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, Anton Burtsev, Cody Cutler, Mike CAT, Tej Chajed, Nelson Elhage,
Saar Ettinger, Alice Ferrazzi, Nathaniel Filardo, Peter Froehlich, Yakir Goaron,
Shivam Handa, Bryan Henry, Jim Huang, Alexander Kapshuk, Anders Kaseorg,
kehao95, Wolfgang Keller, Eddie Kohler, Austin Liew, 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, Grant Wu, Jindong
Zhang, Icenowy Zheng, 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             lapicid(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 int             cpuid(void);
0359 void            exit(void);
0360 int             fork(void);
0361 int             growproc(int);
0362 int             kill(int);
0363 struct cpu*     mycpu(void);
0364 struct proc*    myproc();
0365 void            pinit(void);
0366 void            procdump(void);
0367 void            scheduler(void) __attribute__((noreturn));
0368 void            sched(void);
0369 void            setproc(struct proc*);
0370 void            sleep(void*, struct spinlock*);
0371 void            userinit(void);
0372 int             wait(void);
0373 void            wakeup(void*);
0374 void            yield(void);
0375
0376 // swtch.S
0377 void            swtch(struct context**, struct context*);
0378
0379 // spinlock.c
0380 void            acquire(struct spinlock*);
0381 void            getcallerpcs(void*, uint*);
0382 int             holding(struct spinlock*);
0383 void            initlock(struct spinlock*, char*);
0384 void            release(struct spinlock*);
0385 void            pushcli(void);
0386 void            popcli(void);
0387
0388 // sleeplock.c
0389 void            acquiresleep(struct sleeplock*);
0390 void            releasesleep(struct sleeplock*);
0391 int             holdingsleep(struct sleeplock*);
0392 void            initsleeplock(struct sleeplock*, char*);
0393
0394 // string.c
0395 int             memcmp(const void*, const void*, uint);
0396 void*           memmove(void*, const void*, uint);
0397 void*           memset(void*, int, uint);
0398 char*           safestrcpy(char*, const char*, int);
0399 int             strlen(const char*);
```



```
0400 int             strncmp(const char*, const char*, uint);
0401 char*           strncpy(char*, const char*, int);
0402
0403 // syscall.c
0404 int             argint(int, int*);
0405 int             argptr(int, char**, int);
0406 int             argstr(int, char**);
0407 int             fetchint(uint, int*);
0408 int             fetchstr(uint, char**);
0409 void            syscall(void);
0410
0411 // timer.c
0412 void            timerinit(void);
0413
0414 // trap.c
0415 void            idtinit(void);
0416 extern uint     ticks;
0417 void            tvinit(void);
0418 extern struct spinlock tickslock;
0419
0420 // uart.c
0421 void            uartinit(void);
0422 void            uartintr(void);
0423 void            uartputc(int);
0424
0425 // vm.c
0426 void            seginit(void);
0427 void            kvmalloc(void);
0428 pde_t*          setupkvm(void);
0429 char*           uva2ka(pde_t*, char*);
0430 int             allocuvm(pde_t*, uint, uint);
0431 int             deallocuvm(pde_t*, uint, uint);
0432 void            freevm(pde_t*);
0433 void            inituvm(pde_t*, char*, uint);
0434 int             loaduvm(pde_t*, char*, struct inode*, uint, uint);
0435 pde_t*          copyuvm(pde_t*, uint);
0436 void            switchuvm(struct proc*);
0437 void            switchkvm(void);
0438 int             copyout(pde_t*, uint, void*, uint);
0439 void            clearpteu(pde_t *pgdir, char *uva);
0440
0441 // number of elements in fixed-size array
0442 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
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_UCODE 3  // user code
0745 #define SEG_UDATA 4  // user data+stack
0746 #define SEG_TSS   5  // this process's task state
0747
0748 // cpu->gdt[NSEGS] holds the above segments.
0749 #define NSEGS     6
```

```
0750
0751 #ifndef __ASSEMBLER__
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 // Segment Descriptor
0801 struct segdesc {
0802   uint lim_15_0 : 16;  // Low bits of segment limit
0803   uint base_15_0 : 16; // Low bits of segment base address
0804   uint base_23_16 : 8; // Middle bits of segment base address
0805   uint type : 4;       // Segment type (see STS_ constants)
0806   uint s : 1;          // 0 = system, 1 = application
0807   uint dpl : 2;        // Descriptor Privilege Level
0808   uint p : 1;          // Present
0809   uint lim_19_16 : 4;  // High bits of segment limit
0810   uint avl : 1;        // Unused (available for software use)
0811   uint rsv1 : 1;       // Reserved
0812   uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
0813   uint g : 1;          // Granularity: limit scaled by 4K when set
0814   uint base_31_24 : 8; // High bits of segment base address
0815 };
0816
0817 // Normal segment
0818 #define SEG(type, base, lim, dpl) (struct segdesc)    \
0819 { ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff,      \
0820   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
0821   (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
0822 #define SEG16(type, base, lim, dpl) (struct segdesc)  \
0823 { (lim) & 0xffff, (uint)(base) & 0xffff,              \
0824   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
0825   (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
0826 #endif
0827
0828 #define DPL_USER    0x3     // User DPL
0829
0830 // Application segment type bits
0831 #define STA_X       0x8     // Executable segment
0832 #define STA_E       0x4     // Expand down (non-executable segments)
0833 #define STA_C       0x4     // Conforming code segment (executable only)
0834 #define STA_W       0x2     // Writeable (non-executable segments)
0835 #define STA_R       0x2     // Readable (executable segments)
0836 #define STA_A       0x1     // Accessed
0837
0838 // System segment type bits
0839 #define STS_T16A    0x1     // Available 16-bit TSS
0840 #define STS_LDT     0x2     // Local Descriptor Table
0841 #define STS_T16B    0x3     // Busy 16-bit TSS
0842 #define STS_CG16    0x4     // 16-bit Call Gate
0843 #define STS_TG      0x5     // Task Gate / Coum Transmitions
0844 #define STS_IG16    0x6     // 16-bit Interrupt Gate
0845 #define STS_TG16    0x7     // 16-bit Trap Gate
0846 #define STS_T32A    0x9     // Available 32-bit TSS
0847 #define STS_T32B    0xB     // Busy 32-bit TSS
0848 #define STS_CG32    0xC     // 32-bit Call Gate
0849 #define STS_IG32    0xE     // 32-bit Interrupt Gate
```

```
0850 #define STS_TG32    0xF     // 32-bit Trap Gate
0851
0852 // A virtual address 'la' has a three-part structure as follows:
0853 //
0854 // +--------10------+-------10-------+---------12----------+
0855 // | Page Directory |   Page Table   | Offset within Page  |
0856 // |      Index     |      Index     |                     |
0857 // +----------------+----------------+---------------------+
0858 //  \--- PDX(va) --/ \--- PTX(va) --/
0859
0860 // page directory index
0861 #define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)
0862
0863 // page table index
0864 #define PTX(va)         (((uint)(va) >> PTXSHIFT) & 0x3FF)
0865
0866 // construct virtual address from indexes and offset
0867 #define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
0868
0869 // Page directory and page table constants.
0870 #define NPDENTRIES      1024    // # directory entries per page directory
0871 #define NPTENTRIES      1024    // # PTEs per page table
0872 #define PGSIZE          4096    // bytes mapped by a page
0873
0874 #define PGSHIFT         12      // log2(PGSIZE)
0875 #define PTXSHIFT        12      // offset of PTX in a linear address
0876 #define PDXSHIFT        22      // offset of PDX in a linear address
0877
0878 #define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
0879 #define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
0880
0881 // Page table/directory entry flags.
0882 #define PTE_P           0x001   // Present
0883 #define PTE_W           0x002   // Writeable
0884 #define PTE_U           0x004   // User
0885 #define PTE_PWT         0x008   // Write-Through
0886 #define PTE_PCD         0x010   // Cache-Disable
0887 #define PTE_A           0x020   // Accessed
0888 #define PTE_D           0x040   // Dirty
0889 #define PTE_PS          0x080   // Page Size
0890 #define PTE_MBZ         0x180   // Bits must be zero
0891
0892 // Address in page table or page directory entry
0893 #define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
0894 #define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF)
0895
0896 #ifndef __ASSEMBLER__
0897 typedef uint pte_t;
0898
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   picinit();       // disable pic
1325   ioapicinit();    // another interrupt controller
1326   consoleinit();   // console hardware
1327   uartinit();      // serial port
1328   pinit();         // process table
1329   tvinit();        // trap vectors
1330   binit();         // buffer cache
1331   fileinit();      // file table
1332   ideinit();       // disk
1333   startothers();   // start other processors
1334   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1335   userinit();      // first user process
1336   mpmain();        // finish this processor's setup
1337 }
1338
1339 // Other CPUs jump here from entryother.S.
1340 static void
1341 mpenter(void)
1342 {
1343   switchkvm();
1344   seginit();
1345   lapicinit();
1346   mpmain();
1347 }
1348
1349
```

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



```
1400 // The boot page table used in entry.S and entryother.S.
1401 // Page directories (and page tables) must start on page boundaries,
1402 // hence the __aligned__ attribute.
1403 // PTE_PS in a page directory entry enables 4Mbyte pages.
1404
1405 __attribute__((__aligned__(PGSIZE)))
1406 pde_t entrypgdir[NPDENTRIES] = {
1407   // Map VA's [0, 4MB) to PA's [0, 4MB)
1408   [0] = (0) | PTE_P | PTE_W | PTE_PS,
1409   // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1410   [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1411 };
1412
1413
1414
1415
1416
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 = mycpu();
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 == mycpu();
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(mycpu()->ncli == 0)
1662     mycpu()->intena = eflags & FL_IF;
1663   mycpu()->ncli += 1;
1664 }
1665
1666 void
1667 popcli(void)
1668 {
1669   if(readeflags()&FL_IF)
1670     panic("popcli - interruptible");
1671   if(--mycpu()->ncli < 0)
1672     panic("popcli");
1673   if(mycpu()->ncli == 0 && mycpu()->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[cpuid()];
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   lgdt(c->gdt, sizeof(c->gdt));
1729 }
1730
1731 // Return the address of the PTE in page table pgdir
1732 // that corresponds to virtual address va.  If alloc!=0,
1733 // create any required page table pages.
1734 static pte_t *
1735 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1736 {
1737   pde_t *pde;
1738   pte_t *pgtab;
1739
1740   pde = &pgdir[PDX(va)];
1741   if(*pde & PTE_P){
1742     pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1743   } else {
1744     if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1745       return 0;
1746     // Make sure all those PTE_P bits are zero.
1747     memset(pgtab, 0, PGSIZE);
1748     // The permissions here are overly generous, but they can
1749     // be further restricted by the permissions in the page table
```

```
1750     // entries, if necessary.
1751     *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1752   }
1753   return &pgtab[PTX(va)];
1754 }
1755
1756 // Create PTEs for virtual addresses starting at va that refer to
1757 // physical addresses starting at pa. va and size might not
1758 // be page-aligned.
1759 static int
1760 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1761 {
1762   char *a, *last;
1763   pte_t *pte;
1764
1765   a = (char*)PGROUNDDOWN((uint)va);
1766   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1767   for(;;){
1768     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1769       return -1;
1770     if(*pte & PTE_P)
1771       panic("remap");
1772     *pte = pa | perm | PTE_P;
1773     if(a == last)
1774       break;
1775     a += PGSIZE;
1776     pa += PGSIZE;
1777   }
1778   return 0;
1779 }
1780
1781 // There is one page table per process, plus one that's used when
1782 // a CPU is not running any process (kpgdir). The kernel uses the
1783 // current process's page table during system calls and interrupts;
1784 // page protection bits prevent user code from using the kernel's
1785 // mappings.
1786 //
1787 // setupkvm() and exec() set up every page table like this:
1788 //
1789 //   0..KERNBASE: user memory (text+data+stack+heap), mapped to
1790 //                phys memory allocated by the kernel
1791 //   KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
1792 //   KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
1793 //                for the kernel's instructions and r/o data
1794 //   data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
1795 //                                     rw data + free physical memory
1796 //   0xfe000000..0: mapped direct (devices such as ioapic)
1797 //
1798 // The kernel allocates physical memory for its heap and for user memory
1799 // between V2P(end) and the end of physical memory (PHYSTOP)
```



```
1800 // (directly addressable from end..P2V(PHYSTOP)).
1801
1802 // This table defines the kernel's mappings, which are present in
1803 // every process's page table.
1804 static struct kmap {
1805   void *virt;
1806   uint phys_start;
1807   uint phys_end;
1808   int perm;
1809 } kmap[] = {
1810  { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
1811  { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
1812  { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
1813  { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
1814 };
1815
1816 // Set up kernel part of a page table.
1817 pde_t*
1818 setupkvm(void)
1819 {
1820   pde_t *pgdir;
1821   struct kmap *k;
1822
1823   if((pgdir = (pde_t*)kalloc()) == 0)
1824     return 0;
1825   memset(pgdir, 0, PGSIZE);
1826   if (P2V(PHYSTOP) > (void*)DEVSPACE)
1827     panic("PHYSTOP too high");
1828   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1829     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1830                 (uint)k->phys_start, k->perm) < 0) {
1831       freevm(pgdir);
1832       return 0;
1833     }
1834   return pgdir;
1835 }
1836
1837 // Allocate one page table for the machine for the kernel address
1838 // space for scheduler processes.
1839 void
1840 kvmalloc(void)
1841 {
1842   kpgdir = setupkvm();
1843   switchkvm();
1844 }
1845
1846
1847
1848
1849
```

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



```
1900 // Load a program segment into pgdir.  addr must be page-aligned
1901 // and the pages from addr to addr+sz must already be mapped.
1902 int
1903 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
1904 {
1905   uint i, pa, n;
1906   pte_t *pte;
1907
1908   if((uint) addr % PGSIZE != 0)
1909     panic("loaduvm: addr must be page aligned");
1910   for(i = 0; i < sz; i += PGSIZE){
1911     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1912       panic("loaduvm: address should exist");
1913     pa = PTE_ADDR(*pte);
1914     if(sz - i < PGSIZE)
1915       n = sz - i;
1916     else
1917       n = PGSIZE;
1918     if(readi(ip, P2V(pa), offset+i, n) != n)
1919       return -1;
1920   }
1921   return 0;
1922 }
1923
1924 // Allocate page tables and physical memory to grow process from oldsz to
1925 // newsz, which need not be page aligned.  Returns new size or 0 on error.
1926 int
1927 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1928 {
1929   char *mem;
1930   uint a;
1931
1932   if(newsz >= KERNBASE)
1933     return 0;
1934   if(newsz < oldsz)
1935     return oldsz;
1936
1937   a = PGROUNDUP(oldsz);
1938   for(; a < newsz; a += PGSIZE){
1939     mem = kalloc();
1940     if(mem == 0){
1941       cprintf("allocuvm out of memory\n");
1942       deallocuvm(pgdir, newsz, oldsz);
1943       return 0;
1944     }
1945     memset(mem, 0, PGSIZE);
1946     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
1947       cprintf("allocuvm out of memory (2)\n");
1948       deallocuvm(pgdir, newsz, oldsz);
1949       kfree(mem);
```

```
1950       return 0;
1951     }
1952   }
1953   return newsz;
1954 }
1955
1956 // Deallocate user pages to bring the process size from oldsz to
1957 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
1958 // need to be less than oldsz.  oldsz can be larger than the actual
1959 // process size.  Returns the new process size.
1960 int
1961 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1962 {
1963   pte_t *pte;
1964   uint a, pa;
1965
1966   if(newsz >= oldsz)
1967     return oldsz;
1968
1969   a = PGROUNDUP(newsz);
1970   for(; a  < oldsz; a += PGSIZE){
1971     pte = walkpgdir(pgdir, (char*)a, 0);
1972     if(!pte)
1973       a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
1974     else if((*pte & PTE_P) != 0){
1975       pa = PTE_ADDR(*pte);
1976       if(pa == 0)
1977         panic("kfree");
1978       char *v = P2V(pa);
1979       kfree(v);
1980       *pte = 0;
1981     }
1982   }
1983   return newsz;
1984 }
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
```



```
2000 // Free a page table and all the physical memory pages
2001 // in the user part.
2002 void
2003 freevm(pde_t *pgdir)
2004 {
2005   uint i;
2006
2007   if(pgdir == 0)
2008     panic("freevm: no pgdir");
2009   deallocuvm(pgdir, KERNBASE, 0);
2010   for(i = 0; i < NPDENTRIES; i++){
2011     if(pgdir[i] & PTE_P){
2012       char * v = P2V(PTE_ADDR(pgdir[i]));
2013       kfree(v);
2014     }
2015   }
2016   kfree((char*)pgdir);
2017 }
2018
2019 // Clear PTE_U on a page. Used to create an inaccessible
2020 // page beneath the user stack.
2021 void
2022 clearpteu(pde_t *pgdir, char *uva)
2023 {
2024   pte_t *pte;
2025
2026   pte = walkpgdir(pgdir, uva, 0);
2027   if(pte == 0)
2028     panic("clearpteu");
2029   *pte &= ~PTE_U;
2030 }
2031
2032 // Given a parent process's page table, create a copy
2033 // of it for a child.
2034 pde_t*
2035 copyuvm(pde_t *pgdir, uint sz)
2036 {
2037   pde_t *d;
2038   pte_t *pte;
2039   uint pa, i, flags;
2040   char *mem;
2041
2042   if((d = setupkvm()) == 0)
2043     return 0;
2044   for(i = 0; i < sz; i += PGSIZE){
2045     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
2046       panic("copyuvm: pte should exist");
2047     if(!(*pte & PTE_P))
2048       panic("copyuvm: page not present");
2049     pa = PTE_ADDR(*pte);
```

```
2050     flags = PTE_FLAGS(*pte);
2051     if((mem = kalloc()) == 0)
2052       goto bad;
2053     memmove(mem, (char*)P2V(pa), PGSIZE);
2054     if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
2055       goto bad;
2056   }
2057   return d;
2058
2059 bad:
2060   freevm(d);
2061   return 0;
2062 }
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
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   struct proc *proc;           // The process running on this cpu or null
2310 };
2311
2312 extern struct cpu cpus[NCPU];
2313 extern int ncpu;
2314
2315
2316 // Saved registers for kernel context switches.
2317 // Don't need to save all the segment registers (%cs, etc),
2318 // because they are constant across kernel contexts.
2319 // Don't need to save %eax, %ecx, %edx, because the
2320 // x86 convention is that the caller has saved them.
2321 // Contexts are stored at the bottom of the stack they
2322 // describe; the stack pointer is the address of the context.
2323 // The layout of the context matches the layout of the stack in swtch.S
2324 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2325 // but it is on the stack and allocproc() manipulates it.
2326 struct context {
2327   uint edi;
2328   uint esi;
2329   uint ebx;
2330   uint ebp;
2331   uint eip;
2332 };
2333
2334 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2335
2336 // Per-process state
2337 struct proc {
2338   uint sz;                     // Size of process memory (bytes)
2339   pde_t* pgdir;                // Page table
2340   char *kstack;                // Bottom of kernel stack for this process
2341   enum procstate state;        // Process state
2342   int pid;                     // Process ID
2343   struct proc *parent;         // Parent process
2344   struct trapframe *tf;        // Trap frame for current syscall
2345   struct context *context;     // swtch() here to run process
2346   void *chan;                  // If non-zero, sleeping on chan
2347   int killed;                  // If non-zero, have been killed
2348   struct file *ofile[NOFILE];  // Open files
2349   struct inode *cwd;           // Current directory
```

