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

ACKNOWLEDGMENTS

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

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

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

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

ERROR REPORTS

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

BUILDING AND RUNNING XV6

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

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

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

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

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

    swtch 2658
        0374 2428 2466 2657 2658

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

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



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

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



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

```
0300 struct buf;
0301 struct context;
0302 struct file;
0303 struct inode;
0304 struct pipe;
0305 struct proc;
0306 struct rtcdate;
0307 struct spinlock;
0308 struct stat;
0309 struct superblock;
0310 #ifdef CS333_P2
0311 struct uproc;
0312 #endif
0313
0314
0315 // bio.c
0316 void            binit(void);
0317 struct buf*     bread(uint, uint);
0318 void            brelse(struct buf*);
0319 void            bwrite(struct buf*);
0320 // console.c
0321 void            consoleinit(void);
0322 void            cprintf(char*, ...);
0323 void            consoleintr(int(*)(void));
0324 void            panic(char*) __attribute__((noreturn));
0325
0326 // exec.c
0327 int             exec(char*, char**);
0328
0329 // file.c
0330 struct file*    filealloc(void);
0331 void            fileclose(struct file*);
0332 struct file*    filedup(struct file*);
0333 void            fileinit(void);
0334 int             fileread(struct file*, char*, int n);
0335 int             filestat(struct file*, struct stat*);
0336 int             filewrite(struct file*, char*, int n);
0337
0338 // fs.c
0339 void            readsb(int dev, struct superblock *sb);
0340 int             dirlink(struct inode*, char*, uint);
0341 struct inode*   dirlookup(struct inode*, char*, uint*);
0342 struct inode*   ialloc(uint, short);
0343 struct inode*   idup(struct inode*);
0344 void            iinit(int dev);
0345 void            ilock(struct inode*);
0346 void            iput(struct inode*);
0347 void            iunlock(struct inode*);
0348 void            iunlockput(struct inode*);
0349 void            iupdate(struct inode*);
```



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

```
0400 // pipe.c
0401 int             pipealloc(struct file**, struct file**);
0402 void            pipeclose(struct pipe*, int);
0403 int             piperead(struct pipe*, char*, int);
0404 int             pipewrite(struct pipe*, char*, int);
0405
0406 // proc.c
0407 struct proc*    copyproc(struct proc*);
0408 void            exit(void);
0409 int             fork(void);
0410 int             growproc(int);
0411 int             kill(int);
0412 void            pinit(void);
0413 void            procdump(void);
0414 void            scheduler(void) __attribute__((noreturn));
0415 void            sched(void);
0416 void            sleep(void*, struct spinlock*);
0417 void            userinit(void);
0418 int             wait(void);
0419 void            wakeup(void*);
0420 void            yield(void);
0421 #ifdef CS333_P2
0422 int                                                    getprocs(uint max,
0423 #endif
0424 #ifdef CS333_P3
0425 int                                                    setpriority(int p:
0426 #endif
0427
0428 // swtch.S
0429 void            swtch(struct context**, struct context*);
0430
0431 // spinlock.c
0432 void            acquire(struct spinlock*);
0433 void            getcallerpcs(void*, uint*);
0434 int             holding(struct spinlock*);
0435 void            initlock(struct spinlock*, char*);
0436 void            release(struct spinlock*);
0437 void            pushcli(void);
0438 void            popcli(void);
0439
0440 // string.c
0441 int             memcmp(const void*, const void*, uint);
0442 void*           memmove(void*, const void*, uint);
0443 void*           memset(void*, int, uint);
0444 char*           safestrcpy(char*, const char*, int);
0445 int             strlen(const char*);
0446 int             strncmp(const char*, const char*, uint);
0447 char*           strncpy(char*, const char*, int);
0448
0449
```



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

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



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

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



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

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



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

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



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

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



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

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



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

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



```
1150 #include "asm.h"
1151 #include "memlayout.h"
1152 #include "mmu.h"
1153
1154 # Each non-boot CPU ("AP") is started up in response to a STARTUP
1155 # IPI from the boot CPU.  Section B.4.2 of the Multi-Processor
1156 # Specification says that the AP will start in real mode with CS:IP
1157 # set to XY00:0000, where XY is an 8-bit value sent with the
1158 # STARTUP. Thus this code must start at a 4096-byte boundary.
1159 #
1160 # Because this code sets DS to zero, it must sit
1161 # at an address in the low 2^16 bytes.
1162 #
1163 # Startothers (in main.c) sends the STARTUPs one at a time.
1164 # It copies this code (start) at 0x7000.  It puts the address of
1165 # a newly allocated per-core stack in start-4,the address of the
1166 # place to jump to (mpenter) in start-8, and the physical address
1167 # of entrypgdir in start-12.
1168 #
1169 # This code is identical to bootasm.S except:
1170 #   - it does not need to enable A20
1171 #   - it uses the address at start-4, start-8, and start-12
1172
1173 .code16
1174 .globl start
1175 start:
1176   cli
1177
1178   xorw    %ax,%ax
1179   movw    %ax,%ds
1180   movw    %ax,%es
1181   movw    %ax,%ss
1182
1183   lgdt    gdtdesc
1184   movl    %cr0, %eax
1185   orl     $CR0_PE, %eax
1186   movl    %eax, %cr0
1187
1188   ljmpl    $(SEG_KCODE<<3), $(start32)
1189
1190 .code32
1191 start32:
1192   movw    $(SEG_KDATA<<3), %ax
1193   movw    %ax, %ds
1194   movw    %ax, %es
1195   movw    %ax, %ss
1196   movw    $0, %ax
1197   movw    %ax, %fs
1198   movw    %ax, %gs
1199
```

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



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

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



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

```
1400 // Mutual exclusion lock.
1401 struct spinlock {
1402   uint locked;       // Is the lock held?
1403
1404   // For debugging:
1405   char *name;        // Name of lock.
1406   struct cpu *cpu;   // The cpu holding the lock.
1407   uint pcs[10];      // The call stack (an array of program counters)
1408                      // that locked the lock.
1409 };
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
```



```
1450 // Mutual exclusion spin locks.
1451
1452 #include "types.h"
1453 #include "defs.h"
1454 #include "param.h"
1455 #include "x86.h"
1456 #include "memlayout.h"
1457 #include "mmu.h"
1458 #include "proc.h"
1459 #include "spinlock.h"
1460
1461 void
1462 initlock(struct spinlock *lk, char *name)
1463 {
1464   lk->name = name;
1465   lk->locked = 0;
1466   lk->cpu = 0;
1467 }
1468
1469 // Acquire the lock.
1470 // Loops (spins) until the lock is acquired.
1471 // Holding a lock for a long time may cause
1472 // other CPUs to waste time spinning to acquire it.
1473 void
1474 acquire(struct spinlock *lk)
1475 {
1476   pushcli(); // disable interrupts to avoid deadlock.
1477   if(holding(lk))
1478     panic("acquire");
1479
1480   // The xchg is atomic.
1481   // It also serializes, so that reads after acquire are not
1482   // reordered before it.
1483   while(xchg(&lk->locked, 1) != 0)
1484     ;
1485
1486   // Record info about lock acquisition for debugging.
1487   lk->cpu = cpu;
1488   getcallerpcs(&lk, lk->pcs);
1489 }
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
```

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



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

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



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

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



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

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



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

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



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

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



```
2050 // Segments in proc->gdt.
2051 #define NSEGS     7
2052
2053 // Default UID and GID for init
2054 #define INITUID     0
2055 #define INITGID     0
2056
2057 // Default number of ready processes list
2058 #define NUM_READY_LISTS    7
2059 // Default starting priority number
2060 #define PRIORITY_HIGH      0
2061 // Default lowest priority number
2062 #define PRIORITY_LOW       PRIORITY_HIGH+NUM_READY_LISTS-1
2063 // Default promotion interval
2064 #define TICKS_TO_PROMOTE 200
2065 // Default process budget
2066 #define BUDGET 400
2067
2068 // Per-CPU state
2069 struct cpu {
2070   uchar id;                    // Local APIC ID; index into cpus[] below
2071   struct context *scheduler;   // swtch() here to enter scheduler
2072   struct taskstate ts;         // Used by x86 to find stack for interrupt
2073   struct segdesc gdt[NSEGS];   // x86 global descriptor table
2074   volatile uint started;       // Has the CPU started?
2075   int ncli;                    // Depth of pushcli nesting.
2076   int intena;                  // Were interrupts enabled before pushcli?
2077
2078   // Cpu-local storage variables; see below
2079   struct cpu *cpu;
2080   struct proc *proc;           // The currently-running process.
2081 };
2082
2083 extern struct cpu cpus[NCPU];
2084 extern int ncpu;
2085
2086 // Per-CPU variables, holding pointers to the
2087 // current cpu and to the current process.
2088 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2089 // and "%gs:4" to refer to proc.  seginit sets up the
2090 // %gs segment register so that %gs refers to the memory
2091 // holding those two variables in the local cpu's struct cpu.
2092 // This is similar to how thread-local variables are implemented
2093 // in thread libraries such as Linux pthreads.
2094 extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
2095 extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc
2096
2097
2098
2099
```

```
2100 // Saved registers for kernel context switches.
2101 // Don't need to save all the segment registers (%cs, etc),
2102 // because they are constant across kernel contexts.
2103 // Don't need to save %eax, %ecx, %edx, because the
2104 // x86 convention is that the caller has saved them.
2105 // Contexts are stored at the bottom of the stack they
2106 // describe; the stack pointer is the address of the context.
2107 // The layout of the context matches the layout of the stack in swtch.S
2108 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2109 // but it is on the stack and allocproc() manipulates it.
2110 struct context {
2111   uint edi;
2112   uint esi;
2113   uint ebx;
2114   uint ebp;
2115   uint eip;
2116 };
2117
2118 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2119
2120 // Per-process state
2121 struct proc {
2122   uint sz;                     // Size of process memory (bytes)
2123   pde_t* pgdir;                // Page table
2124   char *kstack;                // Bottom of kernel stack for this process
2125   enum procstate state;        // Process state
2126   uint pid;                    // Process ID
2127   struct proc *parent;         // Parent process
2128   struct trapframe *tf;        // Trap frame for current syscall
2129   struct context *context;     // swtch() here to run process
2130   void *chan;                  // If non-zero, sleeping on chan
2131   int killed;                  // If non-zero, have been killed
2132   struct file *ofile[NOFILE];  // Open files
2133   struct inode *cwd;           // Current directory
2134   char name[16];               // Process name (debugging)
2135   uint start_ticks;                  // Start ticks (debugging)
2136 #ifdef CS333_P2
2137    uint cpu_ticks_total;                          // Total elapsed ticks i
2138    uint cpu_ticks_in;                             // Ticks when scheduled
2139   uint uid;                    // Process owner's user id
2140   uint gid;                    // Process owner's group id
2141 #endif
2142
2143 #ifdef CS333_P3
2144    int priority;                                          // Process prior
2145    int budget;                                                   // A pro
2146    struct proc *next;                       // Next process in the process l
2147 #endif
2148 };
2149
```



```
2150 // Process memory is laid out contiguously, low addresses first:
2151 //   text
2152 //   original data and bss
2153 //   fixed-size stack
2154 //   expandable heap
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
```

```
2200 #include "types.h"
2201 #include "defs.h"
2202 #include "param.h"
2203 #include "memlayout.h"
2204 #include "mmu.h"
2205 #include "x86.h"
2206 #include "proc.h"
2207 #include "spinlock.h"
2208 #include "uproc.h"
2209
2210 struct {
2211   struct spinlock lock;
2212   struct proc proc[NPROC];
2213 #ifdef CS333_P3
2214    struct proc *pReadyList[NUM_READY_LISTS];
2215    struct proc *pFreeList;
2216    uint PromoteAtTime;
2217 #endif
2218 } ptable;
2219
2220 static struct proc *initproc;
2221
2222 int nextpid = 1;
2223 extern void forkret(void);
2224 extern void trapret(void);
2225
2226 static void wakeup1(void *chan);
2227
2228 void
2229 pinit(void)
2230 {
2231   initlock(&ptable.lock, "ptable");
2232 }
2233
2234 #ifdef CS333_P3
2235 // Pops a process off a process queue
2236 // Return -1 if no process in the queue
2237 static struct proc*
2238 popq(struct proc **proclist)
2239 {
2240   if(!holding(&ptable.lock))
2241     panic("popq ptable.lock\n");
2242    if(proclist <= 0 || *proclist <= 0) return 0;
2243    struct proc *ret;
2244    ret = *proclist;
2245    *proclist = (*proclist)->next;
2246    ret->next = 0;
2247    return ret;
2248 }
2249
```



```
2250 // Pushs a process to the pFreeList
2251 static void
2252 pushfreeq(struct proc* input, struct proc **freelist)
2253 {
2254   if(!holding(&ptable.lock))
2255     panic("pushfreeq ptable.lock\n");
2256    else {
2257            input->next = *freelist;
2258            *freelist = input;
2259    }
2260 }
2261
2262 // Pushs a process to the pReadyList
2263 static void
2264 pushreadyq(struct proc* input, struct proc **readylist)
2265 {
2266   if(!holding(&ptable.lock))
2267     panic("pushreadyq ptable.lock\n");
2268    if(!input)
2269            return;
2270    if(!*readylist) {
2271            input->next = 0;
2272            *readylist = input;
2273    }
2274    else {
2275            struct proc* temp = *readylist;
2276            while(temp->next)
2277                    temp = temp->next;
2278            temp->next = input;
2279            input->next = 0;
2280    }
2281 }
2282
2283 // Set process's priority to specified value
2284 // Return 0 if success
2285 // Assumes holding ptable lock
2286 int
2287 setpriority(int pid, int priority)
2288 {
2289    if(pid < 0)
2290     panic("pid out of bound\n");
2291    if(priority < PRIORITY_HIGH || priority > PRIORITY_LOW) {
2292            cprintf("Invalid priority value: %d, need an int between %d and %
2293            return -1;
2294    }
2295
2296    acquire(&ptable.lock);
2297   struct proc *p;
2298    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2299            if(p->pid == pid) {
```

```
2300                    p->priority = priority;
2301                    p->budget = BUDGET;
2302                    release(&ptable.lock);
2303                    return 0;
2304    }
2305    cprintf("Invalid pid: %d\n",pid);
2306    release(&ptable.lock);
2307    return -1;
2308 }
2309 #endif
2310
2311
2312 // Look in the process table for an UNUSED proc.
2313 // If found, change state to EMBRYO and initialize
2314 // state required to run in the kernel.
2315 // Otherwise return 0.
2316 static struct proc*
2317 allocproc(void)
2318 {
2319   struct proc *p;
2320   char *sp;
2321
2322 #ifndef CS333_P3
2323   acquire(&ptable.lock);
2324   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2325     if(p->state == UNUSED)
2326       goto found;
2327   release(&ptable.lock);
2328 #else
2329
2330   acquire(&ptable.lock);
2331    p = popq(&ptable.pFreeList);
2332    if(p && p->state == UNUSED)
2333            goto found;
2334   release(&ptable.lock);
2335 #endif
2336   return 0;
2337
2338 found:
2339   p->state = EMBRYO;
2340   p->pid = nextpid++;
2341   release(&ptable.lock);
2342
2343   // Allocate kernel stack.
2344   if((p->kstack = kalloc()) == 0){
2345     p->state = UNUSED;
2346            acquire(&ptable.lock);
2347            pushfreeq(p, &ptable.pFreeList);
2348            release(&ptable.lock);
2349     return 0;
```



```
2350   }
2351   sp = p->kstack + KSTACKSIZE;
2352
2353   // Leave room for trap frame.
2354   sp -= sizeof *p->tf;
2355   p->tf = (struct trapframe*)sp;
2356
2357   // Set up new context to start executing at forkret,
2358   // which returns to trapret.
2359   sp -= 4;
2360   *(uint*)sp = (uint)trapret;
2361
2362   sp -= sizeof *p->context;
2363   p->context = (struct context*)sp;
2364   memset(p->context, 0, sizeof *p->context);
2365   p->context->eip = (uint)forkret;
2366
2367   acquire(&tickslock);
2368   p->start_ticks = ticks;
2369   release(&tickslock);
2370    p->cpu_ticks_in = 0;
2371
2372   return p;
2373 }
2374
2375 // Check if it's time to promote
2376 // Assume alway hold the lock
2377 // return 1 if it's time to promote
2378 #ifdef CS333_P3
2379 static int
2380 timetopromote(void)
2381 {
2382   if(!holding(&ptable.lock))
2383     panic("timetopromote ptable.lock");
2384    acquire(&tickslock);
2385    if(ticks < ptable.PromoteAtTime) {
2386      release(&tickslock);
2387            return 0; // Not time to promote
2388    }
2389    ptable.PromoteAtTime = ticks + TICKS_TO_PROMOTE;
2390    release(&tickslock);
2391   return 1;
2392 }
2393
2394
2395
2396
2397
2398
2399
```

```
2400 #endif
2401
2402 // Set up first user process.
2403 void
2404 userinit(void)
2405 {
2406   struct proc *p;
2407   extern char _binary_initcode_start[], _binary_initcode_size[];
2408
2409 #ifdef CS333_P3
2410    acquire(&ptable.lock);
2411    ptable.pFreeList = 0;
2412    // Initialize freelist by putting UNUSED processes to the list
2413   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2414     if(p->state == UNUSED)
2415                   pushfreeq(p, &ptable.pFreeList);
2416    // Initialize readylist to empty
2417    int i;
2418    for(i = PRIORITY_HIGH; i < NUM_READY_LISTS; ++i) {
2419            ptable.pReadyList[i] = 0;
2420    }
2421    release(&ptable.lock);
2422 #endif
2423
2424   p = allocproc();
2425   initproc = p;
2426   if((p->pgdir = setupkvm()) == 0)
2427     panic("userinit: out of memory?");
2428   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2429   p->sz = PGSIZE;
2430   memset(p->tf, 0, sizeof(*p->tf));
2431   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2432   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2433   p->tf->es = p->tf->ds;
2434   p->tf->ss = p->tf->ds;
2435   p->tf->eflags = FL_IF;
2436   p->tf->esp = PGSIZE;
2437   p->tf->eip = 0;  // beginning of initcode.S
2438
2439   safestrcpy(p->name, "initcode", sizeof(p->name));
2440   p->cwd = namei("/");
2441
2442   p->state = RUNNABLE;
2443 #ifdef CS333_P3
2444    acquire(&ptable.lock);
2445    ptable.pReadyList[PRIORITY_HIGH] = p;
2446    p->next = 0;
2447    release(&ptable.lock);
2448 #endif
2449
```



```
2450 #ifdef CS333_P2
2451    p->uid = INITUID;
2452    p->gid = INITGID;
2453 #endif
2454 }
2455
2456 // Grow current process's memory by n bytes.
2457 // Return 0 on success, -1 on failure.
2458 int
2459 growproc(int n)
2460 {
2461   uint sz;
2462
2463   sz = proc->sz;
2464   if(n > 0){
2465     if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2466       return -1;
2467   } else if(n < 0){
2468     if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2469       return -1;
2470   }
2471   proc->sz = sz;
2472   switchuvm(proc);
2473   return 0;
2474 }
2475
2476 // Create a new process copying p as the parent.
2477 // Sets up stack to return as if from system call.
2478 // Caller must set state of returned proc to RUNNABLE.
2479 int
2480 fork(void)
2481 {
2482   int i, pid;
2483   struct proc *np;
2484
2485   // Allocate process.
2486   if((np = allocproc()) == 0)
2487     return -1;
2488
2489   // Copy process state from p.
2490   if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2491     kfree(np->kstack);
2492     np->kstack = 0;
2493     np->state = UNUSED;
2494 #ifdef CS333_P3
2495             pushfreeq(np, &ptable.pFreeList);
2496 #endif
2497     return -1;
2498   }
2499   np->sz = proc->sz;
```

```
2500   np->parent = proc;
2501   *np->tf = *proc->tf;
2502
2503 #ifdef CS333_P2
2504    // Copy process UID, GID
2505    np->uid = proc->uid;
2506    np->gid = proc->gid;
2507 #endif
2508
2509   // Clear %eax so that fork returns 0 in the child.
2510   np->tf->eax = 0;
2511
2512   for(i = 0; i < NOFILE; i++)
2513     if(proc->ofile[i])
2514       np->ofile[i] = filedup(proc->ofile[i]);
2515   np->cwd = idup(proc->cwd);
2516
2517   safestrcpy(np->name, proc->name, sizeof(proc->name));
2518
2519   pid = np->pid;
2520
2521   // lock to force the compiler to emit the np->state write last.
2522   acquire(&ptable.lock);
2523   np->state = RUNNABLE;
2524 #ifdef CS333_P3
2525    np->priority = PRIORITY_HIGH;
2526    np->budget = BUDGET;
2527   pushreadyq(np, &ptable.pReadyList[PRIORITY_HIGH]);
2528 #endif
2529   release(&ptable.lock);
2530
2531   return pid;
2532 }
2533
2534 // Exit the current process.  Does not return.
2535 // An exited process remains in the zombie state
2536 // until its parent calls wait() to find out it exited.
2537 void
2538 exit(void)
2539 {
2540   struct proc *p;
2541   int fd;
2542
2543   if(proc == initproc)
2544     panic("init exiting");
2545
2546
2547
2548
2549
```



```
2550   // Close all open files.
2551   for(fd = 0; fd < NOFILE; fd++){
2552     if(proc->ofile[fd]){
2553       fileclose(proc->ofile[fd]);
2554       proc->ofile[fd] = 0;
2555     }
2556   }
2557
2558   begin_op();
2559   iput(proc->cwd);
2560   end_op();
2561   proc->cwd = 0;
2562
2563   acquire(&ptable.lock);
2564
2565   // Parent might be sleeping in wait().
2566   wakeup1(proc->parent);
2567
2568   // Pass abandoned children to init.
2569   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2570     if(p->parent == proc){
2571       p->parent = initproc;
2572       if(p->state == ZOMBIE)
2573         wakeup1(initproc);
2574     }
2575   }
2576
2577   // Jump into the scheduler, never to return.
2578   proc->state = ZOMBIE;
2579   sched();
2580   panic("zombie exit");
2581 }
2582
2583 // Wait for a child process to exit and return its pid.
2584 // Return -1 if this process has no children.
2585 int
2586 wait(void)
2587 {
2588   struct proc *p;
2589   int havekids, pid;
2590
2591   acquire(&ptable.lock);
2592   for(;;){
2593     // Scan through table looking for zombie children.
2594     havekids = 0;
2595     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2596       if(p->parent != proc)
2597         continue;
2598       havekids = 1;
2599       if(p->state == ZOMBIE){
```

```
2600         // Found one.
2601         pid = p->pid;
2602         kfree(p->kstack);
2603         p->kstack = 0;
2604         freevm(p->pgdir);
2605         p->state = UNUSED;
2606 #ifdef CS333_P3
2607                             pushfreeq(p, &ptable.pFreeList);
2608 #endif
2609         p->pid = 0;
2610         p->parent = 0;
2611         p->name[0] = 0;
2612         p->killed = 0;
2613         release(&ptable.lock);
2614         return pid;
2615       }
2616     }
2617
2618     // No point waiting if we don't have any children.
2619     if(!havekids || proc->killed){
2620       release(&ptable.lock);
2621       return -1;
2622     }
2623
2624     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2625     sleep(proc, &ptable.lock);
2626   }
2627 }
2628
2629 // Per-CPU process scheduler.
2630 // Each CPU calls scheduler() after setting itself up.
2631 // Scheduler never returns.  It loops, doing:
2632 //  - choose a process to run
2633 //  - swtch to start running that process
2634 //  - eventually that process transfers control
2635 //      via swtch back to the scheduler.
2636 #ifndef CS333_P3
2637 // original xv6 scheduler. Use if CS333_P3 NOT defined.
2638 void
2639 scheduler(void)
2640 {
2641   struct proc *p;
2642
2643   for(;;){
2644     // Enable interrupts on this processor.
2645     sti();
2646
2647
2648
2649
```



```
2650     // Loop over process table looking for process to run.
2651     acquire(&ptable.lock);
2652     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2653       if(p->state != RUNNABLE)
2654         continue;
2655
2656       // Switch to chosen process.  It is the process's job
2657       // to release ptable.lock and then reacquire it
2658       // before jumping back to us.
2659       proc = p;
2660       switchuvm(p);
2661       p->state = RUNNING;
2662 #ifdef CS333_P2
2663        acquire(&tickslock);
2664                   p->cpu_ticks_in = ticks;
2665                   release(&tickslock);
2666 #endif
2667       swtch(&cpu->scheduler, proc->context);
2668       switchkvm();
2669
2670       // Process is done running for now.
2671       // It should have changed its p->state before coming back.
2672       proc = 0;
2673     }
2674     release(&ptable.lock);
2675
2676   }
2677 }
2678
2679 #else
2680 // CS333_P3 MLFQ scheduler implementation goes here
2681 void
2682 scheduler(void)
2683 {
2684   struct proc *p;
2685
2686   for(;;){
2687           // Enable interrupts on this processor.
2688           sti();
2689
2690           // If promotion timer expires promote all processes one
2691           // level up
2692           acquire(&ptable.lock);
2693           if(timetopromote()) {
2694                   // Increase priority for Running, sleeping processes
2695                   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2696                           if(p->priority <= PRIORITY_HIGH)
2697                                   continue;
2698                           if(p->state == RUNNING || p->state == SLEEPING){
2699                                   p->budget = BUDGET;
```

```
2700                                    p->priority -= 1;
2701                            }
2702                    }
2703                    // Priority queue shift up
2704                    int priority;
2705                    for(priority = PRIORITY_HIGH; priority < PRIORITY_LOW; ++
2706                            cprintf("time to promote\n");
2707                            do {
2708                                    p = popq(&ptable.pReadyList[priority+1]);
2709                                    if(p) {
2710                                            p->priority -= 1;
2711                                            p->budget = BUDGET;
2712                                            pushreadyq(p, &ptable.pReadyList[
2713                                    }
2714                            }while(p);
2715                    }
2716            }
2717
2718            // Find the next runnable process and pop it from the ready
2719            // list
2720            int i;
2721            for(i = PRIORITY_HIGH; i < PRIORITY_LOW+1;) {
2722                    if(!ptable.pReadyList[i]){
2723                            ++i;
2724                            continue;
2725                    }
2726                    p = popq(&ptable.pReadyList[i]);
2727
2728                    if(!p) {
2729                            panic("poping an empty readylist");
2730                    }
2731
2732                    // Switch to chosen process.  It is the process's job
2733                    // to release ptable.lock and then reacquire it
2734                    // before jumping back to us.
2735                    proc = p;
2736                    switchuvm(p);
2737                    p->state = RUNNING;
2738 #ifdef CS333_P2
2739                    acquire(&tickslock);
2740                    p->cpu_ticks_in = ticks;
2741                    release(&tickslock);
2742 #endif
2743                    swtch(&cpu->scheduler, proc->context);
2744                    switchkvm();
2745
2746
2747
2748
2749
```



```
2750                    // Process is done running for now.
2751                    // It should have changed its p->state before coming back
2752                    proc = 0;
2753            }
2754            release(&ptable.lock);
2755    }
2756 }
2757 #endif
2758
2759 // Enter scheduler.  Must hold only ptable.lock
2760 // and have changed proc->state.
2761 void
2762 sched(void)
2763 {
2764   int intena;
2765
2766   if(!holding(&ptable.lock))
2767     panic("sched ptable.lock");
2768   if(cpu->ncli != 1)
2769     panic("sched locks");
2770   if(proc->state == RUNNING)
2771     panic("sched running");
2772   if(readeflags()&FL_IF)
2773     panic("sched interrible");
2774   intena = cpu->intena;
2775 #ifdef CS333_P2
2776    acquire(&tickslock);
2777    proc->cpu_ticks_total += ticks - proc->cpu_ticks_in;
2778 #ifdef CS333_P3
2779    proc->budget -= ticks - proc->cpu_ticks_in;
2780 #endif
2781    release(&tickslock);
2782 #endif
2783
2784 #ifdef CS333_P3
2785    // Check process's budget if its <= 0
2786    // demote to the next lower priority queue
2787    // else add it to the back of current queue.
2788    if(proc->budget <= 0 && proc->priority < PRIORITY_LOW){
2789                    proc->priority += 1;
2790                    proc->budget = BUDGET;
2791    }
2792    if(proc->state == RUNNABLE)
2793                    pushreadyq(proc, &ptable.pReadyList[proc->priority]);
2794 #endif
2795   swtch(&proc->context, cpu->scheduler);
2796   cpu->intena = intena;
2797 }
2798
2799
```

```
2800 // Give up the CPU for one scheduling round.
2801 void
2802 yield(void)
2803 {
2804   acquire(&ptable.lock);
2805   proc->state = RUNNABLE;
2806   sched();
2807   release(&ptable.lock);
2808 }
2809
2810 // A fork child's very first scheduling by scheduler()
2811 // will swtch here.  "Return" to user space.
2812 void
2813 forkret(void)
2814 {
2815   static int first = 1;
2816   // Still holding ptable.lock from scheduler.
2817   release(&ptable.lock);
2818
2819   if (first) {
2820     // Some initialization functions must be run in the context
2821     // of a regular process (e.g., they call sleep), and thus cannot
2822     // be run from main().
2823     first = 0;
2824     iinit(ROOTDEV);
2825     initlog(ROOTDEV);
2826   }
2827
2828   // Return to "caller", actually trapret (see allocproc).
2829 }
2830
2831 // Atomically release lock and sleep on chan.
2832 // Reacquires lock when awakened.
2833 void
2834 sleep(void *chan, struct spinlock *lk)
2835 {
2836   if(proc == 0)
2837     panic("sleep");
2838
2839   if(lk == 0)
2840     panic("sleep without lk");
2841
2842   // Must acquire ptable.lock in order to
2843   // change p->state and then call sched.
2844   // Once we hold ptable.lock, we can be
2845   // guaranteed that we won't miss any wakeup
2846   // (wakeup runs with ptable.lock locked),
2847   // so it's okay to release lk.
2848   if(lk != &ptable.lock){
2849     acquire(&ptable.lock);
```



```
2850     release(lk);
2851   }
2852
2853   // Go to sleep.
2854   proc->chan = chan;
2855   proc->state = SLEEPING;
2856   sched();
2857
2858   // Tidy up.
2859   proc->chan = 0;
2860
2861   // Reacquire original lock.
2862   if(lk != &ptable.lock){
2863     release(&ptable.lock);
2864     acquire(lk);
2865   }
2866 }
2867
2868 // Wake up all processes sleeping on chan.
2869 // The ptable lock must be held.
2870 static void
2871 wakeup1(void *chan)
2872 {
2873   struct proc *p;
2874
2875   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2876 #ifndef CS333_P3
2877     if(p->state == SLEEPING && p->chan == chan)
2878       p->state = RUNNABLE;
2879 #else
2880     if(p->state == SLEEPING && p->chan == chan) {
2881       p->state = RUNNABLE;
2882                   pushreadyq(p, &ptable.pReadyList[p->priority]);
2883           }
2884 #endif
2885 }
2886
2887 // Wake up all processes sleeping on chan.
2888 void
2889 wakeup(void *chan)
2890 {
2891   acquire(&ptable.lock);
2892   wakeup1(chan);
2893   release(&ptable.lock);
2894 }
2895
2896
2897
2898
2899
```

```
2900 // Kill the process with the given pid.
2901 // Process won't exit until it returns
2902 // to user space (see trap in trap.c).
2903 int
2904 kill(int pid)
2905 {
2906   struct proc *p;
2907
2908   acquire(&ptable.lock);
2909   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2910     if(p->pid == pid){
2911       p->killed = 1;
2912       // Wake process from sleep if necessary.
2913 #ifndef CS333_P3
2914       if(p->state == SLEEPING)
2915         p->state = RUNNABLE;
2916 #else
2917       if(p->state == SLEEPING) {
2918         p->state = RUNNABLE;
2919                           pushreadyq(p, &ptable.pReadyList[p->priority]);
2920                     }
2921 #endif
2922
2923       release(&ptable.lock);
2924       return 0;
2925     }
2926   }
2927   release(&ptable.lock);
2928   return -1;
2929 }
2930
2931 // Print a process listing to console.  For debugging.
2932 // Runs when user types ^P on console.
2933 // No lock to avoid wedging a stuck machine further.
2934 static void
2935 print_elapsed(struct proc *p)
2936 {
2937   uint temp = p->start_ticks;
2938   temp = ticks - temp;
2939   cprintf("%d.%d",temp/100, temp%100);
2940 #ifdef CS333_P2
2941   cprintf("  %d.%d",p->cpu_ticks_total/100, p->cpu_ticks_total%100);
2942   cprintf("    %d  ", p->uid);
2943   cprintf("  %d  ", p->gid);
2944   if(p->parent && p->pid != 1)
2945            cprintf("  %d  ", p->parent->pid);
2946    else
2947            cprintf("  %d  ", p->pid);
2948 #ifdef CS333_P3
2949    cprintf("  %d  ", p->priority);
```



```
2950 #endif
2951 #endif
2952 }
2953
2954 void
2955 procdump(void)
2956 {
2957   static char *states[] = {
2958   [UNUSED]    "unused",
2959   [EMBRYO]    "embryo",
2960   [SLEEPING]  "sleep ",
2961   [RUNNABLE]  "runble",
2962   [RUNNING]   "run   ",
2963   [ZOMBIE]    "zombie"
2964   };
2965   int i;
2966   struct proc *p;
2967   char *state;
2968   uint pc[10];
2969
2970 #ifdef CS333_P3
2971   cprintf("\nPID  State  Name  Elapsed    TotalCpuTime    UID    GID    PPID
2972 #else
2973 #ifdef CS333_P2
2974   cprintf("\nPID  State  Name  Elapsed    TotalCpuTime    UID    GID    PPID
2975 #else
2976    cprintf("\nPID  State  Name  Elapsed    PCs\n");
2977 #endif
2978 #endif
2979
2980   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2981     if(p->state == UNUSED)
2982       continue;
2983     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
2984       state = states[p->state];
2985     else
2986       state = "???";
2987     cprintf("%d    %s %s   ", p->pid, state, p->name);
2988     print_elapsed(p);
2989     if(p->state == SLEEPING){
2990       getcallerpcs((uint*)p->context->ebp+2, pc);
2991       for(i=0; i<10 && pc[i] != 0; i++)
2992         cprintf(" %p", pc[i]);
2993     }
2994     cprintf("\n");
2995   }
2996 }
2997
2998
2999
```

```
3000 #ifdef CS333_P2
3001 // Get process information
3002 int
3003 getprocs(uint max, struct uproc* table)
3004 {
3005    if(!table || max == 0) return -1;
3006   static char *states[] = {
3007   [UNUSED]    "unused",
3008   [EMBRYO]    "embryo",
3009   [SLEEPING]  "sleep ",
3010   [RUNNABLE]  "runble",
3011   [RUNNING]   "run   ",
3012   [ZOMBIE]    "zombie"
3013   };
3014
3015    int procscount = 0;
3016   struct proc *p;
3017    if(max > NPROC)
3018            max = NPROC;
3019    acquire(&ptable.lock);
3020    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3021            if(max <= 0) break; // break out of the loop if the max number of
3022            if(p->state == UNUSED || p->state == EMBRYO || p->state == ZOMBIE
3023                    continue;
3024            table->pid = p->pid;
3025            table->uid = p->uid;
3026            table->gid = p->gid;
3027            if(!p->parent || p->pid ==1)
3028                    table->ppid = p->pid;
3029            else
3030                    table->ppid = p->parent->pid;
3031 #ifdef CS333_P3
3032            table->priority = p->priority;
3033 #endif
3034            acquire(&tickslock);
3035            table->elapsed_ticks = ticks - p->start_ticks;
3036            table->CPU_total_ticks = p->cpu_ticks_total;
3037            release(&tickslock);
3038            safestrcpy(table->state, states[p->state], sizeof(table->state));
3039            table->size = p->sz;
3040            safestrcpy(table->name, p->name, sizeof(table->name));
3041            ++procscount;
3042            ++table;
3043            --max;
3044    }
3045    release(&ptable.lock);
3046
3047   return procscount;
3048 }
3049 #endif
```



