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_halt   22
3473 //Added to define position of the system call vector that connects
3474 //to implementation
3475 #define SYS_date   23
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_halt(void);
3601 //Added sys_date to allow routine to be available to other parts of kernel
3602 extern int sys_date(void);
3603 extern int sys_time(void);
3604
3605 static int (*syscalls[])(void) = {
3606 [SYS_fork]    sys_fork,
3607 [SYS_exit]    sys_exit,
3608 [SYS_wait]    sys_wait,
3609 [SYS_pipe]    sys_pipe,
3610 [SYS_read]    sys_read,
3611 [SYS_kill]    sys_kill,
3612 [SYS_exec]    sys_exec,
3613 [SYS_fstat]   sys_fstat,
3614 [SYS_chdir]   sys_chdir,
3615 [SYS_dup]     sys_dup,
3616 [SYS_getpid]  sys_getpid,
3617 [SYS_sbrk]    sys_sbrk,
3618 [SYS_sleep]   sys_sleep,
3619 [SYS_uptime]  sys_uptime,
3620 [SYS_open]    sys_open,
3621 [SYS_write]   sys_write,
3622 [SYS_mknod]   sys_mknod,
3623 [SYS_unlink]  sys_unlink,
3624 [SYS_link]    sys_link,
3625 [SYS_mkdir]   sys_mkdir,
3626 [SYS_close]   sys_close,
3627 [SYS_halt]    sys_halt,
3628 [SYS_date]      sys_date,
3629 };
3630
3631 void
3632 syscall(void)
3633 {
3634 /*
3635     char * syscallnames[] = {
3636     [SYS_fork]    "fork",
3637     [SYS_exit]    "sys_exit",
3638     [SYS_wait]    "sys_wait",
3639     [SYS_pipe]    "sys_pipe",
3640     [SYS_read]    "sys_read",
3641     [SYS_kill]    "sys_kill",
3642     [SYS_exec]    "sys_exec",
3643     [SYS_fstat]   "sys_fstat",
3644     [SYS_chdir]   "sys_chdir",
3645     [SYS_dup]     "sys_dup",
3646     [SYS_getpid]  "sys_getpid",
3647     [SYS_sbrk]    "sys_sbrk",
3648     [SYS_sleep]   "sys_sleep",
3649     [SYS_uptime]  "sys_uptime",
```



```
3650     [SYS_open]    "sys_open",
3651     [SYS_write]   "sys_write",
3652     [SYS_mknod]   "sys_mknod",
3653     [SYS_unlink]  "sys_unlink",
3654     [SYS_link]    "sys_link",
3655     [SYS_mkdir]   "sys_mkdir",
3656     [SYS_close]   "sys_close",
3657     [SYS_halt]    "sys_halt",
3658     };
3659 */
3660   int num;
3661   num = proc->tf->eax;
3662
3663   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3664     proc->tf->eax = syscalls[num]();
3665 //      cprintf("%s -> %d\n", syscallnames[num], proc->tf->eax);
3666   } else {
3667     cprintf("%d %s: unknown sys call %d\n",
3668             proc->pid, proc->name, num);
3669     proc->tf->eax = -1;
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 //Turn of the computer
3801 int sys_halt(void){
3802   cprintf("Shutting down ...\n");
3803   outw (0xB004, 0x0 | 0x2000);
3804   return 0;
3805 }
3806
3807 //Implemented date and time
3808 int
3809 sys_date(void)
3810 {
3811     struct rtcdate *d;
3812
3813     if(argptr(0, (void*)&d, sizeof(*d)) < 0)
3814         return -1;
3815
3816     cmostime(d);
3817     return 0;
3818 }
3819
3820
3821
3822
3823
3824
3825
3826
3827
3828
3829
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
```



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

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



```
3950 #define O_RDONLY  0x000
3951 #define O_WRONLY  0x001
3952 #define O_RDWR    0x002
3953 #define O_CREATE  0x200
3954
3955
3956
3957
3958
3959
3960
3961
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
```

```
4000 #define T_DIR  1   // Directory
4001 #define T_FILE 2   // File
4002 #define T_DEV  3   // Device
4003
4004 struct stat {
4005   short type;  // Type of file
4006   int dev;     // File system's disk device
4007   uint ino;    // Inode number
4008   short nlink; // Number of links to file
4009   uint size;   // Size of file in bytes
4010 };
4011
4012
4013
4014
4015
4016
4017
4018
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
```