```
2350   char name[16];               // Process name (debugging)
2351 };
2352
2353 // Process memory is laid out contiguously, low addresses first:
2354 //   text
2355 //   original data and bss
2356 //   fixed-size stack
2357 //   expandable heap
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
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 // Must be called with interrupts disabled
2429 int
2430 cpuid() {
2431   return mycpu()-cpus;
2432 }
2433
2434 // Must be called with interrupts disabled to avoid the caller being reschedu
2435 // between reading lapicid and running through the loop.
2436 struct cpu*
2437 mycpu(void)
2438 {
2439   int apicid, i;
2440
2441   if(readeflags()&FL_IF)
2442     panic("mycpu called with interrupts enabled\n");
2443
2444   apicid = lapicid();
2445   // APIC IDs are not guaranteed to be contiguous. Maybe we should have
2446   // a reverse map, or reserve a register to store &cpus[i].
2447   for (i = 0; i < ncpu; ++i) {
2448     if (cpus[i].apicid == apicid)
2449       return &cpus[i];
```

```
2450   }
2451   panic("unknown apicid\n");
2452 }
2453
2454 // Disable interrupts so that we are not rescheduled
2455 // while reading proc from the cpu structure
2456 struct proc*
2457 myproc(void) {
2458   struct cpu *c;
2459   struct proc *p;
2460   pushcli();
2461   c = mycpu();
2462   p = c->proc;
2463   popcli();
2464   return p;
2465 }
2466
2467
2468 // Look in the process table for an UNUSED proc.
2469 // If found, change state to EMBRYO and initialize
2470 // state required to run in the kernel.
2471 // Otherwise return 0.
2472 static struct proc*
2473 allocproc(void)
2474 {
2475   struct proc *p;
2476   char *sp;
2477
2478   acquire(&ptable.lock);
2479
2480   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2481     if(p->state == UNUSED)
2482       goto found;
2483
2484   release(&ptable.lock);
2485   return 0;
2486
2487 found:
2488   p->state = EMBRYO;
2489   p->pid = nextpid++;
2490
2491   release(&ptable.lock);
2492
2493   // Allocate kernel stack.
2494   if((p->kstack = kalloc()) == 0){
2495     p->state = UNUSED;
2496     return 0;
2497   }
2498   sp = p->kstack + KSTACKSIZE;
2499
```



```
2500   // Leave room for trap frame.
2501   sp -= sizeof *p->tf;
2502   p->tf = (struct trapframe*)sp;
2503
2504   // Set up new context to start executing at forkret,
2505   // which returns to trapret.
2506   sp -= 4;
2507   *(uint*)sp = (uint)trapret;
2508
2509   sp -= sizeof *p->context;
2510   p->context = (struct context*)sp;
2511   memset(p->context, 0, sizeof *p->context);
2512   p->context->eip = (uint)forkret;
2513
2514   return p;
2515 }
2516
2517
2518 // Set up first user process.
2519 void
2520 userinit(void)
2521 {
2522   struct proc *p;
2523   extern char _binary_initcode_start[], _binary_initcode_size[];
2524
2525   p = allocproc();
2526
2527   initproc = p;
2528   if((p->pgdir = setupkvm()) == 0)
2529     panic("userinit: out of memory?");
2530   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2531   p->sz = PGSIZE;
2532   memset(p->tf, 0, sizeof(*p->tf));
2533   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2534   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2535   p->tf->es = p->tf->ds;
2536   p->tf->ss = p->tf->ds;
2537   p->tf->eflags = FL_IF;
2538   p->tf->esp = PGSIZE;
2539   p->tf->eip = 0;  // beginning of initcode.S
2540
2541   safestrcpy(p->name, "initcode", sizeof(p->name));
2542   p->cwd = namei("/");
2543
2544   // this assignment to p->state lets other cores
2545   // run this process. the acquire forces the above
2546   // writes to be visible, and the lock is also needed
2547   // because the assignment might not be atomic.
2548   acquire(&ptable.lock);
2549
```

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



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

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



```
2700     // No point waiting if we don't have any children.
2701     if(!havekids || curproc->killed){
2702       release(&ptable.lock);
2703       return -1;
2704     }
2705
2706     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2707     sleep(curproc, &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   struct cpu *c = mycpu();
2762   c->proc = 0;
2763
2764   for(;;){
2765     // Enable interrupts on this processor.
2766     sti();
2767
2768     // Loop over process table looking for process to run.
2769     acquire(&ptable.lock);
2770     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2771       if(p->state != RUNNABLE)
2772         continue;
2773
2774       // Switch to chosen process.  It is the process's job
2775       // to release ptable.lock and then reacquire it
2776       // before jumping back to us.
2777       c->proc = p;
2778       switchuvm(p);
2779       p->state = RUNNING;
2780
2781       swtch(&(c->scheduler), p->context);
2782       switchkvm();
2783
2784       // Process is done running for now.
2785       // It should have changed its p->state before coming back.
2786       c->proc = 0;
2787     }
2788     release(&ptable.lock);
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   struct proc *p = myproc();
2812
2813   if(!holding(&ptable.lock))
2814     panic("sched ptable.lock");
2815   if(mycpu()->ncli != 1)
2816     panic("sched locks");
2817   if(p->state == RUNNING)
2818     panic("sched running");
2819   if(readeflags()&FL_IF)
2820     panic("sched interruptible");
2821   intena = mycpu()->intena;
2822   swtch(&p->context, mycpu()->scheduler);
2823   mycpu()->intena = intena;
2824 }
2825
2826 // Give up the CPU for one scheduling round.
2827 void
2828 yield(void)
2829 {
2830   acquire(&ptable.lock);
2831   myproc()->state = RUNNABLE;
2832   sched();
2833   release(&ptable.lock);
2834 }
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
```

```
2850 // A fork child's very first scheduling by scheduler()
2851 // will swtch here.  "Return" to user space.
2852 void
2853 forkret(void)
2854 {
2855   static int first = 1;
2856   // Still holding ptable.lock from scheduler.
2857   release(&ptable.lock);
2858
2859   if (first) {
2860     // Some initialization functions must be run in the context
2861     // of a regular process (e.g., they call sleep), and thus cannot
2862     // be run from main().
2863     first = 0;
2864     iinit(ROOTDEV);
2865     initlog(ROOTDEV);
2866   }
2867
2868   // Return to "caller", actually trapret (see allocproc).
2869 }
2870
2871 // Atomically release lock and sleep on chan.
2872 // Reacquires lock when awakened.
2873 void
2874 sleep(void *chan, struct spinlock *lk)
2875 {
2876   struct proc *p = myproc();
2877
2878   if(p == 0)
2879     panic("sleep");
2880
2881   if(lk == 0)
2882     panic("sleep without lk");
2883
2884   // Must acquire ptable.lock in order to
2885   // change p->state and then call sched.
2886   // Once we hold ptable.lock, we can be
2887   // guaranteed that we won't miss any wakeup
2888   // (wakeup runs with ptable.lock locked),
2889   // so it's okay to release lk.
2890   if(lk != &ptable.lock){
2891     acquire(&ptable.lock);
2892     release(lk);
2893   }
2894   // Go to sleep.
2895   p->chan = chan;
2896   p->state = SLEEPING;
2897
2898   sched();
2899
```



```
2900   // Tidy up.
2901   p->chan = 0;
2902
2903   // Reacquire original lock.
2904   if(lk != &ptable.lock){
2905     release(&ptable.lock);
2906     acquire(lk);
2907   }
2908 }
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
```

```
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}

// Wake up all processes sleeping on chan.
void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}

// Kill the process with the given pid.
// Process won't exit until it returns
// to user space (see trap in trap.c).
int
kill(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING)
        p->state = RUNNABLE;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```



```
// Print a process listing to console.  For debugging.
// Runs when user types ^P on console.
// No lock to avoid wedging a stuck machine further.
void
procdump(void)
{
  static char *states[] = {
  [UNUSED]    "unused",
  [EMBRYO]    "embryo",
  [SLEEPING]  "sleep ",
  [RUNNABLE]  "runble",
  [RUNNING]   "run   ",
  [ZOMBIE]    "zombie"
  };
  int i;
  struct proc *p;
  char *state;
  uint pc[10];

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;
    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
      state = states[p->state];
    else
      state = "???";
    cprintf("%d %s %s", p->pid, state, p->name);
    if(p->state == SLEEPING){
      getcallerpcs((uint*)p->context->ebp+2, pc);
      for(i=0; i<10 && pc[i] != 0; i++)
        cprintf(" %p", pc[i]);
    }
    cprintf("\n");
  }
}
```

```
3050 # Context switch
3051 #
3052 #   void swtch(struct context **old, struct context *new);
3053 #
3054 # Save current register context in old
3055 # and then load register context from new.
3056
3057 .globl swtch
3058 swtch:
3059   movl 4(%esp), %eax
3060   movl 8(%esp), %edx
3061
3062   # Save old callee-save registers
3063   pushl %ebp
3064   pushl %ebx
3065   pushl %esi
3066   pushl %edi
3067
3068   # Switch stacks
3069   movl %esp, (%eax)
3070   movl %edx, %esp
3071
3072   # Load new callee-save registers
3073   popl %edi
3074   popl %esi
3075   popl %ebx
3076   popl %ebp
3077   ret
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
```



```
3100 // Physical memory allocator, intended to allocate
3101 // memory for user processes, kernel stacks, page table pages,
3102 // and pipe buffers. Allocates 4096-byte pages.
3103
3104 #include "types.h"
3105 #include "defs.h"
3106 #include "param.h"
3107 #include "memlayout.h"
3108 #include "mmu.h"
3109 #include "spinlock.h"
3110
3111 void freerange(void *vstart, void *vend);
3112 extern char end[]; // first address after kernel loaded from ELF file
3113                    // defined by the kernel linker script in kernel.ld
3114
3115 struct run {
3116   struct run *next;
3117 };
3118
3119 struct {
3120   struct spinlock lock;
3121   int use_lock;
3122   struct run *freelist;
3123 } kmem;
3124
3125 // Initialization happens in two phases.
3126 // 1. main() calls kinit1() while still using entrypgdir to place just
3127 // the pages mapped by entrypgdir on free list.
3128 // 2. main() calls kinit2() with the rest of the physical pages
3129 // after installing a full page table that maps them on all cores.
3130 void
3131 kinit1(void *vstart, void *vend)
3132 {
3133   initlock(&kmem.lock, "kmem");
3134   kmem.use_lock = 0;
3135   freerange(vstart, vend);
3136 }
3137
3138 void
3139 kinit2(void *vstart, void *vend)
3140 {
3141   freerange(vstart, vend);
3142   kmem.use_lock = 1;
3143 }
3144
3145
3146
3147
3148
3149
```

```
3150 void
3151 freerange(void *vstart, void *vend)
3152 {
3153   char *p;
3154   p = (char*)PGROUNDUP((uint)vstart);
3155   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3156     kfree(p);
3157 }
3158
3159
3160 // Free the page of physical memory pointed at by v,
3161 // which normally should have been returned by a
3162 // call to kalloc().  (The exception is when
3163 // initializing the allocator; see kinit above.)
3164 void
3165 kfree(char *v)
3166 {
3167   struct run *r;
3168
3169   if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
3170     panic("kfree");
3171
3172   // Fill with junk to catch dangling refs.
3173   memset(v, 1, PGSIZE);
3174
3175   if(kmem.use_lock)
3176     acquire(&kmem.lock);
3177   r = (struct run*)v;
3178   r->next = kmem.freelist;
3179   kmem.freelist = r;
3180   if(kmem.use_lock)
3181     release(&kmem.lock);
3182 }
3183
3184 // Allocate one 4096-byte page of physical memory.
3185 // Returns a pointer that the kernel can use.
3186 // Returns 0 if the memory cannot be allocated.
3187 char*
3188 kalloc(void)
3189 {
3190   struct run *r;
3191
3192   if(kmem.use_lock)
3193     acquire(&kmem.lock);
3194   r = kmem.freelist;
3195   if(r)
3196     kmem.freelist = r->next;
3197   if(kmem.use_lock)
3198     release(&kmem.lock);
3199   return (char*)r;
```



