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 #ifdef CS333_P2
0311 struct uproc;
0312 #endif
0313
0314
0315 // bio.c
0316 void            binit(void);
0317 struct buf*     bread(uint, uint);
0318 void            brelse(struct buf*);
0319 void            bwrite(struct buf*);
0320 // console.c
0321 void            consoleinit(void);
0322 void            cprintf(char*, ...);
0323 void            consoleintr(int(*)(void));
0324 void            panic(char*) __attribute__((noreturn));
0325
0326 // exec.c
0327 int             exec(char*, char**);
0328
0329 // file.c
0330 struct file*    filealloc(void);
0331 void            fileclose(struct file*);
0332 struct file*    filedup(struct file*);
0333 void            fileinit(void);
0334 int             fileread(struct file*, char*, int n);
0335 int             filestat(struct file*, struct stat*);
0336 int             filewrite(struct file*, char*, int n);
0337
0338 // fs.c
0339 void            readsb(int dev, struct superblock *sb);
0340 int             dirlink(struct inode*, char*, uint);
0341 struct inode*   dirlookup(struct inode*, char*, uint*);
0342 struct inode*   ialloc(uint, short);
0343 struct inode*   idup(struct inode*);
0344 void            iinit(int dev);
0345 void            ilock(struct inode*);
0346 void            iput(struct inode*);
0347 void            iunlock(struct inode*);
0348 void            iunlockput(struct inode*);
0349 void            iupdate(struct inode*);
```



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

```
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 // proc.c
0407 struct proc*    copyproc(struct proc*);
0408 void            exit(void);
0409 int             fork(void);
0410 int             growproc(int);
0411 int             kill(int);
0412 void            pinit(void);
0413 void            procdump(void);
0414 void            scheduler(void) __attribute__((noreturn));
0415 void            sched(void);
0416 void            sleep(void*, struct spinlock*);
0417 void            userinit(void);
0418 int             wait(void);
0419 void            wakeup(void*);
0420 void            yield(void);
0421 #ifdef CS333_P2
0422 int                                                 getprocs(uint max,
0423 #endif
0424
0425 // swtch.S
0426 void            swtch(struct context**, struct context*);
0427
0428 // spinlock.c
0429 void            acquire(struct spinlock*);
0430 void            getcallerpcs(void*, uint*);
0431 int             holding(struct spinlock*);
0432 void            initlock(struct spinlock*, char*);
0433 void            release(struct spinlock*);
0434 void            pushcli(void);
0435 void            popcli(void);
0436
0437 // string.c
0438 int             memcmp(const void*, const void*, uint);
0439 void*           memmove(void*, const void*, uint);
0440 void*           memset(void*, int, uint);
0441 char*           safestrcpy(char*, const char*, int);
0442 int             strlen(const char*);
0443 int             strncmp(const char*, const char*, uint);
0444 char*           strncpy(char*, const char*, int);
0445
0446
0447
0448
0449
```



```
0450 // syscall.c
0451 int             argint(int, int*);
0452 int             argptr(int, char**, int);
0453 int             argstr(int, char**);
0454 int             fetchint(uint, int*);
0455 int             fetchstr(uint, char**);
0456 void            syscall(void);
0457
0458 // timer.c
0459 void            timerinit(void);
0460
0461 // trap.c
0462 void            idtinit(void);
0463 extern uint     ticks;
0464 void            tvinit(void);
0465 extern struct spinlock tickslock;
0466
0467 // uart.c
0468 void            uartinit(void);
0469 void            uartintr(void);
0470 void            uartputc(int);
0471
0472 // vm.c
0473 void            seginit(void);
0474 void            kvmalloc(void);
0475 void            vmenable(void);
0476 pde_t*          setupkvm(void);
0477 char*           uva2ka(pde_t*, char*);
0478 int             allocuvm(pde_t*, uint, uint);
0479 int             deallocuvm(pde_t*, uint, uint);
0480 void            freevm(pde_t*);
0481 void            inituvm(pde_t*, char*, uint);
0482 int             loaduvm(pde_t*, char*, struct inode*, uint, uint);
0483 pde_t*          copyuvm(pde_t*, uint);
0484 void            switchuvm(struct proc*);
0485 void            switchkvm(void);
0486 int             copyout(pde_t*, uint, void*, uint);
0487 void            clearpteu(pde_t *pgdir, char *uva);
0488
0489 // number of elements in fixed-size array
0490 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
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   ljmpl    $(SEG_KCODE<<3), $(start32)
1189
1190 .code32
1191 start32:
1192   movw    $(SEG_KDATA<<3), %ax
1193   movw    %ax, %ds
1194   movw    %ax, %es
1195   movw    %ax, %ss
1196   movw    $0, %ax
1197   movw    %ax, %fs
1198   movw    %ax, %gs
1199
```

```
1200   # Turn on page size extension for 4Mbyte pages
1201   movl    %cr4, %eax
1202   orl     $(CR4_PSE), %eax
1203   movl    %eax, %cr4
1204   # Use enterpgdir as our initial page table
1205   movl    (start-12), %eax
1206   movl    %eax, %cr3
1207   # Turn on paging.
1208   movl    %cr0, %eax
1209   orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
1210   movl    %eax, %cr0
1211
1212   # Switch to the stack allocated by startothers()
1213   movl    (start-4), %esp
1214   # Call mpenter()
1215   call      *(start-8)
1216
1217   movw    $0x8a00, %ax
1218   movw    %ax, %dx
1219   outw    %ax, %dx
1220   movw    $0x8ae0, %ax
1221   outw    %ax, %dx
1222 spin:
1223   jmp     spin
1224
1225 .p2align 2
1226 gdt:
1227   SEG_NULLASM
1228   SEG_ASM(STA_X|STA_R, 0, 0xffffffff)
1229   SEG_ASM(STA_W, 0, 0xffffffff)
1230
1231
1232 gdtdesc:
1233   .word   (gdtdesc - gdt - 1)
1234   .long   gdt
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
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 // Blank page.
1369 // Blank page.
1370 // Blank page.
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 // Mutual exclusion lock.
1401 struct spinlock {
1402   uint locked;       // Is the lock held?
1403
1404   // For debugging:
1405   char *name;        // Name of lock.
1406   struct cpu *cpu;   // The cpu holding the lock.
1407   uint pcs[10];      // The call stack (an array of program counters)
1408                      // that locked the lock.
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 // Mutual exclusion spin locks.
1451
1452 #include "types.h"
1453 #include "defs.h"
1454 #include "param.h"
1455 #include "x86.h"
1456 #include "memlayout.h"
1457 #include "mmu.h"
1458 #include "proc.h"
1459 #include "spinlock.h"
1460
1461 void
1462 initlock(struct spinlock *lk, char *name)
1463 {
1464   lk->name = name;
1465   lk->locked = 0;
1466   lk->cpu = 0;
1467 }
1468
1469 // Acquire the lock.
1470 // Loops (spins) until the lock is acquired.
1471 // Holding a lock for a long time may cause
1472 // other CPUs to waste time spinning to acquire it.
1473 void
1474 acquire(struct spinlock *lk)
1475 {
1476   pushcli(); // disable interrupts to avoid deadlock.
1477   if(holding(lk))
1478     panic("acquire");
1479
1480   // The xchg is atomic.
1481   // It also serializes, so that reads after acquire are not
1482   // reordered before it.
1483   while(xchg(&lk->locked, 1) != 0)
1484     ;
1485
1486   // Record info about lock acquisition for debugging.
1487   lk->cpu = cpu;
1488   getcallerpcs(&lk, lk->pcs);
1489 }
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
```

```
1500 // Release the lock.
1501 void
1502 release(struct spinlock *lk)
1503 {
1504   if(!holding(lk))
1505     panic("release");
1506
1507   lk->pcs[0] = 0;
1508   lk->cpu = 0;
1509
1510   // The xchg serializes, so that reads before release are
1511   // not reordered after it.  The 1996 PentiumPro manual (Volume 3,
1512   // 7.2) says reads can be carried out speculatively and in
1513   // any order, which implies we need to serialize here.
1514   // But the 2007 Intel 64 Architecture Memory Ordering White
1515   // Paper says that Intel 64 and IA-32 will not move a load
1516   // after a store. So lock->locked = 0 would work here.
1517   // The xchg being asm volatile ensures gcc emits it after
1518   // the above assignments (and after the critical section).
1519   xchg(&lk->locked, 0);
1520
1521   popcli();
1522 }
1523
1524 // Record the current call stack in pcs[] by following the %ebp chain.
1525 void
1526 getcallerpcs(void *v, uint pcs[])
1527 {
1528   uint *ebp;
1529   int i;
1530
1531   ebp = (uint*)v - 2;
1532   for(i = 0; i < 10; i++){
1533     if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
1534       break;
1535     pcs[i] = ebp[1];     // saved %eip
1536     ebp = (uint*)ebp[0]; // saved %ebp
1537   }
1538   for(; i < 10; i++)
1539     pcs[i] = 0;
1540 }
1541
1542 // Check whether this cpu is holding the lock.
1543 int
1544 holding(struct spinlock *lock)
1545 {
1546   return lock->locked && lock->cpu == cpu;
1547 }
1548
1549
```



```
1550 // Pushcli/popcli are like cli/sti except that they are matched:
1551 // it takes two popcli to undo two pushcli.  Also, if interrupts
1552 // are off, then pushcli, popcli leaves them off.
1553
1554 void
1555 pushcli(void)
1556 {
1557   int eflags;
1558
1559   eflags = readeflags();
1560   cli();
1561   if(cpu->ncli++ == 0)
1562     cpu->intena = eflags & FL_IF;
1563 }
1564
1565 void
1566 popcli(void)
1567 {
1568   if(readeflags()&FL_IF)
1569     panic("popcli - interruptible");
1570   if(--cpu->ncli < 0)
1571     panic("popcli");
1572   if(cpu->ncli == 0 && cpu->intena)
1573     sti();
1574 }
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
```

```
1600 #include "param.h"
1601 #include "types.h"
1602 #include "defs.h"
1603 #include "x86.h"
1604 #include "memlayout.h"
1605 #include "mmu.h"
1606 #include "proc.h"
1607 #include "elf.h"
1608
1609 extern char data[];  // defined by kernel.ld
1610 pde_t *kpgdir;  // for use in scheduler()
1611 struct segdesc gdt[NSEGS];
1612
1613 // Set up CPU's kernel segment descriptors.
1614 // Run once on entry on each CPU.
1615 void
1616 seginit(void)
1617 {
1618   struct cpu *c;
1619
1620   // Map "logical" addresses to virtual addresses using identity map.
1621   // Cannot share a CODE descriptor for both kernel and user
1622   // because it would have to have DPL_USR, but the CPU forbids
1623   // an interrupt from CPL=0 to DPL=3.
1624   c = &cpus[cpunum()];
1625   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
1626   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1627   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
1628   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1629
1630   // Map cpu, and curproc
1631   c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
1632
1633   lgdt(c->gdt, sizeof(c->gdt));
1634   loadgs(SEG_KCPU << 3);
1635
1636   // Initialize cpu-local storage.
1637   cpu = c;
1638   proc = 0;
1639 }
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
```



```
1650 // Return the address of the PTE in page table pgdir
1651 // that corresponds to virtual address va.  If alloc!=0,
1652 // create any required page table pages.
1653 static pte_t *
1654 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1655 {
1656   pde_t *pde;
1657   pte_t *pgtab;
1658
1659   pde = &pgdir[PDX(va)];
1660   if(*pde & PTE_P){
1661     pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
1662   } else {
1663     if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1664       return 0;
1665     // Make sure all those PTE_P bits are zero.
1666     memset(pgtab, 0, PGSIZE);
1667     // The permissions here are overly generous, but they can
1668     // be further restricted by the permissions in the page table
1669     // entries, if necessary.
1670     *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
1671   }
1672   return &pgtab[PTX(va)];
1673 }
1674
1675 // Create PTEs for virtual addresses starting at va that refer to
1676 // physical addresses starting at pa. va and size might not
1677 // be page-aligned.
1678 static int
1679 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1680 {
1681   char *a, *last;
1682   pte_t *pte;
1683
1684   a = (char*)PGROUNDDOWN((uint)va);
1685   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1686   for(;;){
1687     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1688       return -1;
1689     if(*pte & PTE_P)
1690       panic("remap");
1691     *pte = pa | perm | PTE_P;
1692     if(a == last)
1693       break;
1694     a += PGSIZE;
1695     pa += PGSIZE;
1696   }
1697   return 0;
1698 }
1699
```

```
1700 // There is one page table per process, plus one that's used when
1701 // a CPU is not running any process (kpgdir). The kernel uses the
1702 // current process's page table during system calls and interrupts;
1703 // page protection bits prevent user code from using the kernel's
1704 // mappings.
1705 //
1706 // setupkvm() and exec() set up every page table like this:
1707 //
1708 //   0..KERNBASE: user memory (text+data+stack+heap), mapped to
1709 //                phys memory allocated by the kernel
1710 //   KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
1711 //   KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
1712 //                for the kernel's instructions and r/o data
1713 //   data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
1714 //                                  rw data + free physical memory
1715 //   0xfe000000..0: mapped direct (devices such as ioapic)
1716 //
1717 // The kernel allocates physical memory for its heap and for user memory
1718 // between V2P(end) and the end of physical memory (PHYSTOP)
1719 // (directly addressable from end..P2V(PHYSTOP)).
1720
1721 // This table defines the kernel's mappings, which are present in
1722 // every process's page table.
1723 static struct kmap {
1724   void *virt;
1725   uint phys_start;
1726   uint phys_end;
1727   int perm;
1728 } kmap[] = {
1729  { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
1730  { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
1731  { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
1732  { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
1733 };
1734
1735 // Set up kernel part of a page table.
1736 pde_t*
1737 setupkvm(void)
1738 {
1739   pde_t *pgdir;
1740   struct kmap *k;
1741
1742   if((pgdir = (pde_t*)kalloc()) == 0)
1743     return 0;
1744   memset(pgdir, 0, PGSIZE);
1745   if (p2v(PHYSTOP) > (void*)DEVSPACE)
1746     panic("PHYSTOP too high");
1747   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1748     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1749                 (uint)k->phys_start, k->perm) < 0)
```



```
1750       return 0;
1751   return pgdir;
1752 }
1753
1754 // Allocate one page table for the machine for the kernel address
1755 // space for scheduler processes.
1756 void
1757 kvmalloc(void)
1758 {
1759   kpgdir = setupkvm();
1760   switchkvm();
1761 }
1762
1763 // Switch h/w page table register to the kernel-only page table,
1764 // for when no process is running.
1765 void
1766 switchkvm(void)
1767 {
1768   lcr3(v2p(kpgdir));   // switch to the kernel page table
1769 }
1770
1771 // Switch TSS and h/w page table to correspond to process p.
1772 void
1773 switchuvm(struct proc *p)
1774 {
1775   pushcli();
1776   cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
1777   cpu->gdt[SEG_TSS].s = 0;
1778   cpu->ts.ss0 = SEG_KDATA << 3;
1779   cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
1780   ltr(SEG_TSS << 3);
1781   if(p->pgdir == 0)
1782     panic("switchuvm: no pgdir");
1783   lcr3(v2p(p->pgdir));  // switch to new address space
1784   popcli();
1785 }
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
```

```
1800 // Load the initcode into address 0 of pgdir.
1801 // sz must be less than a page.
1802 void
1803 inituvm(pde_t *pgdir, char *init, uint sz)
1804 {
1805   char *mem;
1806
1807   if(sz >= PGSIZE)
1808     panic("inituvm: more than a page");
1809   mem = kalloc();
1810   memset(mem, 0, PGSIZE);
1811   mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
1812   memmove(mem, init, sz);
1813 }
1814
1815 // Load a program segment into pgdir.  addr must be page-aligned
1816 // and the pages from addr to addr+sz must already be mapped.
1817 int
1818 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
1819 {
1820   uint i, pa, n;
1821   pte_t *pte;
1822
1823   if((uint) addr % PGSIZE != 0)
1824     panic("loaduvm: addr must be page aligned");
1825   for(i = 0; i < sz; i += PGSIZE){
1826     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1827       panic("loaduvm: address should exist");
1828     pa = PTE_ADDR(*pte);
1829     if(sz - i < PGSIZE)
1830       n = sz - i;
1831     else
1832       n = PGSIZE;
1833     if(readi(ip, p2v(pa), offset+i, n) != n)
1834       return -1;
1835   }
1836   return 0;
1837 }
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
```



```
1850 // Allocate page tables and physical memory to grow process from oldsz to
1851 // newsz, which need not be page aligned.  Returns new size or 0 on error.
1852 int
1853 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1854 {
1855   char *mem;
1856   uint a;
1857
1858   if(newsz >= KERNBASE)
1859     return 0;
1860   if(newsz < oldsz)
1861     return oldsz;
1862
1863   a = PGROUNDUP(oldsz);
1864   for(; a < newsz; a += PGSIZE){
1865     mem = kalloc();
1866     if(mem == 0){
1867       cprintf("allocuvm out of memory\n");
1868       deallocuvm(pgdir, newsz, oldsz);
1869       return 0;
1870     }
1871     memset(mem, 0, PGSIZE);
1872     mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U);
1873   }
1874   return newsz;
1875 }
1876
1877 // Deallocate user pages to bring the process size from oldsz to
1878 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
1879 // need to be less than oldsz.  oldsz can be larger than the actual
1880 // process size.  Returns the new process size.
1881 int
1882 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1883 {
1884   pte_t *pte;
1885   uint a, pa;
1886
1887   if(newsz >= oldsz)
1888     return oldsz;
1889
1890   a = PGROUNDUP(newsz);
1891   for(; a  < oldsz; a += PGSIZE){
1892     pte = walkpgdir(pgdir, (char*)a, 0);
1893     if(!pte)
1894       a += (NPTENTRIES - 1) * PGSIZE;
1895     else if((*pte & PTE_P) != 0){
1896       pa = PTE_ADDR(*pte);
1897       if(pa == 0)
1898         panic("kfree");
1899       char *v = p2v(pa);
```

```
1900       kfree(v);
1901       *pte = 0;
1902     }
1903   }
1904   return newsz;
1905 }
1906
1907 // Free a page table and all the physical memory pages
1908 // in the user part.
1909 void
1910 freevm(pde_t *pgdir)
1911 {
1912   uint i;
1913
1914   if(pgdir == 0)
1915     panic("freevm: no pgdir");
1916   deallocuvm(pgdir, KERNBASE, 0);
1917   for(i = 0; i < NPDENTRIES; i++){
1918     if(pgdir[i] & PTE_P){
1919       char * v = p2v(PTE_ADDR(pgdir[i]));
1920       kfree(v);
1921     }
1922   }
1923   kfree((char*)pgdir);
1924 }
1925
1926 // Clear PTE_U on a page. Used to create an inaccessible
1927 // page beneath the user stack.
1928 void
1929 clearpteu(pde_t *pgdir, char *uva)
1930 {
1931   pte_t *pte;
1932
1933   pte = walkpgdir(pgdir, uva, 0);
1934   if(pte == 0)
1935     panic("clearpteu");
1936   *pte &= ~PTE_U;
1937 }
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
```



```
1950 // Given a parent process's page table, create a copy
1951 // of it for a child.
1952 pde_t*
1953 copyuvm(pde_t *pgdir, uint sz)
1954 {
1955   pde_t *d;
1956   pte_t *pte;
1957   uint pa, i, flags;
1958   char *mem;
1959
1960   if((d = setupkvm()) == 0)
1961     return 0;
1962   for(i = 0; i < sz; i += PGSIZE){
1963     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
1964       panic("copyuvm: pte should exist");
1965     if(!(*pte & PTE_P))
1966       panic("copyuvm: page not present");
1967     pa = PTE_ADDR(*pte);
1968     flags = PTE_FLAGS(*pte);
1969     if((mem = kalloc()) == 0)
1970       goto bad;
1971     memmove(mem, (char*)p2v(pa), PGSIZE);
1972     if(mappages(d, (void*)i, PGSIZE, v2p(mem), flags) < 0)
1973       goto bad;
1974   }
1975   return d;
1976
1977 bad:
1978   freevm(d);
1979   return 0;
1980 }
1981
1982 // Map user virtual address to kernel address.
1983 char*
1984 uva2ka(pde_t *pgdir, char *uva)
1985 {
1986   pte_t *pte;
1987
1988   pte = walkpgdir(pgdir, uva, 0);
1989   if((*pte & PTE_P) == 0)
1990     return 0;
1991   if((*pte & PTE_U) == 0)
1992     return 0;
1993   return (char*)p2v(PTE_ADDR(*pte));
1994 }
1995
1996
1997
1998
1999
```

```
2000 // Copy len bytes from p to user address va in page table pgdir.
2001 // Most useful when pgdir is not the current page table.
2002 // uva2ka ensures this only works for PTE_U pages.
2003 int
2004 copyout(pde_t *pgdir, uint va, void *p, uint len)
2005 {
2006   char *buf, *pa0;
2007   uint n, va0;
2008
2009   buf = (char*)p;
2010   while(len > 0){
2011     va0 = (uint)PGROUNDDOWN(va);
2012     pa0 = uva2ka(pgdir, (char*)va0);
2013     if(pa0 == 0)
2014       return -1;
2015     n = PGSIZE - (va - va0);
2016     if(n > len)
2017       n = len;
2018     memmove(pa0 + (va - va0), buf, n);
2019     len -= n;
2020     buf += n;
2021     va = va0 + PGSIZE;
2022   }
2023   return 0;
2024 }
2025
2026 // Blank page.
2027 // Blank page.
2028 // Blank page.
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
```



```
2050 // Segments in proc->gdt.
2051 #define NSEGS     7
2052
2053 // Default UID and GID for init
2054 #define INITUID     0
2055 #define INITGID     0
2056
2057 // Per-CPU state
2058 struct cpu {
2059   uchar id;                    // Local APIC ID; index into cpus[] below
2060   struct context *scheduler;   // swtch() here to enter scheduler
2061   struct taskstate ts;         // Used by x86 to find stack for interrupt
2062   struct segdesc gdt[NSEGS];   // x86 global descriptor table
2063   volatile uint started;       // Has the CPU started?
2064   int ncli;                    // Depth of pushcli nesting.
2065   int intena;                  // Were interrupts enabled before pushcli?
2066
2067   // Cpu-local storage variables; see below
2068   struct cpu *cpu;
2069   struct proc *proc;           // The currently-running process.
2070 };
2071
2072 extern struct cpu cpus[NCPU];
2073 extern int ncpu;
2074
2075 // Per-CPU variables, holding pointers to the
2076 // current cpu and to the current process.
2077 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2078 // and "%gs:4" to refer to proc.  seginit sets up the
2079 // %gs segment register so that %gs refers to the memory
2080 // holding those two variables in the local cpu's struct cpu.
2081 // This is similar to how thread-local variables are implemented
2082 // in thread libraries such as Linux pthreads.
2083 extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
2084 extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc
2085
2086 // Saved registers for kernel context switches.
2087 // Don't need to save all the segment registers (%cs, etc),
2088 // because they are constant across kernel contexts.
2089 // Don't need to save %eax, %ecx, %edx, because the
2090 // x86 convention is that the caller has saved them.
2091 // Contexts are stored at the bottom of the stack they
2092 // describe; the stack pointer is the address of the context.
2093 // The layout of the context matches the layout of the stack in swtch.S
2094 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2095 // but it is on the stack and allocproc() manipulates it.
2096 struct context {
2097   uint edi;
2098   uint esi;
2099   uint ebx;
```

```
2100   uint ebp;
2101   uint eip;
2102 };
2103
2104 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2105
2106 // Per-process state
2107 struct proc {
2108   uint sz;                     // Size of process memory (bytes)
2109   pde_t* pgdir;                // Page table
2110   char *kstack;                // Bottom of kernel stack for this process
2111   enum procstate state;        // Process state
2112   uint pid;                    // Process ID
2113   struct proc *parent;         // Parent process
2114   struct trapframe *tf;        // Trap frame for current syscall
2115   struct context *context;     // swtch() here to run process
2116   void *chan;                  // If non-zero, sleeping on chan
2117   int killed;                  // If non-zero, have been killed
2118   struct file *ofile[NOFILE];  // Open files
2119   struct inode *cwd;           // Current directory
2120   char name[16];               // Process name (debugging)
2121   uint start_ticks;                  // Start ticks (debugging)
2122 #ifdef CS333_P2
2123    uint cpu_ticks_total;                          // Total elapsed ticks i
2124    uint cpu_ticks_in;                             // Ticks when scheduled
2125   uint uid;                    // Process owner's user id
2126   uint gid;                    // Process owner's group id
2127 #endif
2128 };
2129
2130 // Process memory is laid out contiguously, low addresses first:
2131 //   text
2132 //   original data and bss
2133 //   fixed-size stack
2134 //   expandable heap
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
```



