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 buf;
0251 struct context;
0252 struct file;
0253 struct inode;
0254 struct pipe;
0255 struct proc;
0256 struct rtcdate;
0257 struct spinlock;
0258 struct stat;
0259 struct superblock;
0260
0261 // bio.c
0262 void            binit(void);
0263 struct buf*     bread(uint, uint);
0264 void            brelse(struct buf*);
0265 void            bwrite(struct buf*);
0266
0267 // console.c
0268 void            consoleinit(void);
0269 void            cprintf(char*, ...);
0270 void            consoleintr(int(*)(void));
0271 void            panic(char*) __attribute__((noreturn));
0272
0273 // exec.c
0274 int             exec(char*, char**);
0275
0276 // file.c
0277 struct file*    filealloc(void);
0278 void            fileclose(struct file*);
0279 struct file*    filedup(struct file*);
0280 void            fileinit(void);
0281 int             fileread(struct file*, char*, int n);
0282 int             filestat(struct file*, struct stat*);
0283 int             filewrite(struct file*, char*, int n);
0284
0285 // fs.c
0286 void            readsb(int dev, struct superblock *sb);
0287 int             dirlink(struct inode*, char*, uint);
0288 struct inode*   dirlookup(struct inode*, char*, uint*);
0289 struct inode*   ialloc(uint, short);
0290 struct inode*   idup(struct inode*);
0291 void            iinit(int dev);
0292 void            ilock(struct inode*);
0293 void            iput(struct inode*);
0294 void            iunlock(struct inode*);
0295 void            iunlockput(struct inode*);
0296 void            iupdate(struct inode*);
0297 int             namecmp(const char*, const char*);
0298 struct inode*   namei(char*);
0299 struct inode*   nameiparent(char*, char*);
```

```
0300 int             readi(struct inode*, char*, uint, uint);
0301 void            stati(struct inode*, struct stat*);
0302 int             writei(struct inode*, char*, uint, uint);
0303
0304 // ide.c
0305 void            ideinit(void);
0306 void            ideintr(void);
0307 void            iderw(struct buf*);
0308
0309 // ioapic.c
0310 void            ioapicenable(int irq, int cpu);
0311 extern uchar    ioapicid;
0312 void            ioapicinit(void);
0313
0314 // kalloc.c
0315 char*           kalloc(void);
0316 void            kfree(char*);
0317 void            kinit1(void*, void*);
0318 void            kinit2(void*, void*);
0319
0320 // kbd.c
0321 void            kbdintr(void);
0322
0323 // lapic.c
0324 void            cmostime(struct rtcdate *r);
0325 int             cpunum(void);
0326 extern volatile uint*    lapic;
0327 void            lapiceoi(void);
0328 void            lapicinit(void);
0329 void            lapicstartap(uchar, uint);
0330 void            microdelay(int);
0331
0332 // log.c
0333 void            initlog(int dev);
0334 void            log_write(struct buf*);
0335 void            begin_op();
0336 void            end_op();
0337
0338 // mp.c
0339 extern int      ismp;
0340 int             mpbcpu(void);
0341 void            mpinit(void);
0342 void            mpstartthem(void);
0343
0344 // picirq.c
0345 void            picenable(int);
0346 void            picinit(void);
0347
0348
0349
```



```
0350 // pipe.c
0351 int             pipealloc(struct file**, struct file**);
0352 void            pipeclose(struct pipe*, int);
0353 int             piperead(struct pipe*, char*, int);
0354 int             pipewrite(struct pipe*, char*, int);
0355
0356
0357 // proc.c
0358 struct proc*    copyproc(struct proc*);
0359 void            exit(void);
0360 int             fork(void);
0361 int             growproc(int);
0362 int             kill(int);
0363 void            pinit(void);
0364 void            procdump(void);
0365 void            scheduler(void) __attribute__((noreturn));
0366 void            sched(void);
0367 void            sleep(void*, struct spinlock*);
0368 void            userinit(void);
0369 int             wait(void);
0370 void            wakeup(void*);
0371 void            yield(void);
0372
0373 // swtch.S
0374 void            swtch(struct context**, struct context*);
0375
0376 // spinlock.c
0377 void            acquire(struct spinlock*);
0378 void            getcallerpcs(void*, uint*);
0379 int             holding(struct spinlock*);
0380 void            initlock(struct spinlock*, char*);
0381 void            release(struct spinlock*);
0382 void            pushcli(void);
0383 void            popcli(void);
0384
0385 // string.c
0386 int             memcmp(const void*, const void*, uint);
0387 void*           memmove(void*, const void*, uint);
0388 void*           memset(void*, int, uint);
0389 char*           safestrcpy(char*, const char*, int);
0390 int             strlen(const char*);
0391 int             strncmp(const char*, const char*, uint);
0392 char*           strncpy(char*, const char*, int);
0393
0394 // syscall.c
0395 int             argint(int, int*);
0396 int             argptr(int, char**, int);
0397 int             argstr(int, char**);
0398 int             fetchint(uint, int*);
0399 int             fetchstr(uint, char**);
```

```
0400 void            syscall(void);
0401
0402 // timer.c
0403 void            timerinit(void);
0404
0405 // trap.c
0406 void            idtinit(void);
0407 extern uint     ticks;
0408 void            tvinit(void);
0409 extern struct spinlock tickslock;
0410
0411 // uart.c
0412 void            uartinit(void);
0413 void            uartintr(void);
0414 void            uartputc(int);
0415
0416 // vm.c
0417 void            seginit(void);
0418 void            kvmalloc(void);
0419 void            vmenable(void);
0420 pde_t*          setupkvm(void);
0421 char*           uva2ka(pde_t*, char*);
0422 int             allocuvm(pde_t*, uint, uint);
0423 int             deallocuvm(pde_t*, uint, uint);
0424 void            freevm(pde_t*);
0425 void            inituvm(pde_t*, char*, uint);
0426 int             loaduvm(pde_t*, char*, struct inode*, uint, uint);
0427 pde_t*          copyuvm(pde_t*, uint);
0428 void            switchuvm(struct proc*);
0429 void            switchkvm(void);
0430 int             copyout(pde_t*, uint, void*, uint);
0431 void            clearpteu(pde_t *pgdir, char *uva);
0432
0433 // number of elements in fixed-size array
0434 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
0435
0436
0437
0438
0439
0440
0441
0442
0443
0444
0445
0446
0447
0448
0449
```



```
0450 // Routines to let C code use special x86 instructions.
0451
0452 static inline uchar
0453 inb(ushort port)
0454 {
0455   uchar data;
0456
0457   asm volatile("in %1,%0" : "=a" (data) : "d" (port));
0458   return data;
0459 }
0460
0461 static inline void
0462 insl(int port, void *addr, int cnt)
0463 {
0464   asm volatile("cld; rep insl" :
0465                "=D" (addr), "=c" (cnt) :
0466                "d" (port), "0" (addr), "1" (cnt) :
0467                "memory", "cc");
0468 }
0469
0470 static inline void
0471 outb(ushort port, uchar data)
0472 {
0473   asm volatile("out %0,%1" : : "a" (data), "d" (port));
0474 }
0475
0476 static inline void
0477 outw(ushort port, ushort data)
0478 {
0479   asm volatile("out %0,%1" : : "a" (data), "d" (port));
0480 }
0481
0482 static inline void
0483 outsl(int port, const void *addr, int cnt)
0484 {
0485   asm volatile("cld; rep outsl" :
0486                "=S" (addr), "=c" (cnt) :
0487                "d" (port), "0" (addr), "1" (cnt) :
0488                "cc");
0489 }
0490
0491 static inline void
0492 stosb(void *addr, int data, int cnt)
0493 {
0494   asm volatile("cld; rep stosb" :
0495                "=D" (addr), "=c" (cnt) :
0496                "0" (addr), "1" (cnt), "a" (data) :
0497                "memory", "cc");
0498 }
0499
```

```
0500 static inline void
0501 stosl(void *addr, int data, int cnt)
0502 {
0503   asm volatile("cld; rep stosl" :
0504                "=D" (addr), "=c" (cnt) :
0505                "0" (addr), "1" (cnt), "a" (data) :
0506                "memory", "cc");
0507 }
0508
0509 struct segdesc;
0510
0511 static inline void
0512 lgdt(struct segdesc *p, int size)
0513 {
0514   volatile ushort pd[3];
0515
0516   pd[0] = size-1;
0517   pd[1] = (uint)p;
0518   pd[2] = (uint)p >> 16;
0519
0520   asm volatile("lgdt (%0)" : : "r" (pd));
0521 }
0522
0523 struct gatedesc;
0524
0525 static inline void
0526 lidt(struct gatedesc *p, int size)
0527 {
0528   volatile ushort pd[3];
0529
0530   pd[0] = size-1;
0531   pd[1] = (uint)p;
0532   pd[2] = (uint)p >> 16;
0533
0534   asm volatile("lidt (%0)" : : "r" (pd));
0535 }
0536
0537 static inline void
0538 ltr(ushort sel)
0539 {
0540   asm volatile("ltr %0" : : "r" (sel));
0541 }
0542
0543 static inline uint
0544 readeflags(void)
0545 {
0546   uint eflags;
0547   asm volatile("pushfl; popl %0" : "=r" (eflags));
0548   return eflags;
0549 }
```



```
0550 static inline void
0551 loadgs(ushort v)
0552 {
0553   asm volatile("movw %0, %%gs" : : "r" (v));
0554 }
0555
0556 static inline void
0557 cli(void)
0558 {
0559   asm volatile("cli");
0560 }
0561
0562 static inline void
0563 sti(void)
0564 {
0565   asm volatile("sti");
0566 }
0567
0568 static inline uint
0569 xchg(volatile uint *addr, uint newval)
0570 {
0571   uint result;
0572
0573   // The + in "+m" denotes a read-modify-write operand.
0574   asm volatile("lock; xchgl %0, %1" :
0575                "+m" (*addr), "=a" (result) :
0576                "1" (newval) :
0577                "cc");
0578   return result;
0579 }
0580
0581 static inline uint
0582 rcr2(void)
0583 {
0584   uint val;
0585   asm volatile("movl %%cr2,%0" : "=r" (val));
0586   return val;
0587 }
0588
0589 static inline void
0590 lcr3(uint val)
0591 {
0592   asm volatile("movl %0,%%cr3" : : "r" (val));
0593 }
0594
0595
0596
0597
0598
0599
```

```
0600 // Layout of the trap frame built on the stack by the
0601 // hardware and by trapasm.S, and passed to trap().
0602 struct trapframe {
0603   // registers as pushed by pusha
0604   uint edi;
0605   uint esi;
0606   uint ebp;
0607   uint oesp;      // useless & ignored
0608   uint ebx;
0609   uint edx;
0610   uint ecx;
0611   uint eax;
0612
0613   // rest of trap frame
0614   ushort gs;
0615   ushort padding1;
0616   ushort fs;
0617   ushort padding2;
0618   ushort es;
0619   ushort padding3;
0620   ushort ds;
0621   ushort padding4;
0622   uint trapno;
0623
0624   // below here defined by x86 hardware
0625   uint err;
0626   uint eip;
0627   ushort cs;
0628   ushort padding5;
0629   uint eflags;
0630
0631   // below here only when crossing rings, such as from user to kernel
0632   uint esp;
0633   ushort ss;
0634   ushort padding6;
0635 };
0636
0637
0638
0639
0640
0641
0642
0643
0644
0645
0646
0647
0648
0649
```



```
0650 //
0651 // assembler macros to create x86 segments
0652 //
0653
0654 #define SEG_NULLASM                                             \
0655         .word 0, 0;                                             \
0656         .byte 0, 0, 0, 0
0657
0658 // The 0xC0 means the limit is in 4096-byte units
0659 // and (for executable segments) 32-bit mode.
0660 #define SEG_ASM(type,base,lim)                                  \
0661         .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
0662         .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
0663                 (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
0664
0665 #define STA_X     0x8       // Executable segment
0666 #define STA_E     0x4       // Expand down (non-executable segments)
0667 #define STA_C     0x4       // Conforming code segment (executable only)
0668 #define STA_W     0x2       // Writeable (non-executable segments)
0669 #define STA_R     0x2       // Readable (executable segments)
0670 #define STA_A     0x1       // Accessed
0671
0672
0673
0674
0675
0676
0677
0678
0679
0680
0681
0682
0683
0684
0685
0686
0687
0688
0689
0690
0691
0692
0693
0694
0695
0696
0697
0698
0699
```

```
0700 // This file contains definitions for the
0701 // x86 memory management unit (MMU).
0702
0703 // Eflags register
0704 #define FL_CF           0x00000001      // Carry Flag
0705 #define FL_PF           0x00000004      // Parity Flag
0706 #define FL_AF           0x00000010      // Auxiliary carry Flag
0707 #define FL_ZF           0x00000040      // Zero Flag
0708 #define FL_SF           0x00000080      // Sign Flag
0709 #define FL_TF           0x00000100      // Trap Flag
0710 #define FL_IF           0x00000200      // Interrupt Enable
0711 #define FL_DF           0x00000400      // Direction Flag
0712 #define FL_OF           0x00000800      // Overflow Flag
0713 #define FL_IOPL_MASK    0x00003000      // I/O Privilege Level bitmask
0714 #define FL_IOPL_0       0x00000000      //   IOPL == 0
0715 #define FL_IOPL_1       0x00001000      //   IOPL == 1
0716 #define FL_IOPL_2       0x00002000      //   IOPL == 2
0717 #define FL_IOPL_3       0x00003000      //   IOPL == 3
0718 #define FL_NT           0x00004000      // Nested Task
0719 #define FL_RF           0x00010000      // Resume Flag
0720 #define FL_VM           0x00020000      // Virtual 8086 mode
0721 #define FL_AC           0x00040000      // Alignment Check
0722 #define FL_VIF          0x00080000      // Virtual Interrupt Flag
0723 #define FL_VIP          0x00100000      // Virtual Interrupt Pending
0724 #define FL_ID           0x00200000      // ID flag
0725
0726 // Control Register flags
0727 #define CR0_PE          0x00000001      // Protection Enable
0728 #define CR0_MP          0x00000002      // Monitor coProcessor
0729 #define CR0_EM          0x00000004      // Emulation
0730 #define CR0_TS          0x00000008      // Task Switched
0731 #define CR0_ET          0x00000010      // Extension Type
0732 #define CR0_NE          0x00000020      // Numeric Errror
0733 #define CR0_WP          0x00010000      // Write Protect
0734 #define CR0_AM          0x00040000      // Alignment Mask
0735 #define CR0_NW          0x20000000      // Not Writethrough
0736 #define CR0_CD          0x40000000      // Cache Disable
0737 #define CR0_PG          0x80000000      // Paging
0738
0739 #define CR4_PSE         0x00000010      // Page size extension
0740
0741 #define SEG_KCODE 1  // kernel code
0742 #define SEG_KDATA 2  // kernel data+stack
0743 #define SEG_KCPU  3  // kernel per-cpu data
0744 #define SEG_UCODE 4  // user code
0745 #define SEG_UDATA 5  // user data+stack
0746 #define SEG_TSS   6  // this process's task state
0747
0748
0749
```



```
0750 #ifndef __ASSEMBLER__
0751 // Segment Descriptor
0752 struct segdesc {
0753   uint lim_15_0 : 16;  // Low bits of segment limit
0754   uint base_15_0 : 16; // Low bits of segment base address
0755   uint base_23_16 : 8; // Middle bits of segment base address
0756   uint type : 4;       // Segment type (see STS_ constants)
0757   uint s : 1;          // 0 = system, 1 = application
0758   uint dpl : 2;        // Descriptor Privilege Level
0759   uint p : 1;          // Present
0760   uint lim_19_16 : 4;  // High bits of segment limit
0761   uint avl : 1;        // Unused (available for software use)
0762   uint rsv1 : 1;       // Reserved
0763   uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
0764   uint g : 1;          // Granularity: limit scaled by 4K when set
0765   uint base_31_24 : 8; // High bits of segment base address
0766 };
0767
0768 // Normal segment
0769 #define SEG(type, base, lim, dpl) (struct segdesc)    \
0770 { ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff,      \
0771   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
0772   (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
0773 #define SEG16(type, base, lim, dpl) (struct segdesc)  \
0774 { (lim) & 0xffff, (uint)(base) & 0xffff,              \
0775   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
0776   (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
0777 #endif
0778
0779 #define DPL_USER    0x3     // User DPL
0780
0781 // Application segment type bits
0782 #define STA_X       0x8     // Executable segment
0783 #define STA_E       0x4     // Expand down (non-executable segments)
0784 #define STA_C       0x4     // Conforming code segment (executable only)
0785 #define STA_W       0x2     // Writeable (non-executable segments)
0786 #define STA_R       0x2     // Readable (executable segments)
0787 #define STA_A       0x1     // Accessed
0788
0789 // System segment type bits
0790 #define STS_T16A    0x1     // Available 16-bit TSS
0791 #define STS_LDT     0x2     // Local Descriptor Table
0792 #define STS_T16B    0x3     // Busy 16-bit TSS
0793 #define STS_CG16    0x4     // 16-bit Call Gate
0794 #define STS_TG      0x5     // Task Gate / Coum Transmitions
0795 #define STS_IG16    0x6     // 16-bit Interrupt Gate
0796 #define STS_TG16    0x7     // 16-bit Trap Gate
0797 #define STS_T32A    0x9     // Available 32-bit TSS
0798 #define STS_T32B    0xB     // Busy 32-bit TSS
0799 #define STS_CG32    0xC     // 32-bit Call Gate
```

```
0800 #define STS_IG32    0xE     // 32-bit Interrupt Gate
0801 #define STS_TG32    0xF     // 32-bit Trap Gate
0802
0803 // A virtual address 'la' has a three-part structure as follows:
0804 //
0805 // +--------10------+-------10-------+---------12----------+
0806 // | Page Directory |   Page Table   | Offset within Page  |
0807 // |      Index     |      Index     |                     |
0808 // +----------------+----------------+---------------------+
0809 //  \--- PDX(va) --/ \--- PTX(va) --/
0810
0811 // page directory index
0812 #define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)
0813
0814 // page table index
0815 #define PTX(va)         (((uint)(va) >> PTXSHIFT) & 0x3FF)
0816
0817 // construct virtual address from indexes and offset
0818 #define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
0819
0820 // Page directory and page table constants.
0821 #define NPDENTRIES      1024    // # directory entries per page directory
0822 #define NPTENTRIES      1024    // # PTEs per page table
0823 #define PGSIZE          4096    // bytes mapped by a page
0824
0825 #define PGSHIFT         12      // log2(PGSIZE)
0826 #define PTXSHIFT        12      // offset of PTX in a linear address
0827 #define PDXSHIFT        22      // offset of PDX in a linear address
0828
0829 #define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
0830 #define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
0831
0832 // Page table/directory entry flags.
0833 #define PTE_P           0x001   // Present
0834 #define PTE_W           0x002   // Writeable
0835 #define PTE_U           0x004   // User
0836 #define PTE_PWT         0x008   // Write-Through
0837 #define PTE_PCD         0x010   // Cache-Disable
0838 #define PTE_A           0x020   // Accessed
0839 #define PTE_D           0x040   // Dirty
0840 #define PTE_PS          0x080   // Page Size
0841 #define PTE_MBZ         0x180   // Bits must be zero
0842
0843 // Address in page table or page directory entry
0844 #define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
0845 #define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF)
0846
0847 #ifndef __ASSEMBLER__
0848 typedef uint pte_t;
0849
```



```
0850 // Task state segment format
0851 struct taskstate {
0852   uint link;         // Old ts selector
0853   uint esp0;         // Stack pointers and segment selectors
0854   ushort ss0;        //   after an increase in privilege level
0855   ushort padding1;
0856   uint *esp1;
0857   ushort ss1;
0858   ushort padding2;
0859   uint *esp2;
0860   ushort ss2;
0861   ushort padding3;
0862   void *cr3;         // Page directory base
0863   uint *eip;         // Saved state from last task switch
0864   uint eflags;
0865   uint eax;          // More saved state (registers)
0866   uint ecx;
0867   uint edx;
0868   uint ebx;
0869   uint *esp;
0870   uint *ebp;
0871   uint esi;
0872   uint edi;
0873   ushort es;         // Even more saved state (segment selectors)
0874   ushort padding4;
0875   ushort cs;
0876   ushort padding5;
0877   ushort ss;
0878   ushort padding6;
0879   ushort ds;
0880   ushort padding7;
0881   ushort fs;
0882   ushort padding8;
0883   ushort gs;
0884   ushort padding9;
0885   ushort ldt;
0886   ushort padding10;
0887   ushort t;          // Trap on task switch
0888   ushort iomb;       // I/O map base address
0889 };
0890
0891
0892
0893
0894
0895
0896
0897
0898
0899
```

```
0900 // Gate descriptors for interrupts and traps
0901 struct gatedesc {
0902   uint off_15_0 : 16;   // low 16 bits of offset in segment
0903   uint cs : 16;         // code segment selector
0904   uint args : 5;        // # args, 0 for interrupt/trap gates
0905   uint rsv1 : 3;        // reserved(should be zero I guess)
0906   uint type : 4;        // type(STS_{TG,IG32,TG32})
0907   uint s : 1;           // must be 0 (system)
0908   uint dpl : 2;         // descriptor(meaning new) privilege level
0909   uint p : 1;           // Present
0910   uint off_31_16 : 16;  // high bits of offset in segment
0911 };
0912
0913 // Set up a normal interrupt/trap gate descriptor.
0914 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
0915 //   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
0916 // - sel: Code segment selector for interrupt/trap handler
0917 // - off: Offset in code segment for interrupt/trap handler
0918 // - dpl: Descriptor Privilege Level -
0919 //        the privilege level required for software to invoke
0920 //        this interrupt/trap gate explicitly using an int instruction.
0921 #define SETGATE(gate, istrap, sel, off, d)                \
0922 {                                                         \
0923   (gate).off_15_0 = (uint)(off) & 0xffff;                \
0924   (gate).cs = (sel);                                      \
0925   (gate).args = 0;                                        \
0926   (gate).rsv1 = 0;                                        \
0927   (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
0928   (gate).s = 0;                                           \
0929   (gate).dpl = (d);                                       \
0930   (gate).p = 1;                                           \
0931   (gate).off_31_16 = (uint)(off) >> 16;                  \
0932 }
0933
0934 #endif
0935
0936
0937
0938
0939
0940
0941
0942
0943
0944
0945
0946
0947
0948
0949
```



