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

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14,
2000)). See also http://pdos.csail.mit.edu/6.828/2016/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, and Austin
Clements.

We are also grateful for the bug reports and patches contributed by Silas
Boyd-Wickizer, Mike CAT, Nelson Elhage, Nathaniel Filardo, Peter Froehlich,
Yakir Goaran, Shivam Handa, Bryan Henry, Jim Huang, Anders Kaseorg, kehao95,
Eddie Kohler, Imbar Marinescu, Yandong Mao, Hitoshi Mitake, Carmi Merimovich,
Joel Nider, Greg Price, Ayan Shafqat, Eldar Sehayek, Yongming Shen, Cam Tenny,
Rafael Ubal, Warren Toomey, Stephen Tu, Pablo Ventura, Xi Wang, Keiichi
Watanabe, Nicolas Wolovick, Jindong Zhang, and Zou Chang Wei.

ERROR REPORTS

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

BUILDING AND RUNNING XV6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run "make".
On non-x86 or non-ELF machines (like OS X, even on x86), you will
need to install a cross-compiler gcc suite capable of producing x86 ELF
binaries. See http://pdos.csail.mit.edu/6.828/2016/tools.html.
Then run "make TOOLPREFIX=i386-jos-elf-".

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

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

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

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

```
    swtch 2658
        0374 2428 2466 2657 2658
```

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

```
typedef unsigned int   uint;
typedef unsigned short ushort;
typedef unsigned char  uchar;
typedef uint pde_t;
```

```
#define NPROC        64  // maximum number of processes
#define KSTACKSIZE 4096  // size of per-process kernel stack
#define NCPU          8  // maximum number of CPUs
#define NOFILE       16  // open files per process
#define NFILE       100  // open files per system
#define NINODE       50  // maximum number of active i-nodes
#define NDEV         10  // maximum major device number
#define ROOTDEV       1  // device number of file system root disk
#define MAXARG       32  // max exec arguments
#define MAXOPBLOCKS  10  // max # of blocks any FS op writes
#define LOGSIZE      (MAXOPBLOCKS*3)  // max data blocks in on-disk log
#define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
#define FSSIZE       1000  // size of file system in blocks
```

```
// Memory layout

#define EXTMEM  0x100000            // Start of extended memory
#define PHYSTOP 0xE000000           // Top physical memory
#define DEVSPACE 0xFE000000         // Other devices are at high addresses

// Key addresses for address space layout (see kmap in vm.c for layout)
#define KERNBASE 0x80000000         // First kernel virtual address
#define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked

#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) (((void *) (a)) + KERNBASE)

#define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
#define P2V_WO(x) ((x) + KERNBASE)    // same as P2V, but without casts
```

```
struct buf;
struct context;
struct file;
struct inode;
struct pipe;
struct proc;
struct rtcdate;
struct spinlock;
struct stat;
struct superblock;

// bio.c
void            binit(void);
struct buf*     bread(uint, uint);
void            brelse(struct buf*);
void            bwrite(struct buf*);

// console.c
void            consoleinit(void);
void            cprintf(char*, ...);
void            consoleintr(int(*)(void));
void            panic(char*) __attribute__((noreturn));

// exec.c
int             exec(char*, char**);

// file.c
struct file*    filealloc(void);
void            fileclose(struct file*);
struct file*    filedup(struct file*);
void            fileinit(void);
int             fileread(struct file*, char*, int n);
int             filestat(struct file*, struct stat*);
int             filewrite(struct file*, char*, int n);

// fs.c
void            readsb(int dev, struct superblock *sb);
int             dirlink(struct inode*, char*, uint);
struct inode*   dirlookup(struct inode*, char*, uint*);
struct inode*   ialloc(uint, short);
struct inode*   idup(struct inode*);
void            iinit(int dev);
void            ilock(struct inode*);
void            iput(struct inode*);
void            iunlock(struct inode*);
void            iunlockput(struct inode*);
void            iupdate(struct inode*);
int             namecmp(const char*, const char*);
struct inode*   namei(char*);
struct inode*   nameiparent(char*, char*);
```

```
int             readi(struct inode*, char*, uint, uint);
void            stati(struct inode*, struct stat*);
int             writei(struct inode*, char*, uint, uint);

// ide.c
void            ideinit(void);
void            ideintr(void);
void            iderw(struct buf*);

// ioapic.c
void            ioapicenable(int irq, int cpu);
extern uchar    ioapicid;
void            ioapicinit(void);

// kalloc.c
char*           kalloc(void);
void            kfree(char*);
void            kinit1(void*, void*);
void            kinit2(void*, void*);

// kbd.c
void            kbdintr(void);

// lapic.c
void            cmostime(struct rtcdate *r);
int             cpunum(void);
extern volatile uint*    lapic;
void            lapiceoi(void);
void            lapicinit(void);
void            lapicstartap(uchar, uint);
void            microdelay(int);

// log.c
void            initlog(int dev);
void            log_write(struct buf*);
void            begin_op();
void            end_op();

// mp.c
extern int      ismp;
void            mpinit(void);

// picirq.c
void            picenable(int);
void            picinit(void);





```

```
// pipe.c
int             pipealloc(struct file**, struct file**);
void            pipeclose(struct pipe*, int);
int             piperead(struct pipe*, char*, int);
int             pipewrite(struct pipe*, char*, int);


// proc.c
void            exit(void);
int             fork(void);
int             growproc(int);
int             kill(int);
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(void);
void            wakeup(void*);
void            yield(void);

// swtch.S
void            swtch(struct context**, struct context*);

// spinlock.c
void            acquire(struct spinlock*);
void            getcallerpcs(void*, uint*);
int             holding(struct spinlock*);
void            initlock(struct spinlock*, char*);
void            release(struct spinlock*);
void            pushcli(void);
void            popcli(void);

// string.c
int             memcmp(const void*, const void*, uint);
void*           memmove(void*, const void*, uint);
void*           memset(void*, int, uint);
char*           safestrcpy(char*, const char*, int);
int             strlen(const char*);
int             strncmp(const char*, const char*, uint);
char*           strncpy(char*, const char*, int);

// syscall.c
int             argint(int, int*);
int             argptr(int, char**, int);
int             argstr(int, char**);
int             fetchint(uint, int*);
int             fetchstr(uint, char**);
void            syscall(void);
```

```
// timer.c
void            timerinit(void);

// trap.c
void            idtinit(void);
extern uint     ticks;
void            tvinit(void);
extern struct spinlock tickslock;

// uart.c
void            uartinit(void);
void            uartintr(void);
void            uartputc(int);

// vm.c
void            seginit(void);
void            kvmalloc(void);
pde_t*          setupkvm(void);
char*           uva2ka(pde_t*, char*);
int             allocuvm(pde_t*, uint, uint);
int             deallocuvm(pde_t*, uint, uint);
void            freevm(pde_t*);
void            inituvm(pde_t*, char*, uint);
int             loaduvm(pde_t*, char*, struct inode*, uint, uint);
pde_t*          copyuvm(pde_t*, uint);
void            switchuvm(struct proc*);
void            switchkvm(void);
int             copyout(pde_t*, uint, void*, uint);
void            clearpteu(pde_t *pgdir, char *uva);

// number of elements in fixed-size array
#define NELEM(x) (sizeof(x)/sizeof((x)[0]))
```

```
// Routines to let C code use special x86 instructions.

static inline uchar
inb(ushort port)
{
  uchar data;

  asm volatile("in %1,%0" : "=a" (data) : "d" (port));
  return data;
}

static inline void
insl(int port, void *addr, int cnt)
{
  asm volatile("cld; rep insl" :
               "=D" (addr), "=c" (cnt) :
               "d" (port), "0" (addr), "1" (cnt) :
               "memory", "cc");
}

static inline void
outb(ushort port, uchar data)
{
  asm volatile("out %0,%1" : : "a" (data), "d" (port));
}

static inline void
outw(ushort port, ushort data)
{
  asm volatile("out %0,%1" : : "a" (data), "d" (port));
}

static inline void
outsl(int port, const void *addr, int cnt)
{
  asm volatile("cld; rep outsl" :
               "=S" (addr), "=c" (cnt) :
               "d" (port), "0" (addr), "1" (cnt) :
               "cc");
}

static inline void
stosb(void *addr, int data, int cnt)
{
  asm volatile("cld; rep stosb" :
               "=D" (addr), "=c" (cnt) :
               "0" (addr), "1" (cnt), "a" (data) :
               "memory", "cc");
}
```

```
static inline void
stosl(void *addr, int data, int cnt)
{
  asm volatile("cld; rep stosl" :
               "=D" (addr), "=c" (cnt) :
               "0" (addr), "1" (cnt), "a" (data) :
               "memory", "cc");
}

struct segdesc;

static inline void
lgdt(struct segdesc *p, int size)
{
  volatile ushort pd[3];

  pd[0] = size-1;
  pd[1] = (uint)p;
  pd[2] = (uint)p >> 16;

  asm volatile("lgdt (%0)" : : "r" (pd));
}

struct gatedesc;

static inline void
lidt(struct gatedesc *p, int size)
{
  volatile ushort pd[3];

  pd[0] = size-1;
  pd[1] = (uint)p;
  pd[2] = (uint)p >> 16;

  asm volatile("lidt (%0)" : : "r" (pd));
}

static inline void
ltr(ushort sel)
{
  asm volatile("ltr %0" : : "r" (sel));
}

static inline uint
readeflags(void)
{
  uint eflags;
  asm volatile("pushfl; popl %0" : "=r" (eflags));
  return eflags;
}
```

```
static inline void
loadgs(ushort v)
{
  asm volatile("movw %0, %%gs" : : "r" (v));
}

static inline void
cli(void)
{
  asm volatile("cli");
}

static inline void
sti(void)
{
  asm volatile("sti");
}

static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}

static inline uint
rcr2(void)
{
  uint val;
  asm volatile("movl %%cr2,%0" : "=r" (val));
  return val;
}

static inline void
lcr3(uint val)
{
  asm volatile("movl %0,%%cr3" : : "r" (val));
}
```

```
// Layout of the trap frame built on the stack by the
// hardware and by trapasm.S, and passed to trap().
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```

```
//
// assembler macros to create x86 segments
//

#define SEG_NULLASM                                             \
        .word 0, 0;                                             \
        .byte 0, 0, 0, 0

// The 0xC0 means the limit is in 4096-byte units
// and (for executable segments) 32-bit mode.
#define SEG_ASM(type,base,lim)                                  \
        .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
        .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
                (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)

#define STA_X     0x8       // Executable segment
#define STA_E     0x4       // Expand down (non-executable segments)
#define STA_C     0x4       // Conforming code segment (executable only)
#define STA_W     0x2       // Writeable (non-executable segments)
#define STA_R     0x2       // Readable (executable segments)
#define STA_A     0x1       // Accessed
```

```
// This file contains definitions for the
// x86 memory management unit (MMU).

// Eflags register
#define FL_CF           0x00000001      // Carry Flag
#define FL_PF           0x00000004      // Parity Flag
#define FL_AF           0x00000010      // Auxiliary carry Flag
#define FL_ZF           0x00000040      // Zero Flag
#define FL_SF           0x00000080      // Sign Flag
#define FL_TF           0x00000100      // Trap Flag
#define FL_IF           0x00000200      // Interrupt Enable
#define FL_DF           0x00000400      // Direction Flag
#define FL_OF           0x00000800      // Overflow Flag
#define FL_IOPL_MASK    0x00003000      // I/O Privilege Level bitmask
#define FL_IOPL_0       0x00000000      //   IOPL == 0
#define FL_IOPL_1       0x00001000      //   IOPL == 1
#define FL_IOPL_2       0x00002000      //   IOPL == 2
#define FL_IOPL_3       0x00003000      //   IOPL == 3
#define FL_NT           0x00004000      // Nested Task
#define FL_RF           0x00010000      // Resume Flag
#define FL_VM           0x00020000      // Virtual 8086 mode
#define FL_AC           0x00040000      // Alignment Check
#define FL_VIF          0x00080000      // Virtual Interrupt Flag
#define FL_VIP          0x00100000      // Virtual Interrupt Pending
#define FL_ID           0x00200000      // ID flag

// Control Register flags
#define CR0_PE          0x00000001      // Protection Enable
#define CR0_MP          0x00000002      // Monitor coProcessor
#define CR0_EM          0x00000004      // Emulation
#define CR0_TS          0x00000008      // Task Switched
#define CR0_ET          0x00000010      // Extension Type
#define CR0_NE          0x00000020      // Numeric Errror
#define CR0_WP          0x00010000      // Write Protect
#define CR0_AM          0x00040000      // Alignment Mask
#define CR0_NW          0x20000000      // Not Writethrough
#define CR0_CD          0x40000000      // Cache Disable
#define CR0_PG          0x80000000      // Paging

#define CR4_PSE         0x00000010      // Page size extension

// various segment selectors.
#define SEG_KCODE 1  // kernel code
#define SEG_KDATA 2  // kernel data+stack
#define SEG_KCPU  3  // kernel per-cpu data
#define SEG_UCODE 4  // user code
#define SEG_UDATA 5  // user data+stack
#define SEG_TSS   6  // this process's task state


```

```
// cpu->gdt[NSEGS] holds the above segments.
#define NSEGS     7
```

```
#ifndef __ASSEMBLER__
// Segment Descriptor
struct segdesc {
  uint lim_15_0 : 16;  // Low bits of segment limit
  uint base_15_0 : 16; // Low bits of segment base address
  uint base_23_16 : 8; // Middle bits of segment base address
  uint type : 4;       // Segment type (see STS_ constants)
  uint s : 1;          // 0 = system, 1 = application
  uint dpl : 2;        // Descriptor Privilege Level
  uint p : 1;          // Present
  uint lim_19_16 : 4;  // High bits of segment limit
  uint avl : 1;        // Unused (available for software use)
  uint rsv1 : 1;       // Reserved
  uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
  uint g : 1;          // Granularity: limit scaled by 4K when set
  uint base_31_24 : 8; // High bits of segment base address
};

// Normal segment
#define SEG(type, base, lim, dpl) (struct segdesc)    \
{ ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff,      \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
  (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
#define SEG16(type, base, lim, dpl) (struct segdesc)  \
{ (lim) & 0xffff, (uint)(base) & 0xffff,              \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
  (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
#endif

#define DPL_USER    0x3     // User DPL

// Application segment type bits
#define STA_X       0x8     // Executable segment
#define STA_E       0x4     // Expand down (non-executable segments)
#define STA_C       0x4     // Conforming code segment (executable only)
#define STA_W       0x2     // Writeable (non-executable segments)
#define STA_R       0x2     // Readable (executable segments)
#define STA_A       0x1     // Accessed

// System segment type bits
#define STS_T16A    0x1     // Available 16-bit TSS
#define STS_LDT     0x2     // Local Descriptor Table
#define STS_T16B    0x3     // Busy 16-bit TSS
#define STS_CG16    0x4     // 16-bit Call Gate
#define STS_TG      0x5     // Task Gate / Coum Transmitions
#define STS_IG16    0x6     // 16-bit Interrupt Gate
#define STS_TG16    0x7     // 16-bit Trap Gate
#define STS_T32A    0x9     // Available 32-bit TSS
#define STS_T32B    0xB     // Busy 32-bit TSS
#define STS_CG32    0xC     // 32-bit Call Gate
```