```
2150 #include "types.h"
2151 #include "defs.h"
2152 #include "param.h"
2153 #include "memlayout.h"
2154 #include "mmu.h"
2155 #include "x86.h"
2156 #include "proc.h"
2157 #include "spinlock.h"
2158 #include "uproc.h"
2159
2160 struct {
2161   struct spinlock lock;
2162   struct proc proc[NPROC];
2163 } ptable;
2164
2165 static struct proc *initproc;
2166
2167 int nextpid = 1;
2168 extern void forkret(void);
2169 extern void trapret(void);
2170
2171 static void wakeup1(void *chan);
2172
2173 void
2174 pinit(void)
2175 {
2176   initlock(&ptable.lock, "ptable");
2177 }
2178
2179 // Look in the process table for an UNUSED proc.
2180 // If found, change state to EMBRYO and initialize
2181 // state required to run in the kernel.
2182 // Otherwise return 0.
2183 static struct proc*
2184 allocproc(void)
2185 {
2186   struct proc *p;
2187   char *sp;
2188
2189   acquire(&ptable.lock);
2190   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2191     if(p->state == UNUSED)
2192       goto found;
2193   release(&ptable.lock);
2194   return 0;
2195
2196 found:
2197   p->state = EMBRYO;
2198   p->pid = nextpid++;
2199   release(&ptable.lock);
```

```
2200   // Allocate kernel stack.
2201   if((p->kstack = kalloc()) == 0){
2202     p->state = UNUSED;
2203     return 0;
2204   }
2205   sp = p->kstack + KSTACKSIZE;
2206
2207   // Leave room for trap frame.
2208   sp -= sizeof *p->tf;
2209   p->tf = (struct trapframe*)sp;
2210
2211   // Set up new context to start executing at forkret,
2212   // which returns to trapret.
2213   sp -= 4;
2214   *(uint*)sp = (uint)trapret;
2215
2216   sp -= sizeof *p->context;
2217   p->context = (struct context*)sp;
2218   memset(p->context, 0, sizeof *p->context);
2219   p->context->eip = (uint)forkret;
2220
2221   acquire(&tickslock);
2222   p->start_ticks = ticks;
2223   release(&tickslock);
2224    p->cpu_ticks_in = 0;
2225    p->cpu_ticks_in = 0;
2226
2227   return p;
2228 }
2229
2230 // Set up first user process.
2231 void
2232 userinit(void)
2233 {
2234   struct proc *p;
2235   extern char _binary_initcode_start[], _binary_initcode_size[];
2236
2237   p = allocproc();
2238   initproc = p;
2239   if((p->pgdir = setupkvm()) == 0)
2240     panic("userinit: out of memory?");
2241   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2242   p->sz = PGSIZE;
2243   memset(p->tf, 0, sizeof(*p->tf));
2244   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2245   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2246   p->tf->es = p->tf->ds;
2247   p->tf->ss = p->tf->ds;
2248   p->tf->eflags = FL_IF;
2249   p->tf->esp = PGSIZE;
```



```
2250   p->tf->eip = 0;  // beginning of initcode.S
2251
2252   safestrcpy(p->name, "initcode", sizeof(p->name));
2253   p->cwd = namei("/");
2254
2255   p->state = RUNNABLE;
2256
2257 #ifdef CS333_P2
2258    p->uid = INITUID;
2259    p->gid = INITGID;
2260 #endif
2261 }
2262
2263 // Grow current process's memory by n bytes.
2264 // Return 0 on success, -1 on failure.
2265 int
2266 growproc(int n)
2267 {
2268   uint sz;
2269
2270   sz = proc->sz;
2271   if(n > 0){
2272     if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2273       return -1;
2274   } else if(n < 0){
2275     if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2276       return -1;
2277   }
2278   proc->sz = sz;
2279   switchuvm(proc);
2280   return 0;
2281 }
2282
2283 // Create a new process copying p as the parent.
2284 // Sets up stack to return as if from system call.
2285 // Caller must set state of returned proc to RUNNABLE.
2286 int
2287 fork(void)
2288 {
2289   int i, pid;
2290   struct proc *np;
2291
2292   // Allocate process.
2293   if((np = allocproc()) == 0)
2294     return -1;
2295
2296
2297
2298
2299
```

```
2300   // Copy process state from p.
2301   if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2302     kfree(np->kstack);
2303     np->kstack = 0;
2304     np->state = UNUSED;
2305     return -1;
2306   }
2307   np->sz = proc->sz;
2308   np->parent = proc;
2309   *np->tf = *proc->tf;
2310
2311 #ifdef CS333_P2
2312    // Copy process UID, GID
2313    np->uid = proc->uid;
2314    np->gid = proc->gid;
2315 #endif
2316
2317   // Clear %eax so that fork returns 0 in the child.
2318   np->tf->eax = 0;
2319
2320   for(i = 0; i < NOFILE; i++)
2321     if(proc->ofile[i])
2322       np->ofile[i] = filedup(proc->ofile[i]);
2323   np->cwd = idup(proc->cwd);
2324
2325   safestrcpy(np->name, proc->name, sizeof(proc->name));
2326
2327   pid = np->pid;
2328
2329   // lock to force the compiler to emit the np->state write last.
2330   acquire(&ptable.lock);
2331   np->state = RUNNABLE;
2332   release(&ptable.lock);
2333
2334   return pid;
2335 }
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
```



```
2350 // Exit the current process.  Does not return.
2351 // An exited process remains in the zombie state
2352 // until its parent calls wait() to find out it exited.
2353 void
2354 exit(void)
2355 {
2356   struct proc *p;
2357   int fd;
2358
2359   if(proc == initproc)
2360     panic("init exiting");
2361
2362   // Close all open files.
2363   for(fd = 0; fd < NOFILE; fd++){
2364     if(proc->ofile[fd]){
2365       fileclose(proc->ofile[fd]);
2366       proc->ofile[fd] = 0;
2367     }
2368   }
2369
2370   begin_op();
2371   iput(proc->cwd);
2372   end_op();
2373   proc->cwd = 0;
2374
2375   acquire(&ptable.lock);
2376
2377   // Parent might be sleeping in wait().
2378   wakeup1(proc->parent);
2379
2380   // Pass abandoned children to init.
2381   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2382     if(p->parent == proc){
2383       p->parent = initproc;
2384       if(p->state == ZOMBIE)
2385         wakeup1(initproc);
2386     }
2387   }
2388
2389   // Jump into the scheduler, never to return.
2390   proc->state = ZOMBIE;
2391   sched();
2392   panic("zombie exit");
2393 }
2394
2395
2396
2397
2398
2399
```

```
2400 // Wait for a child process to exit and return its pid.
2401 // Return -1 if this process has no children.
2402 int
2403 wait(void)
2404 {
2405   struct proc *p;
2406   int havekids, pid;
2407
2408   acquire(&ptable.lock);
2409   for(;;){
2410     // Scan through table looking for zombie children.
2411     havekids = 0;
2412     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2413       if(p->parent != proc)
2414         continue;
2415       havekids = 1;
2416       if(p->state == ZOMBIE){
2417         // Found one.
2418         pid = p->pid;
2419         kfree(p->kstack);
2420         p->kstack = 0;
2421         freevm(p->pgdir);
2422         p->state = UNUSED;
2423         p->pid = 0;
2424         p->parent = 0;
2425         p->name[0] = 0;
2426         p->killed = 0;
2427         release(&ptable.lock);
2428         return pid;
2429       }
2430     }
2431
2432     // No point waiting if we don't have any children.
2433     if(!havekids || proc->killed){
2434       release(&ptable.lock);
2435       return -1;
2436     }
2437
2438     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2439     sleep(proc, &ptable.lock);
2440   }
2441 }
2442
2443
2444
2445
2446
2447
2448
2449
```



```
2450 // Per-CPU process scheduler.
2451 // Each CPU calls scheduler() after setting itself up.
2452 // Scheduler never returns.  It loops, doing:
2453 //  - choose a process to run
2454 //  - swtch to start running that process
2455 //  - eventually that process transfers control
2456 //      via swtch back to the scheduler.
2457 #ifndef CS333_P3
2458 // original xv6 scheduler. Use if CS333_P3 NOT defined.
2459 void
2460 scheduler(void)
2461 {
2462   struct proc *p;
2463
2464   for(;;){
2465     // Enable interrupts on this processor.
2466     sti();
2467
2468     // Loop over process table looking for process to run.
2469     acquire(&ptable.lock);
2470     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2471       if(p->state != RUNNABLE)
2472         continue;
2473
2474       // Switch to chosen process.  It is the process's job
2475       // to release ptable.lock and then reacquire it
2476       // before jumping back to us.
2477       proc = p;
2478       switchuvm(p);
2479       p->state = RUNNING;
2480 #ifdef CS333_P2
2481        acquire(&tickslock);
2482                    p->cpu_ticks_in = ticks;
2483                    release(&tickslock);
2484 #endif
2485       swtch(&cpu->scheduler, proc->context);
2486       switchkvm();
2487
2488       // Process is done running for now.
2489       // It should have changed its p->state before coming back.
2490       proc = 0;
2491     }
2492     release(&ptable.lock);
2493
2494   }
2495 }
2496
2497
2498
2499
```

```
2500 #else
2501 // CS333_P3 MLFQ scheduler implementation goes here
2502 void
2503 scheduler(void)
2504 {
2505
2506 }
2507 #endif
2508
2509 // Enter scheduler.  Must hold only ptable.lock
2510 // and have changed proc->state.
2511 void
2512 sched(void)
2513 {
2514   int intena;
2515
2516   if(!holding(&ptable.lock))
2517     panic("sched ptable.lock");
2518   if(cpu->ncli != 1)
2519     panic("sched locks");
2520   if(proc->state == RUNNING)
2521     panic("sched running");
2522   if(readeflags()&FL_IF)
2523     panic("sched interrible");
2524 #ifdef CS333_P2
2525    acquire(&tickslock);
2526    proc->cpu_ticks_total += ticks - proc->cpu_ticks_in;
2527    release(&tickslock);
2528 #endif
2529   intena = cpu->intena;
2530   swtch(&proc->context, cpu->scheduler);
2531   cpu->intena = intena;
2532 }
2533
2534 // Give up the CPU for one scheduling round.
2535 void
2536 yield(void)
2537 {
2538   acquire(&ptable.lock);
2539   proc->state = RUNNABLE;
2540   sched();
2541   release(&ptable.lock);
2542 }
2543
2544
2545
2546
2547
2548
2549
```



```
2550 // A fork child's very first scheduling by scheduler()
2551 // will swtch here.  "Return" to user space.
2552 void
2553 forkret(void)
2554 {
2555   static int first = 1;
2556   // Still holding ptable.lock from scheduler.
2557   release(&ptable.lock);
2558
2559   if (first) {
2560     // Some initialization functions must be run in the context
2561     // of a regular process (e.g., they call sleep), and thus cannot
2562     // be run from main().
2563     first = 0;
2564     iinit(ROOTDEV);
2565     initlog(ROOTDEV);
2566   }
2567
2568   // Return to "caller", actually trapret (see allocproc).
2569 }
2570
2571 // Atomically release lock and sleep on chan.
2572 // Reacquires lock when awakened.
2573 void
2574 sleep(void *chan, struct spinlock *lk)
2575 {
2576   if(proc == 0)
2577     panic("sleep");
2578
2579   if(lk == 0)
2580     panic("sleep without lk");
2581
2582   // Must acquire ptable.lock in order to
2583   // change p->state and then call sched.
2584   // Once we hold ptable.lock, we can be
2585   // guaranteed that we won't miss any wakeup
2586   // (wakeup runs with ptable.lock locked),
2587   // so it's okay to release lk.
2588   if(lk != &ptable.lock){
2589     acquire(&ptable.lock);
2590     release(lk);
2591   }
2592
2593   // Go to sleep.
2594   proc->chan = chan;
2595   proc->state = SLEEPING;
2596   sched();
2597
2598   // Tidy up.
2599   proc->chan = 0;
```

```
2600   // Reacquire original lock.
2601   if(lk != &ptable.lock){
2602     release(&ptable.lock);
2603     acquire(lk);
2604   }
2605 }
2606
2607 // Wake up all processes sleeping on chan.
2608 // The ptable lock must be held.
2609 static void
2610 wakeup1(void *chan)
2611 {
2612   struct proc *p;
2613
2614   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2615     if(p->state == SLEEPING && p->chan == chan)
2616       p->state = RUNNABLE;
2617 }
2618
2619 // Wake up all processes sleeping on chan.
2620 void
2621 wakeup(void *chan)
2622 {
2623   acquire(&ptable.lock);
2624   wakeup1(chan);
2625   release(&ptable.lock);
2626 }
2627
2628 // Kill the process with the given pid.
2629 // Process won't exit until it returns
2630 // to user space (see trap in trap.c).
2631 int
2632 kill(int pid)
2633 {
2634   struct proc *p;
2635
2636   acquire(&ptable.lock);
2637   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2638     if(p->pid == pid){
2639       p->killed = 1;
2640       // Wake process from sleep if necessary.
2641       if(p->state == SLEEPING)
2642         p->state = RUNNABLE;
2643       release(&ptable.lock);
2644       return 0;
2645     }
2646   }
2647   release(&ptable.lock);
2648   return -1;
2649 }
```



```
2650 // Print a process listing to console.  For debugging.
2651 // Runs when user types ^P on console.
2652 // No lock to avoid wedging a stuck machine further.
2653 static void
2654 print_elapsed(struct proc *p)
2655 {
2656   uint temp = p->start_ticks;
2657   temp = ticks - temp;
2658   cprintf("%d.%d",temp/100, temp%100);
2659 #ifdef CS333_P2
2660   cprintf("  %d.%d",p->cpu_ticks_total/100, p->cpu_ticks_total%100);
2661   cprintf("    %d  ", p->uid);
2662   cprintf("  %d  ", p->gid);
2663   if(p->parent && p->pid != 1)
2664             cprintf("  %d  ", p->parent->pid);
2665    else
2666             cprintf("  %d  ", p->pid);
2667 #endif
2668 }
2669
2670 void
2671 procdump(void)
2672 {
2673   static char *states[] = {
2674   [UNUSED]    "unused",
2675   [EMBRYO]    "embryo",
2676   [SLEEPING]  "sleep ",
2677   [RUNNABLE]  "runble",
2678   [RUNNING]   "run   ",
2679   [ZOMBIE]    "zombie"
2680   };
2681   int i;
2682   struct proc *p;
2683   char *state;
2684   uint pc[10];
2685
2686 #ifdef CS333_P2
2687   cprintf("\nPID  State  Name  Elapsed    TotalCpuTime    UID    GID    PPID
2688 #else
2689    cprintf("\nPID  State  Name  Elapsed    PCs\n");
2690 #endif
2691
2692   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2693     if(p->state == UNUSED)
2694       continue;
2695     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
2696       state = states[p->state];
2697     else
2698       state = "???";
2699     cprintf("%d    %s %s   ", p->pid, state, p->name);
```

```
2700     print_elapsed(p);
2701     if(p->state == SLEEPING){
2702       getcallerpcs((uint*)p->context->ebp+2, pc);
2703       for(i=0; i<10 && pc[i] != 0; i++)
2704         cprintf(" %p", pc[i]);
2705     }
2706     cprintf("\n");
2707   }
2708 }
2709
2710 #ifdef CS333_P2
2711 // Get process information
2712 int
2713 getprocs(uint max, struct uproc* table)
2714 {
2715    if(!table || max == 0) return -1;
2716   static char *states[] = {
2717   [UNUSED]    "unused",
2718   [EMBRYO]    "embryo",
2719   [SLEEPING]  "sleep ",
2720   [RUNNABLE]  "runble",
2721   [RUNNING]   "run   ",
2722   [ZOMBIE]    "zombie"
2723   };
2724
2725    int procscount = 0;
2726   struct proc *p;
2727    if(max > NPROC)
2728            max = NPROC;
2729    acquire(&ptable.lock);
2730    for(p = ptable.proc; p < &ptable.proc[max]; p++){
2731            if(p->state == UNUSED || p->state == EMBRYO || p->state == ZOMBIE
2732                    continue;
2733            table->pid = p->pid;
2734            table->uid = p->uid;
2735            table->gid = p->gid;
2736            if(!p->parent || p->pid ==1)
2737                    table->ppid = p->pid;
2738            else
2739                    table->ppid = p->parent->pid;
2740            acquire(&tickslock);
2741            table->elapsed_ticks = ticks - p->start_ticks;
2742            table->CPU_total_ticks = p->cpu_ticks_total;
2743            release(&tickslock);
2744            safestrcpy(table->state, states[p->state], sizeof(table->state));
2745            table->size = p->sz;
2746            safestrcpy(table->name, p->name, sizeof(table->name));
2747            ++procscount;
2748            ++table;
2749    }
```



```
2750    release(&ptable.lock);
2751
2752   return procscount;
2753 }
2754 #endif
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
```

```
2800 # Context switch
2801 #
2802 #   void swtch(struct context **old, struct context *new);
2803 #
2804 # Save current register context in old
2805 # and then load register context from new.
2806
2807 .globl swtch
2808 swtch:
2809   movl 4(%esp), %eax
2810   movl 8(%esp), %edx
2811
2812   # Save old callee-save registers
2813   pushl %ebp
2814   pushl %ebx
2815   pushl %esi
2816   pushl %edi
2817
2818   # Switch stacks
2819   movl %esp, (%eax)
2820   movl %edx, %esp
2821
2822   # Load new callee-save registers
2823   popl %edi
2824   popl %esi
2825   popl %ebx
2826   popl %ebp
2827   ret
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
```



```
2850 // Physical memory allocator, intended to allocate
2851 // memory for user processes, kernel stacks, page table pages,
2852 // and pipe buffers. Allocates 4096-byte pages.
2853
2854 #include "types.h"
2855 #include "defs.h"
2856 #include "param.h"
2857 #include "memlayout.h"
2858 #include "mmu.h"
2859 #include "spinlock.h"
2860
2861 void freerange(void *vstart, void *vend);
2862 extern char end[]; // first address after kernel loaded from ELF file
2863
2864 struct run {
2865   struct run *next;
2866 };
2867
2868 struct {
2869   struct spinlock lock;
2870   int use_lock;
2871   struct run *freelist;
2872 } kmem;
2873
2874 // Initialization happens in two phases.
2875 // 1. main() calls kinit1() while still using entrypgdir to place just
2876 // the pages mapped by entrypgdir on free list.
2877 // 2. main() calls kinit2() with the rest of the physical pages
2878 // after installing a full page table that maps them on all cores.
2879 void
2880 kinit1(void *vstart, void *vend)
2881 {
2882   initlock(&kmem.lock, "kmem");
2883   kmem.use_lock = 0;
2884   freerange(vstart, vend);
2885 }
2886
2887 void
2888 kinit2(void *vstart, void *vend)
2889 {
2890   freerange(vstart, vend);
2891   kmem.use_lock = 1;
2892 }
2893
2894
2895
2896
2897
2898
2899
```

```
2900 void
2901 freerange(void *vstart, void *vend)
2902 {
2903   char *p;
2904   p = (char*)PGROUNDUP((uint)vstart);
2905   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
2906     kfree(p);
2907 }
2908
2909 // Free the page of physical memory pointed at by v,
2910 // which normally should have been returned by a
2911 // call to kalloc().  (The exception is when
2912 // initializing the allocator; see kinit above.)
2913 void
2914 kfree(char *v)
2915 {
2916   struct run *r;
2917
2918   if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
2919     panic("kfree");
2920
2921   // Fill with junk to catch dangling refs.
2922   memset(v, 1, PGSIZE);
2923
2924   if(kmem.use_lock)
2925     acquire(&kmem.lock);
2926   r = (struct run*)v;
2927   r->next = kmem.freelist;
2928   kmem.freelist = r;
2929   if(kmem.use_lock)
2930     release(&kmem.lock);
2931 }
2932
2933 // Allocate one 4096-byte page of physical memory.
2934 // Returns a pointer that the kernel can use.
2935 // Returns 0 if the memory cannot be allocated.
2936 char*
2937 kalloc(void)
2938 {
2939   struct run *r;
2940
2941   if(kmem.use_lock)
2942     acquire(&kmem.lock);
2943   r = kmem.freelist;
2944   if(r)
2945     kmem.freelist = r->next;
2946   if(kmem.use_lock)
2947     release(&kmem.lock);
2948   return (char*)r;
2949 }
```



```
2950 // x86 trap and interrupt constants.
2951
2952 // Processor-defined:
2953 #define T_DIVIDE         0      // divide error
2954 #define T_DEBUG          1      // debug exception
2955 #define T_NMI            2      // non-maskable interrupt
2956 #define T_BRKPT          3      // breakpoint
2957 #define T_OFLOW          4      // overflow
2958 #define T_BOUND          5      // bounds check
2959 #define T_ILLOP          6      // illegal opcode
2960 #define T_DEVICE         7      // device not available
2961 #define T_DBLFLT         8      // double fault
2962 // #define T_COPROC      9      // reserved (not used since 486)
2963 #define T_TSS           10      // invalid task switch segment
2964 #define T_SEGNP         11      // segment not present
2965 #define T_STACK         12      // stack exception
2966 #define T_GPFLT         13      // general protection fault
2967 #define T_PGFLT         14      // page fault
2968 // #define T_RES        15      // reserved
2969 #define T_FPERR         16      // floating point error
2970 #define T_ALIGN         17      // aligment check
2971 #define T_MCHK          18      // machine check
2972 #define T_SIMDERR       19      // SIMD floating point error
2973
2974 // These are arbitrarily chosen, but with care not to overlap
2975 // processor defined exceptions or interrupt vectors.
2976 #define T_SYSCALL       64      // system call
2977 #define T_DEFAULT      500      // catchall
2978
2979 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
2980
2981 #define IRQ_TIMER        0
2982 #define IRQ_KBD          1
2983 #define IRQ_COM1         4
2984 #define IRQ_IDE         14
2985 #define IRQ_ERROR       19
2986 #define IRQ_SPURIOUS    31
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
```

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

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

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

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

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



```
#include "mmu.h"

