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

```
0300 struct inode*   nameiparent(char*, char*);
0301 int             readi(struct inode*, char*, uint, uint);
0302 void            stati(struct inode*, struct stat*);
0303 int             writei(struct inode*, char*, uint, uint);
0304
0305 // ide.c
0306 void            ideinit(void);
0307 void            ideintr(void);
0308 void            iderw(struct buf*);
0309
0310 // ioapic.c
0311 void            ioapicenable(int irq, int cpu);
0312 extern uchar    ioapicid;
0313 void            ioapicinit(void);
0314
0315 // kalloc.c
0316 char*           kalloc(void);
0317 void            kfree(char*);
0318 void            kinit1(void*, void*);
0319 void            kinit2(void*, void*);
0320
0321 // kbd.c
0322 void            kbdintr(void);
0323
0324 // lapic.c
0325 void            cmostime(struct rtcdate *r);
0326 int             cpunum(void);
0327 extern volatile uint*    lapic;
0328 void            lapiceoi(void);
0329 void            lapicinit(void);
0330 void            lapicstartap(uchar, uint);
0331 void            microdelay(int);
0332
0333 // log.c
0334 void            initlog(int dev);
0335 void            log_write(struct buf*);
0336 void            begin_op();
0337 void            end_op();
0338
0339 // mp.c
0340 extern int      ismp;
0341 int             mpbcpu(void);
0342 void            mpinit(void);
0343 void            mpstartthem(void);
0344
0345 // picirq.c
0346 void            picenable(int);
0347 void            picinit(void);
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 int             getprocs(int, struct uproc*);
0373 int                setuid(int);
0374 int                setgid(int);
0375 int                getuid();
0376 int                getgid();
0377 int             addtoq();
0378 int             putinQ(struct proc *);
0379 // swtch.S
0380 void            swtch(struct context**, struct context*);
0381
0382 // spinlock.c
0383 void            acquire(struct spinlock*);
0384 void            getcallerpcs(void*, uint*);
0385 int             holding(struct spinlock*);
0386 void            initlock(struct spinlock*, char*);
0387 void            release(struct spinlock*);
0388 void            pushcli(void);
0389 void            popcli(void);
0390
0391 // string.c
0392 int             memcmp(const void*, const void*, uint);
0393 void*           memmove(void*, const void*, uint);
0394 void*           memset(void*, int, uint);
0395 char*           safestrcpy(char*, const char*, int);
0396 int             strlen(const char*);
0397 int             strncmp(const char*, const char*, uint);
0398 char*           strncpy(char*, const char*, int);
0399
```

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

```
# Multiboot header, for multiboot boot loaders like GNU Grub.
# http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
#
# Using GRUB 2, you can boot xv6 from a file stored in a
# Linux file system by copying kernel or kernelmemfs to /boot
# and then adding this menu entry:
#
# menuentry "xv6" {
#  insmod ext2
#  set root='(hd0,msdos1)'
#  set kernel='/boot/kernel'
#  echo "Loading ${kernel}..."
#  multiboot ${kernel} ${kernel}
#  boot
# }

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

# Multiboot header.  Data to direct multiboot loader.
.p2align 2
.text
.globl multiboot_header
multiboot_header:
  #define magic 0x1badb002
  #define flags 0
  .long magic
  .long flags
  .long (-magic-flags)

# By convention, the _start symbol specifies the ELF entry point.
# Since we haven't set up virtual memory yet, our entry point is
# the physical address of 'entry'.
.globl _start
_start = V2P_WO(entry)

# Entering xv6 on boot processor, with paging off.
.globl entry
entry:
  # Turn on page size extension for 4Mbyte pages
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  # Set page directory
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  # Turn on paging.
  movl    %cr0, %eax
```



```
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0

  # Set up the stack pointer.
  movl $(stack + KSTACKSIZE), %esp

  # Jump to main(), and switch to executing at
  # high addresses. The indirect call is needed because
  # the assembler produces a PC-relative instruction
  # for a direct jump.
  mov $main, %eax
  jmp *%eax

.comm stack, KSTACKSIZE
```

```
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 #define UID_DEFAULT 7
2303 #define GID_DEFAULT 5
2304
2305 // Per-CPU state
2306 struct cpu {
2307   uchar id;                    // Local APIC ID; index into cpus[] below
2308   struct context *scheduler;   // swtch() here to enter scheduler
2309   struct taskstate ts;         // Used by x86 to find stack for interrupt
2310   struct segdesc gdt[NSEGS];   // x86 global descriptor table
2311   volatile uint started;       // Has the CPU started?
2312   int ncli;                    // Depth of pushcli nesting.
2313   int intena;                  // Were interrupts enabled before pushcli?
2314
2315   // Cpu-local storage variables; see below
2316   struct cpu *cpu;
2317   struct proc *proc;           // The currently-running process.
2318 };
2319
2320 extern struct cpu cpus[NCPU];
2321 extern int ncpu;
2322
2323 // Per-CPU variables, holding pointers to the
2324 // current cpu and to the current process.
2325 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2326 // and "%gs:4" to refer to proc.  seginit sets up the
2327 // %gs segment register so that %gs refers to the memory
2328 // holding those two variables in the local cpu's struct cpu.
2329 // This is similar to how thread-local variables are implemented
2330 // in thread libraries such as Linux pthreads.
2331 extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
2332 extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
```



```
2350 // Saved registers for kernel context switches.
2351 // Don't need to save all the segment registers (%cs, etc),
2352 // because they are constant across kernel contexts.
2353 // Don't need to save %eax, %ecx, %edx, because the
2354 // x86 convention is that the caller has saved them.
2355 // Contexts are stored at the bottom of the stack they
2356 // describe; the stack pointer is the address of the context.
2357 // The layout of the context matches the layout of the stack in swtch.S
2358 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2359 // but it is on the stack and allocproc() manipulates it.
2360 struct context {
2361   uint edi;
2362   uint esi;
2363   uint ebx;
2364   uint ebp;
2365   uint eip;
2366 };
2367
2368 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2369 //no one can use processes in embryo state.
2370 // Per-process state
2371 struct proc {
2372   uint sz;                     // Size of process memory (bytes)
2373   pde_t* pgdir;                // Page table
2374   char *kstack;                // Bottom of kernel stack for this process
2375   enum procstate state;        // Process state
2376   int pid;                     // Process ID
2377   uint uid;                    // User ID
2378   uint gid;                    // Group ID
2379   int ppid;
2380   struct proc *parent;         // Parent process
2381   struct trapframe *tf;        // Trap frame for current syscall
2382   struct context *context;     // swtch() here to run process
2383   void *chan;                  // If non-zero, sleeping on chan
2384   int killed;                  // If non-zero, have been killed
2385   struct file *ofile[NOFILE];  // Open files
2386   struct inode *cwd;           // Current directory
2387   char name[16];               // Process name (debugging)
2388   struct proc *next;
2389   int priority;
2390 };
2391
2392
2393
2394
2395 // Process memory is laid out contiguously, low addresses first:
2396 //   text
2397 //   original data and bss
2398 //   fixed-size stack
2399 //   expandable heap
```

```
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 #include "ps.h"
2409
2410 #define NULL 0
2411
2412 static char *states[] = {
2413   [UNUSED]    "UNUSED",
2414   [EMBRYO]    "EMBRYO",
2415   [SLEEPING]  "SLEEPING",
2416   [RUNNABLE]  "RUNNABLE",
2417   [RUNNING]   "RUNNING",
2418   [ZOMBIE]    "ZOMBIE"
2419   };
2420
2421
2422 //uses round robin
2423 //nproc is set to 64
2424 struct {
2425   struct spinlock lock;
2426   struct proc proc[NPROC];
2427   struct proc *pReadyList[2];
2428 //  struct proc *pFreeList;
2429 //  uint TimeToReset;
2430 } ptable;
2431
2432 static struct proc *initproc;
2433
2434 int nextpid = 1;
2435 extern void forkret(void);
2436 extern void trapret(void);
2437
2438 static void wakeup1(void *chan);
2439
2440 void
2441 pinit(void)
2442 {
2443   initlock(&ptable.lock, "ptable");
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   p->uid = UID_DEFAULT;
2527   p->gid = GID_DEFAULT;
2528
2529   p->ppid = 1;
2530   p->priority = 1;
2531   ptable.pReadyList[1] = p;
2532   p->next = NULL;
2533
2534 }
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
```



```
2550 // Grow current process's memory by n bytes.
2551 // Return 0 on success, -1 on failure.
2552 int
2553 growproc(int n)
2554 {
2555   uint sz;
2556
2557   sz = proc->sz;
2558   if(n > 0){
2559     if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2560       return -1;
2561   } else if(n < 0){
2562     if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2563       return -1;
2564   }
2565   proc->sz = sz;
2566   switchuvm(proc);
2567   return 0;
2568 }
2569
2570 // Create a new process copying p as the parent.
2571 // Sets up stack to return as if from system call.
2572 // Caller must set state of returned proc to RUNNABLE.
2573 int
2574 fork(void)
2575 {
2576   int i, pid;
2577   struct proc *np;
2578
2579   // Allocate process.
2580   if((np = allocproc()) == 0)
2581     return -1;
2582
2583   // Copy process state from p.
2584   if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2585     kfree(np->kstack);
2586     np->kstack = 0;
2587     np->state = UNUSED;
2588     return -1;
2589   }
2590   np->sz = proc->sz;
2591   np->parent = proc;
2592   *np->tf = *proc->tf;
2593
2594   // Clear %eax so that fork returns 0 in the child.
2595   np->tf->eax = 0;
2596
2597
2598
2599
```

```
2600   for(i = 0; i < NOFILE; i++)
2601     if(proc->ofile[i])
2602       np->ofile[i] = filedup(proc->ofile[i]);
2603   np->cwd = idup(proc->cwd);
2604
2605   safestrcpy(np->name, proc->name, sizeof(proc->name));
2606
2607 //  acquire(&ptable.lock);
2608   np->uid = proc->uid;
2609   np->gid = proc->gid;
2610   np->ppid = proc->pid;
2611   np->priority = 1;
2612 //  release(&ptable.lock);
2613
2614
2615
2616   pid = np->pid;
2617
2618
2619   // lock to force the compiler to emit the np->state write last.
2620   acquire(&ptable.lock);
2621   np->state = RUNNABLE;
2622   putinQ(np);
2623
2624   release(&ptable.lock);
2625
2626   return pid;
2627 }
2628
2629 // Exit the current process.  Does not return.
2630 // An exited process remains in the zombie state
2631 // until its parent calls wait() to find out it exited.
2632 void
2633 exit(void)
2634 {
2635   struct proc *p;
2636   int fd;
2637
2638   if(proc == initproc)
2639     panic("init exiting");
2640
2641   // Close all open files.
2642   for(fd = 0; fd < NOFILE; fd++){
2643     if(proc->ofile[fd]){
2644       fileclose(proc->ofile[fd]);
2645       proc->ofile[fd] = 0;
2646     }
2647   }
2648
2649
```



```
2650   begin_op();
2651   iput(proc->cwd);
2652   end_op();
2653   proc->cwd = 0;
2654
2655   acquire(&ptable.lock);
2656
2657   // Parent might be sleeping in wait().
2658   wakeup1(proc->parent);
2659
2660   // Pass abandoned children to init.
2661   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2662     if(p->parent == proc){
2663       p->parent = initproc;
2664       if(p->state == ZOMBIE)
2665         wakeup1(initproc);
2666     }
2667   }
2668
2669   // Jump into the scheduler, never to return.
2670   proc->state = ZOMBIE;
2671   sched();
2672   panic("zombie exit");
2673 }
2674
2675 // Wait for a child process to exit and return its pid.
2676 // Return -1 if this process has no children.
2677 int
2678 wait(void)
2679 {
2680   struct proc *p;
2681   int havekids, pid;
2682
2683   acquire(&ptable.lock);
2684   for(;;){
2685     // Scan through table looking for zombie children.
2686     havekids = 0;
2687     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2688       if(p->parent != proc)
2689         continue;
2690       havekids = 1;
2691       if(p->state == ZOMBIE){
2692         // Found one.
2693         pid = p->pid;
2694         kfree(p->kstack);
2695         p->kstack = 0;
2696         freevm(p->pgdir);
2697         p->state = UNUSED;
2698         p->pid = 0;
2699         p->parent = 0;
```

```
2700         p->name[0] = 0;
2701         p->killed = 0;
2702         release(&ptable.lock);
2703         return pid;
2704       }
2705     }
2706
2707     // No point waiting if we don't have any children.
2708     if(!havekids || proc->killed){
2709       release(&ptable.lock);
2710       return -1;
2711     }
2712
2713     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2714     sleep(proc, &ptable.lock);
2715   }
2716 }
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
```



```
2750 // Per-CPU process scheduler.
2751 // Each CPU calls scheduler() after setting itself up.
2752 // Scheduler never returns.  It loops, doing:
2753 //  - choose a process to run
2754 //  - swtch to start running that process
2755 //  - eventually that process transfers control
2756 //      via swtch back to the scheduler.
2757 /*void
2758 scheduler(void)
2759 {
2760   struct proc *p;
2761
2762   for(;;){
2763     // Enable interrupts on this processor.
2764     sti();
2765
2766     // Loop over process table looking for process to run.
2767     acquire(&ptable.lock);
2768     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2769       if(p->state != RUNNABLE)
2770         continue;
2771
2772       // Switch to chosen process.  It is the process's job
2773       // to release ptable.lock and then reacquire it
2774       // before jumping back to us.
2775
2776
2777       proc = p;
2778       switchuvm(p);
2779       p->state = RUNNING;
2780       swtch(&cpu->scheduler, proc->context);//context swtch. proc->context ri
2781       switchkvm();//switch to the correct kernel virtual memory.
2782
2783       // Process is done running for now.
2784       // It should have changed its p->state before coming back.
2785       proc = 0;
2786     }
2787     release(&ptable.lock);
2788
2789   }
2790 }*/
2791
2792 // Enter scheduler.  Must hold only ptable.lock
2793 // and have changed proc->state.
2794 void
2795 sched(void)
2796 {
2797   int intena;
2798
2799   if(!holding(&ptable.lock))
```

```
2800     panic("sched ptable.lock");
2801   if(cpu->ncli != 1)
2802     panic("sched locks");
2803   if(proc->state == RUNNING)
2804     panic("sched running");
2805   if(readeflags()&FL_IF)
2806     panic("sched interruptible");
2807   intena = cpu->intena;
2808   swtch(&proc->context, cpu->scheduler);
2809   cpu->intena = intena;
2810 }
2811
2812 // Give up the CPU for one scheduling round.
2813 void
2814 yield(void)
2815 {
2816   acquire(&ptable.lock);
2817   proc->state = RUNNABLE;
2818   putinQ(proc);
2819   sched();
2820   release(&ptable.lock);
2821 }
2822
2823 // A fork child's very first scheduling by scheduler()
2824 // will swtch here.  "Return" to user space.
2825 void
2826 forkret(void)
2827 {
2828   static int first = 1;
2829   // Still holding ptable.lock from scheduler.
2830   release(&ptable.lock);
2831
2832   if (first) {
2833     // Some initialization functions must be run in the context
2834     // of a regular process (e.g., they call sleep), and thus cannot
2835     // be run from main().
2836     first = 0;
2837     iinit(ROOTDEV);
2838     initlog(ROOTDEV);
2839   }
2840
2841   // Return to "caller", actually trapret (see allocproc).
2842 }
2843
2844
2845
2846
2847
2848
2849
```



```
2850 // Atomically release lock and sleep on chan.
2851 // Reacquires lock when awakened.
2852 void
2853 sleep(void *chan, struct spinlock *lk)
2854 {
2855   if(proc == 0)
2856     panic("sleep");
2857
2858   if(lk == 0)
2859     panic("sleep without lk");
2860
2861   // Must acquire ptable.lock in order to
2862   // change p->state and then call sched.
2863   // Once we hold ptable.lock, we can be
2864   // guaranteed that we won't miss any wakeup
2865   // (wakeup runs with ptable.lock locked),
2866   // so it's okay to release lk.
2867   if(lk != &ptable.lock){
2868     acquire(&ptable.lock);
2869     release(lk);
2870   }
2871
2872   // Go to sleep.
2873   proc->chan = chan;
2874   proc->state = SLEEPING;
2875   sched();
2876
2877   // Tidy up.
2878   proc->chan = 0;
2879
2880   // Reacquire original lock.
2881   if(lk != &ptable.lock){
2882     release(&ptable.lock);
2883     acquire(lk);
2884   }
2885 }
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
```

```
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan){
      p->state = RUNNABLE;
      putinQ(p);
    }
}

// Wake up all processes sleeping on chan.
void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}

// Kill the process with the given pid.
// Process won't exit until it returns
// to user space (see trap in trap.c).
int
kill(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING){
        p->state = RUNNABLE;
   putinQ(p);
      }
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```



```
int
setuid(int uid)
{
    if(uid<0)
   return -1;

    acquire(&ptable.lock);
    proc->uid = uid;
    release(&ptable.lock);

    return 0;

}

int
setgid(int gid)
{

    if(gid<0)
   return -1;

    acquire(&ptable.lock);
    proc->gid = gid;
    release(&ptable.lock);

    return 0;
}

int
getuid()
{
    return proc->uid;
}

int
getgid()
{
    return proc->gid;
}
```

```
3000 // Print a process listing to console.  For debugging.
3001 // Runs when user types ^P on console.
3002 // No lock to avoid wedging a stuck machine further.
3003 void
3004 procdump(void)
3005 {
3006   int i;
3007   struct proc *p;
3008   char *state;
3009   uint pc[10];
3010
3011   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3012     if(p->state == UNUSED)
3013       continue;
3014     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
3015       state = states[p->state];
3016     else
3017       state = "???";
3018     cprintf("%d %d %d %s %s", p->pid, p->uid, p->gid, state, p->name);
3019     if(p->state == SLEEPING){
3020       getcallerpcs((uint*)p->context->ebp+2, pc);
3021       for(i=0; i<10 && pc[i] != 0; i++)
3022         cprintf(" %p", pc[i]);
3023     }
3024     cprintf("\n");
3025   }
3026 }
3027
3028 int
3029 getprocs(int max, struct uproc *table)
3030 {
3031     int count=0;
3032     struct proc *p;
3033
3034     acquire(&ptable.lock);
3035     for(p=ptable.proc;p<&ptable.proc[NPROC]; p++){
3036
3037    if(p->state == UNUSED || p->state == ZOMBIE || p->state == EMBRYO)
3038        continue;
3039    if(count >= max){
3040        release(&ptable.lock);
3041        return count;
3042    }
3043
3044        table[count].pid = p->pid;
3045        table[count].uid = p->uid;
3046        table[count].gid = p->gid;
3047        table[count].ppid = p->ppid;
3048        safestrcpy(table[count].state, states[p->state], sizeof(table[count].s
3049        table[count].size = p->sz;
```



```
3050        safestrcpy(table[count].name, p->name, sizeof(table[count].name));
3051
3052
3053        count = count+1;
3054    }
3055
3056
3057     release(&ptable.lock);
3058     if(max >= count)
3059    return count;
3060
3061     return -1;
3062
3063 }
3064
3065 void scheduler(void){
3066   struct proc *p;
3067   int i;
3068   for(;;){
3069     // Enable interrupts on this processor.
3070     sti();
3071
3072     // Loop over process table looking for process to run.
3073     acquire(&ptable.lock);
3074
3075       p=ptable.pReadyList[0];
3076       i = 0;
3077       if(p == NULL){
3078    p=ptable.pReadyList[1];
3079    i=1;
3080    if(p == NULL){
3081        p=ptable.pReadyList[2];
3082        i=2;
3083    }
3084       }
3085
3086       if(p!=NULL){
3087       proc = p;
3088
3089       ptable.pReadyList[i]= ptable.pReadyList[i]->next;
3090       p->next = NULL;
3091       switchuvm(p);
3092       p->state = RUNNING;
3093       swtch(&cpu->scheduler, proc->context);//context swtch. proc->context ri
3094       switchkvm();//switch to the correct kernel virtual memory.
3095
3096       // Process is done running for now.
3097       // It should have changed its p->state before coming back.
3098       proc = 0;
3099     }
```

```
3100     release(&ptable.lock);
3101   }
3102 }
3103
3104
3105 int putinQ(struct proc *prc){
3106
3107     acquire(&ptable.lock);
3108
3109     if(prc->priority == 0){
3110    addtoq(&ptable.pReadyList[0], prc);
3111    release(&ptable.lock);
3112    return 0;
3113     }
3114     else if (prc->priority == 1){
3115    addtoq(&ptable.pReadyList[1], prc);
3116    release(&ptable.lock);
3117    return 0;
3118     }
3119     else if(prc->priority == 2){
3120    addtoq(&ptable.pReadyList[2], prc);
3121    release(&ptable.lock);
3122    return 0;
3123     }
3124
3125    return 1;
3126
3127 }
3128
3129 int addtoq(struct proc **p, struct proc *prc){
3130     if(!*p)
3131    *p=prc;
3132     else
3133     {
3134    struct proc *current;
3135    current = *p;
3136    while(current->next != NULL)
3137        current = current->next;
3138
3139    current->next = prc;
3140    prc->next = NULL;
3141
3142     }
3143
3144     return 0;
3145 }
3146
3147
3148
3149
```



```
3150 # Context switch
3151 #
3152 #   void swtch(struct context **old, struct context *new);
3153 #
3154 # Save current register context in old
3155 # and then load register context from new.
3156
3157 .globl swtch
3158 swtch:
3159   movl 4(%esp), %eax
3160   movl 8(%esp), %edx
3161
3162   # Save old callee-save registers
3163   pushl %ebp
3164   pushl %ebx
3165   pushl %esi
3166   pushl %edi
3167
3168   # Switch stacks
3169   movl %esp, (%eax)
3170   movl %edx, %esp
3171
3172   # Load new callee-save registers
3173   popl %edi
3174   popl %esi
3175   popl %ebx
3176   popl %ebp
3177   ret
3178
3179
3180
3181
3182
3183
3184
3185
3186
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
```