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

```
3250 // x86 trap and interrupt constants.
3251
3252 // Processor-defined:
3253 #define T_DIVIDE         0      // divide error
3254 #define T_DEBUG          1      // debug exception
3255 #define T_NMI            2      // non-maskable interrupt
3256 #define T_BRKPT          3      // breakpoint
3257 #define T_OFLOW          4      // overflow
3258 #define T_BOUND          5      // bounds check
3259 #define T_ILLOP          6      // illegal opcode
3260 #define T_DEVICE         7      // device not available
3261 #define T_DBLFLT         8      // double fault
3262 // #define T_COPROC      9      // reserved (not used since 486)
3263 #define T_TSS           10      // invalid task switch segment
3264 #define T_SEGNP         11      // segment not present
3265 #define T_STACK         12      // stack exception
3266 #define T_GPFLT         13      // general protection fault
3267 #define T_PGFLT         14      // page fault
3268 // #define T_RES        15      // reserved
3269 #define T_FPERR         16      // floating point error
3270 #define T_ALIGN         17      // aligment check
3271 #define T_MCHK          18      // machine check
3272 #define T_SIMDERR       19      // SIMD floating point error
3273
3274 // These are arbitrarily chosen, but with care not to overlap
3275 // processor defined exceptions or interrupt vectors.
3276 #define T_SYSCALL       64      // system call
3277 #define T_DEFAULT      500      // catchall
3278
3279 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
3280
3281 #define IRQ_TIMER        0
3282 #define IRQ_KBD          1
3283 #define IRQ_COM1         4
3284 #define IRQ_IDE         14
3285 #define IRQ_ERROR       19
3286 #define IRQ_SPURIOUS    31
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
```



```
3300 #!/usr/bin/perl -w
3301
3302 # Generate vectors.S, the trap/interrupt entry points.
3303 # There has to be one entry point per interrupt number
3304 # since otherwise there's no way for trap() to discover
3305 # the interrupt number.
3306
3307 print "# generated by vectors.pl - do not edit\n";
3308 print "# handlers\n";
3309 print ".globl alltraps\n";
3310 for(my $i = 0; $i < 256; $i++){
3311     print ".globl vector$i\n";
3312     print "vector$i:\n";
3313     if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
3314         print "  pushl \$0\n";
3315     }
3316     print "  pushl \$$i\n";
3317     print "  jmp alltraps\n";
3318 }
3319
3320 print "\n# vector table\n";
3321 print ".data\n";
3322 print ".globl vectors\n";
3323 print "vectors:\n";
3324 for(my $i = 0; $i < 256; $i++){
3325     print "  .long vector$i\n";
3326 }
3327
3328 # sample output:
3329 #   # handlers
3330 #   .globl alltraps
3331 #   .globl vector0
3332 #   vector0:
3333 #     pushl $0
3334 #     pushl $0
3335 #     jmp alltraps
3336 #   ...
3337 #
3338 #   # vector table
3339 #   .data
3340 #   .globl vectors
3341 #   vectors:
3342 #     .long vector0
3343 #     .long vector1
3344 #     .long vector2
3345 #   ...
3346
3347
3348
3349
```

```
3350 #include "mmu.h"
3351
3352   # vectors.S sends all traps here.
3353 .globl alltraps
3354 alltraps:
3355   # Build trap frame.
3356   pushl %ds
3357   pushl %es
3358   pushl %fs
3359   pushl %gs
3360   pushal
3361
3362   # Set up data segments.
3363   movw $(SEG_KDATA<<3), %ax
3364   movw %ax, %ds
3365   movw %ax, %es
3366
3367   # Call trap(tf), where tf=%esp
3368   pushl %esp
3369   call trap
3370   addl $4, %esp
3371
3372   # Return falls through to trapret...
3373 .globl trapret
3374 trapret:
3375   popal
3376   popl %gs
3377   popl %fs
3378   popl %es
3379   popl %ds
3380   addl $0x8, %esp  # trapno and errcode
3381   iret
3382
3383
3384
3385
3386
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399
```



```
3400 #include "types.h"
3401 #include "defs.h"
3402 #include "param.h"
3403 #include "memlayout.h"
3404 #include "mmu.h"
3405 #include "proc.h"
3406 #include "x86.h"
3407 #include "traps.h"
3408 #include "spinlock.h"
3409
3410 // Interrupt descriptor table (shared by all CPUs).
3411 struct gatedesc idt[256];
3412 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
3413 struct spinlock tickslock;
3414 uint ticks;
3415
3416 void
3417 tvinit(void)
3418 {
3419   int i;
3420
3421   for(i = 0; i < 256; i++)
3422     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3423   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3424
3425   initlock(&tickslock, "time");
3426 }
3427
3428 void
3429 idtinit(void)
3430 {
3431   lidt(idt, sizeof(idt));
3432 }
3433
3434
3435
3436
3437
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
```

```
3450 void
3451 trap(struct trapframe *tf)
3452 {
3453   if(tf->trapno == T_SYSCALL){
3454     if(myproc()->killed)
3455       exit();
3456     myproc()->tf = tf;
3457     syscall();
3458     if(myproc()->killed)
3459       exit();
3460     return;
3461   }
3462
3463   switch(tf->trapno){
3464   case T_IRQ0 + IRQ_TIMER:
3465     if(cpuid() == 0){
3466       acquire(&tickslock);
3467       ticks++;
3468       wakeup(&ticks);
3469       release(&tickslock);
3470     }
3471     lapiceoi();
3472     break;
3473   case T_IRQ0 + IRQ_IDE:
3474     ideintr();
3475     lapiceoi();
3476     break;
3477   case T_IRQ0 + IRQ_IDE+1:
3478     // Bochs generates spurious IDE1 interrupts.
3479     break;
3480   case T_IRQ0 + IRQ_KBD:
3481     kbdintr();
3482     lapiceoi();
3483     break;
3484   case T_IRQ0 + IRQ_COM1:
3485     uartintr();
3486     lapiceoi();
3487     break;
3488   case T_IRQ0 + 7:
3489   case T_IRQ0 + IRQ_SPURIOUS:
3490     cprintf("cpu%d: spurious interrupt at %x:%x\n",
3491             cpuid(), tf->cs, tf->eip);
3492     lapiceoi();
3493     break;
3494
3495
3496
3497
3498
3499
```



```
3500   default:
3501     if(myproc() == 0 || (tf->cs&3) == 0){
3502       // In kernel, it must be our mistake.
3503       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3504               tf->trapno, cpuid(), tf->eip, rcr2());
3505       panic("trap");
3506     }
3507     // In user space, assume process misbehaved.
3508     cprintf("pid %d %s: trap %d err %d on cpu %d "
3509             "eip 0x%x addr 0x%x--kill proc\n",
3510             myproc()->pid, myproc()->name, tf->trapno, tf->err, cpuid(), tf->
3511             rcr2());
3512     myproc()->killed = 1;
3513   }
3514
3515   // Force process exit if it has been killed and is in user space.
3516   // (If it is still executing in the kernel, let it keep running
3517   // until it gets to the regular system call return.)
3518   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
3519     exit();
3520
3521   // Force process to give up CPU on clock tick.
3522   // If interrupts were on while locks held, would need to check nlock.
3523   if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMEI
3524     yield();
3525
3526   // Check if the process has been killed since we yielded
3527   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
3528     exit();
3529 }
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
```

```
3550 // System call numbers
3551 #define SYS_fork    1
3552 #define SYS_exit    2
3553 #define SYS_wait    3
3554 #define SYS_pipe    4
3555 #define SYS_read    5
3556 #define SYS_kill    6
3557 #define SYS_exec    7
3558 #define SYS_fstat   8
3559 #define SYS_chdir   9
3560 #define SYS_dup    10
3561 #define SYS_getpid 11
3562 #define SYS_sbrk   12
3563 #define SYS_sleep  13
3564 #define SYS_uptime 14
3565 #define SYS_open   15
3566 #define SYS_write  16
3567 #define SYS_mknod  17
3568 #define SYS_unlink 18
3569 #define SYS_link   19
3570 #define SYS_mkdir  20
3571 #define SYS_close  21
3572
3573
3574
3575
3576
3577
3578
3579
3580
3581
3582
3583
3584
3585
3586
3587
3588
3589
3590
3591
3592
3593
3594
3595
3596
3597
3598
3599
```



```
3600 #include "types.h"
3601 #include "defs.h"
3602 #include "param.h"
3603 #include "memlayout.h"
3604 #include "mmu.h"
3605 #include "proc.h"
3606 #include "x86.h"
3607 #include "syscall.h"
3608
3609 // User code makes a system call with INT T_SYSCALL.
3610 // System call number in %eax.
3611 // Arguments on the stack, from the user call to the C
3612 // library system call function. The saved user %esp points
3613 // to a saved program counter, and then the first argument.
3614
3615 // Fetch the int at addr from the current process.
3616 int
3617 fetchint(uint addr, int *ip)
3618 {
3619   struct proc *curproc = myproc();
3620
3621   if(addr >= curproc->sz || addr+4 > curproc->sz)
3622     return -1;
3623   *ip = *(int*)(addr);
3624   return 0;
3625 }
3626
3627 // Fetch the nul-terminated string at addr from the current process.
3628 // Doesn't actually copy the string - just sets *pp to point at it.
3629 // Returns length of string, not including nul.
3630 int
3631 fetchstr(uint addr, char **pp)
3632 {
3633   char *s, *ep;
3634   struct proc *curproc = myproc();
3635
3636   if(addr >= curproc->sz)
3637     return -1;
3638   *pp = (char*)addr;
3639   ep = (char*)curproc->sz;
3640   for(s = *pp; s < ep; s++){
3641     if(*s == 0)
3642       return s - *pp;
3643   }
3644   return -1;
3645 }
3646
3647
3648
3649
```

```
3650 // Fetch the nth 32-bit system call argument.
3651 int
3652 argint(int n, int *ip)
3653 {
3654   return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
3655 }
3656
3657 // Fetch the nth word-sized system call argument as a pointer
3658 // to a block of memory of size bytes.  Check that the pointer
3659 // lies within the process address space.
3660 int
3661 argptr(int n, char **pp, int size)
3662 {
3663   int i;
3664   struct proc *curproc = myproc();
3665
3666   if(argint(n, &i) < 0)
3667     return -1;
3668   if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
3669     return -1;
3670   *pp = (char*)i;
3671   return 0;
3672 }
3673
3674 // Fetch the nth word-sized system call argument as a string pointer.
3675 // Check that the pointer is valid and the string is nul-terminated.
3676 // (There is no shared writable memory, so the string can't change
3677 // between this check and being used by the kernel.)
3678 int
3679 argstr(int n, char **pp)
3680 {
3681   int addr;
3682   if(argint(n, &addr) < 0)
3683     return -1;
3684   return fetchstr(addr, pp);
3685 }
3686
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699
```



```
3700 extern int sys_chdir(void);
3701 extern int sys_close(void);
3702 extern int sys_dup(void);
3703 extern int sys_exec(void);
3704 extern int sys_exit(void);
3705 extern int sys_fork(void);
3706 extern int sys_fstat(void);
3707 extern int sys_getpid(void);
3708 extern int sys_kill(void);
3709 extern int sys_link(void);
3710 extern int sys_mkdir(void);
3711 extern int sys_mknod(void);
3712 extern int sys_open(void);
3713 extern int sys_pipe(void);
3714 extern int sys_read(void);
3715 extern int sys_sbrk(void);
3716 extern int sys_sleep(void);
3717 extern int sys_unlink(void);
3718 extern int sys_wait(void);
3719 extern int sys_write(void);
3720 extern int sys_uptime(void);
3721
3722 static int (*syscalls[])(void) = {
3723 [SYS_fork]    sys_fork,
3724 [SYS_exit]    sys_exit,
3725 [SYS_wait]    sys_wait,
3726 [SYS_pipe]    sys_pipe,
3727 [SYS_read]    sys_read,
3728 [SYS_kill]    sys_kill,
3729 [SYS_exec]    sys_exec,
3730 [SYS_fstat]   sys_fstat,
3731 [SYS_chdir]   sys_chdir,
3732 [SYS_dup]     sys_dup,
3733 [SYS_getpid]  sys_getpid,
3734 [SYS_sbrk]    sys_sbrk,
3735 [SYS_sleep]   sys_sleep,
3736 [SYS_uptime]  sys_uptime,
3737 [SYS_open]    sys_open,
3738 [SYS_write]   sys_write,
3739 [SYS_mknod]   sys_mknod,
3740 [SYS_unlink]  sys_unlink,
3741 [SYS_link]    sys_link,
3742 [SYS_mkdir]   sys_mkdir,
3743 [SYS_close]   sys_close,
3744 };
3745
3746
3747
3748
3749
```

```
3750 void
3751 syscall(void)
3752 {
3753   int num;
3754   struct proc *curproc = myproc();
3755
3756   num = curproc->tf->eax;
3757   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3758     curproc->tf->eax = syscalls[num]();
3759   } else {
3760     cprintf("%d %s: unknown sys call %d\n",
3761             curproc->pid, curproc->name, num);
3762     curproc->tf->eax = -1;
3763   }
3764 }
3765
3766
3767
3768
3769
3770
3771
3772
3773
3774
3775
3776
3777
3778
3779
3780
3781
3782
3783
3784
3785
3786
3787
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3799
```



```
3800 #include "types.h"
3801 #include "x86.h"
3802 #include "defs.h"
3803 #include "date.h"
3804 #include "param.h"
3805 #include "memlayout.h"
3806 #include "mmu.h"
3807 #include "proc.h"
3808
3809 int
3810 sys_fork(void)
3811 {
3812   return fork();
3813 }
3814
3815 int
3816 sys_exit(void)
3817 {
3818   exit();
3819   return 0;  // not reached
3820 }
3821
3822 int
3823 sys_wait(void)
3824 {
3825   return wait();
3826 }
3827
3828 int
3829 sys_kill(void)
3830 {
3831   int pid;
3832
3833   if(argint(0, &pid) < 0)
3834     return -1;
3835   return kill(pid);
3836 }
3837
3838 int
3839 sys_getpid(void)
3840 {
3841   return myproc()->pid;
3842 }
3843
3844
3845
3846
3847
3848
3849
```

```
3850 int
3851 sys_sbrk(void)
3852 {
3853   int addr;
3854   int n;
3855
3856   if(argint(0, &n) < 0)
3857     return -1;
3858   addr = myproc()->sz;
3859   if(growproc(n) < 0)
3860     return -1;
3861   return addr;
3862 }
3863
3864 int
3865 sys_sleep(void)
3866 {
3867   int n;
3868   uint ticks0;
3869
3870   if(argint(0, &n) < 0)
3871     return -1;
3872   acquire(&tickslock);
3873   ticks0 = ticks;
3874   while(ticks - ticks0 < n){
3875     if(myproc()->killed){
3876       release(&tickslock);
3877       return -1;
3878     }
3879     sleep(&ticks, &tickslock);
3880   }
3881   release(&tickslock);
3882   return 0;
3883 }
3884
3885 // return how many clock tick interrupts have occurred
3886 // since start.
3887 int
3888 sys_uptime(void)
3889 {
3890   uint xticks;
3891
3892   acquire(&tickslock);
3893   xticks = ticks;
3894   release(&tickslock);
3895   return xticks;
3896 }
3897
3898
3899
```



```
3900 struct buf {
3901   int flags;
3902   uint dev;
3903   uint blockno;
3904   struct sleeplock lock;
3905   uint refcnt;
3906   struct buf *prev; // LRU cache list
3907   struct buf *next;
3908   struct buf *qnext; // disk queue
3909   uchar data[BSIZE];
3910 };
3911 #define B_VALID 0x2  // buffer has been read from disk
3912 #define B_DIRTY 0x4  // buffer needs to be written to disk
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 // Long-term locks for processes
3951 struct sleeplock {
3952   uint locked;       // Is the lock held?
3953   struct spinlock lk; // spinlock protecting this sleep lock
3954
3955   // For debugging:
3956   char *name;        // Name of lock.
3957   int pid;           // Process holding lock
3958 };
3959
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 #define O_RDONLY  0x000
4001 #define O_WRONLY  0x001
4002 #define O_RDWR    0x002
4003 #define O_CREATE  0x200
4004
4005
4006
4007
4008
4009
4010
4011
4012
4013
4014
4015
4016
4017
4018
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
```

```
4050 #define T_DIR  1   // Directory
4051 #define T_FILE 2   // File
4052 #define T_DEV  3   // Device
4053
4054 struct stat {
4055   short type;  // Type of file
4056   int dev;     // File system's disk device
4057   uint ino;    // Inode number
4058   short nlink; // Number of links to file
4059   uint size;   // Size of file in bytes
4060 };
4061
4062
4063
4064
4065
4066
4067
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 // On-disk file system format.
4101 // Both the kernel and user programs use this header file.
4102
4103
4104 #define ROOTINO 1  // root i-number
4105 #define BSIZE 512  // block size
4106
4107 // Disk layout:
4108 // [ boot block | super block | log | inode blocks |
4109 //                                          free bit map | data blocks]
4110 //
4111 // mkfs computes the super block and builds an initial file system. The
4112 // super block describes the disk layout:
4113 struct superblock {
4114   uint size;         // Size of file system image (blocks)
4115   uint nblocks;      // Number of data blocks
4116   uint ninodes;      // Number of inodes.
4117   uint nlog;         // Number of log blocks
4118   uint logstart;     // Block number of first log block
4119   uint inodestart;   // Block number of first inode block
4120   uint bmapstart;    // Block number of first free map block
4121 };
4122
4123 #define NDIRECT 12
4124 #define NINDIRECT (BSIZE / sizeof(uint))
4125 #define MAXFILE (NDIRECT + NINDIRECT)
4126
4127 // On-disk inode structure
4128 struct dinode {
4129   short type;           // File type
4130   short major;          // Major device number (T_DEV only)
4131   short minor;          // Minor device number (T_DEV only)
4132   short nlink;          // Number of links to inode in file system
4133   uint size;            // Size of file (bytes)
4134   uint addrs[NDIRECT+1];   // Data block addresses
4135 };
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```

```
4150 // Inodes per block.
4151 #define IPB           (BSIZE / sizeof(struct dinode))
4152
4153 // Block containing inode i
4154 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
4155
4156 // Bitmap bits per block
4157 #define BPB           (BSIZE*8)
4158
4159 // Block of free map containing bit for block b
4160 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
4161
4162 // Directory is a file containing a sequence of dirent structures.
4163 #define DIRSIZ 14
4164
4165 struct dirent {
4166   ushort inum;
4167   char name[DIRSIZ];
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 struct file {
4201   enum { FD_NONE, FD_PIPE, FD_INODE } type;
4202   int ref; // reference count
4203   char readable;
4204   char writable;
4205   struct pipe *pipe;
4206   struct inode *ip;
4207   uint off;
4208 };
4209
4210
4211 // in-memory copy of an inode
4212 struct inode {
4213   uint dev;           // Device number
4214   uint inum;          // Inode number
4215   int ref;            // Reference count
4216   struct sleeplock lock; // protects everything below here
4217   int valid;          // inode has been read from disk?
4218
4219   short type;         // copy of disk inode
4220   short major;
4221   short minor;
4222   short nlink;
4223   uint size;
4224   uint addrs[NDIRECT+1];
4225 };
4226
4227 // table mapping major device number to
4228 // device functions
4229 struct devsw {
4230   int (*read)(struct inode*, char*, int);
4231   int (*write)(struct inode*, char*, int);
4232 };
4233
4234 extern struct devsw devsw[];
4235
4236 #define CONSOLE 1
4237
4238
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
```

```
4250 // Blank page.
4251
4252
4253
4254
4255
4256
4257
4258
4259
4260
4261
4262
4263
4264
4265
4266
4267
4268
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
```



```
4300 // Simple PIO-based (non-DMA) IDE driver code.
4301
4302 #include "types.h"
4303 #include "defs.h"
4304 #include "param.h"
4305 #include "memlayout.h"
4306 #include "mmu.h"
4307 #include "proc.h"
4308 #include "x86.h"
4309 #include "traps.h"
4310 #include "spinlock.h"
4311 #include "sleeplock.h"
4312 #include "fs.h"
4313 #include "buf.h"
4314
4315 #define SECTOR_SIZE   512
4316 #define IDE_BSY       0x80
4317 #define IDE_DRDY      0x40
4318 #define IDE_DF        0x20
4319 #define IDE_ERR       0x01
4320
4321 #define IDE_CMD_READ  0x20
4322 #define IDE_CMD_WRITE 0x30
4323 #define IDE_CMD_RDMUL 0xc4
4324 #define IDE_CMD_WRMUL 0xc5
4325
4326 // idequeue points to the buf now being read/written to the disk.
4327 // idequeue->qnext points to the next buf to be processed.
4328 // You must hold idelock while manipulating queue.
4329
4330 static struct spinlock idelock;
4331 static struct buf *idequeue;
4332
4333 static int havedisk1;
4334 static void idestart(struct buf*);
4335
4336 // Wait for IDE disk to become ready.
4337 static int
4338 idewait(int checkerr)
4339 {
4340   int r;
4341
4342   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4343     ;
4344   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4345     return -1;
4346   return 0;
4347 }
4348
4349
```

```
4350 void
4351 ideinit(void)
4352 {
4353   int i;
4354
4355   initlock(&idelock, "ide");
4356   ioapicenable(IRQ_IDE, ncpu - 1);
4357   idewait(0);
4358
4359   // Check if disk 1 is present
4360   outb(0x1f6, 0xe0 | (1<<4));
4361   for(i=0; i<1000; i++){
4362     if(inb(0x1f7) != 0){
4363       havedisk1 = 1;
4364       break;
4365     }
4366   }
4367
4368   // Switch back to disk 0.
4369   outb(0x1f6, 0xe0 | (0<<4));
4370 }
4371
4372 // Start the request for b.  Caller must hold idelock.
4373 static void
4374 idestart(struct buf *b)
4375 {
4376   if(b == 0)
4377     panic("idestart");
4378   if(b->blockno >= FSSIZE)
4379     panic("incorrect blockno");
4380   int sector_per_block =  BSIZE/SECTOR_SIZE;
4381   int sector = b->blockno * sector_per_block;
4382   int read_cmd = (sector_per_block == 1) ? IDE_CMD_READ :  IDE_CMD_RDMUL;
4383   int write_cmd = (sector_per_block == 1) ? IDE_CMD_WRITE : IDE_CMD_WRMUL;
4384
4385   if (sector_per_block > 7) panic("idestart");
4386
4387   idewait(0);
4388   outb(0x3f6, 0);  // generate interrupt
4389   outb(0x1f2, sector_per_block);  // number of sectors
4390   outb(0x1f3, sector & 0xff);
4391   outb(0x1f4, (sector >> 8) & 0xff);
4392   outb(0x1f5, (sector >> 16) & 0xff);
4393   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4394   if(b->flags & B_DIRTY){
4395     outb(0x1f7, write_cmd);
4396     outsl(0x1f0, b->data, BSIZE/4);
4397   } else {
4398     outb(0x1f7, read_cmd);
4399   }
```



```
4400 }
4401
4402 // Interrupt handler.
4403 void
4404 ideintr(void)
4405 {
4406   struct buf *b;
4407
4408   // First queued buffer is the active request.
4409   acquire(&idelock);
4410
4411   if((b = idequeue) == 0){
4412     release(&idelock);
4413     return;
4414   }
4415   idequeue = b->qnext;
4416
4417   // Read data if needed.
4418   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4419     insl(0x1f0, b->data, BSIZE/4);
4420
4421   // Wake process waiting for this buf.
4422   b->flags |= B_VALID;
4423   b->flags &= ~B_DIRTY;
4424   wakeup(b);
4425
4426   // Start disk on next buf in queue.
4427   if(idequeue != 0)
4428     idestart(idequeue);
4429
4430   release(&idelock);
4431 }
4432
4433
4434
4435
4436
4437
4438
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
```

```
// Sync buf with disk.
// If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
// Else if B_VALID is not set, read buf from disk, set B_VALID.
void
iderw(struct buf *b)
{
  struct buf **pp;

  if(!holdingsleep(&b->lock))
    panic("iderw: buf not locked");
  if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
    panic("iderw: nothing to do");
  if(b->dev != 0 && !havedisk1)
    panic("iderw: ide disk 1 not present");

  acquire(&idelock);

  // Append b to idequeue.
  b->qnext = 0;
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
    ;
  *pp = b;

  // Start disk if necessary.
  if(idequeue == b)
    idestart(b);

  // Wait for request to finish.
  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
    sleep(b, &idelock);
  }


  release(&idelock);
}
```



```
// Buffer cache.
//
// The buffer cache is a linked list of buf structures holding
// cached copies of disk block contents.  Caching disk blocks
// in memory reduces the number of disk reads and also provides
// a synchronization point for disk blocks used by multiple processes.
//
// Interface:
// * To get a buffer for a particular disk block, call bread.
// * After changing buffer data, call bwrite to write it to disk.
// * When done with the buffer, call brelse.
// * Do not use the buffer after calling brelse.
// * Only one process at a time can use a buffer,
//     so do not keep them longer than necessary.
//
// The implementation uses two state flags internally:
// * B_VALID: the buffer data has been read from the disk.
// * B_DIRTY: the buffer data has been modified
//     and needs to be written to disk.

#include "types.h"
#include "defs.h"
#include "param.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "fs.h"
#include "buf.h"

struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // head.next is most recently used.
  struct buf head;
} bcache;

void
binit(void)
{
  struct buf *b;

  initlock(&bcache.lock, "bcache");
```

```
4550   // Create linked list of buffers
4551   bcache.head.prev = &bcache.head;
4552   bcache.head.next = &bcache.head;
4553   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4554     b->next = bcache.head.next;
4555     b->prev = &bcache.head;
4556     initsleeplock(&b->lock, "buffer");
4557     bcache.head.next->prev = b;
4558     bcache.head.next = b;
4559   }
4560 }
4561
4562 // Look through buffer cache for block on device dev.
4563 // If not found, allocate a buffer.
4564 // In either case, return locked buffer.
4565 static struct buf*
4566 bget(uint dev, uint blockno)
4567 {
4568   struct buf *b;
4569
4570   acquire(&bcache.lock);
4571
4572   // Is the block already cached?
4573   for(b = bcache.head.next; b != &bcache.head; b = b->next){
4574     if(b->dev == dev && b->blockno == blockno){
4575       b->refcnt++;
4576       release(&bcache.lock);
4577       acquiresleep(&b->lock);
4578       return b;
4579     }
4580   }
4581
4582   // Not cached; recycle an unused buffer.
4583   // Even if refcnt==0, B_DIRTY indicates a buffer is in use
4584   // because log.c has modified it but not yet committed it.
4585   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4586     if(b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
4587       b->dev = dev;
4588       b->blockno = blockno;
4589       b->flags = 0;
4590       b->refcnt = 1;
4591       release(&bcache.lock);
4592       acquiresleep(&b->lock);
4593       return b;
4594     }
4595   }
4596   panic("bget: no buffers");
4597 }
4598
4599
```



```
4600 // Return a locked buf with the contents of the indicated block.
4601 struct buf*
4602 bread(uint dev, uint blockno)
4603 {
4604   struct buf *b;
4605
4606   b = bget(dev, blockno);
4607   if((b->flags & B_VALID) == 0) {
4608     iderw(b);
4609   }
4610   return b;
4611 }
4612
4613 // Write b's contents to disk.  Must be locked.
4614 void
4615 bwrite(struct buf *b)
4616 {
4617   if(!holdingsleep(&b->lock))
4618     panic("bwrite");
4619   b->flags |= B_DIRTY;
4620   iderw(b);
4621 }
4622
4623 // Release a locked buffer.
4624 // Move to the head of the MRU list.
4625 void
4626 brelse(struct buf *b)
4627 {
4628   if(!holdingsleep(&b->lock))
4629     panic("brelse");
4630
4631   releasesleep(&b->lock);
4632
4633   acquire(&bcache.lock);
4634   b->refcnt--;
4635   if (b->refcnt == 0) {
4636     // no one is waiting for it.
4637     b->next->prev = b->prev;
4638     b->prev->next = b->next;
4639     b->next = bcache.head.next;
4640     b->prev = &bcache.head;
4641     bcache.head.next->prev = b;
4642     bcache.head.next = b;
4643   }
4644
4645   release(&bcache.lock);
4646 }
4647
4648
4649
```

