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.

xv6-64 is a 64-bit port of MIT's xv6, by Anthony Shelton and
Jakob Eriksson, for use in UIC's Operating Systems curriculum.

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/6.828/2016/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, Anthony Shelton (x64), Jakob Eriksson (x64), and Xingbo Wu (x64).

ERROR REPORTS

If you spot errors or have suggestions for improvement, please send email to
Jakob Eriksson (jakob@uic.edu), or Frans Kaashoek and Robert Morris
(kaashoek,rtm@csail.mit.edu). If you have suggestions for improvements,
please keep in mind that the main purpose of xv6 is as a teaching operating
system. For example, we are in particular interested in simplifications and
clarifications, instead of suggestions for new systems calls, more portability,
etc.

BUILDING AND RUNNING XV6

To build xv6-64 on an x86_64 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/6.828/2016/tools.html.
Then run "make TOOLPREFIX=<your-tool-prefix>".

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 #pragma once
0101 typedef unsigned int   uint;
0102 typedef unsigned short ushort;
0103 typedef unsigned char  uchar;
0104
0105 typedef long          int64;
0106 typedef unsigned int  uint32;
0107 typedef unsigned long uint64;
0108
0109 typedef unsigned long addr_t;
0110
0111 typedef addr_t pde_t;
0112 typedef addr_t pml4e_t;
0113 typedef addr_t pdpe_t;
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
0205 // Key addresses for address space layout (see kmap in vm.c for layout)
0206 #define KERNBASE 0xFFFF800000000000 // First kernel virtual address
0207
0208 #define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked
0209
0210 #ifndef __ASSEMBLER__
0211 static inline addr_t v2p(void *a) {
0212   return ((addr_t) (a)) - ((addr_t)KERNBASE);
0213 }
0214 static inline void *p2v(addr_t a) {
0215   return (void *) ((a) + ((addr_t)KERNBASE));
0216 }
0217 #endif
0218
0219 #define V2P(a) (((addr_t) (a)) - KERNBASE)
0220 #define P2V(a) (((void *) (a)) + KERNBASE)
0221
0222 #define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
0223 #define P2V_WO(x) ((x) + KERNBASE)    // same as P2V, but without casts
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 sleeplock;
0259 struct stat;
0260 struct superblock;
0261 struct trapframe;
0262
0263 //entry.S
0264 void            wrmsr(uint msr, uint64 val);
0265 void            syscall_entry(void);
0266 void            ignore_sysret(void);
0267
0268 // bio.c
0269 void            binit(void);
0270 struct buf*     bread(uint, uint);
0271 void            brelse(struct buf*);
0272 void            bwrite(struct buf*);
0273
0274 // console.c
0275 void            consoleinit(void);
0276 void            cprintf(char*, ...);
0277 void            consoleintr(int(*)(void));
0278 void            panic(char*) __attribute__((noreturn));
0279
0280 // exec.c
0281 int             exec(char*, char**);
0282
0283 // file.c
0284 struct file*    filealloc(void);
0285 void            fileclose(struct file*);
0286 struct file*    filedup(struct file*);
0287 void            fileinit(void);
0288 int             fileread(struct file*, char*, int n);
0289 int             filestat(struct file*, struct stat*);
0290 int             filewrite(struct file*, char*, int n);
0291
0292 // fs.c
0293 void            readsb(int dev, struct superblock *sb);
0294 int             dirlink(struct inode*, char*, uint);
0295 struct inode*   dirlookup(struct inode*, char*, uint*);
0296 struct inode*   ialloc(uint, short);
0297 struct inode*   idup(struct inode*);
0298 void            iinit(int dev);
0299 void            ilock(struct inode*);
```

```
0300 void            iput(struct inode*);
0301 void            iunlock(struct inode*);
0302 void            iunlockput(struct inode*);
0303 void            iupdate(struct inode*);
0304 int             namecmp(const char*, const char*);
0305 struct inode*   namei(char*);
0306 struct inode*   nameiparent(char*, char*);
0307 int             readi(struct inode*, char*, uint, uint);
0308 void            stati(struct inode*, struct stat*);
0309 int             writei(struct inode*, char*, uint, uint);
0310
0311 // ide.c
0312 void            ideinit(void);
0313 void            ideintr(void);
0314 void            iderw(struct buf*);
0315
0316 // ioapic.c
0317 void            ioapicenable(int irq, int cpu);
0318 extern uchar    ioapicid;
0319 void            ioapicinit(void);
0320
0321 // kalloc.c
0322 char*           kalloc(void);
0323 void            kfree(char*);
0324 void            kinit1(void*, void*);
0325 void            kinit2(void*, void*);
0326
0327 // kbd.c
0328 void            kbdintr(void);
0329
0330 // lapic.c
0331 void            cmostime(struct rtcdate *r);
0332 int             cpunum(void);
0333 extern volatile uint*    lapic;
0334 void            lapiceoi(void);
0335 void            lapicinit(void);
0336 void            lapicstartap(uchar, uint);
0337 void            microdelay(int);
0338
0339 // log.c
0340 void            initlog(int dev);
0341 void            log_write(struct buf*);
0342 void            begin_op();
0343 void            end_op();
0344
0345 // mp.c
0346 extern int      ismp;
0347 void            mpinit(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 void            exit(void);
0359 int             fork(void);
0360 int             growproc(int64);
0361 int             kill(int);
0362 void            pinit(void);
0363 void            procdump(void);
0364 void            scheduler(void) __attribute__((noreturn));
0365 void            sched(void);
0366 void            sleep(void*, struct spinlock*);
0367 void            userinit(void);
0368 int             wait(void);
0369 void            wakeup(void*);
0370 void            yield(void);
0371
0372 // swtch.S
0373 void            swtch(struct context**, struct context*);
0374
0375 // spinlock.c
0376 void            acquire(struct spinlock*);
0377 void            getcallerpcs(void*, addr_t*);
0378 void            getstackpcs(addr_t*, addr_t*);
0379 int             holding(struct spinlock*);
0380 void            initlock(struct spinlock*, char*);
0381 void            release(struct spinlock*);
0382 void            pushcli(void);
0383 void            popcli(void);
0384
0385 // sleeplock.c
0386 void            acquiresleep(struct sleeplock*);
0387 void            releasesleep(struct sleeplock*);
0388 int             holdingsleep(struct sleeplock*);
0389 void            initsleeplock(struct sleeplock*, char*);
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 void            syscall(struct trapframe *);
0402 void            syscallinit(void);
0403 int             argint(int, int*);
0404 int             argptr(int, char**, int);
0405 int             argstr(int, char**);
0406 int             argaddr(int, addr_t*);
0407 int             fetchaddr(addr_t, addr_t*);
0408 int             fetchstr(addr_t, char**);
0409 int             fetchint(addr_t, int*);
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            uartearlyinit(void);
0419 void            uartinit(void);
0420 void            uartintr(void);
0421 void            uartputc(int);
0422
0423 // vm.c
0424 void            seginit(void);
0425 void            kvmalloc(void);
0426 pde_t*          setupkvm(void);
0427 char*           uva2ka(pde_t*, char*);
0428 addr_t          allocuvm(pde_t*, uint64, uint64);
0429 addr_t          deallocuvm(pde_t*, uint64, uint64);
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*, addr_t, void*, uint64);
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 #ifndef __i386__ // suppress warning for bootmain
0510 struct segdesc;
0511
0512 static inline void
0513 lgdt(struct segdesc *p, int size)
0514 {
0515   volatile ushort pd[5];
0516   addr_t addr = (addr_t)p;
0517
0518   pd[0] = size-1;
0519   pd[1] = addr;
0520   pd[2] = addr >> 16;
0521   pd[3] = addr >> 32;
0522   pd[4] = addr >> 48;
0523
0524   asm volatile("lgdt (%0)" : : "r" (pd));
0525 }
0526
0527 struct gatedesc;
0528
0529 static inline void
0530 lidt(struct gatedesc *p, int size)
0531 {
0532   volatile ushort pd[5];
0533   addr_t addr = (addr_t)p;
0534
0535   pd[0] = size-1;
0536   pd[1] = addr;
0537   pd[2] = addr >> 16;
0538   pd[3] = addr >> 32;
0539   pd[4] = addr >> 48;
0540
0541   asm volatile("lidt (%0)" : : "r" (pd));
0542 }
0543 #endif // ndef __i386__
0544
0545 static inline void
0546 ltr(ushort sel)
0547 {
0548   asm volatile("ltr %0" : : "r" (sel));
0549 }
```



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

```
0600 static inline void
0601 lcr3(addr_t val)
0602 {
0603   asm volatile("mov %0,%%cr3" : : "r" (val));
0604 }
0605
0606
0607 // Layout of the trap frame built on the stack by the
0608 // hardware and by trapasm.S, and passed to trap().
0609 struct trapframe {
0610    uint64 rax;
0611    uint64 rbx;
0612    uint64 rcx;
0613    uint64 rdx;
0614    uint64 rbp;
0615    uint64 rsi;
0616    uint64 rdi;
0617    uint64 r8;
0618    uint64 r9;
0619    uint64 r10;
0620    uint64 r11;
0621    uint64 r12;
0622    uint64 r13;
0623    uint64 r14;
0624    uint64 r15;
0625
0626    uint64 trapno;
0627    uint64 err;
0628
0629    uint64 rip;
0630    uint64 cs;
0631    uint64 rflags;
0632    uint64 rsp;
0633    uint64 ss;
0634 };
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 #define CR4_PAE         0x00000020      // Physical address extensions
0741 #define CR4_OSXFSR      0x00000200      // OS supports FXSAVE and FXRSTOR
0742 #define CR4_OSXMMEXCPT  0x00000400      // OS supports SSE exceptions
0743
0744 // Model specific registers
0745 #define MSR_EFER        0xC0000080  // extended feature enable register
0746 #define MSR_STAR        0xC0000081  // stores ring 0&3's segment bases
0747 #define MSR_LSTAR       0xC0000082  // stores syscall's entry rip
0748 #define MSR_CSTAR       0xC0000083  // for compatiblity mode (not used)
0749 #define MSR_SFMASK      0xC0000084  // syscall flag mask
```



```
0750 // various segment selectors.
0751 #define SEG_KCODE    1  // kernel code
0752 #define SEG_KDATA    2  // kernel data+stack
0753 #define SEG_UCODE32  3  // user data+stack
0754 #define SEG_UDATA    4  // user data+stack
0755 #define SEG_UCODE    5  // user code
0756 #define SEG_KCPU     6  // kernel per-cpu data
0757 #define SEG_TSS      7  // this process's task state
0758 // cpu->gdt[NSEGS] holds the above segments.
0759 #define NSEGS     8
0760 #define CALL_GATE 9
0761
0762 // The CS values for user and kernel space
0763 #define USER_CS   ((SEG_UCODE<<3)|DPL_USER)
0764 #define USER_DS   ((SEG_UDATA<<3)|DPL_USER)
0765 #define USER32_CS ((SEG_UCODE32<<3)|DPL_USER)
0766 #define KERNEL_CS (SEG_KCODE<<3)
0767
0768 #ifndef __ASSEMBLER__
0769 // Segment Descriptor
0770 struct segdesc {
0771   uint lim_15_0 : 16;  // Low bits of segment limit
0772   uint base_15_0 : 16; // Low bits of segment base address
0773   uint base_23_16 : 8; // Middle bits of segment base address
0774   uint type : 4;       // Segment type (see STS_ constants)
0775   uint s : 1;          // 0 = system, 1 = application
0776   uint dpl : 2;        // Descriptor Privilege Level
0777   uint p : 1;          // Present
0778   uint lim_19_16 : 4;  // High bits of segment limit
0779   uint avl : 1;        // Unused (available for software use)
0780   uint rsv1 : 1;       // 64-bit segment
0781   uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
0782   uint g : 1;          // Granularity: limit scaled by 4K when set
0783   uint base_31_24 : 8; // High bits of segment base address
0784 };
0785
0786 // Normal segment
0787 #define SEG(type, lim, base, sys, dpl, rsv) (struct segdesc) \
0788 { (addr_t)(lim) & 0xffff, (uint)(base) & 0xffff,             \
0789   ((addr_t)(base) >> 16) & 0xff, type, sys, dpl, 1,          \
0790   (addr_t)(lim) >> 60, 0, rsv, 0, 1, (addr_t)(base) >> 24 }
0791
0792 #define SEG16(type, base, lim, dpl) (struct segdesc)  \
0793 { (lim) & 0xffff, (uint)(base) & 0xffff,              \
0794   ((addr_t)(base) >> 16) & 0xff, type, 1, dpl, 1,     \
0795   (addr_t)(lim) >> 16, 0, 0, 1, 0, (addr_t)(base) >> 24 }
0796 #endif
0797
0798
0799
```

```
0800 #define DPL_USER    0x3     // User DPL
0801 #define APP_SEG     0x1
0802
0803 // Application segment type bits
0804 #define STA_X       0x8     // Executable segment
0805 #define STA_E       0x4     // Expand down (non-executable segments)
0806 #define STA_C       0x4     // Conforming code segment (executable only)
0807 #define STA_W       0x2     // Writeable (non-executable segments)
0808 #define STA_R       0x2     // Readable (executable segments)
0809 #define STA_A       0x1     // Accessed
0810
0811 // System segment type bits
0812 #define STS_T16A    0x1     // Available 16-bit TSS
0813 #define STS_LDT     0x2     // Local Descriptor Table
0814 #define STS_T16B    0x3     // Busy 16-bit TSS
0815 #define STS_CG16    0x4     // 16-bit Call Gate
0816 #define STS_TG      0x5     // Task Gate / Coum Transmitions
0817 #define STS_IG16    0x6     // 16-bit Interrupt Gate
0818 #define STS_TG16    0x7     // 16-bit Trap Gate
0819 #define STS_T64A    0x9     // Available 64-bit TSS
0820 #define STS_T64B    0xB     // Busy 64-bit TSS
0821 #define STS_CG64    0xC     // 64-bit Call Gate
0822 #define STS_IG64    0xE     // 64-bit Interrupt Gate
0823 #define STS_TG64    0xF     // 64-bit Trap Gate
0824
0825
0826
0827
0828
0829
0830
0831
0832
0833
0834
0835
0836
0837
0838
0839
0840
0841
0842
0843
0844
0845
0846
0847
0848
0849
```



```
0850 // A virtual address 'la' has a six-part structure as follows:
0851 //
0852 // +--16--+---9---+------9-------+-----9----+----9-------+----12-------+
0853 // | Sign | PML4  |Page Directory| Page Dir |Page Table  | Offset Page |
0854 // |Extend| Index | Pointer Index|  Index   |  Index     | in Page     |
0855 // +------+-------+--------------+----------+------------+-------------+
0856 //       \-PMX(va)-/\-PDPX(va)--/ \-PDX(va)-/ \-PTX(va)-/
0857
0858 // page map level 4 index
0859 #define PMX(va)         (((addr_t)(va) >> PML4XSHIFT) & PXMASK)
0860 // page directory pointer index
0861 #define PDPX(va)        (((addr_t)(va) >> PDPXSHIFT) & PXMASK)
0862 // page directory index
0863 #define PDX(va)         (((addr_t)(va) >> PDXSHIFT) & PXMASK)
0864 // page table index
0865 #define PTX(va)         (((addr_t)(va) >> PTXSHIFT) & PXMASK)
0866
0867 // Page directory and page table constants.
0868 #define NPDENTRIES      512     // # directory entries per page directory
0869 #define NPTENTRIES      512     // # PTEs per page table
0870 #define PGSIZE          4096    // bytes mapped by a page
0871 #define PGSHIFT         12      // log2(PGSIZE)
0872 #define PTXSHIFT        12      // offset of PTX in a linear address
0873 #define PDXSHIFT        21      // offset of PDX in a linear address
0874 #define PDPXSHIFT       30      // offset of PDPX in a linear address
0875 #define PML4XSHIFT      39      // offset of PML4X in a linear address
0876 #define PXMASK          0X1FF
0877
0878 #define PGROUNDUP(sz)  (((sz)+((addr_t)PGSIZE-1)) & ~((addr_t)(PGSIZE-1)))
0879 #define PGROUNDDOWN(a) (((a)) & ~((addr_t)(PGSIZE-1)))
0880
0881 // Page table/directory entry flags.
0882 #define PTE_P           0x001   // Present
0883 #define PTE_W           0x002   // Writeable
0884 #define PTE_U           0x004   // User
0885 #define PTE_PWT         0x008   // Write-Through
0886 #define PTE_PCD         0x010   // Cache-Disable
0887 #define PTE_A           0x020   // Accessed
0888 #define PTE_D           0x040   // Dirty
0889 #define PTE_PS          0x080   // Page Size
0890 #define PTE_MBZ         0x180   // Bits must be zero
0891
0892 // Address in page table or page directory entry
0893 #define PTE_ADDR(pte)   ((addr_t)(pte) & ~0xFFF)
0894 #define PTE_FLAGS(pte)  ((addr_t)(pte) &  0xFFF)
0895 #ifndef __ASSEMBLER__
0896 typedef addr_t pte_t;
0897 #endif
0898
0899 #define TRAP_GATE 0x100 // trap gate if one, interrupt gate if zero
```

```
0900 // Format of an ELF executable file
0901
0902 #define ELF_MAGIC 0x464C457FU  // "\x7FELF" in little endian
0903
0904 // File header
0905 struct elfhdr {
0906   uint magic;  // must equal ELF_MAGIC
0907   uchar elf[12];
0908   ushort type;
0909   ushort machine;
0910   uint version;
0911   addr_t entry;
0912   addr_t phoff;
0913   addr_t shoff;
0914   uint flags;
0915   ushort ehsize;
0916   ushort phentsize;
0917   ushort phnum;
0918   ushort shentsize;
0919   ushort shnum;
0920   ushort shstrndx;
0921 };
0922
0923 // Program section header
0924 struct proghdr {
0925   uint32 type;
0926   uint32 flags;
0927   uint64 off;
0928   uint64 vaddr;
0929   uint64 paddr;
0930   uint64 filesz;
0931   uint64 memsz;
0932   uint64 align;
0933 };
0934
0935 // Values for Proghdr type
0936 #define ELF_PROG_LOAD           1
0937
0938 // Flag bits for Proghdr flags
0939 #define ELF_PROG_FLAG_EXEC      1
0940 #define ELF_PROG_FLAG_WRITE     2
0941 #define ELF_PROG_FLAG_READ      4
0942
0943
0944
0945
0946
0947
0948
0949
```



```
0950 #include "asm.h"
0951 #include "memlayout.h"
0952 #include "mmu.h"
0953
0954 # Start the first CPU: switch to 32-bit protected mode, jump into C.
0955 # The BIOS loads this code from the first sector of the hard disk into
0956 # memory at physical address 0x7c00 and starts executing in real mode
0957 # with %cs=0 %ip=7c00.
0958
0959 .code16                       # Assemble for 16-bit mode
0960 .global start
0961 start:
0962   cli                         # BIOS enabled interrupts; disable
0963
0964   # Zero data segment registers DS, ES, and SS.
0965   xorw    %ax, %ax            # Set %ax to zero
0966   movw    %ax, %ds            # -> Data Segment
0967   movw    %ax, %es            # -> Extra Segment
0968   movw    %ax, %ss            # -> Stack Segment
0969
0970   # Physical address line A20 is tied to zero so that the first PCs
0971   # with 2 MB would run software that assumed 1 MB.  Undo that.
0972 seta20.1:
0973   inb     $0x64, %al          # Wait for not busy
0974   testb   $0x2, %al
0975   jnz     seta20.1
0976
0977   movb    $0xd1, %al          # 0xd1 -> port 0x64
0978   outb    %al, $0x64
0979
0980 seta20.2:
0981   inb     $0x64, %al          # Wait for not busy
0982   testb   $0x2, %al
0983   jnz     seta20.2
0984
0985   movb    $0xdf, %al          # 0xdf -> port 0x60
0986   outb    %al, $0x60
0987
0988   # Switch from real to protected mode. Use a bootstrap GDT that makes
0989   # virtual addresses map directly to physical addresses so that the
0990   # effective memory map doesn't change during the transition.
0991   lgdt    gdtdesc
0992   movl    %cr0, %eax
0993   orl     $CR0_PE, %eax
0994   movl    %eax, %cr0
0995
0996   # Complete the transition to 32-bit protected mode by using a long jmp
0997   # to reload %cs and %eip. The segment descriptors are set up with no
0998   # translation, so that the mapping is still the identity mapping.
0999   ljmp    $(SEG_KCODE<<3), $start32
```

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

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

  # If bootmain returns (it shouldn't), spin.
spin:
  jmp     spin

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

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



```
// Boot loader.
//
// Part of the boot sector, along with bootasm.S, which calls bootmain().
// bootasm.S has put the processor into protected 32-bit mode.
// bootmain() loads a multiboot kernel image from the disk starting at
// sector 1 and then jumps to the kernel entry routine.

#include "types.h"
#include "x86.h"
#include "memlayout.h"

#define SECTSIZE  512

struct mbheader {
  uint64 magic;
  uint64 flags;
  uint64 checksum;
  uint64 header_addr;
  uint64 load_addr;
  uint64 load_end_addr;
  uint64 bss_end_addr;
  uint64 entry_addr;
};

void readseg(uchar*, uint, uint);

void
bootmain(void)
{
  struct mbheader *hdr;
  void (*entry)(void);
  uint32 *x;
  uint n;

  x = (uint32*) 0x10000; // scratch space

  // multiboot header must be in the first 8192 bytes
  readseg((uchar*)x, 8192, 0);

  for (n = 0; n < 8192/4; n++)
    if (x[n] == 0x1BADB002)
      if ((x[n] + x[n+1] + x[n+2]) == 0)
        goto found_it;

  // failure
  return;

found_it:
  hdr = (struct mbheader *) (x + n);
```

```
  if (!(hdr->flags & 0x10000))
    return; // does not have load_* fields, cannot proceed
  if (hdr->load_addr > hdr->header_addr)
    return; // invalid;
  if (hdr->load_end_addr < hdr->load_addr)
    return; // no idea how much to load

  readseg((uchar*) hdr->load_addr,
    (hdr->load_end_addr - hdr->load_addr),
    (n * 4) - (hdr->header_addr - hdr->load_addr));

  if (hdr->bss_end_addr > hdr->load_end_addr)
    stosb((void*) hdr->load_end_addr, 0,
      hdr->bss_end_addr - hdr->load_end_addr);

  entry = (void(*)(void))(hdr->entry_addr);
  entry();
}

void
waitdisk(void)
{
  while((inb(0x1F7) & 0xC0) != 0x40);
}

// Read a single sector at offset into dst.
void
readsect(void *dst, uint offset)
{
  // Issue command.
  waitdisk();
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
  outb(0x1F5, offset >> 16);
  outb(0x1F6, (offset >> 24) | 0xE0);
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
  insl(0x1F0, dst, SECTSIZE/4);
}
```



```
// Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
// Might copy more than asked.
void
readseg(uchar* pa, uint count, uint offset)
{
  uchar* epa;

  epa = pa + count;

  // Round down to sector boundary.
  pa -= offset % SECTSIZE;

  // Translate from bytes to sectors; kernel starts at sector 1.
  offset = (offset / SECTSIZE) + 1;

  // If this is too slow, we could read lots of sectors at a time.
  // We'd write more to memory than asked, but it doesn't matter --
  // we load in increasing order.
  for(; pa < epa; pa += SECTSIZE, offset++)
    readsect(pa, offset);
}
```

```
/* entry.S
 *
 * Copyright (c) 2013 Brian Swetland
 *
 * Permission is hereby granted, free of charge, to any person obtaining
 * a copy of this software and associated documentation files (the
 * "Software"), to deal in the Software without restriction, including
 * without limitation the rights to use, copy, modify, merge, publish,
 * distribute, sublicense, and/or sell copies of the Software, and to
 * permit persons to whom the Software is furnished to do so, subject to
 * the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
 * LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
 * OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
 * WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 */

#define mboot_magic 0x1badb002
#define mboot_flags 0x00010000
#include "mmu.h"

.code64
.global mboot_header
.global mboot_entry

mboot_header:
  .long mboot_magic
  .long mboot_flags
  .long (-mboot_magic -mboot_flags)        # checksum
  .long mboot_load_addr                    # header_addr
  .long mboot_load_addr
  .long mboot_load_end
  .long mboot_bss_end
  .long mboot_entry_addr

.code32
mboot_entry:
# zero 2 pages for our bootstrap page tables
  xor     %eax, %eax    # value=0
  mov     $0x1000, %edi # starting at 4096
  mov     $0x2000, %ecx # size=8192
  rep     stosb         # memset(4096, 0, 8192)
```



```
# map both virtual address 0 and KERNBASE to the same PDPT
# PML4T[0] -> 0x2000 (PDPT)
# PML4T[256] -> 0x2000 (PDPT)
  mov     $(0x2000 | PTE_P | PTE_W), %eax
  mov     %eax, 0x1000  # PML4T[0]
  mov     %eax, 0x1800  # PML4T[512]

# PDPT[0] -> 0x0 (1 GB flat map page)
  mov     $(0x0 | PTE_P | PTE_PS | PTE_W), %eax
  mov     %eax, 0x2000  # PDPT[0]

# Clear ebx for initial processor boot.
# When secondary processors boot, they'll call through
# entry32mp (from entryother), but with a nonzero ebx.
# We'll reuse these bootstrap pagetables and GDT.
  xor     %ebx, %ebx

.global entry32mp
entry32mp:
# CR3 -> 0x1000 (PML4T)
  mov     $0x1000, %eax
  mov     %eax, %cr3

  lgdt    (gdtr64 - mboot_header + mboot_load_addr)