```
4050 // On-disk file system format.
4051 // Both the kernel and user programs use this header file.
4052
4053
4054 #define ROOTINO 1  // root i-number
4055 #define BSIZE 512  // block size
4056
4057 // Disk layout:
4058 // [ boot block | super block | log | inode blocks | free bit map | data bloc
4059 //
4060 // mkfs computes the super block and builds an initial file system. The super
4061 // the disk layout:
4062 struct superblock {
4063   uint size;         // Size of file system image (blocks)
4064   uint nblocks;      // Number of data blocks
4065   uint ninodes;      // Number of inodes.
4066   uint nlog;         // Number of log blocks
4067   uint logstart;     // Block number of first log block
4068   uint inodestart;   // Block number of first inode block
4069   uint bmapstart;    // Block number of first free map block
4070 };
4071
4072 #define NDIRECT 12
4073 #define NINDIRECT (BSIZE / sizeof(uint))
4074 #define MAXFILE (NDIRECT + NINDIRECT)
4075
4076 // On-disk inode structure
4077 struct dinode {
4078   short type;           // File type
4079   short major;          // Major device number (T_DEV only)
4080   short minor;          // Minor device number (T_DEV only)
4081   short nlink;          // Number of links to inode in file system
4082   uint size;            // Size of file (bytes)
4083   uint addrs[NDIRECT+1];   // Data block addresses
4084 };
4085
4086
4087
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
```

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

```
// See MultiProcessor Specification Version 1.[14]

struct mp {             // floating pointer
  uchar signature[4];           // "_MP_"
  void *physaddr;               // phys addr of MP config table
  uchar length;                 // 1
  uchar specrev;                // [14]
  uchar checksum;               // all bytes must add up to 0
  uchar type;                   // MP system config type
  uchar imcrp;
  uchar reserved[3];
};

struct mpconf {         // configuration table header
  uchar signature[4];           // "PCMP"
  ushort length;                // total table length
  uchar version;                // [14]
  uchar checksum;               // all bytes must add up to 0
  uchar product[20];            // product id
  uint *oemtable;               // OEM table pointer
  ushort oemlength;             // OEM table length
  ushort entry;                 // entry count
  uint *lapicaddr;              // address of local APIC
  ushort xlength;               // extended table length
  uchar xchecksum;              // extended table checksum
  uchar reserved;
};

struct mpproc {         // processor table entry
  uchar type;                   // entry type (0)
  uchar apicid;                 // local APIC id
  uchar version;                // local APIC verison
  uchar flags;                  // CPU flags
    #define MPBOOT 0x02           // This proc is the bootstrap processor.
  uchar signature[4];           // CPU signature
  uint feature;                 // feature flags from CPUID instruction
  uchar reserved[8];
};

struct mpioapic {       // I/O APIC table entry
  uchar type;                   // entry type (2)
  uchar apicno;                 // I/O APIC id
  uchar version;                // I/O APIC version
  uchar flags;                  // I/O APIC flags
  uint *addr;                  // I/O APIC address
};
```



```
// Table entry types
#define MPPROC    0x00  // One per processor
#define MPBUS     0x01  // One per bus
#define MPIOAPIC  0x02  // One per I/O APIC
#define MPIOINTR  0x03  // One per bus interrupt source
#define MPLINTR   0x04  // One per system interrupt source
```

```
7000 // Blank page.
7001
7002
7003
7004
7005
7006
7007
7008
7009
7010
7011
7012
7013
7014
7015
7016
7017
7018
7019
7020
7021
7022
7023
7024
7025
7026
7027
7028
7029
7030
7031
7032
7033
7034
7035
7036
7037
7038
7039
7040
7041
7042
7043
7044
7045
7046
7047
7048
7049
```



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

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



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

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



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

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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