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

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

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

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

```
3100 #include "types.h"
3101 #include "defs.h"
3102 #include "param.h"
3103 #include "memlayout.h"
3104 #include "mmu.h"
3105 #include "proc.h"
3106 #include "x86.h"
3107 #include "traps.h"
3108 #include "spinlock.h"
3109
3110 // Interrupt descriptor table (shared by all CPUs).
3111 struct gatedesc idt[256];
3112 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
3113 struct spinlock tickslock;
3114 uint ticks;
3115
3116 void
3117 tvinit(void)
3118 {
3119   int i;
3120
3121   for(i = 0; i < 256; i++)
3122     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3123   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3124
3125   initlock(&tickslock, "time");
3126 }
3127
3128 void
3129 idtinit(void)
3130 {
3131   lidt(idt, sizeof(idt));
3132 }
3133
3134 void
3135 trap(struct trapframe *tf)
3136 {
3137   if(tf->trapno == T_SYSCALL){
3138     if(proc->killed)
3139       exit();
3140     proc->tf = tf;
3141     syscall();
3142     if(proc->killed)
3143       exit();
3144     return;
3145   }
3146
3147   switch(tf->trapno){
3148   case T_IRQ0 + IRQ_TIMER:
3149     if(cpu->id == 0){
```



```
3150       acquire(&tickslock);
3151       ticks++;
3152       release(&tickslock);    // NOTE: MarkM has reversed these two lines.
3153       wakeup(&ticks);         // wakeup() should not require the tickslock t
3154     }
3155     lapiceoi();
3156     break;
3157   case T_IRQ0 + IRQ_IDE:
3158     ideintr();
3159     lapiceoi();
3160     break;
3161   case T_IRQ0 + IRQ_IDE+1:
3162     // Bochs generates spurious IDE1 interrupts.
3163     break;
3164   case T_IRQ0 + IRQ_KBD:
3165     kbdintr();
3166     lapiceoi();
3167     break;
3168   case T_IRQ0 + IRQ_COM1:
3169     uartintr();
3170     lapiceoi();
3171     break;
3172   case T_IRQ0 + 7:
3173   case T_IRQ0 + IRQ_SPURIOUS:
3174     cprintf("cpu%d: spurious interrupt at %x:%x\n",
3175             cpu->id, tf->cs, tf->eip);
3176     lapiceoi();
3177     break;
3178
3179   default:
3180     if(proc == 0 || (tf->cs&3) == 0){
3181       // In kernel, it must be our mistake.
3182       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3183               tf->trapno, cpu->id, tf->eip, rcr2());
3184       panic("trap");
3185     }
3186     // In user space, assume process misbehaved.
3187     cprintf("pid %d %s: trap %d err %d on cpu %d "
3188             "eip 0x%x addr 0x%x--kill proc\n",
3189             proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
3190             rcr2());
3191     proc->killed = 1;
3192   }
3193
3194   // Force process exit if it has been killed and is in user space.
3195   // (If it is still executing in the kernel, let it keep running
3196   // until it gets to the regular system call return.)
3197   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3198     exit();
3199
```

```
3200   // Force process to give up CPU on clock tick.
3201   // If interrupts were on while locks held, would need to check nlock.
3202   if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3203     yield();
3204
3205   // Check if the process has been killed since we yielded
3206   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3207     exit();
3208 }
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
```



```
3250 // System call numbers
3251 #define SYS_fork                1
3252 #define SYS_exit           SYS_fork+1
3253 #define SYS_wait           SYS_exit+1
3254 #define SYS_pipe           SYS_wait+1
3255 #define SYS_read           SYS_pipe+1
3256 #define SYS_kill           SYS_read+1
3257 #define SYS_exec           SYS_kill+1
3258 #define SYS_fstat          SYS_exec+1
3259 #define SYS_chdir          SYS_fstat+1
3260 #define SYS_dup            SYS_chdir+1
3261 #define SYS_getpid         SYS_dup+1
3262 #define SYS_sbrk           SYS_getpid+1
3263 #define SYS_sleep          SYS_sbrk+1
3264 #define SYS_uptime         SYS_sleep+1
3265 #define SYS_open           SYS_uptime+1
3266 #define SYS_write          SYS_open+1
3267 #define SYS_mknod          SYS_write+1
3268 #define SYS_unlink         SYS_mknod+1
3269 #define SYS_link           SYS_unlink+1
3270 #define SYS_mkdir          SYS_link+1
3271 #define SYS_close          SYS_mkdir+1
3272 #define SYS_halt           SYS_close+1
3273 // student system calls begin here. Follow the existing pattern.
3274 #define SYS_date                SYS_halt+1
3275 #define SYS_getuid         SYS_date+1
3276 #define SYS_getgid         SYS_getuid+1
3277 #define SYS_getppid                SYS_getgid+1
3278 #define SYS_setuid         SYS_getppid+1
3279 #define SYS_setgid         SYS_setuid+1
3280 #define SYS_getprocs  SYS_setgid+1
3281
3282
3283
3284
3285
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
```

```
3300 #include "types.h"
3301 #include "defs.h"
3302 #include "param.h"
3303 #include "memlayout.h"
3304 #include "mmu.h"
3305 #include "proc.h"
3306 #include "x86.h"
3307 #include "syscall.h"
3308
3309 // User code makes a system call with INT T_SYSCALL.
3310 // System call number in %eax.
3311 // Arguments on the stack, from the user call to the C
3312 // library system call function. The saved user %esp points
3313 // to a saved program counter, and then the first argument.
3314
3315 // Fetch the int at addr from the current process.
3316 int
3317 fetchint(uint addr, int *ip)
3318 {
3319   if(addr >= proc->sz || addr+4 > proc->sz)
3320     return -1;
3321   *ip = *(int*)(addr);
3322   return 0;
3323 }
3324
3325 // Fetch the nul-terminated string at addr from the current process.
3326 // Doesn't actually copy the string - just sets *pp to point at it.
3327 // Returns length of string, not including nul.
3328 int
3329 fetchstr(uint addr, char **pp)
3330 {
3331   char *s, *ep;
3332
3333   if(addr >= proc->sz)
3334     return -1;
3335   *pp = (char*)addr;
3336   ep = (char*)proc->sz;
3337   for(s = *pp; s < ep; s++)
3338     if(*s == 0)
3339       return s - *pp;
3340   return -1;
3341 }
3342
3343 // Fetch the nth 32-bit system call argument.
3344 int
3345 argint(int n, int *ip)
3346 {
3347   return fetchint(proc->tf->esp + 4 + 4*n, ip);
3348 }
3349
```



```
3350 // Fetch the nth word-sized system call argument as a pointer
3351 // to a block of memory of size n bytes.  Check that the pointer
3352 // lies within the process address space.
3353 int
3354 argptr(int n, char **pp, int size)
3355 {
3356   int i;
3357
3358   if(argint(n, &i) < 0)
3359     return -1;
3360   if((uint)i >= proc->sz || (uint)i+size > proc->sz)
3361     return -1;
3362   *pp = (char*)i;
3363   return 0;
3364 }
3365
3366 // Fetch the nth word-sized system call argument as a string pointer.
3367 // Check that the pointer is valid and the string is nul-terminated.
3368 // (There is no shared writable memory, so the string can't change
3369 // between this check and being used by the kernel.)
3370 int
3371 argstr(int n, char **pp)
3372 {
3373   int addr;
3374   if(argint(n, &addr) < 0)
3375     return -1;
3376   return fetchstr(addr, pp);
3377 }
3378
3379 extern int sys_chdir(void);
3380 extern int sys_close(void);
3381 extern int sys_dup(void);
3382 extern int sys_exec(void);
3383 extern int sys_exit(void);
3384 extern int sys_fork(void);
3385 extern int sys_fstat(void);
3386 extern int sys_getpid(void);
3387 extern int sys_kill(void);
3388 extern int sys_link(void);
3389 extern int sys_mkdir(void);
3390 extern int sys_mknod(void);
3391 extern int sys_open(void);
3392 extern int sys_pipe(void);
3393 extern int sys_read(void);
3394 extern int sys_sbrk(void);
3395 extern int sys_sleep(void);
3396 extern int sys_unlink(void);
3397 extern int sys_wait(void);
3398 extern int sys_write(void);
3399 extern int sys_uptime(void);
```

```
3400 extern int sys_halt(void);
3401 extern int sys_date(void);
3402 #ifdef CS333_P2
3403 extern int sys_getuid(void);
3404 extern int sys_getgid(void);
3405 extern int sys_getppid(void);
3406 extern int sys_setuid(void);
3407 extern int sys_setgid(void);
3408 extern int sys_getprocs(void);
3409 #endif
3410
3411 static int (*syscalls[])(void) = {
3412 [SYS_fork]    sys_fork,
3413 [SYS_exit]    sys_exit,
3414 [SYS_wait]    sys_wait,
3415 [SYS_pipe]    sys_pipe,
3416 [SYS_read]    sys_read,
3417 [SYS_kill]    sys_kill,
3418 [SYS_exec]    sys_exec,
3419 [SYS_fstat]   sys_fstat,
3420 [SYS_chdir]   sys_chdir,
3421 [SYS_dup]     sys_dup,
3422 [SYS_getpid]  sys_getpid,
3423 [SYS_sbrk]    sys_sbrk,
3424 [SYS_sleep]   sys_sleep,
3425 [SYS_uptime]  sys_uptime,
3426 [SYS_open]    sys_open,
3427 [SYS_write]   sys_write,
3428 [SYS_mknod]   sys_mknod,
3429 [SYS_unlink]  sys_unlink,
3430 [SYS_link]    sys_link,
3431 [SYS_mkdir]   sys_mkdir,
3432 [SYS_close]   sys_close,
3433 [SYS_halt]    sys_halt,
3434 [SYS_date]    sys_date,
3435 #ifdef CS333_P2
3436 [SYS_getuid]  sys_getuid,
3437 [SYS_getgid]  sys_getgid,
3438 [SYS_getppid] sys_getppid,
3439 [SYS_setuid]  sys_setuid,
3440 [SYS_setgid]  sys_setgid,
3441 [SYS_getprocs]  sys_getprocs,
3442 #endif
3443 };
3444
3445
3446
3447
3448
3449
```



```
3450 // put data structure for printing out system call invocation information he:
3451 #ifdef PRINT_SYSCALLS
3452 static const char * (print_syscalls[]) = {
3453 [SYS_fork] = "fork",
3454 [SYS_exit]   = "exit",
3455 [SYS_wait]     = "wait",
3456 [SYS_pipe]     = "pipe",
3457 [SYS_read]     = "read",
3458 [SYS_kill]     = "kill",
3459 [SYS_exec]     = "exec",
3460 [SYS_fstat]    = "fstat",
3461 [SYS_chdir]    = "chdir",
3462 [SYS_dup]      = "dup",
3463 [SYS_getpid]   = "getpid",
3464 [SYS_sbrk]     = "sbrk",
3465 [SYS_sleep]    = "sleep",
3466 [SYS_uptime]   = "uptime",
3467 [SYS_open]     = "open",
3468 [SYS_write]    = "write",
3469 [SYS_mknod]    = "mknod",
3470 [SYS_unlink]   = "unlink",
3471 [SYS_link]     = "link",
3472 [SYS_mkdir]    = "mkdir",
3473 [SYS_close]    = "close",
3474 [SYS_halt]     = "halt",
3475 [SYS_date]     = "date",
3476 #ifdef CS333_P2
3477 [SYS_getgid]   = "getuid",
3478 [SYS_getuid]   = "getgid",
3479 [SYS_getppid]  = "getppid",
3480 [SYS_setgid]   = "setuid",
3481 [SYS_setuid]   = "setgid",
3482 [SYS_getprocs]   = "getprocs",
3483 #endif
3484 };
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
```

```
#endif

void
syscall(void)
{
  int num;

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
// some code goes here
#ifdef PRINT_SYSCALLS
   cprintf("%s -> %d\n", print_syscalls[num], proc->tf->eax);
#endif
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```



```
#include "types.h"
#include "x86.h"
#include "defs.h"
#include "date.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "uproc.h"

int
sys_fork(void)
{
  return fork();
}

int
sys_exit(void)
{
  exit();
  return 0;  // not reached
}

int
sys_wait(void)
{
  return wait();
}

int
sys_kill(void)
{
  int pid;

  if(argint(0, &pid) < 0)
    return -1;
  return kill(pid);
}

int
sys_getpid(void)
{
  return proc->pid;
}
```

```
3600 int
3601 sys_sbrk(void)
3602 {
3603   int addr;
3604   int n;
3605
3606   if(argint(0, &n) < 0)
3607     return -1;
3608   addr = proc->sz;
3609   if(growproc(n) < 0)
3610     return -1;
3611   return addr;
3612 }
3613
3614 int
3615 sys_sleep(void)
3616 {
3617   int n;
3618   uint ticks0;
3619
3620   if(argint(0, &n) < 0)
3621     return -1;
3622   acquire(&tickslock);
3623   ticks0 = ticks;
3624   while(ticks - ticks0 < n){
3625     if(proc->killed){
3626       release(&tickslock);
3627       return -1;
3628     }
3629     sleep(&ticks, &tickslock);
3630   }
3631   release(&tickslock);
3632   return 0;
3633 }
3634
3635 // return how many clock tick interrupts have occurred
3636 // since start.
3637 int
3638 sys_uptime(void)
3639 {
3640   uint xticks;
3641
3642   acquire(&tickslock);
3643   xticks = ticks;
3644   release(&tickslock);
3645   return xticks;
3646 }
3647
3648
3649
```



```
3650 //Turn of the computer
3651 int sys_halt(void){
3652   cprintf("Shutting down ...\n");
3653   //outw (0xB004, 0x0 | 0x2000);
3654    outw( 0x604, 0x0 | 0x2000 );
3655    return 0;
3656
3657 }
3658
3659 //Get current UTC date of the system
3660 int
3661 sys_date(void)
3662 {
3663   struct rtcdate *d;
3664   if(argptr(0, (void*)&d, sizeof(*d)) < 0)
3665     return -1;
3666   cmostime(d);
3667   return 0;
3668 }
3669
3670 #ifdef CS333_P2
3671 // Set UID
3672 int
3673 sys_setuid(void)
3674 {
3675    uint new_uid;
3676   if(argint(0,(int*) &new_uid) < 0)
3677             return -1;
3678    if(new_uid < 0 || new_uid > 32767)
3679             return -1;
3680    proc->uid = new_uid;
3681    return 0;
3682 }
3683
3684 // Set GID
3685 int
3686 sys_setgid(void)
3687 {
3688    uint new_gid;
3689   if(argint(0,(int*) &new_gid) < 0)
3690             return -1;
3691    if(new_gid < 0 || new_gid > 32767)
3692             return -1;
3693    proc->gid = new_gid;
3694    return 0;
3695 }
3696
3697
3698
3699
```

```
3700 // Get UID of current process
3701 uint
3702 sys_getuid(void)
3703 {
3704    return proc->uid;
3705 }
3706
3707 // Get GID of current process
3708 uint
3709 sys_getgid(void)
3710 {
3711    return proc->gid;
3712 }
3713
3714 // Get PPID of current process
3715 uint
3716 sys_getppid(void)
3717 {
3718    if(proc->pid == 1)
3719            return proc->pid;
3720    if(!proc->parent)
3721            return proc->pid;
3722    return proc->parent->pid;
3723 }
3724
3725 // Get process info
3726 int
3727 sys_getprocs(void)
3728 {
3729    uint arg1;
3730    struct uproc* table;
3731    if(argint(0,(int*) &arg1) < 0)
3732            return -1;
3733    if(argptr(1,(void*)&table, sizeof(*table)) < 0)
3734            return -1;;
3735    return getprocs(arg1, table);
3736 }
3737 #endif
3738
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3749
```



```
3750 // halt the system.
3751 #include "types.h"
3752 #include "user.h"
3753
3754 int
3755 main(void) {
3756   halt();
3757   return 0;
3758 }
3759
3760
3761
3762
3763
3764
3765
3766
3767
3768
3769
3770
3771
3772
3773
3774
3775
3776
3777
3778
3779
3780
3781
3782
3783
3784
3785
3786
3787
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3799
```

```
3800 struct buf {
3801   int flags;
3802   uint dev;
3803   uint blockno;
3804   struct buf *prev; // LRU cache list
3805   struct buf *next;
3806   struct buf *qnext; // disk queue
3807   uchar data[BSIZE];
3808 };
3809 #define B_BUSY  0x1  // buffer is locked by some process
3810 #define B_VALID 0x2  // buffer has been read from disk
3811 #define B_DIRTY 0x4  // buffer needs to be written to disk
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 #define O_RDONLY  0x000
3851 #define O_WRONLY  0x001
3852 #define O_RDWR    0x002
3853 #define O_CREATE  0x200
3854
3855
3856
3857
3858
3859
3860
3861
3862
3863
3864
3865
3866
3867
3868
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
```

```
3900 #define T_DIR  1   // Directory
3901 #define T_FILE 2   // File
3902 #define T_DEV  3   // Device
3903
3904 struct stat {
3905   short type;  // Type of file
3906   int dev;     // File system's disk device
3907   uint ino;    // Inode number
3908   short nlink; // Number of links to file
3909   uint size;   // Size of file in bytes
3910 };
3911
3912
3913
3914
3915
3916
3917
3918
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
```



```
3950 // On-disk file system format.
3951 // Both the kernel and user programs use this header file.
3952
3953
3954 #define ROOTINO 1  // root i-number
3955 #define BSIZE 512  // block size
3956
3957 // Disk layout:
3958 // [ boot block | super block | log | inode blocks | free bit map | data blo
3959 //
3960 // mkfs computes the super block and builds an initial file system. The supe
3961 // the disk layout:
3962 struct superblock {
3963   uint size;         // Size of file system image (blocks)
3964   uint nblocks;      // Number of data blocks
3965   uint ninodes;      // Number of inodes.
3966   uint nlog;         // Number of log blocks
3967   uint logstart;     // Block number of first log block
3968   uint inodestart;   // Block number of first inode block
3969   uint bmapstart;    // Block number of first free map block
3970 };
3971
3972 #define NDIRECT 12
3973 #define NINDIRECT (BSIZE / sizeof(uint))
3974 #define MAXFILE (NDIRECT + NINDIRECT)
3975
3976 // On-disk inode structure
3977 struct dinode {
3978   short type;           // File type
3979   short major;          // Major device number (T_DEV only)
3980   short minor;          // Minor device number (T_DEV only)
3981   short nlink;          // Number of links to inode in file system
3982   uint size;            // Size of file (bytes)
3983   uint addrs[NDIRECT+1];   // Data block addresses
3984 };
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
```

```
4000 // Inodes per block.
4001 #define IPB           (BSIZE / sizeof(struct dinode))
4002
4003 // Block containing inode i
4004 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
4005
4006 // Bitmap bits per block
4007 #define BPB           (BSIZE*8)
4008
4009 // Block of free map containing bit for block b
4010 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
4011
4012 // Directory is a file containing a sequence of dirent structures.
4013 #define DIRSIZ 14
4014
4015 struct dirent {
4016   ushort inum;
4017   char name[DIRSIZ];
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 struct file {
4051   enum { FD_NONE, FD_PIPE, FD_INODE } type;
4052   int ref; // reference count
4053   char readable;
4054   char writable;
4055   struct pipe *pipe;
4056   struct inode *ip;
4057   uint off;
4058 };
4059
4060
4061 // in-memory copy of an inode
4062 struct inode {
4063   uint dev;           // Device number
4064   uint inum;          // Inode number
4065   int ref;            // Reference count
4066   int flags;          // I_BUSY, I_VALID
4067
4068   short type;         // copy of disk inode
4069   short major;
4070   short minor;
4071   short nlink;
4072   uint size;
4073   uint addrs[NDIRECT+1];
4074 };
4075 #define I_BUSY 0x1
4076 #define I_VALID 0x2
4077
4078 // table mapping major device number to
4079 // device functions
4080 struct devsw {
4081   int (*read)(struct inode*, char*, int);
4082   int (*write)(struct inode*, char*, int);
4083 };
4084
4085 extern struct devsw devsw[];
4086
4087 #define CONSOLE 1
4088
4089 // Blank page.
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
```

```
4100 // Simple PIO-based (non-DMA) IDE driver code.
4101
4102 #include "types.h"
4103 #include "defs.h"
4104 #include "param.h"
4105 #include "memlayout.h"
4106 #include "mmu.h"
4107 #include "proc.h"
4108 #include "x86.h"
4109 #include "traps.h"
4110 #include "spinlock.h"
4111 #include "fs.h"
4112 #include "buf.h"
4113
4114 #define SECTOR_SIZE   512
4115 #define IDE_BSY       0x80
4116 #define IDE_DRDY      0x40
4117 #define IDE_DF        0x20
4118 #define IDE_ERR       0x01
4119
4120 #define IDE_CMD_READ  0x20
4121 #define IDE_CMD_WRITE 0x30
4122
4123 // idequeue points to the buf now being read/written to the disk.
4124 // idequeue->qnext points to the next buf to be processed.
4125 // You must hold idelock while manipulating queue.
4126
4127 static struct spinlock idelock;
4128 static struct buf *idequeue;
4129
4130 static int havedisk1;
4131 static void idestart(struct buf*);
4132
4133 // Wait for IDE disk to become ready.
4134 static int
4135 idewait(int checkerr)
4136 {
4137   int r;
4138
4139   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4140     ;
4141   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4142     return -1;
4143   return 0;
4144 }
4145
4146
4147
4148
4149
```



```
4150 void
4151 ideinit(void)
4152 {
4153   int i;
4154
4155   initlock(&idelock, "ide");
4156   picenable(IRQ_IDE);
4157   ioapicenable(IRQ_IDE, ncpu - 1);
4158   idewait(0);
4159
4160   // Check if disk 1 is present
4161   outb(0x1f6, 0xe0 | (1<<4));
4162   for(i=0; i<1000; i++){
4163     if(inb(0x1f7) != 0){
4164       havedisk1 = 1;
4165       break;
4166     }
4167   }
4168
4169   // Switch back to disk 0.
4170   outb(0x1f6, 0xe0 | (0<<4));
4171 }
4172
4173 // Start the request for b.  Caller must hold idelock.
4174 static void
4175 idestart(struct buf *b)
4176 {
4177   if(b == 0)
4178     panic("idestart");
4179   if(b->blockno >= FSSIZE)
4180     panic("incorrect blockno");
4181   int sector_per_block =  BSIZE/SECTOR_SIZE;
4182   int sector = b->blockno * sector_per_block;
4183
4184   if (sector_per_block > 7) panic("idestart");
4185
4186   idewait(0);
4187   outb(0x3f6, 0);  // generate interrupt
4188   outb(0x1f2, sector_per_block);  // number of sectors
4189   outb(0x1f3, sector & 0xff);
4190   outb(0x1f4, (sector >> 8) & 0xff);
4191   outb(0x1f5, (sector >> 16) & 0xff);
4192   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4193   if(b->flags & B_DIRTY){
4194     outb(0x1f7, IDE_CMD_WRITE);
4195     outsl(0x1f0, b->data, BSIZE/4);
4196   } else {
4197     outb(0x1f7, IDE_CMD_READ);
4198   }
4199 }
```

```
4200 // Interrupt handler.
4201 void
4202 ideintr(void)
4203 {
4204   struct buf *b;
4205
4206   // First queued buffer is the active request.
4207   acquire(&idelock);
4208   if((b = idequeue) == 0){
4209     release(&idelock);
4210     // cprintf("spurious IDE interrupt\n");
4211     return;
4212   }
4213   idequeue = b->qnext;
4214
4215   // Read data if needed.
4216   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4217     insl(0x1f0, b->data, BSIZE/4);
4218
4219   // Wake process waiting for this buf.
4220   b->flags |= B_VALID;
4221   b->flags &= ~B_DIRTY;
4222   wakeup(b);
4223
4224   // Start disk on next buf in queue.
4225   if(idequeue != 0)
4226     idestart(idequeue);
4227
4228   release(&idelock);
4229 }
4230
4231 // Sync buf with disk.
4232 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
4233 // Else if B_VALID is not set, read buf from disk, set B_VALID.
4234 void
4235 iderw(struct buf *b)
4236 {
4237   struct buf **pp;
4238
4239   if(!(b->flags & B_BUSY))
4240     panic("iderw: buf not busy");
4241   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
4242     panic("iderw: nothing to do");
4243   if(b->dev != 0 && !havedisk1)
4244     panic("iderw: ide disk 1 not present");
4245
4246   acquire(&idelock);
4247
4248
4249
```



```
4250   // Append b to idequeue.
4251   b->qnext = 0;
4252   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
4253     ;
4254   *pp = b;
4255
4256   // Start disk if necessary.
4257   if(idequeue == b)
4258     idestart(b);
4259
4260   // Wait for request to finish.
4261   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
4262     sleep(b, &idelock);
4263   }
4264
4265   release(&idelock);
4266 }
4267
4268
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
```

```
4300 // Buffer cache.
4301 //
4302 // The buffer cache is a linked list of buf structures holding
4303 // cached copies of disk block contents.  Caching disk blocks
4304 // in memory reduces the number of disk reads and also provides
4305 // a synchronization point for disk blocks used by multiple processes.
4306 //
4307 // Interface:
4308 // * To get a buffer for a particular disk block, call bread.
4309 // * After changing buffer data, call bwrite to write it to disk.
4310 // * When done with the buffer, call brelse.
4311 // * Do not use the buffer after calling brelse.
4312 // * Only one process at a time can use a buffer,
4313 //     so do not keep them longer than necessary.
4314 //
4315 // The implementation uses three state flags internally:
4316 // * B_BUSY: the block has been returned from bread
4317 //     and has not been passed back to brelse.
4318 // * B_VALID: the buffer data has been read from the disk.
4319 // * B_DIRTY: the buffer data has been modified
4320 //     and needs to be written to disk.
4321
4322 #include "types.h"
4323 #include "defs.h"
4324 #include "param.h"
4325 #include "spinlock.h"
4326 #include "fs.h"
4327 #include "buf.h"
4328
4329 struct {
4330   struct spinlock lock;
4331   struct buf buf[NBUF];
4332
4333   // Linked list of all buffers, through prev/next.
4334   // head.next is most recently used.
4335   struct buf head;
4336 } bcache;
4337
4338 void
4339 binit(void)
4340 {
4341   struct buf *b;
4342
4343   initlock(&bcache.lock, "bcache");
4344
4345   // Create linked list of buffers
4346   bcache.head.prev = &bcache.head;
4347   bcache.head.next = &bcache.head;
4348   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4349     b->next = bcache.head.next;
```