# PAE is required for 64-bit paging: CR4.PAE=1
  mov     %cr4, %eax
  bts     $5, %eax
  mov     %eax, %cr4

# access EFER Model specific register
  mov     $MSR_EFER, %ecx
  rdmsr
  bts     $0, %eax #enable system call extentions
  bts     $8, %eax #enable long mode
  wrmsr

# enable paging
  mov     %cr0, %eax
  orl     $(CR0_PG | CR0_WP | CR0_MP), %eax
  mov     %eax, %cr0

# shift to 64bit segment
  ljmp    $8, $(entry64low - mboot_header + mboot_load_addr)

.align 16
gdtr64:
  .word gdt64_end - gdt64_begin - 1;
  .quad gdt64_begin - mboot_header + mboot_load_addr

```

```
1300 .align 16
1301 gdt64_begin:
1302   .long 0x00000000 # 0: null desc
1303   .long 0x00000000
1304   .long 0x00000000 # 1: Code, R/X, Nonconforming
1305   .long 0x00209800
1306   .long 0x00000000 # 2: Data, R/W, Expand Down
1307   .long 0x00009000
1308 gdt64_end:
1309
1310 .align 16
1311 .code64
1312 entry64low:
1313   movabs  $entry64high, %rax
1314   jmp     *%rax
1315
1316 .global _start
1317 _start:
1318 entry64high:
1319
1320 # ensure data segment registers are sane
1321   xor     %rax, %rax
1322   mov     %ax, %ss
1323   mov     %ax, %ds
1324   mov     %ax, %es
1325   mov     %ax, %fs
1326   mov     %ax, %gs
1327
1328   # this would enable floating point instructions
1329   # mov     %cr4, %rax
1330   # or      $(CR4_PAE | CR4_OSXFSR | CR4_OSXMMEXCPT) , %rax
1331   # mov     %rax, %cr4
1332
1333 # check to see if we're booting a secondary core
1334   test    %ebx, %ebx
1335   jnz     entry64mp  # jump if booting a secondary code
1336 # setup initial stack
1337   movabs  $0xFFFF800000010000, %rax
1338   mov     %rax, %rsp
1339
1340 # enter main()
1341   jmp     main  # end of initial (the first) core ASM
1342
1343 .global __deadloop
1344 __deadloop:
1345 # we should never return here...
1346   jmp     .
1347
1348
1349
```



