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/6.828/2007/v6.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 (bootother.S, mp.h, mp.c, lapic.c)
    FreeBSD (ioapic.c)
    NetBSD (console.c)

The following people made contributions:
    Russ Cox (context switching, locking)
    Cliff Frey (MP)
    Xiao Yu (MP)

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/6.828/2007/tools.html. Then run "make TOOLPREFIX=i386-jos-elf-".

To run xv6, you can use Bochs or QEMU, both PC simulators. Bochs makes debugging easier, but QEMU is much faster. To run in Bochs, run "make bochs" and then type "c" at the bochs prompt. To run in QEMU, run "make qemu". Both log the xv6 screen output to standard output.

To create a typeset version of the code, run "make xv6.pdf". This requires the "mpage" text formatting 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.

# basic headers

# startup

# locks

# processes

system calls

file system

pipes

string operations

low-level hardware

user-level

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 2256
        0311 1928 1962 2255
        2256
```

indicates that swtch is defined on line 2256 and is mentioned on five lines on sheets 03, 19, and 22.

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

```
#define NPROC        64  // maximum number of processes
#define PAGE       4096  // granularity of user-space memory allocation
#define KSTACKSIZE PAGE  // 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 NBUF         10  // size of disk block cache
#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
```

```c
struct buf;
struct context;
struct file;
struct inode;
struct pipe;
struct proc;
struct spinlock;
struct stat;

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

// console.c
void            console_init(void);
void            cprintf(char*, ...);
void            console_intr(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
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(void);
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);
```

```c
// ide.c
void            ide_init(void);
void            ide_intr(void);
void            ide_rw(struct buf *);

// ioapic.c
void            ioapic_enable(int irq, int cpu);
extern uchar    ioapic_id;
void            ioapic_init(void);

// kalloc.c
char*           kalloc(int);
void            kfree(char*, int);
void            kinit(void);

// kbd.c
void            kbd_intr(void);

// lapic.c
int             cpu(void);
extern volatile uint*    lapic;
void            lapic_disableintr(void);
void            lapic_enableintr(void);
void            lapic_eoi(void);
void            lapic_init(int);
void            lapic_startap(uchar, uint);
void            lapic_timerinit(void);
void            lapic_timerintr(void);

// mp.c
extern int      ismp;
int             mp_bcpu(void);
void            mp_init(void);
void            mp_startthem(void);

// picirq.c
void            pic_enable(int);
void            pic_init(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
struct proc*    copyproc(struct proc*);
void            exit(void);
int             growproc(int);
int             kill(int);
```

```
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            setupsegs(struct proc*);
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*);

// 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(struct proc*, uint, int*);
int             fetchstr(struct proc*, uint, char**);
void            syscall(void);

// timer.c
void            timer_init(void);

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

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


```

```
// Special assembly routines to access x86-specific
// hardware 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\n\trepne\n\tinsl"     :
                   "=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\n\trepne\n\toutsl"    :
                   "=S" (addr), "=c" (cnt)    :
                   "d" (port), "0" (addr), "1" (cnt)  :
                   "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
read_eflags(void)
{
  uint eflags;
  asm volatile("pushfl; popl %0" : "=r" (eflags));
  return eflags;
}

static inline void
write_eflags(uint eflags)
{
  asm volatile("pushl %0; popfl" : : "r" (eflags));
}
```

```
static inline void
cpuid(uint info, uint *eaxp, uint *ebxp, uint *ecxp, uint *edxp)
{
  uint eax, ebx, ecx, edx;

  asm volatile("cpuid" :
               "=a" (eax), "=b" (ebx), "=c" (ecx), "=d" (edx) :
               "a" (info));
  if(eaxp)
    *eaxp = eax;
  if(ebxp)
    *ebxp = ebx;
  if(ecxp)
    *ecxp = ecx;
  if(edxp)
    *edxp = edx;
}

static inline uint
cmpxchg(uint oldval, uint newval, volatile uint* lock_addr)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; cmpxchgl %2, %0" :
                       "+m" (*lock_addr), "=a" (result) :
                       "r"(newval), "1"(oldval) :
                       "cc");
  return result;
}

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

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

```
// Layout of the trap frame on the stack upon entry 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 es;
  ushort padding1;
  ushort ds;
  ushort padding2;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding3;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding4;
};
```

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

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

#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
```

```c
// 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

// 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
};
```

```c
// Null segment
#define SEG_NULL        (struct segdesc){ 0,0,0,0,0,0,0,0,0,0,0,0,0 }

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

#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
```

```
// 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 ss : 16;         // 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).ss = (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;                  \
}
```

```
// 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 offset;
  uint va;
  uint pa;
  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.
```

```
#include "asm.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.

.set PROT_MODE_CSEG, 0x8         # kernel code segment selector
.set PROT_MODE_DSEG, 0x10        # kernel data segment selector
.set CR0_PE_ON,      0x1         # protected mode enable flag

.globl start
start:
  .code16                     # Assemble for 16-bit mode
  cli                         # Disable interrupts
  cld                         # String operations increment

  # Set up the important data segment registers (DS, ES, SS).
  xorw    %ax,%ax             # Segment number zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment

  # Enable A20:
  #   For backwards compatibility with the earliest PCs, physical
  #   address line 20 is tied low, so that addresses higher than
  #   1MB wrap around to zero by default.  This code undoes this.
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, using a bootstrap GDT
  # and segment translation that makes virtual addresses
  # identical to their physical addresses, so that the
  # effective memory map does not change during the switch.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE_ON, %eax
  movl    %eax, %cr0

  # Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $PROT_MODE_CSEG, $protcseg

  .code32                     # Assemble for 32-bit mode
protcseg:
  # Set up the protected-mode data segment registers
  movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS
  movw    %ax, %ss                # -> SS: Stack Segment

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

  # If bootmain returns (it shouldn't), loop.
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   0x17                            # sizeof(gdt) - 1
  .long   gdt                             # address gdt
```

```
#include "asm.h"