```
4350     b->prev = &bcache.head;
4351     b->dev = -1;
4352     bcache.head.next->prev = b;
4353     bcache.head.next = b;
4354   }
4355 }
4356
4357 // Look through buffer cache for block on device dev.
4358 // If not found, allocate a buffer.
4359 // In either case, return B_BUSY buffer.
4360 static struct buf*
4361 bget(uint dev, uint blockno)
4362 {
4363   struct buf *b;
4364
4365   acquire(&bcache.lock);
4366
4367  loop:
4368   // Is the block already cached?
4369   for(b = bcache.head.next; b != &bcache.head; b = b->next){
4370     if(b->dev == dev && b->blockno == blockno){
4371       if(!(b->flags & B_BUSY)){
4372         b->flags |= B_BUSY;
4373         release(&bcache.lock);
4374         return b;
4375       }
4376       sleep(b, &bcache.lock);
4377       goto loop;
4378     }
4379   }
4380
4381   // Not cached; recycle some non-busy and clean buffer.
4382   // "clean" because B_DIRTY and !B_BUSY means log.c
4383   // hasn't yet committed the changes to the buffer.
4384   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4385     if((b->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
4386       b->dev = dev;
4387       b->blockno = blockno;
4388       b->flags = B_BUSY;
4389       release(&bcache.lock);
4390       return b;
4391     }
4392   }
4393   panic("bget: no buffers");
4394 }
4395
4396
4397
4398
4399
```

```
4400 // Return a B_BUSY buf with the contents of the indicated block.
4401 struct buf*
4402 bread(uint dev, uint blockno)
4403 {
4404   struct buf *b;
4405
4406   b = bget(dev, blockno);
4407   if(!(b->flags & B_VALID)) {
4408     iderw(b);
4409   }
4410   return b;
4411 }
4412
4413 // Write b's contents to disk.  Must be B_BUSY.
4414 void
4415 bwrite(struct buf *b)
4416 {
4417   if((b->flags & B_BUSY) == 0)
4418     panic("bwrite");
4419   b->flags |= B_DIRTY;
4420   iderw(b);
4421 }
4422
4423 // Release a B_BUSY buffer.
4424 // Move to the head of the MRU list.
4425 void
4426 brelse(struct buf *b)
4427 {
4428   if((b->flags & B_BUSY) == 0)
4429     panic("brelse");
4430
4431   acquire(&bcache.lock);
4432
4433   b->next->prev = b->prev;
4434   b->prev->next = b->next;
4435   b->next = bcache.head.next;
4436   b->prev = &bcache.head;
4437   bcache.head.next->prev = b;
4438   bcache.head.next = b;
4439
4440   b->flags &= ~B_BUSY;
4441   wakeup(b);
4442
4443   release(&bcache.lock);
4444 }
4445 // Blank page.
4446
4447
4448
4449
```



```
4450 #include "types.h"
4451 #include "defs.h"
4452 #include "param.h"
4453 #include "spinlock.h"
4454 #include "fs.h"
4455 #include "buf.h"
4456
4457 // Simple logging that allows concurrent FS system calls.
4458 //
4459 // A log transaction contains the updates of multiple FS system
4460 // calls. The logging system only commits when there are
4461 // no FS system calls active. Thus there is never
4462 // any reasoning required about whether a commit might
4463 // write an uncommitted system call's updates to disk.
4464 //
4465 // A system call should call begin_op()/end_op() to mark
4466 // its start and end. Usually begin_op() just increments
4467 // the count of in-progress FS system calls and returns.
4468 // But if it thinks the log is close to running out, it
4469 // sleeps until the last outstanding end_op() commits.
4470 //
4471 // The log is a physical re-do log containing disk blocks.
4472 // The on-disk log format:
4473 //   header block, containing block #s for block A, B, C, ...
4474 //   block A
4475 //   block B
4476 //   block C
4477 //   ...
4478 // Log appends are synchronous.
4479
4480 // Contents of the header block, used for both the on-disk header block
4481 // and to keep track in memory of logged block# before commit.
4482 struct logheader {
4483   int n;
4484   int block[LOGSIZE];
4485 };
4486
4487 struct log {
4488   struct spinlock lock;
4489   int start;
4490   int size;
4491   int outstanding; // how many FS sys calls are executing.
4492   int committing;  // in commit(), please wait.
4493   int dev;
4494   struct logheader lh;
4495 };
4496
4497
4498
4499
```

```
4500 struct log log;
4501
4502 static void recover_from_log(void);
4503 static void commit();
4504
4505 void
4506 initlog(int dev)
4507 {
4508   if (sizeof(struct logheader) >= BSIZE)
4509     panic("initlog: too big logheader");
4510
4511   struct superblock sb;
4512   initlock(&log.lock, "log");
4513   readsb(dev, &sb);
4514   log.start = sb.logstart;
4515   log.size = sb.nlog;
4516   log.dev = dev;
4517   recover_from_log();
4518 }
4519
4520 // Copy committed blocks from log to their home location
4521 static void
4522 install_trans(void)
4523 {
4524   int tail;
4525
4526   for (tail = 0; tail < log.lh.n; tail++) {
4527     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
4528     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
4529     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
4530     bwrite(dbuf);  // write dst to disk
4531     brelse(lbuf);
4532     brelse(dbuf);
4533   }
4534 }
4535
4536 // Read the log header from disk into the in-memory log header
4537 static void
4538 read_head(void)
4539 {
4540   struct buf *buf = bread(log.dev, log.start);
4541   struct logheader *lh = (struct logheader *) (buf->data);
4542   int i;
4543   log.lh.n = lh->n;
4544   for (i = 0; i < log.lh.n; i++) {
4545     log.lh.block[i] = lh->block[i];
4546   }
4547   brelse(buf);
4548 }
4549
```



```
4550 // Write in-memory log header to disk.
4551 // This is the true point at which the
4552 // current transaction commits.
4553 static void
4554 write_head(void)
4555 {
4556   struct buf *buf = bread(log.dev, log.start);
4557   struct logheader *hb = (struct logheader *) (buf->data);
4558   int i;
4559   hb->n = log.lh.n;
4560   for (i = 0; i < log.lh.n; i++) {
4561     hb->block[i] = log.lh.block[i];
4562   }
4563   bwrite(buf);
4564   brelse(buf);
4565 }
4566
4567 static void
4568 recover_from_log(void)
4569 {
4570   read_head();
4571   install_trans(); // if committed, copy from log to disk
4572   log.lh.n = 0;
4573   write_head(); // clear the log
4574 }
4575
4576 // called at the start of each FS system call.
4577 void
4578 begin_op(void)
4579 {
4580   acquire(&log.lock);
4581   while(1){
4582     if(log.committing){
4583       sleep(&log, &log.lock);
4584     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
4585       // this op might exhaust log space; wait for commit.
4586       sleep(&log, &log.lock);
4587     } else {
4588       log.outstanding += 1;
4589       release(&log.lock);
4590       break;
4591     }
4592   }
4593 }
4594
4595
4596
4597
4598
4599
```

```
4600 // called at the end of each FS system call.
4601 // commits if this was the last outstanding operation.
4602 void
4603 end_op(void)
4604 {
4605   int do_commit = 0;
4606
4607   acquire(&log.lock);
4608   log.outstanding -= 1;
4609   if(log.committing)
4610     panic("log.committing");
4611   if(log.outstanding == 0){
4612     do_commit = 1;
4613     log.committing = 1;
4614   } else {
4615     // begin_op() may be waiting for log space.
4616     wakeup(&log);
4617   }
4618   release(&log.lock);
4619
4620   if(do_commit){
4621     // call commit w/o holding locks, since not allowed
4622     // to sleep with locks.
4623     commit();
4624     acquire(&log.lock);
4625     log.committing = 0;
4626     wakeup(&log);
4627     release(&log.lock);
4628   }
4629 }
4630
4631 // Copy modified blocks from cache to log.
4632 static void
4633 write_log(void)
4634 {
4635   int tail;
4636
4637   for (tail = 0; tail < log.lh.n; tail++) {
4638     struct buf *to = bread(log.dev, log.start+tail+1); // log block
4639     struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
4640     memmove(to->data, from->data, BSIZE);
4641     bwrite(to);  // write the log
4642     brelse(from);
4643     brelse(to);
4644   }
4645 }
4646
4647
4648
4649
```



```
4650 static void
4651 commit()
4652 {
4653   if (log.lh.n > 0) {
4654     write_log();     // Write modified blocks from cache to log
4655     write_head();    // Write header to disk -- the real commit
4656     install_trans(); // Now install writes to home locations
4657     log.lh.n = 0;
4658     write_head();    // Erase the transaction from the log
4659   }
4660 }
4661
4662 // Caller has modified b->data and is done with the buffer.
4663 // Record the block number and pin in the cache with B_DIRTY.
4664 // commit()/write_log() will do the disk write.
4665 //
4666 // log_write() replaces bwrite(); a typical use is:
4667 //   bp = bread(...)
4668 //   modify bp->data[]
4669 //   log_write(bp)
4670 //   brelse(bp)
4671 void
4672 log_write(struct buf *b)
4673 {
4674   int i;
4675
4676   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
4677     panic("too big a transaction");
4678   if (log.outstanding < 1)
4679     panic("log_write outside of trans");
4680
4681   acquire(&log.lock);
4682   for (i = 0; i < log.lh.n; i++) {
4683     if (log.lh.block[i] == b->blockno)   // log absorbtion
4684       break;
4685   }
4686   log.lh.block[i] = b->blockno;
4687   if (i == log.lh.n)
4688     log.lh.n++;
4689   b->flags |= B_DIRTY; // prevent eviction
4690   release(&log.lock);
4691 }
4692
4693
4694
4695
4696
4697
4698
4699
```

```
4700 // File system implementation.  Five layers:
4701 //   + Blocks: allocator for raw disk blocks.
4702 //   + Log: crash recovery for multi-step updates.
4703 //   + Files: inode allocator, reading, writing, metadata.
4704 //   + Directories: inode with special contents (list of other inodes!)
4705 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
4706 //
4707 // This file contains the low-level file system manipulation
4708 // routines.  The (higher-level) system call implementations
4709 // are in sysfile.c.
4710
4711 #include "types.h"
4712 #include "defs.h"
4713 #include "param.h"
4714 #include "stat.h"
4715 #include "mmu.h"
4716 #include "proc.h"
4717 #include "spinlock.h"
4718 #include "fs.h"
4719 #include "buf.h"
4720 #include "file.h"
4721
4722 #define min(a, b) ((a) < (b) ? (a) : (b))
4723 static void itrunc(struct inode*);
4724 struct superblock sb;   // there should be one per dev, but we run with one 
4725
4726 // Read the super block.
4727 void
4728 readsb(int dev, struct superblock *sb)
4729 {
4730   struct buf *bp;
4731
4732   bp = bread(dev, 1);
4733   memmove(sb, bp->data, sizeof(*sb));
4734   brelse(bp);
4735 }
4736
4737 // Zero a block.
4738 static void
4739 bzero(int dev, int bno)
4740 {
4741   struct buf *bp;
4742
4743   bp = bread(dev, bno);
4744   memset(bp->data, 0, BSIZE);
4745   log_write(bp);
4746   brelse(bp);
4747 }
4748
4749
```



```
4750 // Blocks.
4751
4752 // Allocate a zeroed disk block.
4753 static uint
4754 balloc(uint dev)
4755 {
4756   int b, bi, m;
4757   struct buf *bp;
4758
4759   bp = 0;
4760   for(b = 0; b < sb.size; b += BPB){
4761     bp = bread(dev, BBLOCK(b, sb));
4762     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
4763       m = 1 << (bi % 8);
4764       if((bp->data[bi/8] & m) == 0){  // Is block free?
4765         bp->data[bi/8] |= m;  // Mark block in use.
4766         log_write(bp);
4767         brelse(bp);
4768         bzero(dev, b + bi);
4769         return b + bi;
4770       }
4771     }
4772     brelse(bp);
4773   }
4774   panic("balloc: out of blocks");
4775 }
4776
4777 // Free a disk block.
4778 static void
4779 bfree(int dev, uint b)
4780 {
4781   struct buf *bp;
4782   int bi, m;
4783
4784   readsb(dev, &sb);
4785   bp = bread(dev, BBLOCK(b, sb));
4786   bi = b % BPB;
4787   m = 1 << (bi % 8);
4788   if((bp->data[bi/8] & m) == 0)
4789     panic("freeing free block");
4790   bp->data[bi/8] &= ~m;
4791   log_write(bp);
4792   brelse(bp);
4793 }
4794
4795
4796
4797
4798
4799
```

```
4800 // Inodes.
4801 //
4802 // An inode describes a single unnamed file.
4803 // The inode disk structure holds metadata: the file's type,
4804 // its size, the number of links referring to it, and the
4805 // list of blocks holding the file's content.
4806 //
4807 // The inodes are laid out sequentially on disk at
4808 // sb.startinode. Each inode has a number, indicating its
4809 // position on the disk.
4810 //
4811 // The kernel keeps a cache of in-use inodes in memory
4812 // to provide a place for synchronizing access
4813 // to inodes used by multiple processes. The cached
4814 // inodes include book-keeping information that is
4815 // not stored on disk: ip->ref and ip->flags.
4816 //
4817 // An inode and its in-memory represtative go through a
4818 // sequence of states before they can be used by the
4819 // rest of the file system code.
4820 //
4821 // * Allocation: an inode is allocated if its type (on disk)
4822 //   is non-zero. ialloc() allocates, iput() frees if
4823 //   the link count has fallen to zero.
4824 //
4825 // * Referencing in cache: an entry in the inode cache
4826 //   is free if ip->ref is zero. Otherwise ip->ref tracks
4827 //   the number of in-memory pointers to the entry (open
4828 //   files and current directories). iget() to find or
4829 //   create a cache entry and increment its ref, iput()
4830 //   to decrement ref.
4831 //
4832 // * Valid: the information (type, size, &c) in an inode
4833 //   cache entry is only correct when the I_VALID bit
4834 //   is set in ip->flags. ilock() reads the inode from
4835 //   the disk and sets I_VALID, while iput() clears
4836 //   I_VALID if ip->ref has fallen to zero.
4837 //
4838 // * Locked: file system code may only examine and modify
4839 //   the information in an inode and its content if it
4840 //   has first locked the inode. The I_BUSY flag indicates
4841 //   that the inode is locked. ilock() sets I_BUSY,
4842 //   while iunlock clears it.
4843 //
4844 // Thus a typical sequence is:
4845 //   ip = iget(dev, inum)
4846 //   ilock(ip)
4847 //   ... examine and modify ip->xxx ...
4848 //   iunlock(ip)
4849 //   iput(ip)
```



```
4850 //
4851 // ilock() is separate from iget() so that system calls can
4852 // get a long-term reference to an inode (as for an open file)
4853 // and only lock it for short periods (e.g., in read()).
4854 // The separation also helps avoid deadlock and races during
4855 // pathname lookup. iget() increments ip->ref so that the inode
4856 // stays cached and pointers to it remain valid.
4857 //
4858 // Many internal file system functions expect the caller to
4859 // have locked the inodes involved; this lets callers create
4860 // multi-step atomic operations.
4861
4862 struct {
4863   struct spinlock lock;
4864   struct inode inode[NINODE];
4865 } icache;
4866
4867 void
4868 iinit(int dev)
4869 {
4870   initlock(&icache.lock, "icache");
4871   readsb(dev, &sb);
4872   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d inodestart %
4873           sb.nblocks, sb.ninodes, sb.nlog, sb.logstart, sb.inodestart, sb.bma
4874 }
4875
4876 static struct inode* iget(uint dev, uint inum);
4877
4878 // Allocate a new inode with the given type on device dev.
4879 // A free inode has a type of zero.
4880 struct inode*
4881 ialloc(uint dev, short type)
4882 {
4883   int inum;
4884   struct buf *bp;
4885   struct dinode *dip;
4886
4887   for(inum = 1; inum < sb.ninodes; inum++){
4888     bp = bread(dev, IBLOCK(inum, sb));
4889     dip = (struct dinode*)bp->data + inum%IPB;
4890     if(dip->type == 0){  // a free inode
4891       memset(dip, 0, sizeof(*dip));
4892       dip->type = type;
4893       log_write(bp);   // mark it allocated on the disk
4894       brelse(bp);
4895       return iget(dev, inum);
4896     }
4897     brelse(bp);
4898   }
4899   panic("ialloc: no inodes");
```

```
4900 }
4901
4902 // Copy a modified in-memory inode to disk.
4903 void
4904 iupdate(struct inode *ip)
4905 {
4906   struct buf *bp;
4907   struct dinode *dip;
4908
4909   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
4910   dip = (struct dinode*)bp->data + ip->inum%IPB;
4911   dip->type = ip->type;
4912   dip->major = ip->major;
4913   dip->minor = ip->minor;
4914   dip->nlink = ip->nlink;
4915   dip->size = ip->size;
4916   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
4917   log_write(bp);
4918   brelse(bp);
4919 }
4920
4921 // Find the inode with number inum on device dev
4922 // and return the in-memory copy. Does not lock
4923 // the inode and does not read it from disk.
4924 static struct inode*
4925 iget(uint dev, uint inum)
4926 {
4927   struct inode *ip, *empty;
4928
4929   acquire(&icache.lock);
4930
4931   // Is the inode already cached?
4932   empty = 0;
4933   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
4934     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
4935       ip->ref++;
4936       release(&icache.lock);
4937       return ip;
4938     }
4939     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
4940       empty = ip;
4941   }
4942
4943   // Recycle an inode cache entry.
4944   if(empty == 0)
4945     panic("iget: no inodes");
4946
4947
4948
4949
```



```
4950   ip = empty;
4951   ip->dev = dev;
4952   ip->inum = inum;
4953   ip->ref = 1;
4954   ip->flags = 0;
4955   release(&icache.lock);
4956
4957   return ip;
4958 }
4959
4960 // Increment reference count for ip.
4961 // Returns ip to enable ip = idup(ip1) idiom.
4962 struct inode*
4963 idup(struct inode *ip)
4964 {
4965   acquire(&icache.lock);
4966   ip->ref++;
4967   release(&icache.lock);
4968   return ip;
4969 }
4970
4971 // Lock the given inode.
4972 // Reads the inode from disk if necessary.
4973 void
4974 ilock(struct inode *ip)
4975 {
4976   struct buf *bp;
4977   struct dinode *dip;
4978
4979   if(ip == 0 || ip->ref < 1)
4980     panic("ilock");
4981
4982   acquire(&icache.lock);
4983   while(ip->flags & I_BUSY)
4984     sleep(ip, &icache.lock);
4985   ip->flags |= I_BUSY;
4986   release(&icache.lock);
4987
4988   if(!(ip->flags & I_VALID)){
4989     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
4990     dip = (struct dinode*)bp->data + ip->inum%IPB;
4991     ip->type = dip->type;
4992     ip->major = dip->major;
4993     ip->minor = dip->minor;
4994     ip->nlink = dip->nlink;
4995     ip->size = dip->size;
4996     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
4997     brelse(bp);
4998     ip->flags |= I_VALID;
4999     if(ip->type == 0)
```

```
5000       panic("ilock: no type");
5001   }
5002 }
5003
5004 // Unlock the given inode.
5005 void
5006 iunlock(struct inode *ip)
5007 {
5008   if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
5009     panic("iunlock");
5010
5011   acquire(&icache.lock);
5012   ip->flags &= ~I_BUSY;
5013   wakeup(ip);
5014   release(&icache.lock);
5015 }
5016
5017 // Drop a reference to an in-memory inode.
5018 // If that was the last reference, the inode cache entry can
5019 // be recycled.
5020 // If that was the last reference and the inode has no links
5021 // to it, free the inode (and its content) on disk.
5022 // All calls to iput() must be inside a transaction in
5023 // case it has to free the inode.
5024 void
5025 iput(struct inode *ip)
5026 {
5027   acquire(&icache.lock);
5028   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
5029     // inode has no links and no other references: truncate and free.
5030     if(ip->flags & I_BUSY)
5031       panic("iput busy");
5032     ip->flags |= I_BUSY;
5033     release(&icache.lock);
5034     itrunc(ip);
5035     ip->type = 0;
5036     iupdate(ip);
5037     acquire(&icache.lock);
5038     ip->flags = 0;
5039     wakeup(ip);
5040   }
5041   ip->ref--;
5042   release(&icache.lock);
5043 }
5044
5045
5046
5047
5048
5049
```



```
5050 // Common idiom: unlock, then put.
5051 void
5052 iunlockput(struct inode *ip)
5053 {
5054   iunlock(ip);
5055   iput(ip);
5056 }
5057
5058 // Inode content
5059 //
5060 // The content (data) associated with each inode is stored
5061 // in blocks on the disk. The first NDIRECT block numbers
5062 // are listed in ip->addrs[].  The next NINDIRECT blocks are
5063 // listed in block ip->addrs[NDIRECT].
5064
5065 // Return the disk block address of the nth block in inode ip.
5066 // If there is no such block, bmap allocates one.
5067 static uint
5068 bmap(struct inode *ip, uint bn)
5069 {
5070   uint addr, *a;
5071   struct buf *bp;
5072
5073   if(bn < NDIRECT){
5074     if((addr = ip->addrs[bn]) == 0)
5075       ip->addrs[bn] = addr = balloc(ip->dev);
5076     return addr;
5077   }
5078   bn -= NDIRECT;
5079
5080   if(bn < NINDIRECT){
5081     // Load indirect block, allocating if necessary.
5082     if((addr = ip->addrs[NDIRECT]) == 0)
5083       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5084     bp = bread(ip->dev, addr);
5085     a = (uint*)bp->data;
5086     if((addr = a[bn]) == 0){
5087       a[bn] = addr = balloc(ip->dev);
5088       log_write(bp);
5089     }
5090     brelse(bp);
5091     return addr;
5092   }
5093
5094   panic("bmap: out of range");
5095 }
5096
5097
5098
5099
```

```
5100 // Truncate inode (discard contents).
5101 // Only called when the inode has no links
5102 // to it (no directory entries referring to it)
5103 // and has no in-memory reference to it (is
5104 // not an open file or current directory).
5105 static void
5106 itrunc(struct inode *ip)
5107 {
5108   int i, j;
5109   struct buf *bp;
5110   uint *a;
5111
5112   for(i = 0; i < NDIRECT; i++){
5113     if(ip->addrs[i]){
5114       bfree(ip->dev, ip->addrs[i]);
5115       ip->addrs[i] = 0;
5116     }
5117   }
5118
5119   if(ip->addrs[NDIRECT]){
5120     bp = bread(ip->dev, ip->addrs[NDIRECT]);
5121     a = (uint*)bp->data;
5122     for(j = 0; j < NINDIRECT; j++){
5123       if(a[j])
5124         bfree(ip->dev, a[j]);
5125     }
5126     brelse(bp);
5127     bfree(ip->dev, ip->addrs[NDIRECT]);
5128     ip->addrs[NDIRECT] = 0;
5129   }
5130
5131   ip->size = 0;
5132   iupdate(ip);
5133 }
5134
5135 // Copy stat information from inode.
5136 void
5137 stati(struct inode *ip, struct stat *st)
5138 {
5139   st->dev = ip->dev;
5140   st->ino = ip->inum;
5141   st->type = ip->type;
5142   st->nlink = ip->nlink;
5143   st->size = ip->size;
5144 }
5145
5146
5147
5148
5149
```