```
1350 entry64mp:
1351 # obtain kstack from data block before entryother
1352   mov     $0x7000, %rax
1353   mov     -16(%rax), %rsp
1354   jmp     mpenter  # end of secondary code ASM
1355
1356 .global wrmsr
1357 wrmsr:
1358   mov     %rdi, %rcx     # arg0 -> msrnum
1359   mov     %rsi, %rax     # val.low -> eax
1360   shr     $32, %rsi
1361   mov     %rsi, %rdx     # val.high -> edx
1362   wrmsr
1363   retq
1364
1365 .global ignore_sysret
1366 ignore_sysret: #return error code 38, meaning function unimplemented
1367   mov     $-38, %rax
1368   sysretq
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
```

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

# Each non-boot CPU ("AP") is started up in response to a STARTUP
# IPI from the boot CPU.  Section B.4.2 of the Multi-Processor
# Specification says that the AP will start in real mode with CS:IP
# set to XY00:0000, where XY is an 8-bit value sent with the
# STARTUP. Thus this code must start at a 4096-byte boundary.
#
# Because this code sets DS to zero, it must sit
# at an address in the low 2^16 bytes.
#
# Startothers (in main.c) sends the STARTUPs one at a time.
# It copies this code (start) at 0x7000.  It puts the address of
# a newly allocated per-core stack in start-4,the address of the
# place to jump to (mpenter) in start-8, and the physical address
# of entrypgdir in start-12.
#
# This code is identical to bootasm.S except:
#   - it does not need to enable A20
#   - it uses the address at start-4, start-8, and start-12

.code16
.global start
start:
  cli

  xorw    %ax,%ax
  movw    %ax,%ds
  movw    %ax,%es
  movw    %ax,%ss

  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
  movl    %eax, %cr0

  ljmpl    $(SEG_KCODE<<3), $(start32)
```



```
.code32
start32:
  movw    $(SEG_KDATA<<3), %ax
  movw    %ax, %ds
  movw    %ax, %es
  movw    %ax, %ss
  movw    $0, %ax
  movw    %ax, %fs
  movw    %ax, %gs

  # defer paging until we switch to 64bit mode
  # set ebx=1 so shared boot code knows we're booting a secondary core
  mov     $1, %ebx

  # Switch to the stack allocated by startothers()
  movl    (start-4), %esp
  # Call mpenter()
  call      *(start-8)

  movw    $0x8a00, %ax
  movw    %ax, %dx
  outw    %ax, %dx
  movw    $0x8ae0, %ax
  outw    %ax, %dx
spin:
  jmp     spin

.p2align 2
gdt:
  SEG_NULLASM
  SEG_ASM(STA_X|STA_R, 0, 0xffffffff)
  SEG_ASM(STA_W, 0, 0xffffffff)


gdtdesc:
  .word   (gdtdesc - gdt - 1)
  .long   gdt
```

```
1500 #include "types.h"
1501 #include "defs.h"
1502 #include "param.h"
1503 #include "memlayout.h"
1504 #include "mmu.h"
1505 #include "proc.h"
1506 #include "x86.h"
1507
1508 static void startothers(void);
1509 static void mpmain(void)  __attribute__((noreturn));
1510 extern pde_t *kpgdir;
1511 extern char end[]; // first address after kernel loaded from ELF file
1512
1513 // Bootstrap processor starts running C code here.
1514 // Allocate a real stack and switch to it, first
1515 // doing some setup required for memory allocator to work.
1516 int
1517 main(void)
1518 {
1519   uartearlyinit();
1520   kinit1(end, P2V(4*1024*1024)); // phys page allocator
1521   kvmalloc();      // kernel page table
1522   mpinit();        // detect other processors
1523   lapicinit();     // interrupt controller
1524   tvinit();        // trap vectors
1525   seginit();       // segment descriptors
1526   cprintf("\ncpu%d: starting xv6\n\n", cpunum());
1527   ioapicinit();    // another interrupt controller
1528   consoleinit();   // console hardware
1529   uartinit();      // serial port
1530   pinit();         // process table
1531   binit();         // buffer cache
1532   fileinit();      // file table
1533   ideinit();       // disk
1534   startothers();   // start other processors
1535   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1536   userinit();      // first user process
1537   mpmain();        // finish this processor's setup
1538 }
1539
1540 // Other CPUs jump here from entryother.S.
1541 void
1542 mpenter(void)
1543 {
1544   switchkvm();
1545   seginit();
1546   lapicinit();
1547   mpmain();
1548 }
1549
```



```
1550 // Common CPU setup code.
1551 static void
1552 mpmain(void)
1553 {
1554   cprintf("cpu%d: starting\n", cpunum());
1555   idtinit();       // load idt register
1556   syscallinit();   // syscall set up
1557   xchg(&cpu->started, 1); // tell startothers() we're up
1558   scheduler();     // start running processes
1559 }
1560
1561 void entry32mp(void);
1562
1563 // Start the non-boot (AP) processors.
1564 static void
1565 startothers(void)
1566 {
1567   extern uchar _binary_entryother_start[], _binary_entryother_size[];
1568   uchar *code;
1569   struct cpu *c;
1570   char *stack;
1571
1572   // Write entry code to unused memory at 0x7000.
1573   // The linker has placed the image of entryother.S in
1574   // _binary_entryother_start.
1575   code = P2V(0x7000);
1576   memmove(code, _binary_entryother_start,
1577           (addr_t)_binary_entryother_size);
1578
1579   for(c = cpus; c < cpus+ncpu; c++){
1580     if(c == cpus+cpunum())  // We've started already.
1581       continue;
1582
1583     // Tell entryother.S what stack to use, where to enter, and what
1584     // pgdir to use. We cannot use kpgdir yet, because the AP processor
1585     // is running in low  memory, so we use entrypgdir for the APs too.
1586     stack = kalloc();
1587     *(uint32*)(code-4) = 0x8000; // enough stack to get us to entry64mp
1588     *(uint32*)(code-8) = v2p(entry32mp);
1589     *(uint64*)(code-16) = (uint64) (stack + KSTACKSIZE);
1590
1591     lapicstartap(c->apicid, V2P(code));
1592
1593     // wait for cpu to finish mpmain()
1594     while(c->started == 0)
1595       ;
1596   }
1597 }
1598
1599
```

```
1600 /* Simple linker script for the JOS kernel.
1601    See the GNU ld 'info' manual ("info ld") to learn the syntax. */
1602
1603 /* OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386") */
1604 /* OUTPUT_FORMAT("elf64-x86-64", "elf64-x86-64", "elf64-x86-64") */
1605 OUTPUT_FORMAT("elf64-x86-64")
1606 OUTPUT_ARCH(i386:x86-64)
1607 ENTRY(_start)
1608
1609 mboot_load_addr = 0x00100000;
1610
1611 SECTIONS
1612 {
1613   /* Link the kernel at this address: "." means the current address */
1614   /* Must be equal to KERNLINK */
1615   . = 0xFFFF800000100000;
1616
1617   PROVIDE(begin = .);
1618
1619   .text : AT(mboot_load_addr) {
1620     *(.text .rela.text .stub .text.* .gnu.linkonce.t.*)
1621   }
1622
1623   PROVIDE(etext = .);      /* Define the 'etext' symbol to this value */
1624
1625   .rodata : {
1626     *(.rodata .rodata.* .gnu.linkonce.r.*)
1627   }
1628
1629   /* Adjust the address for the data segment to the next page */
1630   . = ALIGN(0x1000);
1631
1632   /* Conventionally, Unix linkers provide pseudo-symbols
1633    * etext, edata, and end, at the end of the text, data, and bss.
1634    * For the kernel mapping, we need the address at the beginning
1635    * of the data section, but that's not one of the conventional
1636    * symbols, because the convention started before there was a
1637    * read-only rodata section between text and data. */
1638   PROVIDE(data = .);
1639
1640   /* The data segment */
1641   .data : {
1642     *(.data)
1643   }
1644
1645   . = ALIGN(0x1000);
1646
1647   PROVIDE(edata = .);
1648
1649
```



```
1650   .bss : {
1651     *(.bss)
1652       *(COMMON)
1653   }
1654
1655   . = ALIGN(0x1000);
1656
1657   PROVIDE(end = .);
1658
1659   /DISCARD/ : {
1660     *(.eh_frame .rela.eh_frame .note.GNU-stack)
1661   }
1662 }
1663
1664 mboot_load_end = mboot_load_addr + (edata - begin);
1665 mboot_bss_end = mboot_load_addr + (end - begin);
1666 mboot_entry_addr = mboot_load_addr + (mboot_entry - begin);
1667
1668
1669
1670
1671
1672
1673
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 // Mutual exclusion lock.
1701 struct spinlock {
1702   uint locked;       // Is the lock held?
1703
1704   // For debugging:
1705   char *name;        // Name of lock.
1706   struct cpu *cpu;   // The cpu holding the lock.
1707   addr_t pcs[10];    // The call stack (an array of program counters)
1708                      // that locked the lock.
1709 };
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
```



```
1750 // Mutual exclusion spin locks.
1751
1752 #include "types.h"
1753 #include "defs.h"
1754 #include "param.h"
1755 #include "x86.h"
1756 #include "memlayout.h"
1757 #include "mmu.h"
1758 #include "proc.h"
1759 #include "spinlock.h"
1760
1761 void
1762 initlock(struct spinlock *lk, char *name)
1763 {
1764   lk->name = name;
1765   lk->locked = 0;
1766   lk->cpu = 0;
1767 }
1768
1769 // Acquire the lock.
1770 // Loops (spins) until the lock is acquired.
1771 // Holding a lock for a long time may cause
1772 // other CPUs to waste time spinning to acquire it.
1773 void
1774 acquire(struct spinlock *lk)
1775 {
1776   pushcli(); // disable interrupts to avoid deadlock.
1777   if(holding(lk))
1778     panic("acquire");
1779
1780   // The xchg is atomic.
1781   while(xchg(&lk->locked, 1) != 0)
1782     ;
1783
1784   // Tell the C compiler and the processor to not move loads or stores
1785   // past this point, to ensure that the critical section's memory
1786   // references happen after the lock is acquired.
1787   __sync_synchronize();
1788
1789   // Record info about lock acquisition for debugging.
1790   lk->cpu = cpu;
1791   getcallerpcs(&lk, lk->pcs);
1792 }
1793
1794
1795
1796
1797
1798
1799
```

```
1800 // Release the lock.
1801 void
1802 release(struct spinlock *lk)
1803 {
1804   if(!holding(lk))
1805     panic("release");
1806
1807   lk->pcs[0] = 0;
1808   lk->cpu = 0;
1809
1810   // Tell the C compiler and the processor to not move loads or stores
1811   // past this point, to ensure that all the stores in the critical
1812   // section are visible to other cores before the lock is released.
1813   // Both the C compiler and the hardware may re-order loads and
1814   // stores; __sync_synchronize() tells them both not to.
1815   __sync_synchronize();
1816
1817   // Release the lock, equivalent to lk->locked = 0.
1818   // This code can't use a C assignment, since it might
1819   // not be atomic. A real OS would use C atomics here.
1820   asm volatile("movl $0, %0" : "+m" (lk->locked) : );
1821
1822   popcli();
1823 }
1824
1825 // Record the current call stack in pcs[] by following the %rbp chain.
1826 void
1827 getcallerpcs(void *v, addr_t pcs[])
1828 {
1829   addr_t *rbp;
1830
1831   asm volatile("mov %%rbp, %0" : "=r" (rbp));
1832   getstackpcs(rbp, pcs);
1833 }
1834
1835 void
1836 getstackpcs(addr_t *rbp, addr_t pcs[])
1837 {
1838   int i;
1839
1840   for(i = 0; i < 10; i++){
1841     if(rbp == 0 || rbp < (addr_t*)KERNBASE || rbp == (addr_t*)0xffffffff)
1842       break;
1843     pcs[i] = rbp[1];     // saved %rip
1844     rbp = (addr_t*)rbp[0]; // saved %rbp
1845   }
1846   for(; i < 10; i++)
1847     pcs[i] = 0;
1848 }
1849
```



```
1850 // Check whether this cpu is holding the lock.
1851 int
1852 holding(struct spinlock *lock)
1853 {
1854   return lock->locked && lock->cpu == cpu;
1855 }
1856
1857 // Pushcli/popcli are like cli/sti except that they are matched:
1858 // it takes two popcli to undo two pushcli.  Also, if interrupts
1859 // are off, then pushcli, popcli leaves them off.
1860 void
1861 pushcli(void)
1862 {
1863   int eflags;
1864
1865   eflags = readeflags();
1866   cli();
1867   if(cpu->ncli == 0)
1868     cpu->intena = eflags & FL_IF;
1869   cpu->ncli += 1;
1870 }
1871
1872 void
1873 popcli(void)
1874 {
1875   if(readeflags()&FL_IF)
1876     panic("popcli - interruptible");
1877   if(--cpu->ncli < 0)
1878     panic("popcli");
1879   if(cpu->ncli == 0 && cpu->intena)
1880     sti();
1881 }
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
```

```
1900 #include "param.h"
1901 #include "types.h"
1902 #include "defs.h"
1903 #include "x86.h"
1904 #include "memlayout.h"
1905 #include "mmu.h"
1906 #include "proc.h"
1907
1908 extern char data[];  // defined by kernel.ld
1909 pde_t *kpgdir;  // for use in scheduler()
1910
1911 __thread struct cpu *cpu;         // %fs:(-16)
1912 __thread struct proc *proc;       // %fs:(-8)
1913
1914 static pde_t *kpml4;
1915 static pde_t *kpdpt;
1916
1917 void
1918 syscallinit(void)
1919 {
1920   // the MSR/SYSRET wants the segment for 32-bit user data
1921   // next up is 64-bit user data, then code
1922   // This is simply the way the sysret instruction
1923   // is designed to work (it assumes they follow).
1924   wrmsr(MSR_STAR,
1925     ((((uint64)USER32_CS) << 48) | ((uint64)KERNEL_CS << 32)));
1926   wrmsr(MSR_LSTAR, (addr_t)syscall_entry);
1927   wrmsr(MSR_CSTAR, (addr_t)ignore_sysret);
1928
1929   wrmsr(MSR_SFMASK, FL_TF|FL_DF|FL_IF|FL_IOPL_3|FL_AC|FL_NT);
1930 }
1931
1932 // Set up CPU's kernel segment descriptors.
1933 // Run once on entry on each CPU.
1934 void
1935 seginit(void)
1936 {
1937   struct segdesc *gdt;
1938   uint *tss;
1939   uint64 addr;
1940   void *local;
1941   struct cpu *c;
1942
1943   // create a page for cpu local storage
1944   local = kalloc();
1945   memset(local, 0, PGSIZE);
1946
1947   gdt = (struct segdesc*) local;
1948   tss = (uint*) (((char*) local) + 1024);
1949   tss[16] = 0x00680000; // IO Map Base = End of TSS
```



```
1950   // point FS smack in the middle of our local storage page
1951   wrmsr(0xC0000100, ((uint64) local) + 2048);
1952
1953   c = &cpus[cpunum()];
1954   c->local = local;
1955
1956   cpu = c;
1957   proc = 0;
1958
1959   addr = (uint64) tss;
1960   gdt[0] =  (struct segdesc) {};
1961
1962   gdt[SEG_KCODE] = SEG((STA_X|STA_R), 0, 0, APP_SEG, !DPL_USER, 1);
1963   gdt[SEG_KDATA] = SEG(STA_W, 0, 0, APP_SEG, !DPL_USER, 0);
1964   gdt[SEG_UCODE32] = (struct segdesc) {}; // required by syscall/sysret
1965   gdt[SEG_UDATA] = SEG(STA_W, 0, 0, APP_SEG, DPL_USER, 0);
1966   gdt[SEG_UCODE] = SEG((STA_X|STA_R), 0, 0, APP_SEG, DPL_USER, 1);
1967   gdt[SEG_KCPU]  = (struct segdesc) {};
1968   // TSS: See IA32 SDM Figure 7-4
1969   gdt[SEG_TSS]   = SEG(STS_T64A, 0xb, addr, !APP_SEG, DPL_USER, 0);
1970   gdt[SEG_TSS+1] = SEG(0, addr >> 32, addr >> 48, 0, 0, 0);
1971
1972   lgdt((void*) gdt, (NSEGS+1) * sizeof(struct segdesc));
1973
1974   ltr(SEG_TSS << 3);
1975 };
1976
1977
1978 // There is one page table per process, plus one that's used when
1979 // a CPU is not running any process (kpgdir). The kernel uses the
1980 // current process's page table during system calls and interrupts;
1981 // page protection bits prevent user code from using the kernel's
1982 // mappings.
1983 //
1984 // setupkvm() and exec() set up every page table like this:
1985 //
1986 //   0..KERNBASE: user memory (text+data+stack+heap), mapped to
1987 //                phys memory allocated by the kernel
1988 //   KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
1989 //   KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
1990 //                for the kernel's instructions and r/o data
1991 //   data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
1992 //                                     rw data + free physical memory
1993 //   0xfe000000..0: mapped direct (devices such as ioapic)
1994 //
1995 // The kernel allocates physical memory for its heap and for user memory
1996 // between V2P(end) and the end of physical memory (PHYSTOP)
1997 // (directly addressable from end..P2V(PHYSTOP)).
1998
1999
```

```
2000 pde_t*
2001 setupkvm(void)
2002 {
2003   pde_t *pml4 = (pde_t*) kalloc();
2004   memset(pml4, 0, PGSIZE);
2005   pml4[256] = v2p(kpdpt) | PTE_P | PTE_W;
2006   return pml4;
2007 };
2008
2009 // Allocate one page table for the machine for the kernel address
2010 // space for scheduler processes.
2011 //
2012 // linear map the first 4GB of physical memory starting
2013 // at 0xFFFF800000000000
2014 void
2015 kvmalloc(void)
2016 {
2017   kpml4 = (pde_t*) kalloc();
2018   memset(kpml4, 0, PGSIZE);
2019
2020   // the kernel memory region starts at KERNBASE and up
2021   // allocate one PDPT at the bottom of that range.
2022   kpdpt = (pde_t*) kalloc();
2023   memset(kpdpt, 0, PGSIZE);
2024   kpml4[PMX(KERNBASE)] = v2p(kpdpt) | PTE_P | PTE_W;
2025
2026   // direct map first GB of physical addresses to KERNBASE
2027   kpdpt[0] = 0 | PTE_PS | PTE_P | PTE_W;
2028
2029   // direct map 4th GB of physical addresses to KERNBASE+3GB
2030   // this is a very lazy way to map IO memory (for lapic and ioapic)
2031   // PTE_PWT and PTE_PCD for memory mapped I/O correctness.
2032   kpdpt[3] = 0xC0000000 | PTE_PS | PTE_P | PTE_W | PTE_PWT | PTE_PCD;
2033
2034   switchkvm();
2035 }
2036
2037 void
2038 switchuvm(struct proc *p)
2039 {
2040   pushcli();
2041   if(p->pgdir == 0)
2042     panic("switchuvm: no pgdir");
2043   uint *tss = (uint*) (((char*) cpu->local) + 1024);
2044   const addr_t stktop = (addr_t)p->kstack + KSTACKSIZE;
2045   tss[1] = (uint)stktop; // https://wiki.osdev.org/Task_State_Segment
2046   tss[2] = (uint)(stktop >> 32);
2047   lcr3(v2p(p->pgdir));
2048   popcli();
2049 }
```



```
2050 // Return the address of the PTE in page table pgdir
2051 // that corresponds to virtual address va.  If alloc!=0,
2052 // create any required page table pages.
2053 //
2054 // In 64-bit mode, the page table has four levels: PML4, PDPT, PD and PT
2055 // For each level, we dereference the correct entry, or allocate and
2056 // initialize entry if the PTE_P bit is not set
2057 static pte_t *
2058 walkpgdir(pde_t *pml4, const void *va, int alloc)
2059 {
2060   pml4e_t *pml4e;
2061   pdpe_t *pdp, *pdpe;
2062   pde_t *pde, *pd, *pgtab;
2063
2064   // from the PML4, find or allocate the appropriate PDP table
2065   pml4e = &pml4[PMX(va)];
2066   if(*pml4e & PTE_P)
2067     pdp = (pdpe_t*)P2V(PTE_ADDR(*pml4e));
2068   else {
2069     if(!alloc || (pdp = (pdpe_t*)kalloc()) == 0)
2070       return 0;
2071     memset(pdp, 0, PGSIZE);
2072     *pml4e = V2P(pdp) | PTE_P | PTE_W | PTE_U;
2073   }
2074
2075   //from the PDP, find or allocate the appropriate PD (page directory)
2076   pdpe = &pdp[PDPX(va)];
2077   if(*pdpe & PTE_P)
2078     pd = (pde_t*)P2V(PTE_ADDR(*pdpe));
2079   else {
2080     if(!alloc || (pd = (pde_t*)kalloc()) == 0)//allocate page table
2081       return 0;
2082     memset(pd, 0, PGSIZE);
2083     *pdpe = V2P(pd) | PTE_P | PTE_W | PTE_U;
2084   }
2085
2086   // from the PD, find or allocate the appropriate page table
2087   pde = &pd[PDX(va)];
2088   if(*pde & PTE_P)
2089     pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
2090   else {
2091     if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)//allocate page table
2092       return 0;
2093     memset(pgtab, 0, PGSIZE);
2094     *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
2095   }
2096
2097   return &pgtab[PTX(va)];
2098 }
2099
```

```
2100 void
2101 switchkvm(void)
2102 {
2103   lcr3(v2p(kpml4));
2104 }
2105
2106 // Create PTEs for virtual addresses starting at va that refer to
2107 // physical addresses starting at pa. va and size might not
2108 // be page-aligned.
2109 int
2110 mappages(pde_t *pgdir, void *va, addr_t size, addr_t pa, int perm)
2111 {
2112   char *a, *last;
2113   pte_t *pte;
2114
2115   a = (char*)PGROUNDDOWN((addr_t)va);
2116   last = (char*)PGROUNDDOWN(((addr_t)va) + size - 1);
2117   for(;;){
2118     if((pte = walkpgdir(pgdir, a, 1)) == 0)
2119       return -1;
2120     if(*pte & PTE_P)
2121       panic("remap");
2122     *pte = pa | perm | PTE_P;
2123     if(a == last)
2124       break;
2125     a += PGSIZE;
2126     pa += PGSIZE;
2127   }
2128   return 0;
2129 }
2130
2131 // Load the initcode into address 0x1000 (4KB) of pgdir.
2132 // sz must be less than a page.
2133 void
2134 inituvm(pde_t *pgdir, char *init, uint sz)
2135 {
2136   char *mem;
2137
2138   if(sz >= PGSIZE)
2139     panic("inituvm: more than a page");
2140
2141   mem = kalloc();
2142   memset(mem, 0, PGSIZE);
2143   mappages(pgdir, (void *)PGSIZE, PGSIZE, V2P(mem), PTE_W|PTE_U);
2144
2145   memmove(mem, init, sz);
2146 }
2147
2148
2149
```



```
2150 // Load a program segment into pgdir.  addr must be page-aligned
2151 // and the pages from addr to addr+sz must already be mapped.
2152 int
2153 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
2154 {
2155   uint i, n;
2156   addr_t pa;
2157   pte_t *pte;
2158
2159   if((addr_t) addr % PGSIZE != 0)
2160     panic("loaduvm: addr must be page aligned");
2161   for(i = 0; i < sz; i += PGSIZE){
2162     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
2163       panic("loaduvm: address should exist");
2164     pa = PTE_ADDR(*pte);
2165     if(sz - i < PGSIZE)
2166       n = sz - i;
2167     else
2168       n = PGSIZE;
2169     if(readi(ip, P2V(pa), offset+i, n) != n)
2170       return -1;
2171   }
2172   return 0;
2173 }
2174
2175 // Allocate page tables and physical memory to grow process from oldsz to
2176 // newsz, which need not be page aligned.  Returns new size or 0 on error.
2177 uint64
2178 allocuvm(pde_t *pgdir, uint64 oldsz, uint64 newsz)
2179 {
2180   char *mem;
2181   addr_t a;
2182
2183   if(newsz >= KERNBASE)
2184     return 0;
2185   if(newsz < oldsz)
2186     return oldsz;
2187
2188   a = PGROUNDUP(oldsz);
2189   for(; a < newsz; a += PGSIZE){
2190     mem = kalloc();
2191     if(mem == 0){
2192       //cprintf("allocuvm out of memory\n");
2193       deallocuvm(pgdir, newsz, oldsz);
2194       return 0;
2195     }
2196     memset(mem, 0, PGSIZE);
2197     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
2198       //cprintf("allocuvm out of memory (2)\n");
2199       deallocuvm(pgdir, newsz, oldsz);
```

```
2200       kfree(mem);
2201       return 0;
2202     }
2203   }
2204   return newsz;
2205 }
2206
2207 // Deallocate user pages to bring the process size from oldsz to
2208 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
2209 // need to be less than oldsz.  oldsz can be larger than the actual
2210 // process size.  Returns the new process size.
2211 uint64
2212 deallocuvm(pde_t *pgdir, uint64 oldsz, uint64 newsz)
2213 {
2214   pte_t *pte;
2215   addr_t a, pa;
2216
2217   if(newsz >= oldsz)
2218     return oldsz;
2219
2220   a = PGROUNDUP(newsz);
2221   for(; a  < oldsz; a += PGSIZE){
2222     pte = walkpgdir(pgdir, (char*)a, 0);
2223     if(pte && (*pte & PTE_P) != 0){
2224       pa = PTE_ADDR(*pte);
2225       if(pa == 0)
2226         panic("kfree");
2227       char *v = P2V(pa);
2228       kfree(v);
2229       *pte = 0;
2230     }
2231   }
2232   return newsz;
2233 }
2234
2235 // Free all the pages mapped by, and all the memory used for,
2236 // this page table
2237 void
2238 freevm(pde_t *pml4)
2239 {
2240   uint i, j, k, l;
2241   pde_t *pdp, *pd, *pt;
2242
2243   if(pml4 == 0)
2244     panic("freevm: no pgdir");
2245
2246   // then need to loop through pml4 entry
2247   for(i = 0; i < (NPDENTRIES/2); i++){
2248     if(pml4[i] & PTE_P){
2249       pdp = (pdpe_t*)P2V(PTE_ADDR(pml4[i]));
```



```
2250       // and every entry in the corresponding pdpt
2251       for(j = 0; j < NPDENTRIES; j++){
2252         if(pdp[j] & PTE_P){
2253           pd = (pde_t*)P2V(PTE_ADDR(pdp[j]));
2254
2255           // and every entry in the corresponding page directory
2256           for(k = 0; k < (NPDENTRIES); k++){
2257             if(pd[k] & PTE_P) {
2258               pt = (pde_t*)P2V(PTE_ADDR(pd[k]));
2259
2260               // and every entry in the corresponding page table
2261               for(l = 0; l < (NPDENTRIES); l++){
2262                 if(pt[l] & PTE_P) {
2263                   char * v = P2V(PTE_ADDR(pt[l]));
2264
2265                   kfree((char*)v);
2266                 }
2267               }
2268               //freeing every page table
2269               kfree((char*)pt);
2270             }
2271           }
2272           // freeing every page directory
2273           kfree((char*)pd);
2274         }
2275       }
2276       // freeing every page directory pointer table
2277       kfree((char*)pdp);
2278     }
2279   }
2280   // freeing the pml4
2281   kfree((char*)pml4);
2282 }
2283
2284 // Clear PTE_U on a page. Used to create an inaccessible
2285 // page beneath the user stack.
2286 void
2287 clearpteu(pde_t *pgdir, char *uva)
2288 {
2289   pte_t *pte;
2290
2291   pte = walkpgdir(pgdir, uva, 0);
2292   if(pte == 0)
2293     panic("clearpteu");
2294   *pte &= ~PTE_U;
2295 }
2296
2297
2298
2299
```

```
2300 // Given a parent process's page table, create a copy
2301 // of it for a child.
2302 pde_t*
2303 copyuvm(pde_t *pgdir, uint sz)
2304 {
2305   pde_t *d;
2306   pte_t *pte;
2307   addr_t pa, i, flags;
2308   char *mem;
2309
2310   if((d = setupkvm()) == 0)
2311     return 0;
2312   for(i = PGSIZE; i < sz; i += PGSIZE){
2313     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
2314       panic("copyuvm: pte should exist");
2315     if(!(*pte & PTE_P))
2316       panic("copyuvm: page not present");
2317     pa = PTE_ADDR(*pte);
2318     flags = PTE_FLAGS(*pte);
2319     if((mem = kalloc()) == 0)
2320       goto bad;
2321     memmove(mem, (char*)P2V(pa), PGSIZE);
2322     if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
2323       goto bad;
2324   }
2325   return d;
2326
2327 bad:
2328   freevm(d);
2329   return 0;
2330 }
2331
2332 // Map user virtual address to kernel address.
2333 char*
2334 uva2ka(pde_t *pgdir, char *uva)
2335 {
2336   pte_t *pte;
2337
2338   pte = walkpgdir(pgdir, uva, 0);
2339   if((*pte & PTE_P) == 0)
2340     return 0;
2341   if((*pte & PTE_U) == 0)
2342     return 0;
2343   return (char*)P2V(PTE_ADDR(*pte));
2344 }
2345
2346
2347
2348
2349
```



```
2350 // Copy len bytes from p to user address va in page table pgdir.
2351 // Most useful when pgdir is not the current page table.
2352 // uva2ka ensures this only works for PTE_U pages.
2353 int
2354 copyout(pde_t *pgdir, addr_t va, void *p, uint64 len)
2355 {
2356   char *buf, *pa0;
2357   addr_t n, va0;
2358
2359   buf = (char*)p;
2360   while(len > 0){
2361     va0 = PGROUNDDOWN(va);
2362     pa0 = uva2ka(pgdir, (char*)va0);
2363     if(pa0 == 0)
2364       return -1;
2365     n = PGSIZE - (va - va0);
2366     if(n > len)
2367       n = len;
2368     memmove(pa0 + (va - va0), buf, n);
2369     len -= n;
2370     buf += n;
2371     va = va0 + PGSIZE;
2372   }
2373   return 0;
2374 }
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
```

```
2400 // Per-CPU state
2401 struct cpu {
2402   uchar id;
2403   uchar apicid;              // Local APIC ID
2404   struct context *scheduler; // swtch() here to enter scheduler
2405   volatile uint started;     // Has the CPU started?
2406   int ncli;                  // Depth of pushcli nesting.
2407   int intena;                // Were interrupts enabled before pushcli?
2408   void *local;               // CPU-local storage; see seginit()
2409 };
2410
2411 extern struct cpu cpus[NCPU];
2412 extern int ncpu;
2413
2414 // Per-CPU variables, holding pointers to the
2415 // current cpu and to the current process.
2416 // The asm suffix tells gcc to use "%gs:(-16)" to refer to cpu
2417 // and "%gs:(-8)" to refer to proc.  seginit sets up the
2418 // %gs segment register so that %gs refers to the memory
2419 // holding those two variables in the local cpu's struct cpu.
2420 // This is similar to how thread-local variables are implemented
2421 // in thread libraries such as Linux pthreads.
2422 extern __thread struct cpu *cpu;
2423 extern __thread struct proc *proc;
2424
2425
2426 // Saved registers for kernel context switches.
2427 // Don't need to save all the segment registers (%cs, etc),
2428 // because they are constant across kernel contexts.
2429 // Don't need to save %rax, %rcx, etc., because the
2430 // x86 convention is that the caller has saved them.
2431 // Contexts are stored at the bottom of the stack they
2432 // describe; the stack pointer is the address of the context.
2433 // The layout of the context matches the layout of the stack in swtch.S
2434 // at the "Switch stacks" comment. Switch doesn't save rip explicitly,
2435 // but it is on the stack and allocproc() manipulates it.
2436 struct context {
2437   addr_t r15;
2438   addr_t r14;
2439   addr_t r13;
2440   addr_t r12;
2441   addr_t rbx;
2442   addr_t rbp;
2443   addr_t rip;
2444 };
2445
2446
2447
2448
2449
```



```
2450 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2451
2452 // Per-process state
2453 struct proc {
2454   addr_t sz;                     // Size of process memory (bytes)
2455   pde_t* pgdir;                  // Page table
2456   char *kstack;                  // Bottom of kernel stack for this process
2457   enum procstate state;          // Process state
2458   int pid;                       // Process ID
2459   struct proc *parent;           // Parent process
2460   struct trapframe *tf;          // Trap frame for current syscall
2461   struct context *context;       // swtch() here to run process
2462   void *chan;                    // If non-zero, sleeping on chan
2463   int killed;                    // If non-zero, have been killed
2464   struct file *ofile[NOFILE];    // Open files
2465   struct inode *cwd;             // Current directory
2466   char name[16];                 // Process name (debugging)
2467 };
2468
2469 // Process memory is laid out contiguously, low addresses first:
2470 //   text
2471 //   original data and bss
2472 //   fixed-size stack
2473 //   expandable heap
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
```

```
2500 #include "types.h"
2501 #include "defs.h"
2502 #include "param.h"
2503 #include "memlayout.h"
2504 #include "mmu.h"
2505 #include "x86.h"
2506 #include "proc.h"
2507 #include "spinlock.h"
2508
2509 struct {
2510   struct spinlock lock;
2511   struct proc proc[NPROC];
2512 } ptable;
2513
2514 static struct proc *initproc;
2515
2516 int nextpid = 1;
2517 extern void forkret(void);
2518 extern void syscall_trapret(void);
2519
2520 static void wakeup1(void *chan);
2521
2522 void
2523 pinit(void)
2524 {
2525   initlock(&ptable.lock, "ptable");
2526 }
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
```



```
2550 // Look in the process table for an UNUSED proc.
2551 // If found, change state to EMBRYO and initialize
2552 // state required to run in the kernel.
2553 // Otherwise return 0.
2554 static struct proc*
2555 allocproc(void)
2556 {
2557   struct proc *p;
2558   char *sp;
2559
2560   acquire(&ptable.lock);
2561
2562   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2563     if(p->state == UNUSED)
2564       goto found;
2565
2566   release(&ptable.lock);
2567   return 0;
2568
2569 found:
2570   p->state = EMBRYO;
2571   p->pid = nextpid++;
2572
2573   release(&ptable.lock);
2574
2575   // Allocate kernel stack.
2576   if((p->kstack = kalloc()) == 0){
2577     p->state = UNUSED;
2578     return 0;
2579   }
2580   sp = p->kstack + KSTACKSIZE;
2581
2582   // Leave room for trap frame.
2583   sp -= sizeof *p->tf;
2584   p->tf = (struct trapframe*)sp;
2585
2586   // Set up new context to start executing at forkret,
2587   // which returns to trapret.
2588   sp -= sizeof(addr_t);
2589   *(addr_t*)sp = (addr_t)syscall_trapret;
2590
2591   sp -= sizeof *p->context;
2592   p->context = (struct context*)sp;
2593   memset(p->context, 0, sizeof *p->context);
2594   p->context->rip = (addr_t)forkret;
2595
2596   return p;
2597 }
2598
2599
```

```
2600 // Set up first user process.
2601 void
2602 userinit(void)
2603 {
2604   struct proc *p;
2605   extern char _binary_initcode_start[], _binary_initcode_size[];
2606   p = allocproc();
2607
2608   initproc = p;
2609   if((p->pgdir = setupkvm()) == 0)
2610     panic("userinit: out of memory?");
2611
2612   inituvm(p->pgdir, _binary_initcode_start,
2613           (addr_t)_binary_initcode_size);
2614   p->sz = PGSIZE * 2;
2615   memset(p->tf, 0, sizeof(*p->tf));
2616
2617   p->tf->r11 = FL_IF;  // with SYSRET, EFLAGS is in R11
2618   p->tf->rsp = p->sz;
2619   p->tf->rcx = PGSIZE;  // with SYSRET, RIP is in RCX
2620
2621   safestrcpy(p->name, "initcode", sizeof(p->name));
2622   p->cwd = namei("/");
2623
2624   __sync_synchronize();
2625   p->state = RUNNABLE;
2626 }
2627
2628 // Grow current process's memory by n bytes.
2629 // Return 0 on success, -1 on failure.
2630 int
2631 growproc(int64 n)
2632 {
2633   addr_t sz;
2634
2635   sz = proc->sz;
2636   if(n > 0){
2637     if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2638       return -1;
2639   } else if(n < 0){
2640     if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2641       return -1;
2642   }
2643   proc->sz = sz;
2644   switchuvm(proc);
2645   return 0;
2646 }
2647
2648
2649
```



```
2650 // Create a new process copying p as the parent.
2651 // Sets up stack to return as if from system call.
2652 // Caller must set state of returned proc to RUNNABLE.
2653 int
2654 fork(void)
2655 {
2656   int i, pid;
2657   struct proc *np;
2658
2659   // Allocate process.
2660   if((np = allocproc()) == 0)
2661     return -1;
2662
2663   // Copy process state from p.
2664   if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2665     kfree(np->kstack);
2666     np->kstack = 0;
2667     np->state = UNUSED;
2668     return -1;
2669   }
2670   np->sz = proc->sz;
2671   np->parent = proc;
2672   *np->tf = *proc->tf;
2673
2674   // Clear %rax so that fork returns 0 in the child.
2675   np->tf->rax = 0;
2676
2677   for(i = 0; i < NOFILE; i++)
2678     if(proc->ofile[i])
2679       np->ofile[i] = filedup(proc->ofile[i]);
2680   np->cwd = idup(proc->cwd);
2681
2682   safestrcpy(np->name, proc->name, sizeof(proc->name));
2683
2684   pid = np->pid;
2685
2686   __sync_synchronize();
2687   np->state = RUNNABLE;
2688
2689   return pid;
2690 }
2691
2692
2693
2694
2695
2696
2697
2698
2699
```

```
2700 // Exit the current process.  Does not return.
2701 // An exited process remains in the zombie state
2702 // until its parent calls wait() to find out it exited.
2703 void
2704 exit(void)
2705 {
2706   struct proc *p;
2707   int fd;
2708
2709   if(proc == initproc)
2710     panic("init exiting");
2711
2712   // Close all open files.
2713   for(fd = 0; fd < NOFILE; fd++){
2714     if(proc->ofile[fd]){
2715       fileclose(proc->ofile[fd]);
2716       proc->ofile[fd] = 0;
2717     }
2718   }
2719
2720   begin_op();
2721   iput(proc->cwd);
2722   end_op();
2723   proc->cwd = 0;
2724
2725   acquire(&ptable.lock);
2726
2727   // Parent might be sleeping in wait().
2728   wakeup1(proc->parent);
2729
2730   // Pass abandoned children to init.
2731   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2732     if(p->parent == proc){
2733       p->parent = initproc;
2734       if(p->state == ZOMBIE)
2735         wakeup1(initproc);
2736     }
2737   }
2738
2739   // Jump into the scheduler, never to return.
2740   proc->state = ZOMBIE;
2741   sched();
2742   panic("zombie exit");
2743 }
2744
2745
2746
2747
2748
2749
```



```
2750 // Wait for a child process to exit and return its pid.
2751 // Return -1 if this process has no children.
2752 int
2753 wait(void)
2754 {
2755   struct proc *p;
2756   int havekids, pid;
2757
2758   acquire(&ptable.lock);
2759   for(;;){
2760     // Scan through table looking for exited children.
2761     havekids = 0;
2762     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2763       if(p->parent != proc)
2764         continue;
2765       havekids = 1;
2766       if(p->state == ZOMBIE){
2767         // Found one.
2768         pid = p->pid;
2769         kfree(p->kstack);
2770         p->kstack = 0;
2771         freevm(p->pgdir);
2772         p->pid = 0;
2773         p->parent = 0;
2774         p->name[0] = 0;
2775         p->killed = 0;
2776         p->state = UNUSED;
2777         release(&ptable.lock);
2778         return pid;
2779       }
2780     }
2781
2782     // No point waiting if we don't have any children.
2783     if(!havekids || proc->killed){
2784       release(&ptable.lock);
2785       return -1;
2786     }
2787
2788     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2789     sleep(proc, &ptable.lock);
2790   }
2791 }
2792
2793
2794
2795
2796
2797
2798
2799
```

```
2800 // Per-CPU process scheduler.
2801 // Each CPU calls scheduler() after setting itself up.
2802 // Scheduler never returns.  It loops, doing:
2803 //  - choose a process to run
2804 //  - swtch to start running that process
2805 //  - eventually that process transfers control
2806 //      via swtch back to the scheduler.
2807 void
2808 scheduler(void)
2809 {
2810   int i = 0;
2811   struct proc *p;
2812   int skipped = 0;
2813   for(;;){
2814     ++i;
2815     // Enable interrupts on this processor.
2816     sti();
2817     // Loop over process table looking for process to run.
2818     acquire(&ptable.lock);
2819     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2820       if(p->state != RUNNABLE) {
2821         skipped++;
2822         continue;
2823       }
2824       skipped = 0;
2825
2826       // Switch to chosen process.  It is the process's job
2827       // to release ptable.lock and then reacquire it
2828       // before jumping back to us.
2829       proc = p;
2830       switchuvm(p);
2831       p->state = RUNNING;
2832       swtch(&cpu->scheduler, p->context);
2833       switchkvm();
2834
2835       // Process is done running for now.
2836       // It should have changed its p->state before coming back.
2837       proc = 0;
2838     }
2839     release(&ptable.lock);
2840     if (skipped > NPROC) {
2841       hlt();
2842       skipped = 0;
2843     }
2844   }
2845 }
2846
2847
2848
2849
```



```
2850 // Enter scheduler.  Must hold only ptable.lock
2851 // and have changed proc->state. Saves and restores
2852 // intena because intena is a property of this
2853 // kernel thread, not this CPU. It should
2854 // be proc->intena and proc->ncli, but that would
2855 // break in the few places where a lock is held but
2856 // there's no process.
2857 void
2858 sched(void)
2859 {
2860   int intena;
2861
2862
2863   if(!holding(&ptable.lock))
2864     panic("sched ptable.lock");
2865   if(cpu->ncli != 1)
2866     panic("sched locks");
2867   if(proc->state == RUNNING)
2868     panic("sched running");
2869   if(readeflags()&FL_IF)
2870     panic("sched interruptible");
2871   intena = cpu->intena;
2872
2873   swtch(&proc->context, cpu->scheduler);
2874   cpu->intena = intena;
2875 }
2876
2877 // Give up the CPU for one scheduling round.
2878 void
2879 yield(void)
2880 {
2881   acquire(&ptable.lock);
2882   proc->state = RUNNABLE;
2883   sched();
2884   release(&ptable.lock);
2885 }
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
```

```
2900 // A fork child's very first scheduling by scheduler()
2901 // will swtch here.  "Return" to user space.
2902 void
2903 forkret(void)
2904 {
2905   static int first = 1;
2906   // Still holding ptable.lock from scheduler.
2907   release(&ptable.lock);
2908
2909   if (first) {
2910     // Some initialization functions must be run in the context
2911     // of a regular process (e.g., they call sleep), and thus cannot
2912     // be run from main().
2913     first = 0;
2914     iinit(ROOTDEV);
2915     initlog(ROOTDEV);
2916   }
2917
2918   // Return to "caller", actually trapret (see allocproc).
2919 }
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
```



```
2950 // Atomically release lock and sleep on chan.
2951 // Reacquires lock when awakened.
2952 void
2953 sleep(void *chan, struct spinlock *lk)
2954 {
2955   if(proc == 0)
2956     panic("sleep");
2957
2958   if(lk == 0)
2959     panic("sleep without lk");
2960
2961   // Must acquire ptable.lock in order to
2962   // change p->state and then call sched.
2963   // Once we hold ptable.lock, we can be
2964   // guaranteed that we won't miss any wakeup
2965   // (wakeup runs with ptable.lock locked),
2966   // so it's okay to release lk.
2967   if(lk != &ptable.lock){
2968     acquire(&ptable.lock);
2969     release(lk);
2970   }
2971
2972   // Go to sleep.
2973   proc->chan = chan;
2974   proc->state = SLEEPING;
2975   sched();
2976
2977   // Tidy up.
2978   proc->chan = 0;
2979
2980   // Reacquire original lock.
2981   if(lk != &ptable.lock){
2982     release(&ptable.lock);
2983     acquire(lk);
2984   }
2985 }
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
```

```
3000 // Wake up all processes sleeping on chan.
3001 // The ptable lock must be held.
3002 static void
3003 wakeup1(void *chan)
3004 {
3005   struct proc *p;
3006
3007   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
3008     if(p->state == SLEEPING && p->chan == chan)
3009       p->state = RUNNABLE;
3010 }
3011
3012 // Wake up all processes sleeping on chan.
3013 void
3014 wakeup(void *chan)
3015 {
3016   acquire(&ptable.lock);
3017   wakeup1(chan);
3018   release(&ptable.lock);
3019 }
3020
3021 // Kill the process with the given pid.
3022 // Process won't exit until it returns
3023 // to user space (see trap in trap.c).
3024 int
3025 kill(int pid)
3026 {
3027   struct proc *p;
3028
3029   acquire(&ptable.lock);
3030   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3031     if(p->pid == pid){
3032       p->killed = 1;
3033       // Wake process from sleep if necessary.
3034       if(p->state == SLEEPING)
3035         p->state = RUNNABLE;
3036       release(&ptable.lock);
3037       return 0;
3038     }
3039   }
3040   release(&ptable.lock);
3041   return -1;
3042 }
3043
3044
3045
3046
3047
3048
3049
```



```
3050 // Print a process listing to console.  For debugging.
3051 // Runs when user types ^P on console.
3052 // No lock to avoid wedging a stuck machine further.
3053 void
3054 procdump(void)
3055 {
3056   static char *states[] = {
3057   [UNUSED]    "unused",
3058   [EMBRYO]    "embryo",
3059   [SLEEPING]  "sleep ",
3060   [RUNNABLE]  "runble",
3061   [RUNNING]   "run   ",
3062   [ZOMBIE]    "zombie"
3063   };
3064   int i;
3065   struct proc *p;
3066   char *state;
3067   addr_t pc[10];
3068
3069   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3070     if(p->state == UNUSED)
3071       continue;
3072     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
3073       state = states[p->state];
3074     else
3075       state = "???";
3076     cprintf("%d %s %s", p->pid, state, p->name);
3077     if(p->state == SLEEPING){
3078       getstackpcs((addr_t*)p->context->rbp+2, pc);
3079       for(i=0; i<10 && pc[i] != 0; i++)
3080         cprintf(" %p", pc[i]);
3081     }
3082     cprintf("\n");
3083   }
3084 }
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
```

```
3100 # Context switch
3101 #
3102 #   void swtch(struct context **old, struct context *new);
3103 #
3104 # Save current register context in old
3105 # and then load register context from new.
3106
3107 .global swtch
3108 swtch:
3109   # Save old callee-save registers
3110   pushq   %rbp
3111   pushq   %rbx
3112   pushq   %r12
3113   pushq   %r13
3114   pushq   %r14
3115   pushq   %r15
3116
3117   # Switch stacks
3118   movq    %rsp, (%rdi)
3119   movq    %rsi, %rsp
3120
3121   # Load new callee-save registers
3122   popq    %r15
3123   popq    %r14
3124   popq    %r13
3125   popq    %r12
3126   popq    %rbx
3127   popq    %rbp
3128
3129   retq #??
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
```



```
3150 // Physical memory allocator, intended to allocate
3151 // memory for user processes, kernel stacks, page table pages,
3152 // and pipe buffers. Allocates 4096-byte pages.
3153
3154 #include "types.h"
3155 #include "defs.h"
3156 #include "param.h"
3157 #include "memlayout.h"
3158 #include "mmu.h"
3159 #include "spinlock.h"
3160
3161 void freerange(void *vstart, void *vend);
3162 extern char end[]; // first address after kernel loaded from ELF file
3163
3164 struct run {
3165   struct run *next;
3166 };
3167
3168 struct {
3169   struct spinlock lock;
3170   int use_lock;
3171   struct run *freelist;
3172 } kmem;
3173
3174 // Initialization happens in two phases.
3175 // 1. main() calls kinit1() while still using entrypgdir to place just
3176 // the pages mapped by entrypgdir on free list.
3177 // 2. main() calls kinit2() with the rest of the physical pages
3178 // after installing a full page table that maps them on all cores.
3179 void
3180 kinit1(void *vstart, void *vend)
3181 {
3182   initlock(&kmem.lock, "kmem");
3183   kmem.use_lock = 0;
3184   kmem.freelist = 0; // empty
3185   freerange(vstart, vend);
3186 }
3187
3188 void
3189 kinit2(void *vstart, void *vend)
3190 {
3191   freerange(vstart, vend);
3192   kmem.use_lock = 1;
3193 }
3194
3195
3196
3197
3198
3199
```

```
3200 void
3201 freerange(void *vstart, void *vend)
3202 {
3203   char *p;
3204   p = (char*)PGROUNDUP((addr_t)vstart);
3205   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3206     kfree(p);
3207 }
3208
3209 // Free the page of physical memory pointed at by v,
3210 // which normally should have been returned by a
3211 // call to kalloc().  (The exception is when
3212 // initializing the allocator; see kinit above.)
3213 void
3214 kfree(char *v)
3215 {
3216   struct run *r;
3217
3218   if((addr_t)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
3219     panic("kfree");
3220
3221   // Fill with junk to catch dangling refs.
3222   memset(v, 1, PGSIZE);
3223
3224   if(kmem.use_lock)
3225     acquire(&kmem.lock);
3226   r = (struct run*)v;
3227   r->next = kmem.freelist;
3228   kmem.freelist = r;
3229   if(kmem.use_lock)
3230     release(&kmem.lock);
3231 }
3232
3233 // Allocate one 4096-byte page of physical memory.
3234 // Returns a pointer that the kernel can use.
3235 // Returns 0 if the memory cannot be allocated.
3236 char*
3237 kalloc(void)
3238 {
3239   struct run *r;
3240
3241   if(kmem.use_lock)
3242     acquire(&kmem.lock);
3243   r = kmem.freelist;
3244   if(r)
3245     kmem.freelist = r->next;
3246   if(kmem.use_lock)
3247     release(&kmem.lock);
3248   return (char*)r;
3249 }
```



```
3250 // x86 trap and interrupt constants.
3251
3252 // Processor-defined:
3253 #define T_DIVIDE         0      // divide error
3254 #define T_DEBUG          1      // debug exception
3255 #define T_NMI            2      // non-maskable interrupt
3256 #define T_BRKPT          3      // breakpoint
3257 #define T_OFLOW          4      // overflow
3258 #define T_BOUND          5      // bounds check
3259 #define T_ILLOP          6      // illegal opcode
3260 #define T_DEVICE         7      // device not available
3261 #define T_DBLFLT         8      // double fault
3262 // #define T_COPROC      9      // reserved (not used since 486)
3263 #define T_TSS           10      // invalid task switch segment
3264 #define T_SEGNP         11      // segment not present
3265 #define T_STACK         12      // stack exception
3266 #define T_GPFLT         13      // general protection fault
3267 #define T_PGFLT         14      // page fault
3268 // #define T_RES        15      // reserved
3269 #define T_FPERR         16      // floating point error
3270 #define T_ALIGN         17      // aligment check
3271 #define T_MCHK          18      // machine check
3272 #define T_SIMDERR       19      // SIMD floating point error
3273
3274 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
3275
3276 #define IRQ_TIMER        0
3277 #define IRQ_KBD          1
3278 #define IRQ_COM1         4
3279 #define IRQ_IDE         14
3280 #define IRQ_ERROR       19
3281 #define IRQ_SPURIOUS    31
3282
3283
3284
3285
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
```

```
#!/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 ".global alltraps\n";
for(my $i = 0; $i < 256; $i++){
    # print ".global vector$i\n"; # might be useful for debugging
    print "vector$i:\n";
    if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
        print "  push \$0\n";
    }
    print "  push \$$i\n";
    print "  jmp alltraps\n";
}

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



```
# vectors.S sends all traps here.
.global alltraps
alltraps:
  # Build trap frame.
  pushq   %r15
  pushq   %r14
  pushq   %r13
  pushq   %r12
  pushq   %r11
  pushq   %r10
  pushq   %r9
  pushq   %r8
  pushq   %rdi
  pushq   %rsi
  pushq   %rbp
  pushq   %rdx
  pushq   %rcx
  pushq   %rbx
  pushq   %rax

  movq    %rsp, %rdi  # frame in arg1
  callq   trap
# Return falls through to trapret...