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

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



```
3150 void
3151 freerange(void *vstart, void *vend)
3152 {
3153   char *p;
3154   p = (char*)PGROUNDUP((uint)vstart);
3155   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3156     kfree(p);
3157 }
3158
3159 // Free the page of physical memory pointed at by v,
3160 // which normally should have been returned by a
3161 // call to kalloc().  (The exception is when
3162 // initializing the allocator; see kinit above.)
3163 void
3164 kfree(char *v)
3165 {
3166   struct run *r;
3167
3168   if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
3169     panic("kfree");
3170
3171   // Fill with junk to catch dangling refs.
3172   memset(v, 1, PGSIZE);
3173
3174   if(kmem.use_lock)
3175     acquire(&kmem.lock);
3176   r = (struct run*)v;
3177   r->next = kmem.freelist;
3178   kmem.freelist = r;
3179   if(kmem.use_lock)
3180     release(&kmem.lock);
3181 }
3182
3183 // Allocate one 4096-byte page of physical memory.
3184 // Returns a pointer that the kernel can use.
3185 // Returns 0 if the memory cannot be allocated.
3186 char*
3187 kalloc(void)
3188 {
3189   struct run *r;
3190
3191   if(kmem.use_lock)
3192     acquire(&kmem.lock);
3193   r = kmem.freelist;
3194   if(r)
3195     kmem.freelist = r->next;
3196   if(kmem.use_lock)
3197     release(&kmem.lock);
3198   return (char*)r;
3199 }
```

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



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

```
#include "mmu.h"

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

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

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

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



```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "traps.h"
#include "spinlock.h"

// Interrupt descriptor table (shared by all CPUs).
struct gatedesc idt[256];
extern uint vectors[];  // in vectors.S: array of 256 entry pointers
struct spinlock tickslock;
uint ticks;

void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}

void
idtinit(void)
{
  lidt(idt, sizeof(idt));
}

void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(proc->killed)
      exit();
    proc->tf = tf;
    syscall();
    if(proc->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpu->id == 0){
```

```
3400      acquire(&tickslock);
3401      ticks++;
3402      release(&tickslock);    // NOTE: MarkM has reversed these two lines.
3403      wakeup(&ticks);         // wakeup() should not require the tickslock t
3404    }
3405    lapiceoi();
3406    break;
3407  case T_IRQ0 + IRQ_IDE:
3408    ideintr();
3409    lapiceoi();
3410    break;
3411  case T_IRQ0 + IRQ_IDE+1:
3412    // Bochs generates spurious IDE1 interrupts.
3413    break;
3414  case T_IRQ0 + IRQ_KBD:
3415    kbdintr();
3416    lapiceoi();
3417    break;
3418  case T_IRQ0 + IRQ_COM1:
3419    uartintr();
3420    lapiceoi();
3421    break;
3422  case T_IRQ0 + 7:
3423  case T_IRQ0 + IRQ_SPURIOUS:
3424    cprintf("cpu%d: spurious interrupt at %x:%x\n",
3425            cpu->id, tf->cs, tf->eip);
3426    lapiceoi();
3427    break;
3428
3429  default:
3430    if(proc == 0 || (tf->cs&3) == 0){
3431      // In kernel, it must be our mistake.
3432      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3433              tf->trapno, cpu->id, tf->eip, rcr2());
3434      panic("trap");
3435    }
3436    // In user space, assume process misbehaved.
3437    cprintf("pid %d %s: trap %d err %d on cpu %d "
3438            "eip 0x%x addr 0x%x--kill proc\n",
3439            proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
3440            rcr2());
3441    proc->killed = 1;
3442  }
3443
3444  // Force process exit if it has been killed and is in user space.
3445  // (If it is still executing in the kernel, let it keep running
3446  // until it gets to the regular system call return.)
3447  if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3448    exit();
3449
```



```
3450  // Force process to give up CPU on clock tick.
3451  // If interrupts were on while locks held, would need to check nlock.
3452  if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3453    yield();
3454
3455  // Check if the process has been killed since we yielded
3456  if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3457    exit();
3458 }
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3479
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
```

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



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

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



```
3650 extern int sys_halt(void);
3651 extern int sys_date(void);
3652 #ifdef CS333_P2
3653 extern int sys_getuid(void);
3654 extern int sys_getgid(void);
3655 extern int sys_getppid(void);
3656 extern int sys_setuid(void);
3657 extern int sys_setgid(void);
3658 extern int sys_getprocs(void);
3659 #endif
3660 #ifdef CS333_P3
3661 extern int sys_setpriority(void);
3662 #endif
3663
3664 static int (*syscalls[])(void) = {
3665 [SYS_fork]    sys_fork,
3666 [SYS_exit]    sys_exit,
3667 [SYS_wait]    sys_wait,
3668 [SYS_pipe]    sys_pipe,
3669 [SYS_read]    sys_read,
3670 [SYS_kill]    sys_kill,
3671 [SYS_exec]    sys_exec,
3672 [SYS_fstat]   sys_fstat,
3673 [SYS_chdir]   sys_chdir,
3674 [SYS_dup]     sys_dup,
3675 [SYS_getpid]  sys_getpid,
3676 [SYS_sbrk]    sys_sbrk,
3677 [SYS_sleep]   sys_sleep,
3678 [SYS_uptime]  sys_uptime,
3679 [SYS_open]    sys_open,
3680 [SYS_write]   sys_write,
3681 [SYS_mknod]   sys_mknod,
3682 [SYS_unlink]  sys_unlink,
3683 [SYS_link]    sys_link,
3684 [SYS_mkdir]   sys_mkdir,
3685 [SYS_close]   sys_close,
3686 [SYS_halt]    sys_halt,
3687 [SYS_date]    sys_date,
3688 #ifdef CS333_P2
3689 [SYS_getuid]  sys_getuid,
3690 [SYS_getgid]  sys_getgid,
3691 [SYS_getppid] sys_getppid,
3692 [SYS_setuid]  sys_setuid,
3693 [SYS_setgid]  sys_setgid,
3694 [SYS_getprocs]  sys_getprocs,
3695 #endif
3696 #ifdef CS333_P3
3697 [SYS_setpriority]  sys_setpriority,
3698 #endif
3699 };
```

```
3700 // put data structure for printing out system call invocation information he
3701 #ifdef PRINT_SYSCALLS
3702 static const char * (print_syscalls[]) = {
3703 [SYS_fork] = "fork",
3704 [SYS_exit]   = "exit",
3705 [SYS_wait]     = "wait",
3706 [SYS_pipe]     = "pipe",
3707 [SYS_read]     = "read",
3708 [SYS_kill]     = "kill",
3709 [SYS_exec]     = "exec",
3710 [SYS_fstat]    = "fstat",
3711 [SYS_chdir]    = "chdir",
3712 [SYS_dup]      = "dup",
3713 [SYS_getpid]   = "getpid",
3714 [SYS_sbrk]     = "sbrk",
3715 [SYS_sleep]    = "sleep",
3716 [SYS_uptime]   = "uptime",
3717 [SYS_open]     = "open",
3718 [SYS_write]    = "write",
3719 [SYS_mknod]    = "mknod",
3720 [SYS_unlink]   = "unlink",
3721 [SYS_link]     = "link",
3722 [SYS_mkdir]    = "mkdir",
3723 [SYS_close]    = "close",
3724 [SYS_halt]     = "halt",
3725 [SYS_date]     = "date",
3726 #ifdef CS333_P2
3727 [SYS_getgid]   = "getuid",
3728 [SYS_getuid]   = "getgid",
3729 [SYS_getppid]  = "getppid",
3730 [SYS_setgid]   = "setuid",
3731 [SYS_setuid]   = "setgid",
3732 [SYS_getprocs]   = "getprocs",
3733 #endif
3734 #ifdef CS333_P3
3735 [SYS_setpriority]  = "setpriority",
3736 #endif
3737 };
3738
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3749
```



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

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



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

```
3900 //Turn of the computer
3901 int sys_halt(void){
3902   cprintf("Shutting down ...\n");
3903   //outw (0xB004, 0x0 | 0x2000);
3904    outw( 0x604, 0x0 | 0x2000 );
3905    return 0;
3906
3907 }
3908
3909 //Get current UTC date of the system
3910 int
3911 sys_date(void)
3912 {
3913   struct rtcdate *d;
3914   if(argptr(0, (void*)&d, sizeof(*d)) < 0)
3915     return -1;
3916   cmostime(d);
3917   return 0;
3918 }
3919
3920 #ifdef CS333_P2
3921 // Set UID
3922 int
3923 sys_setuid(void)
3924 {
3925    uint new_uid;
3926   if(argint(0,(int*) &new_uid) < 0)
3927            return -1;
3928    if(new_uid < 0 || new_uid > 32767)
3929            return -1;
3930    proc->uid = new_uid;
3931    return 0;
3932 }
3933
3934 // Set GID
3935 int
3936 sys_setgid(void)
3937 {
3938    uint new_gid;
3939   if(argint(0,(int*) &new_gid) < 0)
3940            return -1;
3941    if(new_gid < 0 || new_gid > 32767)
3942            return -1;
3943    proc->gid = new_gid;
3944    return 0;
3945 }
3946
3947
3948
3949
```