```
8750 int
8751 main(void)
8752 {
8753   static char buf[100];
8754   int fd;
8755
8756   // Assumes three file descriptors open.
8757   while((fd = open("console", O_RDWR)) >= 0){
8758     if(fd >= 3){
8759       close(fd);
8760       break;
8761     }
8762   }
8763
8764   // Read and run input commands.
8765   while(getcmd(buf, sizeof(buf)) >= 0){
8766     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
8767       // Clumsy but will have to do for now.
8768       // Chdir has no effect on the parent if run in the child.
8769       buf[strlen(buf)-1] = 0;  // chop \n
8770       if(chdir(buf+3) < 0)
8771         printf(2, "cannot cd %s\n", buf+3);
8772       continue;
8773     }
8774     if(fork1() == 0)
8775       runcmd(parsecmd(buf));
8776     wait();
8777   }
8778   exit();
8779 }
8780
8781 void
8782 panic(char *s)
8783 {
8784   printf(2, "%s\n", s);
8785   exit();
8786 }
8787
8788 int
8789 fork1(void)
8790 {
8791   int pid;
8792
8793   pid = fork();
8794   if(pid == -1)
8795     panic("fork");
8796   return pid;
8797 }
8798
8799
```

```
8800 // Constructors
8801
8802 struct cmd*
8803 execcmd(void)
8804 {
8805   struct execcmd *cmd;
8806
8807   cmd = malloc(sizeof(*cmd));
8808   memset(cmd, 0, sizeof(*cmd));
8809   cmd->type = EXEC;
8810   return (struct cmd*)cmd;
8811 }
8812
8813 struct cmd*
8814 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
8815 {
8816   struct redircmd *cmd;
8817
8818   cmd = malloc(sizeof(*cmd));
8819   memset(cmd, 0, sizeof(*cmd));
8820   cmd->type = REDIR;
8821   cmd->cmd = subcmd;
8822   cmd->file = file;
8823   cmd->efile = efile;
8824   cmd->mode = mode;
8825   cmd->fd = fd;
8826   return (struct cmd*)cmd;
8827 }
8828
8829 struct cmd*
8830 pipecmd(struct cmd *left, struct cmd *right)
8831 {
8832   struct pipecmd *cmd;
8833
8834   cmd = malloc(sizeof(*cmd));
8835   memset(cmd, 0, sizeof(*cmd));
8836   cmd->type = PIPE;
8837   cmd->left = left;
8838   cmd->right = right;
8839   return (struct cmd*)cmd;
8840 }
8841
8842
8843
8844
8845
8846
8847
8848
8849
```



```
8850 struct cmd*
8851 listcmd(struct cmd *left, struct cmd *right)
8852 {
8853   struct listcmd *cmd;
8854
8855   cmd = malloc(sizeof(*cmd));
8856   memset(cmd, 0, sizeof(*cmd));
8857   cmd->type = LIST;
8858   cmd->left = left;
8859   cmd->right = right;
8860   return (struct cmd*)cmd;
8861 }
8862
8863 struct cmd*
8864 backcmd(struct cmd *subcmd)
8865 {
8866   struct backcmd *cmd;
8867
8868   cmd = malloc(sizeof(*cmd));
8869   memset(cmd, 0, sizeof(*cmd));
8870   cmd->type = BACK;
8871   cmd->cmd = subcmd;
8872   return (struct cmd*)cmd;
8873 }
8874
8875
8876
8877
8878
8879
8880
8881
8882
8883
8884
8885
8886
8887
8888
8889
8890
8891
8892
8893
8894
8895
8896
8897
8898
8899
```

```
8900 // Parsing
8901
8902 char whitespace[] = " \t\r\n\v";
8903 char symbols[] = "<|>&;()";
8904
8905 int
8906 gettoken(char **ps, char *es, char **q, char **eq)
8907 {
8908   char *s;
8909   int ret;
8910
8911   s = *ps;
8912   while(s < es && strchr(whitespace, *s))
8913     s++;
8914   if(q)
8915     *q = s;
8916   ret = *s;
8917   switch(*s){
8918   case 0:
8919     break;
8920   case '|':
8921   case '(':
8922   case ')':
8923   case ';':
8924   case '&':
8925   case '<':
8926     s++;
8927     break;
8928   case '>':
8929     s++;
8930     if(*s == '>'){
8931       ret = '+';
8932       s++;
8933     }
8934     break;
8935   default:
8936     ret = 'a';
8937     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
8938       s++;
8939     break;
8940   }
8941   if(eq)
8942     *eq = s;
8943
8944   while(s < es && strchr(whitespace, *s))
8945     s++;
8946   *ps = s;
8947   return ret;
8948 }
8949
```