.global trapret
trapret:
  popq    %rax
  popq    %rbx
  popq    %rcx
  popq    %rdx
  popq    %rbp
  popq    %rsi
  popq    %rdi
  popq    %r8
  popq    %r9
  popq    %r10
  popq    %r11
  popq    %r12
  popq    %r13
  popq    %r14
  popq    %r15

  addq    $16, %rsp  # discard trapnum and errorcode
  iretq
```

```
.global syscall_entry
syscall_entry:
  # switch to kernel stack. With the syscall instruction,
  # this is a kernel resposibility
  # store %rsp on the top of proc->kstack,
  movq    %rax, %fs:(0)      # save %rax above __thread vars
  movq    %fs:(-8), %rax     # %fs:(-8) is proc (the last __thread)
  movq    0x10(%rax), %rax   # get proc->kstack (see struct proc)
  addq    $(4096-16), %rax   # %rax points to tf->rsp
  movq    %rsp, (%rax)       # save user rsp to tf->rsp
  movq    %rax, %rsp         # switch to the kstack
  movq    %fs:(0), %rax      # restore %rax

  pushq   %r11         # rflags
  pushq   $0           # cs is ignored
  pushq   %rcx         # rip (next user insn)

  pushq   $0           # err
  pushq   $0           # trapno ignored

  pushq   %r15
  pushq   %r14
  pushq   %r13
  pushq   %r12
  pushq   %r11
  pushq   %r10
  pushq   %r9
  pushq   %r8
  pushq   %rdi
  pushq   %rsi
  pushq   %rbp
  pushq   %rdx
  pushq   %rcx
  pushq   %rbx
  pushq   %rax

  movq    %rsp, %rdi  # frame in arg1
  callq   syscall
# Return falls through to syscall_trapret...
```



```
.global syscall_trapret
syscall_trapret:
  popq    %rax
  popq    %rbx
  popq    %rcx
  popq    %rdx
  popq    %rbp
  popq    %rsi
  popq    %rdi
  popq    %r8
  popq    %r9
  popq    %r10
  popq    %r11
  popq    %r12
  popq    %r13
  popq    %r14
  popq    %r15

  addq    $40, %rsp  # discard trapnum, errorcode, rip, cs and rflags

  # to make sure we don't get any interrupts on the user stack while in
  # supervisor mode. this is actually slightly unsafe still,
  # since some interrupts are nonmaskable.
  # See https://www.felixcloutier.com/x86/sysret
  cli
  movq    (%rsp), %rsp  # restore the user stack
  sysretq
```

```
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 uint *idt;
3512 extern addr_t vectors[];  // in vectors.S: array of 256 entry pointers
3513 struct spinlock tickslock;
3514 uint ticks;
3515
3516 static void
3517 mkgate(uint *idt, uint n, addr_t kva, uint pl)
3518 {
3519   uint64 addr = (uint64) kva;
3520
3521   n *= 4;
3522   idt[n+0] = (addr & 0xFFFF) | (KERNEL_CS << 16);
3523   idt[n+1] = (addr & 0xFFFF0000) | 0x8E00 | ((pl & 3) << 13);
3524   idt[n+2] = addr >> 32;
3525   idt[n+3] = 0;
3526 }
3527
3528 void idtinit(void)
3529 {
3530   lidt((void*) idt, PGSIZE);
3531 }
3532
3533 void tvinit(void)
3534 {
3535   int n;
3536   idt = (uint*) kalloc();
3537   memset(idt, 0, PGSIZE);
3538
3539   for (n = 0; n < 256; n++)
3540     mkgate(idt, n, vectors[n], 0);
3541 }
3542
3543
3544
3545
3546
3547
3548
3549
```



```
3550 void
3551 trap(struct trapframe *tf)
3552 {
3553   switch(tf->trapno){
3554   case T_IRQ0 + IRQ_TIMER:
3555     if(cpunum() == 0){
3556       acquire(&tickslock);
3557       ticks++;
3558       wakeup(&ticks);
3559       release(&tickslock);
3560     }
3561     lapiceoi();
3562     break;
3563   case T_IRQ0 + IRQ_IDE:
3564     ideintr();
3565     lapiceoi();
3566     break;
3567   case T_IRQ0 + IRQ_IDE+1:
3568     // Bochs generates spurious IDE1 interrupts.
3569     break;
3570   case T_IRQ0 + IRQ_KBD:
3571     kbdintr();
3572     lapiceoi();
3573     break;
3574   case T_IRQ0 + IRQ_COM1:
3575     uartintr();
3576     lapiceoi();
3577     break;
3578   case T_IRQ0 + 7:
3579   case T_IRQ0 + IRQ_SPURIOUS:
3580     cprintf("cpu%d: spurious interrupt at %p:%p\n",
3581             cpunum(), tf->cs, tf->rip);
3582     lapiceoi();
3583     break;
3584
3585
3586   default:
3587     if(proc == 0 || (tf->cs&3) == 0){
3588       // In kernel, it must be our mistake.
3589       cprintf("unexpected trap %d from cpu %d rip %p (cr2=0x%p)\n",
3590               tf->trapno, cpunum(), tf->rip, rcr2());
3591       if (proc)
3592         cprintf("proc id: %d\n", proc->pid);
3593       panic("trap");
3594     }
3595     // In user space, assume process misbehaved.
3596     cprintf("pid %d %s: trap %d err %d on cpu %d "
3597             "rip 0x%p addr 0x%p--kill proc\n",
3598             proc->pid, proc->name, tf->trapno, tf->err, cpunum(), tf->rip,
3599             rcr2());
```

```
3600     proc->killed = 1;
3601   }
3602
3603   // Force process exit if it has been killed and is in user space.
3604   // (If it is still executing in the kernel, let it keep running
3605   // until it gets to the regular system call return.)
3606   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3607     exit();
3608
3609   // Force process to give up CPU on clock tick.
3610   // If interrupts were on while locks held, would need to check nlock.
3611   if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3612     yield();
3613
3614   // Check if the process has been killed since we yielded
3615   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3616     exit();
3617 }
3618
3619
3620
3621
3622
3623
3624
3625
3626
3627
3628
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
3673
3674
3675
3676
3677
3678
3679
3680
3681
3682
3683
3684
3685
3686
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699
```

```
3700 #include "types.h"
3701 #include "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 // Fetch the int at addr from the current process.
3710 int
3711 fetchint(addr_t addr, int *ip)
3712 {
3713   if(addr >= proc->sz || addr+sizeof(int) > proc->sz)
3714     return -1;
3715   *ip = *(int*)(addr);
3716   return 0;
3717 }
3718
3719 int
3720 fetchaddr(addr_t addr, addr_t *ip)
3721 {
3722   if(addr >= proc->sz || addr+sizeof(addr_t) > proc->sz)
3723     return -1;
3724   *ip = *(addr_t*)(addr);
3725   return 0;
3726 }
3727
3728 // Fetch the nul-terminated string at addr from the current process.
3729 // Doesn't actually copy the string - just sets *pp to point at it.
3730 // Returns length of string, not including nul.
3731 int
3732 fetchstr(addr_t addr, char **pp)
3733 {
3734   char *s, *ep;
3735
3736   if(addr >= proc->sz)
3737     return -1;
3738   *pp = (char*)addr;
3739   ep = (char*)proc->sz;
3740   for(s = *pp; s < ep; s++)
3741     if(*s == 0)
3742       return s - *pp;
3743   return -1;
3744 }
3745
3746
3747
3748
3749
```



```
3750 static addr_t
3751 fetcharg(int n)
3752 {
3753   switch (n) {
3754   case 0: return proc->tf->rdi;
3755   case 1: return proc->tf->rsi;
3756   case 2: return proc->tf->rdx;
3757   case 3: return proc->tf->r10;
3758   case 4: return proc->tf->r8;
3759   case 5: return proc->tf->r9;
3760   }
3761   panic("failed fetch");
3762 }
3763
3764 int
3765 argint(int n, int *ip)
3766 {
3767   *ip = fetcharg(n);
3768   return 0;
3769 }
3770
3771 int
3772 argaddr(int n, addr_t *ip)
3773 {
3774   *ip = fetcharg(n);
3775   return 0;
3776 }
3777
3778 // Fetch the nth word-sized system call argument as a pointer
3779 // to a block of memory of size bytes.  Check that the pointer
3780 // lies within the process address space.
3781 int
3782 argptr(int n, char **pp, int size)
3783 {
3784   addr_t i;
3785
3786   if(argaddr(n, &i) < 0)
3787     return -1;
3788   if(size < 0 || (uint)i >= proc->sz || (uint)i+size > proc->sz)
3789     return -1;
3790   *pp = (char*)i;
3791   return 0;
3792 }
3793
3794
3795
3796
3797
3798
3799
```

```
3800 // Fetch the nth word-sized system call argument as a string pointer.
3801 // Check that the pointer is valid and the string is nul-terminated.
3802 // (There is no shared writable memory, so the string can't change
3803 // between this check and being used by the kernel.)
3804 int
3805 argstr(int n, char **pp)
3806 {
3807   int addr;
3808   if(argint(n, &addr) < 0)
3809     return -1;
3810   return fetchstr(addr, pp);
3811 }
3812
3813 extern addr_t sys_chdir(void);
3814 extern addr_t sys_close(void);
3815 extern addr_t sys_dup(void);
3816 extern addr_t sys_exec(void);
3817 extern addr_t sys_exit(void);
3818 extern addr_t sys_fork(void);
3819 extern addr_t sys_fstat(void);
3820 extern addr_t sys_getpid(void);
3821 extern addr_t sys_kill(void);
3822 extern addr_t sys_link(void);
3823 extern addr_t sys_mkdir(void);
3824 extern addr_t sys_mknod(void);
3825 extern addr_t sys_open(void);
3826 extern addr_t sys_pipe(void);
3827 extern addr_t sys_read(void);
3828 extern addr_t sys_sbrk(void);
3829 extern addr_t sys_sleep(void);
3830 extern addr_t sys_unlink(void);
3831 extern addr_t sys_wait(void);
3832 extern addr_t sys_write(void);
3833 extern addr_t sys_uptime(void);
3834
3835
3836
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
```



```
3850 static addr_t (*syscalls[])(void) = {
3851 [SYS_fork]    sys_fork,
3852 [SYS_exit]    sys_exit,
3853 [SYS_wait]    sys_wait,
3854 [SYS_pipe]    sys_pipe,
3855 [SYS_read]    sys_read,
3856 [SYS_kill]    sys_kill,
3857 [SYS_exec]    sys_exec,
3858 [SYS_fstat]   sys_fstat,
3859 [SYS_chdir]   sys_chdir,
3860 [SYS_dup]     sys_dup,
3861 [SYS_getpid]  sys_getpid,
3862 [SYS_sbrk]    sys_sbrk,
3863 [SYS_sleep]   sys_sleep,
3864 [SYS_uptime]  sys_uptime,
3865 [SYS_open]    sys_open,
3866 [SYS_write]   sys_write,
3867 [SYS_mknod]   sys_mknod,
3868 [SYS_unlink]  sys_unlink,
3869 [SYS_link]    sys_link,
3870 [SYS_mkdir]   sys_mkdir,
3871 [SYS_close]   sys_close,
3872 };
3873
3874 void
3875 syscall(struct trapframe *tf)
3876 {
3877   if (proc->killed)
3878     exit();
3879   proc->tf = tf;
3880   uint64 num = proc->tf->rax;
3881   if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3882     tf->rax = syscalls[num]();
3883   } else {
3884     cprintf("%d %s: unknown sys call %d\n",
3885             proc->pid, proc->name, num);
3886     tf->rax = -1;
3887   }
3888   if (proc->killed)
3889     exit();
3890 }
3891
3892
3893
3894
3895
3896
3897
3898
3899
```

```
3900 #include "types.h"
3901 #include "x86.h"
3902 #include "defs.h"
3903 #include "param.h"
3904 #include "memlayout.h"
3905 #include "mmu.h"
3906 #include "proc.h"
3907
3908 int
3909 sys_fork(void)
3910 {
3911   return fork();
3912 }
3913
3914 int
3915 sys_exit(void)
3916 {
3917   exit();
3918   return 0;  // not reached
3919 }
3920
3921 int
3922 sys_wait(void)
3923 {
3924   return wait();
3925 }
3926
3927 int
3928 sys_kill(void)
3929 {
3930   int pid;
3931
3932   if(argint(0, &pid) < 0)
3933     return -1;
3934   return kill(pid);
3935 }
3936
3937 int
3938 sys_getpid(void)
3939 {
3940   return proc->pid;
3941 }
3942
3943
3944
3945
3946
3947
3948
3949
```



```
3950 addr_t
3951 sys_sbrk(void)
3952 {
3953   addr_t addr;
3954   addr_t n;
3955
3956   argaddr(0, &n);
3957   addr = proc->sz;
3958   if(growproc(n) < 0)
3959     return -1;
3960   return addr;
3961 }
3962
3963 int
3964 sys_sleep(void)
3965 {
3966   int n;
3967   uint ticks0;
3968
3969   if(argint(0, &n) < 0)
3970     return -1;
3971   acquire(&tickslock);
3972   ticks0 = ticks;
3973   while(ticks - ticks0 < n){
3974     if(proc->killed){
3975       release(&tickslock);
3976       return -1;
3977     }
3978     sleep(&ticks, &tickslock);
3979   }
3980   release(&tickslock);
3981   return 0;
3982 }
3983
3984 // return how many clock tick interrupts have occurred
3985 // since start.
3986 int
3987 sys_uptime(void)
3988 {
3989   uint xticks;
3990
3991   acquire(&tickslock);
3992   xticks = ticks;
3993   release(&tickslock);
3994   return xticks;
3995 }
3996
3997
3998
3999
```

```
4000 #include "types.h"
4001 #include "param.h"
4002 #include "memlayout.h"
4003 #include "mmu.h"
4004 #include "proc.h"
4005 #include "defs.h"
4006 #include "x86.h"
4007 #include "elf.h"
4008
4009 int
4010 exec(char *path, char **argv)
4011 {
4012   char *s, *last;
4013   int i, off;
4014   addr_t argc, sz, sp, ustack[3+MAXARG+1];
4015   struct elfhdr elf;
4016   struct inode *ip;
4017   struct proghdr ph;
4018   pde_t *pgdir, *oldpgdir;
4019
4020   oldpgdir = proc->pgdir;
4021
4022   begin_op();
4023
4024   if((ip = namei(path)) == 0){
4025     end_op();
4026     return -1;
4027   }
4028   ilock(ip);
4029   pgdir = 0;
4030
4031   // Check ELF header
4032   if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
4033     goto bad;
4034   if(elf.magic != ELF_MAGIC)
4035     goto bad;
4036
4037   if((pgdir = setupkvm()) == 0)
4038     goto bad;
4039
4040   // Load program into memory.
4041   sz = PGSIZE; // skip the first page
4042   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
4043     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
4044       goto bad;
4045     if(ph.type != ELF_PROG_LOAD)
4046       continue;
4047     if(ph.memsz < ph.filesz)
4048       goto bad;
4049     if(ph.vaddr + ph.memsz < ph.vaddr)
```



```
4050       goto bad;
4051     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
4052       goto bad;
4053     if(ph.vaddr % PGSIZE != 0)
4054       goto bad;
4055     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
4056       goto bad;
4057   }
4058   iunlockput(ip);
4059   end_op();
4060   ip = 0;
4061
4062   // Allocate two pages at the next page boundary.
4063   // Make the first inaccessible.  Use the second as the user stack.
4064   sz = PGROUNDUP(sz);
4065   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
4066     goto bad;
4067   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
4068   sp = sz;
4069   // Push argument strings, prepare rest of stack in ustack.
4070   for(argc = 0; argv[argc]; argc++) {
4071     if(argc >= MAXARG)
4072       goto bad;
4073     sp = (sp - (strlen(argv[argc]) + 1)) & ~(sizeof(addr_t)-1);
4074     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
4075       goto bad;
4076     ustack[3+argc] = sp;
4077   }
4078   ustack[3+argc] = 0;
4079
4080   ustack[0] = 0xffffffff;  // fake return PC
4081   ustack[1] = argc;
4082   ustack[2] = sp - (argc+1)*sizeof(addr_t);  // argv pointer
4083
4084   proc->tf->rdi = argc;
4085   proc->tf->rsi = sp - (argc+1)*sizeof(addr_t);
4086
4087
4088   sp -= (3+argc+1) * sizeof(addr_t);
4089   if(copyout(pgdir, sp, ustack, (3+argc+1)*sizeof(addr_t)) < 0)
4090     goto bad;
4091
4092   // Save program name for debugging.
4093   for(last=s=path; *s; s++)
4094     if(*s == '/')
4095       last = s+1;
4096   safestrcpy(proc->name, last, sizeof(proc->name));
4097
4098
4099
```

