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/6.828/2014/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
    Austin Clements

In addition, we are grateful for the bug reports and patches contributed by
Silas Boyd-Wickizer, Peter Froehlich, Shivam Handa, Anders Kaseorg, Eddie
Kohler, Yandong Mao, Hitoshi Mitake, Carmi Merimovich, Joel Nider, Greg Price,
Eldar Sehayek, Yongming Shen, Stephen Tu, and Zouchangwei.

ERROR REPORTS

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

BUILDING AND RUNNING XV6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run "make".
On non-x86 or non-ELF machines (like OS X, even on x86), you will
need to install a cross-compiler gcc suite capable of producing x86 ELF
binaries. See http://pdos.csail.mit.edu/6.828/2014/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 #ifndef __ASSEMBLER__
0211
0212 static inline uint v2p(void *a) { return ((uint) (a))  - KERNBASE; }
0213 static inline void *p2v(uint a) { return (void *) ((a) + KERNBASE); }
0214
0215 #endif
0216
0217 #define V2P(a) (((uint) (a)) - KERNBASE)
0218 #define P2V(a) (((void *) (a)) + KERNBASE)
0219
0220 #define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
0221 #define P2V_WO(x) ((x) + KERNBASE)    // same as P2V, but without casts
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 rtcdate {
0251   uint second;
0252   uint minute;
0253   uint hour;
0254   uint day;
0255   uint month;
0256   uint year;
0257 };
0258
0259
0260
0261
0262
0263
0264
0265
0266
0267
0268
0269
0270
0271
0272
0273
0274
0275
0276
0277
0278
0279
0280
0281
0282
0283
0284
0285
0286
0287
0288
0289
0290
0291
0292
0293
0294
0295
0296
0297
0298
0299
```

```
0300 struct buf;
0301 struct context;
0302 struct file;
0303 struct inode;
0304 struct pipe;
0305 struct proc;
0306 struct rtcdate;
0307 struct spinlock;
0308 struct stat;
0309 struct superblock;
0310
0311 // bio.c
0312 void            binit(void);
0313 struct buf*     bread(uint, uint);
0314 void            brelse(struct buf*);
0315 void            bwrite(struct buf*);
0316
0317 // console.c
0318 void            consoleinit(void);
0319 void            cprintf(char*, ...);
0320 void            consoleintr(int(*)(void));
0321 void            panic(char*) __attribute__((noreturn));
0322
0323 // exec.c
0324 int             exec(char*, char**);
0325
0326 // file.c
0327 struct file*    filealloc(void);
0328 void            fileclose(struct file*);
0329 struct file*    filedup(struct file*);
0330 void            fileinit(void);
0331 int             fileread(struct file*, char*, int n);
0332 int             filestat(struct file*, struct stat*);
0333 int             filewrite(struct file*, char*, int n);
0334
0335 // fs.c
0336 void            readsb(int dev, struct superblock *sb);
0337 int             dirlink(struct inode*, char*, uint);
0338 struct inode*   dirlookup(struct inode*, char*, uint*);
0339 struct inode*   ialloc(uint, short);
0340 struct inode*   idup(struct inode*);
0341 void            iinit(int dev);
0342 void            ilock(struct inode*);
0343 void            iput(struct inode*);
0344 void            iunlock(struct inode*);
0345 void            iunlockput(struct inode*);
0346 void            iupdate(struct inode*);
0347 int             namecmp(const char*, const char*);
0348 struct inode*   namei(char*);
0349 struct inode*   nameiparent(char*, char*);
```



```
0350 int             readi(struct inode*, char*, uint, uint);
0351 void            stati(struct inode*, struct stat*);
0352 int             writei(struct inode*, char*, uint, uint);
0353
0354 // ide.c
0355 void            ideinit(void);
0356 void            ideintr(void);
0357 void            iderw(struct buf*);
0358
0359 // ioapic.c
0360 void            ioapicenable(int irq, int cpu);
0361 extern uchar    ioapicid;
0362 void            ioapicinit(void);
0363
0364 // kalloc.c
0365 char*           kalloc(void);
0366 void            kfree(char*);
0367 void            kinit1(void*, void*);
0368 void            kinit2(void*, void*);
0369
0370 // kbd.c
0371 void            kbdintr(void);
0372
0373 // lapic.c
0374 void            cmostime(struct rtcdate *r);
0375 int             cpunum(void);
0376 extern volatile uint*    lapic;
0377 void            lapiceoi(void);
0378 void            lapicinit(void);
0379 void            lapicstartap(uchar, uint);
0380 void            microdelay(int);
0381
0382 // log.c
0383 void            initlog(int dev);
0384 void            log_write(struct buf*);
0385 void            begin_op();
0386 void            end_op();
0387
0388 // mp.c
0389 extern int      ismp;
0390 int             mpbcpu(void);
0391 void            mpinit(void);
0392 void            mpstartthem(void);
0393
0394 // picirq.c
0395 void            picenable(int);
0396 void            picinit(void);
0397
0398
0399
```

```
0400 // pipe.c
0401 int             pipealloc(struct file**, struct file**);
0402 void            pipeclose(struct pipe*, int);
0403 int             piperead(struct pipe*, char*, int);
0404 int             pipewrite(struct pipe*, char*, int);
0405
0406
0407 // proc.c
0408 struct proc*    copyproc(struct proc*);
0409 void            exit(void);
0410 int             fork(void);
0411 int             growproc(int);
0412 int             kill(int);
0413 void            pinit(void);
0414 void            procdump(void);
0415 void            scheduler(void) __attribute__((noreturn));
0416 void            sched(void);
0417 void            sleep(void*, struct spinlock*);
0418 void            userinit(void);
0419 int             wait(void);
0420 void            wakeup(void*);
0421 void            yield(void);
0422
0423 // swtch.S
0424 void            swtch(struct context**, struct context*);
0425
0426 // spinlock.c
0427 void            acquire(struct spinlock*);
0428 void            getcallerpcs(void*, uint*);
0429 int             holding(struct spinlock*);
0430 void            initlock(struct spinlock*, char*);
0431 void            release(struct spinlock*);
0432 void            pushcli(void);
0433 void            popcli(void);
0434
0435 // string.c
0436 int             memcmp(const void*, const void*, uint);
0437 void*           memmove(void*, const void*, uint);
0438 void*           memset(void*, int, uint);
0439 char*           safestrcpy(char*, const char*, int);
0440 int             strlen(const char*);
0441 int             strncmp(const char*, const char*, uint);
0442 char*           strncpy(char*, const char*, int);
0443
0444 // syscall.c
0445 int             argint(int, int*);
0446 int             argptr(int, char**, int);
0447 int             argstr(int, char**);
0448 int             fetchint(uint, int*);
0449 int             fetchstr(uint, char**);
```



```
0450 void            syscall(void);
0451
0452 // timer.c
0453 void            timerinit(void);
0454
0455 // trap.c
0456 void            idtinit(void);
0457 extern uint     ticks;
0458 void            tvinit(void);
0459 extern struct spinlock tickslock;
0460
0461 // uart.c
0462 void            uartinit(void);
0463 void            uartintr(void);
0464 void            uartputc(int);
0465
0466 // vm.c
0467 void            seginit(void);
0468 void            kvmalloc(void);
0469 void            vmenable(void);
0470 pde_t*          setupkvm(void);
0471 char*           uva2ka(pde_t*, char*);
0472 int             allocuvm(pde_t*, uint, uint);
0473 int             deallocuvm(pde_t*, uint, uint);
0474 void            freevm(pde_t*);
0475 void            inituvm(pde_t*, char*, uint);
0476 int             loaduvm(pde_t*, char*, struct inode*, uint, uint);
0477 pde_t*          copyuvm(pde_t*, uint);
0478 void            switchuvm(struct proc*);
0479 void            switchkvm(void);
0480 int             copyout(pde_t*, uint, void*, uint);
0481 void            clearpteu(pde_t *pgdir, char *uva);
0482
0483 // number of elements in fixed-size array
0484 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
0485
0486
0487
0488
0489
0490
0491
0492
0493
0494
0495
0496
0497
0498
0499
```

```
0500 // Routines to let C code use special x86 instructions.
0501
0502 static inline uchar
0503 inb(ushort port)
0504 {
0505   uchar data;
0506
0507   asm volatile("in %1,%0" : "=a" (data) : "d" (port));
0508   return data;
0509 }
0510
0511 static inline void
0512 insl(int port, void *addr, int cnt)
0513 {
0514   asm volatile("cld; rep insl" :
0515                "=D" (addr), "=c" (cnt) :
0516                "d" (port), "0" (addr), "1" (cnt) :
0517                "memory", "cc");
0518 }
0519
0520 static inline void
0521 outb(ushort port, uchar data)
0522 {
0523   asm volatile("out %0,%1" : : "a" (data), "d" (port));
0524 }
0525
0526 static inline void
0527 outw(ushort port, ushort data)
0528 {
0529   asm volatile("out %0,%1" : : "a" (data), "d" (port));
0530 }
0531
0532 static inline void
0533 outsl(int port, const void *addr, int cnt)
0534 {
0535   asm volatile("cld; rep outsl" :
0536                "=S" (addr), "=c" (cnt) :
0537                "d" (port), "0" (addr), "1" (cnt) :
0538                "cc");
0539 }
0540
0541 static inline void
0542 stosb(void *addr, int data, int cnt)
0543 {
0544   asm volatile("cld; rep stosb" :
0545                "=D" (addr), "=c" (cnt) :
0546                "0" (addr), "1" (cnt), "a" (data) :
0547                "memory", "cc");
0548 }
0549
```



```
0550 static inline void
0551 stosl(void *addr, int data, int cnt)
0552 {
0553   asm volatile("cld; rep stosl" :
0554                "=D" (addr), "=c" (cnt) :
0555                "0" (addr), "1" (cnt), "a" (data) :
0556                "memory", "cc");
0557 }
0558
0559 struct segdesc;
0560
0561 static inline void
0562 lgdt(struct segdesc *p, int size)
0563 {
0564   volatile ushort pd[3];
0565
0566   pd[0] = size-1;
0567   pd[1] = (uint)p;
0568   pd[2] = (uint)p >> 16;
0569
0570   asm volatile("lgdt (%0)" : : "r" (pd));
0571 }
0572
0573 struct gatedesc;
0574
0575 static inline void
0576 lidt(struct gatedesc *p, int size)
0577 {
0578   volatile ushort pd[3];
0579
0580   pd[0] = size-1;
0581   pd[1] = (uint)p;
0582   pd[2] = (uint)p >> 16;
0583
0584   asm volatile("lidt (%0)" : : "r" (pd));
0585 }
0586
0587 static inline void
0588 ltr(ushort sel)
0589 {
0590   asm volatile("ltr %0" : : "r" (sel));
0591 }
0592
0593 static inline uint
0594 readeflags(void)
0595 {
0596   uint eflags;
0597   asm volatile("pushfl; popl %0" : "=r" (eflags));
0598   return eflags;
0599 }
```

```
0600 static inline void
0601 loadgs(ushort v)
0602 {
0603   asm volatile("movw %0, %%gs" : : "r" (v));
0604 }
0605
0606 static inline void
0607 cli(void)
0608 {
0609   asm volatile("cli");
0610 }
0611
0612 static inline void
0613 sti(void)
0614 {
0615   asm volatile("sti");
0616 }
0617
0618 static inline uint
0619 xchg(volatile uint *addr, uint newval)
0620 {
0621   uint result;
0622
0623   // The + in "+m" denotes a read-modify-write operand.
0624   asm volatile("lock; xchgl %0, %1" :
0625                "+m" (*addr), "=a" (result) :
0626                "1" (newval) :
0627                "cc");
0628   return result;
0629 }
0630
0631 static inline uint
0632 rcr2(void)
0633 {
0634   uint val;
0635   asm volatile("movl %%cr2,%0" : "=r" (val));
0636   return val;
0637 }
0638
0639 static inline void
0640 lcr3(uint val)
0641 {
0642   asm volatile("movl %0,%%cr3" : : "r" (val));
0643 }
0644
0645
0646
0647
0648
0649
```



```
0650 // Layout of the trap frame built on the stack by the
0651 // hardware and by trapasm.S, and passed to trap().
0652 struct trapframe {
0653   // registers as pushed by pusha
0654   uint edi;
0655   uint esi;
0656   uint ebp;
0657   uint oesp;      // useless & ignored
0658   uint ebx;
0659   uint edx;
0660   uint ecx;
0661   uint eax;
0662
0663   // rest of trap frame
0664   ushort gs;
0665   ushort padding1;
0666   ushort fs;
0667   ushort padding2;
0668   ushort es;
0669   ushort padding3;
0670   ushort ds;
0671   ushort padding4;
0672   uint trapno;
0673
0674   // below here defined by x86 hardware
0675   uint err;
0676   uint eip;
0677   ushort cs;
0678   ushort padding5;
0679   uint eflags;
0680
0681   // below here only when crossing rings, such as from user to kernel
0682   uint esp;
0683   ushort ss;
0684   ushort padding6;
0685 };
0686
0687
0688
0689
0690
0691
0692
0693
0694
0695
0696
0697
0698
0699
```

```
0700 //
0701 // assembler macros to create x86 segments
0702 //
0703
0704 #define SEG_NULLASM                                             \
0705         .word 0, 0;                                             \
0706         .byte 0, 0, 0, 0
0707
0708 // The 0xC0 means the limit is in 4096-byte units
0709 // and (for executable segments) 32-bit mode.
0710 #define SEG_ASM(type,base,lim)                                  \
0711         .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
0712         .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
0713                 (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
0714
0715 #define STA_X     0x8       // Executable segment
0716 #define STA_E     0x4       // Expand down (non-executable segments)
0717 #define STA_C     0x4       // Conforming code segment (executable only)
0718 #define STA_W     0x2       // Writeable (non-executable segments)
0719 #define STA_R     0x2       // Readable (executable segments)
0720 #define STA_A     0x1       // Accessed
0721
0722
0723
0724
0725
0726
0727
0728
0729
0730
0731
0732
0733
0734
0735
0736
0737
0738
0739
0740
0741
0742
0743
0744
0745
0746
0747
0748
0749
```



```
0750 // This file contains definitions for the
0751 // x86 memory management unit (MMU).
0752
0753 // Eflags register
0754 #define FL_CF           0x00000001      // Carry Flag
0755 #define FL_PF           0x00000004      // Parity Flag
0756 #define FL_AF           0x00000010      // Auxiliary carry Flag
0757 #define FL_ZF           0x00000040      // Zero Flag
0758 #define FL_SF           0x00000080      // Sign Flag
0759 #define FL_TF           0x00000100      // Trap Flag
0760 #define FL_IF           0x00000200      // Interrupt Enable
0761 #define FL_DF           0x00000400      // Direction Flag
0762 #define FL_OF           0x00000800      // Overflow Flag
0763 #define FL_IOPL_MASK    0x00003000      // I/O Privilege Level bitmask
0764 #define FL_IOPL_0       0x00000000      //   IOPL == 0
0765 #define FL_IOPL_1       0x00001000      //   IOPL == 1
0766 #define FL_IOPL_2       0x00002000      //   IOPL == 2
0767 #define FL_IOPL_3       0x00003000      //   IOPL == 3
0768 #define FL_NT           0x00004000      // Nested Task
0769 #define FL_RF           0x00010000      // Resume Flag
0770 #define FL_VM           0x00020000      // Virtual 8086 mode
0771 #define FL_AC           0x00040000      // Alignment Check
0772 #define FL_VIF          0x00080000      // Virtual Interrupt Flag
0773 #define FL_VIP          0x00100000      // Virtual Interrupt Pending
0774 #define FL_ID           0x00200000      // ID flag
0775
0776 // Control Register flags
0777 #define CR0_PE          0x00000001      // Protection Enable
0778 #define CR0_MP          0x00000002      // Monitor coProcessor
0779 #define CR0_EM          0x00000004      // Emulation
0780 #define CR0_TS          0x00000008      // Task Switched
0781 #define CR0_ET          0x00000010      // Extension Type
0782 #define CR0_NE          0x00000020      // Numeric Errror
0783 #define CR0_WP          0x00010000      // Write Protect
0784 #define CR0_AM          0x00040000      // Alignment Mask
0785 #define CR0_NW          0x20000000      // Not Writethrough
0786 #define CR0_CD          0x40000000      // Cache Disable
0787 #define CR0_PG          0x80000000      // Paging
0788
0789 #define CR4_PSE         0x00000010      // Page size extension
0790
0791 #define SEG_KCODE 1  // kernel code
0792 #define SEG_KDATA 2  // kernel data+stack
0793 #define SEG_KCPU  3  // kernel per-cpu data
0794 #define SEG_UCODE 4  // user code
0795 #define SEG_UDATA 5  // user data+stack
0796 #define SEG_TSS   6  // this process's task state
0797
0798
0799
```

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



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

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



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

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



```
1050 # Multiboot header, for multiboot boot loaders like GNU Grub.
1051 # http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
1052 #
1053 # Using GRUB 2, you can boot xv6 from a file stored in a
1054 # Linux file system by copying kernel or kernelmemfs to /boot
1055 # and then adding this menu entry:
1056 #
1057 # menuentry "xv6" {
1058 #  insmod ext2
1059 #  set root='(hd0,msdos1)'
1060 #  set kernel='/boot/kernel'
1061 #  echo "Loading ${kernel}..."
1062 #  multiboot ${kernel} ${kernel}
1063 #  boot
1064 # }
1065
1066 #include "asm.h"
1067 #include "memlayout.h"
1068 #include "mmu.h"
1069 #include "param.h"
1070
1071 # Multiboot header.  Data to direct multiboot loader.
1072 .p2align 2
1073 .text
1074 .globl multiboot_header
1075 multiboot_header:
1076   #define magic 0x1badb002
1077   #define flags 0
1078   .long magic
1079   .long flags
1080   .long (-magic-flags)
1081
1082 # By convention, the _start symbol specifies the ELF entry point.
1083 # Since we haven't set up virtual memory yet, our entry point is
1084 # the physical address of 'entry'.
1085 .globl _start
1086 _start = V2P_WO(entry)
1087
1088 # Entering xv6 on boot processor, with paging off.
1089 .globl entry
1090 entry:
1091   # Turn on page size extension for 4Mbyte pages
1092   movl    %cr4, %eax
1093   orl     $(CR4_PSE), %eax
1094   movl    %eax, %cr4
1095   # Set page directory
1096   movl    $(V2P_WO(entrypgdir)), %eax
1097   movl    %eax, %cr3
1098   # Turn on paging.
1099   movl    %cr0, %eax
```

```
1100   orl     $(CR0_PG|CR0_WP), %eax
1101   movl    %eax, %cr0
1102
1103   # Set up the stack pointer.
1104   movl $(stack + KSTACKSIZE), %esp
1105
1106   # Jump to main(), and switch to executing at
1107   # high addresses. The indirect call is needed because
1108   # the assembler produces a PC-relative instruction
1109   # for a direct jump.
1110   mov $main, %eax
1111   jmp *%eax
1112
1113 .comm stack, KSTACKSIZE
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
```



```
1150 #include "asm.h"
1151 #include "memlayout.h"
1152 #include "mmu.h"
1153
1154 # Each non-boot CPU ("AP") is started up in response to a STARTUP
1155 # IPI from the boot CPU.  Section B.4.2 of the Multi-Processor
1156 # Specification says that the AP will start in real mode with CS:IP
1157 # set to XY00:0000, where XY is an 8-bit value sent with the
1158 # STARTUP. Thus this code must start at a 4096-byte boundary.
1159 #
1160 # Because this code sets DS to zero, it must sit
1161 # at an address in the low 2^16 bytes.
1162 #
1163 # Startothers (in main.c) sends the STARTUPs one at a time.
1164 # It copies this code (start) at 0x7000.  It puts the address of
1165 # a newly allocated per-core stack in start-4,the address of the
1166 # place to jump to (mpenter) in start-8, and the physical address
1167 # of entrypgdir in start-12.
1168 #
1169 # This code is identical to bootasm.S except:
1170 #   - it does not need to enable A20
1171 #   - it uses the address at start-4, start-8, and start-12
1172
1173 .code16
1174 .globl start
1175 start:
1176   cli
1177
1178   xorw    %ax,%ax
1179   movw    %ax,%ds
1180   movw    %ax,%es
1181   movw    %ax,%ss
1182
1183   lgdt    gdtdesc
1184   movl    %cr0, %eax
1185   orl     $CR0_PE, %eax
1186   movl    %eax, %cr0
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
```

```
1200   ljmpl    $(SEG_KCODE<<3), $(start32)
1201
1202 .code32
1203 start32:
1204   movw    $(SEG_KDATA<<3), %ax
1205   movw    %ax, %ds
1206   movw    %ax, %es
1207   movw    %ax, %ss
1208   movw    $0, %ax
1209   movw    %ax, %fs
1210   movw    %ax, %gs
1211
1212   # Turn on page size extension for 4Mbyte pages
1213   movl    %cr4, %eax
1214   orl     $(CR4_PSE), %eax
1215   movl    %eax, %cr4
1216   # Use enterpgdir as our initial page table
1217   movl    (start-12), %eax
1218   movl    %eax, %cr3
1219   # Turn on paging.
1220   movl    %cr0, %eax
1221   orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
1222   movl    %eax, %cr0
1223
1224   # Switch to the stack allocated by startothers()
1225   movl    (start-4), %esp
1226   # Call mpenter()
1227   call      *(start-8)
1228
1229   movw    $0x8a00, %ax
1230   movw    %ax, %dx
1231   outw    %ax, %dx
1232   movw    $0x8ae0, %ax
1233   outw    %ax, %dx
1234 spin:
1235   jmp     spin
1236
1237 .p2align 2
1238 gdt:
1239   SEG_NULLASM
1240   SEG_ASM(STA_X|STA_R, 0, 0xffffffff)
1241   SEG_ASM(STA_W, 0, 0xffffffff)
1242
1243
1244 gdtdesc:
1245   .word   (gdtdesc - gdt - 1)
1246   .long   gdt
1247
1248
1249
```



```
1250 #include "types.h"
1251 #include "defs.h"
1252 #include "param.h"
1253 #include "memlayout.h"
1254 #include "mmu.h"
1255 #include "proc.h"
1256 #include "x86.h"
1257
1258 static void startothers(void);
1259 static void mpmain(void)  __attribute__((noreturn));
1260 extern pde_t *kpgdir;
1261 extern char end[]; // first address after kernel loaded from ELF file
1262
1263 // Bootstrap processor starts running C code here.
1264 // Allocate a real stack and switch to it, first
1265 // doing some setup required for memory allocator to work.
1266 int
1267 main(void)
1268 {
1269   kinit1(end, P2V(4*1024*1024)); // phys page allocator
1270   kvmalloc();      // kernel page table
1271   mpinit();        // collect info about this machine
1272   lapicinit();
1273   seginit();       // set up segments
1274   cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
1275   picinit();       // interrupt controller
1276   ioapicinit();    // another interrupt controller
1277   consoleinit();   // I/O devices & their interrupts
1278   uartinit();      // serial port
1279   pinit();         // process table
1280   tvinit();        // trap vectors
1281   binit();         // buffer cache
1282   fileinit();      // file table
1283   ideinit();       // disk
1284   if(!ismp)
1285     timerinit();   // uniprocessor timer
1286   startothers();   // start other processors
1287   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1288   userinit();      // first user process
1289   // Finish setting up this processor in mpmain.
1290   mpmain();
1291 }
1292
1293
1294
1295
1296
1297
1298
1299
```