```
3200 // Physical memory allocator, intended to allocate
3201 // memory for user processes, kernel stacks, page table pages,
3202 // and pipe buffers. Allocates 4096-byte pages.
3203
3204 #include "types.h"
3205 #include "defs.h"
3206 #include "param.h"
3207 #include "memlayout.h"
3208 #include "mmu.h"
3209 #include "spinlock.h"
3210
3211 void freerange(void *vstart, void *vend);
3212 extern char end[]; // first address after kernel loaded from ELF file
3213
3214 struct run {
3215   struct run *next;
3216 };
3217
3218 struct {
3219   struct spinlock lock;
3220   int use_lock;
3221   struct run *freelist;
3222 } kmem;
3223
3224 // Initialization happens in two phases.
3225 // 1. main() calls kinit1() while still using entrypgdir to place just
3226 // the pages mapped by entrypgdir on free list.
3227 // 2. main() calls kinit2() with the rest of the physical pages
3228 // after installing a full page table that maps them on all cores.
3229 void
3230 kinit1(void *vstart, void *vend)
3231 {
3232   initlock(&kmem.lock, "kmem");
3233   kmem.use_lock = 0;
3234   freerange(vstart, vend);
3235 }
3236
3237 void
3238 kinit2(void *vstart, void *vend)
3239 {
3240   freerange(vstart, vend);
3241   kmem.use_lock = 1;
3242 }
3243
3244
3245
3246
3247
3248
3249
```



```
3250 void
3251 freerange(void *vstart, void *vend)
3252 {
3253   char *p;
3254   p = (char*)PGROUNDUP((uint)vstart);
3255   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3256     kfree(p);
3257 }
3258
3259
3260 // Free the page of physical memory pointed at by v,
3261 // which normally should have been returned by a
3262 // call to kalloc().  (The exception is when
3263 // initializing the allocator; see kinit above.)
3264 void
3265 kfree(char *v)
3266 {
3267   struct run *r;
3268
3269   if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
3270     panic("kfree");
3271
3272   // Fill with junk to catch dangling refs.
3273   memset(v, 1, PGSIZE);
3274
3275   if(kmem.use_lock)
3276     acquire(&kmem.lock);
3277   r = (struct run*)v;
3278   r->next = kmem.freelist;
3279   kmem.freelist = r;
3280   if(kmem.use_lock)
3281     release(&kmem.lock);
3282 }
3283
3284 // Allocate one 4096-byte page of physical memory.
3285 // Returns a pointer that the kernel can use.
3286 // Returns 0 if the memory cannot be allocated.
3287 char*
3288 kalloc(void)
3289 {
3290   struct run *r;
3291
3292   if(kmem.use_lock)
3293     acquire(&kmem.lock);
3294   r = kmem.freelist;
3295   if(r)
3296     kmem.freelist = r->next;
3297   if(kmem.use_lock)
3298     release(&kmem.lock);
3299   return (char*)r;
```

```
3300 }
3301
3302
3303
3304
3305
3306
3307
3308
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
```



```
3350 // x86 trap and interrupt constants.
3351
3352 // Processor-defined:
3353 #define T_DIVIDE         0      // divide error
3354 #define T_DEBUG          1      // debug exception
3355 #define T_NMI            2      // non-maskable interrupt
3356 #define T_BRKPT          3      // breakpoint
3357 #define T_OFLOW          4      // overflow
3358 #define T_BOUND          5      // bounds check
3359 #define T_ILLOP          6      // illegal opcode
3360 #define T_DEVICE         7      // device not available
3361 #define T_DBLFLT         8      // double fault
3362 // #define T_COPROC      9      // reserved (not used since 486)
3363 #define T_TSS           10      // invalid task switch segment
3364 #define T_SEGNP         11      // segment not present
3365 #define T_STACK         12      // stack exception
3366 #define T_GPFLT         13      // general protection fault
3367 #define T_PGFLT         14      // page fault
3368 // #define T_RES        15      // reserved
3369 #define T_FPERR         16      // floating point error
3370 #define T_ALIGN         17      // aligment check
3371 #define T_MCHK          18      // machine check
3372 #define T_SIMDERR       19      // SIMD floating point error
3373
3374 // These are arbitrarily chosen, but with care not to overlap
3375 // processor defined exceptions or interrupt vectors.
3376 #define T_SYSCALL       64      // system call
3377 #define T_DEFAULT      500      // catchall
3378
3379 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
3380
3381 #define IRQ_TIMER        0
3382 #define IRQ_KBD          1
3383 #define IRQ_COM1         4
3384 #define IRQ_IDE         14
3385 #define IRQ_ERROR       19
3386 #define IRQ_SPURIOUS    31
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399
```

```
#!/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
```

```
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 "traps.h"
3508 #include "spinlock.h"
3509
3510 // Interrupt descriptor table (shared by all CPUs).
3511 struct gatedesc idt[256];
3512 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
3513 struct spinlock tickslock;
3514 uint ticks;
3515
3516 void
3517 tvinit(void)
3518 {
3519   int i;
3520
3521   for(i = 0; i < 256; i++)
3522     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3523   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3524
3525   initlock(&tickslock, "time");
3526 }
3527
3528 void
3529 idtinit(void)
3530 {
3531   lidt(idt, sizeof(idt));
3532 }
3533
3534
3535
3536
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
```



```
3550 void
3551 trap(struct trapframe *tf)
3552 {
3553   if(tf->trapno == T_SYSCALL){
3554     if(proc->killed)
3555       exit();
3556     proc->tf = tf;
3557     syscall();
3558     if(proc->killed)
3559       exit();
3560     return;
3561   }
3562
3563   switch(tf->trapno){
3564   case T_IRQ0 + IRQ_TIMER:
3565     if(cpu->id == 0){
3566       acquire(&tickslock);
3567       ticks++;
3568       wakeup(&ticks);
3569       release(&tickslock);
3570     }
3571     lapiceoi();
3572     break;
3573   case T_IRQ0 + IRQ_IDE:
3574     ideintr();
3575     lapiceoi();
3576     break;
3577   case T_IRQ0 + IRQ_IDE+1:
3578     // Bochs generates spurious IDE1 interrupts.
3579     break;
3580   case T_IRQ0 + IRQ_KBD:
3581     kbdintr();
3582     lapiceoi();
3583     break;
3584   case T_IRQ0 + IRQ_COM1:
3585     uartintr();
3586     lapiceoi();
3587     break;
3588   case T_IRQ0 + 7:
3589   case T_IRQ0 + IRQ_SPURIOUS:
3590     cprintf("cpu%d: spurious interrupt at %x:%x\n",
3591             cpu->id, tf->cs, tf->eip);
3592     lapiceoi();
3593     break;
3594
3595
3596
3597
3598
3599
```

```
3600   default:
3601     if(proc == 0 || (tf->cs&3) == 0){
3602       // In kernel, it must be our mistake.
3603       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3604               tf->trapno, cpu->id, tf->eip, rcr2());
3605       panic("trap");
3606     }
3607     // In user space, assume process misbehaved.
3608     cprintf("pid %d %s: trap %d err %d on cpu %d "
3609             "eip 0x%x addr 0x%x--kill proc\n",
3610             proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
3611             rcr2());
3612     proc->killed = 1;
3613   }
3614
3615   // Force process exit if it has been killed and is in user space.
3616   // (If it is still executing in the kernel, let it keep running
3617   // until it gets to the regular system call return.)
3618   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3619     exit();
3620
3621   // Force process to give up CPU on clock tick.
3622   // If interrupts were on while locks held, would need to check nlock.
3623   if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3624     yield();
3625
3626   // Check if the process has been killed since we yielded
3627   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3628     exit();
3629 }
3630
3631
3632
3633
3634
3635
3636
3637
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649
```



```
3650 // System call numbers
3651 #define SYS_fork    1
3652 #define SYS_exit    2
3653 #define SYS_wait    3
3654 #define SYS_pipe    4
3655 #define SYS_read    5
3656 #define SYS_kill    6
3657 #define SYS_exec    7
3658 #define SYS_fstat   8
3659 #define SYS_chdir   9
3660 #define SYS_dup    10
3661 #define SYS_getpid 11
3662 #define SYS_sbrk   12
3663 #define SYS_sleep  13
3664 #define SYS_uptime 14
3665 #define SYS_open   15
3666 #define SYS_write  16
3667 #define SYS_mknod  17
3668 #define SYS_unlink 18
3669 #define SYS_link   19
3670 #define SYS_mkdir  20
3671 #define SYS_close  21
3672 #define SYS_halt   22
3673 #define SYS_date   23
3674 #define SYS_getuid 24
3675 #define SYS_getgid 25
3676 #define SYS_getppid 26
3677 #define SYS_setuid 27
3678 #define SYS_setgid 28
3679 #define SYS_getprocs 29
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 "defs.h"
3702 #include "param.h"
3703 #include "memlayout.h"
3704 #include "mmu.h"
3705 #include "proc.h"
3706 #include "x86.h"
3707 #include "syscall.h"
3708
3709 // User code makes a system call with INT T_SYSCALL.
3710 // System call number in %eax.
3711 // Arguments on the stack, from the user call to the C
3712 // library system call function. The saved user %esp points
3713 // to a saved program counter, and then the first argument.
3714
3715 // Fetch the int at addr from the current process.
3716 int
3717 fetchint(uint addr, int *ip)
3718 {
3719   if(addr >= proc->sz || addr+4 > proc->sz)
3720     return -1;
3721   *ip = *(int*)(addr);
3722   return 0;
3723 }
3724
3725 // Fetch the nul-terminated string at addr from the current process.
3726 // Doesn't actually copy the string - just sets *pp to point at it.
3727 // Returns length of string, not including nul.
3728 int
3729 fetchstr(uint addr, char **pp)
3730 {
3731   char *s, *ep;
3732
3733   if(addr >= proc->sz)
3734     return -1;
3735   *pp = (char*)addr;
3736   ep = (char*)proc->sz;
3737   for(s = *pp; s < ep; s++)
3738     if(*s == 0)
3739       return s - *pp;
3740   return -1;
3741 }
3742
3743 // Fetch the nth 32-bit system call argument.
3744 int
3745 argint(int n, int *ip)
3746 {
3747   return fetchint(proc->tf->esp + 4 + 4*n, ip);
3748 }
3749
```



```
3750 // Fetch the nth word-sized system call argument as a pointer
3751 // to a block of memory of size n bytes.  Check that the pointer
3752 // lies within the process address space.
3753 int
3754 argptr(int n, char **pp, int size)
3755 {
3756   int i;
3757
3758   if(argint(n, &i) < 0)
3759     return -1;
3760   if((uint)i >= proc->sz || (uint)i+size > proc->sz)
3761     return -1;
3762   *pp = (char*)i;
3763   return 0;
3764 }
3765
3766 // Fetch the nth word-sized system call argument as a string pointer.
3767 // Check that the pointer is valid and the string is nul-terminated.
3768 // (There is no shared writable memory, so the string can't change
3769 // between this check and being used by the kernel.)
3770 int
3771 argstr(int n, char **pp)
3772 {
3773   int addr;
3774   if(argint(n, &addr) < 0)
3775     return -1;
3776   return fetchstr(addr, pp);
3777 }
3778
3779 extern int sys_chdir(void);
3780 extern int sys_close(void);
3781 extern int sys_dup(void);
3782 extern int sys_exec(void);
3783 extern int sys_exit(void);
3784 extern int sys_fork(void);
3785 extern int sys_fstat(void);
3786 extern int sys_getpid(void);
3787 extern int sys_kill(void);
3788 extern int sys_link(void);
3789 extern int sys_mkdir(void);
3790 extern int sys_mknod(void);
3791 extern int sys_open(void);
3792 extern int sys_pipe(void);
3793 extern int sys_read(void);
3794 extern int sys_sbrk(void);
3795 extern int sys_sleep(void);
3796 extern int sys_unlink(void);
3797 extern int sys_wait(void);
3798 extern int sys_write(void);
3799 extern int sys_uptime(void);
```

```
3800 extern int sys_halt(void);
3801 extern int sys_date(void);
3802 extern int sys_getuid(void);
3803 extern int sys_getgid(void);
3804 extern int sys_getppid(void);
3805 extern int sys_setuid(void);
3806 extern int sys_setgid(void);
3807 extern int sys_getprocs(void);
3808
3809 static int (*syscalls[])(void) = {
3810 [SYS_fork]    sys_fork,
3811 [SYS_exit]    sys_exit,
3812 [SYS_wait]    sys_wait,
3813 [SYS_pipe]    sys_pipe,
3814 [SYS_read]    sys_read,
3815 [SYS_kill]    sys_kill,
3816 [SYS_exec]    sys_exec,
3817 [SYS_fstat]   sys_fstat,
3818 [SYS_chdir]   sys_chdir,
3819 [SYS_dup]     sys_dup,
3820 [SYS_getpid]  sys_getpid,
3821 [SYS_sbrk]    sys_sbrk,
3822 [SYS_sleep]   sys_sleep,
3823 [SYS_uptime]  sys_uptime,
3824 [SYS_open]    sys_open,
3825 [SYS_write]   sys_write,
3826 [SYS_mknod]   sys_mknod,
3827 [SYS_unlink]  sys_unlink,
3828 [SYS_link]    sys_link,
3829 [SYS_mkdir]   sys_mkdir,
3830 [SYS_close]   sys_close,
3831 [SYS_halt]    sys_halt,
3832 [SYS_date]    sys_date,
3833 [SYS_getuid]  sys_getuid,
3834 [SYS_getgid]  sys_getgid,
3835 [SYS_getppid] sys_getppid,
3836 [SYS_setuid]  sys_setuid,
3837 [SYS_setgid]  sys_setgid,
3838 [SYS_getprocs] sys_getprocs,
3839 };
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
```



```
3850 const char *syscallname[] = {
3851
3852 [SYS_fork]    "fork",
3853 [SYS_exit]    "exit",
3854 [SYS_wait]    "wait",
3855 [SYS_pipe]    "pipe",
3856 [SYS_read]    "read",
3857 [SYS_kill]    "kill",
3858 [SYS_exec]    "exec",
3859 [SYS_fstat]   "fstat",
3860 [SYS_chdir]   "chdir",
3861 [SYS_dup]     "dup",
3862 [SYS_getpid]  "getpid",
3863 [SYS_sbrk]    "sbrk",
3864 [SYS_sleep]   "sleep",
3865 [SYS_uptime]  "uptime",
3866 [SYS_open]    "open",
3867 [SYS_write]   "write",
3868 [SYS_mknod]   "mknod",
3869 [SYS_unlink]  "unlink",
3870 [SYS_link]    "link",
3871 [SYS_mkdir]   "mkdir",
3872 [SYS_close]   "close",
3873 [SYS_halt]    "halt",
3874 [SYS_date]    "date",
3875 [SYS_getuid]  "getuid",
3876 [SYS_getgid]  "getgid",
3877 [SYS_getppid] "getppid",
3878 [SYS_setuid]  "setuid",
3879 [SYS_setgid]  "setgid",
3880 [SYS_getprocs] "getprocs",
3881
3882
3883 };
3884
3885
3886 void
3887 syscall(void)
3888 {
3889   int num;
3890
3891   num = proc->tf->eax;
3892   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3893     proc->tf->eax = syscalls[num]();
3894
3895 //    cprintf("%s %d %d %d %d %d %d:\n", syscallname[num], proc->tf->eax, pr
3896
3897
3898
3899
```

```
3900   } else {
3901     cprintf("%d %s: unknown sys call %d\n",
3902             proc->pid, proc->name, num);
3903     proc->tf->eax = -1;
3904   }
3905 }
3906
3907
3908
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
```



```
3950 #include "types.h"
3951 #include "x86.h"
3952 #include "defs.h"
3953 #include "date.h"
3954 #include "param.h"
3955 #include "memlayout.h"
3956 #include "mmu.h"
3957 #include "proc.h"
3958 #include "ps.h"
3959
3960 int
3961 sys_fork(void)
3962 {
3963   return fork();
3964 }
3965
3966 int
3967 sys_exit(void)
3968 {
3969   exit();
3970   return 0;  // not reached
3971 }
3972
3973 int
3974 sys_wait(void)
3975 {
3976   return wait();
3977 }
3978
3979 int
3980 sys_kill(void)
3981 {
3982   int pid;
3983
3984   if(argint(0, &pid) < 0)
3985     return -1;
3986   return kill(pid);
3987 }
3988
3989 int
3990 sys_getpid(void)
3991 {
3992   return proc->pid;
3993 }
3994
3995 int
3996 sys_getuid(void)
3997 {
3998   return getuid();
3999 }
```

```
4000 int
4001 sys_getgid(void)
4002 {
4003   return getgid();
4004 }
4005
4006 int
4007 sys_getppid(void)
4008 {
4009   return proc->ppid;
4010 }
4011
4012 int
4013 sys_setuid(void)
4014 {
4015     int uid;
4016     if(argint(0, &uid) < 0)
4017    return -1;
4018
4019     return setuid(uid);
4020 }
4021
4022 int
4023 sys_setgid(void)
4024 {
4025     int gid;
4026     if(argint(0, &gid) < 0)
4027    return -1;
4028
4029     return setgid(gid);
4030 }
4031
4032 int
4033 sys_sbrk(void)
4034 {
4035   int addr;
4036   int n;
4037
4038   if(argint(0, &n) < 0)
4039     return -1;
4040   addr = proc->sz;
4041   if(growproc(n) < 0)
4042     return -1;
4043   return addr;
4044 }
4045
4046
4047
4048
4049
```



```
4050 int
4051 sys_sleep(void)
4052 {
4053   int n;
4054   uint ticks0;
4055
4056   if(argint(0, &n) < 0)
4057     return -1;
4058   acquire(&tickslock);
4059   ticks0 = ticks;
4060   while(ticks - ticks0 < n){
4061     if(proc->killed){
4062       release(&tickslock);
4063       return -1;
4064     }
4065     sleep(&ticks, &tickslock);
4066   }
4067   release(&tickslock);
4068   return 0;
4069 }
4070
4071 // return how many clock tick interrupts have occurred
4072 // since start.
4073 int
4074 sys_uptime(void)
4075 {
4076   uint xticks;
4077
4078   acquire(&tickslock);
4079   xticks = ticks;
4080   release(&tickslock);
4081   return xticks;
4082 }
4083
4084 //Turn of the computer
4085 int sys_halt(void){
4086   cprintf("Shutting down ...\n");
4087   outw (0xB004, 0x0 | 0x2000);
4088   return 0;
4089 }
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
```

