to "caller", actually trapret (see allocproc).
2854 }
2855
2856 // Atomically release lock and sleep on chan.
2857 // Reacquires lock when awakened.
2858 void
2859 sleep(void *chan, struct spinlock *lk)
2860 {
2861   if(proc == 0)
2862     panic("sleep");
2863
2864   if(lk == 0)
2865     panic("sleep without lk");
2866
2867   // Must acquire ptable.lock in order to
2868   // change p->state and then call sched.
2869   // Once we hold ptable.lock, we can be
2870   // guaranteed that we won't miss any wakeup
2871   // (wakeup runs with ptable.lock locked),
2872   // so it's okay to release lk.
2873   if(lk != &ptable.lock){
2874     acquire(&ptable.lock);
2875     release(lk);
2876   }
2877
2878   // Go to sleep.
2879   proc->chan = chan;
2880   proc->state = SLEEPING;
2881   sched();
2882
2883   // Tidy up.
2884   proc->chan = 0;
2885
2886   // Reacquire original lock.
2887   if(lk != &ptable.lock){
2888     release(&ptable.lock);
2889     acquire(lk);
2890   }
2891 }
2892
2893
2894
2895
2896
2897
2898
2899
```



```
2900 // Wake up all processes sleeping on chan.
2901 // The ptable lock must be held.
2902 static void
2903 wakeup1(void *chan)
2904 {
2905   struct proc *p;
2906
2907   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2908     if(p->state == SLEEPING && p->chan == chan)
2909       p->state = RUNNABLE;
2910 }
2911
2912 // Wake up all processes sleeping on chan.
2913 void
2914 wakeup(void *chan)
2915 {
2916   acquire(&ptable.lock);
2917   wakeup1(chan);
2918   release(&ptable.lock);
2919 }
2920
2921 // Kill the process with the given pid.
2922 // Process won't exit until it returns
2923 // to user space (see trap in trap.c).
2924 int
2925 kill(int pid)
2926 {
2927   struct proc *p;
2928
2929   acquire(&ptable.lock);
2930   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2931     if(p->pid == pid){
2932       p->killed = 1;
2933       // Wake process from sleep if necessary.
2934       if(p->state == SLEEPING)
2935         p->state = RUNNABLE;
2936       release(&ptable.lock);
2937       return 0;
2938     }
2939   }
2940   release(&ptable.lock);
2941   return -1;
2942 }
2943
2944
2945
2946
2947
2948
2949
```

```
2950 // Print a process listing to console.  For debugging.
2951 // Runs when user types ^P on console.
2952 // No lock to avoid wedging a stuck machine further.
2953 void
2954 procdump(void)
2955 {
2956   static char *states[] = {
2957   [UNUSED]    "unused",
2958   [EMBRYO]    "embryo",
2959   [SLEEPING]  "sleep ",
2960   [RUNNABLE]  "runble",
2961   [RUNNING]   "run   ",
2962   [ZOMBIE]    "zombie"
2963   };
2964   int i;
2965   struct proc *p;
2966   char *state;
2967   uint pc[10];
2968
2969   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2970     if(p->state == UNUSED)
2971       continue;
2972     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
2973       state = states[p->state];
2974     else
2975       state = "???";
2976     cprintf("%d %s %s", p->pid, state, p->name);
2977     if(p->state == SLEEPING){
2978       getcallerpcs((uint*)p->context->ebp+2, pc);
2979       for(i=0; i<10 && pc[i] != 0; i++)
2980         cprintf(" %p", pc[i]);
2981     }
2982     cprintf("\n");
2983   }
2984 }
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
```



```
3000 # Context switch
3001 #
3002 #   void swtch(struct context **old, struct context *new);
3003 #
3004 # Save current register context in old
3005 # and then load register context from new.
3006
3007 .globl swtch
3008 swtch:
3009   movl 4(%esp), %eax
3010   movl 8(%esp), %edx
3011
3012   # Save old callee-save registers
3013   pushl %ebp
3014   pushl %ebx
3015   pushl %esi
3016   pushl %edi
3017
3018   # Switch stacks
3019   movl %esp, (%eax)
3020   movl %edx, %esp
3021
3022   # Load new callee-save registers
3023   popl %edi
3024   popl %esi
3025   popl %ebx
3026   popl %ebp
3027   ret
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
```

```
3050 // Physical memory allocator, intended to allocate
3051 // memory for user processes, kernel stacks, page table pages,
3052 // and pipe buffers. Allocates 4096-byte pages.
3053
3054 #include "types.h"
3055 #include "defs.h"
3056 #include "param.h"
3057 #include "memlayout.h"
3058 #include "mmu.h"
3059 #include "spinlock.h"
3060
3061 void freerange(void *vstart, void *vend);
3062 extern char end[]; // first address after kernel loaded from ELF file
3063
3064 struct run {
3065   struct run *next;
3066 };
3067
3068 struct {
3069   struct spinlock lock;
3070   int use_lock;
3071   struct run *freelist;
3072 } kmem;
3073
3074 // Initialization happens in two phases.
3075 // 1. main() calls kinit1() while still using entrypgdir to place just
3076 // the pages mapped by entrypgdir on free list.
3077 // 2. main() calls kinit2() with the rest of the physical pages
3078 // after installing a full page table that maps them on all cores.
3079 void
3080 kinit1(void *vstart, void *vend)
3081 {
3082   initlock(&kmem.lock, "kmem");
3083   kmem.use_lock = 0;
3084   freerange(vstart, vend);
3085 }
3086
3087 void
3088 kinit2(void *vstart, void *vend)
3089 {
3090   freerange(vstart, vend);
3091   kmem.use_lock = 1;
3092 }
3093
3094
3095
3096
3097
3098
3099
```



```
3100 void
3101 freerange(void *vstart, void *vend)
3102 {
3103   char *p;
3104   p = (char*)PGROUNDUP((uint)vstart);
3105   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3106     kfree(p);
3107 }
3108
3109
3110 // Free the page of physical memory pointed at by v,
3111 // which normally should have been returned by a
3112 // call to kalloc().  (The exception is when
3113 // initializing the allocator; see kinit above.)
3114 void
3115 kfree(char *v)
3116 {
3117   struct run *r;
3118
3119   if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
3120     panic("kfree");
3121
3122   // Fill with junk to catch dangling refs.
3123   memset(v, 1, PGSIZE);
3124
3125   if(kmem.use_lock)
3126     acquire(&kmem.lock);
3127   r = (struct run*)v;
3128   r->next = kmem.freelist;
3129   kmem.freelist = r;
3130   if(kmem.use_lock)
3131     release(&kmem.lock);
3132 }
3133
3134 // Allocate one 4096-byte page of physical memory.
3135 // Returns a pointer that the kernel can use.
3136 // Returns 0 if the memory cannot be allocated.
3137 char*
3138 kalloc(void)
3139 {
3140   struct run *r;
3141
3142   if(kmem.use_lock)
3143     acquire(&kmem.lock);
3144   r = kmem.freelist;
3145   if(r)
3146     kmem.freelist = r->next;
3147   if(kmem.use_lock)
3148     release(&kmem.lock);
3149   return (char*)r;
```

```
3150 }
3151
3152
3153
3154
3155
3156
3157
3158
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3180
3181
3182
3183
3184
3185
3186
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
```



```
3200 // x86 trap and interrupt constants.
3201
3202 // Processor-defined:
3203 #define T_DIVIDE         0      // divide error
3204 #define T_DEBUG          1      // debug exception
3205 #define T_NMI            2      // non-maskable interrupt
3206 #define T_BRKPT          3      // breakpoint
3207 #define T_OFLOW          4      // overflow
3208 #define T_BOUND          5      // bounds check
3209 #define T_ILLOP          6      // illegal opcode
3210 #define T_DEVICE         7      // device not available
3211 #define T_DBLFLT         8      // double fault
3212 // #define T_COPROC      9      // reserved (not used since 486)
3213 #define T_TSS           10      // invalid task switch segment
3214 #define T_SEGNP         11      // segment not present
3215 #define T_STACK         12      // stack exception
3216 #define T_GPFLT         13      // general protection fault
3217 #define T_PGFLT         14      // page fault
3218 // #define T_RES        15      // reserved
3219 #define T_FPERR         16      // floating point error
3220 #define T_ALIGN         17      // aligment check
3221 #define T_MCHK          18      // machine check
3222 #define T_SIMDERR       19      // SIMD floating point error
3223
3224 // These are arbitrarily chosen, but with care not to overlap
3225 // processor defined exceptions or interrupt vectors.
3226 #define T_SYSCALL       64      // system call
3227 #define T_DEFAULT      500      // catchall
3228
3229 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
3230
3231 #define IRQ_TIMER        0
3232 #define IRQ_KBD          1
3233 #define IRQ_COM1         4
3234 #define IRQ_IDE         14
3235 #define IRQ_ERROR       19
3236 #define IRQ_SPURIOUS    31
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
```

```
#!/usr/bin/perl -w

# Generate vectors.S, the trap/interrupt entry points.
# There has to be one entry point per interrupt number
# since otherwise there's no way for trap() to discover
# the interrupt number.

print "# generated by vectors.pl - do not edit\n";
print "# handlers\n";
print ".globl alltraps\n";
for(my $i = 0; $i < 256; $i++){
    print ".globl vector$i\n";
    print "vector$i:\n";
    if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
        print "  pushl \$0\n";
    }
    print "  pushl \$$i\n";
    print "  jmp alltraps\n";
}

print "\n# vector table\n";
print ".data\n";
print ".globl vectors\n";
print "vectors:\n";
for(my $i = 0; $i < 256; $i++){
    print "  .long vector$i\n";
}

# sample output:
#   # handlers
#   .globl alltraps
#   .globl vector0
#   vector0:
#     pushl $0
#     pushl $0
#     jmp alltraps
#   ...
#
#   # vector table
#   .data
#   .globl vectors
#   vectors:
#     .long vector0
#     .long vector1
#     .long vector2
#   ...
```



```
#include "mmu.h"

  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data and per-cpu segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es
  movw $(SEG_KCPU<<3), %ax
  movw %ax, %fs
  movw %ax, %gs

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

```
3350 #include "types.h"
3351 #include "defs.h"
3352 #include "param.h"
3353 #include "memlayout.h"
3354 #include "mmu.h"
3355 #include "proc.h"
3356 #include "x86.h"
3357 #include "traps.h"
3358 #include "spinlock.h"
3359
3360 // Interrupt descriptor table (shared by all CPUs).
3361 struct gatedesc idt[256];
3362 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
3363 struct spinlock tickslock;
3364 uint ticks;
3365
3366 void
3367 tvinit(void)
3368 {
3369   int i;
3370
3371   for(i = 0; i < 256; i++)
3372     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3373   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3374
3375   initlock(&tickslock, "time");
3376 }
3377
3378 void
3379 idtinit(void)
3380 {
3381   lidt(idt, sizeof(idt));
3382 }
3383
3384
3385
3386
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399
```



```
3400 void
3401 trap(struct trapframe *tf)
3402 {
3403   if(tf->trapno == T_SYSCALL){
3404     if(proc->killed)
3405       exit();
3406     proc->tf = tf;
3407     syscall();
3408     if(proc->killed)
3409       exit();
3410     return;
3411   }
3412
3413   switch(tf->trapno){
3414   case T_IRQ0 + IRQ_TIMER:
3415     if(cpunum() == 0){
3416       acquire(&tickslock);
3417       ticks++;
3418       wakeup(&ticks);
3419       release(&tickslock);
3420     }
3421     lapiceoi();
3422     break;
3423   case T_IRQ0 + IRQ_IDE:
3424     ideintr();
3425     lapiceoi();
3426     break;
3427   case T_IRQ0 + IRQ_IDE+1:
3428     // Bochs generates spurious IDE1 interrupts.
3429     break;
3430   case T_IRQ0 + IRQ_KBD:
3431     kbdintr();
3432     lapiceoi();
3433     break;
3434   case T_IRQ0 + IRQ_COM1:
3435     uartintr();
3436     lapiceoi();
3437     break;
3438   case T_IRQ0 + 7:
3439   case T_IRQ0 + IRQ_SPURIOUS:
3440     cprintf("cpu%d: spurious interrupt at %x:%x\n",
3441             cpunum(), tf->cs, tf->eip);
3442     lapiceoi();
3443     break;
3444
3445
3446
3447
3448
3449
```

```
3450   default:
3451     if(proc == 0 || (tf->cs&3) == 0){
3452       // In kernel, it must be our mistake.
3453       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3454               tf->trapno, cpunum(), tf->eip, rcr2());
3455       panic("trap");
3456     }
3457     // In user space, assume process misbehaved.
3458     cprintf("pid %d %s: trap %d err %d on cpu %d "
3459             "eip 0x%x addr 0x%x--kill proc\n",
3460             proc->pid, proc->name, tf->trapno, tf->err, cpunum(), tf->eip,
3461             rcr2());
3462     proc->killed = 1;
3463   }
3464
3465   // Force process exit if it has been killed and is in user space.
3466   // (If it is still executing in the kernel, let it keep running
3467   // until it gets to the regular system call return.)
3468   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3469     exit();
3470
3471   // Force process to give up CPU on clock tick.
3472   // If interrupts were on while locks held, would need to check nlock.
3473   if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3474     yield();
3475
3476   // Check if the process has been killed since we yielded
3477   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3478     exit();
3479 }
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
```



```
3500 // System call numbers
3501 #define SYS_fork    1
3502 #define SYS_exit    2
3503 #define SYS_wait    3
3504 #define SYS_pipe    4
3505 #define SYS_read    5
3506 #define SYS_kill    6
3507 #define SYS_exec    7
3508 #define SYS_fstat   8
3509 #define SYS_chdir   9
3510 #define SYS_dup    10
3511 #define SYS_getpid 11
3512 #define SYS_sbrk   12
3513 #define SYS_sleep  13
3514 #define SYS_uptime 14
3515 #define SYS_open   15
3516 #define SYS_write  16
3517 #define SYS_mknod  17
3518 #define SYS_unlink 18
3519 #define SYS_link   19
3520 #define SYS_mkdir  20
3521 #define SYS_close  21
3522
3523
3524
3525
3526
3527
3528
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
```

```
3550 #include "types.h"
3551 #include "defs.h"
3552 #include "param.h"
3553 #include "memlayout.h"
3554 #include "mmu.h"
3555 #include "proc.h"
3556 #include "x86.h"
3557 #include "syscall.h"
3558
3559 // User code makes a system call with INT T_SYSCALL.
3560 // System call number in %eax.
3561 // Arguments on the stack, from the user call to the C
3562 // library system call function. The saved user %esp points
3563 // to a saved program counter, and then the first argument.
3564
3565 // Fetch the int at addr from the current process.
3566 int
3567 fetchint(uint addr, int *ip)
3568 {
3569   if(addr >= proc->sz || addr+4 > proc->sz)
3570     return -1;
3571   *ip = *(int*)(addr);
3572   return 0;
3573 }
3574
3575 // Fetch the nul-terminated string at addr from the current process.
3576 // Doesn't actually copy the string - just sets *pp to point at it.
3577 // Returns length of string, not including nul.
3578 int
3579 fetchstr(uint addr, char **pp)
3580 {
3581   char *s, *ep;
3582
3583   if(addr >= proc->sz)
3584     return -1;
3585   *pp = (char*)addr;
3586   ep = (char*)proc->sz;
3587   for(s = *pp; s < ep; s++)
3588     if(*s == 0)
3589       return s - *pp;
3590   return -1;
3591 }
3592
3593 // Fetch the nth 32-bit system call argument.
3594 int
3595 argint(int n, int *ip)
3596 {
3597   return fetchint(proc->tf->esp + 4 + 4*n, ip);
3598 }
3599
```



```
3600 // Fetch the nth word-sized system call argument as a pointer
3601 // to a block of memory of size bytes.  Check that the pointer
3602 // lies within the process address space.
3603 int
3604 argptr(int n, char **pp, int size)
3605 {
3606   int i;
3607
3608   if(argint(n, &i) < 0)
3609     return -1;
3610   if(size < 0 || (uint)i >= proc->sz || (uint)i+size > proc->sz)
3611     return -1;
3612   *pp = (char*)i;
3613   return 0;
3614 }
3615
3616 // Fetch the nth word-sized system call argument as a string pointer.
3617 // Check that the pointer is valid and the string is nul-terminated.
3618 // (There is no shared writable memory, so the string can't change
3619 // between this check and being used by the kernel.)
3620 int
3621 argstr(int n, char **pp)
3622 {
3623   int addr;
3624   if(argint(n, &addr) < 0)
3625     return -1;
3626   return fetchstr(addr, pp);
3627 }
3628
3629 extern int sys_chdir(void);
3630 extern int sys_close(void);
3631 extern int sys_dup(void);
3632 extern int sys_exec(void);
3633 extern int sys_exit(void);
3634 extern int sys_fork(void);
3635 extern int sys_fstat(void);
3636 extern int sys_getpid(void);
3637 extern int sys_kill(void);
3638 extern int sys_link(void);
3639 extern int sys_mkdir(void);
3640 extern int sys_mknod(void);
3641 extern int sys_open(void);
3642 extern int sys_pipe(void);
3643 extern int sys_read(void);
3644 extern int sys_sbrk(void);
3645 extern int sys_sleep(void);
3646 extern int sys_unlink(void);
3647 extern int sys_wait(void);
3648 extern int sys_write(void);
3649 extern int sys_uptime(void);
```

```
3650 static int (*syscalls[])(void) = {
3651 [SYS_fork]    sys_fork,
3652 [SYS_exit]    sys_exit,
3653 [SYS_wait]    sys_wait,
3654 [SYS_pipe]    sys_pipe,
3655 [SYS_read]    sys_read,
3656 [SYS_kill]    sys_kill,
3657 [SYS_exec]    sys_exec,
3658 [SYS_fstat]   sys_fstat,
3659 [SYS_chdir]   sys_chdir,
3660 [SYS_dup]     sys_dup,
3661 [SYS_getpid]  sys_getpid,
3662 [SYS_sbrk]    sys_sbrk,
3663 [SYS_sleep]   sys_sleep,
3664 [SYS_uptime]  sys_uptime,
3665 [SYS_open]    sys_open,
3666 [SYS_write]   sys_write,
3667 [SYS_mknod]   sys_mknod,
3668 [SYS_unlink]  sys_unlink,
3669 [SYS_link]    sys_link,
3670 [SYS_mkdir]   sys_mkdir,
3671 [SYS_close]   sys_close,
3672 };
3673
3674 void
3675 syscall(void)
3676 {
3677   int num;
3678
3679   num = proc->tf->eax;
3680   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3681     proc->tf->eax = syscalls[num]();
3682   } else {
3683     cprintf("%d %s: unknown sys call %d\n",
3684             proc->pid, proc->name, num);
3685     proc->tf->eax = -1;
3686   }
3687 }
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699
```



```
3700 #include "types.h"
3701 #include "x86.h"
3702 #include "defs.h"
3703 #include "date.h"
3704 #include "param.h"
3705 #include "memlayout.h"
3706 #include "mmu.h"
3707 #include "proc.h"
3708
3709 int
3710 sys_fork(void)
3711 {
3712   return fork();
3713 }
3714
3715 int
3716 sys_exit(void)
3717 {
3718   exit();
3719   return 0;  // not reached
3720 }
3721
3722 int
3723 sys_wait(void)
3724 {
3725   return wait();
3726 }
3727
3728 int
3729 sys_kill(void)
3730 {
3731   int pid;
3732
3733   if(argint(0, &pid) < 0)
3734     return -1;
3735   return kill(pid);
3736 }
3737
3738 int
3739 sys_getpid(void)
3740 {
3741   return proc->pid;
3742 }
3743
3744
3745
3746
3747
3748
3749
```

```
3750 int
3751 sys_sbrk(void)
3752 {
3753   int addr;
3754   int n;
3755
3756   if(argint(0, &n) < 0)
3757     return -1;
3758   addr = proc->sz;
3759   if(growproc(n) < 0)
3760     return -1;
3761   return addr;
3762 }
3763
3764 int
3765 sys_sleep(void)
3766 {
3767   int n;
3768   uint ticks0;
3769
3770   if(argint(0, &n) < 0)
3771     return -1;
3772   acquire(&tickslock);
3773   ticks0 = ticks;
3774   while(ticks - ticks0 < n){
3775     if(proc->killed){
3776       release(&tickslock);
3777       return -1;
3778     }
3779     sleep(&ticks, &tickslock);
3780   }
3781   release(&tickslock);
3782   return 0;
3783 }
3784
3785 // return how many clock tick interrupts have occurred
3786 // since start.
3787 int
3788 sys_uptime(void)
3789 {
3790   uint xticks;
3791
3792   acquire(&tickslock);
3793   xticks = ticks;
3794   release(&tickslock);
3795   return xticks;
3796 }
3797
3798
3799
```



```
3800 struct buf {
3801   int flags;
3802   uint dev;
3803   uint blockno;
3804   struct sleeplock lock;
3805   uint refcnt;
3806   struct buf *prev; // LRU cache list
3807   struct buf *next;
3808   struct buf *qnext; // disk queue
3809   uchar data[BSIZE];
3810 };
3811 #define B_VALID 0x2  // buffer has been read from disk
3812 #define B_DIRTY 0x4  // buffer needs to be written to disk
3813
3814
3815
3816
3817
3818
3819
3820
3821
3822
3823
3824
3825
3826
3827
3828
3829
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
```

```
3850 // Long-term locks for processes
3851 struct sleeplock {
3852   uint locked;       // Is the lock held?
3853   struct spinlock lk; // spinlock protecting this sleep lock
3854
3855   // For debugging:
3856   char *name;        // Name of lock.
3857   int pid;           // Process holding lock
3858 };
3859
3860
3861
3862
3863
3864
3865
3866
3867
3868
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
```



```
3900 #define O_RDONLY  0x000
3901 #define O_WRONLY  0x001
3902 #define O_RDWR    0x002
3903 #define O_CREATE  0x200
3904
3905
3906
3907
3908
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
```

```
3950 #define T_DIR  1   // Directory
3951 #define T_FILE 2   // File
3952 #define T_DEV  3   // Device
3953
3954 struct stat {
3955   short type;  // Type of file
3956   int dev;     // File system's disk device
3957   uint ino;    // Inode number
3958   short nlink; // Number of links to file
3959   uint size;   // Size of file in bytes
3960 };
3961
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
```



```
4000 // On-disk file system format.
4001 // Both the kernel and user programs use this header file.
4002
4003
4004 #define ROOTINO 1  // root i-number
4005 #define BSIZE 512  // block size
4006
4007 // Disk layout:
4008 // [ boot block | super block | log | inode blocks |
4009 //                                          free bit map | data blocks]
4010 //
4011 // mkfs computes the super block and builds an initial file system. The
4012 // super block describes the disk layout:
4013 struct superblock {
4014   uint size;         // Size of file system image (blocks)
4015   uint nblocks;      // Number of data blocks
4016   uint ninodes;      // Number of inodes.
4017   uint nlog;         // Number of log blocks
4018   uint logstart;     // Block number of first log block
4019   uint inodestart;   // Block number of first inode block
4020   uint bmapstart;    // Block number of first free map block
4021 };
4022
4023 #define NDIRECT 12
4024 #define NINDIRECT (BSIZE / sizeof(uint))
4025 #define MAXFILE (NDIRECT + NINDIRECT)
4026
4027 // On-disk inode structure
4028 struct dinode {
4029   short type;           // File type
4030   short major;          // Major device number (T_DEV only)
4031   short minor;          // Minor device number (T_DEV only)
4032   short nlink;          // Number of links to inode in file system
4033   uint size;            // Size of file (bytes)
4034   uint addrs[NDIRECT+1];   // Data block addresses
4035 };
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
```

```
4050 // Inodes per block.
4051 #define IPB           (BSIZE / sizeof(struct dinode))
4052
4053 // Block containing inode i
4054 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
4055
4056 // Bitmap bits per block
4057 #define BPB           (BSIZE*8)
4058
4059 // Block of free map containing bit for block b
4060 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
4061
4062 // Directory is a file containing a sequence of dirent structures.
4063 #define DIRSIZ 14
4064
4065 struct dirent {
4066   ushort inum;
4067   char name[DIRSIZ];
4068 };
4069
4070
4071
4072
4073
4074
4075
4076
4077
4078
4079
4080
4081
4082
4083
4084
4085
4086
4087
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
```



```
4100 struct file {
4101   enum { FD_NONE, FD_PIPE, FD_INODE } type;
4102   int ref; // reference count
4103   char readable;
4104   char writable;
4105   struct pipe *pipe;
4106   struct inode *ip;
4107   uint off;
4108 };
4109
4110
4111 // in-memory copy of an inode
4112 struct inode {
4113   uint dev;           // Device number
4114   uint inum;          // Inode number
4115   int ref;            // Reference count
4116   struct sleeplock lock;
4117   int flags;          // I_VALID
4118
4119   short type;         // copy of disk inode
4120   short major;
4121   short minor;
4122   short nlink;
4123   uint size;
4124   uint addrs[NDIRECT+1];
4125 };
4126 #define I_VALID 0x2
4127
4128 // table mapping major device number to
4129 // device functions
4130 struct devsw {
4131   int (*read)(struct inode*, char*, int);
4132   int (*write)(struct inode*, char*, int);
4133 };
4134
4135 extern struct devsw devsw[];
4136
4137 #define CONSOLE 1
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```