```
1300 // Other CPUs jump here from entryother.S.
1301 static void
1302 mpenter(void)
1303 {
1304   switchkvm();
1305   seginit();
1306   lapicinit();
1307   mpmain();
1308 }
1309
1310 // Common CPU setup code.
1311 static void
1312 mpmain(void)
1313 {
1314   cprintf("cpu%d: starting\n", cpu->id);
1315   idtinit();       // load idt register
1316   xchg(&cpu->started, 1); // tell startothers() we're up
1317   scheduler();     // start running processes
1318 }
1319
1320 pde_t entrypgdir[];  // For entry.S
1321
1322 // Start the non-boot (AP) processors.
1323 static void
1324 startothers(void)
1325 {
1326   extern uchar _binary_entryother_start[], _binary_entryother_size[];
1327   uchar *code;
1328   struct cpu *c;
1329   char *stack;
1330
1331   // Write entry code to unused memory at 0x7000.
1332   // The linker has placed the image of entryother.S in
1333   // _binary_entryother_start.
1334   code = p2v(0x7000);
1335   memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
1336
1337   for(c = cpus; c < cpus+ncpu; c++){
1338     if(c == cpus+cpunum())  // We've started already.
1339       continue;
1340
1341     // Tell entryother.S what stack to use, where to enter, and what
1342     // pgdir to use. We cannot use kpgdir yet, because the AP processor
1343     // is running in low  memory, so we use entrypgdir for the APs too.
1344     stack = kalloc();
1345     *(void**)(code-4) = stack + KSTACKSIZE;
1346     *(void**)(code-8) = mpenter;
1347     *(int**)(code-12) = (void *) v2p(entrypgdir);
1348
1349     lapicstartap(c->id, v2p(code));
```



```
1350     // wait for cpu to finish mpmain()
1351     while(c->started == 0)
1352       ;
1353   }
1354 }
1355
1356 // Boot page table used in entry.S and entryother.S.
1357 // Page directories (and page tables), must start on a page boundary,
1358 // hence the "__aligned__" attribute.
1359 // Use PTE_PS in page directory entry to enable 4Mbyte pages.
1360 __attribute__((__aligned__(PGSIZE)))
1361 pde_t entrypgdir[NPDENTRIES] = {
1362   // Map VA's [0, 4MB) to PA's [0, 4MB)
1363   [0] = (0) | PTE_P | PTE_W | PTE_PS,
1364   // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1365   [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1366 };
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
```

```
1400 // Blank page.
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
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 // Blank page.
1501
1502
1503
1504
1505
1506
1507
1508
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 lock.
1551 struct spinlock {
1552   uint locked;       // Is the lock held?
1553
1554   // For debugging:
1555   char *name;        // Name of lock.
1556   struct cpu *cpu;   // The cpu holding the lock.
1557   uint pcs[10];      // The call stack (an array of program counters)
1558                      // that locked the lock.
1559 };
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
```

```
1600 // Mutual exclusion spin locks.
1601
1602 #include "types.h"
1603 #include "defs.h"
1604 #include "param.h"
1605 #include "x86.h"
1606 #include "memlayout.h"
1607 #include "mmu.h"
1608 #include "proc.h"
1609 #include "spinlock.h"
1610
1611 void
1612 initlock(struct spinlock *lk, char *name)
1613 {
1614   lk->name = name;
1615   lk->locked = 0;
1616   lk->cpu = 0;
1617 }
1618
1619 // Acquire the lock.
1620 // Loops (spins) until the lock is acquired.
1621 // Holding a lock for a long time may cause
1622 // other CPUs to waste time spinning to acquire it.
1623 void
1624 acquire(struct spinlock *lk)
1625 {
1626   pushcli(); // disable interrupts to avoid deadlock.
1627   if(holding(lk))
1628     panic("acquire");
1629
1630   // The xchg is atomic.
1631   // It also serializes, so that reads after acquire are not
1632   // reordered before it.
1633   while(xchg(&lk->locked, 1) != 0)
1634     ;
1635
1636   // Record info about lock acquisition for debugging.
1637   lk->cpu = cpu;
1638   getcallerpcs(&lk, lk->pcs);
1639 }
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
```



```
1650 // Release the lock.
1651 void
1652 release(struct spinlock *lk)
1653 {
1654   if(!holding(lk))
1655     panic("release");
1656
1657   lk->pcs[0] = 0;
1658   lk->cpu = 0;
1659
1660   // The xchg serializes, so that reads before release are
1661   // not reordered after it.  The 1996 PentiumPro manual (Volume 3,
1662   // 7.2) says reads can be carried out speculatively and in
1663   // any order, which implies we need to serialize here.
1664   // But the 2007 Intel 64 Architecture Memory Ordering White
1665   // Paper says that Intel 64 and IA-32 will not move a load
1666   // after a store. So lock->locked = 0 would work here.
1667   // The xchg being asm volatile ensures gcc emits it after
1668   // the above assignments (and after the critical section).
1669   xchg(&lk->locked, 0);
1670
1671   popcli();
1672 }
1673
1674 // Record the current call stack in pcs[] by following the %ebp chain.
1675 void
1676 getcallerpcs(void *v, uint pcs[])
1677 {
1678   uint *ebp;
1679   int i;
1680
1681   ebp = (uint*)v - 2;
1682   for(i = 0; i < 10; i++){
1683     if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
1684       break;
1685     pcs[i] = ebp[1];     // saved %eip
1686     ebp = (uint*)ebp[0]; // saved %ebp
1687   }
1688   for(; i < 10; i++)
1689     pcs[i] = 0;
1690 }
1691
1692 // Check whether this cpu is holding the lock.
1693 int
1694 holding(struct spinlock *lock)
1695 {
1696   return lock->locked && lock->cpu == cpu;
1697 }
1698
1699
```

```
1700 // Pushcli/popcli are like cli/sti except that they are matched:
1701 // it takes two popcli to undo two pushcli.  Also, if interrupts
1702 // are off, then pushcli, popcli leaves them off.
1703
1704 void
1705 pushcli(void)
1706 {
1707   int eflags;
1708
1709   eflags = readeflags();
1710   cli();
1711   if(cpu->ncli++ == 0)
1712     cpu->intena = eflags & FL_IF;
1713 }
1714
1715 void
1716 popcli(void)
1717 {
1718   if(readeflags()&FL_IF)
1719     panic("popcli - interruptible");
1720   if(--cpu->ncli < 0)
1721     panic("popcli");
1722   if(cpu->ncli == 0 && cpu->intena)
1723     sti();
1724 }
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
```



```
1750 #include "param.h"
1751 #include "types.h"
1752 #include "defs.h"
1753 #include "x86.h"
1754 #include "memlayout.h"
1755 #include "mmu.h"
1756 #include "proc.h"
1757 #include "elf.h"
1758
1759 extern char data[];  // defined by kernel.ld
1760 pde_t *kpgdir;  // for use in scheduler()
1761 struct segdesc gdt[NSEGS];
1762
1763 // Set up CPU's kernel segment descriptors.
1764 // Run once on entry on each CPU.
1765 void
1766 seginit(void)
1767 {
1768   struct cpu *c;
1769
1770   // Map "logical" addresses to virtual addresses using identity map.
1771   // Cannot share a CODE descriptor for both kernel and user
1772   // because it would have to have DPL_USR, but the CPU forbids
1773   // an interrupt from CPL=0 to DPL=3.
1774   c = &cpus[cpunum()];
1775   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
1776   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1777   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
1778   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1779
1780   // Map cpu, and curproc
1781   c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
1782
1783   lgdt(c->gdt, sizeof(c->gdt));
1784   loadgs(SEG_KCPU << 3);
1785
1786   // Initialize cpu-local storage.
1787   cpu = c;
1788   proc = 0;
1789 }
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
```

```
1800 // Return the address of the PTE in page table pgdir
1801 // that corresponds to virtual address va.  If alloc!=0,
1802 // create any required page table pages.
1803 static pte_t *
1804 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1805 {
1806   pde_t *pde;
1807   pte_t *pgtab;
1808
1809   pde = &pgdir[PDX(va)];
1810   if(*pde & PTE_P){
1811     pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
1812   } else {
1813     if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1814       return 0;
1815     // Make sure all those PTE_P bits are zero.
1816     memset(pgtab, 0, PGSIZE);
1817     // The permissions here are overly generous, but they can
1818     // be further restricted by the permissions in the page table
1819     // entries, if necessary.
1820     *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
1821   }
1822   return &pgtab[PTX(va)];
1823 }
1824
1825 // Create PTEs for virtual addresses starting at va that refer to
1826 // physical addresses starting at pa. va and size might not
1827 // be page-aligned.
1828 static int
1829 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1830 {
1831   char *a, *last;
1832   pte_t *pte;
1833
1834   a = (char*)PGROUNDDOWN((uint)va);
1835   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1836   for(;;){
1837     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1838       return -1;
1839     if(*pte & PTE_P)
1840       panic("remap");
1841     *pte = pa | perm | PTE_P;
1842     if(a == last)
1843       break;
1844     a += PGSIZE;
1845     pa += PGSIZE;
1846   }
1847   return 0;
1848 }
1849
```



```
1850 // There is one page table per process, plus one that's used when
1851 // a CPU is not running any process (kpgdir). The kernel uses the
1852 // current process's page table during system calls and interrupts;
1853 // page protection bits prevent user code from using the kernel's
1854 // mappings.
1855 //
1856 // setupkvm() and exec() set up every page table like this:
1857 //
1858 //   0..KERNBASE: user memory (text+data+stack+heap), mapped to
1859 //                phys memory allocated by the kernel
1860 //   KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
1861 //   KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
1862 //                for the kernel's instructions and r/o data
1863 //   data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
1864 //                                     rw data + free physical memory
1865 //   0xfe000000..0: mapped direct (devices such as ioapic)
1866 //
1867 // The kernel allocates physical memory for its heap and for user memory
1868 // between V2P(end) and the end of physical memory (PHYSTOP)
1869 // (directly addressable from end..P2V(PHYSTOP)).
1870
1871 // This table defines the kernel's mappings, which are present in
1872 // every process's page table.
1873 static struct kmap {
1874   void *virt;
1875   uint phys_start;
1876   uint phys_end;
1877   int perm;
1878 } kmap[] = {
1879  { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
1880  { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
1881  { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
1882  { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
1883 };
1884
1885 // Set up kernel part of a page table.
1886 pde_t*
1887 setupkvm(void)
1888 {
1889   pde_t *pgdir;
1890   struct kmap *k;
1891
1892   if((pgdir = (pde_t*)kalloc()) == 0)
1893     return 0;
1894   memset(pgdir, 0, PGSIZE);
1895   if (p2v(PHYSTOP) > (void*)DEVSPACE)
1896     panic("PHYSTOP too high");
1897   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1898     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1899                 (uint)k->phys_start, k->perm) < 0)
```

```
1900       return 0;
1901   return pgdir;
1902 }
1903
1904 // Allocate one page table for the machine for the kernel address
1905 // space for scheduler processes.
1906 void
1907 kvmalloc(void)
1908 {
1909   kpgdir = setupkvm();
1910   switchkvm();
1911 }
1912
1913 // Switch h/w page table register to the kernel-only page table,
1914 // for when no process is running.
1915 void
1916 switchkvm(void)
1917 {
1918   lcr3(v2p(kpgdir));   // switch to the kernel page table
1919 }
1920
1921 // Switch TSS and h/w page table to correspond to process p.
1922 void
1923 switchuvm(struct proc *p)
1924 {
1925   pushcli();
1926   cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
1927   cpu->gdt[SEG_TSS].s = 0;
1928   cpu->ts.ss0 = SEG_KDATA << 3;
1929   cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
1930   ltr(SEG_TSS << 3);
1931   if(p->pgdir == 0)
1932     panic("switchuvm: no pgdir");
1933   lcr3(v2p(p->pgdir));  // switch to new address space
1934   popcli();
1935 }
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
```



```
1950 // Load the initcode into address 0 of pgdir.
1951 // sz must be less than a page.
1952 void
1953 inituvm(pde_t *pgdir, char *init, uint sz)
1954 {
1955   char *mem;
1956
1957   if(sz >= PGSIZE)
1958     panic("inituvm: more than a page");
1959   mem = kalloc();
1960   memset(mem, 0, PGSIZE);
1961   mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
1962   memmove(mem, init, sz);
1963 }
1964
1965 // Load a program segment into pgdir.  addr must be page-aligned
1966 // and the pages from addr to addr+sz must already be mapped.
1967 int
1968 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
1969 {
1970   uint i, pa, n;
1971   pte_t *pte;
1972
1973   if((uint) addr % PGSIZE != 0)
1974     panic("loaduvm: addr must be page aligned");
1975   for(i = 0; i < sz; i += PGSIZE){
1976     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1977       panic("loaduvm: address should exist");
1978     pa = PTE_ADDR(*pte);
1979     if(sz - i < PGSIZE)
1980       n = sz - i;
1981     else
1982       n = PGSIZE;
1983     if(readi(ip, p2v(pa), offset+i, n) != n)
1984       return -1;
1985   }
1986   return 0;
1987 }
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
```

```
2000 // Allocate page tables and physical memory to grow process from oldsz to
2001 // newsz, which need not be page aligned.  Returns new size or 0 on error.
2002 int
2003 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
2004 {
2005   char *mem;
2006   uint a;
2007
2008   if(newsz >= KERNBASE)
2009     return 0;
2010   if(newsz < oldsz)
2011     return oldsz;
2012
2013   a = PGROUNDUP(oldsz);
2014   for(; a < newsz; a += PGSIZE){
2015     mem = kalloc();
2016     if(mem == 0){
2017       cprintf("allocuvm out of memory\n");
2018       deallocuvm(pgdir, newsz, oldsz);
2019       return 0;
2020     }
2021     memset(mem, 0, PGSIZE);
2022     mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U);
2023   }
2024   return newsz;
2025 }
2026
2027 // Deallocate user pages to bring the process size from oldsz to
2028 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
2029 // need to be less than oldsz.  oldsz can be larger than the actual
2030 // process size.  Returns the new process size.
2031 int
2032 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
2033 {
2034   pte_t *pte;
2035   uint a, pa;
2036
2037   if(newsz >= oldsz)
2038     return oldsz;
2039
2040   a = PGROUNDUP(newsz);
2041   for(; a  < oldsz; a += PGSIZE){
2042     pte = walkpgdir(pgdir, (char*)a, 0);
2043     if(!pte)
2044       a += (NPTENTRIES - 1) * PGSIZE;
2045     else if((*pte & PTE_P) != 0){
2046       pa = PTE_ADDR(*pte);
2047       if(pa == 0)
2048         panic("kfree");
2049       char *v = p2v(pa);
```



```
2050       kfree(v);
2051       *pte = 0;
2052     }
2053   }
2054   return newsz;
2055 }
2056
2057 // Free a page table and all the physical memory pages
2058 // in the user part.
2059 void
2060 freevm(pde_t *pgdir)
2061 {
2062   uint i;
2063
2064   if(pgdir == 0)
2065     panic("freevm: no pgdir");
2066   deallocuvm(pgdir, KERNBASE, 0);
2067   for(i = 0; i < NPDENTRIES; i++){
2068     if(pgdir[i] & PTE_P){
2069       char * v = p2v(PTE_ADDR(pgdir[i]));
2070       kfree(v);
2071     }
2072   }
2073   kfree((char*)pgdir);
2074 }
2075
2076 // Clear PTE_U on a page. Used to create an inaccessible
2077 // page beneath the user stack.
2078 void
2079 clearpteu(pde_t *pgdir, char *uva)
2080 {
2081   pte_t *pte;
2082
2083   pte = walkpgdir(pgdir, uva, 0);
2084   if(pte == 0)
2085     panic("clearpteu");
2086   *pte &= ~PTE_U;
2087 }
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
```

```
2100 // Given a parent process's page table, create a copy
2101 // of it for a child.
2102 pde_t*
2103 copyuvm(pde_t *pgdir, uint sz)
2104 {
2105   pde_t *d;
2106   pte_t *pte;
2107   uint pa, i, flags;
2108   char *mem;
2109
2110   if((d = setupkvm()) == 0)
2111     return 0;
2112   for(i = 0; i < sz; i += PGSIZE){
2113     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
2114       panic("copyuvm: pte should exist");
2115     if(!(*pte & PTE_P))
2116       panic("copyuvm: page not present");
2117     pa = PTE_ADDR(*pte);
2118     flags = PTE_FLAGS(*pte);
2119     if((mem = kalloc()) == 0)
2120       goto bad;
2121     memmove(mem, (char*)p2v(pa), PGSIZE);
2122     if(mappages(d, (void*)i, PGSIZE, v2p(mem), flags) < 0)
2123       goto bad;
2124   }
2125   return d;
2126
2127 bad:
2128   freevm(d);
2129   return 0;
2130 }
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
```



```
2150 // Map user virtual address to kernel address.
2151 char*
2152 uva2ka(pde_t *pgdir, char *uva)
2153 {
2154   pte_t *pte;
2155
2156   pte = walkpgdir(pgdir, uva, 0);
2157   if((*pte & PTE_P) == 0)
2158     return 0;
2159   if((*pte & PTE_U) == 0)
2160     return 0;
2161   return (char*)p2v(PTE_ADDR(*pte));
2162 }
2163
2164 // Copy len bytes from p to user address va in page table pgdir.
2165 // Most useful when pgdir is not the current page table.
2166 // uva2ka ensures this only works for PTE_U pages.
2167 int
2168 copyout(pde_t *pgdir, uint va, void *p, uint len)
2169 {
2170   char *buf, *pa0;
2171   uint n, va0;
2172
2173   buf = (char*)p;
2174   while(len > 0){
2175     va0 = (uint)PGROUNDDOWN(va);
2176     pa0 = uva2ka(pgdir, (char*)va0);
2177     if(pa0 == 0)
2178       return -1;
2179     n = PGSIZE - (va - va0);
2180     if(n > len)
2181       n = len;
2182     memmove(pa0 + (va - va0), buf, n);
2183     len -= n;
2184     buf += n;
2185     va = va0 + PGSIZE;
2186   }
2187   return 0;
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 // Blank page.
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
```



```
2350 // Segments in proc->gdt.
2351 #define NSEGS     7
2352
2353 // Per-CPU state
2354 struct cpu {
2355   uchar id;                    // Local APIC ID; index into cpus[] below
2356   struct context *scheduler;   // swtch() here to enter scheduler
2357   struct taskstate ts;         // Used by x86 to find stack for interrupt
2358   struct segdesc gdt[NSEGS];   // x86 global descriptor table
2359   volatile uint started;       // Has the CPU started?
2360   int ncli;                    // Depth of pushcli nesting.
2361   int intena;                  // Were interrupts enabled before pushcli?
2362
2363   // Cpu-local storage variables; see below
2364   struct cpu *cpu;
2365   struct proc *proc;           // The currently-running process.
2366 };
2367
2368 extern struct cpu cpus[NCPU];
2369 extern int ncpu;
2370
2371 // Per-CPU variables, holding pointers to the
2372 // current cpu and to the current process.
2373 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2374 // and "%gs:4" to refer to proc.  seginit sets up the
2375 // %gs segment register so that %gs refers to the memory
2376 // holding those two variables in the local cpu's struct cpu.
2377 // This is similar to how thread-local variables are implemented
2378 // in thread libraries such as Linux pthreads.
2379 extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
2380 extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc
2381
2382
2383 // Saved registers for kernel context switches.
2384 // Don't need to save all the segment registers (%cs, etc),
2385 // because they are constant across kernel contexts.
2386 // Don't need to save %eax, %ecx, %edx, because the
2387 // x86 convention is that the caller has saved them.
2388 // Contexts are stored at the bottom of the stack they
2389 // describe; the stack pointer is the address of the context.
2390 // The layout of the context matches the layout of the stack in swtch.S
2391 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2392 // but it is on the stack and allocproc() manipulates it.
2393 struct context {
2394   uint edi;
2395   uint esi;
2396   uint ebx;
2397   uint ebp;
2398   uint eip;
2399 };
```

```
2400 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2401
2402 // Per-process state
2403 struct proc {
2404   uint sz;                     // Size of process memory (bytes)
2405   pde_t* pgdir;                // Page table
2406   char *kstack;                // Bottom of kernel stack for this process
2407   enum procstate state;        // Process state
2408   uint pid;                    // Process ID
2409   struct proc *parent;         // Parent process
2410   struct trapframe *tf;        // Trap frame for current syscall
2411   struct context *context;     // swtch() here to run process
2412   void *chan;                  // If non-zero, sleeping on chan
2413   int killed;                  // If non-zero, have been killed
2414   struct file *ofile[NOFILE];  // Open files
2415   struct inode *cwd;           // Current directory
2416   char name[16];               // Process name (debugging)
2417 };
2418
2419 // Process memory is laid out contiguously, low addresses first:
2420 //   text
2421 //   original data and bss
2422 //   fixed-size stack
2423 //   expandable heap
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
```



```
2450 #include "types.h"
2451 #include "defs.h"
2452 #include "param.h"
2453 #include "memlayout.h"
2454 #include "mmu.h"
2455 #include "x86.h"
2456 #include "proc.h"
2457 #include "spinlock.h"
2458
2459 struct {
2460   struct spinlock lock;
2461   struct proc proc[NPROC];
2462 } ptable;
2463
2464 static struct proc *initproc;
2465
2466 int nextpid = 1;
2467 extern void forkret(void);
2468 extern void trapret(void);
2469
2470 static void wakeup1(void *chan);
2471
2472 void
2473 pinit(void)
2474 {
2475   initlock(&ptable.lock, "ptable");
2476 }
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
```

```
2500 // Look in the process table for an UNUSED proc.
2501 // If found, change state to EMBRYO and initialize
2502 // state required to run in the kernel.
2503 // Otherwise return 0.
2504 static struct proc*
2505 allocproc(void)
2506 {
2507   struct proc *p;
2508   char *sp;
2509
2510   acquire(&ptable.lock);
2511   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2512     if(p->state == UNUSED)
2513       goto found;
2514   release(&ptable.lock);
2515   return 0;
2516
2517 found:
2518   p->state = EMBRYO;
2519   p->pid = nextpid++;
2520   release(&ptable.lock);
2521
2522   // Allocate kernel stack.
2523   if((p->kstack = kalloc()) == 0){
2524     p->state = UNUSED;
2525     return 0;
2526   }
2527   sp = p->kstack + KSTACKSIZE;
2528
2529   // Leave room for trap frame.
2530   sp -= sizeof *p->tf;
2531   p->tf = (struct trapframe*)sp;
2532
2533   // Set up new context to start executing at forkret,
2534   // which returns to trapret.
2535   sp -= 4;
2536   *(uint*)sp = (uint)trapret;
2537
2538   sp -= sizeof *p->context;
2539   p->context = (struct context*)sp;
2540   memset(p->context, 0, sizeof *p->context);
2541   p->context->eip = (uint)forkret;
2542
2543   return p;
2544 }
2545
2546
2547
2548
2549
```



```
2550 // Set up first user process.
2551 void
2552 userinit(void)
2553 {
2554   struct proc *p;
2555   extern char _binary_initcode_start[], _binary_initcode_size[];
2556
2557   p = allocproc();
2558   initproc = p;
2559   if((p->pgdir = setupkvm()) == 0)
2560     panic("userinit: out of memory?");
2561   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2562   p->sz = PGSIZE;
2563   memset(p->tf, 0, sizeof(*p->tf));
2564   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2565   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2566   p->tf->es = p->tf->ds;
2567   p->tf->ss = p->tf->ds;
2568   p->tf->eflags = FL_IF;
2569   p->tf->esp = PGSIZE;
2570   p->tf->eip = 0;  // beginning of initcode.S
2571
2572   safestrcpy(p->name, "initcode", sizeof(p->name));
2573   p->cwd = namei("/");
2574
2575   p->state = RUNNABLE;
2576 }
2577
2578 // Grow current process's memory by n bytes.
2579 // Return 0 on success, -1 on failure.
2580 int
2581 growproc(int n)
2582 {
2583   uint sz;
2584
2585   sz = proc->sz;
2586   if(n > 0){
2587     if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2588       return -1;
2589   } else if(n < 0){
2590     if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2591       return -1;
2592   }
2593   proc->sz = sz;
2594   switchuvm(proc);
2595   return 0;
2596 }
2597
2598
2599
```

```
2600 // Create a new process copying p as the parent.
2601 // Sets up stack to return as if from system call.
2602 // Caller must set state of returned proc to RUNNABLE.
2603 int
2604 fork(void)
2605 {
2606   int i, pid;
2607   struct proc *np;
2608
2609   // Allocate process.
2610   if((np = allocproc()) == 0)
2611     return -1;
2612
2613   // Copy process state from p.
2614   if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2615     kfree(np->kstack);
2616     np->kstack = 0;
2617     np->state = UNUSED;
2618     return -1;
2619   }
2620   np->sz = proc->sz;
2621   np->parent = proc;
2622   *np->tf = *proc->tf;
2623
2624   // Clear %eax so that fork returns 0 in the child.
2625   np->tf->eax = 0;
2626
2627   for(i = 0; i < NOFILE; i++)
2628     if(proc->ofile[i])
2629       np->ofile[i] = filedup(proc->ofile[i]);
2630   np->cwd = idup(proc->cwd);
2631
2632   safestrcpy(np->name, proc->name, sizeof(proc->name));
2633
2634   pid = np->pid;
2635
2636   // lock to force the compiler to emit the np->state write last.
2637   acquire(&ptable.lock);
2638   np->state = RUNNABLE;
2639   release(&ptable.lock);
2640
2641   return pid;
2642 }
2643
2644
2645
2646
2647
2648
2649
```