```
3950 // Get UID of current process
3951 int
3952 sys_getuid(void)
3953 {
3954    return proc->uid;
3955 }
3956
3957 // Get GID of current process
3958 int
3959 sys_getgid(void)
3960 {
3961    return proc->gid;
3962 }
3963
3964 // Get PPID of current process
3965 int
3966 sys_getppid(void)
3967 {
3968    if(proc->pid == 1)
3969            return proc->pid;
3970    if(!proc->parent)
3971            return proc->pid;
3972    return proc->parent->pid;
3973 }
3974
3975 // Get process info
3976 int
3977 sys_getprocs(void)
3978 {
3979    uint arg1;
3980    struct uproc* table;
3981    if(argint(0,(int*) &arg1) < 0)
3982            return -1;
3983    if(argptr(1,(void*)&table, sizeof(*table)) < 0)
3984            return -1;;
3985    return getprocs(arg1, table);
3986 }
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
```

```
#endif

#ifdef CS333_P3
int
sys_setpriority(void)
{
   int value;
   int pid;
   if(argint(0,(int*) &pid) < 0)
            return -1;
   if(argint(1,(int*) &value) < 0)
            return -1;
  return setpriority(pid, value);
}
#endif
```



```
// halt the system.
#include "types.h"
#include "user.h"

int
main(void) {
  halt();
  return 0;
}
```

```
4100 struct buf {
4101   int flags;
4102   uint dev;
4103   uint blockno;
4104   struct buf *prev; // LRU cache list
4105   struct buf *next;
4106   struct buf *qnext; // disk queue
4107   uchar data[BSIZE];
4108 };
4109 #define B_BUSY  0x1  // buffer is locked by some process
4110 #define B_VALID 0x2  // buffer has been read from disk
4111 #define B_DIRTY 0x4  // buffer needs to be written to disk
4112
4113
4114
4115
4116
4117
4118
4119
4120
4121
4122
4123
4124
4125
4126
4127
4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```



```
4150 #define O_RDONLY  0x000
4151 #define O_WRONLY  0x001
4152 #define O_RDWR    0x002
4153 #define O_CREATE  0x200
4154
4155
4156
4157
4158
4159
4160
4161
4162
4163
4164
4165
4166
4167
4168
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
```

```
4200 #define T_DIR  1   // Directory
4201 #define T_FILE 2   // File
4202 #define T_DEV  3   // Device
4203
4204 struct stat {
4205   short type;  // Type of file
4206   int dev;     // File system's disk device
4207   uint ino;    // Inode number
4208   short nlink; // Number of links to file
4209   uint size;   // Size of file in bytes
4210 };
4211
4212
4213
4214
4215
4216
4217
4218
4219
4220
4221
4222
4223
4224
4225
4226
4227
4228
4229
4230
4231
4232
4233
4234
4235
4236
4237
4238
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
```



```
4250 // On-disk file system format.
4251 // Both the kernel and user programs use this header file.
4252
4253
4254 #define ROOTINO 1  // root i-number
4255 #define BSIZE 512  // block size
4256
4257 // Disk layout:
4258 // [ boot block | super block | log | inode blocks | free bit map | data blo
4259 //
4260 // mkfs computes the super block and builds an initial file system. The supe
4261 // the disk layout:
4262 struct superblock {
4263   uint size;         // Size of file system image (blocks)
4264   uint nblocks;      // Number of data blocks
4265   uint ninodes;      // Number of inodes.
4266   uint nlog;         // Number of log blocks
4267   uint logstart;     // Block number of first log block
4268   uint inodestart;   // Block number of first inode block
4269   uint bmapstart;    // Block number of first free map block
4270 };
4271
4272 #define NDIRECT 12
4273 #define NINDIRECT (BSIZE / sizeof(uint))
4274 #define MAXFILE (NDIRECT + NINDIRECT)
4275
4276 // On-disk inode structure
4277 struct dinode {
4278   short type;           // File type
4279   short major;          // Major device number (T_DEV only)
4280   short minor;          // Minor device number (T_DEV only)
4281   short nlink;          // Number of links to inode in file system
4282   uint size;            // Size of file (bytes)
4283   uint addrs[NDIRECT+1];   // Data block addresses
4284 };
4285
4286
4287
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
```

```
4300 // Inodes per block.
4301 #define IPB           (BSIZE / sizeof(struct dinode))
4302
4303 // Block containing inode i
4304 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
4305
4306 // Bitmap bits per block
4307 #define BPB           (BSIZE*8)
4308
4309 // Block of free map containing bit for block b
4310 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
4311
4312 // Directory is a file containing a sequence of dirent structures.
4313 #define DIRSIZ 14
4314
4315 struct dirent {
4316   ushort inum;
4317   char name[DIRSIZ];
4318 };
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4330
4331
4332
4333
4334
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
```



```
4350 struct file {
4351   enum { FD_NONE, FD_PIPE, FD_INODE } type;
4352   int ref; // reference count
4353   char readable;
4354   char writable;
4355   struct pipe *pipe;
4356   struct inode *ip;
4357   uint off;
4358 };
4359
4360
4361 // in-memory copy of an inode
4362 struct inode {
4363   uint dev;           // Device number
4364   uint inum;          // Inode number
4365   int ref;            // Reference count
4366   int flags;          // I_BUSY, I_VALID
4367
4368   short type;         // copy of disk inode
4369   short major;
4370   short minor;
4371   short nlink;
4372   uint size;
4373   uint addrs[NDIRECT+1];
4374 };
4375 #define I_BUSY 0x1
4376 #define I_VALID 0x2
4377
4378 // table mapping major device number to
4379 // device functions
4380 struct devsw {
4381   int (*read)(struct inode*, char*, int);
4382   int (*write)(struct inode*, char*, int);
4383 };
4384
4385 extern struct devsw devsw[];
4386
4387 #define CONSOLE 1
4388
4389 // Blank page.
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
```

```
4400 // Simple PIO-based (non-DMA) IDE driver code.
4401
4402 #include "types.h"
4403 #include "defs.h"
4404 #include "param.h"
4405 #include "memlayout.h"
4406 #include "mmu.h"
4407 #include "proc.h"
4408 #include "x86.h"
4409 #include "traps.h"
4410 #include "spinlock.h"
4411 #include "fs.h"
4412 #include "buf.h"
4413
4414 #define SECTOR_SIZE   512
4415 #define IDE_BSY       0x80
4416 #define IDE_DRDY      0x40
4417 #define IDE_DF        0x20
4418 #define IDE_ERR       0x01
4419
4420 #define IDE_CMD_READ  0x20
4421 #define IDE_CMD_WRITE 0x30
4422
4423 // idequeue points to the buf now being read/written to the disk.
4424 // idequeue->qnext points to the next buf to be processed.
4425 // You must hold idelock while manipulating queue.
4426
4427 static struct spinlock idelock;
4428 static struct buf *idequeue;
4429
4430 static int havedisk1;
4431 static void idestart(struct buf*);
4432
4433 // Wait for IDE disk to become ready.
4434 static int
4435 idewait(int checkerr)
4436 {
4437   int r;
4438
4439   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4440     ;
4441   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4442     return -1;
4443   return 0;
4444 }
4445
4446
4447
4448
4449
```



```
4450 void
4451 ideinit(void)
4452 {
4453   int i;
4454
4455   initlock(&idelock, "ide");
4456   picenable(IRQ_IDE);
4457   ioapicenable(IRQ_IDE, ncpu - 1);
4458   idewait(0);
4459
4460   // Check if disk 1 is present
4461   outb(0x1f6, 0xe0 | (1<<4));
4462   for(i=0; i<1000; i++){
4463     if(inb(0x1f7) != 0){
4464       havedisk1 = 1;
4465       break;
4466     }
4467   }
4468
4469   // Switch back to disk 0.
4470   outb(0x1f6, 0xe0 | (0<<4));
4471 }
4472
4473 // Start the request for b.  Caller must hold idelock.
4474 static void
4475 idestart(struct buf *b)
4476 {
4477   if(b == 0)
4478     panic("idestart");
4479   if(b->blockno >= FSSIZE)
4480     panic("incorrect blockno");
4481   int sector_per_block =  BSIZE/SECTOR_SIZE;
4482   int sector = b->blockno * sector_per_block;
4483
4484   if (sector_per_block > 7) panic("idestart");
4485
4486   idewait(0);
4487   outb(0x3f6, 0);  // generate interrupt
4488   outb(0x1f2, sector_per_block);  // number of sectors
4489   outb(0x1f3, sector & 0xff);
4490   outb(0x1f4, (sector >> 8) & 0xff);
4491   outb(0x1f5, (sector >> 16) & 0xff);
4492   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4493   if(b->flags & B_DIRTY){
4494     outb(0x1f7, IDE_CMD_WRITE);
4495     outsl(0x1f0, b->data, BSIZE/4);
4496   } else {
4497     outb(0x1f7, IDE_CMD_READ);
4498   }
4499 }
```

```
4500 // Interrupt handler.
4501 void
4502 ideintr(void)
4503 {
4504   struct buf *b;
4505
4506   // First queued buffer is the active request.
4507   acquire(&idelock);
4508   if((b = idequeue) == 0){
4509     release(&idelock);
4510     // cprintf("spurious IDE interrupt\n");
4511     return;
4512   }
4513   idequeue = b->qnext;
4514
4515   // Read data if needed.
4516   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4517     insl(0x1f0, b->data, BSIZE/4);
4518
4519   // Wake process waiting for this buf.
4520   b->flags |= B_VALID;
4521   b->flags &= ~B_DIRTY;
4522   wakeup(b);
4523
4524   // Start disk on next buf in queue.
4525   if(idequeue != 0)
4526     idestart(idequeue);
4527
4528   release(&idelock);
4529 }
4530
4531 // Sync buf with disk.
4532 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
4533 // Else if B_VALID is not set, read buf from disk, set B_VALID.
4534 void
4535 iderw(struct buf *b)
4536 {
4537   struct buf **pp;
4538
4539   if(!(b->flags & B_BUSY))
4540     panic("iderw: buf not busy");
4541   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
4542     panic("iderw: nothing to do");
4543   if(b->dev != 0 && !havedisk1)
4544     panic("iderw: ide disk 1 not present");
4545
4546   acquire(&idelock);
4547
4548
4549
```



```
4550   // Append b to idequeue.
4551   b->qnext = 0;
4552   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
4553     ;
4554   *pp = b;
4555
4556   // Start disk if necessary.
4557   if(idequeue == b)
4558     idestart(b);
4559
4560   // Wait for request to finish.
4561   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
4562     sleep(b, &idelock);
4563   }
4564
4565   release(&idelock);
4566 }
4567
4568
4569
4570
4571
4572
4573
4574
4575
4576
4577
4578
4579
4580
4581
4582
4583
4584
4585
4586
4587
4588
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4599
```

```
4600 // Buffer cache.
4601 //
4602 // The buffer cache is a linked list of buf structures holding
4603 // cached copies of disk block contents.  Caching disk blocks
4604 // in memory reduces the number of disk reads and also provides
4605 // a synchronization point for disk blocks used by multiple processes.
4606 //
4607 // Interface:
4608 // * To get a buffer for a particular disk block, call bread.
4609 // * After changing buffer data, call bwrite to write it to disk.
4610 // * When done with the buffer, call brelse.
4611 // * Do not use the buffer after calling brelse.
4612 // * Only one process at a time can use a buffer,
4613 //     so do not keep them longer than necessary.
4614 //
4615 // The implementation uses three state flags internally:
4616 // * B_BUSY: the block has been returned from bread
4617 //     and has not been passed back to brelse.
4618 // * B_VALID: the buffer data has been read from the disk.
4619 // * B_DIRTY: the buffer data has been modified
4620 //     and needs to be written to disk.
4621
4622 #include "types.h"
4623 #include "defs.h"
4624 #include "param.h"
4625 #include "spinlock.h"
4626 #include "fs.h"
4627 #include "buf.h"
4628
4629 struct {
4630   struct spinlock lock;
4631   struct buf buf[NBUF];
4632
4633   // Linked list of all buffers, through prev/next.
4634   // head.next is most recently used.
4635   struct buf head;
4636 } bcache;
4637
4638 void
4639 binit(void)
4640 {
4641   struct buf *b;
4642
4643   initlock(&bcache.lock, "bcache");
4644
4645   // Create linked list of buffers
4646   bcache.head.prev = &bcache.head;
4647   bcache.head.next = &bcache.head;
4648   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4649     b->next = bcache.head.next;
```



```
4650     b->prev = &bcache.head;
4651     b->dev = -1;
4652     bcache.head.next->prev = b;
4653     bcache.head.next = b;
4654   }
4655 }
4656
4657 // Look through buffer cache for block on device dev.
4658 // If not found, allocate a buffer.
4659 // In either case, return B_BUSY buffer.
4660 static struct buf*
4661 bget(uint dev, uint blockno)
4662 {
4663   struct buf *b;
4664
4665   acquire(&bcache.lock);
4666
4667  loop:
4668   // Is the block already cached?
4669   for(b = bcache.head.next; b != &bcache.head; b = b->next){
4670     if(b->dev == dev && b->blockno == blockno){
4671       if(!(b->flags & B_BUSY)){
4672         b->flags |= B_BUSY;
4673         release(&bcache.lock);
4674         return b;
4675       }
4676       sleep(b, &bcache.lock);
4677       goto loop;
4678     }
4679   }
4680
4681   // Not cached; recycle some non-busy and clean buffer.
4682   // "clean" because B_DIRTY and !B_BUSY means log.c
4683   // hasn't yet committed the changes to the buffer.
4684   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4685     if((b->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
4686       b->dev = dev;
4687       b->blockno = blockno;
4688       b->flags = B_BUSY;
4689       release(&bcache.lock);
4690       return b;
4691     }
4692   }
4693   panic("bget: no buffers");
4694 }
4695
4696
4697
4698
4699
```

```
4700 // Return a B_BUSY buf with the contents of the indicated block.
4701 struct buf*
4702 bread(uint dev, uint blockno)
4703 {
4704   struct buf *b;
4705
4706   b = bget(dev, blockno);
4707   if(!(b->flags & B_VALID)) {
4708     iderw(b);
4709   }
4710   return b;
4711 }
4712
4713 // Write b's contents to disk.  Must be B_BUSY.
4714 void
4715 bwrite(struct buf *b)
4716 {
4717   if((b->flags & B_BUSY) == 0)
4718     panic("bwrite");
4719   b->flags |= B_DIRTY;
4720   iderw(b);
4721 }
4722
4723 // Release a B_BUSY buffer.
4724 // Move to the head of the MRU list.
4725 void
4726 brelse(struct buf *b)
4727 {
4728   if((b->flags & B_BUSY) == 0)
4729     panic("brelse");
4730
4731   acquire(&bcache.lock);
4732
4733   b->next->prev = b->prev;
4734   b->prev->next = b->next;
4735   b->next = bcache.head.next;
4736   b->prev = &bcache.head;
4737   bcache.head.next->prev = b;
4738   bcache.head.next = b;
4739
4740   b->flags &= ~B_BUSY;
4741   wakeup(b);
4742
4743   release(&bcache.lock);
4744 }
4745 // Blank page.
4746
4747
4748
4749
```



```
4750 #include "types.h"
4751 #include "defs.h"
4752 #include "param.h"
4753 #include "spinlock.h"
4754 #include "fs.h"
4755 #include "buf.h"
4756
4757 // Simple logging that allows concurrent FS system calls.
4758 //
4759 // A log transaction contains the updates of multiple FS system
4760 // calls. The logging system only commits when there are
4761 // no FS system calls active. Thus there is never
4762 // any reasoning required about whether a commit might
4763 // write an uncommitted system call's updates to disk.
4764 //
4765 // A system call should call begin_op()/end_op() to mark
4766 // its start and end. Usually begin_op() just increments
4767 // the count of in-progress FS system calls and returns.
4768 // But if it thinks the log is close to running out, it
4769 // sleeps until the last outstanding end_op() commits.
4770 //
4771 // The log is a physical re-do log containing disk blocks.
4772 // The on-disk log format:
4773 //   header block, containing block #s for block A, B, C, ...
4774 //   block A
4775 //   block B
4776 //   block C
4777 //   ...
4778 // Log appends are synchronous.
4779
4780 // Contents of the header block, used for both the on-disk header block
4781 // and to keep track in memory of logged block# before commit.
4782 struct logheader {
4783   int n;
4784   int block[LOGSIZE];
4785 };
4786
4787 struct log {
4788   struct spinlock lock;
4789   int start;
4790   int size;
4791   int outstanding; // how many FS sys calls are executing.
4792   int committing;  // in commit(), please wait.
4793   int dev;
4794   struct logheader lh;
4795 };
4796
4797
4798
4799
```

```
4800 struct log log;
4801
4802 static void recover_from_log(void);
4803 static void commit();
4804
4805 void
4806 initlog(int dev)
4807 {
4808   if (sizeof(struct logheader) >= BSIZE)
4809     panic("initlog: too big logheader");
4810
4811   struct superblock sb;
4812   initlock(&log.lock, "log");
4813   readsb(dev, &sb);
4814   log.start = sb.logstart;
4815   log.size = sb.nlog;
4816   log.dev = dev;
4817   recover_from_log();
4818 }
4819
4820 // Copy committed blocks from log to their home location
4821 static void
4822 install_trans(void)
4823 {
4824   int tail;
4825
4826   for (tail = 0; tail < log.lh.n; tail++) {
4827     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
4828     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
4829     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
4830     bwrite(dbuf);  // write dst to disk
4831     brelse(lbuf);
4832     brelse(dbuf);
4833   }
4834 }
4835
4836 // Read the log header from disk into the in-memory log header
4837 static void
4838 read_head(void)
4839 {
4840   struct buf *buf = bread(log.dev, log.start);
4841   struct logheader *lh = (struct logheader *) (buf->data);
4842   int i;
4843   log.lh.n = lh->n;
4844   for (i = 0; i < log.lh.n; i++) {
4845     log.lh.block[i] = lh->block[i];
4846   }
4847   brelse(buf);
4848 }
4849
```



```
4850 // Write in-memory log header to disk.
4851 // This is the true point at which the
4852 // current transaction commits.
4853 static void
4854 write_head(void)
4855 {
4856   struct buf *buf = bread(log.dev, log.start);
4857   struct logheader *hb = (struct logheader *) (buf->data);
4858   int i;
4859   hb->n = log.lh.n;
4860   for (i = 0; i < log.lh.n; i++) {
4861     hb->block[i] = log.lh.block[i];
4862   }
4863   bwrite(buf);
4864   brelse(buf);
4865 }
4866
4867 static void
4868 recover_from_log(void)
4869 {
4870   read_head();
4871   install_trans(); // if committed, copy from log to disk
4872   log.lh.n = 0;
4873   write_head(); // clear the log
4874 }
4875
4876 // called at the start of each FS system call.
4877 void
4878 begin_op(void)
4879 {
4880   acquire(&log.lock);
4881   while(1){
4882     if(log.committing){
4883       sleep(&log, &log.lock);
4884     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
4885       // this op might exhaust log space; wait for commit.
4886       sleep(&log, &log.lock);
4887     } else {
4888       log.outstanding += 1;
4889       release(&log.lock);
4890       break;
4891     }
4892   }
4893 }
4894
4895
4896
4897
4898
4899
```

```
4900 // called at the end of each FS system call.
4901 // commits if this was the last outstanding operation.
4902 void
4903 end_op(void)
4904 {
4905   int do_commit = 0;
4906
4907   acquire(&log.lock);
4908   log.outstanding -= 1;
4909   if(log.committing)
4910     panic("log.committing");
4911   if(log.outstanding == 0){
4912     do_commit = 1;
4913     log.committing = 1;
4914   } else {
4915     // begin_op() may be waiting for log space.
4916     wakeup(&log);
4917   }
4918   release(&log.lock);
4919
4920   if(do_commit){
4921     // call commit w/o holding locks, since not allowed
4922     // to sleep with locks.
4923     commit();
4924     acquire(&log.lock);
4925     log.committing = 0;
4926     wakeup(&log);
4927     release(&log.lock);
4928   }
4929 }
4930
4931 // Copy modified blocks from cache to log.
4932 static void
4933 write_log(void)
4934 {
4935   int tail;
4936
4937   for (tail = 0; tail < log.lh.n; tail++) {
4938     struct buf *to = bread(log.dev, log.start+tail+1); // log block
4939     struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
4940     memmove(to->data, from->data, BSIZE);
4941     bwrite(to);  // write the log
4942     brelse(from);
4943     brelse(to);
4944   }
4945 }
4946
4947
4948
4949
```



```
4950 static void
4951 commit()
4952 {
4953   if (log.lh.n > 0) {
4954     write_log();     // Write modified blocks from cache to log
4955     write_head();    // Write header to disk -- the real commit
4956     install_trans(); // Now install writes to home locations
4957     log.lh.n = 0;
4958     write_head();    // Erase the transaction from the log
4959   }
4960 }
4961
4962 // Caller has modified b->data and is done with the buffer.
4963 // Record the block number and pin in the cache with B_DIRTY.
4964 // commit()/write_log() will do the disk write.
4965 //
4966 // log_write() replaces bwrite(); a typical use is:
4967 //   bp = bread(...)
4968 //   modify bp->data[]
4969 //   log_write(bp)
4970 //   brelse(bp)
4971 void
4972 log_write(struct buf *b)
4973 {
4974   int i;
4975
4976   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
4977     panic("too big a transaction");
4978   if (log.outstanding < 1)
4979     panic("log_write outside of trans");
4980
4981   acquire(&log.lock);
4982   for (i = 0; i < log.lh.n; i++) {
4983     if (log.lh.block[i] == b->blockno)   // log absorbtion
4984       break;
4985   }
4986   log.lh.block[i] = b->blockno;
4987   if (i == log.lh.n)
4988     log.lh.n++;
4989   b->flags |= B_DIRTY; // prevent eviction
4990   release(&log.lock);
4991 }
4992
4993
4994
4995
4996
4997
4998
4999
```

```
5000 // File system implementation.  Five layers:
5001 //   + Blocks: allocator for raw disk blocks.
5002 //   + Log: crash recovery for multi-step updates.
5003 //   + Files: inode allocator, reading, writing, metadata.
5004 //   + Directories: inode with special contents (list of other inodes!)
5005 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
5006 //
5007 // This file contains the low-level file system manipulation
5008 // routines.  The (higher-level) system call implementations
5009 // are in sysfile.c.
5010
5011 #include "types.h"
5012 #include "defs.h"
5013 #include "param.h"
5014 #include "stat.h"
5015 #include "mmu.h"
5016 #include "proc.h"
5017 #include "spinlock.h"
5018 #include "fs.h"
5019 #include "buf.h"
5020 #include "file.h"
5021
5022 #define min(a, b) ((a) < (b) ? (a) : (b))
5023 static void itrunc(struct inode*);
5024 struct superblock sb;   // there should be one per dev, but we run with one 
5025
5026 // Read the super block.
5027 void
5028 readsb(int dev, struct superblock *sb)
5029 {
5030   struct buf *bp;
5031
5032   bp = bread(dev, 1);
5033   memmove(sb, bp->data, sizeof(*sb));
5034   brelse(bp);
5035 }
5036
5037 // Zero a block.
5038 static void
5039 bzero(int dev, int bno)
5040 {
5041   struct buf *bp;
5042
5043   bp = bread(dev, bno);
5044   memset(bp->data, 0, BSIZE);
5045   log_write(bp);
5046   brelse(bp);
5047 }
5048
5049
```



```
5050 // Blocks.
5051
5052 // Allocate a zeroed disk block.
5053 static uint
5054 balloc(uint dev)
5055 {
5056   int b, bi, m;
5057   struct buf *bp;
5058
5059   bp = 0;
5060   for(b = 0; b < sb.size; b += BPB){
5061     bp = bread(dev, BBLOCK(b, sb));
5062     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
5063       m = 1 << (bi % 8);
5064       if((bp->data[bi/8] & m) == 0){  // Is block free?
5065         bp->data[bi/8] |= m;  // Mark block in use.
5066         log_write(bp);
5067         brelse(bp);
5068         bzero(dev, b + bi);
5069         return b + bi;
5070       }
5071     }
5072     brelse(bp);
5073   }
5074   panic("balloc: out of blocks");
5075 }
5076
5077 // Free a disk block.
5078 static void
5079 bfree(int dev, uint b)
5080 {
5081   struct buf *bp;
5082   int bi, m;
5083
5084   readsb(dev, &sb);
5085   bp = bread(dev, BBLOCK(b, sb));
5086   bi = b % BPB;
5087   m = 1 << (bi % 8);
5088   if((bp->data[bi/8] & m) == 0)
5089     panic("freeing free block");
5090   bp->data[bi/8] &= ~m;
5091   log_write(bp);
5092   brelse(bp);
5093 }
5094
5095
5096
5097
5098
5099
```

```
5100 // Inodes.
5101 //
5102 // An inode describes a single unnamed file.
5103 // The inode disk structure holds metadata: the file's type,
5104 // its size, the number of links referring to it, and the
5105 // list of blocks holding the file's content.
5106 //
5107 // The inodes are laid out sequentially on disk at
5108 // sb.startinode. Each inode has a number, indicating its
5109 // position on the disk.
5110 //
5111 // The kernel keeps a cache of in-use inodes in memory
5112 // to provide a place for synchronizing access
5113 // to inodes used by multiple processes. The cached
5114 // inodes include book-keeping information that is
5115 // not stored on disk: ip->ref and ip->flags.
5116 //
5117 // An inode and its in-memory represtative go through a
5118 // sequence of states before they can be used by the
5119 // rest of the file system code.
5120 //
5121 // * Allocation: an inode is allocated if its type (on disk)
5122 //   is non-zero. ialloc() allocates, iput() frees if
5123 //   the link count has fallen to zero.
5124 //
5125 // * Referencing in cache: an entry in the inode cache
5126 //   is free if ip->ref is zero. Otherwise ip->ref tracks
5127 //   the number of in-memory pointers to the entry (open
5128 //   files and current directories). iget() to find or
5129 //   create a cache entry and increment its ref, iput()
5130 //   to decrement ref.
5131 //
5132 // * Valid: the information (type, size, &c) in an inode
5133 //   cache entry is only correct when the I_VALID bit
5134 //   is set in ip->flags. ilock() reads the inode from
5135 //   the disk and sets I_VALID, while iput() clears
5136 //   I_VALID if ip->ref has fallen to zero.
5137 //
5138 // * Locked: file system code may only examine and modify
5139 //   the information in an inode and its content if it
5140 //   has first locked the inode. The I_BUSY flag indicates
5141 //   that the inode is locked. ilock() sets I_BUSY,
5142 //   while iunlock clears it.
5143 //
5144 // Thus a typical sequence is:
5145 //   ip = iget(dev, inum)
5146 //   ilock(ip)
5147 //   ... examine and modify ip->xxx ...
5148 //   iunlock(ip)
5149 //   iput(ip)
```



```
5150 //
5151 // ilock() is separate from iget() so that system calls can
5152 // get a long-term reference to an inode (as for an open file)
5153 // and only lock it for short periods (e.g., in read()).
5154 // The separation also helps avoid deadlock and races during
5155 // pathname lookup. iget() increments ip->ref so that the inode
5156 // stays cached and pointers to it remain valid.
5157 //
5158 // Many internal file system functions expect the caller to
5159 // have locked the inodes involved; this lets callers create
5160 // multi-step atomic operations.
5161
5162 struct {
5163   struct spinlock lock;
5164   struct inode inode[NINODE];
5165 } icache;
5166
5167 void
5168 iinit(int dev)
5169 {
5170   initlock(&icache.lock, "icache");
5171   readsb(dev, &sb);
5172   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d inodestart %
5173           sb.nblocks, sb.ninodes, sb.nlog, sb.logstart, sb.inodestart, sb.bma
5174 }
5175
5176 static struct inode* iget(uint dev, uint inum);
5177
5178 // Allocate a new inode with the given type on device dev.
5179 // A free inode has a type of zero.
5180 struct inode*
5181 ialloc(uint dev, short type)
5182 {
5183   int inum;
5184   struct buf *bp;
5185   struct dinode *dip;
5186
5187   for(inum = 1; inum < sb.ninodes; inum++){
5188     bp = bread(dev, IBLOCK(inum, sb));
5189     dip = (struct dinode*)bp->data + inum%IPB;
5190     if(dip->type == 0){  // a free inode
5191       memset(dip, 0, sizeof(*dip));
5192       dip->type = type;
5193       log_write(bp);   // mark it allocated on the disk
5194       brelse(bp);
5195       return iget(dev, inum);
5196     }
5197     brelse(bp);
5198   }
5199   panic("ialloc: no inodes");
```

```
5200 }
5201
5202 // Copy a modified in-memory inode to disk.
5203 void
5204 iupdate(struct inode *ip)
5205 {
5206   struct buf *bp;
5207   struct dinode *dip;
5208
5209   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5210   dip = (struct dinode*)bp->data + ip->inum%IPB;
5211   dip->type = ip->type;
5212   dip->major = ip->major;
5213   dip->minor = ip->minor;
5214   dip->nlink = ip->nlink;
5215   dip->size = ip->size;
5216   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5217   log_write(bp);
5218   brelse(bp);
5219 }
5220
5221 // Find the inode with number inum on device dev
5222 // and return the in-memory copy. Does not lock
5223 // the inode and does not read it from disk.
5224 static struct inode*
5225 iget(uint dev, uint inum)
5226 {
5227   struct inode *ip, *empty;
5228
5229   acquire(&icache.lock);
5230
5231   // Is the inode already cached?
5232   empty = 0;
5233   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5234     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5235       ip->ref++;
5236       release(&icache.lock);
5237       return ip;
5238     }
5239     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
5240       empty = ip;
5241   }
5242
5243   // Recycle an inode cache entry.
5244   if(empty == 0)
5245     panic("iget: no inodes");
5246
5247
5248
5249
```



```
5250   ip = empty;
5251   ip->dev = dev;
5252   ip->inum = inum;
5253   ip->ref = 1;
5254   ip->flags = 0;
5255   release(&icache.lock);
5256
5257   return ip;
5258 }
5259
5260 // Increment reference count for ip.
5261 // Returns ip to enable ip = idup(ip1) idiom.
5262 struct inode*
5263 idup(struct inode *ip)
5264 {
5265   acquire(&icache.lock);
5266   ip->ref++;
5267   release(&icache.lock);
5268   return ip;
5269 }
5270
5271 // Lock the given inode.
5272 // Reads the inode from disk if necessary.
5273 void
5274 ilock(struct inode *ip)
5275 {
5276   struct buf *bp;
5277   struct dinode *dip;
5278
5279   if(ip == 0 || ip->ref < 1)
5280     panic("ilock");
5281
5282   acquire(&icache.lock);
5283   while(ip->flags & I_BUSY)
5284     sleep(ip, &icache.lock);
5285   ip->flags |= I_BUSY;
5286   release(&icache.lock);
5287
5288   if(!(ip->flags & I_VALID)){
5289     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5290     dip = (struct dinode*)bp->data + ip->inum%IPB;
5291     ip->type = dip->type;
5292     ip->major = dip->major;
5293     ip->minor = dip->minor;
5294     ip->nlink = dip->nlink;
5295     ip->size = dip->size;
5296     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5297     brelse(bp);
5298     ip->flags |= I_VALID;
5299     if(ip->type == 0)
```

```
5300       panic("ilock: no type");
5301   }
5302 }
5303
5304 // Unlock the given inode.
5305 void
5306 iunlock(struct inode *ip)
5307 {
5308   if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
5309     panic("iunlock");
5310
5311   acquire(&icache.lock);
5312   ip->flags &= ~I_BUSY;
5313   wakeup(ip);
5314   release(&icache.lock);
5315 }
5316
5317 // Drop a reference to an in-memory inode.
5318 // If that was the last reference, the inode cache entry can
5319 // be recycled.
5320 // If that was the last reference and the inode has no links
5321 // to it, free the inode (and its content) on disk.
5322 // All calls to iput() must be inside a transaction in
5323 // case it has to free the inode.
5324 void
5325 iput(struct inode *ip)
5326 {
5327   acquire(&icache.lock);
5328   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
5329     // inode has no links and no other references: truncate and free.
5330     if(ip->flags & I_BUSY)
5331       panic("iput busy");
5332     ip->flags |= I_BUSY;
5333     release(&icache.lock);
5334     itrunc(ip);
5335     ip->type = 0;
5336     iupdate(ip);
5337     acquire(&icache.lock);
5338     ip->flags = 0;
5339     wakeup(ip);
5340   }
5341   ip->ref--;
5342   release(&icache.lock);
5343 }
5344
5345
5346
5347
5348
5349
```



```
5350 // Common idiom: unlock, then put.
5351 void
5352 iunlockput(struct inode *ip)
5353 {
5354   iunlock(ip);
5355   iput(ip);
5356 }
5357
5358 // Inode content
5359 //
5360 // The content (data) associated with each inode is stored
5361 // in blocks on the disk. The first NDIRECT block numbers
5362 // are listed in ip->addrs[].  The next NINDIRECT blocks are
5363 // listed in block ip->addrs[NDIRECT].
5364
5365 // Return the disk block address of the nth block in inode ip.
5366 // If there is no such block, bmap allocates one.
5367 static uint
5368 bmap(struct inode *ip, uint bn)
5369 {
5370   uint addr, *a;
5371   struct buf *bp;
5372
5373   if(bn < NDIRECT){
5374     if((addr = ip->addrs[bn]) == 0)
5375       ip->addrs[bn] = addr = balloc(ip->dev);
5376     return addr;
5377   }
5378   bn -= NDIRECT;
5379
5380   if(bn < NINDIRECT){
5381     // Load indirect block, allocating if necessary.
5382     if((addr = ip->addrs[NDIRECT]) == 0)
5383       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5384     bp = bread(ip->dev, addr);
5385     a = (uint*)bp->data;
5386     if((addr = a[bn]) == 0){
5387       a[bn] = addr = balloc(ip->dev);
5388       log_write(bp);
5389     }
5390     brelse(bp);
5391     return addr;
5392   }
5393
5394   panic("bmap: out of range");
5395 }
5396
5397
5398
5399
```

```
5400 // Truncate inode (discard contents).
5401 // Only called when the inode has no links
5402 // to it (no directory entries referring to it)
5403 // and has no in-memory reference to it (is
5404 // not an open file or current directory).
5405 static void
5406 itrunc(struct inode *ip)
5407 {
5408   int i, j;
5409   struct buf *bp;
5410   uint *a;
5411
5412   for(i = 0; i < NDIRECT; i++){
5413     if(ip->addrs[i]){
5414       bfree(ip->dev, ip->addrs[i]);
5415       ip->addrs[i] = 0;
5416     }
5417   }
5418
5419   if(ip->addrs[NDIRECT]){
5420     bp = bread(ip->dev, ip->addrs[NDIRECT]);
5421     a = (uint*)bp->data;
5422     for(j = 0; j < NINDIRECT; j++){
5423       if(a[j])
5424         bfree(ip->dev, a[j]);
5425     }
5426     brelse(bp);
5427     bfree(ip->dev, ip->addrs[NDIRECT]);
5428     ip->addrs[NDIRECT] = 0;
5429   }
5430
5431   ip->size = 0;
5432   iupdate(ip);
5433 }
5434
5435 // Copy stat information from inode.
5436 void
5437 stati(struct inode *ip, struct stat *st)
5438 {
5439   st->dev = ip->dev;
5440   st->ino = ip->inum;
5441   st->type = ip->type;
5442   st->nlink = ip->nlink;
5443   st->size = ip->size;
5444 }
5445
5446
5447
5448
5449
```