```
4100   // Commit to the user image.
4101   proc->pgdir = pgdir;
4102   proc->sz = sz;
4103   proc->tf->rip = elf.entry;  // main
4104   proc->tf->rcx = elf.entry;
4105   proc->tf->rsp = sp;
4106   switchuvm(proc);
4107   freevm(oldpgdir);
4108   return 0;
4109
4110  bad:
4111   if(pgdir)
4112     freevm(pgdir);
4113   if(ip){
4114     iunlockput(ip);
4115     end_op();
4116   }
4117   return -1;
4118 }
4119
4120
4121
4122
4123
4124
4125
4126
4127
4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```



```
4150 // See MultiProcessor Specification Version 1.[14]
4151
4152 struct mp {             // floating pointer
4153   uchar signature[4];           // "_MP_"
4154   uint32 physaddr;               // 32-bit phys addr of MP config table
4155   uchar length;                 // 1
4156   uchar specrev;                // [14]
4157   uchar checksum;               // all bytes must add up to 0
4158   uchar type;                   // MP system config type
4159   uchar imcrp;
4160   uchar reserved[3];
4161 };
4162
4163 struct mpconf {         // configuration table header
4164   uchar signature[4];           // "PCMP"
4165   ushort length;                // total table length
4166   uchar version;                // [14]
4167   uchar checksum;               // all bytes must add up to 0
4168   uchar product[20];            // product id
4169   uint32 oemtable_p;               // OEM table pointer
4170   ushort oemlength;             // OEM table length
4171   ushort entry;                 // entry count
4172   uint32 lapicaddr_p;              // address of local APIC
4173   ushort xlength;               // extended table length
4174   uchar xchecksum;              // extended table checksum
4175   uchar reserved;
4176 };
4177
4178 struct mpproc {         // processor table entry
4179   uchar type;                   // entry type (0)
4180   uchar apicid;                 // local APIC id
4181   uchar version;                // local APIC verison
4182   uchar flags;                  // CPU flags
4183     #define MPBOOT 0x02           // This proc is the bootstrap processor.
4184   uchar signature[4];           // CPU signature
4185   uint feature;                 // feature flags from CPUID instruction
4186   uchar reserved[8];
4187 };
4188
4189 struct mpioapic {       // I/O APIC table entry
4190   uchar type;                   // entry type (2)
4191   uchar apicno;                 // I/O APIC id
4192   uchar version;                // I/O APIC version
4193   uchar flags;                  // I/O APIC flags
4194   uint32 addr_p;                  // I/O APIC address
4195 };
4196
4197
4198
4199
```

```
4200 // Table entry types
4201 #define MPPROC    0x00  // One per processor
4202 #define MPBUS     0x01  // One per bus
4203 #define MPIOAPIC  0x02  // One per I/O APIC
4204 #define MPIOINTR  0x03  // One per bus interrupt source
4205 #define MPLINTR   0x04  // One per system interrupt source
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 // Multiprocessor support
4251 // Search memory for MP description structures.
4252 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
4253
4254 #include "types.h"
4255 #include "defs.h"
4256 #include "param.h"
4257 #include "memlayout.h"
4258 #include "mp.h"
4259 #include "x86.h"
4260 #include "mmu.h"
4261 #include "proc.h"
4262
4263 struct cpu cpus[NCPU];
4264 int ncpu;
4265 uchar ioapicid;
4266
4267 static uchar
4268 sum(uchar *addr, int len)
4269 {
4270   int i, sum;
4271
4272   sum = 0;
4273   for(i=0; i<len; i++)
4274     sum += addr[i];
4275   return sum;
4276 }
4277
4278 // Look for an MP structure in the len bytes at addr.
4279 static struct mp*
4280 mpsearch1(addr_t a, int len)
4281 {
4282   uchar *e, *p, *addr;
4283   addr = P2V(a);
4284   e = addr+len;
4285   for(p = addr; p < e; p += sizeof(struct mp))
4286     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
4287       return (struct mp*)p;
4288   return 0;
4289 }
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
```

```
4300 // Search for the MP Floating Pointer Structure, which according to the
4301 // spec is in one of the following three locations:
4302 // 1) in the first KB of the EBDA;
4303 // 2) in the last KB of system base memory;
4304 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
4305 static struct mp*
4306 mpsearch(void)
4307 {
4308   uchar *bda;
4309   uint p;
4310   struct mp *mp;
4311
4312   bda = (uchar *) P2V(0x400);
4313   if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
4314     if((mp = mpsearch1(p, 1024)))
4315       return mp;
4316   } else {
4317     p = ((bda[0x14]<<8)|bda[0x13])*1024;
4318     if((mp = mpsearch1(p-1024, 1024)))
4319       return mp;
4320   }
4321   return mpsearch1(0xF0000, 0x10000);
4322 }
4323
4324 // Search for an MP configuration table.  For now,
4325 // don't accept the default configurations (physaddr == 0).
4326 // Check for correct signature, calculate the checksum and,
4327 // if correct, check the version.
4328 // To do: check extended table checksum.
4329 static struct mpconf*
4330 mpconfig(struct mp **pmp)
4331 {
4332   struct mpconf *conf;
4333   struct mp *mp;
4334
4335   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
4336     return 0;
4337   conf = (struct mpconf*) P2V((addr_t) mp->physaddr);
4338   if(memcmp(conf, "PCMP", 4) != 0)
4339     return 0;
4340   if(conf->version != 1 && conf->version != 4)
4341     return 0;
4342   if(sum((uchar*)conf, conf->length) != 0)
4343     return 0;
4344   *pmp = mp;
4345   return conf;
4346 }
4347
4348
4349
```



```
4350 void
4351 mpinit(void)
4352 {
4353   uchar *p, *e;
4354   struct mp *mp;
4355   struct mpconf *conf;
4356   struct mpproc *proc;
4357   struct mpioapic *ioapic;
4358
4359   if((conf = mpconfig(&mp)) == 0) {
4360     cprintf("No other CPUs found.\n");
4361     return;
4362   }
4363   lapic = P2V((addr_t)conf->lapicaddr_p);
4364   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
4365     switch(*p){
4366     case MPPROC:
4367       proc = (struct mpproc*)p;
4368       if(ncpu < NCPU) {
4369         cpus[ncpu].apicid = proc->apicid;  // apicid may differ from ncpu
4370         ncpu++;
4371       }
4372       p += sizeof(struct mpproc);
4373       continue;
4374     case MPIOAPIC:
4375       ioapic = (struct mpioapic*)p;
4376       ioapicid = ioapic->apicno;
4377       p += sizeof(struct mpioapic);
4378       continue;
4379     case MPBUS:
4380     case MPIOINTR:
4381     case MPLINTR:
4382       p += 8;
4383       continue;
4384     default:
4385       panic("Major problem parsing mp config.");
4386       break;
4387     }
4388   }
4389   cprintf("Seems we are SMP, ncpu = %d\n",ncpu);
4390   if(mp->imcrp){
4391     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
4392     // But it would on real hardware.
4393     outb(0x22, 0x70);   // Select IMCR
4394     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
4395   }
4396 }
4397
4398
4399
```

```
4400 // The local APIC manages internal (non-I/O) interrupts.
4401 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
4402
4403 #include "param.h"
4404 #include "types.h"
4405 #include "defs.h"
4406 #include "date.h"
4407 #include "memlayout.h"
4408 #include "traps.h"
4409 #include "mmu.h"
4410 #include "x86.h"
4411 #include "proc.h"  // ncpu
4412
4413 // Local APIC registers, divided by 4 for use as uint[] indices.
4414 #define ID      (0x0020/4)   // ID
4415 #define VER     (0x0030/4)   // Version
4416 #define TPR     (0x0080/4)   // Task Priority
4417 #define EOI     (0x00B0/4)   // EOI
4418 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
4419   #define ENABLE     0x00000100   // Unit Enable
4420 #define ESR     (0x0280/4)   // Error Status
4421 #define ICRLO   (0x0300/4)   // Interrupt Command
4422   #define INIT       0x00000500   // INIT/RESET
4423   #define STARTUP    0x00000600   // Startup IPI
4424   #define DELIVS     0x00001000   // Delivery status
4425   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
4426   #define LEVEL      0x00008000   // Level triggered
4427   #define BCAST      0x00080000   // Send to all APICs, including self.
4428   #define BUSY       0x00001000
4429 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
4430 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
4431   #define X1         0x0000000B   // divide counts by 1
4432   #define PERIODIC   0x00020000   // Periodic
4433 #define PCINT   (0x0340/4)   // Performance Counter LVT
4434 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
4435 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
4436 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
4437   #define MASKED     0x00010000   // Interrupt masked
4438 #define TICR    (0x0380/4)   // Timer Initial Count
4439 #define TCCR    (0x0390/4)   // Timer Current Count
4440 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
4441
4442 volatile uint *lapic;  // Initialized in mp.c
4443
4444 static void
4445 lapicw(int index, int value)
4446 {
4447   lapic[index] = value;
4448   lapic[ID];  // wait for write to finish, by reading
4449 }
```



```
4450 void
4451 lapicinit(void)
4452 {
4453   if(!lapic)
4454     return;
4455
4456   // Enable local APIC; set spurious interrupt vector.
4457   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
4458
4459   // The timer repeatedly counts down at bus frequency
4460   // from lapic[TICR] and then issues an interrupt.
4461   // If xv6 cared more about precise timekeeping,
4462   // TICR would be calibrated using an external time source.
4463   lapicw(TDCR, X1);
4464   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
4465   lapicw(TICR, 10000000);
4466
4467   // Disable logical interrupt lines.
4468   lapicw(LINT0, MASKED);
4469   lapicw(LINT1, MASKED);
4470
4471   // Disable performance counter overflow interrupts
4472   // on machines that provide that interrupt entry.
4473   if(((lapic[VER]>>16) & 0xFF) >= 4)
4474     lapicw(PCINT, MASKED);
4475
4476   // Map error interrupt to IRQ_ERROR.
4477   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
4478
4479   // Clear error status register (requires back-to-back writes).
4480   lapicw(ESR, 0);
4481   lapicw(ESR, 0);
4482
4483   // Ack any outstanding interrupts.
4484   lapicw(EOI, 0);
4485
4486   // Send an Init Level De-Assert to synchronise arbitration ID's.
4487   lapicw(ICRHI, 0);
4488   lapicw(ICRLO, BCAST | INIT | LEVEL);
4489   while(lapic[ICRLO] & DELIVS)
4490     ;
4491
4492   // Enable interrupts on the APIC (but not on the processor).
4493   lapicw(TPR, 0);
4494 }
4495
4496
4497
4498
4499
```

```
4500 int
4501 cpunum(void)
4502 {
4503   int apicid, i;
4504
4505   // Cannot call cpu when interrupts are enabled:
4506   // result not guaranteed to last long enough to be used!
4507   // Would prefer to panic but even printing is chancy here:
4508   // almost everything, including cprintf and panic, calls cpu,
4509   // often indirectly through acquire and release.
4510   if(readeflags()&FL_IF){
4511     static int n;
4512     if(n++ == 0)
4513       cprintf("cpu called from %x with interrupts enabled\n",
4514         __builtin_return_address(0));
4515   }
4516
4517   if (!lapic)
4518     return 0;
4519
4520   apicid = lapic[ID] >> 24;
4521   for (i = 0; i < ncpu; ++i) {
4522     if (cpus[i].apicid == apicid)
4523       return i;
4524   }
4525   panic("unknown apicid\n");
4526 }
4527
4528 // Acknowledge interrupt.
4529 void
4530 lapiceoi(void)
4531 {
4532   if(lapic)
4533     lapicw(EOI, 0);
4534 }
4535
4536 // Spin for a given number of microseconds.
4537 // On real hardware would want to tune this dynamically.
4538 void
4539 microdelay(int us)
4540 {
4541 }
4542
4543
4544
4545
4546
4547
4548
4549
```



```
4550 #define CMOS_PORT    0x70
4551 #define CMOS_RETURN  0x71
4552
4553 // Start additional processor running entry code at addr.
4554 // See Appendix B of MultiProcessor Specification.
4555 void
4556 lapicstartap(uchar apicid, uint addr)
4557 {
4558   int i;
4559   ushort *wrv;
4560
4561   // "The BSP must initialize CMOS shutdown code to 0AH
4562   // and the warm reset vector (DWORD based at 40:67) to point at
4563   // the AP startup code prior to the [universal startup algorithm]."
4564   outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
4565   outb(CMOS_PORT+1, 0x0A);
4566   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
4567   wrv[0] = 0;
4568   wrv[1] = addr >> 4;
4569
4570   // "Universal startup algorithm."
4571   // Send INIT (level-triggered) interrupt to reset other CPU.
4572   lapicw(ICRHI, apicid<<24);
4573   lapicw(ICRLO, INIT | LEVEL | ASSERT);
4574   microdelay(200);
4575   lapicw(ICRLO, INIT | LEVEL);
4576   microdelay(100);    // should be 10ms, but too slow in Bochs!
4577
4578   // Send startup IPI (twice!) to enter code.
4579   // Regular hardware is supposed to only accept a STARTUP
4580   // when it is in the halted state due to an INIT.  So the second
4581   // should be ignored, but it is part of the official Intel algorithm.
4582   // Bochs complains about the second one.  Too bad for Bochs.
4583   for(i = 0; i < 2; i++){
4584     lapicw(ICRHI, apicid<<24);
4585     lapicw(ICRLO, STARTUP | (addr>>12));
4586     microdelay(200);
4587   }
4588 }
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4599
```

```
4600 #define CMOS_STATA   0x0a
4601 #define CMOS_STATB   0x0b
4602 #define CMOS_UIP    (1 << 7)        // RTC update in progress
4603
4604 #define SECS    0x00
4605 #define MINS    0x02
4606 #define HOURS   0x04
4607 #define DAY     0x07
4608 #define MONTH   0x08
4609 #define YEAR    0x09
4610
4611 static uint cmos_read(uint reg)
4612 {
4613   outb(CMOS_PORT,  reg);
4614   microdelay(200);
4615
4616   return inb(CMOS_RETURN);
4617 }
4618
4619 static void fill_rtcdate(struct rtcdate *r)
4620 {
4621   r->second = cmos_read(SECS);
4622   r->minute = cmos_read(MINS);
4623   r->hour   = cmos_read(HOURS);
4624   r->day    = cmos_read(DAY);
4625   r->month  = cmos_read(MONTH);
4626   r->year   = cmos_read(YEAR);
4627 }
4628
4629
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649
```



```
4650 // qemu seems to use 24-hour GWT and the values are BCD encoded
4651 void cmostime(struct rtcdate *r)
4652 {
4653   struct rtcdate t1, t2;
4654   int sb, bcd;
4655
4656   sb = cmos_read(CMOS_STATB);
4657
4658   bcd = (sb & (1 << 2)) == 0;
4659
4660   // make sure CMOS doesn't modify time while we read it
4661   for(;;) {
4662     fill_rtcdate(&t1);
4663     if(cmos_read(CMOS_STATA) & CMOS_UIP)
4664         continue;
4665     fill_rtcdate(&t2);
4666     if(memcmp(&t1, &t2, sizeof(t1)) == 0)
4667       break;
4668   }
4669
4670   // convert
4671   if(bcd) {
4672 #define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
4673     CONV(second);
4674     CONV(minute);
4675     CONV(hour  );
4676     CONV(day   );
4677     CONV(month );
4678     CONV(year  );
4679 #undef     CONV
4680   }
4681
4682   *r = t1;
4683   r->year += 2000;
4684 }
4685
4686
4687
4688
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4699
```

```
4700 // The I/O APIC manages hardware interrupts for an SMP system.
4701 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
4702 // See also picirq.c.
4703
4704 #include "types.h"
4705 #include "defs.h"
4706 #include "traps.h"
4707 #include "memlayout.h"
4708
4709 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
4710
4711 #define REG_ID     0x00  // Register index: ID
4712 #define REG_VER    0x01  // Register index: version
4713 #define REG_TABLE  0x10  // Redirection table base
4714
4715 // The redirection table starts at REG_TABLE and uses
4716 // two registers to configure each interrupt.
4717 // The first (low) register in a pair contains configuration bits.
4718 // The second (high) register contains a bitmask telling which
4719 // CPUs can serve that interrupt.
4720 #define INT_DISABLED   0x00010000  // Interrupt disabled
4721 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
4722 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
4723 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
4724
4725 volatile struct ioapic *ioapic;
4726
4727 // IO APIC MMIO structure: write reg, then read or write data.
4728 struct ioapic {
4729   uint reg;
4730   uint pad[3];
4731   uint data;
4732 };
4733
4734 static uint
4735 ioapicread(int reg)
4736 {
4737   ioapic->reg = reg;
4738   return ioapic->data;
4739 }
4740
4741 static void
4742 ioapicwrite(int reg, uint data)
4743 {
4744   ioapic->reg = reg;
4745   ioapic->data = data;
4746 }
4747
4748
4749
```



```
4750 void
4751 ioapicinit(void)
4752 {
4753   int i, id, maxintr;
4754
4755   ioapic = P2V((volatile struct ioapic*)IOAPIC);
4756   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
4757   id = ioapicread(REG_ID) >> 24;
4758   if(id != ioapicid)
4759     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
4760
4761   // Mark all interrupts edge-triggered, active high, disabled,
4762   // and not routed to any CPUs.
4763   for(i = 0; i <= maxintr; i++){
4764     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
4765     ioapicwrite(REG_TABLE+2*i+1, 0);
4766   }
4767 }
4768
4769 void
4770 ioapicenable(int irq, int cpunum)
4771 {
4772   // Mark interrupt edge-triggered, active high,
4773   // enabled, and routed to the given cpunum,
4774   // which happens to be that cpu's APIC ID.
4775   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
4776   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
4777 }
4778
4779
4780
4781
4782
4783
4784
4785
4786
4787
4788
4789
4790
4791
4792
4793
4794
4795
4796
4797
4798
4799
```

```
4800 // PC keyboard interface constants
4801
4802 #define KBSTATP         0x64    // kbd controller status port(I)
4803 #define KBS_DIB         0x01    // kbd data in buffer
4804 #define KBDATAP         0x60    // kbd data port(I)
4805
4806 #define NO              0
4807
4808 #define SHIFT           (1<<0)
4809 #define CTL             (1<<1)
4810 #define ALT             (1<<2)
4811
4812 #define CAPSLOCK        (1<<3)
4813 #define NUMLOCK         (1<<4)
4814 #define SCROLLLOCK      (1<<5)
4815
4816 #define E0ESC           (1<<6)
4817
4818 // Special keycodes
4819 #define KEY_HOME        0xE0
4820 #define KEY_END         0xE1
4821 #define KEY_UP          0xE2
4822 #define KEY_DN          0xE3
4823 #define KEY_LF          0xE4
4824 #define KEY_RT          0xE5
4825 #define KEY_PGUP        0xE6
4826 #define KEY_PGDN        0xE7
4827 #define KEY_INS         0xE8
4828 #define KEY_DEL         0xE9
4829
4830 // C('A') == Control-A
4831 #define C(x) (x - '@')
4832
4833 static uchar shiftcode[256] =
4834 {
4835   [0x1D] CTL,
4836   [0x2A] SHIFT,
4837   [0x36] SHIFT,
4838   [0x38] ALT,
4839   [0x9D] CTL,
4840   [0xB8] ALT
4841 };
4842
4843 static uchar togglecode[256] =
4844 {
4845   [0x3A] CAPSLOCK,
4846   [0x45] NUMLOCK,
4847   [0x46] SCROLLLOCK
4848 };
4849
```



```
4850 static uchar normalmap[256] =
4851 {
4852   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
4853   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
4854   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
4855   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
4856   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
4857   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
4858   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
4859   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
4860   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
4861   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
4862   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
4863   [0x9C] '\n',      // KP_Enter
4864   [0xB5] '/',       // KP_Div
4865   [0xC8] KEY_UP,    [0xD0] KEY_DN,
4866   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
4867   [0xCB] KEY_LF,    [0xCD] KEY_RT,
4868   [0x97] KEY_HOME,  [0xCF] KEY_END,
4869   [0xD2] KEY_INS,   [0xD3] KEY_DEL
4870 };
4871
4872 static uchar shiftmap[256] =
4873 {
4874   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
4875   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
4876   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
4877   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
4878   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
4879   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
4880   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
4881   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
4882   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
4883   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
4884   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
4885   [0x9C] '\n',      // KP_Enter
4886   [0xB5] '/',       // KP_Div
4887   [0xC8] KEY_UP,    [0xD0] KEY_DN,
4888   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
4889   [0xCB] KEY_LF,    [0xCD] KEY_RT,
4890   [0x97] KEY_HOME,  [0xCF] KEY_END,
4891   [0xD2] KEY_INS,   [0xD3] KEY_DEL
4892 };
4893
4894
4895
4896
4897
4898
4899
```

```
4900 static uchar ctlmap[256] =
4901 {
4902   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
4903   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
4904   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
4905   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
4906   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
4907   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
4908   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
4909   [0x9C] '\r',      // KP_Enter
4910   [0xB5] C('/'),    // KP_Div
4911   [0xC8] KEY_UP,    [0xD0] KEY_DN,
4912   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
4913   [0xCB] KEY_LF,    [0xCD] KEY_RT,
4914   [0x97] KEY_HOME,  [0xCF] KEY_END,
4915   [0xD2] KEY_INS,   [0xD3] KEY_DEL
4916 };
4917
4918
4919
4920
4921
4922
4923
4924
4925
4926
4927
4928
4929
4930
4931
4932
4933
4934
4935
4936
4937
4938
4939
4940
4941
4942
4943
4944
4945
4946
4947
4948
4949
```



```
4950 #include "types.h"
4951 #include "x86.h"
4952 #include "defs.h"
4953 #include "kbd.h"
4954
4955 int
4956 kbdgetc(void)
4957 {
4958   static uint shift;
4959   static uchar *charcode[4] = {
4960     normalmap, shiftmap, ctlmap, ctlmap
4961   };
4962   uint st, data, c;
4963
4964   st = inb(KBSTATP);
4965   if((st & KBS_DIB) == 0)
4966     return -1;
4967   data = inb(KBDATAP);
4968
4969   if(data == 0xE0){
4970     shift |= E0ESC;
4971     return 0;
4972   } else if(data & 0x80){
4973     // Key released
4974     data = (shift & E0ESC ? data : data & 0x7F);
4975     shift &= ~(shiftcode[data] | E0ESC);
4976     return 0;
4977   } else if(shift & E0ESC){
4978     // Last character was an E0 escape; or with 0x80
4979     data |= 0x80;
4980     shift &= ~E0ESC;
4981   }
4982
4983   shift |= shiftcode[data];
4984   shift ^= togglecode[data];
4985   c = charcode[shift & (CTL | SHIFT)][data];
4986   if(shift & CAPSLOCK){
4987     if('a' <= c && c <= 'z')
4988       c += 'A' - 'a';
4989     else if('A' <= c && c <= 'Z')
4990       c += 'a' - 'A';
4991   }
4992   return c;
4993 }
4994
4995 void
4996 kbdintr(void)
4997 {
4998   consoleintr(kbdgetc);
4999 }
```

```
5000 // Console input and output.
5001 // Input is from the keyboard or serial port.
5002 // Output is written to the screen and serial port.
5003
5004 #include <stdarg.h>
5005
5006 #include "types.h"
5007 #include "defs.h"
5008 #include "param.h"
5009 #include "traps.h"
5010 #include "spinlock.h"
5011 #include "sleeplock.h"
5012 #include "fs.h"
5013 #include "file.h"
5014 #include "memlayout.h"
5015 #include "mmu.h"
5016 #include "proc.h"
5017 #include "x86.h"
5018
5019 static void consputc(int);
5020
5021 static int panicked = 0;
5022
5023 static struct {
5024   struct spinlock lock;
5025   int locking;
5026 } cons;
5027
5028 static char digits[] = "0123456789abcdef";
5029
5030   static void
5031 print_x64(addr_t x)
5032 {
5033   int i;
5034   for (i = 0; i < (sizeof(addr_t) * 2); i++, x <<= 4)
5035     consputc(digits[x >> (sizeof(addr_t) * 8 - 4)]);
5036 }
5037
5038   static void
5039 print_x32(uint x)
5040 {
5041   int i;
5042   for (i = 0; i < (sizeof(uint) * 2); i++, x <<= 4)
5043     consputc(digits[x >> (sizeof(uint) * 8 - 4)]);
5044 }
5045
5046
5047
5048
5049
```



```
5050   static void
5051 print_d(int v)
5052 {
5053   char buf[16];
5054   int64 x = v;
5055
5056   if (v < 0)
5057     x = -x;
5058
5059   int i = 0;
5060   do {
5061     buf[i++] = digits[x % 10];
5062     x /= 10;
5063   } while(x != 0);
5064
5065   if (v < 0)
5066     buf[i++] = '-';
5067
5068   while (--i >= 0)
5069     consputc(buf[i]);
5070 }
5071
5072
5073
5074
5075
5076
5077
5078
5079
5080
5081
5082
5083
5084
5085
5086
5087
5088
5089
5090
5091
5092
5093
5094
5095
5096
5097
5098
5099
```

```
5100 // Print to the console. only understands %d, %x, %p, %s.
5101   void
5102 cprintf(char *fmt, ...)
5103 {
5104   va_list ap;
5105   int i, c, locking;
5106   char *s;
5107
5108   va_start(ap, fmt);
5109
5110   locking = cons.locking;
5111   if (locking)
5112     acquire(&cons.lock);
5113
5114   if (fmt == 0)
5115     panic("null fmt");
5116
5117   for (i = 0; (c = fmt[i] & 0xff) != 0; i++) {
5118     if (c != '%') {
5119       consputc(c);
5120       continue;
5121     }
5122     c = fmt[++i] & 0xff;
5123     if (c == 0)
5124       break;
5125     switch(c) {
5126     case 'd':
5127       print_d(va_arg(ap, int));
5128       break;
5129     case 'x':
5130       print_x32(va_arg(ap, uint));
5131       break;
5132     case 'p':
5133       print_x64(va_arg(ap, addr_t));
5134       break;
5135     case 's':
5136       if ((s = va_arg(ap, char*)) == 0)
5137         s = "(null)";
5138       while (*s)
5139         consputc(*(s++));
5140       break;
5141     case '%':
5142       consputc('%');
5143       break;
5144     default:
5145       // Print unknown % sequence to draw attention.
5146       consputc('%');
5147       consputc(c);
5148       break;
5149     }
```



```
5150   }
5151
5152   if (locking)
5153     release(&cons.lock);
5154 }
5155
5156 __attribute__((noreturn))
5157   void
5158 panic(char *s)
5159 {
5160   int i;
5161   addr_t pcs[10];
5162
5163   cli();
5164   cons.locking = 0;
5165   cprintf("cpu%d: panic: ", cpu->id);
5166   cprintf(s);
5167   cprintf("\n");
5168   getcallerpcs(&s, pcs);
5169   for (i=0; i<10; i++)
5170     cprintf(" %p\n", pcs[i]);
5171   panicked = 1; // freeze other CPU
5172   for (;;)
5173     hlt();
5174 }
5175
5176
5177
5178
5179
5180
5181
5182
5183
5184
5185
5186
5187
5188
5189
5190
5191
5192
5193
5194
5195
5196
5197
5198
5199
```

```
5200 #define BACKSPACE 0x100
5201 #define CRTPORT 0x3d4
5202 static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory
5203
5204   static void
5205 cgaputc(int c)
5206 {
5207   int pos;
5208
5209   // Cursor position: col + 80*row.
5210   outb(CRTPORT, 14);
5211   pos = inb(CRTPORT+1) << 8;
5212   outb(CRTPORT, 15);
5213   pos |= inb(CRTPORT+1);
5214
5215   if (c == '\n')
5216     pos += 80 - pos%80;
5217   else if (c == BACKSPACE) {
5218     if (pos > 0) --pos;
5219   } else
5220     crt[pos++] = (c&0xff) | 0x0700;  // gray on black
5221
5222   if ((pos/80) >= 24){  // Scroll up.
5223     memmove(crt, crt+80, sizeof(crt[0])*23*80);
5224     pos -= 80;
5225     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
5226   }
5227
5228   outb(CRTPORT, 14);
5229   outb(CRTPORT+1, pos>>8);
5230   outb(CRTPORT, 15);
5231   outb(CRTPORT+1, pos);
5232   crt[pos] = ' ' | 0x0700;
5233 }
5234
5235   void
5236 consputc(int c)
5237 {
5238   if (panicked) {
5239     cli();
5240     for(;;)
5241       hlt();
5242   }
5243
5244   if (c == BACKSPACE) {
5245     uartputc('\b'); uartputc(' '); uartputc('\b');
5246   } else
5247     uartputc(c);
5248   cgaputc(c);
5249 }
```



```
5250 #define INPUT_BUF 128
5251 struct {
5252   struct spinlock lock;
5253   char buf[INPUT_BUF];
5254   uint r;  // Read index
5255   uint w;  // Write index
5256   uint e;  // Edit index
5257 } input;
5258
5259 #define C(x)  ((x)-'@')  // Control-x
5260
5261   void
5262 consoleintr(int (*getc)(void))
5263 {
5264   int c;
5265
5266   acquire(&input.lock);
5267   while((c = getc()) >= 0){
5268     switch(c){
5269     case C('Z'): // reboot
5270       lidt(0,0);
5271       break;
5272     case C('P'):  // Process listing.
5273       procdump();
5274       break;
5275     case C('U'):  // Kill line.
5276       while(input.e != input.w &&
5277           input.buf[(input.e-1) % INPUT_BUF] != '\n'){
5278         input.e--;
5279         consputc(BACKSPACE);
5280       }
5281       break;
5282     case C('H'): case '\x7f':  // Backspace
5283       if (input.e != input.w) {
5284         input.e--;
5285         consputc(BACKSPACE);
5286       }
5287       break;
5288     default:
5289       if (c != 0 && input.e-input.r < INPUT_BUF) {
5290         c = (c == '\r') ? '\n' : c;
5291         input.buf[input.e++ % INPUT_BUF] = c;
5292         consputc(c);
5293         if (c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF) {
5294           input.w = input.e;
5295           wakeup(&input.r);
5296         }
5297       }
5298       break;
5299     }
```

```
5300   }
5301   release(&input.lock);
5302 }
5303
5304   int
5305 consoleread(struct inode *ip, uint off, char *dst, int n)
5306 {
5307   uint target;
5308   int c;
5309
5310   iunlock(ip);
5311   target = n;
5312   acquire(&input.lock);
5313   while(n > 0){
5314     while(input.r == input.w){
5315       if (proc->killed) {
5316         release(&input.lock);
5317         ilock(ip);
5318         return -1;
5319       }
5320       sleep(&input.r, &input.lock);
5321     }
5322     c = input.buf[input.r++ % INPUT_BUF];
5323     if (c == C('D')) {  // EOF
5324       if (n < target) {
5325         // Save ^D for next time, to make sure
5326         // caller gets a 0-byte result.
5327         input.r--;
5328       }
5329       break;
5330     }
5331     *dst++ = c;
5332     --n;
5333     if (c == '\n')
5334       break;
5335   }
5336   release(&input.lock);
5337   ilock(ip);
5338
5339   return target - n;
5340 }
5341
5342
5343
5344
5345
5346
5347
5348
5349
```



```
5350   int
5351 consolewrite(struct inode *ip, uint off, char *buf, int n)
5352 {
5353   int i;
5354
5355   iunlock(ip);
5356   acquire(&cons.lock);
5357   for(i = 0; i < n; i++)
5358     consputc(buf[i] & 0xff);
5359   release(&cons.lock);
5360   ilock(ip);
5361
5362   return n;
5363 }
5364
5365   void
5366 consoleinit(void)
5367 {
5368   initlock(&cons.lock, "console");
5369   initlock(&input.lock, "input");
5370
5371   devsw[CONSOLE].write = consolewrite;
5372   devsw[CONSOLE].read = consoleread;
5373   cons.locking = 1;
5374
5375   ioapicenable(IRQ_KBD, 0);
5376 }
5377
5378
5379
5380
5381
5382
5383
5384
5385
5386
5387
5388
5389
5390
5391
5392
5393
5394
5395
5396
5397
5398
5399
```

```
5400 // Intel 8250 serial port (UART).
5401
5402 #include "types.h"
5403 #include "defs.h"
5404 #include "param.h"
5405 #include "traps.h"
5406 #include "spinlock.h"
5407 #include "sleeplock.h"
5408 #include "fs.h"
5409 #include "file.h"
5410 #include "mmu.h"
5411 #include "proc.h"
5412 #include "x86.h"
5413
5414 #define COM1    0x3f8
5415
5416 static int uart;    // is there a uart?
5417
5418 void
5419 uartearlyinit(void)
5420 {
5421   char *p;
5422
5423   // Turn off the FIFO
5424   outb(COM1+2, 0);
5425
5426   // 9600 baud, 8 data bits, 1 stop bit, parity off.
5427   outb(COM1+3, 0x80);    // Unlock divisor
5428   outb(COM1+0, 115200/9600);
5429   outb(COM1+1, 0);
5430   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
5431   outb(COM1+4, 0);
5432   outb(COM1+1, 0x01);    // Enable receive interrupts.
5433
5434   // If status is 0xFF, no serial port.
5435   if(inb(COM1+5) == 0xFF)
5436     return;
5437   uart = 1;
5438
5439
5440
5441   // Announce that we're here.
5442   for(p="xv6...\n"; *p; p++)
5443     uartputc(*p);
5444 }
5445
5446
5447
5448
5449
```



```
5450 void
5451 uartinit(void)
5452 {
5453   if(!uart)
5454     return;
5455
5456   // Acknowledge pre-existing interrupt conditions;
5457   // enable interrupts.
5458   inb(COM1+2);
5459   inb(COM1+0);
5460   ioapicenable(IRQ_COM1, 0);
5461
5462 }
5463 void
5464 uartputc(int c)
5465 {
5466   int i;
5467
5468   if(!uart)
5469     return;
5470   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
5471     microdelay(10);
5472   outb(COM1+0, c);
5473 }
5474
5475 static int
5476 uartgetc(void)
5477 {
5478   if(!uart)
5479     return -1;
5480   if(!(inb(COM1+5) & 0x01))
5481     return -1;
5482   return inb(COM1+0);
5483 }
5484
5485 void
5486 uartintr(void)
5487 {
5488   consoleintr(uartgetc);
5489 }
5490
5491
5492
5493
5494
5495
5496
5497
5498
5499
```

```
5500 struct buf {
5501   int flags;
5502   uint dev;
5503   uint blockno;
5504   struct sleeplock lock;
5505   uint refcnt;
5506   struct buf *prev; // LRU cache list
5507   struct buf *next;
5508   struct buf *qnext; // disk queue
5509   uchar data[BSIZE];
5510 };
5511 #define B_VALID 0x2  // buffer has been read from disk
5512 #define B_DIRTY 0x4  // buffer needs to be written to disk
5513
5514
5515
5516
5517
5518
5519
5520
5521
5522
5523
5524
5525
5526
5527
5528
5529
5530
5531
5532
5533
5534
5535
5536
5537
5538
5539
5540
5541
5542
5543
5544
5545
5546
5547
5548
5549
```



```
5550 #pragma once
5551 #define O_RDONLY  0x000
5552 #define O_WRONLY  0x001
5553 #define O_RDWR    0x002
5554 #define O_CREATE  0x200
5555
5556
5557
5558
5559
5560
5561
5562
5563
5564
5565
5566
5567
5568
5569
5570
5571
5572
5573
5574
5575
5576
5577
5578
5579
5580
5581
5582
5583
5584
5585
5586
5587
5588
5589
5590
5591
5592
5593
5594
5595
5596
5597
5598
5599
```

```
5600 #pragma once
5601 #define T_DIR  1   // Directory
5602 #define T_FILE 2   // File
5603 #define T_DEV  3   // Device
5604
5605 struct stat {
5606   short type;  // Type of file
5607   int dev;     // File system's disk device
5608   uint ino;    // Inode number
5609   short nlink; // Number of links to file
5610   uint size;   // Size of file in bytes
5611 };
5612
5613
5614
5615
5616
5617
5618
5619
5620
5621
5622
5623
5624
5625
5626
5627
5628
5629
5630
5631
5632
5633
5634
5635
5636
5637
5638
5639
5640
5641
5642
5643
5644
5645
5646
5647
5648
5649
```



```
5650 // Simple PIO-based (non-DMA) IDE driver code.
5651
5652 #include "types.h"
5653 #include "defs.h"
5654 #include "param.h"
5655 #include "memlayout.h"
5656 #include "mmu.h"
5657 #include "proc.h"
5658 #include "x86.h"
5659 #include "traps.h"
5660 #include "spinlock.h"
5661 #include "sleeplock.h"
5662 #include "fs.h"
5663 #include "buf.h"
5664
5665 #define SECTOR_SIZE   512
5666 #define IDE_BSY       0x80
5667 #define IDE_DRDY      0x40
5668 #define IDE_DF        0x20
5669 #define IDE_ERR       0x01
5670
5671 #define IDE_CMD_READ  0x20
5672 #define IDE_CMD_WRITE 0x30
5673 #define IDE_CMD_RDMUL 0xc4
5674 #define IDE_CMD_WRMUL 0xc5
5675
5676 // idequeue points to the buf now being read/written to the disk.
5677 // idequeue->qnext points to the next buf to be processed.
5678 // You must hold idelock while manipulating queue.
5679
5680 static struct spinlock idelock;
5681 static struct buf *idequeue;
5682
5683 static int havedisk1;
5684 static void idestart(struct buf*);
5685
5686 // Wait for IDE disk to become ready.
5687 static int
5688 idewait(int checkerr)
5689 {
5690   int r;
5691
5692   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
5693     ;
5694   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
5695     return -1;
5696   return 0;
5697 }
5698
5699
```

```
5700 void
5701 ideinit(void)
5702 {
5703   initlock(&idelock, "ide");
5704   ioapicenable(IRQ_IDE, ncpu - 1);
5705   idewait(0);
5706
5707   // Check if disk 1 is present
5708   outb(0x1f6, 0xe0 | (1<<4));
5709   for(int i=0; i<1000; i++){
5710     if(inb(0x1f7) != 0){
5711       havedisk1 = 1;
5712       break;
5713     }
5714   }
5715
5716   // Switch back to disk 0.
5717   outb(0x1f6, 0xe0 | (0<<4));
5718 }
5719
5720 // Start the request for b.  Caller must hold idelock.
5721 static void
5722 idestart(struct buf *b)
5723 {
5724   if(b == 0)
5725     panic("idestart");
5726   if(b->blockno >= FSSIZE)
5727     panic("incorrect blockno");
5728   int sector_per_block =  BSIZE/SECTOR_SIZE;
5729   int sector = b->blockno * sector_per_block;
5730   int read_cmd = (sector_per_block == 1) ? IDE_CMD_READ :  IDE_CMD_RDMUL;
5731   int write_cmd = (sector_per_block == 1) ? IDE_CMD_WRITE : IDE_CMD_WRMUL;
5732
5733   if (sector_per_block > 7) panic("idestart");
5734
5735   idewait(0);
5736   outb(0x3f6, 0);  // generate interrupt
5737   outb(0x1f2, sector_per_block);  // number of sectors
5738   outb(0x1f3, sector & 0xff);
5739   outb(0x1f4, (sector >> 8) & 0xff);
5740   outb(0x1f5, (sector >> 16) & 0xff);
5741   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
5742   if(b->flags & B_DIRTY){
5743     outb(0x1f7, write_cmd);
5744     outsl(0x1f0, b->data, BSIZE/4);
5745   } else {
5746     outb(0x1f7, read_cmd);
5747   }
5748 }
5749
```



```
5750 // Interrupt handler.
5751 void
5752 ideintr(void)
5753 {
5754   struct buf *b;
5755
5756   // First queued buffer is the active request.
5757   acquire(&idelock);
5758   if((b = idequeue) == 0){
5759     release(&idelock);
5760     // cprintf("spurious IDE interrupt\n");
5761     return;
5762   }
5763   idequeue = b->qnext;
5764
5765   // Read data if needed.
5766   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
5767     insl(0x1f0, b->data, BSIZE/4);
5768
5769   // Wake process waiting for this buf.
5770   b->flags |= B_VALID;
5771   b->flags &= ~B_DIRTY;
5772   wakeup(b);
5773
5774   // Start disk on next buf in queue.
5775   if(idequeue != 0)
5776     idestart(idequeue);
5777
5778   release(&idelock);
5779 }
5780
5781
5782
5783
5784
5785
5786
5787
5788
5789
5790
5791
5792
5793
5794
5795
5796
5797
5798
5799
```

```
5800 // Sync buf with disk.
5801 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
5802 // Else if B_VALID is not set, read buf from disk, set B_VALID.
5803 void
5804 iderw(struct buf *b)
5805 {
5806   struct buf **pp;
5807
5808   if(!holdingsleep(&b->lock))
5809     panic("iderw: buf not locked");
5810   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
5811     panic("iderw: nothing to do");
5812   if(b->dev != 0 && !havedisk1)
5813     panic("iderw: ide disk 1 not present");
5814
5815   acquire(&idelock);
5816
5817   // Append b to idequeue.
5818   b->qnext = 0;
5819   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
5820     ;
5821   *pp = b;
5822
5823   // Start disk if necessary.
5824   if(idequeue == b)
5825     idestart(b);
5826
5827   // Wait for request to finish.
5828   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
5829     sleep(b, &idelock);
5830   }
5831
5832   release(&idelock);
5833 }
5834
5835
5836
5837
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```