```
4650 // Blank page.
4651
4652
4653
4654
4655
4656
4657
4658
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 // Sleeping locks
4701
4702 #include "types.h"
4703 #include "defs.h"
4704 #include "param.h"
4705 #include "x86.h"
4706 #include "memlayout.h"
4707 #include "mmu.h"
4708 #include "proc.h"
4709 #include "spinlock.h"
4710 #include "sleeplock.h"
4711
4712 void
4713 initsleeplock(struct sleeplock *lk, char *name)
4714 {
4715   initlock(&lk->lk, "sleep lock");
4716   lk->name = name;
4717   lk->locked = 0;
4718   lk->pid = 0;
4719 }
4720
4721 void
4722 acquiresleep(struct sleeplock *lk)
4723 {
4724   acquire(&lk->lk);
4725   while (lk->locked) {
4726     sleep(lk, &lk->lk);
4727   }
4728   lk->locked = 1;
4729   lk->pid = myproc()->pid;
4730   release(&lk->lk);
4731 }
4732
4733 void
4734 releasesleep(struct sleeplock *lk)
4735 {
4736   acquire(&lk->lk);
4737   lk->locked = 0;
4738   lk->pid = 0;
4739   wakeup(lk);
4740   release(&lk->lk);
4741 }
4742
4743
4744
4745
4746
4747
4748
4749
```

```
4750 int
4751 holdingsleep(struct sleeplock *lk)
4752 {
4753   int r;
4754
4755   acquire(&lk->lk);
4756   r = lk->locked;
4757   release(&lk->lk);
4758   return r;
4759 }
4760
4761
4762
4763
4764
4765
4766
4767
4768
4769
4770
4771
4772
4773
4774
4775
4776
4777
4778
4779
4780
4781
4782
4783
4784
4785
4786
4787
4788
4789
4790
4791
4792
4793
4794
4795
4796
4797
4798
4799
```



```
4800 #include "types.h"
4801 #include "defs.h"
4802 #include "param.h"
4803 #include "spinlock.h"
4804 #include "sleeplock.h"
4805 #include "fs.h"
4806 #include "buf.h"
4807
4808 // Simple logging that allows concurrent FS system calls.
4809 //
4810 // A log transaction contains the updates of multiple FS system
4811 // calls. The logging system only commits when there are
4812 // no FS system calls active. Thus there is never
4813 // any reasoning required about whether a commit might
4814 // write an uncommitted system call's updates to disk.
4815 //
4816 // A system call should call begin_op()/end_op() to mark
4817 // its start and end. Usually begin_op() just increments
4818 // the count of in-progress FS system calls and returns.
4819 // But if it thinks the log is close to running out, it
4820 // sleeps until the last outstanding end_op() commits.
4821 //
4822 // The log is a physical re-do log containing disk blocks.
4823 // The on-disk log format:
4824 //   header block, containing block #s for block A, B, C, ...
4825 //   block A
4826 //   block B
4827 //   block C
4828 //   ...
4829 // Log appends are synchronous.
4830
4831 // Contents of the header block, used for both the on-disk header block
4832 // and to keep track in memory of logged block# before commit.
4833 struct logheader {
4834   int n;
4835   int block[LOGSIZE];
4836 };
4837
4838 struct log {
4839   struct spinlock lock;
4840   int start;
4841   int size;
4842   int outstanding; // how many FS sys calls are executing.
4843   int committing;  // in commit(), please wait.
4844   int dev;
4845   struct logheader lh;
4846 };
4847
4848
4849
```

```
4850 struct log log;
4851
4852 static void recover_from_log(void);
4853 static void commit();
4854
4855 void
4856 initlog(int dev)
4857 {
4858   if (sizeof(struct logheader) >= BSIZE)
4859     panic("initlog: too big logheader");
4860
4861   struct superblock sb;
4862   initlock(&log.lock, "log");
4863   readsb(dev, &sb);
4864   log.start = sb.logstart;
4865   log.size = sb.nlog;
4866   log.dev = dev;
4867   recover_from_log();
4868 }
4869
4870 // Copy committed blocks from log to their home location
4871 static void
4872 install_trans(void)
4873 {
4874   int tail;
4875
4876   for (tail = 0; tail < log.lh.n; tail++) {
4877     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
4878     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
4879     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
4880     bwrite(dbuf);  // write dst to disk
4881     brelse(lbuf);
4882     brelse(dbuf);
4883   }
4884 }
4885
4886 // Read the log header from disk into the in-memory log header
4887 static void
4888 read_head(void)
4889 {
4890   struct buf *buf = bread(log.dev, log.start);
4891   struct logheader *lh = (struct logheader *) (buf->data);
4892   int i;
4893   log.lh.n = lh->n;
4894   for (i = 0; i < log.lh.n; i++) {
4895     log.lh.block[i] = lh->block[i];
4896   }
4897   brelse(buf);
4898 }
4899
```



```
4900 // Write in-memory log header to disk.
4901 // This is the true point at which the
4902 // current transaction commits.
4903 static void
4904 write_head(void)
4905 {
4906   struct buf *buf = bread(log.dev, log.start);
4907   struct logheader *hb = (struct logheader *) (buf->data);
4908   int i;
4909   hb->n = log.lh.n;
4910   for (i = 0; i < log.lh.n; i++) {
4911     hb->block[i] = log.lh.block[i];
4912   }
4913   bwrite(buf);
4914   brelse(buf);
4915 }
4916
4917 static void
4918 recover_from_log(void)
4919 {
4920   read_head();
4921   install_trans(); // if committed, copy from log to disk
4922   log.lh.n = 0;
4923   write_head(); // clear the log
4924 }
4925
4926 // called at the start of each FS system call.
4927 void
4928 begin_op(void)
4929 {
4930   acquire(&log.lock);
4931   while(1){
4932     if(log.committing){
4933       sleep(&log, &log.lock);
4934     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
4935       // this op might exhaust log space; wait for commit.
4936       sleep(&log, &log.lock);
4937     } else {
4938       log.outstanding += 1;
4939       release(&log.lock);
4940       break;
4941     }
4942   }
4943 }
4944
4945
4946
4947
4948
4949
```

```
4950 // called at the end of each FS system call.
4951 // commits if this was the last outstanding operation.
4952 void
4953 end_op(void)
4954 {
4955   int do_commit = 0;
4956
4957   acquire(&log.lock);
4958   log.outstanding -= 1;
4959   if(log.committing)
4960     panic("log.committing");
4961   if(log.outstanding == 0){
4962     do_commit = 1;
4963     log.committing = 1;
4964   } else {
4965     // begin_op() may be waiting for log space,
4966     // and decrementing log.outstanding has decreased
4967     // the amount of reserved space.
4968     wakeup(&log);
4969   }
4970   release(&log.lock);
4971
4972   if(do_commit){
4973     // call commit w/o holding locks, since not allowed
4974     // to sleep with locks.
4975     commit();
4976     acquire(&log.lock);
4977     log.committing = 0;
4978     wakeup(&log);
4979     release(&log.lock);
4980   }
4981 }
4982
4983 // Copy modified blocks from cache to log.
4984 static void
4985 write_log(void)
4986 {
4987   int tail;
4988
4989   for (tail = 0; tail < log.lh.n; tail++) {
4990     struct buf *to = bread(log.dev, log.start+tail+1); // log block
4991     struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
4992     memmove(to->data, from->data, BSIZE);
4993     bwrite(to);  // write the log
4994     brelse(from);
4995     brelse(to);
4996   }
4997 }
4998
4999
```



```
5000 static void
5001 commit()
5002 {
5003   if (log.lh.n > 0) {
5004     write_log();     // Write modified blocks from cache to log
5005     write_head();    // Write header to disk -- the real commit
5006     install_trans(); // Now install writes to home locations
5007     log.lh.n = 0;
5008     write_head();    // Erase the transaction from the log
5009   }
5010 }
5011
5012 // Caller has modified b->data and is done with the buffer.
5013 // Record the block number and pin in the cache with B_DIRTY.
5014 // commit()/write_log() will do the disk write.
5015 //
5016 // log_write() replaces bwrite(); a typical use is:
5017 //   bp = bread(...)
5018 //   modify bp->data[]
5019 //   log_write(bp)
5020 //   brelse(bp)
5021 void
5022 log_write(struct buf *b)
5023 {
5024   int i;
5025
5026   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
5027     panic("too big a transaction");
5028   if (log.outstanding < 1)
5029     panic("log_write outside of trans");
5030
5031   acquire(&log.lock);
5032   for (i = 0; i < log.lh.n; i++) {
5033     if (log.lh.block[i] == b->blockno)   // log absorbtion
5034       break;
5035   }
5036   log.lh.block[i] = b->blockno;
5037   if (i == log.lh.n)
5038     log.lh.n++;
5039   b->flags |= B_DIRTY; // prevent eviction
5040   release(&log.lock);
5041 }
5042
5043
5044
5045
5046
5047
5048
5049
```

```
5050 // File system implementation.  Five layers:
5051 //   + Blocks: allocator for raw disk blocks.
5052 //   + Log: crash recovery for multi-step updates.
5053 //   + Files: inode allocator, reading, writing, metadata.
5054 //   + Directories: inode with special contents (list of other inodes!)
5055 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
5056 //
5057 // This file contains the low-level file system manipulation
5058 // routines.  The (higher-level) system call implementations
5059 // are in sysfile.c.
5060
5061 #include "types.h"
5062 #include "defs.h"
5063 #include "param.h"
5064 #include "stat.h"
5065 #include "mmu.h"
5066 #include "proc.h"
5067 #include "spinlock.h"
5068 #include "sleeplock.h"
5069 #include "fs.h"
5070 #include "buf.h"
5071 #include "file.h"
5072
5073 #define min(a, b) ((a) < (b) ? (a) : (b))
5074 static void itrunc(struct inode*);
5075 // there should be one superblock per disk device, but we run with
5076 // only one device
5077 struct superblock sb;
5078
5079 // Read the super block.
5080 void
5081 readsb(int dev, struct superblock *sb)
5082 {
5083   struct buf *bp;
5084
5085   bp = bread(dev, 1);
5086   memmove(sb, bp->data, sizeof(*sb));
5087   brelse(bp);
5088 }
5089
5090
5091
5092
5093
5094
5095
5096
5097
5098
5099
```



```
5100 // Zero a block.
5101 static void
5102 bzero(int dev, int bno)
5103 {
5104   struct buf *bp;
5105
5106   bp = bread(dev, bno);
5107   memset(bp->data, 0, BSIZE);
5108   log_write(bp);
5109   brelse(bp);
5110 }
5111
5112 // Blocks.
5113
5114 // Allocate a zeroed disk block.
5115 static uint
5116 balloc(uint dev)
5117 {
5118   int b, bi, m;
5119   struct buf *bp;
5120
5121   bp = 0;
5122   for(b = 0; b < sb.size; b += BPB){
5123     bp = bread(dev, BBLOCK(b, sb));
5124     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
5125       m = 1 << (bi % 8);
5126       if((bp->data[bi/8] & m) == 0){  // Is block free?
5127         bp->data[bi/8] |= m;  // Mark block in use.
5128         log_write(bp);
5129         brelse(bp);
5130         bzero(dev, b + bi);
5131         return b + bi;
5132       }
5133     }
5134     brelse(bp);
5135   }
5136   panic("balloc: out of blocks");
5137 }
5138
5139
5140
5141
5142
5143
5144
5145
5146
5147
5148
5149
```

```
5150 // Free a disk block.
5151 static void
5152 bfree(int dev, uint b)
5153 {
5154   struct buf *bp;
5155   int bi, m;
5156
5157   readsb(dev, &sb);
5158   bp = bread(dev, BBLOCK(b, sb));
5159   bi = b % BPB;
5160   m = 1 << (bi % 8);
5161   if((bp->data[bi/8] & m) == 0)
5162     panic("freeing free block");
5163   bp->data[bi/8] &= ~m;
5164   log_write(bp);
5165   brelse(bp);
5166 }
5167
5168 // Inodes.
5169 //
5170 // An inode describes a single unnamed file.
5171 // The inode disk structure holds metadata: the file's type,
5172 // its size, the number of links referring to it, and the
5173 // list of blocks holding the file's content.
5174 //
5175 // The inodes are laid out sequentially on disk at
5176 // sb.startinode. Each inode has a number, indicating its
5177 // position on the disk.
5178 //
5179 // The kernel keeps a cache of in-use inodes in memory
5180 // to provide a place for synchronizing access
5181 // to inodes used by multiple processes. The cached
5182 // inodes include book-keeping information that is
5183 // not stored on disk: ip->ref and ip->valid.
5184 //
5185 // An inode and its in-memory representation go through a
5186 // sequence of states before they can be used by the
5187 // rest of the file system code.
5188 //
5189 // * Allocation: an inode is allocated if its type (on disk)
5190 //   is non-zero. ialloc() allocates, and iput() frees if
5191 //   the reference and link counts have fallen to zero.
5192 //
5193 // * Referencing in cache: an entry in the inode cache
5194 //   is free if ip->ref is zero. Otherwise ip->ref tracks
5195 //   the number of in-memory pointers to the entry (open
5196 //   files and current directories). iget() finds or
5197 //   creates a cache entry and increments its ref; iput()
5198 //   decrements ref.
5199 //
```



```
5200 // * Valid: the information (type, size, &c) in an inode
5201 //   cache entry is only correct when ip->valid is 1.
5202 //   ilock() reads the inode from
5203 //   the disk and sets ip->valid, while iput() clears
5204 //   ip->valid if ip->ref has fallen to zero.
5205 //
5206 // * Locked: file system code may only examine and modify
5207 //   the information in an inode and its content if it
5208 //   has first locked the inode.
5209 //
5210 // Thus a typical sequence is:
5211 //   ip = iget(dev, inum)
5212 //   ilock(ip)
5213 //   ... examine and modify ip->xxx ...
5214 //   iunlock(ip)
5215 //   iput(ip)
5216 //
5217 // ilock() is separate from iget() so that system calls can
5218 // get a long-term reference to an inode (as for an open file)
5219 // and only lock it for short periods (e.g., in read()).
5220 // The separation also helps avoid deadlock and races during
5221 // pathname lookup. iget() increments ip->ref so that the inode
5222 // stays cached and pointers to it remain valid.
5223 //
5224 // Many internal file system functions expect the caller to
5225 // have locked the inodes involved; this lets callers create
5226 // multi-step atomic operations.
5227 //
5228 // The icache.lock spin-lock defends the allocation of icache
5229 // entries. Since ip->ref indicates whether an entry is free,
5230 // and ip->dev and ip->inum indicate which i-node an entry
5231 // holds, one must hold icache.lock while using any of those fields.
5232 //
5233 // An ip->lock sleep-lock defends all ip-> fields other than ref,
5234 // dev, and inum.  One must hold ip->lock in order to
5235 // read or write that inode's ip->valid, ip->size, ip->type, &c.
5236
5237 struct {
5238   struct spinlock lock;
5239   struct inode inode[NINODE];
5240 } icache;
5241
5242 void
5243 iinit(int dev)
5244 {
5245   int i = 0;
5246
5247   initlock(&icache.lock, "icache");
5248   for(i = 0; i < NINODE; i++) {
5249     initsleeplock(&icache.inode[i].lock, "inode");
```

```
5250   }
5251
5252   readsb(dev, &sb);
5253   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d\
5254  inodestart %d bmap start %d\n", sb.size, sb.nblocks,
5255           sb.ninodes, sb.nlog, sb.logstart, sb.inodestart,
5256           sb.bmapstart);
5257 }
5258
5259 static struct inode* iget(uint dev, uint inum);
5260
5261
5262
5263
5264
5265
5266
5267
5268
5269
5270
5271
5272
5273
5274
5275
5276
5277
5278
5279
5280
5281
5282
5283
5284
5285
5286
5287
5288
5289
5290
5291
5292
5293
5294
5295
5296
5297
5298
5299
```



```
5300 // Allocate an inode on device dev.
5301 // Mark it as allocated by  giving it type type.
5302 // Returns an unlocked but allocated and referenced inode.
5303 struct inode*
5304 ialloc(uint dev, short type)
5305 {
5306   int inum;
5307   struct buf *bp;
5308   struct dinode *dip;
5309
5310   for(inum = 1; inum < sb.ninodes; inum++){
5311     bp = bread(dev, IBLOCK(inum, sb));
5312     dip = (struct dinode*)bp->data + inum%IPB;
5313     if(dip->type == 0){  // a free inode
5314       memset(dip, 0, sizeof(*dip));
5315       dip->type = type;
5316       log_write(bp);   // mark it allocated on the disk
5317       brelse(bp);
5318       return iget(dev, inum);
5319     }
5320     brelse(bp);
5321   }
5322   panic("ialloc: no inodes");
5323 }
5324
5325 // Copy a modified in-memory inode to disk.
5326 // Must be called after every change to an ip->xxx field
5327 // that lives on disk, since i-node cache is write-through.
5328 // Caller must hold ip->lock.
5329 void
5330 iupdate(struct inode *ip)
5331 {
5332   struct buf *bp;
5333   struct dinode *dip;
5334
5335   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5336   dip = (struct dinode*)bp->data + ip->inum%IPB;
5337   dip->type = ip->type;
5338   dip->major = ip->major;
5339   dip->minor = ip->minor;
5340   dip->nlink = ip->nlink;
5341   dip->size = ip->size;
5342   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5343   log_write(bp);
5344   brelse(bp);
5345 }
5346
5347
5348
5349
```

```
5350 // Find the inode with number inum on device dev
5351 // and return the in-memory copy. Does not lock
5352 // the inode and does not read it from disk.
5353 static struct inode*
5354 iget(uint dev, uint inum)
5355 {
5356   struct inode *ip, *empty;
5357
5358   acquire(&icache.lock);
5359
5360   // Is the inode already cached?
5361   empty = 0;
5362   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5363     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5364       ip->ref++;
5365       release(&icache.lock);
5366       return ip;
5367     }
5368     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
5369       empty = ip;
5370   }
5371
5372   // Recycle an inode cache entry.
5373   if(empty == 0)
5374     panic("iget: no inodes");
5375
5376   ip = empty;
5377   ip->dev = dev;
5378   ip->inum = inum;
5379   ip->ref = 1;
5380   ip->valid = 0;
5381   release(&icache.lock);
5382
5383   return ip;
5384 }
5385
5386 // Increment reference count for ip.
5387 // Returns ip to enable ip = idup(ip1) idiom.
5388 struct inode*
5389 idup(struct inode *ip)
5390 {
5391   acquire(&icache.lock);
5392   ip->ref++;
5393   release(&icache.lock);
5394   return ip;
5395 }
5396
5397
5398
5399
```



```
5400 // Lock the given inode.
5401 // Reads the inode from disk if necessary.
5402 void
5403 ilock(struct inode *ip)
5404 {
5405   struct buf *bp;
5406   struct dinode *dip;
5407
5408   if(ip == 0 || ip->ref < 1)
5409     panic("ilock");
5410
5411   acquiresleep(&ip->lock);
5412
5413   if(ip->valid == 0){
5414     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5415     dip = (struct dinode*)bp->data + ip->inum%IPB;
5416     ip->type = dip->type;
5417     ip->major = dip->major;
5418     ip->minor = dip->minor;
5419     ip->nlink = dip->nlink;
5420     ip->size = dip->size;
5421     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5422     brelse(bp);
5423     ip->valid = 1;
5424     if(ip->type == 0)
5425       panic("ilock: no type");
5426   }
5427 }
5428
5429 // Unlock the given inode.
5430 void
5431 iunlock(struct inode *ip)
5432 {
5433   if(ip == 0 || !holdingsleep(&ip->lock) || ip->ref < 1)
5434     panic("iunlock");
5435
5436   releasesleep(&ip->lock);
5437 }
5438
5439
5440
5441
5442
5443
5444
5445
5446
5447
5448
5449
```