```
2650 // Exit the current process.  Does not return.
2651 // An exited process remains in the zombie state
2652 // until its parent calls wait() to find out it exited.
2653 void
2654 exit(void)
2655 {
2656   struct proc *p;
2657   int fd;
2658
2659   if(proc == initproc)
2660     panic("init exiting");
2661
2662   // Close all open files.
2663   for(fd = 0; fd < NOFILE; fd++){
2664     if(proc->ofile[fd]){
2665       fileclose(proc->ofile[fd]);
2666       proc->ofile[fd] = 0;
2667     }
2668   }
2669
2670   begin_op();
2671   iput(proc->cwd);
2672   end_op();
2673   proc->cwd = 0;
2674
2675   acquire(&ptable.lock);
2676
2677   // Parent might be sleeping in wait().
2678   wakeup1(proc->parent);
2679
2680   // Pass abandoned children to init.
2681   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2682     if(p->parent == proc){
2683       p->parent = initproc;
2684       if(p->state == ZOMBIE)
2685         wakeup1(initproc);
2686     }
2687   }
2688
2689   // Jump into the scheduler, never to return.
2690   proc->state = ZOMBIE;
2691   sched();
2692   panic("zombie exit");
2693 }
2694
2695
2696
2697
2698
2699
```

```
2700 // Wait for a child process to exit and return its pid.
2701 // Return -1 if this process has no children.
2702 int
2703 wait(void)
2704 {
2705   struct proc *p;
2706   int havekids, pid;
2707
2708   acquire(&ptable.lock);
2709   for(;;){
2710     // Scan through table looking for zombie children.
2711     havekids = 0;
2712     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2713       if(p->parent != proc)
2714         continue;
2715       havekids = 1;
2716       if(p->state == ZOMBIE){
2717         // Found one.
2718         pid = p->pid;
2719         kfree(p->kstack);
2720         p->kstack = 0;
2721         freevm(p->pgdir);
2722         p->state = UNUSED;
2723         p->pid = 0;
2724         p->parent = 0;
2725         p->name[0] = 0;
2726         p->killed = 0;
2727         release(&ptable.lock);
2728         return pid;
2729       }
2730     }
2731
2732     // No point waiting if we don't have any children.
2733     if(!havekids || proc->killed){
2734       release(&ptable.lock);
2735       return -1;
2736     }
2737
2738     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2739     sleep(proc, &ptable.lock);
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 #ifndef CS333_P3
2758 // original xv6 scheduler. Use if CS333_P3 NOT defined.
2759 void
2760 scheduler(void)
2761 {
2762   struct proc *p;
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       proc = p;
2778       switchuvm(p);
2779       p->state = RUNNING;
2780       swtch(&cpu->scheduler, proc->context);
2781       switchkvm();
2782
2783       // Process is done running for now.
2784       // It should have changed its p->state before coming back.
2785       proc = 0;
2786     }
2787     release(&ptable.lock);
2788
2789   }
2790 }
2791
2792 #else
2793 // CS333_P3 MLFQ scheduler implementation goes here
2794 void
2795 scheduler(void)
2796 {
2797
2798 }
2799
```

```
2800 #endif
2801
2802 // Enter scheduler.  Must hold only ptable.lock
2803 // and have changed proc->state.
2804 void
2805 sched(void)
2806 {
2807   int intena;
2808
2809   if(!holding(&ptable.lock))
2810     panic("sched ptable.lock");
2811   if(cpu->ncli != 1)
2812     panic("sched locks");
2813   if(proc->state == RUNNING)
2814     panic("sched running");
2815   if(readeflags()&FL_IF)
2816     panic("sched interruptible");
2817   intena = cpu->intena;
2818   swtch(&proc->context, cpu->scheduler);
2819   cpu->intena = intena;
2820 }
2821
2822 // Give up the CPU for one scheduling round.
2823 void
2824 yield(void)
2825 {
2826   acquire(&ptable.lock);
2827   proc->state = RUNNABLE;
2828   sched();
2829   release(&ptable.lock);
2830 }
2831
2832 // A fork child's very first scheduling by scheduler()
2833 // will swtch here.  "Return" to user space.
2834 void
2835 forkret(void)
2836 {
2837   static int first = 1;
2838   // Still holding ptable.lock from scheduler.
2839   release(&ptable.lock);
2840
2841   if (first) {
2842     // Some initialization functions must be run in the context
2843     // of a regular process (e.g., they call sleep), and thus cannot
2844     // be run from main().
2845     first = 0;
2846     iinit(ROOTDEV);
2847     initlog(ROOTDEV);
2848   }
2849
```



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

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



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

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```



```
// Physical memory allocator, intended to allocate
// memory for user processes, kernel stacks, page table pages,
// and pipe buffers. Allocates 4096-byte pages.

#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "spinlock.h"

void freerange(void *vstart, void *vend);
extern char end[]; // first address after kernel loaded from ELF file

struct run {
  struct run *next;
};

struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
} kmem;

// Initialization happens in two phases.
// 1. main() calls kinit1() while still using entrypgdir to place just
// the pages mapped by entrypgdir on free list.
// 2. main() calls kinit2() with the rest of the physical pages
// after installing a full page table that maps them on all cores.
void
kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;
  freerange(vstart, vend);
}

void
kinit2(void *vstart, void *vend)
{
  freerange(vstart, vend);
  kmem.use_lock = 1;
}
```

```
3100 void
3101 freerange(void *vstart, void *vend)
3102 {
3103   char *p;
3104   p = (char*)PGROUNDUP((uint)vstart);
3105   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3106     kfree(p);
3107 }
3108
3109
3110 // Free the page of physical memory pointed at by v,
3111 // which normally should have been returned by a
3112 // call to kalloc().  (The exception is when
3113 // initializing the allocator; see kinit above.)
3114 void
3115 kfree(char *v)
3116 {
3117   struct run *r;
3118
3119   if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
3120     panic("kfree");
3121
3122   // Fill with junk to catch dangling refs.
3123   memset(v, 1, PGSIZE);
3124
3125   if(kmem.use_lock)
3126     acquire(&kmem.lock);
3127   r = (struct run*)v;
3128   r->next = kmem.freelist;
3129   kmem.freelist = r;
3130   if(kmem.use_lock)
3131     release(&kmem.lock);
3132 }
3133
3134 // Allocate one 4096-byte page of physical memory.
3135 // Returns a pointer that the kernel can use.
3136 // Returns 0 if the memory cannot be allocated.
3137 char*
3138 kalloc(void)
3139 {
3140   struct run *r;
3141
3142   if(kmem.use_lock)
3143     acquire(&kmem.lock);
3144   r = kmem.freelist;
3145   if(r)
3146     kmem.freelist = r->next;
3147   if(kmem.use_lock)
3148     release(&kmem.lock);
3149   return (char*)r;
```



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

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



```
3250 #!/usr/bin/perl -w
3251
3252 # Generate vectors.S, the trap/interrupt entry points.
3253 # There has to be one entry point per interrupt number
3254 # since otherwise there's no way for trap() to discover
3255 # the interrupt number.
3256
3257 print "# generated by vectors.pl - do not edit\n";
3258 print "# handlers\n";
3259 print ".globl alltraps\n";
3260 for(my $i = 0; $i < 256; $i++){
3261     print ".globl vector$i\n";
3262     print "vector$i:\n";
3263     if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
3264         print "  pushl \$0\n";
3265     }
3266     print "  pushl \$$i\n";
3267     print "  jmp alltraps\n";
3268 }
3269
3270 print "\n# vector table\n";
3271 print ".data\n";
3272 print ".globl vectors\n";
3273 print "vectors:\n";
3274 for(my $i = 0; $i < 256; $i++){
3275     print "  .long vector$i\n";
3276 }
3277
3278 # sample output:
3279 #   # handlers
3280 #   .globl alltraps
3281 #   .globl vector0
3282 #   vector0:
3283 #     pushl $0
3284 #     pushl $0
3285 #     jmp alltraps
3286 #   ...
3287 #
3288 #   # vector table
3289 #   .data
3290 #   .globl vectors
3291 #   vectors:
3292 #     .long vector0
3293 #     .long vector1
3294 #     .long vector2
3295 #   ...
3296
3297
3298
3299
```

```
3300 #include "mmu.h"
3301
3302   # vectors.S sends all traps here.
3303 .globl alltraps
3304 alltraps:
3305   # Build trap frame.
3306   pushl %ds
3307   pushl %es
3308   pushl %fs
3309   pushl %gs
3310   pushal
3311
3312   # Set up data and per-cpu segments.
3313   movw $(SEG_KDATA<<3), %ax
3314   movw %ax, %ds
3315   movw %ax, %es
3316   movw $(SEG_KCPU<<3), %ax
3317   movw %ax, %fs
3318   movw %ax, %gs
3319
3320   # Call trap(tf), where tf=%esp
3321   pushl %esp
3322   call trap
3323   addl $4, %esp
3324
3325   # Return falls through to trapret...
3326 .globl trapret
3327 trapret:
3328   popal
3329   popl %gs
3330   popl %fs
3331   popl %es
3332   popl %ds
3333   addl $0x8, %esp  # trapno and errcode
3334   iret
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
```



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

```
3400 void
3401 trap(struct trapframe *tf)
3402 {
3403   if(tf->trapno == T_SYSCALL){
3404     if(proc->killed)
3405       exit();
3406     proc->tf = tf;
3407     syscall();
3408     if(proc->killed)
3409       exit();
3410     return;
3411   }
3412
3413   switch(tf->trapno){
3414   case T_IRQ0 + IRQ_TIMER:
3415     if(cpu->id == 0){
3416       acquire(&tickslock);
3417       ticks++;
3418       release(&tickslock);    // NOTE: MarkM has reversed these two lines.
3419       wakeup(&ticks);         // wakeup() should not require the tickslock t
3420     }
3421     lapiceoi();
3422     break;
3423   case T_IRQ0 + IRQ_IDE:
3424     ideintr();
3425     lapiceoi();
3426     break;
3427   case T_IRQ0 + IRQ_IDE+1:
3428     // Bochs generates spurious IDE1 interrupts.
3429     break;
3430   case T_IRQ0 + IRQ_KBD:
3431     kbdintr();
3432     lapiceoi();
3433     break;
3434   case T_IRQ0 + IRQ_COM1:
3435     uartintr();
3436     lapiceoi();
3437     break;
3438   case T_IRQ0 + 7:
3439   case T_IRQ0 + IRQ_SPURIOUS:
3440     cprintf("cpu%d: spurious interrupt at %x:%x\n",
3441             cpu->id, tf->cs, tf->eip);
3442     lapiceoi();
3443     break;
3444
3445
3446
3447
3448
3449
```



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

```
3500 // System call numbers
3501 #define SYS_fork    1
3502 #define SYS_exit    SYS_fork+1
3503 #define SYS_wait    SYS_exit+1
3504 #define SYS_pipe    SYS_wait+1
3505 #define SYS_read    SYS_pipe+1
3506 #define SYS_kill    SYS_read+1
3507 #define SYS_exec    SYS_kill+1
3508 #define SYS_fstat   SYS_exec+1
3509 #define SYS_chdir   SYS_fstat+1
3510 #define SYS_dup     SYS_chdir+1
3511 #define SYS_getpid  SYS_dup+1
3512 #define SYS_sbrk    SYS_getpid+1
3513 #define SYS_sleep   SYS_sbrk+1
3514 #define SYS_uptime  SYS_sleep+1
3515 #define SYS_open    SYS_uptime+1
3516 #define SYS_write   SYS_open+1
3517 #define SYS_mknod   SYS_write+1
3518 #define SYS_unlink  SYS_mknod+1
3519 #define SYS_link    SYS_unlink+1
3520 #define SYS_mkdir   SYS_link+1
3521 #define SYS_close   SYS_mkdir+1
3522 #define SYS_halt    SYS_close+1
3523 // student system calls begin here. Follow the existing pattern.
3524
3525
3526
3527
3528
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
```



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

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



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

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



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

```
3800 //Turn of the computer
3801 int sys_halt(void){
3802   cprintf("Shutting down ...\n");
3803   outw (0xB004, 0x0 | 0x2000);
3804   return 0;
3805 }
3806
3807
3808
3809
3810
3811
3812
3813
3814
3815
3816
3817
3818
3819
3820
3821
3822
3823
3824
3825
3826
3827
3828
3829
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
```



```
3850 // halt the system.
3851 #include "types.h"
3852 #include "user.h"
3853
3854 int
3855 main(void) {
3856   halt();
3857   return 0;
3858 }
3859
3860
3861
3862
3863
3864
3865
3866
3867
3868
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
```

```
3900 struct buf {
3901   int flags;
3902   uint dev;
3903   uint blockno;
3904   struct buf *prev; // LRU cache list
3905   struct buf *next;
3906   struct buf *qnext; // disk queue
3907   uchar data[BSIZE];
3908 };
3909 #define B_BUSY  0x1  // buffer is locked by some process
3910 #define B_VALID 0x2  // buffer has been read from disk
3911 #define B_DIRTY 0x4  // buffer needs to be written to disk
3912
3913
3914
3915
3916
3917
3918
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
```



```
3950 #define O_RDONLY  0x000
3951 #define O_WRONLY  0x001
3952 #define O_RDWR    0x002
3953 #define O_CREATE  0x200
3954
3955
3956
3957
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 T_DIR  1   // Directory
4001 #define T_FILE 2   // File
4002 #define T_DEV  3   // Device
4003
4004 struct stat {
4005   short type;  // Type of file
4006   int dev;     // File system's disk device
4007   uint ino;    // Inode number
4008   short nlink; // Number of links to file
4009   uint size;   // Size of file in bytes
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 // On-disk file system format.
4051 // Both the kernel and user programs use this header file.
4052
4053
4054 #define ROOTINO 1  // root i-number
4055 #define BSIZE 512  // block size
4056
4057 // Disk layout:
4058 // [ boot block | super block | log | inode blocks | free bit map | data bloc
4059 //
4060 // mkfs computes the super block and builds an initial file system. The super
4061 // the disk layout:
4062 struct superblock {
4063   uint size;         // Size of file system image (blocks)
4064   uint nblocks;      // Number of data blocks
4065   uint ninodes;      // Number of inodes.
4066   uint nlog;         // Number of log blocks
4067   uint logstart;     // Block number of first log block
4068   uint inodestart;   // Block number of first inode block
4069   uint bmapstart;    // Block number of first free map block
4070 };
4071
4072 #define NDIRECT 12
4073 #define NINDIRECT (BSIZE / sizeof(uint))
4074 #define MAXFILE (NDIRECT + NINDIRECT)
4075
4076 // On-disk inode structure
4077 struct dinode {
4078   short type;           // File type
4079   short major;          // Major device number (T_DEV only)
4080   short minor;          // Minor device number (T_DEV only)
4081   short nlink;          // Number of links to inode in file system
4082   uint size;            // Size of file (bytes)
4083   uint addrs[NDIRECT+1];   // Data block addresses
4084 };
4085
4086
4087
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
```