```
#define STS_IG32    0xE     // 32-bit Interrupt Gate
#define STS_TG32    0xF     // 32-bit Trap Gate

// A virtual address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(va) --/ \--- PTX(va) --/

// page directory index
#define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)

// page table index
#define PTX(va)         (((uint)(va) >> PTXSHIFT) & 0x3FF)

// construct virtual address from indexes and offset
#define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))

// Page directory and page table constants.
#define NPDENTRIES      1024    // # directory entries per page directory
#define NPTENTRIES      1024    // # PTEs per page table
#define PGSIZE          4096    // bytes mapped by a page

#define PGSHIFT         12      // log2(PGSIZE)
#define PTXSHIFT        12      // offset of PTX in a linear address
#define PDXSHIFT        22      // offset of PDX in a linear address

#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))

// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PWT         0x008   // Write-Through
#define PTE_PCD         0x010   // Cache-Disable
#define PTE_A           0x020   // Accessed
#define PTE_D           0x040   // Dirty
#define PTE_PS          0x080   // Page Size
#define PTE_MBZ         0x180   // Bits must be zero

// Address in page table or page directory entry
#define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
#define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF)

#ifndef __ASSEMBLER__
typedef uint pte_t;

```

```
// Task state segment format
struct taskstate {
  uint link;         // Old ts selector
  uint esp0;         // Stack pointers and segment selectors
  ushort ss0;        //   after an increase in privilege level
  ushort padding1;
  uint *esp1;
  ushort ss1;
  ushort padding2;
  uint *esp2;
  ushort ss2;
  ushort padding3;
  void *cr3;         // Page directory base
  uint *eip;         // Saved state from last task switch
  uint eflags;
  uint eax;          // More saved state (registers)
  uint ecx;
  uint edx;
  uint ebx;
  uint *esp;
  uint *ebp;
  uint esi;
  uint edi;
  ushort es;         // Even more saved state (segment selectors)
  ushort padding4;
  ushort cs;
  ushort padding5;
  ushort ss;
  ushort padding6;
  ushort ds;
  ushort padding7;
  ushort fs;
  ushort padding8;
  ushort gs;
  ushort padding9;
  ushort ldt;
  ushort padding10;
  ushort t;          // Trap on task switch
  ushort iomb;       // I/O map base address
};
```

```
// Gate descriptors for interrupts and traps
struct gatedesc {
  uint off_15_0 : 16;   // low 16 bits of offset in segment
  uint cs : 16;         // code segment selector
  uint args : 5;        // # args, 0 for interrupt/trap gates
  uint rsv1 : 3;        // reserved(should be zero I guess)
  uint type : 4;        // type(STS_{TG,IG32,TG32})
  uint s : 1;           // must be 0 (system)
  uint dpl : 2;         // descriptor(meaning new) privilege level
  uint p : 1;           // Present
  uint off_31_16 : 16;  // high bits of offset in segment
};

// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
//   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//        the privilege level required for software to invoke
//        this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, d)                \
{                                                         \
  (gate).off_15_0 = (uint)(off) & 0xffff;                \
  (gate).cs = (sel);                                      \
  (gate).args = 0;                                        \
  (gate).rsv1 = 0;                                        \
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
  (gate).s = 0;                                           \
  (gate).dpl = (d);                                       \
  (gate).p = 1;                                           \
  (gate).off_31_16 = (uint)(off) >> 16;                  \
}

#endif
```

```
// Format of an ELF executable file

#define ELF_MAGIC 0x464C457FU  // "\x7FELF" in little endian

// File header
struct elfhdr {
  uint magic;  // must equal ELF_MAGIC
  uchar elf[12];
  ushort type;
  ushort machine;
  uint version;
  uint entry;
  uint phoff;
  uint shoff;
  uint flags;
  ushort ehsize;
  ushort phentsize;
  ushort phnum;
  ushort shentsize;
  ushort shnum;
  ushort shstrndx;
};

// Program section header
struct proghdr {
  uint type;
  uint off;
  uint vaddr;
  uint paddr;
  uint filesz;
  uint memsz;
  uint flags;
  uint align;
};

// Values for Proghdr type
#define ELF_PROG_LOAD           1

// Flag bits for Proghdr flags
#define ELF_PROG_FLAG_EXEC      1
#define ELF_PROG_FLAG_WRITE     2
#define ELF_PROG_FLAG_READ      4
```

```
// Blank page.
```

```
# The xv6 kernel starts executing in this file. This file is linked with
# the kernel C code, so it can refer to kernel symbols such as main().
# The boot block (bootasm.S and bootmain.c) jumps to entry below.

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

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

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

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

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

```
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  # Turn on paging.
  movl    %cr0, %eax
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0

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

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

.comm stack, KSTACKSIZE
```

```
#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 combines elements of bootasm.S and entry.S.

.code16
.globl start
start:
  cli

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax
  movw    %ax,%ds
  movw    %ax,%es
  movw    %ax,%ss

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

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

  # Turn on page size extension for 4Mbyte pages
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  # Use entrypgdir as our initial page table
  movl    (start-12), %eax
  movl    %eax, %cr3
  # Turn on paging.
  movl    %cr0, %eax
  orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0

  # 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
```

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

static void startothers(void);
static void mpmain(void)  __attribute__((noreturn));
extern pde_t *kpgdir;
extern char end[]; // first address after kernel loaded from ELF file

// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  cprintf("\ncpu%d: starting xv6\n\n", cpunum());
  picinit();       // another interrupt controller
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  if(!ismp)
    timerinit();   // uniprocessor timer
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
```

```c
// Other CPUs jump here from entryother.S.
static void
mpenter(void)
{
  switchkvm();
  seginit();
  lapicinit();
  mpmain();
}

// Common CPU setup code.
static void
mpmain(void)
{
  cprintf("cpu%d: starting\n", cpunum());
  idtinit();       // load idt register
  xchg(&cpu->started, 1); // tell startothers() we're up
  scheduler();     // start running processes
}

pde_t entrypgdir[];  // For entry.S

// Start the non-boot (AP) processors.
static void
startothers(void)
{
  extern uchar _binary_entryother_start[], _binary_entryother_size[];
  uchar *code;
  struct cpu *c;
  char *stack;

  // Write entry code to unused memory at 0x7000.
  // The linker has placed the image of entryother.S in
  // _binary_entryother_start.
  code = P2V(0x7000);
  memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);

  for(c = cpus; c < cpus+ncpu; c++){
    if(c == cpus+cpunum())  // We've started already.
      continue;

    // Tell entryother.S what stack to use, where to enter, and what
    // pgdir to use. We cannot use kpgdir yet, because the AP processor
    // is running in low  memory, so we use entrypgdir for the APs too.
    stack = kalloc();
    *(void**)(code-4) = stack + KSTACKSIZE;
    *(void**)(code-8) = mpenter;
    *(int**)(code-12) = (void *) V2P(entrypgdir);

    lapicstartap(c->apicid, V2P(code));
    // wait for cpu to finish mpmain()
    while(c->started == 0)
      ;
  }
}

// The boot page table used in entry.S and entryother.S.
// Page directories (and page tables) must start on page boundaries,
// hence the __aligned__ attribute.
// PTE_PS in a page directory entry enables 4Mbyte pages.

__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

```
// Blank page.
```

```c
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

```c
// Mutual exclusion spin locks.

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

void
initlock(struct spinlock *lk, char *name)
{
  lk->name = name;
  lk->locked = 0;
  lk->cpu = 0;
}

// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = cpu;
  getcallerpcs(&lk, lk->pcs);
}
```

```c
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other cores before the lock is released.
  // Both the C compiler and the hardware may re-order loads and
  // stores; __sync_synchronize() tells them both to not re-order.
  __sync_synchronize();

  // Release the lock.
  lk->locked = 0;

  popcli();
}

// Record the current call stack in pcs[] by following the %ebp chain.
void
getcallerpcs(void *v, uint pcs[])
{
  uint *ebp;
  int i;

  ebp = (uint*)v - 2;
  for(i = 0; i < 10; i++){
    if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
      break;
    pcs[i] = ebp[1];     // saved %eip
    ebp = (uint*)ebp[0]; // saved %ebp
  }
  for(; i < 10; i++)
    pcs[i] = 0;
}

// Check whether this cpu is holding the lock.
int
holding(struct spinlock *lock)
{
  return lock->locked && lock->cpu == cpu;
}
```

```c
// Pushcli/popcli are like cli/sti except that they are matched:
// it takes two popcli to undo two pushcli.  Also, if interrupts
// are off, then pushcli, popcli leaves them off.

void
pushcli(void)
{
  int eflags;

  eflags = readeflags();
  cli();
  if(cpu->ncli == 0)
    cpu->intena = eflags & FL_IF;
  cpu->ncli += 1;
}

void
popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--cpu->ncli < 0)
    panic("popcli");
  if(cpu->ncli == 0 && cpu->intena)
    sti();
}
```

```c
#include "param.h"
#include "types.h"
#include "defs.h"
#include "x86.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "elf.h"

extern char data[];  // defined by kernel.ld
pde_t *kpgdir;  // for use in scheduler()

// Set up CPU's kernel segment descriptors.
// Run once on entry on each CPU.
void
seginit(void)
{
  struct cpu *c;

  // Map "logical" addresses to virtual addresses using identity map.
  // Cannot share a CODE descriptor for both kernel and user
  // because it would have to have DPL_USR, but the CPU forbids
  // an interrupt from CPL=0 to DPL=3.
  c = &cpus[cpunum()];
  c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
  c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);

  // Map cpu and proc -- these are private per cpu.
  c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);

  lgdt(c->gdt, sizeof(c->gdt));
  loadgs(SEG_KCPU << 3);

  // Initialize cpu-local storage.
  cpu = c;
  proc = 0;
}
```

```c
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}

// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

```
// There is one page table per process, plus one that's used when
// a CPU is not running any process (kpgdir). The kernel uses the
// current process's page table during system calls and interrupts;
// page protection bits prevent user code from using the kernel's
// mappings.
//
// setupkvm() and exec() set up every page table like this:
//
//   0..KERNBASE: user memory (text+data+stack+heap), mapped to
//                phys memory allocated by the kernel
//   KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
//   KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
//                for the kernel's instructions and r/o data
//   data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
//                                  rw data + free physical memory
//   0xfe000000..0: mapped direct (devices such as ioapic)
//
// The kernel allocates physical memory for its heap and for user memory
// between V2P(end) and the end of physical memory (PHYSTOP)
// (directly addressable from end..P2V(PHYSTOP)).

// This table defines the kernel's mappings, which are present in
// every process's page table.
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
};

// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
```

```
      return 0;
  return pgdir;
}

// Allocate one page table for the machine for the kernel address
// space for scheduler processes.
void
kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}

// Switch h/w page table register to the kernel-only page table,
// for when no process is running.
void
switchkvm(void)
{
  lcr3(V2P(kpgdir));   // switch to the kernel page table
}

// Switch TSS and h/w page table to correspond to process p.
void
switchuvm(struct proc *p)
{
  pushcli();
  cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
  cpu->gdt[SEG_TSS].s = 0;
  cpu->ts.ss0 = SEG_KDATA << 3;
  cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
  // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
  // forbids I/O instructions (e.g., inb and outb) from user space
  cpu->ts.iomb = (ushort) 0xFFFF;
  ltr(SEG_TSS << 3);
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");
  lcr3(V2P(p->pgdir));  // switch to process's address space
  popcli();
}
```

```
// Load the initcode into address 0 of pgdir.
// sz must be less than a page.
void
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W|PTE_U);
  memmove(mem, init, sz);
}

// Load a program segment into pgdir.  addr must be page-aligned
// and the pages from addr to addr+sz must already be mapped.
int
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
  uint i, pa, n;
  pte_t *pte;

  if((uint) addr % PGSIZE != 0)
    panic("loaduvm: addr must be page aligned");
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

```
// Allocate page tables and physical memory to grow process from oldsz to
// newsz, which need not be page aligned.  Returns new size or 0 on error.
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;

  if(newsz >= KERNBASE)
    return 0;
  if(newsz < oldsz)
    return oldsz;

  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
  return newsz;
}

// Deallocate user pages to bring the process size from oldsz to
// newsz.  oldsz and newsz need not be page-aligned, nor does newsz
// need to be less than oldsz.  oldsz can be larger than the actual
// process size.  Returns the new process size.
int
deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  pte_t *pte;
  uint a, pa;

  if(newsz >= oldsz)
    return oldsz;

  a = PGROUNDUP(newsz);
  for(; a  < oldsz; a += PGSIZE){
    pte = walkpgdir(pgdir, (char*)a, 0);
    if(!pte)
      a += (NPTENTRIES - 1) * PGSIZE;
    else if((*pte & PTE_P) != 0){
      pa = PTE_ADDR(*pte);
      if(pa == 0)
        panic("kfree");
      char *v = P2V(pa);
      kfree(v);
      *pte = 0;
    }
  }
  return newsz;
}

// Free a page table and all the physical memory pages
// in the user part.
void
freevm(pde_t *pgdir)
{
  uint i;

  if(pgdir == 0)
    panic("freevm: no pgdir");
  deallocuvm(pgdir, KERNBASE, 0);
  for(i = 0; i < NPDENTRIES; i++){
    if(pgdir[i] & PTE_P){
      char * v = P2V(PTE_ADDR(pgdir[i]));
      kfree(v);
    }
  }
  kfree((char*)pgdir);
}

// Clear PTE_U on a page. Used to create an inaccessible
// page beneath the user stack.
void
clearpteu(pde_t *pgdir, char *uva)
{
  pte_t *pte;

  pte = walkpgdir(pgdir, uva, 0);
  if(pte == 0)
    panic("clearpteu");
  *pte &= ~PTE_U;
}
```

```
// Given a parent process's page table, create a copy
// of it for a child.
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
      goto bad;
  }
  return d;

bad:
  freevm(d);
  return 0;
}
```

```
// Map user virtual address to kernel address.
char*
uva2ka(pde_t *pgdir, char *uva)
{
  pte_t *pte;

  pte = walkpgdir(pgdir, uva, 0);
  if((*pte & PTE_P) == 0)
    return 0;
  if((*pte & PTE_U) == 0)
    return 0;
  return (char*)P2V(PTE_ADDR(*pte));
}

// Copy len bytes from p to user address va in page table pgdir.
// Most useful when pgdir is not the current page table.
// uva2ka ensures this only works for PTE_U pages.
int
copyout(pde_t *pgdir, uint va, void *p, uint len)
{
  char *buf, *pa0;
  uint n, va0;

  buf = (char*)p;
  while(len > 0){
    va0 = (uint)PGROUNDDOWN(va);
    pa0 = uva2ka(pgdir, (char*)va0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (va - va0);
    if(n > len)
      n = len;
    memmove(pa0 + (va - va0), buf, n);
    len -= n;
    buf += n;
    va = va0 + PGSIZE;
  }
  return 0;
}
```

```
// Blank page.
// Blank page.
```

```
// Blank page.
// Per-CPU state
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?