```
0950 // Format of an ELF executable file
0951
0952 #define ELF_MAGIC 0x464C457FU  // "\x7FELF" in little endian
0953
0954 // File header
0955 struct elfhdr {
0956   uint magic;  // must equal ELF_MAGIC
0957   uchar elf[12];
0958   ushort type;
0959   ushort machine;
0960   uint version;
0961   uint entry;
0962   uint phoff;
0963   uint shoff;
0964   uint flags;
0965   ushort ehsize;
0966   ushort phentsize;
0967   ushort phnum;
0968   ushort shentsize;
0969   ushort shnum;
0970   ushort shstrndx;
0971 };
0972
0973 // Program section header
0974 struct proghdr {
0975   uint type;
0976   uint off;
0977   uint vaddr;
0978   uint paddr;
0979   uint filesz;
0980   uint memsz;
0981   uint flags;
0982   uint align;
0983 };
0984
0985 // Values for Proghdr type
0986 #define ELF_PROG_LOAD           1
0987
0988 // Flag bits for Proghdr flags
0989 #define ELF_PROG_FLAG_EXEC      1
0990 #define ELF_PROG_FLAG_WRITE     2
0991 #define ELF_PROG_FLAG_READ      4
0992
0993
0994
0995
0996
0997
0998
0999
```

```
1000 # Multiboot header, for multiboot boot loaders like GNU Grub.
1001 # http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
1002 #
1003 # Using GRUB 2, you can boot xv6 from a file stored in a
1004 # Linux file system by copying kernel or kernelmemfs to /boot
1005 # and then adding this menu entry:
1006 #
1007 # menuentry "xv6" {
1008 #  insmod ext2
1009 #  set root='(hd0,msdos1)'
1010 #  set kernel='/boot/kernel'
1011 #  echo "Loading ${kernel}..."
1012 #  multiboot ${kernel} ${kernel}
1013 #  boot
1014 # }
1015
1016 #include "asm.h"
1017 #include "memlayout.h"
1018 #include "mmu.h"
1019 #include "param.h"
1020
1021 # Multiboot header.  Data to direct multiboot loader.
1022 .p2align 2
1023 .text
1024 .globl multiboot_header
1025 multiboot_header:
1026   #define magic 0x1badb002
1027   #define flags 0
1028   .long magic
1029   .long flags
1030   .long (-magic-flags)
1031
1032 # By convention, the _start symbol specifies the ELF entry point.
1033 # Since we haven't set up virtual memory yet, our entry point is
1034 # the physical address of 'entry'.
1035 .globl _start
1036 _start = V2P_WO(entry)
1037
1038 # Entering xv6 on boot processor, with paging off.
1039 .globl entry
1040 entry:
1041   # Turn on page size extension for 4Mbyte pages
1042   movl    %cr4, %eax
1043   orl     $(CR4_PSE), %eax
1044   movl    %eax, %cr4
1045   # Set page directory
1046   movl    $(V2P_WO(entrypgdir)), %eax
1047   movl    %eax, %cr3
1048   # Turn on paging.
1049   movl    %cr0, %eax
```



```
1050   orl     $(CR0_PG|CR0_WP), %eax
1051   movl    %eax, %cr0
1052
1053   # Set up the stack pointer.
1054   movl $(stack + KSTACKSIZE), %esp
1055
1056   # Jump to main(), and switch to executing at
1057   # high addresses. The indirect call is needed because
1058   # the assembler produces a PC-relative instruction
1059   # for a direct jump.
1060   mov $main, %eax
1061   jmp *%eax
1062
1063 .comm stack, KSTACKSIZE
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
```

```
1100 #include "asm.h"
1101 #include "memlayout.h"
1102 #include "mmu.h"
1103
1104 # Each non-boot CPU ("AP") is started up in response to a STARTUP
1105 # IPI from the boot CPU.  Section B.4.2 of the Multi-Processor
1106 # Specification says that the AP will start in real mode with CS:IP
1107 # set to XY00:0000, where XY is an 8-bit value sent with the
1108 # STARTUP. Thus this code must start at a 4096-byte boundary.
1109 #
1110 # Because this code sets DS to zero, it must sit
1111 # at an address in the low 2^16 bytes.
1112 #
1113 # Startothers (in main.c) sends the STARTUPs one at a time.
1114 # It copies this code (start) at 0x7000.  It puts the address of
1115 # a newly allocated per-core stack in start-4,the address of the
1116 # place to jump to (mpenter) in start-8, and the physical address
1117 # of entrypgdir in start-12.
1118 #
1119 # This code is identical to bootasm.S except:
1120 #   - it does not need to enable A20
1121 #   - it uses the address at start-4, start-8, and start-12
1122
1123 .code16
1124 .globl start
1125 start:
1126   cli
1127
1128   xorw    %ax,%ax
1129   movw    %ax,%ds
1130   movw    %ax,%es
1131   movw    %ax,%ss
1132
1133   lgdt    gdtdesc
1134   movl    %cr0, %eax
1135   orl     $CR0_PE, %eax
1136   movl    %eax, %cr0
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
```



```
1150   ljmpl   $(SEG_KCODE<<3), $(start32)
1151
1152 .code32
1153 start32:
1154   movw    $(SEG_KDATA<<3), %ax
1155   movw    %ax, %ds
1156   movw    %ax, %es
1157   movw    %ax, %ss
1158   movw    $0, %ax
1159   movw    %ax, %fs
1160   movw    %ax, %gs
1161
1162   # Turn on page size extension for 4Mbyte pages
1163   movl    %cr4, %eax
1164   orl     $(CR4_PSE), %eax
1165   movl    %eax, %cr4
1166   # Use enterpgdir as our initial page table
1167   movl    (start-12), %eax
1168   movl    %eax, %cr3
1169   # Turn on paging.
1170   movl    %cr0, %eax
1171   orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
1172   movl    %eax, %cr0
1173
1174   # Switch to the stack allocated by startothers()
1175   movl    (start-4), %esp
1176   # Call mpenter()
1177   call      *(start-8)
1178
1179   movw    $0x8a00, %ax
1180   movw    %ax, %dx
1181   outw    %ax, %dx
1182   movw    $0x8ae0, %ax
1183   outw    %ax, %dx
1184 spin:
1185   jmp     spin
1186
1187 .p2align 2
1188 gdt:
1189   SEG_NULLASM
1190   SEG_ASM(STA_X|STA_R, 0, 0xffffffff)
1191   SEG_ASM(STA_W, 0, 0xffffffff)
1192
1193
1194 gdtdesc:
1195   .word   (gdtdesc - gdt - 1)
1196   .long   gdt
1197
1198
1199
```

```
1200 #include "types.h"
1201 #include "defs.h"
1202 #include "param.h"
1203 #include "memlayout.h"
1204 #include "mmu.h"
1205 #include "proc.h"
1206 #include "x86.h"
1207
1208 static void startothers(void);
1209 static void mpmain(void)  __attribute__((noreturn));
1210 extern pde_t *kpgdir;
1211 extern char end[]; // first address after kernel loaded from ELF file
1212
1213 // Bootstrap processor starts running C code here.
1214 // Allocate a real stack and switch to it, first
1215 // doing some setup required for memory allocator to work.
1216 int
1217 main(void)
1218 {
1219   kinit1(end, P2V(4*1024*1024)); // phys page allocator
1220   kvmalloc();      // kernel page table
1221   mpinit();        // collect info about this machine
1222   lapicinit();
1223   seginit();       // set up segments
1224   cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
1225   picinit();       // interrupt controller
1226   ioapicinit();    // another interrupt controller
1227   consoleinit();   // I/O devices & their interrupts
1228   uartinit();      // serial port
1229   pinit();         // process table
1230   tvinit();        // trap vectors
1231   binit();         // buffer cache
1232   fileinit();      // file table
1233   ideinit();       // disk
1234   if(!ismp)
1235     timerinit();   // uniprocessor timer
1236   startothers();   // start other processors
1237   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1238   userinit();      // first user process
1239   // Finish setting up this processor in mpmain.
1240   mpmain();
1241 }
1242
1243
1244
1245
1246
1247
1248
1249
```



```
1250 // Other CPUs jump here from entryother.S.
1251 static void
1252 mpenter(void)
1253 {
1254   switchkvm();
1255   seginit();
1256   lapicinit();
1257   mpmain();
1258 }
1259
1260 // Common CPU setup code.
1261 static void
1262 mpmain(void)
1263 {
1264   cprintf("cpu%d: starting\n", cpu->id);
1265   idtinit();       // load idt register
1266   xchg(&cpu->started, 1); // tell startothers() we're up
1267   scheduler();     // start running processes
1268 }
1269
1270 pde_t entrypgdir[];  // For entry.S
1271
1272 // Start the non-boot (AP) processors.
1273 static void
1274 startothers(void)
1275 {
1276   extern uchar _binary_entryother_start[], _binary_entryother_size[];
1277   uchar *code;
1278   struct cpu *c;
1279   char *stack;
1280
1281   // Write entry code to unused memory at 0x7000.
1282   // The linker has placed the image of entryother.S in
1283   // _binary_entryother_start.
1284   code = p2v(0x7000);
1285   memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
1286
1287   for(c = cpus; c < cpus+ncpu; c++){
1288     if(c == cpus+cpunum())  // We've started already.
1289       continue;
1290
1291     // Tell entryother.S what stack to use, where to enter, and what
1292     // pgdir to use. We cannot use kpgdir yet, because the AP processor
1293     // is running in low  memory, so we use entrypgdir for the APs too.
1294     stack = kalloc();
1295     *(void**)(code-4) = stack + KSTACKSIZE;
1296     *(void**)(code-8) = mpenter;
1297     *(int**)(code-12) = (void *) v2p(entrypgdir);
1298
1299     lapicstartap(c->id, v2p(code));
```

```
1300     // wait for cpu to finish mpmain()
1301     while(c->started == 0)
1302       ;
1303   }
1304 }
1305
1306 // Boot page table used in entry.S and entryother.S.
1307 // Page directories (and page tables), must start on a page boundary,
1308 // hence the "__aligned__" attribute.
1309 // Use PTE_PS in page directory entry to enable 4Mbyte pages.
1310 __attribute__((__aligned__(PGSIZE)))
1311 pde_t entrypgdir[NPDENTRIES] = {
1312   // Map VA's [0, 4MB) to PA's [0, 4MB)
1313   [0] = (0) | PTE_P | PTE_W | PTE_PS,
1314   // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1315   [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1316 };
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
```



```
1350 // Blank page.
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
```

```
1400 // Blank page.
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
```



```
1450 // Blank page.
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
```

```
1500 // Mutual exclusion lock.
1501 struct spinlock {
1502   uint locked;       // Is the lock held?
1503
1504   // For debugging:
1505   char *name;        // Name of lock.
1506   struct cpu *cpu;   // The cpu holding the lock.
1507   uint pcs[10];      // The call stack (an array of program counters)
1508                      // that locked the lock.
1509 };
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
```



```
1550 // Mutual exclusion spin locks.
1551
1552 #include "types.h"
1553 #include "defs.h"
1554 #include "param.h"
1555 #include "x86.h"
1556 #include "memlayout.h"
1557 #include "mmu.h"
1558 #include "proc.h"
1559 #include "spinlock.h"
1560
1561 void
1562 initlock(struct spinlock *lk, char *name)
1563 {
1564   lk->name = name;
1565   lk->locked = 0;
1566   lk->cpu = 0;
1567 }
1568
1569 // Acquire the lock.
1570 // Loops (spins) until the lock is acquired.
1571 // Holding a lock for a long time may cause
1572 // other CPUs to waste time spinning to acquire it.
1573 void
1574 acquire(struct spinlock *lk)
1575 {
1576   pushcli(); // disable interrupts to avoid deadlock.
1577   if(holding(lk))
1578     panic("acquire");
1579
1580   // The xchg is atomic.
1581   // It also serializes, so that reads after acquire are not
1582   // reordered before it.
1583   while(xchg(&lk->locked, 1) != 0)
1584     ;
1585
1586   // Record info about lock acquisition for debugging.
1587   lk->cpu = cpu;
1588   getcallerpcs(&lk, lk->pcs);
1589 }
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
```

```
1600 // Release the lock.
1601 void
1602 release(struct spinlock *lk)
1603 {
1604   if(!holding(lk))
1605     panic("release");
1606
1607   lk->pcs[0] = 0;
1608   lk->cpu = 0;
1609
1610   // The xchg serializes, so that reads before release are
1611   // not reordered after it.  The 1996 PentiumPro manual (Volume 3,
1612   // 7.2) says reads can be carried out speculatively and in
1613   // any order, which implies we need to serialize here.
1614   // But the 2007 Intel 64 Architecture Memory Ordering White
1615   // Paper says that Intel 64 and IA-32 will not move a load
1616   // after a store. So lock->locked = 0 would work here.
1617   // The xchg being asm volatile ensures gcc emits it after
1618   // the above assignments (and after the critical section).
1619   xchg(&lk->locked, 0);
1620
1621   popcli();
1622 }
1623
1624 // Record the current call stack in pcs[] by following the %ebp chain.
1625 void
1626 getcallerpcs(void *v, uint pcs[])
1627 {
1628   uint *ebp;
1629   int i;
1630
1631   ebp = (uint*)v - 2;
1632   for(i = 0; i < 10; i++){
1633     if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
1634       break;
1635     pcs[i] = ebp[1];     // saved %eip
1636     ebp = (uint*)ebp[0]; // saved %ebp
1637   }
1638   for(; i < 10; i++)
1639     pcs[i] = 0;
1640 }
1641
1642 // Check whether this cpu is holding the lock.
1643 int
1644 holding(struct spinlock *lock)
1645 {
1646   return lock->locked && lock->cpu == cpu;
1647 }
1648
1649
```



```
1650 // Pushcli/popcli are like cli/sti except that they are matched:
1651 // it takes two popcli to undo two pushcli.  Also, if interrupts
1652 // are off, then pushcli, popcli leaves them off.
1653
1654 void
1655 pushcli(void)
1656 {
1657   int eflags;
1658
1659   eflags = readeflags();
1660   cli();
1661   if(cpu->ncli++ == 0)
1662     cpu->intena = eflags & FL_IF;
1663 }
1664
1665 void
1666 popcli(void)
1667 {
1668   if(readeflags()&FL_IF)
1669     panic("popcli - interruptible");
1670   if(--cpu->ncli < 0)
1671     panic("popcli");
1672   if(cpu->ncli == 0 && cpu->intena)
1673     sti();
1674 }
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
```

```
1700 #include "param.h"
1701 #include "types.h"
1702 #include "defs.h"
1703 #include "x86.h"
1704 #include "memlayout.h"
1705 #include "mmu.h"
1706 #include "proc.h"
1707 #include "elf.h"
1708
1709 extern char data[];  // defined by kernel.ld
1710 pde_t *kpgdir;  // for use in scheduler()
1711 struct segdesc gdt[NSEGS];
1712
1713 // Set up CPU's kernel segment descriptors.
1714 // Run once on entry on each CPU.
1715 void
1716 seginit(void)
1717 {
1718   struct cpu *c;
1719
1720   // Map "logical" addresses to virtual addresses using identity map.
1721   // Cannot share a CODE descriptor for both kernel and user
1722   // because it would have to have DPL_USR, but the CPU forbids
1723   // an interrupt from CPL=0 to DPL=3.
1724   c = &cpus[cpunum()];
1725   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
1726   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1727   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
1728   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1729
1730   // Map cpu, and curproc
1731   c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
1732
1733   lgdt(c->gdt, sizeof(c->gdt));
1734   loadgs(SEG_KCPU << 3);
1735
1736   // Initialize cpu-local storage.
1737   cpu = c;
1738   proc = 0;
1739 }
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
```



```
1750 // Return the address of the PTE in page table pgdir
1751 // that corresponds to virtual address va.  If alloc!=0,
1752 // create any required page table pages.
1753 static pte_t *
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756   pde_t *pde;
1757   pte_t *pgtab;
1758
1759   pde = &pgdir[PDX(va)];
1760   if(*pde & PTE_P){
1761     pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
1762   } else {
1763     if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1764       return 0;
1765     // Make sure all those PTE_P bits are zero.
1766     memset(pgtab, 0, PGSIZE);
1767     // The permissions here are overly generous, but they can
1768     // be further restricted by the permissions in the page table
1769     // entries, if necessary.
1770     *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
1771   }
1772   return &pgtab[PTX(va)];
1773 }
1774
1775 // Create PTEs for virtual addresses starting at va that refer to
1776 // physical addresses starting at pa. va and size might not
1777 // be page-aligned.
1778 static int
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return -1;
1789     if(*pte & PTE_P)
1790       panic("remap");
1791     *pte = pa | perm | PTE_P;
1792     if(a == last)
1793       break;
1794     a += PGSIZE;
1795     pa += PGSIZE;
1796   }
1797   return 0;
1798 }
1799
```

```
1800 // There is one page table per process, plus one that's used when
1801 // a CPU is not running any process (kpgdir). The kernel uses the
1802 // current process's page table during system calls and interrupts;
1803 // page protection bits prevent user code from using the kernel's
1804 // mappings.
1805 //
1806 // setupkvm() and exec() set up every page table like this:
1807 //
1808 //   0..KERNBASE: user memory (text+data+stack+heap), mapped to
1809 //                phys memory allocated by the kernel
1810 //   KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
1811 //   KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
1812 //                for the kernel's instructions and r/o data
1813 //   data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
1814 //                                  rw data + free physical memory
1815 //   0xfe000000..0: mapped direct (devices such as ioapic)
1816 //
1817 // The kernel allocates physical memory for its heap and for user memory
1818 // between V2P(end) and the end of physical memory (PHYSTOP)
1819 // (directly addressable from end..P2V(PHYSTOP)).
1820
1821 // This table defines the kernel's mappings, which are present in
1822 // every process's page table.
1823 static struct kmap {
1824   void *virt;
1825   uint phys_start;
1826   uint phys_end;
1827   int perm;
1828 } kmap[] = {
1829  { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
1830  { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
1831  { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
1832  { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
1833 };
1834
1835 // Set up kernel part of a page table.
1836 pde_t*
1837 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
1845   if (p2v(PHYSTOP) > (void*)DEVSPACE)
1846     panic("PHYSTOP too high");
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                 (uint)k->phys_start, k->perm) < 0)
```



```
1850       return 0;
1851   return pgdir;
1852 }
1853
1854 // Allocate one page table for the machine for the kernel address
1855 // space for scheduler processes.
1856 void
1857 kvmalloc(void)
1858 {
1859   kpgdir = setupkvm();
1860   switchkvm();
1861 }
1862
1863 // Switch h/w page table register to the kernel-only page table,
1864 // for when no process is running.
1865 void
1866 switchkvm(void)
1867 {
1868   lcr3(v2p(kpgdir));   // switch to the kernel page table
1869 }
1870
1871 // Switch TSS and h/w page table to correspond to process p.
1872 void
1873 switchuvm(struct proc *p)
1874 {
1875   pushcli();
1876   cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
1877   cpu->gdt[SEG_TSS].s = 0;
1878   cpu->ts.ss0 = SEG_KDATA << 3;
1879   cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
1880   ltr(SEG_TSS << 3);
1881   if(p->pgdir == 0)
1882     panic("switchuvm: no pgdir");
1883   lcr3(v2p(p->pgdir));  // switch to new address space
1884   popcli();
1885 }
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
```

```
1900 // Load the initcode into address 0 of pgdir.
1901 // sz must be less than a page.
1902 void
1903 inituvm(pde_t *pgdir, char *init, uint sz)
1904 {
1905   char *mem;
1906
1907   if(sz >= PGSIZE)
1908     panic("inituvm: more than a page");
1909   mem = kalloc();
1910   memset(mem, 0, PGSIZE);
1911   mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
1912   memmove(mem, init, sz);
1913 }
1914
1915 // Load a program segment into pgdir.  addr must be page-aligned
1916 // and the pages from addr to addr+sz must already be mapped.
1917 int
1918 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
1919 {
1920   uint i, pa, n;
1921   pte_t *pte;
1922
1923   if((uint) addr % PGSIZE != 0)
1924     panic("loaduvm: addr must be page aligned");
1925   for(i = 0; i < sz; i += PGSIZE){
1926     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1927       panic("loaduvm: address should exist");
1928     pa = PTE_ADDR(*pte);
1929     if(sz - i < PGSIZE)
1930       n = sz - i;
1931     else
1932       n = PGSIZE;
1933     if(readi(ip, p2v(pa), offset+i, n) != n)
1934       return -1;
1935   }
1936   return 0;
1937 }
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
```



```
1950 // Allocate page tables and physical memory to grow process from oldsz to
1951 // newsz, which need not be page aligned.  Returns new size or 0 on error.
1952 int
1953 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1954 {
1955   char *mem;
1956   uint a;
1957
1958   if(newsz >= KERNBASE)
1959     return 0;
1960   if(newsz < oldsz)
1961     return oldsz;
1962
1963   a = PGROUNDUP(oldsz);
1964   for(; a < newsz; a += PGSIZE){
1965     mem = kalloc();
1966     if(mem == 0){
1967       cprintf("allocuvm out of memory\n");
1968       deallocuvm(pgdir, newsz, oldsz);
1969       return 0;
1970     }
1971     memset(mem, 0, PGSIZE);
1972     mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U);
1973   }
1974   return newsz;
1975 }
1976
1977 // Deallocate user pages to bring the process size from oldsz to
1978 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
1979 // need to be less than oldsz.  oldsz can be larger than the actual
1980 // process size.  Returns the new process size.
1981 int
1982 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1983 {
1984   pte_t *pte;
1985   uint a, pa;
1986
1987   if(newsz >= oldsz)
1988     return oldsz;
1989
1990   a = PGROUNDUP(newsz);
1991   for(; a  < oldsz; a += PGSIZE){
1992     pte = walkpgdir(pgdir, (char*)a, 0);
1993     if(!pte)
1994       a += (NPTENTRIES - 1) * PGSIZE;
1995     else if((*pte & PTE_P) != 0){
1996       pa = PTE_ADDR(*pte);
1997       if(pa == 0)
1998         panic("kfree");
1999       char *v = p2v(pa);
```