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

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



```
5550 // Write a new directory entry (name, inum) into the directory dp.
5551 int
5552 dirlink(struct inode *dp, char *name, uint inum)
5553 {
5554   int off;
5555   struct dirent de;
5556   struct inode *ip;
5557
5558   // Check that name is not present.
5559   if((ip = dirlookup(dp, name, 0)) != 0){
5560     iput(ip);
5561     return -1;
5562   }
5563
5564   // Look for an empty dirent.
5565   for(off = 0; off < dp->size; off += sizeof(de)){
5566     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5567       panic("dirlink read");
5568     if(de.inum == 0)
5569       break;
5570   }
5571
5572   strncpy(de.name, name, DIRSIZ);
5573   de.inum = inum;
5574   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5575     panic("dirlink");
5576
5577   return 0;
5578 }
5579
5580 // Paths
5581
5582 // Copy the next path element from path into name.
5583 // Return a pointer to the element following the copied one.
5584 // The returned path has no leading slashes,
5585 // so the caller can check *path=='\0' to see if the name is the last one.
5586 // If no name to remove, return 0.
5587 //
5588 // Examples:
5589 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5590 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5591 //   skipelem("a", name) = "", setting name = "a"
5592 //   skipelem("", name) = skipelem("////", name) = 0
5593 //
5594 static char*
5595 skipelem(char *path, char *name)
5596 {
5597   char *s;
5598   int len;
5599
```

```
5600   while(*path == '/')
5601     path++;
5602   if(*path == 0)
5603     return 0;
5604   s = path;
5605   while(*path != '/' && *path != 0)
5606     path++;
5607   len = path - s;
5608   if(len >= DIRSIZ)
5609     memmove(name, s, DIRSIZ);
5610   else {
5611     memmove(name, s, len);
5612     name[len] = 0;
5613   }
5614   while(*path == '/')
5615     path++;
5616   return path;
5617 }
5618
5619 // Look up and return the inode for a path name.
5620 // If parent != 0, return the inode for the parent and copy the final
5621 // path element into name, which must have room for DIRSIZ bytes.
5622 // Must be called inside a transaction since it calls iput().
5623 static struct inode*
5624 namex(char *path, int nameiparent, char *name)
5625 {
5626   struct inode *ip, *next;
5627
5628   if(*path == '/')
5629     ip = iget(ROOTDEV, ROOTINO);
5630   else
5631     ip = idup(proc->cwd);
5632
5633   while((path = skipelem(path, name)) != 0){
5634     ilock(ip);
5635     if(ip->type != T_DIR){
5636       iunlockput(ip);
5637       return 0;
5638     }
5639     if(nameiparent && *path == '\0'){
5640       // Stop one level early.
5641       iunlock(ip);
5642       return ip;
5643     }
5644     if((next = dirlookup(ip, name, 0)) == 0){
5645       iunlockput(ip);
5646       return 0;
5647     }
5648     iunlockput(ip);
5649     ip = next;
```



```
5650   }
5651   if(nameiparent){
5652     iput(ip);
5653     return 0;
5654   }
5655   return ip;
5656 }
5657
5658 struct inode*
5659 namei(char *path)
5660 {
5661   char name[DIRSIZ];
5662   return namex(path, 0, name);
5663 }
5664
5665 struct inode*
5666 nameiparent(char *path, char *name)
5667 {
5668   return namex(path, 1, name);
5669 }
5670
5671
5672
5673
5674
5675
5676
5677
5678
5679
5680
5681
5682
5683
5684
5685
5686
5687
5688
5689
5690
5691
5692
5693
5694
5695
5696
5697
5698
5699
```

```
5700 //
5701 // File descriptors
5702 //
5703
5704 #include "types.h"
5705 #include "defs.h"
5706 #include "param.h"
5707 #include "fs.h"
5708 #include "file.h"
5709 #include "spinlock.h"
5710
5711 struct devsw devsw[NDEV];
5712 struct {
5713   struct spinlock lock;
5714   struct file file[NFILE];
5715 } ftable;
5716
5717 void
5718 fileinit(void)
5719 {
5720   initlock(&ftable.lock, "ftable");
5721 }
5722
5723 // Allocate a file structure.
5724 struct file*
5725 filealloc(void)
5726 {
5727   struct file *f;
5728
5729   acquire(&ftable.lock);
5730   for(f = ftable.file; f < ftable.file + NFILE; f++){
5731     if(f->ref == 0){
5732       f->ref = 1;
5733       release(&ftable.lock);
5734       return f;
5735     }
5736   }
5737   release(&ftable.lock);
5738   return 0;
5739 }
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749
```



```
5750 // Increment ref count for file f.
5751 struct file*
5752 filedup(struct file *f)
5753 {
5754   acquire(&ftable.lock);
5755   if(f->ref < 1)
5756     panic("filedup");
5757   f->ref++;
5758   release(&ftable.lock);
5759   return f;
5760 }
5761
5762 // Close file f.  (Decrement ref count, close when reaches 0.)
5763 void
5764 fileclose(struct file *f)
5765 {
5766   struct file ff;
5767
5768   acquire(&ftable.lock);
5769   if(f->ref < 1)
5770     panic("fileclose");
5771   if(--f->ref > 0){
5772     release(&ftable.lock);
5773     return;
5774   }
5775   ff = *f;
5776   f->ref = 0;
5777   f->type = FD_NONE;
5778   release(&ftable.lock);
5779
5780   if(ff.type == FD_PIPE)
5781     pipeclose(ff.pipe, ff.writable);
5782   else if(ff.type == FD_INODE){
5783     begin_op();
5784     iput(ff.ip);
5785     end_op();
5786   }
5787 }
5788
5789
5790
5791
5792
5793
5794
5795
5796
5797
5798
5799
```

```
5800 // Get metadata about file f.
5801 int
5802 filestat(struct file *f, struct stat *st)
5803 {
5804   if(f->type == FD_INODE){
5805     ilock(f->ip);
5806     stati(f->ip, st);
5807     iunlock(f->ip);
5808     return 0;
5809   }
5810   return -1;
5811 }
5812
5813 // Read from file f.
5814 int
5815 fileread(struct file *f, char *addr, int n)
5816 {
5817   int r;
5818
5819   if(f->readable == 0)
5820     return -1;
5821   if(f->type == FD_PIPE)
5822     return piperead(f->pipe, addr, n);
5823   if(f->type == FD_INODE){
5824     ilock(f->ip);
5825     if((r = readi(f->ip, addr, f->off, n)) > 0)
5826       f->off += r;
5827     iunlock(f->ip);
5828     return r;
5829   }
5830   panic("fileread");
5831 }
5832
5833 // Write to file f.
5834 int
5835 filewrite(struct file *f, char *addr, int n)
5836 {
5837   int r;
5838
5839   if(f->writable == 0)
5840     return -1;
5841   if(f->type == FD_PIPE)
5842     return pipewrite(f->pipe, addr, n);
5843   if(f->type == FD_INODE){
5844     // write a few blocks at a time to avoid exceeding
5845     // the maximum log transaction size, including
5846     // i-node, indirect block, allocation blocks,
5847     // and 2 blocks of slop for non-aligned writes.
5848     // this really belongs lower down, since writei()
5849     // might be writing a device like the console.
```



```
5850     int max = ((LOGSIZE-1-1-2) / 2) * 512;
5851     int i = 0;
5852     while(i < n){
5853       int n1 = n - i;
5854       if(n1 > max)
5855         n1 = max;
5856
5857       begin_op();
5858       ilock(f->ip);
5859       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
5860         f->off += r;
5861       iunlock(f->ip);
5862       end_op();
5863
5864       if(r < 0)
5865         break;
5866       if(r != n1)
5867         panic("short filewrite");
5868       i += r;
5869     }
5870     return i == n ? n : -1;
5871   }
5872   panic("filewrite");
5873 }
5874
5875
5876
5877
5878
5879
5880
5881
5882
5883
5884
5885
5886
5887
5888
5889
5890
5891
5892
5893
5894
5895
5896
5897
5898
5899
```

```
5900 //
5901 // File-system system calls.
5902 // Mostly argument checking, since we don't trust
5903 // user code, and calls into file.c and fs.c.
5904 //
5905
5906 #include "types.h"
5907 #include "defs.h"
5908 #include "param.h"
5909 #include "stat.h"
5910 #include "mmu.h"
5911 #include "proc.h"
5912 #include "fs.h"
5913 #include "file.h"
5914 #include "fcntl.h"
5915
5916 // Fetch the nth word-sized system call argument as a file descriptor
5917 // and return both the descriptor and the corresponding struct file.
5918 static int
5919 argfd(int n, int *pfd, struct file **pf)
5920 {
5921   int fd;
5922   struct file *f;
5923
5924   if(argint(n, &fd) < 0)
5925     return -1;
5926   if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
5927     return -1;
5928   if(pfd)
5929     *pfd = fd;
5930   if(pf)
5931     *pf = f;
5932   return 0;
5933 }
5934
5935 // Allocate a file descriptor for the given file.
5936 // Takes over file reference from caller on success.
5937 static int
5938 fdalloc(struct file *f)
5939 {
5940   int fd;
5941
5942   for(fd = 0; fd < NOFILE; fd++){
5943     if(proc->ofile[fd] == 0){
5944       proc->ofile[fd] = f;
5945       return fd;
5946     }
5947   }
5948   return -1;
5949 }
```



```
5950 int
5951 sys_dup(void)
5952 {
5953   struct file *f;
5954   int fd;
5955
5956   if(argfd(0, 0, &f) < 0)
5957     return -1;
5958   if((fd=fdalloc(f)) < 0)
5959     return -1;
5960   filedup(f);
5961   return fd;
5962 }
5963
5964 int
5965 sys_read(void)
5966 {
5967   struct file *f;
5968   int n;
5969   char *p;
5970
5971   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5972     return -1;
5973   return fileread(f, p, n);
5974 }
5975
5976 int
5977 sys_write(void)
5978 {
5979   struct file *f;
5980   int n;
5981   char *p;
5982
5983   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5984     return -1;
5985   return filewrite(f, p, n);
5986 }
5987
5988 int
5989 sys_close(void)
5990 {
5991   int fd;
5992   struct file *f;
5993
5994   if(argfd(0, &fd, &f) < 0)
5995     return -1;
5996   proc->ofile[fd] = 0;
5997   fileclose(f);
5998   return 0;
5999 }
```

```
6000 int
6001 sys_fstat(void)
6002 {
6003   struct file *f;
6004   struct stat *st;
6005
6006   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
6007     return -1;
6008   return filestat(f, st);
6009 }
6010
6011 // Create the path new as a link to the same inode as old.
6012 int
6013 sys_link(void)
6014 {
6015   char name[DIRSIZ], *new, *old;
6016   struct inode *dp, *ip;
6017
6018   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
6019     return -1;
6020
6021   begin_op();
6022   if((ip = namei(old)) == 0){
6023     end_op();
6024     return -1;
6025   }
6026
6027   ilock(ip);
6028   if(ip->type == T_DIR){
6029     iunlockput(ip);
6030     end_op();
6031     return -1;
6032   }
6033
6034   ip->nlink++;
6035   iupdate(ip);
6036   iunlock(ip);
6037
6038   if((dp = nameiparent(new, name)) == 0)
6039     goto bad;
6040   ilock(dp);
6041   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
6042     iunlockput(dp);
6043     goto bad;
6044   }
6045   iunlockput(dp);
6046   iput(ip);
6047
6048   end_op();
6049
```



```
6050   return 0;
6051
6052 bad:
6053   ilock(ip);
6054   ip->nlink--;
6055   iupdate(ip);
6056   iunlockput(ip);
6057   end_op();
6058   return -1;
6059 }
6060
6061 // Is the directory dp empty except for "." and ".." ?
6062 static int
6063 isdirempty(struct inode *dp)
6064 {
6065   int off;
6066   struct dirent de;
6067
6068   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
6069     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6070       panic("isdirempty: readi");
6071     if(de.inum != 0)
6072       return 0;
6073   }
6074   return 1;
6075 }
6076
6077 int
6078 sys_unlink(void)
6079 {
6080   struct inode *ip, *dp;
6081   struct dirent de;
6082   char name[DIRSIZ], *path;
6083   uint off;
6084
6085   if(argstr(0, &path) < 0)
6086     return -1;
6087
6088   begin_op();
6089   if((dp = nameiparent(path, name)) == 0){
6090     end_op();
6091     return -1;
6092   }
6093
6094   ilock(dp);
6095
6096   // Cannot unlink "." or "..".
6097   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6098     goto bad;
6099
```

```
6100   if((ip = dirlookup(dp, name, &off)) == 0)
6101     goto bad;
6102   ilock(ip);
6103
6104   if(ip->nlink < 1)
6105     panic("unlink: nlink < 1");
6106   if(ip->type == T_DIR && !isdirempty(ip)){
6107     iunlockput(ip);
6108     goto bad;
6109   }
6110
6111   memset(&de, 0, sizeof(de));
6112   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6113     panic("unlink: writei");
6114   if(ip->type == T_DIR){
6115     dp->nlink--;
6116     iupdate(dp);
6117   }
6118   iunlockput(dp);
6119
6120   ip->nlink--;
6121   iupdate(ip);
6122   iunlockput(ip);
6123
6124   end_op();
6125
6126   return 0;
6127
6128 bad:
6129   iunlockput(dp);
6130   end_op();
6131   return -1;
6132 }
6133
6134 static struct inode*
6135 create(char *path, short type, short major, short minor)
6136 {
6137   uint off;
6138   struct inode *ip, *dp;
6139   char name[DIRSIZ];
6140
6141   if((dp = nameiparent(path, name)) == 0)
6142     return 0;
6143   ilock(dp);
6144
6145   if((ip = dirlookup(dp, name, &off)) != 0){
6146     iunlockput(dp);
6147     ilock(ip);
6148     if(type == T_FILE && ip->type == T_FILE)
6149       return ip;
```



```
6150     iunlockput(ip);
6151     return 0;
6152   }
6153
6154   if((ip = ialloc(dp->dev, type)) == 0)
6155     panic("create: ialloc");
6156
6157   ilock(ip);
6158   ip->major = major;
6159   ip->minor = minor;
6160   ip->nlink = 1;
6161   iupdate(ip);
6162
6163   if(type == T_DIR){  // Create . and .. entries.
6164     dp->nlink++;  // for ".."
6165     iupdate(dp);
6166     // No ip->nlink++ for ".": avoid cyclic ref count.
6167     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6168       panic("create dots");
6169   }
6170
6171   if(dirlink(dp, name, ip->inum) < 0)
6172     panic("create: dirlink");
6173
6174   iunlockput(dp);
6175
6176   return ip;
6177 }
6178
6179 int
6180 sys_open(void)
6181 {
6182   char *path;
6183   int fd, omode;
6184   struct file *f;
6185   struct inode *ip;
6186
6187   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6188     return -1;
6189
6190   begin_op();
6191
6192   if(omode & O_CREATE){
6193     ip = create(path, T_FILE, 0, 0);
6194     if(ip == 0){
6195       end_op();
6196       return -1;
6197     }
6198   } else {
6199     if((ip = namei(path)) == 0){
```

```
6200       end_op();
6201       return -1;
6202     }
6203     ilock(ip);
6204     if(ip->type == T_DIR && omode != O_RDONLY){
6205       iunlockput(ip);
6206       end_op();
6207       return -1;
6208     }
6209   }
6210
6211   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6212     if(f)
6213       fileclose(f);
6214     iunlockput(ip);
6215     end_op();
6216     return -1;
6217   }
6218   iunlock(ip);
6219   end_op();
6220
6221   f->type = FD_INODE;
6222   f->ip = ip;
6223   f->off = 0;
6224   f->readable = !(omode & O_WRONLY);
6225   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6226   return fd;
6227 }
6228
6229 int
6230 sys_mkdir(void)
6231 {
6232   char *path;
6233   struct inode *ip;
6234
6235   begin_op();
6236   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
6237     end_op();
6238     return -1;
6239   }
6240   iunlockput(ip);
6241   end_op();
6242   return 0;
6243 }
6244
6245
6246
6247
6248
6249
```



```
6250 int
6251 sys_mknod(void)
6252 {
6253   struct inode *ip;
6254   char *path;
6255   int len;
6256   int major, minor;
6257
6258   begin_op();
6259   if((len=argstr(0, &path)) < 0 ||
6260      argint(1, &major) < 0 ||
6261      argint(2, &minor) < 0 ||
6262      (ip = create(path, T_DEV, major, minor)) == 0){
6263     end_op();
6264     return -1;
6265   }
6266   iunlockput(ip);
6267   end_op();
6268   return 0;
6269 }
6270
6271 int
6272 sys_chdir(void)
6273 {
6274   char *path;
6275   struct inode *ip;
6276
6277   begin_op();
6278   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
6279     end_op();
6280     return -1;
6281   }
6282   ilock(ip);
6283   if(ip->type != T_DIR){
6284     iunlockput(ip);
6285     end_op();
6286     return -1;
6287   }
6288   iunlock(ip);
6289   iput(proc->cwd);
6290   end_op();
6291   proc->cwd = ip;
6292   return 0;
6293 }
6294
6295
6296
6297
6298
6299
```

```
6300 int
6301 sys_exec(void)
6302 {
6303   char *path, *argv[MAXARG];
6304   int i;
6305   uint uargv, uarg;
6306
6307   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6308     return -1;
6309   }
6310   memset(argv, 0, sizeof(argv));
6311   for(i=0;; i++){
6312     if(i >= NELEM(argv))
6313       return -1;
6314     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6315       return -1;
6316     if(uarg == 0){
6317       argv[i] = 0;
6318       break;
6319     }
6320     if(fetchstr(uarg, &argv[i]) < 0)
6321       return -1;
6322   }
6323   return exec(path, argv);
6324 }
6325
6326 int
6327 sys_pipe(void)
6328 {
6329   int *fd;
6330   struct file *rf, *wf;
6331   int fd0, fd1;
6332
6333   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6334     return -1;
6335   if(pipealloc(&rf, &wf) < 0)
6336     return -1;
6337   fd0 = -1;
6338   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6339     if(fd0 >= 0)
6340       proc->ofile[fd0] = 0;
6341     fileclose(rf);
6342     fileclose(wf);
6343     return -1;
6344   }
6345   fd[0] = fd0;
6346   fd[1] = fd1;
6347   return 0;
6348 }
6349
```



```
6350 #include "types.h"
6351 #include "param.h"
6352 #include "memlayout.h"
6353 #include "mmu.h"
6354 #include "proc.h"
6355 #include "defs.h"
6356 #include "x86.h"
6357 #include "elf.h"
6358
6359 int
6360 exec(char *path, char **argv)
6361 {
6362   char *s, *last;
6363   int i, off;
6364   uint argc, sz, sp, ustack[3+MAXARG+1];
6365   struct elfhdr elf;
6366   struct inode *ip;
6367   struct proghdr ph;
6368   pde_t *pgdir, *oldpgdir;
6369
6370   begin_op();
6371   if((ip = namei(path)) == 0){
6372     end_op();
6373     return -1;
6374   }
6375   ilock(ip);
6376   pgdir = 0;
6377
6378   // Check ELF header
6379   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
6380     goto bad;
6381   if(elf.magic != ELF_MAGIC)
6382     goto bad;
6383
6384   if((pgdir = setupkvm()) == 0)
6385     goto bad;
6386
6387   // Load program into memory.
6388   sz = 0;
6389   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6390     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6391       goto bad;
6392     if(ph.type != ELF_PROG_LOAD)
6393       continue;
6394     if(ph.memsz < ph.filesz)
6395       goto bad;
6396     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6397       goto bad;
6398     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6399       goto bad;
```

```
6400   }
6401   iunlockput(ip);
6402   end_op();
6403   ip = 0;
6404
6405   // Allocate two pages at the next page boundary.
6406   // Make the first inaccessible.  Use the second as the user stack.
6407   sz = PGROUNDUP(sz);
6408   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
6409     goto bad;
6410   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6411   sp = sz;
6412
6413   // Push argument strings, prepare rest of stack in ustack.
6414   for(argc = 0; argv[argc]; argc++) {
6415     if(argc >= MAXARG)
6416       goto bad;
6417     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6418     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6419       goto bad;
6420     ustack[3+argc] = sp;
6421   }
6422   ustack[3+argc] = 0;
6423
6424   ustack[0] = 0xffffffff;  // fake return PC
6425   ustack[1] = argc;
6426   ustack[2] = sp - (argc+1)*4;  // argv pointer
6427
6428   sp -= (3+argc+1) * 4;
6429   if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6430     goto bad;
6431
6432   // Save program name for debugging.
6433   for(last=s=path; *s; s++)
6434     if(*s == '/')
6435       last = s+1;
6436   safestrcpy(proc->name, last, sizeof(proc->name));
6437
6438   // Commit to the user image.
6439   oldpgdir = proc->pgdir;
6440   proc->pgdir = pgdir;
6441   proc->sz = sz;
6442   proc->tf->eip = elf.entry;  // main
6443   proc->tf->esp = sp;
6444   switchuvm(proc);
6445   freevm(oldpgdir);
6446   return 0;
6447
6448
6449
```



```
6450  bad:
6451    if(pgdir)
6452      freevm(pgdir);
6453    if(ip){
6454      iunlockput(ip);
6455      end_op();
6456    }
6457    return -1;
6458  }
6459
6460
6461
6462
6463
6464
6465
6466
6467
6468
6469
6470
6471
6472
6473
6474
6475
6476
6477
6478
6479
6480
6481
6482
6483
6484
6485
6486
6487
6488
6489
6490
6491
6492
6493
6494
6495
6496
6497
6498
6499
```

```
6500 #include "types.h"
6501 #include "defs.h"
6502 #include "param.h"
6503 #include "mmu.h"
6504 #include "proc.h"
6505 #include "fs.h"
6506 #include "file.h"
6507 #include "spinlock.h"
6508
6509 #define PIPESIZE 512
6510
6511 struct pipe {
6512   struct spinlock lock;
6513   char data[PIPESIZE];
6514   uint nread;     // number of bytes read
6515   uint nwrite;    // number of bytes written
6516   int readopen;   // read fd is still open
6517   int writeopen;  // write fd is still open
6518 };
6519
6520 int
6521 pipealloc(struct file **f0, struct file **f1)
6522 {
6523   struct pipe *p;
6524
6525   p = 0;
6526   *f0 = *f1 = 0;
6527   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
6528     goto bad;
6529   if((p = (struct pipe*)kalloc()) == 0)
6530     goto bad;
6531   p->readopen = 1;
6532   p->writeopen = 1;
6533   p->nwrite = 0;
6534   p->nread = 0;