  // Cpu-local storage variables; see below
  struct cpu *cpu;
  struct proc *proc;           // The currently-running process.
};

extern struct cpu cpus[NCPU];
extern int ncpu;

// Per-CPU variables, holding pointers to the
// current cpu and to the current process.
// The asm suffix tells gcc to use "%gs:0" to refer to cpu
// and "%gs:4" to refer to proc.  seginit sets up the
// %gs segment register so that %gs refers to the memory
// holding those two variables in the local cpu's struct cpu.
// This is similar to how thread-local variables are implemented
// in thread libraries such as Linux pthreads.
extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc


// Saved registers for kernel context switches.
// Don't need to save all the segment registers (%cs, etc),
// because they are constant across kernel contexts.
// Don't need to save %eax, %ecx, %edx, because the
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom of the stack they
// describe; the stack pointer is the address of the context.
// The layout of the context matches the layout of the stack in swtch.S
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,
// but it is on the stack and allocproc() manipulates it.
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};

// Process memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap
```

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

struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;

static struct proc *initproc;

int nextpid = 1;
extern void forkret(void);
extern void trapret(void);

static void wakeup1(void *chan);

void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
}
```

```
// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
// Must hold ptable.lock.
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;

  return p;
}
```

```
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  acquire(&ptable.lock);

  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  p->state = RUNNABLE;

  release(&ptable.lock);
}

// Grow current process's memory by n bytes.
// Return 0 on success, -1 on failure.
int
growproc(int n)
{
  uint sz;

  sz = proc->sz;
  if(n > 0){
    if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
      return -1;
  } else if(n < 0){
    if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  proc->sz = sz;
  switchuvm(proc);
  return 0;
```

```
}

// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
int
fork(void)
{
  int i, pid;
  struct proc *np;

  acquire(&ptable.lock);

  // Allocate process.
  if((np = allocproc()) == 0){
    release(&ptable.lock);
    return -1;
  }

  // Copy process state from p.
  if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    release(&ptable.lock);
    return -1;
  }
  np->sz = proc->sz;
  np->parent = proc;
  *np->tf = *proc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(proc->ofile[i])
      np->ofile[i] = filedup(proc->ofile[i]);
  np->cwd = idup(proc->cwd);

  safestrcpy(np->name, proc->name, sizeof(proc->name));

  pid = np->pid;

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}

```

```c
// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait() to find out it exited.
void
exit(void)
{
  struct proc *p;
  int fd;

  if(proc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(proc->ofile[fd]){
      fileclose(proc->ofile[fd]);
      proc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(proc->cwd);
  end_op();
  proc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(proc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == proc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  proc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

```c
// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
int
wait(void)
{
  struct proc *p;
  int havekids, pid;

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for zombie children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != proc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || proc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(proc, &ptable.lock);
  }
}
```

```c
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
```

walk list of processes

```c
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
```

load context of the target process

```c
      p->state = RUNNING;
      swtch(&cpu->scheduler, p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);

  }
}
```

```c
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state. Saves and restores
// intena because intena is a property of this
// kernel thread, not this CPU. It should
// be proc->intena and proc->ncli, but that would
// break in the few places where a lock is held but
// there's no process.
void
sched(void)
```

save the current process and load the context of the scheduler

```c
{
  int intena;

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(cpu->ncli != 1)
    panic("sched locks");
  if(proc->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = cpu->intena;
  swtch(&proc->context, cpu->scheduler);
  cpu->intena = intena;
}

// Give up the CPU for one scheduling round.
void
yield(void)
```

to yield it has to make it runnable, then calls a routine sched(()

```c
{
  acquire(&ptable.lock);
  proc->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}

// A fork child's very first scheduling by scheduler()
// will swtch here.  "Return" to user space.
void
forkret(void)
{
  static int first = 1;
  // Still holding ptable.lock from scheduler.
  release(&ptable.lock);

  if (first) {
    // Some initialization functions must be run in the context
    // of a regular process (e.g., they call sleep), and thus cannot
    // be run from main().
    first = 0;
    iinit(ROOTDEV);
    initlog(ROOTDEV);
  }

  // Return to "caller", actually trapret (see allocproc).
}

// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
sleep(void *chan, struct spinlock *lk)
{
  if(proc == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }

  // Go to sleep.
  proc->chan = chan;
  proc->state = SLEEPING;
  sched();

  // Tidy up.
  proc->chan = 0;

  // Reacquire original lock.
  if(lk != &ptable.lock){
    release(&ptable.lock);
    acquire(lk);
  }
}
```

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

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

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

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

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

walking all the processes

we can have a list of all the processes that are waiting

```
// Print a process listing to console.  For debugging.
// Runs when user types ^P on console.
// No lock to avoid wedging a stuck machine further.
void
procdump(void)
{
  static char *states[] = {
  [UNUSED]    "unused",
  [EMBRYO]    "embryo",
  [SLEEPING]  "sleep ",
  [RUNNABLE]  "runble",
  [RUNNING]   "run   ",
  [ZOMBIE]    "zombie"
  };
  int i;
  struct proc *p;
  char *state;
  uint pc[10];

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;
    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
      state = states[p->state];
    else
      state = "???";
    cprintf("%d %s %s", p->pid, state, p->name);
    if(p->state == SLEEPING){
      getcallerpcs((uint*)p->context->ebp+2, pc);
      for(i=0; i<10 && pc[i] != 0; i++)
        cprintf(" %p", pc[i]);
    }
    cprintf("\n");
  }
}
```

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.

.globl swtch  called by scheduler, sched
swtch:
  movl 4(%esp), %eax   set eax to contain adress of old context
  movl 8(%esp), %edx   set edx to contain new context

  # Save old callee-save registers
  pushl %ebp                    caller saved already saved
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)   copy
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

```c
// Physical memory allocator, intended to allocate
// memory for user processes, kernel stacks, page table pages,
// and pipe buffers. Allocates 4096-byte pages.

#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "spinlock.h"

void freerange(void *vstart, void *vend);
extern char end[]; // first address after kernel loaded from ELF file

struct run {
  struct run *next;
};

struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
} kmem;

// Initialization happens in two phases.
// 1. main() calls kinit1() while still using entrypgdir to place just
// the pages mapped by entrypgdir on free list.
// 2. main() calls kinit2() with the rest of the physical pages
// after installing a full page table that maps them on all cores.
void
kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;
  freerange(vstart, vend);
}

void
kinit2(void *vstart, void *vend)
{
  freerange(vstart, vend);
  kmem.use_lock = 1;
}
```

```c
void
freerange(void *vstart, void *vend)
{
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
    kfree(p);
}


// Free the page of physical memory pointed at by v,
// which normally should have been returned by a
// call to kalloc().  (The exception is when
// initializing the allocator; see kinit above.)
void
kfree(char *v)
{
  struct run *r;

  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock);
}

// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

```
// x86 trap and interrupt constants.

// Processor-defined:
#define T_DIVIDE         0      // divide error
#define T_DEBUG          1      // debug exception
#define T_NMI            2      // non-maskable interrupt
#define T_BRKPT          3      // breakpoint
#define T_OFLOW          4      // overflow
#define T_BOUND          5      // bounds check
#define T_ILLOP          6      // illegal opcode
#define T_DEVICE         7      // device not available
#define T_DBLFLT         8      // double fault
// #define T_COPROC      9      // reserved (not used since 486)
#define T_TSS           10      // invalid task switch segment
#define T_SEGNP         11      // segment not present
#define T_STACK         12      // stack exception
#define T_GPFLT         13      // general protection fault
#define T_PGFLT         14      // page fault
// #define T_RES        15      // reserved
#define T_FPERR         16      // floating point error
#define T_ALIGN         17      // aligment check
#define T_MCHK          18      // machine check
#define T_SIMDERR       19      // SIMD floating point error

// These are arbitrarily chosen, but with care not to overlap
// processor defined exceptions or interrupt vectors.
#define T_SYSCALL       64      // system call
#define T_DEFAULT      500      // catchall

#define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ

#define IRQ_TIMER        0
#define IRQ_KBD          1
#define IRQ_COM1         4
#define IRQ_IDE         14
#define IRQ_ERROR       19
#define IRQ_SPURIOUS    31
```

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

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

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

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

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

```
#include "mmu.h"

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

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

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

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

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

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

void
tvinit(void)
{
  int i;

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

  initlock(&tickslock, "time");
}

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

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

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpunum() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE+1:
    // Bochs generates spurious IDE1 interrupts.
    break;
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_COM1:
    uartintr();
    lapiceoi();
    break;
  case T_IRQ0 + 7:
  case T_IRQ0 + IRQ_SPURIOUS:
    cprintf("cpu%d: spurious interrupt at %x:%x\n",
            cpunum(), tf->cs, tf->eip);
    lapiceoi();
    break;
  default:
    if(proc == 0 || (tf->cs&3) == 0){
      // In kernel, it must be our mistake.
      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
              tf->trapno, cpunum(), tf->eip, rcr2());
      panic("trap");
    }
    // In user space, assume process misbehaved.
    cprintf("pid %d %s: trap %d err %d on cpu %d "
            "eip 0x%x addr 0x%x--kill proc\n",
            proc->pid, proc->name, tf->trapno, tf->err, cpunum(), tf->eip,
            rcr2());
    proc->killed = 1;
  }

  // Force process exit if it has been killed and is in user space.
  // (If it is still executing in the kernel, let it keep running
  // until it gets to the regular system call return.)
  if(proc && proc->killed && (tf->cs&3) == DPL_USER)
    exit();

  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();

  // Check if the process has been killed since we yielded
  if(proc && proc->killed && (tf->cs&3) == DPL_USER)
    exit();
}
```

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
```

```c
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "syscall.h"

// User code makes a system call with INT T_SYSCALL.
// System call number in %eax.
// Arguments on the stack, from the user call to the C
// library system call function. The saved user %esp points
// to a saved program counter, and then the first argument.

// Fetch the int at addr from the current process.
int
fetchint(uint addr, int *ip)
{
  if(addr >= proc->sz || addr+4 > proc->sz)
    return -1;
  *ip = *(int*)(addr);
  return 0;
}

// Fetch the nul-terminated string at addr from the current process.
// Doesn't actually copy the string - just sets *pp to point at it.
// Returns length of string, not including nul.
int
fetchstr(uint addr, char **pp)
{
  char *s, *ep;

  if(addr >= proc->sz)
    return -1;
  *pp = (char*)addr;
  ep = (char*)proc->sz;
  for(s = *pp; s < ep; s++)
    if(*s == 0)
      return s - *pp;
  return -1;
}

// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
  return fetchint(proc->tf->esp + 4 + 4*n, ip);
}

```

```c
// Fetch the nth word-sized system call argument as a pointer
// to a block of memory of size n bytes.  Check that the pointer
// lies within the process address space.
int
argptr(int n, char **pp, int size)
{
  int i;

  if(argint(n, &i) < 0)
    return -1;
  if((uint)i >= proc->sz || (uint)i+size > proc->sz)
    return -1;
  *pp = (char*)i;
  return 0;
}

// Fetch the nth word-sized system call argument as a string pointer.
// Check that the pointer is valid and the string is nul-terminated.
// (There is no shared writable memory, so the string can't change
// between this check and being used by the kernel.)
int
argstr(int n, char **pp)
{
  int addr;
  if(argint(n, &addr) < 0)
    return -1;
  return fetchstr(addr, pp);
}

extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
```

```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
};

void
syscall(void)
{
  int num;

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```

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

int
sys_fork(void)
{
  return fork();
}

int
sys_exit(void)
{
  exit();
  return 0;  // not reached
}

int
sys_wait(void)
{
  return wait();
}

int
sys_kill(void)
{
  int pid;

  if(argint(0, &pid) < 0)
    return -1;
  return kill(pid);
}

int
sys_getpid(void)
{
  return proc->pid;
}
```

```
int
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;
  addr = proc->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}

int
sys_sleep(void)
{
  int n;
  uint ticks0;

  if(argint(0, &n) < 0)
    return -1;
  acquire(&tickslock);
  ticks0 = ticks;
  while(ticks - ticks0 < n){
    if(proc->killed){
      release(&tickslock);
      return -1;
    }
    sleep(&ticks, &tickslock);
  }
  release(&tickslock);
  return 0;
}

// return how many clock tick interrupts have occurred
// since start.
int
sys_uptime(void)
{
  uint xticks;

