xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix
Version 6 (v6). xv6 loosely follows the structure and style of v6,
but is implemented for a modern x86-based multiprocessor using ANSI C.

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14,
2000)). See also http://pdos.csail.mit.edu/6.828/2014/xv6.html, which
provides pointers to on-line resources for v6.

xv6 borrows code from the following sources:
    JOS (asm.h, elf.h, mmu.h, bootasm.S, ide.c, console.c, and others)
    Plan 9 (entryother.S, mp.h, mp.c, lapic.c)
    FreeBSD (ioapic.c)
    NetBSD (console.c)

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

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

ERROR REPORTS

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

BUILDING AND RUNNING XV6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run "make".
On non-x86 or non-ELF machines (like OS X, even on x86), you will
need to install a cross-compiler gcc suite capable of producing x86 ELF
binaries. See http://pdos.csail.mit.edu/6.828/2014/tools.html.
Then run "make TOOLPREFIX=i386-jos-elf-".

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

To create a typeset version of the code, run "make xv6.pdf". This
requires the "mpage" utility. See http://www.mesa.nl/pub/mpage/.

The numbers to the left of the file names in the table are sheet numbers.
The source code has been printed in a double column format with fifty
lines per column, giving one hundred lines per sheet (or page).
Thus there is a convenient relationship between line numbers and sheet numbers.

The source listing is preceded by a cross-reference that lists every defined
constant, struct, global variable, and function in xv6. Each entry gives,
on the same line as the name, the line number (or, in a few cases, numbers)
where the name is defined. Successive lines in an entry list the line
numbers where the name is used. For example, this entry:

    swtch 2658
        0374 2428 2466 2657 2658

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

```
0100 typedef unsigned int   uint;
0101 typedef unsigned short ushort;
0102 typedef unsigned char  uchar;
0103 typedef uint pde_t;
0104
0105
0106
0107
0108
0109
0110
0111
0112
0113
0114
0115
0116
0117
0118
0119
0120
0121
0122
0123
0124
0125
0126
0127
0128
0129
0130
0131
0132
0133
0134
0135
0136
0137
0138
0139
0140
0141
0142
0143
0144
0145
0146
0147
0148
0149
```



```
0150 #define NPROC        64  // maximum number of processes
0151 #define KSTACKSIZE 4096  // size of per-process kernel stack
0152 #define NCPU          8  // maximum number of CPUs
0153 #define NOFILE       16  // open files per process
0154 #define NFILE       100  // open files per system
0155 #define NINODE       50  // maximum number of active i-nodes
0156 #define NDEV         10  // maximum major device number
0157 #define ROOTDEV       1  // device number of file system root disk
0158 #define MAXARG       32  // max exec arguments
0159 #define MAXOPBLOCKS  10  // max # of blocks any FS op writes
0160 #define LOGSIZE      (MAXOPBLOCKS*3)  // max data blocks in on-disk log
0161 #define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
0162 #define FSSIZE       1000  // size of file system in blocks
0163
0164
0165
0166
0167
0168
0169
0170
0171
0172
0173
0174
0175
0176
0177
0178
0179
0180
0181
0182
0183
0184
0185
0186
0187
0188
0189
0190
0191
0192
0193
0194
0195
0196
0197
0198
0199
```

```
0200 // Memory layout
0201
0202 #define EXTMEM  0x100000            // Start of extended memory
0203 #define PHYSTOP 0xE000000           // Top physical memory
0204 #define DEVSPACE 0xFE000000         // Other devices are at high addresses
0205
0206 // Key addresses for address space layout (see kmap in vm.c for layout)
0207 #define KERNBASE 0x80000000         // First kernel virtual address
0208 #define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked
0209
0210 #ifndef __ASSEMBLER__
0211
0212 static inline uint v2p(void *a) { return ((uint) (a))  - KERNBASE; }
0213 static inline void *p2v(uint a) { return (void *) ((a) + KERNBASE); }
0214
0215 #endif
0216
0217 #define V2P(a) (((uint) (a)) - KERNBASE)
0218 #define P2V(a) (((void *) (a)) + KERNBASE)
0219
0220 #define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
0221 #define P2V_WO(x) ((x) + KERNBASE)    // same as P2V, but without casts
0222
0223
0224
0225
0226
0227
0228
0229
0230
0231
0232
0233
0234
0235
0236
0237
0238
0239
0240
0241
0242
0243
0244
0245
0246
0247
0248
0249
```



```
0250 struct rtcdate {
0251   uint second;
0252   uint minute;
0253   uint hour;
0254   uint day;
0255   uint month;
0256   uint year;
0257 };
0258
0259
0260
0261
0262
0263
0264
0265
0266
0267
0268
0269
0270
0271
0272
0273
0274
0275
0276
0277
0278
0279
0280
0281
0282
0283
0284
0285
0286
0287
0288
0289
0290
0291
0292
0293
0294
0295
0296
0297
0298
0299
```

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



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

```
0400 // pipe.c
0401 int             pipealloc(struct file**, struct file**);
0402 void            pipeclose(struct pipe*, int);
0403 int             piperead(struct pipe*, char*, int);
0404 int             pipewrite(struct pipe*, char*, int);
0405
0406 // 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
0422 // swtch.S
0423 void            swtch(struct context**, struct context*);
0424
0425 // spinlock.c
0426 void            acquire(struct spinlock*);
0427 void            getcallerpcs(void*, uint*);
0428 int             holding(struct spinlock*);
0429 void            initlock(struct spinlock*, char*);
0430 void            release(struct spinlock*);
0431 void            pushcli(void);
0432 void            popcli(void);
0433
0434 // string.c
0435 int             memcmp(const void*, const void*, uint);
0436 void*           memmove(void*, const void*, uint);
0437 void*           memset(void*, int, uint);
0438 char*           safestrcpy(char*, const char*, int);
0439 int             strlen(const char*);
0440 int             strncmp(const char*, const char*, uint);
0441 char*           strncpy(char*, const char*, int);
0442
0443 // syscall.c
0444 int             argint(int, int*);
0445 int             argptr(int, char**, int);
0446 int             argstr(int, char**);
0447 int             fetchint(uint, int*);
0448 int             fetchstr(uint, char**);
0449 void            syscall(void);
```