6535   initlock(&p->lock, "pipe");
6536   (*f0)->type = FD_PIPE;
6537   (*f0)->readable = 1;
6538   (*f0)->writable = 0;
6539   (*f0)->pipe = p;
6540   (*f1)->type = FD_PIPE;
6541   (*f1)->readable = 0;
6542   (*f1)->writable = 1;
6543   (*f1)->pipe = p;
6544   return 0;
6545
6546
6547
6548
6549
```



```
6550  bad:
6551   if(p)
6552     kfree((char*)p);
6553   if(*f0)
6554     fileclose(*f0);
6555   if(*f1)
6556     fileclose(*f1);
6557   return -1;
6558 }
6559
6560 void
6561 pipeclose(struct pipe *p, int writable)
6562 {
6563   acquire(&p->lock);
6564   if(writable){
6565     p->writeopen = 0;
6566     wakeup(&p->nread);
6567   } else {
6568     p->readopen = 0;
6569     wakeup(&p->nwrite);
6570   }
6571   if(p->readopen == 0 && p->writeopen == 0){
6572     release(&p->lock);
6573     kfree((char*)p);
6574   } else
6575     release(&p->lock);
6576 }
6577
6578 int
6579 pipewrite(struct pipe *p, char *addr, int n)
6580 {
6581   int i;
6582
6583   acquire(&p->lock);
6584   for(i = 0; i < n; i++){
6585     while(p->nwrite == p->nread + PIPESIZE){
6586       if(p->readopen == 0 || proc->killed){
6587         release(&p->lock);
6588         return -1;
6589       }
6590       wakeup(&p->nread);
6591       sleep(&p->nwrite, &p->lock);
6592     }
6593     p->data[p->nwrite++ % PIPESIZE] = addr[i];
6594   }
6595   wakeup(&p->nread);
6596   release(&p->lock);
6597   return n;
6598 }
6599
```

```
6600 int
6601 piperead(struct pipe *p, char *addr, int n)
6602 {
6603   int i;
6604
6605   acquire(&p->lock);
6606   while(p->nread == p->nwrite && p->writeopen){
6607     if(proc->killed){
6608       release(&p->lock);
6609       return -1;
6610     }
6611     sleep(&p->nread, &p->lock);
6612   }
6613   for(i = 0; i < n; i++){
6614     if(p->nread == p->nwrite)
6615       break;
6616     addr[i] = p->data[p->nread++ % PIPESIZE];
6617   }
6618   wakeup(&p->nwrite);
6619   release(&p->lock);
6620   return i;
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 #include "types.h"
6651 #include "x86.h"
6652
6653 void*
6654 memset(void *dst, int c, uint n)
6655 {
6656   if ((int)dst%4 == 0 && n%4 == 0){
6657     c &= 0xFF;
6658     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
6659   } else
6660     stosb(dst, c, n);
6661   return dst;
6662 }
6663
6664 int
6665 memcmp(const void *v1, const void *v2, uint n)
6666 {
6667   const uchar *s1, *s2;
6668
6669   s1 = v1;
6670   s2 = v2;
6671   while(n-- > 0){
6672     if(*s1 != *s2)
6673       return *s1 - *s2;
6674     s1++, s2++;
6675   }
6676
6677   return 0;
6678 }
6679
6680 void*
6681 memmove(void *dst, const void *src, uint n)
6682 {
6683   const char *s;
6684   char *d;
6685
6686   s = src;
6687   d = dst;
6688   if(s < d && s + n > d){
6689     s += n;
6690     d += n;
6691     while(n-- > 0)
6692       *--d = *--s;
6693   } else
6694     while(n-- > 0)
6695       *d++ = *s++;
6696
6697   return dst;
6698 }
6699
```

```
6700 // memcpy exists to placate GCC.  Use memmove.
6701 void*
6702 memcpy(void *dst, const void *src, uint n)
6703 {
6704   return memmove(dst, src, n);
6705 }
6706
6707 int
6708 strncmp(const char *p, const char *q, uint n)
6709 {
6710   while(n > 0 && *p && *p == *q)
6711     n--, p++, q++;
6712   if(n == 0)
6713     return 0;
6714   return (uchar)*p - (uchar)*q;
6715 }
6716
6717 char*
6718 strncpy(char *s, const char *t, int n)
6719 {
6720   char *os;
6721
6722   os = s;
6723   while(n-- > 0 && (*s++ = *t++) != 0)
6724     ;
6725   while(n-- > 0)
6726     *s++ = 0;
6727   return os;
6728 }
6729
6730 // Like strncpy but guaranteed to NUL-terminate.
6731 char*
6732 safestrcpy(char *s, const char *t, int n)
6733 {
6734   char *os;
6735
6736   os = s;
6737   if(n <= 0)
6738     return os;
6739   while(--n > 0 && (*s++ = *t++) != 0)
6740     ;
6741   *s = 0;
6742   return os;
6743 }
6744
6745
6746
6747
6748
6749
```



```
6750 int
6751 strlen(const char *s)
6752 {
6753   int n;
6754
6755   for(n = 0; s[n]; n++)
6756     ;
6757   return n;
6758 }
6759
6760
6761
6762
6763
6764
6765
6766
6767
6768
6769
6770
6771
6772
6773
6774
6775
6776
6777
6778
6779
6780
6781
6782
6783
6784
6785
6786
6787
6788
6789
6790
6791
6792
6793
6794
6795
6796
6797
6798
6799
```

```
6800 // See MultiProcessor Specification Version 1.[14]
6801
6802 struct mp {             // floating pointer
6803   uchar signature[4];           // "_MP_"
6804   void *physaddr;               // phys addr of MP config table
6805   uchar length;                 // 1
6806   uchar specrev;                // [14]
6807   uchar checksum;               // all bytes must add up to 0
6808   uchar type;                   // MP system config type
6809   uchar imcrp;
6810   uchar reserved[3];
6811 };
6812
6813 struct mpconf {         // configuration table header
6814   uchar signature[4];           // "PCMP"
6815   ushort length;                // total table length
6816   uchar version;                // [14]
6817   uchar checksum;               // all bytes must add up to 0
6818   uchar product[20];            // product id
6819   uint *oemtable;               // OEM table pointer
6820   ushort oemlength;             // OEM table length
6821   ushort entry;                 // entry count
6822   uint *lapicaddr;              // address of local APIC
6823   ushort xlength;               // extended table length
6824   uchar xchecksum;              // extended table checksum
6825   uchar reserved;
6826 };
6827
6828 struct mpproc {         // processor table entry
6829   uchar type;                   // entry type (0)
6830   uchar apicid;                 // local APIC id
6831   uchar version;                // local APIC verison
6832   uchar flags;                  // CPU flags
6833     #define MPBOOT 0x02           // This proc is the bootstrap processor.
6834   uchar signature[4];           // CPU signature
6835   uint feature;                 // feature flags from CPUID instruction
6836   uchar reserved[8];
6837 };
6838
6839 struct mpioapic {       // I/O APIC table entry
6840   uchar type;                   // entry type (2)
6841   uchar apicno;                 // I/O APIC id
6842   uchar version;                // I/O APIC version
6843   uchar flags;                  // I/O APIC flags
6844   uint *addr;                  // I/O APIC address
6845 };
6846
6847
6848
6849
```



```
6850 // Table entry types
6851 #define MPPROC    0x00  // One per processor
6852 #define MPBUS     0x01  // One per bus
6853 #define MPIOAPIC  0x02  // One per I/O APIC
6854 #define MPIOINTR  0x03  // One per bus interrupt source
6855 #define MPLINTR   0x04  // One per system interrupt source
6856
6857 // Blank page.
6858
6859
6860
6861
6862
6863
6864
6865
6866
6867
6868
6869
6870
6871
6872
6873
6874
6875
6876
6877
6878
6879
6880
6881
6882
6883
6884
6885
6886
6887
6888
6889
6890
6891
6892
6893
6894
6895
6896
6897
6898
6899
```

```
6900 // Multiprocessor support
6901 // Search memory for MP description structures.
6902 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
6903
6904 #include "types.h"
6905 #include "defs.h"
6906 #include "param.h"
6907 #include "memlayout.h"
6908 #include "mp.h"
6909 #include "x86.h"
6910 #include "mmu.h"
6911 #include "proc.h"
6912
6913 struct cpu cpus[NCPU];
6914 static struct cpu *bcpu;
6915 int ismp;
6916 int ncpu;
6917 uchar ioapicid;
6918
6919 int
6920 mpbcpu(void)
6921 {
6922   return bcpu-cpus;
6923 }
6924
6925 static uchar
6926 sum(uchar *addr, int len)
6927 {
6928   int i, sum;
6929
6930   sum = 0;
6931   for(i=0; i<len; i++)
6932     sum += addr[i];
6933   return sum;
6934 }
6935
6936 // Look for an MP structure in the len bytes at addr.
6937 static struct mp*
6938 mpsearch1(uint a, int len)
6939 {
6940   uchar *e, *p, *addr;
6941
6942   addr = p2v(a);
6943   e = addr+len;
6944   for(p = addr; p < e; p += sizeof(struct mp))
6945     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
6946       return (struct mp*)p;
6947   return 0;
6948 }
6949
```



```
6950 // Search for the MP Floating Pointer Structure, which according to the
6951 // spec is in one of the following three locations:
6952 // 1) in the first KB of the EBDA;
6953 // 2) in the last KB of system base memory;
6954 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
6955 static struct mp*
6956 mpsearch(void)
6957 {
6958   uchar *bda;
6959   uint p;
6960   struct mp *mp;
6961
6962   bda = (uchar *) P2V(0x400);
6963   if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
6964     if((mp = mpsearch1(p, 1024)))
6965       return mp;
6966   } else {
6967     p = ((bda[0x14]<<8)|bda[0x13])*1024;
6968     if((mp = mpsearch1(p-1024, 1024)))
6969       return mp;
6970   }
6971   return mpsearch1(0xF0000, 0x10000);
6972 }
6973
6974 // Search for an MP configuration table.  For now,
6975 // don't accept the default configurations (physaddr == 0).
6976 // Check for correct signature, calculate the checksum and,
6977 // if correct, check the version.
6978 // To do: check extended table checksum.
6979 static struct mpconf*
6980 mpconfig(struct mp **pmp)
6981 {
6982   struct mpconf *conf;
6983   struct mp *mp;
6984
6985   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
6986     return 0;
6987   conf = (struct mpconf*) p2v((uint) mp->physaddr);
6988   if(memcmp(conf, "PCMP", 4) != 0)
6989     return 0;
6990   if(conf->version != 1 && conf->version != 4)
6991     return 0;
6992   if(sum((uchar*)conf, conf->length) != 0)
6993     return 0;
6994   *pmp = mp;
6995   return conf;
6996 }
6997
6998
6999
```

```
7000 void
7001 mpinit(void)
7002 {
7003   uchar *p, *e;
7004   struct mp *mp;
7005   struct mpconf *conf;
7006   struct mpproc *proc;
7007   struct mpioapic *ioapic;
7008
7009   bcpu = &cpus[0];
7010   if((conf = mpconfig(&mp)) == 0)
7011     return;
7012   ismp = 1;
7013   lapic = (uint*)conf->lapicaddr;
7014   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
7015     switch(*p){
7016     case MPPROC:
7017       proc = (struct mpproc*)p;
7018       if(ncpu != proc->apicid){
7019         cprintf("mpinit: ncpu=%d apicid=%d\n", ncpu, proc->apicid);
7020         ismp = 0;
7021       }
7022       if(proc->flags & MPBOOT)
7023         bcpu = &cpus[ncpu];
7024       cpus[ncpu].id = ncpu;
7025       ncpu++;
7026       p += sizeof(struct mpproc);
7027       continue;
7028     case MPIOAPIC:
7029       ioapic = (struct mpioapic*)p;
7030       ioapicid = ioapic->apicno;
7031       p += sizeof(struct mpioapic);
7032       continue;
7033     case MPBUS:
7034     case MPIOINTR:
7035     case MPLINTR:
7036       p += 8;
7037       continue;
7038     default:
7039       cprintf("mpinit: unknown config type %x\n", *p);
7040       ismp = 0;
7041     }
7042   }
7043   if(!ismp){
7044     // Didn't like what we found; fall back to no MP.
7045     ncpu = 1;
7046     lapic = 0;
7047     ioapicid = 0;
7048     return;
7049   }
```



```
7050   if(mp->imcrp){
7051     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
7052     // But it would on real hardware.
7053     outb(0x22, 0x70);   // Select IMCR
7054     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
7055   }
7056 }
7057
7058
7059
7060
7061
7062
7063
7064
7065
7066
7067
7068
7069
7070
7071
7072
7073
7074
7075
7076
7077
7078
7079
7080
7081
7082
7083
7084
7085
7086
7087
7088
7089
7090
7091
7092
7093
7094
7095
7096
7097
7098
7099
```

```
7100 // The local APIC manages internal (non-I/O) interrupts.
7101 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
7102 // As of 7/26/2016, Intel processor manual Chapter 10 of Volume 3
7103
7104 #include "types.h"
7105 #include "defs.h"
7106 #include "date.h"
7107 #include "memlayout.h"
7108 #include "traps.h"
7109 #include "mmu.h"
7110 #include "x86.h"
7111
7112 // Local APIC registers, divided by 4 for use as uint[] indices.
7113 #define ID      (0x0020/4)   // ID
7114 #define VER     (0x0030/4)   // Version
7115 #define TPR     (0x0080/4)   // Task Priority
7116 #define EOI     (0x00B0/4)   // EOI
7117 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
7118   #define ENABLE     0x00000100   // Unit Enable
7119 #define ESR     (0x0280/4)   // Error Status
7120 #define ICRLO   (0x0300/4)   // Interrupt Command
7121   #define INIT       0x00000500   // INIT/RESET
7122   #define STARTUP    0x00000600   // Startup IPI
7123   #define DELIVS     0x00001000   // Delivery status
7124   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
7125   #define DEASSERT   0x00000000
7126   #define LEVEL      0x00008000   // Level triggered
7127   #define BCAST      0x00080000   // Send to all APICs, including self.
7128   #define BUSY       0x00001000
7129   #define FIXED      0x00000000
7130 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
7131 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
7132   #define X1         0x0000000B   // divide counts by 1
7133   #define PERIODIC   0x00020000   // Periodic
7134 #define PCINT   (0x0340/4)   // Performance Counter LVT
7135 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
7136 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
7137 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
7138   #define MASKED     0x00010000   // Interrupt masked
7139 #define TICR    (0x0380/4)   // Timer Initial Count
7140 #define TCCR    (0x0390/4)   // Timer Current Count
7141 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
7142
7143 volatile uint *lapic;  // Initialized in mp.c
7144
7145 static void
7146 lapicw(int index, int value)
7147 {
7148   lapic[index] = value;
7149   lapic[ID];  // wait for write to finish, by reading
```



```
7150 }
7151
7152 void
7153 lapicinit(void)
7154 {
7155   if(!lapic)
7156     return;
7157
7158   // Enable local APIC; set spurious interrupt vector.
7159   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
7160
7161   // The timer repeatedly counts down at bus frequency
7162   // from lapic[TICR] and then issues an interrupt.
7163   // If xv6 cared more about precise timekeeping,
7164   // TICR would be calibrated using an external time source.
7165   lapicw(TDCR, X1);
7166   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
7167   lapicw(TICR, 10000000);
7168
7169   // Disable logical interrupt lines.
7170   lapicw(LINT0, MASKED);
7171   lapicw(LINT1, MASKED);
7172
7173   // Disable performance counter overflow interrupts
7174   // on machines that provide that interrupt entry.
7175   if(((lapic[VER]>>16) & 0xFF) >= 4)
7176     lapicw(PCINT, MASKED);
7177
7178   // Map error interrupt to IRQ_ERROR.
7179   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
7180
7181   // Clear error status register (requires back-to-back writes).
7182   lapicw(ESR, 0);
7183   lapicw(ESR, 0);
7184
7185   // Ack any outstanding interrupts.
7186   lapicw(EOI, 0);
7187
7188   // Send an Init Level De-Assert to synchronise arbitration ID's.
7189   lapicw(ICRHI, 0);
7190   lapicw(ICRLO, BCAST | INIT | LEVEL);
7191   while(lapic[ICRLO] & DELIVS)
7192     ;
7193
7194   // Enable interrupts on the APIC (but not on the processor).
7195   lapicw(TPR, 0);
7196 }
7197
7198
7199
```

```
7200 int
7201 cpunum(void)
7202 {
7203   // Cannot call cpu when interrupts are enabled:
7204   // result not guaranteed to last long enough to be used!
7205   // Would prefer to panic but even printing is chancy here:
7206   // almost everything, including cprintf and panic, calls cpu,
7207   // often indirectly through acquire and release.
7208   if(readeflags()&FL_IF){
7209     static int n;
7210     if(n++ == 0)
7211       cprintf("cpu called from %x with interrupts enabled\n",
7212         __builtin_return_address(0));
7213   }
7214
7215   if(lapic)
7216     return lapic[ID]>>24;
7217   return 0;
7218 }
7219
7220 // Acknowledge interrupt.
7221 void
7222 lapiceoi(void)
7223 {
7224   if(lapic)
7225     lapicw(EOI, 0);
7226 }
7227
7228 // Spin for a given number of microseconds.
7229 // On real hardware would want to tune this dynamically.
7230 void
7231 microdelay(int us)
7232 {
7233 }
7234
7235 #define CMOS_PORT    0x70
7236 #define CMOS_RETURN  0x71
7237
7238 // Start additional processor running entry code at addr.
7239 // See Appendix B of MultiProcessor Specification.
7240 void
7241 lapicstartap(uchar apicid, uint addr)
7242 {
7243   int i;
7244   ushort *wrv;
7245
7246   // "The BSP must initialize CMOS shutdown code to 0AH
7247   // and the warm reset vector (DWORD based at 40:67) to point at
7248   // the AP startup code prior to the [universal startup algorithm]."
7249   outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
```



```
7250   outb(CMOS_PORT+1, 0x0A);
7251   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
7252   wrv[0] = 0;
7253   wrv[1] = addr >> 4;
7254
7255   // "Universal startup algorithm."
7256   // Send INIT (level-triggered) interrupt to reset other CPU.
7257   lapicw(ICRHI, apicid<<24);
7258   lapicw(ICRLO, INIT | LEVEL | ASSERT);
7259   microdelay(200);
7260   lapicw(ICRLO, INIT | LEVEL);
7261   microdelay(100);    // should be 10ms, but too slow in Bochs!
7262
7263   // Send startup IPI (twice!) to enter code.
7264   // Regular hardware is supposed to only accept a STARTUP
7265   // when it is in the halted state due to an INIT.  So the second
7266   // should be ignored, but it is part of the official Intel algorithm.
7267   // Bochs complains about the second one.  Too bad for Bochs.
7268   for(i = 0; i < 2; i++){
7269     lapicw(ICRHI, apicid<<24);
7270     lapicw(ICRLO, STARTUP | (addr>>12));
7271     microdelay(200);
7272   }
7273 }
7274
7275 #define CMOS_STATA   0x0a
7276 #define CMOS_STATB   0x0b
7277 #define CMOS_UIP    (1 << 7)        // RTC update in progress
7278
7279 #define SECS    0x00
7280 #define MINS    0x02
7281 #define HOURS   0x04
7282 #define DAY     0x07
7283 #define MONTH   0x08
7284 #define YEAR    0x09
7285
7286 static uint cmos_read(uint reg)
7287 {
7288   outb(CMOS_PORT,  reg);
7289   microdelay(200);
7290
7291   return inb(CMOS_RETURN);
7292 }
7293
7294
7295
7296
7297
7298
7299
```

```
7300 static void fill_rtcdate(struct rtcdate *r)
7301 {
7302   r->second = cmos_read(SECS);
7303   r->minute = cmos_read(MINS);
7304   r->hour   = cmos_read(HOURS);
7305   r->day    = cmos_read(DAY);
7306   r->month  = cmos_read(MONTH);
7307   r->year   = cmos_read(YEAR);
7308 }
7309
7310 // qemu seems to use 24-hour GWT and the values are BCD encoded
7311 void cmostime(struct rtcdate *r)
7312 {
7313   struct rtcdate t1, t2;
7314   int sb, bcd;
7315
7316   sb = cmos_read(CMOS_STATB);
7317
7318   bcd = (sb & (1 << 2)) == 0;
7319
7320   // make sure CMOS doesn't modify time while we read it
7321   for (;;) {
7322     fill_rtcdate(&t1);
7323     if (cmos_read(CMOS_STATA) & CMOS_UIP)
7324         continue;
7325     fill_rtcdate(&t2);
7326     if (memcmp(&t1, &t2, sizeof(t1)) == 0)
7327       break;
7328   }
7329
7330   // convert
7331   if (bcd) {
7332 #define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
7333     CONV(second);
7334     CONV(minute);
7335     CONV(hour  );
7336     CONV(day   );
7337     CONV(month );
7338     CONV(year  );
7339 #undef     CONV
7340   }
7341
7342   *r = t1;
7343   r->year += 2000;
7344 }
7345
7346
7347
7348
7349
```



```
7350 // The I/O APIC manages hardware interrupts for an SMP system.
7351 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
7352 // See also picirq.c.
7353
7354 #include "types.h"
7355 #include "defs.h"
7356 #include "traps.h"
7357
7358 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
7359
7360 #define REG_ID     0x00  // Register index: ID
7361 #define REG_VER    0x01  // Register index: version
7362 #define REG_TABLE  0x10  // Redirection table base
7363
7364 // The redirection table starts at REG_TABLE and uses
7365 // two registers to configure each interrupt.
7366 // The first (low) register in a pair contains configuration bits.
7367 // The second (high) register contains a bitmask telling which
7368 // CPUs can serve that interrupt.
7369 #define INT_DISABLED   0x00010000  // Interrupt disabled
7370 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
7371 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
7372 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
7373
7374 volatile struct ioapic *ioapic;
7375
7376 // IO APIC MMIO structure: write reg, then read or write data.
7377 struct ioapic {
7378   uint reg;
7379   uint pad[3];
7380   uint data;
7381 };
7382
7383 static uint
7384 ioapicread(int reg)
7385 {
7386   ioapic->reg = reg;
7387   return ioapic->data;
7388 }
7389
7390 static void
7391 ioapicwrite(int reg, uint data)
7392 {
7393   ioapic->reg = reg;
7394   ioapic->data = data;
7395 }
7396
7397
7398
7399
```

```
7400 void
7401 ioapicinit(void)
7402 {
7403   int i, id, maxintr;
7404
7405   if(!ismp)
7406     return;
7407
7408   ioapic = (volatile struct ioapic*)IOAPIC;