```
4100 //Date
4101 int
4102 sys_date(void)
4103 {
4104
4105     struct rtcdate *d;
4106     if(argptr(0, (void*)&d, sizeof(*d)) < 0)
4107    return -1;
4108     cmostime(d);
4109
4110     return 0;
4111
4112 }
4113
4114 //getprocs
4115 int
4116 sys_getprocs(void)
4117 {
4118     int max;
4119     struct uproc *table;
4120
4121     if(argint(0, &max) < 0)
4122    return -1;
4123
4124
4125     if(argptr(1, (void*)&table, sizeof(*table)) < 0)
4126    return -1;
4127
4128     return getprocs(max, table);
4129
4130
4131 }
4132
4133
4134
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```



```
4150 struct buf {
4151   int flags;
4152   uint dev;
4153   uint blockno;
4154   struct buf *prev; // LRU cache list
4155   struct buf *next;
4156   struct buf *qnext; // disk queue
4157   uchar data[BSIZE];
4158 };
4159 #define B_BUSY  0x1  // buffer is locked by some process
4160 #define B_VALID 0x2  // buffer has been read from disk
4161 #define B_DIRTY 0x4  // buffer needs to be written to disk
4162
4163
4164
4165
4166
4167
4168
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
```

```
4200 #define O_RDONLY  0x000
4201 #define O_WRONLY  0x001
4202 #define O_RDWR    0x002
4203 #define O_CREATE  0x200
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 #define T_DIR  1   // Directory
4251 #define T_FILE 2   // File
4252 #define T_DEV  3   // Device
4253
4254 struct stat {
4255   short type;  // Type of file
4256   int dev;     // File system's disk device
4257   uint ino;    // Inode number
4258   short nlink; // Number of links to file
4259   uint size;   // Size of file in bytes
4260 };
4261
4262
4263
4264
4265
4266
4267
4268
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
```

```
4300 // On-disk file system format.
4301 // Both the kernel and user programs use this header file.
4302
4303
4304 #define ROOTINO 1  // root i-number
4305 #define BSIZE 512  // block size
4306
4307 // Disk layout:
4308 // [ boot block | super block | log | inode blocks | free bit map | data blo
4309 //
4310 // mkfs computes the super block and builds an initial file system. The supe
4311 // the disk layout:
4312 struct superblock {
4313   uint size;         // Size of file system image (blocks)
4314   uint nblocks;      // Number of data blocks
4315   uint ninodes;      // Number of inodes.
4316   uint nlog;         // Number of log blocks
4317   uint logstart;     // Block number of first log block
4318   uint inodestart;   // Block number of first inode block
4319   uint bmapstart;    // Block number of first free map block
4320 };
4321
4322 #define NDIRECT 12
4323 #define NINDIRECT (BSIZE / sizeof(uint))
4324 #define MAXFILE (NDIRECT + NINDIRECT)
4325
4326 // On-disk inode structure
4327 struct dinode {
4328   short type;           // File type
4329   short major;          // Major device number (T_DEV only)
4330   short minor;          // Minor device number (T_DEV only)
4331   short nlink;          // Number of links to inode in file system
4332   uint size;            // Size of file (bytes)
4333   uint addrs[NDIRECT+1];   // Data block addresses
4334 };
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
```



```
4350 // Inodes per block.
4351 #define IPB           (BSIZE / sizeof(struct dinode))
4352
4353 // Block containing inode i
4354 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
4355
4356 // Bitmap bits per block
4357 #define BPB           (BSIZE*8)
4358
4359 // Block of free map containing bit for block b
4360 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
4361
4362 // Directory is a file containing a sequence of dirent structures.
4363 #define DIRSIZ 14
4364
4365 struct dirent {
4366   ushort inum;
4367   char name[DIRSIZ];
4368 };
4369
4370
4371
4372
4373
4374
4375
4376
4377
4378
4379
4380
4381
4382
4383
4384
4385
4386
4387
4388
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
```

```
4400 struct file {
4401   enum { FD_NONE, FD_PIPE, FD_INODE } type;
4402   int ref; // reference count
4403   char readable;
4404   char writable;
4405   struct pipe *pipe;
4406   struct inode *ip;
4407   uint off;
4408 };
4409
4410
4411 // in-memory copy of an inode
4412 struct inode {
4413   uint dev;           // Device number
4414   uint inum;          // Inode number
4415   int ref;            // Reference count
4416   int flags;          // I_BUSY, I_VALID
4417
4418   short type;         // copy of disk inode
4419   short major;
4420   short minor;
4421   short nlink;
4422   uint size;
4423   uint addrs[NDIRECT+1];
4424 };
4425 #define I_BUSY 0x1
4426 #define I_VALID 0x2
4427
4428 // table mapping major device number to
4429 // device functions
4430 struct devsw {
4431   int (*read)(struct inode*, char*, int);
4432   int (*write)(struct inode*, char*, int);
4433 };
4434
4435 extern struct devsw devsw[];
4436
4437 #define CONSOLE 1
4438
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
```



```
4450 // Blank page.
4451
4452
4453
4454
4455
4456
4457
4458
4459
4460
4461
4462
4463
4464
4465
4466
4467
4468
4469
4470
4471
4472
4473
4474
4475
4476
4477
4478
4479
4480
4481
4482
4483
4484
4485
4486
4487
4488
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499
```

```
4500 // Simple PIO-based (non-DMA) IDE driver code.
4501
4502 #include "types.h"
4503 #include "defs.h"
4504 #include "param.h"
4505 #include "memlayout.h"
4506 #include "mmu.h"
4507 #include "proc.h"
4508 #include "x86.h"
4509 #include "traps.h"
4510 #include "spinlock.h"
4511 #include "fs.h"
4512 #include "buf.h"
4513
4514 #define SECTOR_SIZE   512
4515 #define IDE_BSY       0x80
4516 #define IDE_DRDY      0x40
4517 #define IDE_DF        0x20
4518 #define IDE_ERR       0x01
4519
4520 #define IDE_CMD_READ  0x20
4521 #define IDE_CMD_WRITE 0x30
4522
4523 // idequeue points to the buf now being read/written to the disk.
4524 // idequeue->qnext points to the next buf to be processed.
4525 // You must hold idelock while manipulating queue.
4526
4527 static struct spinlock idelock;
4528 static struct buf *idequeue;
4529
4530 static int havedisk1;
4531 static void idestart(struct buf*);
4532
4533 // Wait for IDE disk to become ready.
4534 static int
4535 idewait(int checkerr)
4536 {
4537   int r;
4538
4539   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4540     ;
4541   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4542     return -1;
4543   return 0;
4544 }
4545
4546
4547
4548
4549
```



```
4550 void
4551 ideinit(void)
4552 {
4553   int i;
4554
4555   initlock(&idelock, "ide");
4556   picenable(IRQ_IDE);
4557   ioapicenable(IRQ_IDE, ncpu - 1);
4558   idewait(0);
4559
4560   // Check if disk 1 is present
4561   outb(0x1f6, 0xe0 | (1<<4));
4562   for(i=0; i<1000; i++){
4563     if(inb(0x1f7) != 0){
4564       havedisk1 = 1;
4565       break;
4566     }
4567   }
4568
4569   // Switch back to disk 0.
4570   outb(0x1f6, 0xe0 | (0<<4));
4571 }
4572
4573 // Start the request for b.  Caller must hold idelock.
4574 static void
4575 idestart(struct buf *b)
4576 {
4577   if(b == 0)
4578     panic("idestart");
4579   if(b->blockno >= FSSIZE)
4580     panic("incorrect blockno");
4581   int sector_per_block =  BSIZE/SECTOR_SIZE;
4582   int sector = b->blockno * sector_per_block;
4583
4584   if (sector_per_block > 7) panic("idestart");
4585
4586   idewait(0);
4587   outb(0x3f6, 0);  // generate interrupt
4588   outb(0x1f2, sector_per_block);  // number of sectors
4589   outb(0x1f3, sector & 0xff);
4590   outb(0x1f4, (sector >> 8) & 0xff);
4591   outb(0x1f5, (sector >> 16) & 0xff);
4592   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4593   if(b->flags & B_DIRTY){
4594     outb(0x1f7, IDE_CMD_WRITE);
4595     outsl(0x1f0, b->data, BSIZE/4);
4596   } else {
4597     outb(0x1f7, IDE_CMD_READ);
4598   }
4599 }
```

```
4600 // Interrupt handler.
4601 void
4602 ideintr(void)
4603 {
4604   struct buf *b;
4605
4606   // First queued buffer is the active request.
4607   acquire(&idelock);
4608   if((b = idequeue) == 0){
4609     release(&idelock);
4610     // cprintf("spurious IDE interrupt\n");
4611     return;
4612   }
4613   idequeue = b->qnext;
4614
4615   // Read data if needed.
4616   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4617     insl(0x1f0, b->data, BSIZE/4);
4618
4619   // Wake process waiting for this buf.
4620   b->flags |= B_VALID;
4621   b->flags &= ~B_DIRTY;
4622   wakeup(b);
4623
4624   // Start disk on next buf in queue.
4625   if(idequeue != 0)
4626     idestart(idequeue);
4627
4628   release(&idelock);
4629 }
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649
```



```
4650 // Sync buf with disk.
4651 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
4652 // Else if B_VALID is not set, read buf from disk, set B_VALID.
4653 void
4654 iderw(struct buf *b)
4655 {
4656   struct buf **pp;
4657
4658   if(!(b->flags & B_BUSY))
4659     panic("iderw: buf not busy");
4660   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
4661     panic("iderw: nothing to do");
4662   if(b->dev != 0 && !havedisk1)
4663     panic("iderw: ide disk 1 not present");
4664
4665   acquire(&idelock);
4666
4667   // Append b to idequeue.
4668   b->qnext = 0;
4669   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
4670     ;
4671   *pp = b;
4672
4673   // Start disk if necessary.
4674   if(idequeue == b)
4675     idestart(b);
4676
4677   // Wait for request to finish.
4678   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
4679     sleep(b, &idelock);
4680   }
4681
4682   release(&idelock);
4683 }
4684
4685
4686
4687
4688
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4699
```

```
4700 // Buffer cache.
4701 //
4702 // The buffer cache is a linked list of buf structures holding
4703 // cached copies of disk block contents.  Caching disk blocks
4704 // in memory reduces the number of disk reads and also provides
4705 // a synchronization point for disk blocks used by multiple processes.
4706 //
4707 // Interface:
4708 // * To get a buffer for a particular disk block, call bread.
4709 // * After changing buffer data, call bwrite to write it to disk.
4710 // * When done with the buffer, call brelse.
4711 // * Do not use the buffer after calling brelse.
4712 // * Only one process at a time can use a buffer,
4713 //     so do not keep them longer than necessary.
4714 //
4715 // The implementation uses three state flags internally:
4716 // * B_BUSY: the block has been returned from bread
4717 //     and has not been passed back to brelse.
4718 // * B_VALID: the buffer data has been read from the disk.
4719 // * B_DIRTY: the buffer data has been modified
4720 //     and needs to be written to disk.
4721
4722 #include "types.h"
4723 #include "defs.h"
4724 #include "param.h"
4725 #include "spinlock.h"
4726 #include "fs.h"
4727 #include "buf.h"
4728
4729 struct {
4730   struct spinlock lock;
4731   struct buf buf[NBUF];
4732
4733   // Linked list of all buffers, through prev/next.
4734   // head.next is most recently used.
4735   struct buf head;
4736 } bcache;
4737
4738 void
4739 binit(void)
4740 {
4741   struct buf *b;
4742
4743   initlock(&bcache.lock, "bcache");
4744
4745
4746
4747
4748
4749
```



```
4750   // Create linked list of buffers
4751   bcache.head.prev = &bcache.head;
4752   bcache.head.next = &bcache.head;
4753   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4754     b->next = bcache.head.next;
4755     b->prev = &bcache.head;
4756     b->dev = -1;
4757     bcache.head.next->prev = b;
4758     bcache.head.next = b;
4759   }
4760 }
4761
4762 // Look through buffer cache for block on device dev.
4763 // If not found, allocate a buffer.
4764 // In either case, return B_BUSY buffer.
4765 static struct buf*
4766 bget(uint dev, uint blockno)
4767 {
4768   struct buf *b;
4769
4770   acquire(&bcache.lock);
4771
4772  loop:
4773   // Is the block already cached?
4774   for(b = bcache.head.next; b != &bcache.head; b = b->next){
4775     if(b->dev == dev && b->blockno == blockno){
4776       if(!(b->flags & B_BUSY)){
4777         b->flags |= B_BUSY;
4778         release(&bcache.lock);
4779         return b;
4780       }
4781       sleep(b, &bcache.lock);
4782       goto loop;
4783     }
4784   }
4785
4786   // Not cached; recycle some non-busy and clean buffer.
4787   // "clean" because B_DIRTY and !B_BUSY means log.c
4788   // hasn't yet committed the changes to the buffer.
4789   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4790     if((b->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
4791       b->dev = dev;
4792       b->blockno = blockno;
4793       b->flags = B_BUSY;
4794       release(&bcache.lock);
4795       return b;
4796     }
4797   }
4798   panic("bget: no buffers");
4799 }
```

```
4800 // Return a B_BUSY buf with the contents of the indicated block.
4801 struct buf*
4802 bread(uint dev, uint blockno)
4803 {
4804   struct buf *b;
4805
4806   b = bget(dev, blockno);
4807   if(!(b->flags & B_VALID)) {
4808     iderw(b);
4809   }
4810   return b;
4811 }
4812
4813 // Write b's contents to disk.  Must be B_BUSY.
4814 void
4815 bwrite(struct buf *b)
4816 {
4817   if((b->flags & B_BUSY) == 0)
4818     panic("bwrite");
4819   b->flags |= B_DIRTY;
4820   iderw(b);
4821 }
4822
4823 // Release a B_BUSY buffer.
4824 // Move to the head of the MRU list.
4825 void
4826 brelse(struct buf *b)
4827 {
4828   if((b->flags & B_BUSY) == 0)
4829     panic("brelse");
4830
4831   acquire(&bcache.lock);
4832
4833   b->next->prev = b->prev;
4834   b->prev->next = b->next;
4835   b->next = bcache.head.next;
4836   b->prev = &bcache.head;
4837   bcache.head.next->prev = b;
4838   bcache.head.next = b;
4839
4840   b->flags &= ~B_BUSY;
4841   wakeup(b);
4842
4843   release(&bcache.lock);
4844 }
4845
4846
4847
4848
4849
```



```
4850 // Blank page.
4851
4852
4853
4854
4855
4856
4857
4858
4859
4860
4861
4862
4863
4864
4865
4866
4867
4868
4869
4870
4871
4872
4873
4874
4875
4876
4877
4878
4879
4880
4881
4882
4883
4884
4885
4886
4887
4888
4889
4890
4891
4892
4893
4894
4895
4896
4897
4898
4899
```

```
4900 #include "types.h"
4901 #include "defs.h"
4902 #include "param.h"
4903 #include "spinlock.h"
4904 #include "fs.h"
4905 #include "buf.h"
4906
4907 // Simple logging that allows concurrent FS system calls.
4908 //
4909 // A log transaction contains the updates of multiple FS system
4910 // calls. The logging system only commits when there are
4911 // no FS system calls active. Thus there is never
4912 // any reasoning required about whether a commit might
4913 // write an uncommitted system call's updates to disk.
4914 //
4915 // A system call should call begin_op()/end_op() to mark
4916 // its start and end. Usually begin_op() just increments
4917 // the count of in-progress FS system calls and returns.
4918 // But if it thinks the log is close to running out, it
4919 // sleeps until the last outstanding end_op() commits.
4920 //
4921 // The log is a physical re-do log containing disk blocks.
4922 // The on-disk log format:
4923 //   header block, containing block #s for block A, B, C, ...
4924 //   block A
4925 //   block B
4926 //   block C
4927 //   ...
4928 // Log appends are synchronous.
4929
4930 // Contents of the header block, used for both the on-disk header block
4931 // and to keep track in memory of logged block# before commit.
4932 struct logheader {
4933   int n;
4934   int block[LOGSIZE];
4935 };
4936
4937 struct log {
4938   struct spinlock lock;
4939   int start;
4940   int size;
4941   int outstanding; // how many FS sys calls are executing.
4942   int committing;  // in commit(), please wait.
4943   int dev;
4944   struct logheader lh;
4945 };
4946
4947
4948
4949
```



```
4950 struct log log;
4951
4952 static void recover_from_log(void);
4953 static void commit();
4954
4955 void
4956 initlog(int dev)
4957 {
4958   if (sizeof(struct logheader) >= BSIZE)
4959     panic("initlog: too big logheader");
4960
4961   struct superblock sb;
4962   initlock(&log.lock, "log");
4963   readsb(dev, &sb);
4964   log.start = sb.logstart;
4965   log.size = sb.nlog;
4966   log.dev = dev;
4967   recover_from_log();
4968 }
4969
4970 // Copy committed blocks from log to their home location
4971 static void
4972 install_trans(void)
4973 {
4974   int tail;
4975
4976   for (tail = 0; tail < log.lh.n; tail++) {
4977     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
4978     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
4979     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
4980     bwrite(dbuf);  // write dst to disk
4981     brelse(lbuf);
4982     brelse(dbuf);
4983   }
4984 }
4985
4986 // Read the log header from disk into the in-memory log header
4987 static void
4988 read_head(void)
4989 {
4990   struct buf *buf = bread(log.dev, log.start);
4991   struct logheader *lh = (struct logheader *) (buf->data);
4992   int i;
4993   log.lh.n = lh->n;
4994   for (i = 0; i < log.lh.n; i++) {
4995     log.lh.block[i] = lh->block[i];
4996   }
4997   brelse(buf);
4998 }
4999
```

```
5000 // Write in-memory log header to disk.
5001 // This is the true point at which the
5002 // current transaction commits.
5003 static void
5004 write_head(void)
5005 {
5006   struct buf *buf = bread(log.dev, log.start);
5007   struct logheader *hb = (struct logheader *) (buf->data);
5008   int i;
5009   hb->n = log.lh.n;
5010   for (i = 0; i < log.lh.n; i++) {
5011     hb->block[i] = log.lh.block[i];
5012   }
5013   bwrite(buf);
5014   brelse(buf);
5015 }
5016
5017 static void
5018 recover_from_log(void)
5019 {
5020   read_head();
5021   install_trans(); // if committed, copy from log to disk
5022   log.lh.n = 0;
5023   write_head(); // clear the log
5024 }
5025
5026 // called at the start of each FS system call.
5027 void
5028 begin_op(void)
5029 {
5030   acquire(&log.lock);
5031   while(1){
5032     if(log.committing){
5033       sleep(&log, &log.lock);
5034     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
5035       // this op might exhaust log space; wait for commit.
5036       sleep(&log, &log.lock);
5037     } else {
5038       log.outstanding += 1;
5039       release(&log.lock);
5040       break;
5041     }
5042   }
5043 }
5044
5045
5046
5047
5048
5049
```



```
5050 // called at the end of each FS system call.
5051 // commits if this was the last outstanding operation.
5052 void
5053 end_op(void)
5054 {
5055   int do_commit = 0;
5056
5057   acquire(&log.lock);
5058   log.outstanding -= 1;
5059   if(log.committing)
5060     panic("log.committing");
5061   if(log.outstanding == 0){
5062     do_commit = 1;
5063     log.committing = 1;
5064   } else {
5065     // begin_op() may be waiting for log space.
5066     wakeup(&log);
5067   }
5068   release(&log.lock);
5069
5070   if(do_commit){
5071     // call commit w/o holding locks, since not allowed
5072     // to sleep with locks.
5073     commit();
5074     acquire(&log.lock);
5075     log.committing = 0;
5076     wakeup(&log);
5077     release(&log.lock);
5078   }
5079 }
5080
5081 // Copy modified blocks from cache to log.
5082 static void
5083 write_log(void)
5084 {
5085   int tail;
5086
5087   for (tail = 0; tail < log.lh.n; tail++) {
5088     struct buf *to = bread(log.dev, log.start+tail+1); // log block
5089     struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
5090     memmove(to->data, from->data, BSIZE);
5091     bwrite(to);  // write the log
5092     brelse(from);
5093     brelse(to);
5094   }
5095 }
5096
5097
5098
5099
```

```
5100 static void
5101 commit()
5102 {
5103   if (log.lh.n > 0) {
5104     write_log();     // Write modified blocks from cache to log
5105     write_head();    // Write header to disk -- the real commit
5106     install_trans(); // Now install writes to home locations
5107     log.lh.n = 0;
5108     write_head();    // Erase the transaction from the log
5109   }
5110 }
5111
5112 // Caller has modified b->data and is done with the buffer.
5113 // Record the block number and pin in the cache with B_DIRTY.
5114 // commit()/write_log() will do the disk write.
5115 //
5116 // log_write() replaces bwrite(); a typical use is:
5117 //   bp = bread(...)
5118 //   modify bp->data[]
5119 //   log_write(bp)
5120 //   brelse(bp)
5121 void
5122 log_write(struct buf *b)
5123 {
5124   int i;
5125
5126   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
5127     panic("too big a transaction");
5128   if (log.outstanding < 1)
5129     panic("log_write outside of trans");
5130
5131   acquire(&log.lock);
5132   for (i = 0; i < log.lh.n; i++) {
5133     if (log.lh.block[i] == b->blockno)   // log absorbtion
5134       break;
5135   }
5136   log.lh.block[i] = b->blockno;
5137   if (i == log.lh.n)
5138     log.lh.n++;
5139   b->flags |= B_DIRTY; // prevent eviction
5140   release(&log.lock);
5141 }
5142
5143
5144
5145
5146
5147
5148
5149
```



```
5150 // File system implementation.  Five layers:
5151 //   + Blocks: allocator for raw disk blocks.
5152 //   + Log: crash recovery for multi-step updates.
5153 //   + Files: inode allocator, reading, writing, metadata.
5154 //   + Directories: inode with special contents (list of other inodes!)
5155 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
5156 //
5157 // This file contains the low-level file system manipulation
5158 // routines.  The (higher-level) system call implementations
5159 // are in sysfile.c.
5160
5161 #include "types.h"
5162 #include "defs.h"
5163 #include "param.h"
5164 #include "stat.h"
5165 #include "mmu.h"
5166 #include "proc.h"
5167 #include "spinlock.h"
5168 #include "fs.h"
5169 #include "buf.h"
5170 #include "file.h"
5171
5172 #define min(a, b) ((a) < (b) ? (a) : (b))
5173 static void itrunc(struct inode*);
5174 struct superblock sb;   // there should be one per dev, but we run with one d
5175
5176 // Read the super block.
5177 void
5178 readsb(int dev, struct superblock *sb)
5179 {
5180   struct buf *bp;
5181
5182   bp = bread(dev, 1);
5183   memmove(sb, bp->data, sizeof(*sb));
5184   brelse(bp);
5185 }
5186
5187 // Zero a block.
5188 static void
5189 bzero(int dev, int bno)
5190 {
5191   struct buf *bp;
5192
5193   bp = bread(dev, bno);
5194   memset(bp->data, 0, BSIZE);
5195   log_write(bp);
5196   brelse(bp);
5197 }
5198
5199
```

```
5200 // Blocks.
5201
5202 // Allocate a zeroed disk block.
5203 static uint
5204 balloc(uint dev)
5205 {
5206   int b, bi, m;
5207   struct buf *bp;
5208
5209   bp = 0;
5210   for(b = 0; b < sb.size; b += BPB){
5211     bp = bread(dev, BBLOCK(b, sb));
5212     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
5213       m = 1 << (bi % 8);
5214       if((bp->data[bi/8] & m) == 0){  // Is block free?
5215         bp->data[bi/8] |= m;  // Mark block in use.
5216         log_write(bp);
5217         brelse(bp);
5218         bzero(dev, b + bi);
5219         return b + bi;
5220       }
5221     }
5222     brelse(bp);
5223   }
5224   panic("balloc: out of blocks");
5225 }
5226
5227 // Free a disk block.
5228 static void
5229 bfree(int dev, uint b)
5230 {
5231   struct buf *bp;
5232   int bi, m;
5233
5234   readsb(dev, &sb);
5235   bp = bread(dev, BBLOCK(b, sb));
5236   bi = b % BPB;
5237   m = 1 << (bi % 8);
5238   if((bp->data[bi/8] & m) == 0)
5239     panic("freeing free block");
5240   bp->data[bi/8] &= ~m;
5241   log_write(bp);
5242   brelse(bp);
5243 }
5244
5245
5246
5247
5248
5249
```



```
5250 // Inodes.
5251 //
5252 // An inode describes a single unnamed file.
5253 // The inode disk structure holds metadata: the file's type,
5254 // its size, the number of links referring to it, and the
5255 // list of blocks holding the file's content.
5256 //
5257 // The inodes are laid out sequentially on disk at
5258 // sb.startinode. Each inode has a number, indicating its
5259 // position on the disk.
5260 //
5261 // The kernel keeps a cache of in-use inodes in memory
5262 // to provide a place for synchronizing access
5263 // to inodes used by multiple processes. The cached
5264 // inodes include book-keeping information that is
5265 // not stored on disk: ip->ref and ip->flags.
5266 //
5267 // An inode and its in-memory represtative go through a
5268 // sequence of states before they can be used by the
5269 // rest of the file system code.
5270 //
5271 // * Allocation: an inode is allocated if its type (on disk)
5272 //   is non-zero. ialloc() allocates, iput() frees if
5273 //   the link count has fallen to zero.
5274 //
5275 // * Referencing in cache: an entry in the inode cache
5276 //   is free if ip->ref is zero. Otherwise ip->ref tracks
5277 //   the number of in-memory pointers to the entry (open
5278 //   files and current directories). iget() to find or
5279 //   create a cache entry and increment its ref, iput()
5280 //   to decrement ref.
5281 //
5282 // * Valid: the information (type, size, &c) in an inode
5283 //   cache entry is only correct when the I_VALID bit
5284 //   is set in ip->flags. ilock() reads the inode from
5285 //   the disk and sets I_VALID, while iput() clears
5286 //   I_VALID if ip->ref has fallen to zero.
5287 //
5288 // * Locked: file system code may only examine and modify
5289 //   the information in an inode and its content if it
5290 //   has first locked the inode. The I_BUSY flag indicates
5291 //   that the inode is locked. ilock() sets I_BUSY,
5292 //   while iunlock clears it.
5293 //
5294 // Thus a typical sequence is:
5295 //   ip = iget(dev, inum)
5296 //   ilock(ip)
5297 //   ... examine and modify ip->xxx ...
5298 //   iunlock(ip)
5299 //   iput(ip)
```

```
5300 //
5301 // ilock() is separate from iget() so that system calls can
5302 // get a long-term reference to an inode (as for an open file)
5303 // and only lock it for short periods (e.g., in read()).
5304 // The separation also helps avoid deadlock and races during
5305 // pathname lookup. iget() increments ip->ref so that the inode
5306 // stays cached and pointers to it remain valid.
5307 //
5308 // Many internal file system functions expect the caller to
5309 // have locked the inodes involved; this lets callers create
5310 // multi-step atomic operations.
5311
5312 struct {
5313   struct spinlock lock;
5314   struct inode inode[NINODE];
5315 } icache;
5316
5317 void
5318 iinit(int dev)
5319 {
5320   initlock(&icache.lock, "icache");
5321   readsb(dev, &sb);
5322   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d inodestart %
5323           sb.nblocks, sb.ninodes, sb.nlog, sb.logstart, sb.inodestart, sb.bma
5324 }
5325
5326 static struct inode* iget(uint dev, uint inum);
5327
5328
5329
5330
5331
5332
5333
5334
5335
5336
5337
5338
5339
5340
5341
5342
5343
5344
5345
5346
5347
5348
5349
```



```
5350 // Allocate a new inode with the given type on device dev.
5351 // A free inode has a type of zero.
5352 struct inode*
5353 ialloc(uint dev, short type)
5354 {
5355   int inum;
5356   struct buf *bp;
5357   struct dinode *dip;
5358
5359   for(inum = 1; inum < sb.ninodes; inum++){
5360     bp = bread(dev, IBLOCK(inum, sb));
5361     dip = (struct dinode*)bp->data + inum%IPB;
5362     if(dip->type == 0){  // a free inode
5363       memset(dip, 0, sizeof(*dip));
5364       dip->type = type;
5365       log_write(bp);   // mark it allocated on the disk
5366       brelse(bp);
5367       return iget(dev, inum);
5368     }
5369     brelse(bp);
5370   }
5371   panic("ialloc: no inodes");
5372 }
5373
5374 // Copy a modified in-memory inode to disk.
5375 void
5376 iupdate(struct inode *ip)
5377 {
5378   struct buf *bp;
5379   struct dinode *dip;
5380
5381   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5382   dip = (struct dinode*)bp->data + ip->inum%IPB;
5383   dip->type = ip->type;
5384   dip->major = ip->major;
5385   dip->minor = ip->minor;
5386   dip->nlink = ip->nlink;
5387   dip->size = ip->size;
5388   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5389   log_write(bp);
5390   brelse(bp);
5391 }
5392
5393
5394
5395
5396
5397
5398
5399
```

```
5400 // Find the inode with number inum on device dev
5401 // and return the in-memory copy. Does not lock
5402 // the inode and does not read it from disk.
5403 static struct inode*
5404 iget(uint dev, uint inum)
5405 {
5406   struct inode *ip, *empty;
5407
5408   acquire(&icache.lock);
5409
5410   // Is the inode already cached?
5411   empty = 0;
5412   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5413     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5414       ip->ref++;
5415       release(&icache.lock);
5416       return ip;
5417     }
5418     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
5419       empty = ip;
5420   }
5421
5422   // Recycle an inode cache entry.
5423   if(empty == 0)
5424     panic("iget: no inodes");
5425
5426   ip = empty;
5427   ip->dev = dev;
5428   ip->inum = inum;
5429   ip->ref = 1;
5430   ip->flags = 0;
5431   release(&icache.lock);
5432
5433   return ip;
5434 }
5435
5436 // Increment reference count for ip.
5437 // Returns ip to enable ip = idup(ip1) idiom.
5438 struct inode*
5439 idup(struct inode *ip)
5440 {
5441   acquire(&icache.lock);
5442   ip->ref++;
5443   release(&icache.lock);
5444   return ip;
5445 }
5446
5447
5448
5449
```



```
5450 // Lock the given inode.
5451 // Reads the inode from disk if necessary.
5452 void
5453 ilock(struct inode *ip)
5454 {
5455   struct buf *bp;
5456   struct dinode *dip;
5457
5458   if(ip == 0 || ip->ref < 1)
5459     panic("ilock");
5460
5461   acquire(&icache.lock);
5462   while(ip->flags & I_BUSY)
5463     sleep(ip, &icache.lock);
5464   ip->flags |= I_BUSY;
5465   release(&icache.lock);
5466
5467   if(!(ip->flags & I_VALID)){
5468     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5469     dip = (struct dinode*)bp->data + ip->inum%IPB;
5470     ip->type = dip->type;
5471     ip->major = dip->major;
5472     ip->minor = dip->minor;
5473     ip->nlink = dip->nlink;
5474     ip->size = dip->size;
5475     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5476     brelse(bp);
5477     ip->flags |= I_VALID;
5478     if(ip->type == 0)
5479       panic("ilock: no type");
5480   }
5481 }
5482
5483 // Unlock the given inode.
5484 void
5485 iunlock(struct inode *ip)
5486 {
5487   if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
5488     panic("iunlock");
5489
5490   acquire(&icache.lock);
5491   ip->flags &= ~I_BUSY;
5492   wakeup(ip);
5493   release(&icache.lock);
5494 }
5495
5496
5497
5498
5499
```

```
5500 // Drop a reference to an in-memory inode.
5501 // If that was the last reference, the inode cache entry can
5502 // be recycled.
5503 // If that was the last reference and the inode has no links
5504 // to it, free the inode (and its content) on disk.
5505 // All calls to iput() must be inside a transaction in
5506 // case it has to free the inode.
5507 void
5508 iput(struct inode *ip)
5509 {
5510   acquire(&icache.lock);
5511   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
5512     // inode has no links and no other references: truncate and free.
5513     if(ip->flags & I_BUSY)
5514       panic("iput busy");
5515     ip->flags |= I_BUSY;
5516     release(&icache.lock);
5517     itrunc(ip);
5518     ip->type = 0;
5519     iupdate(ip);
5520     acquire(&icache.lock);
5521     ip->flags = 0;
5522     wakeup(ip);
5523   }
5524   ip->ref--;
5525   release(&icache.lock);
5526 }
5527
5528 // Common idiom: unlock, then put.
5529 void
5530 iunlockput(struct inode *ip)
5531 {
5532   iunlock(ip);
5533   iput(ip);
5534 }
5535
5536
5537
5538
5539
5540
5541
5542
5543
5544
5545
5546
5547
5548
5549
```



```
5550 // Inode content
5551 //
5552 // The content (data) associated with each inode is stored
5553 // in blocks on the disk. The first NDIRECT block numbers
5554 // are listed in ip->addrs[].  The next NINDIRECT blocks are
5555 // listed in block ip->addrs[NDIRECT].
5556
5557 // Return the disk block address of the nth block in inode ip.
5558 // If there is no such block, bmap allocates one.
5559 static uint
5560 bmap(struct inode *ip, uint bn)
5561 {
5562   uint addr, *a;
5563   struct buf *bp;
5564
5565   if(bn < NDIRECT){
5566     if((addr = ip->addrs[bn]) == 0)
5567       ip->addrs[bn] = addr = balloc(ip->dev);
5568     return addr;
5569   }
5570   bn -= NDIRECT;
5571
5572   if(bn < NINDIRECT){
5573     // Load indirect block, allocating if necessary.
5574     if((addr = ip->addrs[NDIRECT]) == 0)
5575       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5576     bp = bread(ip->dev, addr);
5577     a = (uint*)bp->data;
5578     if((addr = a[bn]) == 0){
5579       a[bn] = addr = balloc(ip->dev);
5580       log_write(bp);
5581     }
5582     brelse(bp);
5583     return addr;
5584   }
5585
5586   panic("bmap: out of range");
5587 }
5588
5589
5590
5591
5592
5593
5594
5595
5596
5597
5598
5599
```

```
5600 // Truncate inode (discard contents).
5601 // Only called when the inode has no links
5602 // to it (no directory entries referring to it)
5603 // and has no in-memory reference to it (is
5604 // not an open file or current directory).
5605 static void
5606 itrunc(struct inode *ip)
5607 {
5608   int i, j;
5609   struct buf *bp;
5610   uint *a;
5611
5612   for(i = 0; i < NDIRECT; i++){
5613     if(ip->addrs[i]){
5614       bfree(ip->dev, ip->addrs[i]);
5615       ip->addrs[i] = 0;
5616     }
5617   }
5618
5619   if(ip->addrs[NDIRECT]){
5620     bp = bread(ip->dev, ip->addrs[NDIRECT]);
5621     a = (uint*)bp->data;
5622     for(j = 0; j < NINDIRECT; j++){
5623       if(a[j])
5624         bfree(ip->dev, a[j]);
5625     }
5626     brelse(bp);
5627     bfree(ip->dev, ip->addrs[NDIRECT]);
5628     ip->addrs[NDIRECT] = 0;
5629   }
5630
5631   ip->size = 0;
5632   iupdate(ip);
5633 }
5634
5635 // Copy stat information from inode.
5636 void
5637 stati(struct inode *ip, struct stat *st)
5638 {
5639   st->dev = ip->dev;
5640   st->ino = ip->inum;
5641   st->type = ip->type;
5642   st->nlink = ip->nlink;
5643   st->size = ip->size;
5644 }
5645
5646
5647
5648
5649
```



```
5650 // Read data from inode.
5651 int
5652 readi(struct inode *ip, char *dst, uint off, uint n)
5653 {
5654   uint tot, m;
5655   struct buf *bp;
5656
5657   if(ip->type == T_DEV){
5658     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
5659       return -1;
5660     return devsw[ip->major].read(ip, dst, n);
5661   }
5662
5663   if(off > ip->size || off + n < off)
5664     return -1;
5665   if(off + n > ip->size)
5666     n = ip->size - off;
5667
5668   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5669     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5670     m = min(n - tot, BSIZE - off%BSIZE);
5671     memmove(dst, bp->data + off%BSIZE, m);
5672     brelse(bp);
5673   }
5674   return n;
5675 }
5676
5677
5678
5679
5680
5681
5682
5683
5684
5685
5686
5687
5688
5689
5690
5691
5692
5693
5694
5695
5696
5697
5698
5699
```

```
5700 // Write data to inode.
5701 int
5702 writei(struct inode *ip, char *src, uint off, uint n)
5703 {
5704   uint tot, m;
5705   struct buf *bp;
5706
5707   if(ip->type == T_DEV){
5708     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
5709       return -1;
5710     return devsw[ip->major].write(ip, src, n);
5711   }
5712
5713   if(off > ip->size || off + n < off)
5714     return -1;
5715   if(off + n > MAXFILE*BSIZE)
5716     return -1;
5717
5718   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
5719     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5720     m = min(n - tot, BSIZE - off%BSIZE);
5721     memmove(bp->data + off%BSIZE, src, m);
5722     log_write(bp);
5723     brelse(bp);
5724   }
5725
5726   if(n > 0 && off > ip->size){
5727     ip->size = off;
5728     iupdate(ip);
5729   }
5730   return n;
5731 }
5732
5733
5734
5735
5736
5737
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749
```



```
5750 // Directories
5751
5752 int
5753 namecmp(const char *s, const char *t)
5754 {
5755   return strncmp(s, t, DIRSIZ);
5756 }
5757
5758 // Look for a directory entry in a directory.
5759 // If found, set *poff to byte offset of entry.
5760 struct inode*
5761 dirlookup(struct inode *dp, char *name, uint *poff)
5762 {
5763   uint off, inum;
5764   struct dirent de;
5765
5766   if(dp->type != T_DIR)
5767     panic("dirlookup not DIR");
5768
5769   for(off = 0; off < dp->size; off += sizeof(de)){
5770     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5771       panic("dirlink read");
5772     if(de.inum == 0)
5773       continue;
5774     if(namecmp(name, de.name) == 0){
5775       // entry matches path element
5776       if(poff)
5777         *poff = off;
5778       inum = de.inum;
5779       return iget(dp->dev, inum);
5780     }
5781   }
5782
5783   return 0;
5784 }
5785
5786
5787
5788
5789
5790
5791
5792
5793
5794
5795
5796
5797
5798
5799
```

```
5800 // Write a new directory entry (name, inum) into the directory dp.
5801 int
5802 dirlink(struct inode *dp, char *name, uint inum)
5803 {
5804   int off;
5805   struct dirent de;
5806   struct inode *ip;
5807
5808   // Check that name is not present.
5809   if((ip = dirlookup(dp, name, 0)) != 0){
5810     iput(ip);
5811     return -1;
5812   }
5813
5814   // Look for an empty dirent.
5815   for(off = 0; off < dp->size; off += sizeof(de)){
5816     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5817       panic("dirlink read");
5818     if(de.inum == 0)
5819       break;
5820   }
5821
5822   strncpy(de.name, name, DIRSIZ);
5823   de.inum = inum;
5824   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5825     panic("dirlink");
5826
5827   return 0;
5828 }
5829
5830
5831
5832
5833
5834
5835
5836
5837
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```