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

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



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

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



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

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



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

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



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

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



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

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



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

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



```
1150 #include "asm.h"
1151 #include "memlayout.h"
1152 #include "mmu.h"
1153
1154 # Each non-boot CPU ("AP") is started up in response to a STARTUP
1155 # IPI from the boot CPU.  Section B.4.2 of the Multi-Processor
1156 # Specification says that the AP will start in real mode with CS:IP
1157 # set to XY00:0000, where XY is an 8-bit value sent with the
1158 # STARTUP. Thus this code must start at a 4096-byte boundary.
1159 #
1160 # Because this code sets DS to zero, it must sit
1161 # at an address in the low 2^16 bytes.
1162 #
1163 # Startothers (in main.c) sends the STARTUPs one at a time.
1164 # It copies this code (start) at 0x7000.  It puts the address of
1165 # a newly allocated per-core stack in start-4,the address of the
1166 # place to jump to (mpenter) in start-8, and the physical address
1167 # of entrypgdir in start-12.
1168 #
1169 # This code is identical to bootasm.S except:
1170 #   - it does not need to enable A20
1171 #   - it uses the address at start-4, start-8, and start-12
1172
1173 .code16
1174 .globl start
1175 start:
1176   cli
1177
1178   xorw    %ax,%ax
1179   movw    %ax,%ds
1180   movw    %ax,%es
1181   movw    %ax,%ss
1182
1183   lgdt    gdtdesc
1184   movl    %cr0, %eax
1185   orl     $CR0_PE, %eax
1186   movl    %eax, %cr0
1187
1188   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 #define INITUID 0
2053 #define INITGID 0
2054 #define NUM_READY_LISTS 5
2055 #define TICKS_TO_PROMOTE 100
2056 #define INITBUDGET 50
2057
2058 // Per-CPU state
2059 struct cpu {
2060   uchar id;                    // Local APIC ID; index into cpus[] below
2061   struct context *scheduler;   // swtch() here to enter scheduler
2062   struct taskstate ts;         // Used by x86 to find stack for interrupt
2063   struct segdesc gdt[NSEGS];   // x86 global descriptor table
2064   volatile uint started;       // Has the CPU started?
2065   int ncli;                    // Depth of pushcli nesting.
2066   int intena;                  // Were interrupts enabled before pushcli?
2067
2068   // Cpu-local storage variables; see below
2069   struct cpu *cpu;
2070   struct proc *proc;           // The currently-running process.
2071 };
2072
2073 extern struct cpu cpus[NCPU];
2074 extern int ncpu;
2075
2076 // Per-CPU variables, holding pointers to the
2077 // current cpu and to the current process.
2078 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2079 // and "%gs:4" to refer to proc.  seginit sets up the
2080 // %gs segment register so that %gs refers to the memory
2081 // holding those two variables in the local cpu's struct cpu.
2082 // This is similar to how thread-local variables are implemented
2083 // in thread libraries such as Linux pthreads.
2084 extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
2085 extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc
2086
2087 // Saved registers for kernel context switches.
2088 // Don't need to save all the segment registers (%cs, etc),
2089 // because they are constant across kernel contexts.
2090 // Don't need to save %eax, %ecx, %edx, because the
2091 // x86 convention is that the caller has saved them.
2092 // Contexts are stored at the bottom of the stack they
2093 // describe; the stack pointer is the address of the context.
2094 // The layout of the context matches the layout of the stack in swtch.S
2095 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2096 // but it is on the stack and allocproc() manipulates it.
2097 struct context {
2098   uint edi;
2099   uint esi;
```

```
2100   uint ebx;
2101   uint ebp;
2102   uint eip;
2103 };
2104
2105 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2106
2107 // Per-process state
2108 struct proc {
2109   uint sz;                     // Size of process memory (bytes)
2110   pde_t* pgdir;                // Page table
2111   char *kstack;                // Bottom of kernel stack for this process
2112   enum procstate state;        // Process state
2113   uint pid;                    // Process ID
2114   struct proc *parent;         // Parent process
2115   struct trapframe *tf;        // Trap frame for current syscall
2116   struct context *context;     // swtch() here to run process
2117   void *chan;                  // If non-zero, sleeping on chan
2118   int killed;                  // If non-zero, have been killed
2119   struct file *ofile[NOFILE];  // Open files
2120   struct inode *cwd;           // Current directory
2121   char name[16];               // Process name (debugging)
2122
2123   //STUDENT IMPLEMENTATION
2124   uint start_ticks;            // Start time
2125   uint uid;                    // User ID Number
2126   uint gid;                    // Group ID Number
2127   uint cpu_ticks_total;        // Amount of ticks
2128   uint cpu_ticks_in;           // Total ticks spent in cpu
2129   uint priority;               // The priority of this process
2130   struct proc* next;           // Points to next process in the same list
2131   int budget;                 // Ticks Before Aging
2132
2133 };
2134
2135 // Process memory is laid out contiguously, low addresses first:
2136 //   text
2137 //   original data and bss
2138 //   fixed-size stack
2139 //   expandable heap
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 "asm.h"
2159 #include "uproc.h"
2160 #include "queue.h"
2161
2162 struct {
2163   struct spinlock lock;
2164   struct proc proc[NPROC];
2165   struct queue ReadyList[NUM_READY_LISTS];  // Array of queue for MLFQ
2166   struct queue FreeList;                    // Queue for UNUSED processes
2167   uint PromoteAtTime;
2168
2169 } ptable;
2170
2171 static struct proc *initproc;
2172
2173 int nextpid = 1;
2174 extern void forkret(void);
2175 extern void trapret(void);
2176 // Project 3
2177 // Helper Functions to manipulate the queues
2178
2179 extern int      queue (struct queue* this, struct proc* first);
2180 extern int     enqueue (struct queue* this, struct proc* nproc);
2181 extern  struct proc* dequeue (struct queue* this);
2182 extern int     updateBudget(struct proc* this, int now);
2183 extern void    promote();
2184
2185 static void wakeup1(void *chan);
2186
2187 void
2188 pinit(void)
2189 {
2190   initlock(&ptable.lock, "ptable");
2191 }
2192
2193
2194
2195
2196
2197
2198
2199
```

```
2200 // Look in the process table for an UNUSED proc.
2201 // If found, change state to EMBRYO and initialize
2202 // state required to run in the kernel.
2203 // Otherwise return 0.
2204 static struct proc*
2205 allocproc(void)
2206 {
2207   struct proc *p;
2208   char *sp;
2209
2210   acquire(&ptable.lock);
2211   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2212     if(p->state == UNUSED)
2213       goto found;
2214   release(&ptable.lock);
2215   return 0;
2216
2217   // Project 3 Pull process from Free List
2218   acquire(&ptable.lock);
2219   p = 0;                                  // Must be initialized, so assume th
2220   p = dequeue(&ptable.FreeList);
2221
2222   if(p != 0)
2223       goto found;
2224   release(&ptable.lock);
2225   return 0;
2226
2227 found:
2228   p->state = EMBRYO;
2229   p->pid = nextpid++;
2230   release(&ptable.lock);
2231
2232   // Allocate kernel stack.
2233   if((p->kstack = kalloc()) == 0){
2234     p->state = UNUSED;
2235     enqueue(&ptable.FreeList, p);
2236     return 0;
2237   }
2238   sp = p->kstack + KSTACKSIZE;
2239
2240   // Leave room for trap frame.
2241   sp -= sizeof *p->tf;
2242   p->tf = (struct trapframe*)sp;
2243
2244   // Set up new context to start executing at forkret,
2245   // which returns to trapret.
2246   sp -= 4;
2247   *(uint*)sp = (uint)trapret;
2248
2249
```



```
2250   sp -= sizeof *p->context;
2251   p->context = (struct context*)sp;
2252   memset(p->context, 0, sizeof *p->context);
2253   p->context->eip = (uint)forkret;
2254
2255   // STUDENT CODE
2256   // Grab Start Time
2257   acquire(&tickslock);
2258   p->start_ticks = (uint)ticks;
2259   release(&tickslock);
2260
2261   p->cpu_ticks_total = 0;
2262   p->cpu_ticks_in = 0;
2263   return p;
2264 }
2265
2266 // Set up first user process.
2267 void
2268 userinit(void)
2269 {
2270   struct proc *p;
2271   extern char _binary_initcode_start[], _binary_initcode_size[];
2272
2273   //Project 3
2274   // Before any process is created initialize all processes to
2275   // the free list
2276
2277   acquire(&ptable.lock);
2278   queue(&ptable.FreeList, &ptable.proc[0]);
2279   for(int i = 1; i < NPROC; i++){
2280     enqueue(&ptable.FreeList, &ptable.proc[i]);
2281   }
2282   release(&ptable.lock);
2283
2284
2285   p = allocproc();
2286   initproc = p;
2287   if((p->pgdir = setupkvm()) == 0)
2288     panic("userinit: out of memory?");
2289   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2290   p->sz = PGSIZE;
2291   memset(p->tf, 0, sizeof(*p->tf));
2292   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2293   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2294   p->tf->es = p->tf->ds;
2295   p->tf->ss = p->tf->ds;
2296   p->tf->eflags = FL_IF;
2297   p->tf->esp = PGSIZE;
2298   p->tf->eip = 0;  // beginning of initcode.S
2299
```

```
2300   safestrcpy(p->name, "initcode", sizeof(p->name));
2301   p->cwd = namei("/");
2302
2303   p->state = RUNNABLE;
2304
2305   p->uid = INITUID;
2306   p->gid = INITGID;
2307   p->budget = INITBUDGET;
2308   //Project 3
2309   acquire(&ptable.lock);
2310   enqueue(&ptable.ReadyList[0], p);
2311   release(&ptable.lock);
2312 }
2313
2314 // Grow current process's memory by n bytes.
2315 // Return 0 on success, -1 on failure.
2316 int
2317 growproc(int n)
2318 {
2319   uint sz;
2320
2321   sz = proc->sz;
2322   if(n > 0){
2323     if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2324       return -1;
2325   } else if(n < 0){
2326     if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2327       return -1;
2328   }
2329   proc->sz = sz;
2330   switchuvm(proc);
2331   return 0;
2332 }
2333
2334 // Create a new process copying p as the parent.
2335 // Sets up stack to return as if from system call.
2336 // Caller must set state of returned proc to RUNNABLE.
2337 int
2338 fork(void)
2339 {
2340   int i, pid;
2341   struct proc *np;
2342
2343   // Allocate process.
2344   if((np = allocproc()) == 0)
2345     return -1;
2346
2347
2348
2349
```



```
2350   // Copy process state from p.
2351   if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2352     kfree(np->kstack);
2353     np->kstack = 0;
2354     np->state = UNUSED;
2355     acquire(&ptable.lock);
2356     enqueue(&ptable.FreeList, np);
2357     release(&ptable.lock);
2358     return -1;
2359   }
2360   np->sz = proc->sz;
2361   np->parent = proc;
2362   *np->tf = *proc->tf;
2363
2364   // Clear %eax so that fork returns 0 in the child.
2365   np->tf->eax = 0;
2366
2367   for(i = 0; i < NOFILE; i++)
2368     if(proc->ofile[i])
2369       np->ofile[i] = filedup(proc->ofile[i]);
2370   np->cwd = idup(proc->cwd);
2371
2372   safestrcpy(np->name, proc->name, sizeof(proc->name));
2373
2374   //STUDENT CODE copy all ids
2375   pid = np->pid;
2376   np->uid = proc->uid;
2377   np->gid = proc->gid;
2378
2379   // lock to force the compiler to emit the np->state write last.
2380   acquire(&ptable.lock);
2381   np->state = RUNNABLE;
2382   enqueue(&ptable.ReadyList[0], np);
2383   release(&ptable.lock);
2384
2385   return pid;
2386 }
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
```

```
2400 // Exit the current process.  Does not return.
2401 // An exited process remains in the zombie state
2402 // until its parent calls wait() to find out it exited.
2403 void
2404 exit(void)
2405 {
2406   struct proc *p;
2407   int fd;
2408
2409   if(proc == initproc)
2410     panic("init exiting");
2411
2412   // Close all open files.
2413   for(fd = 0; fd < NOFILE; fd++){
2414     if(proc->ofile[fd]){
2415       fileclose(proc->ofile[fd]);
2416       proc->ofile[fd] = 0;
2417     }
2418   }
2419
2420   begin_op();
2421   iput(proc->cwd);
2422   end_op();
2423   proc->cwd = 0;
2424
2425   acquire(&ptable.lock);
2426
2427   // Parent might be sleeping in wait().
2428   wakeup1(proc->parent);
2429
2430   // Pass abandoned children to init.
2431   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2432     if(p->parent == proc){
2433       p->parent = initproc;
2434       if(p->state == ZOMBIE)
2435         wakeup1(initproc);
2436     }
2437   }
2438   // Jump into the scheduler, never to return.
2439   proc->state = ZOMBIE;
2440   sched();
2441   panic("zombie exit");
2442 }
2443
2444
2445
2446
2447
2448
2449
```



```
2450 // Wait for a child process to exit and return its pid.
2451 // Return -1 if this process has no children.
2452 int
2453 wait(void)
2454 {
2455   struct proc *p;
2456   int havekids, pid;
2457
2458   acquire(&ptable.lock);
2459   for(;;){
2460     // Scan through table looking for zombie children.
2461     havekids = 0;
2462     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2463       if(p->parent != proc)
2464         continue;
2465       havekids = 1;
2466       if(p->state == ZOMBIE){
2467         // Found one.
2468         pid = p->pid;
2469         kfree(p->kstack);
2470         p->kstack = 0;
2471         freevm(p->pgdir);
2472         p->state = UNUSED;
2473         p->pid = 0;
2474         p->parent = 0;
2475         p->name[0] = 0;
2476         p->killed = 0;
2477         enqueue(&ptable.FreeList, p);
2478         release(&ptable.lock);
2479         return pid;
2480       }
2481     }
2482
2483     // No point waiting if we don’t have any children.
2484     if(!havekids || proc->killed){
2485       release(&ptable.lock);
2486       return -1;
2487     }
2488
2489     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2490     sleep(proc, &ptable.lock);
2491   }
2492 }
2493
2494
2495
2496
2497
2498
2499
```

```
2500 // Per-CPU process scheduler.
2501 // Each CPU calls scheduler() after setting itself up.
2502 // Scheduler never returns.  It loops, doing:
2503 //  - choose a process to run
2504 //  - swtch to start running that process
2505 //  - eventually that process transfers control
2506 //      via swtch back to the scheduler.
2507 #ifndef CS333_P3
2508 // original xv6 scheduler. Use if CS333_P3 NOT defined.
2509 void
2510 scheduler(void)
2511 {
2512   struct proc *p;
2513   uint now;
2514
2515   for(;;){
2516     // Enable interrupts on this processor.
2517     sti();
2518
2519     // Loop over process table looking for process to run.
2520     acquire(&ptable.lock);
2521     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2522       if(p->state != RUNNABLE)
2523         continue;
2524
2525       // Switch to chosen process.  It is the process's job
2526       // to release ptable.lock and then reacquire it
2527       // before jumping back to us.
2528       proc = p;
2529       switchuvm(p);
2530       p->state = RUNNING;
2531
2532       acquire(&tickslock);
2533       now = (uint)ticks;
2534       release(&tickslock);
2535       p->cpu_ticks_in = now;
2536
2537       swtch(&cpu->scheduler, proc->context);
2538       switchkvm();
2539
2540       // Process is done running for now.
2541       // It should have changed its p->state before coming back.
2542       proc = 0;
2543     }
2544     release(&ptable.lock);
2545
2546   }
2547 }
2548
2549
```



```
2550 #else
2551 // CS333_P3 MLFQ scheduler implementation goes here
2552 void
2553 scheduler(void)
2554 {
2555
2556   int CURRENT = 0;
2557   struct proc *p = 0;
2558   uint now = 0;
2559
2560 //  acquire(&ptable.lock);
2561 //  ptable.PromoteAtTime = TICKS_TO_PROMOTE;
2562 //  release(&ptable.lock);
2563   for(;;){
2564     // Enable interrupts on this processor.
2565     sti();
2566 //    if(now >= ptable.PromoteAtTime ){
2567 //          promote();
2568 //          ptable.PromoteAtTime = ptable.PromoteAtTime + TICKS_TO_PROMOTE;
2569 //    }
2570     if(CURRENT == NUM_READY_LISTS) //Go back to top of queue
2571         CURRENT = 0;
2572     // Loop over process table looking for process to run.
2573     acquire(&ptable.lock);
2574     p = dequeue(&ptable.ReadyList[CURRENT]);
2575     if(p == 0){
2576        CURRENT++;
2577         release(&ptable.lock);
2578         continue;
2579     }
2580       // Switch to chosen process.  It is the process's job
2581       // to release ptable.lock and then reacquire it
2582       // before jumping back to us.
2583       proc = p;
2584       switchuvm(p);
2585       p->state = RUNNING;
2586
2587       acquire(&tickslock);
2588       now = (uint)ticks;
2589       release(&tickslock);
2590       p->cpu_ticks_in = now;
2591       // Is it time to promote?
2592      swtch(&cpu->scheduler, proc->context);
2593       switchkvm();
2594
2595       // Process is done running for now.
2596       // It should have changed its p->state before coming back.
2597       proc = 0;
2598     release(&ptable.lock);
2599    }
```

```
2600 }
2601 #endif
2602
2603 // Enter scheduler.  Must hold only ptable.lock
2604 // and have changed proc->state.
2605 void
2606 sched(void)
2607 {
2608   int rc;
2609   int intena;
2610   uint now;
2611
2612   if(!holding(&ptable.lock))
2613     panic("sched ptable.lock");
2614   if(cpu->ncli != 1)
2615     panic("sched locks");
2616   if(proc->state == RUNNING)
2617     panic("sched running");
2618   if(readeflags()&FL_IF)
2619     panic("sched interruptible");
2620   intena = cpu->intena;
2621
2622   acquire(&tickslock);
2623   now = (uint)ticks;
2624   release(&tickslock);
2625
2626   proc->cpu_ticks_total = proc->cpu_ticks_total + (now - proc->cpu_ticks_in);
2627
2628   rc = updateBudget(proc, now);
2629   if(rc && (proc->priority< NUM_READY_LISTS)){
2630       cprintf("%s demoted, was %d, now %d\n", proc->name, proc->priority, pr
2631       proc->priority = proc->priority+ 1;
2632   }
2633
2634   swtch(&proc->context, cpu->scheduler);
2635   cpu->intena = intena;
2636 }
2637
2638 // Give up the CPU for one scheduling round.
2639 void
2640 yield(void)
2641 {
2642   acquire(&ptable.lock);
2643   proc->state = RUNNABLE;
2644   enqueue(&ptable.ReadyList[proc->priority], proc);
2645   sched();
2646   release(&ptable.lock);
2647 }
2648
2649
```



```
2650 // A fork child's very first scheduling by scheduler()
2651 // will swtch here.  "Return" to user space.
2652 void
2653 forkret(void)
2654 {
2655   static int first = 1;
2656   // Still holding ptable.lock from scheduler.
2657   release(&ptable.lock);
2658
2659   if (first) {
2660     // Some initialization functions must be run in the context
2661     // of a regular process (e.g., they call sleep), and thus cannot
2662     // be run from main().
2663     first = 0;
2664     iinit(ROOTDEV);
2665     initlog(ROOTDEV);
2666   }
2667
2668   // Return to "caller", actually trapret (see allocproc).
2669 }
2670
2671 // Atomically release lock and sleep on chan.
2672 // Reacquires lock when awakened.
2673 void
2674 sleep(void *chan, struct spinlock *lk)
2675 {
2676   if(proc == 0)
2677     panic("sleep");
2678
2679   if(lk == 0)
2680     panic("sleep without lk");
2681
2682   // Must acquire ptable.lock in order to
2683   // change p->state and then call sched.
2684   // Once we hold ptable.lock, we can be
2685   // guaranteed that we won't miss any wakeup
2686   // (wakeup runs with ptable.lock locked),
2687   // so it's okay to release lk.
2688   if(lk != &ptable.lock){
2689     acquire(&ptable.lock);
2690     release(lk);
2691   }
2692
2693   // Go to sleep.
2694   proc->chan = chan;
2695   proc->state = SLEEPING;
2696   sched();
2697
2698   // Tidy up.
2699   proc->chan = 0;
```

```
2700   // Reacquire original lock.
2701   if(lk != &ptable.lock){
2702     release(&ptable.lock);
2703     acquire(lk);
2704   }
2705 }
2706
2707 // Wake up all processes sleeping on chan.
2708 // The ptable lock must be held.
2709 static void
2710 wakeup1(void *chan)
2711 {
2712   struct proc *p;
2713
2714   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2715     if(p->state == SLEEPING && p->chan == chan){
2716       p->state = RUNNABLE;
2717       enqueue(&ptable.ReadyList[0], p);
2718     }
2719
2720 }
2721
2722 // Wake up all processes sleeping on chan.
2723 void
2724 wakeup(void *chan)
2725 {
2726   acquire(&ptable.lock);
2727   wakeup1(chan);
2728   release(&ptable.lock);
2729 }
2730
2731 // Kill the process with the given pid.
2732 // Process won't exit until it returns
2733 // to user space (see trap in trap.c).
2734 int
2735 kill(int pid)
2736 {
2737   struct proc *p;
2738
2739   acquire(&ptable.lock);
2740   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2741     if(p->pid == pid){
2742       p->killed = 1;
2743       // Wake process from sleep if necessary.
2744       if(p->state == SLEEPING){
2745         p->state = RUNNABLE;
2746         enqueue(&ptable.ReadyList[0], p);
2747       }
2748       release(&ptable.lock);
2749       return 0;
```



```
2750     }
2751   }
2752   release(&ptable.lock);
2753   return -1;
2754 }
2755
2756 // Print a process listing to console.  For debugging.
2757 // Runs when user types ^P on console.
2758 // No lock to avoid wedging a stuck machine further.
2759 void
2760 procdump(void)
2761 {
2762   static char *states[] = {
2763   [UNUSED]    "unused",
2764   [EMBRYO]    "embryo",
2765   [SLEEPING]  "sleep ",
2766   [RUNNABLE]  "runble",
2767   [RUNNING]   "run   ",
2768   [ZOMBIE]    "zombie"
2769   };
2770   int i;
2771   struct proc *p;
2772   char *state;
2773   uint pc[10];
2774
2775   uint now;         //Snag the current ticks and cast
2776
2777   acquire(&tickslock);
2778   now = (uint)ticks;
2779   release(&tickslock);
2780
2781   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2782     if(p->state == UNUSED)
2783       continue;
2784     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
2785       state = states[p->state];
2786     else
2787       state = "???";
2788     cprintf("%d %s %s %d %d.%d %d.%d", p->pid, state, p->name, p->priority,
2789             (now - p->start_ticks) / 100,
2790             (now - p->start_ticks) % 100,
2791              p->cpu_ticks_total / 100,
2792              p->cpu_ticks_total % 100);
2793     if(p->state == SLEEPING){
2794       getcallerpcs((uint*)p->context->ebp+2, pc);
2795       for(i=0; i<10 && pc[i] != 0; i++)
2796         cprintf(" %p", pc[i]);
2797     }
2798     cprintf("\n");
2799   }
```

```
2800 }
2801
2802
2803 int
2804 sys_getprocs(void)
2805 {
2806
2807   static char *states[] = {
2808   [UNUSED]    "unused",
2809   [EMBRYO]    "embryo",
2810   [SLEEPING]  "sleep ",
2811   [RUNNABLE]  "runble",
2812   [RUNNING]   "run   ",
2813   [ZOMBIE]    "zombie"
2814   };
2815
2816   uint now;
2817   acquire(&tickslock);
2818   now = (uint)ticks;
2819   release(&tickslock);
2820
2821
2822     struct proc* p;
2823     struct uproc* up;
2824     int MAX = 0;
2825     int i = 0;
2826
2827     if( argint(0, &MAX) == -1)
2828         return -1;
2829
2830     if(argptr(1, (char**)&up, sizeof(*up)) < 0)
2831         return -1;
2832
2833   acquire(&ptable.lock);
2834
2835   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2836   {
2837
2838         if( p->state && i < MAX){
2839
2840             up[i].pid = p->pid;
2841             up[i].uid = p->uid;
2842             up[i].gid = p->gid;
2843             up[i].CPU_total_ticks = p->cpu_ticks_total;
2844             up[i].elapsed_ticks = now - p->start_ticks;
2845             up[i].size = p->sz;
2846 //            up[i].priority = p->priority;
2847             safestrcpy(up[i].name, p->name, sizeof(p->name));
2848             safestrcpy(up[i].state, states[p->state], sizeof(p->state));
2849             if(up[i].pid == 1)
```



```
2850                 up[i].ppid = 1;
2851             else
2852                 up[i].ppid = p->parent->pid;
2853             i++;
2854         }
2855   }
2856   release(&ptable.lock);
2857   return i;
2858 }
2859
2860 //Implementations for priority queue struct
2861
2862 //Constructor set all values to null
2863 int
2864 queue(struct queue *this, struct proc *first )
2865 {
2866     this->head  = this->tail = first;         // This works fine even if the
2867     return 0;                                 // it just feels more OO to ha
2868                                               // and makes it more readable
2869 }
2870
2871 // add process to the end of the list
2872 int
2873 enqueue(struct queue *this, struct proc *nproc){
2874
2875       if(this->tail == 0){                  // If list is empty initialize
2876           return queue(this, nproc);
2877           return 0;
2878       }
2879
2880       this->tail->next = nproc;            // Otherwise add it in as normal
2881       this->tail = nproc;
2882       this->tail->next = 0;
2883       return 0;
2884 }
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
```

```
2900 // take process from head of list
2901 struct proc*
2902 dequeue(struct queue *this){
2903
2904      struct proc* nproc = 0;;
2905       if(this->tail == 0)
2906           return  nproc;
2907       if(this->tail == this->head){
2908           nproc = this->tail;
2909           this->head = 0;
2910           this->tail = 0;
2911           nproc->next = 0;
2912           return nproc;
2913         }
2914       nproc = this->head;
2915       this->head = this->head->next;
2916       nproc->next = 0;
2917       return nproc;
2918
2919 }
2920
2921 int
2922 updateBudget(struct proc * this, int now){
2923
2924     this->budget = this->budget - (now - this->cpu_ticks_in);
2925     if(this->budget <=0){
2926         this->budget = INITBUDGET;
2927         return 1;
2928     }
2929     return 0;
2930 }
2931
2932 void
2933 promote(){
2934
2935     int i = 1;
2936     struct proc * p;
2937     for(i = 1;  i < NUM_READY_LISTS; i++){
2938         p = dequeue(&ptable.ReadyList[i]);
2939         if(p == 0)
2940             continue;
2941         p->budget = INITBUDGET;
2942         p->priority = p->priority - 1;
2943         enqueue(&ptable.ReadyList[p->priority - 1], p);
2944     }
2945 }
2946
2947
2948
2949
```



```
2950 # Context switch
2951 #
2952 #   void swtch(struct context **old, struct context *new);
2953 #
2954 # Save current register context in old
2955 # and then load register context from new.
2956
2957 .globl swtch
2958 swtch:
2959   movl 4(%esp), %eax
2960   movl 8(%esp), %edx
2961
2962   # Save old callee-save registers
2963   pushl %ebp
2964   pushl %ebx
2965   pushl %esi
2966   pushl %edi
2967
2968   # Switch stacks
2969   movl %esp, (%eax)
2970   movl %edx, %esp
2971
2972   # Load new callee-save registers
2973   popl %edi
2974   popl %esi
2975   popl %ebx
2976   popl %ebp
2977   ret
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
```

```
3000 // Physical memory allocator, intended to allocate
3001 // memory for user processes, kernel stacks, page table pages,
3002 // and pipe buffers. Allocates 4096-byte pages.
3003
3004 #include "types.h"
3005 #include "defs.h"
3006 #include "param.h"
3007 #include "memlayout.h"
3008 #include "mmu.h"
3009 #include "spinlock.h"
3010
3011 void freerange(void *vstart, void *vend);
3012 extern char end[]; // first address after kernel loaded from ELF file
3013
3014 struct run {
3015   struct run *next;
3016 };
3017
3018 struct {
3019   struct spinlock lock;
3020   int use_lock;
3021   struct run *freelist;
3022 } kmem;
3023
3024 // Initialization happens in two phases.
3025 // 1. main() calls kinit1() while still using entrypgdir to place just
3026 // the pages mapped by entrypgdir on free list.
3027 // 2. main() calls kinit2() with the rest of the physical pages
3028 // after installing a full page table that maps them on all cores.
3029 void
3030 kinit1(void *vstart, void *vend)
3031 {
3032   initlock(&kmem.lock, "kmem");
3033   kmem.use_lock = 0;
3034   freerange(vstart, vend);
3035 }
3036
3037 void
3038 kinit2(void *vstart, void *vend)
3039 {
3040   freerange(vstart, vend);
3041   kmem.use_lock = 1;
3042 }
3043
3044
3045
3046
3047
3048
3049
```



```
3050 void
3051 freerange(void *vstart, void *vend)
3052 {
3053   char *p;
3054   p = (char*)PGROUNDUP((uint)vstart);
3055   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3056     kfree(p);
3057 }
3058
3059 // Free the page of physical memory pointed at by v,
3060 // which normally should have been returned by a
3061 // call to kalloc().  (The exception is when
3062 // initializing the allocator; see kinit above.)
3063 void
3064 kfree(char *v)
3065 {
3066   struct run *r;
3067
3068   if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
3069     panic("kfree");
3070
3071   // Fill with junk to catch dangling refs.
3072   memset(v, 1, PGSIZE);
3073
3074   if(kmem.use_lock)
3075     acquire(&kmem.lock);
3076   r = (struct run*)v;
3077   r->next = kmem.freelist;
3078   kmem.freelist = r;
3079   if(kmem.use_lock)
3080     release(&kmem.lock);
3081 }
3082
3083 // Allocate one 4096-byte page of physical memory.
3084 // Returns a pointer that the kernel can use.
3085 // Returns 0 if the memory cannot be allocated.
3086 char*
3087 kalloc(void)
3088 {
3089   struct run *r;
3090
3091   if(kmem.use_lock)
3092     acquire(&kmem.lock);
3093   r = kmem.freelist;
3094   if(r)
3095     kmem.freelist = r->next;
3096   if(kmem.use_lock)
3097     release(&kmem.lock);
3098   return (char*)r;
3099 }
```

```
3100 // x86 trap and interrupt constants.
3101
3102 // Processor-defined:
3103 #define T_DIVIDE         0      // divide error
3104 #define T_DEBUG          1      // debug exception
3105 #define T_NMI            2      // non-maskable interrupt
3106 #define T_BRKPT          3      // breakpoint
3107 #define T_OFLOW          4      // overflow
3108 #define T_BOUND          5      // bounds check
3109 #define T_ILLOP          6      // illegal opcode
3110 #define T_DEVICE         7      // device not available
3111 #define T_DBLFLT         8      // double fault
3112 // #define T_COPROC      9      // reserved (not used since 486)
3113 #define T_TSS           10      // invalid task switch segment
3114 #define T_SEGNP         11      // segment not present
3115 #define T_STACK         12      // stack exception
3116 #define T_GPFLT         13      // general protection fault
3117 #define T_PGFLT         14      // page fault
3118 // #define T_RES        15      // reserved
3119 #define T_FPERR         16      // floating point error
3120 #define T_ALIGN         17      // aligment check
3121 #define T_MCHK          18      // machine check
3122 #define T_SIMDERR       19      // SIMD floating point error
3123
3124 // These are arbitrarily chosen, but with care not to overlap
3125 // processor defined exceptions or interrupt vectors.
3126 #define T_SYSCALL       64      // system call
3127 #define T_DEFAULT      500      // catchall
3128
3129 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
3130
3131 #define IRQ_TIMER        0
3132 #define IRQ_KBD          1
3133 #define IRQ_COM1         4
3134 #define IRQ_IDE         14
3135 #define IRQ_ERROR       19
3136 #define IRQ_SPURIOUS    31
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
```



```
3150 #!/usr/bin/perl -w
3151
3152 # Generate vectors.S, the trap/interrupt entry points.
3153 # There has to be one entry point per interrupt number
3154 # since otherwise there's no way for trap() to discover
3155 # the interrupt number.
3156
3157 print "# generated by vectors.pl - do not edit\n";
3158 print "# handlers\n";
3159 print ".globl alltraps\n";
3160 for(my $i = 0; $i < 256; $i++){
3161     print ".globl vector$i\n";
3162     print "vector$i:\n";
3163     if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
3164         print "  pushl \$0\n";
3165     }
3166     print "  pushl \$$i\n";
3167     print "  jmp alltraps\n";
3168 }
3169
3170 print "\n# vector table\n";
3171 print ".data\n";
3172 print ".globl vectors\n";
3173 print "vectors:\n";
3174 for(my $i = 0; $i < 256; $i++){
3175     print "  .long vector$i\n";
3176 }
3177
3178 # sample output:
3179 #   # handlers
3180 #   .globl alltraps
3181 #   .globl vector0
3182 #   vector0:
3183 #     pushl $0
3184 #     pushl $0
3185 #     jmp alltraps
3186 #   ...
3187 #
3188 #   # vector table
3189 #   .data
3190 #   .globl vectors
3191 #   vectors:
3192 #     .long vector0
3193 #     .long vector1
3194 #     .long vector2
3195 #   ...
3196
3197
3198
3199
```

```
3200 #include "mmu.h"
3201
3202   # vectors.S sends all traps here.
3203 .globl alltraps
3204 alltraps:
3205   # Build trap frame.
3206   pushl %ds
3207   pushl %es
3208   pushl %fs
3209   pushl %gs
3210   pushal
3211
3212   # Set up data and per-cpu segments.
3213   movw $(SEG_KDATA<<3), %ax
3214   movw %ax, %ds
3215   movw %ax, %es
3216   movw $(SEG_KCPU<<3), %ax
3217   movw %ax, %fs
3218   movw %ax, %gs
3219
3220   # Call trap(tf), where tf=%esp
3221   pushl %esp
3222   call trap
3223   addl $4, %esp
3224
3225   # Return falls through to trapret...
3226 .globl trapret
3227 trapret:
3228   popal
3229   popl %gs
3230   popl %fs
3231   popl %es
3232   popl %ds
3233   addl $0x8, %esp  # trapno and errcode
3234   iret
3235
3236
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
```



```
3250 #include "types.h"
3251 #include "defs.h"
3252 #include "param.h"
3253 #include "memlayout.h"
3254 #include "mmu.h"
3255 #include "proc.h"
3256 #include "x86.h"
3257 #include "traps.h"
3258 #include "spinlock.h"
3259
3260 // Interrupt descriptor table (shared by all CPUs).
3261 struct gatedesc idt[256];
3262 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
3263 struct spinlock tickslock;
3264 uint ticks;
3265
3266 void
3267 tvinit(void)
3268 {
3269   int i;
3270
3271   for(i = 0; i < 256; i++)
3272     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3273   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3274
3275   initlock(&tickslock, "time");
3276 }
3277
3278 void
3279 idtinit(void)
3280 {
3281   lidt(idt, sizeof(idt));
3282 }
3283
3284 void
3285 trap(struct trapframe *tf)
3286 {
3287   if(tf->trapno == T_SYSCALL){
3288     if(proc->killed)
3289       exit();
3290     proc->tf = tf;
3291     syscall();
3292     if(proc->killed)
3293       exit();
3294     return;
3295   }
3296
3297   switch(tf->trapno){
3298   case T_IRQ0 + IRQ_TIMER:
3299     if(cpu->id == 0){
```

```
3300      acquire(&tickslock);
3301      ticks++;
3302      release(&tickslock);    // NOTE: MarkM has reversed these two lines.
3303      wakeup(&ticks);         // wakeup() should not require the tickslock t
3304    }
3305    lapiceoi();
3306    break;
3307  case T_IRQ0 + IRQ_IDE:
3308    ideintr();
3309    lapiceoi();
3310    break;
3311  case T_IRQ0 + IRQ_IDE+1:
3312    // Bochs generates spurious IDE1 interrupts.
3313    break;
3314  case T_IRQ0 + IRQ_KBD:
3315    kbdintr();
3316    lapiceoi();
3317    break;
3318  case T_IRQ0 + IRQ_COM1:
3319    uartintr();
3320    lapiceoi();
3321    break;
3322  case T_IRQ0 + 7:
3323  case T_IRQ0 + IRQ_SPURIOUS:
3324    cprintf("cpu%d: spurious interrupt at %x:%x\n",
3325            cpu->id, tf->cs, tf->eip);
3326    lapiceoi();
3327    break;
3328
3329  default:
3330    if(proc == 0 || (tf->cs&3) == 0){
3331      // In kernel, it must be our mistake.
3332      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3333              tf->trapno, cpu->id, tf->eip, rcr2());
3334      panic("trap");
3335    }
3336    // In user space, assume process misbehaved.
3337    cprintf("pid %d %s: trap %d err %d on cpu %d "
3338            "eip 0x%x addr 0x%x--kill proc\n",
3339            proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
3340            rcr2());
3341    proc->killed = 1;
3342  }
3343
3344  // Force process exit if it has been killed and is in user space.
3345  // (If it is still executing in the kernel, let it keep running
3346  // until it gets to the regular system call return.)
3347  if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3348    exit();
3349
```



```
3350  // Force process to give up CPU on clock tick.
3351  // If interrupts were on while locks held, would need to check nlock.
3352  if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3353    yield();
3354
3355  // Check if the process has been killed since we yielded
3356  if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3357    exit();
3358 }
3359
3360
3361
3362
3363
3364
3365
3366
3367
3368
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3379
3380
3381
3382
3383
3384
3385
3386
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399
```

```
3400 // System call numbers
3401 #define SYS_fork    1
3402 #define SYS_exit    SYS_fork+1
3403 #define SYS_wait    SYS_exit+1
3404 #define SYS_pipe    SYS_wait+1
3405 #define SYS_read    SYS_pipe+1
3406 #define SYS_kill    SYS_read+1
3407 #define SYS_exec    SYS_kill+1
3408 #define SYS_fstat   SYS_exec+1
3409 #define SYS_chdir   SYS_fstat+1
3410 #define SYS_dup     SYS_chdir+1
3411 #define SYS_getpid  SYS_dup+1
3412 #define SYS_sbrk    SYS_getpid+1
3413 #define SYS_sleep   SYS_sbrk+1
3414 #define SYS_uptime  SYS_sleep+1
3415 #define SYS_open    SYS_uptime+1
3416 #define SYS_write   SYS_open+1
3417 #define SYS_mknod   SYS_write+1
3418 #define SYS_unlink  SYS_mknod+1
3419 #define SYS_link    SYS_unlink+1
3420 #define SYS_mkdir   SYS_link+1
3421 #define SYS_close   SYS_mkdir+1
3422 #define SYS_halt    SYS_close+1
3423 // student system calls begin here. Follow the existing pattern.
3424
3425 #define SYS_date       SYS_halt+1
3426
3427 #define SYS_getuid     SYS_date+1
3428 #define SYS_getgid     SYS_getuid+1
3429 #define SYS_getppid    SYS_getgid+1
3430
3431 #define SYS_setuid     SYS_getppid+1
3432 #define SYS_setgid     SYS_setuid+1
3433 #define SYS_getprocs   SYS_setgid+1
3434
3435 // Project 3
3436 #define SYS_setpriority SYS_getprocs+1
3437
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
```



```
3450 #include "types.h"
3451 #include "defs.h"
3452 #include "param.h"
3453 #include "memlayout.h"
3454 #include "mmu.h"
3455 #include "proc.h"
3456 #include "x86.h"
3457 #include "syscall.h"
3458
3459 // User code makes a system call with INT T_SYSCALL.
3460 // System call number in %eax.
3461 // Arguments on the stack, from the user call to the C
3462 // library system call function. The saved user %esp points
3463 // to a saved program counter, and then the first argument.
3464
3465 // Fetch the int at addr from the current process.
3466 int
3467 fetchint(uint addr, int *ip)
3468 {
3469   if(addr >= proc->sz || addr+4 > proc->sz)
3470     return -1;
3471   *ip = *(int*)(addr);
3472   return 0;
3473 }
3474
3475 // Fetch the nul-terminated string at addr from the current process.
3476 // Doesn't actually copy the string - just sets *pp to point at it.
3477 // Returns length of string, not including nul.
3478 int
3479 fetchstr(uint addr, char **pp)
3480 {
3481   char *s, *ep;
3482
3483   if(addr >= proc->sz)
3484     return -1;
3485   *pp = (char*)addr;
3486   ep = (char*)proc->sz;
3487   for(s = *pp; s < ep; s++)
3488     if(*s == 0)
3489       return s - *pp;
3490   return -1;
3491 }
3492
3493 // Fetch the nth 32-bit system call argument.
3494 int
3495 argint(int n, int *ip)
3496 {
3497   return fetchint(proc->tf->esp + 4 + 4*n, ip);
3498 }
3499
```

```
3500 // Fetch the nth word-sized system call argument as a pointer
3501 // to a block of memory of size n bytes.  Check that the pointer
3502 // lies within the process address space.
3503 int
3504 argptr(int n, char **pp, int size)
3505 {
3506   int i;
3507
3508   if(argint(n, &i) < 0)
3509     return -1;
3510   if((uint)i >= proc->sz || (uint)i+size > proc->sz)
3511     return -1;
3512   *pp = (char*)i;
3513   return 0;
3514 }
3515
3516 // Fetch the nth word-sized system call argument as a string pointer.
3517 // Check that the pointer is valid and the string is nul-terminated.
3518 // (There is no shared writable memory, so the string can't change
3519 // between this check and being used by the kernel.)
3520 int
3521 argstr(int n, char **pp)
3522 {
3523   int addr;
3524   if(argint(n, &addr) < 0)
3525     return -1;
3526   return fetchstr(addr, pp);
3527 }
3528
3529 extern int sys_chdir(void);
3530 extern int sys_close(void);
3531 extern int sys_dup(void);
3532 extern int sys_exec(void);
3533 extern int sys_exit(void);
3534 extern int sys_fork(void);
3535 extern int sys_fstat(void);
3536 extern int sys_getpid(void);
3537 extern int sys_kill(void);
3538 extern int sys_link(void);
3539 extern int sys_mkdir(void);
3540 extern int sys_mknod(void);
3541 extern int sys_open(void);
3542 extern int sys_pipe(void);
3543 extern int sys_read(void);
3544 extern int sys_sbrk(void);
3545 extern int sys_sleep(void);
3546 extern int sys_unlink(void);
3547 extern int sys_wait(void);
3548 extern int sys_write(void);
3549 extern int sys_uptime(void);
```



```
3550 extern int sys_halt(void);
3551
3552 //Student functions
3553 extern int sys_date(void);
3554
3555 extern int sys_getuid(void);
3556 extern int sys_getgid(void);
3557 extern int sys_getppid(void);
3558
3559 extern int sys_setuid(void);
3560 extern int sys_setgid(void);
3561
3562 extern int sys_getprocs(void);
3563 // Project 3
3564 extern int sys_setpriority(void);
3565
3566 static int (*syscalls[])(void) = {
3567 [SYS_fork]    sys_fork,
3568 [SYS_exit]    sys_exit,
3569 [SYS_wait]    sys_wait,
3570 [SYS_pipe]    sys_pipe,
3571 [SYS_read]    sys_read,
3572 [SYS_kill]    sys_kill,
3573 [SYS_exec]    sys_exec,
3574 [SYS_fstat]   sys_fstat,
3575 [SYS_chdir]   sys_chdir,
3576 [SYS_dup]     sys_dup,
3577 [SYS_getpid]  sys_getpid,
3578 [SYS_sbrk]    sys_sbrk,
3579 [SYS_sleep]   sys_sleep,
3580 [SYS_uptime]  sys_uptime,
3581 [SYS_open]    sys_open,
3582 [SYS_write]   sys_write,
3583 [SYS_mknod]   sys_mknod,
3584 [SYS_unlink]  sys_unlink,
3585 [SYS_link]    sys_link,
3586 [SYS_mkdir]   sys_mkdir,
3587 [SYS_close]   sys_close,
3588 [SYS_halt]    sys_halt,
3589
3590 //Student Functions
3591 [SYS_date]    sys_date,
3592
3593 [SYS_getuid]  sys_getuid,
3594 [SYS_getgid]  sys_getgid,
3595 [SYS_getppid] sys_getppid,
3596
3597 [SYS_setuid]  sys_setuid,
3598 [SYS_setgid]  sys_setgid,
3599
```

```
[SYS_getprocs] sys_getprocs,

// Project 3
[SYS_setpriority] sys_setpriority,

};

// put data structure for printing out system call invocation information he
// This is basically an enum, but can get called differently
#ifdef PRINT_SYSCALLS
char* (sysname[]) = {
[SYS_fork]    "sys_fork",
[SYS_exit]    "sys_exit",
[SYS_wait]    "sys_wait",
[SYS_pipe]    "sys_pipe",
[SYS_read]    "sys_read",
[SYS_kill]    "sys_kill",
[SYS_exec]    "sys_exec",
[SYS_fstat]   "sys_fstat",
[SYS_chdir]   "sys_chdir",
[SYS_dup]     "sys_dup",
[SYS_getpid]  "sys_getpid",
[SYS_sbrk]    "sys_sbrk",
[SYS_sleep]   "sys_sleep",
[SYS_uptime]  "sys_uptime",
[SYS_open]    "sys_open",
[SYS_write]   "sys_write",
[SYS_mknod]   "sys_mknod",
[SYS_unlink]  "sys_unlink",
[SYS_link]    "sys_link",
[SYS_mkdir]   "sys_mkdir",
[SYS_close]   "sys_close",
[SYS_halt]    "sys_halt",
};
#endif


void
syscall(void)
{
  int num;

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
    //Start of Project 1
    //cprintf("Syscall Happening '\n'");
```



```
    #ifdef PRINT_SYSCALLS
    cprintf("\n \t \t \t  %s -> %d \n", sysname[num], proc->tf->eax );
    #endif

  } else {
    cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```

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



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