7409   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
7410   id = ioapicread(REG_ID) >> 24;
7411   if(id != ioapicid)
7412     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
7413
7414   // Mark all interrupts edge-triggered, active high, disabled,
7415   // and not routed to any CPUs.
7416   for(i = 0; i <= maxintr; i++){
7417     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
7418     ioapicwrite(REG_TABLE+2*i+1, 0);
7419   }
7420 }
7421
7422 void
7423 ioapicenable(int irq, int cpunum)
7424 {
7425   if(!ismp)
7426     return;
7427
7428   // Mark interrupt edge-triggered, active high,
7429   // enabled, and routed to the given cpunum,
7430   // which happens to be that cpu's APIC ID.
7431   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
7432   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
7433 }
7434
7435
7436
7437
7438
7439
7440
7441
7442
7443
7444
7445
7446
7447
7448
7449
```



```
7450 // Intel 8259A programmable interrupt controllers.
7451
7452 #include "types.h"
7453 #include "x86.h"
7454 #include "traps.h"
7455
7456 // I/O Addresses of the two programmable interrupt controllers
7457 #define IO_PIC1         0x20    // Master (IRQs 0-7)
7458 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
7459
7460 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
7461
7462 // Current IRQ mask.
7463 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
7464 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
7465
7466 static void
7467 picsetmask(ushort mask)
7468 {
7469   irqmask = mask;
7470   outb(IO_PIC1+1, mask);
7471   outb(IO_PIC2+1, mask >> 8);
7472 }
7473
7474 void
7475 picenable(int irq)
7476 {
7477   picsetmask(irqmask & ~(1<<irq));
7478 }
7479
7480 // Initialize the 8259A interrupt controllers.
7481 void
7482 picinit(void)
7483 {
7484   // mask all interrupts
7485   outb(IO_PIC1+1, 0xFF);
7486   outb(IO_PIC2+1, 0xFF);
7487
7488   // Set up master (8259A-1)
7489
7490   // ICW1:  0001g0hi
7491   //    g:  0 = edge triggering, 1 = level triggering
7492   //    h:  0 = cascaded PICs, 1 = master only
7493   //    i:  0 = no ICW4, 1 = ICW4 required
7494   outb(IO_PIC1, 0x11);
7495
7496   // ICW2:  Vector offset
7497   outb(IO_PIC1+1, T_IRQ0);
7498
7499
```

```
7500   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
7501   //        (slave PIC) 3-bit # of slave's connection to master
7502   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
7503
7504   // ICW4:  000nbmap
7505   //    n:  1 = special fully nested mode
7506   //    b:  1 = buffered mode
7507   //    m:  0 = slave PIC, 1 = master PIC
7508   //      (ignored when b is 0, as the master/slave role
7509   //      can be hardwired).
7510   //    a:  1 = Automatic EOI mode
7511   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
7512   outb(IO_PIC1+1, 0x3);
7513
7514   // Set up slave (8259A-2)
7515   outb(IO_PIC2, 0x11);                  // ICW1
7516   outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
7517   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
7518   // NB Automatic EOI mode doesn't tend to work on the slave.
7519   // Linux source code says it's "to be investigated".
7520   outb(IO_PIC2+1, 0x3);                 // ICW4
7521
7522   // OCW3:  0ef01prs
7523   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
7524   //    p:  0 = no polling, 1 = polling mode
7525   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
7526   outb(IO_PIC1, 0x68);             // clear specific mask
7527   outb(IO_PIC1, 0x0a);             // read IRR by default
7528
7529   outb(IO_PIC2, 0x68);             // OCW3
7530   outb(IO_PIC2, 0x0a);             // OCW3
7531
7532   if(irqmask != 0xFFFF)
7533     picsetmask(irqmask);
7534 }
7535
7536
7537
7538
7539
7540
7541
7542
7543
7544
7545
7546
7547
7548
7549
```



```
7550 // PC keyboard interface constants
7551
7552 #define KBSTATP         0x64    // kbd controller status port(I)
7553 #define KBS_DIB         0x01    // kbd data in buffer
7554 #define KBDATAP         0x60    // kbd data port(I)
7555
7556 #define NO              0
7557
7558 #define SHIFT           (1<<0)
7559 #define CTL             (1<<1)
7560 #define ALT             (1<<2)
7561
7562 #define CAPSLOCK        (1<<3)
7563 #define NUMLOCK         (1<<4)
7564 #define SCROLLLOCK      (1<<5)
7565
7566 #define E0ESC           (1<<6)
7567
7568 // Special keycodes
7569 #define KEY_HOME        0xE0
7570 #define KEY_END         0xE1
7571 #define KEY_UP          0xE2
7572 #define KEY_DN          0xE3
7573 #define KEY_LF          0xE4
7574 #define KEY_RT          0xE5
7575 #define KEY_PGUP        0xE6
7576 #define KEY_PGDN        0xE7
7577 #define KEY_INS         0xE8
7578 #define KEY_DEL         0xE9
7579
7580 // C('A') == Control-A
7581 #define C(x) (x - '@')
7582
7583 static uchar shiftcode[256] =
7584 {
7585   [0x1D] CTL,
7586   [0x2A] SHIFT,
7587   [0x36] SHIFT,
7588   [0x38] ALT,
7589   [0x9D] CTL,
7590   [0xB8] ALT
7591 };
7592
7593 static uchar togglecode[256] =
7594 {
7595   [0x3A] CAPSLOCK,
7596   [0x45] NUMLOCK,
7597   [0x46] SCROLLLOCK
7598 };
7599
```

```
7600 static uchar normalmap[256] =
7601 {
7602   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
7603   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
7604   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
7605   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
7606   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
7607   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
7608   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
7609   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7610   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7611   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7612   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7613   [0x9C] '\n',      // KP_Enter
7614   [0xB5] '/',       // KP_Div
7615   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7616   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7617   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7618   [0x97] KEY_HOME,  [0xCF] KEY_END,
7619   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7620 };
7621
7622 static uchar shiftmap[256] =
7623 {
7624   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
7625   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
7626   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
7627   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
7628   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
7629   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
7630   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
7631   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7632   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7633   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7634   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7635   [0x9C] '\n',      // KP_Enter
7636   [0xB5] '/',       // KP_Div
7637   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7638   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7639   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7640   [0x97] KEY_HOME,  [0xCF] KEY_END,
7641   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7642 };
7643
7644
7645
7646
7647
7648
7649
```



```
7650 static uchar ctlmap[256] =
7651 {
7652   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7653   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7654   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
7655   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
7656   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
7657   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
7658   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
7659   [0x9C] '\r',      // KP_Enter
7660   [0xB5] C('/'),    // KP_Div
7661   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7662   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7663   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7664   [0x97] KEY_HOME,  [0xCF] KEY_END,
7665   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7666 };
7667
7668
7669
7670
7671
7672
7673
7674
7675
7676
7677
7678
7679
7680
7681
7682
7683
7684
7685
7686
7687
7688
7689
7690
7691
7692
7693
7694
7695
7696
7697
7698
7699
```

```
7700 #include "types.h"
7701 #include "x86.h"
7702 #include "defs.h"
7703 #include "kbd.h"
7704
7705 int
7706 kbdgetc(void)
7707 {
7708   static uint shift;
7709   static uchar *charcode[4] = {
7710     normalmap, shiftmap, ctlmap, ctlmap
7711   };
7712   uint st, data, c;
7713
7714   st = inb(KBSTATP);
7715   if((st & KBS_DIB) == 0)
7716     return -1;
7717   data = inb(KBDATAP);
7718
7719   if(data == 0xE0){
7720     shift |= E0ESC;
7721     return 0;
7722   } else if(data & 0x80){
7723     // Key released
7724     data = (shift & E0ESC ? data : data & 0x7F);
7725     shift &= ~(shiftcode[data] | E0ESC);
7726     return 0;
7727   } else if(shift & E0ESC){
7728     // Last character was an E0 escape; or with 0x80
7729     data |= 0x80;
7730     shift &= ~E0ESC;
7731   }
7732
7733   shift |= shiftcode[data];
7734   shift ^= togglecode[data];
7735   c = charcode[shift & (CTL | SHIFT)][data];
7736   if(shift & CAPSLOCK){
7737     if('a' <= c && c <= 'z')
7738       c += 'A' - 'a';
7739     else if('A' <= c && c <= 'Z')
7740       c += 'a' - 'A';
7741   }
7742   return c;
7743 }
7744
7745 void
7746 kbdintr(void)
7747 {
7748   consoleintr(kbdgetc);
7749 }
```



```
7750 // Console input and output.
7751 // Input is from the keyboard or serial port.
7752 // Output is written to the screen and serial port.
7753
7754 #include "types.h"
7755 #include "defs.h"
7756 #include "param.h"
7757 #include "traps.h"
7758 #include "spinlock.h"
7759 #include "fs.h"
7760 #include "file.h"
7761 #include "memlayout.h"
7762 #include "mmu.h"
7763 #include "proc.h"
7764 #include "x86.h"
7765
7766 static void consputc(int);
7767
7768 static int panicked = 0;
7769
7770 static struct {
7771   struct spinlock lock;
7772   int locking;
7773 } cons;
7774
7775 static void
7776 printint(int xx, int base, int sign)
7777 {
7778   static char digits[] = "0123456789abcdef";
7779   char buf[16];
7780   int i;
7781   uint x;
7782
7783   if(sign && (sign = xx < 0))
7784     x = -xx;
7785   else
7786     x = xx;
7787
7788   i = 0;
7789   do{
7790     buf[i++] = digits[x % base];
7791   }while((x /= base) != 0);
7792
7793   if(sign)
7794     buf[i++] = '-';
7795
7796   while(--i >= 0)
7797     consputc(buf[i]);
7798 }
7799
```

```
7800 // Print to the console. only understands %d, %x, %p, %s.
7801 void
7802 cprintf(char *fmt, ...)
7803 {
7804   int i, c, locking;
7805   uint *argp;
7806   char *s;
7807
7808   locking = cons.locking;
7809   if(locking)
7810     acquire(&cons.lock);
7811
7812   if (fmt == 0)
7813     panic("null fmt");
7814
7815   argp = (uint*)(void*)(&fmt + 1);
7816   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
7817     if(c != '%'){
7818       consputc(c);
7819       continue;
7820     }
7821     c = fmt[++i] & 0xff;
7822     if(c == 0)
7823       break;
7824     switch(c){
7825     case 'd':
7826       printint(*argp++, 10, 1);
7827       break;
7828     case 'x':
7829     case 'p':
7830       printint(*argp++, 16, 0);
7831       break;
7832     case 's':
7833       if((s = (char*)*argp++) == 0)
7834         s = "(null)";
7835       for(; *s; s++)
7836         consputc(*s);
7837       break;
7838     case '%':
7839       consputc('%');
7840       break;
7841     default:
7842       // Print unknown % sequence to draw attention.
7843       consputc('%');
7844       consputc(c);
7845       break;
7846     }
7847   }
7848
7849
```



```
7850   if(locking)
7851     release(&cons.lock);
7852 }
7853
7854 void
7855 panic(char *s)
7856 {
7857   int i;
7858   uint pcs[10];
7859
7860   cli();
7861   cons.locking = 0;
7862   cprintf("cpu%d: panic: ", cpu->id);
7863   cprintf(s);
7864   cprintf("\n");
7865   getcallerpcs(&s, pcs);
7866   for(i=0; i<10; i++)
7867     cprintf(" %p", pcs[i]);
7868   panicked = 1; // freeze other CPU
7869   for(;;)
7870     ;
7871 }
7872
7873 #define BACKSPACE 0x100
7874 #define CRTPORT 0x3d4
7875 static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory
7876
7877 static void
7878 cgaputc(int c)
7879 {
7880   int pos;
7881
7882   // Cursor position: col + 80*row.
7883   outb(CRTPORT, 14);
7884   pos = inb(CRTPORT+1) << 8;
7885   outb(CRTPORT, 15);
7886   pos |= inb(CRTPORT+1);
7887
7888   if(c == '\n')
7889     pos += 80 - pos%80;
7890   else if(c == BACKSPACE){
7891     if(pos > 0) --pos;
7892   } else
7893     crt[pos++] = (c&0xff) | 0x0700;  // black on white
7894
7895   if(pos < 0 || pos > 25*80)
7896     panic("pos under/overflow");
7897
7898
7899
```

```
7900   if((pos/80) >= 24){  // Scroll up.
7901     memmove(crt, crt+80, sizeof(crt[0])*23*80);
7902     pos -= 80;
7903     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
7904   }
7905
7906   outb(CRTPORT, 14);
7907   outb(CRTPORT+1, pos>>8);
7908   outb(CRTPORT, 15);
7909   outb(CRTPORT+1, pos);
7910   crt[pos] = ' ' | 0x0700;
7911 }
7912
7913 void
7914 consputc(int c)
7915 {
7916   if(panicked){
7917     cli();
7918     for(;;)
7919       ;
7920   }
7921
7922   if(c == BACKSPACE){
7923     uartputc('\b'); uartputc(' '); uartputc('\b');
7924   } else
7925     uartputc(c);
7926   cgaputc(c);
7927 }
7928
7929 #define INPUT_BUF 128
7930 struct {
7931   char buf[INPUT_BUF];
7932   uint r;  // Read index
7933   uint w;  // Write index
7934   uint e;  // Edit index
7935 } input;
7936
7937 #define C(x)  ((x)-'@')  // Control-x
7938
7939 void
7940 consoleintr(int (*getc)(void))
7941 {
7942   int c, doprocdump = 0;
7943
7944   acquire(&cons.lock);
7945   while((c = getc()) >= 0){
7946     switch(c){
7947     case C('P'):  // Process listing.
7948       doprocdump = 1;   // procdump() locks cons.lock indirectly; invoke lat
7949       break;
```



```
7950     case C('U'):  // Kill line.
7951       while(input.e != input.w &&
7952             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
7953         input.e--;
7954         consputc(BACKSPACE);
7955       }
7956       break;
7957     case C('H'): case '\x7f':  // Backspace
7958       if(input.e != input.w){
7959         input.e--;
7960         consputc(BACKSPACE);
7961       }
7962       break;
7963     default:
7964       if(c != 0 && input.e-input.r < INPUT_BUF){
7965         c = (c == '\r') ? '\n' : c;
7966         input.buf[input.e++ % INPUT_BUF] = c;
7967         consputc(c);
7968         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
7969           input.w = input.e;
7970           wakeup(&input.r);
7971         }
7972       }
7973       break;
7974     }
7975   }
7976   release(&cons.lock);
7977   if(doprocdump) {
7978     procdump();  // now call procdump() wo. cons.lock held
7979   }
7980 }
7981
7982 int
7983 consoleread(struct inode *ip, char *dst, int n)
7984 {
7985   uint target;
7986   int c;
7987
7988   iunlock(ip);
7989   target = n;
7990   acquire(&cons.lock);
7991   while(n > 0){
7992     while(input.r == input.w){
7993       if(proc->killed){
7994         release(&cons.lock);
7995         ilock(ip);
7996         return -1;
7997       }
7998       sleep(&input.r, &cons.lock);
7999     }
```

```
8000     c = input.buf[input.r++ % INPUT_BUF];
8001     if(c == C('D')){  // EOF
8002       if(n < target){
8003         // Save ^D for next time, to make sure
8004         // caller gets a 0-byte result.
8005         input.r--;
8006       }
8007       break;
8008     }
8009     *dst++ = c;
8010     --n;
8011     if(c == '\n')
8012       break;
8013   }
8014   release(&cons.lock);
8015   ilock(ip);
8016
8017   return target - n;
8018 }
8019
8020 int
8021 consolewrite(struct inode *ip, char *buf, int n)
8022 {
8023   int i;
8024
8025   iunlock(ip);
8026   acquire(&cons.lock);
8027   for(i = 0; i < n; i++)
8028     consputc(buf[i] & 0xff);
8029   release(&cons.lock);
8030   ilock(ip);
8031
8032   return n;
8033 }
8034
8035 void
8036 consoleinit(void)
8037 {
8038   initlock(&cons.lock, "console");
8039
8040   devsw[CONSOLE].write = consolewrite;
8041   devsw[CONSOLE].read = consoleread;
8042   cons.locking = 1;
8043
8044   picenable(IRQ_KBD);
8045   ioapicenable(IRQ_KBD, 0);
8046 }
8047
8048
8049
```



```
8050 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
8051 // Only used on uniprocessors;
8052 // SMP machines use the local APIC timer.
8053
8054 #include "types.h"
8055 #include "defs.h"
8056 #include "traps.h"
8057 #include "x86.h"
8058
8059 #define IO_TIMER1       0x040           // 8253 Timer #1
8060
8061 // Frequency of all three count-down timers;
8062 // (TIMER_FREQ/freq) is the appropriate count
8063 // to generate a frequency of freq Hz.
8064
8065 #define TIMER_FREQ      1193182
8066 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
8067
8068 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
8069 #define TIMER_SEL0      0x00    // select counter 0
8070 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
8071 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
8072
8073 void
8074 timerinit(void)
8075 {
8076   // Interrupt 100 times/sec.
8077   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
8078   outb(IO_TIMER1, TIMER_DIV(100) % 256);
8079   outb(IO_TIMER1, TIMER_DIV(100) / 256);
8080   picenable(IRQ_TIMER);
8081 }
8082
8083
8084
8085
8086
8087
8088
8089
8090
8091
8092
8093
8094
8095
8096
8097
8098
8099
```

```
8100 // Intel 8250 serial port (UART).
8101
8102 #include "types.h"
8103 #include "defs.h"
8104 #include "param.h"
8105 #include "traps.h"
8106 #include "spinlock.h"
8107 #include "fs.h"
8108 #include "file.h"
8109 #include "mmu.h"
8110 #include "proc.h"
8111 #include "x86.h"
8112
8113 #define COM1    0x3f8
8114
8115 static int uart;    // is there a uart?
8116
8117 void
8118 uartinit(void)
8119 {
8120   char *p;
8121
8122   // Turn off the FIFO
8123   outb(COM1+2, 0);
8124
8125   // 9600 baud, 8 data bits, 1 stop bit, parity off.
8126   outb(COM1+3, 0x80);    // Unlock divisor
8127   outb(COM1+0, 115200/9600);
8128   outb(COM1+1, 0);
8129   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
8130   outb(COM1+4, 0);
8131   outb(COM1+1, 0x01);    // Enable receive interrupts.
8132
8133   // If status is 0xFF, no serial port.
8134   if(inb(COM1+5) == 0xFF)
8135     return;
8136   uart = 1;
8137
8138   // Acknowledge pre-existing interrupt conditions;
8139   // enable interrupts.
8140   inb(COM1+2);
8141   inb(COM1+0);
8142   picenable(IRQ_COM1);
8143   ioapicenable(IRQ_COM1, 0);
8144
8145   // Announce that we're here.
8146   for(p="xv6...\n"; *p; p++)
8147     uartputc(*p);
8148 }
8149
```



```
8150 void
8151 uartputc(int c)
8152 {
8153   int i;
8154
8155   if(!uart)
8156     return;
8157   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
8158     microdelay(10);
8159   outb(COM1+0, c);
8160 }
8161
8162 static int
8163 uartgetc(void)
8164 {
8165   if(!uart)
8166     return -1;
8167   if(!(inb(COM1+5) & 0x01))
8168     return -1;
8169   return inb(COM1+0);
8170 }
8171
8172 void
8173 uartintr(void)
8174 {
8175   consoleintr(uartgetc);
8176 }
8177
8178
8179
8180
8181
8182
8183
8184
8185
8186
8187
8188
8189
8190
8191
8192
8193
8194
8195
8196
8197
8198
8199
```

```
8200 # Initial process execs /init.
8201
8202 #include "syscall.h"
8203 #include "traps.h"
8204
8205
8206 # exec(init, argv)
8207 .globl start
8208 start:
8209   pushl $argv
8210   pushl $init
8211   pushl $0  // where caller pc would be
8212   movl $SYS_exec, %eax
8213   int $T_SYSCALL
8214
8215 # for(;;) exit();
8216 exit:
8217   movl $SYS_exit, %eax
8218   int $T_SYSCALL
8219   jmp exit
8220
8221 # char init[] = "/init\0";
8222 init:
8223   .string "/init\0"
8224
8225 # char *argv[] = { init, 0 };
8226 .p2align 2
8227 argv:
8228   .long init
8229   .long 0
8230
8231
8232
8233
8234
8235
8236
8237
8238
8239
8240
8241
8242
8243
8244
8245
8246
8247
8248
8249
```



```
8250 #include "syscall.h"
8251 #include "traps.h"
8252
8253 #define SYSCALL(name) \
8254   .globl name; \
8255   name: \
8256     movl $SYS_ ## name, %eax; \
8257     int $T_SYSCALL; \
8258     ret
8259
8260 SYSCALL(fork)
8261 SYSCALL(exit)
8262 SYSCALL(wait)
8263 SYSCALL(pipe)
8264 SYSCALL(read)
8265 SYSCALL(write)
8266 SYSCALL(close)
8267 SYSCALL(kill)
8268 SYSCALL(exec)
8269 SYSCALL(open)
8270 SYSCALL(mknod)
8271 SYSCALL(unlink)
8272 SYSCALL(fstat)
8273 SYSCALL(link)
8274 SYSCALL(mkdir)
8275 SYSCALL(chdir)
8276 SYSCALL(dup)
8277 SYSCALL(getpid)
8278 SYSCALL(sbrk)
8279 SYSCALL(sleep)
8280 SYSCALL(uptime)
8281 SYSCALL(halt)
8282 SYSCALL(date)
8283 SYSCALL(getuid)
8284 SYSCALL(getgid)
8285 SYSCALL(getppid)
8286 SYSCALL(setuid)
8287 SYSCALL(setgid)
8288 SYSCALL(getprocs)
8289 SYSCALL(setpriority)
8290
8291
8292
8293
8294
8295
8296
8297
8298
8299