# Start an Application Processor. This must be placed on a 4KB boundary
# somewhere in the 1st MB of conventional memory (APBOOTSTRAP). However,
# due to some shortcuts below it’s restricted further to within the 1st
# 64KB. The AP starts in real-mode, with
#   CS selector set to the startup memory address/16;
#   CS base set to startup memory address;
#   CS limit set to 64KB;
#   CPL and IP set to 0.
#
# Bootothers (in main.c) starts each non-boot CPU in turn.
# It puts the correct %esp in start-4,
# and the place to jump to in start-8.
#
# This code is identical to bootasm.S except:
#   - it does not need to enable A20
#   - it uses the address at start-4 for the %esp
#   - it jumps to the address at start-8 instead of calling bootmain

.set PROT_MODE_CSEG, 0x8         # kernel code segment selector
.set PROT_MODE_DSEG, 0x10        # kernel data segment selector
.set CR0_PE_ON,      0x1         # protected mode enable flag

.globl start
start:
  .code16                     # Assemble for 16-bit mode
  cli                         # Disable interrupts
  cld                         # String operations increment

  # Set up the important data segment registers (DS, ES, SS).
  xorw    %ax,%ax             # Segment number zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment
```

```
  # Switch from real to protected mode, using a bootstrap GDT
  # and segment translation that makes virtual addresses
  # identical to their physical addresses, so that the
  # effective memory map does not change during the switch.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE_ON, %eax
  movl    %eax, %cr0

  # Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $PROT_MODE_CSEG, $protcseg

  .code32                     # Assemble for 32-bit mode
protcseg:
  # Set up the protected-mode data segment registers
  movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS
  movw    %ax, %ss                # -> SS: Stack Segment

  movl    start-4, %esp
  movl    start-8, %eax
  jmp     *%eax

# 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   0x17                            # sizeof(gdt) - 1
  .long   gdt                             # address gdt
```

```c
// Boot loader.
//
// The BIOS loads boot sector (bootasm.S) from sector 0 of the disk
// into memory and executes it.  The boot sector puts the processor
// in 32-bit mode and calls bootmain below, which 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"

#define SECTSIZE  512

void readseg(uint, uint, uint);

void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uint)elf, SECTSIZE*8, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    goto bad;

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++)
    readseg(ph->va, ph->memsz, ph->offset);

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry & 0xFFFFFF);
  entry();

bad:
  outw(0x8A00, 0x8A00);
  outw(0x8A00, 0x8E00);
  for(;;)
    ;
}
```

```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 virtual address 'va'.
// Might copy more than asked.
void
readseg(uint va, uint count, uint offset)
{
  uint eva;

  va &= 0xFFFFFF;
  eva = va + count;

  // Round down to sector boundary.
  va &= ~(SECTSIZE - 1);

  // 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(; va < eva; va += SECTSIZE, offset++)
    readsect((uchar*)va, offset);
}
```

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

static void bootothers(void);

// Bootstrap processor starts running C code here.
int
main(void)
{
  int i;
  static volatile int bcpu;  // cannot be on stack
  extern char edata[], end[];

  // clear BSS
  memset(edata, 0, end - edata);

  // Prevent release() from enabling interrupts.
  for(i=0; i<NCPU; i++)
    cpus[i].nlock = 1;

  mp_init(); // collect info about this machine
  bcpu = mp_bcpu();

  // Switch to bootstrap processor’s stack
  asm volatile("movl %0, %%esp" : : "r" (cpus[bcpu].mpstack+MPSTACK-32));
  asm volatile("movl %0, %%ebp" : : "r" (cpus[bcpu].mpstack+MPSTACK));

  lapic_init(bcpu);
  cprintf("\ncpu%d: starting xv6\n\n", cpu());

  pinit();         // process table
  binit();         // buffer cache
  pic_init();      // interrupt controller
  ioapic_init();   // another interrupt controller
  kinit();         // physical memory allocator
  tvinit();        // trap vectors
  idtinit();       // interrupt descriptor table
  fileinit();      // file table
  iinit();         // inode cache
  setupsegs(0);    // segments & TSS
  console_init();  // I/O devices & their interrupts
  ide_init();      // disk
  bootothers();    // boot other CPUs
  if(!ismp)
    timer_init(); // uniprocessor timer
  userinit();      // first user process
```

```c
  // enable interrupts on this processor.
  cpus[cpu()].nlock--;
  sti();

  scheduler();
}

// Additional processors start here.
static void
mpmain(void)
{
  cprintf("cpu%d: starting\n", cpu());
  idtinit();
  lapic_init(cpu());
  setupsegs(0);
  cpuid(0, 0, 0, 0, 0);  // memory barrier
  cpus[cpu()].booted = 1;

  // Enable interrupts on this processor.
  cpus[cpu()].nlock--;
  sti();

  scheduler();
}

static void
bootothers(void)
{
  extern uchar _binary_bootother_start[], _binary_bootother_size[];
  uchar *code;
  struct cpu *c;

  // Write bootstrap code to unused memory at 0x7000.
  code = (uchar*)0x7000;
  memmove(code, _binary_bootother_start, (uint)_binary_bootother_size);

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

    // Fill in %esp, %eip and start code on cpu.
    *(void**)(code-4) = c->mpstack + MPSTACK;
    *(void**)(code-8) = mpmain;
    lapic_startap(c->apicid, (uint)code);

    // Wait for cpu to get through bootstrap.
    while(c->booted == 0)
      ;
  }
}
```

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

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

```
// Blank page.
```

```c
// Mutual exclusion spin locks.

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

extern int use_console_lock;

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

// Acquire the lock.
// Loops (spins) until the lock is acquired.
// (Because contention is handled by spinning,
// must not go to sleep holding any locks.)
void
acquire(struct spinlock *lock)
{
  if(holding(lock))
    panic("acquire");

  if(cpus[cpu()].nlock == 0)
    cli();
  cpus[cpu()].nlock++;

  while(cmpxchg(0, 1, &lock->locked) == 1)
    ;

  // Serialize instructions: now that lock is acquired, make sure
  // we wait for all pending writes from other processors.
  cpuid(0, 0, 0, 0, 0);  // memory barrier (see Ch 7, IA-32 manual vol 3)

  // Record info about lock acquisition for debugging.
  // The +10 is only so that we can tell the difference
  // between forgetting to initialize lock->cpu
  // and holding a lock on cpu 0.
  lock->cpu = cpu() + 10;
  getcallerpcs(&lock, lock->pcs);
}
```

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

  lock->pcs[0] = 0;
  lock->cpu = 0xffffffff;