```
2000       kfree(v);
2001       *pte = 0;
2002     }
2003   }
2004   return newsz;
2005 }
2006
2007 // Free a page table and all the physical memory pages
2008 // in the user part.
2009 void
2010 freevm(pde_t *pgdir)
2011 {
2012   uint i;
2013
2014   if(pgdir == 0)
2015     panic("freevm: no pgdir");
2016   deallocuvm(pgdir, KERNBASE, 0);
2017   for(i = 0; i < NPDENTRIES; i++){
2018     if(pgdir[i] & PTE_P){
2019       char * v = p2v(PTE_ADDR(pgdir[i]));
2020       kfree(v);
2021     }
2022   }
2023   kfree((char*)pgdir);
2024 }
2025
2026 // Clear PTE_U on a page. Used to create an inaccessible
2027 // page beneath the user stack.
2028 void
2029 clearpteu(pde_t *pgdir, char *uva)
2030 {
2031   pte_t *pte;
2032
2033   pte = walkpgdir(pgdir, uva, 0);
2034   if(pte == 0)
2035     panic("clearpteu");
2036   *pte &= ~PTE_U;
2037 }
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
```



```
2050 // Given a parent process's page table, create a copy
2051 // of it for a child.
2052 pde_t*
2053 copyuvm(pde_t *pgdir, uint sz)
2054 {
2055   pde_t *d;
2056   pte_t *pte;
2057   uint pa, i, flags;
2058   char *mem;
2059
2060   if((d = setupkvm()) == 0)
2061     return 0;
2062   for(i = 0; i < sz; i += PGSIZE){
2063     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
2064       panic("copyuvm: pte should exist");
2065     if(!(*pte & PTE_P))
2066       panic("copyuvm: page not present");
2067     pa = PTE_ADDR(*pte);
2068     flags = PTE_FLAGS(*pte);
2069     if((mem = kalloc()) == 0)
2070       goto bad;
2071     memmove(mem, (char*)p2v(pa), PGSIZE);
2072     if(mappages(d, (void*)i, PGSIZE, v2p(mem), flags) < 0)
2073       goto bad;
2074   }
2075   return d;
2076
2077 bad:
2078   freevm(d);
2079   return 0;
2080 }
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
```

```
2100 // Map user virtual address to kernel address.
2101 char*
2102 uva2ka(pde_t *pgdir, char *uva)
2103 {
2104   pte_t *pte;
2105
2106   pte = walkpgdir(pgdir, uva, 0);
2107   if((*pte & PTE_P) == 0)
2108     return 0;
2109   if((*pte & PTE_U) == 0)
2110     return 0;
2111   return (char*)p2v(PTE_ADDR(*pte));
2112 }
2113
2114 // Copy len bytes from p to user address va in page table pgdir.
2115 // Most useful when pgdir is not the current page table.
2116 // uva2ka ensures this only works for PTE_U pages.
2117 int
2118 copyout(pde_t *pgdir, uint va, void *p, uint len)
2119 {
2120   char *buf, *pa0;
2121   uint n, va0;
2122
2123   buf = (char*)p;
2124   while(len > 0){
2125     va0 = (uint)PGROUNDDOWN(va);
2126     pa0 = uva2ka(pgdir, (char*)va0);
2127     if(pa0 == 0)
2128       return -1;
2129     n = PGSIZE - (va - va0);
2130     if(n > len)
2131       n = len;
2132     memmove(pa0 + (va - va0), buf, n);
2133     len -= n;
2134     buf += n;
2135     va = va0 + PGSIZE;
2136   }
2137   return 0;
2138 }
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
```



```
2150 // Blank page.
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
```

```
2200 // Blank page.
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
```



```
2250 // Blank page.
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
```

```
2300 // Segments in proc->gdt.
2301 #define NSEGS     7
2302
2303 // Per-CPU state
2304 struct cpu {
2305   uchar id;                    // Local APIC ID; index into cpus[] below
2306   struct context *scheduler;   // swtch() here to enter scheduler
2307   struct taskstate ts;         // Used by x86 to find stack for interrupt
2308   struct segdesc gdt[NSEGS];   // x86 global descriptor table
2309   volatile uint started;       // Has the CPU started?
2310   int ncli;                    // Depth of pushcli nesting.
2311   int intena;                  // Were interrupts enabled before pushcli?
2312
2313   // Cpu-local storage variables; see below
2314   struct cpu *cpu;
2315   struct proc *proc;           // The currently-running process.
2316 };
2317
2318 extern struct cpu cpus[NCPU];
2319 extern int ncpu;
2320
2321 // Per-CPU variables, holding pointers to the
2322 // current cpu and to the current process.
2323 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2324 // and "%gs:4" to refer to proc.  seginit sets up the
2325 // %gs segment register so that %gs refers to the memory
2326 // holding those two variables in the local cpu's struct cpu.
2327 // This is similar to how thread-local variables are implemented
2328 // in thread libraries such as Linux pthreads.
2329 extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
2330 extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc
2331
2332
2333 // Saved registers for kernel context switches.
2334 // Don't need to save all the segment registers (%cs, etc),
2335 // because they are constant across kernel contexts.
2336 // Don't need to save %eax, %ecx, %edx, because the
2337 // x86 convention is that the caller has saved them.
2338 // Contexts are stored at the bottom of the stack they
2339 // describe; the stack pointer is the address of the context.
2340 // The layout of the context matches the layout of the stack in swtch.S
2341 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2342 // but it is on the stack and allocproc() manipulates it.
2343 struct context {
2344   uint edi;
2345   uint esi;
2346   uint ebx;
2347   uint ebp;
2348   uint eip;
2349 };
```



```
2350 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2351
2352 // Per-process state
2353 struct proc {
2354   uint sz;                     // Size of process memory (bytes)
2355   pde_t* pgdir;                // Page table
2356   char *kstack;                // Bottom of kernel stack for this process
2357   enum procstate state;        // Process state
2358   int pid;                     // Process ID
2359   struct proc *parent;         // Parent process
2360   struct trapframe *tf;        // Trap frame for current syscall
2361   struct context *context;     // swtch() here to run process
2362   void *chan;                  // If non-zero, sleeping on chan
2363   int killed;                  // If non-zero, have been killed
2364   struct file *ofile[NOFILE];  // Open files
2365   struct inode *cwd;           // Current directory
2366   char name[16];               // Process name (debugging)
2367 };
2368
2369 // Process memory is laid out contiguously, low addresses first:
2370 //   text
2371 //   original data and bss
2372 //   fixed-size stack
2373 //   expandable heap
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
```

```
2400 #include "types.h"
2401 #include "defs.h"
2402 #include "param.h"
2403 #include "memlayout.h"
2404 #include "mmu.h"
2405 #include "x86.h"
2406 #include "proc.h"
2407 #include "spinlock.h"
2408
2409 struct {
2410   struct spinlock lock;
2411   struct proc proc[NPROC];
2412 } ptable;
2413
2414 static struct proc *initproc;
2415
2416 int nextpid = 1;
2417 extern void forkret(void);
2418 extern void trapret(void);
2419
2420 static void wakeup1(void *chan);
2421
2422 void
2423 pinit(void)
2424 {
2425   initlock(&ptable.lock, "ptable");
2426 }
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
```



```
2450 // Look in the process table for an UNUSED proc.
2451 // If found, change state to EMBRYO and initialize
2452 // state required to run in the kernel.
2453 // Otherwise return 0.
2454 static struct proc*
2455 allocproc(void)
2456 {
2457   struct proc *p;
2458   char *sp;
2459
2460   acquire(&ptable.lock);
2461   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2462     if(p->state == UNUSED)
2463       goto found;
2464   release(&ptable.lock);
2465   return 0;
2466
2467 found:
2468   p->state = EMBRYO;
2469   p->pid = nextpid++;
2470   release(&ptable.lock);
2471
2472   // Allocate kernel stack.
2473   if((p->kstack = kalloc()) == 0){
2474     p->state = UNUSED;
2475     return 0;
2476   }
2477   sp = p->kstack + KSTACKSIZE;
2478
2479   // Leave room for trap frame.
2480   sp -= sizeof *p->tf;
2481   p->tf = (struct trapframe*)sp;
2482
2483   // Set up new context to start executing at forkret,
2484   // which returns to trapret.
2485   sp -= 4;
2486   *(uint*)sp = (uint)trapret;
2487
2488   sp -= sizeof *p->context;
2489   p->context = (struct context*)sp;
2490   memset(p->context, 0, sizeof *p->context);
2491   p->context->eip = (uint)forkret;
2492
2493   return p;
2494 }
2495
2496
2497
2498
2499
```

```
2500 // Set up first user process.
2501 void
2502 userinit(void)
2503 {
2504   struct proc *p;
2505   extern char _binary_initcode_start[], _binary_initcode_size[];
2506
2507   p = allocproc();
2508   initproc = p;
2509   if((p->pgdir = setupkvm()) == 0)
2510     panic("userinit: out of memory?");
2511   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2512   p->sz = PGSIZE;
2513   memset(p->tf, 0, sizeof(*p->tf));
2514   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2515   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2516   p->tf->es = p->tf->ds;
2517   p->tf->ss = p->tf->ds;
2518   p->tf->eflags = FL_IF;
2519   p->tf->esp = PGSIZE;
2520   p->tf->eip = 0;  // beginning of initcode.S
2521
2522   safestrcpy(p->name, "initcode", sizeof(p->name));
2523   p->cwd = namei("/");
2524
2525   p->state = RUNNABLE;
2526 }
2527
2528 // Grow current process's memory by n bytes.
2529 // Return 0 on success, -1 on failure.
2530 int
2531 growproc(int n)
2532 {
2533   uint sz;
2534
2535   sz = proc->sz;
2536   if(n > 0){
2537     if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2538       return -1;
2539   } else if(n < 0){
2540     if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2541       return -1;
2542   }
2543   proc->sz = sz;
2544   switchuvm(proc);
2545   return 0;
2546 }
2547
2548
2549
```



```
2550 // Create a new process copying p as the parent.
2551 // Sets up stack to return as if from system call.
2552 // Caller must set state of returned proc to RUNNABLE.
2553 int
2554 fork(void)
2555 {
2556   int i, pid;
2557   struct proc *np;
2558
2559   // Allocate process.
2560   if((np = allocproc()) == 0)
2561     return -1;
2562
2563   // Copy process state from p.
2564   if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2565     kfree(np->kstack);
2566     np->kstack = 0;
2567     np->state = UNUSED;
2568     return -1;
2569   }
2570   np->sz = proc->sz;
2571   np->parent = proc;
2572   *np->tf = *proc->tf;
2573
2574   // Clear %eax so that fork returns 0 in the child.
2575   np->tf->eax = 0;
2576
2577   for(i = 0; i < NOFILE; i++)
2578     if(proc->ofile[i])
2579       np->ofile[i] = filedup(proc->ofile[i]);
2580   np->cwd = idup(proc->cwd);
2581
2582   safestrcpy(np->name, proc->name, sizeof(proc->name));
2583
2584   pid = np->pid;
2585
2586   // lock to force the compiler to emit the np->state write last.
2587   acquire(&ptable.lock);
2588   np->state = RUNNABLE;
2589   release(&ptable.lock);
2590
2591   return pid;
2592 }
2593
2594
2595
2596
2597
2598
2599
```

```
2600 // Exit the current process.  Does not return.
2601 // An exited process remains in the zombie state
2602 // until its parent calls wait() to find out it exited.
2603 void
2604 exit(void)
2605 {
2606   struct proc *p;
2607   int fd;
2608
2609   if(proc == initproc)
2610     panic("init exiting");
2611
2612   // Close all open files.
2613   for(fd = 0; fd < NOFILE; fd++){
2614     if(proc->ofile[fd]){
2615       fileclose(proc->ofile[fd]);
2616       proc->ofile[fd] = 0;
2617     }
2618   }
2619
2620   begin_op();
2621   iput(proc->cwd);
2622   end_op();
2623   proc->cwd = 0;
2624
2625   acquire(&ptable.lock);
2626
2627   // Parent might be sleeping in wait().
2628   wakeup1(proc->parent);
2629
2630   // Pass abandoned children to init.
2631   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2632     if(p->parent == proc){
2633       p->parent = initproc;
2634       if(p->state == ZOMBIE)
2635         wakeup1(initproc);
2636     }
2637   }
2638
2639   // Jump into the scheduler, never to return.
2640   proc->state = ZOMBIE;
2641   sched();
2642   panic("zombie exit");
2643 }
2644
2645
2646
2647
2648
2649
```



```
2650 // Wait for a child process to exit and return its pid.
2651 // Return -1 if this process has no children.
2652 int
2653 wait(void)
2654 {
2655   struct proc *p;
2656   int havekids, pid;
2657
2658   acquire(&ptable.lock);
2659   for(;;){
2660     // Scan through table looking for zombie children.
2661     havekids = 0;
2662     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2663       if(p->parent != proc)
2664         continue;
2665       havekids = 1;
2666       if(p->state == ZOMBIE){
2667         // Found one.
2668         pid = p->pid;
2669         kfree(p->kstack);
2670         p->kstack = 0;
2671         freevm(p->pgdir);
2672         p->state = UNUSED;
2673         p->pid = 0;
2674         p->parent = 0;
2675         p->name[0] = 0;
2676         p->killed = 0;
2677         release(&ptable.lock);
2678         return pid;
2679       }
2680     }
2681
2682     // No point waiting if we don't have any children.
2683     if(!havekids || proc->killed){
2684       release(&ptable.lock);
2685       return -1;
2686     }
2687
2688     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2689     sleep(proc, &ptable.lock);
2690   }
2691 }
2692
2693
2694
2695
2696
2697
2698
2699
```

```
2700 // Per-CPU process scheduler.
2701 // Each CPU calls scheduler() after setting itself up.
2702 // Scheduler never returns.  It loops, doing:
2703 //  - choose a process to run
2704 //  - swtch to start running that process
2705 //  - eventually that process transfers control
2706 //      via swtch back to the scheduler.
2707 void
2708 scheduler(void)
2709 {
2710   struct proc *p;
2711
2712   for(;;){
2713     // Enable interrupts on this processor.
2714     sti();
2715
2716     // Loop over process table looking for process to run.
2717     acquire(&ptable.lock);
2718     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2719       if(p->state != RUNNABLE)
2720         continue;
2721
2722       // Switch to chosen process.  It is the process's job
2723       // to release ptable.lock and then reacquire it
2724       // before jumping back to us.
2725       proc = p;
2726       switchuvm(p);
2727       p->state = RUNNING;
2728       swtch(&cpu->scheduler, proc->context);
2729       switchkvm();
2730
2731       // Process is done running for now.
2732       // It should have changed its p->state before coming back.
2733       proc = 0;
2734     }
2735     release(&ptable.lock);
2736
2737   }
2738 }
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
```



```
2750 // Enter scheduler.  Must hold only ptable.lock
2751 // and have changed proc->state.
2752 void
2753 sched(void)
2754 {
2755   int intena;
2756
2757   if(!holding(&ptable.lock))
2758     panic("sched ptable.lock");
2759   if(cpu->ncli != 1)
2760     panic("sched locks");
2761   if(proc->state == RUNNING)
2762     panic("sched running");
2763   if(readeflags()&FL_IF)
2764     panic("sched interruptible");
2765   intena = cpu->intena;
2766   swtch(&proc->context, cpu->scheduler);
2767   cpu->intena = intena;
2768 }
2769
2770 // Give up the CPU for one scheduling round.
2771 void
2772 yield(void)
2773 {
2774   acquire(&ptable.lock);
2775   proc->state = RUNNABLE;
2776   sched();
2777   release(&ptable.lock);
2778 }
2779
2780 // A fork child's very first scheduling by scheduler()
2781 // will swtch here.  "Return" to user space.
2782 void
2783 forkret(void)
2784 {
2785   static int first = 1;
2786   // Still holding ptable.lock from scheduler.
2787   release(&ptable.lock);
2788
2789   if (first) {
2790     // Some initialization functions must be run in the context
2791     // of a regular process (e.g., they call sleep), and thus cannot
2792     // be run from main().
2793     first = 0;
2794     iinit(ROOTDEV);
2795     initlog(ROOTDEV);
2796   }
2797
2798   // Return to "caller", actually trapret (see allocproc).
2799 }
```

```
2800 // Atomically release lock and sleep on chan.
2801 // Reacquires lock when awakened.
2802 void
2803 sleep(void *chan, struct spinlock *lk)
2804 {
2805   if(proc == 0)
2806     panic("sleep");
2807
2808   if(lk == 0)
2809     panic("sleep without lk");
2810
2811   // Must acquire ptable.lock in order to
2812   // change p->state and then call sched.
2813   // Once we hold ptable.lock, we can be
2814   // guaranteed that we won't miss any wakeup
2815   // (wakeup runs with ptable.lock locked),
2816   // so it's okay to release lk.
2817   if(lk != &ptable.lock){
2818     acquire(&ptable.lock);
2819     release(lk);
2820   }
2821
2822   // Go to sleep.
2823   proc->chan = chan;
2824   proc->state = SLEEPING;
2825   sched();
2826
2827   // Tidy up.
2828   proc->chan = 0;
2829
2830   // Reacquire original lock.
2831   if(lk != &ptable.lock){
2832     release(&ptable.lock);
2833     acquire(lk);
2834   }
2835 }
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
```



```
2850 // Wake up all processes sleeping on chan.
2851 // The ptable lock must be held.
2852 static void
2853 wakeup1(void *chan)
2854 {
2855   struct proc *p;
2856
2857   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2858     if(p->state == SLEEPING && p->chan == chan)
2859       p->state = RUNNABLE;
2860 }
2861
2862 // Wake up all processes sleeping on chan.
2863 void
2864 wakeup(void *chan)
2865 {
2866   acquire(&ptable.lock);
2867   wakeup1(chan);
2868   release(&ptable.lock);
2869 }
2870
2871 // Kill the process with the given pid.
2872 // Process won't exit until it returns
2873 // to user space (see trap in trap.c).
2874 int
2875 kill(int pid)
2876 {
2877   struct proc *p;
2878
2879   acquire(&ptable.lock);
2880   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2881     if(p->pid == pid){
2882       p->killed = 1;
2883       // Wake process from sleep if necessary.
2884       if(p->state == SLEEPING)
2885         p->state = RUNNABLE;
2886       release(&ptable.lock);
2887       return 0;
2888     }
2889   }
2890   release(&ptable.lock);
2891   return -1;
2892 }
2893
2894
2895
2896
2897
2898
2899
```

```
2900 // Print a process listing to console.  For debugging.
2901 // Runs when user types ^P on console.
2902 // No lock to avoid wedging a stuck machine further.
2903 void
2904 procdump(void)
2905 {
2906   static char *states[] = {
2907   [UNUSED]    "unused",
2908   [EMBRYO]    "embryo",
2909   [SLEEPING]  "sleep ",
2910   [RUNNABLE]  "runble",
2911   [RUNNING]   "run   ",
2912   [ZOMBIE]    "zombie"
2913   };
2914   int i;
2915   struct proc *p;
2916   char *state;
2917   uint pc[10];
2918
2919   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2920     if(p->state == UNUSED)
2921       continue;
2922     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
2923       state = states[p->state];
2924     else
2925       state = "???";
2926     cprintf("%d %s %s", p->pid, state, p->name);
2927     if(p->state == SLEEPING){
2928       getcallerpcs((uint*)p->context->ebp+2, pc);
2929       for(i=0; i<10 && pc[i] != 0; i++)
2930         cprintf(" %p", pc[i]);
2931     }
2932     cprintf("\n");
2933   }
2934 }
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
```



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

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



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

```
3100 }
3101
3102
3103
3104
3105
3106
3107
3108
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
```



```
3150 // x86 trap and interrupt constants.
3151
3152 // Processor-defined:
3153 #define T_DIVIDE         0      // divide error
3154 #define T_DEBUG          1      // debug exception
3155 #define T_NMI            2      // non-maskable interrupt
3156 #define T_BRKPT          3      // breakpoint
3157 #define T_OFLOW          4      // overflow
3158 #define T_BOUND          5      // bounds check
3159 #define T_ILLOP          6      // illegal opcode
3160 #define T_DEVICE         7      // device not available
3161 #define T_DBLFLT         8      // double fault
3162 // #define T_COPROC      9      // reserved (not used since 486)
3163 #define T_TSS           10      // invalid task switch segment
3164 #define T_SEGNP         11      // segment not present
3165 #define T_STACK         12      // stack exception
3166 #define T_GPFLT         13      // general protection fault
3167 #define T_PGFLT         14      // page fault
3168 // #define T_RES        15      // reserved
3169 #define T_FPERR         16      // floating point error
3170 #define T_ALIGN         17      // aligment check
3171 #define T_MCHK          18      // machine check
3172 #define T_SIMDERR       19      // SIMD floating point error
3173
3174 // These are arbitrarily chosen, but with care not to overlap
3175 // processor defined exceptions or interrupt vectors.
3176 #define T_SYSCALL       64      // system call
3177 #define T_DEFAULT      500      // catchall
3178
3179 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
3180
3181 #define IRQ_TIMER        0
3182 #define IRQ_KBD          1
3183 #define IRQ_COM1         4
3184 #define IRQ_IDE         14
3185 #define IRQ_ERROR       19
3186 #define IRQ_SPURIOUS    31
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
```

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

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

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

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

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



```
#include "mmu.h"