```
3800 //Turn of the computer
3801 int sys_halt(void){
3802   cprintf("Shutting down ...\n");
3803   outw (0xB004, 0x0 | 0x2000);
3804   return 0;
3805 }
3806
3807 int sys_date(void){
3808
3809     struct rtcdate* d;
3810
3811     if(argptr(0, (char**)&d, sizeof(*d)) <0)
3812         return -1;
3813
3814     cmostime(d);
3815     return 0;
3816 }
3817
3818
3819 uint
3820 sys_getuid(void)
3821 {
3822     return proc->uid;
3823 }
3824
3825 uint
3826 sys_getgid(void)
3827 {
3828     return proc->gid;
3829 }
3830
3831 uint
3832 sys_getppid(void)
3833 {
3834
3835     if(proc->pid == 1)
3836         return 1;
3837     return proc->parent->pid;
3838 }
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
```



```
3850 int
3851 sys_setuid(void)
3852 {
3853
3854
3855     if(argint(0, (int*)&proc->uid))
3856         return -1;
3857
3858     if(proc->uid < 0 || proc->uid > 32767){
3859         proc->uid = 0;
3860         return -1;
3861     }
3862     else
3863         return 0;
3864 }
3865
3866 int
3867 sys_setgid(void)
3868 {
3869
3870     if(argint(0, (int*)&proc->gid) )
3871         return -1;
3872     if(proc->gid < 0 || proc -> gid > 32767){
3873         proc->gid = 0;
3874         return -1;
3875     }
3876     else
3877         return 0;
3878 }
3879
3880
3881 // Project 3
3882 int
3883 sys_setpriority(void){
3884
3885     int pid, priority;
3886    if(argint(0, &pid) < 0 )
3887         return -1;
3888    if(argint(0, &priority) < 0 && priority <= NUM_READY_LISTS ) // validate s
3889         return -1;
3890    if(proc->pid == pid){   // garuntee this is correct process
3891        proc->priority = priority; //set new priority
3892        return 0;
3893     }
3894    else
3895        return -1;
3896
3897     return 0;        //Error not reached!
3898 }
3899
```

```
3900 // halt the system.
3901 #include "types.h"
3902 #include "user.h"
3903
3904 int
3905 main(void) {
3906   halt();
3907   return 0;
3908 }
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
```



```
3950 struct buf {
3951   int flags;
3952   uint dev;
3953   uint blockno;
3954   struct buf *prev; // LRU cache list
3955   struct buf *next;
3956   struct buf *qnext; // disk queue
3957   uchar data[BSIZE];
3958 };
3959 #define B_BUSY  0x1  // buffer is locked by some process
3960 #define B_VALID 0x2  // buffer has been read from disk
3961 #define B_DIRTY 0x4  // buffer needs to be written to disk
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
```

```
4000 #define O_RDONLY  0x000
4001 #define O_WRONLY  0x001
4002 #define O_RDWR    0x002
4003 #define O_CREATE  0x200
4004
4005
4006
4007
4008
4009
4010
4011
4012
4013
4014
4015
4016
4017
4018
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
```



```
4050 #define T_DIR  1   // Directory
4051 #define T_FILE 2   // File
4052 #define T_DEV  3   // Device
4053
4054 struct stat {
4055   short type;  // Type of file
4056   int dev;     // File system's disk device
4057   uint ino;    // Inode number
4058   short nlink; // Number of links to file
4059   uint size;   // Size of file in bytes
4060 };
4061
4062
4063
4064
4065
4066
4067
4068
4069
4070
4071
4072
4073
4074
4075
4076
4077
4078
4079
4080
4081
4082
4083
4084
4085
4086
4087
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
```

```
4100 // On-disk file system format.
4101 // Both the kernel and user programs use this header file.
4102
4103
4104 #define ROOTINO 1  // root i-number
4105 #define BSIZE 512  // block size
4106
4107 // Disk layout:
4108 // [ boot block | super block | log | inode blocks | free bit map | data blo
4109 //
4110 // mkfs computes the super block and builds an initial file system. The supe
4111 // the disk layout:
4112 struct superblock {
4113   uint size;         // Size of file system image (blocks)
4114   uint nblocks;      // Number of data blocks
4115   uint ninodes;      // Number of inodes.
4116   uint nlog;         // Number of log blocks
4117   uint logstart;     // Block number of first log block
4118   uint inodestart;   // Block number of first inode block
4119   uint bmapstart;    // Block number of first free map block
4120 };
4121
4122 #define NDIRECT 12
4123 #define NINDIRECT (BSIZE / sizeof(uint))
4124 #define MAXFILE (NDIRECT + NINDIRECT)
4125
4126 // On-disk inode structure
4127 struct dinode {
4128   short type;           // File type
4129   short major;          // Major device number (T_DEV only)
4130   short minor;          // Minor device number (T_DEV only)
4131   short nlink;          // Number of links to inode in file system
4132   uint size;            // Size of file (bytes)
4133   uint addrs[NDIRECT+1];   // Data block addresses
4134 };
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```



```
4150 // Inodes per block.
4151 #define IPB           (BSIZE / sizeof(struct dinode))
4152
4153 // Block containing inode i
4154 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
4155
4156 // Bitmap bits per block
4157 #define BPB           (BSIZE*8)
4158
4159 // Block of free map containing bit for block b
4160 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
4161
4162 // Directory is a file containing a sequence of dirent structures.
4163 #define DIRSIZ 14
4164
4165 struct dirent {
4166   ushort inum;
4167   char name[DIRSIZ];
4168 };
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
```

```
4200 struct file {
4201   enum { FD_NONE, FD_PIPE, FD_INODE } type;
4202   int ref; // reference count
4203   char readable;
4204   char writable;
4205   struct pipe *pipe;
4206   struct inode *ip;
4207   uint off;
4208 };
4209
4210
4211 // in-memory copy of an inode
4212 struct inode {
4213   uint dev;           // Device number
4214   uint inum;          // Inode number
4215   int ref;            // Reference count
4216   int flags;          // I_BUSY, I_VALID
4217
4218   short type;         // copy of disk inode
4219   short major;
4220   short minor;
4221   short nlink;
4222   uint size;
4223   uint addrs[NDIRECT+1];
4224 };
4225 #define I_BUSY 0x1
4226 #define I_VALID 0x2
4227
4228 // table mapping major device number to
4229 // device functions
4230 struct devsw {
4231   int (*read)(struct inode*, char*, int);
4232   int (*write)(struct inode*, char*, int);
4233 };
4234
4235 extern struct devsw devsw[];
4236
4237 #define CONSOLE 1
4238
4239 // Blank page.
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
```



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

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



```
4350 // Interrupt handler.
4351 void
4352 ideintr(void)
4353 {
4354   struct buf *b;
4355
4356   // First queued buffer is the active request.
4357   acquire(&idelock);
4358   if((b = idequeue) == 0){
4359     release(&idelock);
4360     // cprintf("spurious IDE interrupt\n");
4361     return;
4362   }
4363   idequeue = b->qnext;
4364
4365   // Read data if needed.
4366   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4367     insl(0x1f0, b->data, BSIZE/4);
4368
4369   // Wake process waiting for this buf.
4370   b->flags |= B_VALID;
4371   b->flags &= ~B_DIRTY;
4372   wakeup(b);
4373
4374   // Start disk on next buf in queue.
4375   if(idequeue != 0)
4376     idestart(idequeue);
4377
4378   release(&idelock);
4379 }
4380
4381 // Sync buf with disk.
4382 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
4383 // Else if B_VALID is not set, read buf from disk, set B_VALID.
4384 void
4385 iderw(struct buf *b)
4386 {
4387   struct buf **pp;
4388
4389   if(!(b->flags & B_BUSY))
4390     panic("iderw: buf not busy");
4391   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
4392     panic("iderw: nothing to do");
4393   if(b->dev != 0 && !havedisk1)
4394     panic("iderw: ide disk 1 not present");
4395
4396   acquire(&idelock);
4397
4398
4399
```

```
4400   // Append b to idequeue.
4401   b->qnext = 0;
4402   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
4403     ;
4404   *pp = b;
4405
4406   // Start disk if necessary.
4407   if(idequeue == b)
4408     idestart(b);
4409
4410   // Wait for request to finish.
4411   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
4412     sleep(b, &idelock);
4413   }
4414
4415   release(&idelock);
4416 }
4417
4418
4419
4420
4421
4422
4423
4424
4425
4426
4427
4428
4429
4430
4431
4432
4433
4434
4435
4436
4437
4438
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
```