```
4150 // Blank page.
4151
4152
4153
4154
4155
4156
4157
4158
4159
4160
4161
4162
4163
4164
4165
4166
4167
4168
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
```



```
4200 // Simple PIO-based (non-DMA) IDE driver code.
4201
4202 #include "types.h"
4203 #include "defs.h"
4204 #include "param.h"
4205 #include "memlayout.h"
4206 #include "mmu.h"
4207 #include "proc.h"
4208 #include "x86.h"
4209 #include "traps.h"
4210 #include "spinlock.h"
4211 #include "sleeplock.h"
4212 #include "fs.h"
4213 #include "buf.h"
4214
4215 #define SECTOR_SIZE   512
4216 #define IDE_BSY       0x80
4217 #define IDE_DRDY      0x40
4218 #define IDE_DF        0x20
4219 #define IDE_ERR       0x01
4220
4221 #define IDE_CMD_READ  0x20
4222 #define IDE_CMD_WRITE 0x30
4223 #define IDE_CMD_RDMUL 0xc4
4224 #define IDE_CMD_WRMUL 0xc5
4225
4226 // idequeue points to the buf now being read/written to the disk.
4227 // idequeue->qnext points to the next buf to be processed.
4228 // You must hold idelock while manipulating queue.
4229
4230 static struct spinlock idelock;
4231 static struct buf *idequeue;
4232
4233 static int havedisk1;
4234 static void idestart(struct buf*);
4235
4236 // Wait for IDE disk to become ready.
4237 static int
4238 idewait(int checkerr)
4239 {
4240   int r;
4241
4242   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4243     ;
4244   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4245     return -1;
4246   return 0;
4247 }
4248
4249
```

```
4250 void
4251 ideinit(void)
4252 {
4253   int i;
4254
4255   initlock(&idelock, "ide");
4256   picenable(IRQ_IDE);
4257   ioapicenable(IRQ_IDE, ncpu - 1);
4258   idewait(0);
4259
4260   // Check if disk 1 is present
4261   outb(0x1f6, 0xe0 | (1<<4));
4262   for(i=0; i<1000; i++){
4263     if(inb(0x1f7) != 0){
4264       havedisk1 = 1;
4265       break;
4266     }
4267   }
4268
4269   // Switch back to disk 0.
4270   outb(0x1f6, 0xe0 | (0<<4));
4271 }
4272
4273 // Start the request for b.  Caller must hold idelock.
4274 static void
4275 idestart(struct buf *b)
4276 {
4277   if(b == 0)
4278     panic("idestart");
4279   if(b->blockno >= FSSIZE)
4280     panic("incorrect blockno");
4281   int sector_per_block =  BSIZE/SECTOR_SIZE;
4282   int sector = b->blockno * sector_per_block;
4283   int read_cmd = (sector_per_block == 1) ? IDE_CMD_READ :  IDE_CMD_RDMUL;
4284   int write_cmd = (sector_per_block == 1) ? IDE_CMD_WRITE : IDE_CMD_WRMUL;
4285
4286   if (sector_per_block > 7) panic("idestart");
4287
4288   idewait(0);
4289   outb(0x3f6, 0);  // generate interrupt
4290   outb(0x1f2, sector_per_block);  // number of sectors
4291   outb(0x1f3, sector & 0xff);
4292   outb(0x1f4, (sector >> 8) & 0xff);
4293   outb(0x1f5, (sector >> 16) & 0xff);
4294   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4295   if(b->flags & B_DIRTY){
4296     outb(0x1f7, write_cmd);
4297     outsl(0x1f0, b->data, BSIZE/4);
4298   } else {
4299     outb(0x1f7, read_cmd);
```



```
4300   }
4301 }
4302
4303 // Interrupt handler.
4304 void
4305 ideintr(void)
4306 {
4307   struct buf *b;
4308
4309   // First queued buffer is the active request.
4310   acquire(&idelock);
4311   if((b = idequeue) == 0){
4312     release(&idelock);
4313     // cprintf("spurious IDE interrupt\n");
4314     return;
4315   }
4316   idequeue = b->qnext;
4317
4318   // Read data if needed.
4319   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4320     insl(0x1f0, b->data, BSIZE/4);
4321
4322   // Wake process waiting for this buf.
4323   b->flags |= B_VALID;
4324   b->flags &= ~B_DIRTY;
4325   wakeup(b);
4326
4327   // Start disk on next buf in queue.
4328   if(idequeue != 0)
4329     idestart(idequeue);
4330
4331   release(&idelock);
4332 }
4333
4334
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
```

```
4350 // Sync buf with disk.
4351 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
4352 // Else if B_VALID is not set, read buf from disk, set B_VALID.
4353 void
4354 iderw(struct buf *b)
4355 {
4356   struct buf **pp;
4357
4358   if(!holdingsleep(&b->lock))
4359     panic("iderw: buf not locked");
4360   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
4361     panic("iderw: nothing to do");
4362   if(b->dev != 0 && !havedisk1)
4363     panic("iderw: ide disk 1 not present");
4364
4365   acquire(&idelock);
4366
4367   // Append b to idequeue.
4368   b->qnext = 0;
4369   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
4370     ;
4371   *pp = b;
4372
4373   // Start disk if necessary.
4374   if(idequeue == b)
4375     idestart(b);
4376
4377   // Wait for request to finish.
4378   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
4379     sleep(b, &idelock);
4380   }
4381
4382   release(&idelock);
4383 }
4384
4385
4386
4387
4388
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
```



```
4400 // Buffer cache.
4401 //
4402 // The buffer cache is a linked list of buf structures holding
4403 // cached copies of disk block contents.  Caching disk blocks
4404 // in memory reduces the number of disk reads and also provides
4405 // a synchronization point for disk blocks used by multiple processes.
4406 //
4407 // Interface:
4408 // * To get a buffer for a particular disk block, call bread.
4409 // * After changing buffer data, call bwrite to write it to disk.
4410 // * When done with the buffer, call brelse.
4411 // * Do not use the buffer after calling brelse.
4412 // * Only one process at a time can use a buffer,
4413 //     so do not keep them longer than necessary.
4414 //
4415 // The implementation uses two state flags internally:
4416 // * B_VALID: the buffer data has been read from the disk.
4417 // * B_DIRTY: the buffer data has been modified
4418 //     and needs to be written to disk.
4419
4420 #include "types.h"
4421 #include "defs.h"
4422 #include "param.h"
4423 #include "spinlock.h"
4424 #include "sleeplock.h"
4425 #include "fs.h"
4426 #include "buf.h"
4427
4428 struct {
4429   struct spinlock lock;
4430   struct buf buf[NBUF];
4431
4432   // Linked list of all buffers, through prev/next.
4433   // head.next is most recently used.
4434   struct buf head;
4435 } bcache;
4436
4437 void
4438 binit(void)
4439 {
4440   struct buf *b;
4441
4442   initlock(&bcache.lock, "bcache");
4443
4444
4445
4446
4447
4448
4449
```

```
4450   // Create linked list of buffers
4451   bcache.head.prev = &bcache.head;
4452   bcache.head.next = &bcache.head;
4453   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4454     b->next = bcache.head.next;
4455     b->prev = &bcache.head;
4456     initsleeplock(&b->lock, "buffer");
4457     bcache.head.next->prev = b;
4458     bcache.head.next = b;
4459   }
4460 }
4461
4462 // Look through buffer cache for block on device dev.
4463 // If not found, allocate a buffer.
4464 // In either case, return locked buffer.
4465 static struct buf*
4466 bget(uint dev, uint blockno)
4467 {
4468   struct buf *b;
4469
4470   acquire(&bcache.lock);
4471
4472   // Is the block already cached?
4473   for(b = bcache.head.next; b != &bcache.head; b = b->next){
4474     if(b->dev == dev && b->blockno == blockno){
4475       b->refcnt++;
4476       release(&bcache.lock);
4477       acquiresleep(&b->lock);
4478       return b;
4479     }
4480   }
4481
4482   // Not cached; recycle some unused buffer and clean buffer
4483   // "clean" because B_DIRTY and not locked means log.c
4484   // hasn't yet committed the changes to the buffer.
4485   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4486     if(b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
4487       b->dev = dev;
4488       b->blockno = blockno;
4489       b->flags = 0;
4490       b->refcnt = 1;
4491       release(&bcache.lock);
4492       acquiresleep(&b->lock);
4493       return b;
4494     }
4495   }
4496   panic("bget: no buffers");
4497 }
4498
4499
```



```
4500 // Return a locked buf with the contents of the indicated block.
4501 struct buf*
4502 bread(uint dev, uint blockno)
4503 {
4504   struct buf *b;
4505
4506   b = bget(dev, blockno);
4507   if(!(b->flags & B_VALID)) {
4508     iderw(b);
4509   }
4510   return b;
4511 }
4512
4513 // Write b's contents to disk.  Must be locked.
4514 void
4515 bwrite(struct buf *b)
4516 {
4517   if(!holdingsleep(&b->lock))
4518     panic("bwrite");
4519   b->flags |= B_DIRTY;
4520   iderw(b);
4521 }
4522
4523 // Release a locked buffer.
4524 // Move to the head of the MRU list.
4525 void
4526 brelse(struct buf *b)
4527 {
4528   if(!holdingsleep(&b->lock))
4529     panic("brelse");
4530
4531   releasesleep(&b->lock);
4532
4533   acquire(&bcache.lock);
4534   b->refcnt--;
4535   if (b->refcnt == 0) {
4536     // no one is waiting for it.
4537     b->next->prev = b->prev;
4538     b->prev->next = b->next;
4539     b->next = bcache.head.next;
4540     b->prev = &bcache.head;
4541     bcache.head.next->prev = b;
4542     bcache.head.next = b;
4543   }
4544
4545   release(&bcache.lock);
4546 }
4547
4548
4549
```

```
4550 // Blank page.
4551
4552
4553
4554
4555
4556
4557
4558
4559
4560
4561
4562
4563
4564
4565
4566
4567
4568
4569
4570
4571
4572
4573
4574
4575
4576
4577
4578
4579
4580
4581
4582
4583
4584
4585
4586
4587
4588
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4599
```



```
4600 // Sleeping locks
4601
4602 #include "types.h"
4603 #include "defs.h"
4604 #include "param.h"
4605 #include "x86.h"
4606 #include "memlayout.h"
4607 #include "mmu.h"
4608 #include "proc.h"
4609 #include "spinlock.h"
4610 #include "sleeplock.h"
4611
4612 void
4613 initsleeplock(struct sleeplock *lk, char *name)
4614 {
4615   initlock(&lk->lk, "sleep lock");
4616   lk->name = name;
4617   lk->locked = 0;
4618   lk->pid = 0;
4619 }
4620
4621 void
4622 acquiresleep(struct sleeplock *lk)
4623 {
4624   acquire(&lk->lk);
4625   while (lk->locked) {
4626     sleep(lk, &lk->lk);
4627   }
4628   lk->locked = 1;
4629   lk->pid = proc->pid;
4630   release(&lk->lk);
4631 }
4632
4633 void
4634 releasesleep(struct sleeplock *lk)
4635 {
4636   acquire(&lk->lk);
4637   lk->locked = 0;
4638   lk->pid = 0;
4639   wakeup(lk);
4640   release(&lk->lk);
4641 }
4642
4643
4644
4645
4646
4647
4648
4649
```

```
4650 int
4651 holdingsleep(struct sleeplock *lk)
4652 {
4653   int r;
4654
4655   acquire(&lk->lk);
4656   r = lk->locked;
4657   release(&lk->lk);
4658   return r;
4659 }
4660
4661
4662
4663
4664
4665
4666
4667
4668
4669
4670
4671
4672
4673
4674
4675
4676
4677
4678
4679
4680
4681
4682
4683
4684
4685
4686
4687
4688
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4699
```