```
8950 int
8951 peek(char **ps, char *es, char *toks)
8952 {
8953   char *s;
8954
8955   s = *ps;
8956   while(s < es && strchr(whitespace, *s))
8957     s++;
8958   *ps = s;
8959   return *s && strchr(toks, *s);
8960 }
8961
8962 struct cmd *parseline(char**, char*);
8963 struct cmd *parsepipe(char**, char*);
8964 struct cmd *parseexec(char**, char*);
8965 struct cmd *nulterminate(struct cmd*);
8966
8967 struct cmd*
8968 parsecmd(char *s)
8969 {
8970   char *es;
8971   struct cmd *cmd;
8972
8973   es = s + strlen(s);
8974   cmd = parseline(&s, es);
8975   peek(&s, es, "");
8976   if(s != es){
8977     printf(2, "leftovers: %s\n", s);
8978     panic("syntax");
8979   }
8980   nulterminate(cmd);
8981   return cmd;
8982 }
8983
8984 struct cmd*
8985 parseline(char **ps, char *es)
8986 {
8987   struct cmd *cmd;
8988
8989   cmd = parsepipe(ps, es);
8990   while(peek(ps, es, "&")){
8991     gettoken(ps, es, 0, 0);
8992     cmd = backcmd(cmd);
8993   }
8994   if(peek(ps, es, ";")){
8995     gettoken(ps, es, 0, 0);
8996     cmd = listcmd(cmd, parseline(ps, es));
8997   }
8998   return cmd;
8999 }
```

```
9000 struct cmd*
9001 parsepipe(char **ps, char *es)
9002 {
9003   struct cmd *cmd;
9004
9005   cmd = parseexec(ps, es);
9006   if(peek(ps, es, "|")){
9007     gettoken(ps, es, 0, 0);
9008     cmd = pipecmd(cmd, parsepipe(ps, es));
9009   }
9010   return cmd;
9011 }
9012
9013 struct cmd*
9014 parseredirs(struct cmd *cmd, char **ps, char *es)
9015 {
9016   int tok;
9017   char *q, *eq;
9018
9019   while(peek(ps, es, "<>")){
9020     tok = gettoken(ps, es, 0, 0);
9021     if(gettoken(ps, es, &q, &eq) != 'a')
9022       panic("missing file for redirection");
9023     switch(tok){
9024     case '<':
9025       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
9026       break;
9027     case '>':
9028       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9029       break;
9030     case '+':  // >>
9031       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9032       break;
9033     }
9034   }
9035   return cmd;
9036 }
9037
9038
9039
9040
9041
9042
9043
9044
9045
9046
9047
9048
9049
```



```
9050 struct cmd*
9051 parseblock(char **ps, char *es)
9052 {
9053   struct cmd *cmd;
9054
9055   if(!peek(ps, es, "("))
9056     panic("parseblock");
9057   gettoken(ps, es, 0, 0);
9058   cmd = parseline(ps, es);
9059   if(!peek(ps, es, ")"))
9060     panic("syntax - missing )");
9061   gettoken(ps, es, 0, 0);
9062   cmd = parseredirs(cmd, ps, es);
9063   return cmd;
9064 }
9065
9066 struct cmd*
9067 parseexec(char **ps, char *es)
9068 {
9069   char *q, *eq;
9070   int tok, argc;
9071   struct execcmd *cmd;
9072   struct cmd *ret;
9073
9074   if(peek(ps, es, "("))
9075     return parseblock(ps, es);
9076
9077   ret = execcmd();
9078   cmd = (struct execcmd*)ret;
9079
9080   argc = 0;
9081   ret = parseredirs(ret, ps, es);
9082   while(!peek(ps, es, "|)&;")){
9083     if((tok=gettoken(ps, es, &q, &eq)) == 0)
9084       break;
9085     if(tok != 'a')
9086       panic("syntax");
9087     cmd->argv[argc] = q;
9088     cmd->eargv[argc] = eq;
9089     argc++;
9090     if(argc >= MAXARGS)
9091       panic("too many args");
9092     ret = parseredirs(ret, ps, es);
9093   }
9094   cmd->argv[argc] = 0;
9095   cmd->eargv[argc] = 0;
9096   return ret;
9097 }
9098
9099
```