```
5850 // Paths
5851
5852 // Copy the next path element from path into name.
5853 // Return a pointer to the element following the copied one.
5854 // The returned path has no leading slashes,
5855 // so the caller can check *path=='\0' to see if the name is the last one.
5856 // If no name to remove, return 0.
5857 //
5858 // Examples:
5859 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5860 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5861 //   skipelem("a", name) = "", setting name = "a"
5862 //   skipelem("", name) = skipelem("////", name) = 0
5863 //
5864 static char*
5865 skipelem(char *path, char *name)
5866 {
5867   char *s;
5868   int len;
5869
5870   while(*path == '/')
5871     path++;
5872   if(*path == 0)
5873     return 0;
5874   s = path;
5875   while(*path != '/' && *path != 0)
5876     path++;
5877   len = path - s;
5878   if(len >= DIRSIZ)
5879     memmove(name, s, DIRSIZ);
5880   else {
5881     memmove(name, s, len);
5882     name[len] = 0;
5883   }
5884   while(*path == '/')
5885     path++;
5886   return path;
5887 }
5888
5889
5890
5891
5892
5893
5894
5895
5896
5897
5898
5899
```

```
5900 // Look up and return the inode for a path name.
5901 // If parent != 0, return the inode for the parent and copy the final
5902 // path element into name, which must have room for DIRSIZ bytes.
5903 // Must be called inside a transaction since it calls iput().
5904 static struct inode*
5905 namex(char *path, int nameiparent, char *name)
5906 {
5907   struct inode *ip, *next;
5908
5909   if(*path == '/')
5910     ip = iget(ROOTDEV, ROOTINO);
5911   else
5912     ip = idup(proc->cwd);
5913
5914   while((path = skipelem(path, name)) != 0){
5915     ilock(ip);
5916     if(ip->type != T_DIR){
5917       iunlockput(ip);
5918       return 0;
5919     }
5920     if(nameiparent && *path == '\0'){
5921       // Stop one level early.
5922       iunlock(ip);
5923       return ip;
5924     }
5925     if((next = dirlookup(ip, name, 0)) == 0){
5926       iunlockput(ip);
5927       return 0;
5928     }
5929     iunlockput(ip);
5930     ip = next;
5931   }
5932   if(nameiparent){
5933     iput(ip);
5934     return 0;
5935   }
5936   return ip;
5937 }
5938
5939 struct inode*
5940 namei(char *path)
5941 {
5942   char name[DIRSIZ];
5943   return namex(path, 0, name);
5944 }
5945
5946
5947
5948
5949
```



```
5950 struct inode*
5951 nameiparent(char *path, char *name)
5952 {
5953   return namex(path, 1, name);
5954 }
5955
5956
5957
5958
5959
5960
5961
5962
5963
5964
5965
5966
5967
5968
5969
5970
5971
5972
5973
5974
5975
5976
5977
5978
5979
5980
5981
5982
5983
5984
5985
5986
5987
5988
5989
5990
5991
5992
5993
5994
5995
5996
5997
5998
5999
```

```
6000 //
6001 // File descriptors
6002 //
6003
6004 #include "types.h"
6005 #include "defs.h"
6006 #include "param.h"
6007 #include "fs.h"
6008 #include "file.h"
6009 #include "spinlock.h"
6010
6011 struct devsw devsw[NDEV];
6012 struct {
6013   struct spinlock lock;
6014   struct file file[NFILE];
6015 } ftable;
6016
6017 void
6018 fileinit(void)
6019 {
6020   initlock(&ftable.lock, "ftable");
6021 }
6022
6023 // Allocate a file structure.
6024 struct file*
6025 filealloc(void)
6026 {
6027   struct file *f;
6028
6029   acquire(&ftable.lock);
6030   for(f = ftable.file; f < ftable.file + NFILE; f++){
6031     if(f->ref == 0){
6032       f->ref = 1;
6033       release(&ftable.lock);
6034       return f;
6035     }
6036   }
6037   release(&ftable.lock);
6038   return 0;
6039 }
6040
6041
6042
6043
6044
6045
6046
6047
6048
6049
```



```
6050 // Increment ref count for file f.
6051 struct file*
6052 filedup(struct file *f)
6053 {
6054   acquire(&ftable.lock);
6055   if(f->ref < 1)
6056     panic("filedup");
6057   f->ref++;
6058   release(&ftable.lock);
6059   return f;
6060 }
6061
6062 // Close file f.  (Decrement ref count, close when reaches 0.)
6063 void
6064 fileclose(struct file *f)
6065 {
6066   struct file ff;
6067
6068   acquire(&ftable.lock);
6069   if(f->ref < 1)
6070     panic("fileclose");
6071   if(--f->ref > 0){
6072     release(&ftable.lock);
6073     return;
6074   }
6075   ff = *f;
6076   f->ref = 0;
6077   f->type = FD_NONE;
6078   release(&ftable.lock);
6079
6080   if(ff.type == FD_PIPE)
6081     pipeclose(ff.pipe, ff.writable);
6082   else if(ff.type == FD_INODE){
6083     begin_op();
6084     iput(ff.ip);
6085     end_op();
6086   }
6087 }
6088
6089
6090
6091
6092
6093
6094
6095
6096
6097
6098
6099
```

```
6100 // Get metadata about file f.
6101 int
6102 filestat(struct file *f, struct stat *st)
6103 {
6104   if(f->type == FD_INODE){
6105     ilock(f->ip);
6106     stati(f->ip, st);
6107     iunlock(f->ip);
6108     return 0;
6109   }
6110   return -1;
6111 }
6112
6113 // Read from file f.
6114 int
6115 fileread(struct file *f, char *addr, int n)
6116 {
6117   int r;
6118
6119   if(f->readable == 0)
6120     return -1;
6121   if(f->type == FD_PIPE)
6122     return piperead(f->pipe, addr, n);
6123   if(f->type == FD_INODE){
6124     ilock(f->ip);
6125     if((r = readi(f->ip, addr, f->off, n)) > 0)
6126       f->off += r;
6127     iunlock(f->ip);
6128     return r;
6129   }
6130   panic("fileread");
6131 }
6132
6133
6134
6135
6136
6137
6138
6139
6140
6141
6142
6143
6144
6145
6146
6147
6148
6149
```



```
6150 // Write to file f.
6151 int
6152 filewrite(struct file *f, char *addr, int n)
6153 {
6154   int r;
6155
6156   if(f->writable == 0)
6157     return -1;
6158   if(f->type == FD_PIPE)
6159     return pipewrite(f->pipe, addr, n);
6160   if(f->type == FD_INODE){
6161     // write a few blocks at a time to avoid exceeding
6162     // the maximum log transaction size, including
6163     // i-node, indirect block, allocation blocks,
6164     // and 2 blocks of slop for non-aligned writes.
6165     // this really belongs lower down, since writei()
6166     // might be writing a device like the console.
6167     int max = ((LOGSIZE-1-1-2) / 2) * 512;
6168     int i = 0;
6169     while(i < n){
6170       int n1 = n - i;
6171       if(n1 > max)
6172         n1 = max;
6173
6174       begin_op();
6175       ilock(f->ip);
6176       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
6177         f->off += r;
6178       iunlock(f->ip);
6179       end_op();
6180
6181       if(r < 0)
6182         break;
6183       if(r != n1)
6184         panic("short filewrite");
6185       i += r;
6186     }
6187     return i == n ? n : -1;
6188   }
6189   panic("filewrite");
6190 }
6191
6192
6193
6194
6195
6196
6197
6198
6199
```