  // Serialize instructions: before unlocking the lock, make sure
  // to flush any pending memory writes from this processor.
  cpuid(0, 0, 0, 0, 0);  // memory barrier (see Ch 7, IA-32 manual vol 3)

  lock->locked = 0;
  if(--cpus[cpu()].nlock == 0)
    sti();
}

// 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*)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() + 10;
}
```

```
// Segments in proc->gdt
#define SEG_KCODE 1  // kernel code
#define SEG_KDATA 2  // kernel data+stack
#define SEG_UCODE 3
#define SEG_UDATA 4
#define SEG_TSS   5  // this process’s task state
#define NSEGS     6

// Saved registers for kernel context switches.
// Don’t need to save all the %fs etc. segment registers,
// because they are constant across kernel contexts.
// Save all the regular registers so we don’t need to care
// which are caller save, but not the return register %eax.
// (Not saving %eax just simplifies the switching code.)
// The layout of context must match code in swtch.S.
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

enum proc_state { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  char *mem;                // Start of process memory (kernel address)
  uint sz;                  // Size of process memory (bytes)
  char *kstack;             // Bottom of kernel stack for this process
  enum proc_state state;    // Process state
  int pid;                  // Process ID
  struct proc *parent;      // Parent 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
  struct context context;   // Switch here to run process
  struct trapframe *tf;     // Trap frame for current interrupt
  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

// Arrange that cp point to the struct proc that this
// CPU is currently running.  Such preprocessor
// subterfuge can be confusing, but saves a lot of typing.
extern struct proc *curproc[NCPU];  // Current (running) process per CPU
#define cp (curproc[cpu()])  // Current process on this CPU


#define MPSTACK 512

// Per-CPU state
struct cpu {
  uchar apicid;               // Local APIC ID
  struct context context;     // Switch here to enter scheduler
  struct taskstate ts;        // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];  // x86 global descriptor table
  char mpstack[MPSTACK];      // Per-CPU startup stack
  volatile int booted;        // Has the CPU started?
  int nlock;                  // Number of locks currently held
};

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

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

struct spinlock proc_table_lock;

struct proc proc[NPROC];
struct proc *curproc[NCPU];
static struct proc *initproc;

int nextpid = 1;
extern void forkret(void);
extern void forkret1(struct trapframe*);

void
pinit(void)
{
  initlock(&proc_table_lock, "proc_table");
}

// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and return it.
// Otherwise return 0.
static struct proc*
allocproc(void)
{
  int i;
  struct proc *p;

  acquire(&proc_table_lock);
  for(i = 0; i < NPROC; i++){
    p = &proc[i];
    if(p->state == UNUSED){
      p->state = EMBRYO;
      p->pid = nextpid++;
      release(&proc_table_lock);
      return p;
    }
  }
  release(&proc_table_lock);
  return 0;
}
```

```
// Grow current process's memory by n bytes.
// Return old size on success, -1 on failure.
int
growproc(int n)
{
  char *newmem, *oldmem;

  newmem = kalloc(cp->sz + n);
  if(newmem == 0)
    return -1;
  memmove(newmem, cp->mem, cp->sz);
  memset(newmem + cp->sz, 0, n);
  oldmem = cp->mem;
  cp->mem = newmem;
  kfree(oldmem, cp->sz);
  cp->sz += n;
  return cp->sz - n;
}

// Set up CPU's segment descriptors and task state for a given process.
// If p==0, set up for "idle" state for when scheduler() is running.
void
setupsegs(struct proc *p)
{
  struct cpu *c;

  c = &cpus[cpu()];
  c->ts.ss0 = SEG_KDATA << 3;
  if(p)
    c->ts.esp0 = (uint)(p->kstack + KSTACKSIZE);
  else
    c->ts.esp0 = 0xffffffff;

  c->gdt[0] = SEG_NULL;
  c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0x100000 + 64*1024-1, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c->gdt[SEG_TSS] = SEG16(STS_T32A, (uint)&c->ts, sizeof(c->ts)-1, 0);
  c->gdt[SEG_TSS].s = 0;
  if(p){
    c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, (uint)p->mem, p->sz-1, DPL_USER);
    c->gdt[SEG_UDATA] = SEG(STA_W, (uint)p->mem, p->sz-1, DPL_USER);
  } else {
    c->gdt[SEG_UCODE] = SEG_NULL;
    c->gdt[SEG_UDATA] = SEG_NULL;
  }

  lgdt(c->gdt, sizeof(c->gdt));
  ltr(SEG_TSS << 3);
}
```

```
// 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.
struct proc*
copyproc(struct proc *p)
{
  int i;
  struct proc *np;

  // Allocate process.
  if((np = allocproc()) == 0)
    return 0;

  // Allocate kernel stack.
  if((np->kstack = kalloc(KSTACKSIZE)) == 0){
    np->state = UNUSED;
    return 0;
  }
  np->tf = (struct trapframe*)(np->kstack + KSTACKSIZE) - 1;

  if(p){  // Copy process state from p.
    np->parent = p;
    memmove(np->tf, p->tf, sizeof(*np->tf));

    np->sz = p->sz;
    if((np->mem = kalloc(np->sz)) == 0){
      kfree(np->kstack, KSTACKSIZE);
      np->kstack = 0;
      np->state = UNUSED;
      return 0;
    }
    memmove(np->mem, p->mem, np->sz);

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

  // Set up new context to start executing at forkret (see below).
  memset(&np->context, 0, sizeof(np->context));
  np->context.eip = (uint)forkret;
  np->context.esp = (uint)np->tf;

  // Clear %eax so that fork system call returns 0 in child.
  np->tf->eax = 0;
  return np;
}
```

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

  p = copyproc(0);
  p->sz = PAGE;
  p->mem = kalloc(p->sz);
  p->cwd = namei("/");
  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 = p->sz;

  // Make return address readable; needed for some gcc.
  p->tf->esp -= 4;
  *(uint*)(p->mem + p->tf->esp) = 0xefefefef;

  // On entry to user space, start executing at beginning of initcode.S.
  p->tf->eip = 0;
  memmove(p->mem, _binary_initcode_start, (int)_binary_initcode_size);
  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->state = RUNNABLE;

  initproc = p;
}
```

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

  for(;;){
    // Loop over process table looking for process to run.
    acquire(&proc_table_lock);

    for(i = 0; i < NPROC; i++){
      p = &proc[i];
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release proc_table_lock and then reacquire it
      // before jumping back to us.
      cp = p;
      setupsegs(p);
      p->state = RUNNING;
      swtch(&cpus[cpu()].context, &p->context);