```
9100 // NUL-terminate all the counted strings.
9101 struct cmd*
9102 nulterminate(struct cmd *cmd)
9103 {
9104   int i;
9105   struct backcmd *bcmd;
9106   struct execcmd *ecmd;
9107   struct listcmd *lcmd;
9108   struct pipecmd *pcmd;
9109   struct redircmd *rcmd;
9110
9111   if(cmd == 0)
9112     return 0;
9113
9114   switch(cmd->type){
9115   case EXEC:
9116     ecmd = (struct execcmd*)cmd;
9117     for(i=0; ecmd->argv[i]; i++)
9118       *ecmd->eargv[i] = 0;
9119     break;
9120
9121   case REDIR:
9122     rcmd = (struct redircmd*)cmd;
9123     nulterminate(rcmd->cmd);
9124     *rcmd->efile = 0;
9125     break;
9126
9127   case PIPE:
9128     pcmd = (struct pipecmd*)cmd;
9129     nulterminate(pcmd->left);
9130     nulterminate(pcmd->right);
9131     break;
9132
9133   case LIST:
9134     lcmd = (struct listcmd*)cmd;
9135     nulterminate(lcmd->left);
9136     nulterminate(lcmd->right);
9137     break;
9138
9139   case BACK:
9140     bcmd = (struct backcmd*)cmd;
9141     nulterminate(bcmd->cmd);
9142     break;
9143   }
9144   return cmd;
9145 }
9146
9147
9148
9149
```



```
9150 #include "asm.h"
9151 #include "memlayout.h"
9152 #include "mmu.h"
9153
9154 # Start the first CPU: switch to 32-bit protected mode, jump into C.
9155 # The BIOS loads this code from the first sector of the hard disk into
9156 # memory at physical address 0x7c00 and starts executing in real mode
9157 # with %cs=0 %ip=7c00.
9158
9159 .code16                       # Assemble for 16-bit mode
9160 .globl start
9161 start:
9162   cli                         # BIOS enabled interrupts; disable
9163
9164   # Zero data segment registers DS, ES, and SS.
9165   xorw    %ax,%ax             # Set %ax to zero
9166   movw    %ax,%ds             # -> Data Segment
9167   movw    %ax,%es             # -> Extra Segment
9168   movw    %ax,%ss             # -> Stack Segment
9169
9170   # Physical address line A20 is tied to zero so that the first PCs
9171   # with 2 MB would run software that assumed 1 MB.  Undo that.
9172 seta20.1:
9173   inb     $0x64,%al               # Wait for not busy
9174   testb   $0x2,%al
9175   jnz     seta20.1
9176
9177   movb    $0xd1,%al               # 0xd1 -> port 0x64
9178   outb    %al,$0x64
9179
9180 seta20.2:
9181   inb     $0x64,%al               # Wait for not busy
9182   testb   $0x2,%al
9183   jnz     seta20.2
9184
9185   movb    $0xdf,%al               # 0xdf -> port 0x60
9186   outb    %al,$0x60
9187
9188   # Switch from real to protected mode.  Use a bootstrap GDT that makes
9189   # virtual addresses map directly to physical addresses so that the
9190   # effective memory map doesn't change during the transition.
9191   lgdt    gdtdesc
9192   movl    %cr0, %eax
9193   orl     $CR0_PE, %eax
9194   movl    %eax, %cr0
9195
9196
9197
9198
9199
```