```
6200 //
6201 // File-system system calls.
6202 // Mostly argument checking, since we don't trust
6203 // user code, and calls into file.c and fs.c.
6204 //
6205
6206 #include "types.h"
6207 #include "defs.h"
6208 #include "param.h"
6209 #include "stat.h"
6210 #include "mmu.h"
6211 #include "proc.h"
6212 #include "fs.h"
6213 #include "file.h"
6214 #include "fcntl.h"
6215
6216 // Fetch the nth word-sized system call argument as a file descriptor
6217 // and return both the descriptor and the corresponding struct file.
6218 static int
6219 argfd(int n, int *pfd, struct file **pf)
6220 {
6221   int fd;
6222   struct file *f;
6223
6224   if(argint(n, &fd) < 0)
6225     return -1;
6226   if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
6227     return -1;
6228   if(pfd)
6229     *pfd = fd;
6230   if(pf)
6231     *pf = f;
6232   return 0;
6233 }
6234
6235 // Allocate a file descriptor for the given file.
6236 // Takes over file reference from caller on success.
6237 static int
6238 fdalloc(struct file *f)
6239 {
6240   int fd;
6241
6242   for(fd = 0; fd < NOFILE; fd++){
6243     if(proc->ofile[fd] == 0){
6244       proc->ofile[fd] = f;
6245       return fd;
6246     }
6247   }
6248   return -1;
6249 }
```



```
6250 int
6251 sys_dup(void)
6252 {
6253   struct file *f;
6254   int fd;
6255
6256   if(argfd(0, 0, &f) < 0)
6257     return -1;
6258   if((fd=fdalloc(f)) < 0)
6259     return -1;
6260   filedup(f);
6261   return fd;
6262 }
6263
6264 int
6265 sys_read(void)
6266 {
6267   struct file *f;
6268   int n;
6269   char *p;
6270
6271   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6272     return -1;
6273   return fileread(f, p, n);
6274 }
6275
6276 int
6277 sys_write(void)
6278 {
6279   struct file *f;
6280   int n;
6281   char *p;
6282
6283   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6284     return -1;
6285   return filewrite(f, p, n);
6286 }
6287
6288 int
6289 sys_close(void)
6290 {
6291   int fd;
6292   struct file *f;
6293
6294   if(argfd(0, &fd, &f) < 0)
6295     return -1;
6296   proc->ofile[fd] = 0;
6297   fileclose(f);
6298   return 0;
6299 }
```

```
6300 int
6301 sys_fstat(void)
6302 {
6303   struct file *f;
6304   struct stat *st;
6305
6306   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
6307     return -1;
6308   return filestat(f, st);
6309 }
6310
6311 // Create the path new as a link to the same inode as old.
6312 int
6313 sys_link(void)
6314 {
6315   char name[DIRSIZ], *new, *old;
6316   struct inode *dp, *ip;
6317
6318   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
6319     return -1;
6320
6321   begin_op();
6322   if((ip = namei(old)) == 0){
6323     end_op();
6324     return -1;
6325   }
6326
6327   ilock(ip);
6328   if(ip->type == T_DIR){
6329     iunlockput(ip);
6330     end_op();
6331     return -1;
6332   }
6333
6334   ip->nlink++;
6335   iupdate(ip);
6336   iunlock(ip);
6337
6338   if((dp = nameiparent(new, name)) == 0)
6339     goto bad;
6340   ilock(dp);
6341   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
6342     iunlockput(dp);
6343     goto bad;
6344   }
6345   iunlockput(dp);
6346   iput(ip);
6347
6348   end_op();
6349
```



```
6350   return 0;
6351
6352 bad:
6353   ilock(ip);
6354   ip->nlink--;
6355   iupdate(ip);
6356   iunlockput(ip);
6357   end_op();
6358   return -1;
6359 }
6360
6361 // Is the directory dp empty except for "." and ".." ?
6362 static int
6363 isdirempty(struct inode *dp)
6364 {
6365   int off;
6366   struct dirent de;
6367
6368   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
6369     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6370       panic("isdirempty: readi");
6371     if(de.inum != 0)
6372       return 0;
6373   }
6374   return 1;
6375 }
6376
6377
6378
6379
6380
6381
6382
6383
6384
6385
6386
6387
6388
6389
6390
6391
6392
6393
6394
6395
6396
6397
6398
6399
```

```
6400 int
6401 sys_unlink(void)
6402 {
6403   struct inode *ip, *dp;
6404   struct dirent de;
6405   char name[DIRSIZ], *path;
6406   uint off;
6407
6408   if(argstr(0, &path) < 0)
6409     return -1;
6410
6411   begin_op();
6412   if((dp = nameiparent(path, name)) == 0){
6413     end_op();
6414     return -1;
6415   }
6416
6417   ilock(dp);
6418
6419   // Cannot unlink "." or "..".
6420   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6421     goto bad;
6422
6423   if((ip = dirlookup(dp, name, &off)) == 0)
6424     goto bad;
6425   ilock(ip);
6426
6427   if(ip->nlink < 1)
6428     panic("unlink: nlink < 1");
6429   if(ip->type == T_DIR && !isdirempty(ip)){
6430     iunlockput(ip);
6431     goto bad;
6432   }
6433
6434   memset(&de, 0, sizeof(de));
6435   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6436     panic("unlink: writei");
6437   if(ip->type == T_DIR){
6438     dp->nlink--;
6439     iupdate(dp);
6440   }
6441   iunlockput(dp);
6442
6443   ip->nlink--;
6444   iupdate(ip);
6445   iunlockput(ip);
6446
6447   end_op();
6448
6449   return 0;
```



```
6450 bad:
6451   iunlockput(dp);
6452   end_op();
6453   return -1;
6454 }
6455
6456 static struct inode*
6457 create(char *path, short type, short major, short minor)
6458 {
6459   uint off;
6460   struct inode *ip, *dp;
6461   char name[DIRSIZ];
6462
6463   if((dp = nameiparent(path, name)) == 0)
6464     return 0;
6465   ilock(dp);
6466
6467   if((ip = dirlookup(dp, name, &off)) != 0){
6468     iunlockput(dp);
6469     ilock(ip);
6470     if(type == T_FILE && ip->type == T_FILE)
6471       return ip;
6472     iunlockput(ip);
6473     return 0;
6474   }
6475
6476   if((ip = ialloc(dp->dev, type)) == 0)
6477     panic("create: ialloc");
6478
6479   ilock(ip);
6480   ip->major = major;
6481   ip->minor = minor;
6482   ip->nlink = 1;
6483   iupdate(ip);
6484
6485   if(type == T_DIR){  // Create . and .. entries.
6486     dp->nlink++;  // for ".."
6487     iupdate(dp);
6488     // No ip->nlink++ for ".": avoid cyclic ref count.
6489     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6490       panic("create dots");
6491   }
6492
6493   if(dirlink(dp, name, ip->inum) < 0)
6494     panic("create: dirlink");
6495
6496   iunlockput(dp);
6497
6498   return ip;
6499 }
```

```
6500 int
6501 sys_open(void)
6502 {
6503   char *path;
6504   int fd, omode;
6505   struct file *f;
6506   struct inode *ip;
6507
6508   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6509     return -1;
6510
6511   begin_op();
6512
6513   if(omode & O_CREATE){
6514     ip = create(path, T_FILE, 0, 0);
6515     if(ip == 0){
6516       end_op();
6517       return -1;
6518     }
6519   } else {
6520     if((ip = namei(path)) == 0){
6521       end_op();
6522       return -1;
6523     }
6524     ilock(ip);
6525     if(ip->type == T_DIR && omode != O_RDONLY){
6526       iunlockput(ip);
6527       end_op();
6528       return -1;
6529     }
6530   }
6531
6532   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6533     if(f)
6534       fileclose(f);
6535     iunlockput(ip);
6536     end_op();
6537     return -1;
6538   }
6539   iunlock(ip);
6540   end_op();
6541
6542   f->type = FD_INODE;
6543   f->ip = ip;
6544   f->off = 0;
6545   f->readable = !(omode & O_WRONLY);
6546   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6547   return fd;
6548 }
6549
```



```
6550 int
6551 sys_mkdir(void)
6552 {
6553   char *path;
6554   struct inode *ip;
6555
6556   begin_op();
6557   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
6558     end_op();
6559     return -1;
6560   }
6561   iunlockput(ip);
6562   end_op();
6563   return 0;
6564 }
6565
6566 int
6567 sys_mknod(void)
6568 {
6569   struct inode *ip;
6570   char *path;
6571   int len;
6572   int major, minor;
6573
6574   begin_op();
6575   if((len=argstr(0, &path)) < 0 ||
6576      argint(1, &major) < 0 ||
6577      argint(2, &minor) < 0 ||
6578      (ip = create(path, T_DEV, major, minor)) == 0){
6579     end_op();
6580     return -1;
6581   }
6582   iunlockput(ip);
6583   end_op();
6584   return 0;
6585 }
6586
6587
6588
6589
6590
6591
6592
6593
6594
6595
6596
6597
6598
6599
```

```
6600 int
6601 sys_chdir(void)
6602 {
6603   char *path;
6604   struct inode *ip;
6605
6606   begin_op();
6607   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
6608     end_op();
6609     return -1;
6610   }
6611   ilock(ip);
6612   if(ip->type != T_DIR){
6613     iunlockput(ip);
6614     end_op();
6615     return -1;
6616   }
6617   iunlock(ip);
6618   iput(proc->cwd);
6619   end_op();
6620   proc->cwd = ip;
6621   return 0;
6622 }
6623
6624 int
6625 sys_exec(void)
6626 {
6627   char *path, *argv[MAXARG];
6628   int i;
6629   uint uargv, uarg;
6630
6631   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6632     return -1;
6633   }
6634   memset(argv, 0, sizeof(argv));
6635   for(i=0;; i++){
6636     if(i >= NELEM(argv))
6637       return -1;
6638     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6639       return -1;
6640     if(uarg == 0){
6641       argv[i] = 0;
6642       break;
6643     }
6644     if(fetchstr(uarg, &argv[i]) < 0)
6645       return -1;
6646   }
6647   return exec(path, argv);
6648 }
6649
```



```
6650 int
6651 sys_pipe(void)
6652 {
6653   int *fd;
6654   struct file *rf, *wf;
6655   int fd0, fd1;
6656
6657   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6658     return -1;
6659   if(pipealloc(&rf, &wf) < 0)
6660     return -1;
6661   fd0 = -1;
6662   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6663     if(fd0 >= 0)
6664       proc->ofile[fd0] = 0;
6665     fileclose(rf);
6666     fileclose(wf);
6667     return -1;
6668   }
6669   fd[0] = fd0;
6670   fd[1] = fd1;
6671   return 0;
6672 }
6673
6674
6675
6676
6677
6678
6679
6680
6681
6682
6683
6684
6685
6686
6687
6688
6689
6690
6691
6692
6693
6694
6695
6696
6697
6698
6699
```

```
6700 #include "types.h"
6701 #include "param.h"
6702 #include "memlayout.h"
6703 #include "mmu.h"
6704 #include "proc.h"
6705 #include "defs.h"
6706 #include "x86.h"
6707 #include "elf.h"
6708
6709 int
6710 exec(char *path, char **argv)
6711 {
6712   char *s, *last;
6713   int i, off;
6714   uint argc, sz, sp, ustack[3+MAXARG+1];
6715   struct elfhdr elf;
6716   struct inode *ip;
6717   struct proghdr ph;
6718   pde_t *pgdir, *oldpgdir;
6719
6720   begin_op();
6721   if((ip = namei(path)) == 0){
6722     end_op();
6723     return -1;
6724   }
6725   ilock(ip);
6726   pgdir = 0;
6727
6728   // Check ELF header
6729   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
6730     goto bad;
6731   if(elf.magic != ELF_MAGIC)
6732     goto bad;
6733
6734   if((pgdir = setupkvm()) == 0)
6735     goto bad;
6736
6737   // Load program into memory.
6738   sz = 0;
6739   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6740     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6741       goto bad;
6742     if(ph.type != ELF_PROG_LOAD)
6743       continue;
6744     if(ph.memsz < ph.filesz)
6745       goto bad;
6746     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6747       goto bad;
6748     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6749       goto bad;
```



```
6750   }
6751   iunlockput(ip);
6752   end_op();
6753   ip = 0;
6754
6755   // Allocate two pages at the next page boundary.
6756   // Make the first inaccessible.  Use the second as the user stack.
6757   sz = PGROUNDUP(sz);
6758   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
6759     goto bad;
6760   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6761   sp = sz;
6762
6763   // Push argument strings, prepare rest of stack in ustack.
6764   for(argc = 0; argv[argc]; argc++) {
6765     if(argc >= MAXARG)
6766       goto bad;
6767     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6768     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6769       goto bad;
6770     ustack[3+argc] = sp;
6771   }
6772   ustack[3+argc] = 0;
6773
6774   ustack[0] = 0xffffffff;  // fake return PC
6775   ustack[1] = argc;
6776   ustack[2] = sp - (argc+1)*4;  // argv pointer
6777
6778   sp -= (3+argc+1) * 4;
6779   if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6780     goto bad;
6781
6782   // Save program name for debugging.
6783   for(last=s=path; *s; s++)
6784     if(*s == '/')
6785       last = s+1;
6786   safestrcpy(proc->name, last, sizeof(proc->name));
6787
6788   // Commit to the user image.
6789   oldpgdir = proc->pgdir;
6790   proc->pgdir = pgdir;
6791   proc->sz = sz;
6792   proc->tf->eip = elf.entry;  // main
6793   proc->tf->esp = sp;
6794   switchuvm(proc);
6795   freevm(oldpgdir);
6796   return 0;
6797
6798
6799
```

```
6800  bad:
6801    if(pgdir)
6802      freevm(pgdir);
6803    if(ip){
6804      iunlockput(ip);
6805      end_op();
6806    }
6807    return -1;
6808 }
6809
6810
6811
6812
6813
6814
6815
6816
6817
6818
6819
6820
6821
6822
6823
6824
6825
6826
6827
6828
6829
6830
6831
6832
6833
6834
6835
6836
6837
6838
6839
6840
6841
6842
6843
6844
6845
6846
6847
6848
6849
```



```
6850 #include "types.h"
6851 #include "defs.h"
6852 #include "param.h"
6853 #include "mmu.h"
6854 #include "proc.h"
6855 #include "fs.h"
6856 #include "file.h"
6857 #include "spinlock.h"
6858
6859 #define PIPESIZE 512
6860
6861 struct pipe {
6862   struct spinlock lock;
6863   char data[PIPESIZE];
6864   uint nread;     // number of bytes read
6865   uint nwrite;    // number of bytes written
6866   int readopen;   // read fd is still open
6867   int writeopen;  // write fd is still open
6868 };
6869
6870 int
6871 pipealloc(struct file **f0, struct file **f1)
6872 {
6873   struct pipe *p;
6874
6875   p = 0;
6876   *f0 = *f1 = 0;
6877   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
6878     goto bad;
6879   if((p = (struct pipe*)kalloc()) == 0)
6880     goto bad;
6881   p->readopen = 1;
6882   p->writeopen = 1;
6883   p->nwrite = 0;
6884   p->nread = 0;
6885   initlock(&p->lock, "pipe");
6886   (*f0)->type = FD_PIPE;
6887   (*f0)->readable = 1;
6888   (*f0)->writable = 0;
6889   (*f0)->pipe = p;
6890   (*f1)->type = FD_PIPE;
6891   (*f1)->readable = 0;
6892   (*f1)->writable = 1;
6893   (*f1)->pipe = p;
6894   return 0;
6895
6896
6897
6898
6899
```

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

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


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

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



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

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

```
7000 #include "types.h"
7001 #include "x86.h"
7002
7003 void*
7004 memset(void *dst, int c, uint n)
7005 {
7006   if ((int)dst%4 == 0 && n%4 == 0){
7007     c &= 0xFF;
7008     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
7009   } else
7010     stosb(dst, c, n);
7011   return dst;
7012 }
7013
7014 int
7015 memcmp(const void *v1, const void *v2, uint n)
7016 {
7017   const uchar *s1, *s2;
7018
7019   s1 = v1;
7020   s2 = v2;
7021   while(n-- > 0){
7022     if(*s1 != *s2)
7023       return *s1 - *s2;
7024     s1++, s2++;
7025   }
7026
7027   return 0;
7028 }
7029
7030 void*
7031 memmove(void *dst, const void *src, uint n)
7032 {
7033   const char *s;
7034   char *d;
7035
7036   s = src;
7037   d = dst;
7038   if(s < d && s + n > d){
7039     s += n;
7040     d += n;
7041     while(n-- > 0)
7042       *--d = *--s;
7043   } else
7044     while(n-- > 0)
7045       *d++ = *s++;
7046
7047   return dst;
7048 }
7049
```



```
7050 // memcpy exists to placate GCC.  Use memmove.
7051 void*
7052 memcpy(void *dst, const void *src, uint n)
7053 {
7054   return memmove(dst, src, n);
7055 }
7056
7057 int
7058 strncmp(const char *p, const char *q, uint n)
7059 {
7060   while(n > 0 && *p && *p == *q)
7061     n--, p++, q++;
7062   if(n == 0)
7063     return 0;
7064   return (uchar)*p - (uchar)*q;
7065 }
7066
7067 char*
7068 strncpy(char *s, const char *t, int n)
7069 {
7070   char *os;
7071
7072   os = s;
7073   while(n-- > 0 && (*s++ = *t++) != 0)
7074     ;
7075   while(n-- > 0)
7076     *s++ = 0;
7077   return os;
7078 }
7079
7080 // Like strncpy but guaranteed to NUL-terminate.
7081 char*
7082 safestrcpy(char *s, const char *t, int n)
7083 {
7084   char *os;
7085
7086   os = s;
7087   if(n <= 0)
7088     return os;
7089   while(--n > 0 && (*s++ = *t++) != 0)
7090     ;
7091   *s = 0;
7092   return os;
7093 }
7094
7095
7096
7097
7098
7099
```

```
7100 int
7101 strlen(const char *s)
7102 {
7103   int n;
7104
7105   for(n = 0; s[n]; n++)
7106     ;
7107   return n;
7108 }
7109
7110
7111
7112
7113
7114
7115
7116
7117
7118
7119
7120
7121
7122
7123
7124
7125
7126
7127
7128
7129
7130
7131
7132
7133
7134
7135
7136
7137
7138
7139
7140
7141
7142
7143
7144
7145
7146
7147
7148
7149
```



```
7150 // See MultiProcessor Specification Version 1.[14]
7151
7152 struct mp {             // floating pointer
7153   uchar signature[4];           // "_MP_"
7154   void *physaddr;               // phys addr of MP config table
7155   uchar length;                 // 1
7156   uchar specrev;                // [14]
7157   uchar checksum;               // all bytes must add up to 0
7158   uchar type;                   // MP system config type
7159   uchar imcrp;
7160   uchar reserved[3];
7161 };
7162
7163 struct mpconf {         // configuration table header
7164   uchar signature[4];           // "PCMP"
7165   ushort length;                // total table length
7166   uchar version;                // [14]
7167   uchar checksum;               // all bytes must add up to 0
7168   uchar product[20];            // product id
7169   uint *oemtable;               // OEM table pointer
7170   ushort oemlength;             // OEM table length
7171   ushort entry;                 // entry count
7172   uint *lapicaddr;              // address of local APIC
7173   ushort xlength;               // extended table length
7174   uchar xchecksum;              // extended table checksum
7175   uchar reserved;
7176 };
7177
7178 struct mpproc {         // processor table entry
7179   uchar type;                   // entry type (0)
7180   uchar apicid;                 // local APIC id
7181   uchar version;                // local APIC verison
7182   uchar flags;                  // CPU flags
7183     #define MPBOOT 0x02           // This proc is the bootstrap processor.
7184   uchar signature[4];           // CPU signature
7185   uint feature;                 // feature flags from CPUID instruction
7186   uchar reserved[8];
7187 };
7188
7189 struct mpioapic {       // I/O APIC table entry
7190   uchar type;                   // entry type (2)
7191   uchar apicno;                 // I/O APIC id
7192   uchar version;                // I/O APIC version
7193   uchar flags;                  // I/O APIC flags
7194   uint *addr;                  // I/O APIC address
7195 };
7196
7197
7198
7199
```

```
7200 // Table entry types
7201 #define MPPROC    0x00  // One per processor
7202 #define MPBUS     0x01  // One per bus
7203 #define MPIOAPIC  0x02  // One per I/O APIC
7204 #define MPIOINTR  0x03  // One per bus interrupt source
7205 #define MPLINTR   0x04  // One per system interrupt source
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 // Blank page.
7251
7252
7253
7254
7255
7256
7257
7258
7259
7260
7261
7262
7263
7264
7265
7266
7267
7268
7269
7270
7271
7272
7273
7274
7275
7276
7277
7278
7279
7280
7281
7282
7283
7284
7285
7286
7287
7288
7289
7290
7291
7292
7293
7294
7295
7296
7297
7298
7299
```