  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
}
```

```
struct buf {
  int flags;
  uint dev;
  uint blockno;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};
#define B_BUSY  0x1  // buffer is locked by some process
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

```
#define O_RDONLY  0x000
#define O_WRONLY  0x001
#define O_RDWR    0x002
#define O_CREATE  0x200
```

```
#define T_DIR  1   // Directory
#define T_FILE 2   // File
#define T_DEV  3   // Device

struct stat {
  short type;  // Type of file
  int dev;     // File system's disk device
  uint ino;    // Inode number
  short nlink; // Number of links to file
  uint size;   // Size of file in bytes
};
```

```
// On-disk file system format.
// Both the kernel and user programs use this header file.


#define ROOTINO 1  // root i-number
#define BSIZE 512  // block size

// Disk layout:
// [ boot block | super block | log | inode blocks |
//                                          free bit map | data blocks]
//
// mkfs computes the super block and builds an initial file system. The
// super block describes the disk layout:
struct superblock {
  uint size;         // Size of file system image (blocks)
  uint nblocks;      // Number of data blocks
  uint ninodes;      // Number of inodes.
  uint nlog;         // Number of log blocks
  uint logstart;     // Block number of first log block
  uint inodestart;   // Block number of first inode block
  uint bmapstart;    // Block number of first free map block
};

#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT)

// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};
```

```
// Inodes per block.
#define IPB           (BSIZE / sizeof(struct dinode))

// Block containing inode i
#define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)

// Bitmap bits per block
#define BPB           (BSIZE*8)

// Block of free map containing bit for block b
#define BBLOCK(b, sb) (b/BPB + sb.bmapstart)

// Directory is a file containing a sequence of dirent structures.
#define DIRSIZ 14

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};


// in-memory copy of an inode
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  int flags;          // I_BUSY, I_VALID

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
#define I_BUSY 0x1
#define I_VALID 0x2

// table mapping major device number to
// device functions
struct devsw {
  int (*read)(struct inode*, char*, int);
  int (*write)(struct inode*, char*, int);
};

extern struct devsw devsw[];

#define CONSOLE 1
```

```
// Blank page.
// Simple PIO-based (non-DMA) IDE driver code.

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

#define SECTOR_SIZE   512
#define IDE_BSY       0x80
#define IDE_DRDY      0x40
#define IDE_DF        0x20
#define IDE_ERR       0x01

#define IDE_CMD_READ  0x20
#define IDE_CMD_WRITE 0x30
#define IDE_CMD_RDMUL 0xc4
#define IDE_CMD_WRMUL 0xc5

// idequeue points to the buf now being read/written to the disk.
// idequeue->qnext points to the next buf to be processed.
// You must hold idelock while manipulating queue.

static struct spinlock idelock;
static struct buf *idequeue;

static int havedisk1;
static void idestart(struct buf*);

// Wait for IDE disk to become ready.
static int
idewait(int checkerr)
{
  int r;

  while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
    ;
  if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
    return -1;
  return 0;
}
```

```
void
ideinit(void)
{
  int i;

  initlock(&idelock, "ide");
  picenable(IRQ_IDE);
  ioapicenable(IRQ_IDE, ncpu - 1);
  idewait(0);

  // Check if disk 1 is present
  outb(0x1f6, 0xe0 | (1<<4));
  for(i=0; i<1000; i++){
    if(inb(0x1f7) != 0){
      havedisk1 = 1;
      break;
    }
  }

  // Switch back to disk 0.
  outb(0x1f6, 0xe0 | (0<<4));
}

// Start the request for b.  Caller must hold idelock.
static void
idestart(struct buf *b)
{
  if(b == 0)
    panic("idestart");
  if(b->blockno >= FSSIZE)
    panic("incorrect blockno");
  int sector_per_block =  BSIZE/SECTOR_SIZE;
  int sector = b->blockno * sector_per_block;
  int read_cmd = (sector_per_block == 1) ? IDE_CMD_READ :  IDE_CMD_RDMUL;
  int write_cmd = (sector_per_block == 1) ? IDE_CMD_WRITE : IDE_CMD_WRMUL;

  if (sector_per_block > 7) panic("idestart");

  idewait(0);
  outb(0x3f6, 0);  // generate interrupt
  outb(0x1f2, sector_per_block);  // number of sectors
  outb(0x1f3, sector & 0xff);
  outb(0x1f4, (sector >> 8) & 0xff);
  outb(0x1f5, (sector >> 16) & 0xff);
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, write_cmd);
    outsl(0x1f0, b->data, BSIZE/4);
  } else {
    outb(0x1f7, read_cmd);
  }
}

// Interrupt handler.
void
ideintr(void)
{
  struct buf *b;

  // First queued buffer is the active request.
  acquire(&idelock);
  if((b = idequeue) == 0){
    release(&idelock);
    // cprintf("spurious IDE interrupt\n");
    return;
  }
  idequeue = b->qnext;

  // Read data if needed.
  if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
    insl(0x1f0, b->data, BSIZE/4);

  // Wake process waiting for this buf.
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);

  // Start disk on next buf in queue.
  if(idequeue != 0)
    idestart(idequeue);

  release(&idelock);
}
```

```
// Sync buf with disk.
// If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
// Else if B_VALID is not set, read buf from disk, set B_VALID.
void
iderw(struct buf *b)
{
  struct buf **pp;

  if(!(b->flags & B_BUSY))
    panic("iderw: buf not busy");
  if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
    panic("iderw: nothing to do");
  if(b->dev != 0 && !havedisk1)
    panic("iderw: ide disk 1 not present");

  acquire(&idelock);

  // Append b to idequeue.
  b->qnext = 0;
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
    ;
  *pp = b;

  // Start disk if necessary.
  if(idequeue == b)
    idestart(b);

  // Wait for request to finish.
  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
    sleep(b, &idelock);
  }

  release(&idelock);
}
```

```
// Buffer cache.
//
// The buffer cache is a linked list of buf structures holding
// cached copies of disk block contents.  Caching disk blocks
// in memory reduces the number of disk reads and also provides
// a synchronization point for disk blocks used by multiple processes.
//
// Interface:
// * To get a buffer for a particular disk block, call bread.
// * After changing buffer data, call bwrite to write it to disk.
// * When done with the buffer, call brelse.
// * Do not use the buffer after calling brelse.
// * Only one process at a time can use a buffer,
//     so do not keep them longer than necessary.
//
// The implementation uses three state flags internally:
// * B_BUSY: the block has been returned from bread
//     and has not been passed back to brelse.
// * B_VALID: the buffer data has been read from the disk.
// * B_DIRTY: the buffer data has been modified
//     and needs to be written to disk.

#include "types.h"
#include "defs.h"
#include "param.h"
#include "spinlock.h"
#include "fs.h"
#include "buf.h"

struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // head.next is most recently used.
  struct buf head;
} bcache;

void
binit(void)
{
  struct buf *b;

  initlock(&bcache.lock, "bcache");
```

```
  // Create linked list of buffers
  bcache.head.prev = &bcache.head;
  bcache.head.next = &bcache.head;
  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    b->next = bcache.head.next;
    b->prev = &bcache.head;
    b->dev = -1;
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }
}

// Look through buffer cache for block on device dev.
// If not found, allocate a buffer.
// In either case, return B_BUSY buffer.
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;

  acquire(&bcache.lock);

 loop:
  // Is the block already cached?
  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      if(!(b->flags & B_BUSY)){
        b->flags |= B_BUSY;
        release(&bcache.lock);
        return b;
      }
      sleep(b, &bcache.lock);
      goto loop;
    }
  }

  // Not cached; recycle some non-busy and clean buffer.
  // "clean" because B_DIRTY and !B_BUSY means log.c
  // hasn't yet committed the changes to the buffer.
  for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
    if((b->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
      b->dev = dev;
      b->blockno = blockno;
      b->flags = B_BUSY;
      release(&bcache.lock);
      return b;
    }
  }
  panic("bget: no buffers");
}
```

```
// Return a B_BUSY buf with the contents of the indicated block.
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;

  b = bget(dev, blockno);
  if(!(b->flags & B_VALID)) {
    iderw(b);
  }
  return b;
}

// Write b's contents to disk.  Must be B_BUSY.
void
bwrite(struct buf *b)
{
  if((b->flags & B_BUSY) == 0)
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}

// Release a B_BUSY buffer.
// Move to the head of the MRU list.
void
brelse(struct buf *b)
{
  if((b->flags & B_BUSY) == 0)
    panic("brelse");

  acquire(&bcache.lock);

  b->next->prev = b->prev;
  b->prev->next = b->next;
  b->next = bcache.head.next;
  b->prev = &bcache.head;
  bcache.head.next->prev = b;
  bcache.head.next = b;

  b->flags &= ~B_BUSY;
  wakeup(b);

  release(&bcache.lock);
}
```

```
// Blank page.
```

```c
#include "types.h"
#include "defs.h"
#include "param.h"
#include "spinlock.h"
#include "fs.h"
#include "buf.h"

// Simple logging that allows concurrent FS system calls.
//
// A log transaction contains the updates of multiple FS system
// calls. The logging system only commits when there are
// no FS system calls active. Thus there is never
// any reasoning required about whether a commit might
// write an uncommitted system call’s updates to disk.
//
// A system call should call begin_op()/end_op() to mark
// its start and end. Usually begin_op() just increments
// the count of in-progress FS system calls and returns.
// But if it thinks the log is close to running out, it
// sleeps until the last outstanding end_op() commits.
//
// The log is a physical re-do log containing disk blocks.
// The on-disk log format:
//   header block, containing block #s for block A, B, C, ...
//   block A
//   block B
//   block C
//   ...
// Log appends are synchronous.

// Contents of the header block, used for both the on-disk header block
// and to keep track in memory of logged block# before commit.
struct logheader {
  int n;
  int block[LOGSIZE];
};

struct log {
  struct spinlock lock;
  int start;
  int size;
  int outstanding; // how many FS sys calls are executing.
  int committing;  // in commit(), please wait.
  int dev;
  struct logheader lh;
};
```

```c
struct log log;

static void recover_from_log(void);
static void commit();

void
initlog(int dev)
{
  if (sizeof(struct logheader) >= BSIZE)
    panic("initlog: too big logheader");

  struct superblock sb;
  initlock(&log.lock, "log");
  readsb(dev, &sb);
  log.start = sb.logstart;
  log.size = sb.nlog;
  log.dev = dev;
  recover_from_log();
}

// Copy committed blocks from log to their home location
static void
install_trans(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
    struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
    memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
    bwrite(dbuf);  // write dst to disk
    brelse(lbuf);
    brelse(dbuf);
  }
}

// Read the log header from disk into the in-memory log header
static void
read_head(void)
{
  struct buf *buf = bread(log.dev, log.start);
  struct logheader *lh = (struct logheader *) (buf->data);
  int i;
  log.lh.n = lh->n;
  for (i = 0; i < log.lh.n; i++) {
    log.lh.block[i] = lh->block[i];
  }
  brelse(buf);
}
```

```
// Write in-memory log header to disk.
// This is the true point at which the
// current transaction commits.
static void
write_head(void)
{
  struct buf *buf = bread(log.dev, log.start);
  struct logheader *hb = (struct logheader *) (buf->data);
  int i;
  hb->n = log.lh.n;
  for (i = 0; i < log.lh.n; i++) {
    hb->block[i] = log.lh.block[i];
  }
  bwrite(buf);
  brelse(buf);
}

static void
recover_from_log(void)
{
  read_head();
  install_trans(); // if committed, copy from log to disk
  log.lh.n = 0;
  write_head(); // clear the log
}

// called at the start of each FS system call.
void
begin_op(void)
{
  acquire(&log.lock);
  while(1){
    if(log.committing){
      sleep(&log, &log.lock);
    } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
      // this op might exhaust log space; wait for commit.
      sleep(&log, &log.lock);
    } else {
      log.outstanding += 1;
      release(&log.lock);
      break;
    }
  }
}
```

```
// called at the end of each FS system call.
// commits if this was the last outstanding operation.
void
end_op(void)
{
  int do_commit = 0;

  acquire(&log.lock);
  log.outstanding -= 1;
  if(log.committing)
    panic("log.committing");
  if(log.outstanding == 0){
    do_commit = 1;
    log.committing = 1;
  } else {
    // begin_op() may be waiting for log space.
    wakeup(&log);
  }
  release(&log.lock);

  if(do_commit){
    // call commit w/o holding locks, since not allowed
    // to sleep with locks.
    commit();
    acquire(&log.lock);
    log.committing = 0;
    wakeup(&log);
    release(&log.lock);
  }
}

// Copy modified blocks from cache to log.
static void
write_log(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *to = bread(log.dev, log.start+tail+1); // log block
    struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
    memmove(to->data, from->data, BSIZE);
    bwrite(to);  // write the log
    brelse(from);
    brelse(to);
  }
}
```

```
static void
commit()
{
  if (log.lh.n > 0) {
    write_log();     // Write modified blocks from cache to log
    write_head();    // Write header to disk -- the real commit
    install_trans(); // Now install writes to home locations
    log.lh.n = 0;
    write_head();    // Erase the transaction from the log
  }
}

// Caller has modified b->data and is done with the buffer.
// Record the block number and pin in the cache with B_DIRTY.
// commit()/write_log() will do the disk write.
//
// log_write() replaces bwrite(); a typical use is:
//   bp = bread(...)
//   modify bp->data[]
//   log_write(bp)
//   brelse(bp)
void
log_write(struct buf *b)
{
  int i;

  if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
    panic("too big a transaction");
  if (log.outstanding < 1)
    panic("log_write outside of trans");

  acquire(&log.lock);
  for (i = 0; i < log.lh.n; i++) {
    if (log.lh.block[i] == b->blockno)   // log absorbtion
      break;
  }
  log.lh.block[i] = b->blockno;
  if (i == log.lh.n)
    log.lh.n++;
  b->flags |= B_DIRTY; // prevent eviction
  release(&log.lock);
}
```

```
// File system implementation.  Five layers:
//   + Blocks: allocator for raw disk blocks.
//   + Log: crash recovery for multi-step updates.
//   + Files: inode allocator, reading, writing, metadata.
//   + Directories: inode with special contents (list of other inodes!)
//   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
//
// This file contains the low-level file system manipulation
// routines.  The (higher-level) system call implementations
// are in sysfile.c.

#include "types.h"
#include "defs.h"
#include "param.h"
#include "stat.h"
#include "mmu.h"
#include "proc.h"
#include "spinlock.h"
#include "fs.h"
#include "buf.h"
#include "file.h"

#define min(a, b) ((a) < (b) ? (a) : (b))
static void itrunc(struct inode*);
// there should be one superblock per disk device, but we run with
// only one device
struct superblock sb;

// Read the super block.
void
readsb(int dev, struct superblock *sb)
{
  struct buf *bp;

  bp = bread(dev, 1);
  memmove(sb, bp->data, sizeof(*sb));
  brelse(bp);
}

// Zero a block.
static void
bzero(int dev, int bno)
{
  struct buf *bp;

  bp = bread(dev, bno);
  memset(bp->data, 0, BSIZE);
  log_write(bp);
  brelse(bp);
}
```

```
// Blocks.

// Allocate a zeroed disk block.
static uint
balloc(uint dev)
{
  int b, bi, m;
  struct buf *bp;

  bp = 0;
  for(b = 0; b < sb.size; b += BPB){
    bp = bread(dev, BBLOCK(b, sb));
    for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
      m = 1 << (bi % 8);
      if((bp->data[bi/8] & m) == 0){  // Is block free?
        bp->data[bi/8] |= m;  // Mark block in use.
        log_write(bp);
        brelse(bp);
        bzero(dev, b + bi);
        return b + bi;
      }
    }
    brelse(bp);
  }
  panic("balloc: out of blocks");
}

// Free a disk block.
static void
bfree(int dev, uint b)
{
  struct buf *bp;
  int bi, m;

  readsb(dev, &sb);
  bp = bread(dev, BBLOCK(b, sb));
  bi = b % BPB;
  m = 1 << (bi % 8);
  if((bp->data[bi/8] & m) == 0)
    panic("freeing free block");
  bp->data[bi/8] &= ~m;
  log_write(bp);
  brelse(bp);
}
```

```
// Inodes.
//
// An inode describes a single unnamed file.
// The inode disk structure holds metadata: the file's type,
// its size, the number of links referring to it, and the
// list of blocks holding the file's content.
//
// The inodes are laid out sequentially on disk at
// sb.startinode. Each inode has a number, indicating its
// position on the disk.
//
// The kernel keeps a cache of in-use inodes in memory
// to provide a place for synchronizing access
// to inodes used by multiple processes. The cached
// inodes include book-keeping information that is
// not stored on disk: ip->ref and ip->flags.
//
// An inode and its in-memory represtative go through a
// sequence of states before they can be used by the
// rest of the file system code.
//
// * Allocation: an inode is allocated if its type (on disk)
//   is non-zero. ialloc() allocates, iput() frees if
//   the link count has fallen to zero.
//
// * Referencing in cache: an entry in the inode cache
//   is free if ip->ref is zero. Otherwise ip->ref tracks
//   the number of in-memory pointers to the entry (open
//   files and current directories). iget() to find or
//   create a cache entry and increment its ref, iput()
//   to decrement ref.
//
// * Valid: the information (type, size, &c) in an inode
//   cache entry is only correct when the I_VALID bit
//   is set in ip->flags. ilock() reads the inode from
//   the disk and sets I_VALID, while iput() clears
//   I_VALID if ip->ref has fallen to zero.
//
// * Locked: file system code may only examine and modify
//   the information in an inode and its content if it
//   has first locked the inode. The I_BUSY flag indicates
//   that the inode is locked. ilock() sets I_BUSY,
//   while iunlock clears it.
//
// Thus a typical sequence is:
//   ip = iget(dev, inum)
//   ilock(ip)
//   ... examine and modify ip->xxx ...
//   iunlock(ip)
//   iput(ip)
```

```
//
// ilock() is separate from iget() so that system calls can
// get a long-term reference to an inode (as for an open file)
// and only lock it for short periods (e.g., in read()).
// The separation also helps avoid deadlock and races during
// pathname lookup. iget() increments ip->ref so that the inode
// stays cached and pointers to it remain valid.
//
// Many internal file system functions expect the caller to
// have locked the inodes involved; this lets callers create
// multi-step atomic operations.

struct {
  struct spinlock lock;
  struct inode inode[NINODE];
} icache;

void
iinit(int dev)
{
  initlock(&icache.lock, "icache");
  readsb(dev, &sb);
  cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d\
          inodestart %d bmap start %d\n", sb.size, sb.nblocks,
          sb.ninodes, sb.nlog, sb.logstart, sb.inodestart,
          sb.bmapstart);
}

static struct inode* iget(uint dev, uint inum);
```

```
// Allocate a new inode with the given type on device dev.
// A free inode has a type of zero.
struct inode*
ialloc(uint dev, short type)
{
  int inum;
  struct buf *bp;
  struct dinode *dip;

  for(inum = 1; inum < sb.ninodes; inum++){
    bp = bread(dev, IBLOCK(inum, sb));
    dip = (struct dinode*)bp->data + inum%IPB;
    if(dip->type == 0){  // a free inode
      memset(dip, 0, sizeof(*dip));
      dip->type = type;
      log_write(bp);   // mark it allocated on the disk
      brelse(bp);
      return iget(dev, inum);
    }
    brelse(bp);
  }
  panic("ialloc: no inodes");
}

// Copy a modified in-memory inode to disk.
void
iupdate(struct inode *ip)
{
  struct buf *bp;
  struct dinode *dip;

  bp = bread(ip->dev, IBLOCK(ip->inum, sb));
  dip = (struct dinode*)bp->data + ip->inum%IPB;
  dip->type = ip->type;
  dip->major = ip->major;
  dip->minor = ip->minor;
  dip->nlink = ip->nlink;
  dip->size = ip->size;
  memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
  log_write(bp);
  brelse(bp);
}
```

```
// Find the inode with number inum on device dev
// and return the in-memory copy. Does not lock
// the inode and does not read it from disk.
static struct inode*
iget(uint dev, uint inum)
{
  struct inode *ip, *empty;

  acquire(&icache.lock);

  // Is the inode already cached?
  empty = 0;
  for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
    if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
      ip->ref++;
      release(&icache.lock);
      return ip;
    }
    if(empty == 0 && ip->ref == 0)    // Remember empty slot.
      empty = ip;
  }

  // Recycle an inode cache entry.
  if(empty == 0)
    panic("iget: no inodes");

  ip = empty;
  ip->dev = dev;
  ip->inum = inum;
  ip->ref = 1;
  ip->flags = 0;
  release(&icache.lock);

  return ip;
}

// Increment reference count for ip.
// Returns ip to enable ip = idup(ip1) idiom.
struct inode*
idup(struct inode *ip)
{
  acquire(&icache.lock);
  ip->ref++;
  release(&icache.lock);
  return ip;
}
```

```
// Lock the given inode.
// Reads the inode from disk if necessary.
void
ilock(struct inode *ip)
{
  struct buf *bp;
  struct dinode *dip;

  if(ip == 0 || ip->ref < 1)
    panic("ilock");

  acquire(&icache.lock);
  while(ip->flags & I_BUSY)
    sleep(ip, &icache.lock);
  ip->flags |= I_BUSY;
  release(&icache.lock);

  if(!(ip->flags & I_VALID)){
    bp = bread(ip->dev, IBLOCK(ip->inum, sb));
    dip = (struct dinode*)bp->data + ip->inum%IPB;
    ip->type = dip->type;
    ip->major = dip->major;
    ip->minor = dip->minor;
    ip->nlink = dip->nlink;
    ip->size = dip->size;
    memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
    brelse(bp);
    ip->flags |= I_VALID;
    if(ip->type == 0)
      panic("ilock: no type");
  }
}

// Unlock the given inode.
void
iunlock(struct inode *ip)
{
  if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
    panic("iunlock");

  acquire(&icache.lock);
  ip->flags &= ~I_BUSY;
  wakeup(ip);
  release(&icache.lock);
}
```

```
// Drop a reference to an in-memory inode.
// If that was the last reference, the inode cache entry can
// be recycled.
// If that was the last reference and the inode has no links
// to it, free the inode (and its content) on disk.
// All calls to iput() must be inside a transaction in
// case it has to free the inode.
void
iput(struct inode *ip)
{
  acquire(&icache.lock);
  if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
    // inode has no links and no other references: truncate and free.
    if(ip->flags & I_BUSY)
      panic("iput busy");
    ip->flags |= I_BUSY;
    release(&icache.lock);
    itrunc(ip);
    ip->type = 0;
    iupdate(ip);
    acquire(&icache.lock);
    ip->flags = 0;
    wakeup(ip);
  }
  ip->ref--;
  release(&icache.lock);
}

// Common idiom: unlock, then put.
void
iunlockput(struct inode *ip)
{
  iunlock(ip);
  iput(ip);
}
```

```
// Inode content
//
// The content (data) associated with each inode is stored
// in blocks on the disk. The first NDIRECT block numbers
// are listed in ip->addrs[].  The next NINDIRECT blocks are
// listed in block ip->addrs[NDIRECT].

// Return the disk block address of the nth block in inode ip.
// If there is no such block, bmap allocates one.
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;

  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0)
      ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){
      a[bn] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }

  panic("bmap: out of range");
}
```

```
// Truncate inode (discard contents).
// Only called when the inode has no links
// to it (no directory entries referring to it)
// and has no in-memory reference to it (is
// not an open file or current directory).
static void
itrunc(struct inode *ip)
{
  int i, j;
  struct buf *bp;
  uint *a;

  for(i = 0; i < NDIRECT; i++){
    if(ip->addrs[i]){
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0;
    }
  }

  if(ip->addrs[NDIRECT]){
    bp = bread(ip->dev, ip->addrs[NDIRECT]);
    a = (uint*)bp->data;
    for(j = 0; j < NINDIRECT; j++){
      if(a[j])
        bfree(ip->dev, a[j]);
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT]);
    ip->addrs[NDIRECT] = 0;
  }

  ip->size = 0;
  iupdate(ip);
}

// Copy stat information from inode.
void
stati(struct inode *ip, struct stat *st)
{
  st->dev = ip->dev;
  st->ino = ip->inum;
  st->type = ip->type;
  st->nlink = ip->nlink;
  st->size = ip->size;
}
```

```
// Read data from inode.
int
readi(struct inode *ip, char *dst, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

  if(ip->type == T_DEV){
    if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
      return -1;
    return devsw[ip->major].read(ip, dst, n);
  }

  if(off > ip->size || off + n < off)
    return -1;
  if(off + n > ip->size)
    n = ip->size - off;

  for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(dst, bp->data + off%BSIZE, m);
    brelse(bp);
  }
  return n;
}
```

```
// Write data to inode.
int
writei(struct inode *ip, char *src, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

  if(ip->type == T_DEV){
    if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
      return -1;
    return devsw[ip->major].write(ip, src, n);
  }

  if(off > ip->size || off + n < off)
    return -1;
  if(off + n > MAXFILE*BSIZE)
    return -1;

  for(tot=0; tot<n; tot+=m, off+=m, src+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(bp->data + off%BSIZE, src, m);
    log_write(bp);
    brelse(bp);
  }

  if(n > 0 && off > ip->size){
    ip->size = off;
    iupdate(ip);
  }
  return n;
}
```

```
// Directories

int
namecmp(const char *s, const char *t)
{
  return strncmp(s, t, DIRSIZ);
}

// Look for a directory entry in a directory.
// If found, set *poff to byte offset of entry.
struct inode*
dirlookup(struct inode *dp, char *name, uint *poff)
{
  uint off, inum;
  struct dirent de;

  if(dp->type != T_DIR)
    panic("dirlookup not DIR");

  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlink read");
    if(de.inum == 0)
      continue;
    if(namecmp(name, de.name) == 0){
      // entry matches path element
      if(poff)
        *poff = off;
      inum = de.inum;
      return iget(dp->dev, inum);
    }
  }

  return 0;
}
```

```c
// Write a new directory entry (name, inum) into the directory dp.
int
dirlink(struct inode *dp, char *name, uint inum)
{
  int off;
  struct dirent de;
  struct inode *ip;

  // Check that name is not present.
  if((ip = dirlookup(dp, name, 0)) != 0){
    iput(ip);
    return -1;
  }

  // Look for an empty dirent.
  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlink read");
    if(de.inum == 0)
      break;
  }

  strncpy(de.name, name, DIRSIZ);
  de.inum = inum;
  if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
    panic("dirlink");

  return 0;
}
```

```c
// Paths

// Copy the next path element from path into name.
// Return a pointer to the element following the copied one.
// The returned path has no leading slashes,
// so the caller can check *path=='\0' to see if the name is the last one.
// If no name to remove, return 0.
//
// Examples:
//   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
//   skipelem("///a//bb", name) = "bb", setting name = "a"
//   skipelem("a", name) = "", setting name = "a"
//   skipelem("", name) = skipelem("////", name) = 0
//
static char*
skipelem(char *path, char *name)
{
  char *s;
  int len;

  while(*path == '/')
    path++;
  if(*path == 0)
    return 0;
  s = path;
  while(*path != '/' && *path != 0)
    path++;
  len = path - s;
  if(len >= DIRSIZ)
    memmove(name, s, DIRSIZ);
  else {
    memmove(name, s, len);
    name[len] = 0;
  }
  while(*path == '/')
    path++;
  return path;
}
```

```
// Look up and return the inode for a path name.
// If parent != 0, return the inode for the parent and copy the final
// path element into name, which must have room for DIRSIZ bytes.
// Must be called inside a transaction since it calls iput().
static struct inode*
namex(char *path, int nameiparent, char *name)
{
  struct inode *ip, *next;

  if(*path == '/')
    ip = iget(ROOTDEV, ROOTINO);
  else
    ip = idup(proc->cwd);

  while((path = skipelem(path, name)) != 0){
    ilock(ip);
    if(ip->type != T_DIR){
      iunlockput(ip);
      return 0;
    }
    if(nameiparent && *path == '\0'){
      // Stop one level early.
      iunlock(ip);
      return ip;
    }
    if((next = dirlookup(ip, name, 0)) == 0){
      iunlockput(ip);
      return 0;
    }
    iunlockput(ip);
    ip = next;
  }
  if(nameiparent){
    iput(ip);
    return 0;
  }
  return ip;
}

struct inode*
namei(char *path)
{
  char name[DIRSIZ];
  return namex(path, 0, name);
}
```

```
struct inode*
nameiparent(char *path, char *name)
{
  return namex(path, 1, name);
}
```

```c
//
// File descriptors
//

#include "types.h"
#include "defs.h"
#include "param.h"
#include "fs.h"
#include "file.h"
#include "spinlock.h"

struct devsw devsw[NDEV];
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;

void
fileinit(void)
{
  initlock(&ftable.lock, "ftable");
}

// Allocate a file structure.
struct file*
filealloc(void)
{
  struct file *f;

  acquire(&ftable.lock);
  for(f = ftable.file; f < ftable.file + NFILE; f++){
    if(f->ref == 0){
      f->ref = 1;
      release(&ftable.lock);
      return f;
    }
  }
  release(&ftable.lock);
  return 0;
}
```

```c
// Increment ref count for file f.
struct file*
filedup(struct file *f)
{
  acquire(&ftable.lock);
  if(f->ref < 1)
    panic("filedup");
  f->ref++;
  release(&ftable.lock);
  return f;
}

// Close file f.  (Decrement ref count, close when reaches 0.)
void
fileclose(struct file *f)
{
  struct file ff;

  acquire(&ftable.lock);
  if(f->ref < 1)
    panic("fileclose");
  if(--f->ref > 0){
    release(&ftable.lock);
    return;
  }
  ff = *f;
  f->ref = 0;
  f->type = FD_NONE;
  release(&ftable.lock);

  if(ff.type == FD_PIPE)
    pipeclose(ff.pipe, ff.writable);
  else if(ff.type == FD_INODE){
    begin_op();
    iput(ff.ip);
    end_op();
  }
}
```

```
// Get metadata about file f.
int
filestat(struct file *f, struct stat *st)
{
  if(f->type == FD_INODE){
    ilock(f->ip);
    stati(f->ip, st);
    iunlock(f->ip);
    return 0;
  }
  return -1;
}

// Read from file f.
int
fileread(struct file *f, char *addr, int n)
{
  int r;

  if(f->readable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return piperead(f->pipe, addr, n);
  if(f->type == FD_INODE){
    ilock(f->ip);
    if((r = readi(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
  panic("fileread");
}
```

```
// Write to file f.
int
filewrite(struct file *f, char *addr, int n)
{
  int r;

  if(f->writable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return pipewrite(f->pipe, addr, n);
  if(f->type == FD_INODE){
    // write a few blocks at a time to avoid exceeding
    // the maximum log transaction size, including
    // i-node, indirect block, allocation blocks,
    // and 2 blocks of slop for non-aligned writes.
    // this really belongs lower down, since writei()
    // might be writing a device like the console.
    int max = ((LOGSIZE-1-1-2) / 2) * 512;
    int i = 0;
    while(i < n){
      int n1 = n - i;
      if(n1 > max)
        n1 = max;

      begin_op();
      ilock(f->ip);
      if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
        f->off += r;
      iunlock(f->ip);
      end_op();

      if(r < 0)
        break;
      if(r != n1)
        panic("short filewrite");
      i += r;
    }
    return i == n ? n : -1;
  }
  panic("filewrite");
}
```

```c
//
// File-system system calls.
// Mostly argument checking, since we don't trust
// user code, and calls into file.c and fs.c.
//

#include "types.h"
#include "defs.h"
#include "param.h"
#include "stat.h"
#include "mmu.h"
#include "proc.h"
#include "fs.h"
#include "file.h"
#include "fcntl.h"

// Fetch the nth word-sized system call argument as a file descriptor
// and return both the descriptor and the corresponding struct file.
static int
argfd(int n, int *pfd, struct file **pf)
{
  int fd;
  struct file *f;

  if(argint(n, &fd) < 0)
    return -1;
  if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
    return -1;
  if(pfd)
    *pfd = fd;
  if(pf)
    *pf = f;
  return 0;
}

// Allocate a file descriptor for the given file.
// Takes over file reference from caller on success.
static int
fdalloc(struct file *f)
{
  int fd;

  for(fd = 0; fd < NOFILE; fd++){
    if(proc->ofile[fd] == 0){
      proc->ofile[fd] = f;
      return fd;
    }
  }
  return -1;
}
```

```c
int
sys_dup(void)
{
  struct file *f;
  int fd;

  if(argfd(0, 0, &f) < 0)
    return -1;
  if((fd=fdalloc(f)) < 0)
    return -1;
  filedup(f);
  return fd;
}

int
sys_read(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return fileread(f, p, n);
}

int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}

int
sys_close(void)
{
  int fd;
  struct file *f;

  if(argfd(0, &fd, &f) < 0)
    return -1;
  proc->ofile[fd] = 0;
  fileclose(f);
  return 0;
}
```

```
int
sys_fstat(void)
{
  struct file *f;
  struct stat *st;

  if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
    return -1;
  return filestat(f, st);
}

// Create the path new as a link to the same inode as old.
int
sys_link(void)
{
  char name[DIRSIZ], *new, *old;
  struct inode *dp, *ip;

  if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
    return -1;

  begin_op();
  if((ip = namei(old)) == 0){
    end_op();
    return -1;
  }

  ilock(ip);
  if(ip->type == T_DIR){
    iunlockput(ip);
    end_op();
    return -1;
  }

  ip->nlink++;
  iupdate(ip);
  iunlock(ip);

  if((dp = nameiparent(new, name)) == 0)
    goto bad;
  ilock(dp);
  if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
    iunlockput(dp);
    goto bad;
  }
  iunlockput(dp);
  iput(ip);

  end_op();