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      cp = 0;
      setupsegs(0);
    }

    release(&proc_table_lock);
  }
}
```

```
// Enter scheduler.  Must already hold proc_table_lock
// and have changed curproc[cpu()]->state.
void
sched(void)
{
  if(cp->state == RUNNING)
    panic("sched running");
  if(!holding(&proc_table_lock))
    panic("sched proc_table_lock");
  if(cpus[cpu()].nlock != 1)
    panic("sched locks");

  swtch(&cp->context, &cpus[cpu()].context);
}

// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&proc_table_lock);
  cp->state = RUNNABLE;
  sched();
  release(&proc_table_lock);
}

// A fork child's very first scheduling by scheduler()
// will longjmp here.  "Return" to user space.
void
forkret(void)
{
  // Still holding proc_table_lock from scheduler.
  release(&proc_table_lock);

  // Jump into assembly, never to return.
  forkret1(cp->tf);
}
```

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

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

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

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

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

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

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

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

// Wake up all processes sleeping on chan.
// Proc_table_lock is acquired and released.
void
wakeup(void *chan)
{
  acquire(&proc_table_lock);
  wakeup1(chan);
  release(&proc_table_lock);
}

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

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

```
// Exit the current process.  Does not return.
// Exited processes remain in the zombie state
// until their parent calls wait() to find out they exited.
void
exit(void)
{
  struct proc *p;
  int fd;

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

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

  iput(cp->cwd);
  cp->cwd = 0;

  acquire(&proc_table_lock);

  // Parent might be sleeping in proc_wait.
  wakeup1(cp->parent);

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

  // Jump into the scheduler, never to return.
  cp->killed = 0;
  cp->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

```
// 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 i, havekids, pid;

  acquire(&proc_table_lock);
  for(;;){
    // Scan through table looking for zombie children.
    havekids = 0;
    for(i = 0; i < NPROC; i++){
      p = &proc[i];
      if(p->state == UNUSED)
        continue;
      if(p->parent == cp){
        if(p->state == ZOMBIE){
          // Found one.
          kfree(p->mem, p->sz);
          kfree(p->kstack, KSTACKSIZE);
          pid = p->pid;
          p->state = UNUSED;
          p->pid = 0;
          p->parent = 0;
          p->name[0] = 0;
          release(&proc_table_lock);
          return pid;
        }
        havekids = 1;
      }
    }

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

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

```
// 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, j;
  struct proc *p;
  char *state;
  uint pc[10];

  for(i = 0; i < NPROC; i++){
    p = &proc[i];
    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(j=0; j<10 && pc[j] != 0; j++)
        cprintf(" %p", pc[j]);
    }
    cprintf("\n");
  }
}
```

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

.globl swtch
swtch:
  # Save old registers
  movl 4(%esp), %eax

  popl 0(%eax)  # %eip
  movl %esp, 4(%eax)
  movl %ebx, 8(%eax)
  movl %ecx, 12(%eax)
  movl %edx, 16(%eax)
  movl %esi, 20(%eax)
  movl %edi, 24(%eax)
  movl %ebp, 28(%eax)

  # Load new registers
  movl 4(%esp), %eax  # not 8(%esp) - popped return address above

  movl 28(%eax), %ebp
  movl 24(%eax), %edi
  movl 20(%eax), %esi
  movl 16(%eax), %edx
  movl 12(%eax), %ecx
  movl 8(%eax), %ebx
  movl 4(%eax), %esp
  pushl 0(%eax)  # %eip

  ret
```

```
// Physical memory allocator, intended to allocate
// memory for user processes. Allocates in 4096-byte "pages".
// Free list is kept sorted and combines adjacent pages into
// long runs, to make it easier to allocate big segments.
// One reason the page size is 4k is that the x86 segment size
// granularity is 4k.

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

struct spinlock kalloc_lock;

struct run {
  struct run *next;
  int len; // bytes
};
struct run *freelist;

// Initialize free list of physical pages.
// This code cheats by just considering one megabyte of
// pages after _end.  Real systems would determine the
// amount of memory available in the system and use it all.
void
kinit(void)
{
  extern int end;
  uint mem;
  char *start;

  initlock(&kalloc_lock, "kalloc");
  start = (char*) &end;
  start = (char*) (((uint)start + PAGE) & ~(PAGE-1));
  mem = 256; // assume computer has 256 pages of RAM
  cprintf("mem = %d\n", mem * PAGE);
  kfree(start, mem * PAGE);
}
```

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

  if(len <= 0 || len % PAGE)
    panic("kfree");

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

  acquire(&kalloc_lock);
  p = (struct run*)v;
  pend = (struct run*)(v + len);
  for(rp=&freelist; (r=*rp) != 0 && r <= pend; rp=&r->next){
    rend = (struct run*)((char*)r + r->len);
    if(r <= p && p < rend)
      panic("freeing free page");
    if(pend == r){  // p next to r: replace r with p
      p->len = len + r->len;
      p->next = r->next;
      *rp = p;
      goto out;
    }
    if(rend == p){  // r next to p: replace p with r
      r->len += len;
      if(r->next && r->next == pend){  // r now next to r->next?
        r->len += r->next->len;
        r->next = r->next->next;
      }
      goto out;
    }
  }
  // Insert p before r in list.
  p->len = len;
  p->next = r;
  *rp = p;

 out:
  release(&kalloc_lock);
}
```

```
// Allocate n bytes of physical memory.
// Returns a kernel-segment pointer.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(int n)
{
  char *p;
  struct run *r, **rp;

  if(n % PAGE || n <= 0)
    panic("kalloc");

  acquire(&kalloc_lock);
  for(rp=&freelist; (r=*rp) != 0; rp=&r->next){
    if(r->len == n){
      *rp = r->next;
      release(&kalloc_lock);
      return (char*)r;
    }
    if(r->len > n){
      r->len -= n;
      p = (char*)r + r->len;
      release(&kalloc_lock);
      return p;
    }
  }
  release(&kalloc_lock);

  cprintf("kalloc: out of memory\n");
  return 0;
}
```

```
// 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       48      // system call
#define T_DEFAULT      500      // catchall