```
7300 // Multiprocessor support
7301 // Search memory for MP description structures.
7302 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
7303
7304 #include "types.h"
7305 #include "defs.h"
7306 #include "param.h"
7307 #include "memlayout.h"
7308 #include "mp.h"
7309 #include "x86.h"
7310 #include "mmu.h"
7311 #include "proc.h"
7312
7313 struct cpu cpus[NCPU];
7314 static struct cpu *bcpu;
7315 int ismp;
7316 int ncpu;
7317 uchar ioapicid;
7318
7319 int
7320 mpbcpu(void)
7321 {
7322   return bcpu-cpus;
7323 }
7324
7325 static uchar
7326 sum(uchar *addr, int len)
7327 {
7328   int i, sum;
7329
7330   sum = 0;
7331   for(i=0; i<len; i++)
7332     sum += addr[i];
7333   return sum;
7334 }
7335
7336 // Look for an MP structure in the len bytes at addr.
7337 static struct mp*
7338 mpsearch1(uint a, int len)
7339 {
7340   uchar *e, *p, *addr;
7341
7342   addr = p2v(a);
7343   e = addr+len;
7344   for(p = addr; p < e; p += sizeof(struct mp))
7345     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
7346       return (struct mp*)p;
7347   return 0;
7348 }
7349
```



```
7350 // Search for the MP Floating Pointer Structure, which according to the
7351 // spec is in one of the following three locations:
7352 // 1) in the first KB of the EBDA;
7353 // 2) in the last KB of system base memory;
7354 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
7355 static struct mp*
7356 mpsearch(void)
7357 {
7358   uchar *bda;
7359   uint p;
7360   struct mp *mp;
7361
7362   bda = (uchar *) P2V(0x400);
7363   if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
7364     if((mp = mpsearch1(p, 1024)))
7365       return mp;
7366   } else {
7367     p = ((bda[0x14]<<8)|bda[0x13])*1024;
7368     if((mp = mpsearch1(p-1024, 1024)))
7369       return mp;
7370   }
7371   return mpsearch1(0xF0000, 0x10000);
7372 }
7373
7374 // Search for an MP configuration table.  For now,
7375 // don't accept the default configurations (physaddr == 0).
7376 // Check for correct signature, calculate the checksum and,
7377 // if correct, check the version.
7378 // To do: check extended table checksum.
7379 static struct mpconf*
7380 mpconfig(struct mp **pmp)
7381 {
7382   struct mpconf *conf;
7383   struct mp *mp;
7384
7385   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
7386     return 0;
7387   conf = (struct mpconf*) p2v((uint) mp->physaddr);
7388   if(memcmp(conf, "PCMP", 4) != 0)
7389     return 0;
7390   if(conf->version != 1 && conf->version != 4)
7391     return 0;
7392   if(sum((uchar*)conf, conf->length) != 0)
7393     return 0;
7394   *pmp = mp;
7395   return conf;
7396 }
7397
7398
7399
```

```
7400 void
7401 mpinit(void)
7402 {
7403   uchar *p, *e;
7404   struct mp *mp;
7405   struct mpconf *conf;
7406   struct mpproc *proc;
7407   struct mpioapic *ioapic;
7408
7409   bcpu = &cpus[0];
7410   if((conf = mpconfig(&mp)) == 0)
7411     return;
7412   ismp = 1;
7413   lapic = (uint*)conf->lapicaddr;
7414   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
7415     switch(*p){
7416     case MPPROC:
7417       proc = (struct mpproc*)p;
7418       if(ncpu != proc->apicid){
7419         cprintf("mpinit: ncpu=%d apicid=%d\n", ncpu, proc->apicid);
7420         ismp = 0;
7421       }
7422       if(proc->flags & MPBOOT)
7423         bcpu = &cpus[ncpu];
7424       cpus[ncpu].id = ncpu;
7425       ncpu++;
7426       p += sizeof(struct mpproc);
7427       continue;
7428     case MPIOAPIC:
7429       ioapic = (struct mpioapic*)p;
7430       ioapicid = ioapic->apicno;
7431       p += sizeof(struct mpioapic);
7432       continue;
7433     case MPBUS:
7434     case MPIOINTR:
7435     case MPLINTR:
7436       p += 8;
7437       continue;
7438     default:
7439       cprintf("mpinit: unknown config type %x\n", *p);
7440       ismp = 0;
7441     }
7442   }
7443   if(!ismp){
7444     // Didn't like what we found; fall back to no MP.
7445     ncpu = 1;
7446     lapic = 0;
7447     ioapicid = 0;
7448     return;
7449   }
```



```
7450   if(mp->imcrp){
7451     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
7452     // But it would on real hardware.
7453     outb(0x22, 0x70);   // Select IMCR
7454     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
7455   }
7456 }
7457
7458
7459
7460
7461
7462
7463
7464
7465
7466
7467
7468
7469
7470
7471
7472
7473
7474
7475
7476
7477
7478
7479
7480
7481
7482
7483
7484
7485
7486
7487
7488
7489
7490
7491
7492
7493
7494
7495
7496
7497
7498
7499
```

```
7500 // The local APIC manages internal (non-I/O) interrupts.
7501 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
7502
7503 #include "types.h"
7504 #include "defs.h"
7505 #include "date.h"
7506 #include "memlayout.h"
7507 #include "traps.h"
7508 #include "mmu.h"
7509 #include "x86.h"
7510
7511 // Local APIC registers, divided by 4 for use as uint[] indices.
7512 #define ID      (0x0020/4)   // ID
7513 #define VER     (0x0030/4)   // Version
7514 #define TPR     (0x0080/4)   // Task Priority
7515 #define EOI     (0x00B0/4)   // EOI
7516 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
7517   #define ENABLE     0x00000100   // Unit Enable
7518 #define ESR     (0x0280/4)   // Error Status
7519 #define ICRLO   (0x0300/4)   // Interrupt Command
7520   #define INIT       0x00000500   // INIT/RESET
7521   #define STARTUP    0x00000600   // Startup IPI
7522   #define DELIVS     0x00001000   // Delivery status
7523   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
7524   #define DEASSERT   0x00000000
7525   #define LEVEL      0x00008000   // Level triggered
7526   #define BCAST      0x00080000   // Send to all APICs, including self.
7527   #define BUSY       0x00001000
7528   #define FIXED      0x00000000
7529 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
7530 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
7531   #define X1         0x0000000B   // divide counts by 1
7532   #define PERIODIC   0x00020000   // Periodic
7533 #define PCINT   (0x0340/4)   // Performance Counter LVT
7534 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
7535 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
7536 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
7537   #define MASKED     0x00010000   // Interrupt masked
7538 #define TICR    (0x0380/4)   // Timer Initial Count
7539 #define TCCR    (0x0390/4)   // Timer Current Count
7540 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
7541
7542 volatile uint *lapic;  // Initialized in mp.c
7543
7544 static void
7545 lapicw(int index, int value)
7546 {
7547   lapic[index] = value;
7548   lapic[ID];  // wait for write to finish, by reading
7549 }
```

```
7600 void
7601 lapicinit(void)
7602 {
7603   if(!lapic)
7604     return;
7605
7606   // Enable local APIC; set spurious interrupt vector.
7607   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
7608
7609   // The timer repeatedly counts down at bus frequency
7610   // from lapic[TICR] and then issues an interrupt.
7611   // If xv6 cared more about precise timekeeping,
7612   // TICR would be calibrated using an external time source.
7613   lapicw(TDCR, X1);
7614   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
7615   lapicw(TICR, 10000000);
7616
7617   // Disable logical interrupt lines.
7618   lapicw(LINT0, MASKED);
7619   lapicw(LINT1, MASKED);
7620
7621   // Disable performance counter overflow interrupts
7622   // on machines that provide that interrupt entry.
7623   if(((lapic[VER]>>16) & 0xFF) >= 4)
7624     lapicw(PCINT, MASKED);
7625
7626   // Map error interrupt to IRQ_ERROR.
7627   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
7628
7629   // Clear error status register (requires back-to-back writes).
7630   lapicw(ESR, 0);
7631   lapicw(ESR, 0);
7632
7633   // Ack any outstanding interrupts.
7634   lapicw(EOI, 0);
7635
7636   // Send an Init Level De-Assert to synchronise arbitration ID's.
7637   lapicw(ICRHI, 0);
7638   lapicw(ICRLO, BCAST | INIT | LEVEL);
7639   while(lapic[ICRLO] & DELIVS)
7640     ;
7641
7642   // Enable interrupts on the APIC (but not on the processor).
7643   lapicw(TPR, 0);
7644 }
7645
7646
7647
7648
7649
```



```
7650 int
7651 cpunum(void)
7652 {
7653   // Cannot call cpu when interrupts are enabled:
7654   // result not guaranteed to last long enough to be used!
7655   // Would prefer to panic but even printing is chancy here:
7656   // almost everything, including cprintf and panic, calls cpu,
7657   // often indirectly through acquire and release.
7658   if(readeflags()&FL_IF){
7659     static int n;
7660     if(n++ == 0)
7661       cprintf("cpu called from %x with interrupts enabled\n",
7662         __builtin_return_address(0));
7663   }
7664
7665   if(lapic)
7666     return lapic[ID]>>24;
7667   return 0;
7668 }
7669
7670 // Acknowledge interrupt.
7671 void
7672 lapiceoi(void)
7673 {
7674   if(lapic)
7675     lapicw(EOI, 0);
7676 }
7677
7678 // Spin for a given number of microseconds.
7679 // On real hardware would want to tune this dynamically.
7680 void
7681 microdelay(int us)
7682 {
7683 }
7684
7685 #define CMOS_PORT    0x70
7686 #define CMOS_RETURN  0x71
7687
7688 // Start additional processor running entry code at addr.
7689 // See Appendix B of MultiProcessor Specification.
7690 void
7691 lapicstartap(uchar apicid, uint addr)
7692 {
7693   int i;
7694   ushort *wrv;
7695
7696   // "The BSP must initialize CMOS shutdown code to 0AH
7697   // and the warm reset vector (DWORD based at 40:67) to point at
7698   // the AP startup code prior to the [universal startup algorithm]."
7699   outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
```

```
7700   outb(CMOS_PORT+1, 0x0A);
7701   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
7702   wrv[0] = 0;
7703   wrv[1] = addr >> 4;
7704
7705   // "Universal startup algorithm."
7706   // Send INIT (level-triggered) interrupt to reset other CPU.
7707   lapicw(ICRHI, apicid<<24);
7708   lapicw(ICRLO, INIT | LEVEL | ASSERT);
7709   microdelay(200);
7710   lapicw(ICRLO, INIT | LEVEL);
7711   microdelay(100);    // should be 10ms, but too slow in Bochs!
7712
7713   // Send startup IPI (twice!) to enter code.
7714   // Regular hardware is supposed to only accept a STARTUP
7715   // when it is in the halted state due to an INIT.  So the second
7716   // should be ignored, but it is part of the official Intel algorithm.
7717   // Bochs complains about the second one.  Too bad for Bochs.
7718   for(i = 0; i < 2; i++){
7719     lapicw(ICRHI, apicid<<24);
7720     lapicw(ICRLO, STARTUP | (addr>>12));
7721     microdelay(200);
7722   }
7723 }
7724
7725 #define CMOS_STATA   0x0a
7726 #define CMOS_STATB   0x0b
7727 #define CMOS_UIP    (1 << 7)        // RTC update in progress
7728
7729 #define SECS    0x00
7730 #define MINS    0x02
7731 #define HOURS   0x04
7732 #define DAY     0x07
7733 #define MONTH   0x08
7734 #define YEAR    0x09
7735
7736 static uint cmos_read(uint reg)
7737 {
7738   outb(CMOS_PORT,  reg);
7739   microdelay(200);
7740
7741   return inb(CMOS_RETURN);
7742 }
7743
7744
7745
7746
7747
7748
7749
```



```
7750 static void fill_rtcdate(struct rtcdate *r)
7751 {
7752   r->second = cmos_read(SECS);
7753   r->minute = cmos_read(MINS);
7754   r->hour   = cmos_read(HOURS);
7755   r->day    = cmos_read(DAY);
7756   r->month  = cmos_read(MONTH);
7757   r->year   = cmos_read(YEAR);
7758 }
7759
7760 // qemu seems to use 24-hour GWT and the values are BCD encoded
7761 void cmostime(struct rtcdate *r)
7762 {
7763   struct rtcdate t1, t2;
7764   int sb, bcd;
7765
7766   sb = cmos_read(CMOS_STATB);
7767
7768   bcd = (sb & (1 << 2)) == 0;
7769
7770   // make sure CMOS doesn't modify time while we read it
7771   for (;;) {
7772     fill_rtcdate(&t1);
7773     if (cmos_read(CMOS_STATA) & CMOS_UIP)
7774         continue;
7775     fill_rtcdate(&t2);
7776     if (memcmp(&t1, &t2, sizeof(t1)) == 0)
7777       break;
7778   }
7779
7780   // convert
7781   if (bcd) {
7782 #define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
7783     CONV(second);
7784     CONV(minute);
7785     CONV(hour  );
7786     CONV(day   );
7787     CONV(month );
7788     CONV(year  );
7789 #undef     CONV
7790   }
7791
7792   *r = t1;
7793   r->year += 2000;
7794 }
7795
7796
7797
7798
7799
```

```
7800 // The I/O APIC manages hardware interrupts for an SMP system.
7801 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
7802 // See also picirq.c.
7803
7804 #include "types.h"
7805 #include "defs.h"
7806 #include "traps.h"
7807
7808 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
7809
7810 #define REG_ID     0x00  // Register index: ID
7811 #define REG_VER    0x01  // Register index: version
7812 #define REG_TABLE  0x10  // Redirection table base
7813
7814 // The redirection table starts at REG_TABLE and uses
7815 // two registers to configure each interrupt.
7816 // The first (low) register in a pair contains configuration bits.
7817 // The second (high) register contains a bitmask telling which
7818 // CPUs can serve that interrupt.
7819 #define INT_DISABLED   0x00010000  // Interrupt disabled
7820 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
7821 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
7822 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
7823
7824 volatile struct ioapic *ioapic;
7825
7826 // IO APIC MMIO structure: write reg, then read or write data.
7827 struct ioapic {
7828   uint reg;
7829   uint pad[3];
7830   uint data;
7831 };
7832
7833 static uint
7834 ioapicread(int reg)
7835 {
7836   ioapic->reg = reg;
7837   return ioapic->data;
7838 }
7839
7840 static void
7841 ioapicwrite(int reg, uint data)
7842 {
7843   ioapic->reg = reg;
7844   ioapic->data = data;
7845 }
7846
7847
7848
7849
```



```
7850 void
7851 ioapicinit(void)
7852 {
7853   int i, id, maxintr;
7854
7855   if(!ismp)
7856     return;
7857
7858   ioapic = (volatile struct ioapic*)IOAPIC;
7859   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
7860   id = ioapicread(REG_ID) >> 24;
7861   if(id != ioapicid)
7862     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
7863
7864   // Mark all interrupts edge-triggered, active high, disabled,
7865   // and not routed to any CPUs.
7866   for(i = 0; i <= maxintr; i++){
7867     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
7868     ioapicwrite(REG_TABLE+2*i+1, 0);
7869   }
7870 }
7871
7872 void
7873 ioapicenable(int irq, int cpunum)
7874 {
7875   if(!ismp)
7876     return;
7877
7878   // Mark interrupt edge-triggered, active high,
7879   // enabled, and routed to the given cpunum,
7880   // which happens to be that cpu's APIC ID.
7881   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
7882   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
7883 }
7884
7885
7886
7887
7888
7889
7890
7891
7892
7893
7894
7895
7896
7897
7898
7899
```

```
7900 // Intel 8259A programmable interrupt controllers.
7901
7902 #include "types.h"
7903 #include "x86.h"
7904 #include "traps.h"
7905
7906 // I/O Addresses of the two programmable interrupt controllers
7907 #define IO_PIC1         0x20    // Master (IRQs 0-7)
7908 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
7909
7910 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
7911
7912 // Current IRQ mask.
7913 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
7914 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
7915
7916 static void
7917 picsetmask(ushort mask)
7918 {
7919   irqmask = mask;
7920   outb(IO_PIC1+1, mask);
7921   outb(IO_PIC2+1, mask >> 8);
7922 }
7923
7924 void
7925 picenable(int irq)
7926 {
7927   picsetmask(irqmask & ~(1<<irq));
7928 }
7929
7930 // Initialize the 8259A interrupt controllers.
7931 void
7932 picinit(void)
7933 {
7934   // mask all interrupts
7935   outb(IO_PIC1+1, 0xFF);
7936   outb(IO_PIC2+1, 0xFF);
7937
7938   // Set up master (8259A-1)
7939
7940   // ICW1:  0001g0hi
7941   //    g:  0 = edge triggering, 1 = level triggering
7942   //    h:  0 = cascaded PICs, 1 = master only
7943   //    i:  0 = no ICW4, 1 = ICW4 required
7944   outb(IO_PIC1, 0x11);
7945
7946   // ICW2:  Vector offset
7947   outb(IO_PIC1+1, T_IRQ0);
7948
7949
```



```
7950   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
7951   //        (slave PIC) 3-bit # of slave's connection to master
7952   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
7953
7954   // ICW4:  000nbmap
7955   //    n:  1 = special fully nested mode
7956   //    b:  1 = buffered mode
7957   //    m:  0 = slave PIC, 1 = master PIC
7958   //      (ignored when b is 0, as the master/slave role
7959   //      can be hardwired).
7960   //    a:  1 = Automatic EOI mode
7961   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
7962   outb(IO_PIC1+1, 0x3);
7963
7964   // Set up slave (8259A-2)
7965   outb(IO_PIC2, 0x11);                  // ICW1
7966   outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
7967   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
7968   // NB Automatic EOI mode doesn't tend to work on the slave.
7969   // Linux source code says it's "to be investigated".
7970   outb(IO_PIC2+1, 0x3);                 // ICW4
7971
7972   // OCW3:  0ef01prs
7973   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
7974   //    p:  0 = no polling, 1 = polling mode
7975   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
7976   outb(IO_PIC1, 0x68);             // clear specific mask
7977   outb(IO_PIC1, 0x0a);             // read IRR by default
7978
7979   outb(IO_PIC2, 0x68);             // OCW3
7980   outb(IO_PIC2, 0x0a);             // OCW3
7981
7982   if(irqmask != 0xFFFF)
7983     picsetmask(irqmask);
7984 }
7985
7986
7987
7988
7989
7990
7991
7992
7993
7994
7995
7996
7997
7998
7999
```

```
8000 // PC keyboard interface constants
8001
8002 #define KBSTATP         0x64    // kbd controller status port(I)
8003 #define KBS_DIB         0x01    // kbd data in buffer
8004 #define KBDATAP         0x60    // kbd data port(I)
8005
8006 #define NO              0
8007
8008 #define SHIFT           (1<<0)
8009 #define CTL             (1<<1)
8010 #define ALT             (1<<2)
8011
8012 #define CAPSLOCK        (1<<3)
8013 #define NUMLOCK         (1<<4)
8014 #define SCROLLLOCK      (1<<5)
8015
8016 #define E0ESC           (1<<6)
8017
8018 // Special keycodes
8019 #define KEY_HOME        0xE0
8020 #define KEY_END         0xE1
8021 #define KEY_UP          0xE2
8022 #define KEY_DN          0xE3
8023 #define KEY_LF          0xE4
8024 #define KEY_RT          0xE5
8025 #define KEY_PGUP        0xE6
8026 #define KEY_PGDN        0xE7
8027 #define KEY_INS         0xE8
8028 #define KEY_DEL         0xE9
8029
8030 // C('A') == Control-A
8031 #define C(x) (x - '@')
8032
8033 static uchar shiftcode[256] =
8034 {
8035   [0x1D] CTL,
8036   [0x2A] SHIFT,
8037   [0x36] SHIFT,
8038   [0x38] ALT,
8039   [0x9D] CTL,
8040   [0xB8] ALT
8041 };
8042
8043 static uchar togglecode[256] =
8044 {
8045   [0x3A] CAPSLOCK,
8046   [0x45] NUMLOCK,
8047   [0x46] SCROLLLOCK
8048 };
8049
```



```
8050 static uchar normalmap[256] =
8051 {
8052   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
8053   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
8054   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
8055   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
8056   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
8057   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
8058   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
8059   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
8060   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
8061   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
8062   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
8063   [0x9C] '\n',      // KP_Enter
8064   [0xB5] '/',       // KP_Div
8065   [0xC8] KEY_UP,    [0xD0] KEY_DN,
8066   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
8067   [0xCB] KEY_LF,    [0xCD] KEY_RT,
8068   [0x97] KEY_HOME,  [0xCF] KEY_END,
8069   [0xD2] KEY_INS,   [0xD3] KEY_DEL
8070 };
8071
8072 static uchar shiftmap[256] =
8073 {
8074   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
8075   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
8076   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
8077   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
8078   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
8079   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
8080   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
8081   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
8082   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
8083   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
8084   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
8085   [0x9C] '\n',      // KP_Enter
8086   [0xB5] '/',       // KP_Div
8087   [0xC8] KEY_UP,    [0xD0] KEY_DN,
8088   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
8089   [0xCB] KEY_LF,    [0xCD] KEY_RT,
8090   [0x97] KEY_HOME,  [0xCF] KEY_END,
8091   [0xD2] KEY_INS,   [0xD3] KEY_DEL
8092 };
8093
8094
8095
8096
8097
8098
8099
```

```
8100 static uchar ctlmap[256] =
8101 {
8102   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
8103   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
8104   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
8105   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
8106   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
8107   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
8108   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
8109   [0x9C] '\r',      // KP_Enter
8110   [0xB5] C('/'),    // KP_Div
8111   [0xC8] KEY_UP,    [0xD0] KEY_DN,
8112   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
8113   [0xCB] KEY_LF,    [0xCD] KEY_RT,
8114   [0x97] KEY_HOME,  [0xCF] KEY_END,
8115   [0xD2] KEY_INS,   [0xD3] KEY_DEL
8116 };
8117
8118
8119
8120
8121
8122
8123
8124
8125
8126
8127
8128
8129
8130
8131
8132
8133
8134
8135
8136
8137
8138
8139
8140
8141
8142
8143
8144
8145
8146
8147
8148
8149
```