```
4700 #include "types.h"
4701 #include "defs.h"
4702 #include "param.h"
4703 #include "spinlock.h"
4704 #include "sleeplock.h"
4705 #include "fs.h"
4706 #include "buf.h"
4707
4708 // Simple logging that allows concurrent FS system calls.
4709 //
4710 // A log transaction contains the updates of multiple FS system
4711 // calls. The logging system only commits when there are
4712 // no FS system calls active. Thus there is never
4713 // any reasoning required about whether a commit might
4714 // write an uncommitted system call's updates to disk.
4715 //
4716 // A system call should call begin_op()/end_op() to mark
4717 // its start and end. Usually begin_op() just increments
4718 // the count of in-progress FS system calls and returns.
4719 // But if it thinks the log is close to running out, it
4720 // sleeps until the last outstanding end_op() commits.
4721 //
4722 // The log is a physical re-do log containing disk blocks.
4723 // The on-disk log format:
4724 //   header block, containing block #s for block A, B, C, ...
4725 //   block A
4726 //   block B
4727 //   block C
4728 //   ...
4729 // Log appends are synchronous.
4730
4731 // Contents of the header block, used for both the on-disk header block
4732 // and to keep track in memory of logged block# before commit.
4733 struct logheader {
4734   int n;
4735   int block[LOGSIZE];
4736 };
4737
4738 struct log {
4739   struct spinlock lock;
4740   int start;
4741   int size;
4742   int outstanding; // how many FS sys calls are executing.
4743   int committing;  // in commit(), please wait.
4744   int dev;
4745   struct logheader lh;
4746 };
4747
4748
4749
```

```
4750 struct log log;
4751
4752 static void recover_from_log(void);
4753 static void commit();
4754
4755 void
4756 initlog(int dev)
4757 {
4758   if (sizeof(struct logheader) >= BSIZE)
4759     panic("initlog: too big logheader");
4760
4761   struct superblock sb;
4762   initlock(&log.lock, "log");
4763   readsb(dev, &sb);
4764   log.start = sb.logstart;
4765   log.size = sb.nlog;
4766   log.dev = dev;
4767   recover_from_log();
4768 }
4769
4770 // Copy committed blocks from log to their home location
4771 static void
4772 install_trans(void)
4773 {
4774   int tail;
4775
4776   for (tail = 0; tail < log.lh.n; tail++) {
4777     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
4778     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
4779     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
4780     bwrite(dbuf);  // write dst to disk
4781     brelse(lbuf);
4782     brelse(dbuf);
4783   }
4784 }
4785
4786 // Read the log header from disk into the in-memory log header
4787 static void
4788 read_head(void)
4789 {
4790   struct buf *buf = bread(log.dev, log.start);
4791   struct logheader *lh = (struct logheader *) (buf->data);
4792   int i;
4793   log.lh.n = lh->n;
4794   for (i = 0; i < log.lh.n; i++) {
4795     log.lh.block[i] = lh->block[i];
4796   }
4797   brelse(buf);
4798 }
4799
```



```
4800 // Write in-memory log header to disk.
4801 // This is the true point at which the
4802 // current transaction commits.
4803 static void
4804 write_head(void)
4805 {
4806   struct buf *buf = bread(log.dev, log.start);
4807   struct logheader *hb = (struct logheader *) (buf->data);
4808   int i;
4809   hb->n = log.lh.n;
4810   for (i = 0; i < log.lh.n; i++) {
4811     hb->block[i] = log.lh.block[i];
4812   }
4813   bwrite(buf);
4814   brelse(buf);
4815 }
4816
4817 static void
4818 recover_from_log(void)
4819 {
4820   read_head();
4821   install_trans(); // if committed, copy from log to disk
4822   log.lh.n = 0;
4823   write_head(); // clear the log
4824 }
4825
4826 // called at the start of each FS system call.
4827 void
4828 begin_op(void)
4829 {
4830   acquire(&log.lock);
4831   while(1){
4832     if(log.committing){
4833       sleep(&log, &log.lock);
4834     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
4835       // this op might exhaust log space; wait for commit.
4836       sleep(&log, &log.lock);
4837     } else {
4838       log.outstanding += 1;
4839       release(&log.lock);
4840       break;
4841     }
4842   }
4843 }
4844
4845
4846
4847
4848
4849
```

```
4850 // called at the end of each FS system call.
4851 // commits if this was the last outstanding operation.
4852 void
4853 end_op(void)
4854 {
4855   int do_commit = 0;
4856
4857   acquire(&log.lock);
4858   log.outstanding -= 1;
4859   if(log.committing)
4860     panic("log.committing");
4861   if(log.outstanding == 0){
4862     do_commit = 1;
4863     log.committing = 1;
4864   } else {
4865     // begin_op() may be waiting for log space.
4866     wakeup(&log);
4867   }
4868   release(&log.lock);
4869
4870   if(do_commit){
4871     // call commit w/o holding locks, since not allowed
4872     // to sleep with locks.
4873     commit();
4874     acquire(&log.lock);
4875     log.committing = 0;
4876     wakeup(&log);
4877     release(&log.lock);
4878   }
4879 }
4880
4881 // Copy modified blocks from cache to log.
4882 static void
4883 write_log(void)
4884 {
4885   int tail;
4886
4887   for (tail = 0; tail < log.lh.n; tail++) {
4888     struct buf *to = bread(log.dev, log.start+tail+1); // log block
4889     struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
4890     memmove(to->data, from->data, BSIZE);
4891     bwrite(to);  // write the log
4892     brelse(from);
4893     brelse(to);
4894   }
4895 }
4896
4897
4898
4899
```



```
4900 static void
4901 commit()
4902 {
4903   if (log.lh.n > 0) {
4904     write_log();     // Write modified blocks from cache to log
4905     write_head();    // Write header to disk -- the real commit
4906     install_trans(); // Now install writes to home locations
4907     log.lh.n = 0;
4908     write_head();    // Erase the transaction from the log
4909   }
4910 }
4911
4912 // Caller has modified b->data and is done with the buffer.
4913 // Record the block number and pin in the cache with B_DIRTY.
4914 // commit()/write_log() will do the disk write.
4915 //
4916 // log_write() replaces bwrite(); a typical use is:
4917 //   bp = bread(...)
4918 //   modify bp->data[]
4919 //   log_write(bp)
4920 //   brelse(bp)
4921 void
4922 log_write(struct buf *b)
4923 {
4924   int i;
4925
4926   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
4927     panic("too big a transaction");
4928   if (log.outstanding < 1)
4929     panic("log_write outside of trans");
4930
4931   acquire(&log.lock);
4932   for (i = 0; i < log.lh.n; i++) {
4933     if (log.lh.block[i] == b->blockno)   // log absorbtion
4934       break;
4935   }
4936   log.lh.block[i] = b->blockno;
4937   if (i == log.lh.n)
4938     log.lh.n++;
4939   b->flags |= B_DIRTY; // prevent eviction
4940   release(&log.lock);
4941 }
4942
4943
4944
4945
4946
4947
4948
4949
```

```
4950 // File system implementation.  Five layers:
4951 //   + Blocks: allocator for raw disk blocks.
4952 //   + Log: crash recovery for multi-step updates.
4953 //   + Files: inode allocator, reading, writing, metadata.
4954 //   + Directories: inode with special contents (list of other inodes!)
4955 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
4956 //
4957 // This file contains the low-level file system manipulation
4958 // routines.  The (higher-level) system call implementations
4959 // are in sysfile.c.
4960
4961 #include "types.h"
4962 #include "defs.h"
4963 #include "param.h"
4964 #include "stat.h"
4965 #include "mmu.h"
4966 #include "proc.h"
4967 #include "spinlock.h"
4968 #include "sleeplock.h"
4969 #include "fs.h"
4970 #include "buf.h"
4971 #include "file.h"
4972
4973 #define min(a, b) ((a) < (b) ? (a) : (b))
4974 static void itrunc(struct inode*);
4975 // there should be one superblock per disk device, but we run with
4976 // only one device
4977 struct superblock sb;
4978
4979 // Read the super block.
4980 void
4981 readsb(int dev, struct superblock *sb)
4982 {
4983   struct buf *bp;
4984
4985   bp = bread(dev, 1);
4986   memmove(sb, bp->data, sizeof(*sb));
4987   brelse(bp);
4988 }
4989
4990
4991
4992
4993
4994
4995
4996
4997
4998
4999
```



```
5000 // Zero a block.
5001 static void
5002 bzero(int dev, int bno)
5003 {
5004   struct buf *bp;
5005
5006   bp = bread(dev, bno);
5007   memset(bp->data, 0, BSIZE);
5008   log_write(bp);
5009   brelse(bp);
5010 }
5011
5012 // Blocks.
5013
5014 // Allocate a zeroed disk block.
5015 static uint
5016 balloc(uint dev)
5017 {
5018   int b, bi, m;
5019   struct buf *bp;
5020
5021   bp = 0;
5022   for(b = 0; b < sb.size; b += BPB){
5023     bp = bread(dev, BBLOCK(b, sb));
5024     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
5025       m = 1 << (bi % 8);
5026       if((bp->data[bi/8] & m) == 0){  // Is block free?
5027         bp->data[bi/8] |= m;  // Mark block in use.
5028         log_write(bp);
5029         brelse(bp);
5030         bzero(dev, b + bi);
5031         return b + bi;
5032       }
5033     }
5034     brelse(bp);
5035   }
5036   panic("balloc: out of blocks");
5037 }
5038
5039
5040
5041
5042
5043
5044
5045
5046
5047
5048
5049
```

```
5050 // Free a disk block.
5051 static void
5052 bfree(int dev, uint b)
5053 {
5054   struct buf *bp;
5055   int bi, m;
5056
5057   readsb(dev, &sb);
5058   bp = bread(dev, BBLOCK(b, sb));
5059   bi = b % BPB;
5060   m = 1 << (bi % 8);
5061   if((bp->data[bi/8] & m) == 0)
5062     panic("freeing free block");
5063   bp->data[bi/8] &= ~m;
5064   log_write(bp);
5065   brelse(bp);
5066 }
5067
5068 // Inodes.
5069 //
5070 // An inode describes a single unnamed file.
5071 // The inode disk structure holds metadata: the file's type,
5072 // its size, the number of links referring to it, and the
5073 // list of blocks holding the file's content.
5074 //
5075 // The inodes are laid out sequentially on disk at
5076 // sb.startinode. Each inode has a number, indicating its
5077 // position on the disk.
5078 //
5079 // The kernel keeps a cache of in-use inodes in memory
5080 // to provide a place for synchronizing access
5081 // to inodes used by multiple processes. The cached
5082 // inodes include book-keeping information that is
5083 // not stored on disk: ip->ref and ip->flags.
5084 //
5085 // An inode and its in-memory represtative go through a
5086 // sequence of states before they can be used by the
5087 // rest of the file system code.
5088 //
5089 // * Allocation: an inode is allocated if its type (on disk)
5090 //   is non-zero. ialloc() allocates, iput() frees if
5091 //   the link count has fallen to zero.
5092 //
5093 // * Referencing in cache: an entry in the inode cache
5094 //   is free if ip->ref is zero. Otherwise ip->ref tracks
5095 //   the number of in-memory pointers to the entry (open
5096 //   files and current directories). iget() to find or
5097 //   create a cache entry and increment its ref, iput()
5098 //   to decrement ref.
5099 //
```



```
5100 // * Valid: the information (type, size, &c) in an inode
5101 //   cache entry is only correct when the I_VALID bit
5102 //   is set in ip->flags. ilock() reads the inode from
5103 //   the disk and sets I_VALID, while iput() clears
5104 //   I_VALID if ip->ref has fallen to zero.
5105 //
5106 // * Locked: file system code may only examine and modify
5107 //   the information in an inode and its content if it
5108 //   has first locked the inode.
5109 //
5110 // Thus a typical sequence is:
5111 //   ip = iget(dev, inum)
5112 //   ilock(ip)
5113 //   ... examine and modify ip->xxx ...
5114 //   iunlock(ip)
5115 //   iput(ip)
5116 //
5117 // ilock() is separate from iget() so that system calls can
5118 // get a long-term reference to an inode (as for an open file)
5119 // and only lock it for short periods (e.g., in read()).
5120 // The separation also helps avoid deadlock and races during
5121 // pathname lookup. iget() increments ip->ref so that the inode
5122 // stays cached and pointers to it remain valid.
5123 //
5124 // Many internal file system functions expect the caller to
5125 // have locked the inodes involved; this lets callers create
5126 // multi-step atomic operations.
5127
5128 struct {
5129   struct spinlock lock;
5130   struct inode inode[NINODE];
5131 } icache;
5132
5133 void
5134 iinit(int dev)
5135 {
5136   int i = 0;
5137
5138   initlock(&icache.lock, "icache");
5139   for(i = 0; i < NINODE; i++) {
5140     initsleeplock(&icache.inode[i].lock, "inode");
5141   }
5142
5143   readsb(dev, &sb);
5144   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d\
5145  inodestart %d bmap start %d\n", sb.size, sb.nblocks,
5146          sb.ninodes, sb.nlog, sb.logstart, sb.inodestart,
5147          sb.bmapstart);
5148 }
5149
```

```
5150 static struct inode* iget(uint dev, uint inum);
5151
5152
5153
5154
5155
5156
5157
5158
5159
5160
5161
5162
5163
5164
5165
5166
5167
5168
5169
5170
5171
5172
5173
5174
5175
5176
5177
5178
5179
5180
5181
5182
5183
5184
5185
5186
5187
5188
5189
5190
5191
5192
5193
5194
5195
5196
5197
5198
5199
```



```
5200 // Allocate a new inode with the given type on device dev.
5201 // A free inode has a type of zero.
5202 struct inode*
5203 ialloc(uint dev, short type)
5204 {
5205   int inum;
5206   struct buf *bp;
5207   struct dinode *dip;
5208
5209   for(inum = 1; inum < sb.ninodes; inum++){
5210     bp = bread(dev, IBLOCK(inum, sb));
5211     dip = (struct dinode*)bp->data + inum%IPB;
5212     if(dip->type == 0){  // a free inode
5213       memset(dip, 0, sizeof(*dip));
5214       dip->type = type;
5215       log_write(bp);   // mark it allocated on the disk
5216       brelse(bp);
5217       return iget(dev, inum);
5218     }
5219     brelse(bp);
5220   }
5221   panic("ialloc: no inodes");
5222 }
5223
5224 // Copy a modified in-memory inode to disk.
5225 void
5226 iupdate(struct inode *ip)
5227 {
5228   struct buf *bp;
5229   struct dinode *dip;
5230
5231   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5232   dip = (struct dinode*)bp->data + ip->inum%IPB;
5233   dip->type = ip->type;
5234   dip->major = ip->major;
5235   dip->minor = ip->minor;
5236   dip->nlink = ip->nlink;
5237   dip->size = ip->size;
5238   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5239   log_write(bp);
5240   brelse(bp);
5241 }
5242
5243
5244
5245
5246
5247
5248
5249
```

```
5250 // Find the inode with number inum on device dev
5251 // and return the in-memory copy. Does not lock
5252 // the inode and does not read it from disk.
5253 static struct inode*
5254 iget(uint dev, uint inum)
5255 {
5256   struct inode *ip, *empty;
5257
5258   acquire(&icache.lock);
5259
5260   // Is the inode already cached?
5261   empty = 0;
5262   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5263     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5264       ip->ref++;
5265       release(&icache.lock);
5266       return ip;
5267     }
5268     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
5269       empty = ip;
5270   }
5271
5272   // Recycle an inode cache entry.
5273   if(empty == 0)
5274     panic("iget: no inodes");
5275
5276   ip = empty;
5277   ip->dev = dev;
5278   ip->inum = inum;
5279   ip->ref = 1;
5280   ip->flags = 0;
5281   release(&icache.lock);
5282
5283   return ip;
5284 }
5285
5286 // Increment reference count for ip.
5287 // Returns ip to enable ip = idup(ip1) idiom.
5288 struct inode*
5289 idup(struct inode *ip)
5290 {
5291   acquire(&icache.lock);
5292   ip->ref++;
5293   release(&icache.lock);
5294   return ip;
5295 }
5296
5297
5298
5299
```



```
5300 // Lock the given inode.
5301 // Reads the inode from disk if necessary.
5302 void
5303 ilock(struct inode *ip)
5304 {
5305   struct buf *bp;
5306   struct dinode *dip;
5307
5308   if(ip == 0 || ip->ref < 1)
5309     panic("ilock");
5310
5311   acquiresleep(&ip->lock);
5312
5313   if(!(ip->flags & I_VALID)){
5314     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5315     dip = (struct dinode*)bp->data + ip->inum%IPB;
5316     ip->type = dip->type;
5317     ip->major = dip->major;
5318     ip->minor = dip->minor;
5319     ip->nlink = dip->nlink;
5320     ip->size = dip->size;
5321     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5322     brelse(bp);
5323     ip->flags |= I_VALID;
5324     if(ip->type == 0)
5325       panic("ilock: no type");
5326   }
5327 }
5328
5329 // Unlock the given inode.
5330 void
5331 iunlock(struct inode *ip)
5332 {
5333   if(ip == 0 || !holdingsleep(&ip->lock) || ip->ref < 1)
5334     panic("iunlock");
5335
5336   releasesleep(&ip->lock);
5337 }
5338
5339
5340
5341
5342
5343
5344
5345
5346
5347
5348
5349
```

```
5350 // Drop a reference to an in-memory inode.
5351 // If that was the last reference, the inode cache entry can
5352 // be recycled.
5353 // If that was the last reference and the inode has no links
5354 // to it, free the inode (and its content) on disk.
5355 // All calls to iput() must be inside a transaction in
5356 // case it has to free the inode.
5357 void
5358 iput(struct inode *ip)
5359 {
5360   acquire(&icache.lock);
5361   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
5362     // inode has no links and no other references: truncate and free.
5363     release(&icache.lock);
5364     itrunc(ip);
5365     ip->type = 0;
5366     iupdate(ip);
5367     acquire(&icache.lock);
5368     ip->flags = 0;
5369   }
5370   ip->ref--;
5371   release(&icache.lock);
5372 }
5373
5374 // Common idiom: unlock, then put.
5375 void
5376 iunlockput(struct inode *ip)
5377 {
5378   iunlock(ip);
5379   iput(ip);
5380 }
5381
5382
5383
5384
5385
5386
5387
5388
5389
5390
5391
5392
5393
5394
5395
5396
5397
5398
5399
```