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

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

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

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

```
3300 #include "types.h"
3301 #include "defs.h"
3302 #include "param.h"
3303 #include "memlayout.h"
3304 #include "mmu.h"
3305 #include "proc.h"
3306 #include "x86.h"
3307 #include "traps.h"
3308 #include "spinlock.h"
3309
3310 // Interrupt descriptor table (shared by all CPUs).
3311 struct gatedesc idt[256];
3312 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
3313 struct spinlock tickslock;
3314 uint ticks;
3315
3316 void
3317 tvinit(void)
3318 {
3319   int i;
3320
3321   for(i = 0; i < 256; i++)
3322     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3323   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3324
3325   initlock(&tickslock, "time");
3326 }
3327
3328 void
3329 idtinit(void)
3330 {
3331   lidt(idt, sizeof(idt));
3332 }
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
```



```
3350 void
3351 trap(struct trapframe *tf)
3352 {
3353   if(tf->trapno == T_SYSCALL){
3354     if(proc->killed)
3355       exit();
3356     proc->tf = tf;
3357     syscall();
3358     if(proc->killed)
3359       exit();
3360     return;
3361   }
3362
3363   switch(tf->trapno){
3364   case T_IRQ0 + IRQ_TIMER:
3365     if(cpu->id == 0){
3366       acquire(&tickslock);
3367       ticks++;
3368       wakeup(&ticks);
3369       release(&tickslock);
3370     }
3371     lapiceoi();
3372     break;
3373   case T_IRQ0 + IRQ_IDE:
3374     ideintr();
3375     lapiceoi();
3376     break;
3377   case T_IRQ0 + IRQ_IDE+1:
3378     // Bochs generates spurious IDE1 interrupts.
3379     break;
3380   case T_IRQ0 + IRQ_KBD:
3381     kbdintr();
3382     lapiceoi();
3383     break;
3384   case T_IRQ0 + IRQ_COM1:
3385     uartintr();
3386     lapiceoi();
3387     break;
3388   case T_IRQ0 + 7:
3389   case T_IRQ0 + IRQ_SPURIOUS:
3390     cprintf("cpu%d: spurious interrupt at %x:%x\n",
3391             cpu->id, tf->cs, tf->eip);
3392     lapiceoi();
3393     break;
3394
3395
3396
3397
3398
3399
```

```
3400   default:
3401     if(proc == 0 || (tf->cs&3) == 0){
3402       // In kernel, it must be our mistake.
3403       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3404               tf->trapno, cpu->id, tf->eip, rcr2());
3405       panic("trap");
3406     }
3407     // In user space, assume process misbehaved.
3408     cprintf("pid %d %s: trap %d err %d on cpu %d "
3409             "eip 0x%x addr 0x%x--kill proc\n",
3410             proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
3411             rcr2());
3412     proc->killed = 1;
3413   }
3414
3415   // Force process exit if it has been killed and is in user space.
3416   // (If it is still executing in the kernel, let it keep running
3417   // until it gets to the regular system call return.)
3418   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3419     exit();
3420
3421   // Force process to give up CPU on clock tick.
3422   // If interrupts were on while locks held, would need to check nlock.
3423   if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3424     yield();
3425
3426   // Check if the process has been killed since we yielded
3427   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3428     exit();
3429 }
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
```



```
3450 // System call numbers
3451 #define SYS_fork    1
3452 #define SYS_exit    2
3453 #define SYS_wait    3
3454 #define SYS_pipe    4
3455 #define SYS_read    5
3456 #define SYS_kill    6
3457 #define SYS_exec    7
3458 #define SYS_fstat   8
3459 #define SYS_chdir   9
3460 #define SYS_dup    10
3461 #define SYS_getpid 11
3462 #define SYS_sbrk   12
3463 #define SYS_sleep  13
3464 #define SYS_uptime 14
3465 #define SYS_open   15
3466 #define SYS_write  16
3467 #define SYS_mknod  17
3468 #define SYS_unlink 18
3469 #define SYS_link   19
3470 #define SYS_mkdir  20
3471 #define SYS_close  21
3472 #define SYS_date   22
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 #include "types.h"
3501 #include "defs.h"
3502 #include "param.h"
3503 #include "memlayout.h"
3504 #include "mmu.h"
3505 #include "proc.h"
3506 #include "x86.h"
3507 #include "syscall.h"
3508
3509 // User code makes a system call with INT T_SYSCALL.
3510 // System call number in %eax.
3511 // Arguments on the stack, from the user call to the C
3512 // library system call function. The saved user %esp points
3513 // to a saved program counter, and then the first argument.
3514
3515 // Fetch the int at addr from the current process.
3516 int
3517 fetchint(uint addr, int *ip)
3518 {
3519   if(addr >= proc->sz || addr+4 > proc->sz)
3520     return -1;
3521   *ip = *(int*)(addr);
3522   return 0;
3523 }
3524
3525 // Fetch the nul-terminated string at addr from the current process.
3526 // Doesn't actually copy the string - just sets *pp to point at it.
3527 // Returns length of string, not including nul.
3528 int
3529 fetchstr(uint addr, char **pp)
3530 {
3531   char *s, *ep;
3532
3533   if(addr >= proc->sz)
3534     return -1;
3535   *pp = (char*)addr;
3536   ep = (char*)proc->sz;
3537   for(s = *pp; s < ep; s++)
3538     if(*s == 0)
3539       return s - *pp;
3540   return -1;
3541 }
3542
3543 // Fetch the nth 32-bit system call argument.
3544 int
3545 argint(int n, int *ip)
3546 {
3547   return fetchint(proc->tf->esp + 4 + 4*n, ip);
3548 }
3549
```



```
3550 // Fetch the nth word-sized system call argument as a pointer
3551 // to a block of memory of size n bytes.  Check that the pointer
3552 // lies within the process address space.
3553 int
3554 argptr(int n, char **pp, int size)
3555 {
3556   int i;
3557
3558   if(argint(n, &i) < 0)
3559     return -1;
3560   if((uint)i >= proc->sz || (uint)i+size > proc->sz)
3561     return -1;
3562   *pp = (char*)i;
3563   return 0;
3564 }
3565
3566 // Fetch the nth word-sized system call argument as a string pointer.
3567 // Check that the pointer is valid and the string is nul-terminated.
3568 // (There is no shared writable memory, so the string can't change
3569 // between this check and being used by the kernel.)
3570 int
3571 argstr(int n, char **pp)
3572 {
3573   int addr;
3574   if(argint(n, &addr) < 0)
3575     return -1;
3576   return fetchstr(addr, pp);
3577 }
3578
3579 extern int sys_chdir(void);
3580 extern int sys_close(void);
3581 extern int sys_dup(void);
3582 extern int sys_exec(void);
3583 extern int sys_exit(void);
3584 extern int sys_fork(void);
3585 extern int sys_fstat(void);
3586 extern int sys_getpid(void);
3587 extern int sys_kill(void);
3588 extern int sys_link(void);
3589 extern int sys_mkdir(void);
3590 extern int sys_mknod(void);
3591 extern int sys_open(void);
3592 extern int sys_pipe(void);
3593 extern int sys_read(void);
3594 extern int sys_sbrk(void);
3595 extern int sys_sleep(void);
3596 extern int sys_unlink(void);
3597 extern int sys_wait(void);
3598 extern int sys_write(void);
3599 extern int sys_uptime(void);
```

```
3600 extern int sys_date(void);
3601
3602 static int (*syscalls[])(void) = {
3603 [SYS_fork]    sys_fork,
3604 [SYS_exit]    sys_exit,
3605 [SYS_wait]    sys_wait,
3606 [SYS_pipe]    sys_pipe,
3607 [SYS_read]    sys_read,
3608 [SYS_kill]    sys_kill,
3609 [SYS_exec]    sys_exec,
3610 [SYS_fstat]   sys_fstat,
3611 [SYS_chdir]   sys_chdir,
3612 [SYS_dup]     sys_dup,
3613 [SYS_getpid]  sys_getpid,
3614 [SYS_sbrk]    sys_sbrk,
3615 [SYS_sleep]   sys_sleep,
3616 [SYS_uptime]  sys_uptime,
3617 [SYS_open]    sys_open,
3618 [SYS_write]   sys_write,
3619 [SYS_mknod]   sys_mknod,
3620 [SYS_unlink]  sys_unlink,
3621 [SYS_link]    sys_link,
3622 [SYS_mkdir]   sys_mkdir,
3623 [SYS_close]   sys_close,
3624 [SYS_date]    sys_date
3625 };
3626
3627 void
3628 syscall(void)
3629 {
3630   int num;
3631
3632   num = proc->tf->eax;
3633   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3634     proc->tf->eax = syscalls[num]();
3635     #ifdef PRINT_SYSCALLS
3636     static char* syscallnames[] = {
3637       [SYS_fork]    "fork",
3638       [SYS_exit]    "exit",
3639       [SYS_wait]    "wait",
3640       [SYS_pipe]    "pipe",
3641       [SYS_read]    "read",
3642       [SYS_kill]    "kill",
3643       [SYS_exec]    "exec",
3644       [SYS_fstat]   "fstat",
3645       [SYS_chdir]   "chdir",
3646       [SYS_dup]     "dup",
3647       [SYS_getpid]  "getpid",
3648       [SYS_sbrk]    "sbrk",
3649       [SYS_sleep]   "sleep",
```



```
3650       [SYS_uptime]  "uptime",
3651       [SYS_open]    "open",
3652       [SYS_write]   "write",
3653       [SYS_mknod]   "mknod",
3654       [SYS_unlink]  "unlink",
3655       [SYS_link]    "link",
3656       [SYS_mkdir]   "mkdir",
3657       [SYS_close]   "close",
3658       [SYS_date]    "date"
3659     };
3660     char* syscall = syscallnames[num];
3661     cprintf("%s -> %d\n",
3662             syscall, proc->tf->eax);
3663     #endif
3664   } else {
3665     cprintf("%d %s: unknown sys call %d\n",
3666             proc->pid, proc->name, num);
3667     proc->tf->eax = -1;
3668   }
3669 }
3670
3671
3672
3673
3674
3675
3676
3677
3678
3679
3680
3681
3682
3683
3684
3685
3686
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699
```

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



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

```
3800 int
3801 sys_date(void)
3802 {
3803    struct rtcdate *r;
3804
3805    if(argptr(0, (void*)&r, sizeof(*r)) < 0)
3806      return -1;
3807    cmostime(r);
3808    return 0;
3809 }
3810
3811
3812
3813
3814
3815
3816
3817
3818
3819
3820
3821
3822
3823
3824
3825
3826
3827
3828
3829
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
```



```
3850 struct buf {
3851   int flags;
3852   uint dev;
3853   uint blockno;
3854   struct buf *prev; // LRU cache list
3855   struct buf *next;
3856   struct buf *qnext; // disk queue
3857   uchar data[BSIZE];
3858 };
3859 #define B_BUSY  0x1  // buffer is locked by some process
3860 #define B_VALID 0x2  // buffer has been read from disk
3861 #define B_DIRTY 0x4  // buffer needs to be written to disk
3862
3863
3864
3865
3866
3867
3868
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
```

```
3900 #define O_RDONLY  0x000
3901 #define O_WRONLY  0x001
3902 #define O_RDWR    0x002
3903 #define O_CREATE  0x200
3904
3905
3906
3907
3908
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
```



```
3950 #define T_DIR  1   // Directory
3951 #define T_FILE 2   // File
3952 #define T_DEV  3   // Device
3953
3954 struct stat {
3955   short type;  // Type of file
3956   int dev;     // File system's disk device
3957   uint ino;    // Inode number
3958   short nlink; // Number of links to file
3959   uint size;   // Size of file in bytes
3960 };
3961
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
```

```
4000 // On-disk file system format.
4001 // Both the kernel and user programs use this header file.
4002
4003
4004 #define ROOTINO 1  // root i-number
4005 #define BSIZE 512  // block size
4006
4007 // Disk layout:
4008 // [ boot block | super block | log | inode blocks | free bit map | data blo
4009 //
4010 // mkfs computes the super block and builds an initial file system. The supe
4011 // the disk layout:
4012 struct superblock {
4013   uint size;         // Size of file system image (blocks)
4014   uint nblocks;      // Number of data blocks
4015   uint ninodes;      // Number of inodes.
4016   uint nlog;         // Number of log blocks
4017   uint logstart;     // Block number of first log block
4018   uint inodestart;   // Block number of first inode block
4019   uint bmapstart;    // Block number of first free map block
4020 };
4021
4022 #define NDIRECT 12
4023 #define NINDIRECT (BSIZE / sizeof(uint))
4024 #define MAXFILE (NDIRECT + NINDIRECT)
4025
4026 // On-disk inode structure
4027 struct dinode {
4028   short type;           // File type
4029   short major;          // Major device number (T_DEV only)
4030   short minor;          // Minor device number (T_DEV only)
4031   short nlink;          // Number of links to inode in file system
4032   uint size;            // Size of file (bytes)
4033   uint addrs[NDIRECT+1];   // Data block addresses
4034 };
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
```



```
4050 // Inodes per block.
4051 #define IPB           (BSIZE / sizeof(struct dinode))
4052
4053 // Block containing inode i
4054 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
4055
4056 // Bitmap bits per block
4057 #define BPB           (BSIZE*8)
4058
4059 // Block of free map containing bit for block b
4060 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
4061
4062 // Directory is a file containing a sequence of dirent structures.
4063 #define DIRSIZ 14
4064
4065 struct dirent {
4066   ushort inum;
4067   char name[DIRSIZ];
4068 };
4069
4070
4071
4072
4073
4074
4075
4076
4077
4078
4079
4080
4081
4082
4083
4084
4085
4086
4087
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
```

```
4100 struct file {
4101   enum { FD_NONE, FD_PIPE, FD_INODE } type;
4102   int ref; // reference count
4103   char readable;
4104   char writable;
4105   struct pipe *pipe;
4106   struct inode *ip;
4107   uint off;
4108 };
4109
4110
4111 // in-memory copy of an inode
4112 struct inode {
4113   uint dev;           // Device number
4114   uint inum;          // Inode number
4115   int ref;            // Reference count
4116   int flags;          // I_BUSY, I_VALID
4117
4118   short type;         // copy of disk inode
4119   short major;
4120   short minor;
4121   short nlink;
4122   uint size;
4123   uint addrs[NDIRECT+1];
4124 };
4125 #define I_BUSY 0x1
4126 #define I_VALID 0x2
4127
4128 // table mapping major device number to
4129 // device functions
4130 struct devsw {
4131   int (*read)(struct inode*, char*, int);
4132   int (*write)(struct inode*, char*, int);
4133 };
4134
4135 extern struct devsw devsw[];
4136
4137 #define CONSOLE 1
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```



```
4150 // Blank page.
4151
4152
4153
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 // Simple PIO-based (non-DMA) IDE driver code.
4201
4202 #include "types.h"
4203 #include "defs.h"
4204 #include "param.h"
4205 #include "memlayout.h"
4206 #include "mmu.h"
4207 #include "proc.h"
4208 #include "x86.h"
4209 #include "traps.h"
4210 #include "spinlock.h"
4211 #include "fs.h"
4212 #include "buf.h"
4213
4214 #define SECTOR_SIZE   512
4215 #define IDE_BSY       0x80
4216 #define IDE_DRDY      0x40
4217 #define IDE_DF        0x20
4218 #define IDE_ERR       0x01
4219
4220 #define IDE_CMD_READ  0x20
4221 #define IDE_CMD_WRITE 0x30
4222
4223 // idequeue points to the buf now being read/written to the disk.
4224 // idequeue->qnext points to the next buf to be processed.
4225 // You must hold idelock while manipulating queue.
4226
4227 static struct spinlock idelock;
4228 static struct buf *idequeue;
4229
4230 static int havedisk1;
4231 static void idestart(struct buf*);
4232
4233 // Wait for IDE disk to become ready.
4234 static int
4235 idewait(int checkerr)
4236 {
4237   int r;
4238
4239   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4240     ;
4241   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4242     return -1;
4243   return 0;
4244 }
4245
4246
4247
4248
4249
```



```
4250 void
4251 ideinit(void)
4252 {
4253   int i;
4254
4255   initlock(&idelock, "ide");
4256   picenable(IRQ_IDE);
4257   ioapicenable(IRQ_IDE, ncpu - 1);
4258   idewait(0);
4259
4260   // Check if disk 1 is present
4261   outb(0x1f6, 0xe0 | (1<<4));
4262   for(i=0; i<1000; i++){
4263     if(inb(0x1f7) != 0){
4264       havedisk1 = 1;
4265       break;
4266     }
4267   }
4268
4269   // Switch back to disk 0.
4270   outb(0x1f6, 0xe0 | (0<<4));
4271 }
4272
4273 // Start the request for b.  Caller must hold idelock.
4274 static void
4275 idestart(struct buf *b)
4276 {
4277   if(b == 0)
4278     panic("idestart");
4279   if(b->blockno >= FSSIZE)
4280     panic("incorrect blockno");
4281   int sector_per_block =  BSIZE/SECTOR_SIZE;
4282   int sector = b->blockno * sector_per_block;
4283
4284   if (sector_per_block > 7) panic("idestart");
4285
4286   idewait(0);
4287   outb(0x3f6, 0);  // generate interrupt
4288   outb(0x1f2, sector_per_block);  // number of sectors
4289   outb(0x1f3, sector & 0xff);
4290   outb(0x1f4, (sector >> 8) & 0xff);
4291   outb(0x1f5, (sector >> 16) & 0xff);
4292   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4293   if(b->flags & B_DIRTY){
4294     outb(0x1f7, IDE_CMD_WRITE);
4295     outsl(0x1f0, b->data, BSIZE/4);
4296   } else {
4297     outb(0x1f7, IDE_CMD_READ);
4298   }
4299 }
```

```
4300 // Interrupt handler.
4301 void
4302 ideintr(void)
4303 {
4304   struct buf *b;
4305
4306   // First queued buffer is the active request.
4307   acquire(&idelock);
4308   if((b = idequeue) == 0){
4309     release(&idelock);
4310     // cprintf("spurious IDE interrupt\n");
4311     return;
4312   }
4313   idequeue = b->qnext;
4314
4315   // Read data if needed.
4316   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4317     insl(0x1f0, b->data, BSIZE/4);
4318
4319   // Wake process waiting for this buf.
4320   b->flags |= B_VALID;
4321   b->flags &= ~B_DIRTY;
4322   wakeup(b);
4323
4324   // Start disk on next buf in queue.
4325   if(idequeue != 0)
4326     idestart(idequeue);
4327
4328   release(&idelock);
4329 }
4330
4331
4332
4333
4334
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
```



```
4350 // Sync buf with disk.
4351 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
4352 // Else if B_VALID is not set, read buf from disk, set B_VALID.
4353 void
4354 iderw(struct buf *b)
4355 {
4356   struct buf **pp;
4357
4358   if(!(b->flags & B_BUSY))
4359     panic("iderw: buf not busy");
4360   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
4361     panic("iderw: nothing to do");
4362   if(b->dev != 0 && !havedisk1)
4363     panic("iderw: ide disk 1 not present");
4364
4365   acquire(&idelock);
4366
4367   // Append b to idequeue.
4368   b->qnext = 0;
4369   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
4370     ;
4371   *pp = b;
4372
4373   // Start disk if necessary.
4374   if(idequeue == b)
4375     idestart(b);
4376
4377   // Wait for request to finish.
4378   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
4379     sleep(b, &idelock);
4380   }
4381
4382   release(&idelock);
4383 }
4384
4385
4386
4387
4388
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
```

```
4400 // Buffer cache.
4401 //
4402 // The buffer cache is a linked list of buf structures holding
4403 // cached copies of disk block contents.  Caching disk blocks
4404 // in memory reduces the number of disk reads and also provides
4405 // a synchronization point for disk blocks used by multiple processes.
4406 //
4407 // Interface:
4408 // * To get a buffer for a particular disk block, call bread.
4409 // * After changing buffer data, call bwrite to write it to disk.
4410 // * When done with the buffer, call brelse.
4411 // * Do not use the buffer after calling brelse.
4412 // * Only one process at a time can use a buffer,
4413 //     so do not keep them longer than necessary.
4414 //
4415 // The implementation uses three state flags internally:
4416 // * B_BUSY: the block has been returned from bread
4417 //     and has not been passed back to brelse.
4418 // * B_VALID: the buffer data has been read from the disk.
4419 // * B_DIRTY: the buffer data has been modified
4420 //     and needs to be written to disk.
4421
4422 #include "types.h"
4423 #include "defs.h"
4424 #include "param.h"
4425 #include "spinlock.h"
4426 #include "fs.h"
4427 #include "buf.h"
4428
4429 struct {
4430   struct spinlock lock;
4431   struct buf buf[NBUF];
4432
4433   // Linked list of all buffers, through prev/next.
4434   // head.next is most recently used.
4435   struct buf head;
4436 } bcache;
4437
4438 void
4439 binit(void)
4440 {
4441   struct buf *b;
4442
4443   initlock(&bcache.lock, "bcache");
4444
4445
4446
4447
4448
4449
```



```
4450   // Create linked list of buffers
4451   bcache.head.prev = &bcache.head;
4452   bcache.head.next = &bcache.head;
4453   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4454     b->next = bcache.head.next;
4455     b->prev = &bcache.head;
4456     b->dev = -1;
4457     bcache.head.next->prev = b;
4458     bcache.head.next = b;
4459   }
4460 }
4461
4462 // Look through buffer cache for block on device dev.
4463 // If not found, allocate a buffer.
4464 // In either case, return B_BUSY buffer.
4465 static struct buf*
4466 bget(uint dev, uint blockno)
4467 {
4468   struct buf *b;
4469
4470   acquire(&bcache.lock);
4471
4472  loop:
4473   // Is the block already cached?
4474   for(b = bcache.head.next; b != &bcache.head; b = b->next){
4475     if(b->dev == dev && b->blockno == blockno){
4476       if(!(b->flags & B_BUSY)){
4477         b->flags |= B_BUSY;
4478         release(&bcache.lock);
4479         return b;
4480       }
4481       sleep(b, &bcache.lock);
4482       goto loop;
4483     }
4484   }
4485
4486   // Not cached; recycle some non-busy and clean buffer.
4487   // "clean" because B_DIRTY and !B_BUSY means log.c
4488   // hasn't yet committed the changes to the buffer.
4489   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4490     if((b->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
4491       b->dev = dev;
4492       b->blockno = blockno;
4493       b->flags = B_BUSY;
4494       release(&bcache.lock);
4495       return b;
4496     }
4497   }
4498   panic("bget: no buffers");
4499 }
```

```
4500 // Return a B_BUSY buf with the contents of the indicated block.
4501 struct buf*
4502 bread(uint dev, uint blockno)
4503 {
4504   struct buf *b;
4505
4506   b = bget(dev, blockno);
4507   if(!(b->flags & B_VALID)) {
4508     iderw(b);
4509   }
4510   return b;
4511 }
4512
4513 // Write b's contents to disk.  Must be B_BUSY.
4514 void
4515 bwrite(struct buf *b)
4516 {
4517   if((b->flags & B_BUSY) == 0)
4518     panic("bwrite");
4519   b->flags |= B_DIRTY;
4520   iderw(b);
4521 }
4522
4523 // Release a B_BUSY buffer.
4524 // Move to the head of the MRU list.
4525 void
4526 brelse(struct buf *b)
4527 {
4528   if((b->flags & B_BUSY) == 0)
4529     panic("brelse");
4530
4531   acquire(&bcache.lock);
4532
4533   b->next->prev = b->prev;
4534   b->prev->next = b->next;
4535   b->next = bcache.head.next;
4536   b->prev = &bcache.head;
4537   bcache.head.next->prev = b;
4538   bcache.head.next = b;
4539
4540   b->flags &= ~B_BUSY;
4541   wakeup(b);
4542
4543   release(&bcache.lock);
4544 }
4545
4546
4547
4548
4549
```