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

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



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

```
4600 #include "types.h"
4601 #include "defs.h"
4602 #include "param.h"
4603 #include "spinlock.h"
4604 #include "fs.h"
4605 #include "buf.h"
4606
4607 // Simple logging that allows concurrent FS system calls.
4608 //
4609 // A log transaction contains the updates of multiple FS system
4610 // calls. The logging system only commits when there are
4611 // no FS system calls active. Thus there is never
4612 // any reasoning required about whether a commit might
4613 // write an uncommitted system call's updates to disk.
4614 //
4615 // A system call should call begin_op()/end_op() to mark
4616 // its start and end. Usually begin_op() just increments
4617 // the count of in-progress FS system calls and returns.
4618 // But if it thinks the log is close to running out, it
4619 // sleeps until the last outstanding end_op() commits.
4620 //
4621 // The log is a physical re-do log containing disk blocks.
4622 // The on-disk log format:
4623 //   header block, containing block #s for block A, B, C, ...
4624 //   block A
4625 //   block B
4626 //   block C
4627 //   ...
4628 // Log appends are synchronous.
4629
4630 // Contents of the header block, used for both the on-disk header block
4631 // and to keep track in memory of logged block# before commit.
4632 struct logheader {
4633   int n;
4634   int block[LOGSIZE];
4635 };
4636
4637 struct log {
4638   struct spinlock lock;
4639   int start;
4640   int size;
4641   int outstanding; // how many FS sys calls are executing.
4642   int committing;  // in commit(), please wait.
4643   int dev;
4644   struct logheader lh;
4645 };
4646
4647
4648
4649
```



```
4650 struct log log;
4651
4652 static void recover_from_log(void);
4653 static void commit();
4654
4655 void
4656 initlog(int dev)
4657 {
4658   if (sizeof(struct logheader) >= BSIZE)
4659     panic("initlog: too big logheader");
4660
4661   struct superblock sb;
4662   initlock(&log.lock, "log");
4663   readsb(dev, &sb);
4664   log.start = sb.logstart;
4665   log.size = sb.nlog;
4666   log.dev = dev;
4667   recover_from_log();
4668 }
4669
4670 // Copy committed blocks from log to their home location
4671 static void
4672 install_trans(void)
4673 {
4674   int tail;
4675
4676   for (tail = 0; tail < log.lh.n; tail++) {
4677     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
4678     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
4679     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
4680     bwrite(dbuf);  // write dst to disk
4681     brelse(lbuf);
4682     brelse(dbuf);
4683   }
4684 }
4685
4686 // Read the log header from disk into the in-memory log header
4687 static void
4688 read_head(void)
4689 {
4690   struct buf *buf = bread(log.dev, log.start);
4691   struct logheader *lh = (struct logheader *) (buf->data);
4692   int i;
4693   log.lh.n = lh->n;
4694   for (i = 0; i < log.lh.n; i++) {
4695     log.lh.block[i] = lh->block[i];
4696   }
4697   brelse(buf);
4698 }
4699
```

```
4700 // Write in-memory log header to disk.
4701 // This is the true point at which the
4702 // current transaction commits.
4703 static void
4704 write_head(void)
4705 {
4706   struct buf *buf = bread(log.dev, log.start);
4707   struct logheader *hb = (struct logheader *) (buf->data);
4708   int i;
4709   hb->n = log.lh.n;
4710   for (i = 0; i < log.lh.n; i++) {
4711     hb->block[i] = log.lh.block[i];
4712   }
4713   bwrite(buf);
4714   brelse(buf);
4715 }
4716
4717 static void
4718 recover_from_log(void)
4719 {
4720   read_head();
4721   install_trans(); // if committed, copy from log to disk
4722   log.lh.n = 0;
4723   write_head(); // clear the log
4724 }
4725
4726 // called at the start of each FS system call.
4727 void
4728 begin_op(void)
4729 {
4730   acquire(&log.lock);
4731   while(1){
4732     if(log.committing){
4733       sleep(&log, &log.lock);
4734     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
4735       // this op might exhaust log space; wait for commit.
4736       sleep(&log, &log.lock);
4737     } else {
4738       log.outstanding += 1;
4739       release(&log.lock);
4740       break;
4741     }
4742   }
4743 }
4744
4745
4746
4747
4748
4749
```



```
4750 // called at the end of each FS system call.
4751 // commits if this was the last outstanding operation.
4752 void
4753 end_op(void)
4754 {
4755   int do_commit = 0;
4756
4757   acquire(&log.lock);
4758   log.outstanding -= 1;
4759   if(log.committing)
4760     panic("log.committing");
4761   if(log.outstanding == 0){
4762     do_commit = 1;
4763     log.committing = 1;
4764   } else {
4765     // begin_op() may be waiting for log space.
4766     wakeup(&log);
4767   }
4768   release(&log.lock);
4769
4770   if(do_commit){
4771     // call commit w/o holding locks, since not allowed
4772     // to sleep with locks.
4773     commit();
4774     acquire(&log.lock);
4775     log.committing = 0;
4776     wakeup(&log);
4777     release(&log.lock);
4778   }
4779 }
4780
4781 // Copy modified blocks from cache to log.
4782 static void
4783 write_log(void)
4784 {
4785   int tail;
4786
4787   for (tail = 0; tail < log.lh.n; tail++) {
4788     struct buf *to = bread(log.dev, log.start+tail+1); // log block
4789     struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
4790     memmove(to->data, from->data, BSIZE);
4791     bwrite(to);  // write the log
4792     brelse(from);
4793     brelse(to);
4794   }
4795 }
4796
4797
4798
4799
```

```
4800 static void
4801 commit()
4802 {
4803   if (log.lh.n > 0) {
4804     write_log();     // Write modified blocks from cache to log
4805     write_head();    // Write header to disk -- the real commit
4806     install_trans(); // Now install writes to home locations
4807     log.lh.n = 0;
4808     write_head();    // Erase the transaction from the log
4809   }
4810 }
4811
4812 // Caller has modified b->data and is done with the buffer.
4813 // Record the block number and pin in the cache with B_DIRTY.
4814 // commit()/write_log() will do the disk write.
4815 //
4816 // log_write() replaces bwrite(); a typical use is:
4817 //   bp = bread(...)
4818 //   modify bp->data[]
4819 //   log_write(bp)
4820 //   brelse(bp)
4821 void
4822 log_write(struct buf *b)
4823 {
4824   int i;
4825
4826   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
4827     panic("too big a transaction");
4828   if (log.outstanding < 1)
4829     panic("log_write outside of trans");
4830
4831   acquire(&log.lock);
4832   for (i = 0; i < log.lh.n; i++) {
4833     if (log.lh.block[i] == b->blockno)   // log absorbtion
4834       break;
4835   }
4836   log.lh.block[i] = b->blockno;
4837   if (i == log.lh.n)
4838     log.lh.n++;
4839   b->flags |= B_DIRTY; // prevent eviction
4840   release(&log.lock);
4841 }
4842
4843
4844
4845
4846
4847
4848
4849
```



```
4850 // File system implementation.  Five layers:
4851 //   + Blocks: allocator for raw disk blocks.
4852 //   + Log: crash recovery for multi-step updates.
4853 //   + Files: inode allocator, reading, writing, metadata.
4854 //   + Directories: inode with special contents (list of other inodes!)
4855 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
4856 //
4857 // This file contains the low-level file system manipulation
4858 // routines.  The (higher-level) system call implementations
4859 // are in sysfile.c.
4860
4861 #include "types.h"
4862 #include "defs.h"
4863 #include "param.h"
4864 #include "stat.h"
4865 #include "mmu.h"
4866 #include "proc.h"
4867 #include "spinlock.h"
4868 #include "fs.h"
4869 #include "buf.h"
4870 #include "file.h"
4871
4872 #define min(a, b) ((a) < (b) ? (a) : (b))
4873 static void itrunc(struct inode*);
4874 struct superblock sb;   // there should be one per dev, but we run with one d
4875
4876 // Read the super block.
4877 void
4878 readsb(int dev, struct superblock *sb)
4879 {
4880   struct buf *bp;
4881
4882   bp = bread(dev, 1);
4883   memmove(sb, bp->data, sizeof(*sb));
4884   brelse(bp);
4885 }
4886
4887 // Zero a block.
4888 static void
4889 bzero(int dev, int bno)
4890 {
4891   struct buf *bp;
4892
4893   bp = bread(dev, bno);
4894   memset(bp->data, 0, BSIZE);
4895   log_write(bp);
4896   brelse(bp);
4897 }
4898
4899
```

```
4900 // Blocks.
4901
4902 // Allocate a zeroed disk block.
4903 static uint
4904 balloc(uint dev)
4905 {
4906   int b, bi, m;
4907   struct buf *bp;
4908
4909   bp = 0;
4910   for(b = 0; b < sb.size; b += BPB){
4911     bp = bread(dev, BBLOCK(b, sb));
4912     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
4913       m = 1 << (bi % 8);
4914       if((bp->data[bi/8] & m) == 0){  // Is block free?
4915         bp->data[bi/8] |= m;  // Mark block in use.
4916         log_write(bp);
4917         brelse(bp);
4918         bzero(dev, b + bi);
4919         return b + bi;
4920       }
4921     }
4922     brelse(bp);
4923   }
4924   panic("balloc: out of blocks");
4925 }
4926
4927 // Free a disk block.
4928 static void
4929 bfree(int dev, uint b)
4930 {
4931   struct buf *bp;
4932   int bi, m;
4933
4934   readsb(dev, &sb);
4935   bp = bread(dev, BBLOCK(b, sb));
4936   bi = b % BPB;
4937   m = 1 << (bi % 8);
4938   if((bp->data[bi/8] & m) == 0)
4939     panic("freeing free block");
4940   bp->data[bi/8] &= ~m;
4941   log_write(bp);
4942   brelse(bp);
4943 }
4944
4945
4946
4947
4948
4949
```



```
4950 // Inodes.
4951 //
4952 // An inode describes a single unnamed file.
4953 // The inode disk structure holds metadata: the file's type,
4954 // its size, the number of links referring to it, and the
4955 // list of blocks holding the file's content.
4956 //
4957 // The inodes are laid out sequentially on disk at
4958 // sb.startinode. Each inode has a number, indicating its
4959 // position on the disk.
4960 //
4961 // The kernel keeps a cache of in-use inodes in memory
4962 // to provide a place for synchronizing access
4963 // to inodes used by multiple processes. The cached
4964 // inodes include book-keeping information that is
4965 // not stored on disk: ip->ref and ip->flags.
4966 //
4967 // An inode and its in-memory represtative go through a
4968 // sequence of states before they can be used by the
4969 // rest of the file system code.
4970 //
4971 // * Allocation: an inode is allocated if its type (on disk)
4972 //   is non-zero. ialloc() allocates, iput() frees if
4973 //   the link count has fallen to zero.
4974 //
4975 // * Referencing in cache: an entry in the inode cache
4976 //   is free if ip->ref is zero. Otherwise ip->ref tracks
4977 //   the number of in-memory pointers to the entry (open
4978 //   files and current directories). iget() to find or
4979 //   create a cache entry and increment its ref, iput()
4980 //   to decrement ref.
4981 //
4982 // * Valid: the information (type, size, &c) in an inode
4983 //   cache entry is only correct when the I_VALID bit
4984 //   is set in ip->flags. ilock() reads the inode from
4985 //   the disk and sets I_VALID, while iput() clears
4986 //   I_VALID if ip->ref has fallen to zero.
4987 //
4988 // * Locked: file system code may only examine and modify
4989 //   the information in an inode and its content if it
4990 //   has first locked the inode. The I_BUSY flag indicates
4991 //   that the inode is locked. ilock() sets I_BUSY,
4992 //   while iunlock clears it.
4993 //
4994 // Thus a typical sequence is:
4995 //   ip = iget(dev, inum)
4996 //   ilock(ip)
4997 //   ... examine and modify ip->xxx ...
4998 //   iunlock(ip)
4999 //   iput(ip)
```

```
5000 //
5001 // ilock() is separate from iget() so that system calls can
5002 // get a long-term reference to an inode (as for an open file)
5003 // and only lock it for short periods (e.g., in read()).
5004 // The separation also helps avoid deadlock and races during
5005 // pathname lookup. iget() increments ip->ref so that the inode
5006 // stays cached and pointers to it remain valid.
5007 //
5008 // Many internal file system functions expect the caller to
5009 // have locked the inodes involved; this lets callers create
5010 // multi-step atomic operations.
5011
5012 struct {
5013   struct spinlock lock;
5014   struct inode inode[NINODE];
5015 } icache;
5016
5017 void
5018 iinit(int dev)
5019 {
5020   initlock(&icache.lock, "icache");
5021   readsb(dev, &sb);
5022   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d inodestart %
5023           sb.nblocks, sb.ninodes, sb.nlog, sb.logstart, sb.inodestart, sb.bma
5024 }
5025
5026 static struct inode* iget(uint dev, uint inum);
5027
5028 // Allocate a new inode with the given type on device dev.
5029 // A free inode has a type of zero.
5030 struct inode*
5031 ialloc(uint dev, short type)
5032 {
5033   int inum;
5034   struct buf *bp;
5035   struct dinode *dip;
5036
5037   for(inum = 1; inum < sb.ninodes; inum++){
5038     bp = bread(dev, IBLOCK(inum, sb));
5039     dip = (struct dinode*)bp->data + inum%IPB;
5040     if(dip->type == 0){  // a free inode
5041       memset(dip, 0, sizeof(*dip));
5042       dip->type = type;
5043       log_write(bp);   // mark it allocated on the disk
5044       brelse(bp);
5045       return iget(dev, inum);
5046     }
5047     brelse(bp);
5048   }
5049   panic("ialloc: no inodes");
```



```
5050 }
5051
5052 // Copy a modified in-memory inode to disk.
5053 void
5054 iupdate(struct inode *ip)
5055 {
5056   struct buf *bp;
5057   struct dinode *dip;
5058
5059   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5060   dip = (struct dinode*)bp->data + ip->inum%IPB;
5061   dip->type = ip->type;
5062   dip->major = ip->major;
5063   dip->minor = ip->minor;
5064   dip->nlink = ip->nlink;
5065   dip->size = ip->size;
5066   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5067   log_write(bp);
5068   brelse(bp);
5069 }
5070
5071 // Find the inode with number inum on device dev
5072 // and return the in-memory copy. Does not lock
5073 // the inode and does not read it from disk.
5074 static struct inode*
5075 iget(uint dev, uint inum)
5076 {
5077   struct inode *ip, *empty;
5078
5079   acquire(&icache.lock);
5080
5081   // Is the inode already cached?
5082   empty = 0;
5083   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5084     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5085       ip->ref++;
5086       release(&icache.lock);
5087       return ip;
5088     }
5089     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
5090       empty = ip;
5091   }
5092
5093   // Recycle an inode cache entry.
5094   if(empty == 0)
5095     panic("iget: no inodes");
5096
5097
5098
5099
```

```
5100   ip = empty;
5101   ip->dev = dev;
5102   ip->inum = inum;
5103   ip->ref = 1;
5104   ip->flags = 0;
5105   release(&icache.lock);
5106
5107   return ip;
5108 }
5109
5110 // Increment reference count for ip.
5111 // Returns ip to enable ip = idup(ip1) idiom.
5112 struct inode*
5113 idup(struct inode *ip)
5114 {
5115   acquire(&icache.lock);
5116   ip->ref++;
5117   release(&icache.lock);
5118   return ip;
5119 }
5120
5121 // Lock the given inode.
5122 // Reads the inode from disk if necessary.
5123 void
5124 ilock(struct inode *ip)
5125 {
5126   struct buf *bp;
5127   struct dinode *dip;
5128
5129   if(ip == 0 || ip->ref < 1)
5130     panic("ilock");
5131
5132   acquire(&icache.lock);
5133   while(ip->flags & I_BUSY)
5134     sleep(ip, &icache.lock);
5135   ip->flags |= I_BUSY;
5136   release(&icache.lock);
5137
5138   if(!(ip->flags & I_VALID)){
5139     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5140     dip = (struct dinode*)bp->data + ip->inum%IPB;
5141     ip->type = dip->type;
5142     ip->major = dip->major;
5143     ip->minor = dip->minor;
5144     ip->nlink = dip->nlink;
5145     ip->size = dip->size;
5146     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5147     brelse(bp);
5148     ip->flags |= I_VALID;
5149     if(ip->type == 0)
```



```
5150       panic("ilock: no type");
5151   }
5152 }
5153
5154 // Unlock the given inode.
5155 void
5156 iunlock(struct inode *ip)
5157 {
5158   if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
5159     panic("iunlock");
5160
5161   acquire(&icache.lock);
5162   ip->flags &= ~I_BUSY;
5163   wakeup(ip);
5164   release(&icache.lock);
5165 }
5166
5167 // Drop a reference to an in-memory inode.
5168 // If that was the last reference, the inode cache entry can
5169 // be recycled.
5170 // If that was the last reference and the inode has no links
5171 // to it, free the inode (and its content) on disk.
5172 // All calls to iput() must be inside a transaction in
5173 // case it has to free the inode.
5174 void
5175 iput(struct inode *ip)
5176 {
5177   acquire(&icache.lock);
5178   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
5179     // inode has no links and no other references: truncate and free.
5180     if(ip->flags & I_BUSY)
5181       panic("iput busy");
5182     ip->flags |= I_BUSY;
5183     release(&icache.lock);
5184     itrunc(ip);
5185     ip->type = 0;
5186     iupdate(ip);
5187     acquire(&icache.lock);
5188     ip->flags = 0;
5189     wakeup(ip);
5190   }
5191   ip->ref--;
5192   release(&icache.lock);
5193 }
5194
5195
5196
5197
5198
5199
```

```
5200 // Common idiom: unlock, then put.
5201 void
5202 iunlockput(struct inode *ip)
5203 {
5204   iunlock(ip);
5205   iput(ip);
5206 }
5207
5208 // Inode content
5209 //
5210 // The content (data) associated with each inode is stored
5211 // in blocks on the disk. The first NDIRECT block numbers
5212 // are listed in ip->addrs[].  The next NINDIRECT blocks are
5213 // listed in block ip->addrs[NDIRECT].
5214
5215 // Return the disk block address of the nth block in inode ip.
5216 // If there is no such block, bmap allocates one.
5217 static uint
5218 bmap(struct inode *ip, uint bn)
5219 {
5220   uint addr, *a;
5221   struct buf *bp;
5222
5223   if(bn < NDIRECT){
5224     if((addr = ip->addrs[bn]) == 0)
5225       ip->addrs[bn] = addr = balloc(ip->dev);
5226     return addr;
5227   }
5228   bn -= NDIRECT;
5229
5230   if(bn < NINDIRECT){
5231     // Load indirect block, allocating if necessary.
5232     if((addr = ip->addrs[NDIRECT]) == 0)
5233       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5234     bp = bread(ip->dev, addr);
5235     a = (uint*)bp->data;
5236     if((addr = a[bn]) == 0){
5237       a[bn] = addr = balloc(ip->dev);
5238       log_write(bp);
5239     }
5240     brelse(bp);
5241     return addr;
5242   }
5243
5244   panic("bmap: out of range");
5245 }
5246
5247
5248
5249
```



```
5250 // Truncate inode (discard contents).
5251 // Only called when the inode has no links
5252 // to it (no directory entries referring to it)
5253 // and has no in-memory reference to it (is
5254 // not an open file or current directory).
5255 static void
5256 itrunc(struct inode *ip)
5257 {
5258   int i, j;
5259   struct buf *bp;
5260   uint *a;
5261
5262   for(i = 0; i < NDIRECT; i++){
5263     if(ip->addrs[i]){
5264       bfree(ip->dev, ip->addrs[i]);
5265       ip->addrs[i] = 0;
5266     }
5267   }
5268
5269   if(ip->addrs[NDIRECT]){
5270     bp = bread(ip->dev, ip->addrs[NDIRECT]);
5271     a = (uint*)bp->data;
5272     for(j = 0; j < NINDIRECT; j++){
5273       if(a[j])
5274         bfree(ip->dev, a[j]);
5275     }
5276     brelse(bp);
5277     bfree(ip->dev, ip->addrs[NDIRECT]);
5278     ip->addrs[NDIRECT] = 0;
5279   }
5280
5281   ip->size = 0;
5282   iupdate(ip);
5283 }
5284
5285 // Copy stat information from inode.
5286 void
5287 stati(struct inode *ip, struct stat *st)
5288 {
5289   st->dev = ip->dev;
5290   st->ino = ip->inum;
5291   st->type = ip->type;
5292   st->nlink = ip->nlink;
5293   st->size = ip->size;
5294 }
5295
5296
5297
5298
5299
```

```
5300 // Read data from inode.
5301 int
5302 readi(struct inode *ip, char *dst, uint off, uint n)
5303 {
5304   uint tot, m;
5305   struct buf *bp;
5306
5307   if(ip->type == T_DEV){
5308     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
5309       return -1;
5310     return devsw[ip->major].read(ip, dst, n);
5311   }
5312
5313   if(off > ip->size || off + n < off)
5314     return -1;
5315   if(off + n > ip->size)
5316     n = ip->size - off;
5317
5318   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5319     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5320     m = min(n - tot, BSIZE - off%BSIZE);
5321     memmove(dst, bp->data + off%BSIZE, m);
5322     brelse(bp);
5323   }
5324   return n;
5325 }
5326
5327 // Write data to inode.
5328 int
5329 writei(struct inode *ip, char *src, uint off, uint n)
5330 {
5331   uint tot, m;
5332   struct buf *bp;
5333
5334   if(ip->type == T_DEV){
5335     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
5336       return -1;
5337     return devsw[ip->major].write(ip, src, n);
5338   }
5339
5340   if(off > ip->size || off + n < off)
5341     return -1;
5342   if(off + n > MAXFILE*BSIZE)
5343     return -1;
5344
5345   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
5346     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5347     m = min(n - tot, BSIZE - off%BSIZE);
5348     memmove(bp->data + off%BSIZE, src, m);
5349     log_write(bp);
```



```
5350     brelse(bp);
5351   }
5352
5353   if(n > 0 && off > ip->size){
5354     ip->size = off;
5355     iupdate(ip);
5356   }
5357   return n;
5358 }
5359
5360 // Directories
5361
5362 int
5363 namecmp(const char *s, const char *t)
5364 {
5365   return strncmp(s, t, DIRSIZ);
5366 }
5367
5368 // Look for a directory entry in a directory.
5369 // If found, set *poff to byte offset of entry.
5370 struct inode*
5371 dirlookup(struct inode *dp, char *name, uint *poff)
5372 {
5373   uint off, inum;
5374   struct dirent de;
5375
5376   if(dp->type != T_DIR)
5377     panic("dirlookup not DIR");
5378
5379   for(off = 0; off < dp->size; off += sizeof(de)){
5380     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5381       panic("dirlink read");
5382     if(de.inum == 0)
5383       continue;
5384     if(namecmp(name, de.name) == 0){
5385       // entry matches path element
5386       if(poff)
5387         *poff = off;
5388       inum = de.inum;
5389       return iget(dp->dev, inum);
5390     }
5391   }
5392
5393   return 0;
5394 }
5395
5396
5397
5398
5399
```

```
5400 // Write a new directory entry (name, inum) into the directory dp.
5401 int
5402 dirlink(struct inode *dp, char *name, uint inum)
5403 {
5404   int off;
5405   struct dirent de;
5406   struct inode *ip;
5407
5408   // Check that name is not present.
5409   if((ip = dirlookup(dp, name, 0)) != 0){
5410     iput(ip);
5411     return -1;
5412   }
5413
5414   // Look for an empty dirent.
5415   for(off = 0; off < dp->size; off += sizeof(de)){
5416     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5417       panic("dirlink read");
5418     if(de.inum == 0)
5419       break;
5420   }
5421
5422   strncpy(de.name, name, DIRSIZ);
5423   de.inum = inum;
5424   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5425     panic("dirlink");
5426
5427   return 0;
5428 }
5429
5430 // Paths
5431
5432 // Copy the next path element from path into name.
5433 // Return a pointer to the element following the copied one.
5434 // The returned path has no leading slashes,
5435 // so the caller can check *path=='\0' to see if the name is the last one.
5436 // If no name to remove, return 0.
5437 //
5438 // Examples:
5439 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5440 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5441 //   skipelem("a", name) = "", setting name = "a"
5442 //   skipelem("", name) = skipelem("////", name) = 0
5443 //
5444 static char*
5445 skipelem(char *path, char *name)
5446 {
5447   char *s;
5448   int len;
5449
```



```
5450   while(*path == '/')
5451     path++;
5452   if(*path == 0)
5453     return 0;
5454   s = path;
5455   while(*path != '/' && *path != 0)
5456     path++;
5457   len = path - s;
5458   if(len >= DIRSIZ)
5459     memmove(name, s, DIRSIZ);
5460   else {
5461     memmove(name, s, len);
5462     name[len] = 0;
5463   }
5464   while(*path == '/')
5465     path++;
5466   return path;
5467 }
5468
5469 // Look up and return the inode for a path name.
5470 // If parent != 0, return the inode for the parent and copy the final
5471 // path element into name, which must have room for DIRSIZ bytes.
5472 // Must be called inside a transaction since it calls iput().
5473 static struct inode*
5474 namex(char *path, int nameiparent, char *name)
5475 {
5476   struct inode *ip, *next;
5477
5478   if(*path == '/')
5479     ip = iget(ROOTDEV, ROOTINO);
5480   else
5481     ip = idup(proc->cwd);
5482
5483   while((path = skipelem(path, name)) != 0){
5484     ilock(ip);
5485     if(ip->type != T_DIR){
5486       iunlockput(ip);
5487       return 0;
5488     }
5489     if(nameiparent && *path == '\0'){
5490       // Stop one level early.
5491       iunlock(ip);
5492       return ip;
5493     }
5494     if((next = dirlookup(ip, name, 0)) == 0){
5495       iunlockput(ip);
5496       return 0;
5497     }
5498     iunlockput(ip);
5499     ip = next;
```

```
5500   }
5501   if(nameiparent){
5502     iput(ip);
5503     return 0;
5504   }
5505   return ip;
5506 }
5507
5508 struct inode*
5509 namei(char *path)
5510 {
5511   char name[DIRSIZ];
5512   return namex(path, 0, name);
5513 }
5514
5515 struct inode*
5516 nameiparent(char *path, char *name)
5517 {
5518   return namex(path, 1, name);
5519 }
5520
5521
5522
5523
5524
5525
5526
5527
5528
5529
5530
5531
5532
5533
5534
5535
5536
5537
5538
5539
5540
5541
5542
5543
5544
5545
5546
5547
5548
5549
```



```
5550 //
5551 // File descriptors
5552 //
5553
5554 #include "types.h"
5555 #include "defs.h"
5556 #include "param.h"
5557 #include "fs.h"
5558 #include "file.h"
5559 #include "spinlock.h"
5560
5561 struct devsw devsw[NDEV];
5562 struct {
5563   struct spinlock lock;
5564   struct file file[NFILE];
5565 } ftable;
5566
5567 void
5568 fileinit(void)
5569 {
5570   initlock(&ftable.lock, "ftable");
5571 }
5572
5573 // Allocate a file structure.
5574 struct file*
5575 filealloc(void)
5576 {
5577   struct file *f;
5578
5579   acquire(&ftable.lock);
5580   for(f = ftable.file; f < ftable.file + NFILE; f++){
5581     if(f->ref == 0){
5582       f->ref = 1;
5583       release(&ftable.lock);
5584       return f;
5585     }
5586   }
5587   release(&ftable.lock);
5588   return 0;
5589 }
5590
5591
5592
5593
5594
5595
5596
5597
5598
5599
```

```
5600 // Increment ref count for file f.
5601 struct file*
5602 filedup(struct file *f)
5603 {
5604   acquire(&ftable.lock);