```

```
8300 // init: The initial user-level program
8301
8302 #include "types.h"
8303 #include "stat.h"
8304 #include "user.h"
8305 #include "fcntl.h"
8306
8307 char *argv[] = { "sh", 0 };
8308
8309 int
8310 main(void)
8311 {
8312   int pid, wpid;
8313
8314   if(open("console", O_RDWR) < 0){
8315     mknod("console", 1, 1);
8316     open("console", O_RDWR);
8317   }
8318   dup(0);  // stdout
8319   dup(0);  // stderr
8320
8321   for(;;){
8322     printf(1, "init: starting sh\n");
8323     pid = fork();
8324     if(pid < 0){
8325       printf(1, "init: fork failed\n");
8326       exit();
8327     }
8328     if(pid == 0){
8329       exec("sh", argv);
8330       printf(1, "init: exec sh failed\n");
8331       exit();
8332     }
8333     while((wpid=wait()) >= 0 && wpid != pid)
8334       printf(1, "zombie!\n");
8335   }
8336 }
8337
8338
8339
8340
8341
8342
8343
8344
8345
8346
8347
8348
8349
```



```
8350 // Shell.
8351 // 2015-12-21. Added very simple processing for builtin commands
8352
8353 #include "types.h"
8354 #include "user.h"
8355 #include "fcntl.h"
8356
8357 // Parsed command representation
8358 #define EXEC  1
8359 #define REDIR 2
8360 #define PIPE  3
8361 #define LIST  4
8362 #define BACK  5
8363
8364 #define MAXARGS 10
8365
8366 struct cmd {
8367   int type;
8368 };
8369
8370 struct execcmd {
8371   int type;
8372   char *argv[MAXARGS];
8373   char *eargv[MAXARGS];
8374 };
8375
8376 struct redircmd {
8377   int type;
8378   struct cmd *cmd;
8379   char *file;
8380   char *efile;
8381   int mode;
8382   int fd;
8383 };
8384
8385 struct pipecmd {
8386   int type;
8387   struct cmd *left;
8388   struct cmd *right;
8389 };
8390
8391 struct listcmd {
8392   int type;
8393   struct cmd *left;
8394   struct cmd *right;
8395 };
8396
8397
8398
8399
```

```
8400 struct backcmd {
8401   int type;
8402   struct cmd *cmd;
8403 };
8404
8405 int fork1(void);  // Fork but panics on failure.
8406 void panic(char*);
8407 struct cmd *parsecmd(char*);
8408
8409 // Execute cmd.  Never returns.
8410 void
8411 runcmd(struct cmd *cmd)
8412 {
8413   int p[2];
8414   struct backcmd *bcmd;
8415   struct execcmd *ecmd;
8416   struct listcmd *lcmd;
8417   struct pipecmd *pcmd;
8418   struct redircmd *rcmd;
8419
8420   if(cmd == 0)
8421     exit();
8422
8423   switch(cmd->type){
8424   default:
8425     panic("runcmd");
8426
8427   case EXEC:
8428     ecmd = (struct execcmd*)cmd;
8429     if(ecmd->argv[0] == 0)
8430       exit();
8431     exec(ecmd->argv[0], ecmd->argv);
8432     printf(2, "exec %s failed\n", ecmd->argv[0]);
8433     break;
8434
8435   case REDIR:
8436     rcmd = (struct redircmd*)cmd;
8437     close(rcmd->fd);
8438     if(open(rcmd->file, rcmd->mode) < 0){
8439       printf(2, "open %s failed\n", rcmd->file);
8440       exit();
8441     }
8442     runcmd(rcmd->cmd);
8443     break;
8444
8445   case LIST:
8446     lcmd = (struct listcmd*)cmd;
8447     if(fork1() == 0)
8448       runcmd(lcmd->left);
8449     wait();
```



```
8450     runcmd(lcmd->right);
8451     break;
8452
8453   case PIPE:
8454     pcmd = (struct pipecmd*)cmd;
8455     if(pipe(p) < 0)
8456       panic("pipe");
8457     if(fork1() == 0){
8458       close(1);
8459       dup(p[1]);
8460       close(p[0]);
8461       close(p[1]);
8462       runcmd(pcmd->left);
8463     }
8464     if(fork1() == 0){
8465       close(0);
8466       dup(p[0]);
8467       close(p[0]);
8468       close(p[1]);
8469       runcmd(pcmd->right);
8470     }
8471     close(p[0]);
8472     close(p[1]);
8473     wait();
8474     wait();
8475     break;
8476
8477   case BACK:
8478     bcmd = (struct backcmd*)cmd;
8479     if(fork1() == 0)
8480       runcmd(bcmd->cmd);
8481     break;
8482   }
8483   exit();
8484 }
8485
8486 int
8487 getcmd(char *buf, int nbuf)
8488 {
8489   printf(2, "$ ");
8490   memset(buf, 0, nbuf);
8491   gets(buf, nbuf);
8492   if(buf[0] == 0) // EOF
8493     return -1;
8494   return 0;
8495 }
8496
8497
8498
8499
```

```
8500 #ifdef USE_BUILTINS
8501 // ***** processing for shell builtins begins here *****
8502
8503 int
8504 strncmp(const char *p, const char *q, uint n)
8505 {
8506     while(n > 0 && *p && *p == *q)
8507       n--, p++, q++;
8508     if(n == 0)
8509       return 0;
8510     return (uchar)*p - (uchar)*q;
8511 }
8512
8513 int
8514 makeint(char *p)
8515 {
8516   int val = 0;
8517
8518   while ((*p >= '0') && (*p <= '9')) {
8519     val = 10*val + (*p-'0');
8520     ++p;
8521   }
8522   return val;
8523 }
8524
8525 int
8526 setbuiltin(char *p)
8527 {
8528   int i;
8529
8530   p += strlen("_set");
8531   while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8532   if (strncmp("uid", p, 3) == 0) {
8533     p += strlen("uid");
8534     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8535     i = makeint(p); // ugly
8536     return (setuid(i));
8537   } else
8538   if (strncmp("gid", p, 3) == 0) {
8539     p += strlen("gid");
8540     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8541     i = makeint(p); // ugly
8542     return (setgid(i));
8543   }
8544   printf(2, "Invalid _set parameter\n");
8545   return -1;
8546 }
8547
8548
8549
```



```
8550 int
8551 getbuiltin(char *p)
8552 {
8553   p += strlen("_get");
8554   while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8555   if (strncmp("uid", p, 3) == 0) {
8556     printf(2, "%d\n", getuid());
8557     return 0;
8558   }
8559   if (strncmp("gid", p, 3) == 0) {
8560     printf(2, "%d\n", getgid());
8561     return 0;
8562   }
8563   printf(2, "Invalid _get parameter\n");
8564   return -1;
8565 }
8566
8567 typedef int funcPtr_t(char *);
8568 typedef struct {
8569   char       *cmd;
8570   funcPtr_t  *name;
8571 } dispatchTableEntry_t;
8572
8573 // Use a simple function dispatch table (FDT) to process builtin commands
8574 dispatchTableEntry_t fdt[] = {
8575   {"_set", setbuiltin},
8576   {"_get", getbuiltin}
8577 };
8578 int FDTcount = sizeof(fdt) / sizeof(fdt[0]); // # entris in FDT
8579
8580 void
8581 dobuiltin(char *cmd) {
8582   int i;
8583
8584   for (i=0; i<FDTcount; i++)
8585     if (strncmp(cmd, fdt[i].cmd, strlen(fdt[i].cmd)) == 0)
8586       (*fdt[i].name)(cmd);
8587 }
8588
8589
8590
8591
8592
8593
8594
8595
8596
8597
8598
8599
```

```
8600 // ***** processing for shell builtins ends here *****
8601 #endif
8602
8603 int
8604 main(void)
8605 {
8606   static char buf[100];
8607   int fd;
8608
8609   // Assumes three file descriptors open.
8610   while((fd = open("console", O_RDWR)) >= 0){
8611     if(fd >= 3){
8612       close(fd);
8613       break;
8614     }
8615   }
8616
8617   // Read and run input commands.
8618   while(getcmd(buf, sizeof(buf)) >= 0){
8619 // add support for built-ins here. cd is a built-in
8620     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
8621       // Clumsy but will have to do for now.
8622       // Chdir has no effect on the parent if run in the child.
8623       buf[strlen(buf)-1] = 0;  // chop \n
8624       if(chdir(buf+3) < 0)
8625         printf(2, "cannot cd %s\n", buf+3);
8626       continue;
8627     }
8628 #ifdef USE_BUILTINS
8629     if (buf[0]=='_') {     // assume it is a builtin command
8630       dobuiltin(buf);
8631       continue;
8632     }
8633 #endif
8634     if(fork1() == 0)
8635       runcmd(parsecmd(buf));
8636     wait();
8637   }
8638   exit();
8639 }
8640
8641 void
8642 panic(char *s)
8643 {
8644   printf(2, "%s\n", s);
8645   exit();
8646 }
8647
8648
8649
```



```
8650 int
8651 fork1(void)
8652 {
8653   int pid;
8654
8655   pid = fork();
8656   if(pid == -1)
8657     panic("fork");
8658   return pid;
8659 }
8660
8661 // Constructors
8662
8663 struct cmd*
8664 execcmd(void)
8665 {
8666   struct execcmd *cmd;
8667
8668   cmd = malloc(sizeof(*cmd));
8669   memset(cmd, 0, sizeof(*cmd));
8670   cmd->type = EXEC;
8671   return (struct cmd*)cmd;
8672 }
8673
8674 struct cmd*
8675 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
8676 {
8677   struct redircmd *cmd;
8678
8679   cmd = malloc(sizeof(*cmd));
8680   memset(cmd, 0, sizeof(*cmd));
8681   cmd->type = REDIR;
8682   cmd->cmd = subcmd;
8683   cmd->file = file;
8684   cmd->efile = efile;
8685   cmd->mode = mode;
8686   cmd->fd = fd;
8687   return (struct cmd*)cmd;
8688 }
8689
8690
8691
8692
8693
8694
8695
8696
8697
8698
8699
```

```
8700 struct cmd*
8701 pipecmd(struct cmd *left, struct cmd *right)
8702 {
8703   struct pipecmd *cmd;
8704
8705   cmd = malloc(sizeof(*cmd));
8706   memset(cmd, 0, sizeof(*cmd));
8707   cmd->type = PIPE;
8708   cmd->left = left;
8709   cmd->right = right;
8710   return (struct cmd*)cmd;
8711 }
8712
8713 struct cmd*
8714 listcmd(struct cmd *left, struct cmd *right)
8715 {
8716   struct listcmd *cmd;
8717
8718   cmd = malloc(sizeof(*cmd));
8719   memset(cmd, 0, sizeof(*cmd));
8720   cmd->type = LIST;
8721   cmd->left = left;
8722   cmd->right = right;
8723   return (struct cmd*)cmd;
8724 }
8725
8726 struct cmd*
8727 backcmd(struct cmd *subcmd)
8728 {
8729   struct backcmd *cmd;
8730
8731   cmd = malloc(sizeof(*cmd));
8732   memset(cmd, 0, sizeof(*cmd));
8733   cmd->type = BACK;
8734   cmd->cmd = subcmd;
8735   return (struct cmd*)cmd;
8736 }
8737
8738
8739
8740
8741
8742
8743
8744
8745
8746
8747
8748
8749
```



```
8750 // Parsing
8751
8752 char whitespace[] = " \t\r\n\v";
8753 char symbols[] = "<|>&;()";
8754
8755 int
8756 gettoken(char **ps, char *es, char **q, char **eq)
8757 {
8758   char *s;
8759   int ret;
8760
8761   s = *ps;
8762   while(s < es && strchr(whitespace, *s))
8763     s++;
8764   if(q)
8765     *q = s;
8766   ret = *s;
8767   switch(*s){
8768   case 0:
8769     break;
8770   case '|':
8771   case '(':
8772   case ')':
8773   case ';':
8774   case '&':
8775   case '<':
8776     s++;
8777     break;
8778   case '>':
8779     s++;
8780     if(*s == '>'){
8781       ret = '+';
8782       s++;
8783     }
8784     break;
8785   default:
8786     ret = 'a';
8787     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
8788       s++;
8789     break;
8790   }
8791   if(eq)
8792     *eq = s;
8793
8794   while(s < es && strchr(whitespace, *s))
8795     s++;
8796   *ps = s;
8797   return ret;
8798 }
8799
```

```
8800 int
8801 peek(char **ps, char *es, char *toks)
8802 {
8803   char *s;
8804
8805   s = *ps;
8806   while(s < es && strchr(whitespace, *s))
8807     s++;
8808   *ps = s;
8809   return *s && strchr(toks, *s);
8810 }
8811
8812 struct cmd *parseline(char**, char*);
8813 struct cmd *parsepipe(char**, char*);
8814 struct cmd *parseexec(char**, char*);
8815 struct cmd *nulterminate(struct cmd*);
8816
8817 struct cmd*
8818 parsecmd(char *s)
8819 {
8820   char *es;
8821   struct cmd *cmd;
8822
8823   es = s + strlen(s);
8824   cmd = parseline(&s, es);
8825   peek(&s, es, "");
8826   if(s != es){
8827     printf(2, "leftovers: %s\n", s);
8828     panic("syntax");
8829   }
8830   nulterminate(cmd);
8831   return cmd;
8832 }
8833
8834 struct cmd*
8835 parseline(char **ps, char *es)
8836 {
8837   struct cmd *cmd;
8838
8839   cmd = parsepipe(ps, es);
8840   while(peek(ps, es, "&")){
8841     gettoken(ps, es, 0, 0);
8842     cmd = backcmd(cmd);
8843   }
8844   if(peek(ps, es, ";")){
8845     gettoken(ps, es, 0, 0);
8846     cmd = listcmd(cmd, parseline(ps, es));
8847   }
8848   return cmd;
8849 }
```



```
8850 struct cmd*
8851 parsepipe(char **ps, char *es)
8852 {
8853   struct cmd *cmd;
8854
8855   cmd = parseexec(ps, es);
8856   if(peek(ps, es, "|")){
8857     gettoken(ps, es, 0, 0);
8858     cmd = pipecmd(cmd, parsepipe(ps, es));
8859   }
8860   return cmd;
8861 }
8862
8863 struct cmd*
8864 parseredirs(struct cmd *cmd, char **ps, char *es)
8865 {
8866   int tok;
8867   char *q, *eq;
8868
8869   while(peek(ps, es, "<>")){
8870     tok = gettoken(ps, es, 0, 0);
8871     if(gettoken(ps, es, &q, &eq) != 'a')
8872       panic("missing file for redirection");
8873     switch(tok){
8874     case '<':
8875       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
8876       break;
8877     case '>':
8878       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8879       break;
8880     case '+':  // >>
8881       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8882       break;
8883     }
8884   }
8885   return cmd;
8886 }
8887
8888
8889
8890
8891
8892
8893
8894
8895
8896
8897
8898
8899
```

```
8900 struct cmd*
8901 parseblock(char **ps, char *es)
8902 {
8903   struct cmd *cmd;
8904
8905   if(!peek(ps, es, "("))
8906     panic("parseblock");
8907   gettoken(ps, es, 0, 0);
8908   cmd = parseline(ps, es);
8909   if(!peek(ps, es, ")"))
8910     panic("syntax - missing )");
8911   gettoken(ps, es, 0, 0);
8912   cmd = parseredirs(cmd, ps, es);
8913   return cmd;
8914 }
8915
8916 struct cmd*
8917 parseexec(char **ps, char *es)
8918 {
8919   char *q, *eq;
8920   int tok, argc;
8921   struct execcmd *cmd;
8922   struct cmd *ret;
8923
8924   if(peek(ps, es, "("))
8925     return parseblock(ps, es);
8926
8927   ret = execcmd();
8928   cmd = (struct execcmd*)ret;
8929
8930   argc = 0;
8931   ret = parseredirs(ret, ps, es);
8932   while(!peek(ps, es, "|)&;")){
8933     if((tok=gettoken(ps, es, &q, &eq)) == 0)
8934       break;
8935     if(tok != 'a')
8936       panic("syntax");
8937     cmd->argv[argc] = q;
8938     cmd->eargv[argc] = eq;
8939     argc++;
8940     if(argc >= MAXARGS)
8941       panic("too many args");
8942     ret = parseredirs(ret, ps, es);
8943   }
8944   cmd->argv[argc] = 0;
8945   cmd->eargv[argc] = 0;
8946   return ret;
8947 }
8948
8949
```



```
8950 // NUL-terminate all the counted strings.
8951 struct cmd*
8952 nulterminate(struct cmd *cmd)
8953 {
8954   int i;
8955   struct backcmd *bcmd;
8956   struct execcmd *ecmd;
8957   struct listcmd *lcmd;
8958   struct pipecmd *pcmd;
8959   struct redircmd *rcmd;
8960
8961   if(cmd == 0)
8962     return 0;
8963
8964   switch(cmd->type){
8965   case EXEC:
8966     ecmd = (struct execcmd*)cmd;
8967     for(i=0; ecmd->argv[i]; i++)
8968       *ecmd->eargv[i] = 0;
8969     break;
8970
8971   case REDIR:
8972     rcmd = (struct redircmd*)cmd;
8973     nulterminate(rcmd->cmd);
8974     *rcmd->efile = 0;
8975     break;
8976
8977   case PIPE:
8978     pcmd = (struct pipecmd*)cmd;
8979     nulterminate(pcmd->left);
8980     nulterminate(pcmd->right);
8981     break;
8982
8983   case LIST:
8984     lcmd = (struct listcmd*)cmd;
8985     nulterminate(lcmd->left);
8986     nulterminate(lcmd->right);
8987     break;
8988
8989   case BACK:
8990     bcmd = (struct backcmd*)cmd;
8991     nulterminate(bcmd->cmd);
8992     break;
8993   }
8994   return cmd;
8995 }
8996
8997
8998
8999
```

```
9000 #include "asm.h"
9001 #include "memlayout.h"
9002 #include "mmu.h"
9003
9004 # Start the first CPU: switch to 32-bit protected mode, jump into C.
9005 # The BIOS loads this code from the first sector of the hard disk into
9006 # memory at physical address 0x7c00 and starts executing in real mode
9007 # with %cs=0 %ip=7c00.
9008
9009 .code16                       # Assemble for 16-bit mode
9010 .globl start
9011 start:
9012   cli                         # BIOS enabled interrupts; disable
9013
9014   # Zero data segment registers DS, ES, and SS.
9015   xorw    %ax,%ax             # Set %ax to zero
9016   movw    %ax,%ds             # -> Data Segment
9017   movw    %ax,%es             # -> Extra Segment
9018   movw    %ax,%ss             # -> Stack Segment
9019
9020   # Physical address line A20 is tied to zero so that the first PCs
9021   # with 2 MB would run software that assumed 1 MB.  Undo that.
9022 seta20.1:
9023   inb     $0x64,%al               # Wait for not busy
9024   testb   $0x2,%al
9025   jnz     seta20.1
9026
9027   movb    $0xd1,%al               # 0xd1 -> port 0x64
9028   outb    %al,$0x64
9029
9030 seta20.2:
9031   inb     $0x64,%al               # Wait for not busy
9032   testb   $0x2,%al
9033   jnz     seta20.2
9034
9035   movb    $0xdf,%al               # 0xdf -> port 0x60
9036   outb    %al,$0x60
9037
9038   # Switch from real to protected mode.  Use a bootstrap GDT that makes
9039   # virtual addresses map directly to physical addresses so that the
9040   # effective memory map doesn't change during the transition.
9041   lgdt    gdtdesc
9042   movl    %cr0, %eax
9043   orl     $CR0_PE, %eax
9044   movl    %eax, %cr0
9045
9046   # Complete transition to 32-bit protected mode by using long jmp
9047   # to reload %cs and %eip.  The segment descriptors are set up with no
9048   # translation, so that the mapping is still the identity mapping.
9049   ljmp    $(SEG_KCODE<<3), $start32
```



```
9050 .code32  # Tell assembler to generate 32-bit code now.
9051 start32:
9052   # Set up the protected-mode data segment registers
9053   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
9054   movw    %ax, %ds                # -> DS: Data Segment
9055   movw    %ax, %es                # -> ES: Extra Segment
9056   movw    %ax, %ss                # -> SS: Stack Segment
9057   movw    $0, %ax                 # Zero segments not ready for use
9058   movw    %ax, %fs                # -> FS
9059   movw    %ax, %gs                # -> GS
9060
9061   # Set up the stack pointer and call into C.
9062   movl    $start, %esp
9063   call    bootmain
9064
9065   # If bootmain returns (it shouldn't), trigger a Bochs
9066   # breakpoint if running under Bochs, then loop.
9067   movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
9068   movw    %ax, %dx
9069   outw    %ax, %dx
9070   movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
9071   outw    %ax, %dx
9072 spin:
9073   jmp     spin
9074
9075 # Bootstrap GDT
9076 .p2align 2                                # force 4 byte alignment
9077 gdt:
9078   SEG_NULLASM                             # null seg
9079   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
9080   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
9081
9082 gdtdesc:
9083   .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
9084   .long   gdt                             # address gdt
9085
9086
9087
9088
9089
9090
9091
9092
9093
9094
9095
9096
9097
9098
9099
```

```
9100 // Boot loader.
9101 //
9102 // Part of the boot block, along with bootasm.S, which calls bootmain().
9103 // bootasm.S has put the processor into protected 32-bit mode.
9104 // bootmain() loads an ELF kernel image from the disk starting at
9105 // sector 1 and then jumps to the kernel entry routine.
9106
9107 #include "types.h"
9108 #include "elf.h"
9109 #include "x86.h"
9110 #include "memlayout.h"
9111
9112 #define SECTSIZE  512
9113
9114 void readseg(uchar*, uint, uint);
9115
9116 void
9117 bootmain(void)
9118 {
9119   struct elfhdr *elf;
9120   struct proghdr *ph, *eph;
9121   void (*entry)(void);
9122   uchar* pa;
9123
9124   elf = (struct elfhdr*)0x10000;  // scratch space
9125
9126   // Read 1st page off disk
9127   readseg((uchar*)elf, 4096, 0);
9128
9129   // Is this an ELF executable?
9130   if(elf->magic != ELF_MAGIC)
9131     return;  // let bootasm.S handle error
9132
9133   // Load each program segment (ignores ph flags).
9134   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
9135   eph = ph + elf->phnum;
9136   for(; ph < eph; ph++){
9137     pa = (uchar*)ph->paddr;
9138     readseg(pa, ph->filesz, ph->off);
9139     if(ph->memsz > ph->filesz)
9140       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
9141   }
9142
9143   // Call the entry point from the ELF header.
9144   // Does not return!
9145   entry = (void(*)(void))(elf->entry);
9146   entry();
9147 }
9148
9149
```



```
9150 void
9151 waitdisk(void)
9152 {
9153   // Wait for disk ready.
9154   while((inb(0x1F7) & 0xC0) != 0x40)
9155     ;
9156 }
9157
9158 // Read a single sector at offset into dst.
9159 void
9160 readsect(void *dst, uint offset)
9161 {
9162   // Issue command.
9163   waitdisk();
9164   outb(0x1F2, 1);   // count = 1
9165   outb(0x1F3, offset);
9166   outb(0x1F4, offset >> 8);
9167   outb(0x1F5, offset >> 16);
9168   outb(0x1F6, (offset >> 24) | 0xE0);
9169   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
9170
9171   // Read data.
9172   waitdisk();
9173   insl(0x1F0, dst, SECTSIZE/4);
9174 }
9175
9176 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
9177 // Might copy more than asked.
9178 void
9179 readseg(uchar* pa, uint count, uint offset)
9180 {
9181   uchar* epa;
9182
9183   epa = pa + count;
9184
9185   // Round down to sector boundary.
9186   pa -= offset % SECTSIZE;
9187
9188   // Translate from bytes to sectors; kernel starts at sector 1.
9189   offset = (offset / SECTSIZE) + 1;
9190
9191   // If this is too slow, we could read lots of sectors at a time.
9192   // We'd write more to memory than asked, but it doesn't matter --
9193   // we load in increasing order.
9194   for(; pa < epa; pa += SECTSIZE, offset++)
9195     readsect(pa, offset);
9196 }
9197
9198
9199
```