```
4550 // Blank page.
4551
4552
4553
4554
4555
4556
4557
4558
4559
4560
4561
4562
4563
4564
4565
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 #include "types.h"
4601 #include "defs.h"
4602 #include "param.h"
4603 #include "spinlock.h"
4604 #include "fs.h"
4605 #include "buf.h"
4606
4607 // Simple logging that allows concurrent FS system calls.
4608 //
4609 // A log transaction contains the updates of multiple FS system
4610 // calls. The logging system only commits when there are
4611 // no FS system calls active. Thus there is never
4612 // any reasoning required about whether a commit might
4613 // write an uncommitted system call's updates to disk.
4614 //
4615 // A system call should call begin_op()/end_op() to mark
4616 // its start and end. Usually begin_op() just increments
4617 // the count of in-progress FS system calls and returns.
4618 // But if it thinks the log is close to running out, it
4619 // sleeps until the last outstanding end_op() commits.
4620 //
4621 // The log is a physical re-do log containing disk blocks.
4622 // The on-disk log format:
4623 //   header block, containing block #s for block A, B, C, ...
4624 //   block A
4625 //   block B
4626 //   block C
4627 //   ...
4628 // Log appends are synchronous.
4629
4630 // Contents of the header block, used for both the on-disk header block
4631 // and to keep track in memory of logged block# before commit.
4632 struct logheader {
4633   int n;
4634   int block[LOGSIZE];
4635 };
4636
4637 struct log {
4638   struct spinlock lock;
4639   int start;
4640   int size;
4641   int outstanding; // how many FS sys calls are executing.
4642   int committing;  // in commit(), please wait.
4643   int dev;
4644   struct logheader lh;
4645 };
4646
4647
4648
4649
```



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

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



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

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



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

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



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

```
5000 //
5001 // ilock() is separate from iget() so that system calls can
5002 // get a long-term reference to an inode (as for an open file)
5003 // and only lock it for short periods (e.g., in read()).
5004 // The separation also helps avoid deadlock and races during
5005 // pathname lookup. iget() increments ip->ref so that the inode
5006 // stays cached and pointers to it remain valid.
5007 //
5008 // Many internal file system functions expect the caller to
5009 // have locked the inodes involved; this lets callers create
5010 // multi-step atomic operations.
5011
5012 struct {
5013   struct spinlock lock;
5014   struct inode inode[NINODE];
5015 } icache;
5016
5017 void
5018 iinit(int dev)
5019 {
5020   initlock(&icache.lock, "icache");
5021   readsb(dev, &sb);
5022   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d inodestart %
5023           sb.nblocks, sb.ninodes, sb.nlog, sb.logstart, sb.inodestart, sb.bma
5024 }
5025
5026 static struct inode* iget(uint dev, uint inum);
5027
5028
5029
5030
5031
5032
5033
5034
5035
5036
5037
5038
5039
5040
5041
5042
5043
5044
5045
5046
5047
5048
5049
```



```
5050 // Allocate a new inode with the given type on device dev.
5051 // A free inode has a type of zero.
5052 struct inode*
5053 ialloc(uint dev, short type)
5054 {
5055   int inum;
5056   struct buf *bp;
5057   struct dinode *dip;
5058
5059   for(inum = 1; inum < sb.ninodes; inum++){
5060     bp = bread(dev, IBLOCK(inum, sb));
5061     dip = (struct dinode*)bp->data + inum%IPB;
5062     if(dip->type == 0){  // a free inode
5063       memset(dip, 0, sizeof(*dip));
5064       dip->type = type;
5065       log_write(bp);   // mark it allocated on the disk
5066       brelse(bp);
5067       return iget(dev, inum);
5068     }
5069     brelse(bp);
5070   }
5071   panic("ialloc: no inodes");
5072 }
5073
5074 // Copy a modified in-memory inode to disk.
5075 void
5076 iupdate(struct inode *ip)
5077 {
5078   struct buf *bp;
5079   struct dinode *dip;
5080
5081   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5082   dip = (struct dinode*)bp->data + ip->inum%IPB;
5083   dip->type = ip->type;
5084   dip->major = ip->major;
5085   dip->minor = ip->minor;
5086   dip->nlink = ip->nlink;
5087   dip->size = ip->size;
5088   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5089   log_write(bp);
5090   brelse(bp);
5091 }
5092
5093
5094
5095
5096
5097
5098
5099
```

```
5100 // Find the inode with number inum on device dev
5101 // and return the in-memory copy. Does not lock
5102 // the inode and does not read it from disk.
5103 static struct inode*
5104 iget(uint dev, uint inum)
5105 {
5106   struct inode *ip, *empty;
5107
5108   acquire(&icache.lock);
5109
5110   // Is the inode already cached?
5111   empty = 0;
5112   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5113     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5114       ip->ref++;
5115       release(&icache.lock);
5116       return ip;
5117     }
5118     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
5119       empty = ip;
5120   }
5121
5122   // Recycle an inode cache entry.
5123   if(empty == 0)
5124     panic("iget: no inodes");
5125
5126   ip = empty;
5127   ip->dev = dev;
5128   ip->inum = inum;
5129   ip->ref = 1;
5130   ip->flags = 0;
5131   release(&icache.lock);
5132
5133   return ip;
5134 }
5135
5136 // Increment reference count for ip.
5137 // Returns ip to enable ip = idup(ip1) idiom.
5138 struct inode*
5139 idup(struct inode *ip)
5140 {
5141   acquire(&icache.lock);
5142   ip->ref++;
5143   release(&icache.lock);
5144   return ip;
5145 }
5146
5147
5148
5149
```



```
5150 // Lock the given inode.
5151 // Reads the inode from disk if necessary.
5152 void
5153 ilock(struct inode *ip)
5154 {
5155   struct buf *bp;
5156   struct dinode *dip;
5157
5158   if(ip == 0 || ip->ref < 1)
5159     panic("ilock");
5160
5161   acquire(&icache.lock);
5162   while(ip->flags & I_BUSY)
5163     sleep(ip, &icache.lock);
5164   ip->flags |= I_BUSY;
5165   release(&icache.lock);
5166
5167   if(!(ip->flags & I_VALID)){
5168     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5169     dip = (struct dinode*)bp->data + ip->inum%IPB;
5170     ip->type = dip->type;
5171     ip->major = dip->major;
5172     ip->minor = dip->minor;
5173     ip->nlink = dip->nlink;
5174     ip->size = dip->size;
5175     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5176     brelse(bp);
5177     ip->flags |= I_VALID;
5178     if(ip->type == 0)
5179       panic("ilock: no type");
5180   }
5181 }
5182
5183 // Unlock the given inode.
5184 void
5185 iunlock(struct inode *ip)
5186 {
5187   if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
5188     panic("iunlock");
5189
5190   acquire(&icache.lock);
5191   ip->flags &= ~I_BUSY;
5192   wakeup(ip);
5193   release(&icache.lock);
5194 }
5195
5196
5197
5198
5199
```

```
5200 // Drop a reference to an in-memory inode.
5201 // If that was the last reference, the inode cache entry can
5202 // be recycled.
5203 // If that was the last reference and the inode has no links
5204 // to it, free the inode (and its content) on disk.
5205 // All calls to iput() must be inside a transaction in
5206 // case it has to free the inode.
5207 void
5208 iput(struct inode *ip)
5209 {
5210   acquire(&icache.lock);
5211   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
5212     // inode has no links and no other references: truncate and free.
5213     if(ip->flags & I_BUSY)
5214       panic("iput busy");
5215     ip->flags |= I_BUSY;
5216     release(&icache.lock);
5217     itrunc(ip);
5218     ip->type = 0;
5219     iupdate(ip);
5220     acquire(&icache.lock);
5221     ip->flags = 0;
5222     wakeup(ip);
5223   }
5224   ip->ref--;
5225   release(&icache.lock);
5226 }
5227
5228 // Common idiom: unlock, then put.
5229 void
5230 iunlockput(struct inode *ip)
5231 {
5232   iunlock(ip);
5233   iput(ip);
5234 }
5235
5236
5237
5238
5239
5240
5241
5242
5243
5244
5245
5246
5247
5248
5249
```



```
5250 // Inode content
5251 //
5252 // The content (data) associated with each inode is stored
5253 // in blocks on the disk. The first NDIRECT block numbers
5254 // are listed in ip->addrs[].  The next NINDIRECT blocks are
5255 // listed in block ip->addrs[NDIRECT].
5256
5257 // Return the disk block address of the nth block in inode ip.
5258 // If there is no such block, bmap allocates one.
5259 static uint
5260 bmap(struct inode *ip, uint bn)
5261 {
5262   uint addr, *a;
5263   struct buf *bp;
5264
5265   if(bn < NDIRECT){
5266     if((addr = ip->addrs[bn]) == 0)
5267       ip->addrs[bn] = addr = balloc(ip->dev);
5268     return addr;
5269   }
5270   bn -= NDIRECT;
5271
5272   if(bn < NINDIRECT){
5273     // Load indirect block, allocating if necessary.
5274     if((addr = ip->addrs[NDIRECT]) == 0)
5275       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5276     bp = bread(ip->dev, addr);
5277     a = (uint*)bp->data;
5278     if((addr = a[bn]) == 0){
5279       a[bn] = addr = balloc(ip->dev);
5280       log_write(bp);
5281     }
5282     brelse(bp);
5283     return addr;
5284   }
5285
5286   panic("bmap: out of range");
5287 }
5288
5289
5290
5291
5292
5293
5294
5295
5296
5297
5298
5299
```

```
5300 // Truncate inode (discard contents).
5301 // Only called when the inode has no links
5302 // to it (no directory entries referring to it)
5303 // and has no in-memory reference to it (is
5304 // not an open file or current directory).
5305 static void
5306 itrunc(struct inode *ip)
5307 {
5308   int i, j;
5309   struct buf *bp;
5310   uint *a;
5311
5312   for(i = 0; i < NDIRECT; i++){
5313     if(ip->addrs[i]){
5314       bfree(ip->dev, ip->addrs[i]);
5315       ip->addrs[i] = 0;
5316     }
5317   }
5318
5319   if(ip->addrs[NDIRECT]){
5320     bp = bread(ip->dev, ip->addrs[NDIRECT]);
5321     a = (uint*)bp->data;
5322     for(j = 0; j < NINDIRECT; j++){
5323       if(a[j])
5324         bfree(ip->dev, a[j]);
5325     }
5326     brelse(bp);
5327     bfree(ip->dev, ip->addrs[NDIRECT]);
5328     ip->addrs[NDIRECT] = 0;
5329   }
5330
5331   ip->size = 0;
5332   iupdate(ip);
5333 }
5334
5335 // Copy stat information from inode.
5336 void
5337 stati(struct inode *ip, struct stat *st)
5338 {
5339   st->dev = ip->dev;
5340   st->ino = ip->inum;
5341   st->type = ip->type;
5342   st->nlink = ip->nlink;
5343   st->size = ip->size;
5344 }
5345
5346
5347
5348
5349
```



```
5350 // Read data from inode.
5351 int
5352 readi(struct inode *ip, char *dst, uint off, uint n)
5353 {
5354   uint tot, m;
5355   struct buf *bp;
5356
5357   if(ip->type == T_DEV){
5358     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
5359       return -1;
5360     return devsw[ip->major].read(ip, dst, n);
5361   }
5362
5363   if(off > ip->size || off + n < off)
5364     return -1;
5365   if(off + n > ip->size)
5366     n = ip->size - off;
5367
5368   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5369     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5370     m = min(n - tot, BSIZE - off%BSIZE);
5371     memmove(dst, bp->data + off%BSIZE, m);
5372     brelse(bp);
5373   }
5374   return n;
5375 }
5376
5377
5378
5379
5380
5381
5382
5383
5384
5385
5386
5387
5388
5389
5390
5391
5392
5393
5394
5395
5396
5397
5398
5399
```

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



```
5450 // Directories
5451
5452 int
5453 namecmp(const char *s, const char *t)
5454 {
5455   return strncmp(s, t, DIRSIZ);
5456 }
5457
5458 // Look for a directory entry in a directory.
5459 // If found, set *poff to byte offset of entry.
5460 struct inode*
5461 dirlookup(struct inode *dp, char *name, uint *poff)
5462 {
5463   uint off, inum;
5464   struct dirent de;
5465
5466   if(dp->type != T_DIR)
5467     panic("dirlookup not DIR");
5468
5469   for(off = 0; off < dp->size; off += sizeof(de)){
5470     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5471       panic("dirlink read");
5472     if(de.inum == 0)
5473       continue;
5474     if(namecmp(name, de.name) == 0){
5475       // entry matches path element
5476       if(poff)
5477         *poff = off;
5478       inum = de.inum;
5479       return iget(dp->dev, inum);
5480     }
5481   }
5482
5483   return 0;
5484 }
5485
5486
5487
5488
5489
5490
5491
5492
5493
5494
5495
5496
5497
5498
5499
```

```
5500 // Write a new directory entry (name, inum) into the directory dp.
5501 int
5502 dirlink(struct inode *dp, char *name, uint inum)
5503 {
5504   int off;
5505   struct dirent de;
5506   struct inode *ip;
5507
5508   // Check that name is not present.
5509   if((ip = dirlookup(dp, name, 0)) != 0){
5510     iput(ip);
5511     return -1;
5512   }
5513
5514   // Look for an empty dirent.
5515   for(off = 0; off < dp->size; off += sizeof(de)){
5516     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5517       panic("dirlink read");
5518     if(de.inum == 0)
5519       break;
5520   }
5521
5522   strncpy(de.name, name, DIRSIZ);
5523   de.inum = inum;
5524   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5525     panic("dirlink");
5526
5527   return 0;
5528 }
5529
5530
5531
5532
5533
5534
5535
5536
5537
5538
5539
5540
5541
5542
5543
5544
5545
5546
5547
5548
5549
```



```
5550 // Paths
5551
5552 // Copy the next path element from path into name.
5553 // Return a pointer to the element following the copied one.
5554 // The returned path has no leading slashes,
5555 // so the caller can check *path=='\0' to see if the name is the last one.
5556 // If no name to remove, return 0.
5557 //
5558 // Examples:
5559 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5560 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5561 //   skipelem("a", name) = "", setting name = "a"
5562 //   skipelem("", name) = skipelem("////", name) = 0
5563 //
5564 static char*
5565 skipelem(char *path, char *name)
5566 {
5567   char *s;
5568   int len;
5569
5570   while(*path == '/')
5571     path++;
5572   if(*path == 0)
5573     return 0;
5574   s = path;
5575   while(*path != '/' && *path != 0)
5576     path++;
5577   len = path - s;
5578   if(len >= DIRSIZ)
5579     memmove(name, s, DIRSIZ);
5580   else {
5581     memmove(name, s, len);
5582     name[len] = 0;
5583   }
5584   while(*path == '/')
5585     path++;
5586   return path;
5587 }
5588
5589
5590
5591
5592
5593
5594
5595
5596
5597
5598
5599
```

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



```
5650 struct inode*
5651 nameiparent(char *path, char *name)
5652 {
5653   return namex(path, 1, name);
5654 }
5655
5656
5657
5658
5659
5660
5661
5662
5663
5664
5665
5666
5667
5668
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
5834
5835
5836
5837
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```



```
5850 // Write to file f.
5851 int
5852 filewrite(struct file *f, char *addr, int n)
5853 {
5854   int r;
5855
5856   if(f->writable == 0)
5857     return -1;
5858   if(f->type == FD_PIPE)
5859     return pipewrite(f->pipe, addr, n);
5860   if(f->type == FD_INODE){
5861     // write a few blocks at a time to avoid exceeding
5862     // the maximum log transaction size, including
5863     // i-node, indirect block, allocation blocks,
5864     // and 2 blocks of slop for non-aligned writes.
5865     // this really belongs lower down, since writei()
5866     // might be writing a device like the console.
5867     int max = ((LOGSIZE-1-1-2) / 2) * 512;
5868     int i = 0;
5869     while(i < n){
5870       int n1 = n - i;
5871       if(n1 > max)
5872         n1 = max;
5873
5874       begin_op();
5875       ilock(f->ip);
5876       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
5877         f->off += r;
5878       iunlock(f->ip);
5879       end_op();
5880
5881       if(r < 0)
5882         break;
5883       if(r != n1)
5884         panic("short filewrite");
5885       i += r;
5886     }
5887     return i == n ? n : -1;
5888   }
5889   panic("filewrite");
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
6078
6079
6080
6081
6082
6083
6084
6085
6086
6087
6088
6089
6090
6091
6092
6093
6094
6095
6096
6097
6098
6099
```

```
6100 int
6101 sys_unlink(void)
6102 {
6103   struct inode *ip, *dp;
6104   struct dirent de;
6105   char name[DIRSIZ], *path;
6106   uint off;
6107
6108   if(argstr(0, &path) < 0)
6109     return -1;
6110
6111   begin_op();
6112   if((dp = nameiparent(path, name)) == 0){
6113     end_op();
6114     return -1;
6115   }
6116
6117   ilock(dp);
6118
6119   // Cannot unlink "." or "..".
6120   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6121     goto bad;
6122
6123   if((ip = dirlookup(dp, name, &off)) == 0)
6124     goto bad;
6125   ilock(ip);
6126
6127   if(ip->nlink < 1)
6128     panic("unlink: nlink < 1");
6129   if(ip->type == T_DIR && !isdirempty(ip)){
6130     iunlockput(ip);
6131     goto bad;
6132   }
6133
6134   memset(&de, 0, sizeof(de));
6135   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6136     panic("unlink: writei");
6137   if(ip->type == T_DIR){
6138     dp->nlink--;
6139     iupdate(dp);
6140   }
6141   iunlockput(dp);
6142
6143   ip->nlink--;
6144   iupdate(ip);
6145   iunlockput(ip);
6146
6147   end_op();
6148
6149   return 0;
```