```
5150 // Read data from inode.
5151 int
5152 readi(struct inode *ip, char *dst, uint off, uint n)
5153 {
5154   uint tot, m;
5155   struct buf *bp;
5156
5157   if(ip->type == T_DEV){
5158     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
5159       return -1;
5160     return devsw[ip->major].read(ip, dst, n);
5161   }
5162
5163   if(off > ip->size || off + n < off)
5164     return -1;
5165   if(off + n > ip->size)
5166     n = ip->size - off;
5167
5168   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5169     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5170     m = min(n - tot, BSIZE - off%BSIZE);
5171     memmove(dst, bp->data + off%BSIZE, m);
5172     brelse(bp);
5173   }
5174   return n;
5175 }
5176
5177 // Write data to inode.
5178 int
5179 writei(struct inode *ip, char *src, uint off, uint n)
5180 {
5181   uint tot, m;
5182   struct buf *bp;
5183
5184   if(ip->type == T_DEV){
5185     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
5186       return -1;
5187     return devsw[ip->major].write(ip, src, n);
5188   }
5189
5190   if(off > ip->size || off + n < off)
5191     return -1;
5192   if(off + n > MAXFILE*BSIZE)
5193     return -1;
5194
5195   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
5196     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5197     m = min(n - tot, BSIZE - off%BSIZE);
5198     memmove(bp->data + off%BSIZE, src, m);
5199     log_write(bp);
```

```
5200     brelse(bp);
5201   }
5202
5203   if(n > 0 && off > ip->size){
5204     ip->size = off;
5205     iupdate(ip);
5206   }
5207   return n;
5208 }
5209
5210 // Directories
5211
5212 int
5213 namecmp(const char *s, const char *t)
5214 {
5215   return strncmp(s, t, DIRSIZ);
5216 }
5217
5218 // Look for a directory entry in a directory.
5219 // If found, set *poff to byte offset of entry.
5220 struct inode*
5221 dirlookup(struct inode *dp, char *name, uint *poff)
5222 {
5223   uint off, inum;
5224   struct dirent de;
5225
5226   if(dp->type != T_DIR)
5227     panic("dirlookup not DIR");
5228
5229   for(off = 0; off < dp->size; off += sizeof(de)){
5230     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5231       panic("dirlink read");
5232     if(de.inum == 0)
5233       continue;
5234     if(namecmp(name, de.name) == 0){
5235       // entry matches path element
5236       if(poff)
5237         *poff = off;
5238       inum = de.inum;
5239       return iget(dp->dev, inum);
5240     }
5241   }
5242
5243   return 0;
5244 }
5245
5246
5247
5248
5249
```



```
5250 // Write a new directory entry (name, inum) into the directory dp.
5251 int
5252 dirlink(struct inode *dp, char *name, uint inum)
5253 {
5254   int off;
5255   struct dirent de;
5256   struct inode *ip;
5257
5258   // Check that name is not present.
5259   if((ip = dirlookup(dp, name, 0)) != 0){
5260     iput(ip);
5261     return -1;
5262   }
5263
5264   // Look for an empty dirent.
5265   for(off = 0; off < dp->size; off += sizeof(de)){
5266     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5267       panic("dirlink read");
5268     if(de.inum == 0)
5269       break;
5270   }
5271
5272   strncpy(de.name, name, DIRSIZ);
5273   de.inum = inum;
5274   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5275     panic("dirlink");
5276
5277   return 0;
5278 }
5279
5280 // Paths
5281
5282 // Copy the next path element from path into name.
5283 // Return a pointer to the element following the copied one.
5284 // The returned path has no leading slashes,
5285 // so the caller can check *path=='\0' to see if the name is the last one.
5286 // If no name to remove, return 0.
5287 //
5288 // Examples:
5289 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5290 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5291 //   skipelem("a", name) = "", setting name = "a"
5292 //   skipelem("", name) = skipelem("////", name) = 0
5293 //
5294 static char*
5295 skipelem(char *path, char *name)
5296 {
5297   char *s;
5298   int len;
5299
```

```
5300   while(*path == '/')
5301     path++;
5302   if(*path == 0)
5303     return 0;
5304   s = path;
5305   while(*path != '/' && *path != 0)
5306     path++;
5307   len = path - s;
5308   if(len >= DIRSIZ)
5309     memmove(name, s, DIRSIZ);
5310   else {
5311     memmove(name, s, len);
5312     name[len] = 0;
5313   }
5314   while(*path == '/')
5315     path++;
5316   return path;
5317 }
5318
5319 // Look up and return the inode for a path name.
5320 // If parent != 0, return the inode for the parent and copy the final
5321 // path element into name, which must have room for DIRSIZ bytes.
5322 // Must be called inside a transaction since it calls iput().
5323 static struct inode*
5324 namex(char *path, int nameiparent, char *name)
5325 {
5326   struct inode *ip, *next;
5327
5328   if(*path == '/')
5329     ip = iget(ROOTDEV, ROOTINO);
5330   else
5331     ip = idup(proc->cwd);
5332
5333   while((path = skipelem(path, name)) != 0){
5334     ilock(ip);
5335     if(ip->type != T_DIR){
5336       iunlockput(ip);
5337       return 0;
5338     }
5339     if(nameiparent && *path == '\0'){
5340       // Stop one level early.
5341       iunlock(ip);
5342       return ip;
5343     }
5344     if((next = dirlookup(ip, name, 0)) == 0){
5345       iunlockput(ip);
5346       return 0;
5347     }
5348     iunlockput(ip);
5349     ip = next;
```



```
5350   }
5351   if(nameiparent){
5352     iput(ip);
5353     return 0;
5354   }
5355   return ip;
5356 }
5357
5358 struct inode*
5359 namei(char *path)
5360 {
5361   char name[DIRSIZ];
5362   return namex(path, 0, name);
5363 }
5364
5365 struct inode*
5366 nameiparent(char *path, char *name)
5367 {
5368   return namex(path, 1, name);
5369 }
5370
5371
5372
5373
5374
5375
5376
5377
5378
5379
5380
5381
5382
5383
5384
5385
5386
5387
5388
5389
5390
5391
5392
5393
5394
5395
5396
5397
5398
5399
```

```
5400 //
5401 // File descriptors
5402 //
5403
5404 #include "types.h"
5405 #include "defs.h"
5406 #include "param.h"
5407 #include "fs.h"
5408 #include "file.h"
5409 #include "spinlock.h"
5410
5411 struct devsw devsw[NDEV];
5412 struct {
5413   struct spinlock lock;
5414   struct file file[NFILE];
5415 } ftable;
5416
5417 void
5418 fileinit(void)
5419 {
5420   initlock(&ftable.lock, "ftable");
5421 }
5422
5423 // Allocate a file structure.
5424 struct file*
5425 filealloc(void)
5426 {
5427   struct file *f;
5428
5429   acquire(&ftable.lock);
5430   for(f = ftable.file; f < ftable.file + NFILE; f++){
5431     if(f->ref == 0){
5432       f->ref = 1;
5433       release(&ftable.lock);
5434       return f;
5435     }
5436   }
5437   release(&ftable.lock);
5438   return 0;
5439 }
5440
5441
5442
5443
5444
5445
5446
5447
5448
5449
```



```
5450 // Increment ref count for file f.
5451 struct file*
5452 filedup(struct file *f)
5453 {
5454   acquire(&ftable.lock);
5455   if(f->ref < 1)
5456     panic("filedup");
5457   f->ref++;
5458   release(&ftable.lock);
5459   return f;
5460 }
5461
5462 // Close file f.  (Decrement ref count, close when reaches 0.)
5463 void
5464 fileclose(struct file *f)
5465 {
5466   struct file ff;
5467
5468   acquire(&ftable.lock);
5469   if(f->ref < 1)
5470     panic("fileclose");
5471   if(--f->ref > 0){
5472     release(&ftable.lock);
5473     return;
5474   }
5475   ff = *f;
5476   f->ref = 0;
5477   f->type = FD_NONE;
5478   release(&ftable.lock);
5479
5480   if(ff.type == FD_PIPE)
5481     pipeclose(ff.pipe, ff.writable);
5482   else if(ff.type == FD_INODE){
5483     begin_op();
5484     iput(ff.ip);
5485     end_op();
5486   }
5487 }
5488
5489
5490
5491
5492
5493
5494
5495
5496
5497
5498
5499
```

```
5500 // Get metadata about file f.
5501 int
5502 filestat(struct file *f, struct stat *st)
5503 {
5504   if(f->type == FD_INODE){
5505     ilock(f->ip);
5506     stati(f->ip, st);
5507     iunlock(f->ip);
5508     return 0;
5509   }
5510   return -1;
5511 }
5512
5513 // Read from file f.
5514 int
5515 fileread(struct file *f, char *addr, int n)
5516 {
5517   int r;
5518
5519   if(f->readable == 0)
5520     return -1;
5521   if(f->type == FD_PIPE)
5522     return piperead(f->pipe, addr, n);
5523   if(f->type == FD_INODE){
5524     ilock(f->ip);
5525     if((r = readi(f->ip, addr, f->off, n)) > 0)
5526       f->off += r;
5527     iunlock(f->ip);
5528     return r;
5529   }
5530   panic("fileread");
5531 }
5532
5533 // Write to file f.
5534 int
5535 filewrite(struct file *f, char *addr, int n)
5536 {
5537   int r;
5538
5539   if(f->writable == 0)
5540     return -1;
5541   if(f->type == FD_PIPE)
5542     return pipewrite(f->pipe, addr, n);
5543   if(f->type == FD_INODE){
5544     // write a few blocks at a time to avoid exceeding
5545     // the maximum log transaction size, including
5546     // i-node, indirect block, allocation blocks,
5547     // and 2 blocks of slop for non-aligned writes.
5548     // this really belongs lower down, since writei()
5549     // might be writing a device like the console.
```



```
5550     int max = ((LOGSIZE-1-1-2) / 2) * 512;
5551     int i = 0;
5552     while(i < n){
5553       int n1 = n - i;
5554       if(n1 > max)
5555         n1 = max;
5556
5557       begin_op();
5558       ilock(f->ip);
5559       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
5560         f->off += r;
5561       iunlock(f->ip);
5562       end_op();
5563
5564       if(r < 0)
5565         break;
5566       if(r != n1)
5567         panic("short filewrite");
5568       i += r;
5569     }
5570     return i == n ? n : -1;
5571   }
5572   panic("filewrite");
5573 }
5574
5575
5576
5577
5578
5579
5580
5581
5582
5583
5584
5585
5586
5587
5588
5589
5590
5591
5592
5593
5594
5595
5596
5597
5598
5599
```

```
5600 //
5601 // File-system system calls.
5602 // Mostly argument checking, since we don't trust
5603 // user code, and calls into file.c and fs.c.
5604 //
5605
5606 #include "types.h"
5607 #include "defs.h"
5608 #include "param.h"
5609 #include "stat.h"
5610 #include "mmu.h"
5611 #include "proc.h"
5612 #include "fs.h"
5613 #include "file.h"
5614 #include "fcntl.h"
5615
5616 // Fetch the nth word-sized system call argument as a file descriptor
5617 // and return both the descriptor and the corresponding struct file.
5618 static int
5619 argfd(int n, int *pfd, struct file **pf)
5620 {
5621   int fd;
5622   struct file *f;
5623
5624   if(argint(n, &fd) < 0)
5625     return -1;
5626   if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
5627     return -1;
5628   if(pfd)
5629     *pfd = fd;
5630   if(pf)
5631     *pf = f;
5632   return 0;
5633 }
5634
5635 // Allocate a file descriptor for the given file.
5636 // Takes over file reference from caller on success.
5637 static int
5638 fdalloc(struct file *f)
5639 {
5640   int fd;
5641
5642   for(fd = 0; fd < NOFILE; fd++){
5643     if(proc->ofile[fd] == 0){
5644       proc->ofile[fd] = f;
5645       return fd;
5646     }
5647   }
5648   return -1;
5649 }
```



```
5650 int
5651 sys_dup(void)
5652 {
5653   struct file *f;
5654   int fd;
5655
5656   if(argfd(0, 0, &f) < 0)
5657     return -1;
5658   if((fd=fdalloc(f)) < 0)
5659     return -1;
5660   filedup(f);
5661   return fd;
5662 }
5663
5664 int
5665 sys_read(void)
5666 {
5667   struct file *f;
5668   int n;
5669   char *p;
5670
5671   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5672     return -1;
5673   return fileread(f, p, n);
5674 }
5675
5676 int
5677 sys_write(void)
5678 {
5679   struct file *f;
5680   int n;
5681   char *p;
5682
5683   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5684     return -1;
5685   return filewrite(f, p, n);
5686 }
5687
5688 int
5689 sys_close(void)
5690 {
5691   int fd;
5692   struct file *f;
5693
5694   if(argfd(0, &fd, &f) < 0)
5695     return -1;
5696   proc->ofile[fd] = 0;
5697   fileclose(f);
5698   return 0;
5699 }
```

```
5700 int
5701 sys_fstat(void)
5702 {
5703   struct file *f;
5704   struct stat *st;
5705
5706   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
5707     return -1;
5708   return filestat(f, st);
5709 }
5710
5711 // Create the path new as a link to the same inode as old.
5712 int
5713 sys_link(void)
5714 {
5715   char name[DIRSIZ], *new, *old;
5716   struct inode *dp, *ip;
5717
5718   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
5719     return -1;
5720
5721   begin_op();
5722   if((ip = namei(old)) == 0){
5723     end_op();
5724     return -1;
5725   }
5726
5727   ilock(ip);
5728   if(ip->type == T_DIR){
5729     iunlockput(ip);
5730     end_op();
5731     return -1;
5732   }
5733
5734   ip->nlink++;
5735   iupdate(ip);
5736   iunlock(ip);
5737
5738   if((dp = nameiparent(new, name)) == 0)
5739     goto bad;
5740   ilock(dp);
5741   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
5742     iunlockput(dp);
5743     goto bad;
5744   }
5745   iunlockput(dp);
5746   iput(ip);
5747
5748   end_op();
5749
```



```
5750   return 0;
5751
5752 bad:
5753   ilock(ip);
5754   ip->nlink--;
5755   iupdate(ip);
5756   iunlockput(ip);
5757   end_op();
5758   return -1;
5759 }
5760
5761 // Is the directory dp empty except for "." and ".." ?
5762 static int
5763 isdirempty(struct inode *dp)
5764 {
5765   int off;
5766   struct dirent de;
5767
5768   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
5769     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5770       panic("isdirempty: readi");
5771     if(de.inum != 0)
5772       return 0;
5773   }
5774   return 1;
5775 }
5776
5777 int
5778 sys_unlink(void)
5779 {
5780   struct inode *ip, *dp;
5781   struct dirent de;
5782   char name[DIRSIZ], *path;
5783   uint off;
5784
5785   if(argstr(0, &path) < 0)
5786     return -1;
5787
5788   begin_op();
5789   if((dp = nameiparent(path, name)) == 0){
5790     end_op();
5791     return -1;
5792   }
5793
5794   ilock(dp);
5795
5796   // Cannot unlink "." or "..".
5797   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
5798     goto bad;
5799
```

```
5800   if((ip = dirlookup(dp, name, &off)) == 0)
5801     goto bad;
5802   ilock(ip);
5803
5804   if(ip->nlink < 1)
5805     panic("unlink: nlink < 1");
5806   if(ip->type == T_DIR && !isdirempty(ip)){
5807     iunlockput(ip);
5808     goto bad;
5809   }
5810
5811   memset(&de, 0, sizeof(de));
5812   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5813     panic("unlink: writei");
5814   if(ip->type == T_DIR){
5815     dp->nlink--;
5816     iupdate(dp);
5817   }
5818   iunlockput(dp);
5819
5820   ip->nlink--;
5821   iupdate(ip);
5822   iunlockput(ip);
5823
5824   end_op();
5825
5826   return 0;
5827
5828 bad:
5829   iunlockput(dp);
5830   end_op();
5831   return -1;
5832 }
5833
5834 static struct inode*
5835 create(char *path, short type, short major, short minor)
5836 {
5837   uint off;
5838   struct inode *ip, *dp;
5839   char name[DIRSIZ];
5840
5841   if((dp = nameiparent(path, name)) == 0)
5842     return 0;
5843   ilock(dp);
5844
5845   if((ip = dirlookup(dp, name, &off)) != 0){
5846     iunlockput(dp);
5847     ilock(ip);
5848     if(type == T_FILE && ip->type == T_FILE)
5849       return ip;
```



```
5850     iunlockput(ip);
5851     return 0;
5852   }
5853
5854   if((ip = ialloc(dp->dev, type)) == 0)
5855     panic("create: ialloc");
5856
5857   ilock(ip);
5858   ip->major = major;
5859   ip->minor = minor;
5860   ip->nlink = 1;
5861   iupdate(ip);
5862
5863   if(type == T_DIR){  // Create . and .. entries.
5864     dp->nlink++;  // for ".."
5865     iupdate(dp);
5866     // No ip->nlink++ for ".": avoid cyclic ref count.
5867     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
5868       panic("create dots");
5869   }
5870
5871   if(dirlink(dp, name, ip->inum) < 0)
5872     panic("create: dirlink");
5873
5874   iunlockput(dp);
5875
5876   return ip;
5877 }
5878
5879 int
5880 sys_open(void)
5881 {
5882   char *path;
5883   int fd, omode;
5884   struct file *f;
5885   struct inode *ip;
5886
5887   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
5888     return -1;
5889
5890   begin_op();
5891
5892   if(omode & O_CREATE){
5893     ip = create(path, T_FILE, 0, 0);
5894     if(ip == 0){
5895       end_op();
5896       return -1;
5897     }
5898   } else {
5899     if((ip = namei(path)) == 0){
```

```
5900       end_op();
5901       return -1;
5902     }
5903     ilock(ip);
5904     if(ip->type == T_DIR && omode != O_RDONLY){
5905       iunlockput(ip);
5906       end_op();
5907       return -1;
5908     }
5909   }
5910
5911   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
5912     if(f)
5913       fileclose(f);
5914     iunlockput(ip);
5915     end_op();
5916     return -1;
5917   }
5918   iunlock(ip);
5919   end_op();
5920
5921   f->type = FD_INODE;
5922   f->ip = ip;
5923   f->off = 0;
5924   f->readable = !(omode & O_WRONLY);
5925   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
5926   return fd;
5927 }
5928
5929 int
5930 sys_mkdir(void)
5931 {
5932   char *path;
5933   struct inode *ip;
5934
5935   begin_op();
5936   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
5937     end_op();
5938     return -1;
5939   }
5940   iunlockput(ip);
5941   end_op();
5942   return 0;
5943 }
5944
5945
5946
5947
5948
5949
```



```
5950 int
5951 sys_mknod(void)
5952 {
5953   struct inode *ip;
5954   char *path;
5955   int len;
5956   int major, minor;
5957
5958   begin_op();
5959   if((len=argstr(0, &path)) < 0 ||
5960      argint(1, &major) < 0 ||
5961      argint(2, &minor) < 0 ||
5962      (ip = create(path, T_DEV, major, minor)) == 0){
5963     end_op();
5964     return -1;
5965   }
5966   iunlockput(ip);
5967   end_op();
5968   return 0;
5969 }
5970
5971 int
5972 sys_chdir(void)
5973 {
5974   char *path;
5975   struct inode *ip;
5976
5977   begin_op();
5978   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
5979     end_op();
5980     return -1;
5981   }
5982   ilock(ip);
5983   if(ip->type != T_DIR){
5984     iunlockput(ip);
5985     end_op();
5986     return -1;
5987   }
5988   iunlock(ip);
5989   iput(proc->cwd);
5990   end_op();
5991   proc->cwd = ip;
5992   return 0;
5993 }
5994
5995
5996
5997
5998
5999
```

```
6000 int
6001 sys_exec(void)
6002 {
6003   char *path, *argv[MAXARG];
6004   int i;
6005   uint uargv, uarg;
6006
6007   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6008     return -1;
6009   }
6010   memset(argv, 0, sizeof(argv));
6011   for(i=0;; i++){
6012     if(i >= NELEM(argv))
6013       return -1;
6014     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6015       return -1;
6016     if(uarg == 0){
6017       argv[i] = 0;
6018       break;
6019     }
6020     if(fetchstr(uarg, &argv[i]) < 0)
6021       return -1;
6022   }
6023   return exec(path, argv);
6024 }
6025
6026 int
6027 sys_pipe(void)
6028 {
6029   int *fd;
6030   struct file *rf, *wf;
6031   int fd0, fd1;
6032
6033   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6034     return -1;
6035   if(pipealloc(&rf, &wf) < 0)
6036     return -1;
6037   fd0 = -1;
6038   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6039     if(fd0 >= 0)
6040       proc->ofile[fd0] = 0;
6041     fileclose(rf);
6042     fileclose(wf);
6043     return -1;
6044   }
6045   fd[0] = fd0;
6046   fd[1] = fd1;
6047   return 0;
6048 }
6049
```



```
6050 #include "types.h"
6051 #include "param.h"
6052 #include "memlayout.h"
6053 #include "mmu.h"
6054 #include "proc.h"
6055 #include "defs.h"
6056 #include "x86.h"
6057 #include "elf.h"
6058
6059 int
6060 exec(char *path, char **argv)
6061 {
6062   char *s, *last;
6063   int i, off;
6064   uint argc, sz, sp, ustack[3+MAXARG+1];
6065   struct elfhdr elf;
6066   struct inode *ip;
6067   struct proghdr ph;
6068   pde_t *pgdir, *oldpgdir;
6069
6070   begin_op();
6071   if((ip = namei(path)) == 0){
6072     end_op();
6073     return -1;
6074   }
6075   ilock(ip);
6076   pgdir = 0;
6077
6078   // Check ELF header
6079   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
6080     goto bad;
6081   if(elf.magic != ELF_MAGIC)
6082     goto bad;
6083
6084   if((pgdir = setupkvm()) == 0)
6085     goto bad;
6086
6087   // Load program into memory.
6088   sz = 0;
6089   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6090     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6091       goto bad;
6092     if(ph.type != ELF_PROG_LOAD)
6093       continue;
6094     if(ph.memsz < ph.filesz)
6095       goto bad;
6096     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6097       goto bad;
6098     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6099       goto bad;
```

```
  }
  iunlockput(ip);
  end_op();
  ip = 0;

  // Allocate two pages at the next page boundary.
  // Make the first inaccessible.  Use the second as the user stack.
  sz = PGROUNDUP(sz);
  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
  sp = sz;

  // Push argument strings, prepare rest of stack in ustack.
  for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
      goto bad;
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
      goto bad;
    ustack[3+argc] = sp;
  }
  ustack[3+argc] = 0;

  ustack[0] = 0xffffffff;  // fake return PC
  ustack[1] = argc;
  ustack[2] = sp - (argc+1)*4;  // argv pointer

  sp -= (3+argc+1) * 4;
  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;

  // Save program name for debugging.
  for(last=s=path; *s; s++)
    if(*s == '/')
      last = s+1;
  safestrcpy(proc->name, last, sizeof(proc->name));

  // Commit to the user image.
  oldpgdir = proc->pgdir;
  proc->pgdir = pgdir;
  proc->sz = sz;
  proc->tf->eip = elf.entry;  // main
  proc->tf->esp = sp;
  switchuvm(proc);
  freevm(oldpgdir);
  return 0;


```



```
 bad:
  if(pgdir)
    freevm(pgdir);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
}
```

```
6200 #include "types.h"
6201 #include "defs.h"
6202 #include "param.h"
6203 #include "mmu.h"
6204 #include "proc.h"
6205 #include "fs.h"
6206 #include "file.h"
6207 #include "spinlock.h"
6208
6209 #define PIPESIZE 512
6210
6211 struct pipe {
6212   struct spinlock lock;
6213   char data[PIPESIZE];
6214   uint nread;     // number of bytes read
6215   uint nwrite;    // number of bytes written
6216   int readopen;   // read fd is still open
6217   int writeopen;  // write fd is still open
6218 };
6219
6220 int
6221 pipealloc(struct file **f0, struct file **f1)
6222 {
6223   struct pipe *p;
6224
6225   p = 0;
6226   *f0 = *f1 = 0;
6227   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
6228     goto bad;
6229   if((p = (struct pipe*)kalloc()) == 0)
6230     goto bad;
6231   p->readopen = 1;
6232   p->writeopen = 1;
6233   p->nwrite = 0;
6234   p->nread = 0;
6235   initlock(&p->lock, "pipe");
6236   (*f0)->type = FD_PIPE;
6237   (*f0)->readable = 1;
6238   (*f0)->writable = 0;
6239   (*f0)->pipe = p;
6240   (*f1)->type = FD_PIPE;
6241   (*f1)->readable = 0;
6242   (*f1)->writable = 1;
6243   (*f1)->pipe = p;
6244   return 0;
6245
6246
6247
6248
6249
```