#define IRQ_OFFSET      32      // IRQ 0 corresponds to int IRQ_OFFSET

#define IRQ_TIMER        0
#define IRQ_KBD          1
#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 ".text\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 > 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
#   .text
#   .globl alltraps
#   .globl vector0
#   vector0:
#     pushl $0
#     pushl $0
#     jmp alltraps
#   ...
#
#   # vector table
#   .data
#   .globl vectors
#   vectors:
#     .long vector0
#     .long vector1
#     .long vector2
#   ...
```

```
.text

.set SEG_KDATA_SEL, 0x10   # selector for SEG_KDATA

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

  # Set up data segments.
  movl $SEG_KDATA_SEL, %eax
  movw %ax,%ds
  movw %ax,%es

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

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

  # A forked process switches to user mode by calling
  # forkret1(tf), where tf is the trap frame to use.
.globl forkret1
forkret1:
  movl 4(%esp), %esp
  jmp trapret
```

```c
#include "types.h"
#include "defs.h"
#include "param.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;
int 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], 0, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}

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

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

  // Increment nlock to make sure interrupts stay off
  // during interrupt handler.  Decrement before returning.
  cpus[cpu()].nlock++;

```

```c
  switch(tf->trapno){
  case IRQ_OFFSET + IRQ_TIMER:
    if(cpu() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapic_eoi();
    break;
  case IRQ_OFFSET + IRQ_IDE:
    ide_intr();
    lapic_eoi();
    break;
  case IRQ_OFFSET + IRQ_KBD:
    kbd_intr();
    lapic_eoi();
    break;
  case IRQ_OFFSET + IRQ_SPURIOUS:
    cprintf("spurious interrupt from cpu %d eip %x\n", cpu(), tf->eip);
    lapic_eoi();
    break;

  default:
    if(cp == 0){
      // Otherwise it's our mistake.
      cprintf("unexpected trap %d from cpu %d eip %x\n",
              tf->trapno, cpu(), tf->eip);
      panic("trap");
    }
    // Assume process divided by zero or dereferenced null, etc.
    cprintf("pid %d %s: trap %d err %d on cpu %d eip %x -- kill proc\n",
            cp->pid, cp->name, tf->trapno, tf->err, cpu(), tf->eip);
    cp->killed = 1;
  }
  cpus[cpu()].nlock--;

  // 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(cp && cp->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(cp && cp->state == RUNNING && tf->trapno == IRQ_OFFSET+IRQ_TIMER)
    yield();
}


```

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

```
#include "types.h"
#include "defs.h"
#include "param.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 process p.
int
fetchint(struct proc *p, uint addr, int *ip)
{
  if(addr >= p->sz || addr+4 > p->sz)
    return -1;
  *ip = *(int*)(p->mem + addr);
  return 0;
}

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

  if(addr >= p->sz)
    return -1;
  *pp = p->mem + addr;
  ep = p->mem + p->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(cp, cp->tf->esp + 4 + 4*n, ip);
}
```

```
// 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 >= cp->sz || (uint)i+size >= cp->sz)
    return -1;
  *pp = cp->mem + 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(cp, 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);

```

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

void
syscall(void)
{
  int num;

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

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "mmu.h"
#include "proc.h"

int
sys_fork(void)
{
  struct proc *np;

  if((np = copyproc(cp)) == 0)
    return -1;
  np->state = RUNNABLE;
  return np->pid;
}

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 cp->pid;
}
```

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

  if(argint(0, &n) < 0)
    return -1;
  if((addr = growproc(n)) < 0)
    return -1;
  setupsegs(cp);
  return addr;
}

int
sys_sleep(void)
{
  int n, ticks0;

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

```
struct buf {
  int flags;
  uint dev;
  uint sector;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[512];
};
#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
```

```
struct devsw {
  int (*read)(struct inode*, char*, int);
  int (*write)(struct inode*, char*, int);
};

extern struct devsw devsw[];

#define CONSOLE 1
```

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

```
struct stat {
  int dev;     // Device number
  uint ino;    // Inode number on device
  short type;  // Type of file
  short nlink; // Number of links to file
  uint size;   // Size of file in bytes
};
```

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

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

// Block 0 is unused.
// Block 1 is super block.
// Inodes start at block 2.

#define BSIZE 512  // block size

// File system super block
struct superblock {
  uint size;         // Size of file system image (blocks)
  uint nblocks;      // Number of data blocks
  uint ninodes;      // Number of inodes.
};

#define NADDRS (NDIRECT+1)
#define NDIRECT 12
#define INDIRECT 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[NADDRS];   // Data block addresses
};

#define T_DIR  1   // Directory
#define T_FILE 2   // File
#define T_DEV  3   // Special device

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

// Block containing inode i
#define IBLOCK(i)     ((i) / IPB + 2)

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

// Block containing bit for block b
#define BBLOCK(b, ninodes) (b/BPB + (ninodes)/IPB + 3)
```

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

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

```
// in-core file system types

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[NADDRS];
};

#define I_BUSY 0x1
#define I_VALID 0x2
```

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

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

#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

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

static struct spinlock ide_lock;
static struct buf *ide_queue;

static int disk_1_present;
static void ide_start_request();

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

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

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

  initlock(&ide_lock, "ide");
  pic_enable(IRQ_IDE);
  ioapic_enable(IRQ_IDE, ncpu - 1);
  ide_wait_ready(0);

  // Check if disk 1 is present
  outb(0x1f6, 0xe0 | (1<<4));
  for(i=0; i<1000; i++){
    if(inb(0x1f7) != 0){
      disk_1_present = 1;
      break;
    }
  }

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

// Start the request for b.  Caller must hold ide_lock.
static void
ide_start_request(struct buf *b)
{
  if(b == 0)
    panic("ide_start_request");

  ide_wait_ready(0);
  outb(0x3f6, 0);  // generate interrupt
  outb(0x1f2, 1);  // number of sectors
  outb(0x1f3, b->sector & 0xff);
  outb(0x1f4, (b->sector >> 8) & 0xff);
  outb(0x1f5, (b->sector >> 16) & 0xff);
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);
    outsl(0x1f0, b->data, 512/4);
  } else {
    outb(0x1f7, IDE_CMD_READ);
  }
}
```

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

  acquire(&ide_lock);
  if((b = ide_queue) == 0){
    cprintf("stray ide interrupt\n");
    release(&ide_lock);
    return;
  }