```
4100 // Inodes per block.
4101 #define IPB           (BSIZE / sizeof(struct dinode))
4102
4103 // Block containing inode i
4104 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
4105
4106 // Bitmap bits per block
4107 #define BPB           (BSIZE*8)
4108
4109 // Block of free map containing bit for block b
4110 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
4111
4112 // Directory is a file containing a sequence of dirent structures.
4113 #define DIRSIZ 14
4114
4115 struct dirent {
4116   ushort inum;
4117   char name[DIRSIZ];
4118 };
4119
4120
4121
4122
4123
4124
4125
4126
4127
4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```



```
4150 struct file {
4151   enum { FD_NONE, FD_PIPE, FD_INODE } type;
4152   int ref; // reference count
4153   char readable;
4154   char writable;
4155   struct pipe *pipe;
4156   struct inode *ip;
4157   uint off;
4158 };
4159
4160
4161 // in-memory copy of an inode
4162 struct inode {
4163   uint dev;           // Device number
4164   uint inum;          // Inode number
4165   int ref;            // Reference count
4166   int flags;          // I_BUSY, I_VALID
4167
4168   short type;         // copy of disk inode
4169   short major;
4170   short minor;
4171   short nlink;
4172   uint size;
4173   uint addrs[NDIRECT+1];
4174 };
4175 #define I_BUSY 0x1
4176 #define I_VALID 0x2
4177
4178 // table mapping major device number to
4179 // device functions
4180 struct devsw {
4181   int (*read)(struct inode*, char*, int);
4182   int (*write)(struct inode*, char*, int);
4183 };
4184
4185 extern struct devsw devsw[];
4186
4187 #define CONSOLE 1
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
```

```
4200 // Blank page.
4201
4202
4203
4204
4205
4206
4207
4208
4209
4210
4211
4212
4213
4214
4215
4216
4217
4218
4219
4220
4221
4222
4223
4224
4225
4226
4227
4228
4229
4230
4231
4232
4233
4234
4235
4236
4237
4238
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
```



```
4250 // Simple PIO-based (non-DMA) IDE driver code.
4251
4252 #include "types.h"
4253 #include "defs.h"
4254 #include "param.h"
4255 #include "memlayout.h"
4256 #include "mmu.h"
4257 #include "proc.h"
4258 #include "x86.h"
4259 #include "traps.h"
4260 #include "spinlock.h"
4261 #include "fs.h"
4262 #include "buf.h"
4263
4264 #define SECTOR_SIZE   512
4265 #define IDE_BSY       0x80
4266 #define IDE_DRDY      0x40
4267 #define IDE_DF        0x20
4268 #define IDE_ERR       0x01
4269
4270 #define IDE_CMD_READ  0x20
4271 #define IDE_CMD_WRITE 0x30
4272
4273 // idequeue points to the buf now being read/written to the disk.
4274 // idequeue->qnext points to the next buf to be processed.
4275 // You must hold idelock while manipulating queue.
4276
4277 static struct spinlock idelock;
4278 static struct buf *idequeue;
4279
4280 static int havedisk1;
4281 static void idestart(struct buf*);
4282
4283 // Wait for IDE disk to become ready.
4284 static int
4285 idewait(int checkerr)
4286 {
4287   int r;
4288
4289   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4290     ;
4291   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4292     return -1;
4293   return 0;
4294 }
4295
4296
4297
4298
4299
```

```
4300 void
4301 ideinit(void)
4302 {
4303   int i;
4304
4305   initlock(&idelock, "ide");
4306   picenable(IRQ_IDE);
4307   ioapicenable(IRQ_IDE, ncpu - 1);
4308   idewait(0);
4309
4310   // Check if disk 1 is present
4311   outb(0x1f6, 0xe0 | (1<<4));
4312   for(i=0; i<1000; i++){
4313     if(inb(0x1f7) != 0){
4314       havedisk1 = 1;
4315       break;
4316     }
4317   }
4318
4319   // Switch back to disk 0.
4320   outb(0x1f6, 0xe0 | (0<<4));
4321 }
4322
4323 // Start the request for b.  Caller must hold idelock.
4324 static void
4325 idestart(struct buf *b)
4326 {
4327   if(b == 0)
4328     panic("idestart");
4329   if(b->blockno >= FSSIZE)
4330     panic("incorrect blockno");
4331   int sector_per_block =  BSIZE/SECTOR_SIZE;
4332   int sector = b->blockno * sector_per_block;
4333
4334   if (sector_per_block > 7) panic("idestart");
4335
4336   idewait(0);
4337   outb(0x3f6, 0);  // generate interrupt
4338   outb(0x1f2, sector_per_block);  // number of sectors
4339   outb(0x1f3, sector & 0xff);
4340   outb(0x1f4, (sector >> 8) & 0xff);
4341   outb(0x1f5, (sector >> 16) & 0xff);
4342   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4343   if(b->flags & B_DIRTY){
4344     outb(0x1f7, IDE_CMD_WRITE);
4345     outsl(0x1f0, b->data, BSIZE/4);
4346   } else {
4347     outb(0x1f7, IDE_CMD_READ);
4348   }
4349 }
```



```
4350 // Interrupt handler.
4351 void
4352 ideintr(void)
4353 {
4354   struct buf *b;
4355
4356   // First queued buffer is the active request.
4357   acquire(&idelock);
4358   if((b = idequeue) == 0){
4359     release(&idelock);
4360     // cprintf("spurious IDE interrupt\n");
4361     return;
4362   }
4363   idequeue = b->qnext;
4364
4365   // Read data if needed.
4366   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4367     insl(0x1f0, b->data, BSIZE/4);
4368
4369   // Wake process waiting for this buf.
4370   b->flags |= B_VALID;
4371   b->flags &= ~B_DIRTY;
4372   wakeup(b);
4373
4374   // Start disk on next buf in queue.
4375   if(idequeue != 0)
4376     idestart(idequeue);
4377
4378   release(&idelock);
4379 }
4380
4381
4382
4383
4384
4385
4386
4387
4388
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
```

```
4400 // Sync buf with disk.
4401 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
4402 // Else if B_VALID is not set, read buf from disk, set B_VALID.
4403 void
4404 iderw(struct buf *b)
4405 {
4406   struct buf **pp;
4407
4408   if(!(b->flags & B_BUSY))
4409     panic("iderw: buf not busy");
4410   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
4411     panic("iderw: nothing to do");
4412   if(b->dev != 0 && !havedisk1)
4413     panic("iderw: ide disk 1 not present");
4414
4415   acquire(&idelock);
4416
4417   // Append b to idequeue.
4418   b->qnext = 0;
4419   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
4420     ;
4421   *pp = b;
4422
4423   // Start disk if necessary.
4424   if(idequeue == b)
4425     idestart(b);
4426
4427   // Wait for request to finish.
4428   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
4429     sleep(b, &idelock);
4430   }
4431
4432   release(&idelock);
4433 }
4434
4435
4436
4437
4438
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
```



```
4450 // Buffer cache.
4451 //
4452 // The buffer cache is a linked list of buf structures holding
4453 // cached copies of disk block contents.  Caching disk blocks
4454 // in memory reduces the number of disk reads and also provides
4455 // a synchronization point for disk blocks used by multiple processes.
4456 //
4457 // Interface:
4458 // * To get a buffer for a particular disk block, call bread.
4459 // * After changing buffer data, call bwrite to write it to disk.
4460 // * When done with the buffer, call brelse.
4461 // * Do not use the buffer after calling brelse.
4462 // * Only one process at a time can use a buffer,
4463 //     so do not keep them longer than necessary.
4464 //
4465 // The implementation uses three state flags internally:
4466 // * B_BUSY: the block has been returned from bread
4467 //     and has not been passed back to brelse.
4468 // * B_VALID: the buffer data has been read from the disk.
4469 // * B_DIRTY: the buffer data has been modified
4470 //     and needs to be written to disk.
4471
4472 #include "types.h"
4473 #include "defs.h"
4474 #include "param.h"
4475 #include "spinlock.h"
4476 #include "fs.h"
4477 #include "buf.h"
4478
4479 struct {
4480   struct spinlock lock;
4481   struct buf buf[NBUF];
4482
4483   // Linked list of all buffers, through prev/next.
4484   // head.next is most recently used.
4485   struct buf head;
4486 } bcache;
4487
4488 void
4489 binit(void)
4490 {
4491   struct buf *b;
4492
4493   initlock(&bcache.lock, "bcache");
4494
4495
4496
4497
4498
4499
```

```
4500   // Create linked list of buffers
4501   bcache.head.prev = &bcache.head;
4502   bcache.head.next = &bcache.head;
4503   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4504     b->next = bcache.head.next;
4505     b->prev = &bcache.head;
4506     b->dev = -1;
4507     bcache.head.next->prev = b;
4508     bcache.head.next = b;
4509   }
4510 }
4511
4512 // Look through buffer cache for block on device dev.
4513 // If not found, allocate a buffer.
4514 // In either case, return B_BUSY buffer.
4515 static struct buf*
4516 bget(uint dev, uint blockno)
4517 {
4518   struct buf *b;
4519
4520   acquire(&bcache.lock);
4521
4522  loop:
4523   // Is the block already cached?
4524   for(b = bcache.head.next; b != &bcache.head; b = b->next){
4525     if(b->dev == dev && b->blockno == blockno){
4526       if(!(b->flags & B_BUSY)){
4527         b->flags |= B_BUSY;
4528         release(&bcache.lock);
4529         return b;
4530       }
4531       sleep(b, &bcache.lock);
4532       goto loop;
4533     }
4534   }
4535
4536   // Not cached; recycle some non-busy and clean buffer.
4537   // "clean" because B_DIRTY and !B_BUSY means log.c
4538   // hasn't yet committed the changes to the buffer.
4539   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4540     if((b->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
4541       b->dev = dev;
4542       b->blockno = blockno;
4543       b->flags = B_BUSY;
4544       release(&bcache.lock);
4545       return b;
4546     }
4547   }
4548   panic("bget: no buffers");
4549 }
```



```
4550 // Return a B_BUSY buf with the contents of the indicated block.
4551 struct buf*
4552 bread(uint dev, uint blockno)
4553 {
4554   struct buf *b;
4555
4556   b = bget(dev, blockno);
4557   if(!(b->flags & B_VALID)) {
4558     iderw(b);
4559   }
4560   return b;
4561 }
4562
4563 // Write b's contents to disk.  Must be B_BUSY.
4564 void
4565 bwrite(struct buf *b)
4566 {
4567   if((b->flags & B_BUSY) == 0)
4568     panic("bwrite");
4569   b->flags |= B_DIRTY;
4570   iderw(b);
4571 }
4572
4573 // Release a B_BUSY buffer.
4574 // Move to the head of the MRU list.
4575 void
4576 brelse(struct buf *b)
4577 {
4578   if((b->flags & B_BUSY) == 0)
4579     panic("brelse");
4580
4581   acquire(&bcache.lock);
4582
4583   b->next->prev = b->prev;
4584   b->prev->next = b->next;
4585   b->next = bcache.head.next;
4586   b->prev = &bcache.head;
4587   bcache.head.next->prev = b;
4588   bcache.head.next = b;
4589
4590   b->flags &= ~B_BUSY;
4591   wakeup(b);
4592
4593   release(&bcache.lock);
4594 }
4595
4596
4597
4598
4599
```

```
4600 // Blank page.
4601
4602
4603
4604
4605
4606
4607
4608
4609
4610
4611
4612
4613
4614
4615
4616
4617
4618
4619
4620
4621
4622
4623
4624
4625
4626
4627
4628
4629
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649
```



```
4650 #include "types.h"
4651 #include "defs.h"
4652 #include "param.h"
4653 #include "spinlock.h"
4654 #include "fs.h"
4655 #include "buf.h"
4656
4657 // Simple logging that allows concurrent FS system calls.
4658 //
4659 // A log transaction contains the updates of multiple FS system
4660 // calls. The logging system only commits when there are
4661 // no FS system calls active. Thus there is never
4662 // any reasoning required about whether a commit might
4663 // write an uncommitted system call's updates to disk.
4664 //
4665 // A system call should call begin_op()/end_op() to mark
4666 // its start and end. Usually begin_op() just increments
4667 // the count of in-progress FS system calls and returns.
4668 // But if it thinks the log is close to running out, it
4669 // sleeps until the last outstanding end_op() commits.
4670 //
4671 // The log is a physical re-do log containing disk blocks.
4672 // The on-disk log format:
4673 //   header block, containing block #s for block A, B, C, ...
4674 //   block A
4675 //   block B
4676 //   block C
4677 //   ...
4678 // Log appends are synchronous.
4679
4680 // Contents of the header block, used for both the on-disk header block
4681 // and to keep track in memory of logged block# before commit.
4682 struct logheader {
4683   int n;
4684   int block[LOGSIZE];
4685 };
4686
4687 struct log {
4688   struct spinlock lock;
4689   int start;
4690   int size;
4691   int outstanding; // how many FS sys calls are executing.
4692   int committing;  // in commit(), please wait.
4693   int dev;
4694   struct logheader lh;
4695 };
4696
4697
4698
4699
```

```
4700 struct log log;
4701
4702 static void recover_from_log(void);
4703 static void commit();
4704
4705 void
4706 initlog(int dev)
4707 {
4708   if (sizeof(struct logheader) >= BSIZE)
4709     panic("initlog: too big logheader");
4710
4711   struct superblock sb;
4712   initlock(&log.lock, "log");
4713   readsb(dev, &sb);
4714   log.start = sb.logstart;
4715   log.size = sb.nlog;
4716   log.dev = dev;
4717   recover_from_log();
4718 }
4719
4720 // Copy committed blocks from log to their home location
4721 static void
4722 install_trans(void)
4723 {
4724   int tail;
4725
4726   for (tail = 0; tail < log.lh.n; tail++) {
4727     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
4728     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
4729     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
4730     bwrite(dbuf);  // write dst to disk
4731     brelse(lbuf);
4732     brelse(dbuf);
4733   }
4734 }
4735
4736 // Read the log header from disk into the in-memory log header
4737 static void
4738 read_head(void)
4739 {
4740   struct buf *buf = bread(log.dev, log.start);
4741   struct logheader *lh = (struct logheader *) (buf->data);
4742   int i;
4743   log.lh.n = lh->n;
4744   for (i = 0; i < log.lh.n; i++) {
4745     log.lh.block[i] = lh->block[i];
4746   }
4747   brelse(buf);
4748 }
4749
```



```
4750 // Write in-memory log header to disk.
4751 // This is the true point at which the
4752 // current transaction commits.
4753 static void
4754 write_head(void)
4755 {
4756   struct buf *buf = bread(log.dev, log.start);
4757   struct logheader *hb = (struct logheader *) (buf->data);
4758   int i;
4759   hb->n = log.lh.n;
4760   for (i = 0; i < log.lh.n; i++) {
4761     hb->block[i] = log.lh.block[i];
4762   }
4763   bwrite(buf);
4764   brelse(buf);
4765 }
4766
4767 static void
4768 recover_from_log(void)
4769 {
4770   read_head();
4771   install_trans(); // if committed, copy from log to disk
4772   log.lh.n = 0;
4773   write_head(); // clear the log
4774 }
4775
4776 // called at the start of each FS system call.
4777 void
4778 begin_op(void)
4779 {
4780   acquire(&log.lock);
4781   while(1){
4782     if(log.committing){
4783       sleep(&log, &log.lock);
4784     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
4785       // this op might exhaust log space; wait for commit.
4786       sleep(&log, &log.lock);
4787     } else {
4788       log.outstanding += 1;
4789       release(&log.lock);
4790       break;
4791     }
4792   }
4793 }
4794
4795
4796
4797
4798
4799
```

```
4800 // called at the end of each FS system call.
4801 // commits if this was the last outstanding operation.
4802 void
4803 end_op(void)
4804 {
4805   int do_commit = 0;
4806
4807   acquire(&log.lock);
4808   log.outstanding -= 1;
4809   if(log.committing)
4810     panic("log.committing");
4811   if(log.outstanding == 0){
4812     do_commit = 1;
4813     log.committing = 1;
4814   } else {
4815     // begin_op() may be waiting for log space.
4816     wakeup(&log);
4817   }
4818   release(&log.lock);
4819
4820   if(do_commit){
4821     // call commit w/o holding locks, since not allowed
4822     // to sleep with locks.
4823     commit();
4824     acquire(&log.lock);
4825     log.committing = 0;
4826     wakeup(&log);
4827     release(&log.lock);
4828   }
4829 }
4830
4831 // Copy modified blocks from cache to log.
4832 static void
4833 write_log(void)
4834 {
4835   int tail;
4836
4837   for (tail = 0; tail < log.lh.n; tail++) {
4838     struct buf *to = bread(log.dev, log.start+tail+1); // log block
4839     struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
4840     memmove(to->data, from->data, BSIZE);
4841     bwrite(to);  // write the log
4842     brelse(from);
4843     brelse(to);
4844   }
4845 }
4846
4847
4848
4849
```



```
4850 static void
4851 commit()
4852 {
4853   if (log.lh.n > 0) {
4854     write_log();     // Write modified blocks from cache to log
4855     write_head();    // Write header to disk -- the real commit
4856     install_trans(); // Now install writes to home locations
4857     log.lh.n = 0;
4858     write_head();    // Erase the transaction from the log
4859   }
4860 }
4861
4862 // Caller has modified b->data and is done with the buffer.
4863 // Record the block number and pin in the cache with B_DIRTY.
4864 // commit()/write_log() will do the disk write.
4865 //
4866 // log_write() replaces bwrite(); a typical use is:
4867 //   bp = bread(...)
4868 //   modify bp->data[]
4869 //   log_write(bp)
4870 //   brelse(bp)
4871 void
4872 log_write(struct buf *b)
4873 {
4874   int i;
4875
4876   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
4877     panic("too big a transaction");
4878   if (log.outstanding < 1)
4879     panic("log_write outside of trans");
4880
4881   acquire(&log.lock);
4882   for (i = 0; i < log.lh.n; i++) {
4883     if (log.lh.block[i] == b->blockno)   // log absorbtion
4884       break;
4885   }
4886   log.lh.block[i] = b->blockno;
4887   if (i == log.lh.n)
4888     log.lh.n++;
4889   b->flags |= B_DIRTY; // prevent eviction
4890   release(&log.lock);
4891 }
4892
4893
4894
4895
4896
4897
4898
4899
```

```
4900 // File system implementation.  Five layers:
4901 //   + Blocks: allocator for raw disk blocks.
4902 //   + Log: crash recovery for multi-step updates.
4903 //   + Files: inode allocator, reading, writing, metadata.
4904 //   + Directories: inode with special contents (list of other inodes!)
4905 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
4906 //
4907 // This file contains the low-level file system manipulation
4908 // routines.  The (higher-level) system call implementations
4909 // are in sysfile.c.
4910
4911 #include "types.h"
4912 #include "defs.h"
4913 #include "param.h"
4914 #include "stat.h"
4915 #include "mmu.h"
4916 #include "proc.h"
4917 #include "spinlock.h"
4918 #include "fs.h"
4919 #include "buf.h"
4920 #include "file.h"
4921
4922 #define min(a, b) ((a) < (b) ? (a) : (b))
4923 static void itrunc(struct inode*);
4924 struct superblock sb;   // there should be one per dev, but we run with one d
4925
4926 // Read the super block.
4927 void
4928 readsb(int dev, struct superblock *sb)
4929 {
4930   struct buf *bp;
4931
4932   bp = bread(dev, 1);
4933   memmove(sb, bp->data, sizeof(*sb));
4934   brelse(bp);
4935 }
4936
4937 // Zero a block.
4938 static void
4939 bzero(int dev, int bno)
4940 {
4941   struct buf *bp;
4942
4943   bp = bread(dev, bno);
4944   memset(bp->data, 0, BSIZE);
4945   log_write(bp);
4946   brelse(bp);
4947 }
4948
4949
```



```
4950 // Blocks.
4951
4952 // Allocate a zeroed disk block.
4953 static uint
4954 balloc(uint dev)
4955 {
4956   int b, bi, m;
4957   struct buf *bp;
4958
4959   bp = 0;
4960   for(b = 0; b < sb.size; b += BPB){
4961     bp = bread(dev, BBLOCK(b, sb));
4962     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
4963       m = 1 << (bi % 8);
4964       if((bp->data[bi/8] & m) == 0){  // Is block free?
4965         bp->data[bi/8] |= m;  // Mark block in use.
4966         log_write(bp);
4967         brelse(bp);
4968         bzero(dev, b + bi);
4969         return b + bi;
4970       }
4971     }
4972     brelse(bp);
4973   }
4974   panic("balloc: out of blocks");
4975 }
4976
4977 // Free a disk block.
4978 static void
4979 bfree(int dev, uint b)
4980 {
4981   struct buf *bp;
4982   int bi, m;
4983
4984   readsb(dev, &sb);
4985   bp = bread(dev, BBLOCK(b, sb));
4986   bi = b % BPB;
4987   m = 1 << (bi % 8);
4988   if((bp->data[bi/8] & m) == 0)
4989     panic("freeing free block");
4990   bp->data[bi/8] &= ~m;
4991   log_write(bp);
4992   brelse(bp);
4993 }
4994
4995
4996
4997
4998
4999
```

```
5000 // Inodes.
5001 //
5002 // An inode describes a single unnamed file.
5003 // The inode disk structure holds metadata: the file's type,
5004 // its size, the number of links referring to it, and the
5005 // list of blocks holding the file's content.
5006 //
5007 // The inodes are laid out sequentially on disk at
5008 // sb.startinode. Each inode has a number, indicating its
5009 // position on the disk.
5010 //
5011 // The kernel keeps a cache of in-use inodes in memory
5012 // to provide a place for synchronizing access
5013 // to inodes used by multiple processes. The cached
5014 // inodes include book-keeping information that is
5015 // not stored on disk: ip->ref and ip->flags.
5016 //
5017 // An inode and its in-memory represtative go through a
5018 // sequence of states before they can be used by the
5019 // rest of the file system code.
5020 //
5021 // * Allocation: an inode is allocated if its type (on disk)
5022 //   is non-zero. ialloc() allocates, iput() frees if
5023 //   the link count has fallen to zero.
5024 //
5025 // * Referencing in cache: an entry in the inode cache
5026 //   is free if ip->ref is zero. Otherwise ip->ref tracks
5027 //   the number of in-memory pointers to the entry (open
5028 //   files and current directories). iget() to find or
5029 //   create a cache entry and increment its ref, iput()
5030 //   to decrement ref.
5031 //
5032 // * Valid: the information (type, size, &c) in an inode
5033 //   cache entry is only correct when the I_VALID bit
5034 //   is set in ip->flags. ilock() reads the inode from
5035 //   the disk and sets I_VALID, while iput() clears
5036 //   I_VALID if ip->ref has fallen to zero.
5037 //
5038 // * Locked: file system code may only examine and modify
5039 //   the information in an inode and its content if it
5040 //   has first locked the inode. The I_BUSY flag indicates
5041 //   that the inode is locked. ilock() sets I_BUSY,
5042 //   while iunlock clears it.
5043 //
5044 // Thus a typical sequence is:
5045 //   ip = iget(dev, inum)
5046 //   ilock(ip)
5047 //   ... examine and modify ip->xxx ...
5048 //   iunlock(ip)
5049 //   iput(ip)
```



```
5050 //
5051 // ilock() is separate from iget() so that system calls can
5052 // get a long-term reference to an inode (as for an open file)
5053 // and only lock it for short periods (e.g., in read()).
5054 // The separation also helps avoid deadlock and races during
5055 // pathname lookup. iget() increments ip->ref so that the inode
5056 // stays cached and pointers to it remain valid.
5057 //
5058 // Many internal file system functions expect the caller to
5059 // have locked the inodes involved; this lets callers create
5060 // multi-step atomic operations.
5061
5062 struct {
5063   struct spinlock lock;
5064   struct inode inode[NINODE];
5065 } icache;
5066
5067 void
5068 iinit(int dev)
5069 {
5070   initlock(&icache.lock, "icache");
5071   readsb(dev, &sb);
5072   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d inodestart %
5073           sb.nblocks, sb.ninodes, sb.nlog, sb.logstart, sb.inodestart, sb.bma
5074 }
5075
5076 static struct inode* iget(uint dev, uint inum);
5077
5078
5079
5080
5081
5082
5083
5084
5085
5086
5087
5088
5089
5090
5091
5092
5093
5094
5095
5096
5097
5098
5099
```

```
5100 // Allocate a new inode with the given type on device dev.
5101 // A free inode has a type of zero.
5102 struct inode*
5103 ialloc(uint dev, short type)
5104 {
5105   int inum;
5106   struct buf *bp;
5107   struct dinode *dip;
5108
5109   for(inum = 1; inum < sb.ninodes; inum++){
5110     bp = bread(dev, IBLOCK(inum, sb));
5111     dip = (struct dinode*)bp->data + inum%IPB;
5112     if(dip->type == 0){  // a free inode
5113       memset(dip, 0, sizeof(*dip));
5114       dip->type = type;
5115       log_write(bp);   // mark it allocated on the disk
5116       brelse(bp);
5117       return iget(dev, inum);
5118     }
5119     brelse(bp);
5120   }
5121   panic("ialloc: no inodes");
5122 }
5123
5124 // Copy a modified in-memory inode to disk.
5125 void
5126 iupdate(struct inode *ip)
5127 {
5128   struct buf *bp;
5129   struct dinode *dip;
5130
5131   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5132   dip = (struct dinode*)bp->data + ip->inum%IPB;
5133   dip->type = ip->type;
5134   dip->major = ip->major;
5135   dip->minor = ip->minor;
5136   dip->nlink = ip->nlink;
5137   dip->size = ip->size;
5138   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5139   log_write(bp);
5140   brelse(bp);
5141 }
5142
5143
5144
5145
5146
5147
5148
5149
```



```
5150 // Find the inode with number inum on device dev
5151 // and return the in-memory copy. Does not lock
5152 // the inode and does not read it from disk.
5153 static struct inode*
5154 iget(uint dev, uint inum)
5155 {
5156   struct inode *ip, *empty;
5157
5158   acquire(&icache.lock);
5159
5160   // Is the inode already cached?
5161   empty = 0;
5162   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5163     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5164       ip->ref++;
5165       release(&icache.lock);
5166       return ip;
5167     }
5168     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
5169       empty = ip;
5170   }
5171
5172   // Recycle an inode cache entry.
5173   if(empty == 0)
5174     panic("iget: no inodes");
5175
5176   ip = empty;
5177   ip->dev = dev;
5178   ip->inum = inum;
5179   ip->ref = 1;
5180   ip->flags = 0;
5181   release(&icache.lock);
5182
5183   return ip;
5184 }
5185
5186 // Increment reference count for ip.
5187 // Returns ip to enable ip = idup(ip1) idiom.
5188 struct inode*
5189 idup(struct inode *ip)
5190 {
5191   acquire(&icache.lock);
5192   ip->ref++;
5193   release(&icache.lock);
5194   return ip;
5195 }
5196
5197
5198
5199
```

```
5200 // Lock the given inode.
5201 // Reads the inode from disk if necessary.
5202 void
5203 ilock(struct inode *ip)
5204 {
5205   struct buf *bp;
5206   struct dinode *dip;
5207
5208   if(ip == 0 || ip->ref < 1)
5209     panic("ilock");
5210
5211   acquire(&icache.lock);
5212   while(ip->flags & I_BUSY)
5213     sleep(ip, &icache.lock);
5214   ip->flags |= I_BUSY;
5215   release(&icache.lock);
5216
5217   if(!(ip->flags & I_VALID)){
5218     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5219     dip = (struct dinode*)bp->data + ip->inum%IPB;
5220     ip->type = dip->type;
5221     ip->major = dip->major;
5222     ip->minor = dip->minor;
5223     ip->nlink = dip->nlink;
5224     ip->size = dip->size;
5225     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5226     brelse(bp);
5227     ip->flags |= I_VALID;
5228     if(ip->type == 0)
5229       panic("ilock: no type");
5230   }
5231 }
5232
5233 // Unlock the given inode.
5234 void
5235 iunlock(struct inode *ip)
5236 {
5237   if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
5238     panic("iunlock");
5239
5240   acquire(&icache.lock);
5241   ip->flags &= ~I_BUSY;
5242   wakeup(ip);
5243   release(&icache.lock);
5244 }
5245
5246
5247
5248
5249
```



```
5250 // Drop a reference to an in-memory inode.
5251 // If that was the last reference, the inode cache entry can
5252 // be recycled.
5253 // If that was the last reference and the inode has no links
5254 // to it, free the inode (and its content) on disk.
5255 // All calls to iput() must be inside a transaction in
5256 // case it has to free the inode.
5257 void
5258 iput(struct inode *ip)
5259 {
5260   acquire(&icache.lock);
5261   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
5262     // inode has no links and no other references: truncate and free.
5263     if(ip->flags & I_BUSY)
5264       panic("iput busy");
5265     ip->flags |= I_BUSY;
5266     release(&icache.lock);
5267     itrunc(ip);
5268     ip->type = 0;
5269     iupdate(ip);
5270     acquire(&icache.lock);
5271     ip->flags = 0;
5272     wakeup(ip);
5273   }
5274   ip->ref--;
5275   release(&icache.lock);
5276 }
5277
5278 // Common idiom: unlock, then put.
5279 void
5280 iunlockput(struct inode *ip)
5281 {
5282   iunlock(ip);
5283   iput(ip);
5284 }
5285
5286
5287
5288
5289
5290
5291
5292
5293
5294
5295
5296
5297
5298
5299
```

```
5300 // Inode content
5301 //
5302 // The content (data) associated with each inode is stored
5303 // in blocks on the disk. The first NDIRECT block numbers
5304 // are listed in ip->addrs[].  The next NINDIRECT blocks are
5305 // listed in block ip->addrs[NDIRECT].
5306
5307 // Return the disk block address of the nth block in inode ip.
5308 // If there is no such block, bmap allocates one.
5309 static uint
5310 bmap(struct inode *ip, uint bn)
5311 {
5312   uint addr, *a;
5313   struct buf *bp;
5314
5315   if(bn < NDIRECT){
5316     if((addr = ip->addrs[bn]) == 0)
5317       ip->addrs[bn] = addr = balloc(ip->dev);
5318     return addr;
5319   }
5320   bn -= NDIRECT;
5321
5322   if(bn < NINDIRECT){
5323     // Load indirect block, allocating if necessary.
5324     if((addr = ip->addrs[NDIRECT]) == 0)
5325       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5326     bp = bread(ip->dev, addr);
5327     a = (uint*)bp->data;
5328     if((addr = a[bn]) == 0){
5329       a[bn] = addr = balloc(ip->dev);
5330       log_write(bp);
5331     }
5332     brelse(bp);
5333     return addr;
5334   }
5335
5336   panic("bmap: out of range");
5337 }
5338
5339
5340
5341
5342
5343
5344
5345
5346
5347
5348
5349
```



```
5350 // Truncate inode (discard contents).
5351 // Only called when the inode has no links
5352 // to it (no directory entries referring to it)
5353 // and has no in-memory reference to it (is
5354 // not an open file or current directory).
5355 static void
5356 itrunc(struct inode *ip)
5357 {
5358   int i, j;
5359   struct buf *bp;
5360   uint *a;
5361
5362   for(i = 0; i < NDIRECT; i++){
5363     if(ip->addrs[i]){
5364       bfree(ip->dev, ip->addrs[i]);
5365       ip->addrs[i] = 0;
5366     }
5367   }
5368
5369   if(ip->addrs[NDIRECT]){
5370     bp = bread(ip->dev, ip->addrs[NDIRECT]);
5371     a = (uint*)bp->data;
5372     for(j = 0; j < NINDIRECT; j++){
5373       if(a[j])
5374         bfree(ip->dev, a[j]);
5375     }
5376     brelse(bp);
5377     bfree(ip->dev, ip->addrs[NDIRECT]);
5378     ip->addrs[NDIRECT] = 0;
5379   }
5380
5381   ip->size = 0;
5382   iupdate(ip);
5383 }
5384
5385 // Copy stat information from inode.
5386 void
5387 stati(struct inode *ip, struct stat *st)
5388 {
5389   st->dev = ip->dev;
5390   st->ino = ip->inum;
5391   st->type = ip->type;
5392   st->nlink = ip->nlink;
5393   st->size = ip->size;
5394 }
5395
5396
5397
5398
5399
```

```
5400 // Read data from inode.
5401 int
5402 readi(struct inode *ip, char *dst, uint off, uint n)
5403 {
5404   uint tot, m;
5405   struct buf *bp;
5406
5407   if(ip->type == T_DEV){
5408     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
5409       return -1;
5410     return devsw[ip->major].read(ip, dst, n);
5411   }
5412
5413   if(off > ip->size || off + n < off)
5414     return -1;
5415   if(off + n > ip->size)
5416     n = ip->size - off;
5417
5418   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5419     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5420     m = min(n - tot, BSIZE - off%BSIZE);
5421     memmove(dst, bp->data + off%BSIZE, m);
5422     brelse(bp);
5423   }
5424   return n;
5425 }
5426
5427
5428
5429
5430
5431
5432
5433
5434
5435
5436
5437
5438
5439
5440
5441
5442
5443
5444
5445
5446
5447
5448
5449
```



```
5450 // Write data to inode.
5451 int
5452 writei(struct inode *ip, char *src, uint off, uint n)
5453 {
5454   uint tot, m;
5455   struct buf *bp;
5456
5457   if(ip->type == T_DEV){
5458     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
5459       return -1;
5460     return devsw[ip->major].write(ip, src, n);
5461   }
5462
5463   if(off > ip->size || off + n < off)
5464     return -1;
5465   if(off + n > MAXFILE*BSIZE)
5466     return -1;
5467
5468   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
5469     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5470     m = min(n - tot, BSIZE - off%BSIZE);
5471     memmove(bp->data + off%BSIZE, src, m);
5472     log_write(bp);
5473     brelse(bp);
5474   }
5475
5476   if(n > 0 && off > ip->size){
5477     ip->size = off;
5478     iupdate(ip);
5479   }
5480   return n;
5481 }
5482
5483
5484
5485
5486
5487
5488
5489
5490
5491
5492
5493
5494
5495
5496
5497
5498
5499
```

```
5500 // Directories
5501
5502 int
5503 namecmp(const char *s, const char *t)
5504 {
5505   return strncmp(s, t, DIRSIZ);
5506 }
5507
5508 // Look for a directory entry in a directory.
5509 // If found, set *poff to byte offset of entry.
5510 struct inode*
5511 dirlookup(struct inode *dp, char *name, uint *poff)
5512 {
5513   uint off, inum;
5514   struct dirent de;
5515
5516   if(dp->type != T_DIR)
5517     panic("dirlookup not DIR");
5518
5519   for(off = 0; off < dp->size; off += sizeof(de)){
5520     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5521       panic("dirlink read");
5522     if(de.inum == 0)
5523       continue;
5524     if(namecmp(name, de.name) == 0){
5525       // entry matches path element
5526       if(poff)
5527         *poff = off;
5528       inum = de.inum;
5529       return iget(dp->dev, inum);
5530     }
5531   }
5532
5533   return 0;
5534 }
5535
5536
5537
5538
5539
5540
5541
5542
5543
5544
5545
5546
5547
5548
5549
```



```
5550 // Write a new directory entry (name, inum) into the directory dp.
5551 int
5552 dirlink(struct inode *dp, char *name, uint inum)
5553 {
5554   int off;
5555   struct dirent de;
5556   struct inode *ip;
5557
5558   // Check that name is not present.
5559   if((ip = dirlookup(dp, name, 0)) != 0){
5560     iput(ip);
5561     return -1;
5562   }
5563
5564   // Look for an empty dirent.
5565   for(off = 0; off < dp->size; off += sizeof(de)){
5566     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5567       panic("dirlink read");
5568     if(de.inum == 0)
5569       break;
5570   }
5571
5572   strncpy(de.name, name, DIRSIZ);
5573   de.inum = inum;
5574   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5575     panic("dirlink");
5576
5577   return 0;
5578 }
5579
5580
5581
5582
5583
5584
5585
5586
5587
5588
5589
5590
5591
5592
5593
5594
5595
5596
5597
5598
5599
```

```
5600 // Paths
5601
5602 // Copy the next path element from path into name.
5603 // Return a pointer to the element following the copied one.
5604 // The returned path has no leading slashes,
5605 // so the caller can check *path=='\0' to see if the name is the last one.
5606 // If no name to remove, return 0.
5607 //
5608 // Examples:
5609 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5610 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5611 //   skipelem("a", name) = "", setting name = "a"
5612 //   skipelem("", name) = skipelem("////", name) = 0
5613 //
5614 static char*
5615 skipelem(char *path, char *name)
5616 {
5617   char *s;
5618   int len;
5619
5620   while(*path == '/')
5621     path++;
5622   if(*path == 0)
5623     return 0;
5624   s = path;
5625   while(*path != '/' && *path != 0)
5626     path++;
5627   len = path - s;
5628   if(len >= DIRSIZ)
5629     memmove(name, s, DIRSIZ);
5630   else {
5631     memmove(name, s, len);
5632     name[len] = 0;
5633   }
5634   while(*path == '/')
5635     path++;
5636   return path;
5637 }
5638
5639
5640
5641
5642
5643
5644
5645
5646
5647
5648
5649
```



```
5650 // Look up and return the inode for a path name.
5651 // If parent != 0, return the inode for the parent and copy the final
5652 // path element into name, which must have room for DIRSIZ bytes.
5653 // Must be called inside a transaction since it calls iput().
5654 static struct inode*
5655 namex(char *path, int nameiparent, char *name)
5656 {
5657   struct inode *ip, *next;
5658
5659   if(*path == '/')
5660     ip = iget(ROOTDEV, ROOTINO);
5661   else
5662     ip = idup(proc->cwd);
5663
5664   while((path = skipelem(path, name)) != 0){
5665     ilock(ip);
5666     if(ip->type != T_DIR){
5667       iunlockput(ip);
5668       return 0;
5669     }
5670     if(nameiparent && *path == '\0'){
5671       // Stop one level early.
5672       iunlock(ip);
5673       return ip;
5674     }
5675     if((next = dirlookup(ip, name, 0)) == 0){
5676       iunlockput(ip);
5677       return 0;
5678     }
5679     iunlockput(ip);
5680     ip = next;
5681   }
5682   if(nameiparent){
5683     iput(ip);
5684     return 0;
5685   }
5686   return ip;
5687 }
5688
5689 struct inode*
5690 namei(char *path)
5691 {
5692   char name[DIRSIZ];
5693   return namex(path, 0, name);
5694 }
5695
5696
5697
5698
5699
```

```
5700 struct inode*
5701 nameiparent(char *path, char *name)
5702 {
5703   return namex(path, 1, name);
5704 }
5705
5706
5707
5708
5709
5710
5711
5712
5713
5714
5715
5716
5717
5718
5719
5720
5721
5722
5723
5724
5725
5726
5727
5728
5729
5730
5731
5732
5733
5734
5735
5736
5737
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749
```



```
5750 //
5751 // File descriptors
5752 //
5753
5754 #include "types.h"
5755 #include "defs.h"
5756 #include "param.h"
5757 #include "fs.h"
5758 #include "file.h"
5759 #include "spinlock.h"
5760
5761 struct devsw devsw[NDEV];
5762 struct {
5763   struct spinlock lock;
5764   struct file file[NFILE];
5765 } ftable;
5766
5767 void
5768 fileinit(void)
5769 {
5770   initlock(&ftable.lock, "ftable");
5771 }
5772
5773 // Allocate a file structure.
5774 struct file*
5775 filealloc(void)
5776 {
5777   struct file *f;
5778
5779   acquire(&ftable.lock);
5780   for(f = ftable.file; f < ftable.file + NFILE; f++){
5781     if(f->ref == 0){
5782       f->ref = 1;
5783       release(&ftable.lock);
5784       return f;
5785     }
5786   }
5787   release(&ftable.lock);
5788   return 0;
5789 }
5790
5791
5792
5793
5794
5795
5796
5797
5798
5799
```

```
5800 // Increment ref count for file f.
5801 struct file*
5802 filedup(struct file *f)
5803 {
5804   acquire(&ftable.lock);
5805   if(f->ref < 1)
5806     panic("filedup");
5807   f->ref++;
5808   release(&ftable.lock);
5809   return f;
5810 }
5811
5812 // Close file f.  (Decrement ref count, close when reaches 0.)
5813 void
5814 fileclose(struct file *f)
5815 {
5816   struct file ff;
5817
5818   acquire(&ftable.lock);
5819   if(f->ref < 1)
5820     panic("fileclose");
5821   if(--f->ref > 0){
5822     release(&ftable.lock);
5823     return;
5824   }
5825   ff = *f;
5826   f->ref = 0;
5827   f->type = FD_NONE;
5828   release(&ftable.lock);
5829
5830   if(ff.type == FD_PIPE)
5831     pipeclose(ff.pipe, ff.writable);
5832   else if(ff.type == FD_INODE){
5833     begin_op();
5834     iput(ff.ip);
5835     end_op();
5836   }
5837 }
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```