5605   if(f->ref < 1)
5606     panic("filedup");
5607   f->ref++;
5608   release(&ftable.lock);
5609   return f;
5610 }
5611
5612 // Close file f.  (Decrement ref count, close when reaches 0.)
5613 void
5614 fileclose(struct file *f)
5615 {
5616   struct file ff;
5617
5618   acquire(&ftable.lock);
5619   if(f->ref < 1)
5620     panic("fileclose");
5621   if(--f->ref > 0){
5622     release(&ftable.lock);
5623     return;
5624   }
5625   ff = *f;
5626   f->ref = 0;
5627   f->type = FD_NONE;
5628   release(&ftable.lock);
5629
5630   if(ff.type == FD_PIPE)
5631     pipeclose(ff.pipe, ff.writable);
5632   else if(ff.type == FD_INODE){
5633     begin_op();
5634     iput(ff.ip);
5635     end_op();
5636   }
5637 }
5638
5639
5640
5641
5642
5643
5644
5645
5646
5647
5648
5649
```



```
5650 // Get metadata about file f.
5651 int
5652 filestat(struct file *f, struct stat *st)
5653 {
5654   if(f->type == FD_INODE){
5655     ilock(f->ip);
5656     stati(f->ip, st);
5657     iunlock(f->ip);
5658     return 0;
5659   }
5660   return -1;
5661 }
5662
5663 // Read from file f.
5664 int
5665 fileread(struct file *f, char *addr, int n)
5666 {
5667   int r;
5668
5669   if(f->readable == 0)
5670     return -1;
5671   if(f->type == FD_PIPE)
5672     return piperead(f->pipe, addr, n);
5673   if(f->type == FD_INODE){
5674     ilock(f->ip);
5675     if((r = readi(f->ip, addr, f->off, n)) > 0)
5676       f->off += r;
5677     iunlock(f->ip);
5678     return r;
5679   }
5680   panic("fileread");
5681 }
5682
5683 // Write to file f.
5684 int
5685 filewrite(struct file *f, char *addr, int n)
5686 {
5687   int r;
5688
5689   if(f->writable == 0)
5690     return -1;
5691   if(f->type == FD_PIPE)
5692     return pipewrite(f->pipe, addr, n);
5693   if(f->type == FD_INODE){
5694     // write a few blocks at a time to avoid exceeding
5695     // the maximum log transaction size, including
5696     // i-node, indirect block, allocation blocks,
5697     // and 2 blocks of slop for non-aligned writes.
5698     // this really belongs lower down, since writei()
5699     // might be writing a device like the console.
```

```
5700     int max = ((LOGSIZE-1-1-2) / 2) * 512;
5701     int i = 0;
5702     while(i < n){
5703       int n1 = n - i;
5704       if(n1 > max)
5705         n1 = max;
5706
5707       begin_op();
5708       ilock(f->ip);
5709       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
5710         f->off += r;
5711       iunlock(f->ip);
5712       end_op();
5713
5714       if(r < 0)
5715         break;
5716       if(r != n1)
5717         panic("short filewrite");
5718       i += r;
5719     }
5720     return i == n ? n : -1;
5721   }
5722   panic("filewrite");
5723 }
5724
5725
5726
5727
5728
5729
5730
5731
5732
5733
5734
5735
5736
5737
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749
```



```
5750 //
5751 // File-system system calls.
5752 // Mostly argument checking, since we don't trust
5753 // user code, and calls into file.c and fs.c.
5754 //
5755
5756 #include "types.h"
5757 #include "defs.h"
5758 #include "param.h"
5759 #include "stat.h"
5760 #include "mmu.h"
5761 #include "proc.h"
5762 #include "fs.h"
5763 #include "file.h"
5764 #include "fcntl.h"
5765
5766 // Fetch the nth word-sized system call argument as a file descriptor
5767 // and return both the descriptor and the corresponding struct file.
5768 static int
5769 argfd(int n, int *pfd, struct file **pf)
5770 {
5771   int fd;
5772   struct file *f;
5773
5774   if(argint(n, &fd) < 0)
5775     return -1;
5776   if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
5777     return -1;
5778   if(pfd)
5779     *pfd = fd;
5780   if(pf)
5781     *pf = f;
5782   return 0;
5783 }
5784
5785 // Allocate a file descriptor for the given file.
5786 // Takes over file reference from caller on success.
5787 static int
5788 fdalloc(struct file *f)
5789 {
5790   int fd;
5791
5792   for(fd = 0; fd < NOFILE; fd++){
5793     if(proc->ofile[fd] == 0){
5794       proc->ofile[fd] = f;
5795       return fd;
5796     }
5797   }
5798   return -1;
5799 }
```

```
5800 int
5801 sys_dup(void)
5802 {
5803   struct file *f;
5804   int fd;
5805
5806   if(argfd(0, 0, &f) < 0)
5807     return -1;
5808   if((fd=fdalloc(f)) < 0)
5809     return -1;
5810   filedup(f);
5811   return fd;
5812 }
5813
5814 int
5815 sys_read(void)
5816 {
5817   struct file *f;
5818   int n;
5819   char *p;
5820
5821   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5822     return -1;
5823   return fileread(f, p, n);
5824 }
5825
5826 int
5827 sys_write(void)
5828 {
5829   struct file *f;
5830   int n;
5831   char *p;
5832
5833   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5834     return -1;
5835   return filewrite(f, p, n);
5836 }
5837
5838 int
5839 sys_close(void)
5840 {
5841   int fd;
5842   struct file *f;
5843
5844   if(argfd(0, &fd, &f) < 0)
5845     return -1;
5846   proc->ofile[fd] = 0;
5847   fileclose(f);
5848   return 0;
5849 }
```



```
5850 int
5851 sys_fstat(void)
5852 {
5853   struct file *f;
5854   struct stat *st;
5855
5856   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
5857     return -1;
5858   return filestat(f, st);
5859 }
5860
5861 // Create the path new as a link to the same inode as old.
5862 int
5863 sys_link(void)
5864 {
5865   char name[DIRSIZ], *new, *old;
5866   struct inode *dp, *ip;
5867
5868   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
5869     return -1;
5870
5871   begin_op();
5872   if((ip = namei(old)) == 0){
5873     end_op();
5874     return -1;
5875   }
5876
5877   ilock(ip);
5878   if(ip->type == T_DIR){
5879     iunlockput(ip);
5880     end_op();
5881     return -1;
5882   }
5883
5884   ip->nlink++;
5885   iupdate(ip);
5886   iunlock(ip);
5887
5888   if((dp = nameiparent(new, name)) == 0)
5889     goto bad;
5890   ilock(dp);
5891   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
5892     iunlockput(dp);
5893     goto bad;
5894   }
5895   iunlockput(dp);
5896   iput(ip);
5897
5898   end_op();
5899
```

```
5900   return 0;
5901
5902 bad:
5903   ilock(ip);
5904   ip->nlink--;
5905   iupdate(ip);
5906   iunlockput(ip);
5907   end_op();
5908   return -1;
5909 }
5910
5911 // Is the directory dp empty except for "." and ".." ?
5912 static int
5913 isdirempty(struct inode *dp)
5914 {
5915   int off;
5916   struct dirent de;
5917
5918   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
5919     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5920       panic("isdirempty: readi");
5921     if(de.inum != 0)
5922       return 0;
5923   }
5924   return 1;
5925 }
5926
5927 int
5928 sys_unlink(void)
5929 {
5930   struct inode *ip, *dp;
5931   struct dirent de;
5932   char name[DIRSIZ], *path;
5933   uint off;
5934
5935   if(argstr(0, &path) < 0)
5936     return -1;
5937
5938   begin_op();
5939   if((dp = nameiparent(path, name)) == 0){
5940     end_op();
5941     return -1;
5942   }
5943
5944   ilock(dp);
5945
5946   // Cannot unlink "." or "..".
5947   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
5948     goto bad;
5949
```



```
5950   if((ip = dirlookup(dp, name, &off)) == 0)
5951     goto bad;
5952   ilock(ip);
5953
5954   if(ip->nlink < 1)
5955     panic("unlink: nlink < 1");
5956   if(ip->type == T_DIR && !isdirempty(ip)){
5957     iunlockput(ip);
5958     goto bad;
5959   }
5960
5961   memset(&de, 0, sizeof(de));
5962   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5963     panic("unlink: writei");
5964   if(ip->type == T_DIR){
5965     dp->nlink--;
5966     iupdate(dp);
5967   }
5968   iunlockput(dp);
5969
5970   ip->nlink--;
5971   iupdate(ip);
5972   iunlockput(ip);
5973
5974   end_op();
5975
5976   return 0;
5977
5978 bad:
5979   iunlockput(dp);
5980   end_op();
5981   return -1;
5982 }
5983
5984 static struct inode*
5985 create(char *path, short type, short major, short minor)
5986 {
5987   uint off;
5988   struct inode *ip, *dp;
5989   char name[DIRSIZ];
5990
5991   if((dp = nameiparent(path, name)) == 0)
5992     return 0;
5993   ilock(dp);
5994
5995   if((ip = dirlookup(dp, name, &off)) != 0){
5996     iunlockput(dp);
5997     ilock(ip);
5998     if(type == T_FILE && ip->type == T_FILE)
5999       return ip;
```

```
6000     iunlockput(ip);
6001     return 0;
6002   }
6003
6004   if((ip = ialloc(dp->dev, type)) == 0)
6005     panic("create: ialloc");
6006
6007   ilock(ip);
6008   ip->major = major;
6009   ip->minor = minor;
6010   ip->nlink = 1;
6011   iupdate(ip);
6012
6013   if(type == T_DIR){  // Create . and .. entries.
6014     dp->nlink++;  // for ".."
6015     iupdate(dp);
6016     // No ip->nlink++ for ".": avoid cyclic ref count.
6017     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6018       panic("create dots");
6019   }
6020
6021   if(dirlink(dp, name, ip->inum) < 0)
6022     panic("create: dirlink");
6023
6024   iunlockput(dp);
6025
6026   return ip;
6027 }
6028
6029 int
6030 sys_open(void)
6031 {
6032   char *path;
6033   int fd, omode;
6034   struct file *f;
6035   struct inode *ip;
6036
6037   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6038     return -1;
6039
6040   begin_op();
6041
6042   if(omode & O_CREATE){
6043     ip = create(path, T_FILE, 0, 0);
6044     if(ip == 0){
6045       end_op();
6046       return -1;
6047     }
6048   } else {
6049     if((ip = namei(path)) == 0){
```



```
6050       end_op();
6051       return -1;
6052     }
6053     ilock(ip);
6054     if(ip->type == T_DIR && omode != O_RDONLY){
6055       iunlockput(ip);
6056       end_op();
6057       return -1;
6058     }
6059   }
6060
6061   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6062     if(f)
6063       fileclose(f);
6064     iunlockput(ip);
6065     end_op();
6066     return -1;
6067   }
6068   iunlock(ip);
6069   end_op();
6070
6071   f->type = FD_INODE;
6072   f->ip = ip;
6073   f->off = 0;
6074   f->readable = !(omode & O_WRONLY);
6075   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6076   return fd;
6077 }
6078
6079 int
6080 sys_mkdir(void)
6081 {
6082   char *path;
6083   struct inode *ip;
6084
6085   begin_op();
6086   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
6087     end_op();
6088     return -1;
6089   }
6090   iunlockput(ip);
6091   end_op();
6092   return 0;
6093 }
6094
6095
6096
6097
6098
6099
```

```
6100 int
6101 sys_mknod(void)
6102 {
6103   struct inode *ip;
6104   char *path;
6105   int len;
6106   int major, minor;
6107
6108   begin_op();
6109   if((len=argstr(0, &path)) < 0 ||
6110      argint(1, &major) < 0 ||
6111      argint(2, &minor) < 0 ||
6112      (ip = create(path, T_DEV, major, minor)) == 0){
6113     end_op();
6114     return -1;
6115   }
6116   iunlockput(ip);
6117   end_op();
6118   return 0;
6119 }
6120
6121 int
6122 sys_chdir(void)
6123 {
6124   char *path;
6125   struct inode *ip;
6126
6127   begin_op();
6128   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
6129     end_op();
6130     return -1;
6131   }
6132   ilock(ip);
6133   if(ip->type != T_DIR){
6134     iunlockput(ip);
6135     end_op();
6136     return -1;
6137   }
6138   iunlock(ip);
6139   iput(proc->cwd);
6140   end_op();
6141   proc->cwd = ip;
6142   return 0;
6143 }
6144
6145
6146
6147
6148
6149
```



```
6150 int
6151 sys_exec(void)
6152 {
6153   char *path, *argv[MAXARG];
6154   int i;
6155   uint uargv, uarg;
6156
6157   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6158     return -1;
6159   }
6160   memset(argv, 0, sizeof(argv));
6161   for(i=0;; i++){
6162     if(i >= NELEM(argv))
6163       return -1;
6164     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6165       return -1;
6166     if(uarg == 0){
6167       argv[i] = 0;
6168       break;
6169     }
6170     if(fetchstr(uarg, &argv[i]) < 0)
6171       return -1;
6172   }
6173   return exec(path, argv);
6174 }
6175
6176 int
6177 sys_pipe(void)
6178 {
6179   int *fd;
6180   struct file *rf, *wf;
6181   int fd0, fd1;
6182
6183   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6184     return -1;
6185   if(pipealloc(&rf, &wf) < 0)
6186     return -1;
6187   fd0 = -1;
6188   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6189     if(fd0 >= 0)
6190       proc->ofile[fd0] = 0;
6191     fileclose(rf);
6192     fileclose(wf);
6193     return -1;
6194   }
6195   fd[0] = fd0;
6196   fd[1] = fd1;
6197   return 0;
6198 }
6199
```

```
6200 #include "types.h"
6201 #include "param.h"
6202 #include "memlayout.h"
6203 #include "mmu.h"
6204 #include "proc.h"
6205 #include "defs.h"
6206 #include "x86.h"
6207 #include "elf.h"
6208
6209 int
6210 exec(char *path, char **argv)
6211 {
6212   char *s, *last;
6213   int i, off;
6214   uint argc, sz, sp, ustack[3+MAXARG+1];
6215   struct elfhdr elf;
6216   struct inode *ip;
6217   struct proghdr ph;
6218   pde_t *pgdir, *oldpgdir;
6219
6220   begin_op();
6221   if((ip = namei(path)) == 0){
6222     end_op();
6223     return -1;
6224   }
6225   ilock(ip);
6226   pgdir = 0;
6227
6228   // Check ELF header
6229   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
6230     goto bad;
6231   if(elf.magic != ELF_MAGIC)
6232     goto bad;
6233
6234   if((pgdir = setupkvm()) == 0)
6235     goto bad;
6236
6237   // Load program into memory.
6238   sz = 0;
6239   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6240     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6241       goto bad;
6242     if(ph.type != ELF_PROG_LOAD)
6243       continue;
6244     if(ph.memsz < ph.filesz)
6245       goto bad;
6246     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6247       goto bad;
6248     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6249       goto bad;
```



```
6250   }
6251   iunlockput(ip);
6252   end_op();
6253   ip = 0;
6254
6255   // Allocate two pages at the next page boundary.
6256   // Make the first inaccessible.  Use the second as the user stack.
6257   sz = PGROUNDUP(sz);
6258   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
6259     goto bad;
6260   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6261   sp = sz;
6262
6263   // Push argument strings, prepare rest of stack in ustack.
6264   for(argc = 0; argv[argc]; argc++) {
6265     if(argc >= MAXARG)
6266       goto bad;
6267     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6268     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6269       goto bad;
6270     ustack[3+argc] = sp;
6271   }
6272   ustack[3+argc] = 0;
6273
6274   ustack[0] = 0xffffffff;  // fake return PC
6275   ustack[1] = argc;
6276   ustack[2] = sp - (argc+1)*4;  // argv pointer
6277
6278   sp -= (3+argc+1) * 4;
6279   if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6280     goto bad;
6281
6282   // Save program name for debugging.
6283   for(last=s=path; *s; s++)
6284     if(*s == '/')
6285       last = s+1;
6286   safestrcpy(proc->name, last, sizeof(proc->name));
6287
6288   // Commit to the user image.
6289   oldpgdir = proc->pgdir;
6290   proc->pgdir = pgdir;
6291   proc->sz = sz;
6292   proc->tf->eip = elf.entry;  // main
6293   proc->tf->esp = sp;
6294   switchuvm(proc);
6295   freevm(oldpgdir);
6296   return 0;
6297
6298
6299
```

```
6300  bad:
6301    if(pgdir)
6302      freevm(pgdir);
6303    if(ip){
6304      iunlockput(ip);
6305      end_op();
6306    }
6307    return -1;
6308  }
6309
6310
6311
6312
6313
6314
6315
6316
6317
6318
6319
6320
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 "defs.h"
6352 #include "param.h"
6353 #include "mmu.h"
6354 #include "proc.h"
6355 #include "fs.h"
6356 #include "file.h"
6357 #include "spinlock.h"
6358
6359 #define PIPESIZE 512
6360
6361 struct pipe {
6362   struct spinlock lock;
6363   char data[PIPESIZE];
6364   uint nread;     // number of bytes read
6365   uint nwrite;    // number of bytes written
6366   int readopen;   // read fd is still open
6367   int writeopen;  // write fd is still open
6368 };
6369
6370 int
6371 pipealloc(struct file **f0, struct file **f1)
6372 {
6373   struct pipe *p;
6374
6375   p = 0;
6376   *f0 = *f1 = 0;
6377   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
6378     goto bad;
6379   if((p = (struct pipe*)kalloc()) == 0)
6380     goto bad;
6381   p->readopen = 1;
6382   p->writeopen = 1;
6383   p->nwrite = 0;
6384   p->nread = 0;
6385   initlock(&p->lock, "pipe");
6386   (*f0)->type = FD_PIPE;
6387   (*f0)->readable = 1;
6388   (*f0)->writable = 0;
6389   (*f0)->pipe = p;
6390   (*f1)->type = FD_PIPE;
6391   (*f1)->readable = 0;
6392   (*f1)->writable = 1;
6393   (*f1)->pipe = p;
6394   return 0;
6395
6396
6397
6398
6399
```

```
6400  bad:
6401    if(p)
6402      kfree((char*)p);
6403    if(*f0)
6404      fileclose(*f0);
6405    if(*f1)
6406      fileclose(*f1);
6407    return -1;
6408  }
6409
6410  void
6411  pipeclose(struct pipe *p, int writable)
6412  {
6413    acquire(&p->lock);
6414    if(writable){
6415      p->writeopen = 0;
6416      wakeup(&p->nread);
6417    } else {
6418      p->readopen = 0;
6419      wakeup(&p->nwrite);
6420    }
6421    if(p->readopen == 0 && p->writeopen == 0){
6422      release(&p->lock);
6423      kfree((char*)p);
6424    } else
6425      release(&p->lock);
6426  }
6427
6428  int
6429  pipewrite(struct pipe *p, char *addr, int n)
6430  {
6431    int i;
6432
6433    acquire(&p->lock);
6434    for(i = 0; i < n; i++){
6435      while(p->nwrite == p->nread + PIPESIZE){
6436        if(p->readopen == 0 || proc->killed){
6437          release(&p->lock);
6438          return -1;
6439        }
6440        wakeup(&p->nread);
6441        sleep(&p->nwrite, &p->lock);
6442      }
6443      p->data[p->nwrite++ % PIPESIZE] = addr[i];
6444    }
6445    wakeup(&p->nread);
6446    release(&p->lock);
6447    return n;
6448  }
6449
```



```
6450  int
6451  piperead(struct pipe *p, char *addr, int n)
6452  {
6453    int i;
6454
6455    acquire(&p->lock);
6456    while(p->nread == p->nwrite && p->writeopen){
6457      if(proc->killed){
6458        release(&p->lock);
6459        return -1;
6460      }
6461      sleep(&p->nread, &p->lock);
6462    }
6463    for(i = 0; i < n; i++){
6464      if(p->nread == p->nwrite)
6465        break;
6466      addr[i] = p->data[p->nread++ % PIPESIZE];
6467    }
6468    wakeup(&p->nwrite);
6469    release(&p->lock);
6470    return i;
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 #include "types.h"
6501 #include "x86.h"
6502
6503 void*
6504 memset(void *dst, int c, uint n)
6505 {
6506   if ((int)dst%4 == 0 && n%4 == 0){
6507     c &= 0xFF;
6508     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
6509   } else
6510     stosb(dst, c, n);
6511   return dst;
6512 }
6513
6514 int
6515 memcmp(const void *v1, const void *v2, uint n)
6516 {
6517   const uchar *s1, *s2;
6518
6519   s1 = v1;
6520   s2 = v2;
6521   while(n-- > 0){
6522     if(*s1 != *s2)
6523       return *s1 - *s2;
6524     s1++, s2++;
6525   }
6526
6527   return 0;
6528 }
6529
6530 void*
6531 memmove(void *dst, const void *src, uint n)
6532 {
6533   const char *s;
6534   char *d;
6535
6536   s = src;
6537   d = dst;
6538   if(s < d && s + n > d){
6539     s += n;
6540     d += n;
6541     while(n-- > 0)
6542       *--d = *--s;
6543   } else
6544     while(n-- > 0)
6545       *d++ = *s++;
6546
6547   return dst;
6548 }
6549
```



```
6550 // memcpy exists to placate GCC.  Use memmove.
6551 void*
6552 memcpy(void *dst, const void *src, uint n)
6553 {
6554   return memmove(dst, src, n);
6555 }
6556
6557 int
6558 strncmp(const char *p, const char *q, uint n)
6559 {
6560   while(n > 0 && *p && *p == *q)
6561     n--, p++, q++;
6562   if(n == 0)
6563     return 0;
6564   return (uchar)*p - (uchar)*q;
6565 }
6566
6567 char*
6568 strncpy(char *s, const char *t, int n)
6569 {
6570   char *os;
6571
6572   os = s;
6573   while(n-- > 0 && (*s++ = *t++) != 0)
6574     ;
6575   while(n-- > 0)
6576     *s++ = 0;
6577   return os;
6578 }
6579
6580 // Like strncpy but guaranteed to NUL-terminate.
6581 char*
6582 safestrcpy(char *s, const char *t, int n)
6583 {
6584   char *os;
6585
6586   os = s;
6587   if(n <= 0)
6588     return os;
6589   while(--n > 0 && (*s++ = *t++) != 0)
6590     ;
6591   *s = 0;
6592   return os;
6593 }
6594
6595
6596
6597
6598
6599
```

```
6600 int
6601 strlen(const char *s)
6602 {
6603   int n;
6604
6605   for(n = 0; s[n]; n++)
6606     ;
6607   return n;
6608 }
6609
6610
6611
6612
6613
6614
6615
6616
6617
6618
6619
6620
6621
6622
6623
6624
6625
6626
6627
6628
6629
6630
6631
6632
6633
6634
6635
6636
6637
6638
6639
6640
6641
6642
6643
6644
6645
6646
6647
6648
6649
```



```
6650 // See MultiProcessor Specification Version 1.[14]
6651
6652 struct mp {             // floating pointer
6653   uchar signature[4];           // "_MP_"
6654   void *physaddr;               // phys addr of MP config table
6655   uchar length;                 // 1
6656   uchar specrev;                // [14]
6657   uchar checksum;               // all bytes must add up to 0
6658   uchar type;                   // MP system config type
6659   uchar imcrp;
6660   uchar reserved[3];
6661 };
6662
6663 struct mpconf {         // configuration table header
6664   uchar signature[4];           // "PCMP"
6665   ushort length;                // total table length
6666   uchar version;                // [14]
6667   uchar checksum;               // all bytes must add up to 0
6668   uchar product[20];            // product id
6669   uint *oemtable;               // OEM table pointer
6670   ushort oemlength;             // OEM table length
6671   ushort entry;                 // entry count
6672   uint *lapicaddr;              // address of local APIC
6673   ushort xlength;               // extended table length
6674   uchar xchecksum;              // extended table checksum
6675   uchar reserved;
6676 };
6677
6678 struct mpproc {         // processor table entry
6679   uchar type;                   // entry type (0)
6680   uchar apicid;                 // local APIC id
6681   uchar version;                // local APIC verison
6682   uchar flags;                  // CPU flags
6683     #define MPBOOT 0x02           // This proc is the bootstrap processor.
6684   uchar signature[4];           // CPU signature
6685   uint feature;                 // feature flags from CPUID instruction
6686   uchar reserved[8];
6687 };
6688
6689 struct mpioapic {       // I/O APIC table entry
6690   uchar type;                   // entry type (2)
6691   uchar apicno;                 // I/O APIC id
6692   uchar version;                // I/O APIC version
6693   uchar flags;                  // I/O APIC flags
6694   uint *addr;                  // I/O APIC address
6695 };
6696
6697
6698
6699
```

```
6700 // Table entry types
6701 #define MPPROC    0x00  // One per processor
6702 #define MPBUS     0x01  // One per bus
6703 #define MPIOAPIC  0x02  // One per I/O APIC
6704 #define MPIOINTR  0x03  // One per bus interrupt source
6705 #define MPLINTR   0x04  // One per system interrupt source
6706
6707 // Blank page.
6708
6709
6710
6711
6712
6713
6714
6715
6716
6717
6718
6719
6720
6721
6722
6723
6724
6725
6726
6727
6728
6729
6730
6731
6732
6733
6734
6735
6736
6737
6738
6739
6740
6741
6742
6743
6744
6745
6746
6747
6748
6749
```



```
6750 // Multiprocessor support
6751 // Search memory for MP description structures.
6752 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
6753
6754 #include "types.h"
6755 #include "defs.h"
6756 #include "param.h"
6757 #include "memlayout.h"
6758 #include "mp.h"
6759 #include "x86.h"
6760 #include "mmu.h"
6761 #include "proc.h"
6762
6763 struct cpu cpus[NCPU];
6764 static struct cpu *bcpu;
6765 int ismp;
6766 int ncpu;
6767 uchar ioapicid;
6768
6769 int
6770 mpbcpu(void)
6771 {
6772   return bcpu-cpus;
6773 }
6774
6775 static uchar
6776 sum(uchar *addr, int len)
6777 {
6778   int i, sum;
6779
6780   sum = 0;
6781   for(i=0; i<len; i++)
6782     sum += addr[i];
6783   return sum;
6784 }
6785
6786 // Look for an MP structure in the len bytes at addr.
6787 static struct mp*
6788 mpsearch1(uint a, int len)
6789 {
6790   uchar *e, *p, *addr;
6791
6792   addr = p2v(a);
6793   e = addr+len;
6794   for(p = addr; p < e; p += sizeof(struct mp))
6795     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
6796       return (struct mp*)p;
6797   return 0;
6798 }
6799
```

```
6800 // Search for the MP Floating Pointer Structure, which according to the
6801 // spec is in one of the following three locations:
6802 // 1) in the first KB of the EBDA;
6803 // 2) in the last KB of system base memory;
6804 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
6805 static struct mp*
6806 mpsearch(void)
6807 {
6808   uchar *bda;
6809   uint p;
6810   struct mp *mp;
6811
6812   bda = (uchar *) P2V(0x400);
6813   if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
6814     if((mp = mpsearch1(p, 1024)))
6815       return mp;
6816   } else {
6817     p = ((bda[0x14]<<8)|bda[0x13])*1024;
6818     if((mp = mpsearch1(p-1024, 1024)))
6819       return mp;
6820   }
6821   return mpsearch1(0xF0000, 0x10000);
6822 }
6823
6824 // Search for an MP configuration table.  For now,
6825 // don't accept the default configurations (physaddr == 0).
6826 // Check for correct signature, calculate the checksum and,
6827 // if correct, check the version.
6828 // To do: check extended table checksum.
6829 static struct mpconf*
6830 mpconfig(struct mp **pmp)
6831 {
6832   struct mpconf *conf;
6833   struct mp *mp;
6834
6835   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
6836     return 0;
6837   conf = (struct mpconf*) p2v((uint) mp->physaddr);
6838   if(memcmp(conf, "PCMP", 4) != 0)
6839     return 0;
6840   if(conf->version != 1 && conf->version != 4)
6841     return 0;
6842   if(sum((uchar*)conf, conf->length) != 0)
6843     return 0;
6844   *pmp = mp;
6845   return conf;
6846 }
6847
6848
6849
```



```
6850 void
6851 mpinit(void)
6852 {
6853   uchar *p, *e;
6854   struct mp *mp;
6855   struct mpconf *conf;
6856   struct mpproc *proc;
6857   struct mpioapic *ioapic;
6858
6859   bcpu = &cpus[0];
6860   if((conf = mpconfig(&mp)) == 0)
6861     return;
6862   ismp = 1;
6863   lapic = (uint*)conf->lapicaddr;
6864   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
6865     switch(*p){
6866     case MPPROC:
6867       proc = (struct mpproc*)p;
6868       if(ncpu != proc->apicid){
6869         cprintf("mpinit: ncpu=%d apicid=%d\n", ncpu, proc->apicid);
6870         ismp = 0;
6871       }
6872       if(proc->flags & MPBOOT)
6873         bcpu = &cpus[ncpu];
6874       cpus[ncpu].id = ncpu;
6875       ncpu++;
6876       p += sizeof(struct mpproc);
6877       continue;
6878     case MPIOAPIC:
6879       ioapic = (struct mpioapic*)p;