```
5400 // Inode content
5401 //
5402 // The content (data) associated with each inode is stored
5403 // in blocks on the disk. The first NDIRECT block numbers
5404 // are listed in ip->addrs[].  The next NINDIRECT blocks are
5405 // listed in block ip->addrs[NDIRECT].
5406
5407 // Return the disk block address of the nth block in inode ip.
5408 // If there is no such block, bmap allocates one.
5409 static uint
5410 bmap(struct inode *ip, uint bn)
5411 {
5412   uint addr, *a;
5413   struct buf *bp;
5414
5415   if(bn < NDIRECT){
5416     if((addr = ip->addrs[bn]) == 0)
5417       ip->addrs[bn] = addr = balloc(ip->dev);
5418     return addr;
5419   }
5420   bn -= NDIRECT;
5421
5422   if(bn < NINDIRECT){
5423     // Load indirect block, allocating if necessary.
5424     if((addr = ip->addrs[NDIRECT]) == 0)
5425       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5426     bp = bread(ip->dev, addr);
5427     a = (uint*)bp->data;
5428     if((addr = a[bn]) == 0){
5429       a[bn] = addr = balloc(ip->dev);
5430       log_write(bp);
5431     }
5432     brelse(bp);
5433     return addr;
5434   }
5435
5436   panic("bmap: out of range");
5437 }
5438
5439
5440
5441
5442
5443
5444
5445
5446
5447
5448
5449
```

```
5450 // Truncate inode (discard contents).
5451 // Only called when the inode has no links
5452 // to it (no directory entries referring to it)
5453 // and has no in-memory reference to it (is
5454 // not an open file or current directory).
5455 static void
5456 itrunc(struct inode *ip)
5457 {
5458   int i, j;
5459   struct buf *bp;
5460   uint *a;
5461
5462   for(i = 0; i < NDIRECT; i++){
5463     if(ip->addrs[i]){
5464       bfree(ip->dev, ip->addrs[i]);
5465       ip->addrs[i] = 0;
5466     }
5467   }
5468
5469   if(ip->addrs[NDIRECT]){
5470     bp = bread(ip->dev, ip->addrs[NDIRECT]);
5471     a = (uint*)bp->data;
5472     for(j = 0; j < NINDIRECT; j++){
5473       if(a[j])
5474         bfree(ip->dev, a[j]);
5475     }
5476     brelse(bp);
5477     bfree(ip->dev, ip->addrs[NDIRECT]);
5478     ip->addrs[NDIRECT] = 0;
5479   }
5480
5481   ip->size = 0;
5482   iupdate(ip);
5483 }
5484
5485 // Copy stat information from inode.
5486 void
5487 stati(struct inode *ip, struct stat *st)
5488 {
5489   st->dev = ip->dev;
5490   st->ino = ip->inum;
5491   st->type = ip->type;
5492   st->nlink = ip->nlink;
5493   st->size = ip->size;
5494 }
5495
5496
5497
5498
5499
```



```
5500 // Read data from inode.
5501 int
5502 readi(struct inode *ip, char *dst, uint off, uint n)
5503 {
5504   uint tot, m;
5505   struct buf *bp;
5506
5507   if(ip->type == T_DEV){
5508     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
5509       return -1;
5510     return devsw[ip->major].read(ip, dst, n);
5511   }
5512
5513   if(off > ip->size || off + n < off)
5514     return -1;
5515   if(off + n > ip->size)
5516     n = ip->size - off;
5517
5518   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5519     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5520     m = min(n - tot, BSIZE - off%BSIZE);
5521     /*
5522     cprintf("data off %d:\n", off);
5523     for (int j = 0; j < min(m, 10); j++) {
5524       cprintf("%x ", bp->data[off%BSIZE+j]);
5525     }
5526     cprintf("\n");
5527     */
5528     memmove(dst, bp->data + off%BSIZE, m);
5529     brelse(bp);
5530   }
5531   return n;
5532 }
5533
5534
5535
5536
5537
5538
5539
5540
5541
5542
5543
5544
5545
5546
5547
5548
5549
```

```
5550 // Write data to inode.
5551 int
5552 writei(struct inode *ip, char *src, uint off, uint n)
5553 {
5554   uint tot, m;
5555   struct buf *bp;
5556
5557   if(ip->type == T_DEV){
5558     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
5559       return -1;
5560     return devsw[ip->major].write(ip, src, n);
5561   }
5562
5563   if(off > ip->size || off + n < off)
5564     return -1;
5565   if(off + n > MAXFILE*BSIZE)
5566     return -1;
5567
5568   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
5569     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5570     m = min(n - tot, BSIZE - off%BSIZE);
5571     memmove(bp->data + off%BSIZE, src, m);
5572     log_write(bp);
5573     brelse(bp);
5574   }
5575
5576   if(n > 0 && off > ip->size){
5577     ip->size = off;
5578     iupdate(ip);
5579   }
5580   return n;
5581 }
5582
5583
5584
5585
5586
5587
5588
5589
5590
5591
5592
5593
5594
5595
5596
5597
5598
5599
```



```
5600 // Directories
5601
5602 int
5603 namecmp(const char *s, const char *t)
5604 {
5605   return strncmp(s, t, DIRSIZ);
5606 }
5607
5608 // Look for a directory entry in a directory.
5609 // If found, set *poff to byte offset of entry.
5610 struct inode*
5611 dirlookup(struct inode *dp, char *name, uint *poff)
5612 {
5613   uint off, inum;
5614   struct dirent de;
5615
5616   if(dp->type != T_DIR)
5617     panic("dirlookup not DIR");
5618
5619   for(off = 0; off < dp->size; off += sizeof(de)){
5620     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5621       panic("dirlink read");
5622     if(de.inum == 0)
5623       continue;
5624     if(namecmp(name, de.name) == 0){
5625       // entry matches path element
5626       if(poff)
5627         *poff = off;
5628       inum = de.inum;
5629       return iget(dp->dev, inum);
5630     }
5631   }
5632
5633   return 0;
5634 }
5635
5636
5637
5638
5639
5640
5641
5642
5643
5644
5645
5646
5647
5648
5649
```

```
5650 // Write a new directory entry (name, inum) into the directory dp.
5651 int
5652 dirlink(struct inode *dp, char *name, uint inum)
5653 {
5654   int off;
5655   struct dirent de;
5656   struct inode *ip;
5657
5658   // Check that name is not present.
5659   if((ip = dirlookup(dp, name, 0)) != 0){
5660     iput(ip);
5661     return -1;
5662   }
5663
5664   // Look for an empty dirent.
5665   for(off = 0; off < dp->size; off += sizeof(de)){
5666     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5667       panic("dirlink read");
5668     if(de.inum == 0)
5669       break;
5670   }
5671
5672   strncpy(de.name, name, DIRSIZ);
5673   de.inum = inum;
5674   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5675     panic("dirlink");
5676
5677   return 0;
5678 }
5679
5680
5681
5682
5683
5684
5685
5686
5687
5688
5689
5690
5691
5692
5693
5694
5695
5696
5697
5698
5699
```



```
5700 // Paths
5701
5702 // Copy the next path element from path into name.
5703 // Return a pointer to the element following the copied one.
5704 // The returned path has no leading slashes,
5705 // so the caller can check *path=='\0' to see if the name is the last one.
5706 // If no name to remove, return 0.
5707 //
5708 // Examples:
5709 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5710 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5711 //   skipelem("a", name) = "", setting name = "a"
5712 //   skipelem("", name) = skipelem("////", name) = 0
5713 //
5714 static char*
5715 skipelem(char *path, char *name)
5716 {
5717   char *s;
5718   int len;
5719
5720   while(*path == '/')
5721     path++;
5722   if(*path == 0)
5723     return 0;
5724   s = path;
5725   while(*path != '/' && *path != 0)
5726     path++;
5727   len = path - s;
5728   if(len >= DIRSIZ)
5729     memmove(name, s, DIRSIZ);
5730   else {
5731     memmove(name, s, len);
5732     name[len] = 0;
5733   }
5734   while(*path == '/')
5735     path++;
5736   return path;
5737 }
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749
```

```
5750 // Look up and return the inode for a path name.
5751 // If parent != 0, return the inode for the parent and copy the final
5752 // path element into name, which must have room for DIRSIZ bytes.
5753 // Must be called inside a transaction since it calls iput().
5754 static struct inode*
5755 namex(char *path, int nameiparent, char *name)
5756 {
5757   struct inode *ip, *next;
5758
5759   if(*path == '/')
5760     ip = iget(ROOTDEV, ROOTINO);
5761   else
5762     ip = idup(proc->cwd);
5763
5764   while((path = skipelem(path, name)) != 0){
5765     ilock(ip);
5766     if(ip->type != T_DIR){
5767       iunlockput(ip);
5768       return 0;
5769     }
5770     if(nameiparent && *path == '\0'){
5771       // Stop one level early.
5772       iunlock(ip);
5773       return ip;
5774     }
5775     if((next = dirlookup(ip, name, 0)) == 0){
5776       iunlockput(ip);
5777       return 0;
5778     }
5779     iunlockput(ip);
5780     ip = next;
5781   }
5782   if(nameiparent){
5783     iput(ip);
5784     return 0;
5785   }
5786   return ip;
5787 }
5788
5789 struct inode*
5790 namei(char *path)
5791 {
5792   char name[DIRSIZ];
5793   return namex(path, 0, name);
5794 }
5795
5796
5797
5798
5799
```