```
5850 // Get metadata about file f.
5851 int
5852 filestat(struct file *f, struct stat *st)
5853 {
5854   if(f->type == FD_INODE){
5855     ilock(f->ip);
5856     stati(f->ip, st);
5857     iunlock(f->ip);
5858     return 0;
5859   }
5860   return -1;
5861 }
5862
5863 // Read from file f.
5864 int
5865 fileread(struct file *f, char *addr, int n)
5866 {
5867   int r;
5868
5869   if(f->readable == 0)
5870     return -1;
5871   if(f->type == FD_PIPE)
5872     return piperead(f->pipe, addr, n);
5873   if(f->type == FD_INODE){
5874     ilock(f->ip);
5875     if((r = readi(f->ip, addr, f->off, n)) > 0)
5876       f->off += r;
5877     iunlock(f->ip);
5878     return r;
5879   }
5880   panic("fileread");
5881 }
5882
5883
5884
5885
5886
5887
5888
5889
5890
5891
5892
5893
5894
5895
5896
5897
5898
5899
```

```
5900 // Write to file f.
5901 int
5902 filewrite(struct file *f, char *addr, int n)
5903 {
5904   int r;
5905
5906   if(f->writable == 0)
5907     return -1;
5908   if(f->type == FD_PIPE)
5909     return pipewrite(f->pipe, addr, n);
5910   if(f->type == FD_INODE){
5911     // write a few blocks at a time to avoid exceeding
5912     // the maximum log transaction size, including
5913     // i-node, indirect block, allocation blocks,
5914     // and 2 blocks of slop for non-aligned writes.
5915     // this really belongs lower down, since writei()
5916     // might be writing a device like the console.
5917     int max = ((LOGSIZE-1-1-2) / 2) * 512;
5918     int i = 0;
5919     while(i < n){
5920       int n1 = n - i;
5921       if(n1 > max)
5922         n1 = max;
5923
5924       begin_op();
5925       ilock(f->ip);
5926       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
5927         f->off += r;
5928       iunlock(f->ip);
5929       end_op();
5930
5931       if(r < 0)
5932         break;
5933       if(r != n1)
5934         panic("short filewrite");
5935       i += r;
5936     }
5937     return i == n ? n : -1;
5938   }
5939   panic("filewrite");
5940 }
5941
5942
5943
5944
5945
5946
5947
5948
5949
```



```
5950 //
5951 // File-system system calls.
5952 // Mostly argument checking, since we don't trust
5953 // user code, and calls into file.c and fs.c.
5954 //
5955
5956 #include "types.h"
5957 #include "defs.h"
5958 #include "param.h"
5959 #include "stat.h"
5960 #include "mmu.h"
5961 #include "proc.h"
5962 #include "fs.h"
5963 #include "file.h"
5964 #include "fcntl.h"
5965
5966 // Fetch the nth word-sized system call argument as a file descriptor
5967 // and return both the descriptor and the corresponding struct file.
5968 static int
5969 argfd(int n, int *pfd, struct file **pf)
5970 {
5971   int fd;
5972   struct file *f;
5973
5974   if(argint(n, &fd) < 0)
5975     return -1;
5976   if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
5977     return -1;
5978   if(pfd)
5979     *pfd = fd;
5980   if(pf)
5981     *pf = f;
5982   return 0;
5983 }
5984
5985 // Allocate a file descriptor for the given file.
5986 // Takes over file reference from caller on success.
5987 static int
5988 fdalloc(struct file *f)
5989 {
5990   int fd;
5991
5992   for(fd = 0; fd < NOFILE; fd++){
5993     if(proc->ofile[fd] == 0){
5994       proc->ofile[fd] = f;
5995       return fd;
5996     }
5997   }
5998   return -1;
5999 }
```

```
6000 int
6001 sys_dup(void)
6002 {
6003   struct file *f;
6004   int fd;
6005
6006   if(argfd(0, 0, &f) < 0)
6007     return -1;
6008   if((fd=fdalloc(f)) < 0)
6009     return -1;
6010   filedup(f);
6011   return fd;
6012 }
6013
6014 int
6015 sys_read(void)
6016 {
6017   struct file *f;
6018   int n;
6019   char *p;
6020
6021   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6022     return -1;
6023   return fileread(f, p, n);
6024 }
6025
6026 int
6027 sys_write(void)
6028 {
6029   struct file *f;
6030   int n;
6031   char *p;
6032
6033   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6034     return -1;
6035   return filewrite(f, p, n);
6036 }
6037
6038 int
6039 sys_close(void)
6040 {
6041   int fd;
6042   struct file *f;
6043
6044   if(argfd(0, &fd, &f) < 0)
6045     return -1;
6046   proc->ofile[fd] = 0;
6047   fileclose(f);
6048   return 0;
6049 }
```



```
6050 int
6051 sys_fstat(void)
6052 {
6053   struct file *f;
6054   struct stat *st;
6055
6056   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
6057     return -1;
6058   return filestat(f, st);
6059 }
6060
6061 // Create the path new as a link to the same inode as old.
6062 int
6063 sys_link(void)
6064 {
6065   char name[DIRSIZ], *new, *old;
6066   struct inode *dp, *ip;
6067
6068   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
6069     return -1;
6070
6071   begin_op();
6072   if((ip = namei(old)) == 0){
6073     end_op();
6074     return -1;
6075   }
6076
6077   ilock(ip);
6078   if(ip->type == T_DIR){
6079     iunlockput(ip);
6080     end_op();
6081     return -1;
6082   }
6083
6084   ip->nlink++;
6085   iupdate(ip);
6086   iunlock(ip);
6087
6088   if((dp = nameiparent(new, name)) == 0)
6089     goto bad;
6090   ilock(dp);
6091   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
6092     iunlockput(dp);
6093     goto bad;
6094   }
6095   iunlockput(dp);
6096   iput(ip);
6097
6098   end_op();
6099
```

```
6100   return 0;
6101
6102 bad:
6103   ilock(ip);
6104   ip->nlink--;
6105   iupdate(ip);
6106   iunlockput(ip);
6107   end_op();
6108   return -1;
6109 }
6110
6111 // Is the directory dp empty except for "." and ".." ?
6112 static int
6113 isdirempty(struct inode *dp)
6114 {
6115   int off;
6116   struct dirent de;
6117
6118   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
6119     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6120       panic("isdirempty: readi");
6121     if(de.inum != 0)
6122       return 0;
6123   }
6124   return 1;
6125 }
6126
6127
6128
6129
6130
6131
6132
6133
6134
6135
6136
6137
6138
6139
6140
6141
6142
6143
6144
6145
6146
6147
6148
6149
```



```
6150 int
6151 sys_unlink(void)
6152 {
6153   struct inode *ip, *dp;
6154   struct dirent de;
6155   char name[DIRSIZ], *path;
6156   uint off;
6157
6158   if(argstr(0, &path) < 0)
6159     return -1;
6160
6161   begin_op();
6162   if((dp = nameiparent(path, name)) == 0){
6163     end_op();
6164     return -1;
6165   }
6166
6167   ilock(dp);
6168
6169   // Cannot unlink "." or "..".
6170   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6171     goto bad;
6172
6173   if((ip = dirlookup(dp, name, &off)) == 0)
6174     goto bad;
6175   ilock(ip);
6176
6177   if(ip->nlink < 1)
6178     panic("unlink: nlink < 1");
6179   if(ip->type == T_DIR && !isdirempty(ip)){
6180     iunlockput(ip);
6181     goto bad;
6182   }
6183
6184   memset(&de, 0, sizeof(de));
6185   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6186     panic("unlink: writei");
6187   if(ip->type == T_DIR){
6188     dp->nlink--;
6189     iupdate(dp);
6190   }
6191   iunlockput(dp);
6192
6193   ip->nlink--;
6194   iupdate(ip);
6195   iunlockput(ip);
6196
6197   end_op();
6198
6199   return 0;
```

```
6200 bad:
6201   iunlockput(dp);
6202   end_op();
6203   return -1;
6204 }
6205
6206 static struct inode*
6207 create(char *path, short type, short major, short minor)
6208 {
6209   uint off;
6210   struct inode *ip, *dp;
6211   char name[DIRSIZ];
6212
6213   if((dp = nameiparent(path, name)) == 0)
6214     return 0;
6215   ilock(dp);
6216
6217   if((ip = dirlookup(dp, name, &off)) != 0){
6218     iunlockput(dp);
6219     ilock(ip);
6220     if(type == T_FILE && ip->type == T_FILE)
6221       return ip;
6222     iunlockput(ip);
6223     return 0;
6224   }
6225
6226   if((ip = ialloc(dp->dev, type)) == 0)
6227     panic("create: ialloc");
6228
6229   ilock(ip);
6230   ip->major = major;
6231   ip->minor = minor;
6232   ip->nlink = 1;
6233   iupdate(ip);
6234
6235   if(type == T_DIR){  // Create . and .. entries.
6236     dp->nlink++;  // for ".."
6237     iupdate(dp);
6238     // No ip->nlink++ for ".": avoid cyclic ref count.
6239     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6240       panic("create dots");
6241   }
6242
6243   if(dirlink(dp, name, ip->inum) < 0)
6244     panic("create: dirlink");
6245
6246   iunlockput(dp);
6247
6248   return ip;
6249 }
```



```
6250 int
6251 sys_open(void)
6252 {
6253   char *path;
6254   int fd, omode;
6255   struct file *f;
6256   struct inode *ip;
6257
6258   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6259     return -1;
6260
6261   begin_op();
6262
6263   if(omode & O_CREATE){
6264     ip = create(path, T_FILE, 0, 0);
6265     if(ip == 0){
6266       end_op();
6267       return -1;
6268     }
6269   } else {
6270     if((ip = namei(path)) == 0){
6271       end_op();
6272       return -1;
6273     }
6274     ilock(ip);
6275     if(ip->type == T_DIR && omode != O_RDONLY){
6276       iunlockput(ip);
6277       end_op();
6278       return -1;
6279     }
6280   }
6281
6282   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6283     if(f)
6284       fileclose(f);
6285     iunlockput(ip);
6286     end_op();
6287     return -1;
6288   }
6289   iunlock(ip);
6290   end_op();
6291
6292   f->type = FD_INODE;
6293   f->ip = ip;
6294   f->off = 0;
6295   f->readable = !(omode & O_WRONLY);
6296   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6297   return fd;
6298 }
6299
```

```
6300 int
6301 sys_mkdir(void)
6302 {
6303   char *path;
6304   struct inode *ip;
6305
6306   begin_op();
6307   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
6308     end_op();
6309     return -1;
6310   }
6311   iunlockput(ip);
6312   end_op();
6313   return 0;
6314 }
6315
6316 int
6317 sys_mknod(void)
6318 {
6319   struct inode *ip;
6320   char *path;
6321   int len;
6322   int major, minor;
6323
6324   begin_op();
6325   if((len=argstr(0, &path)) < 0 ||
6326      argint(1, &major) < 0 ||
6327      argint(2, &minor) < 0 ||
6328      (ip = create(path, T_DEV, major, minor)) == 0){
6329     end_op();
6330     return -1;
6331   }
6332   iunlockput(ip);
6333   end_op();
6334   return 0;
6335 }
6336
6337
6338
6339
6340
6341
6342
6343
6344
6345
6346
6347
6348
6349
```



```
6350 int
6351 sys_chdir(void)
6352 {
6353   char *path;
6354   struct inode *ip;
6355
6356   begin_op();
6357   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
6358     end_op();
6359     return -1;
6360   }
6361   ilock(ip);
6362   if(ip->type != T_DIR){
6363     iunlockput(ip);
6364     end_op();
6365     return -1;
6366   }
6367   iunlock(ip);
6368   iput(proc->cwd);
6369   end_op();
6370   proc->cwd = ip;
6371   return 0;
6372 }
6373
6374 int
6375 sys_exec(void)
6376 {
6377   char *path, *argv[MAXARG];
6378   int i;
6379   uint uargv, uarg;
6380
6381   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6382     return -1;
6383   }
6384   memset(argv, 0, sizeof(argv));
6385   for(i=0;; i++){
6386     if(i >= NELEM(argv))
6387       return -1;
6388     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6389       return -1;
6390     if(uarg == 0){
6391       argv[i] = 0;
6392       break;
6393     }
6394     if(fetchstr(uarg, &argv[i]) < 0)
6395       return -1;
6396   }
6397   return exec(path, argv);
6398 }
6399
```

```
6400 int
6401 sys_pipe(void)
6402 {
6403   int *fd;
6404   struct file *rf, *wf;
6405   int fd0, fd1;
6406
6407   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6408     return -1;
6409   if(pipealloc(&rf, &wf) < 0)
6410     return -1;
6411   fd0 = -1;
6412   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6413     if(fd0 >= 0)
6414       proc->ofile[fd0] = 0;
6415     fileclose(rf);
6416     fileclose(wf);
6417     return -1;
6418   }
6419   fd[0] = fd0;
6420   fd[1] = fd1;
6421   return 0;
6422 }
6423
6424
6425
6426
6427
6428
6429
6430
6431
6432
6433
6434
6435
6436
6437
6438
6439
6440
6441
6442
6443
6444
6445
6446
6447
6448
6449
```



```
6450 #include "types.h"
6451 #include "param.h"
6452 #include "memlayout.h"
6453 #include "mmu.h"
6454 #include "proc.h"
6455 #include "defs.h"
6456 #include "x86.h"
6457 #include "elf.h"
6458
6459 int
6460 exec(char *path, char **argv)
6461 {
6462   char *s, *last;
6463   int i, off;
6464   uint argc, sz, sp, ustack[3+MAXARG+1];
6465   struct elfhdr elf;
6466   struct inode *ip;
6467   struct proghdr ph;
6468   pde_t *pgdir, *oldpgdir;
6469
6470   begin_op();
6471   if((ip = namei(path)) == 0){
6472     end_op();
6473     return -1;
6474   }
6475   ilock(ip);
6476   pgdir = 0;
6477
6478   // Check ELF header
6479   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
6480     goto bad;
6481   if(elf.magic != ELF_MAGIC)
6482     goto bad;
6483
6484   if((pgdir = setupkvm()) == 0)
6485     goto bad;
6486
6487   // Load program into memory.
6488   sz = 0;
6489   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6490     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6491       goto bad;
6492     if(ph.type != ELF_PROG_LOAD)
6493       continue;
6494     if(ph.memsz < ph.filesz)
6495       goto bad;
6496     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6497       goto bad;
6498     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6499       goto bad;
```

```
6500   }
6501   iunlockput(ip);
6502   end_op();
6503   ip = 0;
6504
6505   // Allocate two pages at the next page boundary.
6506   // Make the first inaccessible.  Use the second as the user stack.
6507   sz = PGROUNDUP(sz);
6508   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
6509     goto bad;
6510   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6511   sp = sz;
6512
6513   // Push argument strings, prepare rest of stack in ustack.
6514   for(argc = 0; argv[argc]; argc++) {
6515     if(argc >= MAXARG)
6516       goto bad;
6517     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6518     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6519       goto bad;
6520     ustack[3+argc] = sp;
6521   }
6522   ustack[3+argc] = 0;
6523
6524   ustack[0] = 0xffffffff;  // fake return PC
6525   ustack[1] = argc;
6526   ustack[2] = sp - (argc+1)*4;  // argv pointer
6527
6528   sp -= (3+argc+1) * 4;
6529   if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6530     goto bad;
6531
6532   // Save program name for debugging.
6533   for(last=s=path; *s; s++)
6534     if(*s == '/')
6535       last = s+1;
6536   safestrcpy(proc->name, last, sizeof(proc->name));
6537
6538   // Commit to the user image.
6539   oldpgdir = proc->pgdir;
6540   proc->pgdir = pgdir;
6541   proc->sz = sz;
6542   proc->tf->eip = elf.entry;  // main
6543   proc->tf->esp = sp;
6544   switchuvm(proc);
6545   freevm(oldpgdir);
6546   return 0;
6547
6548
6549
```