6880       ioapicid = ioapic->apicno;
6881       p += sizeof(struct mpioapic);
6882       continue;
6883     case MPBUS:
6884     case MPIOINTR:
6885     case MPLINTR:
6886       p += 8;
6887       continue;
6888     default:
6889       cprintf("mpinit: unknown config type %x\n", *p);
6890       ismp = 0;
6891     }
6892   }
6893   if(!ismp){
6894     // Didn't like what we found; fall back to no MP.
6895     ncpu = 1;
6896     lapic = 0;
6897     ioapicid = 0;
6898     return;
6899   }
```

```
6900   if(mp->imcrp){
6901     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
6902     // But it would on real hardware.
6903     outb(0x22, 0x70);   // Select IMCR
6904     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
6905   }
6906 }
6907
6908
6909
6910
6911
6912
6913
6914
6915
6916
6917
6918
6919
6920
6921
6922
6923
6924
6925
6926
6927
6928
6929
6930
6931
6932
6933
6934
6935
6936
6937
6938
6939
6940
6941
6942
6943
6944
6945
6946
6947
6948
6949
```



```
6950 // The local APIC manages internal (non-I/O) interrupts.
6951 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
6952 // As of 7/26/2016, Intel processor manual Chapter 10 of Volume 3
6953
6954 #include "types.h"
6955 #include "defs.h"
6956 #include "date.h"
6957 #include "memlayout.h"
6958 #include "traps.h"
6959 #include "mmu.h"
6960 #include "x86.h"
6961
6962 // Local APIC registers, divided by 4 for use as uint[] indices.
6963 #define ID      (0x0020/4)   // ID
6964 #define VER     (0x0030/4)   // Version
6965 #define TPR     (0x0080/4)   // Task Priority
6966 #define EOI     (0x00B0/4)   // EOI
6967 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
6968   #define ENABLE     0x00000100   // Unit Enable
6969 #define ESR     (0x0280/4)   // Error Status
6970 #define ICRLO   (0x0300/4)   // Interrupt Command
6971   #define INIT       0x00000500   // INIT/RESET
6972   #define STARTUP    0x00000600   // Startup IPI
6973   #define DELIVS     0x00001000   // Delivery status
6974   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
6975   #define DEASSERT   0x00000000
6976   #define LEVEL      0x00008000   // Level triggered
6977   #define BCAST      0x00080000   // Send to all APICs, including self.
6978   #define BUSY       0x00001000
6979   #define FIXED      0x00000000
6980 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
6981 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
6982   #define X1         0x0000000B   // divide counts by 1
6983   #define PERIODIC   0x00020000   // Periodic
6984 #define PCINT   (0x0340/4)   // Performance Counter LVT
6985 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
6986 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
6987 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
6988   #define MASKED     0x00010000   // Interrupt masked
6989 #define TICR    (0x0380/4)   // Timer Initial Count
6990 #define TCCR    (0x0390/4)   // Timer Current Count
6991 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
6992
6993 volatile uint *lapic;  // Initialized in mp.c
6994
6995 static void
6996 lapicw(int index, int value)
6997 {
6998   lapic[index] = value;
6999   lapic[ID];  // wait for write to finish, by reading
```

```
7000 }
7001
7002 void
7003 lapicinit(void)
7004 {
7005   if(!lapic)
7006     return;
7007
7008   // Enable local APIC; set spurious interrupt vector.
7009   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
7010
7011   // The timer repeatedly counts down at bus frequency
7012   // from lapic[TICR] and then issues an interrupt.
7013   // If xv6 cared more about precise timekeeping,
7014   // TICR would be calibrated using an external time source.
7015   lapicw(TDCR, X1);
7016   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
7017   lapicw(TICR, 10000000);
7018
7019   // Disable logical interrupt lines.
7020   lapicw(LINT0, MASKED);
7021   lapicw(LINT1, MASKED);
7022
7023   // Disable performance counter overflow interrupts
7024   // on machines that provide that interrupt entry.
7025   if(((lapic[VER]>>16) & 0xFF) >= 4)
7026     lapicw(PCINT, MASKED);
7027
7028   // Map error interrupt to IRQ_ERROR.
7029   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
7030
7031   // Clear error status register (requires back-to-back writes).
7032   lapicw(ESR, 0);
7033   lapicw(ESR, 0);
7034
7035   // Ack any outstanding interrupts.
7036   lapicw(EOI, 0);
7037
7038   // Send an Init Level De-Assert to synchronise arbitration ID's.
7039   lapicw(ICRHI, 0);
7040   lapicw(ICRLO, BCAST | INIT | LEVEL);
7041   while(lapic[ICRLO] & DELIVS)
7042     ;
7043
7044   // Enable interrupts on the APIC (but not on the processor).
7045   lapicw(TPR, 0);
7046 }
7047
7048
7049
```



```
7050 int
7051 cpunum(void)
7052 {
7053   // Cannot call cpu when interrupts are enabled:
7054   // result not guaranteed to last long enough to be used!
7055   // Would prefer to panic but even printing is chancy here:
7056   // almost everything, including cprintf and panic, calls cpu,
7057   // often indirectly through acquire and release.
7058   if(readeflags()&FL_IF){
7059     static int n;
7060     if(n++ == 0)
7061       cprintf("cpu called from %x with interrupts enabled\n",
7062         __builtin_return_address(0));
7063   }
7064
7065   if(lapic)
7066     return lapic[ID]>>24;
7067   return 0;
7068 }
7069
7070 // Acknowledge interrupt.
7071 void
7072 lapiceoi(void)
7073 {
7074   if(lapic)
7075     lapicw(EOI, 0);
7076 }
7077
7078 // Spin for a given number of microseconds.
7079 // On real hardware would want to tune this dynamically.
7080 void
7081 microdelay(int us)
7082 {
7083 }
7084
7085 #define CMOS_PORT    0x70
7086 #define CMOS_RETURN  0x71
7087
7088 // Start additional processor running entry code at addr.
7089 // See Appendix B of MultiProcessor Specification.
7090 void
7091 lapicstartap(uchar apicid, uint addr)
7092 {
7093   int i;
7094   ushort *wrv;
7095
7096   // "The BSP must initialize CMOS shutdown code to 0AH
7097   // and the warm reset vector (DWORD based at 40:67) to point at
7098   // the AP startup code prior to the [universal startup algorithm]."
7099   outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
```

```
7100   outb(CMOS_PORT+1, 0x0A);
7101   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
7102   wrv[0] = 0;
7103   wrv[1] = addr >> 4;
7104
7105   // "Universal startup algorithm."
7106   // Send INIT (level-triggered) interrupt to reset other CPU.
7107   lapicw(ICRHI, apicid<<24);
7108   lapicw(ICRLO, INIT | LEVEL | ASSERT);
7109   microdelay(200);
7110   lapicw(ICRLO, INIT | LEVEL);
7111   microdelay(100);    // should be 10ms, but too slow in Bochs!
7112
7113   // Send startup IPI (twice!) to enter code.
7114   // Regular hardware is supposed to only accept a STARTUP
7115   // when it is in the halted state due to an INIT.  So the second
7116   // should be ignored, but it is part of the official Intel algorithm.
7117   // Bochs complains about the second one.  Too bad for Bochs.
7118   for(i = 0; i < 2; i++){
7119     lapicw(ICRHI, apicid<<24);
7120     lapicw(ICRLO, STARTUP | (addr>>12));
7121     microdelay(200);
7122   }
7123 }
7124
7125 #define CMOS_STATA   0x0a
7126 #define CMOS_STATB   0x0b
7127 #define CMOS_UIP    (1 << 7)        // RTC update in progress
7128
7129 #define SECS    0x00
7130 #define MINS    0x02
7131 #define HOURS   0x04
7132 #define DAY     0x07
7133 #define MONTH   0x08
7134 #define YEAR    0x09
7135
7136 static uint cmos_read(uint reg)
7137 {
7138   outb(CMOS_PORT,  reg);
7139   microdelay(200);
7140
7141   return inb(CMOS_RETURN);
7142 }
7143
7144
7145
7146
7147
7148
7149
```



```
7150 static void fill_rtcdate(struct rtcdate *r)
7151 {
7152   r->second = cmos_read(SECS);
7153   r->minute = cmos_read(MINS);
7154   r->hour   = cmos_read(HOURS);
7155   r->day    = cmos_read(DAY);
7156   r->month  = cmos_read(MONTH);
7157   r->year   = cmos_read(YEAR);
7158 }
7159
7160 // qemu seems to use 24-hour GWT and the values are BCD encoded
7161 void cmostime(struct rtcdate *r)
7162 {
7163   struct rtcdate t1, t2;
7164   int sb, bcd;
7165
7166   sb = cmos_read(CMOS_STATB);
7167
7168   bcd = (sb & (1 << 2)) == 0;
7169
7170   // make sure CMOS doesn't modify time while we read it
7171   for (;;) {
7172     fill_rtcdate(&t1);
7173     if (cmos_read(CMOS_STATA) & CMOS_UIP)
7174         continue;
7175     fill_rtcdate(&t2);
7176     if (memcmp(&t1, &t2, sizeof(t1)) == 0)
7177       break;
7178   }
7179
7180   // convert
7181   if (bcd) {
7182 #define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
7183     CONV(second);
7184     CONV(minute);
7185     CONV(hour  );
7186     CONV(day   );
7187     CONV(month );
7188     CONV(year  );
7189 #undef     CONV
7190   }
7191
7192   *r = t1;
7193   r->year += 2000;
7194 }
7195
7196
7197
7198
7199
```

```
7200 // The I/O APIC manages hardware interrupts for an SMP system.
7201 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
7202 // See also picirq.c.
7203
7204 #include "types.h"
7205 #include "defs.h"
7206 #include "traps.h"
7207
7208 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
7209
7210 #define REG_ID     0x00  // Register index: ID
7211 #define REG_VER    0x01  // Register index: version
7212 #define REG_TABLE  0x10  // Redirection table base
7213
7214 // The redirection table starts at REG_TABLE and uses
7215 // two registers to configure each interrupt.
7216 // The first (low) register in a pair contains configuration bits.
7217 // The second (high) register contains a bitmask telling which
7218 // CPUs can serve that interrupt.
7219 #define INT_DISABLED   0x00010000  // Interrupt disabled
7220 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
7221 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
7222 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
7223
7224 volatile struct ioapic *ioapic;
7225
7226 // IO APIC MMIO structure: write reg, then read or write data.
7227 struct ioapic {
7228   uint reg;
7229   uint pad[3];
7230   uint data;
7231 };
7232
7233 static uint
7234 ioapicread(int reg)
7235 {
7236   ioapic->reg = reg;
7237   return ioapic->data;
7238 }
7239
7240 static void
7241 ioapicwrite(int reg, uint data)
7242 {
7243   ioapic->reg = reg;
7244   ioapic->data = data;
7245 }
7246
7247
7248
7249
```



```
7250 void
7251 ioapicinit(void)
7252 {
7253   int i, id, maxintr;
7254
7255   if(!ismp)
7256     return;
7257
7258   ioapic = (volatile struct ioapic*)IOAPIC;
7259   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
7260   id = ioapicread(REG_ID) >> 24;
7261   if(id != ioapicid)
7262     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
7263
7264   // Mark all interrupts edge-triggered, active high, disabled,
7265   // and not routed to any CPUs.
7266   for(i = 0; i <= maxintr; i++){
7267     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
7268     ioapicwrite(REG_TABLE+2*i+1, 0);
7269   }
7270 }
7271
7272 void
7273 ioapicenable(int irq, int cpunum)
7274 {
7275   if(!ismp)
7276     return;
7277
7278   // Mark interrupt edge-triggered, active high,
7279   // enabled, and routed to the given cpunum,
7280   // which happens to be that cpu's APIC ID.
7281   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
7282   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
7283 }
7284
7285
7286
7287
7288
7289
7290
7291
7292
7293
7294
7295
7296
7297
7298
7299
```

```
7300 // Intel 8259A programmable interrupt controllers.
7301
7302 #include "types.h"
7303 #include "x86.h"
7304 #include "traps.h"
7305
7306 // I/O Addresses of the two programmable interrupt controllers
7307 #define IO_PIC1         0x20    // Master (IRQs 0-7)
7308 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
7309
7310 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
7311
7312 // Current IRQ mask.
7313 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
7314 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
7315
7316 static void
7317 picsetmask(ushort mask)
7318 {
7319   irqmask = mask;
7320   outb(IO_PIC1+1, mask);
7321   outb(IO_PIC2+1, mask >> 8);
7322 }
7323
7324 void
7325 picenable(int irq)
7326 {
7327   picsetmask(irqmask & ~(1<<irq));
7328 }
7329
7330 // Initialize the 8259A interrupt controllers.
7331 void
7332 picinit(void)
7333 {
7334   // mask all interrupts
7335   outb(IO_PIC1+1, 0xFF);
7336   outb(IO_PIC2+1, 0xFF);
7337
7338   // Set up master (8259A-1)
7339
7340   // ICW1:  0001g0hi
7341   //    g:  0 = edge triggering, 1 = level triggering
7342   //    h:  0 = cascaded PICs, 1 = master only
7343   //    i:  0 = no ICW4, 1 = ICW4 required
7344   outb(IO_PIC1, 0x11);
7345
7346   // ICW2:  Vector offset
7347   outb(IO_PIC1+1, T_IRQ0);
7348
7349
```



```
7350   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
7351   //        (slave PIC) 3-bit # of slave's connection to master
7352   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
7353
7354   // ICW4:  000nbmap
7355   //    n:  1 = special fully nested mode
7356   //    b:  1 = buffered mode
7357   //    m:  0 = slave PIC, 1 = master PIC
7358   //      (ignored when b is 0, as the master/slave role
7359   //      can be hardwired).
7360   //    a:  1 = Automatic EOI mode
7361   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
7362   outb(IO_PIC1+1, 0x3);
7363
7364   // Set up slave (8259A-2)
7365   outb(IO_PIC2, 0x11);                  // ICW1
7366   outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
7367   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
7368   // NB Automatic EOI mode doesn't tend to work on the slave.
7369   // Linux source code says it's "to be investigated".
7370   outb(IO_PIC2+1, 0x3);                 // ICW4
7371
7372   // OCW3:  0ef01prs
7373   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
7374   //    p:  0 = no polling, 1 = polling mode
7375   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
7376   outb(IO_PIC1, 0x68);             // clear specific mask
7377   outb(IO_PIC1, 0x0a);             // read IRR by default
7378
7379   outb(IO_PIC2, 0x68);             // OCW3
7380   outb(IO_PIC2, 0x0a);             // OCW3
7381
7382   if(irqmask != 0xFFFF)
7383     picsetmask(irqmask);
7384 }
7385
7386
7387
7388
7389
7390
7391
7392
7393
7394
7395
7396
7397
7398
7399
```

```
7400 // PC keyboard interface constants
7401
7402 #define KBSTATP         0x64    // kbd controller status port(I)
7403 #define KBS_DIB         0x01    // kbd data in buffer
7404 #define KBDATAP         0x60    // kbd data port(I)
7405
7406 #define NO              0
7407
7408 #define SHIFT           (1<<0)
7409 #define CTL             (1<<1)
7410 #define ALT             (1<<2)
7411
7412 #define CAPSLOCK        (1<<3)
7413 #define NUMLOCK         (1<<4)
7414 #define SCROLLLOCK      (1<<5)
7415
7416 #define E0ESC           (1<<6)
7417
7418 // Special keycodes
7419 #define KEY_HOME        0xE0
7420 #define KEY_END         0xE1
7421 #define KEY_UP          0xE2
7422 #define KEY_DN          0xE3
7423 #define KEY_LF          0xE4
7424 #define KEY_RT          0xE5
7425 #define KEY_PGUP        0xE6
7426 #define KEY_PGDN        0xE7
7427 #define KEY_INS         0xE8
7428 #define KEY_DEL         0xE9
7429
7430 // C('A') == Control-A
7431 #define C(x) (x - '@')
7432
7433 static uchar shiftcode[256] =
7434 {
7435   [0x1D] CTL,
7436   [0x2A] SHIFT,
7437   [0x36] SHIFT,
7438   [0x38] ALT,
7439   [0x9D] CTL,
7440   [0xB8] ALT
7441 };
7442
7443 static uchar togglecode[256] =
7444 {
7445   [0x3A] CAPSLOCK,
7446   [0x45] NUMLOCK,
7447   [0x46] SCROLLLOCK
7448 };
7449
```



```
7450 static uchar normalmap[256] =
7451 {
7452   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
7453   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
7454   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
7455   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
7456   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
7457   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
7458   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
7459   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7460   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7461   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7462   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7463   [0x9C] '\n',      // KP_Enter
7464   [0xB5] '/',       // KP_Div
7465   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7466   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7467   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7468   [0x97] KEY_HOME,  [0xCF] KEY_END,
7469   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7470 };
7471
7472 static uchar shiftmap[256] =
7473 {
7474   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
7475   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
7476   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
7477   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
7478   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
7479   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
7480   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
7481   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7482   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7483   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7484   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7485   [0x9C] '\n',      // KP_Enter
7486   [0xB5] '/',       // KP_Div
7487   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7488   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7489   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7490   [0x97] KEY_HOME,  [0xCF] KEY_END,
7491   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7492 };
7493
7494
7495
7496
7497
7498
7499
```

```
7500 static uchar ctlmap[256] =
7501 {
7502   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7503   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7504   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
7505   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
7506   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
7507   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
7508   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
7509   [0x9C] '\r',      // KP_Enter
7510   [0xB5] C('/'),    // KP_Div
7511   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7512   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7513   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7514   [0x97] KEY_HOME,  [0xCF] KEY_END,
7515   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7516 };
7517
7518
7519
7520
7521
7522
7523
7524
7525
7526
7527
7528
7529
7530
7531
7532
7533
7534
7535
7536
7537
7538
7539
7540
7541
7542
7543
7544
7545
7546
7547
7548
7549
```



```
7550 #include "types.h"
7551 #include "x86.h"
7552 #include "defs.h"
7553 #include "kbd.h"
7554
7555 int
7556 kbdgetc(void)
7557 {
7558   static uint shift;
7559   static uchar *charcode[4] = {
7560     normalmap, shiftmap, ctlmap, ctlmap
7561   };
7562   uint st, data, c;
7563
7564   st = inb(KBSTATP);
7565   if((st & KBS_DIB) == 0)
7566     return -1;
7567   data = inb(KBDATAP);
7568
7569   if(data == 0xE0){
7570     shift |= E0ESC;
7571     return 0;
7572   } else if(data & 0x80){
7573     // Key released
7574     data = (shift & E0ESC ? data : data & 0x7F);
7575     shift &= ~(shiftcode[data] | E0ESC);
7576     return 0;
7577   } else if(shift & E0ESC){
7578     // Last character was an E0 escape; or with 0x80
7579     data |= 0x80;
7580     shift &= ~E0ESC;
7581   }
7582
7583   shift |= shiftcode[data];
7584   shift ^= togglecode[data];
7585   c = charcode[shift & (CTL | SHIFT)][data];
7586   if(shift & CAPSLOCK){
7587     if('a' <= c && c <= 'z')
7588       c += 'A' - 'a';
7589     else if('A' <= c && c <= 'Z')
7590       c += 'a' - 'A';
7591   }
7592   return c;
7593 }
7594
7595 void
7596 kbdintr(void)
7597 {
7598   consoleintr(kbdgetc);
7599 }
```

```
7600 // Console input and output.
7601 // Input is from the keyboard or serial port.
7602 // Output is written to the screen and serial port.
7603
7604 #include "types.h"
7605 #include "defs.h"
7606 #include "param.h"
7607 #include "traps.h"
7608 #include "spinlock.h"
7609 #include "fs.h"
7610 #include "file.h"
7611 #include "memlayout.h"
7612 #include "mmu.h"
7613 #include "proc.h"
7614 #include "x86.h"
7615
7616 static void consputc(int);
7617
7618 static int panicked = 0;
7619
7620 static struct {
7621   struct spinlock lock;
7622   int locking;
7623 } cons;
7624
7625 static void
7626 printint(int xx, int base, int sign)
7627 {
7628   static char digits[] = "0123456789abcdef";
7629   char buf[16];
7630   int i;
7631   uint x;
7632
7633   if(sign && (sign = xx < 0))
7634     x = -xx;
7635   else
7636     x = xx;
7637
7638   i = 0;
7639   do{
7640     buf[i++] = digits[x % base];
7641   }while((x /= base) != 0);
7642
7643   if(sign)
7644     buf[i++] = '-';
7645
7646   while(--i >= 0)
7647     consputc(buf[i]);
7648 }
7649
```



```
7650 // Print to the console. only understands %d, %x, %p, %s.
7651 void
7652 cprintf(char *fmt, ...)
7653 {
7654   int i, c, locking;
7655   uint *argp;
7656   char *s;
7657
7658   locking = cons.locking;
7659   if(locking)
7660     acquire(&cons.lock);
7661
7662   if (fmt == 0)
7663     panic("null fmt");
7664
7665   argp = (uint*)(void*)(&fmt + 1);
7666   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
7667     if(c != '%'){
7668       consputc(c);
7669       continue;
7670     }
7671     c = fmt[++i] & 0xff;
7672     if(c == 0)
7673       break;
7674     switch(c){
7675     case 'd':
7676       printint(*argp++, 10, 1);
7677       break;
7678     case 'x':
7679     case 'p':
7680       printint(*argp++, 16, 0);
7681       break;
7682     case 's':
7683       if((s = (char*)*argp++) == 0)
7684         s = "(null)";
7685       for(; *s; s++)
7686         consputc(*s);
7687       break;
7688     case '%':
7689       consputc('%');
7690       break;
7691     default:
7692       // Print unknown % sequence to draw attention.
7693       consputc('%');
7694       consputc(c);
7695       break;
7696     }
7697   }
7698
7699
```

```
7700   if(locking)
7701     release(&cons.lock);
7702 }
7703
7704 void
7705 panic(char *s)
7706 {
7707   int i;
7708   uint pcs[10];
7709
7710   cli();
7711   cons.locking = 0;
7712   cprintf("cpu%d: panic: ", cpu->id);
7713   cprintf(s);
7714   cprintf("\n");
7715   getcallerpcs(&s, pcs);
7716   for(i=0; i<10; i++)
7717     cprintf(" %p", pcs[i]);
7718   panicked = 1; // freeze other CPU
7719   for(;;)
7720     ;
7721 }
7722
7723 #define BACKSPACE 0x100
7724 #define CRTPORT 0x3d4
7725 static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory
7726
7727 static void
7728 cgaputc(int c)
7729 {
7730   int pos;
7731
7732   // Cursor position: col + 80*row.
7733   outb(CRTPORT, 14);
7734   pos = inb(CRTPORT+1) << 8;
7735   outb(CRTPORT, 15);
7736   pos |= inb(CRTPORT+1);
7737
7738   if(c == '\n')
7739     pos += 80 - pos%80;
7740   else if(c == BACKSPACE){
7741     if(pos > 0) --pos;
7742   } else
7743     crt[pos++] = (c&0xff) | 0x0700;  // black on white
7744
7745   if(pos < 0 || pos > 25*80)
7746     panic("pos under/overflow");
7747
7748
7749
```



```
7750   if((pos/80) >= 24){  // Scroll up.
7751     memmove(crt, crt+80, sizeof(crt[0])*23*80);
7752     pos -= 80;
7753     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
7754   }
7755
7756   outb(CRTPORT, 14);
7757   outb(CRTPORT+1, pos>>8);
7758   outb(CRTPORT, 15);
7759   outb(CRTPORT+1, pos);
7760   crt[pos] = ' ' | 0x0700;
7761 }
7762
7763 void
7764 consputc(int c)
7765 {
7766   if(panicked){
7767     cli();
7768     for(;;)
7769       ;
7770   }
7771
7772   if(c == BACKSPACE){
7773     uartputc('\b'); uartputc(' '); uartputc('\b');
7774   } else
7775     uartputc(c);
7776   cgaputc(c);
7777 }
7778
7779 #define INPUT_BUF 128
7780 struct {
7781   char buf[INPUT_BUF];
7782   uint r;  // Read index
7783   uint w;  // Write index
7784   uint e;  // Edit index
7785 } input;
7786
7787 #define C(x)  ((x)-'@')  // Control-x
7788
7789 void
7790 consoleintr(int (*getc)(void))
7791 {
7792   int c, doprocdump = 0;
7793
7794   acquire(&cons.lock);
7795   while((c = getc()) >= 0){
7796     switch(c){
7797     case C('P'):  // Process listing.
7798       doprocdump = 1;   // procdump() locks cons.lock indirectly; invoke late
7799       break;
```

```
7800     case C('U'):  // Kill line.
7801       while(input.e != input.w &&
7802             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
7803         input.e--;
7804         consputc(BACKSPACE);
7805       }
7806       break;
7807     case C('H'): case '\x7f':  // Backspace
7808       if(input.e != input.w){
7809         input.e--;
7810         consputc(BACKSPACE);
7811       }
7812       break;
7813     default:
7814       if(c != 0 && input.e-input.r < INPUT_BUF){
7815         c = (c == '\r') ? '\n' : c;
7816         input.buf[input.e++ % INPUT_BUF] = c;
7817         consputc(c);
7818         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
7819           input.w = input.e;
7820           wakeup(&input.r);
7821         }
7822       }
7823       break;
7824     }
7825   }
7826   release(&cons.lock);
7827   if(doprocdump) {
7828     procdump();  // now call procdump() wo. cons.lock held
7829   }
7830 }
7831
7832 int
7833 consoleread(struct inode *ip, char *dst, int n)
7834 {
7835   uint target;
7836   int c;
7837
7838   iunlock(ip);
7839   target = n;
7840   acquire(&cons.lock);
7841   while(n > 0){
7842     while(input.r == input.w){
7843       if(proc->killed){
7844         release(&cons.lock);
7845         ilock(ip);
7846         return -1;
7847       }
7848       sleep(&input.r, &cons.lock);
7849     }
```



```
7850     c = input.buf[input.r++ % INPUT_BUF];
7851     if(c == C('D')){  // EOF
7852       if(n < target){
7853         // Save ^D for next time, to make sure
7854         // caller gets a 0-byte result.
7855         input.r--;
7856       }
7857       break;
7858     }
7859     *dst++ = c;
7860     --n;
7861     if(c == '\n')
7862       break;
7863   }
7864   release(&cons.lock);
7865   ilock(ip);
7866
7867   return target - n;
7868 }
7869
7870 int
7871 consolewrite(struct inode *ip, char *buf, int n)
7872 {
7873   int i;
7874
7875   iunlock(ip);
7876   acquire(&cons.lock);
7877   for(i = 0; i < n; i++)
7878     consputc(buf[i] & 0xff);
7879   release(&cons.lock);
7880   ilock(ip);
7881
7882   return n;
7883 }
7884
7885 void
7886 consoleinit(void)
7887 {
7888   initlock(&cons.lock, "console");
7889
7890   devsw[CONSOLE].write = consolewrite;
7891   devsw[CONSOLE].read = consoleread;