```
5850 // Buffer cache.
5851 //
5852 // The buffer cache is a linked list of buf structures holding
5853 // cached copies of disk block contents.  Caching disk blocks
5854 // in memory reduces the number of disk reads and also provides
5855 // a synchronization point for disk blocks used by multiple processes.
5856 //
5857 // Interface:
5858 // * To get a buffer for a particular disk block, call bread.
5859 // * After changing buffer data, call bwrite to write it to disk.
5860 // * When done with the buffer, call brelse.
5861 // * Do not use the buffer after calling brelse.
5862 // * Only one process at a time can use a buffer,
5863 //     so do not keep them longer than necessary.
5864 //
5865 // The implementation uses two state flags internally:
5866 // * B_VALID: the buffer data has been read from the disk.
5867 // * B_DIRTY: the buffer data has been modified
5868 //     and needs to be written to disk.
5869
5870 #include "types.h"
5871 #include "defs.h"
5872 #include "param.h"
5873 #include "spinlock.h"
5874 #include "sleeplock.h"
5875 #include "fs.h"
5876 #include "buf.h"
5877
5878 struct {
5879   struct spinlock lock;
5880   struct buf buf[NBUF];
5881
5882   // Linked list of all buffers, through prev/next.
5883   // head.next is most recently used.
5884   struct buf head;
5885 } bcache;
5886
5887 void
5888 binit(void)
5889 {
5890   struct buf *b;
5891
5892   initlock(&bcache.lock, "bcache");
5893
5894
5895
5896
5897
5898
5899
```

```
5900   // Create linked list of buffers
5901   bcache.head.prev = &bcache.head;
5902   bcache.head.next = &bcache.head;
5903   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
5904     b->next = bcache.head.next;
5905     b->prev = &bcache.head;
5906     initsleeplock(&b->lock, "buffer");
5907     bcache.head.next->prev = b;
5908     bcache.head.next = b;
5909   }
5910 }
5911
5912 // Look through buffer cache for block on device dev.
5913 // If not found, allocate a buffer.
5914 // In either case, return locked buffer.
5915 static struct buf*
5916 bget(uint dev, uint blockno)
5917 {
5918   struct buf *b;
5919
5920   acquire(&bcache.lock);
5921
5922   // Is the block already cached?
5923   for(b = bcache.head.next; b != &bcache.head; b = b->next){
5924     if(b->dev == dev && b->blockno == blockno){
5925       b->refcnt++;
5926       release(&bcache.lock);
5927       acquiresleep(&b->lock);
5928       return b;
5929     }
5930   }
5931
5932   // Not cached; recycle some unused buffer and clean buffer
5933   // "clean" because B_DIRTY and not locked means log.c
5934   // hasn't yet committed the changes to the buffer.
5935   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
5936     if(b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
5937       b->dev = dev;
5938       b->blockno = blockno;
5939       b->flags = 0;
5940       b->refcnt = 1;
5941       release(&bcache.lock);
5942       acquiresleep(&b->lock);
5943       return b;
5944     }
5945   }
5946   panic("bget: no buffers");
5947 }
5948
5949
```



```
5950 // Return a locked buf with the contents of the indicated block.
5951 struct buf*
5952 bread(uint dev, uint blockno)
5953 {
5954   struct buf *b;
5955
5956   b = bget(dev, blockno);
5957   if(!(b->flags & B_VALID)) {
5958     iderw(b);
5959   }
5960   return b;
5961 }
5962
5963 // Write b's contents to disk.  Must be locked.
5964 void
5965 bwrite(struct buf *b)
5966 {
5967   if(!holdingsleep(&b->lock))
5968     panic("bwrite");
5969   b->flags |= B_DIRTY;
5970   iderw(b);
5971 }
5972
5973 // Release a locked buffer.
5974 // Move to the head of the MRU list.
5975 void
5976 brelse(struct buf *b)
5977 {
5978   if(!holdingsleep(&b->lock))
5979     panic("brelse");
5980
5981   releasesleep(&b->lock);
5982
5983   acquire(&bcache.lock);
5984   b->refcnt--;
5985   if (b->refcnt == 0) {
5986     // no one is waiting for it.
5987     b->next->prev = b->prev;
5988     b->prev->next = b->next;
5989     b->next = bcache.head.next;
5990     b->prev = &bcache.head;
5991     bcache.head.next->prev = b;
5992     bcache.head.next = b;
5993   }
5994
5995   release(&bcache.lock);
5996 }
5997
5998
5999
```

```
6000 // Long-term locks for processes
6001 struct sleeplock {
6002   uint locked;       // Is the lock held?
6003   struct spinlock lk; // spinlock protecting this sleep lock
6004
6005   // For debugging:
6006   char *name;        // Name of lock.
6007   int pid;           // Process holding lock
6008 };
6009
6010
6011
6012
6013
6014
6015
6016
6017
6018
6019
6020
6021
6022
6023
6024
6025
6026
6027
6028
6029
6030
6031
6032
6033
6034
6035
6036
6037
6038
6039
6040
6041
6042
6043
6044
6045
6046
6047
6048
6049
```



```
6050 // Sleeping locks
6051
6052 #include "types.h"
6053 #include "defs.h"
6054 #include "param.h"
6055 #include "x86.h"
6056 #include "memlayout.h"
6057 #include "mmu.h"
6058 #include "proc.h"
6059 #include "spinlock.h"
6060 #include "sleeplock.h"
6061
6062 void
6063 initsleeplock(struct sleeplock *lk, char *name)
6064 {
6065   initlock(&lk->lk, "sleep lock");
6066   lk->name = name;
6067   lk->locked = 0;
6068   lk->pid = 0;
6069 }
6070
6071 void
6072 acquiresleep(struct sleeplock *lk)
6073 {
6074   acquire(&lk->lk);
6075   while (lk->locked)
6076     sleep(lk, &lk->lk);
6077   lk->locked = 1;
6078   lk->pid = proc->pid;
6079   release(&lk->lk);
6080 }
6081
6082 void
6083 releasesleep(struct sleeplock *lk)
6084 {
6085   acquire(&lk->lk);
6086   lk->locked = 0;
6087   lk->pid = 0;
6088   wakeup(lk);
6089   release(&lk->lk);
6090 }
6091
6092 int
6093 holdingsleep(struct sleeplock *lk)
6094 {
6095   acquire(&lk->lk);
6096   int r = lk->locked;
6097   release(&lk->lk);
6098   return r;
6099 }
```

```
6100 #include "types.h"
6101 #include "defs.h"
6102 #include "param.h"
6103 #include "spinlock.h"
6104 #include "sleeplock.h"
6105 #include "fs.h"
6106 #include "buf.h"
6107
6108 // Simple logging that allows concurrent FS system calls.
6109 //
6110 // A log transaction contains the updates of multiple FS system
6111 // calls. The logging system only commits when there are
6112 // no FS system calls active. Thus there is never
6113 // any reasoning required about whether a commit might
6114 // write an uncommitted system call's updates to disk.
6115 //
6116 // A system call should call begin_op()/end_op() to mark
6117 // its start and end. Usually begin_op() just increments
6118 // the count of in-progress FS system calls and returns.
6119 // But if it thinks the log is close to running out, it
6120 // sleeps until the last outstanding end_op() commits.
6121 //
6122 // The log is a physical re-do log containing disk blocks.
6123 // The on-disk log format:
6124 //   header block, containing block #s for block A, B, C, ...
6125 //   block A
6126 //   block B
6127 //   block C
6128 //   ...
6129 // Log appends are synchronous.
6130
6131 // Contents of the header block, used for both the on-disk header block
6132 // and to keep track in memory of logged block# before commit.
6133 struct logheader {
6134   int n;
6135   int block[LOGSIZE];
6136 };
6137
6138 struct log {
6139   struct spinlock lock;
6140   int start;
6141   int size;
6142   int outstanding; // how many FS sys calls are executing.
6143   int committing;  // in commit(), please wait.
6144   int dev;
6145   struct logheader lh;
6146 };
6147
6148
6149
```



```
6150 struct log log;
6151
6152 static void recover_from_log(void);
6153 static void commit();
6154
6155 void
6156 initlog(int dev)
6157 {
6158   if (sizeof(struct logheader) >= BSIZE)
6159     panic("initlog: too big logheader");
6160
6161   struct superblock sb;
6162   initlock(&log.lock, "log");
6163   readsb(dev, &sb);
6164   log.start = sb.logstart;
6165   log.size = sb.nlog;
6166   log.dev = dev;
6167   recover_from_log();
6168 }
6169
6170 // Copy committed blocks from log to their home location
6171 static void
6172 install_trans(void)
6173 {
6174   int tail;
6175
6176   for (tail = 0; tail < log.lh.n; tail++) {
6177     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
6178     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
6179     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
6180     bwrite(dbuf);  // write dst to disk
6181     brelse(lbuf);
6182     brelse(dbuf);
6183   }
6184 }
6185
6186 // Read the log header from disk into the in-memory log header
6187 static void
6188 read_head(void)
6189 {
6190   struct buf *buf = bread(log.dev, log.start);
6191   struct logheader *lh = (struct logheader *) (buf->data);
6192   int i;
6193   log.lh.n = lh->n;
6194   for (i = 0; i < log.lh.n; i++) {
6195     log.lh.block[i] = lh->block[i];
6196   }
6197   brelse(buf);
6198 }
6199
```

```
6200 // Write in-memory log header to disk.
6201 // This is the true point at which the
6202 // current transaction commits.
6203 static void
6204 write_head(void)
6205 {
6206   struct buf *buf = bread(log.dev, log.start);
6207   struct logheader *hb = (struct logheader *) (buf->data);
6208   int i;
6209   hb->n = log.lh.n;
6210   for (i = 0; i < log.lh.n; i++) {
6211     hb->block[i] = log.lh.block[i];
6212   }
6213   bwrite(buf);
6214   brelse(buf);
6215 }
6216
6217 static void
6218 recover_from_log(void)
6219 {
6220   read_head();
6221   install_trans(); // if committed, copy from log to disk
6222   log.lh.n = 0;
6223   write_head(); // clear the log
6224 }
6225
6226 // called at the start of each FS system call.
6227 void
6228 begin_op(void)
6229 {
6230   acquire(&log.lock);
6231   while(1){
6232     if(log.committing){
6233       sleep(&log, &log.lock);
6234     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
6235       // this op might exhaust log space; wait for commit.
6236       sleep(&log, &log.lock);
6237     } else {
6238       log.outstanding += 1;
6239       release(&log.lock);
6240       break;
6241     }
6242   }
6243 }
6244
6245
6246
6247
6248
6249
```



```
6250 // called at the end of each FS system call.
6251 // commits if this was the last outstanding operation.
6252 void
6253 end_op(void)
6254 {
6255   int do_commit = 0;
6256
6257   acquire(&log.lock);
6258   log.outstanding -= 1;
6259   if(log.committing)
6260     panic("log.committing");
6261   if(log.outstanding == 0){
6262     do_commit = 1;
6263     log.committing = 1;
6264   } else {
6265     // begin_op() may be waiting for log space.
6266     wakeup(&log);
6267   }
6268   release(&log.lock);
6269
6270   if(do_commit){
6271     // call commit w/o holding locks, since not allowed
6272     // to sleep with locks.
6273     commit();
6274     acquire(&log.lock);
6275     log.committing = 0;
6276     wakeup(&log);
6277     release(&log.lock);
6278   }
6279 }
6280
6281 // Copy modified blocks from cache to log.
6282 static void
6283 write_log(void)
6284 {
6285   int tail;
6286
6287   for (tail = 0; tail < log.lh.n; tail++) {
6288     struct buf *to = bread(log.dev, log.start+tail+1); // log block
6289     struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
6290     memmove(to->data, from->data, BSIZE);
6291     bwrite(to);  // write the log
6292     brelse(from);
6293     brelse(to);
6294   }
6295 }
6296
6297
6298
6299
```

```
6300 static void
6301 commit()
6302 {
6303   if (log.lh.n > 0) {
6304     write_log();     // Write modified blocks from cache to log
6305     write_head();    // Write header to disk -- the real commit
6306     install_trans(); // Now install writes to home locations
6307     log.lh.n = 0;
6308     write_head();    // Erase the transaction from the log
6309   }
6310 }
6311
6312 // Caller has modified b->data and is done with the buffer.
6313 // Record the block number and pin in the cache with B_DIRTY.
6314 // commit()/write_log() will do the disk write.
6315 //
6316 // log_write() replaces bwrite(); a typical use is:
6317 //   bp = bread(...)
6318 //   modify bp->data[]
6319 //   log_write(bp)
6320 //   brelse(bp)
6321 void
6322 log_write(struct buf *b)
6323 {
6324   int i;
6325
6326   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
6327     panic("too big a transaction");
6328   if (log.outstanding < 1)
6329     panic("log_write outside of trans");
6330
6331   acquire(&log.lock);
6332   for (i = 0; i < log.lh.n; i++) {
6333     if (log.lh.block[i] == b->blockno)   // log absorbtion
6334       break;
6335   }
6336   log.lh.block[i] = b->blockno;
6337   if (i == log.lh.n)
6338     log.lh.n++;
6339   b->flags |= B_DIRTY; // prevent eviction
6340   release(&log.lock);
6341 }
6342
6343
6344
6345
6346
6347
6348
6349
```



```
6350 // On-disk file system format.
6351 // Both the kernel and user programs use this header file.
6352
6353
6354 #define ROOTINO 1  // root i-number
6355 #define BSIZE 512  // block size
6356
6357 // Disk layout:
6358 // [ boot block | super block | log | inode blocks |
6359 //                                          free bit map | data blocks]
6360 //
6361 // mkfs computes the super block and builds an initial file system. The
6362 // super block describes the disk layout:
6363 struct superblock {
6364   uint size;         // Size of file system image (blocks)
6365   uint nblocks;      // Number of data blocks
6366   uint ninodes;      // Number of inodes.
6367   uint nlog;         // Number of log blocks
6368   uint logstart;     // Block number of first log block
6369   uint inodestart;   // Block number of first inode block
6370   uint bmapstart;    // Block number of first free map block
6371 };
6372
6373 #define NDIRECT 12
6374 #define NINDIRECT (BSIZE / sizeof(uint))
6375 #define MAXFILE (NDIRECT + NINDIRECT)
6376
6377 // On-disk inode structure
6378 struct dinode {
6379   short type;           // File type
6380   short major;          // Major device number (T_DEV only)
6381   short minor;          // Minor device number (T_DEV only)
6382   short nlink;          // Number of links to inode in file system
6383   uint size;            // Size of file (bytes)
6384   uint addrs[NDIRECT+1];   // Data block addresses
6385 };
6386
6387
6388
6389
6390
6391
6392
6393
6394
6395
6396
6397
6398
6399
```

```
6400 // Inodes per block.
6401 #define IPB           (BSIZE / sizeof(struct dinode))
6402
6403 // Block containing inode i
6404 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
6405
6406 // Bitmap bits per block
6407 #define BPB           (BSIZE*8)
6408
6409 // Block of free map containing bit for block b
6410 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
6411
6412 // Directory is a file containing a sequence of dirent structures.
6413 #define DIRSIZ 14
6414
6415 struct dirent {
6416   ushort inum;
6417   char name[DIRSIZ];
6418 };
6419
6420
6421
6422
6423
6424
6425
6426
6427
6428
6429
6430
6431
6432
6433
6434
6435
6436
6437
6438
6439
6440
6441
6442
6443
6444
6445
6446
6447
6448
6449
```