```
6550  bad:
6551    if(pgdir)
6552      freevm(pgdir);
6553    if(ip){
6554      iunlockput(ip);
6555      end_op();
6556    }
6557    return -1;
6558  }
6559
6560
6561
6562
6563
6564
6565
6566
6567
6568
6569
6570
6571
6572
6573
6574
6575
6576
6577
6578
6579
6580
6581
6582
6583
6584
6585
6586
6587
6588
6589
6590
6591
6592
6593
6594
6595
6596
6597
6598
6599
```

```
6600 #include "types.h"
6601 #include "defs.h"
6602 #include "param.h"
6603 #include "mmu.h"
6604 #include "proc.h"
6605 #include "fs.h"
6606 #include "file.h"
6607 #include "spinlock.h"
6608
6609 #define PIPESIZE 512
6610
6611 struct pipe {
6612   struct spinlock lock;
6613   char data[PIPESIZE];
6614   uint nread;     // number of bytes read
6615   uint nwrite;    // number of bytes written
6616   int readopen;   // read fd is still open
6617   int writeopen;  // write fd is still open
6618 };
6619
6620 int
6621 pipealloc(struct file **f0, struct file **f1)
6622 {
6623   struct pipe *p;
6624
6625   p = 0;
6626   *f0 = *f1 = 0;
6627   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
6628     goto bad;
6629   if((p = (struct pipe*)kalloc()) == 0)
6630     goto bad;
6631   p->readopen = 1;
6632   p->writeopen = 1;
6633   p->nwrite = 0;
6634   p->nread = 0;
6635   initlock(&p->lock, "pipe");
6636   (*f0)->type = FD_PIPE;
6637   (*f0)->readable = 1;
6638   (*f0)->writable = 0;
6639   (*f0)->pipe = p;
6640   (*f1)->type = FD_PIPE;
6641   (*f1)->readable = 0;
6642   (*f1)->writable = 1;
6643   (*f1)->pipe = p;
6644   return 0;
6645
6646
6647
6648
6649
```



```
6650  bad:
6651   if(p)
6652     kfree((char*)p);
6653   if(*f0)
6654     fileclose(*f0);
6655   if(*f1)
6656     fileclose(*f1);
6657   return -1;
6658 }
6659
6660 void
6661 pipeclose(struct pipe *p, int writable)
6662 {
6663   acquire(&p->lock);
6664   if(writable){
6665     p->writeopen = 0;
6666     wakeup(&p->nread);
6667   } else {
6668     p->readopen = 0;
6669     wakeup(&p->nwrite);
6670   }
6671   if(p->readopen == 0 && p->writeopen == 0){
6672     release(&p->lock);
6673     kfree((char*)p);
6674   } else
6675     release(&p->lock);
6676 }
6677
6678
6679 int
6680 pipewrite(struct pipe *p, char *addr, int n)
6681 {
6682   int i;
6683
6684   acquire(&p->lock);
6685   for(i = 0; i < n; i++){
6686     while(p->nwrite == p->nread + PIPESIZE){
6687       if(p->readopen == 0 || proc->killed){
6688         release(&p->lock);
6689         return -1;
6690       }
6691       wakeup(&p->nread);
6692       sleep(&p->nwrite, &p->lock);
6693     }
6694     p->data[p->nwrite++ % PIPESIZE] = addr[i];
6695   }
6696   wakeup(&p->nread);
6697   release(&p->lock);
6698   return n;
6699 }
```

```
6700 int
6701 piperead(struct pipe *p, char *addr, int n)
6702 {
6703   int i;
6704
6705   acquire(&p->lock);
6706   while(p->nread == p->nwrite && p->writeopen){
6707     if(proc->killed){
6708       release(&p->lock);
6709       return -1;
6710     }
6711     sleep(&p->nread, &p->lock);
6712   }
6713   for(i = 0; i < n; i++){
6714     if(p->nread == p->nwrite)
6715       break;
6716     addr[i] = p->data[p->nread++ % PIPESIZE];
6717   }
6718   wakeup(&p->nwrite);
6719   release(&p->lock);
6720   return i;
6721 }
6722
6723
6724
6725
6726
6727
6728
6729
6730
6731
6732
6733
6734
6735
6736
6737
6738
6739
6740
6741
6742
6743
6744
6745
6746
6747
6748
6749
```



```
6750 #include "types.h"
6751 #include "x86.h"
6752
6753 void*
6754 memset(void *dst, int c, uint n)
6755 {
6756   if ((int)dst%4 == 0 && n%4 == 0){
6757     c &= 0xFF;
6758     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
6759   } else
6760     stosb(dst, c, n);
6761   return dst;
6762 }
6763
6764 int
6765 memcmp(const void *v1, const void *v2, uint n)
6766 {
6767   const uchar *s1, *s2;
6768
6769   s1 = v1;
6770   s2 = v2;
6771   while(n-- > 0){
6772     if(*s1 != *s2)
6773       return *s1 - *s2;
6774     s1++, s2++;
6775   }
6776
6777   return 0;
6778 }
6779
6780 void*
6781 memmove(void *dst, const void *src, uint n)
6782 {
6783   const char *s;
6784   char *d;
6785
6786   s = src;
6787   d = dst;
6788   if(s < d && s + n > d){
6789     s += n;
6790     d += n;
6791     while(n-- > 0)
6792       *--d = *--s;
6793   } else
6794     while(n-- > 0)
6795       *d++ = *s++;
6796
6797   return dst;
6798 }
6799
```

```
6800 // memcpy exists to placate GCC.  Use memmove.
6801 void*
6802 memcpy(void *dst, const void *src, uint n)
6803 {
6804   return memmove(dst, src, n);
6805 }
6806
6807 int
6808 strncmp(const char *p, const char *q, uint n)
6809 {
6810   while(n > 0 && *p && *p == *q)
6811     n--, p++, q++;
6812   if(n == 0)
6813     return 0;
6814   return (uchar)*p - (uchar)*q;
6815 }
6816
6817 char*
6818 strncpy(char *s, const char *t, int n)
6819 {
6820   char *os;
6821
6822   os = s;
6823   while(n-- > 0 && (*s++ = *t++) != 0)
6824     ;
6825   while(n-- > 0)
6826     *s++ = 0;
6827   return os;
6828 }
6829
6830 // Like strncpy but guaranteed to NUL-terminate.
6831 char*
6832 safestrcpy(char *s, const char *t, int n)
6833 {
6834   char *os;
6835
6836   os = s;
6837   if(n <= 0)
6838     return os;
6839   while(--n > 0 && (*s++ = *t++) != 0)
6840     ;
6841   *s = 0;
6842   return os;
6843 }
6844
6845
6846
6847
6848
6849
```



```
6850 int
6851 strlen(const char *s)
6852 {
6853   int n;
6854
6855   for(n = 0; s[n]; n++)
6856     ;
6857   return n;
6858 }
6859
6860
6861
6862
6863
6864
6865
6866
6867
6868
6869
6870
6871
6872
6873
6874
6875
6876
6877
6878
6879
6880
6881
6882
6883
6884
6885
6886
6887
6888
6889
6890
6891
6892
6893
6894
6895
6896
6897
6898
6899
```

```
6900 // See MultiProcessor Specification Version 1.[14]
6901
6902 struct mp {             // floating pointer
6903   uchar signature[4];           // "_MP_"
6904   void *physaddr;               // phys addr of MP config table
6905   uchar length;                 // 1
6906   uchar specrev;                // [14]
6907   uchar checksum;               // all bytes must add up to 0
6908   uchar type;                   // MP system config type
6909   uchar imcrp;
6910   uchar reserved[3];
6911 };
6912
6913 struct mpconf {         // configuration table header
6914   uchar signature[4];           // "PCMP"
6915   ushort length;                // total table length
6916   uchar version;                // [14]
6917   uchar checksum;               // all bytes must add up to 0
6918   uchar product[20];            // product id
6919   uint *oemtable;               // OEM table pointer
6920   ushort oemlength;             // OEM table length
6921   ushort entry;                 // entry count
6922   uint *lapicaddr;              // address of local APIC
6923   ushort xlength;               // extended table length
6924   uchar xchecksum;              // extended table checksum
6925   uchar reserved;
6926 };
6927
6928 struct mpproc {         // processor table entry
6929   uchar type;                   // entry type (0)
6930   uchar apicid;                 // local APIC id
6931   uchar version;                // local APIC verison
6932   uchar flags;                  // CPU flags
6933     #define MPBOOT 0x02           // This proc is the bootstrap processor.
6934   uchar signature[4];           // CPU signature
6935   uint feature;                 // feature flags from CPUID instruction
6936   uchar reserved[8];
6937 };
6938
6939 struct mpioapic {       // I/O APIC table entry
6940   uchar type;                   // entry type (2)
6941   uchar apicno;                 // I/O APIC id
6942   uchar version;                // I/O APIC version
6943   uchar flags;                  // I/O APIC flags
6944   uint *addr;                  // I/O APIC address
6945 };
6946
6947
6948
6949
```



```
6950 // Table entry types
6951 #define MPPROC    0x00  // One per processor
6952 #define MPBUS     0x01  // One per bus
6953 #define MPIOAPIC  0x02  // One per I/O APIC
6954 #define MPIOINTR  0x03  // One per bus interrupt source
6955 #define MPLINTR   0x04  // One per system interrupt source
6956
6957
6958
6959
6960
6961
6962
6963
6964
6965
6966
6967
6968
6969
6970
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 // Blank page.
7001
7002
7003
7004
7005
7006
7007
7008
7009
7010
7011
7012
7013
7014
7015
7016
7017
7018
7019
7020
7021
7022
7023
7024
7025
7026
7027
7028
7029
7030
7031
7032
7033
7034
7035
7036
7037
7038
7039
7040
7041
7042
7043
7044
7045
7046
7047
7048
7049
```



```
7050 // Multiprocessor support
7051 // Search memory for MP description structures.
7052 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
7053
7054 #include "types.h"
7055 #include "defs.h"
7056 #include "param.h"
7057 #include "memlayout.h"
7058 #include "mp.h"
7059 #include "x86.h"
7060 #include "mmu.h"
7061 #include "proc.h"
7062
7063 struct cpu cpus[NCPU];
7064 static struct cpu *bcpu;
7065 int ismp;
7066 int ncpu;
7067 uchar ioapicid;
7068
7069 int
7070 mpbcpu(void)
7071 {
7072   return bcpu-cpus;
7073 }
7074
7075 static uchar
7076 sum(uchar *addr, int len)
7077 {
7078   int i, sum;
7079
7080   sum = 0;
7081   for(i=0; i<len; i++)
7082     sum += addr[i];
7083   return sum;
7084 }
7085
7086 // Look for an MP structure in the len bytes at addr.
7087 static struct mp*
7088 mpsearch1(uint a, int len)
7089 {
7090   uchar *e, *p, *addr;
7091
7092   addr = p2v(a);
7093   e = addr+len;
7094   for(p = addr; p < e; p += sizeof(struct mp))
7095     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
7096       return (struct mp*)p;
7097   return 0;
7098 }
7099
```

```
7100 // Search for the MP Floating Pointer Structure, which according to the
7101 // spec is in one of the following three locations:
7102 // 1) in the first KB of the EBDA;
7103 // 2) in the last KB of system base memory;
7104 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
7105 static struct mp*
7106 mpsearch(void)
7107 {
7108   uchar *bda;
7109   uint p;
7110   struct mp *mp;
7111
7112   bda = (uchar *) P2V(0x400);
7113   if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
7114     if((mp = mpsearch1(p, 1024)))
7115       return mp;
7116   } else {
7117     p = ((bda[0x14]<<8)|bda[0x13])*1024;
7118     if((mp = mpsearch1(p-1024, 1024)))
7119       return mp;
7120   }
7121   return mpsearch1(0xF0000, 0x10000);
7122 }
7123
7124 // Search for an MP configuration table.  For now,
7125 // don't accept the default configurations (physaddr == 0).
7126 // Check for correct signature, calculate the checksum and,
7127 // if correct, check the version.
7128 // To do: check extended table checksum.
7129 static struct mpconf*
7130 mpconfig(struct mp **pmp)
7131 {
7132   struct mpconf *conf;
7133   struct mp *mp;
7134
7135   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
7136     return 0;
7137   conf = (struct mpconf*) p2v((uint) mp->physaddr);
7138   if(memcmp(conf, "PCMP", 4) != 0)
7139     return 0;
7140   if(conf->version != 1 && conf->version != 4)
7141     return 0;
7142   if(sum((uchar*)conf, conf->length) != 0)
7143     return 0;
7144   *pmp = mp;
7145   return conf;
7146 }
7147
7148
7149
```



```
7150 void
7151 mpinit(void)
7152 {
7153   uchar *p, *e;
7154   struct mp *mp;
7155   struct mpconf *conf;
7156   struct mpproc *proc;
7157   struct mpioapic *ioapic;
7158
7159   bcpu = &cpus[0];
7160   if((conf = mpconfig(&mp)) == 0)
7161     return;
7162   ismp = 1;
7163   lapic = (uint*)conf->lapicaddr;
7164   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
7165     switch(*p){
7166     case MPPROC:
7167       proc = (struct mpproc*)p;
7168       if(ncpu != proc->apicid){
7169         cprintf("mpinit: ncpu=%d apicid=%d\n", ncpu, proc->apicid);
7170         ismp = 0;
7171       }
7172       if(proc->flags & MPBOOT)
7173         bcpu = &cpus[ncpu];
7174       cpus[ncpu].id = ncpu;
7175       ncpu++;
7176       p += sizeof(struct mpproc);
7177       continue;
7178     case MPIOAPIC:
7179       ioapic = (struct mpioapic*)p;
7180       ioapicid = ioapic->apicno;
7181       p += sizeof(struct mpioapic);
7182       continue;
7183     case MPBUS:
7184     case MPIOINTR:
7185     case MPLINTR:
7186       p += 8;
7187       continue;
7188     default:
7189       cprintf("mpinit: unknown config type %x\n", *p);
7190       ismp = 0;
7191     }
7192   }
7193   if(!ismp){
7194     // Didn't like what we found; fall back to no MP.
7195     ncpu = 1;
7196     lapic = 0;
7197     ioapicid = 0;
7198     return;
7199   }
```

```
7200   if(mp->imcrp){
7201     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
7202     // But it would on real hardware.
7203     outb(0x22, 0x70);   // Select IMCR
7204     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
7205   }
7206 }
7207
7208
7209
7210
7211
7212
7213
7214
7215
7216
7217
7218
7219
7220
7221
7222
7223
7224
7225
7226
7227
7228
7229
7230
7231
7232
7233
7234
7235
7236
7237
7238
7239
7240
7241
7242
7243
7244
7245
7246
7247
7248
7249
```



```
7250 // The local APIC manages internal (non-I/O) interrupts.
7251 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
7252 // As of 7/26/2016, Intel processor manual Chapter 10 of Volume 3
7253
7254 #include "types.h"
7255 #include "defs.h"
7256 #include "date.h"
7257 #include "memlayout.h"
7258 #include "traps.h"
7259 #include "mmu.h"
7260 #include "x86.h"
7261
7262 // Local APIC registers, divided by 4 for use as uint[] indices.
7263 #define ID      (0x0020/4)   // ID
7264 #define VER     (0x0030/4)   // Version
7265 #define TPR     (0x0080/4)   // Task Priority
7266 #define EOI     (0x00B0/4)   // EOI
7267 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
7268   #define ENABLE     0x00000100   // Unit Enable
7269 #define ESR     (0x0280/4)   // Error Status
7270 #define ICRLO   (0x0300/4)   // Interrupt Command
7271   #define INIT       0x00000500   // INIT/RESET
7272   #define STARTUP    0x00000600   // Startup IPI
7273   #define DELIVS     0x00001000   // Delivery status
7274   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
7275   #define DEASSERT   0x00000000
7276   #define LEVEL      0x00008000   // Level triggered
7277   #define BCAST      0x00080000   // Send to all APICs, including self.
7278   #define BUSY       0x00001000
7279   #define FIXED      0x00000000
7280 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
7281 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
7282   #define X1         0x0000000B   // divide counts by 1
7283   #define PERIODIC   0x00020000   // Periodic
7284 #define PCINT   (0x0340/4)   // Performance Counter LVT
7285 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
7286 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
7287 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
7288   #define MASKED     0x00010000   // Interrupt masked
7289 #define TICR    (0x0380/4)   // Timer Initial Count
7290 #define TCCR    (0x0390/4)   // Timer Current Count
7291 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
7292
7293 volatile uint *lapic;  // Initialized in mp.c
7294
7295 static void
7296 lapicw(int index, int value)
7297 {
7298   lapic[index] = value;
7299   lapic[ID];  // wait for write to finish, by reading
```

```
7300 }
7301
7302
7303
7304
7305
7306
7307
7308
7309
7310
7311
7312
7313
7314
7315
7316
7317
7318
7319
7320
7321
7322
7323
7324
7325
7326
7327
7328
7329
7330
7331
7332
7333
7334
7335
7336
7337
7338
7339
7340
7341
7342
7343
7344
7345
7346
7347
7348
7349
```



```
7350 void
7351 lapicinit(void)
7352 {
7353   if(!lapic)
7354     return;
7355
7356   // Enable local APIC; set spurious interrupt vector.
7357   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
7358
7359   // The timer repeatedly counts down at bus frequency
7360   // from lapic[TICR] and then issues an interrupt.
7361   // If xv6 cared more about precise timekeeping,
7362   // TICR would be calibrated using an external time source.
7363   lapicw(TDCR, X1);
7364   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
7365   lapicw(TICR, 10000000);
7366
7367   // Disable logical interrupt lines.
7368   lapicw(LINT0, MASKED);
7369   lapicw(LINT1, MASKED);
7370
7371   // Disable performance counter overflow interrupts
7372   // on machines that provide that interrupt entry.
7373   if(((lapic[VER]>>16) & 0xFF) >= 4)
7374     lapicw(PCINT, MASKED);
7375
7376   // Map error interrupt to IRQ_ERROR.
7377   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
7378
7379   // Clear error status register (requires back-to-back writes).
7380   lapicw(ESR, 0);
7381   lapicw(ESR, 0);
7382
7383   // Ack any outstanding interrupts.
7384   lapicw(EOI, 0);
7385
7386   // Send an Init Level De-Assert to synchronise arbitration ID's.
7387   lapicw(ICRHI, 0);
7388   lapicw(ICRLO, BCAST | INIT | LEVEL);
7389   while(lapic[ICRLO] & DELIVS)
7390     ;
7391
7392   // Enable interrupts on the APIC (but not on the processor).
7393   lapicw(TPR, 0);
7394 }
7395
7396
7397
7398
7399
```

```
7400 int
7401 cpunum(void)
7402 {
7403   // Cannot call cpu when interrupts are enabled:
7404   // result not guaranteed to last long enough to be used!
7405   // Would prefer to panic but even printing is chancy here:
7406   // almost everything, including cprintf and panic, calls cpu,
7407   // often indirectly through acquire and release.
7408   if(readeflags()&FL_IF){
7409     static int n;
7410     if(n++ == 0)
7411       cprintf("cpu called from %x with interrupts enabled\n",
7412         __builtin_return_address(0));
7413   }
7414
7415   if(lapic)
7416     return lapic[ID]>>24;
7417   return 0;
7418 }
7419
7420 // Acknowledge interrupt.
7421 void
7422 lapiceoi(void)
7423 {
7424   if(lapic)
7425     lapicw(EOI, 0);
7426 }
7427
7428 // Spin for a given number of microseconds.
7429 // On real hardware would want to tune this dynamically.
7430 void
7431 microdelay(int us)
7432 {
7433 }
7434
7435 #define CMOS_PORT    0x70
7436 #define CMOS_RETURN  0x71
7437
7438 // Start additional processor running entry code at addr.
7439 // See Appendix B of MultiProcessor Specification.
7440 void
7441 lapicstartap(uchar apicid, uint addr)
7442 {
7443   int i;
7444   ushort *wrv;
7445
7446   // "The BSP must initialize CMOS shutdown code to 0AH
7447   // and the warm reset vector (DWORD based at 40:67) to point at
7448   // the AP startup code prior to the [universal startup algorithm]."
7449   outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
```



```
7450   outb(CMOS_PORT+1, 0x0A);
7451   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
7452   wrv[0] = 0;
7453   wrv[1] = addr >> 4;
7454
7455   // "Universal startup algorithm."
7456   // Send INIT (level-triggered) interrupt to reset other CPU.
7457   lapicw(ICRHI, apicid<<24);
7458   lapicw(ICRLO, INIT | LEVEL | ASSERT);
7459   microdelay(200);
7460   lapicw(ICRLO, INIT | LEVEL);
7461   microdelay(100);    // should be 10ms, but too slow in Bochs!
7462
7463   // Send startup IPI (twice!) to enter code.
7464   // Regular hardware is supposed to only accept a STARTUP
7465   // when it is in the halted state due to an INIT.  So the second
7466   // should be ignored, but it is part of the official Intel algorithm.
7467   // Bochs complains about the second one.  Too bad for Bochs.
7468   for(i = 0; i < 2; i++){
7469     lapicw(ICRHI, apicid<<24);
7470     lapicw(ICRLO, STARTUP | (addr>>12));
7471     microdelay(200);
7472   }
7473 }
7474
7475 #define CMOS_STATA   0x0a
7476 #define CMOS_STATB   0x0b
7477 #define CMOS_UIP    (1 << 7)        // RTC update in progress
7478
7479 #define SECS    0x00
7480 #define MINS    0x02
7481 #define HOURS   0x04
7482 #define DAY     0x07
7483 #define MONTH   0x08
7484 #define YEAR    0x09
7485
7486 static uint cmos_read(uint reg)
7487 {
7488   outb(CMOS_PORT,  reg);
7489   microdelay(200);
7490
7491   return inb(CMOS_RETURN);
7492 }
7493
7494
7495
7496
7497
7498
7499
```

```
7500 static void fill_rtcdate(struct rtcdate *r)
7501 {
7502   r->second = cmos_read(SECS);
7503   r->minute = cmos_read(MINS);
7504   r->hour   = cmos_read(HOURS);
7505   r->day    = cmos_read(DAY);
7506   r->month  = cmos_read(MONTH);
7507   r->year   = cmos_read(YEAR);
7508 }
7509
7510 // qemu seems to use 24-hour GWT and the values are BCD encoded
7511 void cmostime(struct rtcdate *r)
7512 {
7513   struct rtcdate t1, t2;
7514   int sb, bcd;
7515
7516   sb = cmos_read(CMOS_STATB);
7517
7518   bcd = (sb & (1 << 2)) == 0;
7519
7520   // make sure CMOS doesn't modify time while we read it
7521   for (;;) {
7522     fill_rtcdate(&t1);
7523     if (cmos_read(CMOS_STATA) & CMOS_UIP)
7524         continue;
7525     fill_rtcdate(&t2);
7526     if (memcmp(&t1, &t2, sizeof(t1)) == 0)
7527       break;
7528   }
7529
7530   // convert
7531   if (bcd) {
7532 #define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
7533     CONV(second);
7534     CONV(minute);
7535     CONV(hour  );
7536     CONV(day   );
7537     CONV(month );
7538     CONV(year  );
7539 #undef     CONV
7540   }
7541
7542   *r = t1;
7543   r->year += 2000;
7544 }
7545
7546
7547
7548
7549
```



```
7550 // The I/O APIC manages hardware interrupts for an SMP system.
7551 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
7552 // See also picirq.c.
7553
7554 #include "types.h"
7555 #include "defs.h"
7556 #include "traps.h"
7557
7558 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
7559
7560 #define REG_ID     0x00  // Register index: ID
7561 #define REG_VER    0x01  // Register index: version
7562 #define REG_TABLE  0x10  // Redirection table base
7563
7564 // The redirection table starts at REG_TABLE and uses
7565 // two registers to configure each interrupt.
7566 // The first (low) register in a pair contains configuration bits.
7567 // The second (high) register contains a bitmask telling which
7568 // CPUs can serve that interrupt.
7569 #define INT_DISABLED   0x00010000  // Interrupt disabled
7570 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
7571 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
7572 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
7573
7574 volatile struct ioapic *ioapic;
7575
7576 // IO APIC MMIO structure: write reg, then read or write data.
7577 struct ioapic {
7578   uint reg;
7579   uint pad[3];
7580   uint data;
7581 };
7582
7583 static uint
7584 ioapicread(int reg)
7585 {
7586   ioapic->reg = reg;
7587   return ioapic->data;
7588 }
7589
7590 static void
7591 ioapicwrite(int reg, uint data)
7592 {
7593   ioapic->reg = reg;
7594   ioapic->data = data;
7595 }
7596
7597
7598
7599
```