7892   cons.locking = 1;
7893
7894   picenable(IRQ_KBD);
7895   ioapicenable(IRQ_KBD, 0);
7896 }
7897
7898
7899
```

```
7900 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
7901 // Only used on uniprocessors;
7902 // SMP machines use the local APIC timer.
7903
7904 #include "types.h"
7905 #include "defs.h"
7906 #include "traps.h"
7907 #include "x86.h"
7908
7909 #define IO_TIMER1       0x040           // 8253 Timer #1
7910
7911 // Frequency of all three count-down timers;
7912 // (TIMER_FREQ/freq) is the appropriate count
7913 // to generate a frequency of freq Hz.
7914
7915 #define TIMER_FREQ      1193182
7916 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
7917
7918 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
7919 #define TIMER_SEL0      0x00    // select counter 0
7920 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
7921 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
7922
7923 void
7924 timerinit(void)
7925 {
7926   // Interrupt 100 times/sec.
7927   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
7928   outb(IO_TIMER1, TIMER_DIV(100) % 256);
7929   outb(IO_TIMER1, TIMER_DIV(100) / 256);
7930   picenable(IRQ_TIMER);
7931 }
7932
7933
7934
7935
7936
7937
7938
7939
7940
7941
7942
7943
7944
7945
7946
7947
7948
7949
```



```
7950 // Intel 8250 serial port (UART).
7951
7952 #include "types.h"
7953 #include "defs.h"
7954 #include "param.h"
7955 #include "traps.h"
7956 #include "spinlock.h"
7957 #include "fs.h"
7958 #include "file.h"
7959 #include "mmu.h"
7960 #include "proc.h"
7961 #include "x86.h"
7962
7963 #define COM1    0x3f8
7964
7965 static int uart;    // is there a uart?
7966
7967 void
7968 uartinit(void)
7969 {
7970   char *p;
7971
7972   // Turn off the FIFO
7973   outb(COM1+2, 0);
7974
7975   // 9600 baud, 8 data bits, 1 stop bit, parity off.
7976   outb(COM1+3, 0x80);    // Unlock divisor
7977   outb(COM1+0, 115200/9600);
7978   outb(COM1+1, 0);
7979   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
7980   outb(COM1+4, 0);
7981   outb(COM1+1, 0x01);    // Enable receive interrupts.
7982
7983   // If status is 0xFF, no serial port.
7984   if(inb(COM1+5) == 0xFF)
7985     return;
7986   uart = 1;
7987
7988   // Acknowledge pre-existing interrupt conditions;
7989   // enable interrupts.
7990   inb(COM1+2);
7991   inb(COM1+0);
7992   picenable(IRQ_COM1);
7993   ioapicenable(IRQ_COM1, 0);
7994
7995   // Announce that we're here.
7996   for(p="xv6...\n"; *p; p++)
7997     uartputc(*p);
7998 }
7999
```

```
8000 void
8001 uartputc(int c)
8002 {
8003   int i;
8004
8005   if(!uart)
8006     return;
8007   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
8008     microdelay(10);
8009   outb(COM1+0, c);
8010 }
8011
8012 static int
8013 uartgetc(void)
8014 {
8015   if(!uart)
8016     return -1;
8017   if(!(inb(COM1+5) & 0x01))
8018     return -1;
8019   return inb(COM1+0);
8020 }
8021
8022 void
8023 uartintr(void)
8024 {
8025   consoleintr(uartgetc);
8026 }
8027
8028
8029
8030
8031
8032
8033
8034
8035
8036
8037
8038
8039
8040
8041
8042
8043
8044
8045
8046
8047
8048
8049
```



```
8050 # Initial process execs /init.
8051
8052 #include "syscall.h"
8053 #include "traps.h"
8054
8055
8056 # exec(init, argv)
8057 .globl start
8058 start:
8059   pushl $argv
8060   pushl $init
8061   pushl $0  // where caller pc would be
8062   movl $SYS_exec, %eax
8063   int $T_SYSCALL
8064
8065 # for(;;) exit();
8066 exit:
8067   movl $SYS_exit, %eax
8068   int $T_SYSCALL
8069   jmp exit
8070
8071 # char init[] = "/init\0";
8072 init:
8073   .string "/init\0"
8074
8075 # char *argv[] = { init, 0 };
8076 .p2align 2
8077 argv:
8078   .long init
8079   .long 0
8080
8081
8082
8083
8084
8085
8086
8087
8088
8089
8090
8091
8092
8093
8094
8095
8096
8097
8098
8099
```

```
8100 #include "syscall.h"
8101 #include "traps.h"
8102
8103 #define SYSCALL(name) \
8104   .globl name; \
8105   name: \
8106     movl $SYS_ ## name, %eax; \
8107     int $T_SYSCALL; \
8108     ret
8109
8110 SYSCALL(fork)
8111 SYSCALL(exit)
8112 SYSCALL(wait)
8113 SYSCALL(pipe)
8114 SYSCALL(read)
8115 SYSCALL(write)
8116 SYSCALL(close)
8117 SYSCALL(kill)
8118 SYSCALL(exec)
8119 SYSCALL(open)
8120 SYSCALL(mknod)
8121 SYSCALL(unlink)
8122 SYSCALL(fstat)
8123 SYSCALL(link)
8124 SYSCALL(mkdir)
8125 SYSCALL(chdir)
8126 SYSCALL(dup)
8127 SYSCALL(getpid)
8128 SYSCALL(sbrk)
8129 SYSCALL(sleep)
8130 SYSCALL(uptime)
8131 SYSCALL(halt)
8132 //Student Implementations
8133 SYSCALL(date)
8134
8135 SYSCALL(getuid)
8136 SYSCALL(getgid)
8137 SYSCALL(getppid)
8138
8139 SYSCALL(setuid)
8140 SYSCALL(setgid)
8141 SYSCALL(getprocs)
8142 // Project 3
8143 SYSCALL(setpriority)
8144
8145
8146
8147
8148
8149
```



```
8150 // init: The initial user-level program
8151
8152 #include "types.h"
8153 #include "stat.h"
8154 #include "user.h"
8155 #include "fcntl.h"
8156
8157 char *argv[] = { "sh", 0 };
8158
8159 int
8160 main(void ){
8161
8162   int pid, wpid;
8163
8164   if(open("console", O_RDWR) < 0){
8165     mknod("console", 1, 1);
8166     open("console", O_RDWR);
8167   }
8168   dup(0);  // stdout
8169   dup(0);  // stderr
8170
8171   for(;;){
8172     printf(1, "init: starting sh\n");
8173     pid = fork();
8174     if(pid < 0){
8175       printf(1, "init: fork failed\n");
8176       exit();
8177     }
8178     if(pid == 0){
8179       exec("sh", argv);
8180       printf(1, "init: exec sh failed\n");
8181       exit();
8182     }
8183     while((wpid=wait()) >= 0 && wpid != pid)
8184       printf(1, "zombie!\n");
8185   }
8186 }
8187
8188
8189
8190
8191
8192
8193
8194
8195
8196
8197
8198
8199
```

```
8200 // Shell.
8201 // 2015-12-21. Added very simple processing for builtin commands
8202
8203 #include "types.h"
8204 #include "user.h"
8205 #include "fcntl.h"
8206
8207 // Parsed command representation
8208 #define EXEC  1
8209 #define REDIR 2
8210 #define PIPE  3
8211 #define LIST  4
8212 #define BACK  5
8213
8214 #define MAXARGS 10
8215
8216 struct cmd {
8217   int type;
8218 };
8219
8220 struct execcmd {
8221   int type;
8222   char *argv[MAXARGS];
8223   char *eargv[MAXARGS];
8224 };
8225
8226 struct redircmd {
8227   int type;
8228   struct cmd *cmd;
8229   char *file;
8230   char *efile;
8231   int mode;
8232   int fd;
8233 };
8234
8235 struct pipecmd {
8236   int type;
8237   struct cmd *left;
8238   struct cmd *right;
8239 };
8240
8241 struct listcmd {
8242   int type;
8243   struct cmd *left;
8244   struct cmd *right;
8245 };
8246
8247
8248
8249
```



```
8250 struct backcmd {
8251   int type;
8252   struct cmd *cmd;
8253 };
8254
8255 int fork1(void);  // Fork but panics on failure.
8256 void panic(char*);
8257 struct cmd *parsecmd(char*);
8258
8259 // Execute cmd.  Never returns.
8260 void
8261 runcmd(struct cmd *cmd)
8262 {
8263   int p[2];
8264   struct backcmd *bcmd;
8265   struct execcmd *ecmd;
8266   struct listcmd *lcmd;
8267   struct pipecmd *pcmd;
8268   struct redircmd *rcmd;
8269
8270   if(cmd == 0)
8271     exit();
8272
8273   switch(cmd->type){
8274   default:
8275     panic("runcmd");
8276
8277   case EXEC:
8278     ecmd = (struct execcmd*)cmd;
8279     if(ecmd->argv[0] == 0)
8280       exit();
8281     exec(ecmd->argv[0], ecmd->argv);
8282     printf(2, "exec %s failed\n", ecmd->argv[0]);
8283     break;
8284
8285   case REDIR:
8286     rcmd = (struct redircmd*)cmd;
8287     close(rcmd->fd);
8288     if(open(rcmd->file, rcmd->mode) < 0){
8289       printf(2, "open %s failed\n", rcmd->file);
8290       exit();
8291     }
8292     runcmd(rcmd->cmd);
8293     break;
8294
8295   case LIST:
8296     lcmd = (struct listcmd*)cmd;
8297     if(fork1() == 0)
8298       runcmd(lcmd->left);
8299     wait();
```

```
8300     runcmd(lcmd->right);
8301     break;
8302
8303   case PIPE:
8304     pcmd = (struct pipecmd*)cmd;
8305     if(pipe(p) < 0)
8306       panic("pipe");
8307     if(fork1() == 0){
8308       close(1);
8309       dup(p[1]);
8310       close(p[0]);
8311       close(p[1]);
8312       runcmd(pcmd->left);
8313     }
8314     if(fork1() == 0){
8315       close(0);
8316       dup(p[0]);
8317       close(p[0]);
8318       close(p[1]);
8319       runcmd(pcmd->right);
8320     }
8321     close(p[0]);
8322     close(p[1]);
8323     wait();
8324     wait();
8325     break;
8326
8327   case BACK:
8328     bcmd = (struct backcmd*)cmd;
8329     if(fork1() == 0)
8330       runcmd(bcmd->cmd);
8331     break;
8332   }
8333   exit();
8334 }
8335
8336 int
8337 getcmd(char *buf, int nbuf)
8338 {
8339   printf(2, "$ ");
8340   memset(buf, 0, nbuf);
8341   gets(buf, nbuf);
8342   if(buf[0] == 0) // EOF
8343     return -1;
8344   return 0;
8345 }
8346
8347
8348
8349
```



```
8350 #ifdef USE_BUILTINS
8351 // ***** processing for shell builtins begins here *****
8352
8353 int
8354 strncmp(const char *p, const char *q, uint n)
8355 {
8356     while(n > 0 && *p && *p == *q)
8357       n--, p++, q++;
8358     if(n == 0)
8359       return 0;
8360     return (uchar)*p - (uchar)*q;
8361 }
8362
8363 int
8364 makeint(char *p)
8365 {
8366   int val = 0;
8367
8368   while ((*p >= '0') && (*p <= '9')) {
8369     val = 10*val + (*p-'0');
8370     ++p;
8371   }
8372   return val;
8373 }
8374
8375 int
8376 setbuiltin(char *p)
8377 {
8378   int i;
8379
8380   p += strlen("_set");
8381   while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8382   if (strncmp("uid", p, 3) == 0) {
8383     p += strlen("uid");
8384     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8385     i = makeint(p); // ugly
8386     return (setuid(i));
8387   } else
8388   if (strncmp("gid", p, 3) == 0) {
8389     p += strlen("gid");
8390     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8391     i = makeint(p); // ugly
8392     return (setgid(i));
8393   }
8394   printf(2, "Invalid _set parameter\n");
8395   return -1;
8396 }
8397
8398
8399
```

```
8400 int
8401 getbuiltin(char *p)
8402 {
8403   p += strlen("_get");
8404   while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8405   if (strncmp("uid", p, 3) == 0) {
8406     printf(2, "%d\n", getuid());
8407     return 0;
8408   }
8409   if (strncmp("gid", p, 3) == 0) {
8410     printf(2, "%d\n", getgid());
8411     return 0;
8412   }
8413   printf(2, "Invalid _get parameter\n");
8414   return -1;
8415 }
8416
8417 typedef int funcPtr_t(char *);
8418 typedef struct {
8419   char       *cmd;
8420   funcPtr_t  *name;
8421 } dispatchTableEntry_t;
8422
8423 // Use a simple function dispatch table (FDT) to process builtin commands
8424 dispatchTableEntry_t fdt[] = {
8425   {"_set", setbuiltin},
8426   {"_get", getbuiltin}
8427 };
8428 int FDTcount = sizeof(fdt) / sizeof(fdt[0]); // # entris in FDT
8429
8430 void
8431 dobuiltin(char *cmd) {
8432   int i;
8433
8434   for (i=0; i<FDTcount; i++)
8435     if (strncmp(cmd, fdt[i].cmd, strlen(fdt[i].cmd)) == 0)
8436      (*fdt[i].name)(cmd);
8437 }
8438
8439
8440
8441
8442
8443
8444
8445
8446
8447
8448
8449
```



```
8450 // ***** processing for shell builtins ends here *****
8451 #endif
8452
8453 int
8454 main(void)
8455 {
8456   static char buf[100];
8457   int fd;
8458
8459   // Assumes three file descriptors open.
8460   while((fd = open("console", O_RDWR)) >= 0){
8461     if(fd >= 3){
8462       close(fd);
8463       break;
8464     }
8465   }
8466
8467   // Read and run input commands.
8468   while(getcmd(buf, sizeof(buf)) >= 0){
8469 // add support for built-ins here. cd is a built-in
8470     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
8471       // Clumsy but will have to do for now.
8472       // Chdir has no effect on the parent if run in the child.
8473       buf[strlen(buf)-1] = 0;  // chop \n
8474       if(chdir(buf+3) < 0)
8475         printf(2, "cannot cd %s\n", buf+3);
8476       continue;
8477     }
8478 #ifdef USE_BUILTINS
8479     if (buf[0]=='_') {     // assume it is a builtin command
8480       dobuiltin(buf);
8481       continue;
8482     }
8483 #endif
8484     if(fork1() == 0)
8485       runcmd(parsecmd(buf));
8486     wait();
8487   }
8488   exit();
8489 }
8490
8491 void
8492 panic(char *s)
8493 {
8494   printf(2, "%s\n", s);
8495   exit();
8496 }
8497
8498
8499
```

```
8500 int
8501 fork1(void)
8502 {
8503   int pid;
8504
8505   pid = fork();
8506   if(pid == -1)
8507     panic("fork");
8508   return pid;
8509 }
8510
8511 // Constructors
8512
8513 struct cmd*
8514 execcmd(void)
8515 {
8516   struct execcmd *cmd;
8517
8518   cmd = malloc(sizeof(*cmd));
8519   memset(cmd, 0, sizeof(*cmd));
8520   cmd->type = EXEC;
8521   return (struct cmd*)cmd;
8522 }
8523
8524 struct cmd*
8525 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
8526 {
8527   struct redircmd *cmd;
8528
8529   cmd = malloc(sizeof(*cmd));
8530   memset(cmd, 0, sizeof(*cmd));
8531   cmd->type = REDIR;
8532   cmd->cmd = subcmd;
8533   cmd->file = file;
8534   cmd->efile = efile;
8535   cmd->mode = mode;
8536   cmd->fd = fd;
8537   return (struct cmd*)cmd;
8538 }
8539
8540
8541
8542
8543
8544
8545
8546
8547
8548
8549
```



```
8550 struct cmd*
8551 pipecmd(struct cmd *left, struct cmd *right)
8552 {
8553   struct pipecmd *cmd;
8554
8555   cmd = malloc(sizeof(*cmd));
8556   memset(cmd, 0, sizeof(*cmd));
8557   cmd->type = PIPE;
8558   cmd->left = left;
8559   cmd->right = right;
8560   return (struct cmd*)cmd;
8561 }
8562
8563 struct cmd*
8564 listcmd(struct cmd *left, struct cmd *right)
8565 {
8566   struct listcmd *cmd;
8567
8568   cmd = malloc(sizeof(*cmd));
8569   memset(cmd, 0, sizeof(*cmd));
8570   cmd->type = LIST;
8571   cmd->left = left;
8572   cmd->right = right;
8573   return (struct cmd*)cmd;
8574 }
8575
8576 struct cmd*
8577 backcmd(struct cmd *subcmd)
8578 {
8579   struct backcmd *cmd;
8580
8581   cmd = malloc(sizeof(*cmd));
8582   memset(cmd, 0, sizeof(*cmd));
8583   cmd->type = BACK;
8584   cmd->cmd = subcmd;
8585   return (struct cmd*)cmd;
8586 }
8587
8588
8589
8590
8591
8592
8593
8594
8595
8596
8597
8598
8599
```

```
8600 // Parsing
8601
8602 char whitespace[] = " \t\r\n\v";
8603 char symbols[] = "<|>&;()";
8604
8605 int
8606 gettoken(char **ps, char *es, char **q, char **eq)
8607 {
8608   char *s;
8609   int ret;
8610
8611   s = *ps;
8612   while(s < es && strchr(whitespace, *s))
8613     s++;
8614   if(q)
8615     *q = s;
8616   ret = *s;
8617   switch(*s){
8618   case 0:
8619     break;
8620   case '|':
8621   case '(':
8622   case ')':
8623   case ';':
8624   case '&':
8625   case '<':
8626     s++;
8627     break;
8628   case '>':
8629     s++;
8630     if(*s == '>'){
8631       ret = '+';
8632       s++;
8633     }
8634     break;
8635   default:
8636     ret = 'a';
8637     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
8638       s++;
8639     break;
8640   }
8641   if(eq)
8642     *eq = s;
8643
8644   while(s < es && strchr(whitespace, *s))
8645     s++;
8646   *ps = s;
8647   return ret;
8648 }
8649
```



```
8650 int
8651 peek(char **ps, char *es, char *toks)
8652 {
8653   char *s;
8654
8655   s = *ps;
8656   while(s < es && strchr(whitespace, *s))
8657     s++;
8658   *ps = s;
8659   return *s && strchr(toks, *s);
8660 }
8661
8662 struct cmd *parseline(char**, char*);
8663 struct cmd *parsepipe(char**, char*);
8664 struct cmd *parseexec(char**, char*);
8665 struct cmd *nulterminate(struct cmd*);
8666
8667 struct cmd*
8668 parsecmd(char *s)
8669 {
8670   char *es;
8671   struct cmd *cmd;
8672
8673   es = s + strlen(s);
8674   cmd = parseline(&s, es);
8675   peek(&s, es, "");
8676   if(s != es){
8677     printf(2, "leftovers: %s\n", s);
8678     panic("syntax");
8679   }
8680   nulterminate(cmd);
8681   return cmd;
8682 }
8683
8684 struct cmd*
8685 parseline(char **ps, char *es)
8686 {
8687   struct cmd *cmd;
8688
8689   cmd = parsepipe(ps, es);
8690   while(peek(ps, es, "&")){
8691     gettoken(ps, es, 0, 0);
8692     cmd = backcmd(cmd);
8693   }
8694   if(peek(ps, es, ";")){
8695     gettoken(ps, es, 0, 0);
8696     cmd = listcmd(cmd, parseline(ps, es));
8697   }
8698   return cmd;
8699 }
```

```
8700 struct cmd*
8701 parsepipe(char **ps, char *es)
8702 {
8703   struct cmd *cmd;
8704
8705   cmd = parseexec(ps, es);
8706   if(peek(ps, es, "|")){
8707     gettoken(ps, es, 0, 0);
8708     cmd = pipecmd(cmd, parsepipe(ps, es));
8709   }
8710   return cmd;
8711 }
8712
8713 struct cmd*
8714 parseredirs(struct cmd *cmd, char **ps, char *es)
8715 {
8716   int tok;
8717   char *q, *eq;
8718
8719   while(peek(ps, es, "<>")){
8720     tok = gettoken(ps, es, 0, 0);
8721     if(gettoken(ps, es, &q, &eq) != 'a')
8722       panic("missing file for redirection");
8723     switch(tok){
8724     case '<':
8725       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
8726       break;
8727     case '>':
8728       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8729       break;
8730     case '+':  // >>
8731       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8732       break;
8733     }
8734   }
8735   return cmd;
8736 }
8737
8738
8739
8740
8741
8742
8743
8744
8745
8746
8747
8748
8749
```



```
8750 struct cmd*
8751 parseblock(char **ps, char *es)
8752 {
8753   struct cmd *cmd;
8754
8755   if(!peek(ps, es, "("))
8756     panic("parseblock");
8757   gettoken(ps, es, 0, 0);
8758   cmd = parseline(ps, es);
8759   if(!peek(ps, es, ")"))
8760     panic("syntax - missing )");
8761   gettoken(ps, es, 0, 0);
8762   cmd = parseredirs(cmd, ps, es);
8763   return cmd;
8764 }
8765
8766 struct cmd*
8767 parseexec(char **ps, char *es)
8768 {
8769   char *q, *eq;
8770   int tok, argc;
8771   struct execcmd *cmd;
8772   struct cmd *ret;
8773
8774   if(peek(ps, es, "("))
8775     return parseblock(ps, es);
8776
8777   ret = execcmd();
8778   cmd = (struct execcmd*)ret;
8779
8780   argc = 0;
8781   ret = parseredirs(ret, ps, es);
8782   while(!peek(ps, es, "|)&;")){
8783     if((tok=gettoken(ps, es, &q, &eq)) == 0)
8784       break;
8785     if(tok != 'a')
8786       panic("syntax");
8787     cmd->argv[argc] = q;
8788     cmd->eargv[argc] = eq;
8789     argc++;
8790     if(argc >= MAXARGS)
8791       panic("too many args");
8792     ret = parseredirs(ret, ps, es);
8793   }
8794   cmd->argv[argc] = 0;
8795   cmd->eargv[argc] = 0;
8796   return ret;
8797 }
8798
8799
```

```
8800 // NUL-terminate all the counted strings.
8801 struct cmd*
8802 nulterminate(struct cmd *cmd)
8803 {
8804   int i;
8805   struct backcmd *bcmd;
8806   struct execcmd *ecmd;
8807   struct listcmd *lcmd;
8808   struct pipecmd *pcmd;
8809   struct redircmd *rcmd;
8810
8811   if(cmd == 0)
8812     return 0;
8813
8814   switch(cmd->type){
8815   case EXEC:
8816     ecmd = (struct execcmd*)cmd;
8817     for(i=0; ecmd->argv[i]; i++)
8818       *ecmd->eargv[i] = 0;
8819     break;
8820
8821   case REDIR:
8822     rcmd = (struct redircmd*)cmd;
8823     nulterminate(rcmd->cmd);
8824     *rcmd->efile = 0;
8825     break;
8826
8827   case PIPE:
8828     pcmd = (struct pipecmd*)cmd;
8829     nulterminate(pcmd->left);
8830     nulterminate(pcmd->right);
8831     break;
8832
8833   case LIST:
8834     lcmd = (struct listcmd*)cmd;
8835     nulterminate(lcmd->left);
8836     nulterminate(lcmd->right);
8837     break;
8838
8839   case BACK:
8840     bcmd = (struct backcmd*)cmd;
8841     nulterminate(bcmd->cmd);
8842     break;
8843   }
8844   return cmd;
8845 }
8846
8847
8848
8849
```



```
8850 #include "asm.h"
8851 #include "memlayout.h"
8852 #include "mmu.h"
8853
8854 # Start the first CPU: switch to 32-bit protected mode, jump into C.
8855 # The BIOS loads this code from the first sector of the hard disk into
8856 # memory at physical address 0x7c00 and starts executing in real mode
8857 # with %cs=0 %ip=7c00.
8858
8859 .code16                       # Assemble for 16-bit mode
8860 .globl start
8861 start:
8862   cli                         # BIOS enabled interrupts; disable
8863
8864   # Zero data segment registers DS, ES, and SS.
8865   xorw    %ax,%ax             # Set %ax to zero
8866   movw    %ax,%ds             # -> Data Segment
8867   movw    %ax,%es             # -> Extra Segment
8868   movw    %ax,%ss             # -> Stack Segment
8869
8870   # Physical address line A20 is tied to zero so that the first PCs
8871   # with 2 MB would run software that assumed 1 MB.  Undo that.
8872 seta20.1:
8873   inb     $0x64,%al               # Wait for not busy
8874   testb   $0x2,%al
8875   jnz     seta20.1
8876
8877   movb    $0xd1,%al               # 0xd1 -> port 0x64
8878   outb    %al,$0x64
8879
8880 seta20.2:
8881   inb     $0x64,%al               # Wait for not busy
8882   testb   $0x2,%al
8883   jnz     seta20.2
8884
8885   movb    $0xdf,%al               # 0xdf -> port 0x60
8886   outb    %al,$0x60
8887
8888   # Switch from real to protected mode.  Use a bootstrap GDT that makes
8889   # virtual addresses map directly to physical addresses so that the
8890   # effective memory map doesn’t change during the transition.
8891   lgdt    gdtdesc
8892   movl    %cr0, %eax
8893   orl     $CR0_PE, %eax
8894   movl    %eax, %cr0
8895
8896   # Complete transition to 32-bit protected mode by using long jmp
8897   # to reload %cs and %eip.  The segment descriptors are set up with no
8898   # translation, so that the mapping is still the identity mapping.
8899   ljmp    $(SEG_KCODE<<3), $start32
```

```
8900 .code32  # Tell assembler to generate 32-bit code now.
8901 start32:
8902   # Set up the protected-mode data segment registers
8903   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
8904   movw    %ax, %ds                # -> DS: Data Segment
8905   movw    %ax, %es                # -> ES: Extra Segment
8906   movw    %ax, %ss                # -> SS: Stack Segment
8907   movw    $0, %ax                 # Zero segments not ready for use
8908   movw    %ax, %fs                # -> FS
8909   movw    %ax, %gs                # -> GS
8910
8911   # Set up the stack pointer and call into C.
8912   movl    $start, %esp
8913   call    bootmain
8914
8915   # If bootmain returns (it shouldn't), trigger a Bochs
8916   # breakpoint if running under Bochs, then loop.
8917   movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
8918   movw    %ax, %dx
8919   outw    %ax, %dx
8920   movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
8921   outw    %ax, %dx
8922 spin:
8923   jmp     spin
8924
8925 # Bootstrap GDT
8926 .p2align 2                                # force 4 byte alignment
8927 gdt:
8928   SEG_NULLASM                             # null seg
8929   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
8930   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
8931
8932 gdtdesc:
8933   .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
8934   .long   gdt                             # address gdt
8935
8936
8937
8938
8939
8940
8941
8942
8943
8944
8945
8946
8947
8948
8949
```



```
8950 // Boot loader.
8951 //
8952 // Part of the boot block, along with bootasm.S, which calls bootmain().
8953 // bootasm.S has put the processor into protected 32-bit mode.
8954 // bootmain() loads an ELF kernel image from the disk starting at
8955 // sector 1 and then jumps to the kernel entry routine.
8956
8957 #include "types.h"
8958 #include "elf.h"
8959 #include "x86.h"
8960 #include "memlayout.h"
8961
8962 #define SECTSIZE  512
8963
8964 void readseg(uchar*, uint, uint);
8965
8966 void
8967 bootmain(void)
8968 {
8969   struct elfhdr *elf;
8970   struct proghdr *ph, *eph;
8971   void (*entry)(void);
8972   uchar* pa;
8973
8974   elf = (struct elfhdr*)0x10000;  // scratch space
8975
8976   // Read 1st page off disk
8977   readseg((uchar*)elf, 4096, 0);
8978
8979   // Is this an ELF executable?
8980   if(elf->magic != ELF_MAGIC)
8981     return;  // let bootasm.S handle error
8982
8983   // Load each program segment (ignores ph flags).
8984   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
8985   eph = ph + elf->phnum;
8986   for(; ph < eph; ph++){
8987     pa = (uchar*)ph->paddr;
8988     readseg(pa, ph->filesz, ph->off);
8989     if(ph->memsz > ph->filesz)
8990       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
8991   }
8992
8993   // Call the entry point from the ELF header.
8994   // Does not return!
8995   entry = (void(*)(void))(elf->entry);
8996   entry();
8997 }
8998
8999
```

```
9000 void
9001 waitdisk(void)
9002 {
9003   // Wait for disk ready.
9004   while((inb(0x1F7) & 0xC0) != 0x40)
9005     ;
9006 }
9007
9008 // Read a single sector at offset into dst.
9009 void
9010 readsect(void *dst, uint offset)
9011 {
9012   // Issue command.
9013   waitdisk();
9014   outb(0x1F2, 1);   // count = 1
9015   outb(0x1F3, offset);
9016   outb(0x1F4, offset >> 8);
9017   outb(0x1F5, offset >> 16);
9018   outb(0x1F6, (offset >> 24) | 0xE0);
9019   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
9020
9021   // Read data.
9022   waitdisk();
9023   insl(0x1F0, dst, SECTSIZE/4);
9024 }
9025
9026 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
9027 // Might copy more than asked.
9028 void
9029 readseg(uchar* pa, uint count, uint offset)
9030 {
9031   uchar* epa;
9032
9033   epa = pa + count;
9034
9035   // Round down to sector boundary.
9036   pa -= offset % SECTSIZE;
9037
9038   // Translate from bytes to sectors; kernel starts at sector 1.
9039   offset = (offset / SECTSIZE) + 1;
9040
9041   // If this is too slow, we could read lots of sectors at a time.
9042   // We'd write more to memory than asked, but it doesn't matter --
9043   // we load in increasing order.
9044   for(; pa < epa; pa += SECTSIZE, offset++)
9045     readsect(pa, offset);
9046 }
9047
9048
9049
```



```
9050 #ifdef CS333_P4
9051 // this is an ugly series of if statements but it works
9052 void
9053 print_mode(struct stat* st)
9054 {
9055   switch (st->type) {
9056     case T_DIR: printf(1, "d"); break;
9057     case T_FILE: printf(1, "-"); break;
9058     case T_DEV: printf(1, "c"); break;
9059     default: printf(1, "?");
9060   }
9061
9062   if (st->mode.flags.u_r)
9063     printf(1, "r");
9064   else
9065     printf(1, "-");
9066
9067   if (st->mode.flags.u_w)
9068     printf(1, "w");
9069   else
9070     printf(1, "-");
9071
9072   if ((st->mode.flags.u_x) & (st->mode.flags.setuid))
9073     printf(1, "S");
9074   else if (st->mode.flags.u_x)
9075     printf(1, "x");
9076   else
9077     printf(1, "-");
9078
9079   if (st->mode.flags.g_r)
9080     printf(1, "r");
9081   else
9082     printf(1, "-");
9083
9084   if (st->mode.flags.g_w)
9085     printf(1, "w");
9086   else
9087     printf(1, "-");
9088
9089   if (st->mode.flags.g_x)
9090     printf(1, "x");
9091   else
9092     printf(1, "-");
9093
9094   if (st->mode.flags.o_r)
9095     printf(1, "r");
9096   else
9097     printf(1, "-");
9098
9099
```