  // Read data if needed.
  if(!(b->flags & B_DIRTY) && ide_wait_ready(1) >= 0)
    insl(0x1f0, b->data, 512/4);

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

  // Start disk on next buf in queue.
  if((ide_queue = b->qnext) != 0)
    ide_start_request(ide_queue);

  release(&ide_lock);
}
```

```
// 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
ide_rw(struct buf *b)
{
  struct buf **pp;

  if(!(b->flags & B_BUSY))
    panic("ide_rw: buf not busy");
  if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
    panic("ide_rw: nothing to do");
  if(b->dev != 0 && !disk_1_present)
    panic("ide disk 1 not present");

  acquire(&ide_lock);

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

  // Start disk if necessary.
  if(ide_queue == b)
    ide_start_request(b);

  // Wait for request to finish.
  // Assuming will not sleep too long: ignore cp->killed.
  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
    sleep(b, &ide_lock);

  release(&ide_lock);
}
```

```
// 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 flush 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 initialized
//     with the associated disk block contents.
// * 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 "buf.h"

struct buf buf[NBUF];
struct spinlock buf_table_lock;

// Linked list of all buffers, through prev/next.
// bufhead->next is most recently used.
// bufhead->tail is least recently used.
struct buf bufhead;

void
binit(void)
{
  struct buf *b;

  initlock(&buf_table_lock, "buf_table");
```

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

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

  acquire(&buf_table_lock);

 loop:
  // Try for cached block.
  for(b = bufhead.next; b != &bufhead; b = b->next){
    if((b->flags & (B_BUSY|B_VALID)) &&
       b->dev == dev && b->sector == sector){
      if(b->flags & B_BUSY){
        sleep(buf, &buf_table_lock);
        goto loop;
      }
      b->flags |= B_BUSY;
      release(&buf_table_lock);
      return b;
    }
  }

  // Allocate fresh block.
  for(b = bufhead.prev; b != &bufhead; b = b->prev){
    if((b->flags & B_BUSY) == 0){
      b->flags = B_BUSY;
      b->dev = dev;
      b->sector = sector;
      release(&buf_table_lock);
      return b;
    }
  }
  panic("bget: no buffers");
}
```

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

  b = bget(dev, sector);
  if(!(b->flags & B_VALID))
    ide_rw(b);
  return b;
}

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

// Release the buffer buf.
void
brelse(struct buf *b)
{
  if((b->flags & B_BUSY) == 0)
    panic("brelse");

  acquire(&buf_table_lock);

  b->next->prev = b->prev;
  b->prev->next = b->next;
  b->next = bufhead.next;
  b->prev = &bufhead;
  bufhead.next->prev = b;
  bufhead.next = b;

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

  release(&buf_table_lock);
}
```

```
// File system implementation.  Four layers:
//   + Blocks: allocator for raw disk blocks.
//   + 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.
//
// Disk layout is: superblock, inodes, block in-use bitmap, data blocks.
//
// 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 "buf.h"
#include "fs.h"
#include "fsvar.h"
#include "dev.h"

#define min(a, b) ((a) < (b) ? (a) : (b))
static void itrunc(struct inode*);

// Read the super block.
static 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);
  bwrite(bp);
  brelse(bp);
}
```

```
// Blocks.

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

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

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

  bzero(dev, b);

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

```
// Inodes.
//
// An inode is a single, unnamed file in the file system.
// The inode disk structure holds metadata (the type, device numbers,
// and data size) along with a list of blocks where the associated
// data can be found.
//
// The inodes are laid out sequentially on disk immediately after
// the superblock.  The kernel keeps a cache of the in-use
// on-disk structures to provide a place for synchronizing access
// to inodes shared between multiple processes.
//
// ip->ref counts the number of pointer references to this cached
// inode; references are typically kept in struct file and in cp->cwd.
// When ip->ref falls to zero, the inode is no longer cached.
// It is an error to use an inode without holding a reference to it.
//
// Processes are only allowed to read and write inode
// metadata and contents when holding the inode’s lock,
// represented by the I_BUSY flag in the in-memory copy.
// Because inode locks are held during disk accesses,
// they are implemented using a flag rather than with
// spin locks.  Callers are responsible for locking
// inodes before passing them to routines in this file; leaving
// this responsibility with the caller makes it possible for them
// to create arbitrarily-sized atomic operations.
//
// To give maximum control over locking to the callers,
// the routines in this file that return inode pointers
// return pointers to *unlocked* inodes.  It is the callers’
// responsibility to lock them before using them.  A non-zero
// ip->ref keeps these unlocked inodes in the cache.

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

void
iinit(void)
{
  initlock(&icache.lock, "icache.lock");
}
```

```
// Find the inode with number inum on device dev
// and return the in-memory copy.
static struct inode*
iget(uint dev, uint inum)
{
  struct inode *ip, *empty;

  acquire(&icache.lock);

  // Try for cached inode.
  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;
  }

  // Allocate fresh inode.
  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.
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));
    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);
}
```

```
// Caller holds reference to unlocked ip.  Drop reference.
void
iput(struct inode *ip)
{
  acquire(&icache.lock);
  if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
    // inode is no longer used: truncate and free inode.
    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 &= ~I_BUSY;
    wakeup(ip);
  }
  ip->ref--;
  release(&icache.lock);
}

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

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

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

// Copy inode, which has changed, from memory to disk.
void
iupdate(struct inode *ip)
{
  struct buf *bp;
  struct dinode *dip;

  bp = bread(ip->dev, IBLOCK(ip->inum));
  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));
  bwrite(bp);
  brelse(bp);
}
```

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

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

  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0){
      if(!alloc)
        return -1;
      ip->addrs[bn] = addr = balloc(ip->dev);
    }
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[INDIRECT]) == 0){
      if(!alloc)
        return -1;
      ip->addrs[INDIRECT] = addr = balloc(ip->dev);
    }
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;

    if((addr = a[bn]) == 0){
      if(!alloc){
        brelse(bp);
        return -1;
      }
      a[bn] = addr = balloc(ip->dev);
      bwrite(bp);
    }
    brelse(bp);
    return addr;
  }