```
7600 void
7601 ioapicinit(void)
7602 {
7603   int i, id, maxintr;
7604
7605   if(!ismp)
7606     return;
7607
7608   ioapic = (volatile struct ioapic*)IOAPIC;
7609   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
7610   id = ioapicread(REG_ID) >> 24;
7611   if(id != ioapicid)
7612     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
7613
7614   // Mark all interrupts edge-triggered, active high, disabled,
7615   // and not routed to any CPUs.
7616   for(i = 0; i <= maxintr; i++){
7617     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
7618     ioapicwrite(REG_TABLE+2*i+1, 0);
7619   }
7620 }
7621
7622 void
7623 ioapicenable(int irq, int cpunum)
7624 {
7625   if(!ismp)
7626     return;
7627
7628   // Mark interrupt edge-triggered, active high,
7629   // enabled, and routed to the given cpunum,
7630   // which happens to be that cpu's APIC ID.
7631   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
7632   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
7633 }
7634
7635
7636
7637
7638
7639
7640
7641
7642
7643
7644
7645
7646
7647
7648
7649
```



```
7650 // Intel 8259A programmable interrupt controllers.
7651
7652 #include "types.h"
7653 #include "x86.h"
7654 #include "traps.h"
7655
7656 // I/O Addresses of the two programmable interrupt controllers
7657 #define IO_PIC1         0x20    // Master (IRQs 0-7)
7658 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
7659
7660 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
7661
7662 // Current IRQ mask.
7663 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
7664 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
7665
7666 static void
7667 picsetmask(ushort mask)
7668 {
7669   irqmask = mask;
7670   outb(IO_PIC1+1, mask);
7671   outb(IO_PIC2+1, mask >> 8);
7672 }
7673
7674 void
7675 picenable(int irq)
7676 {
7677   picsetmask(irqmask & ~(1<<irq));
7678 }
7679
7680 // Initialize the 8259A interrupt controllers.
7681 void
7682 picinit(void)
7683 {
7684   // mask all interrupts
7685   outb(IO_PIC1+1, 0xFF);
7686   outb(IO_PIC2+1, 0xFF);
7687
7688   // Set up master (8259A-1)
7689
7690   // ICW1:  0001g0hi
7691   //    g:  0 = edge triggering, 1 = level triggering
7692   //    h:  0 = cascaded PICs, 1 = master only
7693   //    i:  0 = no ICW4, 1 = ICW4 required
7694   outb(IO_PIC1, 0x11);
7695
7696   // ICW2:  Vector offset
7697   outb(IO_PIC1+1, T_IRQ0);
7698
7699
```

```
7700   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
7701   //        (slave PIC) 3-bit # of slave's connection to master
7702   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
7703
7704   // ICW4:  000nbmap
7705   //    n:  1 = special fully nested mode
7706   //    b:  1 = buffered mode
7707   //    m:  0 = slave PIC, 1 = master PIC
7708   //      (ignored when b is 0, as the master/slave role
7709   //      can be hardwired).
7710   //    a:  1 = Automatic EOI mode
7711   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
7712   outb(IO_PIC1+1, 0x3);
7713
7714   // Set up slave (8259A-2)
7715   outb(IO_PIC2, 0x11);                  // ICW1
7716   outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
7717   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
7718   // NB Automatic EOI mode doesn't tend to work on the slave.
7719   // Linux source code says it's "to be investigated".
7720   outb(IO_PIC2+1, 0x3);                 // ICW4
7721
7722   // OCW3:  0ef01prs
7723   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
7724   //    p:  0 = no polling, 1 = polling mode
7725   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
7726   outb(IO_PIC1, 0x68);             // clear specific mask
7727   outb(IO_PIC1, 0x0a);             // read IRR by default
7728
7729   outb(IO_PIC2, 0x68);             // OCW3
7730   outb(IO_PIC2, 0x0a);             // OCW3
7731
7732   if(irqmask != 0xFFFF)
7733     picsetmask(irqmask);
7734 }
7735
7736
7737
7738
7739
7740
7741
7742
7743
7744
7745
7746
7747
7748
7749
```



```
7750 // PC keyboard interface constants
7751
7752 #define KBSTATP         0x64    // kbd controller status port(I)
7753 #define KBS_DIB         0x01    // kbd data in buffer
7754 #define KBDATAP         0x60    // kbd data port(I)
7755
7756 #define NO              0
7757
7758 #define SHIFT           (1<<0)
7759 #define CTL             (1<<1)
7760 #define ALT             (1<<2)
7761
7762 #define CAPSLOCK        (1<<3)
7763 #define NUMLOCK         (1<<4)
7764 #define SCROLLLOCK      (1<<5)
7765
7766 #define E0ESC           (1<<6)
7767
7768 // Special keycodes
7769 #define KEY_HOME        0xE0
7770 #define KEY_END         0xE1
7771 #define KEY_UP          0xE2
7772 #define KEY_DN          0xE3
7773 #define KEY_LF          0xE4
7774 #define KEY_RT          0xE5
7775 #define KEY_PGUP        0xE6
7776 #define KEY_PGDN        0xE7
7777 #define KEY_INS         0xE8
7778 #define KEY_DEL         0xE9
7779
7780 // C('A') == Control-A
7781 #define C(x) (x - '@')
7782
7783 static uchar shiftcode[256] =
7784 {
7785   [0x1D] CTL,
7786   [0x2A] SHIFT,
7787   [0x36] SHIFT,
7788   [0x38] ALT,
7789   [0x9D] CTL,
7790   [0xB8] ALT
7791 };
7792
7793 static uchar togglecode[256] =
7794 {
7795   [0x3A] CAPSLOCK,
7796   [0x45] NUMLOCK,
7797   [0x46] SCROLLLOCK
7798 };
7799
```

```
7800 static uchar normalmap[256] =
7801 {
7802   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
7803   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
7804   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
7805   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
7806   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
7807   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
7808   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
7809   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7810   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7811   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7812   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7813   [0x9C] '\n',      // KP_Enter
7814   [0xB5] '/',       // KP_Div
7815   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7816   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7817   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7818   [0x97] KEY_HOME,  [0xCF] KEY_END,
7819   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7820 };
7821
7822 static uchar shiftmap[256] =
7823 {
7824   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
7825   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
7826   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
7827   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
7828   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
7829   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
7830   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
7831   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7832   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7833   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7834   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7835   [0x9C] '\n',      // KP_Enter
7836   [0xB5] '/',       // KP_Div
7837   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7838   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7839   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7840   [0x97] KEY_HOME,  [0xCF] KEY_END,
7841   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7842 };
7843
7844
7845
7846
7847
7848
7849
```



```
7850 static uchar ctlmap[256] =
7851 {
7852   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7853   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7854   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
7855   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
7856   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
7857   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
7858   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
7859   [0x9C] '\r',      // KP_Enter
7860   [0xB5] C('/'),    // KP_Div
7861   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7862   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7863   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7864   [0x97] KEY_HOME,  [0xCF] KEY_END,
7865   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7866 };
7867
7868
7869
7870
7871
7872
7873
7874
7875
7876
7877
7878
7879
7880
7881
7882
7883
7884
7885
7886
7887
7888
7889
7890
7891
7892
7893
7894
7895
7896
7897
7898
7899
```

```
7900 #include "types.h"
7901 #include "x86.h"
7902 #include "defs.h"
7903 #include "kbd.h"
7904
7905 int
7906 kbdgetc(void)
7907 {
7908   static uint shift;
7909   static uchar *charcode[4] = {
7910     normalmap, shiftmap, ctlmap, ctlmap
7911   };
7912   uint st, data, c;
7913
7914   st = inb(KBSTATP);
7915   if((st & KBS_DIB) == 0)
7916     return -1;
7917   data = inb(KBDATAP);
7918
7919   if(data == 0xE0){
7920     shift |= E0ESC;
7921     return 0;
7922   } else if(data & 0x80){
7923     // Key released
7924     data = (shift & E0ESC ? data : data & 0x7F);
7925     shift &= ~(shiftcode[data] | E0ESC);
7926     return 0;
7927   } else if(shift & E0ESC){
7928     // Last character was an E0 escape; or with 0x80
7929     data |= 0x80;
7930     shift &= ~E0ESC;
7931   }
7932
7933   shift |= shiftcode[data];
7934   shift ^= togglecode[data];
7935   c = charcode[shift & (CTL | SHIFT)][data];
7936   if(shift & CAPSLOCK){
7937     if('a' <= c && c <= 'z')
7938       c += 'A' - 'a';
7939     else if('A' <= c && c <= 'Z')
7940       c += 'a' - 'A';
7941   }
7942   return c;
7943 }
7944
7945 void
7946 kbdintr(void)
7947 {
7948   consoleintr(kbdgetc);
7949 }
```



```
7950 // Console input and output.
7951 // Input is from the keyboard or serial port.
7952 // Output is written to the screen and serial port.
7953
7954 #include "types.h"
7955 #include "defs.h"
7956 #include "param.h"
7957 #include "traps.h"
7958 #include "spinlock.h"
7959 #include "fs.h"
7960 #include "file.h"
7961 #include "memlayout.h"
7962 #include "mmu.h"
7963 #include "proc.h"
7964 #include "x86.h"
7965
7966 static void consputc(int);
7967
7968 static int panicked = 0;
7969
7970 static struct {
7971   struct spinlock lock;
7972   int locking;
7973 } cons;
7974
7975 static void
7976 printint(int xx, int base, int sign)
7977 {
7978   static char digits[] = "0123456789abcdef";
7979   char buf[16];
7980   int i;
7981   uint x;
7982
7983   if(sign && (sign = xx < 0))
7984     x = -xx;
7985   else
7986     x = xx;
7987
7988   i = 0;
7989   do{
7990     buf[i++] = digits[x % base];
7991   }while((x /= base) != 0);
7992
7993   if(sign)
7994     buf[i++] = '-';
7995
7996   while(--i >= 0)
7997     consputc(buf[i]);
7998 }
7999
```

```
8000 // Print to the console. only understands %d, %x, %p, %s.
8001 void
8002 cprintf(char *fmt, ...)
8003 {
8004   int i, c, locking;
8005   uint *argp;
8006   char *s;
8007
8008   locking = cons.locking;
8009   if(locking)
8010     acquire(&cons.lock);
8011
8012   if (fmt == 0)
8013     panic("null fmt");
8014
8015   argp = (uint*)(void*)(&fmt + 1);
8016   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
8017     if(c != '%'){
8018       consputc(c);
8019       continue;
8020     }
8021     c = fmt[++i] & 0xff;
8022     if(c == 0)
8023       break;
8024     switch(c){
8025     case 'd':
8026       printint(*argp++, 10, 1);
8027       break;
8028     case 'x':
8029     case 'p':
8030       printint(*argp++, 16, 0);
8031       break;
8032     case 's':
8033       if((s = (char*)*argp++) == 0)
8034         s = "(null)";
8035       for(; *s; s++)
8036         consputc(*s);
8037       break;
8038     case '%':
8039       consputc('%');
8040       break;
8041     default:
8042       // Print unknown % sequence to draw attention.
8043       consputc('%');
8044       consputc(c);
8045       break;
8046     }
8047   }
8048
8049
```



```
8050   if(locking)
8051     release(&cons.lock);
8052 }
8053
8054 void
8055 panic(char *s)
8056 {
8057   int i;
8058   uint pcs[10];
8059
8060   cli();
8061   cons.locking = 0;
8062   cprintf("cpu%d: panic: ", cpu->id);
8063   cprintf(s);
8064   cprintf("\n");
8065   getcallerpcs(&s, pcs);
8066   for(i=0; i<10; i++)
8067     cprintf(" %p", pcs[i]);
8068   panicked = 1; // freeze other CPU
8069   for(;;)
8070     ;
8071 }
8072
8073
8074
8075
8076
8077
8078
8079
8080
8081
8082
8083
8084
8085
8086
8087
8088
8089
8090
8091
8092
8093
8094
8095
8096
8097
8098
8099
```

```
8100 #define BACKSPACE 0x100
8101 #define CRTPORT 0x3d4
8102 static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory
8103
8104 static void
8105 cgaputc(int c)
8106 {
8107   int pos;
8108
8109   // Cursor position: col + 80*row.
8110   outb(CRTPORT, 14);
8111   pos = inb(CRTPORT+1) << 8;
8112   outb(CRTPORT, 15);
8113   pos |= inb(CRTPORT+1);
8114
8115   if(c == '\n')
8116     pos += 80 - pos%80;
8117   else if(c == BACKSPACE){
8118     if(pos > 0) --pos;
8119   } else
8120     crt[pos++] = (c&0xff) | 0x0700;  // black on white
8121
8122   if(pos < 0 || pos > 25*80)
8123     panic("pos under/overflow");
8124
8125   if((pos/80) >= 24){  // Scroll up.
8126     memmove(crt, crt+80, sizeof(crt[0])*23*80);
8127     pos -= 80;
8128     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
8129   }
8130
8131   outb(CRTPORT, 14);
8132   outb(CRTPORT+1, pos>>8);
8133   outb(CRTPORT, 15);
8134   outb(CRTPORT+1, pos);
8135   crt[pos] = ' ' | 0x0700;
8136 }
8137
8138
8139
8140
8141
8142
8143
8144
8145
8146
8147
8148
8149
```



```
8150 void
8151 consputc(int c)
8152 {
8153   if(panicked){
8154     cli();
8155     for(;;)
8156       ;
8157   }
8158
8159   if(c == BACKSPACE){
8160     uartputc('\b'); uartputc(' '); uartputc('\b');
8161   } else
8162     uartputc(c);
8163   cgaputc(c);
8164 }
8165
8166 #define INPUT_BUF 128
8167 struct {
8168   char buf[INPUT_BUF];
8169   uint r;  // Read index
8170   uint w;  // Write index
8171   uint e;  // Edit index
8172 } input;
8173
8174 #define C(x)  ((x)-'@')  // Control-x
8175
8176 void
8177 consoleintr(int (*getc)(void))
8178 {
8179   int c, doprocdump = 0;
8180
8181   acquire(&cons.lock);
8182   while((c = getc()) >= 0){
8183     switch(c){
8184     case C('P'):  // Process listing.
8185       doprocdump = 1;   // procdump() locks cons.lock indirectly; invoke lat
8186       break;
8187     case C('U'):  // Kill line.
8188       while(input.e != input.w &&
8189             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
8190         input.e--;
8191         consputc(BACKSPACE);
8192       }
8193       break;
8194     case C('H'): case '\x7f':  // Backspace
8195       if(input.e != input.w){
8196         input.e--;
8197         consputc(BACKSPACE);
8198       }
8199       break;
```

```
8200     default:
8201       if(c != 0 && input.e-input.r < INPUT_BUF){
8202         c = (c == '\r') ? '\n' : c;
8203         input.buf[input.e++ % INPUT_BUF] = c;
8204         consputc(c);
8205         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
8206           input.w = input.e;
8207           wakeup(&input.r);
8208         }
8209       }
8210       break;
8211     }
8212   }
8213   release(&cons.lock);
8214   if(doprocdump) {
8215     procdump();  // now call procdump() wo. cons.lock held
8216   }
8217 }
8218
8219 int
8220 consoleread(struct inode *ip, char *dst, int n)
8221 {
8222   uint target;
8223   int c;
8224
8225   iunlock(ip);
8226   target = n;
8227   acquire(&cons.lock);
8228   while(n > 0){
8229     while(input.r == input.w){
8230       if(proc->killed){
8231         release(&cons.lock);
8232         ilock(ip);
8233         return -1;
8234       }
8235       sleep(&input.r, &cons.lock);
8236     }
8237     c = input.buf[input.r++ % INPUT_BUF];
8238     if(c == C('D')){  // EOF
8239       if(n < target){
8240         // Save ^D for next time, to make sure
8241         // caller gets a 0-byte result.
8242         input.r--;
8243       }
8244       break;
8245     }
8246     *dst++ = c;
8247     --n;
8248     if(c == '\n')
8249       break;
```



```
8250   }
8251   release(&cons.lock);
8252   ilock(ip);
8253
8254   return target - n;
8255 }
8256
8257 int
8258 consolewrite(struct inode *ip, char *buf, int n)
8259 {
8260   int i;
8261
8262   iunlock(ip);
8263   acquire(&cons.lock);
8264   for(i = 0; i < n; i++)
8265     consputc(buf[i] & 0xff);
8266   release(&cons.lock);
8267   ilock(ip);
8268
8269   return n;
8270 }
8271
8272 void
8273 consoleinit(void)
8274 {
8275   initlock(&cons.lock, "console");
8276
8277   devsw[CONSOLE].write = consolewrite;
8278   devsw[CONSOLE].read = consoleread;
8279   cons.locking = 1;
8280
8281   picenable(IRQ_KBD);
8282   ioapicenable(IRQ_KBD, 0);
8283 }
8284
8285
8286
8287
8288
8289
8290
8291
8292
8293
8294
8295
8296
8297
8298
8299
```

```
8300 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
8301 // Only used on uniprocessors;
8302 // SMP machines use the local APIC timer.
8303
8304 #include "types.h"
8305 #include "defs.h"
8306 #include "traps.h"
8307 #include "x86.h"
8308
8309 #define IO_TIMER1       0x040           // 8253 Timer #1
8310
8311 // Frequency of all three count-down timers;
8312 // (TIMER_FREQ/freq) is the appropriate count
8313 // to generate a frequency of freq Hz.
8314
8315 #define TIMER_FREQ      1193182
8316 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
8317
8318 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
8319 #define TIMER_SEL0      0x00    // select counter 0
8320 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
8321 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
8322
8323 void
8324 timerinit(void)
8325 {
8326   // Interrupt 100 times/sec.
8327   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
8328   outb(IO_TIMER1, TIMER_DIV(100) % 256);
8329   outb(IO_TIMER1, TIMER_DIV(100) / 256);
8330   picenable(IRQ_TIMER);
8331 }
8332
8333
8334
8335
8336
8337
8338
8339
8340
8341
8342
8343
8344
8345
8346
8347
8348
8349
```



```
8350 // Intel 8250 serial port (UART).
8351
8352 #include "types.h"
8353 #include "defs.h"
8354 #include "param.h"
8355 #include "traps.h"
8356 #include "spinlock.h"
8357 #include "fs.h"
8358 #include "file.h"
8359 #include "mmu.h"
8360 #include "proc.h"
8361 #include "x86.h"
8362
8363 #define COM1    0x3f8
8364
8365 static int uart;    // is there a uart?
8366
8367 void
8368 uartinit(void)
8369 {
8370   char *p;
8371
8372   // Turn off the FIFO
8373   outb(COM1+2, 0);
8374
8375   // 9600 baud, 8 data bits, 1 stop bit, parity off.
8376   outb(COM1+3, 0x80);    // Unlock divisor
8377   outb(COM1+0, 115200/9600);
8378   outb(COM1+1, 0);
8379   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
8380   outb(COM1+4, 0);
8381   outb(COM1+1, 0x01);    // Enable receive interrupts.
8382
8383   // If status is 0xFF, no serial port.
8384   if(inb(COM1+5) == 0xFF)
8385     return;
8386   uart = 1;
8387
8388   // Acknowledge pre-existing interrupt conditions;
8389   // enable interrupts.
8390   inb(COM1+2);
8391   inb(COM1+0);
8392   picenable(IRQ_COM1);
8393   ioapicenable(IRQ_COM1, 0);
8394
8395   // Announce that we're here.
8396   for(p="xv6...\n"; *p; p++)
8397     uartputc(*p);
8398 }
8399
```

```
8400 void
8401 uartputc(int c)
8402 {
8403   int i;
8404
8405   if(!uart)
8406     return;
8407   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
8408     microdelay(10);
8409   outb(COM1+0, c);
8410 }
8411
8412 static int
8413 uartgetc(void)
8414 {
8415   if(!uart)
8416     return -1;
8417   if(!(inb(COM1+5) & 0x01))
8418     return -1;
8419   return inb(COM1+0);
8420 }
8421
8422 void
8423 uartintr(void)
8424 {
8425   consoleintr(uartgetc);
8426 }
8427
8428
8429
8430
8431
8432
8433
8434
8435
8436
8437
8438
8439
8440
8441
8442
8443
8444
8445
8446
8447
8448
8449
```



```
8450 # Initial process execs /init.
8451
8452 #include "syscall.h"
8453 #include "traps.h"
8454
8455
8456 # exec(init, argv)
8457 .globl start
8458 start:
8459   pushl $argv
8460   pushl $init
8461   pushl $0  // where caller pc would be
8462   movl $SYS_exec, %eax
8463   int $T_SYSCALL
8464
8465 # for(;;) exit();
8466 exit:
8467   movl $SYS_exit, %eax
8468   int $T_SYSCALL
8469   jmp exit
8470
8471 # char init[] = "/init\0";
8472 init:
8473   .string "/init\0"
8474
8475 # char *argv[] = { init, 0 };
8476 .p2align 2
8477 argv:
8478   .long init
8479   .long 0
8480
8481
8482
8483
8484
8485
8486
8487
8488
8489
8490
8491
8492
8493
8494
8495
8496
8497
8498
8499
```

```
8500 #include "syscall.h"
8501 #include "traps.h"
8502
8503 #define SYSCALL(name) \
8504   .globl name; \
8505   name: \
8506     movl $SYS_ ## name, %eax; \
8507     int $T_SYSCALL; \
8508     ret
8509
8510 SYSCALL(fork)
8511 SYSCALL(exit)
8512 SYSCALL(wait)
8513 SYSCALL(pipe)
8514 SYSCALL(read)
8515 SYSCALL(write)
8516 SYSCALL(close)
8517 SYSCALL(kill)
8518 SYSCALL(exec)
8519 SYSCALL(open)
8520 SYSCALL(mknod)
8521 SYSCALL(unlink)
8522 SYSCALL(fstat)
8523 SYSCALL(link)
8524 SYSCALL(mkdir)
8525 SYSCALL(chdir)
8526 SYSCALL(dup)
8527 SYSCALL(getpid)
8528 SYSCALL(sbrk)
8529 SYSCALL(sleep)
8530 SYSCALL(uptime)
8531 SYSCALL(halt)
8532
8533
8534
8535
8536
8537
8538
8539
8540
8541
8542
8543
8544
8545
8546
8547
8548
8549
```



```
8550 // init: The initial user-level program
8551
8552 #include "types.h"
8553 #include "stat.h"
8554 #include "user.h"
8555 #include "fcntl.h"
8556
8557 char *argv[] = { "sh", 0 };
8558
8559 int
8560 main(void)
8561 {
8562   int pid, wpid;
8563
8564   if(open("console", O_RDWR) < 0){
8565     mknod("console", 1, 1);
8566     open("console", O_RDWR);
8567   }
8568   dup(0);  // stdout
8569   dup(0);  // stderr
8570
8571   for(;;){
8572     printf(1, "init: starting sh\n");
8573     pid = fork();
8574     if(pid < 0){
8575       printf(1, "init: fork failed\n");
8576       exit();
8577     }
8578     if(pid == 0){
8579       exec("sh", argv);
8580       printf(1, "init: exec sh failed\n");
8581       exit();
8582     }
8583     while((wpid=wait()) >= 0 && wpid != pid)
8584       printf(1, "zombie!\n");
8585   }
8586 }
8587
8588
8589
8590
8591
8592
8593
8594
8595
8596
8597
8598
8599
```

```
8600 // Shell.
8601 // 2015-12-21. Added very simple processing for builtin commands
8602
8603 #include "types.h"
8604 #include "user.h"
8605 #include "fcntl.h"
8606
8607 // Parsed command representation
8608 #define EXEC  1
8609 #define REDIR 2
8610 #define PIPE  3
8611 #define LIST  4
8612 #define BACK  5
8613
8614 #define MAXARGS 10
8615
8616 struct cmd {
8617   int type;
8618 };
8619
8620 struct execcmd {
8621   int type;
8622   char *argv[MAXARGS];
8623   char *eargv[MAXARGS];
8624 };
8625
8626 struct redircmd {
8627   int type;
8628   struct cmd *cmd;
8629   char *file;
8630   char *efile;
8631   int mode;
8632   int fd;
8633 };
8634
8635 struct pipecmd {
8636   int type;
8637   struct cmd *left;
8638   struct cmd *right;
8639 };
8640
8641 struct listcmd {
8642   int type;
8643   struct cmd *left;
8644   struct cmd *right;
8645 };
8646
8647
8648
8649
```