```
5450 // Drop a reference to an in-memory inode.
5451 // If that was the last reference, the inode cache entry can
5452 // be recycled.
5453 // If that was the last reference and the inode has no links
5454 // to it, free the inode (and its content) on disk.
5455 // All calls to iput() must be inside a transaction in
5456 // case it has to free the inode.
5457 void
5458 iput(struct inode *ip)
5459 {
5460   acquiresleep(&ip->lock);
5461   if(ip->valid && ip->nlink == 0){
5462     acquire(&icache.lock);
5463     int r = ip->ref;
5464     release(&icache.lock);
5465     if(r == 1){
5466       // inode has no links and no other references: truncate and free.
5467       itrunc(ip);
5468       ip->type = 0;
5469       iupdate(ip);
5470       ip->valid = 0;
5471     }
5472   }
5473   releasesleep(&ip->lock);
5474
5475   acquire(&icache.lock);
5476   ip->ref--;
5477   release(&icache.lock);
5478 }
5479
5480 // Common idiom: unlock, then put.
5481 void
5482 iunlockput(struct inode *ip)
5483 {
5484   iunlock(ip);
5485   iput(ip);
5486 }
5487
5488
5489
5490
5491
5492
5493
5494
5495
5496
5497
5498
5499
```



```
5500 // Inode content
5501 //
5502 // The content (data) associated with each inode is stored
5503 // in blocks on the disk. The first NDIRECT block numbers
5504 // are listed in ip->addrs[].  The next NINDIRECT blocks are
5505 // listed in block ip->addrs[NDIRECT].
5506
5507 // Return the disk block address of the nth block in inode ip.
5508 // If there is no such block, bmap allocates one.
5509 static uint
5510 bmap(struct inode *ip, uint bn)
5511 {
5512   uint addr, *a;
5513   struct buf *bp;
5514
5515   if(bn < NDIRECT){
5516     if((addr = ip->addrs[bn]) == 0)
5517       ip->addrs[bn] = addr = balloc(ip->dev);
5518     return addr;
5519   }
5520   bn -= NDIRECT;
5521
5522   if(bn < NINDIRECT){
5523     // Load indirect block, allocating if necessary.
5524     if((addr = ip->addrs[NDIRECT]) == 0)
5525       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5526     bp = bread(ip->dev, addr);
5527     a = (uint*)bp->data;
5528     if((addr = a[bn]) == 0){
5529       a[bn] = addr = balloc(ip->dev);
5530       log_write(bp);
5531     }
5532     brelse(bp);
5533     return addr;
5534   }
5535
5536   panic("bmap: out of range");
5537 }
5538
5539
5540
5541
5542
5543
5544
5545
5546
5547
5548
5549
```

```
5550 // Truncate inode (discard contents).
5551 // Only called when the inode has no links
5552 // to it (no directory entries referring to it)
5553 // and has no in-memory reference to it (is
5554 // not an open file or current directory).
5555 static void
5556 itrunc(struct inode *ip)
5557 {
5558   int i, j;
5559   struct buf *bp;
5560   uint *a;
5561
5562   for(i = 0; i < NDIRECT; i++){
5563     if(ip->addrs[i]){
5564       bfree(ip->dev, ip->addrs[i]);
5565       ip->addrs[i] = 0;
5566     }
5567   }
5568
5569   if(ip->addrs[NDIRECT]){
5570     bp = bread(ip->dev, ip->addrs[NDIRECT]);
5571     a = (uint*)bp->data;
5572     for(j = 0; j < NINDIRECT; j++){
5573       if(a[j])
5574         bfree(ip->dev, a[j]);
5575     }
5576     brelse(bp);
5577     bfree(ip->dev, ip->addrs[NDIRECT]);
5578     ip->addrs[NDIRECT] = 0;
5579   }
5580
5581   ip->size = 0;
5582   iupdate(ip);
5583 }
5584
5585 // Copy stat information from inode.
5586 // Caller must hold ip->lock.
5587 void
5588 stati(struct inode *ip, struct stat *st)
5589 {
5590   st->dev = ip->dev;
5591   st->ino = ip->inum;
5592   st->type = ip->type;
5593   st->nlink = ip->nlink;
5594   st->size = ip->size;
5595 }
5596
5597
5598
5599
```



```
5600 // Read data from inode.
5601 // Caller must hold ip->lock.
5602 int
5603 readi(struct inode *ip, char *dst, uint off, uint n)
5604 {
5605   uint tot, m;
5606   struct buf *bp;
5607
5608   if(ip->type == T_DEV){
5609     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
5610       return -1;
5611     return devsw[ip->major].read(ip, dst, n);
5612   }
5613
5614   if(off > ip->size || off + n < off)
5615     return -1;
5616   if(off + n > ip->size)
5617     n = ip->size - off;
5618
5619   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5620     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5621     m = min(n - tot, BSIZE - off%BSIZE);
5622     memmove(dst, bp->data + off%BSIZE, m);
5623     brelse(bp);
5624   }
5625   return n;
5626 }
5627
5628
5629
5630
5631
5632
5633
5634
5635
5636
5637
5638
5639
5640
5641
5642
5643
5644
5645
5646
5647
5648
5649
```

```
5650 // Write data to inode.
5651 // Caller must hold ip->lock.
5652 int
5653 writei(struct inode *ip, char *src, uint off, uint n)
5654 {
5655   uint tot, m;
5656   struct buf *bp;
5657
5658   if(ip->type == T_DEV){
5659     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
5660       return -1;
5661     return devsw[ip->major].write(ip, src, n);
5662   }
5663
5664   if(off > ip->size || off + n < off)
5665     return -1;
5666   if(off + n > MAXFILE*BSIZE)
5667     return -1;
5668
5669   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
5670     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5671     m = min(n - tot, BSIZE - off%BSIZE);
5672     memmove(bp->data + off%BSIZE, src, m);
5673     log_write(bp);
5674     brelse(bp);
5675   }
5676
5677   if(n > 0 && off > ip->size){
5678     ip->size = off;
5679     iupdate(ip);
5680   }
5681   return n;
5682 }
5683
5684
5685
5686
5687
5688
5689
5690
5691
5692
5693
5694
5695
5696
5697
5698
5699
```



```
5700 // Directories
5701
5702 int
5703 namecmp(const char *s, const char *t)
5704 {
5705   return strncmp(s, t, DIRSIZ);
5706 }
5707
5708 // Look for a directory entry in a directory.
5709 // If found, set *poff to byte offset of entry.
5710 struct inode*
5711 dirlookup(struct inode *dp, char *name, uint *poff)
5712 {
5713   uint off, inum;
5714   struct dirent de;
5715
5716   if(dp->type != T_DIR)
5717     panic("dirlookup not DIR");
5718
5719   for(off = 0; off < dp->size; off += sizeof(de)){
5720     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5721       panic("dirlookup read");
5722     if(de.inum == 0)
5723       continue;
5724     if(namecmp(name, de.name) == 0){
5725       // entry matches path element
5726       if(poff)
5727         *poff = off;
5728       inum = de.inum;
5729       return iget(dp->dev, inum);
5730     }
5731   }
5732
5733   return 0;
5734 }
5735
5736
5737
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749
```

```
5750 // Write a new directory entry (name, inum) into the directory dp.
5751 int
5752 dirlink(struct inode *dp, char *name, uint inum)
5753 {
5754   int off;
5755   struct dirent de;
5756   struct inode *ip;
5757
5758   // Check that name is not present.
5759   if((ip = dirlookup(dp, name, 0)) != 0){
5760     iput(ip);
5761     return -1;
5762   }
5763
5764   // Look for an empty dirent.
5765   for(off = 0; off < dp->size; off += sizeof(de)){
5766     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5767       panic("dirlink read");
5768     if(de.inum == 0)
5769       break;
5770   }
5771
5772   strncpy(de.name, name, DIRSIZ);
5773   de.inum = inum;
5774   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5775     panic("dirlink");
5776
5777   return 0;
5778 }
5779
5780
5781
5782
5783
5784
5785
5786
5787
5788
5789
5790
5791
5792
5793
5794
5795
5796
5797
5798
5799
```