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

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

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

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

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

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

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

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

```



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

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

  return;
}
#endif
```

```
9300 #include "types.h"
9301 #include "user.h"
9302 #include "date.h"
9303
9304
9305 int
9306 main(int argc, char *argv[])
9307 {
9308   struct rtcdate r;
9309   if(date(&r)) {
9310     printf(2,"date failed\n");
9311     exit();
9312   }
9313   printf(1, "Current UTC time is: %d/%d/%d - %d:%d:%d\n",r.year, r.month, r.
9314
9315   exit();
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
```



```
9350 #define STRMAX 32
9351
9352 struct uproc {
9353    uint pid;
9354    uint uid;
9355    uint gid;
9356    uint ppid;
9357    uint elapsed_ticks;
9358    uint CPU_total_ticks;
9359    char state[STRMAX];
9360    uint size;
9361    char name[STRMAX];
9362 #ifdef CS333_P3
9363    int priority;
9364 #endif
9365 };
9366
9367
9368
9369
9370
9371
9372
9373
9374
9375
9376
9377
9378
9379
9380
9381
9382
9383
9384
9385
9386
9387
9388
9389
9390
9391
9392
9393
9394
9395
9396
9397
9398
9399
```

```
9400 #include "types.h"
9401 #include "user.h"
9402
9403 // Test GID and UID to be in the correct range
9404 #ifdef CS333_P2
9405 int
9406 testgiduid(void)
9407 {
9408    uint uid, gid, ppid;
9409
9410    uid = getuid();
9411    printf(2, "Current UID is : %d\n", uid);
9412    printf(2, "Setting UID to 100\n");
9413    setuid(100);
9414    uid = getuid();
9415    printf(2, "Current UID is : %d\n", uid);
9416
9417    gid = getgid();
9418    printf(2, "Current GID is : %d\n", gid);
9419    printf(2, "Setting GID to 100\n");
9420    setgid(100);
9421    gid = getgid();
9422    printf(2, "Current UID is : %d\n", gid);
9423
9424    ppid = getppid();
9425    printf(2, "My parent process is : %d\n", ppid);
9426    printf(2, "Done!\n");
9427
9428    return 0;
9429 }
9430
9431 int
9432 main(int argc, char *argv[])
9433 {
9434    testgiduid();
9435    exit();
9436 }
9437 #else
9438 int
9439 main(int argc, char *argv[])
9440 {
9441    printf(2, "Please compile with CS333_P2 on to enable this feature.\n");
9442    exit();
9443 }
9444 #endif
9445
9446
9447
9448
9449
```



```
9450 #include "types.h"
9451 #include "uproc.h"
9452 #include "user.h"
9453
9454 #ifdef CS333_P2
9455 int
9456 main(int argc, char *argv[])
9457 {
9458    int ptable_size;
9459    uint display_size;
9460    display_size = 64;
9461    struct uproc* ps;
9462    ps = malloc(sizeof(struct uproc) * display_size);
9463    ptable_size = getprocs(display_size, ps);
9464    if(ptable_size <= 0) {
9465            printf(1,"\nGetting processes information failed\n");
9466            exit();
9467    }
9468    printf(1,"\nNumber of processes is :%d\n",ptable_size);
9469 #ifdef CS333_P3
9470    printf(1,"\nPID       State     Name      UID       GID       PPID    Pri
9471    int i;
9472    for(i=0; i < ptable_size; ++i){
9473     printf(1,"\n%d         %s    %s    %d    %d    %d    %d    %d.%d    %d.%
9474            ps->state,\
9475            ps->name,\
9476            ps->uid,\
9477            ps->gid,\
9478            ps->ppid,\
9479            ps->priority,\
9480            ps->elapsed_ticks/100, ps->elapsed_ticks%100, ps->CPU_total_ticks,
9481            ++ps;
9482    }
9483 #else
9484    printf(1,"\nPID       State     Name      UID       GID       PPID      E
9485    int i;
9486    for(i=0; i < ptable_size; ++i){
9487     printf(1,"\n%d         %s    %s    %d    %d    %d    %d.%d    %d.%d    %
9488            ps->state,\
9489            ps->name,\
9490            ps->uid,\
9491            ps->gid,\
9492            ps->ppid,\
9493            ps->elapsed_ticks/100, ps->elapsed_ticks%100, ps->CPU_total_ticks,
9494            ++ps;
9495    }
9496 #endif
9497    free(ps);
9498   exit();
9499 }
```

```
9500 #else
9501 int
9502 main(int argc, char *argv[])
9503 {
9504    printf(2, "Please compile with CS333_P2 on to enable this feature.\n");
9505    exit();
9506 }
9507 #endif
9508
9509
9510
9511
9512
9513
9514
9515
9516
9517
9518
9519
9520
9521
9522
9523
9524
9525
9526
9527
9528
9529
9530
9531
9532
9533
9534
9535
9536
9537
9538
9539
9540
9541
9542
9543
9544
9545
9546
9547
9548
9549
```



```
9550 #include "types.h"
9551 #include "user.h"
9552
9553 #ifdef CS333_P2
9554 int
9555 main(int argc, char *argv[])
9556 {
9557    int elapsed_t = 0;
9558    int pid;
9559    int start_t = 0;
9560    int end_t = start_t;
9561    if(argc > 1) {
9562            start_t = uptime();
9563            pid = fork();
9564            if(pid > 0) {
9565                    pid = wait();
9566                    end_t= uptime();
9567                    }
9568            else if(pid == 0) {
9569                    //child process running
9570                    if(exec(argv[1], argv+1) < 0)
9571                            printf(2,"%s failed to execute.", argv[1]);
9572                    exit();
9573                    }
9574            else {
9575                    // error: fork failed
9576                    printf(2,"Error: Fork failed");
9577                    exit();
9578                    }
9579            }
9580    elapsed_t = end_t - start_t;
9581    char *proc_name = argv[1] ? argv[1] : "";
9582   printf(1,"%s ran in %d.%d seconds\n",proc_name, elapsed_t/100, elapsed_t%1(
9583
9584   exit();
9585 }
9586 #else
9587 int
9588 main(int argc, char *argv[])
9589 {
9590    printf(2, "Please compile with CS333_P2 on to enable this feature.\n");
9591    exit();
9592 }
9593 #endif
9594
9595
9596
9597
9598
9599
```

```
9600 // This program can be freely used to test your scheduler. It is
9601 // by no means a complete test.
9602
9603 #include "types.h"
9604 #include "user.h"
9605
9606 // PrioCount should be set to the nummber of priority levels
9607 #define PrioCount 3
9608 #define numChildren 10
9609
9610 void
9611 countForever(int p)
9612 {
9613   int j;
9614   unsigned long count = 0;
9615
9616   j = getpid();
9617   p = p%PrioCount;
9618   setpriority(j, p);
9619   printf(1, "%d: start prio %d\n", j, p);
9620
9621   while (1) {
9622     count++;
9623     if ((count & 0xFFFFFFF) == 0) {
9624       p = (p+1) % PrioCount;
9625       setpriority(j, p);
9626       printf(1, "%d: new prio %d\n", j, p);
9627     }
9628   }
9629 }
9630
9631 int
9632 main(void)
9633 {
9634   int i, rc;
9635
9636   for (i=0; i<numChildren; i++) {
9637     rc = fork();
9638     if (!rc) { // child
9639       countForever(i);
9640     }
9641   }
9642   // what the heck, let's have the parent waste time as well!
9643   countForever(1);
9644   exit();
9645 }
9646
9647
9648
9649
```