```

```
  return 0;

bad:
  ilock(ip);
  ip->nlink--;
  iupdate(ip);
  iunlockput(ip);
  end_op();
  return -1;
}

// Is the directory dp empty except for "." and ".." ?
static int
isdirempty(struct inode *dp)
{
  int off;
  struct dirent de;

  for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("isdirempty: readi");
    if(de.inum != 0)
      return 0;
  }
  return 1;
}
```

```
int
sys_unlink(void)
{
  struct inode *ip, *dp;
  struct dirent de;
  char name[DIRSIZ], *path;
  uint off;

  if(argstr(0, &path) < 0)
    return -1;

  begin_op();
  if((dp = nameiparent(path, name)) == 0){
    end_op();
    return -1;
  }

  ilock(dp);

  // Cannot unlink "." or "..".
  if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
    goto bad;

  if((ip = dirlookup(dp, name, &off)) == 0)
    goto bad;
  ilock(ip);

  if(ip->nlink < 1)
    panic("unlink: nlink < 1");
  if(ip->type == T_DIR && !isdirempty(ip)){
    iunlockput(ip);
    goto bad;
  }

  memset(&de, 0, sizeof(de));
  if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
    panic("unlink: writei");
  if(ip->type == T_DIR){
    dp->nlink--;
    iupdate(dp);
  }
  iunlockput(dp);

  ip->nlink--;
  iupdate(ip);
  iunlockput(ip);

  end_op();

  return 0;
```

```
bad:
  iunlockput(dp);
  end_op();
  return -1;
}

static struct inode*
create(char *path, short type, short major, short minor)
{
  uint off;
  struct inode *ip, *dp;
  char name[DIRSIZ];

  if((dp = nameiparent(path, name)) == 0)
    return 0;
  ilock(dp);

  if((ip = dirlookup(dp, name, &off)) != 0){
    iunlockput(dp);
    ilock(ip);
    if(type == T_FILE && ip->type == T_FILE)
      return ip;
    iunlockput(ip);
    return 0;
  }

  if((ip = ialloc(dp->dev, type)) == 0)
    panic("create: ialloc");

  ilock(ip);
  ip->major = major;
  ip->minor = minor;
  ip->nlink = 1;
  iupdate(ip);

  if(type == T_DIR){  // Create . and .. entries.
    dp->nlink++;  // for ".."
    iupdate(dp);
    // No ip->nlink++ for ".": avoid cyclic ref count.
    if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
      panic("create dots");
  }

  if(dirlink(dp, name, ip->inum) < 0)
    panic("create: dirlink");

  iunlockput(dp);

  return ip;
}
```

```
int
sys_open(void)
{
  char *path;
  int fd, omode;
  struct file *f;
  struct inode *ip;

  if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
    return -1;

  begin_op();

  if(omode & O_CREATE){
    ip = create(path, T_FILE, 0, 0);
    if(ip == 0){
      end_op();
      return -1;
    }
  } else {
    if((ip = namei(path)) == 0){
      end_op();
      return -1;
    }
    ilock(ip);
    if(ip->type == T_DIR && omode != O_RDONLY){
      iunlockput(ip);
      end_op();
      return -1;
    }
  }

  if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
    if(f)
      fileclose(f);
    iunlockput(ip);
    end_op();
    return -1;
  }
  iunlock(ip);
  end_op();

  f->type = FD_INODE;
  f->ip = ip;
  f->off = 0;
  f->readable = !(omode & O_WRONLY);
  f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
  return fd;
}
```

```
int
sys_mkdir(void)
{
  char *path;
  struct inode *ip;

  begin_op();
  if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
    end_op();
    return -1;
  }
  iunlockput(ip);
  end_op();
  return 0;
}

int
sys_mknod(void)
{
  struct inode *ip;
  char *path;
  int major, minor;

  begin_op();
  if((argstr(0, &path)) < 0 ||
     argint(1, &major) < 0 ||
     argint(2, &minor) < 0 ||
     (ip = create(path, T_DEV, major, minor)) == 0){
    end_op();
    return -1;
  }
  iunlockput(ip);
  end_op();
  return 0;
}
```

```
int
sys_chdir(void)
{
  char *path;
  struct inode *ip;

  begin_op();
  if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
    end_op();
    return -1;
  }
  ilock(ip);
  if(ip->type != T_DIR){
    iunlockput(ip);
    end_op();
    return -1;
  }
  iunlock(ip);
  iput(proc->cwd);
  end_op();
  proc->cwd = ip;
  return 0;
}

int
sys_exec(void)
{
  char *path, *argv[MAXARG];
  int i;
  uint uargv, uarg;

  if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
    return -1;
  }
  memset(argv, 0, sizeof(argv));
  for(i=0;; i++){
    if(i >= NELEM(argv))
      return -1;
    if(fetchint(uargv+4*i, (int*)&uarg) < 0)
      return -1;
    if(uarg == 0){
      argv[i] = 0;
      break;
    }
    if(fetchstr(uarg, &argv[i]) < 0)
      return -1;
  }
  return exec(path, argv);
}
```

```
int
sys_pipe(void)
{
  int *fd;
  struct file *rf, *wf;
  int fd0, fd1;

  if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
    return -1;
  if(pipealloc(&rf, &wf) < 0)
    return -1;
  fd0 = -1;
  if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
    if(fd0 >= 0)
      proc->ofile[fd0] = 0;
    fileclose(rf);
    fileclose(wf);
    return -1;
  }
  fd[0] = fd0;
  fd[1] = fd1;
  return 0;
}
```

```c
#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "defs.h"
#include "x86.h"
#include "elf.h"

int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  uint argc, sz, sp, ustack[3+MAXARG+1];
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pde_t *pgdir, *oldpgdir;

  begin_op();
  if((ip = namei(path)) == 0){
    end_op();
    return -1;
  }
  ilock(ip);
  pgdir = 0;

  // Check ELF header
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;

  if((pgdir = setupkvm()) == 0)
    goto bad;

  // Load program into memory.
  sz = 0;
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
      goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
```

```c
    if(ph.vaddr % PGSIZE != 0)
      goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
  }
  iunlockput(ip);
  end_op();
  ip = 0;

  // Allocate two pages at the next page boundary.
  // Make the first inaccessible.  Use the second as the user stack.
  sz = PGROUNDUP(sz);
  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
  sp = sz;

  // Push argument strings, prepare rest of stack in ustack.
  for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
      goto bad;
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
      goto bad;
    ustack[3+argc] = sp;
  }
  ustack[3+argc] = 0;

  ustack[0] = 0xffffffff;  // fake return PC
  ustack[1] = argc;
  ustack[2] = sp - (argc+1)*4;  // argv pointer

  sp -= (3+argc+1) * 4;
  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;

  // Save program name for debugging.
  for(last=s=path; *s; s++)
    if(*s == '/')
      last = s+1;
  safestrcpy(proc->name, last, sizeof(proc->name));

  // Commit to the user image.
  oldpgdir = proc->pgdir;
  proc->pgdir = pgdir;
  proc->sz = sz;
  proc->tf->eip = elf.entry;  // main
  proc->tf->esp = sp;
  switchuvm(proc);
  freevm(oldpgdir);
  return 0;

 bad:
  if(pgdir)
    freevm(pgdir);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
}
```