```
5800 // Paths
5801
5802 // Copy the next path element from path into name.
5803 // Return a pointer to the element following the copied one.
5804 // The returned path has no leading slashes,
5805 // so the caller can check *path=='\0' to see if the name is the last one.
5806 // If no name to remove, return 0.
5807 //
5808 // Examples:
5809 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5810 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5811 //   skipelem("a", name) = "", setting name = "a"
5812 //   skipelem("", name) = skipelem("////", name) = 0
5813 //
5814 static char*
5815 skipelem(char *path, char *name)
5816 {
5817   char *s;
5818   int len;
5819
5820   while(*path == '/')
5821     path++;
5822   if(*path == 0)
5823     return 0;
5824   s = path;
5825   while(*path != '/' && *path != 0)
5826     path++;
5827   len = path - s;
5828   if(len >= DIRSIZ)
5829     memmove(name, s, DIRSIZ);
5830   else {
5831     memmove(name, s, len);
5832     name[len] = 0;
5833   }
5834   while(*path == '/')
5835     path++;
5836   return path;
5837 }
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```

```
5850 // Look up and return the inode for a path name.
5851 // If parent != 0, return the inode for the parent and copy the final
5852 // path element into name, which must have room for DIRSIZ bytes.
5853 // Must be called inside a transaction since it calls iput().
5854 static struct inode*
5855 namex(char *path, int nameiparent, char *name)
5856 {
5857   struct inode *ip, *next;
5858
5859   if(*path == '/')
5860     ip = iget(ROOTDEV, ROOTINO);
5861   else
5862     ip = idup(myproc()->cwd);
5863
5864   while((path = skipelem(path, name)) != 0){
5865     ilock(ip);
5866     if(ip->type != T_DIR){
5867       iunlockput(ip);
5868       return 0;
5869     }
5870     if(nameiparent && *path == '\0'){
5871       // Stop one level early.
5872       iunlock(ip);
5873       return ip;
5874     }
5875     if((next = dirlookup(ip, name, 0)) == 0){
5876       iunlockput(ip);
5877       return 0;
5878     }
5879     iunlockput(ip);
5880     ip = next;
5881   }
5882   if(nameiparent){
5883     iput(ip);
5884     return 0;
5885   }
5886   return ip;
5887 }
5888
5889 struct inode*
5890 namei(char *path)
5891 {
5892   char name[DIRSIZ];
5893   return namex(path, 0, name);
5894 }
5895
5896
5897
5898
5899
```



```
5900 struct inode*
5901 nameiparent(char *path, char *name)
5902 {
5903   return namex(path, 1, name);
5904 }
5905
5906
5907
5908
5909
5910
5911
5912
5913
5914
5915
5916
5917
5918
5919
5920
5921
5922
5923
5924
5925
5926
5927
5928
5929
5930
5931
5932
5933
5934
5935
5936
5937
5938
5939
5940
5941
5942
5943
5944
5945
5946
5947
5948
5949
```

```
5950 //
5951 // File descriptors
5952 //
5953
5954 #include "types.h"
5955 #include "defs.h"
5956 #include "param.h"
5957 #include "fs.h"
5958 #include "spinlock.h"
5959 #include "sleeplock.h"
5960 #include "file.h"
5961
5962 struct devsw devsw[NDEV];
5963 struct {
5964   struct spinlock lock;
5965   struct file file[NFILE];
5966 } ftable;
5967
5968 void
5969 fileinit(void)
5970 {
5971   initlock(&ftable.lock, "ftable");
5972 }
5973
5974 // Allocate a file structure.
5975 struct file*
5976 filealloc(void)
5977 {
5978   struct file *f;
5979
5980   acquire(&ftable.lock);
5981   for(f = ftable.file; f < ftable.file + NFILE; f++){
5982     if(f->ref == 0){
5983       f->ref = 1;
5984       release(&ftable.lock);
5985       return f;
5986     }
5987   }
5988   release(&ftable.lock);
5989   return 0;
5990 }
5991
5992
5993
5994
5995
5996
5997
5998
5999
```



```
6000 // Increment ref count for file f.
6001 struct file*
6002 filedup(struct file *f)
6003 {
6004   acquire(&ftable.lock);
6005   if(f->ref < 1)
6006     panic("filedup");
6007   f->ref++;
6008   release(&ftable.lock);
6009   return f;
6010 }
6011
6012 // Close file f.  (Decrement ref count, close when reaches 0.)
6013 void
6014 fileclose(struct file *f)
6015 {
6016   struct file ff;
6017
6018   acquire(&ftable.lock);
6019   if(f->ref < 1)
6020     panic("fileclose");
6021   if(--f->ref > 0){
6022     release(&ftable.lock);
6023     return;
6024   }
6025   ff = *f;
6026   f->ref = 0;
6027   f->type = FD_NONE;
6028   release(&ftable.lock);
6029
6030   if(ff.type == FD_PIPE)
6031     pipeclose(ff.pipe, ff.writable);
6032   else if(ff.type == FD_INODE){
6033     begin_op();
6034     iput(ff.ip);
6035     end_op();
6036   }
6037 }
6038
6039
6040
6041
6042
6043
6044
6045
6046
6047
6048
6049
```

```
6050 // Get metadata about file f.
6051 int
6052 filestat(struct file *f, struct stat *st)
6053 {
6054   if(f->type == FD_INODE){
6055     ilock(f->ip);
6056     stati(f->ip, st);
6057     iunlock(f->ip);
6058     return 0;
6059   }
6060   return -1;
6061 }
6062
6063 // Read from file f.
6064 int
6065 fileread(struct file *f, char *addr, int n)
6066 {
6067   int r;
6068
6069   if(f->readable == 0)
6070     return -1;
6071   if(f->type == FD_PIPE)
6072     return piperead(f->pipe, addr, n);
6073   if(f->type == FD_INODE){
6074     ilock(f->ip);
6075     if((r = readi(f->ip, addr, f->off, n)) > 0)
6076       f->off += r;
6077     iunlock(f->ip);
6078     return r;
6079   }
6080   panic("fileread");
6081 }
6082
6083
6084
6085
6086
6087
6088
6089
6090
6091
6092
6093
6094
6095
6096
6097
6098
6099
```



```
6100 // Write to file f.
6101 int
6102 filewrite(struct file *f, char *addr, int n)
6103 {
6104   int r;
6105
6106   if(f->writable == 0)
6107     return -1;
6108   if(f->type == FD_PIPE)
6109     return pipewrite(f->pipe, addr, n);
6110   if(f->type == FD_INODE){
6111     // write a few blocks at a time to avoid exceeding
6112     // the maximum log transaction size, including
6113     // i-node, indirect block, allocation blocks,
6114     // and 2 blocks of slop for non-aligned writes.
6115     // this really belongs lower down, since writei()
6116     // might be writing a device like the console.
6117     int max = ((LOGSIZE-1-1-2) / 2) * 512;
6118     int i = 0;
6119     while(i < n){
6120       int n1 = n - i;
6121       if(n1 > max)
6122         n1 = max;
6123
6124       begin_op();
6125       ilock(f->ip);
6126       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
6127         f->off += r;
6128       iunlock(f->ip);
6129       end_op();
6130
6131       if(r < 0)
6132         break;
6133       if(r != n1)
6134         panic("short filewrite");
6135       i += r;
6136     }
6137     return i == n ? n : -1;
6138   }
6139   panic("filewrite");
6140 }
6141
6142
6143
6144
6145
6146
6147
6148
6149
```

```
6150 //
6151 // File-system system calls.
6152 // Mostly argument checking, since we don't trust
6153 // user code, and calls into file.c and fs.c.
6154 //
6155
6156 #include "types.h"
6157 #include "defs.h"
6158 #include "param.h"
6159 #include "stat.h"
6160 #include "mmu.h"
6161 #include "proc.h"
6162 #include "fs.h"
6163 #include "spinlock.h"
6164 #include "sleeplock.h"
6165 #include "file.h"
6166 #include "fcntl.h"
6167
6168 // Fetch the nth word-sized system call argument as a file descriptor
6169 // and return both the descriptor and the corresponding struct file.
6170 static int
6171 argfd(int n, int *pfd, struct file **pf)
6172 {
6173   int fd;
6174   struct file *f;
6175
6176   if(argint(n, &fd) < 0)
6177     return -1;
6178   if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
6179     return -1;
6180   if(pfd)
6181     *pfd = fd;
6182   if(pf)
6183     *pf = f;
6184   return 0;
6185 }
6186
6187
6188
6189
6190
6191
6192
6193
6194
6195
6196
6197
6198
6199
```



```
6200 // Allocate a file descriptor for the given file.
6201 // Takes over file reference from caller on success.
6202 static int
6203 fdalloc(struct file *f)
6204 {
6205   int fd;
6206   struct proc *curproc = myproc();
6207
6208   for(fd = 0; fd < NOFILE; fd++){
6209     if(curproc->ofile[fd] == 0){
6210       curproc->ofile[fd] = f;
6211       return fd;
6212     }
6213   }
6214   return -1;
6215 }
6216
6217 int
6218 sys_dup(void)
6219 {
6220   struct file *f;
6221   int fd;
6222
6223   if(argfd(0, 0, &f) < 0)
6224     return -1;
6225   if((fd=fdalloc(f)) < 0)
6226     return -1;
6227   filedup(f);
6228   return fd;
6229 }
6230
6231 int
6232 sys_read(void)
6233 {
6234   struct file *f;
6235   int n;
6236   char *p;
6237
6238   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6239     return -1;
6240   return fileread(f, p, n);
6241 }
6242
6243
6244
6245
6246
6247
6248
6249
```

```
6250 int
6251 sys_write(void)
6252 {
6253   struct file *f;
6254   int n;
6255   char *p;
6256
6257   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6258     return -1;
6259   return filewrite(f, p, n);
6260 }
6261
6262 int
6263 sys_close(void)
6264 {
6265   int fd;
6266   struct file *f;
6267
6268   if(argfd(0, &fd, &f) < 0)
6269     return -1;
6270   myproc()->ofile[fd] = 0;
6271   fileclose(f);
6272   return 0;
6273 }
6274
6275 int
6276 sys_fstat(void)
6277 {
6278   struct file *f;
6279   struct stat *st;
6280
6281   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
6282     return -1;
6283   return filestat(f, st);
6284 }
6285
6286
6287
6288
6289
6290
6291
6292
6293
6294
6295
6296
6297
6298
6299
```



```
6300 // Create the path new as a link to the same inode as old.
6301 int
6302 sys_link(void)
6303 {
6304   char name[DIRSIZ], *new, *old;
6305   struct inode *dp, *ip;
6306
6307   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
6308     return -1;
6309
6310   begin_op();
6311   if((ip = namei(old)) == 0){
6312     end_op();
6313     return -1;
6314   }
6315
6316   ilock(ip);
6317   if(ip->type == T_DIR){
6318     iunlockput(ip);
6319     end_op();
6320     return -1;
6321   }
6322
6323   ip->nlink++;
6324   iupdate(ip);
6325   iunlock(ip);
6326
6327   if((dp = nameiparent(new, name)) == 0)
6328     goto bad;
6329   ilock(dp);
6330   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
6331     iunlockput(dp);
6332     goto bad;
6333   }
6334   iunlockput(dp);
6335   iput(ip);
6336
6337   end_op();
6338
6339   return 0;
6340
6341 bad:
6342   ilock(ip);
6343   ip->nlink--;
6344   iupdate(ip);
6345   iunlockput(ip);
6346   end_op();
6347   return -1;
6348 }
6349
```

```
// Is the directory dp empty except for "." and ".." ?
static int
isdirempty(struct inode *dp)
{
  int off;
  struct dirent de;

  for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("isdirempty: readi");
    if(de.inum != 0)
      return 0;
  }
  return 1;
}
```



```
int
sys_unlink(void)
{
  struct inode *ip, *dp;
  struct dirent de;
  char name[DIRSIZ], *path;
  uint off;

  if(argstr(0, &path) < 0)
    return -1;

  begin_op();
  if((dp = nameiparent(path, name)) == 0){
    end_op();
    return -1;
  }

  ilock(dp);

  // Cannot unlink "." or "..".
  if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
    goto bad;

  if((ip = dirlookup(dp, name, &off)) == 0)
    goto bad;
  ilock(ip);

  if(ip->nlink < 1)
    panic("unlink: nlink < 1");
  if(ip->type == T_DIR && !isdirempty(ip)){
    iunlockput(ip);
    goto bad;
  }

  memset(&de, 0, sizeof(de));
  if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
    panic("unlink: writei");
  if(ip->type == T_DIR){
    dp->nlink--;
    iupdate(dp);
  }
  iunlockput(dp);

  ip->nlink--;
  iupdate(ip);
  iunlockput(ip);

  end_op();

  return 0;
```

```
6450 bad:
6451   iunlockput(dp);
6452   end_op();
6453   return -1;
6454 }
6455
6456 static struct inode*
6457 create(char *path, short type, short major, short minor)
6458 {
6459   uint off;
6460   struct inode *ip, *dp;
6461   char name[DIRSIZ];
6462
6463   if((dp = nameiparent(path, name)) == 0)
6464     return 0;
6465   ilock(dp);
6466
6467   if((ip = dirlookup(dp, name, &off)) != 0){
6468     iunlockput(dp);
6469     ilock(ip);
6470     if(type == T_FILE && ip->type == T_FILE)
6471       return ip;
6472     iunlockput(ip);
6473     return 0;
6474   }
6475
6476   if((ip = ialloc(dp->dev, type)) == 0)
6477     panic("create: ialloc");
6478
6479   ilock(ip);
6480   ip->major = major;
6481   ip->minor = minor;
6482   ip->nlink = 1;
6483   iupdate(ip);
6484
6485   if(type == T_DIR){  // Create . and .. entries.
6486     dp->nlink++;  // for ".."
6487     iupdate(dp);
6488     // No ip->nlink++ for ".": avoid cyclic ref count.
6489     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6490       panic("create dots");
6491   }
6492
6493   if(dirlink(dp, name, ip->inum) < 0)
6494     panic("create: dirlink");
6495
6496   iunlockput(dp);
6497
6498   return ip;
6499 }
```



```
6500 int
6501 sys_open(void)
6502 {
6503   char *path;
6504   int fd, omode;
6505   struct file *f;
6506   struct inode *ip;
6507
6508   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6509     return -1;
6510
6511   begin_op();
6512
6513   if(omode & O_CREATE){
6514     ip = create(path, T_FILE, 0, 0);
6515     if(ip == 0){
6516       end_op();
6517       return -1;
6518     }
6519   } else {
6520     if((ip = namei(path)) == 0){
6521       end_op();
6522       return -1;
6523     }
6524     ilock(ip);
6525     if(ip->type == T_DIR && omode != O_RDONLY){
6526       iunlockput(ip);
6527       end_op();
6528       return -1;
6529     }
6530   }
6531
6532   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6533     if(f)
6534       fileclose(f);
6535     iunlockput(ip);
6536     end_op();
6537     return -1;
6538   }
6539   iunlock(ip);
6540   end_op();
6541
6542   f->type = FD_INODE;
6543   f->ip = ip;
6544   f->off = 0;
6545   f->readable = !(omode & O_WRONLY);
6546   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6547   return fd;
6548 }
6549
```

```
6550 int
6551 sys_mkdir(void)
6552 {
6553   char *path;
6554   struct inode *ip;
6555
6556   begin_op();
6557   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
6558     end_op();
6559     return -1;
6560   }
6561   iunlockput(ip);
6562   end_op();
6563   return 0;
6564 }
6565
6566 int
6567 sys_mknod(void)
6568 {
6569   struct inode *ip;
6570   char *path;
6571   int major, minor;
6572
6573   begin_op();
6574   if((argstr(0, &path)) < 0 ||
6575      argint(1, &major) < 0 ||
6576      argint(2, &minor) < 0 ||
6577      (ip = create(path, T_DEV, major, minor)) == 0){
6578     end_op();
6579     return -1;
6580   }
6581   iunlockput(ip);
6582   end_op();
6583   return 0;
6584 }
6585
6586
6587
6588
6589
6590
6591
6592
6593
6594
6595
6596
6597
6598
6599
```



```
6600 int
6601 sys_chdir(void)
6602 {
6603   char *path;
6604   struct inode *ip;
6605   struct proc *curproc = myproc();
6606
6607   begin_op();
6608   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
6609     end_op();
6610     return -1;
6611   }
6612   ilock(ip);
6613   if(ip->type != T_DIR){
6614     iunlockput(ip);
6615     end_op();
6616     return -1;
6617   }
6618   iunlock(ip);
6619   iput(curproc->cwd);
6620   end_op();
6621   curproc->cwd = ip;
6622   return 0;
6623 }
6624
6625 int
6626 sys_exec(void)
6627 {
6628   char *path, *argv[MAXARG];
6629   int i;
6630   uint uargv, uarg;
6631
6632   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6633     return -1;
6634   }
6635   memset(argv, 0, sizeof(argv));
6636   for(i=0;; i++){
6637     if(i >= NELEM(argv))
6638       return -1;
6639     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6640       return -1;
6641     if(uarg == 0){
6642       argv[i] = 0;
6643       break;
6644     }
6645     if(fetchstr(uarg, &argv[i]) < 0)
6646       return -1;
6647   }
6648   return exec(path, argv);
6649 }
```

```
6650 int
6651 sys_pipe(void)
6652 {
6653   int *fd;
6654   struct file *rf, *wf;
6655   int fd0, fd1;
6656
6657   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6658     return -1;
6659   if(pipealloc(&rf, &wf) < 0)
6660     return -1;
6661   fd0 = -1;
6662   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6663     if(fd0 >= 0)
6664       myproc()->ofile[fd0] = 0;
6665     fileclose(rf);
6666     fileclose(wf);
6667     return -1;
6668   }
6669   fd[0] = fd0;
6670   fd[1] = fd1;
6671   return 0;
6672 }
6673
6674
6675
6676
6677
6678
6679
6680
6681
6682
6683
6684
6685
6686
6687
6688
6689
6690
6691
6692
6693
6694
6695
6696
6697
6698
6699
```



```
6700 #include "types.h"
6701 #include "param.h"
6702 #include "memlayout.h"
6703 #include "mmu.h"
6704 #include "proc.h"
6705 #include "defs.h"
6706 #include "x86.h"
6707 #include "elf.h"
6708
6709 int
6710 exec(char *path, char **argv)
6711 {
6712   char *s, *last;
6713   int i, off;
6714   uint argc, sz, sp, ustack[3+MAXARG+1];
6715   struct elfhdr elf;
6716   struct inode *ip;
6717   struct proghdr ph;
6718   pde_t *pgdir, *oldpgdir;
6719   struct proc *curproc = myproc();
6720
6721   begin_op();
6722
6723   if((ip = namei(path)) == 0){
6724     end_op();
6725     cprintf("exec: fail\n");
6726     return -1;
6727   }
6728   ilock(ip);
6729   pgdir = 0;
6730
6731   // Check ELF header
6732   if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
6733     goto bad;
6734   if(elf.magic != ELF_MAGIC)
6735     goto bad;
6736
6737   if((pgdir = setupkvm()) == 0)
6738     goto bad;
6739
6740   // Load program into memory.
6741   sz = 0;
6742   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6743     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6744       goto bad;
6745     if(ph.type != ELF_PROG_LOAD)
6746       continue;
6747     if(ph.memsz < ph.filesz)
6748       goto bad;
6749     if(ph.vaddr + ph.memsz < ph.vaddr)
```

```
6750      goto bad;
6751    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6752      goto bad;
6753    if(ph.vaddr % PGSIZE != 0)
6754      goto bad;
6755    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6756      goto bad;
6757  }
6758  iunlockput(ip);
6759  end_op();
6760  ip = 0;
6761
6762  // Allocate two pages at the next page boundary.
6763  // Make the first inaccessible.  Use the second as the user stack.
6764  sz = PGROUNDUP(sz);
6765  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
6766    goto bad;
6767  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6768  sp = sz;
6769
6770  // Push argument strings, prepare rest of stack in ustack.
6771  for(argc = 0; argv[argc]; argc++) {
6772    if(argc >= MAXARG)
6773      goto bad;
6774    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6775    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6776      goto bad;