```
6450 // File system implementation.  Five layers:
6451 //   + Blocks: allocator for raw disk blocks.
6452 //   + Log: crash recovery for multi-step updates.
6453 //   + Files: inode allocator, reading, writing, metadata.
6454 //   + Directories: inode with special contents (list of other inodes!)
6455 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
6456 //
6457 // This file contains the low-level file system manipulation
6458 // routines.  The (higher-level) system call implementations
6459 // are in sysfile.c.
6460
6461 #include "types.h"
6462 #include "defs.h"
6463 #include "param.h"
6464 #include "stat.h"
6465 #include "mmu.h"
6466 #include "proc.h"
6467 #include "spinlock.h"
6468 #include "sleeplock.h"
6469 #include "fs.h"
6470 #include "buf.h"
6471 #include "file.h"
6472
6473 #define min(a, b) ((a) < (b) ? (a) : (b))
6474 static void itrunc(struct inode*);
6475 // there should be one superblock per disk device, but we run with
6476 // only one device
6477 struct superblock sb;
6478
6479 // Read the super block.
6480 void
6481 readsb(int dev, struct superblock *sb)
6482 {
6483   struct buf *bp = bread(dev, 1);
6484   memmove(sb, bp->data, sizeof(*sb));
6485   brelse(bp);
6486 }
6487
6488 // Zero a block.
6489 static void
6490 bzero(int dev, int bno)
6491 {
6492   struct buf *bp = bread(dev, bno);
6493   memset(bp->data, 0, BSIZE);
6494   log_write(bp);
6495   brelse(bp);
6496 }
6497
6498
6499
```

```
6500 // Blocks.
6501
6502 // Allocate a zeroed disk block.
6503 static uint
6504 balloc(uint dev)
6505 {
6506   int b, bi, m;
6507   struct buf *bp;
6508   for(b = 0; b < sb.size; b += BPB){
6509     bp = bread(dev, BBLOCK(b, sb));
6510     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
6511       m = 1 << (bi % 8);
6512       if((bp->data[bi/8] & m) == 0){  // Is block free?
6513         bp->data[bi/8] |= m;  // Mark block in use.
6514         log_write(bp);
6515         brelse(bp);
6516         bzero(dev, b + bi);
6517         return b + bi;
6518       }
6519     }
6520     brelse(bp);
6521   }
6522   panic("balloc: out of blocks");
6523 }
6524
6525 // Free a disk block.
6526 static void
6527 bfree(int dev, uint b)
6528 {
6529   int bi, m;
6530
6531   readsb(dev, &sb);
6532   struct buf *bp = bread(dev, BBLOCK(b, sb));
6533   bi = b % BPB;
6534   m = 1 << (bi % 8);
6535   if((bp->data[bi/8] & m) == 0)
6536     panic("freeing free block");
6537   bp->data[bi/8] &= ~m;
6538   log_write(bp);
6539   brelse(bp);
6540 }
6541
6542
6543
6544
6545
6546
6547
6548
6549
```



```
6550 // Inodes.
6551 //
6552 // An inode describes a single unnamed file.
6553 // The inode disk structure holds metadata: the file's type,
6554 // its size, the number of links referring to it, and the
6555 // list of blocks holding the file's content.
6556 //
6557 // The inodes are laid out sequentially on disk at
6558 // sb.startinode. Each inode has a number, indicating its
6559 // position on the disk.
6560 //
6561 // The kernel keeps a cache of in-use inodes in memory
6562 // to provide a place for synchronizing access
6563 // to inodes used by multiple processes. The cached
6564 // inodes include book-keeping information that is
6565 // not stored on disk: ip->ref and ip->flags.
6566 //
6567 // An inode and its in-memory represtative go through a
6568 // sequence of states before they can be used by the
6569 // rest of the file system code.
6570 //
6571 // * Allocation: an inode is allocated if its type (on disk)
6572 //   is non-zero. ialloc() allocates, iput() frees if
6573 //   the link count has fallen to zero.
6574 //
6575 // * Referencing in cache: an entry in the inode cache
6576 //   is free if ip->ref is zero. Otherwise ip->ref tracks
6577 //   the number of in-memory pointers to the entry (open
6578 //   files and current directories). iget() to find or
6579 //   create a cache entry and increment its ref, iput()
6580 //   to decrement ref.
6581 //
6582 // * Valid: the information (type, size, &c) in an inode
6583 //   cache entry is only correct when the I_VALID bit
6584 //   is set in ip->flags. ilock() reads the inode from
6585 //   the disk and sets I_VALID, while iput() clears
6586 //   I_VALID if ip->ref has fallen to zero.
6587 //
6588 // * Locked: file system code may only examine and modify
6589 //   the information in an inode and its content if it
6590 //   has first locked the inode.
6591 //
6592 // Thus a typical sequence is:
6593 //   ip = iget(dev, inum)
6594 //   ilock(ip)
6595 //   ... examine and modify ip->xxx ...
6596 //   iunlock(ip)
6597 //   iput(ip)
6598 //
6599 // ilock() is separate from iget() so that system calls can
```

```
6600 // get a long-term reference to an inode (as for an open file)
6601 // and only lock it for short periods (e.g., in read()).
6602 // The separation also helps avoid deadlock and races during
6603 // pathname lookup. iget() increments ip->ref so that the inode
6604 // stays cached and pointers to it remain valid.
6605 //
6606 // Many internal file system functions expect the caller to
6607 // have locked the inodes involved; this lets callers create
6608 // multi-step atomic operations.
6609
6610 struct {
6611   struct spinlock lock;
6612   struct inode inode[NINODE];
6613 } icache;
6614
6615 void
6616 iinit(int dev)
6617 {
6618   int i = 0;
6619
6620   initlock(&icache.lock, "icache");
6621   for(i = 0; i < NINODE; i++) {
6622     initsleeplock(&icache.inode[i].lock, "inode");
6623   }
6624
6625   readsb(dev, &sb);
6626   /*cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d\
6627  inodestart %d bmap start %d\n", sb.size, sb.nblocks,
6628           sb.ninodes, sb.nlog, sb.logstart, sb.inodestart,
6629           sb.bmapstart);*/
6630 }
6631
6632 static struct inode* iget(uint dev, uint inum);
6633
6634
6635 // Allocate a new inode with the given type on device dev.
6636 // A free inode has a type of zero.
6637 struct inode*
6638 ialloc(uint dev, short type)
6639 {
6640   int inum;
6641   struct buf *bp;
6642   struct dinode *dip;
6643
6644   for(inum = 1; inum < sb.ninodes; inum++){
6645     bp = bread(dev, IBLOCK(inum, sb));
6646     dip = (struct dinode*)bp->data + inum%IPB;
6647     if(dip->type == 0){  // a free inode
6648       memset(dip, 0, sizeof(*dip));
6649       dip->type = type;
```



```
6650       log_write(bp);   // mark it allocated on the disk
6651       brelse(bp);
6652       return iget(dev, inum);
6653     }
6654     brelse(bp);
6655   }
6656   panic("ialloc: no inodes");
6657 }
6658
6659 // Copy a modified in-memory inode to disk.
6660 void
6661 iupdate(struct inode *ip)
6662 {
6663   struct buf *bp;
6664   struct dinode *dip;
6665
6666   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
6667   dip = (struct dinode*)bp->data + ip->inum%IPB;
6668   dip->type = ip->type;
6669   dip->major = ip->major;
6670   dip->minor = ip->minor;
6671   dip->nlink = ip->nlink;
6672   dip->size = ip->size;
6673   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
6674   log_write(bp);
6675   brelse(bp);
6676 }
6677
6678 // Find the inode with number inum on device dev
6679 // and return the in-memory copy. Does not lock
6680 // the inode and does not read it from disk.
6681 static struct inode*
6682 iget(uint dev, uint inum)
6683 {
6684   struct inode *ip, *empty;
6685
6686   acquire(&icache.lock);
6687
6688   // Is the inode already cached?
6689   empty = 0;
6690   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
6691     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
6692       ip->ref++;
6693       release(&icache.lock);
6694       return ip;
6695     }
6696     if(empty == 0 && ip->ref == 0) // Remember empty slot.
6697       empty = ip;
6698   }
6699
```

```
6700   // Recycle an inode cache entry.
6701   if(empty == 0)
6702     panic("iget: no inodes");
6703
6704   ip = empty;
6705   ip->dev = dev;
6706   ip->inum = inum;
6707   ip->ref = 1;
6708   ip->flags = 0;
6709   release(&icache.lock);
6710
6711   return ip;
6712 }
6713
6714 // Increment reference count for ip.
6715 // Returns ip to enable ip = idup(ip1) idiom.
6716 struct inode*
6717 idup(struct inode *ip)
6718 {
6719   acquire(&icache.lock);
6720   ip->ref++;
6721   release(&icache.lock);
6722   return ip;
6723 }
6724
6725 // Lock the given inode.
6726 // Reads the inode from disk if necessary.
6727 void
6728 ilock(struct inode *ip)
6729 {
6730   struct buf *bp;
6731   struct dinode *dip;
6732
6733   if(ip == 0 || ip->ref < 1)
6734     panic("ilock");
6735
6736   acquiresleep(&ip->lock);
6737
6738   if(!(ip->flags & I_VALID)){
6739     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
6740     dip = (struct dinode*)bp->data + ip->inum%IPB;
6741     ip->type = dip->type;
6742     ip->major = dip->major;
6743     ip->minor = dip->minor;
6744     ip->nlink = dip->nlink;
6745     ip->size = dip->size;
6746     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
6747     brelse(bp);
6748     ip->flags |= I_VALID;
6749     if(ip->type == 0)
```



```
6750       panic("ilock: no type");
6751   }
6752 }
6753
6754 // Unlock the given inode.
6755 void
6756 iunlock(struct inode *ip)
6757 {
6758   if(ip == 0 || !holdingsleep(&ip->lock) || ip->ref < 1)
6759     panic("iunlock");
6760
6761   releasesleep(&ip->lock);
6762 }
6763
6764 // Drop a reference to an in-memory inode.
6765 // If that was the last reference, the inode cache entry can
6766 // be recycled.
6767 // If that was the last reference and the inode has no links
6768 // to it, free the inode (and its content) on disk.
6769 // All calls to iput() must be inside a transaction in
6770 // case it has to free the inode.
6771 void
6772 iput(struct inode *ip)
6773 {
6774   acquire(&icache.lock);
6775   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
6776     // inode has no links and no other references: truncate and free.
6777     release(&icache.lock);
6778     itrunc(ip);
6779     ip->type = 0;
6780     iupdate(ip);
6781     acquire(&icache.lock);
6782     ip->flags = 0;
6783   }
6784   ip->ref--;
6785   release(&icache.lock);
6786 }
6787
6788 // Common idiom: unlock, then put.
6789 void
6790 iunlockput(struct inode *ip)
6791 {
6792   iunlock(ip);
6793   iput(ip);
6794 }
6795
6796
6797
6798
6799
```

```
6800 // Inode content
6801 //
6802 // The content (data) associated with each inode is stored
6803 // in blocks on the disk. The first NDIRECT block numbers
6804 // are listed in ip->addrs[].  The next NINDIRECT blocks are
6805 // listed in block ip->addrs[NDIRECT].
6806
6807 // Return the disk block address of the nth block in inode ip.
6808 // If there is no such block, bmap allocates one.
6809 static uint
6810 bmap(struct inode *ip, uint bn)
6811 {
6812   uint addr, *a;
6813   struct buf *bp;
6814
6815   if(bn < NDIRECT){
6816     if((addr = ip->addrs[bn]) == 0)
6817       ip->addrs[bn] = addr = balloc(ip->dev);
6818     return addr;
6819   }
6820   bn -= NDIRECT;
6821
6822   if(bn < NINDIRECT){
6823     // Load indirect block, allocating if necessary.
6824     if((addr = ip->addrs[NDIRECT]) == 0)
6825       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
6826     bp = bread(ip->dev, addr);
6827     a = (uint*)bp->data;
6828     if((addr = a[bn]) == 0){
6829       a[bn] = addr = balloc(ip->dev);
6830       log_write(bp);
6831     }
6832     brelse(bp);
6833     return addr;
6834   }
6835
6836   panic("bmap: out of range");
6837 }
6838
6839
6840
6841
6842
6843
6844
6845
6846
6847
6848
6849
```



```
6850 // Truncate inode (discard contents).
6851 // Only called when the inode has no links
6852 // to it (no directory entries referring to it)
6853 // and has no in-memory reference to it (is
6854 // not an open file or current directory).
6855 static void
6856 itrunc(struct inode *ip)
6857 {
6858   int i, j;
6859   struct buf *bp;
6860   uint *a;
6861
6862   for(i = 0; i < NDIRECT; i++){
6863     if(ip->addrs[i]){
6864       bfree(ip->dev, ip->addrs[i]);
6865       ip->addrs[i] = 0;
6866     }
6867   }
6868
6869   if(ip->addrs[NDIRECT]){
6870     bp = bread(ip->dev, ip->addrs[NDIRECT]);
6871     a = (uint*)bp->data;
6872     for(j = 0; j < NINDIRECT; j++){
6873       if(a[j])
6874         bfree(ip->dev, a[j]);
6875     }
6876     brelse(bp);
6877     bfree(ip->dev, ip->addrs[NDIRECT]);
6878     ip->addrs[NDIRECT] = 0;
6879   }
6880
6881   ip->size = 0;
6882   iupdate(ip);
6883 }
6884
6885 // Copy stat information from inode.
6886 void
6887 stati(struct inode *ip, struct stat *st)
6888 {
6889   st->dev = ip->dev;
6890   st->ino = ip->inum;
6891   st->type = ip->type;
6892   st->nlink = ip->nlink;
6893   st->size = ip->size;
6894 }
6895
6896
6897
6898
6899
```

```
6900 // Read data from inode.
6901 int
6902 readi(struct inode *ip, char *dst, uint off, uint n)
6903 {
6904   uint tot, m;
6905   struct buf *bp;
6906
6907   if(ip->type == T_DEV){
6908     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
6909       return -1;
6910     return devsw[ip->major].read(ip, off, dst, n);
6911   }
6912
6913   if(off > ip->size || off + n < off)
6914     return -1;
6915   if(off + n > ip->size)
6916     n = ip->size - off;
6917
6918   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
6919     bp = bread(ip->dev, bmap(ip, off/BSIZE));
6920     m = min(n - tot, BSIZE - off%BSIZE);
6921     memmove(dst, bp->data + off%BSIZE, m);
6922     brelse(bp);
6923   }
6924   return n;
6925 }
6926
6927
6928
6929
6930
6931
6932
6933
6934
6935
6936
6937
6938
6939
6940
6941
6942
6943
6944
6945
6946
6947
6948
6949
```



```
6950 // Write data to inode.
6951 int
6952 writei(struct inode *ip, char *src, uint off, uint n)
6953 {
6954   uint tot, m;
6955   struct buf *bp;
6956
6957   if(ip->type == T_DEV){
6958     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
6959       return -1;
6960     return devsw[ip->major].write(ip, off, src, n);
6961   }
6962
6963   if(off > ip->size || off + n < off)
6964     return -1;
6965   if(off + n > MAXFILE*BSIZE)
6966     return -1;
6967
6968   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
6969     bp = bread(ip->dev, bmap(ip, off/BSIZE));
6970     m = min(n - tot, BSIZE - off%BSIZE);
6971     memmove(bp->data + off%BSIZE, src, m);
6972     log_write(bp);
6973     brelse(bp);
6974   }
6975
6976   if(n > 0 && off > ip->size){
6977     ip->size = off;
6978     iupdate(ip);
6979   }
6980   return n;
6981 }
6982
6983
6984
6985
6986
6987
6988
6989
6990
6991
6992
6993
6994
6995
6996
6997
6998
6999
```

```
7000 // Directories
7001
7002 int
7003 namecmp(const char *s, const char *t)
7004 {
7005   return strncmp(s, t, DIRSIZ);
7006 }
7007
7008 // Look for a directory entry in a directory.
7009 // If found, set *poff to byte offset of entry.
7010 struct inode*
7011 dirlookup(struct inode *dp, char *name, uint *poff)
7012 {
7013   uint off, inum;
7014   struct dirent de;
7015
7016   if(dp->type != T_DIR)
7017     panic("dirlookup not DIR");
7018
7019   for(off = 0; off < dp->size; off += sizeof(de)){
7020     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
7021       panic("dirlookup read");
7022     if(de.inum == 0)
7023       continue;
7024     if(namecmp(name, de.name) == 0){
7025       // entry matches path element
7026       if(poff)
7027         *poff = off;
7028       inum = de.inum;
7029       return iget(dp->dev, inum);
7030     }
7031   }
7032
7033   return 0;
7034 }
7035
7036
7037
7038
7039
7040
7041
7042
7043
7044
7045
7046
7047
7048
7049
```



```
7050 // Write a new directory entry (name, inum) into the directory dp.
7051 int
7052 dirlink(struct inode *dp, char *name, uint inum)
7053 {
7054   int off;
7055   struct dirent de;
7056   struct inode *ip;
7057
7058   // Check that name is not present.
7059   if((ip = dirlookup(dp, name, 0)) != 0){
7060     iput(ip);
7061     return -1;
7062   }
7063
7064   // Look for an empty dirent.
7065   for(off = 0; off < dp->size; off += sizeof(de)){
7066     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
7067       panic("dirlink read");
7068     if(de.inum == 0)
7069       break;
7070   }
7071
7072   strncpy(de.name, name, DIRSIZ);
7073   de.inum = inum;
7074   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
7075     panic("dirlink");
7076
7077   return 0;
7078 }
7079
7080
7081
7082
7083
7084
7085
7086
7087
7088
7089
7090
7091
7092
7093
7094
7095
7096
7097
7098
7099
```

```
7100 // Paths
7101
7102 // Copy the next path element from path into name.
7103 // Return a pointer to the element following the copied one.
7104 // The returned path has no leading slashes,
7105 // so the caller can check *path=='\0' to see if the name is the last one.
7106 // If no name to remove, return 0.
7107 //
7108 // Examples:
7109 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
7110 //   skipelem("///a//bb", name) = "bb", setting name = "a"
7111 //   skipelem("a", name) = "", setting name = "a"
7112 //   skipelem("", name) = skipelem("////", name) = 0
7113 //
7114 static char*
7115 skipelem(char *path, char *name)
7116 {
7117   char *s;
7118   int len;
7119
7120   while(*path == '/')
7121     path++;
7122   if(*path == 0)
7123     return 0;
7124   s = path;
7125   while(*path != '/' && *path != 0)
7126     path++;
7127   len = path - s;
7128   if(len >= DIRSIZ)
7129     memmove(name, s, DIRSIZ);
7130   else {
7131     memmove(name, s, len);
7132     name[len] = 0;
7133   }
7134   while(*path == '/')
7135     path++;
7136   return path;
7137 }
7138
7139
7140
7141
7142
7143
7144
7145
7146
7147
7148
7149
```



```
7150 // Look up and return the inode for a path name.
7151 // If parent != 0, return the inode for the parent and copy the final
7152 // path element into name, which must have room for DIRSIZ bytes.
7153 // Must be called inside a transaction since it calls iput().
7154 static struct inode*
7155 namex(char *path, int nameiparent, char *name)
7156 {
7157   struct inode *ip, *next;
7158
7159   if(*path == '/')
7160     ip = iget(ROOTDEV, ROOTINO);
7161   else
7162     ip = idup(proc->cwd);
7163
7164   while((path = skipelem(path, name)) != 0){
7165     ilock(ip);
7166     if(ip->type != T_DIR){
7167       iunlockput(ip);
7168       return 0;
7169     }
7170     if(nameiparent && *path == '\0'){
7171       iunlock(ip);  // Stop one level early.
7172       return ip;
7173     }
7174     if((next = dirlookup(ip, name, 0)) == 0){
7175       iunlockput(ip);
7176       return 0;
7177     }
7178     iunlockput(ip);
7179     ip = next;
7180   }
7181   if(nameiparent){
7182     iput(ip);
7183     return 0;
7184   }
7185   return ip;
7186 }
7187
7188 struct inode*
7189 namei(char *path)
7190 {
7191   char name[DIRSIZ];
7192   return namex(path, 0, name);
7193 }
7194
7195 struct inode*
7196 nameiparent(char *path, char *name)
7197 {
7198   return namex(path, 1, name);
7199 }
```

```
7200 struct file {
7201   enum { FD_NONE, FD_PIPE, FD_INODE } type;
7202   int ref; // reference count
7203   char readable;
7204   char writable;
7205   struct pipe *pipe;
7206   struct inode *ip;
7207   uint off;
7208 };
7209
7210
7211 // in-memory copy of an inode
7212 struct inode {
7213   uint dev;           // Device number
7214   uint inum;          // Inode number
7215   int ref;            // Reference count
7216   struct sleeplock lock;
7217   int flags;          // I_VALID
7218
7219   short type;         // copy of disk inode
7220   short major;
7221   short minor;
7222   short nlink;
7223   uint size;
7224   uint addrs[NDIRECT+1];
7225 };
7226 #define I_VALID 0x2
7227
7228 // table mapping major device number to
7229 // device functions
7230 struct devsw {
7231   int (*read)(struct inode*, uint, char*, int);
7232   int (*write)(struct inode*, uint, char*, int);
7233 };
7234
7235 extern struct devsw devsw[];
7236
7237 #define CONSOLE 1
7238
7239
7240
7241
7242
7243
7244
7245
7246
7247
7248
7249
```



```
7250 //
7251 // File descriptors
7252 //
7253
7254 #include "types.h"
7255 #include "defs.h"
7256 #include "param.h"
7257 #include "fs.h"
7258 #include "spinlock.h"
7259 #include "sleeplock.h"
7260 #include "file.h"
7261
7262 struct devsw devsw[NDEV];
7263 struct {
7264   struct spinlock lock;
7265   struct file file[NFILE];
7266 } ftable;
7267
7268 void
7269 fileinit(void)
7270 {
7271   initlock(&ftable.lock, "ftable");
7272 }
7273
7274 // Allocate a file structure.
7275 struct file*
7276 filealloc(void)
7277 {
7278   struct file *f;
7279
7280   acquire(&ftable.lock);
7281   for(f = ftable.file; f < ftable.file + NFILE; f++){
7282     if(f->ref == 0){
7283       f->ref = 1;
7284       release(&ftable.lock);
7285       return f;
7286     }
7287   }
7288   release(&ftable.lock);
7289   return 0;
7290 }
7291
7292
7293
7294
7295
7296
7297
7298
7299
```

```
7300 // Increment ref count for file f.
7301 struct file*
7302 filedup(struct file *f)
7303 {
7304   acquire(&ftable.lock);
7305   if(f->ref < 1)
7306     panic("filedup");
7307   f->ref++;
7308   release(&ftable.lock);
7309   return f;
7310 }
7311
7312 // Close file f.  (Decrement ref count, close when reaches 0.)
7313 void
7314 fileclose(struct file *f)
7315 {
7316   struct file ff;
7317
7318   acquire(&ftable.lock);
7319   if(f->ref < 1)
7320     panic("fileclose");
7321   if(--f->ref > 0){
7322     release(&ftable.lock);
7323     return;
7324   }
7325   ff = *f;
7326   f->ref = 0;
7327   f->type = FD_NONE;
7328   release(&ftable.lock);
7329
7330   if(ff.type == FD_PIPE)
7331     pipeclose(ff.pipe, ff.writable);
7332   else if(ff.type == FD_INODE){
7333     begin_op();
7334     iput(ff.ip);
7335     end_op();
7336   }
7337 }
7338
7339
7340
7341
7342
7343
7344
7345
7346
7347
7348
7349
```



```
7350 // Get metadata about file f.
7351 int
7352 filestat(struct file *f, struct stat *st)
7353 {
7354   if(f->type == FD_INODE){
7355     ilock(f->ip);
7356     stati(f->ip, st);
7357     iunlock(f->ip);
7358     return 0;
7359   }
7360   return -1;
7361 }
7362
7363 // Read from file f.
7364 int
7365 fileread(struct file *f, char *addr, int n)
7366 {
7367   int r;
7368
7369   if(f->readable == 0)
7370     return -1;
7371   if(f->type == FD_PIPE)
7372     return piperead(f->pipe, addr, n);
7373   if(f->type == FD_INODE){
7374     ilock(f->ip);
7375     if((r = readi(f->ip, addr, f->off, n)) > 0)
7376       f->off += r;
7377     iunlock(f->ip);
7378     return r;
7379   }
7380   panic("fileread");
7381 }
7382
7383
7384
7385
7386
7387
7388
7389
7390
7391
7392
7393
7394
7395
7396
7397
7398
7399
```

```
7400 // Write to file f.
7401 int
7402 filewrite(struct file *f, char *addr, int n)
7403 {
7404   int r;
7405
7406   if(f->writable == 0)
7407     return -1;
7408   if(f->type == FD_PIPE)
7409     return pipewrite(f->pipe, addr, n);
7410   if(f->type == FD_INODE){
7411     // write a few blocks at a time to avoid exceeding
7412     // the maximum log transaction size, including
7413     // i-node, indirect block, allocation blocks,
7414     // and 2 blocks of slop for non-aligned writes.
7415     // this really belongs lower down, since writei()
7416     // might be writing a device like the console.
7417     int max = ((LOGSIZE-1-1-2) / 2) * 512;
7418     int i = 0;
7419     while(i < n){
7420       int n1 = n - i;
7421       if(n1 > max)
7422         n1 = max;
7423
7424       begin_op();
7425       ilock(f->ip);
7426       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
7427         f->off += r;
7428       iunlock(f->ip);
7429       end_op();
7430
7431       if(r < 0)
7432         break;
7433       if(r != n1)
7434         panic("short filewrite");
7435       i += r;
7436     }
7437     return i == n ? n : -1;
7438   }
7439   panic("filewrite");
7440 }
7441
7442
7443
7444
7445
7446
7447
7448
7449
```