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "mmu.h"
#include "proc.h"
#include "fs.h"
#include "file.h"
#include "spinlock.h"

#define PIPESIZE 512

struct pipe {
  struct spinlock lock;
  char data[PIPESIZE];
  uint nread;     // number of bytes read
  uint nwrite;    // number of bytes written
  int readopen;   // read fd is still open
  int writeopen;  // write fd is still open
};

int
pipealloc(struct file **f0, struct file **f1)
{
  struct pipe *p;

  p = 0;
  *f0 = *f1 = 0;
  if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
    goto bad;
  if((p = (struct pipe*)kalloc()) == 0)
    goto bad;
  p->readopen = 1;
  p->writeopen = 1;
  p->nwrite = 0;
  p->nread = 0;
  initlock(&p->lock, "pipe");
  (*f0)->type = FD_PIPE;
  (*f0)->readable = 1;
  (*f0)->writable = 0;
  (*f0)->pipe = p;
  (*f1)->type = FD_PIPE;
  (*f1)->readable = 0;
  (*f1)->writable = 1;
  (*f1)->pipe = p;
  return 0;
```

```
 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;
}
```

```c
#include "types.h"
#include "x86.h"

void*
memset(void *dst, int c, uint n)
{
  if ((int)dst%4 == 0 && n%4 == 0){
    c &= 0xFF;
    stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
  } else
    stosb(dst, c, n);
  return dst;
}

int
memcmp(const void *v1, const void *v2, uint n)
{
  const uchar *s1, *s2;

  s1 = v1;
  s2 = v2;
  while(n-- > 0){
    if(*s1 != *s2)
      return *s1 - *s2;
    s1++, s2++;
  }

  return 0;
}

void*
memmove(void *dst, const void *src, uint n)
{
  const char *s;
  char *d;

  s = src;
  d = dst;
  if(s < d && s + n > d){
    s += n;
    d += n;
    while(n-- > 0)
      *--d = *--s;
  } else
    while(n-- > 0)
      *d++ = *s++;

  return dst;
}

```

```c
// memcpy exists to placate GCC.  Use memmove.
void*
memcpy(void *dst, const void *src, uint n)
{
  return memmove(dst, src, n);
}

int
strncmp(const char *p, const char *q, uint n)
{
  while(n > 0 && *p && *p == *q)
    n--, p++, q++;
  if(n == 0)
    return 0;
  return (uchar)*p - (uchar)*q;
}

char*
strncpy(char *s, const char *t, int n)
{
  char *os;

  os = s;
  while(n-- > 0 && (*s++ = *t++) != 0)
    ;
  while(n-- > 0)
    *s++ = 0;
  return os;
}

// Like strncpy but guaranteed to NUL-terminate.
char*
safestrcpy(char *s, const char *t, int n)
{
  char *os;

  os = s;
  if(n <= 0)
    return os;
  while(--n > 0 && (*s++ = *t++) != 0)
    ;
  *s = 0;
  return os;
}






```

```
int
strlen(const char *s)
{
  int n;

  for(n = 0; s[n]; n++)
    ;
  return n;
}
```

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

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

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

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

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

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

```
// Blank page.
```

```c
// Multiprocessor support
// Search memory for MP description structures.
// http://developer.intel.com/design/pentium/datashts/24201606.pdf

#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mp.h"
#include "x86.h"
#include "mmu.h"
#include "proc.h"

struct cpu cpus[NCPU];
int ismp;
int ncpu;
uchar ioapicid;

static uchar
sum(uchar *addr, int len)
{
  int i, sum;

  sum = 0;
  for(i=0; i<len; i++)
    sum += addr[i];
  return sum;
}

// Look for an MP structure in the len bytes at addr.
static struct mp*
mpsearch1(uint a, int len)
{
  uchar *e, *p, *addr;

  addr = P2V(a);
  e = addr+len;
  for(p = addr; p < e; p += sizeof(struct mp))
    if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
      return (struct mp*)p;
  return 0;
}
```

```c
// Search for the MP Floating Pointer Structure, which according to the
// spec is in one of the following three locations:
// 1) in the first KB of the EBDA;
// 2) in the last KB of system base memory;
// 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
static struct mp*
mpsearch(void)
{
  uchar *bda;
  uint p;
  struct mp *mp;

  bda = (uchar *) P2V(0x400);
  if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
    if((mp = mpsearch1(p, 1024)))
      return mp;
  } else {
    p = ((bda[0x14]<<8)|bda[0x13])*1024;
    if((mp = mpsearch1(p-1024, 1024)))
      return mp;
  }
  return mpsearch1(0xF0000, 0x10000);
}

// Search for an MP configuration table.  For now,
// don’t accept the default configurations (physaddr == 0).
// Check for correct signature, calculate the checksum and,
// if correct, check the version.
// To do: check extended table checksum.
static struct mpconf*
mpconfig(struct mp **pmp)
{
  struct mpconf *conf;
  struct mp *mp;

  if((mp = mpsearch()) == 0 || mp->physaddr == 0)
    return 0;
  conf = (struct mpconf*) P2V((uint) mp->physaddr);
  if(memcmp(conf, "PCMP", 4) != 0)
    return 0;
  if(conf->version != 1 && conf->version != 4)
    return 0;
  if(sum((uchar*)conf, conf->length) != 0)
    return 0;
  *pmp = mp;
  return conf;
}
```

```
void
mpinit(void)
{
  uchar *p, *e;
  struct mp *mp;
  struct mpconf *conf;
  struct mpproc *proc;
  struct mpioapic *ioapic;

  if((conf = mpconfig(&mp)) == 0)
    return;
  ismp = 1;
  lapic = (uint*)conf->lapicaddr;
  for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
    switch(*p){
    case MPPROC:
      proc = (struct mpproc*)p;
      if(ncpu < NCPU) {
        cpus[ncpu].apicid = proc->apicid;  // apicid may differ from ncpu
        ncpu++;
      }
      p += sizeof(struct mpproc);
      continue;
    case MPIOAPIC:
      ioapic = (struct mpioapic*)p;
      ioapicid = ioapic->apicno;
      p += sizeof(struct mpioapic);
      continue;
    case MPBUS:
    case MPIOINTR:
    case MPLINTR:
      p += 8;
      continue;
    default:
      ismp = 0;
      break;
    }
  }
  if(!ismp){
    // Didn't like what we found; fall back to no MP.
    ncpu = 1;
    lapic = 0;
    ioapicid = 0;
    return;
  }
```

```
  if(mp->imcrp){
    // Bochs doesn't support IMCR, so this doesn't run on Bochs.
    // But it would on real hardware.
    outb(0x22, 0x70);   // Select IMCR
    outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
  }
}
```

```
// The local APIC manages internal (non-I/O) interrupts.
// See Chapter 8 & Appendix C of Intel processor manual volume 3.