```
8150 #include "types.h"
8151 #include "x86.h"
8152 #include "defs.h"
8153 #include "kbd.h"
8154
8155 int
8156 kbdgetc(void)
8157 {
8158   static uint shift;
8159   static uchar *charcode[4] = {
8160     normalmap, shiftmap, ctlmap, ctlmap
8161   };
8162   uint st, data, c;
8163
8164   st = inb(KBSTATP);
8165   if((st & KBS_DIB) == 0)
8166     return -1;
8167   data = inb(KBDATAP);
8168
8169   if(data == 0xE0){
8170     shift |= E0ESC;
8171     return 0;
8172   } else if(data & 0x80){
8173     // Key released
8174     data = (shift & E0ESC ? data : data & 0x7F);
8175     shift &= ~(shiftcode[data] | E0ESC);
8176     return 0;
8177   } else if(shift & E0ESC){
8178     // Last character was an E0 escape; or with 0x80
8179     data |= 0x80;
8180     shift &= ~E0ESC;
8181   }
8182
8183   shift |= shiftcode[data];
8184   shift ^= togglecode[data];
8185   c = charcode[shift & (CTL | SHIFT)][data];
8186   if(shift & CAPSLOCK){
8187     if('a' <= c && c <= 'z')
8188       c += 'A' - 'a';
8189     else if('A' <= c && c <= 'Z')
8190       c += 'a' - 'A';
8191   }
8192   return c;
8193 }
8194
8195 void
8196 kbdintr(void)
8197 {
8198   consoleintr(kbdgetc);
8199 }
```

```
8200 // Console input and output.
8201 // Input is from the keyboard or serial port.
8202 // Output is written to the screen and serial port.
8203
8204 #include "types.h"
8205 #include "defs.h"
8206 #include "param.h"
8207 #include "traps.h"
8208 #include "spinlock.h"
8209 #include "fs.h"
8210 #include "file.h"
8211 #include "memlayout.h"
8212 #include "mmu.h"
8213 #include "proc.h"
8214 #include "x86.h"
8215
8216 static void consputc(int);
8217
8218 static int panicked = 0;
8219
8220 static struct {
8221   struct spinlock lock;
8222   int locking;
8223 } cons;
8224
8225 static void
8226 printint(int xx, int base, int sign)
8227 {
8228   static char digits[] = "0123456789abcdef";
8229   char buf[16];
8230   int i;
8231   uint x;
8232
8233   if(sign && (sign = xx < 0))
8234     x = -xx;
8235   else
8236     x = xx;
8237
8238   i = 0;
8239   do{
8240     buf[i++] = digits[x % base];
8241   }while((x /= base) != 0);
8242
8243   if(sign)
8244     buf[i++] = '-';
8245
8246   while(--i >= 0)
8247     consputc(buf[i]);
8248 }
8249
```



```
8250 // Print to the console. only understands %d, %x, %p, %s.
8251 void
8252 cprintf(char *fmt, ...)
8253 {
8254   int i, c, locking;
8255   uint *argp;
8256   char *s;
8257
8258   locking = cons.locking;
8259   if(locking)
8260     acquire(&cons.lock);
8261
8262   if (fmt == 0)
8263     panic("null fmt");
8264
8265   argp = (uint*)(void*)(&fmt + 1);
8266   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
8267     if(c != '%'){
8268       consputc(c);
8269       continue;
8270     }
8271     c = fmt[++i] & 0xff;
8272     if(c == 0)
8273       break;
8274     switch(c){
8275     case 'd':
8276       printint(*argp++, 10, 1);
8277       break;
8278     case 'x':
8279     case 'p':
8280       printint(*argp++, 16, 0);
8281       break;
8282     case 's':
8283       if((s = (char*)*argp++) == 0)
8284         s = "(null)";
8285       for(; *s; s++)
8286         consputc(*s);
8287       break;
8288     case '%':
8289       consputc('%');
8290       break;
8291     default:
8292       // Print unknown % sequence to draw attention.
8293       consputc('%');
8294       consputc(c);
8295       break;
8296     }
8297   }
8298
8299
```

```
  if(locking)
    release(&cons.lock);
}

void
panic(char *s)
{
  int i;
  uint pcs[10];

  cli();
  cons.locking = 0;
  cprintf("cpu%d: panic: ", cpu->id);
  cprintf(s);
  cprintf("\n");
  getcallerpcs(&s, pcs);
  for(i=0; i<10; i++)
    cprintf(" %p", pcs[i]);
  panicked = 1; // freeze other CPU
  for(;;)
    ;
}
```



```
#define BACKSPACE 0x100
#define CRTPORT 0x3d4
static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory

static void
cgaputc(int c)
{
  int pos;

  // Cursor position: col + 80*row.
  outb(CRTPORT, 14);
  pos = inb(CRTPORT+1) << 8;
  outb(CRTPORT, 15);
  pos |= inb(CRTPORT+1);

  if(c == '\n')
    pos += 80 - pos%80;
  else if(c == BACKSPACE){
    if(pos > 0) --pos;
  } else
    crt[pos++] = (c&0xff) | 0x0700;  // black on white

  if(pos < 0 || pos > 25*80)
    panic("pos under/overflow");

  if((pos/80) >= 24){  // Scroll up.
    memmove(crt, crt+80, sizeof(crt[0])*23*80);
    pos -= 80;
    memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
  }

  outb(CRTPORT, 14);
  outb(CRTPORT+1, pos>>8);
  outb(CRTPORT, 15);
  outb(CRTPORT+1, pos);
  crt[pos] = ' ' | 0x0700;
}
```

```
8400 void
8401 consputc(int c)
8402 {
8403   if(panicked){
8404     cli();
8405     for(;;)
8406       ;
8407   }
8408
8409   if(c == BACKSPACE){
8410     uartputc('\b'); uartputc(' '); uartputc('\b');
8411   } else
8412     uartputc(c);
8413   cgaputc(c);
8414 }
8415
8416 #define INPUT_BUF 128
8417 struct {
8418   char buf[INPUT_BUF];
8419   uint r;  // Read index
8420   uint w;  // Write index
8421   uint e;  // Edit index
8422 } input;
8423
8424 #define C(x)  ((x)-'@')  // Control-x
8425
8426 void
8427 consoleintr(int (*getc)(void))
8428 {
8429   int c, doprocdump = 0;
8430
8431   acquire(&cons.lock);
8432   while((c = getc()) >= 0){
8433     switch(c){
8434     case C('P'):  // Process listing.
8435       doprocdump = 1;   // procdump() locks cons.lock indirectly; invoke lat
8436       break;
8437     case C('U'):  // Kill line.
8438       while(input.e != input.w &&
8439             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
8440         input.e--;
8441         consputc(BACKSPACE);
8442       }
8443       break;
8444     case C('H'): case '\x7f':  // Backspace
8445       if(input.e != input.w){
8446         input.e--;
8447         consputc(BACKSPACE);
8448       }
8449       break;
```



```
8450     default:
8451       if(c != 0 && input.e-input.r < INPUT_BUF){
8452         c = (c == '\r') ? '\n' : c;
8453         input.buf[input.e++ % INPUT_BUF] = c;
8454         consputc(c);
8455         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
8456           input.w = input.e;
8457           wakeup(&input.r);
8458         }
8459       }
8460       break;
8461     }
8462   }
8463   release(&cons.lock);
8464   if(doprocdump) {
8465     procdump();  // now call procdump() wo. cons.lock held
8466   }
8467 }
8468
8469 int
8470 consoleread(struct inode *ip, char *dst, int n)
8471 {
8472   uint target;
8473   int c;
8474
8475   iunlock(ip);
8476   target = n;
8477   acquire(&cons.lock);
8478   while(n > 0){
8479     while(input.r == input.w){
8480       if(proc->killed){
8481         release(&cons.lock);
8482         ilock(ip);
8483         return -1;
8484       }
8485       sleep(&input.r, &cons.lock);
8486     }
8487     c = input.buf[input.r++ % INPUT_BUF];
8488     if(c == C('D')){  // EOF
8489       if(n < target){
8490         // Save ^D for next time, to make sure
8491         // caller gets a 0-byte result.
8492         input.r--;
8493       }
8494       break;
8495     }
8496     *dst++ = c;
8497     --n;
8498     if(c == '\n')
8499       break;
```

```
8500   }
8501   release(&cons.lock);
8502   ilock(ip);
8503
8504   return target - n;
8505 }
8506
8507 int
8508 consolewrite(struct inode *ip, char *buf, int n)
8509 {
8510   int i;
8511
8512   iunlock(ip);
8513   acquire(&cons.lock);
8514   for(i = 0; i < n; i++)
8515     consputc(buf[i] & 0xff);
8516   release(&cons.lock);
8517   ilock(ip);
8518
8519   return n;
8520 }
8521
8522 void
8523 consoleinit(void)
8524 {
8525   initlock(&cons.lock, "console");
8526
8527   devsw[CONSOLE].write = consolewrite;
8528   devsw[CONSOLE].read = consoleread;
8529   cons.locking = 1;
8530
8531   picenable(IRQ_KBD);
8532   ioapicenable(IRQ_KBD, 0);
8533 }
8534
8535
8536
8537
8538
8539
8540
8541
8542
8543
8544
8545
8546
8547
8548
8549
```



```
8550 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
8551 // Only used on uniprocessors;
8552 // SMP machines use the local APIC timer.
8553
8554 #include "types.h"
8555 #include "defs.h"
8556 #include "traps.h"
8557 #include "x86.h"
8558
8559 #define IO_TIMER1       0x040           // 8253 Timer #1
8560
8561 // Frequency of all three count-down timers;
8562 // (TIMER_FREQ/freq) is the appropriate count
8563 // to generate a frequency of freq Hz.
8564
8565 #define TIMER_FREQ      1193182
8566 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
8567
8568 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
8569 #define TIMER_SEL0      0x00    // select counter 0
8570 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
8571 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
8572
8573 void
8574 timerinit(void)
8575 {
8576   // Interrupt 100 times/sec.
8577   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
8578   outb(IO_TIMER1, TIMER_DIV(100) % 256);
8579   outb(IO_TIMER1, TIMER_DIV(100) / 256);
8580   picenable(IRQ_TIMER);
8581 }
8582
8583
8584
8585
8586
8587
8588
8589
8590
8591
8592
8593
8594
8595
8596
8597
8598
8599
```

```
8600 // Intel 8250 serial port (UART).
8601
8602 #include "types.h"
8603 #include "defs.h"
8604 #include "param.h"
8605 #include "traps.h"
8606 #include "spinlock.h"
8607 #include "fs.h"
8608 #include "file.h"
8609 #include "mmu.h"
8610 #include "proc.h"
8611 #include "x86.h"
8612
8613 #define COM1    0x3f8
8614
8615 static int uart;    // is there a uart?
8616
8617 void
8618 uartinit(void)
8619 {
8620   char *p;
8621
8622   // Turn off the FIFO
8623   outb(COM1+2, 0);
8624
8625   // 9600 baud, 8 data bits, 1 stop bit, parity off.
8626   outb(COM1+3, 0x80);    // Unlock divisor
8627   outb(COM1+0, 115200/9600);
8628   outb(COM1+1, 0);
8629   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
8630   outb(COM1+4, 0);
8631   outb(COM1+1, 0x01);    // Enable receive interrupts.
8632
8633   // If status is 0xFF, no serial port.
8634   if(inb(COM1+5) == 0xFF)
8635     return;
8636   uart = 1;
8637
8638   // Acknowledge pre-existing interrupt conditions;
8639   // enable interrupts.
8640   inb(COM1+2);
8641   inb(COM1+0);
8642   picenable(IRQ_COM1);
8643   ioapicenable(IRQ_COM1, 0);
8644
8645   // Announce that we're here.
8646   for(p="xv6...\n"; *p; p++)
8647     uartputc(*p);
8648 }
8649
```



```
8650 void
8651 uartputc(int c)
8652 {
8653   int i;
8654
8655   if(!uart)
8656     return;
8657   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
8658     microdelay(10);
8659   outb(COM1+0, c);
8660 }
8661
8662 static int
8663 uartgetc(void)
8664 {
8665   if(!uart)
8666     return -1;
8667   if(!(inb(COM1+5) & 0x01))
8668     return -1;
8669   return inb(COM1+0);
8670 }
8671
8672 void
8673 uartintr(void)
8674 {
8675   consoleintr(uartgetc);
8676 }
8677
8678
8679
8680
8681
8682
8683
8684
8685
8686
8687
8688
8689
8690
8691
8692
8693
8694
8695
8696
8697
8698
8699
```

```
# Initial process execs /init.