6777    ustack[3+argc] = sp;
6778  }
6779  ustack[3+argc] = 0;
6780
6781  ustack[0] = 0xffffffff;  // fake return PC
6782  ustack[1] = argc;
6783  ustack[2] = sp - (argc+1)*4;  // argv pointer
6784
6785  sp -= (3+argc+1) * 4;
6786  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6787    goto bad;
6788
6789  // Save program name for debugging.
6790  for(last=s=path; *s; s++)
6791    if(*s == '/')
6792      last = s+1;
6793  safestrcpy(curproc->name, last, sizeof(curproc->name));
6794
6795  // Commit to the user image.
6796  oldpgdir = curproc->pgdir;
6797  curproc->pgdir = pgdir;
6798  curproc->sz = sz;
6799  curproc->tf->eip = elf.entry;  // main
```



```
6800   curproc->tf->esp = sp;
6801   switchuvm(curproc);
6802   freevm(oldpgdir);
6803   return 0;
6804
6805  bad:
6806   if(pgdir)
6807     freevm(pgdir);
6808   if(ip){
6809     iunlockput(ip);
6810     end_op();
6811   }
6812   return -1;
6813 }
6814
6815
6816
6817
6818
6819
6820
6821
6822
6823
6824
6825
6826
6827
6828
6829
6830
6831
6832
6833
6834
6835
6836
6837
6838
6839
6840
6841
6842
6843
6844
6845
6846
6847
6848
6849
```

```
6850 #include "types.h"
6851 #include "defs.h"
6852 #include "param.h"
6853 #include "mmu.h"
6854 #include "proc.h"
6855 #include "fs.h"
6856 #include "spinlock.h"
6857 #include "sleeplock.h"
6858 #include "file.h"
6859
6860 #define PIPESIZE 512
6861
6862 struct pipe {
6863   struct spinlock lock;
6864   char data[PIPESIZE];
6865   uint nread;     // number of bytes read
6866   uint nwrite;    // number of bytes written
6867   int readopen;   // read fd is still open
6868   int writeopen;  // write fd is still open
6869 };
6870
6871 int
6872 pipealloc(struct file **f0, struct file **f1)
6873 {
6874   struct pipe *p;
6875
6876   p = 0;
6877   *f0 = *f1 = 0;
6878   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
6879     goto bad;
6880   if((p = (struct pipe*)kalloc()) == 0)
6881     goto bad;
6882   p->readopen = 1;
6883   p->writeopen = 1;
6884   p->nwrite = 0;
6885   p->nread = 0;
6886   initlock(&p->lock, "pipe");
6887   (*f0)->type = FD_PIPE;
6888   (*f0)->readable = 1;
6889   (*f0)->writable = 0;
6890   (*f0)->pipe = p;
6891   (*f1)->type = FD_PIPE;
6892   (*f1)->readable = 0;
6893   (*f1)->writable = 1;
6894   (*f1)->pipe = p;
6895   return 0;
6896
6897
6898
6899
```



```
6900  bad:
6901   if(p)
6902     kfree((char*)p);
6903   if(*f0)
6904     fileclose(*f0);
6905   if(*f1)
6906     fileclose(*f1);
6907   return -1;
6908 }
6909
6910 void
6911 pipeclose(struct pipe *p, int writable)
6912 {
6913   acquire(&p->lock);
6914   if(writable){
6915     p->writeopen = 0;
6916     wakeup(&p->nread);
6917   } else {
6918     p->readopen = 0;
6919     wakeup(&p->nwrite);
6920   }
6921   if(p->readopen == 0 && p->writeopen == 0){
6922     release(&p->lock);
6923     kfree((char*)p);
6924   } else
6925     release(&p->lock);
6926 }
6927
6928
6929 int
6930 pipewrite(struct pipe *p, char *addr, int n)
6931 {
6932   int i;
6933
6934   acquire(&p->lock);
6935   for(i = 0; i < n; i++){
6936     while(p->nwrite == p->nread + PIPESIZE){
6937       if(p->readopen == 0 || myproc()->killed){
6938         release(&p->lock);
6939         return -1;
6940       }
6941       wakeup(&p->nread);
6942       sleep(&p->nwrite, &p->lock);
6943     }
6944     p->data[p->nwrite++ % PIPESIZE] = addr[i];
6945   }
6946   wakeup(&p->nread);
6947   release(&p->lock);
6948   return n;
6949 }
```

```
6950 int
6951 piperead(struct pipe *p, char *addr, int n)
6952 {
6953   int i;
6954
6955   acquire(&p->lock);
6956   while(p->nread == p->nwrite && p->writeopen){
6957     if(myproc()->killed){
6958       release(&p->lock);
6959       return -1;
6960     }
6961     sleep(&p->nread, &p->lock);
6962   }
6963   for(i = 0; i < n; i++){
6964     if(p->nread == p->nwrite)
6965       break;
6966     addr[i] = p->data[p->nread++ % PIPESIZE];
6967   }
6968   wakeup(&p->nwrite);
6969   release(&p->lock);
6970   return i;
6971 }
6972
6973
6974
6975
6976
6977
6978
6979
6980
6981
6982
6983
6984
6985
6986
6987
6988
6989
6990
6991
6992
6993
6994
6995
6996
6997
6998
6999
```



```
7000 #include "types.h"
7001 #include "x86.h"
7002
7003 void*
7004 memset(void *dst, int c, uint n)
7005 {
7006   if ((int)dst%4 == 0 && n%4 == 0){
7007     c &= 0xFF;
7008     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
7009   } else
7010     stosb(dst, c, n);
7011   return dst;
7012 }
7013
7014 int
7015 memcmp(const void *v1, const void *v2, uint n)
7016 {
7017   const uchar *s1, *s2;
7018
7019   s1 = v1;
7020   s2 = v2;
7021   while(n-- > 0){
7022     if(*s1 != *s2)
7023       return *s1 - *s2;
7024     s1++, s2++;
7025   }
7026
7027   return 0;
7028 }
7029
7030 void*
7031 memmove(void *dst, const void *src, uint n)
7032 {
7033   const char *s;
7034   char *d;
7035
7036   s = src;
7037   d = dst;
7038   if(s < d && s + n > d){
7039     s += n;
7040     d += n;
7041     while(n-- > 0)
7042       *--d = *--s;
7043   } else
7044     while(n-- > 0)
7045       *d++ = *s++;
7046
7047   return dst;
7048 }
7049
```

```
7050 // memcpy exists to placate GCC.  Use memmove.
7051 void*
7052 memcpy(void *dst, const void *src, uint n)
7053 {
7054   return memmove(dst, src, n);
7055 }
7056
7057 int
7058 strncmp(const char *p, const char *q, uint n)
7059 {
7060   while(n > 0 && *p && *p == *q)
7061     n--, p++, q++;
7062   if(n == 0)
7063     return 0;
7064   return (uchar)*p - (uchar)*q;
7065 }
7066
7067 char*
7068 strncpy(char *s, const char *t, int n)
7069 {
7070   char *os;
7071
7072   os = s;
7073   while(n-- > 0 && (*s++ = *t++) != 0)
7074     ;
7075   while(n-- > 0)
7076     *s++ = 0;
7077   return os;
7078 }
7079
7080 // Like strncpy but guaranteed to NUL-terminate.
7081 char*
7082 safestrcpy(char *s, const char *t, int n)
7083 {
7084   char *os;
7085
7086   os = s;
7087   if(n <= 0)
7088     return os;
7089   while(--n > 0 && (*s++ = *t++) != 0)
7090     ;
7091   *s = 0;
7092   return os;
7093 }
7094
7095
7096
7097
7098
7099
```



```
7100 int
7101 strlen(const char *s)
7102 {
7103   int n;
7104
7105   for(n = 0; s[n]; n++)
7106     ;
7107   return n;
7108 }
7109
7110
7111
7112
7113
7114
7115
7116
7117
7118
7119
7120
7121
7122
7123
7124
7125
7126
7127
7128
7129
7130
7131
7132
7133
7134
7135
7136
7137
7138
7139
7140
7141
7142
7143
7144
7145
7146
7147
7148
7149
```

```
// 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
```

```
7250 // Blank page.
7251
7252
7253
7254
7255
7256
7257
7258
7259
7260
7261
7262
7263
7264
7265
7266
7267
7268
7269
7270
7271
7272
7273
7274
7275
7276
7277
7278
7279
7280
7281
7282
7283
7284
7285
7286
7287
7288
7289
7290
7291
7292
7293
7294
7295
7296
7297
7298
7299
```



```
7300 // Multiprocessor support
7301 // Search memory for MP description structures.
7302 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
7303
7304 #include "types.h"
7305 #include "defs.h"
7306 #include "param.h"
7307 #include "memlayout.h"
7308 #include "mp.h"
7309 #include "x86.h"
7310 #include "mmu.h"
7311 #include "proc.h"
7312
7313 struct cpu cpus[NCPU];
7314 int ncpu;
7315 uchar ioapicid;
7316
7317 static uchar
7318 sum(uchar *addr, int len)
7319 {
7320   int i, sum;
7321
7322   sum = 0;
7323   for(i=0; i<len; i++)
7324     sum += addr[i];
7325   return sum;
7326 }
7327
7328 // Look for an MP structure in the len bytes at addr.
7329 static struct mp*
7330 mpsearch1(uint a, int len)
7331 {
7332   uchar *e, *p, *addr;
7333
7334   addr = P2V(a);
7335   e = addr+len;
7336   for(p = addr; p < e; p += sizeof(struct mp))
7337     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
7338       return (struct mp*)p;
7339   return 0;
7340 }
7341
7342
7343
7344
7345
7346
7347
7348
7349
```

```
7350 // Search for the MP Floating Pointer Structure, which according to the
7351 // spec is in one of the following three locations:
7352 // 1) in the first KB of the EBDA;
7353 // 2) in the last KB of system base memory;
7354 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
7355 static struct mp*
7356 mpsearch(void)
7357 {
7358   uchar *bda;
7359   uint p;
7360   struct mp *mp;
7361
7362   bda = (uchar *) P2V(0x400);
7363   if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
7364     if((mp = mpsearch1(p, 1024)))
7365       return mp;
7366   } else {
7367     p = ((bda[0x14]<<8)|bda[0x13])*1024;
7368     if((mp = mpsearch1(p-1024, 1024)))
7369       return mp;
7370   }
7371   return mpsearch1(0xF0000, 0x10000);
7372 }
7373
7374 // Search for an MP configuration table.  For now,
7375 // don't accept the default configurations (physaddr == 0).
7376 // Check for correct signature, calculate the checksum and,
7377 // if correct, check the version.
7378 // To do: check extended table checksum.
7379 static struct mpconf*
7380 mpconfig(struct mp **pmp)
7381 {
7382   struct mpconf *conf;
7383   struct mp *mp;
7384
7385   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
7386     return 0;
7387   conf = (struct mpconf*) P2V((uint) mp->physaddr);
7388   if(memcmp(conf, "PCMP", 4) != 0)
7389     return 0;
7390   if(conf->version != 1 && conf->version != 4)
7391     return 0;
7392   if(sum((uchar*)conf, conf->length) != 0)
7393     return 0;
7394   *pmp = mp;
7395   return conf;
7396 }
7397
7398
7399
```



```
7400 void
7401 mpinit(void)
7402 {
7403   uchar *p, *e;
7404   int ismp;
7405   struct mp *mp;
7406   struct mpconf *conf;
7407   struct mpproc *proc;
7408   struct mpioapic *ioapic;
7409
7410   if((conf = mpconfig(&mp)) == 0)
7411     panic("Expect to run on an SMP");
7412   ismp = 1;
7413   lapic = (uint*)conf->lapicaddr;
7414   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
7415     switch(*p){
7416     case MPPROC:
7417       proc = (struct mpproc*)p;
7418       if(ncpu < NCPU) {
7419         cpus[ncpu].apicid = proc->apicid;  // apicid may differ from ncpu
7420         ncpu++;
7421       }
7422       p += sizeof(struct mpproc);
7423       continue;
7424     case MPIOAPIC:
7425       ioapic = (struct mpioapic*)p;
7426       ioapicid = ioapic->apicno;
7427       p += sizeof(struct mpioapic);
7428       continue;
7429     case MPBUS:
7430     case MPIOINTR:
7431     case MPLINTR:
7432       p += 8;
7433       continue;
7434     default:
7435       ismp = 0;
7436       break;
7437     }
7438   }
7439   if(!ismp)
7440     panic("Didn't find a suitable machine");
7441
7442   if(mp->imcrp){
7443     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
7444     // But it would on real hardware.
7445     outb(0x22, 0x70);   // Select IMCR
7446     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
7447   }
7448 }
7449
```

```
7450 // The local APIC manages internal (non-I/O) interrupts.
7451 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
7452
7453 #include "param.h"
7454 #include "types.h"
7455 #include "defs.h"
7456 #include "date.h"
7457 #include "memlayout.h"
7458 #include "traps.h"
7459 #include "mmu.h"
7460 #include "x86.h"
7461
7462 // Local APIC registers, divided by 4 for use as uint[] indices.
7463 #define ID      (0x0020/4)   // ID
7464 #define VER     (0x0030/4)   // Version
7465 #define TPR     (0x0080/4)   // Task Priority
7466 #define EOI     (0x00B0/4)   // EOI
7467 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
7468   #define ENABLE     0x00000100   // Unit Enable
7469 #define ESR     (0x0280/4)   // Error Status
7470 #define ICRLO   (0x0300/4)   // Interrupt Command
7471   #define INIT       0x00000500   // INIT/RESET
7472   #define STARTUP    0x00000600   // Startup IPI
7473   #define DELIVS     0x00001000   // Delivery status
7474   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
7475   #define DEASSERT   0x00000000
7476   #define LEVEL      0x00008000   // Level triggered
7477   #define BCAST      0x00080000   // Send to all APICs, including self.
7478   #define BUSY       0x00001000
7479   #define FIXED      0x00000000
7480 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
7481 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
7482   #define X1         0x0000000B   // divide counts by 1
7483   #define PERIODIC   0x00020000   // Periodic
7484 #define PCINT   (0x0340/4)   // Performance Counter LVT
7485 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
7486 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
7487 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
7488   #define MASKED     0x00010000   // Interrupt masked
7489 #define TICR    (0x0380/4)   // Timer Initial Count
7490 #define TCCR    (0x0390/4)   // Timer Current Count
7491 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
7492
7493 volatile uint *lapic;  // Initialized in mp.c
7494
7495 static void
7496 lapicw(int index, int value)
7497 {
7498   lapic[index] = value;
7499   lapic[ID];  // wait for write to finish, by reading
```



```
7500 }
7501
7502
7503
7504
7505
7506
7507
7508
7509
7510
7511
7512
7513
7514
7515
7516
7517
7518
7519
7520
7521
7522
7523
7524
7525
7526
7527
7528
7529
7530
7531
7532
7533
7534
7535
7536
7537
7538
7539
7540
7541
7542
7543
7544
7545
7546
7547
7548
7549
```

```
7550 void
7551 lapicinit(void)
7552 {
7553   if(!lapic)
7554     return;
7555
7556   // Enable local APIC; set spurious interrupt vector.
7557   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
7558
7559   // The timer repeatedly counts down at bus frequency
7560   // from lapic[TICR] and then issues an interrupt.
7561   // If xv6 cared more about precise timekeeping,
7562   // TICR would be calibrated using an external time source.
7563   lapicw(TDCR, X1);
7564   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
7565   lapicw(TICR, 10000000);
7566
7567   // Disable logical interrupt lines.
7568   lapicw(LINT0, MASKED);
7569   lapicw(LINT1, MASKED);
7570
7571   // Disable performance counter overflow interrupts
7572   // on machines that provide that interrupt entry.
7573   if(((lapic[VER]>>16) & 0xFF) >= 4)
7574     lapicw(PCINT, MASKED);
7575
7576   // Map error interrupt to IRQ_ERROR.
7577   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
7578
7579   // Clear error status register (requires back-to-back writes).
7580   lapicw(ESR, 0);
7581   lapicw(ESR, 0);
7582
7583   // Ack any outstanding interrupts.
7584   lapicw(EOI, 0);
7585
7586   // Send an Init Level De-Assert to synchronise arbitration ID's.
7587   lapicw(ICRHI, 0);
7588   lapicw(ICRLO, BCAST | INIT | LEVEL);
7589   while(lapic[ICRLO] & DELIVS)
7590     ;
7591
7592   // Enable interrupts on the APIC (but not on the processor).
7593   lapicw(TPR, 0);
7594 }
7595
7596
7597
7598
7599
```



```
7600 int
7601 lapicid(void)
7602 {
7603   if (!lapic)
7604     return 0;
7605   return lapic[ID] >> 24;
7606 }
7607
7608 // Acknowledge interrupt.
7609 void
7610 lapiceoi(void)
7611 {
7612   if(lapic)
7613     lapicw(EOI, 0);
7614 }
7615
7616 // Spin for a given number of microseconds.
7617 // On real hardware would want to tune this dynamically.
7618 void
7619 microdelay(int us)
7620 {
7621 }
7622
7623 #define CMOS_PORT    0x70
7624 #define CMOS_RETURN  0x71
7625
7626 // Start additional processor running entry code at addr.
7627 // See Appendix B of MultiProcessor Specification.
7628 void
7629 lapicstartap(uchar apicid, uint addr)
7630 {
7631   int i;
7632   ushort *wrv;
7633
7634   // "The BSP must initialize CMOS shutdown code to 0AH
7635   // and the warm reset vector (DWORD based at 40:67) to point at
7636   // the AP startup code prior to the [universal startup algorithm]."
7637   outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
7638   outb(CMOS_PORT+1, 0x0A);
7639   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
7640   wrv[0] = 0;
7641   wrv[1] = addr >> 4;
7642
7643   // "Universal startup algorithm."
7644   // Send INIT (level-triggered) interrupt to reset other CPU.
7645   lapicw(ICRHI, apicid<<24);
7646   lapicw(ICRLO, INIT | LEVEL | ASSERT);
7647   microdelay(200);
7648   lapicw(ICRLO, INIT | LEVEL);
7649   microdelay(100);    // should be 10ms, but too slow in Bochs!
```

```
7650   // Send startup IPI (twice!) to enter code.
7651   // Regular hardware is supposed to only accept a STARTUP
7652   // when it is in the halted state due to an INIT.  So the second
7653   // should be ignored, but it is part of the official Intel algorithm.
7654   // Bochs complains about the second one.  Too bad for Bochs.
7655   for(i = 0; i < 2; i++){
7656     lapicw(ICRHI, apicid<<24);
7657     lapicw(ICRLO, STARTUP | (addr>>12));
7658     microdelay(200);
7659   }
7660 }
7661
7662 #define CMOS_STATA   0x0a
7663 #define CMOS_STATB   0x0b
7664 #define CMOS_UIP    (1 << 7)        // RTC update in progress
7665
7666 #define SECS    0x00
7667 #define MINS    0x02
7668 #define HOURS   0x04
7669 #define DAY     0x07
7670 #define MONTH   0x08
7671 #define YEAR    0x09
7672
7673 static uint cmos_read(uint reg)
7674 {
7675   outb(CMOS_PORT,  reg);
7676   microdelay(200);
7677
7678   return inb(CMOS_RETURN);
7679 }
7680
7681 static void fill_rtcdate(struct rtcdate *r)
7682 {
7683   r->second = cmos_read(SECS);
7684   r->minute = cmos_read(MINS);
7685   r->hour   = cmos_read(HOURS);
7686   r->day    = cmos_read(DAY);
7687   r->month  = cmos_read(MONTH);
7688   r->year   = cmos_read(YEAR);
7689 }
7690
7691
7692
7693
7694
7695
7696
7697
7698
7699
```



```
7700 // qemu seems to use 24-hour GWT and the values are BCD encoded
7701 void cmostime(struct rtcdate *r)
7702 {
7703   struct rtcdate t1, t2;
7704   int sb, bcd;
7705
7706   sb = cmos_read(CMOS_STATB);
7707
7708   bcd = (sb & (1 << 2)) == 0;
7709
7710   // make sure CMOS doesn't modify time while we read it
7711   for(;;) {
7712     fill_rtcdate(&t1);
7713     if(cmos_read(CMOS_STATA) & CMOS_UIP)
7714         continue;
7715     fill_rtcdate(&t2);
7716     if(memcmp(&t1, &t2, sizeof(t1)) == 0)
7717       break;
7718   }
7719
7720   // convert
7721   if(bcd) {
7722 #define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
7723     CONV(second);
7724     CONV(minute);
7725     CONV(hour  );
7726     CONV(day   );
7727     CONV(month );
7728     CONV(year  );
7729 #undef     CONV
7730   }
7731
7732   *r = t1;
7733   r->year += 2000;
7734 }
7735
7736
7737
7738
7739
7740
7741
7742
7743
7744
7745
7746
7747
7748
7749
```