#include "param.h"
#include "types.h"
#include "defs.h"
#include "date.h"
#include "memlayout.h"
#include "traps.h"
#include "mmu.h"
#include "x86.h"
#include "proc.h"  // ncpu

// Local APIC registers, divided by 4 for use as uint[] indices.
#define ID      (0x0020/4)   // ID
#define VER     (0x0030/4)   // Version
#define TPR     (0x0080/4)   // Task Priority
#define EOI     (0x00B0/4)   // EOI
#define SVR     (0x00F0/4)   // Spurious Interrupt Vector
  #define ENABLE     0x00000100   // Unit Enable
#define ESR     (0x0280/4)   // Error Status
#define ICRLO   (0x0300/4)   // Interrupt Command
  #define INIT       0x00000500   // INIT/RESET
  #define STARTUP    0x00000600   // Startup IPI
  #define DELIVS     0x00001000   // Delivery status
  #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
  #define DEASSERT   0x00000000
  #define LEVEL      0x00008000   // Level triggered
  #define BCAST      0x00080000   // Send to all APICs, including self.
  #define BUSY       0x00001000
  #define FIXED      0x00000000
#define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
#define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
  #define X1         0x0000000B   // divide counts by 1
  #define PERIODIC   0x00020000   // Periodic
#define PCINT   (0x0340/4)   // Performance Counter LVT
#define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
#define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
#define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
  #define MASKED     0x00010000   // Interrupt masked
#define TICR    (0x0380/4)   // Timer Initial Count
#define TCCR    (0x0390/4)   // Timer Current Count
#define TDCR    (0x03E0/4)   // Timer Divide Configuration

volatile uint *lapic;  // Initialized in mp.c
```

```
static void
lapicw(int index, int value)
{
  lapic[index] = value;
  lapic[ID];  // wait for write to finish, by reading
}
```

```
void
lapicinit(void)
{
  if(!lapic)
    return;

  // Enable local APIC; set spurious interrupt vector.
  lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));

  // The timer repeatedly counts down at bus frequency
  // from lapic[TICR] and then issues an interrupt.
  // If xv6 cared more about precise timekeeping,
  // TICR would be calibrated using an external time source.
  lapicw(TDCR, X1);
  lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
  lapicw(TICR, 10000000);

  // Disable logical interrupt lines.
  lapicw(LINT0, MASKED);
  lapicw(LINT1, MASKED);

  // Disable performance counter overflow interrupts
  // on machines that provide that interrupt entry.
  if(((lapic[VER]>>16) & 0xFF) >= 4)
    lapicw(PCINT, MASKED);

  // Map error interrupt to IRQ_ERROR.
  lapicw(ERROR, T_IRQ0 + IRQ_ERROR);

  // Clear error status register (requires back-to-back writes).
  lapicw(ESR, 0);
  lapicw(ESR, 0);

  // Ack any outstanding interrupts.
  lapicw(EOI, 0);

  // Send an Init Level De-Assert to synchronise arbitration ID's.
  lapicw(ICRHI, 0);
  lapicw(ICRLO, BCAST | INIT | LEVEL);
  while(lapic[ICRLO] & DELIVS)
    ;

  // Enable interrupts on the APIC (but not on the processor).
  lapicw(TPR, 0);
}
```

```
int
cpunum(void)
{
  int apicid, i;

  // Cannot call cpu when interrupts are enabled:
  // result not guaranteed to last long enough to be used!
  // Would prefer to panic but even printing is chancy here:
  // almost everything, including cprintf and panic, calls cpu,
  // often indirectly through acquire and release.
  if(readeflags()&FL_IF){
    static int n;
    if(n++ == 0)
      cprintf("cpu called from %x with interrupts enabled\n",
        __builtin_return_address(0));
  }

  if (!lapic)
    return 0;

  apicid = lapic[ID] >> 24;
  for (i = 0; i < ncpu; ++i) {
    if (cpus[i].apicid == apicid)
      return i;
  }
  panic("unknown apicid\n");
}

// Acknowledge interrupt.
void
lapiceoi(void)
{
  if(lapic)
    lapicw(EOI, 0);
}

// Spin for a given number of microseconds.
// On real hardware would want to tune this dynamically.
void
microdelay(int us)
{
}
```

```c
#define CMOS_PORT    0x70
#define CMOS_RETURN  0x71

// Start additional processor running entry code at addr.
// See Appendix B of MultiProcessor Specification.
void
lapicstartap(uchar apicid, uint addr)
{
  int i;
  ushort *wrv;

  // "The BSP must initialize CMOS shutdown code to 0AH
  // and the warm reset vector (DWORD based at 40:67) to point at
  // the AP startup code prior to the [universal startup algorithm]."
  outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
  outb(CMOS_PORT+1, 0x0A);
  wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
  wrv[0] = 0;
  wrv[1] = addr >> 4;

  // "Universal startup algorithm."
  // Send INIT (level-triggered) interrupt to reset other CPU.
  lapicw(ICRHI, apicid<<24);
  lapicw(ICRLO, INIT | LEVEL | ASSERT);
  microdelay(200);
  lapicw(ICRLO, INIT | LEVEL);
  microdelay(100);    // should be 10ms, but too slow in Bochs!

  // Send startup IPI (twice!) to enter code.
  // Regular hardware is supposed to only accept a STARTUP
  // when it is in the halted state due to an INIT.  So the second
  // should be ignored, but it is part of the official Intel algorithm.
  // Bochs complains about the second one.  Too bad for Bochs.
  for(i = 0; i < 2; i++){
    lapicw(ICRHI, apicid<<24);
    lapicw(ICRLO, STARTUP | (addr>>12));
    microdelay(200);
  }
}
```

```c
#define CMOS_STATA   0x0a
#define CMOS_STATB   0x0b
#define CMOS_UIP    (1 << 7)        // RTC update in progress

#define SECS    0x00
#define MINS    0x02
#define HOURS   0x04
#define DAY     0x07
#define MONTH   0x08
#define YEAR    0x09

static uint cmos_read(uint reg)
{
  outb(CMOS_PORT,  reg);
  microdelay(200);

  return inb(CMOS_RETURN);
}

static void fill_rtcdate(struct rtcdate *r)
{
  r->second = cmos_read(SECS);
  r->minute = cmos_read(MINS);
  r->hour   = cmos_read(HOURS);
  r->day    = cmos_read(DAY);
  r->month  = cmos_read(MONTH);
  r->year   = cmos_read(YEAR);
}

// qemu seems to use 24-hour GWT and the values are BCD encoded
void cmostime(struct rtcdate *r)
{
  struct rtcdate t1, t2;
  int sb, bcd;

  sb = cmos_read(CMOS_STATB);

  bcd = (sb & (1 << 2)) == 0;

  // make sure CMOS doesn't modify time while we read it
  for(;;) {
    fill_rtcdate(&t1);
    if(cmos_read(CMOS_STATA) & CMOS_UIP)
        continue;
    fill_rtcdate(&t2);
    if(memcmp(&t1, &t2, sizeof(t1)) == 0)
      break;
  }
  // convert
  if(bcd) {
#define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
    CONV(second);
    CONV(minute);
    CONV(hour  );
    CONV(day   );
    CONV(month );
    CONV(year  );
#undef     CONV
  }

  *r = t1;
  r->year += 2000;
}
```

```
// The I/O APIC manages hardware interrupts for an SMP system.
// http://www.intel.com/design/chipsets/datashts/29056601.pdf
// See also picirq.c.

#include "types.h"
#include "defs.h"
#include "traps.h"

#define IOAPIC  0xFEC00000   // Default physical address of IO APIC

#define REG_ID     0x00  // Register index: ID
#define REG_VER    0x01  // Register index: version
#define REG_TABLE  0x10  // Redirection table base

// The redirection table starts at REG_TABLE and uses
// two registers to configure each interrupt.
// The first (low) register in a pair contains configuration bits.
// The second (high) register contains a bitmask telling which
// CPUs can serve that interrupt.
#define INT_DISABLED   0x00010000  // Interrupt disabled
#define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
#define INT_ACTIVELOW  0x00002000  // Active low (vs high)
#define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)

volatile struct ioapic *ioapic;

// IO APIC MMIO structure: write reg, then read or write data.
struct ioapic {
  uint reg;
  uint pad[3];
  uint data;
};

static uint
ioapicread(int reg)
{
  ioapic->reg = reg;
  return ioapic->data;
}

static void
ioapicwrite(int reg, uint data)
{
  ioapic->reg = reg;
  ioapic->data = data;
}
```

```
void
ioapicinit(void)
{
  int i, id, maxintr;

  if(!ismp)
    return;

  ioapic = (volatile struct ioapic*)IOAPIC;
  maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
  id = ioapicread(REG_ID) >> 24;
  if(id != ioapicid)
    cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");

  // Mark all interrupts edge-triggered, active high, disabled,
  // and not routed to any CPUs.
  for(i = 0; i <= maxintr; i++){
    ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
    ioapicwrite(REG_TABLE+2*i+1, 0);
  }
}

void
ioapicenable(int irq, int cpunum)
{
  if(!ismp)
    return;

  // Mark interrupt edge-triggered, active high,
  // enabled, and routed to the given cpunum,
  // which happens to be that cpu's APIC ID.
  ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
  ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
}
```

```
// Intel 8259A programmable interrupt controllers.

#include "types.h"
#include "x86.h"
#include "traps.h"

// I/O Addresses of the two programmable interrupt controllers
#define IO_PIC1         0x20    // Master (IRQs 0-7)
#define IO_PIC2         0xA0    // Slave (IRQs 8-15)

#define IRQ_SLAVE       2       // IRQ at which slave connects to master

// Current IRQ mask.
// Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);

static void
picsetmask(ushort mask)
{
  irqmask = mask;
  outb(IO_PIC1+1, mask);
  outb(IO_PIC2+1, mask >> 8);
}

void
picenable(int irq)
{
  picsetmask(irqmask & ~(1<<irq));
}

// Initialize the 8259A interrupt controllers.
void
picinit(void)
{
  // mask all interrupts
  outb(IO_PIC1+1, 0xFF);
  outb(IO_PIC2+1, 0xFF);

  // Set up master (8259A-1)

  // ICW1:  0001g0hi
  //    g:  0 = edge triggering, 1 = level triggering
  //    h:  0 = cascaded PICs, 1 = master only
  //    i:  0 = no ICW4, 1 = ICW4 required
  outb(IO_PIC1, 0x11);

  // ICW2:  Vector offset
  outb(IO_PIC1+1, T_IRQ0);
  // ICW3:  (master PIC) bit mask of IR lines connected to slaves
  //        (slave PIC) 3-bit # of slave's connection to master
  outb(IO_PIC1+1, 1<<IRQ_SLAVE);

  // ICW4:  000nbmap
  //    n:  1 = special fully nested mode
  //    b:  1 = buffered mode
  //    m:  0 = slave PIC, 1 = master PIC
  //      (ignored when b is 0, as the master/slave role
  //      can be hardwired).
  //    a:  1 = Automatic EOI mode
  //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
  outb(IO_PIC1+1, 0x3);

  // Set up slave (8259A-2)
  outb(IO_PIC2, 0x11);                  // ICW1
  outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
  outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
  // NB Automatic EOI mode doesn't tend to work on the slave.
  // Linux source code says it's "to be investigated".
  outb(IO_PIC2+1, 0x3);                 // ICW4

  // OCW3:  0ef01prs
  //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
  //    p:  0 = no polling, 1 = polling mode
  //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
  outb(IO_PIC1, 0x68);             // clear specific mask
  outb(IO_PIC1, 0x0a);             // read IRR by default

  outb(IO_PIC2, 0x68);             // OCW3
  outb(IO_PIC2, 0x0a);             // OCW3

  if(irqmask != 0xFFFF)
    picsetmask(irqmask);
}
```

```
// Blank page.
// PC keyboard interface constants

#define KBSTATP         0x64    // kbd controller status port(I)
#define KBS_DIB         0x01    // kbd data in buffer
#define KBDATAP         0x60    // kbd data port(I)

#define NO              0

#define SHIFT           (1<<0)
#define CTL             (1<<1)
#define ALT             (1<<2)

#define CAPSLOCK        (1<<3)
#define NUMLOCK         (1<<4)
#define SCROLLLOCK      (1<<5)

#define E0ESC           (1<<6)

// Special keycodes
#define KEY_HOME        0xE0
#define KEY_END         0xE1
#define KEY_UP          0xE2
#define KEY_DN          0xE3
#define KEY_LF          0xE4
#define KEY_RT          0xE5
#define KEY_PGUP        0xE6
#define KEY_PGDN        0xE7
#define KEY_INS         0xE8
#define KEY_DEL         0xE9

// C('A') == Control-A
#define C(x) (x - '@')

static uchar shiftcode[256] =
{
  [0x1D] CTL,
  [0x2A] SHIFT,
  [0x36] SHIFT,
  [0x38] ALT,
  [0x9D] CTL,
  [0xB8] ALT
};

static uchar togglecode[256] =
{
  [0x3A] CAPSLOCK,
  [0x45] NUMLOCK,
  [0x46] SCROLLLOCK
};

```

```
static uchar normalmap[256] =
{
  NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
  '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
  'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
  'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
  'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
  '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
  'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
  NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
  NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
  '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
  '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
  [0x9C] '\n',      // KP_Enter
  [0xB5] '/',       // KP_Div
  [0xC8] KEY_UP,    [0xD0] KEY_DN,
  [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
  [0xCB] KEY_LF,    [0xCD] KEY_RT,
  [0x97] KEY_HOME,  [0xCF] KEY_END,
  [0xD2] KEY_INS,   [0xD3] KEY_DEL
};

static uchar shiftmap[256] =
{
  NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
  '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
  'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
  'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
  'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
  '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
  'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
  NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
  NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
  '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
  '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
  [0x9C] '\n',      // KP_Enter
  [0xB5] '/',       // KP_Div
  [0xC8] KEY_UP,    [0xD0] KEY_DN,
  [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
  [0xCB] KEY_LF,    [0xCD] KEY_RT,
  [0x97] KEY_HOME,  [0xCF] KEY_END,
  [0xD2] KEY_INS,   [0xD3] KEY_DEL
};
```

```
static uchar ctlmap[256] =
{
  NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
  NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
  C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
  C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
  C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
  NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
  C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
  [0x9C] '\r',      // KP_Enter
  [0xB5] C('/'),    // KP_Div
  [0xC8] KEY_UP,    [0xD0] KEY_DN,
  [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
  [0xCB] KEY_LF,    [0xCD] KEY_RT,
  [0x97] KEY_HOME,  [0xCF] KEY_END,
  [0xD2] KEY_INS,   [0xD3] KEY_DEL
};
```

```
#include "types.h"
#include "x86.h"
#include "defs.h"
#include "kbd.h"

int
kbdgetc(void)
{
  static uint shift;
  static uchar *charcode[4] = {
    normalmap, shiftmap, ctlmap, ctlmap
  };
  uint st, data, c;

  st = inb(KBSTATP);
  if((st & KBS_DIB) == 0)
    return -1;
  data = inb(KBDATAP);

  if(data == 0xE0){
    shift |= E0ESC;
    return 0;
  } else if(data & 0x80){
    // Key released
    data = (shift & E0ESC ? data : data & 0x7F);
    shift &= ~(shiftcode[data] | E0ESC);
    return 0;
  } else if(shift & E0ESC){
    // Last character was an E0 escape; or with 0x80
    data |= 0x80;
    shift &= ~E0ESC;
  }

  shift |= shiftcode[data];
  shift ^= togglecode[data];
  c = charcode[shift & (CTL | SHIFT)][data];
  if(shift & CAPSLOCK){
    if('a' <= c && c <= 'z')
      c += 'A' - 'a';
    else if('A' <= c && c <= 'Z')
      c += 'a' - 'A';
  }
  return c;
}

void
kbdintr(void)
{
  consoleintr(kbdgetc);
}
```

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

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

static void consputc(int);

static int panicked = 0;

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

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

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

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

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

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

```

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

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

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

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


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

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

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

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

static void
cgaputc(int c)
{
  int pos;