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

```
// Truncate inode (discard contents).
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[INDIRECT]){
    bp = bread(ip->dev, ip->addrs[INDIRECT]);
    a = (uint*)bp->data;
    for(j = 0; j < NINDIRECT; j++){
      if(a[j])
        bfree(ip->dev, a[j]);
    }
    brelse(bp);
    ip->addrs[INDIRECT] = 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, 0));
    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 + n < off)
    return -1;
  if(off + n > MAXFILE*BSIZE)
    n = MAXFILE*BSIZE - off;

  for(tot=0; tot<n; tot+=m, off+=m, src+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE, 1));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(bp->data + off%BSIZE, src, m);
    bwrite(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.
// Caller must have already locked dp.
struct inode*
dirlookup(struct inode *dp, char *name, uint *poff)
{
  uint off, inum;
  struct buf *bp;
  struct dirent *de;

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

  for(off = 0; off < dp->size; off += BSIZE){
    bp = bread(dp->dev, bmap(dp, off / BSIZE, 0));
    for(de = (struct dirent*)bp->data;
        de < (struct dirent*)(bp->data + BSIZE);
        de++){
      if(de->inum == 0)
        continue;
      if(namecmp(name, de->name) == 0){
        // entry matches path element
        if(poff)
          *poff = off + (uchar*)de - bp->data;
        inum = de->inum;
        brelse(bp);
        return iget(dp->dev, inum);
      }
    }
    brelse(bp);
  }
  return 0;
}
```

```
// Write a new directory entry (name, ino) into the directory dp.
int
dirlink(struct inode *dp, char *name, uint ino)
{
  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 = ino;
  if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
    panic("dirlink");

  return 0;
}
```

```
// 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("", 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.
static struct inode*
_namei(char *path, int parent, char *name)
{
  struct inode *ip, *next;

  if(*path == ’/’)
    ip = iget(ROOTDEV, 1);
  else
    ip = idup(cp->cwd);

  while((path = skipelem(path, name)) != 0){
    ilock(ip);
    if(ip->type != T_DIR){
      iunlockput(ip);
      return 0;
    }
    if(parent && *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(parent){
    iput(ip);
    return 0;
  }
  return ip;
}

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

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

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "file.h"
#include "spinlock.h"
#include "dev.h"

struct devsw devsw[NDEV];
struct spinlock file_table_lock;
struct file file[NFILE];

void
fileinit(void)
{
  initlock(&file_table_lock, "file_table");
}

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

  acquire(&file_table_lock);
  for(i = 0; i < NFILE; i++){
    if(file[i].type == FD_CLOSED){
      file[i].type = FD_NONE;
      file[i].ref = 1;
      release(&file_table_lock);
      return file + i;
    }
  }
  release(&file_table_lock);
  return 0;
}

// Increment ref count for file f.
struct file*
filedup(struct file *f)
{
  acquire(&file_table_lock);
  if(f->ref < 1 || f->type == FD_CLOSED)
    panic("filedup");
  f->ref++;
  release(&file_table_lock);
  return f;
}
```

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

  acquire(&file_table_lock);
  if(f->ref < 1 || f->type == FD_CLOSED)
    panic("fileclose");
  if(--f->ref > 0){
    release(&file_table_lock);
    return;
  }
  ff = *f;
  f->ref = 0;
  f->type = FD_CLOSED;
  release(&file_table_lock);

  if(ff.type == FD_PIPE)
    pipeclose(ff.pipe, ff.writable);
  else if(ff.type == FD_INODE)
    iput(ff.ip);
  else
    panic("fileclose");
}

// 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.  Addr is kernel address.
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.  Addr is kernel address.
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){
    ilock(f->ip);
    if((r = writei(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
  panic("filewrite");
}
```

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "stat.h"
#include "mmu.h"
#include "proc.h"
#include "fs.h"
#include "fsvar.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=cp->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(cp->ofile[fd] == 0){
      cp->ofile[fd] = f;
      return fd;
    }
  }
  return -1;
}
```

```c
int
sys_read(void)
{
  struct file *f;
  int n;
  char *cp;

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

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

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

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

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

```c
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;
  if((ip = namei(old)) == 0)
    return -1;
  ilock(ip);
  if(ip->type == T_DIR){
    iunlockput(ip);
    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)
    goto bad;
  iunlockput(dp);
  iput(ip);
  return 0;

bad:
  if(dp)
    iunlockput(dp);
  ilock(ip);
  ip->nlink--;
  iupdate(ip);
  iunlockput(ip);
  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;
  if((dp = nameiparent(path, name)) == 0)
    return -1;
  ilock(dp);

  // Cannot unlink "." or "..".
  if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0){
    iunlockput(dp);
    return -1;
  }

  if((ip = dirlookup(dp, name, &off)) == 0){
    iunlockput(dp);
    return -1;
  }
  ilock(ip);

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

  memset(&de, 0, sizeof(de));
  if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
    panic("unlink: writei");
  iunlockput(dp);

  ip->nlink--;
  iupdate(ip);
  iunlockput(ip);
  return 0;
}
```

```
static struct inode*
create(char *path, int canexist, 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(canexist && (ip = dirlookup(dp, name, &off)) != 0){
    iunlockput(dp);
    ilock(ip);
    if(ip->type != type || ip->major != major || ip->minor != minor){
      iunlockput(ip);
      return 0;
    }
    return ip;
  }

  if((ip = ialloc(dp->dev, type)) == 0){
    iunlockput(dp);
    return 0;
  }
  ilock(ip);
  ip->major = major;
  ip->minor = minor;
  ip->nlink = 1;
  iupdate(ip);

  if(dirlink(dp, name, ip->inum) < 0){
    ip->nlink = 0;
    iunlockput(ip);
    iunlockput(dp);
    return 0;
  }

  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");
  }
  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;

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

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

  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;
}
```

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

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

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

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

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

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

```c
int
sys_exec(void)
{
  char *path, *argv[20];
  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(cp, uargv+4*i, (int*)&uarg) < 0)
      return -1;
    if(uarg == 0){
      argv[i] = 0;
      break;
    }
    if(fetchstr(cp, 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)
      cp->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 "mmu.h"
#include "proc.h"
#include "defs.h"
#include "x86.h"
#include "elf.h"

int
exec(char *path, char **argv)
{
  char *mem, *s, *last;
  int i, argc, arglen, len, off;
  uint sz, sp, argp;
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;

  if((ip = namei(path)) == 0)
    return -1;
  ilock(ip);