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

```
7850 // PC keyboard interface constants
7851
7852 #define KBSTATP         0x64    // kbd controller status port(I)
7853 #define KBS_DIB         0x01    // kbd data in buffer
7854 #define KBDATAP         0x60    // kbd data port(I)
7855
7856 #define NO              0
7857
7858 #define SHIFT           (1<<0)
7859 #define CTL             (1<<1)
7860 #define ALT             (1<<2)
7861
7862 #define CAPSLOCK        (1<<3)
7863 #define NUMLOCK         (1<<4)
7864 #define SCROLLLOCK      (1<<5)
7865
7866 #define E0ESC           (1<<6)
7867
7868 // Special keycodes
7869 #define KEY_HOME        0xE0
7870 #define KEY_END         0xE1
7871 #define KEY_UP          0xE2
7872 #define KEY_DN          0xE3
7873 #define KEY_LF          0xE4
7874 #define KEY_RT          0xE5
7875 #define KEY_PGUP        0xE6
7876 #define KEY_PGDN        0xE7
7877 #define KEY_INS         0xE8
7878 #define KEY_DEL         0xE9
7879
7880 // C('A') == Control-A
7881 #define C(x) (x - '@')
7882
7883 static uchar shiftcode[256] =
7884 {
7885   [0x1D] CTL,
7886   [0x2A] SHIFT,
7887   [0x36] SHIFT,
7888   [0x38] ALT,
7889   [0x9D] CTL,
7890   [0xB8] ALT
7891 };
7892
7893 static uchar togglecode[256] =
7894 {
7895   [0x3A] CAPSLOCK,
7896   [0x45] NUMLOCK,
7897   [0x46] SCROLLLOCK
7898 };
7899
```



```
7900 static uchar normalmap[256] =
7901 {
7902   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
7903   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
7904   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
7905   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
7906   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
7907   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
7908   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
7909   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7910   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7911   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7912   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7913   [0x9C] '\n',      // KP_Enter
7914   [0xB5] '/',       // KP_Div
7915   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7916   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7917   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7918   [0x97] KEY_HOME,  [0xCF] KEY_END,
7919   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7920 };
7921
7922 static uchar shiftmap[256] =
7923 {
7924   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
7925   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
7926   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
7927   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
7928   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
7929   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
7930   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
7931   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7932   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7933   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7934   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7935   [0x9C] '\n',      // KP_Enter
7936   [0xB5] '/',       // KP_Div
7937   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7938   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7939   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7940   [0x97] KEY_HOME,  [0xCF] KEY_END,
7941   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7942 };
7943
7944
7945
7946
7947
7948
7949
```

```
7950 static uchar ctlmap[256] =
7951 {
7952   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7953   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7954   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
7955   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
7956   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
7957   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
7958   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
7959   [0x9C] '\r',      // KP_Enter
7960   [0xB5] C('/'),    // KP_Div
7961   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7962   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7963   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7964   [0x97] KEY_HOME,  [0xCF] KEY_END,
7965   [0xD2] KEY_INS,   [0xD3] KEY_DEL
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 #include "types.h"
8001 #include "x86.h"
8002 #include "defs.h"
8003 #include "kbd.h"
8004
8005 int
8006 kbdgetc(void)
8007 {
8008   static uint shift;
8009   static uchar *charcode[4] = {
8010     normalmap, shiftmap, ctlmap, ctlmap
8011   };
8012   uint st, data, c;
8013
8014   st = inb(KBSTATP);
8015   if((st & KBS_DIB) == 0)
8016     return -1;
8017   data = inb(KBDATAP);
8018
8019   if(data == 0xE0){
8020     shift |= E0ESC;
8021     return 0;
8022   } else if(data & 0x80){
8023     // Key released
8024     data = (shift & E0ESC ? data : data & 0x7F);
8025     shift &= ~(shiftcode[data] | E0ESC);
8026     return 0;
8027   } else if(shift & E0ESC){
8028     // Last character was an E0 escape; or with 0x80
8029     data |= 0x80;
8030     shift &= ~E0ESC;
8031   }
8032
8033   shift |= shiftcode[data];
8034   shift ^= togglecode[data];
8035   c = charcode[shift & (CTL | SHIFT)][data];
8036   if(shift & CAPSLOCK){
8037     if('a' <= c && c <= 'z')
8038       c += 'A' - 'a';
8039     else if('A' <= c && c <= 'Z')
8040       c += 'a' - 'A';
8041   }
8042   return c;
8043 }
8044
8045 void
8046 kbdintr(void)
8047 {
8048   consoleintr(kbdgetc);
8049 }
```

```
8050 // Console input and output.
8051 // Input is from the keyboard or serial port.
8052 // Output is written to the screen and serial port.
8053
8054 #include "types.h"
8055 #include "defs.h"
8056 #include "param.h"
8057 #include "traps.h"
8058 #include "spinlock.h"
8059 #include "sleeplock.h"
8060 #include "fs.h"
8061 #include "file.h"
8062 #include "memlayout.h"
8063 #include "mmu.h"
8064 #include "proc.h"
8065 #include "x86.h"
8066
8067 static void consputc(int);
8068
8069 static int panicked = 0;
8070
8071 static struct {
8072   struct spinlock lock;
8073   int locking;
8074 } cons;
8075
8076 static void
8077 printint(int xx, int base, int sign)
8078 {
8079   static char digits[] = "0123456789abcdef";
8080   char buf[16];
8081   int i;
8082   uint x;
8083
8084   if(sign && (sign = xx < 0))
8085     x = -xx;
8086   else
8087     x = xx;
8088
8089   i = 0;
8090   do{
8091     buf[i++] = digits[x % base];
8092   }while((x /= base) != 0);
8093
8094   if(sign)
8095     buf[i++] = '-';
8096
8097   while(--i >= 0)
8098     consputc(buf[i]);
8099 }
```



```
8100
8101
8102
8103
8104
8105
8106
8107
8108
8109
8110
8111
8112
8113
8114
8115
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 // Print to the console. only understands %d, %x, %p, %s.
8151 void
8152 cprintf(char *fmt, ...)
8153 {
8154   int i, c, locking;
8155   uint *argp;
8156   char *s;
8157
8158   locking = cons.locking;
8159   if(locking)
8160     acquire(&cons.lock);
8161
8162   if (fmt == 0)
8163     panic("null fmt");
8164
8165   argp = (uint*)(void*)(&fmt + 1);
8166   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
8167     if(c != '%'){
8168       consputc(c);
8169       continue;
8170     }
8171     c = fmt[++i] & 0xff;
8172     if(c == 0)
8173       break;
8174     switch(c){
8175     case 'd':
8176       printint(*argp++, 10, 1);
8177       break;
8178     case 'x':
8179     case 'p':
8180       printint(*argp++, 16, 0);
8181       break;
8182     case 's':
8183       if((s = (char*)*argp++) == 0)
8184         s = "(null)";
8185       for(; *s; s++)
8186         consputc(*s);
8187       break;
8188     case '%':
8189       consputc('%');
8190       break;
8191     default:
8192       // Print unknown % sequence to draw attention.
8193       consputc('%');
8194       consputc(c);
8195       break;
8196     }
8197   }
8198
8199
```



```
8200   if(locking)
8201     release(&cons.lock);
8202 }
8203
8204 void
8205 panic(char *s)
8206 {
8207   int i;
8208   uint pcs[10];
8209
8210   cli();
8211   cons.locking = 0;
8212   // use lapiccpunum so that we can call panic from mycpu()
8213   cprintf("lapicid %d: panic: ", lapicid());
8214   cprintf(s);
8215   cprintf("\n");
8216   getcallerpcs(&s, pcs);
8217   for(i=0; i<10; i++)
8218     cprintf(" %p", pcs[i]);
8219   panicked = 1; // freeze other CPU
8220   for(;;)
8221     ;
8222 }
8223
8224
8225
8226
8227
8228
8229
8230
8231
8232
8233
8234
8235
8236
8237
8238
8239
8240
8241
8242
8243
8244
8245
8246
8247
8248
8249
```

```
#define BACKSPACE 0x100
#define CRTPORT 0x3d4
static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory

static void
cgaputc(int c)
{
  int pos;

  // Cursor position: col + 80*row.
  outb(CRTPORT, 14);
  pos = inb(CRTPORT+1) << 8;
  outb(CRTPORT, 15);
  pos |= inb(CRTPORT+1);

  if(c == '\n')
    pos += 80 - pos%80;
  else if(c == BACKSPACE){
    if(pos > 0) --pos;
  } else
    crt[pos++] = (c&0xff) | 0x0700;  // black on white

  if(pos < 0 || pos > 25*80)
    panic("pos under/overflow");

  if((pos/80) >= 24){  // Scroll up.
    memmove(crt, crt+80, sizeof(crt[0])*23*80);
    pos -= 80;
    memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
  }

  outb(CRTPORT, 14);
  outb(CRTPORT+1, pos>>8);
  outb(CRTPORT, 15);
  outb(CRTPORT+1, pos);
  crt[pos] = ' ' | 0x0700;
}
```



```
void
consputc(int c)
{
  if(panicked){
    cli();
    for(;;)
      ;
  }

  if(c == BACKSPACE){
    uartputc('\b'); uartputc(' '); uartputc('\b');
  } else
    uartputc(c);
  cgaputc(c);
}

#define INPUT_BUF 128
struct {
  char buf[INPUT_BUF];
  uint r;  // Read index
  uint w;  // Write index
  uint e;  // Edit index
} input;

#define C(x)  ((x)-'@')  // Control-x

void
consoleintr(int (*getc)(void))
{
  int c, doprocdump = 0;

  acquire(&cons.lock);
  while((c = getc()) >= 0){
    switch(c){
    case C('P'):  // Process listing.
      // procdump() locks cons.lock indirectly; invoke later
      doprocdump = 1;
      break;
    case C('U'):  // Kill line.
      while(input.e != input.w &&
            input.buf[(input.e-1) % INPUT_BUF] != '\n'){
        input.e--;
        consputc(BACKSPACE);
      }
      break;
    case C('H'): case '\x7f':  // Backspace
      if(input.e != input.w){
        input.e--;
        consputc(BACKSPACE);
      }
```

```
8350       break;
8351     default:
8352       if(c != 0 && input.e-input.r < INPUT_BUF){
8353         c = (c == '\r') ? '\n' : c;
8354         input.buf[input.e++ % INPUT_BUF] = c;
8355         consputc(c);
8356         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
8357           input.w = input.e;
8358           wakeup(&input.r);
8359         }
8360       }
8361       break;
8362     }
8363   }
8364   release(&cons.lock);
8365   if(doprocdump) {
8366     procdump();  // now call procdump() wo. cons.lock held
8367   }
8368 }
8369
8370 int
8371 consoleread(struct inode *ip, char *dst, int n)
8372 {
8373   uint target;
8374   int c;
8375
8376   iunlock(ip);
8377   target = n;
8378   acquire(&cons.lock);
8379   while(n > 0){
8380     while(input.r == input.w){
8381       if(myproc()->killed){
8382         release(&cons.lock);
8383         ilock(ip);
8384         return -1;
8385       }
8386       sleep(&input.r, &cons.lock);
8387     }
8388     c = input.buf[input.r++ % INPUT_BUF];
8389     if(c == C('D')){  // EOF
8390       if(n < target){
8391         // Save ^D for next time, to make sure
8392         // caller gets a 0-byte result.
8393         input.r--;
8394       }
8395       break;
8396     }
8397     *dst++ = c;
8398     --n;
8399     if(c == '\n')
```



```
8400       break;
8401   }
8402   release(&cons.lock);
8403   ilock(ip);
8404
8405   return target - n;
8406 }
8407
8408 int
8409 consolewrite(struct inode *ip, char *buf, int n)
8410 {
8411   int i;
8412
8413   iunlock(ip);
8414   acquire(&cons.lock);
8415   for(i = 0; i < n; i++)
8416     consputc(buf[i] & 0xff);
8417   release(&cons.lock);
8418   ilock(ip);
8419
8420   return n;
8421 }
8422
8423 void
8424 consoleinit(void)
8425 {
8426   initlock(&cons.lock, "console");
8427
8428   devsw[CONSOLE].write = consolewrite;
8429   devsw[CONSOLE].read = consoleread;
8430   cons.locking = 1;
8431
8432   ioapicenable(IRQ_KBD, 0);
8433 }
8434
8435
8436
8437
8438
8439
8440
8441
8442
8443
8444
8445
8446
8447
8448
8449
```

```
8450 // Intel 8250 serial port (UART).
8451
8452 #include "types.h"
8453 #include "defs.h"
8454 #include "param.h"
8455 #include "traps.h"
8456 #include "spinlock.h"
8457 #include "sleeplock.h"
8458 #include "fs.h"
8459 #include "file.h"
8460 #include "mmu.h"
8461 #include "proc.h"
8462 #include "x86.h"
8463
8464 #define COM1    0x3f8
8465
8466 static int uart;    // is there a uart?
8467
8468 void
8469 uartinit(void)
8470 {
8471   char *p;
8472
8473   // Turn off the FIFO
8474   outb(COM1+2, 0);
8475
8476   // 9600 baud, 8 data bits, 1 stop bit, parity off.
8477   outb(COM1+3, 0x80);    // Unlock divisor
8478   outb(COM1+0, 115200/9600);
8479   outb(COM1+1, 0);
8480   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
8481   outb(COM1+4, 0);
8482   outb(COM1+1, 0x01);    // Enable receive interrupts.
8483
8484   // If status is 0xFF, no serial port.
8485   if(inb(COM1+5) == 0xFF)
8486     return;
8487   uart = 1;
8488
8489   // Acknowledge pre-existing interrupt conditions;
8490   // enable interrupts.
8491   inb(COM1+2);
8492   inb(COM1+0);
8493   ioapicenable(IRQ_COM1, 0);
8494
8495   // Announce that we're here.
8496   for(p="xv6...\n"; *p; p++)
8497     uartputc(*p);
8498 }
8499
```



```
8500 void
8501 uartputc(int c)
8502 {
8503   int i;
8504
8505   if(!uart)
8506     return;
8507   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
8508     microdelay(10);
8509   outb(COM1+0, c);
8510 }
8511
8512 static int
8513 uartgetc(void)
8514 {
8515   if(!uart)
8516     return -1;
8517   if(!(inb(COM1+5) & 0x01))
8518     return -1;
8519   return inb(COM1+0);
8520 }
8521
8522 void
8523 uartintr(void)
8524 {
8525   consoleintr(uartgetc);
8526 }
8527
8528
8529
8530
8531
8532
8533
8534
8535
8536
8537
8538
8539
8540
8541
8542
8543
8544
8545
8546
8547
8548
8549
```

```
# Initial process execs /init.
# This code runs in user space.

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


# exec(init, argv)
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL

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

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

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



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

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
```

```
8650 // init: The initial user-level program
8651
8652 #include "types.h"
8653 #include "stat.h"
8654 #include "user.h"
8655 #include "fcntl.h"
8656
8657 char *argv[] = { "sh", 0 };
8658
8659 int
8660 main(void)
8661 {
8662   int pid, wpid;
8663
8664   if(open("console", O_RDWR) < 0){
8665     mknod("console", 1, 1);
8666     open("console", O_RDWR);
8667   }
8668   dup(0);  // stdout
8669   dup(0);  // stderr
8670
8671   for(;;){
8672     printf(1, "init: starting sh\n");
8673     pid = fork();
8674     if(pid < 0){
8675       printf(1, "init: fork failed\n");
8676       exit();
8677     }
8678     if(pid == 0){
8679       exec("sh", argv);
8680       printf(1, "init: exec sh failed\n");
8681       exit();
8682     }
8683     while((wpid=wait()) >= 0 && wpid != pid)
8684       printf(1, "zombie!\n");
8685   }
8686 }
8687
8688
8689
8690
8691
8692
8693
8694
8695
8696
8697
8698
8699
```



```
8700 // Shell.
8701
8702 #include "types.h"
8703 #include "user.h"
8704 #include "fcntl.h"
8705
8706 // Parsed command representation
8707 #define EXEC  1
8708 #define REDIR 2
8709 #define PIPE  3
8710 #define LIST  4
8711 #define BACK  5
8712
8713 #define MAXARGS 10
8714
8715 struct cmd {
8716   int type;
8717 };
8718
8719 struct execcmd {
8720   int type;
8721   char *argv[MAXARGS];
8722   char *eargv[MAXARGS];
8723 };
8724
8725 struct redircmd {
8726   int type;
8727   struct cmd *cmd;
8728   char *file;
8729   char *efile;
8730   int mode;
8731   int fd;
8732 };
8733
8734 struct pipecmd {
8735   int type;
8736   struct cmd *left;
8737   struct cmd *right;
8738 };
8739
8740 struct listcmd {
8741   int type;
8742   struct cmd *left;
8743   struct cmd *right;
8744 };
8745
8746 struct backcmd {
8747   int type;
8748   struct cmd *cmd;
8749 };
```

```
8750 int fork1(void);  // Fork but panics on failure.
8751 void panic(char*);
8752 struct cmd *parsecmd(char*);
8753
8754 // Execute cmd.  Never returns.
8755 void
8756 runcmd(struct cmd *cmd)
8757 {
8758   int p[2];
8759   struct backcmd *bcmd;
8760   struct execcmd *ecmd;
8761   struct listcmd *lcmd;
8762   struct pipecmd *pcmd;
8763   struct redircmd *rcmd;
8764
8765   if(cmd == 0)
8766     exit();
8767
8768   switch(cmd->type){
8769   default:
8770     panic("runcmd");
8771
8772   case EXEC:
8773     ecmd = (struct execcmd*)cmd;
8774     if(ecmd->argv[0] == 0)
8775       exit();
8776     exec(ecmd->argv[0], ecmd->argv);
8777     printf(2, "exec %s failed\n", ecmd->argv[0]);
8778     break;
8779
8780   case REDIR:
8781     rcmd = (struct redircmd*)cmd;
8782     close(rcmd->fd);
8783     if(open(rcmd->file, rcmd->mode) < 0){
8784       printf(2, "open %s failed\n", rcmd->file);
8785       exit();
8786     }
8787     runcmd(rcmd->cmd);
8788     break;
8789
8790   case LIST:
8791     lcmd = (struct listcmd*)cmd;
8792     if(fork1() == 0)
8793       runcmd(lcmd->left);
8794     wait();
8795     runcmd(lcmd->right);
8796     break;
8797
8798
8799
```



```
8800   case PIPE:
8801     pcmd = (struct pipecmd*)cmd;
8802     if(pipe(p) < 0)
8803       panic("pipe");
8804     if(fork1() == 0){
8805       close(1);
8806       dup(p[1]);
8807       close(p[0]);
8808       close(p[1]);
8809       runcmd(pcmd->left);
8810     }
8811     if(fork1() == 0){
8812       close(0);
8813       dup(p[0]);
8814       close(p[0]);
8815       close(p[1]);
8816       runcmd(pcmd->right);
8817     }
8818     close(p[0]);
8819     close(p[1]);
8820     wait();
8821     wait();
8822     break;
8823
8824   case BACK:
8825     bcmd = (struct backcmd*)cmd;
8826     if(fork1() == 0)
8827       runcmd(bcmd->cmd);
8828     break;
8829   }
8830   exit();
8831 }
8832
8833 int
8834 getcmd(char *buf, int nbuf)
8835 {
8836   printf(2, "$ ");
8837   memset(buf, 0, nbuf);
8838   gets(buf, nbuf);
8839   if(buf[0] == 0) // EOF
8840     return -1;
8841   return 0;
8842 }
8843
8844
8845
8846
8847
8848
8849
```

```
8850 int
8851 main(void)
8852 {
8853   static char buf[100];
8854   int fd;
8855
8856   // Ensure that three file descriptors are open.
8857   while((fd = open("console", O_RDWR)) >= 0){
8858     if(fd >= 3){
8859       close(fd);
8860       break;
8861     }
8862   }
8863
8864   // Read and run input commands.
8865   while(getcmd(buf, sizeof(buf)) >= 0){
8866     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
8867       // Chdir must be called by the parent, not the child.
8868       buf[strlen(buf)-1] = 0;  // chop \n
8869       if(chdir(buf+3) < 0)
8870         printf(2, "cannot cd %s\n", buf+3);
8871       continue;
8872     }
8873     if(fork1() == 0)
8874       runcmd(parsecmd(buf));
8875     wait();
8876   }
8877   exit();
8878 }
8879
8880 void
8881 panic(char *s)
8882 {
8883   printf(2, "%s\n", s);
8884   exit();
8885 }
8886
8887 int
8888 fork1(void)
8889 {
8890   int pid;
8891
8892   pid = fork();
8893   if(pid == -1)
8894     panic("fork");
8895   return pid;
8896 }
8897
8898
8899
```



```
8900 // Constructors
8901
8902 struct cmd*
8903 execcmd(void)
8904 {
8905   struct execcmd *cmd;
8906
8907   cmd = malloc(sizeof(*cmd));
8908   memset(cmd, 0, sizeof(*cmd));
8909   cmd->type = EXEC;
8910   return (struct cmd*)cmd;
8911 }
8912
8913 struct cmd*
8914 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
8915 {
8916   struct redircmd *cmd;
8917
8918   cmd = malloc(sizeof(*cmd));
8919   memset(cmd, 0, sizeof(*cmd));
8920   cmd->type = REDIR;
8921   cmd->cmd = subcmd;
8922   cmd->file = file;
8923   cmd->efile = efile;
8924   cmd->mode = mode;
8925   cmd->fd = fd;
8926   return (struct cmd*)cmd;
8927 }
8928
8929 struct cmd*
8930 pipecmd(struct cmd *left, struct cmd *right)
8931 {
8932   struct pipecmd *cmd;
8933
8934   cmd = malloc(sizeof(*cmd));
8935   memset(cmd, 0, sizeof(*cmd));
8936   cmd->type = PIPE;
8937   cmd->left = left;
8938   cmd->right = right;
8939   return (struct cmd*)cmd;
8940 }
8941
8942
8943
8944
8945
8946
8947
8948
8949
```

```
8950 struct cmd*
8951 listcmd(struct cmd *left, struct cmd *right)
8952 {
8953   struct listcmd *cmd;
8954
8955   cmd = malloc(sizeof(*cmd));
8956   memset(cmd, 0, sizeof(*cmd));
8957   cmd->type = LIST;
8958   cmd->left = left;
8959   cmd->right = right;
8960   return (struct cmd*)cmd;
8961 }
8962
8963 struct cmd*
8964 backcmd(struct cmd *subcmd)
8965 {
8966   struct backcmd *cmd;
8967
8968   cmd = malloc(sizeof(*cmd));
8969   memset(cmd, 0, sizeof(*cmd));
8970   cmd->type = BACK;
8971   cmd->cmd = subcmd;
8972   return (struct cmd*)cmd;
8973 }
8974
8975
8976
8977
8978
8979
8980
8981
8982
8983
8984
8985
8986
8987
8988
8989
8990
8991
8992
8993
8994
8995
8996
8997
8998
8999
```



```
9000 // Parsing
9001
9002 char whitespace[] = " \t\r\n\v";
9003 char symbols[] = "<|>&;()";
9004
9005 int
9006 gettoken(char **ps, char *es, char **q, char **eq)
9007 {
9008   char *s;
9009   int ret;
9010
9011   s = *ps;
9012   while(s < es && strchr(whitespace, *s))
9013     s++;
9014   if(q)
9015     *q = s;
9016   ret = *s;
9017   switch(*s){
9018   case 0:
9019     break;
9020   case '|':
9021   case '(':
9022   case ')':
9023   case ';':
9024   case '&':
9025   case '<':
9026     s++;
9027     break;
9028   case '>':
9029     s++;
9030     if(*s == '>'){
9031       ret = '+';
9032       s++;
9033     }
9034     break;
9035   default:
9036     ret = 'a';
9037     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
9038       s++;
9039     break;
9040   }
9041   if(eq)
9042     *eq = s;
9043
9044   while(s < es && strchr(whitespace, *s))
9045     s++;
9046   *ps = s;
9047   return ret;
9048 }
9049
```

```
9050 int
9051 peek(char **ps, char *es, char *toks)
9052 {
9053   char *s;
9054
9055   s = *ps;
9056   while(s < es && strchr(whitespace, *s))
9057     s++;
9058   *ps = s;
9059   return *s && strchr(toks, *s);
9060 }
9061
9062 struct cmd *parseline(char**, char*);
9063 struct cmd *parsepipe(char**, char*);
9064 struct cmd *parseexec(char**, char*);
9065 struct cmd *nulterminate(struct cmd*);
9066
9067 struct cmd*
9068 parsecmd(char *s)
9069 {
9070   char *es;
9071   struct cmd *cmd;
9072
9073   es = s + strlen(s);
9074   cmd = parseline(&s, es);
9075   peek(&s, es, "");
9076   if(s != es){
9077     printf(2, "leftovers: %s\n", s);
9078     panic("syntax");
9079   }
9080   nulterminate(cmd);
9081   return cmd;
9082 }
9083
9084 struct cmd*
9085 parseline(char **ps, char *es)
9086 {
9087   struct cmd *cmd;
9088
9089   cmd = parsepipe(ps, es);
9090   while(peek(ps, es, "&")){
9091     gettoken(ps, es, 0, 0);
9092     cmd = backcmd(cmd);
9093   }
9094   if(peek(ps, es, ";")){
9095     gettoken(ps, es, 0, 0);
9096     cmd = listcmd(cmd, parseline(ps, es));
9097   }
9098   return cmd;
9099 }
```



```
9100 struct cmd*
9101 parsepipe(char **ps, char *es)
9102 {
9103   struct cmd *cmd;
9104
9105   cmd = parseexec(ps, es);
9106   if(peek(ps, es, "|")){
9107     gettoken(ps, es, 0, 0);
9108     cmd = pipecmd(cmd, parsepipe(ps, es));
9109   }
9110   return cmd;
9111 }
9112
9113 struct cmd*
9114 parseredirs(struct cmd *cmd, char **ps, char *es)
9115 {
9116   int tok;
9117   char *q, *eq;
9118
9119   while(peek(ps, es, "<>")){
9120     tok = gettoken(ps, es, 0, 0);
9121     if(gettoken(ps, es, &q, &eq) != 'a')
9122       panic("missing file for redirection");
9123     switch(tok){
9124     case '<':
9125       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
9126       break;
9127     case '>':
9128       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9129       break;
9130     case '+':  // >>
9131       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9132       break;
9133     }
9134   }
9135   return cmd;
9136 }
9137
9138
9139
9140
9141
9142
9143
9144
9145
9146
9147
9148
9149
```

```
9150 struct cmd*
9151 parseblock(char **ps, char *es)
9152 {
9153   struct cmd *cmd;
9154
9155   if(!peek(ps, es, "("))
9156     panic("parseblock");
9157   gettoken(ps, es, 0, 0);
9158   cmd = parseline(ps, es);
9159   if(!peek(ps, es, ")"))
9160     panic("syntax - missing )");
9161   gettoken(ps, es, 0, 0);
9162   cmd = parseredirs(cmd, ps, es);
9163   return cmd;
9164 }
9165
9166 struct cmd*
9167 parseexec(char **ps, char *es)
9168 {
9169   char *q, *eq;
9170   int tok, argc;
9171   struct execcmd *cmd;
9172   struct cmd *ret;
9173
9174   if(peek(ps, es, "("))
9175     return parseblock(ps, es);
9176
9177   ret = execcmd();
9178   cmd = (struct execcmd*)ret;
9179
9180   argc = 0;
9181   ret = parseredirs(ret, ps, es);
9182   while(!peek(ps, es, "|)&;")){
9183     if((tok=gettoken(ps, es, &q, &eq)) == 0)
9184       break;
9185     if(tok != 'a')
9186       panic("syntax");
9187     cmd->argv[argc] = q;
9188     cmd->eargv[argc] = eq;
9189     argc++;
9190     if(argc >= MAXARGS)
9191       panic("too many args");
9192     ret = parseredirs(ret, ps, es);
9193   }
9194   cmd->argv[argc] = 0;
9195   cmd->eargv[argc] = 0;
9196   return ret;
9197 }
9198
9199
```



```
9200 // NUL-terminate all the counted strings.
9201 struct cmd*
9202 nulterminate(struct cmd *cmd)
9203 {
9204   int i;
9205   struct backcmd *bcmd;
9206   struct execcmd *ecmd;
9207   struct listcmd *lcmd;
9208   struct pipecmd *pcmd;
9209   struct redircmd *rcmd;
9210
9211   if(cmd == 0)
9212     return 0;
9213
9214   switch(cmd->type){
9215   case EXEC:
9216     ecmd = (struct execcmd*)cmd;
9217     for(i=0; ecmd->argv[i]; i++)
9218       *ecmd->eargv[i] = 0;
9219     break;
9220
9221   case REDIR:
9222     rcmd = (struct redircmd*)cmd;
9223     nulterminate(rcmd->cmd);
9224     *rcmd->efile = 0;
9225     break;
9226
9227   case PIPE:
9228     pcmd = (struct pipecmd*)cmd;
9229     nulterminate(pcmd->left);
9230     nulterminate(pcmd->right);
9231     break;
9232
9233   case LIST:
9234     lcmd = (struct listcmd*)cmd;
9235     nulterminate(lcmd->left);
9236     nulterminate(lcmd->right);
9237     break;
9238
9239   case BACK:
9240     bcmd = (struct backcmd*)cmd;
9241     nulterminate(bcmd->cmd);
9242     break;
9243   }
9244   return cmd;
9245 }
9246
9247
9248
9249
```

```
#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
```

```
// Boot loader.
//
// Part of the boot block, along with bootasm.S, which calls bootmain().
// bootasm.S has put the processor into protected 32-bit mode.
// bootmain() loads an ELF kernel image from the disk starting at
// sector 1 and then jumps to the kernel entry routine.

#include "types.h"
#include "elf.h"
#include "x86.h"
#include "memlayout.h"

#define SECTSIZE  512

void readseg(uchar*, uint, uint);

void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```



```
void
waitdisk(void)
{
  // Wait for disk ready.
  while((inb(0x1F7) & 0xC0) != 0x40)
    ;
}

// Read a single sector at offset into dst.
void
readsect(void *dst, uint offset)
{
  // Issue command.
  waitdisk();
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
  outb(0x1F5, offset >> 16);
  outb(0x1F6, (offset >> 24) | 0xE0);
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
  insl(0x1F0, dst, SECTSIZE/4);
}

// Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
// Might copy more than asked.
void
readseg(uchar* pa, uint count, uint offset)
{
  uchar* epa;

  epa = pa + count;

  // Round down to sector boundary.
  pa -= offset % SECTSIZE;

  // Translate from bytes to sectors; kernel starts at sector 1.
  offset = (offset / SECTSIZE) + 1;

  // If this is too slow, we could read lots of sectors at a time.
  // We'd write more to memory than asked, but it doesn't matter --
  // we load in increasing order.
  for(; pa < epa; pa += SECTSIZE, offset++)
    readsect(pa, offset);
}
```