```
7450 // File-system system calls.
7451 // Mostly argument checking, since we don't trust
7452 // user code, and calls into file.c and fs.c.
7453 //
7454
7455 #include "types.h"
7456 #include "defs.h"
7457 #include "param.h"
7458 #include "stat.h"
7459 #include "mmu.h"
7460 #include "proc.h"
7461 #include "fs.h"
7462 #include "spinlock.h"
7463 #include "sleeplock.h"
7464 #include "file.h"
7465 #include "fcntl.h"
7466
7467 // Fetch the nth word-sized system call argument as a file descriptor
7468 // and return both the descriptor and the corresponding struct file.
7469 static int
7470 argfd(int n, int *pfd, struct file **pf)
7471 {
7472   int fd;
7473   struct file *f;
7474
7475   if(argint(n, &fd) < 0)
7476     return -1;
7477   if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
7478     return -1;
7479   if(pfd)
7480     *pfd = fd;
7481   if(pf)
7482     *pf = f;
7483   return 0;
7484 }
7485
7486
7487
7488
7489
7490
7491
7492
7493
7494
7495
7496
7497
7498
7499
```

```
7500 // Allocate a file descriptor for the given file.
7501 // Takes over file reference from caller on success.
7502 static int
7503 fdalloc(struct file *f)
7504 {
7505   int fd;
7506
7507   for(fd = 0; fd < NOFILE; fd++){
7508     if(proc->ofile[fd] == 0){
7509       proc->ofile[fd] = f;
7510       return fd;
7511     }
7512   }
7513   return -1;
7514 }
7515
7516 int
7517 sys_dup(void)
7518 {
7519   struct file *f;
7520   int fd;
7521
7522   if(argfd(0, 0, &f) < 0)
7523     return -1;
7524   if((fd=fdalloc(f)) < 0)
7525     return -1;
7526   filedup(f);
7527   return fd;
7528 }
7529
7530 int
7531 sys_read(void)
7532 {
7533   struct file *f;
7534   int n;
7535   char *p;
7536
7537   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
7538     return -1;
7539   return fileread(f, p, n);
7540 }
7541
7542
7543
7544
7545
7546
7547
7548
7549
```



```
7550 int
7551 sys_write(void)
7552 {
7553   struct file *f;
7554   int n;
7555   char *p;
7556
7557   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
7558     return -1;
7559   return filewrite(f, p, n);
7560 }
7561
7562 int
7563 sys_close(void)
7564 {
7565   int fd;
7566   struct file *f;
7567
7568   if(argfd(0, &fd, &f) < 0)
7569     return -1;
7570   proc->ofile[fd] = 0;
7571   fileclose(f);
7572   return 0;
7573 }
7574
7575 int
7576 sys_fstat(void)
7577 {
7578   struct file *f;
7579   struct stat *st;
7580
7581   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
7582     return -1;
7583   return filestat(f, st);
7584 }
7585
7586 static int
7587 isdirempty(struct inode *dp)
7588 {
7589   int off;
7590   struct dirent de;
7591   // Is the directory dp empty except for "." and ".." ?
7592   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
7593     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
7594       panic("isdirempty: readi");
7595     if(de.inum != 0)
7596       return 0;
7597   }
7598   return 1;
7599 }
```

```
7600 // Create the path new as a link to the same inode as old.
7601 int
7602 sys_link(void)
7603 {
7604   char name[DIRSIZ], *new, *old;
7605   struct inode *dp, *ip;
7606
7607   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
7608     return -1;
7609
7610   begin_op();
7611   if((ip = namei(old)) == 0){
7612     end_op();
7613     return -1;
7614   }
7615
7616   ilock(ip);
7617   if(ip->type == T_DIR){
7618     iunlockput(ip);
7619     end_op();
7620     return -1;
7621   }
7622
7623   ip->nlink++;
7624   iupdate(ip);
7625   iunlock(ip);
7626
7627   if((dp = nameiparent(new, name)) == 0)
7628     goto bad;
7629   ilock(dp);
7630   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
7631     iunlockput(dp);
7632     goto bad;
7633   }
7634   iunlockput(dp);
7635   iput(ip);
7636
7637   end_op();
7638
7639   return 0;
7640
7641 bad:
7642   ilock(ip);
7643   ip->nlink--;
7644   iupdate(ip);
7645   iunlockput(ip);
7646   end_op();
7647   return -1;
7648 }
7649
```



```
7650 int
7651 sys_unlink(void)
7652 {
7653   struct inode *ip, *dp;
7654   struct dirent de;
7655   char name[DIRSIZ], *path;
7656   uint off;
7657
7658   if(argstr(0, &path) < 0)
7659     return -1;
7660
7661   begin_op();
7662   if((dp = nameiparent(path, name)) == 0){
7663     end_op();
7664     return -1;
7665   }
7666
7667   ilock(dp);
7668
7669   // Cannot unlink "." or "..".
7670   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
7671     goto bad;
7672
7673   if((ip = dirlookup(dp, name, &off)) == 0)
7674     goto bad;
7675   ilock(ip);
7676
7677   if(ip->nlink < 1)
7678     panic("unlink: nlink < 1");
7679   if(ip->type == T_DIR && !isdirempty(ip)){
7680     iunlockput(ip);
7681     goto bad;
7682   }
7683
7684   memset(&de, 0, sizeof(de));
7685   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
7686     panic("unlink: writei");
7687   if(ip->type == T_DIR){
7688     dp->nlink--;
7689     iupdate(dp);
7690   }
7691   iunlockput(dp);
7692
7693   ip->nlink--;
7694   iupdate(ip);
7695   iunlockput(ip);
7696
7697   end_op();
7698
7699   return 0;
```

```
7700 bad:
7701   iunlockput(dp);
7702   end_op();
7703   return -1;
7704 }
7705
7706 static struct inode*
7707 create(char *path, short type, short major, short minor)
7708 {
7709   uint off;
7710   struct inode *ip, *dp;
7711   char name[DIRSIZ];
7712
7713   if((dp = nameiparent(path, name)) == 0)
7714     return 0;
7715   ilock(dp);
7716
7717   if((ip = dirlookup(dp, name, &off)) != 0){
7718     iunlockput(dp);
7719     ilock(ip);
7720     if(type == T_FILE && ip->type == T_FILE)
7721       return ip;
7722     iunlockput(ip);
7723     return 0;
7724   }
7725
7726   if((ip = ialloc(dp->dev, type)) == 0)
7727     panic("create: ialloc");
7728
7729   ilock(ip);
7730   ip->major = major;
7731   ip->minor = minor;
7732   ip->nlink = 1;
7733   iupdate(ip);
7734
7735   if(type == T_DIR){  // Create . and .. entries.
7736     dp->nlink++;  // for ".."
7737     iupdate(dp);
7738     // No ip->nlink++ for ".": avoid cyclic ref count.
7739     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
7740       panic("create dots");
7741   }
7742
7743   if(dirlink(dp, name, ip->inum) < 0)
7744     panic("create: dirlink");
7745
7746   iunlockput(dp);
7747
7748   return ip;
7749 }
```



```
7750 int
7751 sys_open(void)
7752 {
7753   char *path;
7754   int fd, omode;
7755   struct file *f;
7756   struct inode *ip;
7757
7758   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
7759     return -1;
7760
7761   begin_op();
7762
7763   if(omode & O_CREATE){
7764     ip = create(path, T_FILE, 0, 0);
7765     if(ip == 0){
7766       end_op();
7767       return -1;
7768     }
7769   } else {
7770     if((ip = namei(path)) == 0){
7771       end_op();
7772       return -1;
7773     }
7774     ilock(ip);
7775     if(ip->type == T_DIR && omode != O_RDONLY){
7776       iunlockput(ip);
7777       end_op();
7778       return -1;
7779     }
7780   }
7781
7782   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
7783     if(f)
7784       fileclose(f);
7785     iunlockput(ip);
7786     end_op();
7787     return -1;
7788   }
7789   iunlock(ip);
7790   end_op();
7791
7792   f->type = FD_INODE;
7793   f->ip = ip;
7794   f->off = 0;
7795   f->readable = !(omode & O_WRONLY);
7796   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
7797   return fd;
7798 }
7799
```

```
7800 int
7801 sys_mkdir(void)
7802 {
7803   char *path;
7804   struct inode *ip;
7805
7806   begin_op();
7807   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
7808     end_op();
7809     return -1;
7810   }
7811   iunlockput(ip);
7812   end_op();
7813   return 0;
7814 }
7815
7816 int
7817 sys_mknod(void)
7818 {
7819   struct inode *ip;
7820   char *path;
7821   int major, minor;
7822
7823   begin_op();
7824   if((argstr(0, &path)) < 0 ||
7825      argint(1, &major) < 0 ||
7826      argint(2, &minor) < 0 ||
7827      (ip = create(path, T_DEV, major, minor)) == 0){
7828     end_op();
7829     return -1;
7830   }
7831   iunlockput(ip);
7832   end_op();
7833   return 0;
7834 }
7835
7836
7837
7838
7839
7840
7841
7842
7843
7844
7845
7846
7847
7848
7849
```



```
7850 int
7851 sys_chdir(void)
7852 {
7853   char *path;
7854   struct inode *ip;
7855
7856   begin_op();
7857   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
7858     end_op();
7859     return -1;
7860   }
7861   ilock(ip);
7862   if(ip->type != T_DIR){
7863     iunlockput(ip);
7864     end_op();
7865     return -1;
7866   }
7867   iunlock(ip);
7868   iput(proc->cwd);
7869   end_op();
7870   proc->cwd = ip;
7871   return 0;
7872 }
7873
7874 int
7875 sys_exec(void)
7876 {
7877   char *path, *argv[MAXARG];
7878   int i;
7879   addr_t uargv, uarg;
7880
7881   if(argstr(0, &path) < 0 || argaddr(1, &uargv) < 0){
7882     return -1;
7883   }
7884   memset(argv, 0, sizeof(argv));
7885   for(i=0;; i++){
7886     if(i >= NELEM(argv))
7887       return -1;
7888     if(fetchaddr(uargv+(sizeof(addr_t))*i, (addr_t*)&uarg) < 0)
7889       return -1;
7890     if(uarg == 0){
7891       argv[i] = 0;
7892       break;
7893     }
7894     if(fetchstr(uarg, &argv[i]) < 0)
7895       return -1;
7896   }
7897   return exec(path, argv);
7898 }
7899
```

```
7900 int
7901 sys_pipe(void)
7902 {
7903   int *fd;
7904   struct file *rf, *wf;
7905   int fd0, fd1;
7906
7907   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
7908     return -1;
7909   if(pipealloc(&rf, &wf) < 0)
7910     return -1;
7911   fd0 = -1;
7912   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
7913     if(fd0 >= 0)
7914       proc->ofile[fd0] = 0;
7915     fileclose(rf);
7916     fileclose(wf);
7917     return -1;
7918   }
7919   fd[0] = fd0;
7920   fd[1] = fd1;
7921   return 0;
7922 }
7923
7924
7925
7926
7927
7928
7929
7930
7931
7932
7933
7934
7935
7936
7937
7938
7939
7940
7941
7942
7943
7944
7945
7946
7947
7948
7949
```



```
7950 #include "types.h"
7951 #include "defs.h"
7952 #include "param.h"
7953 #include "mmu.h"
7954 #include "proc.h"
7955 #include "fs.h"
7956 #include "spinlock.h"
7957 #include "sleeplock.h"
7958 #include "file.h"
7959
7960 #define PIPESIZE 512
7961
7962 struct pipe {
7963   struct spinlock lock;
7964   char data[PIPESIZE];
7965   uint nread;     // number of bytes read
7966   uint nwrite;    // number of bytes written
7967   int readopen;   // read fd is still open
7968   int writeopen;  // write fd is still open
7969 };
7970
7971 int
7972 pipealloc(struct file **f0, struct file **f1)
7973 {
7974   struct pipe *p;
7975
7976   p = 0;
7977   *f0 = *f1 = 0;
7978   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
7979     goto bad;
7980   if((p = (struct pipe*)kalloc()) == 0)
7981     goto bad;
7982   p->readopen = 1;
7983   p->writeopen = 1;
7984   p->nwrite = 0;
7985   p->nread = 0;
7986   initlock(&p->lock, "pipe");
7987   (*f0)->type = FD_PIPE;
7988   (*f0)->readable = 1;
7989   (*f0)->writable = 0;
7990   (*f0)->pipe = p;
7991   (*f1)->type = FD_PIPE;
7992   (*f1)->readable = 0;
7993   (*f1)->writable = 1;
7994   (*f1)->pipe = p;
7995   return 0;
7996
7997
7998
7999
```

```
 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;
}
```

```
8100 #include "types.h"
8101 #include "x86.h"
8102
8103 void*
8104 memset(void *dst, int c, uint64 n)
8105 {
8106   if ((addr_t)dst%4 == 0 && n%4 == 0){
8107     c &= 0xFF;
8108     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
8109   } else
8110     stosb(dst, c, n);
8111   return dst;
8112 }
8113
8114 int
8115 memcmp(const void *v1, const void *v2, uint n)
8116 {
8117   const uchar *s1, *s2;
8118
8119   s1 = v1;
8120   s2 = v2;
8121   while(n-- > 0){
8122     if(*s1 != *s2)
8123       return *s1 - *s2;
8124     s1++, s2++;
8125   }
8126
8127   return 0;
8128 }
8129
8130 void*
8131 memmove(void *dst, const void *src, uint n)
8132 {
8133   const char *s;
8134   char *d;
8135
8136   s = src;
8137   d = dst;
8138   if(s < d && s + n > d){
8139     s += n;
8140     d += n;
8141     while(n-- > 0)
8142       *--d = *--s;
8143   } else
8144     while(n-- > 0)
8145       *d++ = *s++;
8146
8147   return dst;
8148 }
8149
```



```
8150 // memcpy exists to placate GCC.  Use memmove.
8151 void*
8152 memcpy(void *dst, const void *src, uint n)
8153 {
8154   return memmove(dst, src, n);
8155 }
8156
8157 int
8158 strncmp(const char *p, const char *q, uint n)
8159 {
8160   while(n > 0 && *p && *p == *q)
8161     n--, p++, q++;
8162   if(n == 0)
8163     return 0;
8164   return (uchar)*p - (uchar)*q;
8165 }
8166
8167 char*
8168 strncpy(char *s, const char *t, int n)
8169 {
8170   char *os = s;
8171   while(n-- > 0 && (*s++ = *t++) != 0)
8172     ;
8173   while(n-- > 0)
8174     *s++ = 0;
8175   return os;
8176 }
8177
8178 // Like strncpy but guaranteed to NUL-terminate.
8179 char*
8180 safestrcpy(char *s, const char *t, int n)
8181 {
8182   char *os = s;
8183   if(n <= 0)
8184     return os;
8185   while(--n > 0 && (*s++ = *t++) != 0)
8186     ;
8187   *s = 0;
8188   return os;
8189 }
8190
8191 int
8192 strlen(const char *s)
8193 {
8194   int n;
8195
8196   for(n = 0; s[n]; n++)
8197     ;
8198   return n;
8199 }
```

```
8200 # Initial process execs /init.
8201
8202 #include "syscall.h"
8203 #include "traps.h"
8204
8205 # exec(init, argv)
8206 .code64
8207 .global start
8208 start:
8209   mov $init, %rdi
8210   mov $argv, %rsi
8211   mov $SYS_exec, %rax
8212   syscall
8213
8214 # for(;;) exit();
8215 exit:
8216   mov $SYS_exit, %rax
8217   syscall
8218   jmp exit
8219
8220 # char init[] = "/init\0";
8221 init:
8222   .string "/init\0"
8223
8224 # char *argv[] = { init, 0 };
8225 .p2align 3
8226 argv:
8227   .quad init
8228   .quad 0
8229
8230
8231
8232
8233
8234
8235
8236
8237
8238
8239
8240
8241
8242
8243
8244
8245
8246
8247
8248
8249
```



```
8250 #include "syscall.h"
8251 #include "traps.h"
8252
8253 #define SYSCALL(name) \
8254   .global name; \
8255   name: \
8256     mov $SYS_ ## name, %rax; \
8257     mov %rcx, %r10 ;\
8258     syscall                   ;\
8259     ret
8260
8261 SYSCALL(fork)
8262 SYSCALL(exit)
8263 SYSCALL(wait)
8264 SYSCALL(pipe)
8265 SYSCALL(read)
8266 SYSCALL(write)
8267 SYSCALL(close)
8268 SYSCALL(kill)
8269 SYSCALL(exec)
8270 SYSCALL(open)
8271 SYSCALL(mknod)
8272 SYSCALL(unlink)
8273 SYSCALL(fstat)
8274 SYSCALL(link)
8275 SYSCALL(mkdir)
8276 SYSCALL(chdir)
8277 SYSCALL(dup)
8278 SYSCALL(getpid)
8279 SYSCALL(sbrk)
8280 SYSCALL(sleep)
8281 SYSCALL(uptime)
8282
8283
8284
8285
8286
8287
8288
8289
8290
8291
8292
8293
8294
8295
8296
8297
8298
8299
```

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



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

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



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

```
8500 int
8501 main(void)
8502 {
8503   static char buf[100];
8504   int fd;
8505
8506   // Ensure that three file descriptors are open.
8507   while((fd = open("console", O_RDWR)) >= 0){
8508     if(fd >= 3){
8509       close(fd);
8510       break;
8511     }
8512   }
8513
8514   // Read and run input commands.
8515   while(getcmd(buf, sizeof(buf)) >= 0){
8516     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
8517       // Chdir must be called by the parent, not the child.
8518       buf[strlen(buf)-1] = 0;  // chop \n
8519       if(chdir(buf+3) < 0)
8520         printf(2, "cannot cd %s\n", buf+3);
8521       continue;
8522     }
8523     if(fork1() == 0)
8524       runcmd(parsecmd(buf));
8525     wait();
8526   }
8527   exit();
8528 }
8529
8530 void
8531 panic(char *s)
8532 {
8533   printf(2, "%s\n", s);
8534   exit();
8535 }
8536
8537 int
8538 fork1(void)
8539 {
8540   int pid;
8541
8542   pid = fork();
8543   if(pid == -1)
8544     panic("fork");
8545   return pid;
8546 }
8547
8548
8549
```



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

```
8600 struct cmd*
8601 listcmd(struct cmd *left, struct cmd *right)
8602 {
8603   struct listcmd *cmd;
8604
8605   cmd = malloc(sizeof(*cmd));
8606   memset(cmd, 0, sizeof(*cmd));
8607   cmd->type = LIST;
8608   cmd->left = left;
8609   cmd->right = right;
8610   return (struct cmd*)cmd;
8611 }
8612
8613 struct cmd*
8614 backcmd(struct cmd *subcmd)
8615 {
8616   struct backcmd *cmd;
8617
8618   cmd = malloc(sizeof(*cmd));
8619   memset(cmd, 0, sizeof(*cmd));
8620   cmd->type = BACK;
8621   cmd->cmd = subcmd;
8622   return (struct cmd*)cmd;
8623 }
8624
8625
8626
8627
8628
8629
8630
8631
8632
8633
8634
8635
8636
8637
8638
8639
8640
8641
8642
8643
8644
8645
8646
8647
8648
8649
```



```
8650 // Parsing
8651
8652 char whitespace[] = " \t\r\n\v";
8653 char symbols[] = "<|>&;()";
8654
8655 int
8656 gettoken(char **ps, char *es, char **q, char **eq)
8657 {
8658   char *s;
8659   int ret;
8660
8661   s = *ps;
8662   while(s < es && strchr(whitespace, *s))
8663     s++;
8664   if(q)
8665     *q = s;
8666   ret = *s;
8667   switch(*s){
8668   case 0:
8669     break;
8670   case '|':
8671   case '(':
8672   case ')':
8673   case ';':
8674   case '&':
8675   case '<':
8676     s++;
8677     break;
8678   case '>':
8679     s++;
8680     if(*s == '>'){
8681       ret = '+';
8682       s++;
8683     }
8684     break;
8685   default:
8686     ret = 'a';
8687     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
8688       s++;
8689     break;
8690   }
8691   if(eq)
8692     *eq = s;
8693
8694   while(s < es && strchr(whitespace, *s))
8695     s++;
8696   *ps = s;
8697   return ret;
8698 }
8699
```

```
8700 int
8701 peek(char **ps, char *es, char *toks)
8702 {
8703   char *s;
8704
8705   s = *ps;
8706   while(s < es && strchr(whitespace, *s))
8707     s++;
8708   *ps = s;
8709   return *s && strchr(toks, *s);
8710 }
8711
8712 struct cmd *parseline(char**, char*);
8713 struct cmd *parsepipe(char**, char*);
8714 struct cmd *parseexec(char**, char*);
8715 struct cmd *nulterminate(struct cmd*);
8716
8717 struct cmd*
8718 parsecmd(char *s)
8719 {
8720   char *es;
8721   struct cmd *cmd;
8722
8723   es = s + strlen(s);
8724   cmd = parseline(&s, es);
8725   peek(&s, es, "");
8726   if(s != es){
8727     printf(2, "leftovers: %s\n", s);
8728     panic("syntax");
8729   }
8730   nulterminate(cmd);
8731   return cmd;
8732 }
8733
8734 struct cmd*
8735 parseline(char **ps, char *es)
8736 {
8737   struct cmd *cmd;
8738
8739   cmd = parsepipe(ps, es);
8740   while(peek(ps, es, "&")){
8741     gettoken(ps, es, 0, 0);
8742     cmd = backcmd(cmd);
8743   }
8744   if(peek(ps, es, ";")){
8745     gettoken(ps, es, 0, 0);
8746     cmd = listcmd(cmd, parseline(ps, es));
8747   }
8748   return cmd;
8749 }
```



```
8750 struct cmd*
8751 parsepipe(char **ps, char *es)
8752 {
8753   struct cmd *cmd;
8754
8755   cmd = parseexec(ps, es);
8756   if(peek(ps, es, "|")){
8757     gettoken(ps, es, 0, 0);
8758     cmd = pipecmd(cmd, parsepipe(ps, es));
8759   }
8760   return cmd;
8761 }
8762
8763 struct cmd*
8764 parseredirs(struct cmd *cmd, char **ps, char *es)
8765 {
8766   int tok;
8767   char *q, *eq;
8768
8769   while(peek(ps, es, "<>")){
8770     tok = gettoken(ps, es, 0, 0);
8771     if(gettoken(ps, es, &q, &eq) != 'a')
8772       panic("missing file for redirection");
8773     switch(tok){
8774     case '<':
8775       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
8776       break;
8777     case '>':
8778       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8779       break;
8780     case '+':  // >>
8781       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8782       break;
8783     }
8784   }
8785   return cmd;
8786 }
8787
8788
8789
8790
8791
8792
8793
8794
8795
8796
8797
8798
8799
```

```
8800 struct cmd*
8801 parseblock(char **ps, char *es)
8802 {
8803   struct cmd *cmd;
8804
8805   if(!peek(ps, es, "("))
8806     panic("parseblock");
8807   gettoken(ps, es, 0, 0);
8808   cmd = parseline(ps, es);
8809   if(!peek(ps, es, ")"))
8810     panic("syntax - missing )");
8811   gettoken(ps, es, 0, 0);
8812   cmd = parseredirs(cmd, ps, es);
8813   return cmd;
8814 }
8815
8816 struct cmd*
8817 parseexec(char **ps, char *es)
8818 {
8819   char *q, *eq;
8820   int tok, argc;
8821   struct execcmd *cmd;
8822   struct cmd *ret;
8823
8824   if(peek(ps, es, "("))
8825     return parseblock(ps, es);
8826
8827   ret = execcmd();
8828   cmd = (struct execcmd*)ret;
8829
8830   argc = 0;
8831   ret = parseredirs(ret, ps, es);
8832   while(!peek(ps, es, "|)&;")){
8833     if((tok=gettoken(ps, es, &q, &eq)) == 0)
8834       break;
8835     if(tok != 'a')
8836       panic("syntax");
8837     cmd->argv[argc] = q;
8838     cmd->eargv[argc] = eq;
8839     argc++;
8840     if(argc >= MAXARGS)
8841       panic("too many args");
8842     ret = parseredirs(ret, ps, es);
8843   }
8844   cmd->argv[argc] = 0;
8845   cmd->eargv[argc] = 0;
8846   return ret;
8847 }
8848
8849
```



```
8850 // NUL-terminate all the counted strings.
8851 struct cmd*
8852 nulterminate(struct cmd *cmd)
8853 {
8854   int i;
8855   struct backcmd *bcmd;
8856   struct execcmd *ecmd;
8857   struct listcmd *lcmd;
8858   struct pipecmd *pcmd;
8859   struct redircmd *rcmd;
8860
8861   if(cmd == 0)
8862     return 0;
8863
8864   switch(cmd->type){
8865   case EXEC:
8866     ecmd = (struct execcmd*)cmd;
8867     for(i=0; ecmd->argv[i]; i++)
8868       *ecmd->eargv[i] = 0;
8869     break;
8870
8871   case REDIR:
8872     rcmd = (struct redircmd*)cmd;
8873     nulterminate(rcmd->cmd);
8874     *rcmd->efile = 0;
8875     break;
8876
8877   case PIPE:
8878     pcmd = (struct pipecmd*)cmd;
8879     nulterminate(pcmd->left);
8880     nulterminate(pcmd->right);
8881     break;
8882
8883   case LIST:
8884     lcmd = (struct listcmd*)cmd;
8885     nulterminate(lcmd->left);
8886     nulterminate(lcmd->right);
8887     break;
8888
8889   case BACK:
8890     bcmd = (struct backcmd*)cmd;
8891     nulterminate(bcmd->cmd);
8892     break;
8893   }
8894   return cmd;
8895 }
8896
8897
8898
8899
```