```
5800 struct inode*
5801 nameiparent(char *path, char *name)
5802 {
5803   return namex(path, 1, name);
5804 }
5805
5806
5807
5808
5809
5810
5811
5812
5813
5814
5815
5816
5817
5818
5819
5820
5821
5822
5823
5824
5825
5826
5827
5828
5829
5830
5831
5832
5833
5834
5835
5836
5837
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```

```
5850 //
5851 // File descriptors
5852 //
5853
5854 #include "types.h"
5855 #include "defs.h"
5856 #include "param.h"
5857 #include "fs.h"
5858 #include "spinlock.h"
5859 #include "sleeplock.h"
5860 #include "file.h"
5861
5862 struct devsw devsw[NDEV];
5863 struct {
5864   struct spinlock lock;
5865   struct file file[NFILE];
5866 } ftable;
5867
5868 void
5869 fileinit(void)
5870 {
5871   initlock(&ftable.lock, "ftable");
5872 }
5873
5874 // Allocate a file structure.
5875 struct file*
5876 filealloc(void)
5877 {
5878   struct file *f;
5879
5880   acquire(&ftable.lock);
5881   for(f = ftable.file; f < ftable.file + NFILE; f++){
5882     if(f->ref == 0){
5883       f->ref = 1;
5884       release(&ftable.lock);
5885       return f;
5886     }
5887   }
5888   release(&ftable.lock);
5889   return 0;
5890 }
5891
5892
5893
5894
5895
5896
5897
5898
5899
```



```
5900 // Increment ref count for file f.
5901 struct file*
5902 filedup(struct file *f)
5903 {
5904   acquire(&ftable.lock);
5905   if(f->ref < 1)
5906     panic("filedup");
5907   f->ref++;
5908   release(&ftable.lock);
5909   return f;
5910 }
5911
5912 // Close file f.  (Decrement ref count, close when reaches 0.)
5913 void
5914 fileclose(struct file *f)
5915 {
5916   struct file ff;
5917
5918   acquire(&ftable.lock);
5919   if(f->ref < 1)
5920     panic("fileclose");
5921   if(--f->ref > 0){
5922     release(&ftable.lock);
5923     return;
5924   }
5925   ff = *f;
5926   f->ref = 0;
5927   f->type = FD_NONE;
5928   release(&ftable.lock);
5929
5930   if(ff.type == FD_PIPE)
5931     pipeclose(ff.pipe, ff.writable);
5932   else if(ff.type == FD_INODE){
5933     begin_op();
5934     iput(ff.ip);
5935     end_op();
5936   }
5937 }
5938
5939
5940
5941
5942
5943
5944
5945
5946
5947
5948
5949
```

```
// Get metadata about file f.
int
filestat(struct file *f, struct stat *st)
{
  if(f->type == FD_INODE){
    ilock(f->ip);
    stati(f->ip, st);
    iunlock(f->ip);
    return 0;
  }
  return -1;
}

// Read from file f.
int
fileread(struct file *f, char *addr, int n)
{
  int r;

  if(f->readable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return piperead(f->pipe, addr, n);
  if(f->type == FD_INODE){
    ilock(f->ip);
    if((r = readi(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
  panic("fileread");
}
```



```
// Write to file f.
int
filewrite(struct file *f, char *addr, int n)
{
  int r;

  if(f->writable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return pipewrite(f->pipe, addr, n);
  if(f->type == FD_INODE){
    // write a few blocks at a time to avoid exceeding
    // the maximum log transaction size, including
    // i-node, indirect block, allocation blocks,
    // and 2 blocks of slop for non-aligned writes.
    // this really belongs lower down, since writei()
    // might be writing a device like the console.
    int max = ((LOGSIZE-1-1-2) / 2) * 512;
    int i = 0;
    while(i < n){
      int n1 = n - i;
      if(n1 > max)
        n1 = max;

      begin_op();
      ilock(f->ip);
      if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
        f->off += r;
      iunlock(f->ip);
      end_op();

      if(r < 0)
        break;
      if(r != n1)
        panic("short filewrite");
      i += r;
    }
    return i == n ? n : -1;
  }
  panic("filewrite");
}
```

```
6050 //
6051 // File-system system calls.
6052 // Mostly argument checking, since we don't trust
6053 // user code, and calls into file.c and fs.c.
6054 //
6055
6056 #include "types.h"
6057 #include "defs.h"
6058 #include "param.h"
6059 #include "stat.h"
6060 #include "mmu.h"
6061 #include "proc.h"
6062 #include "fs.h"
6063 #include "spinlock.h"
6064 #include "sleeplock.h"
6065 #include "file.h"
6066 #include "fcntl.h"
6067
6068 // Fetch the nth word-sized system call argument as a file descriptor
6069 // and return both the descriptor and the corresponding struct file.
6070 static int
6071 argfd(int n, int *pfd, struct file **pf)
6072 {
6073   int fd;
6074   struct file *f;
6075
6076   if(argint(n, &fd) < 0)
6077     return -1;
6078   if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
6079     return -1;
6080   if(pfd)
6081     *pfd = fd;
6082   if(pf)
6083     *pf = f;
6084   return 0;
6085 }
6086
6087
6088
6089
6090
6091
6092
6093
6094
6095
6096
6097
6098
6099
```



```
6100 // Allocate a file descriptor for the given file.
6101 // Takes over file reference from caller on success.
6102 static int
6103 fdalloc(struct file *f)
6104 {
6105   int fd;
6106
6107   for(fd = 0; fd < NOFILE; fd++){
6108     if(proc->ofile[fd] == 0){
6109       proc->ofile[fd] = f;
6110       return fd;
6111     }
6112   }
6113   return -1;
6114 }
6115
6116 int
6117 sys_dup(void)
6118 {
6119   struct file *f;
6120   int fd;
6121
6122   if(argfd(0, 0, &f) < 0)
6123     return -1;
6124   if((fd=fdalloc(f)) < 0)
6125     return -1;
6126   filedup(f);
6127   return fd;
6128 }
6129
6130 int
6131 sys_read(void)
6132 {
6133   struct file *f;
6134   int n;
6135   char *p;
6136
6137   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6138     return -1;
6139   return fileread(f, p, n);
6140 }
6141
6142
6143
6144
6145
6146
6147
6148
6149
```

```
6150 int
6151 sys_write(void)
6152 {
6153   struct file *f;
6154   int n;
6155   char *p;
6156
6157   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6158     return -1;
6159   return filewrite(f, p, n);
6160 }
6161
6162 int
6163 sys_close(void)
6164 {
6165   int fd;
6166   struct file *f;
6167
6168   if(argfd(0, &fd, &f) < 0)
6169     return -1;
6170   proc->ofile[fd] = 0;
6171   fileclose(f);
6172   return 0;
6173 }
6174
6175 int
6176 sys_fstat(void)
6177 {
6178   struct file *f;
6179   struct stat *st;
6180
6181   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
6182     return -1;
6183   return filestat(f, st);
6184 }
6185
6186
6187
6188
6189
6190
6191
6192
6193
6194
6195
6196
6197
6198
6199
```



```
6200 // Create the path new as a link to the same inode as old.
6201 int
6202 sys_link(void)
6203 {
6204   char name[DIRSIZ], *new, *old;
6205   struct inode *dp, *ip;
6206
6207   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
6208     return -1;
6209
6210   begin_op();
6211   if((ip = namei(old)) == 0){
6212     end_op();
6213     return -1;
6214   }
6215
6216   ilock(ip);
6217   if(ip->type == T_DIR){
6218     iunlockput(ip);
6219     end_op();
6220     return -1;
6221   }
6222
6223   ip->nlink++;
6224   iupdate(ip);
6225   iunlock(ip);
6226
6227   if((dp = nameiparent(new, name)) == 0)
6228     goto bad;
6229   ilock(dp);
6230   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
6231     iunlockput(dp);
6232     goto bad;
6233   }
6234   iunlockput(dp);
6235   iput(ip);
6236
6237   end_op();
6238
6239   return 0;
6240
6241 bad:
6242   ilock(ip);
6243   ip->nlink--;
6244   iupdate(ip);
6245   iunlockput(ip);
6246   end_op();
6247   return -1;
6248 }
6249
```

```
6250 // Is the directory dp empty except for "." and ".." ?
6251 static int
6252 isdirempty(struct inode *dp)
6253 {
6254   int off;
6255   struct dirent de;
6256
6257   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
6258     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6259       panic("isdirempty: readi");
6260     if(de.inum != 0)
6261       return 0;
6262   }
6263   return 1;
6264 }
6265
6266
6267
6268
6269
6270
6271
6272
6273
6274
6275
6276
6277
6278
6279
6280
6281
6282
6283
6284
6285
6286
6287
6288
6289
6290
6291
6292
6293
6294
6295
6296
6297
6298
6299
```



```
6300 int
6301 sys_unlink(void)
6302 {
6303   struct inode *ip, *dp;
6304   struct dirent de;
6305   char name[DIRSIZ], *path;
6306   uint off;
6307
6308   if(argstr(0, &path) < 0)
6309     return -1;
6310
6311   begin_op();
6312   if((dp = nameiparent(path, name)) == 0){
6313     end_op();
6314     return -1;
6315   }
6316
6317   ilock(dp);
6318
6319   // Cannot unlink "." or "..".
6320   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6321     goto bad;
6322
6323   if((ip = dirlookup(dp, name, &off)) == 0)
6324     goto bad;
6325   ilock(ip);
6326
6327   if(ip->nlink < 1)
6328     panic("unlink: nlink < 1");
6329   if(ip->type == T_DIR && !isdirempty(ip)){
6330     iunlockput(ip);
6331     goto bad;
6332   }
6333
6334   memset(&de, 0, sizeof(de));
6335   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6336     panic("unlink: writei");
6337   if(ip->type == T_DIR){
6338     dp->nlink--;
6339     iupdate(dp);
6340   }
6341   iunlockput(dp);
6342
6343   ip->nlink--;
6344   iupdate(ip);
6345   iunlockput(ip);
6346
6347   end_op();
6348
6349   return 0;
```

```
6350 bad:
6351   iunlockput(dp);
6352   end_op();
6353   return -1;
6354 }
6355
6356 static struct inode*
6357 create(char *path, short type, short major, short minor)
6358 {
6359   uint off;
6360   struct inode *ip, *dp;
6361   char name[DIRSIZ];
6362
6363   if((dp = nameiparent(path, name)) == 0)
6364     return 0;
6365   ilock(dp);
6366
6367   if((ip = dirlookup(dp, name, &off)) != 0){
6368     iunlockput(dp);
6369     ilock(ip);
6370     if(type == T_FILE && ip->type == T_FILE)
6371       return ip;
6372     iunlockput(ip);
6373     return 0;
6374   }
6375
6376   if((ip = ialloc(dp->dev, type)) == 0)
6377     panic("create: ialloc");
6378
6379   ilock(ip);
6380   ip->major = major;
6381   ip->minor = minor;
6382   ip->nlink = 1;
6383   iupdate(ip);
6384
6385   if(type == T_DIR){  // Create . and .. entries.
6386     dp->nlink++;  // for ".."
6387     iupdate(dp);
6388     // No ip->nlink++ for ".": avoid cyclic ref count.
6389     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6390       panic("create dots");
6391   }
6392
6393   if(dirlink(dp, name, ip->inum) < 0)
6394     panic("create: dirlink");
6395
6396   iunlockput(dp);
6397
6398   return ip;
6399 }
```



```
6400 int
6401 sys_open(void)
6402 {
6403   char *path;
6404   int fd, omode;
6405   struct file *f;
6406   struct inode *ip;
6407
6408   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6409     return -1;
6410
6411   begin_op();
6412
6413   if(omode & O_CREATE){
6414     ip = create(path, T_FILE, 0, 0);
6415     if(ip == 0){
6416       end_op();
6417       return -1;
6418     }
6419   } else {
6420     if((ip = namei(path)) == 0){
6421       end_op();
6422       return -1;
6423     }
6424     ilock(ip);
6425     if(ip->type == T_DIR && omode != O_RDONLY){
6426       iunlockput(ip);
6427       end_op();
6428       return -1;
6429     }
6430   }
6431
6432   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6433     if(f)
6434       fileclose(f);
6435     iunlockput(ip);
6436     end_op();
6437     return -1;
6438   }
6439   iunlock(ip);
6440   end_op();
6441
6442   f->type = FD_INODE;
6443   f->ip = ip;
6444   f->off = 0;
6445   f->readable = !(omode & O_WRONLY);
6446   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6447   return fd;
6448 }
6449
```

```
6450 int
6451 sys_mkdir(void)
6452 {
6453   char *path;
6454   struct inode *ip;
6455
6456   begin_op();
6457   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
6458     end_op();
6459     return -1;
6460   }
6461   iunlockput(ip);
6462   end_op();
6463   return 0;
6464 }
6465
6466 int
6467 sys_mknod(void)
6468 {
6469   struct inode *ip;
6470   char *path;
6471   int major, minor;
6472
6473   begin_op();
6474   if((argstr(0, &path)) < 0 ||
6475      argint(1, &major) < 0 ||
6476      argint(2, &minor) < 0 ||
6477      (ip = create(path, T_DEV, major, minor)) == 0){
6478     end_op();
6479     return -1;
6480   }
6481   iunlockput(ip);
6482   end_op();
6483   return 0;
6484 }
6485
6486
6487
6488
6489
6490
6491
6492
6493
6494
6495
6496
6497
6498
6499
```



```
6500 int
6501 sys_chdir(void)
6502 {
6503   char *path;
6504   struct inode *ip;
6505
6506   begin_op();
6507   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
6508     end_op();
6509     return -1;
6510   }
6511   ilock(ip);
6512   if(ip->type != T_DIR){
6513     iunlockput(ip);
6514     end_op();
6515     return -1;
6516   }
6517   iunlock(ip);
6518   iput(proc->cwd);
6519   end_op();
6520   proc->cwd = ip;
6521   return 0;
6522 }
6523
6524 int
6525 sys_exec(void)
6526 {
6527   char *path, *argv[MAXARG];
6528   int i;
6529   uint uargv, uarg;
6530
6531   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6532     return -1;
6533   }
6534   memset(argv, 0, sizeof(argv));
6535   for(i=0;; i++){
6536     if(i >= NELEM(argv))
6537       return -1;
6538     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6539       return -1;
6540     if(uarg == 0){
6541       argv[i] = 0;
6542       break;
6543     }
6544     if(fetchstr(uarg, &argv[i]) < 0)
6545       return -1;
6546   }
6547   return exec(path, argv);
6548 }
6549
```

```
6550 int
6551 sys_pipe(void)
6552 {
6553   int *fd;
6554   struct file *rf, *wf;
6555   int fd0, fd1;
6556
6557   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6558     return -1;
6559   if(pipealloc(&rf, &wf) < 0)
6560     return -1;
6561   fd0 = -1;
6562   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6563     if(fd0 >= 0)
6564       proc->ofile[fd0] = 0;
6565     fileclose(rf);
6566     fileclose(wf);
6567     return -1;
6568   }
6569   fd[0] = fd0;
6570   fd[1] = fd1;
6571   return 0;
6572 }
6573
6574
6575
6576
6577
6578
6579
6580
6581
6582
6583
6584
6585
6586
6587
6588
6589
6590
6591
6592
6593
6594
6595
6596
6597
6598
6599
```



```
6600 #include "types.h"
6601 #include "param.h"
6602 #include "memlayout.h"
6603 #include "mmu.h"
6604 #include "proc.h"
6605 #include "defs.h"
6606 #include "x86.h"
6607 #include "elf.h"
6608
6609 int
6610 exec(char *path, char **argv)
6611 {
6612   char *s, *last;
6613   int i, off;
6614   uint argc, sz, sp, ustack[3+MAXARG+1];
6615   struct elfhdr elf;
6616   struct inode *ip;
6617   struct proghdr ph;
6618   pde_t *pgdir, *oldpgdir;
6619
6620   begin_op();
6621
6622   if((ip = namei(path)) == 0){
6623     end_op();
6624     return -1;
6625   }
6626   ilock(ip);
6627   pgdir = 0;
6628
6629   // Check ELF header
6630   if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
6631     goto bad;
6632   if(elf.magic != ELF_MAGIC)
6633     goto bad;
6634
6635   if((pgdir = setupkvm()) == 0)
6636     goto bad;
6637
6638   // Load program into memory.
6639   sz = 0;
6640   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6641     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6642       goto bad;
6643     if(ph.type != ELF_PROG_LOAD)
6644       continue;
6645     if(ph.memsz < ph.filesz)
6646       goto bad;
6647     if(ph.vaddr + ph.memsz < ph.vaddr)
6648       goto bad;
6649     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
```

```
6650      goto bad;
6651    if(ph.vaddr % PGSIZE != 0)
6652      goto bad;
6653    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6654      goto bad;
6655  }
6656  iunlockput(ip);
6657  end_op();
6658  ip = 0;
6659
6660  // Allocate two pages at the next page boundary.
6661  // Make the first inaccessible.  Use the second as the user stack.
6662  sz = PGROUNDUP(sz);
6663  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
6664    goto bad;
6665  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6666  sp = sz;
6667
6668  // Push argument strings, prepare rest of stack in ustack.
6669  for(argc = 0; argv[argc]; argc++) {
6670    if(argc >= MAXARG)
6671      goto bad;
6672    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6673    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6674      goto bad;
6675    ustack[3+argc] = sp;
6676  }
6677  ustack[3+argc] = 0;
6678
6679  ustack[0] = 0xffffffff;  // fake return PC
6680  ustack[1] = argc;
6681  ustack[2] = sp - (argc+1)*4;  // argv pointer
6682
6683  sp -= (3+argc+1) * 4;
6684  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6685    goto bad;
6686
6687  // Save program name for debugging.
6688  for(last=s=path; *s; s++)
6689    if(*s == '/')
6690      last = s+1;
6691  safestrcpy(proc->name, last, sizeof(proc->name));
6692
6693  // Commit to the user image.
6694  oldpgdir = proc->pgdir;
6695  proc->pgdir = pgdir;
6696  proc->sz = sz;
6697  proc->tf->eip = elf.entry;  // main
6698  proc->tf->esp = sp;
6699  switchuvm(proc);
```



```
6700  freevm(oldpgdir);
6701  return 0;
6702
6703 bad:
6704  if(pgdir)
6705    freevm(pgdir);
6706  if(ip){
6707    iunlockput(ip);
6708    end_op();
6709  }
6710  return -1;
6711 }
6712
6713
6714
6715
6716
6717
6718
6719
6720
6721
6722
6723
6724
6725
6726
6727
6728
6729
6730
6731
6732
6733
6734
6735
6736
6737
6738
6739
6740
6741
6742
6743
6744
6745
6746
6747
6748
6749
```

```
6750 #include "types.h"
6751 #include "defs.h"
6752 #include "param.h"
6753 #include "mmu.h"
6754 #include "proc.h"
6755 #include "fs.h"
6756 #include "spinlock.h"
6757 #include "sleeplock.h"
6758 #include "file.h"
6759
6760 #define PIPESIZE 512
6761
6762 struct pipe {
6763   struct spinlock lock;
6764   char data[PIPESIZE];
6765   uint nread;     // number of bytes read
6766   uint nwrite;    // number of bytes written
6767   int readopen;   // read fd is still open
6768   int writeopen;  // write fd is still open
6769 };
6770
6771 int
6772 pipealloc(struct file **f0, struct file **f1)
6773 {
6774   struct pipe *p;
6775
6776   p = 0;
6777   *f0 = *f1 = 0;
6778   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
6779     goto bad;
6780   if((p = (struct pipe*)kalloc()) == 0)
6781     goto bad;
6782   p->readopen = 1;
6783   p->writeopen = 1;
6784   p->nwrite = 0;
6785   p->nread = 0;
6786   initlock(&p->lock, "pipe");
6787   (*f0)->type = FD_PIPE;
6788   (*f0)->readable = 1;
6789   (*f0)->writable = 0;
6790   (*f0)->pipe = p;
6791   (*f1)->type = FD_PIPE;