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

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



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

```
6300 int
6301 sys_chdir(void)
6302 {
6303   char *path;
6304   struct inode *ip;
6305
6306   begin_op();
6307   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
6308     end_op();
6309     return -1;
6310   }
6311   ilock(ip);
6312   if(ip->type != T_DIR){
6313     iunlockput(ip);
6314     end_op();
6315     return -1;
6316   }
6317   iunlock(ip);
6318   iput(proc->cwd);
6319   end_op();
6320   proc->cwd = ip;
6321   return 0;
6322 }
6323
6324 int
6325 sys_exec(void)
6326 {
6327   char *path, *argv[MAXARG];
6328   int i;
6329   uint uargv, uarg;
6330
6331   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6332     return -1;
6333   }
6334   memset(argv, 0, sizeof(argv));
6335   for(i=0;; i++){
6336     if(i >= NELEM(argv))
6337       return -1;
6338     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6339       return -1;
6340     if(uarg == 0){
6341       argv[i] = 0;
6342       break;
6343     }
6344     if(fetchstr(uarg, &argv[i]) < 0)
6345       return -1;
6346   }
6347   return exec(path, argv);
6348 }
6349
```



```
6350 int
6351 sys_pipe(void)
6352 {
6353   int *fd;
6354   struct file *rf, *wf;
6355   int fd0, fd1;
6356
6357   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6358     return -1;
6359   if(pipealloc(&rf, &wf) < 0)
6360     return -1;
6361   fd0 = -1;
6362   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6363     if(fd0 >= 0)
6364       proc->ofile[fd0] = 0;
6365     fileclose(rf);
6366     fileclose(wf);
6367     return -1;
6368   }
6369   fd[0] = fd0;
6370   fd[1] = fd1;
6371   return 0;
6372 }
6373
6374
6375
6376
6377
6378
6379
6380
6381
6382
6383
6384
6385
6386
6387
6388
6389
6390
6391
6392
6393
6394
6395
6396
6397
6398
6399
```

```
6400 #include "types.h"
6401 #include "param.h"
6402 #include "memlayout.h"
6403 #include "mmu.h"
6404 #include "proc.h"
6405 #include "defs.h"
6406 #include "x86.h"
6407 #include "elf.h"
6408
6409 int
6410 exec(char *path, char **argv)
6411 {
6412   char *s, *last;
6413   int i, off;
6414   uint argc, sz, sp, ustack[3+MAXARG+1];
6415   struct elfhdr elf;
6416   struct inode *ip;
6417   struct proghdr ph;
6418   pde_t *pgdir, *oldpgdir;
6419
6420   begin_op();
6421   if((ip = namei(path)) == 0){
6422     end_op();
6423     return -1;
6424   }
6425   ilock(ip);
6426   pgdir = 0;
6427
6428   // Check ELF header
6429   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
6430     goto bad;
6431   if(elf.magic != ELF_MAGIC)
6432     goto bad;
6433
6434   if((pgdir = setupkvm()) == 0)
6435     goto bad;
6436
6437   // Load program into memory.
6438   sz = 0;
6439   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6440     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6441       goto bad;
6442     if(ph.type != ELF_PROG_LOAD)
6443       continue;
6444     if(ph.memsz < ph.filesz)
6445       goto bad;
6446     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6447       goto bad;
6448     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6449       goto bad;
```



```
6450   }
6451   iunlockput(ip);
6452   end_op();
6453   ip = 0;
6454
6455   // Allocate two pages at the next page boundary.
6456   // Make the first inaccessible.  Use the second as the user stack.
6457   sz = PGROUNDUP(sz);
6458   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
6459     goto bad;
6460   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6461   sp = sz;
6462
6463   // Push argument strings, prepare rest of stack in ustack.
6464   for(argc = 0; argv[argc]; argc++) {
6465     if(argc >= MAXARG)
6466       goto bad;
6467     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6468     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6469       goto bad;
6470     ustack[3+argc] = sp;
6471   }
6472   ustack[3+argc] = 0;
6473
6474   ustack[0] = 0xffffffff;  // fake return PC
6475   ustack[1] = argc;
6476   ustack[2] = sp - (argc+1)*4;  // argv pointer
6477
6478   sp -= (3+argc+1) * 4;
6479   if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6480     goto bad;
6481
6482   // Save program name for debugging.
6483   for(last=s=path; *s; s++)
6484     if(*s == '/')
6485       last = s+1;
6486   safestrcpy(proc->name, last, sizeof(proc->name));
6487
6488   // Commit to the user image.
6489   oldpgdir = proc->pgdir;
6490   proc->pgdir = pgdir;
6491   proc->sz = sz;
6492   proc->tf->eip = elf.entry;  // main
6493   proc->tf->esp = sp;
6494   switchuvm(proc);
6495   freevm(oldpgdir);
6496   return 0;
6497
6498
6499
```

```
6500  bad:
6501    if(pgdir)
6502      freevm(pgdir);
6503    if(ip){
6504      iunlockput(ip);
6505      end_op();
6506    }
6507    return -1;
6508 }
6509
6510
6511
6512
6513
6514
6515
6516
6517
6518
6519
6520
6521
6522
6523
6524
6525
6526
6527
6528
6529
6530
6531
6532
6533
6534
6535
6536
6537
6538
6539
6540
6541
6542
6543
6544
6545
6546
6547
6548
6549
```



```
6550 #include "types.h"
6551 #include "defs.h"
6552 #include "param.h"
6553 #include "mmu.h"
6554 #include "proc.h"
6555 #include "fs.h"
6556 #include "file.h"
6557 #include "spinlock.h"
6558
6559 #define PIPESIZE 512
6560
6561 struct pipe {
6562   struct spinlock lock;
6563   char data[PIPESIZE];
6564   uint nread;     // number of bytes read
6565   uint nwrite;    // number of bytes written
6566   int readopen;   // read fd is still open
6567   int writeopen;  // write fd is still open
6568 };
6569
6570 int
6571 pipealloc(struct file **f0, struct file **f1)
6572 {
6573   struct pipe *p;
6574
6575   p = 0;
6576   *f0 = *f1 = 0;
6577   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
6578     goto bad;
6579   if((p = (struct pipe*)kalloc()) == 0)
6580     goto bad;
6581   p->readopen = 1;
6582   p->writeopen = 1;
6583   p->nwrite = 0;
6584   p->nread = 0;
6585   initlock(&p->lock, "pipe");
6586   (*f0)->type = FD_PIPE;
6587   (*f0)->readable = 1;
6588   (*f0)->writable = 0;
6589   (*f0)->pipe = p;
6590   (*f1)->type = FD_PIPE;
6591   (*f1)->readable = 0;
6592   (*f1)->writable = 1;
6593   (*f1)->pipe = p;
6594   return 0;
6595
6596
6597
6598
6599
```

```
 bad:
  if(p)
    kfree((char*)p);
  if(*f0)
    fileclose(*f0);
  if(*f1)
    fileclose(*f1);
  return -1;
}

void
pipeclose(struct pipe *p, int writable)
{
  acquire(&p->lock);
  if(writable){
    p->writeopen = 0;
    wakeup(&p->nread);
  } else {
    p->readopen = 0;
    wakeup(&p->nwrite);
  }
  if(p->readopen == 0 && p->writeopen == 0){
    release(&p->lock);
    kfree((char*)p);
  } else
    release(&p->lock);
}


int
pipewrite(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  for(i = 0; i < n; i++){
    while(p->nwrite == p->nread + PIPESIZE){
      if(p->readopen == 0 || proc->killed){
        release(&p->lock);
        return -1;
      }
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
  wakeup(&p->nread);
  release(&p->lock);
  return n;
}
```



```
int
piperead(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  while(p->nread == p->nwrite && p->writeopen){
    if(proc->killed){
      release(&p->lock);
      return -1;
    }
    sleep(&p->nread, &p->lock);
  }
  for(i = 0; i < n; i++){
    if(p->nread == p->nwrite)
      break;
    addr[i] = p->data[p->nread++ % PIPESIZE];
  }
  wakeup(&p->nwrite);
  release(&p->lock);
  return i;
}
```

```
6700 #include "types.h"
6701 #include "x86.h"
6702
6703 void*
6704 memset(void *dst, int c, uint n)
6705 {
6706   if ((int)dst%4 == 0 && n%4 == 0){
6707     c &= 0xFF;
6708     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
6709   } else
6710     stosb(dst, c, n);
6711   return dst;
6712 }
6713
6714 int
6715 memcmp(const void *v1, const void *v2, uint n)
6716 {
6717   const uchar *s1, *s2;
6718
6719   s1 = v1;
6720   s2 = v2;
6721   while(n-- > 0){
6722     if(*s1 != *s2)
6723       return *s1 - *s2;
6724     s1++, s2++;
6725   }
6726
6727   return 0;
6728 }
6729
6730 void*
6731 memmove(void *dst, const void *src, uint n)
6732 {
6733   const char *s;
6734   char *d;
6735
6736   s = src;
6737   d = dst;
6738   if(s < d && s + n > d){
6739     s += n;
6740     d += n;
6741     while(n-- > 0)
6742       *--d = *--s;
6743   } else
6744     while(n-- > 0)
6745       *d++ = *s++;
6746
6747   return dst;
6748 }
6749
```



```
6750 // memcpy exists to placate GCC.  Use memmove.
6751 void*
6752 memcpy(void *dst, const void *src, uint n)
6753 {
6754   return memmove(dst, src, n);
6755 }
6756
6757 int
6758 strncmp(const char *p, const char *q, uint n)
6759 {
6760   while(n > 0 && *p && *p == *q)
6761     n--, p++, q++;
6762   if(n == 0)
6763     return 0;
6764   return (uchar)*p - (uchar)*q;
6765 }
6766
6767 char*
6768 strncpy(char *s, const char *t, int n)
6769 {
6770   char *os;
6771
6772   os = s;
6773   while(n-- > 0 && (*s++ = *t++) != 0)
6774     ;
6775   while(n-- > 0)
6776     *s++ = 0;
6777   return os;
6778 }
6779
6780 // Like strncpy but guaranteed to NUL-terminate.
6781 char*
6782 safestrcpy(char *s, const char *t, int n)
6783 {
6784   char *os;
6785
6786   os = s;
6787   if(n <= 0)
6788     return os;
6789   while(--n > 0 && (*s++ = *t++) != 0)
6790     ;
6791   *s = 0;
6792   return os;
6793 }
6794
6795
6796
6797
6798
6799
```

```
6800 int
6801 strlen(const char *s)
6802 {
6803   int n;
6804
6805   for(n = 0; s[n]; n++)
6806     ;
6807   return n;
6808 }
6809
6810
6811
6812
6813
6814
6815
6816
6817
6818
6819
6820
6821
6822
6823
6824
6825
6826
6827
6828
6829
6830
6831
6832
6833
6834
6835
6836
6837
6838
6839
6840
6841
6842
6843
6844
6845
6846
6847
6848
6849
```



```
6850 // See MultiProcessor Specification Version 1.[14]
6851
6852 struct mp {             // floating pointer
6853   uchar signature[4];           // "_MP_"
6854   void *physaddr;               // phys addr of MP config table
6855   uchar length;                 // 1
6856   uchar specrev;                // [14]
6857   uchar checksum;               // all bytes must add up to 0
6858   uchar type;                   // MP system config type
6859   uchar imcrp;
6860   uchar reserved[3];
6861 };
6862
6863 struct mpconf {         // configuration table header
6864   uchar signature[4];           // "PCMP"
6865   ushort length;                // total table length
6866   uchar version;                // [14]
6867   uchar checksum;               // all bytes must add up to 0
6868   uchar product[20];            // product id
6869   uint *oemtable;               // OEM table pointer
6870   ushort oemlength;             // OEM table length
6871   ushort entry;                 // entry count
6872   uint *lapicaddr;              // address of local APIC
6873   ushort xlength;               // extended table length
6874   uchar xchecksum;              // extended table checksum
6875   uchar reserved;
6876 };
6877
6878 struct mpproc {         // processor table entry
6879   uchar type;                   // entry type (0)
6880   uchar apicid;                 // local APIC id
6881   uchar version;                // local APIC verison
6882   uchar flags;                  // CPU flags
6883     #define MPBOOT 0x02           // This proc is the bootstrap processor.
6884   uchar signature[4];           // CPU signature
6885   uint feature;                 // feature flags from CPUID instruction
6886   uchar reserved[8];
6887 };
6888
6889 struct mpioapic {       // I/O APIC table entry
6890   uchar type;                   // entry type (2)
6891   uchar apicno;                 // I/O APIC id
6892   uchar version;                // I/O APIC version
6893   uchar flags;                  // I/O APIC flags
6894   uint *addr;                  // I/O APIC address
6895 };
6896
6897
6898
6899
```

```
6900 // Table entry types
6901 #define MPPROC    0x00  // One per processor
6902 #define MPBUS     0x01  // One per bus
6903 #define MPIOAPIC  0x02  // One per I/O APIC
6904 #define MPIOINTR  0x03  // One per bus interrupt source
6905 #define MPLINTR   0x04  // One per system interrupt source
6906
6907
6908
6909
6910
6911
6912
6913
6914
6915
6916
6917
6918
6919
6920
6921
6922
6923
6924
6925
6926
6927
6928
6929
6930
6931
6932
6933
6934
6935
6936
6937
6938
6939
6940
6941
6942
6943
6944
6945
6946
6947
6948
6949
```



```
6950 // Blank page.
6951
6952
6953
6954
6955
6956
6957
6958
6959
6960
6961
6962
6963
6964
6965
6966
6967
6968
6969
6970
6971
6972
6973
6974
6975
6976
6977
6978
6979
6980
6981
6982
6983
6984
6985
6986
6987
6988
6989
6990
6991
6992
6993
6994
6995
6996
6997
6998
6999
```

```
7000 // Multiprocessor support
7001 // Search memory for MP description structures.
7002 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
7003
7004 #include "types.h"
7005 #include "defs.h"
7006 #include "param.h"
7007 #include "memlayout.h"
7008 #include "mp.h"
7009 #include "x86.h"
7010 #include "mmu.h"
7011 #include "proc.h"
7012
7013 struct cpu cpus[NCPU];
7014 static struct cpu *bcpu;
7015 int ismp;
7016 int ncpu;
7017 uchar ioapicid;
7018
7019 int
7020 mpbcpu(void)
7021 {
7022   return bcpu-cpus;
7023 }
7024
7025 static uchar
7026 sum(uchar *addr, int len)
7027 {
7028   int i, sum;
7029
7030   sum = 0;
7031   for(i=0; i<len; i++)
7032     sum += addr[i];
7033   return sum;
7034 }
7035
7036 // Look for an MP structure in the len bytes at addr.
7037 static struct mp*
7038 mpsearch1(uint a, int len)
7039 {
7040   uchar *e, *p, *addr;
7041
7042   addr = p2v(a);
7043   e = addr+len;
7044   for(p = addr; p < e; p += sizeof(struct mp))
7045     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
7046       return (struct mp*)p;
7047   return 0;
7048 }
7049
```



```
7050 // Search for the MP Floating Pointer Structure, which according to the
7051 // spec is in one of the following three locations:
7052 // 1) in the first KB of the EBDA;
7053 // 2) in the last KB of system base memory;
7054 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
7055 static struct mp*
7056 mpsearch(void)
7057 {
7058   uchar *bda;
7059   uint p;
7060   struct mp *mp;
7061
7062   bda = (uchar *) P2V(0x400);
7063   if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
7064     if((mp = mpsearch1(p, 1024)))
7065       return mp;
7066   } else {
7067     p = ((bda[0x14]<<8)|bda[0x13])*1024;
7068     if((mp = mpsearch1(p-1024, 1024)))
7069       return mp;
7070   }
7071   return mpsearch1(0xF0000, 0x10000);
7072 }
7073
7074 // Search for an MP configuration table.  For now,
7075 // don't accept the default configurations (physaddr == 0).
7076 // Check for correct signature, calculate the checksum and,
7077 // if correct, check the version.
7078 // To do: check extended table checksum.
7079 static struct mpconf*
7080 mpconfig(struct mp **pmp)
7081 {
7082   struct mpconf *conf;
7083   struct mp *mp;
7084
7085   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
7086     return 0;
7087   conf = (struct mpconf*) p2v((uint) mp->physaddr);
7088   if(memcmp(conf, "PCMP", 4) != 0)
7089     return 0;
7090   if(conf->version != 1 && conf->version != 4)
7091     return 0;
7092   if(sum((uchar*)conf, conf->length) != 0)
7093     return 0;
7094   *pmp = mp;
7095   return conf;
7096 }
7097
7098
7099
```

```
7100 void
7101 mpinit(void)
7102 {
7103   uchar *p, *e;
7104   struct mp *mp;
7105   struct mpconf *conf;
7106   struct mpproc *proc;
7107   struct mpioapic *ioapic;
7108
7109   bcpu = &cpus[0];
7110   if((conf = mpconfig(&mp)) == 0)
7111     return;
7112   ismp = 1;
7113   lapic = (uint*)conf->lapicaddr;
7114   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
7115     switch(*p){
7116     case MPPROC:
7117       proc = (struct mpproc*)p;
7118       if(ncpu != proc->apicid){
7119         cprintf("mpinit: ncpu=%d apicid=%d\n", ncpu, proc->apicid);
7120         ismp = 0;
7121       }
7122       if(proc->flags & MPBOOT)
7123         bcpu = &cpus[ncpu];
7124       cpus[ncpu].id = ncpu;
7125       ncpu++;
7126       p += sizeof(struct mpproc);
7127       continue;
7128     case MPIOAPIC:
7129       ioapic = (struct mpioapic*)p;
7130       ioapicid = ioapic->apicno;
7131       p += sizeof(struct mpioapic);
7132       continue;
7133     case MPBUS:
7134     case MPIOINTR:
7135     case MPLINTR:
7136       p += 8;
7137       continue;
7138     default:
7139       cprintf("mpinit: unknown config type %x\n", *p);
7140       ismp = 0;
7141     }
7142   }
7143   if(!ismp){
7144     // Didn't like what we found; fall back to no MP.
7145     ncpu = 1;
7146     lapic = 0;
7147     ioapicid = 0;
7148     return;
7149   }
```



```
7150   if(mp->imcrp){
7151     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
7152     // But it would on real hardware.
7153     outb(0x22, 0x70);   // Select IMCR
7154     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
7155   }
7156 }
7157
7158
7159
7160
7161
7162
7163
7164
7165
7166
7167
7168
7169
7170
7171
7172
7173
7174
7175
7176
7177
7178
7179
7180
7181
7182
7183
7184
7185
7186
7187
7188
7189
7190
7191
7192
7193
7194
7195
7196
7197
7198
7199
```

```
7200 // The local APIC manages internal (non-I/O) interrupts.
7201 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
7202
7203 #include "types.h"
7204 #include "defs.h"
7205 #include "date.h"
7206 #include "memlayout.h"
7207 #include "traps.h"
7208 #include "mmu.h"
7209 #include "x86.h"
7210
7211 // Local APIC registers, divided by 4 for use as uint[] indices.
7212 #define ID      (0x0020/4)   // ID
7213 #define VER     (0x0030/4)   // Version
7214 #define TPR     (0x0080/4)   // Task Priority
7215 #define EOI     (0x00B0/4)   // EOI
7216 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
7217   #define ENABLE     0x00000100   // Unit Enable
7218 #define ESR     (0x0280/4)   // Error Status
7219 #define ICRLO   (0x0300/4)   // Interrupt Command
7220   #define INIT       0x00000500   // INIT/RESET
7221   #define STARTUP    0x00000600   // Startup IPI
7222   #define DELIVS     0x00001000   // Delivery status
7223   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
7224   #define DEASSERT   0x00000000
7225   #define LEVEL      0x00008000   // Level triggered
7226   #define BCAST      0x00080000   // Send to all APICs, including self.
7227   #define BUSY       0x00001000
7228   #define FIXED      0x00000000