```
6250  bad:
6251   if(p)
6252     kfree((char*)p);
6253   if(*f0)
6254     fileclose(*f0);
6255   if(*f1)
6256     fileclose(*f1);
6257   return -1;
6258 }
6259
6260 void
6261 pipeclose(struct pipe *p, int writable)
6262 {
6263   acquire(&p->lock);
6264   if(writable){
6265     p->writeopen = 0;
6266     wakeup(&p->nread);
6267   } else {
6268     p->readopen = 0;
6269     wakeup(&p->nwrite);
6270   }
6271   if(p->readopen == 0 && p->writeopen == 0){
6272     release(&p->lock);
6273     kfree((char*)p);
6274   } else
6275     release(&p->lock);
6276 }
6277
6278 int
6279 pipewrite(struct pipe *p, char *addr, int n)
6280 {
6281   int i;
6282
6283   acquire(&p->lock);
6284   for(i = 0; i < n; i++){
6285     while(p->nwrite == p->nread + PIPESIZE){
6286       if(p->readopen == 0 || proc->killed){
6287         release(&p->lock);
6288         return -1;
6289       }
6290       wakeup(&p->nread);
6291       sleep(&p->nwrite, &p->lock);
6292     }
6293     p->data[p->nwrite++ % PIPESIZE] = addr[i];
6294   }
6295   wakeup(&p->nread);
6296   release(&p->lock);
6297   return n;
6298 }
6299
```

```
6300 int
6301 piperead(struct pipe *p, char *addr, int n)
6302 {
6303   int i;
6304
6305   acquire(&p->lock);
6306   while(p->nread == p->nwrite && p->writeopen){
6307     if(proc->killed){
6308       release(&p->lock);
6309       return -1;
6310     }
6311     sleep(&p->nread, &p->lock);
6312   }
6313   for(i = 0; i < n; i++){
6314     if(p->nread == p->nwrite)
6315       break;
6316     addr[i] = p->data[p->nread++ % PIPESIZE];
6317   }
6318   wakeup(&p->nwrite);
6319   release(&p->lock);
6320   return i;
6321 }
6322
6323
6324
6325
6326
6327
6328
6329
6330
6331
6332
6333
6334
6335
6336
6337
6338
6339
6340
6341
6342
6343
6344
6345
6346
6347
6348
6349
```



```
6350 #include "types.h"
6351 #include "x86.h"
6352
6353 void*
6354 memset(void *dst, int c, uint n)
6355 {
6356   if ((int)dst%4 == 0 && n%4 == 0){
6357     c &= 0xFF;
6358     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
6359   } else
6360     stosb(dst, c, n);
6361   return dst;
6362 }
6363
6364 int
6365 memcmp(const void *v1, const void *v2, uint n)
6366 {
6367   const uchar *s1, *s2;
6368
6369   s1 = v1;
6370   s2 = v2;
6371   while(n-- > 0){
6372     if(*s1 != *s2)
6373       return *s1 - *s2;
6374     s1++, s2++;
6375   }
6376
6377   return 0;
6378 }
6379
6380 void*
6381 memmove(void *dst, const void *src, uint n)
6382 {
6383   const char *s;
6384   char *d;
6385
6386   s = src;
6387   d = dst;
6388   if(s < d && s + n > d){
6389     s += n;
6390     d += n;
6391     while(n-- > 0)
6392       *--d = *--s;
6393   } else
6394     while(n-- > 0)
6395       *d++ = *s++;
6396
6397   return dst;
6398 }
6399
```

```
6400 // memcpy exists to placate GCC.  Use memmove.
6401 void*
6402 memcpy(void *dst, const void *src, uint n)
6403 {
6404   return memmove(dst, src, n);
6405 }
6406
6407 int
6408 strncmp(const char *p, const char *q, uint n)
6409 {
6410   while(n > 0 && *p && *p == *q)
6411     n--, p++, q++;
6412   if(n == 0)
6413     return 0;
6414   return (uchar)*p - (uchar)*q;
6415 }
6416
6417 char*
6418 strncpy(char *s, const char *t, int n)
6419 {
6420   char *os;
6421
6422   os = s;
6423   while(n-- > 0 && (*s++ = *t++) != 0)
6424     ;
6425   while(n-- > 0)
6426     *s++ = 0;
6427   return os;
6428 }
6429
6430 // Like strncpy but guaranteed to NUL-terminate.
6431 char*
6432 safestrcpy(char *s, const char *t, int n)
6433 {
6434   char *os;
6435
6436   os = s;
6437   if(n <= 0)
6438     return os;
6439   while(--n > 0 && (*s++ = *t++) != 0)
6440     ;
6441   *s = 0;
6442   return os;
6443 }
6444
6445
6446
6447
6448
6449
```



```
6450 int
6451 strlen(const char *s)
6452 {
6453   int n;
6454
6455   for(n = 0; s[n]; n++)
6456     ;
6457   return n;
6458 }
6459
6460
6461
6462
6463
6464
6465
6466
6467
6468
6469
6470
6471
6472
6473
6474
6475
6476
6477
6478
6479
6480
6481
6482
6483
6484
6485
6486
6487
6488
6489
6490
6491
6492
6493
6494
6495
6496
6497
6498
6499
```

```
6500 // See MultiProcessor Specification Version 1.[14]
6501
6502 struct mp {             // floating pointer
6503   uchar signature[4];           // "_MP_"
6504   void *physaddr;               // phys addr of MP config table
6505   uchar length;                 // 1
6506   uchar specrev;                // [14]
6507   uchar checksum;               // all bytes must add up to 0
6508   uchar type;                   // MP system config type
6509   uchar imcrp;
6510   uchar reserved[3];
6511 };
6512
6513 struct mpconf {         // configuration table header
6514   uchar signature[4];           // "PCMP"
6515   ushort length;                // total table length
6516   uchar version;                // [14]
6517   uchar checksum;               // all bytes must add up to 0
6518   uchar product[20];            // product id
6519   uint *oemtable;               // OEM table pointer
6520   ushort oemlength;             // OEM table length
6521   ushort entry;                 // entry count
6522   uint *lapicaddr;              // address of local APIC
6523   ushort xlength;               // extended table length
6524   uchar xchecksum;              // extended table checksum
6525   uchar reserved;
6526 };
6527
6528 struct mpproc {         // processor table entry
6529   uchar type;                   // entry type (0)
6530   uchar apicid;                 // local APIC id
6531   uchar version;                // local APIC verison
6532   uchar flags;                  // CPU flags
6533     #define MPBOOT 0x02           // This proc is the bootstrap processor.
6534   uchar signature[4];           // CPU signature
6535   uint feature;                 // feature flags from CPUID instruction
6536   uchar reserved[8];
6537 };
6538
6539 struct mpioapic {       // I/O APIC table entry
6540   uchar type;                   // entry type (2)
6541   uchar apicno;                 // I/O APIC id
6542   uchar version;                // I/O APIC version
6543   uchar flags;                  // I/O APIC flags
6544   uint *addr;                  // I/O APIC address
6545 };
6546
6547
6548
6549
```



```
6550 // Table entry types
6551 #define MPPROC    0x00  // One per processor
6552 #define MPBUS     0x01  // One per bus
6553 #define MPIOAPIC  0x02  // One per I/O APIC
6554 #define MPIOINTR  0x03  // One per bus interrupt source
6555 #define MPLINTR   0x04  // One per system interrupt source
6556
6557 // Blank page.
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 // Multiprocessor support
6601 // Search memory for MP description structures.
6602 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
6603
6604 #include "types.h"
6605 #include "defs.h"
6606 #include "param.h"
6607 #include "memlayout.h"
6608 #include "mp.h"
6609 #include "x86.h"
6610 #include "mmu.h"
6611 #include "proc.h"
6612
6613 struct cpu cpus[NCPU];
6614 static struct cpu *bcpu;
6615 int ismp;
6616 int ncpu;
6617 uchar ioapicid;
6618
6619 int
6620 mpbcpu(void)
6621 {
6622   return bcpu-cpus;
6623 }
6624
6625 static uchar
6626 sum(uchar *addr, int len)
6627 {
6628   int i, sum;
6629
6630   sum = 0;
6631   for(i=0; i<len; i++)
6632     sum += addr[i];
6633   return sum;
6634 }
6635
6636 // Look for an MP structure in the len bytes at addr.
6637 static struct mp*
6638 mpsearch1(uint a, int len)
6639 {
6640   uchar *e, *p, *addr;
6641
6642   addr = p2v(a);
6643   e = addr+len;
6644   for(p = addr; p < e; p += sizeof(struct mp))
6645     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
6646       return (struct mp*)p;
6647   return 0;
6648 }
6649
```



```
6650 // Search for the MP Floating Pointer Structure, which according to the
6651 // spec is in one of the following three locations:
6652 // 1) in the first KB of the EBDA;
6653 // 2) in the last KB of system base memory;
6654 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
6655 static struct mp*
6656 mpsearch(void)
6657 {
6658   uchar *bda;
6659   uint p;
6660   struct mp *mp;
6661
6662   bda = (uchar *) P2V(0x400);
6663   if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
6664     if((mp = mpsearch1(p, 1024)))
6665       return mp;
6666   } else {
6667     p = ((bda[0x14]<<8)|bda[0x13])*1024;
6668     if((mp = mpsearch1(p-1024, 1024)))
6669       return mp;
6670   }
6671   return mpsearch1(0xF0000, 0x10000);
6672 }
6673
6674 // Search for an MP configuration table.  For now,
6675 // don't accept the default configurations (physaddr == 0).
6676 // Check for correct signature, calculate the checksum and,
6677 // if correct, check the version.
6678 // To do: check extended table checksum.
6679 static struct mpconf*
6680 mpconfig(struct mp **pmp)
6681 {
6682   struct mpconf *conf;
6683   struct mp *mp;
6684
6685   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
6686     return 0;
6687   conf = (struct mpconf*) p2v((uint) mp->physaddr);
6688   if(memcmp(conf, "PCMP", 4) != 0)
6689     return 0;
6690   if(conf->version != 1 && conf->version != 4)
6691     return 0;
6692   if(sum((uchar*)conf, conf->length) != 0)
6693     return 0;
6694   *pmp = mp;
6695   return conf;
6696 }
6697
6698
6699
```

```
6700 void
6701 mpinit(void)
6702 {
6703   uchar *p, *e;
6704   struct mp *mp;
6705   struct mpconf *conf;
6706   struct mpproc *proc;
6707   struct mpioapic *ioapic;
6708
6709   bcpu = &cpus[0];
6710   if((conf = mpconfig(&mp)) == 0)
6711     return;
6712   ismp = 1;
6713   lapic = (uint*)conf->lapicaddr;
6714   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
6715     switch(*p){
6716     case MPPROC:
6717       proc = (struct mpproc*)p;
6718       if(ncpu != proc->apicid){
6719         cprintf("mpinit: ncpu=%d apicid=%d\n", ncpu, proc->apicid);
6720         ismp = 0;
6721       }
6722       if(proc->flags & MPBOOT)
6723         bcpu = &cpus[ncpu];
6724       cpus[ncpu].id = ncpu;
6725       ncpu++;
6726       p += sizeof(struct mpproc);
6727       continue;
6728     case MPIOAPIC:
6729       ioapic = (struct mpioapic*)p;
6730       ioapicid = ioapic->apicno;
6731       p += sizeof(struct mpioapic);
6732       continue;
6733     case MPBUS:
6734     case MPIOINTR:
6735     case MPLINTR:
6736       p += 8;
6737       continue;
6738     default:
6739       cprintf("mpinit: unknown config type %x\n", *p);
6740       ismp = 0;
6741     }
6742   }
6743   if(!ismp){
6744     // Didn't like what we found; fall back to no MP.
6745     ncpu = 1;
6746     lapic = 0;
6747     ioapicid = 0;
6748     return;
6749   }
```



```
6750   if(mp->imcrp){
6751     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
6752     // But it would on real hardware.
6753     outb(0x22, 0x70);   // Select IMCR
6754     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
6755   }
6756 }
6757
6758
6759
6760
6761
6762
6763
6764
6765
6766
6767
6768
6769
6770
6771
6772
6773
6774
6775
6776
6777
6778
6779
6780
6781
6782
6783
6784
6785
6786
6787
6788
6789
6790
6791
6792
6793
6794
6795
6796
6797
6798
6799
```

```
6800 // The local APIC manages internal (non-I/O) interrupts.
6801 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
6802 // As of 7/26/2016, Intel processor manual Chapter 10 of Volume 3
6803
6804 #include "types.h"
6805 #include "defs.h"
6806 #include "date.h"
6807 #include "memlayout.h"
6808 #include "traps.h"
6809 #include "mmu.h"
6810 #include "x86.h"
6811
6812 // Local APIC registers, divided by 4 for use as uint[] indices.
6813 #define ID      (0x0020/4)   // ID
6814 #define VER     (0x0030/4)   // Version
6815 #define TPR     (0x0080/4)   // Task Priority
6816 #define EOI     (0x00B0/4)   // EOI
6817 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
6818   #define ENABLE     0x00000100   // Unit Enable
6819 #define ESR     (0x0280/4)   // Error Status
6820 #define ICRLO   (0x0300/4)   // Interrupt Command
6821   #define INIT       0x00000500   // INIT/RESET
6822   #define STARTUP    0x00000600   // Startup IPI
6823   #define DELIVS     0x00001000   // Delivery status
6824   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
6825   #define DEASSERT   0x00000000
6826   #define LEVEL      0x00008000   // Level triggered
6827   #define BCAST      0x00080000   // Send to all APICs, including self.
6828   #define BUSY       0x00001000
6829   #define FIXED      0x00000000
6830 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
6831 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
6832   #define X1         0x0000000B   // divide counts by 1
6833   #define PERIODIC   0x00020000   // Periodic
6834 #define PCINT   (0x0340/4)   // Performance Counter LVT
6835 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
6836 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
6837 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
6838   #define MASKED     0x00010000   // Interrupt masked
6839 #define TICR    (0x0380/4)   // Timer Initial Count
6840 #define TCCR    (0x0390/4)   // Timer Current Count
6841 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
6842
6843 volatile uint *lapic;  // Initialized in mp.c
6844
6845 static void
6846 lapicw(int index, int value)
6847 {
6848   lapic[index] = value;
6849   lapic[ID];  // wait for write to finish, by reading
```



```
6850 }
6851
6852 void
6853 lapicinit(void)
6854 {
6855   if(!lapic)
6856     return;
6857
6858   // Enable local APIC; set spurious interrupt vector.
6859   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
6860
6861   // The timer repeatedly counts down at bus frequency
6862   // from lapic[TICR] and then issues an interrupt.
6863   // If xv6 cared more about precise timekeeping,
6864   // TICR would be calibrated using an external time source.
6865   lapicw(TDCR, X1);
6866   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
6867   lapicw(TICR, 10000000);
6868
6869   // Disable logical interrupt lines.
6870   lapicw(LINT0, MASKED);
6871   lapicw(LINT1, MASKED);
6872
6873   // Disable performance counter overflow interrupts
6874   // on machines that provide that interrupt entry.
6875   if(((lapic[VER]>>16) & 0xFF) >= 4)
6876     lapicw(PCINT, MASKED);
6877
6878   // Map error interrupt to IRQ_ERROR.
6879   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
6880
6881   // Clear error status register (requires back-to-back writes).
6882   lapicw(ESR, 0);
6883   lapicw(ESR, 0);
6884
6885   // Ack any outstanding interrupts.
6886   lapicw(EOI, 0);
6887
6888   // Send an Init Level De-Assert to synchronise arbitration ID's.
6889   lapicw(ICRHI, 0);
6890   lapicw(ICRLO, BCAST | INIT | LEVEL);
6891   while(lapic[ICRLO] & DELIVS)
6892     ;
6893
6894   // Enable interrupts on the APIC (but not on the processor).
6895   lapicw(TPR, 0);
6896 }
6897
6898
6899
```

```
6900 int
6901 cpunum(void)
6902 {
6903   // Cannot call cpu when interrupts are enabled:
6904   // result not guaranteed to last long enough to be used!
6905   // Would prefer to panic but even printing is chancy here:
6906   // almost everything, including cprintf and panic, calls cpu,
6907   // often indirectly through acquire and release.
6908   if(readeflags()&FL_IF){
6909     static int n;
6910     if(n++ == 0)
6911       cprintf("cpu called from %x with interrupts enabled\n",
6912         __builtin_return_address(0));
6913   }
6914
6915   if(lapic)
6916     return lapic[ID]>>24;
6917   return 0;
6918 }
6919
6920 // Acknowledge interrupt.
6921 void
6922 lapiceoi(void)
6923 {
6924   if(lapic)
6925     lapicw(EOI, 0);
6926 }
6927
6928 // Spin for a given number of microseconds.
6929 // On real hardware would want to tune this dynamically.
6930 void
6931 microdelay(int us)
6932 {
6933 }
6934
6935 #define CMOS_PORT    0x70
6936 #define CMOS_RETURN  0x71
6937
6938 // Start additional processor running entry code at addr.
6939 // See Appendix B of MultiProcessor Specification.
6940 void
6941 lapicstartap(uchar apicid, uint addr)
6942 {
6943   int i;
6944   ushort *wrv;
6945
6946   // "The BSP must initialize CMOS shutdown code to 0AH
6947   // and the warm reset vector (DWORD based at 40:67) to point at
6948   // the AP startup code prior to the [universal startup algorithm]."
6949   outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
```



```
6950   outb(CMOS_PORT+1, 0x0A);
6951   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
6952   wrv[0] = 0;
6953   wrv[1] = addr >> 4;
6954
6955   // "Universal startup algorithm."
6956   // Send INIT (level-triggered) interrupt to reset other CPU.
6957   lapicw(ICRHI, apicid<<24);
6958   lapicw(ICRLO, INIT | LEVEL | ASSERT);
6959   microdelay(200);
6960   lapicw(ICRLO, INIT | LEVEL);
6961   microdelay(100);    // should be 10ms, but too slow in Bochs!
6962
6963   // Send startup IPI (twice!) to enter code.
6964   // Regular hardware is supposed to only accept a STARTUP
6965   // when it is in the halted state due to an INIT.  So the second
6966   // should be ignored, but it is part of the official Intel algorithm.
6967   // Bochs complains about the second one.  Too bad for Bochs.
6968   for(i = 0; i < 2; i++){
6969     lapicw(ICRHI, apicid<<24);
6970     lapicw(ICRLO, STARTUP | (addr>>12));
6971     microdelay(200);
6972   }
6973 }
6974
6975 #define CMOS_STATA   0x0a
6976 #define CMOS_STATB   0x0b
6977 #define CMOS_UIP    (1 << 7)        // RTC update in progress
6978
6979 #define SECS    0x00
6980 #define MINS    0x02
6981 #define HOURS   0x04
6982 #define DAY     0x07
6983 #define MONTH   0x08
6984 #define YEAR    0x09
6985
6986 static uint cmos_read(uint reg)
6987 {
6988   outb(CMOS_PORT,  reg);
6989   microdelay(200);
6990
6991   return inb(CMOS_RETURN);
6992 }
6993
6994
6995
6996
6997
6998
6999
```

```
7000 static void fill_rtcdate(struct rtcdate *r)
7001 {
7002   r->second = cmos_read(SECS);
7003   r->minute = cmos_read(MINS);
7004   r->hour   = cmos_read(HOURS);
7005   r->day    = cmos_read(DAY);
7006   r->month  = cmos_read(MONTH);
7007   r->year   = cmos_read(YEAR);
7008 }
7009
7010 // qemu seems to use 24-hour GWT and the values are BCD encoded
7011 void cmostime(struct rtcdate *r)
7012 {
7013   struct rtcdate t1, t2;
7014   int sb, bcd;
7015
7016   sb = cmos_read(CMOS_STATB);
7017
7018   bcd = (sb & (1 << 2)) == 0;
7019
7020   // make sure CMOS doesn't modify time while we read it
7021   for (;;) {
7022     fill_rtcdate(&t1);
7023     if (cmos_read(CMOS_STATA) & CMOS_UIP)
7024         continue;
7025     fill_rtcdate(&t2);
7026     if (memcmp(&t1, &t2, sizeof(t1)) == 0)
7027       break;
7028   }
7029
7030   // convert
7031   if (bcd) {
7032 #define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
7033     CONV(second);
7034     CONV(minute);
7035     CONV(hour  );
7036     CONV(day   );
7037     CONV(month );
7038     CONV(year  );
7039 #undef     CONV
7040   }
7041
7042   *r = t1;
7043   r->year += 2000;
7044 }
7045
7046
7047
7048
7049
```



```
7050 // The I/O APIC manages hardware interrupts for an SMP system.
7051 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
7052 // See also picirq.c.
7053
7054 #include "types.h"
7055 #include "defs.h"
7056 #include "traps.h"
7057
7058 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
7059
7060 #define REG_ID     0x00  // Register index: ID
7061 #define REG_VER    0x01  // Register index: version
7062 #define REG_TABLE  0x10  // Redirection table base
7063
7064 // The redirection table starts at REG_TABLE and uses
7065 // two registers to configure each interrupt.
7066 // The first (low) register in a pair contains configuration bits.
7067 // The second (high) register contains a bitmask telling which
7068 // CPUs can serve that interrupt.
7069 #define INT_DISABLED   0x00010000  // Interrupt disabled
7070 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
7071 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
7072 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
7073
7074 volatile struct ioapic *ioapic;
7075
7076 // IO APIC MMIO structure: write reg, then read or write data.
7077 struct ioapic {
7078   uint reg;
7079   uint pad[3];
7080   uint data;
7081 };
7082
7083 static uint
7084 ioapicread(int reg)
7085 {
7086   ioapic->reg = reg;
7087   return ioapic->data;
7088 }
7089
7090 static void
7091 ioapicwrite(int reg, uint data)
7092 {
7093   ioapic->reg = reg;
7094   ioapic->data = data;
7095 }
7096
7097
7098
7099
```

```
7100 void
7101 ioapicinit(void)
7102 {
7103   int i, id, maxintr;
7104
7105   if(!ismp)
7106     return;
7107
7108   ioapic = (volatile struct ioapic*)IOAPIC;
7109   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
7110   id = ioapicread(REG_ID) >> 24;
7111   if(id != ioapicid)
7112     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
7113
7114   // Mark all interrupts edge-triggered, active high, disabled,
7115   // and not routed to any CPUs.
7116   for(i = 0; i <= maxintr; i++){
7117     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
7118     ioapicwrite(REG_TABLE+2*i+1, 0);
7119   }
7120 }
7121
7122 void
7123 ioapicenable(int irq, int cpunum)
7124 {
7125   if(!ismp)
7126     return;
7127
7128   // Mark interrupt edge-triggered, active high,
7129   // enabled, and routed to the given cpunum,
7130   // which happens to be that cpu's APIC ID.
7131   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
7132   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
7133 }
7134
7135
7136
7137
7138
7139
7140
7141
7142
7143
7144
7145
7146
7147
7148
7149
```



```
7150 // Intel 8259A programmable interrupt controllers.
7151
7152 #include "types.h"
7153 #include "x86.h"
7154 #include "traps.h"
7155
7156 // I/O Addresses of the two programmable interrupt controllers
7157 #define IO_PIC1         0x20    // Master (IRQs 0-7)
7158 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
7159
7160 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
7161
7162 // Current IRQ mask.
7163 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
7164 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
7165
7166 static void
7167 picsetmask(ushort mask)
7168 {
7169   irqmask = mask;
7170   outb(IO_PIC1+1, mask);
7171   outb(IO_PIC2+1, mask >> 8);
7172 }
7173
7174 void
7175 picenable(int irq)
7176 {
7177   picsetmask(irqmask & ~(1<<irq));
7178 }
7179
7180 // Initialize the 8259A interrupt controllers.
7181 void
7182 picinit(void)
7183 {
7184   // mask all interrupts
7185   outb(IO_PIC1+1, 0xFF);
7186   outb(IO_PIC2+1, 0xFF);
7187
7188   // Set up master (8259A-1)
7189
7190   // ICW1:  0001g0hi
7191   //    g:  0 = edge triggering, 1 = level triggering
7192   //    h:  0 = cascaded PICs, 1 = master only
7193   //    i:  0 = no ICW4, 1 = ICW4 required
7194   outb(IO_PIC1, 0x11);
7195
7196   // ICW2:  Vector offset
7197   outb(IO_PIC1+1, T_IRQ0);
7198
7199
```