6792   (*f1)->readable = 0;
6793   (*f1)->writable = 1;
6794   (*f1)->pipe = p;
6795   return 0;
6796
6797
6798
6799
```



```
6800  bad:
6801   if(p)
6802     kfree((char*)p);
6803   if(*f0)
6804     fileclose(*f0);
6805   if(*f1)
6806     fileclose(*f1);
6807   return -1;
6808 }
6809
6810 void
6811 pipeclose(struct pipe *p, int writable)
6812 {
6813   acquire(&p->lock);
6814   if(writable){
6815     p->writeopen = 0;
6816     wakeup(&p->nread);
6817   } else {
6818     p->readopen = 0;
6819     wakeup(&p->nwrite);
6820   }
6821   if(p->readopen == 0 && p->writeopen == 0){
6822     release(&p->lock);
6823     kfree((char*)p);
6824   } else
6825     release(&p->lock);
6826 }
6827
6828
6829 int
6830 pipewrite(struct pipe *p, char *addr, int n)
6831 {
6832   int i;
6833
6834   acquire(&p->lock);
6835   for(i = 0; i < n; i++){
6836     while(p->nwrite == p->nread + PIPESIZE){
6837       if(p->readopen == 0 || proc->killed){
6838         release(&p->lock);
6839         return -1;
6840       }
6841       wakeup(&p->nread);
6842       sleep(&p->nwrite, &p->lock);
6843     }
6844     p->data[p->nwrite++ % PIPESIZE] = addr[i];
6845   }
6846   wakeup(&p->nread);
6847   release(&p->lock);
6848   return n;
6849 }
```

```
6850 int
6851 piperead(struct pipe *p, char *addr, int n)
6852 {
6853   int i;
6854
6855   acquire(&p->lock);
6856   while(p->nread == p->nwrite && p->writeopen){
6857     if(proc->killed){
6858       release(&p->lock);
6859       return -1;
6860     }
6861     sleep(&p->nread, &p->lock);
6862   }
6863   for(i = 0; i < n; i++){
6864     if(p->nread == p->nwrite)
6865       break;
6866     addr[i] = p->data[p->nread++ % PIPESIZE];
6867   }
6868   wakeup(&p->nwrite);
6869   release(&p->lock);
6870   return i;
6871 }
6872
6873
6874
6875
6876
6877
6878
6879
6880
6881
6882
6883
6884
6885
6886
6887
6888
6889
6890
6891
6892
6893
6894
6895
6896
6897
6898
6899
```



```
6900 #include "types.h"
6901 #include "x86.h"
6902
6903 void*
6904 memset(void *dst, int c, uint n)
6905 {
6906   if ((int)dst%4 == 0 && n%4 == 0){
6907     c &= 0xFF;
6908     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
6909   } else
6910     stosb(dst, c, n);
6911   return dst;
6912 }
6913
6914 int
6915 memcmp(const void *v1, const void *v2, uint n)
6916 {
6917   const uchar *s1, *s2;
6918
6919   s1 = v1;
6920   s2 = v2;
6921   while(n-- > 0){
6922     if(*s1 != *s2)
6923       return *s1 - *s2;
6924     s1++, s2++;
6925   }
6926
6927   return 0;
6928 }
6929
6930 void*
6931 memmove(void *dst, const void *src, uint n)
6932 {
6933   const char *s;
6934   char *d;
6935
6936   s = src;
6937   d = dst;
6938   if(s < d && s + n > d){
6939     s += n;
6940     d += n;
6941     while(n-- > 0)
6942       *--d = *--s;
6943   } else
6944     while(n-- > 0)
6945       *d++ = *s++;
6946
6947   return dst;
6948 }
6949
```

```
6950 // memcpy exists to placate GCC.  Use memmove.
6951 void*
6952 memcpy(void *dst, const void *src, uint n)
6953 {
6954   return memmove(dst, src, n);
6955 }
6956
6957 int
6958 strncmp(const char *p, const char *q, uint n)
6959 {
6960   while(n > 0 && *p && *p == *q)
6961     n--, p++, q++;
6962   if(n == 0)
6963     return 0;
6964   return (uchar)*p - (uchar)*q;
6965 }
6966
6967 char*
6968 strncpy(char *s, const char *t, int n)
6969 {
6970   char *os;
6971
6972   os = s;
6973   while(n-- > 0 && (*s++ = *t++) != 0)
6974     ;
6975   while(n-- > 0)
6976     *s++ = 0;
6977   return os;
6978 }
6979
6980 // Like strncpy but guaranteed to NUL-terminate.
6981 char*
6982 safestrcpy(char *s, const char *t, int n)
6983 {
6984   char *os;
6985
6986   os = s;
6987   if(n <= 0)
6988     return os;
6989   while(--n > 0 && (*s++ = *t++) != 0)
6990     ;
6991   *s = 0;
6992   return os;
6993 }
6994
6995
6996
6997
6998
6999
```



```
7000 int
7001 strlen(const char *s)
7002 {
7003   int n;
7004
7005   for(n = 0; s[n]; n++)
7006     ;
7007   return n;
7008 }
7009
7010
7011
7012
7013
7014
7015
7016
7017
7018
7019
7020
7021
7022
7023
7024
7025
7026
7027
7028
7029
7030
7031
7032
7033
7034
7035
7036
7037
7038
7039
7040
7041
7042
7043
7044
7045
7046
7047
7048
7049
```

```
// See MultiProcessor Specification Version 1.[14]

struct mp {             // floating pointer
  uchar signature[4];           // "_MP_"
  void *physaddr;               // phys addr of MP config table
  uchar length;                 // 1
  uchar specrev;                // [14]
  uchar checksum;               // all bytes must add up to 0
  uchar type;                   // MP system config type
  uchar imcrp;
  uchar reserved[3];
};

struct mpconf {         // configuration table header
  uchar signature[4];           // "PCMP"
  ushort length;                // total table length
  uchar version;                // [14]
  uchar checksum;               // all bytes must add up to 0
  uchar product[20];            // product id
  uint *oemtable;               // OEM table pointer
  ushort oemlength;             // OEM table length
  ushort entry;                 // entry count
  uint *lapicaddr;              // address of local APIC
  ushort xlength;               // extended table length
  uchar xchecksum;              // extended table checksum
  uchar reserved;
};

struct mpproc {         // processor table entry
  uchar type;                   // entry type (0)
  uchar apicid;                 // local APIC id
  uchar version;                // local APIC verison
  uchar flags;                  // CPU flags
    #define MPBOOT 0x02           // This proc is the bootstrap processor.
  uchar signature[4];           // CPU signature
  uint feature;                 // feature flags from CPUID instruction
  uchar reserved[8];
};

struct mpioapic {       // I/O APIC table entry
  uchar type;                   // entry type (2)
  uchar apicno;                 // I/O APIC id
  uchar version;                // I/O APIC version
  uchar flags;                  // I/O APIC flags
  uint *addr;                  // I/O APIC address
};
```



```
// Table entry types
#define MPPROC    0x00  // One per processor
#define MPBUS     0x01  // One per bus
#define MPIOAPIC  0x02  // One per I/O APIC
#define MPIOINTR  0x03  // One per bus interrupt source
#define MPLINTR   0x04  // One per system interrupt source
```

```
7150 // Blank page.
7151
7152
7153
7154
7155
7156
7157
7158
7159
7160
7161
7162
7163
7164
7165
7166
7167
7168
7169
7170
7171
7172
7173
7174
7175
7176
7177
7178
7179
7180
7181
7182
7183
7184
7185
7186
7187
7188
7189
7190
7191
7192
7193
7194
7195
7196
7197
7198
7199
```



```
7200 // Multiprocessor support
7201 // Search memory for MP description structures.
7202 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
7203
7204 #include "types.h"
7205 #include "defs.h"
7206 #include "param.h"
7207 #include "memlayout.h"
7208 #include "mp.h"
7209 #include "x86.h"
7210 #include "mmu.h"
7211 #include "proc.h"
7212
7213 struct cpu cpus[NCPU];
7214 int ismp;
7215 int ncpu;
7216 uchar ioapicid;
7217
7218 static uchar
7219 sum(uchar *addr, int len)
7220 {
7221   int i, sum;
7222
7223   sum = 0;
7224   for(i=0; i<len; i++)
7225     sum += addr[i];
7226   return sum;
7227 }
7228
7229 // Look for an MP structure in the len bytes at addr.
7230 static struct mp*
7231 mpsearch1(uint a, int len)
7232 {
7233   uchar *e, *p, *addr;
7234
7235   addr = P2V(a);
7236   e = addr+len;
7237   for(p = addr; p < e; p += sizeof(struct mp))
7238     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
7239       return (struct mp*)p;
7240   return 0;
7241 }
7242
7243
7244
7245
7246
7247
7248
7249
```

```
7250 // Search for the MP Floating Pointer Structure, which according to the
7251 // spec is in one of the following three locations:
7252 // 1) in the first KB of the EBDA;
7253 // 2) in the last KB of system base memory;
7254 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
7255 static struct mp*
7256 mpsearch(void)
7257 {
7258   uchar *bda;
7259   uint p;
7260   struct mp *mp;
7261
7262   bda = (uchar *) P2V(0x400);
7263   if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
7264     if((mp = mpsearch1(p, 1024)))
7265       return mp;
7266   } else {
7267     p = ((bda[0x14]<<8)|bda[0x13])*1024;
7268     if((mp = mpsearch1(p-1024, 1024)))
7269       return mp;
7270   }
7271   return mpsearch1(0xF0000, 0x10000);
7272 }
7273
7274 // Search for an MP configuration table.  For now,
7275 // don't accept the default configurations (physaddr == 0).
7276 // Check for correct signature, calculate the checksum and,
7277 // if correct, check the version.
7278 // To do: check extended table checksum.
7279 static struct mpconf*
7280 mpconfig(struct mp **pmp)
7281 {
7282   struct mpconf *conf;
7283   struct mp *mp;
7284
7285   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
7286     return 0;
7287   conf = (struct mpconf*) P2V((uint) mp->physaddr);
7288   if(memcmp(conf, "PCMP", 4) != 0)
7289     return 0;
7290   if(conf->version != 1 && conf->version != 4)
7291     return 0;
7292   if(sum((uchar*)conf, conf->length) != 0)
7293     return 0;
7294   *pmp = mp;
7295   return conf;
7296 }
7297
7298
7299
```



```
7300 void
7301 mpinit(void)
7302 {
7303   uchar *p, *e;
7304   struct mp *mp;
7305   struct mpconf *conf;
7306   struct mpproc *proc;
7307   struct mpioapic *ioapic;
7308
7309   if((conf = mpconfig(&mp)) == 0)
7310     return;
7311   ismp = 1;
7312   lapic = (uint*)conf->lapicaddr;
7313   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
7314     switch(*p){
7315     case MPPROC:
7316       proc = (struct mpproc*)p;
7317       if(ncpu < NCPU) {
7318         cpus[ncpu].apicid = proc->apicid;  // apicid may differ from ncpu
7319         ncpu++;
7320       }
7321       p += sizeof(struct mpproc);
7322       continue;
7323     case MPIOAPIC:
7324       ioapic = (struct mpioapic*)p;
7325       ioapicid = ioapic->apicno;
7326       p += sizeof(struct mpioapic);
7327       continue;
7328     case MPBUS:
7329     case MPIOINTR:
7330     case MPLINTR:
7331       p += 8;
7332       continue;
7333     default:
7334       ismp = 0;
7335       break;
7336     }
7337   }
7338   if(!ismp){
7339     // Didn't like what we found; fall back to no MP.
7340     ncpu = 1;
7341     lapic = 0;
7342     ioapicid = 0;
7343     return;
7344   }
7345
7346
7347
7348
7349
```

```
7350   if(mp->imcrp){
7351     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
7352     // But it would on real hardware.
7353     outb(0x22, 0x70);   // Select IMCR
7354     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
7355   }
7356 }
7357
7358
7359
7360
7361
7362
7363
7364
7365
7366
7367
7368
7369
7370
7371
7372
7373
7374
7375
7376
7377
7378
7379
7380
7381
7382
7383
7384
7385
7386
7387
7388
7389
7390
7391
7392
7393
7394
7395
7396
7397
7398
7399
```



```
7400 // The local APIC manages internal (non-I/O) interrupts.
7401 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
7402
7403 #include "param.h"
7404 #include "types.h"
7405 #include "defs.h"
7406 #include "date.h"
7407 #include "memlayout.h"
7408 #include "traps.h"
7409 #include "mmu.h"
7410 #include "x86.h"
7411 #include "proc.h"  // ncpu
7412
7413 // Local APIC registers, divided by 4 for use as uint[] indices.
7414 #define ID      (0x0020/4)   // ID
7415 #define VER     (0x0030/4)   // Version
7416 #define TPR     (0x0080/4)   // Task Priority
7417 #define EOI     (0x00B0/4)   // EOI
7418 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
7419   #define ENABLE     0x00000100   // Unit Enable
7420 #define ESR     (0x0280/4)   // Error Status
7421 #define ICRLO   (0x0300/4)   // Interrupt Command
7422   #define INIT       0x00000500   // INIT/RESET
7423   #define STARTUP    0x00000600   // Startup IPI
7424   #define DELIVS     0x00001000   // Delivery status
7425   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
7426   #define DEASSERT   0x00000000
7427   #define LEVEL      0x00008000   // Level triggered
7428   #define BCAST      0x00080000   // Send to all APICs, including self.
7429   #define BUSY       0x00001000
7430   #define FIXED      0x00000000
7431 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
7432 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
7433   #define X1         0x0000000B   // divide counts by 1
7434   #define PERIODIC   0x00020000   // Periodic
7435 #define PCINT   (0x0340/4)   // Performance Counter LVT
7436 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
7437 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
7438 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
7439   #define MASKED     0x00010000   // Interrupt masked
7440 #define TICR    (0x0380/4)   // Timer Initial Count
7441 #define TCCR    (0x0390/4)   // Timer Current Count
7442 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
7443
7444 volatile uint *lapic;  // Initialized in mp.c
7445
7446
7447
7448
7449
```

```
7450 static void
7451 lapicw(int index, int value)
7452 {
7453   lapic[index] = value;
7454   lapic[ID];  // wait for write to finish, by reading
7455 }
7456
7457
7458
7459
7460
7461
7462
7463
7464
7465
7466
7467
7468
7469
7470
7471
7472
7473
7474
7475
7476
7477
7478
7479
7480
7481
7482
7483
7484
7485
7486
7487
7488
7489
7490
7491
7492
7493
7494
7495
7496
7497
7498
7499
```



```
7500 void
7501 lapicinit(void)
7502 {
7503   if(!lapic)
7504     return;
7505
7506   // Enable local APIC; set spurious interrupt vector.
7507   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
7508
7509   // The timer repeatedly counts down at bus frequency
7510   // from lapic[TICR] and then issues an interrupt.
7511   // If xv6 cared more about precise timekeeping,
7512   // TICR would be calibrated using an external time source.
7513   lapicw(TDCR, X1);
7514   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
7515   lapicw(TICR, 10000000);
7516
7517   // Disable logical interrupt lines.
7518   lapicw(LINT0, MASKED);
7519   lapicw(LINT1, MASKED);
7520
7521   // Disable performance counter overflow interrupts
7522   // on machines that provide that interrupt entry.
7523   if(((lapic[VER]>>16) & 0xFF) >= 4)
7524     lapicw(PCINT, MASKED);
7525
7526   // Map error interrupt to IRQ_ERROR.
7527   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
7528
7529   // Clear error status register (requires back-to-back writes).
7530   lapicw(ESR, 0);
7531   lapicw(ESR, 0);
7532
7533   // Ack any outstanding interrupts.
7534   lapicw(EOI, 0);
7535
7536   // Send an Init Level De-Assert to synchronise arbitration ID's.
7537   lapicw(ICRHI, 0);
7538   lapicw(ICRLO, BCAST | INIT | LEVEL);
7539   while(lapic[ICRLO] & DELIVS)
7540     ;
7541
7542   // Enable interrupts on the APIC (but not on the processor).
7543   lapicw(TPR, 0);
7544 }
7545
7546
7547
7548
7549
```

```
int
cpunum(void)
{
  int apicid, i;

  // Cannot call cpu when interrupts are enabled:
  // result not guaranteed to last long enough to be used!
  // Would prefer to panic but even printing is chancy here:
  // almost everything, including cprintf and panic, calls cpu,
  // often indirectly through acquire and release.
  if(readeflags()&FL_IF){
    static int n;
    if(n++ == 0)
      cprintf("cpu called from %x with interrupts enabled\n",
        __builtin_return_address(0));
  }

  if (!lapic)
    return 0;

  apicid = lapic[ID] >> 24;
  for (i = 0; i < ncpu; ++i) {
    if (cpus[i].apicid == apicid)
      return i;
  }
  panic("unknown apicid\n");
}

// Acknowledge interrupt.
void
lapiceoi(void)
{
  if(lapic)
    lapicw(EOI, 0);
}

// Spin for a given number of microseconds.
// On real hardware would want to tune this dynamically.
void
microdelay(int us)
{
}
```



```
#define CMOS_PORT    0x70
#define CMOS_RETURN  0x71

// Start additional processor running entry code at addr.
// See Appendix B of MultiProcessor Specification.
void
lapicstartap(uchar apicid, uint addr)
{
  int i;
  ushort *wrv;

  // "The BSP must initialize CMOS shutdown code to 0AH
  // and the warm reset vector (DWORD based at 40:67) to point at
  // the AP startup code prior to the [universal startup algorithm]."
  outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
  outb(CMOS_PORT+1, 0x0A);
  wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
  wrv[0] = 0;
  wrv[1] = addr >> 4;

  // "Universal startup algorithm."
  // Send INIT (level-triggered) interrupt to reset other CPU.
  lapicw(ICRHI, apicid<<24);
  lapicw(ICRLO, INIT | LEVEL | ASSERT);
  microdelay(200);
  lapicw(ICRLO, INIT | LEVEL);
  microdelay(100);    // should be 10ms, but too slow in Bochs!

  // Send startup IPI (twice!) to enter code.
  // Regular hardware is supposed to only accept a STARTUP
  // when it is in the halted state due to an INIT.  So the second
  // should be ignored, but it is part of the official Intel algorithm.
  // Bochs complains about the second one.  Too bad for Bochs.
  for(i = 0; i < 2; i++){
    lapicw(ICRHI, apicid<<24);
    lapicw(ICRLO, STARTUP | (addr>>12));
    microdelay(200);
  }
}
```

```
#define CMOS_STATA   0x0a
#define CMOS_STATB   0x0b
#define CMOS_UIP    (1 << 7)        // RTC update in progress

#define SECS    0x00
#define MINS    0x02
#define HOURS   0x04
#define DAY     0x07
#define MONTH   0x08
#define YEAR    0x09

static uint cmos_read(uint reg)
{
  outb(CMOS_PORT,  reg);
  microdelay(200);

  return inb(CMOS_RETURN);
}

static void fill_rtcdate(struct rtcdate *r)
{
  r->second = cmos_read(SECS);
  r->minute = cmos_read(MINS);
  r->hour   = cmos_read(HOURS);
  r->day    = cmos_read(DAY);
  r->month  = cmos_read(MONTH);
  r->year   = cmos_read(YEAR);
}

// qemu seems to use 24-hour GWT and the values are BCD encoded
void cmostime(struct rtcdate *r)
{
  struct rtcdate t1, t2;
  int sb, bcd;

  sb = cmos_read(CMOS_STATB);

  bcd = (sb & (1 << 2)) == 0;

  // make sure CMOS doesn't modify time while we read it
  for(;;) {
    fill_rtcdate(&t1);
    if(cmos_read(CMOS_STATA) & CMOS_UIP)
        continue;
    fill_rtcdate(&t2);
    if(memcmp(&t1, &t2, sizeof(t1)) == 0)
      break;
  }