```
9200   # Complete transition to 32-bit protected mode by using long jmp
9201   # to reload %cs and %eip.  The segment descriptors are set up with no
9202   # translation, so that the mapping is still the identity mapping.
9203   ljmp    $(SEG_KCODE<<3), $start32
9204
9205 .code32  # Tell assembler to generate 32-bit code now.
9206 start32:
9207   # Set up the protected-mode data segment registers
9208   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
9209   movw    %ax, %ds                # -> DS: Data Segment
9210   movw    %ax, %es                # -> ES: Extra Segment
9211   movw    %ax, %ss                # -> SS: Stack Segment
9212   movw    $0, %ax                 # Zero segments not ready for use
9213   movw    %ax, %fs                # -> FS
9214   movw    %ax, %gs                # -> GS
9215
9216   # Set up the stack pointer and call into C.
9217   movl    $start, %esp
9218   call    bootmain
9219
9220   # If bootmain returns (it shouldn't), trigger a Bochs
9221   # breakpoint if running under Bochs, then loop.
9222   movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
9223   movw    %ax, %dx
9224   outw    %ax, %dx
9225   movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
9226   outw    %ax, %dx
9227 spin:
9228   jmp     spin
9229
9230 # Bootstrap GDT
9231 .p2align 2                                # force 4 byte alignment
9232 gdt:
9233   SEG_NULLASM                             # null seg
9234   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
9235   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
9236
9237 gdtdesc:
9238   .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
9239   .long   gdt                             # address gdt
9240
9241
9242
9243
9244
9245
9246
9247
9248
9249
```



```
9250 // Boot loader.
9251 //
9252 // Part of the boot block, along with bootasm.S, which calls bootmain().
9253 // bootasm.S has put the processor into protected 32-bit mode.
9254 // bootmain() loads an ELF kernel image from the disk starting at
9255 // sector 1 and then jumps to the kernel entry routine.
9256
9257 #include "types.h"
9258 #include "elf.h"
9259 #include "x86.h"
9260 #include "memlayout.h"
9261
9262 #define SECTSIZE  512
9263
9264 void readseg(uchar*, uint, uint);
9265
9266 void
9267 bootmain(void)
9268 {
9269   struct elfhdr *elf;
9270   struct proghdr *ph, *eph;
9271   void (*entry)(void);
9272   uchar* pa;
9273
9274   elf = (struct elfhdr*)0x10000;  // scratch space
9275
9276   // Read 1st page off disk
9277   readseg((uchar*)elf, 4096, 0);
9278
9279   // Is this an ELF executable?
9280   if(elf->magic != ELF_MAGIC)
9281     return;  // let bootasm.S handle error
9282
9283   // Load each program segment (ignores ph flags).
9284   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
9285   eph = ph + elf->phnum;
9286   for(; ph < eph; ph++){
9287     pa = (uchar*)ph->paddr;
9288     readseg(pa, ph->filesz, ph->off);
9289     if(ph->memsz > ph->filesz)
9290       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
9291   }
9292
9293   // Call the entry point from the ELF header.
9294   // Does not return!
9295   entry = (void(*)(void))(elf->entry);
9296   entry();
9297 }
9298
9299
```

```
9300 void
9301 waitdisk(void)
9302 {
9303   // Wait for disk ready.
9304   while((inb(0x1F7) & 0xC0) != 0x40)
9305     ;
9306 }
9307
9308 // Read a single sector at offset into dst.
9309 void
9310 readsect(void *dst, uint offset)
9311 {
9312   // Issue command.
9313   waitdisk();
9314   outb(0x1F2, 1);   // count = 1
9315   outb(0x1F3, offset);
9316   outb(0x1F4, offset >> 8);
9317   outb(0x1F5, offset >> 16);
9318   outb(0x1F6, (offset >> 24) | 0xE0);
9319   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
9320
9321   // Read data.
9322   waitdisk();
9323   insl(0x1F0, dst, SECTSIZE/4);
9324 }
9325
9326 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
9327 // Might copy more than asked.
9328 void
9329 readseg(uchar* pa, uint count, uint offset)
9330 {
9331   uchar* epa;
9332
9333   epa = pa + count;
9334
9335   // Round down to sector boundary.
9336   pa -= offset % SECTSIZE;
9337
9338   // Translate from bytes to sectors; kernel starts at sector 1.
9339   offset = (offset / SECTSIZE) + 1;
9340
9341   // If this is too slow, we could read lots of sectors at a time.
9342   // We'd write more to memory than asked, but it doesn't matter --
9343   // we load in increasing order.
9344   for(; pa < epa; pa += SECTSIZE, offset++)
9345     readsect(pa, offset);
9346 }
9347
9348
9349
```