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

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

  return;
}
#endif
```



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

//This is a SHELL PROGRAM that should only
//be used to EXECUTE SYSTEM CALLS

int main (int argc, char* argv[]){


    //contains all the pieces of time
    //with resolution of one second
    struct rtcdate r;

    if (date(&r)) {

        printf(2, "Date_failed\n");
        exit();
    }

    if( date(&r) == 0){
        printf(1, "day: %d month: %d year: %d \t hour: %d minute: %d second:
        exit();
    }
}
```

```
9200 #include "types.h"
9201 #include "user.h"
9202
9203 //This is a SHELL PROGRAM that should only
9204 //be used to EXECUTE SYSTEM CALLS
9205
9206 int main (int argc, char* argv[]){
9207
9208
9209   if( argc < 2)
9210       printf(1, "Usage: Report runtime of programs provided as arguments, no
9211
9212
9213   uint start = (uint)uptime();
9214   uint finish = 0;
9215   uint pid = fork();
9216
9217   if(pid == 0 ){
9218
9219       if(exec(argv[1], argv +1)){
9220           printf(1, "Exec Failed");
9221           exit();
9222       }
9223   }
9224   else{
9225     wait();
9226   }
9227
9228     finish  = (uint)uptime();
9229     printf(1, "%s took %d.%d time to run\n", argv[1], (finish - start) / 100,
9230
9231   exit();
9232 }
9233
9234
9235
9236
9237
9238
9239
9240
9241
9242
9243
9244
9245
9246
9247
9248
9249
```



```
9250 #include "types.h"
9251 #include "user.h"
9252 #include "uproc.h"
9253
9254 //This is a SHELL PROGRAM that should only
9255 //be used to EXECUTE SYSTEM CALLS
9256
9257 int main (int argc, char* argv[]){
9258
9259
9260     //contains all the pieces of time
9261     //with resolution of one second
9262     struct uproc* up;
9263     int MAX = 64;
9264     char  *n = "Name",
9265           *p = "PID",
9266           *u = "UID",
9267           *g = "GID",
9268           *pp = "PPID",
9269           *tot = "CPU (s)",
9270           *e = "Elapsed (s)",
9271           *st = "State",
9272           *si = "Size";
9273 //          *pr = "Priority";
9274
9275     up =  malloc( sizeof(&up) * MAX);
9276     MAX =  getprocs(MAX, up);
9277
9278     if(MAX < 0){
9279         printf(1, "getprocs failed\n");
9280     exit();
9281     }
9282
9283
9284     printf(1, "%s\t | %s\t | %s\t | %s\t | %s\t | %s | %s | %s\t | %s\t | \n"
9285                 n, p,u,g,pp, tot,e,st,si);
9286     for(int  i = 0; i < MAX; i++){
9287
9288        printf(1, " %s\t | %d\t | %d\t | %d\t | %d\t |  %d.%d\t | %d.%d\t | %s
9289                      up[i].name, up[i].pid, up[i].uid, up[i].gid,
9290                      up[i].ppid,  up[i].CPU_total_ticks / 100, up[i].CPU_tota
9291                      up[i].elapsed_ticks / 100, up[i].elapsed_ticks % 100,
9292                      up[i].state, up[i].size);
9293
9294     }
9295
9296     exit();
9297 }
9298
9299
```

```
9300 struct queue {
9301
9302     struct proc *head;      // Head, Take Process from here
9303     struct proc *tail;      // Tail, Add Proccess to here
9304 };
9305
9306
9307
9308
9309
9310
9311
9312
9313
9314
9315
9316
9317
9318
9319
9320
9321
9322
9323
9324
9325
9326
9327
9328
9329
9330
9331
9332
9333
9334
9335
9336
9337
9338
9339
9340
9341
9342
9343
9344
9345
9346
9347
9348
9349
```