```



```
  // convert
  if(bcd) {
#define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
    CONV(second);
    CONV(minute);
    CONV(hour  );
    CONV(day   );
    CONV(month );
    CONV(year  );
#undef     CONV
  }

  *r = t1;
  r->year += 2000;
}
```

```
7750 // The I/O APIC manages hardware interrupts for an SMP system.
7751 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
7752 // See also picirq.c.
7753
7754 #include "types.h"
7755 #include "defs.h"
7756 #include "traps.h"
7757
7758 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
7759
7760 #define REG_ID     0x00  // Register index: ID
7761 #define REG_VER    0x01  // Register index: version
7762 #define REG_TABLE  0x10  // Redirection table base
7763
7764 // The redirection table starts at REG_TABLE and uses
7765 // two registers to configure each interrupt.
7766 // The first (low) register in a pair contains configuration bits.
7767 // The second (high) register contains a bitmask telling which
7768 // CPUs can serve that interrupt.
7769 #define INT_DISABLED   0x00010000  // Interrupt disabled
7770 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
7771 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
7772 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
7773
7774 volatile struct ioapic *ioapic;
7775
7776 // IO APIC MMIO structure: write reg, then read or write data.
7777 struct ioapic {
7778   uint reg;
7779   uint pad[3];
7780   uint data;
7781 };
7782
7783 static uint
7784 ioapicread(int reg)
7785 {
7786   ioapic->reg = reg;
7787   return ioapic->data;
7788 }
7789
7790 static void
7791 ioapicwrite(int reg, uint data)
7792 {
7793   ioapic->reg = reg;
7794   ioapic->data = data;
7795 }
7796
7797
7798
7799
```



```
7800 void
7801 ioapicinit(void)
7802 {
7803   int i, id, maxintr;
7804
7805   if(!ismp)
7806     return;
7807
7808   ioapic = (volatile struct ioapic*)IOAPIC;
7809   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
7810   id = ioapicread(REG_ID) >> 24;
7811   if(id != ioapicid)
7812     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
7813
7814   // Mark all interrupts edge-triggered, active high, disabled,
7815   // and not routed to any CPUs.
7816   for(i = 0; i <= maxintr; i++){
7817     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
7818     ioapicwrite(REG_TABLE+2*i+1, 0);
7819   }
7820 }
7821
7822 void
7823 ioapicenable(int irq, int cpunum)
7824 {
7825   if(!ismp)
7826     return;
7827
7828   // Mark interrupt edge-triggered, active high,
7829   // enabled, and routed to the given cpunum,
7830   // which happens to be that cpu's APIC ID.
7831   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
7832   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
7833 }
7834
7835
7836
7837
7838
7839
7840
7841
7842
7843
7844
7845
7846
7847
7848
7849
```

```
7850 // Intel 8259A programmable interrupt controllers.
7851
7852 #include "types.h"
7853 #include "x86.h"
7854 #include "traps.h"
7855
7856 // I/O Addresses of the two programmable interrupt controllers
7857 #define IO_PIC1         0x20    // Master (IRQs 0-7)
7858 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
7859
7860 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
7861
7862 // Current IRQ mask.
7863 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
7864 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
7865
7866 static void
7867 picsetmask(ushort mask)
7868 {
7869   irqmask = mask;
7870   outb(IO_PIC1+1, mask);
7871   outb(IO_PIC2+1, mask >> 8);
7872 }
7873
7874 void
7875 picenable(int irq)
7876 {
7877   picsetmask(irqmask & ~(1<<irq));
7878 }
7879
7880 // Initialize the 8259A interrupt controllers.
7881 void
7882 picinit(void)
7883 {
7884   // mask all interrupts
7885   outb(IO_PIC1+1, 0xFF);
7886   outb(IO_PIC2+1, 0xFF);
7887
7888   // Set up master (8259A-1)
7889
7890   // ICW1:  0001g0hi
7891   //    g:  0 = edge triggering, 1 = level triggering
7892   //    h:  0 = cascaded PICs, 1 = master only
7893   //    i:  0 = no ICW4, 1 = ICW4 required
7894   outb(IO_PIC1, 0x11);
7895
7896   // ICW2:  Vector offset
7897   outb(IO_PIC1+1, T_IRQ0);
7898
7899
```



```
7900   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
7901   //        (slave PIC) 3-bit # of slave's connection to master
7902   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
7903
7904   // ICW4:  000nbmap
7905   //    n:  1 = special fully nested mode
7906   //    b:  1 = buffered mode
7907   //    m:  0 = slave PIC, 1 = master PIC
7908   //      (ignored when b is 0, as the master/slave role
7909   //      can be hardwired).
7910   //    a:  1 = Automatic EOI mode
7911   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
7912   outb(IO_PIC1+1, 0x3);
7913
7914   // Set up slave (8259A-2)
7915   outb(IO_PIC2, 0x11);                  // ICW1
7916   outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
7917   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
7918   // NB Automatic EOI mode doesn't tend to work on the slave.
7919   // Linux source code says it's "to be investigated".
7920   outb(IO_PIC2+1, 0x3);                 // ICW4
7921
7922   // OCW3:  0ef01prs
7923   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
7924   //    p:  0 = no polling, 1 = polling mode
7925   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
7926   outb(IO_PIC1, 0x68);             // clear specific mask
7927   outb(IO_PIC1, 0x0a);             // read IRR by default
7928
7929   outb(IO_PIC2, 0x68);             // OCW3
7930   outb(IO_PIC2, 0x0a);             // OCW3
7931
7932   if(irqmask != 0xFFFF)
7933     picsetmask(irqmask);
7934 }
7935
7936
7937
7938
7939
7940
7941
7942
7943
7944
7945
7946
7947
7948
7949
```

```
7950 // Blank page.
7951
7952
7953
7954
7955
7956
7957
7958
7959
7960
7961
7962
7963
7964
7965
7966
7967
7968
7969
7970
7971
7972
7973
7974
7975
7976
7977
7978
7979
7980
7981
7982
7983
7984
7985
7986
7987
7988
7989
7990
7991
7992
7993
7994
7995
7996
7997
7998
7999
```



```
8000 // PC keyboard interface constants
8001
8002 #define KBSTATP         0x64    // kbd controller status port(I)
8003 #define KBS_DIB         0x01    // kbd data in buffer
8004 #define KBDATAP         0x60    // kbd data port(I)
8005
8006 #define NO              0
8007
8008 #define SHIFT           (1<<0)
8009 #define CTL             (1<<1)
8010 #define ALT             (1<<2)
8011
8012 #define CAPSLOCK        (1<<3)
8013 #define NUMLOCK         (1<<4)
8014 #define SCROLLLOCK      (1<<5)
8015
8016 #define E0ESC           (1<<6)
8017
8018 // Special keycodes
8019 #define KEY_HOME        0xE0
8020 #define KEY_END         0xE1
8021 #define KEY_UP          0xE2
8022 #define KEY_DN          0xE3
8023 #define KEY_LF          0xE4
8024 #define KEY_RT          0xE5
8025 #define KEY_PGUP        0xE6
8026 #define KEY_PGDN        0xE7
8027 #define KEY_INS         0xE8
8028 #define KEY_DEL         0xE9
8029
8030 // C('A') == Control-A
8031 #define C(x) (x - '@')
8032
8033 static uchar shiftcode[256] =
8034 {
8035   [0x1D] CTL,
8036   [0x2A] SHIFT,
8037   [0x36] SHIFT,
8038   [0x38] ALT,
8039   [0x9D] CTL,
8040   [0xB8] ALT
8041 };
8042
8043 static uchar togglecode[256] =
8044 {
8045   [0x3A] CAPSLOCK,
8046   [0x45] NUMLOCK,
8047   [0x46] SCROLLLOCK
8048 };
8049
```

```
8050 static uchar normalmap[256] =
8051 {
8052   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
8053   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
8054   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
8055   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
8056   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
8057   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
8058   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
8059   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
8060   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
8061   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
8062   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
8063   [0x9C] '\n',      // KP_Enter
8064   [0xB5] '/',       // KP_Div
8065   [0xC8] KEY_UP,    [0xD0] KEY_DN,
8066   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
8067   [0xCB] KEY_LF,    [0xCD] KEY_RT,
8068   [0x97] KEY_HOME,  [0xCF] KEY_END,
8069   [0xD2] KEY_INS,   [0xD3] KEY_DEL
8070 };
8071
8072 static uchar shiftmap[256] =
8073 {
8074   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
8075   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
8076   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
8077   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
8078   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
8079   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
8080   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
8081   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
8082   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
8083   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
8084   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
8085   [0x9C] '\n',      // KP_Enter
8086   [0xB5] '/',       // KP_Div
8087   [0xC8] KEY_UP,    [0xD0] KEY_DN,
8088   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
8089   [0xCB] KEY_LF,    [0xCD] KEY_RT,
8090   [0x97] KEY_HOME,  [0xCF] KEY_END,
8091   [0xD2] KEY_INS,   [0xD3] KEY_DEL
8092 };
8093
8094
8095
8096
8097
8098
8099
```



```
8100 static uchar ctlmap[256] =
8101 {
8102   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
8103   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
8104   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
8105   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
8106   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
8107   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
8108   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
8109   [0x9C] '\r',      // KP_Enter
8110   [0xB5] C('/'),    // KP_Div
8111   [0xC8] KEY_UP,    [0xD0] KEY_DN,
8112   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
8113   [0xCB] KEY_LF,    [0xCD] KEY_RT,
8114   [0x97] KEY_HOME,  [0xCF] KEY_END,
8115   [0xD2] KEY_INS,   [0xD3] KEY_DEL
8116 };
8117
8118
8119
8120
8121
8122
8123
8124
8125
8126
8127
8128
8129
8130
8131
8132
8133
8134
8135
8136
8137
8138
8139
8140
8141
8142
8143
8144
8145
8146
8147
8148
8149
```

```
8150 #include "types.h"
8151 #include "x86.h"
8152 #include "defs.h"
8153 #include "kbd.h"
8154
8155 int
8156 kbdgetc(void)
8157 {
8158   static uint shift;
8159   static uchar *charcode[4] = {
8160     normalmap, shiftmap, ctlmap, ctlmap
8161   };
8162   uint st, data, c;
8163
8164   st = inb(KBSTATP);
8165   if((st & KBS_DIB) == 0)
8166     return -1;
8167   data = inb(KBDATAP);
8168
8169   if(data == 0xE0){
8170     shift |= E0ESC;
8171     return 0;
8172   } else if(data & 0x80){
8173     // Key released
8174     data = (shift & E0ESC ? data : data & 0x7F);
8175     shift &= ~(shiftcode[data] | E0ESC);
8176     return 0;
8177   } else if(shift & E0ESC){
8178     // Last character was an E0 escape; or with 0x80
8179     data |= 0x80;
8180     shift &= ~E0ESC;
8181   }
8182
8183   shift |= shiftcode[data];
8184   shift ^= togglecode[data];
8185   c = charcode[shift & (CTL | SHIFT)][data];
8186   if(shift & CAPSLOCK){
8187     if('a' <= c && c <= 'z')
8188       c += 'A' - 'a';
8189     else if('A' <= c && c <= 'Z')
8190       c += 'a' - 'A';
8191   }
8192   return c;
8193 }
8194
8195 void
8196 kbdintr(void)
8197 {
8198   consoleintr(kbdgetc);
8199 }
```



```
8200 // Console input and output.
8201 // Input is from the keyboard or serial port.
8202 // Output is written to the screen and serial port.
8203
8204 #include "types.h"
8205 #include "defs.h"
8206 #include "param.h"
8207 #include "traps.h"
8208 #include "spinlock.h"
8209 #include "sleeplock.h"
8210 #include "fs.h"
8211 #include "file.h"
8212 #include "memlayout.h"
8213 #include "mmu.h"
8214 #include "proc.h"
8215 #include "x86.h"
8216
8217 static void consputc(int);
8218
8219 static int panicked = 0;
8220
8221 static struct {
8222   struct spinlock lock;
8223   int locking;
8224 } cons;
8225
8226 static void
8227 printint(int xx, int base, int sign)
8228 {
8229   static char digits[] = "0123456789abcdef";
8230   char buf[16];
8231   int i;
8232   uint x;
8233
8234   if(sign && (sign = xx < 0))
8235     x = -xx;
8236   else
8237     x = xx;
8238
8239   i = 0;
8240   do{
8241     buf[i++] = digits[x % base];
8242   }while((x /= base) != 0);
8243
8244   if(sign)
8245     buf[i++] = '-';
8246
8247   while(--i >= 0)
8248     consputc(buf[i]);
8249 }
```

```
8250
8251
8252
8253
8254
8255
8256
8257
8258
8259
8260
8261
8262
8263
8264
8265
8266
8267
8268
8269
8270
8271
8272
8273
8274
8275
8276
8277
8278
8279
8280
8281
8282
8283
8284
8285
8286
8287
8288
8289
8290
8291
8292
8293
8294
8295
8296
8297
8298
8299
```



```
8300 // Print to the console. only understands %d, %x, %p, %s.
8301 void
8302 cprintf(char *fmt, ...)
8303 {
8304   int i, c, locking;
8305   uint *argp;
8306   char *s;
8307
8308   locking = cons.locking;
8309   if(locking)
8310     acquire(&cons.lock);
8311
8312   if (fmt == 0)
8313     panic("null fmt");
8314
8315   argp = (uint*)(void*)(&fmt + 1);
8316   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
8317     if(c != '%'){
8318       consputc(c);
8319       continue;
8320     }
8321     c = fmt[++i] & 0xff;
8322     if(c == 0)
8323       break;
8324     switch(c){
8325     case 'd':
8326       printint(*argp++, 10, 1);
8327       break;
8328     case 'x':
8329     case 'p':
8330       printint(*argp++, 16, 0);
8331       break;
8332     case 's':
8333       if((s = (char*)*argp++) == 0)
8334         s = "(null)";
8335       for(; *s; s++)
8336         consputc(*s);
8337       break;
8338     case '%':
8339       consputc('%');
8340       break;
8341     default:
8342       // Print unknown % sequence to draw attention.
8343       consputc('%');
8344       consputc(c);
8345       break;
8346     }
8347   }
8348
8349
```

```
8350   if(locking)
8351     release(&cons.lock);
8352 }
8353
8354 void
8355 panic(char *s)
8356 {
8357   int i;
8358   uint pcs[10];
8359
8360   cli();
8361   cons.locking = 0;
8362   cprintf("cpu with apicid %d: panic: ", cpu->apicid);
8363   cprintf(s);
8364   cprintf("\n");
8365   getcallerpcs(&s, pcs);
8366   for(i=0; i<10; i++)
8367     cprintf(" %p", pcs[i]);
8368   panicked = 1; // freeze other CPU
8369   for(;;)
8370     ;
8371 }
8372
8373
8374
8375
8376
8377
8378
8379
8380
8381
8382
8383
8384
8385
8386
8387
8388
8389
8390
8391
8392
8393
8394
8395
8396
8397
8398
8399
```



```
8400 #define BACKSPACE 0x100
8401 #define CRTPORT 0x3d4
8402 static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory
8403
8404 static void
8405 cgaputc(int c)
8406 {
8407   int pos;
8408
8409   // Cursor position: col + 80*row.
8410   outb(CRTPORT, 14);
8411   pos = inb(CRTPORT+1) << 8;
8412   outb(CRTPORT, 15);
8413   pos |= inb(CRTPORT+1);
8414
8415   if(c == '\n')
8416     pos += 80 - pos%80;
8417   else if(c == BACKSPACE){
8418     if(pos > 0) --pos;
8419   } else
8420     crt[pos++] = (c&0xff) | 0x0700;  // black on white
8421
8422   if(pos < 0 || pos > 25*80)
8423     panic("pos under/overflow");
8424
8425   if((pos/80) >= 24){  // Scroll up.
8426     memmove(crt, crt+80, sizeof(crt[0])*23*80);
8427     pos -= 80;
8428     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
8429   }
8430
8431   outb(CRTPORT, 14);
8432   outb(CRTPORT+1, pos>>8);
8433   outb(CRTPORT, 15);
8434   outb(CRTPORT+1, pos);
8435   crt[pos] = ' ' | 0x0700;
8436 }
8437
8438
8439
8440
8441
8442
8443
8444
8445
8446
8447
8448
8449
```

```
8450 void
8451 consputc(int c)
8452 {
8453   if(panicked){
8454     cli();
8455     for(;;)
8456       ;
8457   }
8458
8459   if(c == BACKSPACE){
8460     uartputc('\b'); uartputc(' '); uartputc('\b');
8461   } else
8462     uartputc(c);
8463   cgaputc(c);
8464 }
8465
8466 #define INPUT_BUF 128
8467 struct {
8468   char buf[INPUT_BUF];
8469   uint r;  // Read index
8470   uint w;  // Write index
8471   uint e;  // Edit index
8472 } input;
8473
8474 #define C(x)  ((x)-'@')  // Control-x
8475
8476 void
8477 consoleintr(int (*getc)(void))
8478 {
8479   int c, doprocdump = 0;
8480
8481   acquire(&cons.lock);
8482   while((c = getc()) >= 0){
8483     switch(c){
8484     case C('P'):  // Process listing.
8485       // procdump() locks cons.lock indirectly; invoke later
8486       doprocdump = 1;
8487       break;
8488     case C('U'):  // Kill line.
8489       while(input.e != input.w &&
8490             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
8491         input.e--;
8492         consputc(BACKSPACE);
8493       }
8494       break;
8495     case C('H'): case '\x7f':  // Backspace
8496       if(input.e != input.w){
8497         input.e--;
8498         consputc(BACKSPACE);
8499       }
```



```
8500       break;
8501     default:
8502       if(c != 0 && input.e-input.r < INPUT_BUF){
8503         c = (c == '\r') ? '\n' : c;
8504         input.buf[input.e++ % INPUT_BUF] = c;
8505         consputc(c);
8506         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
8507           input.w = input.e;
8508           wakeup(&input.r);
8509         }
8510       }
8511       break;
8512     }
8513   }
8514   release(&cons.lock);
8515   if(doprocdump) {
8516     procdump();  // now call procdump() wo. cons.lock held
8517   }
8518 }
8519
8520 int
8521 consoleread(struct inode *ip, char *dst, int n)
8522 {
8523   uint target;
8524   int c;
8525
8526   iunlock(ip);
8527   target = n;
8528   acquire(&cons.lock);
8529   while(n > 0){
8530     while(input.r == input.w){
8531       if(proc->killed){
8532         release(&cons.lock);
8533         ilock(ip);
8534         return -1;
8535       }
8536       sleep(&input.r, &cons.lock);
8537     }
8538     c = input.buf[input.r++ % INPUT_BUF];
8539     if(c == C('D')){  // EOF
8540       if(n < target){
8541         // Save ^D for next time, to make sure
8542         // caller gets a 0-byte result.
8543         input.r--;
8544       }
8545       break;
8546     }
8547     *dst++ = c;
8548     --n;
8549     if(c == '\n')
```

```
8550       break;
8551   }
8552   release(&cons.lock);
8553   ilock(ip);
8554
8555   return target - n;
8556 }
8557
8558 int
8559 consolewrite(struct inode *ip, char *buf, int n)
8560 {
8561   int i;
8562
8563   iunlock(ip);
8564   acquire(&cons.lock);
8565   for(i = 0; i < n; i++)
8566     consputc(buf[i] & 0xff);
8567   release(&cons.lock);
8568   ilock(ip);
8569
8570   return n;
8571 }
8572
8573 void
8574 consoleinit(void)
8575 {
8576   initlock(&cons.lock, "console");
8577
8578   devsw[CONSOLE].write = consolewrite;
8579   devsw[CONSOLE].read = consoleread;
8580   cons.locking = 1;
8581
8582   picenable(IRQ_KBD);
8583   ioapicenable(IRQ_KBD, 0);
8584 }
8585
8586
8587
8588
8589
8590
8591
8592
8593
8594
8595
8596
8597
8598
8599
```



```
8600 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
8601 // Only used on uniprocessors;
8602 // SMP machines use the local APIC timer.
8603
8604 #include "types.h"
8605 #include "defs.h"
8606 #include "traps.h"
8607 #include "x86.h"
8608
8609 #define IO_TIMER1       0x040           // 8253 Timer #1
8610
8611 // Frequency of all three count-down timers;
8612 // (TIMER_FREQ/freq) is the appropriate count
8613 // to generate a frequency of freq Hz.
8614
8615 #define TIMER_FREQ      1193182
8616 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
8617
8618 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
8619 #define TIMER_SEL0      0x00    // select counter 0
8620 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
8621 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
8622
8623 void
8624 timerinit(void)
8625 {
8626   // Interrupt 100 times/sec.
8627   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
8628   outb(IO_TIMER1, TIMER_DIV(100) % 256);
8629   outb(IO_TIMER1, TIMER_DIV(100) / 256);
8630   picenable(IRQ_TIMER);
8631 }
8632
8633
8634
8635
8636
8637
8638
8639
8640
8641
8642
8643