7229 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
7230 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
7231   #define X1         0x0000000B   // divide counts by 1
7232   #define PERIODIC   0x00020000   // Periodic
7233 #define PCINT   (0x0340/4)   // Performance Counter LVT
7234 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
7235 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
7236 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
7237   #define MASKED     0x00010000   // Interrupt masked
7238 #define TICR    (0x0380/4)   // Timer Initial Count
7239 #define TCCR    (0x0390/4)   // Timer Current Count
7240 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
7241
7242 volatile uint *lapic;  // Initialized in mp.c
7243
7244 static void
7245 lapicw(int index, int value)
7246 {
7247   lapic[index] = value;
7248   lapic[ID];  // wait for write to finish, by reading
7249 }
```

```
7300 void
7301 lapicinit(void)
7302 {
7303   if(!lapic)
7304     return;
7305
7306   // Enable local APIC; set spurious interrupt vector.
7307   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
7308
7309   // The timer repeatedly counts down at bus frequency
7310   // from lapic[TICR] and then issues an interrupt.
7311   // If xv6 cared more about precise timekeeping,
7312   // TICR would be calibrated using an external time source.
7313   lapicw(TDCR, X1);
7314   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
7315   lapicw(TICR, 10000000);
7316
7317   // Disable logical interrupt lines.
7318   lapicw(LINT0, MASKED);
7319   lapicw(LINT1, MASKED);
7320
7321   // Disable performance counter overflow interrupts
7322   // on machines that provide that interrupt entry.
7323   if(((lapic[VER]>>16) & 0xFF) >= 4)
7324     lapicw(PCINT, MASKED);
7325
7326   // Map error interrupt to IRQ_ERROR.
7327   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
7328
7329   // Clear error status register (requires back-to-back writes).
7330   lapicw(ESR, 0);
7331   lapicw(ESR, 0);
7332
7333   // Ack any outstanding interrupts.
7334   lapicw(EOI, 0);
7335
7336   // Send an Init Level De-Assert to synchronise arbitration ID's.
7337   lapicw(ICRHI, 0);
7338   lapicw(ICRLO, BCAST | INIT | LEVEL);
7339   while(lapic[ICRLO] & DELIVS)
7340     ;
7341
7342   // Enable interrupts on the APIC (but not on the processor).
7343   lapicw(TPR, 0);
7344 }
7345
7346
7347
7348
7349
```



```
7350 int
7351 cpunum(void)
7352 {
7353   // Cannot call cpu when interrupts are enabled:
7354   // result not guaranteed to last long enough to be used!
7355   // Would prefer to panic but even printing is chancy here:
7356   // almost everything, including cprintf and panic, calls cpu,
7357   // often indirectly through acquire and release.
7358   if(readeflags()&FL_IF){
7359     static int n;
7360     if(n++ == 0)
7361       cprintf("cpu called from %x with interrupts enabled\n",
7362         __builtin_return_address(0));
7363   }
7364
7365   if(lapic)
7366     return lapic[ID]>>24;
7367   return 0;
7368 }
7369
7370 // Acknowledge interrupt.
7371 void
7372 lapiceoi(void)
7373 {
7374   if(lapic)
7375     lapicw(EOI, 0);
7376 }
7377
7378 // Spin for a given number of microseconds.
7379 // On real hardware would want to tune this dynamically.
7380 void
7381 microdelay(int us)
7382 {
7383 }
7384
7385 #define CMOS_PORT    0x70
7386 #define CMOS_RETURN  0x71
7387
7388 // Start additional processor running entry code at addr.
7389 // See Appendix B of MultiProcessor Specification.
7390 void
7391 lapicstartap(uchar apicid, uint addr)
7392 {
7393   int i;
7394   ushort *wrv;
7395
7396   // "The BSP must initialize CMOS shutdown code to 0AH
7397   // and the warm reset vector (DWORD based at 40:67) to point at
7398   // the AP startup code prior to the [universal startup algorithm]."
7399   outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
```

```
7400   outb(CMOS_PORT+1, 0x0A);
7401   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
7402   wrv[0] = 0;
7403   wrv[1] = addr >> 4;
7404
7405   // "Universal startup algorithm."
7406   // Send INIT (level-triggered) interrupt to reset other CPU.
7407   lapicw(ICRHI, apicid<<24);
7408   lapicw(ICRLO, INIT | LEVEL | ASSERT);
7409   microdelay(200);
7410   lapicw(ICRLO, INIT | LEVEL);
7411   microdelay(100);    // should be 10ms, but too slow in Bochs!
7412
7413   // Send startup IPI (twice!) to enter code.
7414   // Regular hardware is supposed to only accept a STARTUP
7415   // when it is in the halted state due to an INIT.  So the second
7416   // should be ignored, but it is part of the official Intel algorithm.
7417   // Bochs complains about the second one.  Too bad for Bochs.
7418   for(i = 0; i < 2; i++){
7419     lapicw(ICRHI, apicid<<24);
7420     lapicw(ICRLO, STARTUP | (addr>>12));
7421     microdelay(200);
7422   }
7423 }
7424
7425 #define CMOS_STATA   0x0a
7426 #define CMOS_STATB   0x0b
7427 #define CMOS_UIP    (1 << 7)        // RTC update in progress
7428
7429 #define SECS    0x00
7430 #define MINS    0x02
7431 #define HOURS   0x04
7432 #define DAY     0x07
7433 #define MONTH   0x08
7434 #define YEAR    0x09
7435
7436 static uint cmos_read(uint reg)
7437 {
7438   outb(CMOS_PORT,  reg);
7439   microdelay(200);
7440
7441   return inb(CMOS_RETURN);
7442 }
7443
7444
7445
7446
7447
7448
7449
```



```
7450 static void fill_rtcdate(struct rtcdate *r)
7451 {
7452   r->second = cmos_read(SECS);
7453   r->minute = cmos_read(MINS);
7454   r->hour   = cmos_read(HOURS);
7455   r->day    = cmos_read(DAY);
7456   r->month  = cmos_read(MONTH);
7457   r->year   = cmos_read(YEAR);
7458 }
7459
7460 // qemu seems to use 24-hour GWT and the values are BCD encoded
7461 void cmostime(struct rtcdate *r)
7462 {
7463   struct rtcdate t1, t2;
7464   int sb, bcd;
7465
7466   sb = cmos_read(CMOS_STATB);
7467
7468   bcd = (sb & (1 << 2)) == 0;
7469
7470   // make sure CMOS doesn't modify time while we read it
7471   for (;;) {
7472     fill_rtcdate(&t1);
7473     if (cmos_read(CMOS_STATA) & CMOS_UIP)
7474         continue;
7475     fill_rtcdate(&t2);
7476     if (memcmp(&t1, &t2, sizeof(t1)) == 0)
7477       break;
7478   }
7479
7480   // convert
7481   if (bcd) {
7482 #define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
7483     CONV(second);
7484     CONV(minute);
7485     CONV(hour  );
7486     CONV(day   );
7487     CONV(month );
7488     CONV(year  );
7489 #undef     CONV
7490   }
7491
7492   *r = t1;
7493   r->year += 2000;
7494 }
7495
7496
7497
7498
7499
```

```
7500 // The I/O APIC manages hardware interrupts for an SMP system.
7501 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
7502 // See also picirq.c.
7503
7504 #include "types.h"
7505 #include "defs.h"
7506 #include "traps.h"
7507
7508 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
7509
7510 #define REG_ID     0x00  // Register index: ID
7511 #define REG_VER    0x01  // Register index: version
7512 #define REG_TABLE  0x10  // Redirection table base
7513
7514 // The redirection table starts at REG_TABLE and uses
7515 // two registers to configure each interrupt.
7516 // The first (low) register in a pair contains configuration bits.
7517 // The second (high) register contains a bitmask telling which
7518 // CPUs can serve that interrupt.
7519 #define INT_DISABLED   0x00010000  // Interrupt disabled
7520 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
7521 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
7522 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
7523
7524 volatile struct ioapic *ioapic;
7525
7526 // IO APIC MMIO structure: write reg, then read or write data.
7527 struct ioapic {
7528   uint reg;
7529   uint pad[3];
7530   uint data;
7531 };
7532
7533 static uint
7534 ioapicread(int reg)
7535 {
7536   ioapic->reg = reg;
7537   return ioapic->data;
7538 }
7539
7540 static void
7541 ioapicwrite(int reg, uint data)
7542 {
7543   ioapic->reg = reg;
7544   ioapic->data = data;
7545 }
7546
7547
7548
7549
```



```
7550 void
7551 ioapicinit(void)
7552 {
7553   int i, id, maxintr;
7554
7555   if(!ismp)
7556     return;
7557
7558   ioapic = (volatile struct ioapic*)IOAPIC;
7559   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
7560   id = ioapicread(REG_ID) >> 24;
7561   if(id != ioapicid)
7562     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
7563
7564   // Mark all interrupts edge-triggered, active high, disabled,
7565   // and not routed to any CPUs.
7566   for(i = 0; i <= maxintr; i++){
7567     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
7568     ioapicwrite(REG_TABLE+2*i+1, 0);
7569   }
7570 }
7571
7572 void
7573 ioapicenable(int irq, int cpunum)
7574 {
7575   if(!ismp)
7576     return;
7577
7578   // Mark interrupt edge-triggered, active high,
7579   // enabled, and routed to the given cpunum,
7580   // which happens to be that cpu's APIC ID.
7581   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
7582   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
7583 }
7584
7585
7586
7587
7588
7589
7590
7591
7592
7593
7594
7595
7596
7597
7598
7599
```

```
7600 // Intel 8259A programmable interrupt controllers.
7601
7602 #include "types.h"
7603 #include "x86.h"
7604 #include "traps.h"
7605
7606 // I/O Addresses of the two programmable interrupt controllers
7607 #define IO_PIC1         0x20    // Master (IRQs 0-7)
7608 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
7609
7610 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
7611
7612 // Current IRQ mask.
7613 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
7614 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
7615
7616 static void
7617 picsetmask(ushort mask)
7618 {
7619   irqmask = mask;
7620   outb(IO_PIC1+1, mask);
7621   outb(IO_PIC2+1, mask >> 8);
7622 }
7623
7624 void
7625 picenable(int irq)
7626 {
7627   picsetmask(irqmask & ~(1<<irq));
7628 }
7629
7630 // Initialize the 8259A interrupt controllers.
7631 void
7632 picinit(void)
7633 {
7634   // mask all interrupts
7635   outb(IO_PIC1+1, 0xFF);
7636   outb(IO_PIC2+1, 0xFF);
7637
7638   // Set up master (8259A-1)
7639
7640   // ICW1:  0001g0hi
7641   //    g:  0 = edge triggering, 1 = level triggering
7642   //    h:  0 = cascaded PICs, 1 = master only
7643   //    i:  0 = no ICW4, 1 = ICW4 required
7644   outb(IO_PIC1, 0x11);
7645
7646   // ICW2:  Vector offset
7647   outb(IO_PIC1+1, T_IRQ0);
7648
7649
```



```
7650   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
7651   //        (slave PIC) 3-bit # of slave's connection to master
7652   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
7653
7654   // ICW4:  000nbmap
7655   //    n:  1 = special fully nested mode
7656   //    b:  1 = buffered mode
7657   //    m:  0 = slave PIC, 1 = master PIC
7658   //      (ignored when b is 0, as the master/slave role
7659   //      can be hardwired).
7660   //    a:  1 = Automatic EOI mode
7661   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
7662   outb(IO_PIC1+1, 0x3);
7663
7664   // Set up slave (8259A-2)
7665   outb(IO_PIC2, 0x11);                  // ICW1
7666   outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
7667   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
7668   // NB Automatic EOI mode doesn't tend to work on the slave.
7669   // Linux source code says it's "to be investigated".
7670   outb(IO_PIC2+1, 0x3);                 // ICW4
7671
7672   // OCW3:  0ef01prs
7673   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
7674   //    p:  0 = no polling, 1 = polling mode
7675   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
7676   outb(IO_PIC1, 0x68);             // clear specific mask
7677   outb(IO_PIC1, 0x0a);             // read IRR by default
7678
7679   outb(IO_PIC2, 0x68);             // OCW3
7680   outb(IO_PIC2, 0x0a);             // OCW3
7681
7682   if(irqmask != 0xFFFF)
7683     picsetmask(irqmask);
7684 }
7685
7686
7687
7688
7689
7690
7691
7692
7693
7694
7695
7696
7697
7698
7699
```

```
7700 // PC keyboard interface constants
7701
7702 #define KBSTATP         0x64    // kbd controller status port(I)
7703 #define KBS_DIB         0x01    // kbd data in buffer
7704 #define KBDATAP         0x60    // kbd data port(I)
7705
7706 #define NO              0
7707
7708 #define SHIFT           (1<<0)
7709 #define CTL             (1<<1)
7710 #define ALT             (1<<2)
7711
7712 #define CAPSLOCK        (1<<3)
7713 #define NUMLOCK         (1<<4)
7714 #define SCROLLLOCK      (1<<5)
7715
7716 #define E0ESC           (1<<6)
7717
7718 // Special keycodes
7719 #define KEY_HOME        0xE0
7720 #define KEY_END         0xE1
7721 #define KEY_UP          0xE2
7722 #define KEY_DN          0xE3
7723 #define KEY_LF          0xE4
7724 #define KEY_RT          0xE5
7725 #define KEY_PGUP        0xE6
7726 #define KEY_PGDN        0xE7
7727 #define KEY_INS         0xE8
7728 #define KEY_DEL         0xE9
7729
7730 // C('A') == Control-A
7731 #define C(x) (x - '@')
7732
7733 static uchar shiftcode[256] =
7734 {
7735   [0x1D] CTL,
7736   [0x2A] SHIFT,
7737   [0x36] SHIFT,
7738   [0x38] ALT,
7739   [0x9D] CTL,
7740   [0xB8] ALT
7741 };
7742
7743 static uchar togglecode[256] =
7744 {
7745   [0x3A] CAPSLOCK,
7746   [0x45] NUMLOCK,
7747   [0x46] SCROLLLOCK
7748 };
7749
```



```
7750 static uchar normalmap[256] =
7751 {
7752   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
7753   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
7754   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
7755   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
7756   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
7757   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
7758   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
7759   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7760   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7761   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7762   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7763   [0x9C] '\n',      // KP_Enter
7764   [0xB5] '/',       // KP_Div
7765   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7766   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7767   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7768   [0x97] KEY_HOME,  [0xCF] KEY_END,
7769   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7770 };
7771
7772 static uchar shiftmap[256] =
7773 {
7774   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
7775   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
7776   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
7777   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
7778   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
7779   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
7780   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
7781   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7782   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7783   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7784   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7785   [0x9C] '\n',      // KP_Enter
7786   [0xB5] '/',       // KP_Div
7787   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7788   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7789   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7790   [0x97] KEY_HOME,  [0xCF] KEY_END,
7791   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7792 };
7793
7794
7795
7796
7797
7798
7799
```

```
7800 static uchar ctlmap[256] =
7801 {
7802   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7803   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7804   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
7805   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
7806   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
7807   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
7808   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
7809   [0x9C] '\r',      // KP_Enter
7810   [0xB5] C('/'),    // KP_Div
7811   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7812   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7813   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7814   [0x97] KEY_HOME,  [0xCF] KEY_END,
7815   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7816 };
7817
7818
7819
7820
7821
7822
7823
7824
7825
7826
7827
7828
7829
7830
7831
7832
7833
7834
7835
7836
7837
7838
7839
7840
7841
7842
7843
7844
7845
7846
7847
7848
7849
```



```
7850 #include "types.h"
7851 #include "x86.h"
7852 #include "defs.h"
7853 #include "kbd.h"
7854
7855 int
7856 kbdgetc(void)
7857 {
7858   static uint shift;
7859   static uchar *charcode[4] = {
7860     normalmap, shiftmap, ctlmap, ctlmap
7861   };
7862   uint st, data, c;
7863
7864   st = inb(KBSTATP);
7865   if((st & KBS_DIB) == 0)
7866     return -1;
7867   data = inb(KBDATAP);
7868
7869   if(data == 0xE0){
7870     shift |= E0ESC;
7871     return 0;
7872   } else if(data & 0x80){
7873     // Key released
7874     data = (shift & E0ESC ? data : data & 0x7F);
7875     shift &= ~(shiftcode[data] | E0ESC);
7876     return 0;
7877   } else if(shift & E0ESC){
7878     // Last character was an E0 escape; or with 0x80
7879     data |= 0x80;
7880     shift &= ~E0ESC;
7881   }
7882
7883   shift |= shiftcode[data];
7884   shift ^= togglecode[data];
7885   c = charcode[shift & (CTL | SHIFT)][data];
7886   if(shift & CAPSLOCK){
7887     if('a' <= c && c <= 'z')
7888       c += 'A' - 'a';
7889     else if('A' <= c && c <= 'Z')
7890       c += 'a' - 'A';
7891   }
7892   return c;
7893 }
7894
7895 void
7896 kbdintr(void)
7897 {
7898   consoleintr(kbdgetc);
7899 }
```

```
// Console input and output.
// Input is from the keyboard or serial port.
// Output is written to the screen and serial port.

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

static void consputc(int);

static int panicked = 0;

static struct {
  struct spinlock lock;
  int locking;
} cons;

static void
printint(int xx, int base, int sign)
{
  static char digits[] = "0123456789abcdef";
  char buf[16];
  int i;
  uint x;

  if(sign && (sign = xx < 0))
    x = -xx;
  else
    x = xx;

  i = 0;
  do{
    buf[i++] = digits[x % base];
  }while((x /= base) != 0);

  if(sign)
    buf[i++] = '-';

  while(--i >= 0)
    consputc(buf[i]);
}