```
8650 struct backcmd {
8651   int type;
8652   struct cmd *cmd;
8653 };
8654
8655 int fork1(void);  // Fork but panics on failure.
8656 void panic(char*);
8657 struct cmd *parsecmd(char*);
8658
8659 // Execute cmd.  Never returns.
8660 void
8661 runcmd(struct cmd *cmd)
8662 {
8663   int p[2];
8664   struct backcmd *bcmd;
8665   struct execcmd *ecmd;
8666   struct listcmd *lcmd;
8667   struct pipecmd *pcmd;
8668   struct redircmd *rcmd;
8669
8670   if(cmd == 0)
8671     exit();
8672
8673   switch(cmd->type){
8674   default:
8675     panic("runcmd");
8676
8677   case EXEC:
8678     ecmd = (struct execcmd*)cmd;
8679     if(ecmd->argv[0] == 0)
8680       exit();
8681     exec(ecmd->argv[0], ecmd->argv);
8682     printf(2, "exec %s failed\n", ecmd->argv[0]);
8683     break;
8684
8685   case REDIR:
8686     rcmd = (struct redircmd*)cmd;
8687     close(rcmd->fd);
8688     if(open(rcmd->file, rcmd->mode) < 0){
8689       printf(2, "open %s failed\n", rcmd->file);
8690       exit();
8691     }
8692     runcmd(rcmd->cmd);
8693     break;
8694
8695   case LIST:
8696     lcmd = (struct listcmd*)cmd;
8697     if(fork1() == 0)
8698       runcmd(lcmd->left);
8699     wait();
```

```
8700     runcmd(lcmd->right);
8701     break;
8702
8703   case PIPE:
8704     pcmd = (struct pipecmd*)cmd;
8705     if(pipe(p) < 0)
8706       panic("pipe");
8707     if(fork1() == 0){
8708       close(1);
8709       dup(p[1]);
8710       close(p[0]);
8711       close(p[1]);
8712       runcmd(pcmd->left);
8713     }
8714     if(fork1() == 0){
8715       close(0);
8716       dup(p[0]);
8717       close(p[0]);
8718       close(p[1]);
8719       runcmd(pcmd->right);
8720     }
8721     close(p[0]);
8722     close(p[1]);
8723     wait();
8724     wait();
8725     break;
8726
8727   case BACK:
8728     bcmd = (struct backcmd*)cmd;
8729     if(fork1() == 0)
8730       runcmd(bcmd->cmd);
8731     break;
8732   }
8733   exit();
8734 }
8735
8736 int
8737 getcmd(char *buf, int nbuf)
8738 {
8739   printf(2, "$ ");
8740   memset(buf, 0, nbuf);
8741   gets(buf, nbuf);
8742   if(buf[0] == 0) // EOF
8743     return -1;
8744   return 0;
8745 }
8746
8747
8748
8749
```



```
8750 #ifdef USE_BUILTINS
8751 // ***** processing for shell builtins begins here *****
8752
8753 int
8754 strncmp(const char *p, const char *q, uint n)
8755 {
8756     while(n > 0 && *p && *p == *q)
8757       n--, p++, q++;
8758     if(n == 0)
8759       return 0;
8760     return (uchar)*p - (uchar)*q;
8761 }
8762
8763 int
8764 makeint(char *p)
8765 {
8766   int val = 0;
8767
8768   while ((*p >= '0') && (*p <= '9')) {
8769     val = 10*val + (*p-'0');
8770     ++p;
8771   }
8772   return val;
8773 }
8774
8775 int
8776 setbuiltin(char *p)
8777 {
8778   int i;
8779
8780   p += strlen("_set");
8781   while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8782   if (strncmp("uid", p, 3) == 0) {
8783     p += strlen("uid");
8784     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8785     i = makeint(p); // ugly
8786     return (setuid(i));
8787   } else
8788   if (strncmp("gid", p, 3) == 0) {
8789     p += strlen("gid");
8790     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8791     i = makeint(p); // ugly
8792     return (setgid(i));
8793   }
8794   printf(2, "Invalid _set parameter\n");
8795   return -1;
8796 }
8797
8798
8799
```

```
8800 int
8801 getbuiltin(char *p)
8802 {
8803   p += strlen("_get");
8804   while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8805   if (strncmp("uid", p, 3) == 0) {
8806     printf(2, "%d\n", getuid());
8807     return 0;
8808   }
8809   if (strncmp("gid", p, 3) == 0) {
8810     printf(2, "%d\n", getgid());
8811     return 0;
8812   }
8813   printf(2, "Invalid _get parameter\n");
8814   return -1;
8815 }
8816
8817 typedef int funcPtr_t(char *);
8818 typedef struct {
8819   char       *cmd;
8820   funcPtr_t  *name;
8821 } dispatchTableEntry_t;
8822
8823 // Use a simple function dispatch table (FDT) to process builtin commands
8824 dispatchTableEntry_t fdt[] = {
8825   {"_set", setbuiltin},
8826   {"_get", getbuiltin}
8827 };
8828 int FDTcount = sizeof(fdt) / sizeof(fdt[0]); // # entris in FDT
8829
8830 void
8831 dobuiltin(char *cmd) {
8832   int i;
8833
8834   for (i=0; i<FDTcount; i++)
8835     if (strncmp(cmd, fdt[i].cmd, strlen(fdt[i].cmd)) == 0)
8836      (*fdt[i].name)(cmd);
8837 }
8838
8839
8840
8841
8842
8843
8844
8845
8846
8847
8848
8849
```



```
8850 // ***** processing for shell builtins ends here *****
8851 #endif
8852
8853 int
8854 main(void)
8855 {
8856   static char buf[100];
8857   int fd;
8858
8859   // Assumes three file descriptors open.
8860   while((fd = open("console", O_RDWR)) >= 0){
8861     if(fd >= 3){
8862       close(fd);
8863       break;
8864     }
8865   }
8866
8867   // Read and run input commands.
8868   while(getcmd(buf, sizeof(buf)) >= 0){
8869 // add support for built-ins here. cd is a built-in
8870     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
8871       // Clumsy but will have to do for now.
8872       // Chdir has no effect on the parent if run in the child.
8873       buf[strlen(buf)-1] = 0;  // chop \n
8874       if(chdir(buf+3) < 0)
8875         printf(2, "cannot cd %s\n", buf+3);
8876       continue;
8877     }
8878 #ifdef USE_BUILTINS
8879     if (buf[0]=='_') {     // assume it is a builtin command
8880       dobuiltin(buf);
8881       continue;
8882     }
8883 #endif
8884     if(fork1() == 0)
8885       runcmd(parsecmd(buf));
8886     wait();
8887   }
8888   exit();
8889 }
8890
8891 void
8892 panic(char *s)
8893 {
8894   printf(2, "%s\n", s);
8895   exit();
8896 }
8897
8898
8899
```

```
8900 int
8901 fork1(void)
8902 {
8903   int pid;
8904
8905   pid = fork();
8906   if(pid == -1)
8907     panic("fork");
8908   return pid;
8909 }
8910
8911
8912
8913
8914
8915
8916
8917
8918
8919
8920
8921
8922
8923
8924
8925
8926
8927
8928
8929
8930
8931
8932
8933
8934
8935
8936
8937
8938
8939
8940
8941
8942
8943
8944
8945
8946
8947
8948
8949
```



```
8950 // Constructors
8951
8952 struct cmd*
8953 execcmd(void)
8954 {
8955   struct execcmd *cmd;
8956
8957   cmd = malloc(sizeof(*cmd));
8958   memset(cmd, 0, sizeof(*cmd));
8959   cmd->type = EXEC;
8960   return (struct cmd*)cmd;
8961 }
8962
8963 struct cmd*
8964 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
8965 {
8966   struct redircmd *cmd;
8967
8968   cmd = malloc(sizeof(*cmd));
8969   memset(cmd, 0, sizeof(*cmd));
8970   cmd->type = REDIR;
8971   cmd->cmd = subcmd;
8972   cmd->file = file;
8973   cmd->efile = efile;
8974   cmd->mode = mode;
8975   cmd->fd = fd;
8976   return (struct cmd*)cmd;
8977 }
8978
8979 struct cmd*
8980 pipecmd(struct cmd *left, struct cmd *right)
8981 {
8982   struct pipecmd *cmd;
8983
8984   cmd = malloc(sizeof(*cmd));
8985   memset(cmd, 0, sizeof(*cmd));
8986   cmd->type = PIPE;
8987   cmd->left = left;
8988   cmd->right = right;
8989   return (struct cmd*)cmd;
8990 }
8991
8992
8993
8994
8995
8996
8997
8998
8999
```

```
9000 struct cmd*
9001 listcmd(struct cmd *left, struct cmd *right)
9002 {
9003   struct listcmd *cmd;
9004
9005   cmd = malloc(sizeof(*cmd));
9006   memset(cmd, 0, sizeof(*cmd));
9007   cmd->type = LIST;
9008   cmd->left = left;
9009   cmd->right = right;
9010   return (struct cmd*)cmd;
9011 }
9012
9013 struct cmd*
9014 backcmd(struct cmd *subcmd)
9015 {
9016   struct backcmd *cmd;
9017
9018   cmd = malloc(sizeof(*cmd));
9019   memset(cmd, 0, sizeof(*cmd));
9020   cmd->type = BACK;
9021   cmd->cmd = subcmd;
9022   return (struct cmd*)cmd;
9023 }
9024
9025
9026
9027
9028
9029
9030
9031
9032
9033
9034
9035
9036
9037
9038
9039
9040
9041
9042
9043
9044
9045
9046
9047
9048
9049
```



```
9050 // Parsing
9051
9052 char whitespace[] = " \t\r\n\v";
9053 char symbols[] = "<|>&;()";
9054
9055 int
9056 gettoken(char **ps, char *es, char **q, char **eq)
9057 {
9058   char *s;
9059   int ret;
9060
9061   s = *ps;
9062   while(s < es && strchr(whitespace, *s))
9063     s++;
9064   if(q)
9065     *q = s;
9066   ret = *s;
9067   switch(*s){
9068   case 0:
9069     break;
9070   case '|':
9071   case '(':
9072   case ')':
9073   case ';':
9074   case '&':
9075   case '<':
9076     s++;
9077     break;
9078   case '>':
9079     s++;
9080     if(*s == '>'){
9081       ret = '+';
9082       s++;
9083     }
9084     break;
9085   default:
9086     ret = 'a';
9087     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
9088       s++;
9089     break;
9090   }
9091   if(eq)
9092     *eq = s;
9093
9094   while(s < es && strchr(whitespace, *s))
9095     s++;
9096   *ps = s;
9097   return ret;
9098 }
9099
```

```
9100 int
9101 peek(char **ps, char *es, char *toks)
9102 {
9103   char *s;
9104
9105   s = *ps;
9106   while(s < es && strchr(whitespace, *s))
9107     s++;
9108   *ps = s;
9109   return *s && strchr(toks, *s);
9110 }
9111
9112 struct cmd *parseline(char**, char*);
9113 struct cmd *parsepipe(char**, char*);
9114 struct cmd *parseexec(char**, char*);
9115 struct cmd *nulterminate(struct cmd*);
9116
9117 struct cmd*
9118 parsecmd(char *s)
9119 {
9120   char *es;
9121   struct cmd *cmd;
9122
9123   es = s + strlen(s);
9124   cmd = parseline(&s, es);
9125   peek(&s, es, "");
9126   if(s != es){
9127     printf(2, "leftovers: %s\n", s);
9128     panic("syntax");
9129   }
9130   nulterminate(cmd);
9131   return cmd;
9132 }
9133
9134 struct cmd*
9135 parseline(char **ps, char *es)
9136 {
9137   struct cmd *cmd;
9138
9139   cmd = parsepipe(ps, es);
9140   while(peek(ps, es, "&")){
9141     gettoken(ps, es, 0, 0);
9142     cmd = backcmd(cmd);
9143   }
9144   if(peek(ps, es, ";")){
9145     gettoken(ps, es, 0, 0);
9146     cmd = listcmd(cmd, parseline(ps, es));
9147   }
9148   return cmd;
9149 }
```



```
9150 struct cmd*
9151 parsepipe(char **ps, char *es)
9152 {
9153   struct cmd *cmd;
9154
9155   cmd = parseexec(ps, es);
9156   if(peek(ps, es, "|")){
9157     gettoken(ps, es, 0, 0);
9158     cmd = pipecmd(cmd, parsepipe(ps, es));
9159   }
9160   return cmd;
9161 }
9162
9163 struct cmd*
9164 parseredirs(struct cmd *cmd, char **ps, char *es)
9165 {
9166   int tok;
9167   char *q, *eq;
9168
9169   while(peek(ps, es, "<>")){
9170     tok = gettoken(ps, es, 0, 0);
9171     if(gettoken(ps, es, &q, &eq) != 'a')
9172       panic("missing file for redirection");
9173     switch(tok){
9174     case '<':
9175       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
9176       break;
9177     case '>':
9178       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9179       break;
9180     case '+':  // >>
9181       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9182       break;
9183     }
9184   }
9185   return cmd;
9186 }
9187
9188
9189
9190
9191
9192
9193
9194
9195
9196
9197
9198
9199
```

```
9200 struct cmd*
9201 parseblock(char **ps, char *es)
9202 {
9203   struct cmd *cmd;
9204
9205   if(!peek(ps, es, "("))
9206     panic("parseblock");
9207   gettoken(ps, es, 0, 0);
9208   cmd = parseline(ps, es);
9209   if(!peek(ps, es, ")"))
9210     panic("syntax - missing )");
9211   gettoken(ps, es, 0, 0);
9212   cmd = parseredirs(cmd, ps, es);
9213   return cmd;
9214 }
9215
9216 struct cmd*
9217 parseexec(char **ps, char *es)
9218 {
9219   char *q, *eq;
9220   int tok, argc;
9221   struct execcmd *cmd;
9222   struct cmd *ret;
9223
9224   if(peek(ps, es, "("))
9225     return parseblock(ps, es);
9226
9227   ret = execcmd();
9228   cmd = (struct execcmd*)ret;
9229
9230   argc = 0;
9231   ret = parseredirs(ret, ps, es);
9232   while(!peek(ps, es, "|)&;")){
9233     if((tok=gettoken(ps, es, &q, &eq)) == 0)
9234       break;
9235     if(tok != 'a')
9236       panic("syntax");
9237     cmd->argv[argc] = q;
9238     cmd->eargv[argc] = eq;
9239     argc++;
9240     if(argc >= MAXARGS)
9241       panic("too many args");
9242     ret = parseredirs(ret, ps, es);
9243   }
9244   cmd->argv[argc] = 0;
9245   cmd->eargv[argc] = 0;
9246   return ret;
9247 }
9248
9249
```



```
9250 // NUL-terminate all the counted strings.
9251 struct cmd*
9252 nulterminate(struct cmd *cmd)
9253 {
9254   int i;
9255   struct backcmd *bcmd;
9256   struct execcmd *ecmd;
9257   struct listcmd *lcmd;
9258   struct pipecmd *pcmd;
9259   struct redircmd *rcmd;
9260
9261   if(cmd == 0)
9262     return 0;
9263
9264   switch(cmd->type){
9265   case EXEC:
9266     ecmd = (struct execcmd*)cmd;
9267     for(i=0; ecmd->argv[i]; i++)
9268       *ecmd->eargv[i] = 0;
9269     break;
9270
9271   case REDIR:
9272     rcmd = (struct redircmd*)cmd;
9273     nulterminate(rcmd->cmd);
9274     *rcmd->efile = 0;
9275     break;
9276
9277   case PIPE:
9278     pcmd = (struct pipecmd*)cmd;
9279     nulterminate(pcmd->left);
9280     nulterminate(pcmd->right);
9281     break;
9282
9283   case LIST:
9284     lcmd = (struct listcmd*)cmd;
9285     nulterminate(lcmd->left);
9286     nulterminate(lcmd->right);
9287     break;
9288
9289   case BACK:
9290     bcmd = (struct backcmd*)cmd;
9291     nulterminate(bcmd->cmd);
9292     break;
9293   }
9294   return cmd;
9295 }
9296
9297
9298
9299
```

```
9300 #include "asm.h"
9301 #include "memlayout.h"
9302 #include "mmu.h"
9303
9304 # Start the first CPU: switch to 32-bit protected mode, jump into C.
9305 # The BIOS loads this code from the first sector of the hard disk into
9306 # memory at physical address 0x7c00 and starts executing in real mode
9307 # with %cs=0 %ip=7c00.
9308
9309 .code16                       # Assemble for 16-bit mode
9310 .globl start
9311 start:
9312   cli                         # BIOS enabled interrupts; disable
9313
9314   # Zero data segment registers DS, ES, and SS.
9315   xorw    %ax,%ax             # Set %ax to zero
9316   movw    %ax,%ds             # -> Data Segment
9317   movw    %ax,%es             # -> Extra Segment
9318   movw    %ax,%ss             # -> Stack Segment
9319
9320   # Physical address line A20 is tied to zero so that the first PCs
9321   # with 2 MB would run software that assumed 1 MB.  Undo that.
9322 seta20.1:
9323   inb     $0x64,%al               # Wait for not busy
9324   testb   $0x2,%al
9325   jnz     seta20.1
9326
9327   movb    $0xd1,%al               # 0xd1 -> port 0x64
9328   outb    %al,$0x64
9329
9330 seta20.2:
9331   inb     $0x64,%al               # Wait for not busy
9332   testb   $0x2,%al
9333   jnz     seta20.2
9334
9335   movb    $0xdf,%al               # 0xdf -> port 0x60
9336   outb    %al,$0x60
9337
9338   # Switch from real to protected mode.  Use a bootstrap GDT that makes
9339   # virtual addresses map directly to physical addresses so that the
9340   # effective memory map doesn't change during the transition.
9341   lgdt    gdtdesc
9342   movl    %cr0, %eax
9343   orl     $CR0_PE, %eax
9344   movl    %eax, %cr0
9345
9346
9347
9348
9349
```



```
9350   # Complete transition to 32-bit protected mode by using long jmp
9351   # to reload %cs and %eip.  The segment descriptors are set up with no
9352   # translation, so that the mapping is still the identity mapping.
9353   ljmp    $(SEG_KCODE<<3), $start32
9354
9355 .code32  # Tell assembler to generate 32-bit code now.
9356 start32:
9357   # Set up the protected-mode data segment registers
9358   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
9359   movw    %ax, %ds                # -> DS: Data Segment
9360   movw    %ax, %es                # -> ES: Extra Segment
9361   movw    %ax, %ss                # -> SS: Stack Segment
9362   movw    $0, %ax                 # Zero segments not ready for use
9363   movw    %ax, %fs                # -> FS
9364   movw    %ax, %gs                # -> GS
9365
9366   # Set up the stack pointer and call into C.
9367   movl    $start, %esp
9368   call    bootmain
9369
9370   # If bootmain returns (it shouldn't), trigger a Bochs
9371   # breakpoint if running under Bochs, then loop.
9372   movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
9373   movw    %ax, %dx
9374   outw    %ax, %dx
9375   movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
9376   outw    %ax, %dx
9377 spin:
9378   jmp     spin
9379
9380 # Bootstrap GDT
9381 .p2align 2                                # force 4 byte alignment
9382 gdt:
9383   SEG_NULLASM                             # null seg
9384   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
9385   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
9386
9387 gdtdesc:
9388   .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
9389   .long   gdt                             # address gdt
9390
9391
9392
9393
9394
9395
9396
9397
9398
9399
```

```
9400 // Boot loader.
9401 //
9402 // Part of the boot block, along with bootasm.S, which calls bootmain().
9403 // bootasm.S has put the processor into protected 32-bit mode.
9404 // bootmain() loads an ELF kernel image from the disk starting at
9405 // sector 1 and then jumps to the kernel entry routine.
9406
9407 #include "types.h"
9408 #include "elf.h"
9409 #include "x86.h"
9410 #include "memlayout.h"
9411
9412 #define SECTSIZE  512
9413
9414 void readseg(uchar*, uint, uint);
9415
9416 void
9417 bootmain(void)
9418 {
9419   struct elfhdr *elf;
9420   struct proghdr *ph, *eph;
9421   void (*entry)(void);
9422   uchar* pa;
9423
9424   elf = (struct elfhdr*)0x10000;  // scratch space
9425
9426   // Read 1st page off disk
9427   readseg((uchar*)elf, 4096, 0);
9428
9429   // Is this an ELF executable?
9430   if(elf->magic != ELF_MAGIC)
9431     return;  // let bootasm.S handle error
9432
9433   // Load each program segment (ignores ph flags).
9434   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
9435   eph = ph + elf->phnum;
9436   for(; ph < eph; ph++){
9437     pa = (uchar*)ph->paddr;
9438     readseg(pa, ph->filesz, ph->off);
9439     if(ph->memsz > ph->filesz)
9440       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
9441   }
9442
9443   // Call the entry point from the ELF header.
9444   // Does not return!
9445   entry = (void(*)(void))(elf->entry);
9446   entry();
9447 }
9448
9449
```



```
9450 void
9451 waitdisk(void)
9452 {
9453   // Wait for disk ready.
9454   while((inb(0x1F7) & 0xC0) != 0x40)
9455     ;
9456 }
9457
9458 // Read a single sector at offset into dst.
9459 void
9460 readsect(void *dst, uint offset)
9461 {
9462   // Issue command.
9463   waitdisk();
9464   outb(0x1F2, 1);   // count = 1
9465   outb(0x1F3, offset);
9466   outb(0x1F4, offset >> 8);
9467   outb(0x1F5, offset >> 16);
9468   outb(0x1F6, (offset >> 24) | 0xE0);
9469   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
9470
9471   // Read data.
9472   waitdisk();
9473   insl(0x1F0, dst, SECTSIZE/4);
9474 }
9475
9476 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
9477 // Might copy more than asked.
9478 void
9479 readseg(uchar* pa, uint count, uint offset)
9480 {
9481   uchar* epa;
9482
9483   epa = pa + count;
9484
9485   // Round down to sector boundary.
9486   pa -= offset % SECTSIZE;
9487
9488   // Translate from bytes to sectors; kernel starts at sector 1.
9489   offset = (offset / SECTSIZE) + 1;
9490
9491   // If this is too slow, we could read lots of sectors at a time.
9492   // We'd write more to memory than asked, but it doesn't matter --
9493   // we load in increasing order.
9494   for(; pa < epa; pa += SECTSIZE, offset++)
9495     readsect(pa, offset);
9496 }
9497
9498
9499
```

```
#ifdef CS333_P4
// this is an ugly series of if statements but it works
void
print_mode(struct stat* st)
{
  switch (st->type) {
    case T_DIR: printf(1, "d"); break;
    case T_FILE: printf(1, "-"); break;
    case T_DEV: printf(1, "c"); break;
    default: printf(1, "?");
  }

  if (st->mode.flags.u_r)
    printf(1, "r");
  else
    printf(1, "-");

  if (st->mode.flags.u_w)
    printf(1, "w");
  else
    printf(1, "-");

  if ((st->mode.flags.u_x) & (st->mode.flags.setuid))
    printf(1, "S");
  else if (st->mode.flags.u_x)
    printf(1, "x");
  else
    printf(1, "-");

  if (st->mode.flags.g_r)
    printf(1, "r");
  else
    printf(1, "-");

  if (st->mode.flags.g_w)
    printf(1, "w");
  else
    printf(1, "-");

  if (st->mode.flags.g_x)
    printf(1, "x");
  else
    printf(1, "-");

  if (st->mode.flags.o_r)
    printf(1, "r");
  else
    printf(1, "-");

```



```
  if (st->mode.flags.o_w)
    printf(1, "w");
  else
    printf(1, "-");

  if (st->mode.flags.o_x)
    printf(1, "x");
  else
    printf(1, "-");

  return;
}
#endif
```