```
9350 #include "types.h"
9351 #include "user.h"
9352 #include "date.h"
9353
9354 int main (int argc, char *argv[])
9355 {
9356     struct rtcdate r;
9357
9358     if(date(&r))
9359     {
9360         printf(2, "date failed \n" ) ;
9361         exit();
9362     }
9363
9364     //CODE to print time in any format
9365     printf(1, "%d:%d:%d    %d/%d/%d\n", r.hour, r.minute, r.second, r.month,
9366     exit();
9367 }
9368
9369
9370
9371
9372
9373
9374
9375
9376
9377
9378
9379
9380
9381
9382
9383
9384
9385
9386
9387
9388
9389
9390
9391
9392
9393
9394
9395
9396
9397
9398
9399
```

```
9400 struct rtcdate {
9401   uint second;
9402   uint minute;
9403   uint hour;
9404   uint day;
9405   uint month;
9406   uint year;
9407 };
9408
9409
9410
9411
9412
9413
9414
9415
9416
9417
9418
9419
9420
9421
9422
9423
9424
9425
9426
9427
9428
9429
9430
9431
9432
9433
9434
9435
9436
9437
9438
9439
9440
9441
9442
9443
9444
9445
9446
9447
9448
9449
```



```
9450 #include "types.h"
9451 #include "user.h"
9452 #include "date.h"
9453
9454 int main (int argc, char *argv[])
9455 {
9456     struct rtcdate r1;
9457     struct rtcdate r2;
9458     int pid = 0;
9459     int hour = 0;
9460     int minute = 0;
9461     int second = 0;
9462
9463     if(date(&r1))           //Get time start
9464     {
9465         printf(2, "date failed \n" ) ;
9466         exit();
9467     }
9468
9469     pid = fork();
9470     if(pid > 0)             //parent exits and waits for child process to ex
9471     {
9472         pid = wait();
9473         if(date(&r2))       //Get time finish
9474         {
9475             printf(2, "date failed \n" ) ;
9476             exit();
9477         }
9478     }
9479     else if(pid == 0)   //child exits
9480     {
9481         exec(argv[1], argv+2); //run the process with name located in argv[1
9482         if(date(&r2))       //Get time finish
9483         {
9484             printf(2, "date failed \n" ) ;
9485             exit();
9486         }
9487     }
9488     else
9489     {
9490         printf(0, "fork error\n");
9491     }
9492
9493     hour = r2.hour - r1.hour;
9494     minute = r2.minute - r1.minute;
9495
9496     if(r2.second > r1.second)
9497         second = r2.second - r1.second;
9498     else
9499         second = r1.second - r2.second;
```

```
9500     //Print elapsed time
9501     printf(1, "Elapsed Time: %d hours %d minutes %d seconds\n", hour, minute,
9502     exit();
9503 }
9504
9505
9506
9507
9508
9509
9510
9511
9512
9513
9514
9515
9516
9517
9518
9519
9520
9521
9522
9523
9524
9525
9526
9527
9528
9529
9530
9531
9532
9533
9534
9535
9536
9537
9538
9539
9540
9541
9542
9543
9544
9545
9546
9547
9548
9549
```



```
9550 struct stat;
9551 struct rtcdate;
9552
9553 // system calls
9554 int fork(void);
9555 int exit(void) __attribute__((noreturn));
9556 int wait(void);
9557 int pipe(int*);
9558 int write(int, void*, int);
9559 int read(int, void*, int);
9560 int close(int);
9561 int kill(int);
9562 int exec(char*, char**);
9563 int open(char*, int);
9564 int mknod(char*, short, short);
9565 int unlink(char*);
9566 int fstat(int fd, struct stat*);
9567 int link(char*, char*);
9568 int mkdir(char*);
9569 int chdir(char*);
9570 int dup(int);
9571 int getpid(void);
9572 char* sbrk(int);
9573 int sleep(int);
9574 int uptime(void);
9575 int halt(void);
9576 //Defined date function that allows user to call through shell
9577 int date(struct rtcdate*);
9578
9579
9580 // ulib.c
9581 int stat(char*, struct stat*);
9582 char* strcpy(char*, char*);
9583 void *memmove(void*, void*, int);
9584 char* strchr(const char*, char c);
9585 int strcmp(const char*, const char*);
9586 void printf(int, char*, ...);
9587 char* gets(char*, int max);
9588 uint strlen(char*);
9589 void* memset(void*, int, uint);
9590 void* malloc(uint);
9591 void free(void*);
9592 int atoi(const char*);
9593
9594
9595
9596
9597
9598
9599
```