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

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

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

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

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

```
void
consputc(int c)
{
  if(panicked){
    cli();
    for(;;)
      ;
  }

  if(c == BACKSPACE){
    uartputc('\b'); uartputc(' '); uartputc('\b');
  } else
    uartputc(c);
  cgaputc(c);
}

#define INPUT_BUF 128
struct {
  char buf[INPUT_BUF];
  uint r;  // Read index
  uint w;  // Write index
  uint e;  // Edit index
} input;

#define C(x)  ((x)-'@')  // Control-x

void
consoleintr(int (*getc)(void))
{
  int c, doprocdump = 0;

  acquire(&cons.lock);
  while((c = getc()) >= 0){
    switch(c){
    case C('P'):  // Process listing.
      // procdump() locks cons.lock indirectly; invoke later
      doprocdump = 1;
      break;
    case C('U'):  // Kill line.
      while(input.e != input.w &&
            input.buf[(input.e-1) % INPUT_BUF] != '\n'){
        input.e--;
        consputc(BACKSPACE);
      }
      break;
    case C('H'): case '\x7f':  // Backspace
      if(input.e != input.w){
        input.e--;
        consputc(BACKSPACE);
      }
```

```
      break;
    default:
      if(c != 0 && input.e-input.r < INPUT_BUF){
        c = (c == '\r') ? '\n' : c;
        input.buf[input.e++ % INPUT_BUF] = c;
        consputc(c);
        if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
          input.w = input.e;
          wakeup(&input.r);
        }
      }
      break;
    }
  }
  release(&cons.lock);
  if(doprocdump) {
    procdump();  // now call procdump() wo. cons.lock held
  }
}

int
consoleread(struct inode *ip, char *dst, int n)
{
  uint target;
  int c;

  iunlock(ip);
  target = n;
  acquire(&cons.lock);
  while(n > 0){
    while(input.r == input.w){
      if(proc->killed){
        release(&cons.lock);
        ilock(ip);
        return -1;
      }
      sleep(&input.r, &cons.lock);
    }
    c = input.buf[input.r++ % INPUT_BUF];
    if(c == C('D')){  // EOF
      if(n < target){
        // Save ^D for next time, to make sure
        // caller gets a 0-byte result.
        input.r--;
      }
      break;
    }
    *dst++ = c;
    --n;
    if(c == '\n')
```

```
      break;
  }
  release(&cons.lock);
  ilock(ip);

  return target - n;
}

int
consolewrite(struct inode *ip, char *buf, int n)
{
  int i;

  iunlock(ip);
  acquire(&cons.lock);
  for(i = 0; i < n; i++)
    consputc(buf[i] & 0xff);
  release(&cons.lock);
  ilock(ip);

  return n;
}

void
consoleinit(void)
{
  initlock(&cons.lock, "console");

  devsw[CONSOLE].write = consolewrite;
  devsw[CONSOLE].read = consoleread;
  cons.locking = 1;

  picenable(IRQ_KBD);
  ioapicenable(IRQ_KBD, 0);
}
```

```
// Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
// Only used on uniprocessors;
// SMP machines use the local APIC timer.

#include "types.h"
#include "defs.h"
#include "traps.h"
#include "x86.h"

#define IO_TIMER1       0x040           // 8253 Timer #1

// Frequency of all three count-down timers;
// (TIMER_FREQ/freq) is the appropriate count
// to generate a frequency of freq Hz.

#define TIMER_FREQ      1193182
#define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))

#define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
#define TIMER_SEL0      0x00    // select counter 0
#define TIMER_RATEGEN   0x04    // mode 2, rate generator
#define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first

void
timerinit(void)
{
  // Interrupt 100 times/sec.
  outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
  outb(IO_TIMER1, TIMER_DIV(100) % 256);
  outb(IO_TIMER1, TIMER_DIV(100) / 256);
  picenable(IRQ_TIMER);
}
```

```
// Intel 8250 serial port (UART).

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

#define COM1    0x3f8

static int uart;    // is there a uart?

void
uartinit(void)
{
  char *p;

  // Turn off the FIFO
  outb(COM1+2, 0);

  // 9600 baud, 8 data bits, 1 stop bit, parity off.
  outb(COM1+3, 0x80);    // Unlock divisor
  outb(COM1+0, 115200/9600);
  outb(COM1+1, 0);
  outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
  outb(COM1+4, 0);
  outb(COM1+1, 0x01);    // Enable receive interrupts.

  // If status is 0xFF, no serial port.
  if(inb(COM1+5) == 0xFF)
    return;
  uart = 1;

  // Acknowledge pre-existing interrupt conditions;
  // enable interrupts.
  inb(COM1+2);
  inb(COM1+0);
  picenable(IRQ_COM1);
  ioapicenable(IRQ_COM1, 0);

  // Announce that we're here.
  for(p="xv6...\n"; *p; p++)
    uartputc(*p);
}

```

```
void
uartputc(int c)
{
  int i;

  if(!uart)
    return;
  for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
    microdelay(10);
  outb(COM1+0, c);
}

static int
uartgetc(void)
{
  if(!uart)
    return -1;
  if(!(inb(COM1+5) & 0x01))
    return -1;
  return inb(COM1+0);
}

void
uartintr(void)
{
  consoleintr(uartgetc);
}
```

```
# Initial process execs /init.
# This code runs in user space.

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


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

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

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

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

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

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

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
```

```c
// init: The initial user-level program

#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

char *argv[] = { "sh", 0 };

int
main(void)
{
  int pid, wpid;

  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr

  for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

```c
// Shell.

#include "types.h"
#include "user.h"
#include "fcntl.h"

// Parsed command representation
#define EXEC  1
#define REDIR 2
#define PIPE  3
#define LIST  4
#define BACK  5

#define MAXARGS 10

struct cmd {
  int type;
};

struct execcmd {
  int type;
  char *argv[MAXARGS];
  char *eargv[MAXARGS];
};

struct redircmd {
  int type;
  struct cmd *cmd;
  char *file;
  char *efile;
  int mode;
  int fd;
};

struct pipecmd {
  int type;
  struct cmd *left;
  struct cmd *right;
};

struct listcmd {
  int type;
  struct cmd *left;
  struct cmd *right;
};

struct backcmd {
  int type;
  struct cmd *cmd;
};
```

```
int fork1(void);  // Fork but panics on failure.
void panic(char*);
struct cmd *parsecmd(char*);

// Execute cmd.  Never returns.
void
runcmd(struct cmd *cmd)
{
  int p[2];
  struct backcmd *bcmd;
  struct execcmd *ecmd;
  struct listcmd *lcmd;
  struct pipecmd *pcmd;
  struct redircmd *rcmd;

  if(cmd == 0)
    exit();

  switch(cmd->type){
  default:
    panic("runcmd");

  case EXEC:
    ecmd = (struct execcmd*)cmd;
    if(ecmd->argv[0] == 0)
      exit();
    exec(ecmd->argv[0], ecmd->argv);
    printf(2, "exec %s failed\n", ecmd->argv[0]);
    break;

  case REDIR:
    rcmd = (struct redircmd*)cmd;
    close(rcmd->fd);
    if(open(rcmd->file, rcmd->mode) < 0){
      printf(2, "open %s failed\n", rcmd->file);
      exit();
    }
    runcmd(rcmd->cmd);
    break;

  case LIST:
    lcmd = (struct listcmd*)cmd;
    if(fork1() == 0)
      runcmd(lcmd->left);
    wait();
    runcmd(lcmd->right);
    break;



```

```
  case PIPE:
    pcmd = (struct pipecmd*)cmd;
    if(pipe(p) < 0)
      panic("pipe");
    if(fork1() == 0){
      close(1);
      dup(p[1]);
      close(p[0]);
      close(p[1]);
      runcmd(pcmd->left);
    }
    if(fork1() == 0){
      close(0);
      dup(p[0]);
      close(p[0]);
      close(p[1]);
      runcmd(pcmd->right);
    }
    close(p[0]);
    close(p[1]);
    wait();
    wait();
    break;

  case BACK:
    bcmd = (struct backcmd*)cmd;
    if(fork1() == 0)
      runcmd(bcmd->cmd);
    break;
  }
  exit();
}

int
getcmd(char *buf, int nbuf)
{
  printf(2, "$ ");
  memset(buf, 0, nbuf);
  gets(buf, nbuf);
  if(buf[0] == 0) // EOF
    return -1;
  return 0;
}
```

```
int
main(void)
{
  static char buf[100];
  int fd;

  // Ensure that three file descriptors are open.
  while((fd = open("console", O_RDWR)) >= 0){
    if(fd >= 3){
      close(fd);
      break;
    }
  }

  // Read and run input commands.
  while(getcmd(buf, sizeof(buf)) >= 0){
    if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
      // Chdir must be called by the parent, not the child.
      buf[strlen(buf)-1] = 0;  // chop \n
      if(chdir(buf+3) < 0)
        printf(2, "cannot cd %s\n", buf+3);
      continue;
    }
    if(fork1() == 0)
      runcmd(parsecmd(buf));
    wait();
  }
  exit();
}

void
panic(char *s)
{
  printf(2, "%s\n", s);
  exit();
}

int
fork1(void)
{
  int pid;

  pid = fork();
  if(pid == -1)
    panic("fork");
  return pid;
}
```

```
// Constructors

struct cmd*
execcmd(void)
{
  struct execcmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = EXEC;
  return (struct cmd*)cmd;
}

struct cmd*
redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
{
  struct redircmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = REDIR;
  cmd->cmd = subcmd;
  cmd->file = file;
  cmd->efile = efile;
  cmd->mode = mode;
  cmd->fd = fd;
  return (struct cmd*)cmd;
}

struct cmd*
pipecmd(struct cmd *left, struct cmd *right)
{
  struct pipecmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = PIPE;
  cmd->left = left;
  cmd->right = right;
  return (struct cmd*)cmd;
}
```

```
struct cmd*
listcmd(struct cmd *left, struct cmd *right)
{
  struct listcmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = LIST;
  cmd->left = left;
  cmd->right = right;
  return (struct cmd*)cmd;
}

struct cmd*
backcmd(struct cmd *subcmd)
{
  struct backcmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = BACK;
  cmd->cmd = subcmd;
  return (struct cmd*)cmd;
}
```

```
// Parsing

char whitespace[] = " \t\r\n\v";
char symbols[] = "<|>&;()";

int
gettoken(char **ps, char *es, char **q, char **eq)
{
  char *s;
  int ret;

  s = *ps;
  while(s < es && strchr(whitespace, *s))
    s++;
  if(q)
    *q = s;
  ret = *s;
  switch(*s){
  case 0:
    break;
  case '|':
  case '(':
  case ')':
  case ';':
  case '&':
  case '<':
    s++;
    break;
  case '>':
    s++;
    if(*s == '>'){
      ret = '+';
      s++;
    }
    break;
  default:
    ret = 'a';
    while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
      s++;
    break;
  }
  if(eq)
    *eq = s;

  while(s < es && strchr(whitespace, *s))
    s++;
  *ps = s;
  return ret;
}
```

```
int
peek(char **ps, char *es, char *toks)
{
  char *s;

  s = *ps;
  while(s < es && strchr(whitespace, *s))
    s++;
  *ps = s;
  return *s && strchr(toks, *s);
}

struct cmd *parseline(char**, char*);
struct cmd *parsepipe(char**, char*);
struct cmd *parseexec(char**, char*);
struct cmd *nulterminate(struct cmd*);

struct cmd*
parsecmd(char *s)
{
  char *es;
  struct cmd *cmd;

  es = s + strlen(s);
  cmd = parseline(&s, es);
  peek(&s, es, "");
  if(s != es){
    printf(2, "leftovers: %s\n", s);
    panic("syntax");
  }
  nulterminate(cmd);
  return cmd;
}

struct cmd*
parseline(char **ps, char *es)
{
  struct cmd *cmd;

  cmd = parsepipe(ps, es);
  while(peek(ps, es, "&")){
    gettoken(ps, es, 0, 0);
    cmd = backcmd(cmd);
  }
  if(peek(ps, es, ";")){
    gettoken(ps, es, 0, 0);
    cmd = listcmd(cmd, parseline(ps, es));
  }
  return cmd;
}
```

```
struct cmd*
parsepipe(char **ps, char *es)
{
  struct cmd *cmd;

  cmd = parseexec(ps, es);
  if(peek(ps, es, "|")){
    gettoken(ps, es, 0, 0);
    cmd = pipecmd(cmd, parsepipe(ps, es));
  }
  return cmd;
}

struct cmd*
parseredirs(struct cmd *cmd, char **ps, char *es)
{
  int tok;
  char *q, *eq;

  while(peek(ps, es, "<>")){
    tok = gettoken(ps, es, 0, 0);
    if(gettoken(ps, es, &q, &eq) != 'a')
      panic("missing file for redirection");
    switch(tok){
    case '<':
      cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
      break;
    case '>':
      cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
      break;
    case '+':  // >>
      cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
      break;
    }
  }
  return cmd;
}
```

```
struct cmd*
parseblock(char **ps, char *es)
{
  struct cmd *cmd;

  if(!peek(ps, es, "("))
    panic("parseblock");
  gettoken(ps, es, 0, 0);
  cmd = parseline(ps, es);
  if(!peek(ps, es, ")"))
    panic("syntax - missing )");
  gettoken(ps, es, 0, 0);
  cmd = parseredirs(cmd, ps, es);
  return cmd;
}

struct cmd*
parseexec(char **ps, char *es)
{
  char *q, *eq;
  int tok, argc;
  struct execcmd *cmd;
  struct cmd *ret;

  if(peek(ps, es, "("))
    return parseblock(ps, es);

  ret = execcmd();
  cmd = (struct execcmd*)ret;

  argc = 0;
  ret = parseredirs(ret, ps, es);
  while(!peek(ps, es, "|)&;")){
    if((tok=gettoken(ps, es, &q, &eq)) == 0)
      break;
    if(tok != 'a')
      panic("syntax");
    cmd->argv[argc] = q;
    cmd->eargv[argc] = eq;
    argc++;
    if(argc >= MAXARGS)
      panic("too many args");
    ret = parseredirs(ret, ps, es);
  }
  cmd->argv[argc] = 0;
  cmd->eargv[argc] = 0;
  return ret;
}
```

```
// NUL-terminate all the counted strings.
struct cmd*
nulterminate(struct cmd *cmd)
{
  int i;
  struct backcmd *bcmd;
  struct execcmd *ecmd;
  struct listcmd *lcmd;
  struct pipecmd *pcmd;
  struct redircmd *rcmd;

  if(cmd == 0)
    return 0;

  switch(cmd->type){
  case EXEC:
    ecmd = (struct execcmd*)cmd;
    for(i=0; ecmd->argv[i]; i++)
      *ecmd->eargv[i] = 0;
    break;

  case REDIR:
    rcmd = (struct redircmd*)cmd;
    nulterminate(rcmd->cmd);
    *rcmd->efile = 0;
    break;

  case PIPE:
    pcmd = (struct pipecmd*)cmd;
    nulterminate(pcmd->left);
    nulterminate(pcmd->right);
    break;

  case LIST:
    lcmd = (struct listcmd*)cmd;
    nulterminate(lcmd->left);
    nulterminate(lcmd->right);
    break;

  case BACK:
    bcmd = (struct backcmd*)cmd;
    nulterminate(bcmd->cmd);
    break;
  }
  return cmd;
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

#define SECTSIZE  512

void readseg(uchar*, uint, uint);

void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}


```

```c
void
waitdisk(void)
{
  // Wait for disk ready.
  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);
}



```