8644
8645
8646
8647
8648
8649
```

```
8650 // Intel 8250 serial port (UART).
8651
8652 #include "types.h"
8653 #include "defs.h"
8654 #include "param.h"
8655 #include "traps.h"
8656 #include "spinlock.h"
8657 #include "sleeplock.h"
8658 #include "fs.h"
8659 #include "file.h"
8660 #include "mmu.h"
8661 #include "proc.h"
8662 #include "x86.h"
8663
8664 #define COM1    0x3f8
8665
8666 static int uart;    // is there a uart?
8667
8668 void
8669 uartinit(void)
8670 {
8671   char *p;
8672
8673   // Turn off the FIFO
8674   outb(COM1+2, 0);
8675
8676   // 9600 baud, 8 data bits, 1 stop bit, parity off.
8677   outb(COM1+3, 0x80);    // Unlock divisor
8678   outb(COM1+0, 115200/9600);
8679   outb(COM1+1, 0);
8680   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
8681   outb(COM1+4, 0);
8682   outb(COM1+1, 0x01);    // Enable receive interrupts.
8683
8684   // If status is 0xFF, no serial port.
8685   if(inb(COM1+5) == 0xFF)
8686     return;
8687   uart = 1;
8688
8689   // Acknowledge pre-existing interrupt conditions;
8690   // enable interrupts.
8691   inb(COM1+2);
8692   inb(COM1+0);
8693   picenable(IRQ_COM1);
8694   ioapicenable(IRQ_COM1, 0);
8695
8696   // Announce that we're here.
8697   for(p="xv6...\n"; *p; p++)
8698     uartputc(*p);
8699 }
```



```
8700 void
8701 uartputc(int c)
8702 {
8703   int i;
8704
8705   if(!uart)
8706     return;
8707   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
8708     microdelay(10);
8709   outb(COM1+0, c);
8710 }
8711
8712 static int
8713 uartgetc(void)
8714 {
8715   if(!uart)
8716     return -1;
8717   if(!(inb(COM1+5) & 0x01))
8718     return -1;
8719   return inb(COM1+0);
8720 }
8721
8722 void
8723 uartintr(void)
8724 {
8725   consoleintr(uartgetc);
8726 }
8727
8728
8729
8730
8731
8732
8733
8734
8735
8736
8737
8738
8739
8740
8741
8742
8743
8744
8745
8746
8747
8748
8749
```

```
8750 # Initial process execs /init.
8751 # This code runs in user space.
8752
8753 #include "syscall.h"
8754 #include "traps.h"
8755
8756
8757 # exec(init, argv)
8758 .globl start
8759 start:
8760   pushl $argv
8761   pushl $init
8762   pushl $0  // where caller pc would be
8763   movl $SYS_exec, %eax
8764   int $T_SYSCALL
8765
8766 # for(;;) exit();
8767 exit:
8768   movl $SYS_exit, %eax
8769   int $T_SYSCALL
8770   jmp exit
8771
8772 # char init[] = "/init\0";
8773 init:
8774   .string "/init\0"
8775
8776 # char *argv[] = { init, 0 };
8777 .p2align 2
8778 argv:
8779   .long init
8780   .long 0
8781
8782
8783
8784
8785
8786
8787
8788
8789
8790
8791
8792
8793
8794
8795
8796
8797
8798
8799
```



```
8800 #include "syscall.h"
8801 #include "traps.h"
8802
8803 #define SYSCALL(name) \
8804   .globl name; \
8805   name: \
8806     movl $SYS_ ## name, %eax; \
8807     int $T_SYSCALL; \
8808     ret
8809
8810 SYSCALL(fork)
8811 SYSCALL(exit)
8812 SYSCALL(wait)
8813 SYSCALL(pipe)
8814 SYSCALL(read)
8815 SYSCALL(write)
8816 SYSCALL(close)
8817 SYSCALL(kill)
8818 SYSCALL(exec)
8819 SYSCALL(open)
8820 SYSCALL(mknod)
8821 SYSCALL(unlink)
8822 SYSCALL(fstat)
8823 SYSCALL(link)
8824 SYSCALL(mkdir)
8825 SYSCALL(chdir)
8826 SYSCALL(dup)
8827 SYSCALL(getpid)
8828 SYSCALL(sbrk)
8829 SYSCALL(sleep)
8830 SYSCALL(uptime)
8831
8832
8833
8834
8835
8836
8837
8838
8839
8840
8841
8842
8843
8844
8845
8846
8847
8848
8849
```

```
8850 // init: The initial user-level program
8851
8852 #include "types.h"
8853 #include "stat.h"
8854 #include "user.h"
8855 #include "fcntl.h"
8856
8857 char *argv[] = { "sh", 0 };
8858
8859 int
8860 main(void)
8861 {
8862   int pid, wpid;
8863
8864   if(open("console", O_RDWR) < 0){
8865     mknod("console", 1, 1);
8866     open("console", O_RDWR);
8867   }
8868   dup(0);  // stdout
8869   dup(0);  // stderr
8870
8871   for(;;){
8872     printf(1, "init: starting sh\n");
8873     pid = fork();
8874     if(pid < 0){
8875       printf(1, "init: fork failed\n");
8876       exit();
8877     }
8878     if(pid == 0){
8879       exec("sh", argv);
8880       printf(1, "init: exec sh failed\n");
8881       exit();
8882     }
8883     while((wpid=wait()) >= 0 && wpid != pid)
8884       printf(1, "zombie!\n");
8885   }
8886 }
8887
8888
8889
8890
8891
8892
8893
8894
8895
8896
8897
8898
8899
```



```
8900 // Shell.
8901
8902 #include "types.h"
8903 #include "user.h"
8904 #include "fcntl.h"
8905
8906 // Parsed command representation
8907 #define EXEC  1
8908 #define REDIR 2
8909 #define PIPE  3
8910 #define LIST  4
8911 #define BACK  5
8912
8913 #define MAXARGS 10
8914
8915 struct cmd {
8916   int type;
8917 };
8918
8919 struct execcmd {
8920   int type;
8921   char *argv[MAXARGS];
8922   char *eargv[MAXARGS];
8923 };
8924
8925 struct redircmd {
8926   int type;
8927   struct cmd *cmd;
8928   char *file;
8929   char *efile;
8930   int mode;
8931   int fd;
8932 };
8933
8934 struct pipecmd {
8935   int type;
8936   struct cmd *left;
8937   struct cmd *right;
8938 };
8939
8940 struct listcmd {
8941   int type;
8942   struct cmd *left;
8943   struct cmd *right;
8944 };
8945
8946 struct backcmd {
8947   int type;
8948   struct cmd *cmd;
8949 };
```

```
8950 int fork1(void);  // Fork but panics on failure.
8951 void panic(char*);
8952 struct cmd *parsecmd(char*);
8953
8954 // Execute cmd.  Never returns.
8955 void
8956 runcmd(struct cmd *cmd)
8957 {
8958   int p[2];
8959   struct backcmd *bcmd;
8960   struct execcmd *ecmd;
8961   struct listcmd *lcmd;
8962   struct pipecmd *pcmd;
8963   struct redircmd *rcmd;
8964
8965   if(cmd == 0)
8966     exit();
8967
8968   switch(cmd->type){
8969   default:
8970     panic("runcmd");
8971
8972   case EXEC:
8973     ecmd = (struct execcmd*)cmd;
8974     if(ecmd->argv[0] == 0)
8975       exit();
8976     exec(ecmd->argv[0], ecmd->argv);
8977     printf(2, "exec %s failed\n", ecmd->argv[0]);
8978     break;
8979
8980   case REDIR:
8981     rcmd = (struct redircmd*)cmd;
8982     close(rcmd->fd);
8983     if(open(rcmd->file, rcmd->mode) < 0){
8984       printf(2, "open %s failed\n", rcmd->file);
8985       exit();
8986     }
8987     runcmd(rcmd->cmd);
8988     break;
8989
8990   case LIST:
8991     lcmd = (struct listcmd*)cmd;
8992     if(fork1() == 0)
8993       runcmd(lcmd->left);
8994     wait();
8995     runcmd(lcmd->right);
8996     break;
8997
8998
8999
```



```
9000   case PIPE:
9001     pcmd = (struct pipecmd*)cmd;
9002     if(pipe(p) < 0)
9003       panic("pipe");
9004     if(fork1() == 0){
9005       close(1);
9006       dup(p[1]);
9007       close(p[0]);
9008       close(p[1]);
9009       runcmd(pcmd->left);
9010     }
9011     if(fork1() == 0){
9012       close(0);
9013       dup(p[0]);
9014       close(p[0]);
9015       close(p[1]);
9016       runcmd(pcmd->right);
9017     }
9018     close(p[0]);
9019     close(p[1]);
9020     wait();
9021     wait();
9022     break;
9023
9024   case BACK:
9025     bcmd = (struct backcmd*)cmd;
9026     if(fork1() == 0)
9027       runcmd(bcmd->cmd);
9028     break;
9029   }
9030   exit();
9031 }
9032
9033 int
9034 getcmd(char *buf, int nbuf)
9035 {
9036   printf(2, "$ ");
9037   memset(buf, 0, nbuf);
9038   gets(buf, nbuf);
9039   if(buf[0] == 0) // EOF
9040     return -1;
9041   return 0;
9042 }
9043
9044
9045
9046
9047
9048
9049
```

```
9050 int
9051 main(void)
9052 {
9053   static char buf[100];
9054   int fd;
9055
9056   // Ensure that three file descriptors are open.
9057   while((fd = open("console", O_RDWR)) >= 0){
9058     if(fd >= 3){
9059       close(fd);
9060       break;
9061     }
9062   }
9063
9064   // Read and run input commands.
9065   while(getcmd(buf, sizeof(buf)) >= 0){
9066     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
9067       // Chdir must be called by the parent, not the child.
9068       buf[strlen(buf)-1] = 0;  // chop \n
9069       if(chdir(buf+3) < 0)
9070         printf(2, "cannot cd %s\n", buf+3);
9071       continue;
9072     }
9073     if(fork1() == 0)
9074       runcmd(parsecmd(buf));
9075     wait();
9076   }
9077   exit();
9078 }
9079
9080 void
9081 panic(char *s)
9082 {
9083   printf(2, "%s\n", s);
9084   exit();
9085 }
9086
9087 int
9088 fork1(void)
9089 {
9090   int pid;
9091
9092   pid = fork();
9093   if(pid == -1)
9094     panic("fork");
9095   return pid;
9096 }
9097
9098
9099
```



```
9100 // Constructors
9101
9102 struct cmd*
9103 execcmd(void)
9104 {
9105   struct execcmd *cmd;
9106
9107   cmd = malloc(sizeof(*cmd));
9108   memset(cmd, 0, sizeof(*cmd));
9109   cmd->type = EXEC;
9110   return (struct cmd*)cmd;
9111 }
9112
9113 struct cmd*
9114 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
9115 {
9116   struct redircmd *cmd;
9117
9118   cmd = malloc(sizeof(*cmd));
9119   memset(cmd, 0, sizeof(*cmd));
9120   cmd->type = REDIR;
9121   cmd->cmd = subcmd;
9122   cmd->file = file;
9123   cmd->efile = efile;
9124   cmd->mode = mode;
9125   cmd->fd = fd;
9126   return (struct cmd*)cmd;
9127 }
9128
9129 struct cmd*
9130 pipecmd(struct cmd *left, struct cmd *right)
9131 {
9132   struct pipecmd *cmd;
9133
9134   cmd = malloc(sizeof(*cmd));
9135   memset(cmd, 0, sizeof(*cmd));
9136   cmd->type = PIPE;
9137   cmd->left = left;
9138   cmd->right = right;
9139   return (struct cmd*)cmd;
9140 }
9141
9142
9143
9144
9145
9146
9147
9148
9149
```

```
9150 struct cmd*
9151 listcmd(struct cmd *left, struct cmd *right)
9152 {
9153   struct listcmd *cmd;
9154
9155   cmd = malloc(sizeof(*cmd));
9156   memset(cmd, 0, sizeof(*cmd));
9157   cmd->type = LIST;
9158   cmd->left = left;
9159   cmd->right = right;
9160   return (struct cmd*)cmd;
9161 }
9162
9163 struct cmd*
9164 backcmd(struct cmd *subcmd)
9165 {
9166   struct backcmd *cmd;
9167
9168   cmd = malloc(sizeof(*cmd));
9169   memset(cmd, 0, sizeof(*cmd));
9170   cmd->type = BACK;
9171   cmd->cmd = subcmd;
9172   return (struct cmd*)cmd;
9173 }
9174
9175
9176
9177
9178
9179
9180
9181
9182
9183
9184
9185
9186
9187
9188
9189
9190
9191
9192
9193
9194
9195
9196
9197
9198
9199
```



```
9200 // Parsing
9201
9202 char whitespace[] = " \t\r\n\v";
9203 char symbols[] = "<|>&;()";
9204
9205 int
9206 gettoken(char **ps, char *es, char **q, char **eq)
9207 {
9208   char *s;
9209   int ret;
9210
9211   s = *ps;
9212   while(s < es && strchr(whitespace, *s))
9213     s++;
9214   if(q)
9215     *q = s;
9216   ret = *s;
9217   switch(*s){
9218   case 0:
9219     break;
9220   case '|':
9221   case '(':
9222   case ')':
9223   case ';':
9224   case '&':
9225   case '<':
9226     s++;
9227     break;
9228   case '>':
9229     s++;
9230     if(*s == '>'){
9231       ret = '+';
9232       s++;
9233     }
9234     break;
9235   default:
9236     ret = 'a';
9237     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
9238       s++;
9239     break;
9240   }
9241   if(eq)
9242     *eq = s;
9243
9244   while(s < es && strchr(whitespace, *s))
9245     s++;
9246   *ps = s;
9247   return ret;
9248 }
9249
```

```
9250 int
9251 peek(char **ps, char *es, char *toks)
9252 {
9253   char *s;
9254
9255   s = *ps;
9256   while(s < es && strchr(whitespace, *s))
9257     s++;
9258   *ps = s;
9259   return *s && strchr(toks, *s);
9260 }
9261
9262 struct cmd *parseline(char**, char*);
9263 struct cmd *parsepipe(char**, char*);
9264 struct cmd *parseexec(char**, char*);
9265 struct cmd *nulterminate(struct cmd*);
9266
9267 struct cmd*
9268 parsecmd(char *s)
9269 {
9270   char *es;
9271   struct cmd *cmd;
9272
9273   es = s + strlen(s);
9274   cmd = parseline(&s, es);
9275   peek(&s, es, "");
9276   if(s != es){
9277     printf(2, "leftovers: %s\n", s);
9278     panic("syntax");
9279   }
9280   nulterminate(cmd);
9281   return cmd;
9282 }
9283
9284 struct cmd*
9285 parseline(char **ps, char *es)
9286 {
9287   struct cmd *cmd;
9288
9289   cmd = parsepipe(ps, es);
9290   while(peek(ps, es, "&")){
9291     gettoken(ps, es, 0, 0);
9292     cmd = backcmd(cmd);
9293   }
9294   if(peek(ps, es, ";")){
9295     gettoken(ps, es, 0, 0);
9296     cmd = listcmd(cmd, parseline(ps, es));
9297   }
9298   return cmd;
9299 }
```



```
9300 struct cmd*
9301 parsepipe(char **ps, char *es)
9302 {
9303   struct cmd *cmd;
9304
9305   cmd = parseexec(ps, es);
9306   if(peek(ps, es, "|")){
9307     gettoken(ps, es, 0, 0);
9308     cmd = pipecmd(cmd, parsepipe(ps, es));
9309   }
9310   return cmd;
9311 }
9312
9313 struct cmd*
9314 parseredirs(struct cmd *cmd, char **ps, char *es)
9315 {
9316   int tok;
9317   char *q, *eq;
9318
9319   while(peek(ps, es, "<>")){
9320     tok = gettoken(ps, es, 0, 0);
9321     if(gettoken(ps, es, &q, &eq) != 'a')
9322       panic("missing file for redirection");
9323     switch(tok){
9324     case '<':
9325       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
9326       break;
9327     case '>':
9328       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9329       break;
9330     case '+':  // >>
9331       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9332       break;
9333     }
9334   }
9335   return cmd;
9336 }
9337
9338
9339
9340
9341
9342
9343
9344
9345
9346
9347
9348
9349
```

```
9350 struct cmd*
9351 parseblock(char **ps, char *es)
9352 {
9353   struct cmd *cmd;
9354
9355   if(!peek(ps, es, "("))
9356     panic("parseblock");
9357   gettoken(ps, es, 0, 0);
9358   cmd = parseline(ps, es);
9359   if(!peek(ps, es, ")"))
9360     panic("syntax - missing )");
9361   gettoken(ps, es, 0, 0);
9362   cmd = parseredirs(cmd, ps, es);
9363   return cmd;
9364 }
9365
9366 struct cmd*
9367 parseexec(char **ps, char *es)
9368 {
9369   char *q, *eq;
9370   int tok, argc;
9371   struct execcmd *cmd;
9372   struct cmd *ret;
9373
9374   if(peek(ps, es, "("))
9375     return parseblock(ps, es);
9376
9377   ret = execcmd();
9378   cmd = (struct execcmd*)ret;
9379
9380   argc = 0;
9381   ret = parseredirs(ret, ps, es);
9382   while(!peek(ps, es, "|)&;")){
9383     if((tok=gettoken(ps, es, &q, &eq)) == 0)
9384       break;
9385     if(tok != 'a')
9386       panic("syntax");
9387     cmd->argv[argc] = q;
9388     cmd->eargv[argc] = eq;
9389     argc++;
9390     if(argc >= MAXARGS)
9391       panic("too many args");
9392     ret = parseredirs(ret, ps, es);
9393   }
9394   cmd->argv[argc] = 0;
9395   cmd->eargv[argc] = 0;
9396   return ret;
9397 }
9398
9399
```



```
9400 // NUL-terminate all the counted strings.
9401 struct cmd*
9402 nulterminate(struct cmd *cmd)
9403 {
9404   int i;
9405   struct backcmd *bcmd;
9406   struct execcmd *ecmd;
9407   struct listcmd *lcmd;
9408   struct pipecmd *pcmd;
9409   struct redircmd *rcmd;
9410
9411   if(cmd == 0)
9412     return 0;
9413
9414   switch(cmd->type){
9415   case EXEC:
9416     ecmd = (struct execcmd*)cmd;
9417     for(i=0; ecmd->argv[i]; i++)
9418       *ecmd->eargv[i] = 0;
9419     break;
9420
9421   case REDIR:
9422     rcmd = (struct redircmd*)cmd;
9423     nulterminate(rcmd->cmd);
9424     *rcmd->efile = 0;
9425     break;
9426
9427   case PIPE:
9428     pcmd = (struct pipecmd*)cmd;
9429     nulterminate(pcmd->left);
9430     nulterminate(pcmd->right);
9431     break;
9432
9433   case LIST:
9434     lcmd = (struct listcmd*)cmd;
9435     nulterminate(lcmd->left);
9436     nulterminate(lcmd->right);
9437     break;
9438
9439   case BACK:
9440     bcmd = (struct backcmd*)cmd;
9441     nulterminate(bcmd->cmd);
9442     break;
9443   }
9444   return cmd;
9445 }
9446
9447
9448
9449
```

```
#include "asm.h"
#include "memlayout.h"
#include "mmu.h"

# Start the first CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.code16                       # Assemble for 16-bit mode
.globl start
start:
  cli                         # BIOS enabled interrupts; disable

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax             # Set %ax to zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment

  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60

  # Switch from real to protected mode.  Use a bootstrap GDT that makes
  # virtual addresses map directly to physical addresses so that the
  # effective memory map doesn't change during the transition.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
  movl    %eax, %cr0
```



```
  # Complete the transition to 32-bit protected mode by using a long jmp
  # to reload %cs and %eip.  The segment descriptors are set up with no
  # translation, so that the mapping is still the identity mapping.
  ljmp    $(SEG_KCODE<<3), $start32

.code32  # Tell assembler to generate 32-bit code now.
start32:
  # Set up the protected-mode data segment registers
  movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %ss                # -> SS: Stack Segment
  movw    $0, %ax                 # Zero segments not ready for use
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS

  # Set up the stack pointer and call into C.
  movl    $start, %esp
  call    bootmain

  # If bootmain returns (it shouldn't), trigger a Bochs
  # breakpoint if running under Bochs, then loop.
  movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
  movw    %ax, %dx
  outw    %ax, %dx
  movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
  outw    %ax, %dx
spin:
  jmp     spin

# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULLASM                             # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg

gdtdesc:
  .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
  .long   gdt                             # address gdt
```

```
9550 // Boot loader.
9551 //
9552 // Part of the boot block, along with bootasm.S, which calls bootmain().
9553 // bootasm.S has put the processor into protected 32-bit mode.
9554 // bootmain() loads an ELF kernel image from the disk starting at
9555 // sector 1 and then jumps to the kernel entry routine.
9556
9557 #include "types.h"
9558 #include "elf.h"
9559 #include "x86.h"
9560 #include "memlayout.h"
9561
9562 #define SECTSIZE  512
9563
9564 void readseg(uchar*, uint, uint);
9565
9566 void
9567 bootmain(void)
9568 {
9569   struct elfhdr *elf;
9570   struct proghdr *ph, *eph;
9571   void (*entry)(void);
9572   uchar* pa;
9573
9574   elf = (struct elfhdr*)0x10000;  // scratch space
9575
9576   // Read 1st page off disk
9577   readseg((uchar*)elf, 4096, 0);
9578
9579   // Is this an ELF executable?
9580   if(elf->magic != ELF_MAGIC)
9581     return;  // let bootasm.S handle error
9582
9583   // Load each program segment (ignores ph flags).
9584   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
9585   eph = ph + elf->phnum;
9586   for(; ph < eph; ph++){
9587     pa = (uchar*)ph->paddr;
9588     readseg(pa, ph->filesz, ph->off);
9589     if(ph->memsz > ph->filesz)
9590       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
9591   }
9592
9593   // Call the entry point from the ELF header.
9594   // Does not return!
9595   entry = (void(*)(void))(elf->entry);
9596   entry();
9597 }
9598
9599
```



```
9600 void
9601 waitdisk(void)
9602 {
9603   // Wait for disk ready.
9604   while((inb(0x1F7) & 0xC0) != 0x40)
9605     ;
9606 }
9607
9608 // Read a single sector at offset into dst.
9609 void
9610 readsect(void *dst, uint offset)
9611 {
9612   // Issue command.
9613   waitdisk();
9614   outb(0x1F2, 1);   // count = 1
9615   outb(0x1F3, offset);
9616   outb(0x1F4, offset >> 8);
9617   outb(0x1F5, offset >> 16);
9618   outb(0x1F6, (offset >> 24) | 0xE0);
9619   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
9620
9621   // Read data.
9622   waitdisk();
9623   insl(0x1F0, dst, SECTSIZE/4);
9624 }
9625
9626 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
9627 // Might copy more than asked.
9628 void
9629 readseg(uchar* pa, uint count, uint offset)
9630 {
9631   uchar* epa;
9632
9633   epa = pa + count;
9634
9635   // Round down to sector boundary.
9636   pa -= offset % SECTSIZE;
9637
9638   // Translate from bytes to sectors; kernel starts at sector 1.
9639   offset = (offset / SECTSIZE) + 1;
9640
9641   // If this is too slow, we could read lots of sectors at a time.
9642   // We'd write more to memory than asked, but it doesn't matter --
9643   // we load in increasing order.
9644   for(; pa < epa; pa += SECTSIZE, offset++)
9645     readsect(pa, offset);
9646 }
9647
9648
9649
```