```
7200   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
7201   //        (slave PIC) 3-bit # of slave's connection to master
7202   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
7203
7204   // ICW4:  000nbmap
7205   //    n:  1 = special fully nested mode
7206   //    b:  1 = buffered mode
7207   //    m:  0 = slave PIC, 1 = master PIC
7208   //      (ignored when b is 0, as the master/slave role
7209   //      can be hardwired).
7210   //    a:  1 = Automatic EOI mode
7211   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
7212   outb(IO_PIC1+1, 0x3);
7213
7214   // Set up slave (8259A-2)
7215   outb(IO_PIC2, 0x11);                  // ICW1
7216   outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
7217   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
7218   // NB Automatic EOI mode doesn't tend to work on the slave.
7219   // Linux source code says it's "to be investigated".
7220   outb(IO_PIC2+1, 0x3);                 // ICW4
7221
7222   // OCW3:  0ef01prs
7223   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
7224   //    p:  0 = no polling, 1 = polling mode
7225   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
7226   outb(IO_PIC1, 0x68);             // clear specific mask
7227   outb(IO_PIC1, 0x0a);             // read IRR by default
7228
7229   outb(IO_PIC2, 0x68);             // OCW3
7230   outb(IO_PIC2, 0x0a);             // OCW3
7231
7232   if(irqmask != 0xFFFF)
7233     picsetmask(irqmask);
7234 }
7235
7236
7237
7238
7239
7240
7241
7242
7243
7244
7245
7246
7247
7248
7249
```



```
7250 // PC keyboard interface constants
7251
7252 #define KBSTATP         0x64    // kbd controller status port(I)
7253 #define KBS_DIB         0x01    // kbd data in buffer
7254 #define KBDATAP         0x60    // kbd data port(I)
7255
7256 #define NO              0
7257
7258 #define SHIFT           (1<<0)
7259 #define CTL             (1<<1)
7260 #define ALT             (1<<2)
7261
7262 #define CAPSLOCK        (1<<3)
7263 #define NUMLOCK         (1<<4)
7264 #define SCROLLLOCK      (1<<5)
7265
7266 #define E0ESC           (1<<6)
7267
7268 // Special keycodes
7269 #define KEY_HOME        0xE0
7270 #define KEY_END         0xE1
7271 #define KEY_UP          0xE2
7272 #define KEY_DN          0xE3
7273 #define KEY_LF          0xE4
7274 #define KEY_RT          0xE5
7275 #define KEY_PGUP        0xE6
7276 #define KEY_PGDN        0xE7
7277 #define KEY_INS         0xE8
7278 #define KEY_DEL         0xE9
7279
7280 // C('A') == Control-A
7281 #define C(x) (x - '@')
7282
7283 static uchar shiftcode[256] =
7284 {
7285   [0x1D] CTL,
7286   [0x2A] SHIFT,
7287   [0x36] SHIFT,
7288   [0x38] ALT,
7289   [0x9D] CTL,
7290   [0xB8] ALT
7291 };
7292
7293 static uchar togglecode[256] =
7294 {
7295   [0x3A] CAPSLOCK,
7296   [0x45] NUMLOCK,
7297   [0x46] SCROLLLOCK
7298 };
7299
```

```
7300 static uchar normalmap[256] =
7301 {
7302   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
7303   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
7304   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
7305   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
7306   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
7307   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
7308   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
7309   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7310   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7311   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7312   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7313   [0x9C] '\n',      // KP_Enter
7314   [0xB5] '/',       // KP_Div
7315   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7316   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7317   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7318   [0x97] KEY_HOME,  [0xCF] KEY_END,
7319   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7320 };
7321
7322 static uchar shiftmap[256] =
7323 {
7324   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
7325   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
7326   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
7327   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
7328   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
7329   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
7330   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
7331   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7332   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7333   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7334   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7335   [0x9C] '\n',      // KP_Enter
7336   [0xB5] '/',       // KP_Div
7337   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7338   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7339   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7340   [0x97] KEY_HOME,  [0xCF] KEY_END,
7341   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7342 };
7343
7344
7345
7346
7347
7348
7349
```



```
7350 static uchar ctlmap[256] =
7351 {
7352   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7353   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7354   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
7355   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
7356   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
7357   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
7358   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
7359   [0x9C] '\r',      // KP_Enter
7360   [0xB5] C('/'),    // KP_Div
7361   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7362   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7363   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7364   [0x97] KEY_HOME,  [0xCF] KEY_END,
7365   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7366 };
7367
7368
7369
7370
7371
7372
7373
7374
7375
7376
7377
7378
7379
7380
7381
7382
7383
7384
7385
7386
7387
7388
7389
7390
7391
7392
7393
7394
7395
7396
7397
7398
7399
```

```
7400 #include "types.h"
7401 #include "x86.h"
7402 #include "defs.h"
7403 #include "kbd.h"
7404
7405 int
7406 kbdgetc(void)
7407 {
7408   static uint shift;
7409   static uchar *charcode[4] = {
7410     normalmap, shiftmap, ctlmap, ctlmap
7411   };
7412   uint st, data, c;
7413
7414   st = inb(KBSTATP);
7415   if((st & KBS_DIB) == 0)
7416     return -1;
7417   data = inb(KBDATAP);
7418
7419   if(data == 0xE0){
7420     shift |= E0ESC;
7421     return 0;
7422   } else if(data & 0x80){
7423     // Key released
7424     data = (shift & E0ESC ? data : data & 0x7F);
7425     shift &= ~(shiftcode[data] | E0ESC);
7426     return 0;
7427   } else if(shift & E0ESC){
7428     // Last character was an E0 escape; or with 0x80
7429     data |= 0x80;
7430     shift &= ~E0ESC;
7431   }
7432
7433   shift |= shiftcode[data];
7434   shift ^= togglecode[data];
7435   c = charcode[shift & (CTL | SHIFT)][data];
7436   if(shift & CAPSLOCK){
7437     if('a' <= c && c <= 'z')
7438       c += 'A' - 'a';
7439     else if('A' <= c && c <= 'Z')
7440       c += 'a' - 'A';
7441   }
7442   return c;
7443 }
7444
7445 void
7446 kbdintr(void)
7447 {
7448   consoleintr(kbdgetc);
7449 }
```



```
7450 // Console input and output.
7451 // Input is from the keyboard or serial port.
7452 // Output is written to the screen and serial port.
7453
7454 #include "types.h"
7455 #include "defs.h"
7456 #include "param.h"
7457 #include "traps.h"
7458 #include "spinlock.h"
7459 #include "fs.h"
7460 #include "file.h"
7461 #include "memlayout.h"
7462 #include "mmu.h"
7463 #include "proc.h"
7464 #include "x86.h"
7465
7466 static void consputc(int);
7467
7468 static int panicked = 0;
7469
7470 static struct {
7471   struct spinlock lock;
7472   int locking;
7473 } cons;
7474
7475 static void
7476 printint(int xx, int base, int sign)
7477 {
7478   static char digits[] = "0123456789abcdef";
7479   char buf[16];
7480   int i;
7481   uint x;
7482
7483   if(sign && (sign = xx < 0))
7484     x = -xx;
7485   else
7486     x = xx;
7487
7488   i = 0;
7489   do{
7490     buf[i++] = digits[x % base];
7491   }while((x /= base) != 0);
7492
7493   if(sign)
7494     buf[i++] = '-';
7495
7496   while(--i >= 0)
7497     consputc(buf[i]);
7498 }
7499
```

```
7500 // Print to the console. only understands %d, %x, %p, %s.
7501 void
7502 cprintf(char *fmt, ...)
7503 {
7504   int i, c, locking;
7505   uint *argp;
7506   char *s;
7507
7508   locking = cons.locking;
7509   if(locking)
7510     acquire(&cons.lock);
7511
7512   if (fmt == 0)
7513     panic("null fmt");
7514
7515   argp = (uint*)(void*)(&fmt + 1);
7516   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
7517     if(c != '%'){
7518       consputc(c);
7519       continue;
7520     }
7521     c = fmt[++i] & 0xff;
7522     if(c == 0)
7523       break;
7524     switch(c){
7525     case 'd':
7526       printint(*argp++, 10, 1);
7527       break;
7528     case 'x':
7529     case 'p':
7530       printint(*argp++, 16, 0);
7531       break;
7532     case 's':
7533       if((s = (char*)*argp++) == 0)
7534         s = "(null)";
7535       for(; *s; s++)
7536         consputc(*s);
7537       break;
7538     case '%':
7539       consputc('%');
7540       break;
7541     default:
7542       // Print unknown % sequence to draw attention.
7543       consputc('%');
7544       consputc(c);
7545       break;
7546     }
7547   }
7548
7549
```



```
7550   if(locking)
7551     release(&cons.lock);
7552 }
7553
7554 void
7555 panic(char *s)
7556 {
7557   int i;
7558   uint pcs[10];
7559
7560   cli();
7561   cons.locking = 0;
7562   cprintf("cpu%d: panic: ", cpu->id);
7563   cprintf(s);
7564   cprintf("\n");
7565   getcallerpcs(&s, pcs);
7566   for(i=0; i<10; i++)
7567     cprintf(" %p", pcs[i]);
7568   panicked = 1; // freeze other CPU
7569   for(;;)
7570     ;
7571 }
7572
7573 #define BACKSPACE 0x100
7574 #define CRTPORT 0x3d4
7575 static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory
7576
7577 static void
7578 cgaputc(int c)
7579 {
7580   int pos;
7581
7582   // Cursor position: col + 80*row.
7583   outb(CRTPORT, 14);
7584   pos = inb(CRTPORT+1) << 8;
7585   outb(CRTPORT, 15);
7586   pos |= inb(CRTPORT+1);
7587
7588   if(c == '\n')
7589     pos += 80 - pos%80;
7590   else if(c == BACKSPACE){
7591     if(pos > 0) --pos;
7592   } else
7593     crt[pos++] = (c&0xff) | 0x0700;  // black on white
7594
7595   if(pos < 0 || pos > 25*80)
7596     panic("pos under/overflow");
7597
7598
7599
```

```
7600   if((pos/80) >= 24){  // Scroll up.
7601     memmove(crt, crt+80, sizeof(crt[0])*23*80);
7602     pos -= 80;
7603     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
7604   }
7605
7606   outb(CRTPORT, 14);
7607   outb(CRTPORT+1, pos>>8);
7608   outb(CRTPORT, 15);
7609   outb(CRTPORT+1, pos);
7610   crt[pos] = ' ' | 0x0700;
7611 }
7612
7613 void
7614 consputc(int c)
7615 {
7616   if(panicked){
7617     cli();
7618     for(;;)
7619       ;
7620   }
7621
7622   if(c == BACKSPACE){
7623     uartputc('\b'); uartputc(' '); uartputc('\b');
7624   } else
7625     uartputc(c);
7626   cgaputc(c);
7627 }
7628
7629 #define INPUT_BUF 128
7630 struct {
7631   char buf[INPUT_BUF];
7632   uint r;  // Read index
7633   uint w;  // Write index
7634   uint e;  // Edit index
7635 } input;
7636
7637 #define C(x)  ((x)-'@')  // Control-x
7638
7639 void
7640 consoleintr(int (*getc)(void))
7641 {
7642   int c, doprocdump = 0;
7643
7644   acquire(&cons.lock);
7645   while((c = getc()) >= 0){
7646     switch(c){
7647     case C('P'):  // Process listing.
7648       doprocdump = 1;   // procdump() locks cons.lock indirectly; invoke lat
7649       break;
```



```
7650     case C('U'):  // Kill line.
7651       while(input.e != input.w &&
7652             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
7653         input.e--;
7654         consputc(BACKSPACE);
7655       }
7656       break;
7657     case C('H'): case '\x7f':  // Backspace
7658       if(input.e != input.w){
7659         input.e--;
7660         consputc(BACKSPACE);
7661       }
7662       break;
7663     default:
7664       if(c != 0 && input.e-input.r < INPUT_BUF){
7665         c = (c == '\r') ? '\n' : c;
7666         input.buf[input.e++ % INPUT_BUF] = c;
7667         consputc(c);
7668         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
7669           input.w = input.e;
7670           wakeup(&input.r);
7671         }
7672       }
7673       break;
7674     }
7675   }
7676   release(&cons.lock);
7677   if(doprocdump) {
7678     procdump();  // now call procdump() wo. cons.lock held
7679   }
7680 }
7681
7682 int
7683 consoleread(struct inode *ip, char *dst, int n)
7684 {
7685   uint target;
7686   int c;
7687
7688   iunlock(ip);
7689   target = n;
7690   acquire(&cons.lock);
7691   while(n > 0){
7692     while(input.r == input.w){
7693       if(proc->killed){
7694         release(&cons.lock);
7695         ilock(ip);
7696         return -1;
7697       }
7698       sleep(&input.r, &cons.lock);
7699     }
```

```
7700     c = input.buf[input.r++ % INPUT_BUF];
7701     if(c == C('D')){  // EOF
7702       if(n < target){
7703         // Save ^D for next time, to make sure
7704         // caller gets a 0-byte result.
7705         input.r--;
7706       }
7707       break;
7708     }
7709     *dst++ = c;
7710     --n;
7711     if(c == '\n')
7712       break;
7713   }
7714   release(&cons.lock);
7715   ilock(ip);
7716
7717   return target - n;
7718 }
7719
7720 int
7721 consolewrite(struct inode *ip, char *buf, int n)
7722 {
7723   int i;
7724
7725   iunlock(ip);
7726   acquire(&cons.lock);
7727   for(i = 0; i < n; i++)
7728     consputc(buf[i] & 0xff);
7729   release(&cons.lock);
7730   ilock(ip);
7731
7732   return n;
7733 }
7734
7735 void
7736 consoleinit(void)
7737 {
7738   initlock(&cons.lock, "console");
7739
7740   devsw[CONSOLE].write = consolewrite;
7741   devsw[CONSOLE].read = consoleread;
7742   cons.locking = 1;
7743
7744   picenable(IRQ_KBD);
7745   ioapicenable(IRQ_KBD, 0);
7746 }
7747
7748
7749
```



```
7750 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
7751 // Only used on uniprocessors;
7752 // SMP machines use the local APIC timer.
7753
7754 #include "types.h"
7755 #include "defs.h"
7756 #include "traps.h"
7757 #include "x86.h"
7758
7759 #define IO_TIMER1       0x040           // 8253 Timer #1
7760
7761 // Frequency of all three count-down timers;
7762 // (TIMER_FREQ/freq) is the appropriate count
7763 // to generate a frequency of freq Hz.
7764
7765 #define TIMER_FREQ      1193182
7766 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
7767
7768 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
7769 #define TIMER_SEL0      0x00    // select counter 0
7770 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
7771 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
7772
7773 void
7774 timerinit(void)
7775 {
7776   // Interrupt 100 times/sec.
7777   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
7778   outb(IO_TIMER1, TIMER_DIV(100) % 256);
7779   outb(IO_TIMER1, TIMER_DIV(100) / 256);
7780   picenable(IRQ_TIMER);
7781 }
7782
7783
7784
7785
7786
7787
7788
7789
7790
7791
7792
7793
7794
7795
7796
7797
7798
7799
```

```
7800 // Intel 8250 serial port (UART).
7801
7802 #include "types.h"
7803 #include "defs.h"
7804 #include "param.h"
7805 #include "traps.h"
7806 #include "spinlock.h"
7807 #include "fs.h"
7808 #include "file.h"
7809 #include "mmu.h"
7810 #include "proc.h"
7811 #include "x86.h"
7812
7813 #define COM1    0x3f8
7814
7815 static int uart;    // is there a uart?
7816
7817 void
7818 uartinit(void)
7819 {
7820   char *p;
7821
7822   // Turn off the FIFO
7823   outb(COM1+2, 0);
7824
7825   // 9600 baud, 8 data bits, 1 stop bit, parity off.
7826   outb(COM1+3, 0x80);    // Unlock divisor
7827   outb(COM1+0, 115200/9600);
7828   outb(COM1+1, 0);
7829   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
7830   outb(COM1+4, 0);
7831   outb(COM1+1, 0x01);    // Enable receive interrupts.
7832
7833   // If status is 0xFF, no serial port.
7834   if(inb(COM1+5) == 0xFF)
7835     return;
7836   uart = 1;
7837
7838   // Acknowledge pre-existing interrupt conditions;
7839   // enable interrupts.
7840   inb(COM1+2);
7841   inb(COM1+0);
7842   picenable(IRQ_COM1);
7843   ioapicenable(IRQ_COM1, 0);
7844
7845   // Announce that we're here.
7846   for(p="xv6...\n"; *p; p++)
7847     uartputc(*p);
7848 }
7849
```



```
7850 void
7851 uartputc(int c)
7852 {
7853   int i;
7854
7855   if(!uart)
7856     return;
7857   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
7858     microdelay(10);
7859   outb(COM1+0, c);
7860 }
7861
7862 static int
7863 uartgetc(void)
7864 {
7865   if(!uart)
7866     return -1;
7867   if(!(inb(COM1+5) & 0x01))
7868     return -1;
7869   return inb(COM1+0);
7870 }
7871
7872 void
7873 uartintr(void)
7874 {
7875   consoleintr(uartgetc);
7876 }
7877
7878
7879
7880
7881
7882
7883
7884
7885
7886
7887
7888
7889
7890
7891
7892
7893
7894
7895
7896
7897
7898
7899
```

```
# Initial process execs /init.

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


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

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

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

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



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

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

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(halt)
SYSCALL(date)
SYSCALL(getuid)
SYSCALL(getgid)
SYSCALL(getppid)
SYSCALL(setuid)
SYSCALL(setgid)
SYSCALL(getprocs)
```

```
8000 // init: The initial user-level program
8001
8002 #include "types.h"
8003 #include "stat.h"
8004 #include "user.h"
8005 #include "fcntl.h"
8006
8007 char *argv[] = { "sh", 0 };
8008
8009 int
8010 main(void)
8011 {
8012   int pid, wpid;
8013
8014   if(open("console", O_RDWR) < 0){
8015     mknod("console", 1, 1);
8016     open("console", O_RDWR);
8017   }
8018   dup(0);  // stdout
8019   dup(0);  // stderr
8020
8021   for(;;){
8022     printf(1, "init: starting sh\n");
8023     pid = fork();
8024     if(pid < 0){
8025       printf(1, "init: fork failed\n");
8026       exit();
8027     }
8028     if(pid == 0){
8029       exec("sh", argv);
8030       printf(1, "init: exec sh failed\n");
8031       exit();
8032     }
8033     while((wpid=wait()) >= 0 && wpid != pid)
8034       printf(1, "zombie!\n");
8035   }
8036 }
8037
8038
8039
8040
8041
8042
8043
8044
8045
8046
8047
8048
8049
```



```
8050 // Shell.
8051 // 2015-12-21. Added very simple processing for builtin commands
8052
8053 #include "types.h"
8054 #include "user.h"
8055 #include "fcntl.h"
8056
8057 // Parsed command representation
8058 #define EXEC  1
8059 #define REDIR 2
8060 #define PIPE  3
8061 #define LIST  4
8062 #define BACK  5
8063
8064 #define MAXARGS 10
8065
8066 struct cmd {
8067   int type;
8068 };
8069
8070 struct execcmd {
8071   int type;
8072   char *argv[MAXARGS];
8073   char *eargv[MAXARGS];
8074 };
8075
8076 struct redircmd {
8077   int type;
8078   struct cmd *cmd;
8079   char *file;
8080   char *efile;
8081   int mode;
8082   int fd;
8083 };
8084
8085 struct pipecmd {
8086   int type;
8087   struct cmd *left;
8088   struct cmd *right;
8089 };
8090
8091 struct listcmd {
8092   int type;
8093   struct cmd *left;
8094   struct cmd *right;
8095 };
8096
8097
8098
8099
```

```
8100 struct backcmd {
8101   int type;
8102   struct cmd *cmd;
8103 };
8104
8105 int fork1(void);  // Fork but panics on failure.
8106 void panic(char*);
8107 struct cmd *parsecmd(char*);
8108
8109 // Execute cmd.  Never returns.
8110 void
8111 runcmd(struct cmd *cmd)
8112 {
8113   int p[2];
8114   struct backcmd *bcmd;
8115   struct execcmd *ecmd;
8116   struct listcmd *lcmd;
8117   struct pipecmd *pcmd;
8118   struct redircmd *rcmd;
8119
8120   if(cmd == 0)
8121     exit();
8122
8123   switch(cmd->type){
8124   default:
8125     panic("runcmd");
8126
8127   case EXEC:
8128     ecmd = (struct execcmd*)cmd;
8129     if(ecmd->argv[0] == 0)
8130       exit();
8131     exec(ecmd->argv[0], ecmd->argv);
8132     printf(2, "exec %s failed\n", ecmd->argv[0]);
8133     break;
8134
8135   case REDIR:
8136     rcmd = (struct redircmd*)cmd;
8137     close(rcmd->fd);
8138     if(open(rcmd->file, rcmd->mode) < 0){
8139       printf(2, "open %s failed\n", rcmd->file);
8140       exit();
8141     }
8142     runcmd(rcmd->cmd);
8143     break;
8144
8145   case LIST:
8146     lcmd = (struct listcmd*)cmd;
8147     if(fork1() == 0)
8148       runcmd(lcmd->left);
8149     wait();
```



```
8150     runcmd(lcmd->right);
8151     break;
8152
8153   case PIPE:
8154     pcmd = (struct pipecmd*)cmd;
8155     if(pipe(p) < 0)
8156       panic("pipe");
8157     if(fork1() == 0){
8158       close(1);
8159       dup(p[1]);
8160       close(p[0]);
8161       close(p[1]);
8162       runcmd(pcmd->left);
8163     }
8164     if(fork1() == 0){
8165       close(0);
8166       dup(p[0]);
8167       close(p[0]);
8168       close(p[1]);
8169       runcmd(pcmd->right);
8170     }
8171     close(p[0]);
8172     close(p[1]);
8173     wait();
8174     wait();
8175     break;
8176
8177   case BACK:
8178     bcmd = (struct backcmd*)cmd;
8179     if(fork1() == 0)
8180       runcmd(bcmd->cmd);
8181     break;
8182   }
8183   exit();
8184 }
8185
8186 int
8187 getcmd(char *buf, int nbuf)
8188 {
8189   printf(2, "$ ");
8190   memset(buf, 0, nbuf);
8191   gets(buf, nbuf);
8192   if(buf[0] == 0) // EOF
8193     return -1;
8194   return 0;
8195 }
8196
8197
8198
8199
```

```
8200 #ifdef USE_BUILTINS
8201 // ***** processing for shell builtins begins here *****
8202
8203 int
8204 strncmp(const char *p, const char *q, uint n)
8205 {
8206     while(n > 0 && *p && *p == *q)
8207       n--, p++, q++;
8208     if(n == 0)
8209       return 0;
8210     return (uchar)*p - (uchar)*q;
8211 }
8212
8213 int
8214 makeint(char *p)
8215 {
8216   int val = 0;
8217
8218   while ((*p >= '0') && (*p <= '9')) {
8219     val = 10*val + (*p-'0');
8220     ++p;
8221   }
8222   return val;
8223 }
8224
8225 int
8226 setbuiltin(char *p)
8227 {
8228   int i;
8229
8230   p += strlen("_set");
8231   while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8232   if (strncmp("uid", p, 3) == 0) {
8233     p += strlen("uid");
8234     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8235     i = makeint(p); // ugly
8236     return (setuid(i));
8237   } else
8238   if (strncmp("gid", p, 3) == 0) {
8239     p += strlen("gid");
8240     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8241     i = makeint(p); // ugly
8242     return (setgid(i));
8243   }
8244   printf(2, "Invalid _set parameter\n");
8245   return -1;
8246 }
8247
8248
8249
```