```



```
// Print to the console. only understands %d, %x, %p, %s.
void
cprintf(char *fmt, ...)
{
  int i, c, locking;
  uint *argp;
  char *s;

  locking = cons.locking;
  if(locking)
    acquire(&cons.lock);

  if (fmt == 0)
    panic("null fmt");

  argp = (uint*)(void*)(&fmt + 1);
  for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
    if(c != '%'){
      consputc(c);
      continue;
    }
    c = fmt[++i] & 0xff;
    if(c == 0)
      break;
    switch(c){
    case 'd':
      printint(*argp++, 10, 1);
      break;
    case 'x':
    case 'p':
      printint(*argp++, 16, 0);
      break;
    case 's':
      if((s = (char*)*argp++) == 0)
        s = "(null)";
      for(; *s; s++)
        consputc(*s);
      break;
    case '%':
      consputc('%');
      break;
    default:
      // Print unknown % sequence to draw attention.
      consputc('%');
      consputc(c);
      break;
    }
  }


```

```
8000   if(locking)
8001     release(&cons.lock);
8002 }
8003
8004 void
8005 panic(char *s)
8006 {
8007   int i;
8008   uint pcs[10];
8009
8010   cli();
8011   cons.locking = 0;
8012   cprintf("cpu%d: panic: ", cpu->id);
8013   cprintf(s);
8014   cprintf("\n");
8015   getcallerpcs(&s, pcs);
8016   for(i=0; i<10; i++)
8017     cprintf(" %p", pcs[i]);
8018   panicked = 1; // freeze other CPU
8019   for(;;)
8020     ;
8021 }
8022
8023
8024
8025
8026
8027
8028
8029
8030
8031
8032
8033
8034
8035
8036
8037
8038
8039
8040
8041
8042
8043
8044
8045
8046
8047
8048
8049
```



```
8050 #define BACKSPACE 0x100
8051 #define CRTPORT 0x3d4
8052 static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory
8053
8054 static void
8055 cgaputc(int c)
8056 {
8057   int pos;
8058
8059   // Cursor position: col + 80*row.
8060   outb(CRTPORT, 14);
8061   pos = inb(CRTPORT+1) << 8;
8062   outb(CRTPORT, 15);
8063   pos |= inb(CRTPORT+1);
8064
8065   if(c == '\n')
8066     pos += 80 - pos%80;
8067   else if(c == BACKSPACE){
8068     if(pos > 0) --pos;
8069   } else
8070     crt[pos++] = (c&0xff) | 0x0700;  // black on white
8071
8072   if(pos < 0 || pos > 25*80)
8073     panic("pos under/overflow");
8074
8075   if((pos/80) >= 24){  // Scroll up.
8076     memmove(crt, crt+80, sizeof(crt[0])*23*80);
8077     pos -= 80;
8078     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
8079   }
8080
8081   outb(CRTPORT, 14);
8082   outb(CRTPORT+1, pos>>8);
8083   outb(CRTPORT, 15);
8084   outb(CRTPORT+1, pos);
8085   crt[pos] = ' ' | 0x0700;
8086 }
8087
8088
8089
8090
8091
8092
8093
8094
8095
8096
8097
8098
8099
```

```
8100 void
8101 consputc(int c)
8102 {
8103   if(panicked){
8104     cli();
8105     for(;;)
8106       ;
8107   }
8108
8109   if(c == BACKSPACE){
8110     uartputc('\b'); uartputc(' '); uartputc('\b');
8111   } else
8112     uartputc(c);
8113   cgaputc(c);
8114 }
8115
8116 #define INPUT_BUF 128
8117 struct {
8118   char buf[INPUT_BUF];
8119   uint r;  // Read index
8120   uint w;  // Write index
8121   uint e;  // Edit index
8122 } input;
8123
8124 #define C(x)  ((x)-'@')  // Control-x
8125
8126 void
8127 consoleintr(int (*getc)(void))
8128 {
8129   int c, doprocdump = 0;
8130
8131   acquire(&cons.lock);
8132   while((c = getc()) >= 0){
8133     switch(c){
8134     case C('P'):  // Process listing.
8135       doprocdump = 1;   // procdump() locks cons.lock indirectly; invoke lat
8136       break;
8137     case C('U'):  // Kill line.
8138       while(input.e != input.w &&
8139             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
8140         input.e--;
8141         consputc(BACKSPACE);
8142       }
8143       break;
8144     case C('H'): case '\x7f':  // Backspace
8145       if(input.e != input.w){
8146         input.e--;
8147         consputc(BACKSPACE);
8148       }
8149       break;
```



```
8150     default:
8151       if(c != 0 && input.e-input.r < INPUT_BUF){
8152         c = (c == '\r') ? '\n' : c;
8153         input.buf[input.e++ % INPUT_BUF] = c;
8154         consputc(c);
8155         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
8156           input.w = input.e;
8157           wakeup(&input.r);
8158         }
8159       }
8160       break;
8161     }
8162   }
8163   release(&cons.lock);
8164   if(doprocdump) {
8165     procdump();  // now call procdump() wo. cons.lock held
8166   }
8167 }
8168
8169 int
8170 consoleread(struct inode *ip, char *dst, int n)
8171 {
8172   uint target;
8173   int c;
8174
8175   iunlock(ip);
8176   target = n;
8177   acquire(&cons.lock);
8178   while(n > 0){
8179     while(input.r == input.w){
8180       if(proc->killed){
8181         release(&cons.lock);
8182         ilock(ip);
8183         return -1;
8184       }
8185       sleep(&input.r, &cons.lock);
8186     }
8187     c = input.buf[input.r++ % INPUT_BUF];
8188     if(c == C('D')){  // EOF
8189       if(n < target){
8190         // Save ^D for next time, to make sure
8191         // caller gets a 0-byte result.
8192         input.r--;
8193       }
8194       break;
8195     }
8196     *dst++ = c;
8197     --n;
8198     if(c == '\n')
8199       break;
```

```
8200   }
8201   release(&cons.lock);
8202   ilock(ip);
8203
8204   return target - n;
8205 }
8206
8207 int
8208 consolewrite(struct inode *ip, char *buf, int n)
8209 {
8210   int i;
8211
8212   iunlock(ip);
8213   acquire(&cons.lock);
8214   for(i = 0; i < n; i++)
8215     consputc(buf[i] & 0xff);
8216   release(&cons.lock);
8217   ilock(ip);
8218
8219   return n;
8220 }
8221
8222 void
8223 consoleinit(void)
8224 {
8225   initlock(&cons.lock, "console");
8226
8227   devsw[CONSOLE].write = consolewrite;
8228   devsw[CONSOLE].read = consoleread;
8229   cons.locking = 1;
8230
8231   picenable(IRQ_KBD);
8232   ioapicenable(IRQ_KBD, 0);
8233 }
8234
8235
8236
8237
8238
8239
8240
8241
8242
8243
8244
8245
8246
8247
8248
8249
```



```
8250 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
8251 // Only used on uniprocessors;
8252 // SMP machines use the local APIC timer.
8253
8254 #include "types.h"
8255 #include "defs.h"
8256 #include "traps.h"
8257 #include "x86.h"
8258
8259 #define IO_TIMER1       0x040           // 8253 Timer #1
8260
8261 // Frequency of all three count-down timers;
8262 // (TIMER_FREQ/freq) is the appropriate count
8263 // to generate a frequency of freq Hz.
8264
8265 #define TIMER_FREQ      1193182
8266 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
8267
8268 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
8269 #define TIMER_SEL0      0x00    // select counter 0
8270 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
8271 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
8272
8273 void
8274 timerinit(void)
8275 {
8276   // Interrupt 100 times/sec.
8277   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
8278   outb(IO_TIMER1, TIMER_DIV(100) % 256);
8279   outb(IO_TIMER1, TIMER_DIV(100) / 256);
8280   picenable(IRQ_TIMER);
8281 }
8282
8283
8284
8285
8286
8287
8288
8289
8290
8291
8292
8293
8294
8295
8296
8297
8298
8299
```

```
8300 // Intel 8250 serial port (UART).
8301
8302 #include "types.h"
8303 #include "defs.h"
8304 #include "param.h"
8305 #include "traps.h"
8306 #include "spinlock.h"
8307 #include "fs.h"
8308 #include "file.h"
8309 #include "mmu.h"
8310 #include "proc.h"
8311 #include "x86.h"
8312
8313 #define COM1    0x3f8
8314
8315 static int uart;    // is there a uart?
8316
8317 void
8318 uartinit(void)
8319 {
8320   char *p;
8321
8322   // Turn off the FIFO
8323   outb(COM1+2, 0);
8324
8325   // 9600 baud, 8 data bits, 1 stop bit, parity off.
8326   outb(COM1+3, 0x80);    // Unlock divisor
8327   outb(COM1+0, 115200/9600);
8328   outb(COM1+1, 0);
8329   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
8330   outb(COM1+4, 0);
8331   outb(COM1+1, 0x01);    // Enable receive interrupts.
8332
8333   // If status is 0xFF, no serial port.
8334   if(inb(COM1+5) == 0xFF)
8335     return;
8336   uart = 1;
8337
8338   // Acknowledge pre-existing interrupt conditions;
8339   // enable interrupts.
8340   inb(COM1+2);
8341   inb(COM1+0);
8342   picenable(IRQ_COM1);
8343   ioapicenable(IRQ_COM1, 0);
8344
8345   // Announce that we're here.
8346   for(p="xv6...\n"; *p; p++)
8347     uartputc(*p);
8348 }
8349
```



```
8350 void
8351 uartputc(int c)
8352 {
8353   int i;
8354
8355   if(!uart)
8356     return;
8357   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
8358     microdelay(10);
8359   outb(COM1+0, c);
8360 }
8361
8362 static int
8363 uartgetc(void)
8364 {
8365   if(!uart)
8366     return -1;
8367   if(!(inb(COM1+5) & 0x01))
8368     return -1;
8369   return inb(COM1+0);
8370 }
8371
8372 void
8373 uartintr(void)
8374 {
8375   consoleintr(uartgetc);
8376 }
8377
8378
8379
8380
8381
8382
8383
8384
8385
8386
8387
8388
8389
8390
8391
8392
8393
8394
8395
8396
8397
8398
8399
```

```
8400 # Initial process execs /init.
8401
8402 #include "syscall.h"
8403 #include "traps.h"
8404
8405
8406 # exec(init, argv)
8407 .globl start
8408 start:
8409   pushl $argv
8410   pushl $init
8411   pushl $0  // where caller pc would be
8412   movl $SYS_exec, %eax
8413   int $T_SYSCALL
8414
8415 # for(;;) exit();
8416 exit:
8417   movl $SYS_exit, %eax
8418   int $T_SYSCALL
8419   jmp exit
8420
8421 # char init[] = "/init\0";
8422 init:
8423   .string "/init\0"
8424
8425 # char *argv[] = { init, 0 };
8426 .p2align 2
8427 argv:
8428   .long init
8429   .long 0
8430
8431
8432
8433
8434
8435
8436
8437
8438
8439
8440
8441
8442
8443
8444
8445
8446
8447
8448
8449
```



```
8450 #include "syscall.h"
8451 #include "traps.h"
8452
8453 #define SYSCALL(name) \
8454   .globl name; \
8455   name: \
8456     movl $SYS_ ## name, %eax; \
8457     int $T_SYSCALL; \
8458     ret
8459
8460 SYSCALL(fork)
8461 SYSCALL(exit)
8462 SYSCALL(wait)
8463 SYSCALL(pipe)
8464 SYSCALL(read)
8465 SYSCALL(write)
8466 SYSCALL(close)
8467 SYSCALL(kill)
8468 SYSCALL(exec)
8469 SYSCALL(open)
8470 SYSCALL(mknod)
8471 SYSCALL(unlink)
8472 SYSCALL(fstat)
8473 SYSCALL(link)
8474 SYSCALL(mkdir)
8475 SYSCALL(chdir)
8476 SYSCALL(dup)
8477 SYSCALL(getpid)
8478 SYSCALL(sbrk)
8479 SYSCALL(sleep)
8480 SYSCALL(uptime)
8481 SYSCALL(date)
8482
8483
8484
8485
8486
8487
8488
8489
8490
8491
8492
8493
8494
8495
8496
8497
8498
8499
```

```
8500 // init: The initial user-level program
8501
8502 #include "types.h"
8503 #include "stat.h"
8504 #include "user.h"
8505 #include "fcntl.h"
8506
8507 char *argv[] = { "sh", 0 };
8508
8509 int
8510 main(void)
8511 {
8512   int pid, wpid;
8513
8514   if(open("console", O_RDWR) < 0){
8515     mknod("console", 1, 1);
8516     open("console", O_RDWR);
8517   }
8518   dup(0);  // stdout
8519   dup(0);  // stderr
8520
8521   for(;;){
8522     printf(1, "init: starting sh\n");
8523     pid = fork();
8524     if(pid < 0){
8525       printf(1, "init: fork failed\n");
8526       exit();
8527     }
8528     if(pid == 0){
8529       exec("sh", argv);
8530       printf(1, "init: exec sh failed\n");
8531       exit();
8532     }
8533     while((wpid=wait()) >= 0 && wpid != pid)
8534       printf(1, "zombie!\n");
8535   }
8536 }
8537
8538
8539
8540
8541
8542
8543
8544
8545
8546
8547
8548
8549
```



```
8550 // Shell.
8551
8552 #include "types.h"
8553 #include "user.h"
8554 #include "fcntl.h"
8555
8556 // Parsed command representation
8557 #define EXEC  1
8558 #define REDIR 2
8559 #define PIPE  3
8560 #define LIST  4
8561 #define BACK  5
8562
8563 #define MAXARGS 10
8564
8565 struct cmd {
8566   int type;
8567 };
8568
8569 struct execcmd {
8570   int type;
8571   char *argv[MAXARGS];
8572   char *eargv[MAXARGS];
8573 };
8574
8575 struct redircmd {
8576   int type;
8577   struct cmd *cmd;
8578   char *file;
8579   char *efile;
8580   int mode;
8581   int fd;
8582 };
8583
8584 struct pipecmd {
8585   int type;
8586   struct cmd *left;
8587   struct cmd *right;
8588 };
8589
8590 struct listcmd {
8591   int type;
8592   struct cmd *left;
8593   struct cmd *right;
8594 };
8595
8596 struct backcmd {
8597   int type;
8598   struct cmd *cmd;
8599 };
```

```
8600 int fork1(void);  // Fork but panics on failure.
8601 void panic(char*);
8602 struct cmd *parsecmd(char*);
8603
8604 // Execute cmd.  Never returns.
8605 void
8606 runcmd(struct cmd *cmd)
8607 {
8608   int p[2];
8609   struct backcmd *bcmd;
8610   struct execcmd *ecmd;
8611   struct listcmd *lcmd;
8612   struct pipecmd *pcmd;
8613   struct redircmd *rcmd;
8614
8615   if(cmd == 0)
8616     exit();
8617
8618   switch(cmd->type){
8619   default:
8620     panic("runcmd");
8621
8622   case EXEC:
8623     ecmd = (struct execcmd*)cmd;
8624     if(ecmd->argv[0] == 0)
8625       exit();
8626     exec(ecmd->argv[0], ecmd->argv);
8627     printf(2, "exec %s failed\n", ecmd->argv[0]);
8628     break;
8629
8630   case REDIR:
8631     rcmd = (struct redircmd*)cmd;
8632     close(rcmd->fd);
8633     if(open(rcmd->file, rcmd->mode) < 0){
8634       printf(2, "open %s failed\n", rcmd->file);
8635       exit();
8636     }
8637     runcmd(rcmd->cmd);
8638     break;
8639
8640   case LIST:
8641     lcmd = (struct listcmd*)cmd;
8642     if(fork1() == 0)
8643       runcmd(lcmd->left);
8644     wait();
8645     runcmd(lcmd->right);
8646     break;
8647
8648
8649
```



```
8650   case PIPE:
8651     pcmd = (struct pipecmd*)cmd;
8652     if(pipe(p) < 0)
8653       panic("pipe");
8654     if(fork1() == 0){
8655       close(1);
8656       dup(p[1]);
8657       close(p[0]);
8658       close(p[1]);
8659       runcmd(pcmd->left);
8660     }
8661     if(fork1() == 0){
8662       close(0);
8663       dup(p[0]);
8664       close(p[0]);
8665       close(p[1]);
8666       runcmd(pcmd->right);
8667     }
8668     close(p[0]);
8669     close(p[1]);
8670     wait();
8671     wait();
8672     break;
8673
8674   case BACK:
8675     bcmd = (struct backcmd*)cmd;
8676     if(fork1() == 0)
8677       runcmd(bcmd->cmd);
8678     break;
8679   }
8680   exit();
8681 }
8682
8683 int
8684 getcmd(char *buf, int nbuf)
8685 {
8686   printf(2, "$ ");
8687   memset(buf, 0, nbuf);
8688   gets(buf, nbuf);
8689   if(buf[0] == 0) // EOF
8690     return -1;
8691   return 0;
8692 }
8693
8694
8695
8696
8697
8698
8699
```

```
8700 int
8701 main(void)
8702 {
8703   static char buf[100];
8704   int fd;
8705
8706   // Assumes three file descriptors open.
8707   while((fd = open("console", O_RDWR)) >= 0){
8708     if(fd >= 3){
8709       close(fd);
8710       break;
8711     }
8712   }
8713
8714   // Read and run input commands.
8715   while(getcmd(buf, sizeof(buf)) >= 0){
8716     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
8717       // Clumsy but will have to do for now.
8718       // Chdir has no effect on the parent if run in the child.
8719       buf[strlen(buf)-1] = 0;  // chop \n
8720       if(chdir(buf+3) < 0)
8721         printf(2, "cannot cd %s\n", buf+3);
8722       continue;
8723     }
8724     if(fork1() == 0)
8725       runcmd(parsecmd(buf));
8726     wait();
8727   }
8728   exit();
8729 }
8730
8731 void
8732 panic(char *s)
8733 {
8734   printf(2, "%s\n", s);
8735   exit();
8736 }
8737
8738 int
8739 fork1(void)
8740 {
8741   int pid;
8742
8743   pid = fork();
8744   if(pid == -1)
8745     panic("fork");
8746   return pid;
8747 }
8748
8749
```



```
8750 // Constructors
8751
8752 struct cmd*
8753 execcmd(void)
8754 {
8755   struct execcmd *cmd;
8756
8757   cmd = malloc(sizeof(*cmd));
8758   memset(cmd, 0, sizeof(*cmd));
8759   cmd->type = EXEC;
8760   return (struct cmd*)cmd;
8761 }
8762
8763 struct cmd*
8764 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
8765 {
8766   struct redircmd *cmd;
8767
8768   cmd = malloc(sizeof(*cmd));
8769   memset(cmd, 0, sizeof(*cmd));
8770   cmd->type = REDIR;
8771   cmd->cmd = subcmd;
8772   cmd->file = file;
8773   cmd->efile = efile;
8774   cmd->mode = mode;
8775   cmd->fd = fd;
8776   return (struct cmd*)cmd;
8777 }
8778
8779 struct cmd*
8780 pipecmd(struct cmd *left, struct cmd *right)
8781 {
8782   struct pipecmd *cmd;
8783
8784   cmd = malloc(sizeof(*cmd));
8785   memset(cmd, 0, sizeof(*cmd));
8786   cmd->type = PIPE;
8787   cmd->left = left;
8788   cmd->right = right;
8789   return (struct cmd*)cmd;
8790 }
8791
8792
8793
8794
8795
8796
8797
8798
8799
```

```
8800 struct cmd*
8801 listcmd(struct cmd *left, struct cmd *right)
8802 {
8803   struct listcmd *cmd;
8804
8805   cmd = malloc(sizeof(*cmd));
8806   memset(cmd, 0, sizeof(*cmd));
8807   cmd->type = LIST;
8808   cmd->left = left;
8809   cmd->right = right;
8810   return (struct cmd*)cmd;
8811 }
8812
8813 struct cmd*
8814 backcmd(struct cmd *subcmd)
8815 {
8816   struct backcmd *cmd;
8817
8818   cmd = malloc(sizeof(*cmd));
8819   memset(cmd, 0, sizeof(*cmd));
8820   cmd->type = BACK;
8821   cmd->cmd = subcmd;
8822   return (struct cmd*)cmd;
8823 }
8824
8825
8826
8827
8828
8829
8830
8831
8832
8833
8834
8835
8836
8837
8838
8839
8840
8841
8842
8843
8844
8845
8846
8847
8848
8849
```



```
8850 // Parsing
8851
8852 char whitespace[] = " \t\r\n\v";
8853 char symbols[] = "<|>&;()";
8854
8855 int
8856 gettoken(char **ps, char *es, char **q, char **eq)
8857 {
8858   char *s;
8859   int ret;
8860
8861   s = *ps;
8862   while(s < es && strchr(whitespace, *s))
8863     s++;
8864   if(q)
8865     *q = s;
8866   ret = *s;
8867   switch(*s){
8868   case 0:
8869     break;
8870   case '|':
8871   case '(':
8872   case ')':
8873   case ';':
8874   case '&':
8875   case '<':
8876     s++;
8877     break;
8878   case '>':
8879     s++;
8880     if(*s == '>'){
8881       ret = '+';
8882       s++;
8883     }
8884     break;
8885   default:
8886     ret = 'a';
8887     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
8888       s++;
8889     break;
8890   }
8891   if(eq)
8892     *eq = s;
8893
8894   while(s < es && strchr(whitespace, *s))
8895     s++;
8896   *ps = s;
8897   return ret;
8898 }
8899
```

```
8900 int
8901 peek(char **ps, char *es, char *toks)
8902 {
8903   char *s;
8904
8905   s = *ps;
8906   while(s < es && strchr(whitespace, *s))
8907     s++;
8908   *ps = s;
8909   return *s && strchr(toks, *s);
8910 }
8911
8912 struct cmd *parseline(char**, char*);
8913 struct cmd *parsepipe(char**, char*);
8914 struct cmd *parseexec(char**, char*);
8915 struct cmd *nulterminate(struct cmd*);
8916
8917 struct cmd*
8918 parsecmd(char *s)
8919 {
8920   char *es;
8921   struct cmd *cmd;
8922
8923   es = s + strlen(s);
8924   cmd = parseline(&s, es);
8925   peek(&s, es, "");
8926   if(s != es){
8927     printf(2, "leftovers: %s\n", s);
8928     panic("syntax");
8929   }
8930   nulterminate(cmd);
8931   return cmd;
8932 }
8933
8934 struct cmd*
8935 parseline(char **ps, char *es)
8936 {
8937   struct cmd *cmd;
8938
8939   cmd = parsepipe(ps, es);
8940   while(peek(ps, es, "&")){
8941     gettoken(ps, es, 0, 0);
8942     cmd = backcmd(cmd);
8943   }
8944   if(peek(ps, es, ";")){
8945     gettoken(ps, es, 0, 0);
8946     cmd = listcmd(cmd, parseline(ps, es));
8947   }
8948   return cmd;
8949 }
```



```
8950 struct cmd*
8951 parsepipe(char **ps, char *es)
8952 {
8953   struct cmd *cmd;
8954
8955   cmd = parseexec(ps, es);
8956   if(peek(ps, es, "|")){
8957     gettoken(ps, es, 0, 0);
8958     cmd = pipecmd(cmd, parsepipe(ps, es));
8959   }
8960   return cmd;
8961 }
8962
8963 struct cmd*
8964 parseredirs(struct cmd *cmd, char **ps, char *es)
8965 {
8966   int tok;
8967   char *q, *eq;
8968
8969   while(peek(ps, es, "<>")){
8970     tok = gettoken(ps, es, 0, 0);
8971     if(gettoken(ps, es, &q, &eq) != 'a')
8972       panic("missing file for redirection");
8973     switch(tok){
8974     case '<':
8975       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
8976       break;
8977     case '>':
8978       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8979       break;
8980     case '+':  // >>
8981       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8982       break;
8983     }
8984   }
8985   return cmd;
8986 }
8987
8988
8989
8990
8991
8992
8993
8994
8995
8996
8997
8998
8999
```

```
9000 struct cmd*
9001 parseblock(char **ps, char *es)
9002 {
9003   struct cmd *cmd;
9004
9005   if(!peek(ps, es, "("))
9006     panic("parseblock");
9007   gettoken(ps, es, 0, 0);
9008   cmd = parseline(ps, es);
9009   if(!peek(ps, es, ")"))
9010     panic("syntax - missing )");
9011   gettoken(ps, es, 0, 0);
9012   cmd = parseredirs(cmd, ps, es);
9013   return cmd;
9014 }
9015
9016 struct cmd*
9017 parseexec(char **ps, char *es)
9018 {
9019   char *q, *eq;
9020   int tok, argc;
9021   struct execcmd *cmd;
9022   struct cmd *ret;
9023
9024   if(peek(ps, es, "("))
9025     return parseblock(ps, es);
9026
9027   ret = execcmd();
9028   cmd = (struct execcmd*)ret;
9029
9030   argc = 0;
9031   ret = parseredirs(ret, ps, es);
9032   while(!peek(ps, es, "|)&;")){
9033     if((tok=gettoken(ps, es, &q, &eq)) == 0)
9034       break;
9035     if(tok != 'a')
9036       panic("syntax");
9037     cmd->argv[argc] = q;
9038     cmd->eargv[argc] = eq;
9039     argc++;
9040     if(argc >= MAXARGS)
9041       panic("too many args");
9042     ret = parseredirs(ret, ps, es);
9043   }
9044   cmd->argv[argc] = 0;
9045   cmd->eargv[argc] = 0;
9046   return ret;
9047 }
9048
9049
```



```
9050 // NUL-terminate all the counted strings.
9051 struct cmd*
9052 nulterminate(struct cmd *cmd)
9053 {
9054   int i;
9055   struct backcmd *bcmd;
9056   struct execcmd *ecmd;
9057   struct listcmd *lcmd;
9058   struct pipecmd *pcmd;
9059   struct redircmd *rcmd;
9060
9061   if(cmd == 0)
9062     return 0;
9063
9064   switch(cmd->type){
9065   case EXEC:
9066     ecmd = (struct execcmd*)cmd;
9067     for(i=0; ecmd->argv[i]; i++)
9068       *ecmd->eargv[i] = 0;
9069     break;
9070
9071   case REDIR:
9072     rcmd = (struct redircmd*)cmd;
9073     nulterminate(rcmd->cmd);
9074     *rcmd->efile = 0;
9075     break;
9076
9077   case PIPE:
9078     pcmd = (struct pipecmd*)cmd;
9079     nulterminate(pcmd->left);
9080     nulterminate(pcmd->right);
9081     break;
9082
9083   case LIST:
9084     lcmd = (struct listcmd*)cmd;
9085     nulterminate(lcmd->left);
9086     nulterminate(lcmd->right);
9087     break;
9088
9089   case BACK:
9090     bcmd = (struct backcmd*)cmd;
9091     nulterminate(bcmd->cmd);
9092     break;
9093   }
9094   return cmd;
9095 }
9096
9097
9098
9099
```

```
#include "asm.h"
#include "memlayout.h"
#include "mmu.h"

# Start the first CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.code16                       # Assemble for 16-bit mode
.globl start
start:
  cli                         # BIOS enabled interrupts; disable

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax             # Set %ax to zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment

  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60

  # Switch from real to protected mode.  Use a bootstrap GDT that makes
  # virtual addresses map directly to physical addresses so that the
  # effective memory map doesn't change during the transition.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
  movl    %eax, %cr0
```



```
  # Complete transition to 32-bit protected mode by using long jmp
  # to reload %cs and %eip.  The segment descriptors are set up with no
  # translation, so that the mapping is still the identity mapping.
  ljmp    $(SEG_KCODE<<3), $start32

.code32  # Tell assembler to generate 32-bit code now.
start32:
  # Set up the protected-mode data segment registers
  movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %ss                # -> SS: Stack Segment
  movw    $0, %ax                 # Zero segments not ready for use
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS

  # Set up the stack pointer and call into C.
  movl    $start, %esp
  call    bootmain

  # If bootmain returns (it shouldn't), trigger a Bochs
  # breakpoint if running under Bochs, then loop.
  movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
  movw    %ax, %dx
  outw    %ax, %dx
  movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
  outw    %ax, %dx
spin:
  jmp     spin

# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULLASM                             # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg

gdtdesc:
  .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
  .long   gdt                             # address gdt
```

```
9200 // Boot loader.
9201 //
9202 // Part of the boot block, along with bootasm.S, which calls bootmain().
9203 // bootasm.S has put the processor into protected 32-bit mode.
9204 // bootmain() loads an ELF kernel image from the disk starting at
9205 // sector 1 and then jumps to the kernel entry routine.
9206
9207 #include "types.h"
9208 #include "elf.h"
9209 #include "x86.h"
9210 #include "memlayout.h"
9211
9212 #define SECTSIZE  512
9213
9214 void readseg(uchar*, uint, uint);
9215
9216 void
9217 bootmain(void)
9218 {
9219   struct elfhdr *elf;
9220   struct proghdr *ph, *eph;
9221   void (*entry)(void);
9222   uchar* pa;
9223
9224   elf = (struct elfhdr*)0x10000;  // scratch space
9225
9226   // Read 1st page off disk
9227   readseg((uchar*)elf, 4096, 0);
9228
9229   // Is this an ELF executable?
9230   if(elf->magic != ELF_MAGIC)
9231     return;  // let bootasm.S handle error
9232
9233   // Load each program segment (ignores ph flags).
9234   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
9235   eph = ph + elf->phnum;
9236   for(; ph < eph; ph++){
9237     pa = (uchar*)ph->paddr;
9238     readseg(pa, ph->filesz, ph->off);
9239     if(ph->memsz > ph->filesz)
9240       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
9241   }
9242
9243   // Call the entry point from the ELF header.
9244   // Does not return!
9245   entry = (void(*)(void))(elf->entry);
9246   entry();
9247 }
9248
9249
```



```
9250 void
9251 waitdisk(void)
9252 {
9253   // Wait for disk ready.
9254   while((inb(0x1F7) & 0xC0) != 0x40)
9255     ;
9256 }
9257
9258 // Read a single sector at offset into dst.
9259 void
9260 readsect(void *dst, uint offset)
9261 {
9262   // Issue command.
9263   waitdisk();
9264   outb(0x1F2, 1);   // count = 1
9265   outb(0x1F3, offset);
9266   outb(0x1F4, offset >> 8);
9267   outb(0x1F5, offset >> 16);
9268   outb(0x1F6, (offset >> 24) | 0xE0);
9269   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
9270
9271   // Read data.
9272   waitdisk();
9273   insl(0x1F0, dst, SECTSIZE/4);
9274 }
9275
9276 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
9277 // Might copy more than asked.
9278 void
9279 readseg(uchar* pa, uint count, uint offset)
9280 {
9281   uchar* epa;
9282
9283   epa = pa + count;
9284
9285   // Round down to sector boundary.
9286   pa -= offset % SECTSIZE;
9287
9288   // Translate from bytes to sectors; kernel starts at sector 1.
9289   offset = (offset / SECTSIZE) + 1;
9290
9291   // If this is too slow, we could read lots of sectors at a time.
9292   // We'd write more to memory than asked, but it doesn't matter --
9293   // we load in increasing order.
9294   for(; pa < epa; pa += SECTSIZE, offset++)
9295     readsect(pa, offset);
9296 }
9297
9298
9299
```

```
9300 struct rtcdate {
9301   uint second;
9302   uint minute;
9303   uint hour;
9304   uint day;
9305   uint month;
9306   uint year;
9307 };
9308
9309
9310
9311
9312
9313
9314
9315
9316
9317
9318
9319
9320
9321
9322
9323
9324
9325
9326
9327
9328
9329
9330
9331
9332
9333
9334
9335
9336
9337
9338
9339
9340
9341
9342
9343
9344
9345
9346
9347
9348
9349
```



```
9350 #include "types.h"
9351 #include "user.h"
9352 #include "date.h"
9353
9354 int
9355 main(int argc, char *argv[])
9356 {
9357   struct rtcdate r;
9358
9359   if (date(&r)) {
9360     printf(2, "date_failed\n");
9361     exit();
9362   }
9363
9364   char *months[] = {
9365     "Jan",
9366     "Feb",
9367     "Mar",
9368     "Apr",
9369     "May",
9370     "Jun",
9371     "Jul",
9372     "Aug",
9373     "Sep",
9374     "Oct",
9375     "Nov",
9376     "Dec"
9377   };
9378
9379   printf(1, "%s %d %d:%d:%d UTC %d\n",
9380          months[r.month-1], r.day, r.hour, r.minute, r.second, r.year);
9381
9382   exit();
9383 }
9384
9385
9386
9387
9388
9389
9390
9391
9392
9393
9394
9395
9396
9397
9398
9399
```

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

#define MAXARGS 20

int
main(int argc, char *argv[]) {
  int i;
  char *p[MAXARGS];
  struct rtcdate r1, r2;
  int pid;

  if (argc >= (MAXARGS)) {  // we count from 0
    printf(2, "Error: too many args\n");
    exit();
  }

  for (i=0; i<argc-1; i++) {
    p[i] = strcpy(p[i], argv[i+1]);
  }
  p[i] = '\0';

  if (date(&r1) != 0) {
    printf(2, "sys_date failed\n");
    exit();
  }

  pid = fork();
  if (pid < 0) {
    printf(2, "fork() failed\n");
    exit();
  } else if (pid == 0) {  // child
    exec(p[0], p);
    printf(2, "Error: exec failed. Arg(s) probably needs full path\n");
    exit();
  } else {                // parent
    wait();
    if (date(&r2) != 0) {
      printf(2, "date failed");
      exit();
    }
  }

  uint min, sec;
```



```
  if (r1.second > r2.second) {
    min = r2.minute - r1.minute - 1;
    sec = r1.second - r2.second;
  } else {
    min = r2.minute - r1.minute;
    sec = r2.second - r1.second;
  }

  printf(1, "\n%dm%ds\n", min, sec);

  exit();
}
```