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


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

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

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

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



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

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

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

```
8800 // init: The initial user-level program
8801
8802 #include "types.h"
8803 #include "stat.h"
8804 #include "user.h"
8805 #include "fcntl.h"
8806
8807 char *argv[] = { "sh", 0 };
8808
8809 int
8810 main(void)
8811 {
8812   int pid, wpid;
8813
8814   if(open("console", O_RDWR) < 0){
8815     mknod("console", 1, 1);
8816     open("console", O_RDWR);
8817   }
8818   dup(0);  // stdout
8819   dup(0);  // stderr
8820
8821   for(;;){
8822     printf(1, "init: starting sh\n");
8823     pid = fork();
8824     if(pid < 0){
8825       printf(1, "init: fork failed\n");
8826       exit();
8827     }
8828     if(pid == 0){
8829       exec("sh", argv);
8830       printf(1, "init: exec sh failed\n");
8831       exit();
8832     }
8833     while((wpid=wait()) >= 0 && wpid != pid)
8834       printf(1, "zombie!\n");
8835   }
8836 }
8837
8838
8839
8840
8841
8842
8843
8844
8845
8846
8847
8848
8849
```



```
8850 // Shell.
8851 // 2015-12-21. Added very simple processing for builtin commands
8852
8853 #include "types.h"
8854 #include "user.h"
8855 #include "fcntl.h"
8856
8857 // Parsed command representation
8858 #define EXEC  1
8859 #define REDIR 2
8860 #define PIPE  3
8861 #define LIST  4
8862 #define BACK  5
8863
8864 #define MAXARGS 10
8865
8866 struct cmd {
8867   int type;
8868 };
8869
8870 struct execcmd {
8871   int type;
8872   char *argv[MAXARGS];
8873   char *eargv[MAXARGS];
8874 };
8875
8876 struct redircmd {
8877   int type;
8878   struct cmd *cmd;
8879   char *file;
8880   char *efile;
8881   int mode;
8882   int fd;
8883 };
8884
8885 struct pipecmd {
8886   int type;
8887   struct cmd *left;
8888   struct cmd *right;
8889 };
8890
8891 struct listcmd {
8892   int type;
8893   struct cmd *left;
8894   struct cmd *right;
8895 };
8896
8897
8898
8899
```

```
8900 struct backcmd {
8901   int type;
8902   struct cmd *cmd;
8903 };
8904
8905 int fork1(void);  // Fork but panics on failure.
8906 void panic(char*);
8907 struct cmd *parsecmd(char*);
8908
8909 // Execute cmd.  Never returns.
8910 void
8911 runcmd(struct cmd *cmd)
8912 {
8913   int p[2];
8914   struct backcmd *bcmd;
8915   struct execcmd *ecmd;
8916   struct listcmd *lcmd;
8917   struct pipecmd *pcmd;
8918   struct redircmd *rcmd;
8919
8920   if(cmd == 0)
8921     exit();
8922
8923   switch(cmd->type){
8924   default:
8925     panic("runcmd");
8926
8927   case EXEC:
8928     ecmd = (struct execcmd*)cmd;
8929     if(ecmd->argv[0] == 0)
8930       exit();
8931     exec(ecmd->argv[0], ecmd->argv);
8932     printf(2, "exec %s failed\n", ecmd->argv[0]);
8933     break;
8934
8935   case REDIR:
8936     rcmd = (struct redircmd*)cmd;
8937     close(rcmd->fd);
8938     if(open(rcmd->file, rcmd->mode) < 0){
8939       printf(2, "open %s failed\n", rcmd->file);
8940       exit();
8941     }
8942     runcmd(rcmd->cmd);
8943     break;
8944
8945   case LIST:
8946     lcmd = (struct listcmd*)cmd;
8947     if(fork1() == 0)
8948       runcmd(lcmd->left);
8949     wait();
```



```
8950     runcmd(lcmd->right);
8951     break;
8952
8953   case PIPE:
8954     pcmd = (struct pipecmd*)cmd;
8955     if(pipe(p) < 0)
8956       panic("pipe");
8957     if(fork1() == 0){
8958       close(1);
8959       dup(p[1]);
8960       close(p[0]);
8961       close(p[1]);
8962       runcmd(pcmd->left);
8963     }
8964     if(fork1() == 0){
8965       close(0);
8966       dup(p[0]);
8967       close(p[0]);
8968       close(p[1]);
8969       runcmd(pcmd->right);
8970     }
8971     close(p[0]);
8972     close(p[1]);
8973     wait();
8974     wait();
8975     break;
8976
8977   case BACK:
8978     bcmd = (struct backcmd*)cmd;
8979     if(fork1() == 0)
8980       runcmd(bcmd->cmd);
8981     break;
8982   }
8983   exit();
8984 }
8985
8986 int
8987 getcmd(char *buf, int nbuf)
8988 {
8989   printf(2, "$ ");
8990   memset(buf, 0, nbuf);
8991   gets(buf, nbuf);
8992   if(buf[0] == 0) // EOF
8993     return -1;
8994   return 0;
8995 }
8996
8997
8998
8999
```

```
9000 // ***** processing for shell builtins begins here *****
9001
9002 int
9003 strncmp(const char *p, const char *q, uint n)
9004 {
9005     while(n > 0 && *p && *p == *q)
9006       n--, p++, q++;
9007     if(n == 0)
9008       return 0;
9009     return (uchar)*p - (uchar)*q;
9010 }
9011
9012 int
9013 makeint(char *p)
9014 {
9015   int val = 0;
9016
9017   while ((*p >= '0') && (*p <= '9')) {
9018     val = 10*val + (*p-'0');
9019     ++p;
9020   }
9021   return val;
9022 }
9023
9024 int
9025 setbuiltin(char *p)
9026 {
9027   int i;
9028
9029   p += strlen("_set");
9030   while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
9031   if (strncmp("uid", p, 3) == 0) {
9032     p += strlen("uid");
9033     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
9034     i = makeint(p); // ugly
9035     return (setuid(i));
9036   } else
9037   if (strncmp("gid", p, 3) == 0) {
9038     p += strlen("gid");
9039     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
9040     i = makeint(p); // ugly
9041     return (setgid(i));
9042   }
9043   printf(2, "Invalid _set parameter\n");
9044   return -1;
9045 }
9046
9047
9048
9049
```



```
9050 int
9051 getbuiltin(char *p)
9052 {
9053   p += strlen("_get");
9054   while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
9055   if (strncmp("uid", p, 3) == 0) {
9056     printf(2, "%d\n", getuid());
9057     return 0;
9058   }
9059   if (strncmp("gid", p, 3) == 0) {
9060     printf(2, "%d\n", getgid());
9061     return 0;
9062   }
9063   printf(2, "Invalid _get parameter\n");
9064   return -1;
9065 }
9066
9067 typedef int funcPtr_t(char *);
9068 typedef struct {
9069   char       *cmd;
9070   funcPtr_t  *name;
9071 } dispatchTableEntry_t;
9072
9073 // Use a simple function dispatch table (FDT) to process builtin commands
9074 dispatchTableEntry_t fdt[] = {
9075   {"_set", setbuiltin},
9076   {"_get", getbuiltin}
9077 };
9078 int FDTcount = sizeof(fdt) / sizeof(fdt[0]); // # entris in FDT
9079
9080 void
9081 dobuiltin(char *cmd) {
9082   int i;
9083
9084   for (i=0; i<FDTcount; i++)
9085     if (strncmp(cmd, fdt[i].cmd, strlen(fdt[i].cmd)) == 0)
9086      (*fdt[i].name)(cmd);
9087 }
9088
9089
9090
9091
9092
9093
9094
9095
9096
9097
9098
9099
```

```
9100 // ***** processing for shell builtins ends here *****
9101
9102 int
9103 main(void)
9104 {
9105   static char buf[100];
9106   int fd;
9107
9108   // Assumes three file descriptors open.
9109   while((fd = open("console", O_RDWR)) >= 0){
9110     if(fd >= 3){
9111       close(fd);
9112       break;
9113     }
9114   }
9115
9116   // Read and run input commands.
9117   while(getcmd(buf, sizeof(buf)) >= 0){
9118 // add support for built-ins here. cd is a built-in
9119     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
9120       // Clumsy but will have to do for now.
9121       // Chdir has no effect on the parent if run in the child.
9122       buf[strlen(buf)-1] = 0;  // chop \n
9123       if(chdir(buf+3) < 0)
9124         printf(2, "cannot cd %s\n", buf+3);
9125       continue;
9126     }
9127     if (buf[0]=='_') {     // assume it is a builtin command
9128       dobuiltin(buf);
9129       continue;
9130     }
9131     if(fork1() == 0)
9132       runcmd(parsecmd(buf));
9133     wait();
9134   }
9135   exit();
9136 }
9137
9138 void
9139 panic(char *s)
9140 {
9141   printf(2, "%s\n", s);
9142   exit();
9143 }
9144
9145
9146
9147
9148
9149
```



```
9150 int
9151 fork1(void)
9152 {
9153   int pid;
9154
9155   pid = fork();
9156   if(pid == -1)
9157     panic("fork");
9158   return pid;
9159 }
9160
9161
9162
9163
9164
9165
9166
9167
9168
9169
9170
9171
9172
9173
9174
9175
9176
9177
9178
9179
9180
9181
9182
9183
9184
9185
9186
9187
9188
9189
9190
9191
9192
9193
9194
9195
9196
9197
9198
9199
```

```
9200 // Constructors
9201
9202 struct cmd*
9203 execcmd(void)
9204 {
9205   struct execcmd *cmd;
9206
9207   cmd = malloc(sizeof(*cmd));
9208   memset(cmd, 0, sizeof(*cmd));
9209   cmd->type = EXEC;
9210   return (struct cmd*)cmd;
9211 }
9212
9213 struct cmd*
9214 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
9215 {
9216   struct redircmd *cmd;
9217
9218   cmd = malloc(sizeof(*cmd));
9219   memset(cmd, 0, sizeof(*cmd));
9220   cmd->type = REDIR;
9221   cmd->cmd = subcmd;
9222   cmd->file = file;
9223   cmd->efile = efile;
9224   cmd->mode = mode;
9225   cmd->fd = fd;
9226   return (struct cmd*)cmd;
9227 }
9228
9229 struct cmd*
9230 pipecmd(struct cmd *left, struct cmd *right)
9231 {
9232   struct pipecmd *cmd;
9233
9234   cmd = malloc(sizeof(*cmd));
9235   memset(cmd, 0, sizeof(*cmd));
9236   cmd->type = PIPE;
9237   cmd->left = left;
9238   cmd->right = right;
9239   return (struct cmd*)cmd;
9240 }
9241
9242
9243
9244
9245
9246
9247
9248
9249
```



```
9250 struct cmd*
9251 listcmd(struct cmd *left, struct cmd *right)
9252 {
9253   struct listcmd *cmd;
9254
9255   cmd = malloc(sizeof(*cmd));
9256   memset(cmd, 0, sizeof(*cmd));
9257   cmd->type = LIST;
9258   cmd->left = left;
9259   cmd->right = right;
9260   return (struct cmd*)cmd;
9261 }
9262
9263 struct cmd*
9264 backcmd(struct cmd *subcmd)
9265 {
9266   struct backcmd *cmd;
9267
9268   cmd = malloc(sizeof(*cmd));
9269   memset(cmd, 0, sizeof(*cmd));
9270   cmd->type = BACK;
9271   cmd->cmd = subcmd;
9272   return (struct cmd*)cmd;
9273 }
9274
9275
9276
9277
9278
9279
9280
9281
9282
9283
9284
9285
9286
9287
9288
9289
9290
9291
9292
9293
9294
9295
9296
9297
9298
9299
```

```
9300 // Parsing
9301
9302 char whitespace[] = " \t\r\n\v";
9303 char symbols[] = "<|>&;()";
9304
9305 int
9306 gettoken(char **ps, char *es, char **q, char **eq)
9307 {
9308   char *s;
9309   int ret;
9310
9311   s = *ps;
9312   while(s < es && strchr(whitespace, *s))
9313     s++;
9314   if(q)
9315     *q = s;
9316   ret = *s;
9317   switch(*s){
9318   case 0:
9319     break;
9320   case '|':
9321   case '(':
9322   case ')':
9323   case ';':
9324   case '&':
9325   case '<':
9326     s++;
9327     break;
9328   case '>':
9329     s++;
9330     if(*s == '>'){
9331       ret = '+';
9332       s++;
9333     }
9334     break;
9335   default:
9336     ret = 'a';
9337     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
9338       s++;
9339     break;
9340   }
9341   if(eq)
9342     *eq = s;
9343
9344   while(s < es && strchr(whitespace, *s))
9345     s++;
9346   *ps = s;
9347   return ret;
9348 }
9349
```



```
9350 int
9351 peek(char **ps, char *es, char *toks)
9352 {
9353   char *s;
9354
9355   s = *ps;
9356   while(s < es && strchr(whitespace, *s))
9357     s++;
9358   *ps = s;
9359   return *s && strchr(toks, *s);
9360 }
9361
9362 struct cmd *parseline(char**, char*);
9363 struct cmd *parsepipe(char**, char*);
9364 struct cmd *parseexec(char**, char*);
9365 struct cmd *nulterminate(struct cmd*);
9366
9367 struct cmd*
9368 parsecmd(char *s)
9369 {
9370   char *es;
9371   struct cmd *cmd;
9372
9373   es = s + strlen(s);
9374   cmd = parseline(&s, es);
9375   peek(&s, es, "");
9376   if(s != es){
9377     printf(2, "leftovers: %s\n", s);
9378     panic("syntax");
9379   }
9380   nulterminate(cmd);
9381   return cmd;
9382 }
9383
9384 struct cmd*
9385 parseline(char **ps, char *es)
9386 {
9387   struct cmd *cmd;
9388
9389   cmd = parsepipe(ps, es);
9390   while(peek(ps, es, "&")){
9391     gettoken(ps, es, 0, 0);
9392     cmd = backcmd(cmd);
9393   }
9394   if(peek(ps, es, ";")){
9395     gettoken(ps, es, 0, 0);
9396     cmd = listcmd(cmd, parseline(ps, es));
9397   }
9398   return cmd;
9399 }
```

```
9400 struct cmd*
9401 parsepipe(char **ps, char *es)
9402 {
9403   struct cmd *cmd;
9404
9405   cmd = parseexec(ps, es);
9406   if(peek(ps, es, "|")){
9407     gettoken(ps, es, 0, 0);
9408     cmd = pipecmd(cmd, parsepipe(ps, es));
9409   }
9410   return cmd;
9411 }
9412
9413 struct cmd*
9414 parseredirs(struct cmd *cmd, char **ps, char *es)
9415 {
9416   int tok;
9417   char *q, *eq;
9418
9419   while(peek(ps, es, "<>")){
9420     tok = gettoken(ps, es, 0, 0);
9421     if(gettoken(ps, es, &q, &eq) != 'a')
9422       panic("missing file for redirection");
9423     switch(tok){
9424     case '<':
9425       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
9426       break;
9427     case '>':
9428       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9429       break;
9430     case '+':  // >>
9431       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9432       break;
9433     }
9434   }
9435   return cmd;
9436 }
9437
9438
9439
9440
9441
9442
9443
9444
9445
9446
9447
9448
9449
```



```
9450 struct cmd*
9451 parseblock(char **ps, char *es)
9452 {
9453   struct cmd *cmd;
9454
9455   if(!peek(ps, es, "("))
9456     panic("parseblock");
9457   gettoken(ps, es, 0, 0);
9458   cmd = parseline(ps, es);
9459   if(!peek(ps, es, ")"))
9460     panic("syntax - missing )");
9461   gettoken(ps, es, 0, 0);
9462   cmd = parseredirs(cmd, ps, es);
9463   return cmd;
9464 }
9465
9466 struct cmd*
9467 parseexec(char **ps, char *es)
9468 {
9469   char *q, *eq;
9470   int tok, argc;
9471   struct execcmd *cmd;
9472   struct cmd *ret;
9473
9474   if(peek(ps, es, "("))
9475     return parseblock(ps, es);
9476
9477   ret = execcmd();
9478   cmd = (struct execcmd*)ret;
9479
9480   argc = 0;
9481   ret = parseredirs(ret, ps, es);
9482   while(!peek(ps, es, "|)&;")){
9483     if((tok=gettoken(ps, es, &q, &eq)) == 0)
9484       break;
9485     if(tok != 'a')
9486       panic("syntax");
9487     cmd->argv[argc] = q;
9488     cmd->eargv[argc] = eq;
9489     argc++;
9490     if(argc >= MAXARGS)
9491       panic("too many args");
9492     ret = parseredirs(ret, ps, es);
9493   }
9494   cmd->argv[argc] = 0;
9495   cmd->eargv[argc] = 0;
9496   return ret;
9497 }
9498
9499
```

```
9500 // NUL-terminate all the counted strings.
9501 struct cmd*
9502 nulterminate(struct cmd *cmd)
9503 {
9504   int i;
9505   struct backcmd *bcmd;
9506   struct execcmd *ecmd;
9507   struct listcmd *lcmd;
9508   struct pipecmd *pcmd;
9509   struct redircmd *rcmd;
9510
9511   if(cmd == 0)
9512     return 0;
9513
9514   switch(cmd->type){
9515   case EXEC:
9516     ecmd = (struct execcmd*)cmd;
9517     for(i=0; ecmd->argv[i]; i++)
9518       *ecmd->eargv[i] = 0;
9519     break;
9520
9521   case REDIR:
9522     rcmd = (struct redircmd*)cmd;
9523     nulterminate(rcmd->cmd);
9524     *rcmd->efile = 0;
9525     break;
9526
9527   case PIPE:
9528     pcmd = (struct pipecmd*)cmd;
9529     nulterminate(pcmd->left);
9530     nulterminate(pcmd->right);
9531     break;
9532
9533   case LIST:
9534     lcmd = (struct listcmd*)cmd;
9535     nulterminate(lcmd->left);
9536     nulterminate(lcmd->right);
9537     break;
9538
9539   case BACK:
9540     bcmd = (struct backcmd*)cmd;
9541     nulterminate(bcmd->cmd);
9542     break;
9543   }
9544   return cmd;
9545 }
9546
9547
9548
9549
```



```
9550 #include "asm.h"
9551 #include "memlayout.h"
9552 #include "mmu.h"
9553
9554 # Start the first CPU: switch to 32-bit protected mode, jump into C.
9555 # The BIOS loads this code from the first sector of the hard disk into
9556 # memory at physical address 0x7c00 and starts executing in real mode
9557 # with %cs=0 %ip=7c00.
9558
9559 .code16                       # Assemble for 16-bit mode
9560 .globl start
9561 start:
9562   cli                         # BIOS enabled interrupts; disable
9563
9564   # Zero data segment registers DS, ES, and SS.
9565   xorw    %ax,%ax             # Set %ax to zero
9566   movw    %ax,%ds             # -> Data Segment
9567   movw    %ax,%es             # -> Extra Segment
9568   movw    %ax,%ss             # -> Stack Segment
9569
9570   # Physical address line A20 is tied to zero so that the first PCs
9571   # with 2 MB would run software that assumed 1 MB.  Undo that.
9572 seta20.1:
9573   inb     $0x64,%al               # Wait for not busy
9574   testb   $0x2,%al
9575   jnz     seta20.1
9576
9577   movb    $0xd1,%al               # 0xd1 -> port 0x64
9578   outb    %al,$0x64
9579
9580 seta20.2:
9581   inb     $0x64,%al               # Wait for not busy
9582   testb   $0x2,%al
9583   jnz     seta20.2
9584
9585   movb    $0xdf,%al               # 0xdf -> port 0x60
9586   outb    %al,$0x60
9587
9588   # Switch from real to protected mode.  Use a bootstrap GDT that makes
9589   # virtual addresses map directly to physical addresses so that the
9590   # effective memory map doesn't change during the transition.
9591   lgdt    gdtdesc
9592   movl    %cr0, %eax
9593   orl     $CR0_PE, %eax
9594   movl    %eax, %cr0
9595
9596
9597
9598
9599
```

```
9600   # Complete transition to 32-bit protected mode by using long jmp
9601   # to reload %cs and %eip.  The segment descriptors are set up with no
9602   # translation, so that the mapping is still the identity mapping.
9603   ljmp    $(SEG_KCODE<<3), $start32
9604
9605 .code32  # Tell assembler to generate 32-bit code now.
9606 start32:
9607   # Set up the protected-mode data segment registers
9608   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
9609   movw    %ax, %ds                # -> DS: Data Segment
9610   movw    %ax, %es                # -> ES: Extra Segment
9611   movw    %ax, %ss                # -> SS: Stack Segment
9612   movw    $0, %ax                 # Zero segments not ready for use
9613   movw    %ax, %fs                # -> FS
9614   movw    %ax, %gs                # -> GS
9615
9616   # Set up the stack pointer and call into C.
9617   movl    $start, %esp
9618   call    bootmain
9619
9620   # If bootmain returns (it shouldn't), trigger a Bochs
9621   # breakpoint if running under Bochs, then loop.
9622   movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
9623   movw    %ax, %dx
9624   outw    %ax, %dx
9625   movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
9626   outw    %ax, %dx
9627 spin:
9628   jmp     spin
9629
9630 # Bootstrap GDT
9631 .p2align 2                                # force 4 byte alignment
9632 gdt:
9633   SEG_NULLASM                             # null seg
9634   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
9635   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
9636
9637 gdtdesc:
9638   .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
9639   .long   gdt                             # address gdt
9640
9641
9642
9643
9644
9645
9646
9647
9648
9649
```



```
9650 // Boot loader.
9651 //
9652 // Part of the boot block, along with bootasm.S, which calls bootmain().
9653 // bootasm.S has put the processor into protected 32-bit mode.
9654 // bootmain() loads an ELF kernel image from the disk starting at
9655 // sector 1 and then jumps to the kernel entry routine.
9656
9657 #include "types.h"
9658 #include "elf.h"
9659 #include "x86.h"
9660 #include "memlayout.h"
9661
9662 #define SECTSIZE  512
9663
9664 void readseg(uchar*, uint, uint);
9665
9666 void
9667 bootmain(void)
9668 {
9669   struct elfhdr *elf;
9670   struct proghdr *ph, *eph;
9671   void (*entry)(void);
9672   uchar* pa;
9673
9674   elf = (struct elfhdr*)0x10000;  // scratch space
9675
9676   // Read 1st page off disk
9677   readseg((uchar*)elf, 4096, 0);
9678
9679   // Is this an ELF executable?
9680   if(elf->magic != ELF_MAGIC)
9681     return;  // let bootasm.S handle error
9682
9683   // Load each program segment (ignores ph flags).
9684   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
9685   eph = ph + elf->phnum;
9686   for(; ph < eph; ph++){
9687     pa = (uchar*)ph->paddr;
9688     readseg(pa, ph->filesz, ph->off);
9689     if(ph->memsz > ph->filesz)
9690       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
9691   }
9692
9693   // Call the entry point from the ELF header.
9694   // Does not return!
9695   entry = (void(*)(void))(elf->entry);
9696   entry();
9697 }
9698
9699
```

```
9700 void
9701 waitdisk(void)
9702 {
9703   // Wait for disk ready.
9704   while((inb(0x1F7) & 0xC0) != 0x40)
9705     ;
9706 }
9707
9708 // Read a single sector at offset into dst.
9709 void
9710 readsect(void *dst, uint offset)
9711 {
9712   // Issue command.
9713   waitdisk();
9714   outb(0x1F2, 1);   // count = 1
9715   outb(0x1F3, offset);
9716   outb(0x1F4, offset >> 8);
9717   outb(0x1F5, offset >> 16);
9718   outb(0x1F6, (offset >> 24) | 0xE0);
9719   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
9720
9721   // Read data.
9722   waitdisk();
9723   insl(0x1F0, dst, SECTSIZE/4);
9724 }
9725
9726 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
9727 // Might copy more than asked.
9728 void
9729 readseg(uchar* pa, uint count, uint offset)
9730 {
9731   uchar* epa;
9732
9733   epa = pa + count;
9734
9735   // Round down to sector boundary.
9736   pa -= offset % SECTSIZE;
9737
9738   // Translate from bytes to sectors; kernel starts at sector 1.
9739   offset = (offset / SECTSIZE) + 1;
9740
9741   // If this is too slow, we could read lots of sectors at a time.
9742   // We'd write more to memory than asked, but it doesn't matter --
9743   // we load in increasing order.
9744   for(; pa < epa; pa += SECTSIZE, offset++)
9745     readsect(pa, offset);
9746 }
9747
9748
9749
```



```
9750 #include "types.h"
9751 #include "user.h"
9752 #include "date.h"
9753
9754 int main (int argc, char *argv[])
9755 {
9756     struct rtcdate r;
9757
9758     if(date(&r)){
9759    printf(2, "date_failed\n");
9760    exit();
9761     }
9762
9763     printf(1,"%d/%d/%d %d: %d: %d\n", r.month, r.day, r.year, r.hour, r.minut
9764     exit();
9765 }
9766
9767
9768
9769
9770
9771
9772
9773
9774
9775
9776
9777
9778
9779
9780
9781
9782
9783
9784
9785
9786
9787
9788
9789
9790
9791
9792
9793
9794
9795
9796
9797
9798
9799
```

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

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

   int min;
   int sec;

   struct rtcdate startTime;
   struct rtcdate endTime;



   if(date(&startTime)){
   printf(2, "date_failed");
   exit();
   }



   int pid = fork();

   if(pid > 0){
      wait();

   if(date(&endTime)){
       printf(2, "date2_failed");
       exit();
   }

    min = endTime.minute - startTime.minute;
    sec = endTime.second - startTime.second;

   if(sec < 0){
   sec = sec + 60;
   min = min-1;
   }

   printf(1, "%s %s %d %s %d %s\n", argv[1],"runs in", min, "minute(s)", sec,

   exit();
   }
   else if(pid == 0){
       exec(argv[1], argv+1);
       printf(1, "Error: exec returned");

       exit();
   }
   else{
```



```
       printf(2,"fork error\n");
       exit();
   }


exit();

}
```

```
9900 struct stat;
9901 struct rtcdate;
9902 struct uproc;
9903
9904 // system calls
9905 int fork(void);
9906 int exit(void) __attribute__((noreturn));
9907 int wait(void);
9908 int pipe(int*);
9909 int write(int, void*, int);
9910 int read(int, void*, int);
9911 int close(int);
9912 int kill(int);
9913 int exec(char*, char**);
9914 int open(char*, int);
9915 int mknod(char*, short, short);
9916 int unlink(char*);
9917 int fstat(int fd, struct stat*);
9918 int link(char*, char*);
9919 int mkdir(char*);
9920 int chdir(char*);
9921 int dup(int);
9922 int getpid(void);
9923 char* sbrk(int);
9924 int sleep(int);
9925 int uptime(void);
9926 int halt(void);
9927 int date(struct rtcdate*);
9928 int getuid(void);
9929 int getgid(void);
9930 int getppid(void);
9931 int setuid(int);
9932 int setgid(int);
9933 int getprocs(int, struct uproc*);
9934 // ulib.c
9935 int stat(char*, struct stat*);
9936 char* strcpy(char*, char*);
9937 void *memmove(void*, void*, int);
9938 char* strchr(const char*, char c);
9939 int strcmp(const char*, const char*);
9940 void printf(int, char*, ...);
9941 char* gets(char*, int max);
9942 uint strlen(char*);
9943 void* memset(void*, int, uint);
9944 void* malloc(uint);
9945 void free(void*);
9946 int atoi(const char*);
9947
9948
9949
```



```
9950 // halt the system.
9951 #include "types.h"
9952 #include "user.h"
9953
9954 int
9955 main(void) {
9956   halt();
9957   return 0;
9958 }
9959
9960
9961
9962
9963
9964
9965
9966
9967
9968
9969
9970
9971
9972
9973
9974
9975
9976
9977
9978
9979
9980
9981
9982
9983
9984
9985
9986
9987
9988
9989
9990
9991
9992
9993
9994
9995
9996
9997
9998
9999
```

```
10000 struct uproc {
10001     int pid;
10002     uint uid;
10003     uint gid;
10004     int ppid;
10005     uint size;
10006     char name[16];
10007     char state[20];
10008 };
10009
10010
10011
10012
10013
10014
10015
10016
10017
10018
10019
10020
10021
10022
10023
10024
10025
10026
10027
10028
10029
10030
10031
10032
10033
10034
10035
10036
10037
10038
10039
10040
10041
10042
10043
10044
10045
10046
10047
10048
10049
```



```
10050 #include "types.h"
10051 #include "user.h"
10052 #include "ps.h"
10053
10054 #define MAX_PROC 7
10055 int
10056 main(int argc, char *argv[])
10057 {
10058     int processes;
10059     int i;
10060     struct uproc prc[MAX_PROC];
10061
10062     processes = getprocs(MAX_PROC, prc);
10063 //    printf(1,"%d\n", processes);
10064
10065   if(processes == -1){
10066       printf(2, "getprocs failed\n");
10067       exit();
10068   }
10069
10070        printf(1, "\nPID PPID UID GID STATE SIZE NAME");
10071
10072        for(i=0; i<processes;i++){
10073       printf(1,"\n%d  %d  %d  %d  %s  %d  %s\n ", prc[i].pid, prc[i].ppid, |
10074
10075        }
10076     exit();
10077
10078 }
10079
10080
10081
10082
10083
10084
10085
10086
10087
10088
10089
10090
10091
10092
10093
10094
10095
10096
10097
10098
10099
```

```
10100 #include "param.h"
10101 #include "types.h"
10102 #include "stat.h"
10103 #include "user.h"
10104 #include "fs.h"
10105 #include "fcntl.h"
10106 #include "syscall.h"
10107 #include "traps.h"
10108 #include "memlayout.h"
10109
10110
10111 int
10112 main(int argc, char *argv[])
10113 {
10114     int uid, gid, ppid;
10115     uid = getuid();
10116     printf(1, "Current UID is: %d\n", uid);
10117     printf(1, "Setting UID to 4\n");
10118
10119     setuid(4);
10120     uid = getuid();
10121     printf(1, "Current UID is: %d\n", uid);
10122
10123     gid = getgid();
10124     printf(1, "Current GID is: %d\n", gid);
10125     printf(1, "Setting GID to 100\n");
10126
10127     setgid(100);
10128     gid = getgid();
10129     printf(1, "Current GID is: %d\n", gid);
10130
10131     ppid = getppid();
10132     printf(1, "Current PPID is: %d\n", ppid);
10133
10134
10135
10136     exit();
10137 }
10138
10139
10140
10141
10142
10143
10144
10145
10146
10147
10148
10149
```