```
8250 int
8251 getbuiltin(char *p)
8252 {
8253   p += strlen("_get");
8254   while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8255   if (strncmp("uid", p, 3) == 0) {
8256     printf(2, "%d\n", getuid());
8257     return 0;
8258   }
8259   if (strncmp("gid", p, 3) == 0) {
8260     printf(2, "%d\n", getgid());
8261     return 0;
8262   }
8263   printf(2, "Invalid _get parameter\n");
8264   return -1;
8265 }
8266
8267 typedef int funcPtr_t(char *);
8268 typedef struct {
8269   char       *cmd;
8270   funcPtr_t  *name;
8271 } dispatchTableEntry_t;
8272
8273 // Use a simple function dispatch table (FDT) to process builtin commands
8274 dispatchTableEntry_t fdt[] = {
8275   {"_set", setbuiltin},
8276   {"_get", getbuiltin}
8277 };
8278 int FDTcount = sizeof(fdt) / sizeof(fdt[0]); // # entris in FDT
8279
8280 void
8281 dobuiltin(char *cmd) {
8282   int i;
8283
8284   for (i=0; i<FDTcount; i++)
8285     if (strncmp(cmd, fdt[i].cmd, strlen(fdt[i].cmd)) == 0)
8286       (*fdt[i].name)(cmd);
8287 }
8288
8289
8290
8291
8292
8293
8294
8295
8296
8297
8298
8299
```

```
8300 // ***** processing for shell builtins ends here *****
8301 #endif
8302
8303 int
8304 main(void)
8305 {
8306   static char buf[100];
8307   int fd;
8308
8309   // Assumes three file descriptors open.
8310   while((fd = open("console", O_RDWR)) >= 0){
8311     if(fd >= 3){
8312       close(fd);
8313       break;
8314     }
8315   }
8316
8317   // Read and run input commands.
8318   while(getcmd(buf, sizeof(buf)) >= 0){
8319 // add support for built-ins here. cd is a built-in
8320     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
8321       // Clumsy but will have to do for now.
8322       // Chdir has no effect on the parent if run in the child.
8323       buf[strlen(buf)-1] = 0;  // chop \n
8324       if(chdir(buf+3) < 0)
8325         printf(2, "cannot cd %s\n", buf+3);
8326       continue;
8327     }
8328 #ifdef USE_BUILTINS
8329     if (buf[0]=='_') {     // assume it is a builtin command
8330       dobuiltin(buf);
8331       continue;
8332     }
8333 #endif
8334     if(fork1() == 0)
8335       runcmd(parsecmd(buf));
8336     wait();
8337   }
8338   exit();
8339 }
8340
8341 void
8342 panic(char *s)
8343 {
8344   printf(2, "%s\n", s);
8345   exit();
8346 }
8347
8348
8349
```



```
8350 int
8351 fork1(void)
8352 {
8353   int pid;
8354
8355   pid = fork();
8356   if(pid == -1)
8357     panic("fork");
8358   return pid;
8359 }
8360
8361 // Constructors
8362
8363 struct cmd*
8364 execcmd(void)
8365 {
8366   struct execcmd *cmd;
8367
8368   cmd = malloc(sizeof(*cmd));
8369   memset(cmd, 0, sizeof(*cmd));
8370   cmd->type = EXEC;
8371   return (struct cmd*)cmd;
8372 }
8373
8374 struct cmd*
8375 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
8376 {
8377   struct redircmd *cmd;
8378
8379   cmd = malloc(sizeof(*cmd));
8380   memset(cmd, 0, sizeof(*cmd));
8381   cmd->type = REDIR;
8382   cmd->cmd = subcmd;
8383   cmd->file = file;
8384   cmd->efile = efile;
8385   cmd->mode = mode;
8386   cmd->fd = fd;
8387   return (struct cmd*)cmd;
8388 }
8389
8390
8391
8392
8393
8394
8395
8396
8397
8398
8399
```

```
8400 struct cmd*
8401 pipecmd(struct cmd *left, struct cmd *right)
8402 {
8403   struct pipecmd *cmd;
8404
8405   cmd = malloc(sizeof(*cmd));
8406   memset(cmd, 0, sizeof(*cmd));
8407   cmd->type = PIPE;
8408   cmd->left = left;
8409   cmd->right = right;
8410   return (struct cmd*)cmd;
8411 }
8412
8413 struct cmd*
8414 listcmd(struct cmd *left, struct cmd *right)
8415 {
8416   struct listcmd *cmd;
8417
8418   cmd = malloc(sizeof(*cmd));
8419   memset(cmd, 0, sizeof(*cmd));
8420   cmd->type = LIST;
8421   cmd->left = left;
8422   cmd->right = right;
8423   return (struct cmd*)cmd;
8424 }
8425
8426 struct cmd*
8427 backcmd(struct cmd *subcmd)
8428 {
8429   struct backcmd *cmd;
8430
8431   cmd = malloc(sizeof(*cmd));
8432   memset(cmd, 0, sizeof(*cmd));
8433   cmd->type = BACK;
8434   cmd->cmd = subcmd;
8435   return (struct cmd*)cmd;
8436 }
8437
8438
8439
8440
8441
8442
8443
8444
8445
8446
8447
8448
8449
```



```
8450 // Parsing
8451
8452 char whitespace[] = " \t\r\n\v";
8453 char symbols[] = "<|>&;()";
8454
8455 int
8456 gettoken(char **ps, char *es, char **q, char **eq)
8457 {
8458   char *s;
8459   int ret;
8460
8461   s = *ps;
8462   while(s < es && strchr(whitespace, *s))
8463     s++;
8464   if(q)
8465     *q = s;
8466   ret = *s;
8467   switch(*s){
8468   case 0:
8469     break;
8470   case '|':
8471   case '(':
8472   case ')':
8473   case ';':
8474   case '&':
8475   case '<':
8476     s++;
8477     break;
8478   case '>':
8479     s++;
8480     if(*s == '>'){
8481       ret = '+';
8482       s++;
8483     }
8484     break;
8485   default:
8486     ret = 'a';
8487     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
8488       s++;
8489     break;
8490   }
8491   if(eq)
8492     *eq = s;
8493
8494   while(s < es && strchr(whitespace, *s))
8495     s++;
8496   *ps = s;
8497   return ret;
8498 }
8499
```

```
8500 int
8501 peek(char **ps, char *es, char *toks)
8502 {
8503   char *s;
8504
8505   s = *ps;
8506   while(s < es && strchr(whitespace, *s))
8507     s++;
8508   *ps = s;
8509   return *s && strchr(toks, *s);
8510 }
8511
8512 struct cmd *parseline(char**, char*);
8513 struct cmd *parsepipe(char**, char*);
8514 struct cmd *parseexec(char**, char*);
8515 struct cmd *nulterminate(struct cmd*);
8516
8517 struct cmd*
8518 parsecmd(char *s)
8519 {
8520   char *es;
8521   struct cmd *cmd;
8522
8523   es = s + strlen(s);
8524   cmd = parseline(&s, es);
8525   peek(&s, es, "");
8526   if(s != es){
8527     printf(2, "leftovers: %s\n", s);
8528     panic("syntax");
8529   }
8530   nulterminate(cmd);
8531   return cmd;
8532 }
8533
8534 struct cmd*
8535 parseline(char **ps, char *es)
8536 {
8537   struct cmd *cmd;
8538
8539   cmd = parsepipe(ps, es);
8540   while(peek(ps, es, "&")){
8541     gettoken(ps, es, 0, 0);
8542     cmd = backcmd(cmd);
8543   }
8544   if(peek(ps, es, ";")){
8545     gettoken(ps, es, 0, 0);
8546     cmd = listcmd(cmd, parseline(ps, es));
8547   }
8548   return cmd;
8549 }
```



```
8550 struct cmd*
8551 parsepipe(char **ps, char *es)
8552 {
8553   struct cmd *cmd;
8554
8555   cmd = parseexec(ps, es);
8556   if(peek(ps, es, "|")){
8557     gettoken(ps, es, 0, 0);
8558     cmd = pipecmd(cmd, parsepipe(ps, es));
8559   }
8560   return cmd;
8561 }
8562
8563 struct cmd*
8564 parseredirs(struct cmd *cmd, char **ps, char *es)
8565 {
8566   int tok;
8567   char *q, *eq;
8568
8569   while(peek(ps, es, "<>")){
8570     tok = gettoken(ps, es, 0, 0);
8571     if(gettoken(ps, es, &q, &eq) != 'a')
8572       panic("missing file for redirection");
8573     switch(tok){
8574     case '<':
8575       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
8576       break;
8577     case '>':
8578       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8579       break;
8580     case '+':  // >>
8581       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8582       break;
8583     }
8584   }
8585   return cmd;
8586 }
8587
8588
8589
8590
8591
8592
8593
8594
8595
8596
8597
8598
8599
```

```
8600 struct cmd*
8601 parseblock(char **ps, char *es)
8602 {
8603   struct cmd *cmd;
8604
8605   if(!peek(ps, es, "("))
8606     panic("parseblock");
8607   gettoken(ps, es, 0, 0);
8608   cmd = parseline(ps, es);
8609   if(!peek(ps, es, ")"))
8610     panic("syntax - missing )");
8611   gettoken(ps, es, 0, 0);
8612   cmd = parseredirs(cmd, ps, es);
8613   return cmd;
8614 }
8615
8616 struct cmd*
8617 parseexec(char **ps, char *es)
8618 {
8619   char *q, *eq;
8620   int tok, argc;
8621   struct execcmd *cmd;
8622   struct cmd *ret;
8623
8624   if(peek(ps, es, "("))
8625     return parseblock(ps, es);
8626
8627   ret = execcmd();
8628   cmd = (struct execcmd*)ret;
8629
8630   argc = 0;
8631   ret = parseredirs(ret, ps, es);
8632   while(!peek(ps, es, "|)&;")){
8633     if((tok=gettoken(ps, es, &q, &eq)) == 0)
8634       break;
8635     if(tok != 'a')
8636       panic("syntax");
8637     cmd->argv[argc] = q;
8638     cmd->eargv[argc] = eq;
8639     argc++;
8640     if(argc >= MAXARGS)
8641       panic("too many args");
8642     ret = parseredirs(ret, ps, es);
8643   }
8644   cmd->argv[argc] = 0;
8645   cmd->eargv[argc] = 0;
8646   return ret;
8647 }
8648
8649
```



```
8650 // NUL-terminate all the counted strings.
8651 struct cmd*
8652 nulterminate(struct cmd *cmd)
8653 {
8654   int i;
8655   struct backcmd *bcmd;
8656   struct execcmd *ecmd;
8657   struct listcmd *lcmd;
8658   struct pipecmd *pcmd;
8659   struct redircmd *rcmd;
8660
8661   if(cmd == 0)
8662     return 0;
8663
8664   switch(cmd->type){
8665   case EXEC:
8666     ecmd = (struct execcmd*)cmd;
8667     for(i=0; ecmd->argv[i]; i++)
8668       *ecmd->eargv[i] = 0;
8669     break;
8670
8671   case REDIR:
8672     rcmd = (struct redircmd*)cmd;
8673     nulterminate(rcmd->cmd);
8674     *rcmd->efile = 0;
8675     break;
8676
8677   case PIPE:
8678     pcmd = (struct pipecmd*)cmd;
8679     nulterminate(pcmd->left);
8680     nulterminate(pcmd->right);
8681     break;
8682
8683   case LIST:
8684     lcmd = (struct listcmd*)cmd;
8685     nulterminate(lcmd->left);
8686     nulterminate(lcmd->right);
8687     break;
8688
8689   case BACK:
8690     bcmd = (struct backcmd*)cmd;
8691     nulterminate(bcmd->cmd);
8692     break;
8693   }
8694   return cmd;
8695 }
8696
8697
8698
8699
```

```
8700 #include "asm.h"
8701 #include "memlayout.h"
8702 #include "mmu.h"
8703
8704 # Start the first CPU: switch to 32-bit protected mode, jump into C.
8705 # The BIOS loads this code from the first sector of the hard disk into
8706 # memory at physical address 0x7c00 and starts executing in real mode
8707 # with %cs=0 %ip=7c00.
8708
8709 .code16                       # Assemble for 16-bit mode
8710 .globl start
8711 start:
8712   cli                         # BIOS enabled interrupts; disable
8713
8714   # Zero data segment registers DS, ES, and SS.
8715   xorw    %ax,%ax             # Set %ax to zero
8716   movw    %ax,%ds             # -> Data Segment
8717   movw    %ax,%es             # -> Extra Segment
8718   movw    %ax,%ss             # -> Stack Segment
8719
8720   # Physical address line A20 is tied to zero so that the first PCs
8721   # with 2 MB would run software that assumed 1 MB.  Undo that.
8722 seta20.1:
8723   inb     $0x64,%al               # Wait for not busy
8724   testb   $0x2,%al
8725   jnz     seta20.1
8726
8727   movb    $0xd1,%al               # 0xd1 -> port 0x64
8728   outb    %al,$0x64
8729
8730 seta20.2:
8731   inb     $0x64,%al               # Wait for not busy
8732   testb   $0x2,%al
8733   jnz     seta20.2
8734
8735   movb    $0xdf,%al               # 0xdf -> port 0x60
8736   outb    %al,$0x60
8737
8738   # Switch from real to protected mode.  Use a bootstrap GDT that makes
8739   # virtual addresses map directly to physical addresses so that the
8740   # effective memory map doesn't change during the transition.
8741   lgdt    gdtdesc
8742   movl    %cr0, %eax
8743   orl     $CR0_PE, %eax
8744   movl    %eax, %cr0
8745
8746   # Complete transition to 32-bit protected mode by using long jmp
8747   # to reload %cs and %eip.  The segment descriptors are set up with no
8748   # translation, so that the mapping is still the identity mapping.
8749   ljmp    $(SEG_KCODE<<3), $start32
```



```
8750 .code32  # Tell assembler to generate 32-bit code now.
8751 start32:
8752   # Set up the protected-mode data segment registers
8753   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
8754   movw    %ax, %ds                # -> DS: Data Segment
8755   movw    %ax, %es                # -> ES: Extra Segment
8756   movw    %ax, %ss                # -> SS: Stack Segment
8757   movw    $0, %ax                 # Zero segments not ready for use
8758   movw    %ax, %fs                # -> FS
8759   movw    %ax, %gs                # -> GS
8760
8761   # Set up the stack pointer and call into C.
8762   movl    $start, %esp
8763   call    bootmain
8764
8765   # If bootmain returns (it shouldn't), trigger a Bochs
8766   # breakpoint if running under Bochs, then loop.
8767   movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
8768   movw    %ax, %dx
8769   outw    %ax, %dx
8770   movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
8771   outw    %ax, %dx
8772 spin:
8773   jmp     spin
8774
8775 # Bootstrap GDT
8776 .p2align 2                                # force 4 byte alignment
8777 gdt:
8778   SEG_NULLASM                             # null seg
8779   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
8780   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
8781
8782 gdtdesc:
8783   .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
8784   .long   gdt                             # address gdt
8785
8786
8787
8788
8789
8790
8791
8792
8793
8794
8795
8796
8797
8798
8799
```

```
8800 // Boot loader.
8801 //
8802 // Part of the boot block, along with bootasm.S, which calls bootmain().
8803 // bootasm.S has put the processor into protected 32-bit mode.
8804 // bootmain() loads an ELF kernel image from the disk starting at
8805 // sector 1 and then jumps to the kernel entry routine.
8806
8807 #include "types.h"
8808 #include "elf.h"
8809 #include "x86.h"
8810 #include "memlayout.h"
8811
8812 #define SECTSIZE  512
8813
8814 void readseg(uchar*, uint, uint);
8815
8816 void
8817 bootmain(void)
8818 {
8819   struct elfhdr *elf;
8820   struct proghdr *ph, *eph;
8821   void (*entry)(void);
8822   uchar* pa;
8823
8824   elf = (struct elfhdr*)0x10000;  // scratch space
8825
8826   // Read 1st page off disk
8827   readseg((uchar*)elf, 4096, 0);
8828
8829   // Is this an ELF executable?
8830   if(elf->magic != ELF_MAGIC)
8831     return;  // let bootasm.S handle error
8832
8833   // Load each program segment (ignores ph flags).
8834   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
8835   eph = ph + elf->phnum;
8836   for(; ph < eph; ph++){
8837     pa = (uchar*)ph->paddr;
8838     readseg(pa, ph->filesz, ph->off);
8839     if(ph->memsz > ph->filesz)
8840       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
8841   }
8842
8843   // Call the entry point from the ELF header.
8844   // Does not return!
8845   entry = (void(*)(void))(elf->entry);
8846   entry();
8847 }
8848
8849
```



```
8850 void
8851 waitdisk(void)
8852 {
8853   // Wait for disk ready.
8854   while((inb(0x1F7) & 0xC0) != 0x40)
8855     ;
8856 }
8857
8858 // Read a single sector at offset into dst.
8859 void
8860 readsect(void *dst, uint offset)
8861 {
8862   // Issue command.
8863   waitdisk();
8864   outb(0x1F2, 1);   // count = 1
8865   outb(0x1F3, offset);
8866   outb(0x1F4, offset >> 8);
8867   outb(0x1F5, offset >> 16);
8868   outb(0x1F6, (offset >> 24) | 0xE0);
8869   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
8870
8871   // Read data.
8872   waitdisk();
8873   insl(0x1F0, dst, SECTSIZE/4);
8874 }
8875
8876 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
8877 // Might copy more than asked.
8878 void
8879 readseg(uchar* pa, uint count, uint offset)
8880 {
8881   uchar* epa;
8882
8883   epa = pa + count;
8884
8885   // Round down to sector boundary.
8886   pa -= offset % SECTSIZE;
8887
8888   // Translate from bytes to sectors; kernel starts at sector 1.
8889   offset = (offset / SECTSIZE) + 1;
8890
8891   // If this is too slow, we could read lots of sectors at a time.
8892   // We'd write more to memory than asked, but it doesn't matter --
8893   // we load in increasing order.
8894   for(; pa < epa; pa += SECTSIZE, offset++)
8895     readsect(pa, offset);
8896 }
8897
8898
8899
```

```
8900 #ifdef CS333_P4
8901 // this is an ugly series of if statements but it works
8902 void
8903 print_mode(struct stat* st)
8904 {
8905   switch (st->type) {
8906     case T_DIR: printf(1, "d"); break;
8907     case T_FILE: printf(1, "-"); break;
8908     case T_DEV: printf(1, "c"); break;
8909     default: printf(1, "?");
8910   }
8911
8912   if (st->mode.flags.u_r)
8913     printf(1, "r");
8914   else
8915     printf(1, "-");
8916
8917   if (st->mode.flags.u_w)
8918     printf(1, "w");
8919   else
8920     printf(1, "-");
8921
8922   if ((st->mode.flags.u_x) & (st->mode.flags.setuid))
8923     printf(1, "S");
8924   else if (st->mode.flags.u_x)
8925     printf(1, "x");
8926   else
8927     printf(1, "-");
8928
8929   if (st->mode.flags.g_r)
8930     printf(1, "r");
8931   else
8932     printf(1, "-");
8933
8934   if (st->mode.flags.g_w)
8935     printf(1, "w");
8936   else
8937     printf(1, "-");
8938
8939   if (st->mode.flags.g_x)
8940     printf(1, "x");
8941   else
8942     printf(1, "-");
8943
8944   if (st->mode.flags.o_r)
8945     printf(1, "r");
8946   else
8947     printf(1, "-");
8948
8949
```



```
8950   if (st->mode.flags.o_w)
8951     printf(1, "w");
8952   else
8953     printf(1, "-");
8954
8955   if (st->mode.flags.o_x)
8956     printf(1, "x");
8957   else
8958     printf(1, "-");
8959
8960   return;
8961 }
8962 #endif
8963
8964
8965
8966
8967
8968
8969
8970
8971
8972
8973
8974
8975
8976
8977
8978
8979
8980
8981
8982
8983
8984
8985
8986
8987
8988
8989
8990
8991
8992
8993
8994
8995
8996
8997
8998
8999
```

```
#include "types.h"
#include "user.h"
#include "date.h"


int
main(int argc, char *argv[])
{
  struct rtcdate r;
  if(date(&r)) {
    printf(2,"date failed\n");
    exit();
  }
  printf(1, "Current UTC time is: %d/%d/%d - %d:%d:%d\n",r.year, r.month, r.
  exit();
}
```



```
#define STRMAX 32

struct uproc {
   uint pid;
   uint uid;
   uint gid;
   uint ppid;
   uint elapsed_ticks;
   uint CPU_total_ticks;
   char state[STRMAX];
   uint size;
   char name[STRMAX];
};
```

```
9100 #include "types.h"
9101 #include "user.h"
9102
9103 // Test GID and UID to be in the correct range
9104 #ifdef CS333_P2
9105 int
9106 testgiduid(void)
9107 {
9108    uint uid, gid, ppid;
9109
9110    uid = getuid();
9111    printf(2, "Current UID is : %d\n", uid);
9112    printf(2, "Setting UID to 100\n");
9113    setuid(100);
9114    uid = getuid();
9115    printf(2, "Current UID is : %d\n", uid);
9116
9117    gid = getgid();
9118    printf(2, "Current GID is : %d\n", gid);
9119    printf(2, "Setting GID to 100\n");
9120    setgid(100);
9121    gid = getgid();
9122    printf(2, "Current UID is : %d\n", gid);
9123
9124    ppid = getppid();
9125    printf(2, "My parent process is : %d\n", ppid);
9126    printf(2, "Done!\n");
9127
9128    return 0;
9129 }
9130
9131 int
9132 main(int argc, char *argv[])
9133 {
9134    testgiduid();
9135    exit();
9136 }
9137 #else
9138 int
9139 main(int argc, char *argv[])
9140 {
9141    printf(2, "Please compile with CS333_P2 on to enable this feature.\n");
9142    exit();
9143 }
9144 #endif
9145
9146
9147
9148
9149
```



```
9150 #include "types.h"
9151 #include "uproc.h"
9152 #include "user.h"
9153
9154 #ifdef CS333_P2
9155 int
9156 main(int argc, char *argv[])
9157 {
9158    int ptable_size;
9159    uint display_size;
9160    display_size = 64;
9161    struct uproc* ps;
9162    ps = malloc(sizeof(struct uproc) * display_size);
9163    ptable_size = getprocs(display_size, ps);
9164    if(ptable_size <= 0) {
9165            printf(1,"\nGetting processes information failed\n");
9166            exit();
9167    }
9168    printf(1,"\nNumber of processes is :%d\n",ptable_size);
9169    printf(1,"\nPID       State     Name      UID       GID       PPID      I
9170    int i;
9171    for(i=0; i < ptable_size; ++i){
9172     printf(1,"\n%d         %s    %s    %d    %d    %d    %d.%d    %d.%d    %(
9173            ps->state,\
9174            ps->name,\
9175            ps->uid,\
9176            ps->gid,\
9177            ps->ppid, ps->elapsed_ticks/100, ps->elapsed_ticks%100, ps->CPU_t(
9178            ++ps;
9179    }
9180   exit();
9181 }
9182 #else
9183 int
9184 main(int argc, char *argv[])
9185 {
9186    printf(2, "Please compile with CS333_P2 on to enable this feature.\n");
9187    exit();
9188 }
9189 #endif
9190
9191
9192
9193
9194
9195
9196
9197
9198
9199
```

```
9200 #include "types.h"
9201 #include "user.h"
9202
9203 #ifdef CS333_P2
9204 int
9205 main(int argc, char *argv[])
9206 {
9207    int elapsed_t = 0;
9208    int pid;
9209    int start_t = uptime();
9210    int end_t = start_t;
9211    if(argc > 1) {
9212            pid = fork();
9213            if(pid > 0) {
9214                    pid = wait();
9215                    end_t= uptime();
9216                    }
9217            else if(pid == 0) {
9218                    //child process running
9219                    char **nargv = ++argv;
9220                    exec(argv[0], nargv);
9221                    exit();
9222                    }
9223            else {
9224                    // error
9225                    exit();
9226                    }
9227            }
9228    elapsed_t = end_t - start_t;
9229    char *proc_name = argv[1] ? argv[1] : "";
9230   printf(1,"%s ran in %d.%d seconds\n",proc_name, elapsed_t/100, elapsed_t%1
9231
9232   exit();
9233 }
9234 #else
9235 int
9236 main(int argc, char *argv[])
9237 {
9238    printf(2, "Please compile with CS333_P2 on to enable this feature.\n");
9239    exit();
9240 }
9241 #endif
9242
9243
9244
9245
9246
9247
9248
9249
```