  // Compute memory size of new process.
  mem = 0;
  sz = 0;

  // Program segments.
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;
  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;
    sz += ph.memsz;
  }

  // Arguments.
  arglen = 0;
  for(argc=0; argv[argc]; argc++)
    arglen += strlen(argv[i]) + 1;
  arglen = (arglen+3) & ~3;
  sz += arglen + 4*(argc+1);

  // Stack.
  sz += PAGE;
```

```c
  // Allocate program memory.
  sz = (sz+PAGE-1) & ~(PAGE-1);
  mem = kalloc(sz);
  if(mem == 0)
    goto bad;
  memset(mem, 0, sz);

  // Load program into memory.
  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.va + ph.memsz > sz)
      goto bad;
    if(readi(ip, mem + ph.va, ph.offset, ph.filesz) != ph.filesz)
      goto bad;
    memset(mem + ph.va + ph.filesz, 0, ph.memsz - ph.filesz);
  }
  iunlockput(ip);

  // Initialize stack.
  sp = sz;
  argp = sz - arglen - 4*(argc+1);

  // Copy argv strings and pointers to stack.
  *(uint*)(mem+argp + 4*argc) = 0;  // argv[argc]
  for(i=argc-1; i>=0; i--){
    len = strlen(argv[i]) + 1;
    sp -= len;
    memmove(mem+sp, argv[i], len);
    *(uint*)(mem+argp + 4*i) = sp;  // argv[i]
  }

  // Stack frame for main(argc, argv), below arguments.
  sp = argp;
  sp -= 4;
  *(uint*)(mem+sp) = argp;
  sp -= 4;
  *(uint*)(mem+sp) = argc;
  sp -= 4;
  *(uint*)(mem+sp) = 0xffffffff;   // fake return pc

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


  // Commit to the new image.
  kfree(cp->mem, cp->sz);
  cp->mem = mem;
  cp->sz = sz;
  cp->tf->eip = elf.entry;  // main
  cp->tf->esp = sp;
  setupsegs(cp);
  return 0;

 bad:
  if(mem)
    kfree(mem, sz);
  iunlockput(ip);
  return -1;
}
```

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

#define PIPESIZE 512

struct pipe {
  int readopen;   // read fd is still open
  int writeopen;  // write fd is still open
  int writep;     // next index to write
  int readp;      // next index to read
  struct spinlock lock;
  char data[PIPESIZE];
};

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(PAGE)) == 0)
    goto bad;
  p->readopen = 1;
  p->writeopen = 1;
  p->writep = 0;
  p->readp = 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, PAGE);
  if(*f0){
    (*f0)->type = FD_NONE;
    fileclose(*f0);
  }
  if(*f1){
    (*f1)->type = FD_NONE;
    fileclose(*f1);
  }
  return -1;
}

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

  if(p->readopen == 0 && p->writeopen == 0)
    kfree((char*)p, PAGE);
}
```

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

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

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

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

```c
#include "types.h"

void*
memset(void *dst, int c, uint n)
{
  char *d;

  d = (char*)dst;
  while(n-- > 0)
    *d++ = c;

  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
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
```

```c
// Multiprocessor bootstrap.
// 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 "mp.h"
#include "x86.h"
#include "mmu.h"
#include "proc.h"

struct cpu cpus[NCPU];
static struct cpu *bcpu;
int ismp;
int ncpu;
uchar ioapic_id;

int
mp_bcpu(void)
{
  return bcpu-cpus;
}

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*
mp_search1(uchar *addr, int len)
{
  uchar *e, *p;

  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*
mp_search(void)
{
  uchar *bda;
  uint p;
  struct mp *mp;

  bda = (uchar*)0x400;
  if((p = (bda[0x0F]<<8)|bda[0x0E])){
    if((mp = mp_search1((uchar*)p, 1024)))
      return mp;
  } else {
    p = ((bda[0x14]<<8)|bda[0x13])*1024;
    if((mp = mp_search1((uchar*)p-1024, 1024)))
      return mp;
  }
  return mp_search1((uchar*)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*
mp_config(struct mp **pmp)
{
  struct mpconf *conf;
  struct mp *mp;

  if((mp = mp_search()) == 0 || mp->physaddr == 0)
    return 0;
  conf = (struct mpconf*)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
mp_init(void)
{
  uchar *p, *e;
  struct mp *mp;
  struct mpconf *conf;
  struct mpproc *proc;
  struct mpioapic *ioapic;

  bcpu = &cpus[ncpu];
  if((conf = mp_config(&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;
      cpus[ncpu].apicid = proc->apicid;
      if(proc->flags & MPBOOT)
        bcpu = &cpus[ncpu];
      ncpu++;
      p += sizeof(struct mpproc);
      continue;
    case MPIOAPIC:
      ioapic = (struct mpioapic*)p;
      ioapic_id = ioapic->apicno;
      p += sizeof(struct mpioapic);
      continue;
    case MPBUS:
    case MPIOINTR:
    case MPLINTR:
      p += 8;
      continue;
    default:
      cprintf("mp_init: unknown config type %x\n", *p);
      panic("mp_init");
    }
  }

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

// 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 LEVEL      0x00008000   // Level triggered
  #define BCAST      0x00080000   // Send to all APICs, including self.
#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
```

```c
void
lapic_init(int c)
{
  if(!lapic)
    return;

  // Enable local APIC; set spurious interrupt vector.
  lapic[SVR] = ENABLE | (IRQ_OFFSET+IRQ_SPURIOUS);

  // The timer repeatedly counts down at bus frequency
  // from lapic[TICR] and then issues an interrupt.
  // Lapic[TCCR] is the current counter value.
  // If xv6 cared more about precise timekeeping, the
  // values of TICR and TCCR would be calibrated using
  // an external time source.
  lapic[TDCR] = X1;
  lapic[TICR] = 10000000;
  lapic[TCCR] = 10000000;
  lapic[TIMER] = PERIODIC | (IRQ_OFFSET + IRQ_TIMER);

  // Disable logical interrupt lines.
  lapic[LINT0] = MASKED;
  lapic[LINT1] = MASKED;

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

  // Map error interrupt to IRQ_ERROR.
  lapic[ERROR] = IRQ_OFFSET+IRQ_ERROR;

  // Clear error status register (requires back-to-back writes).
  lapic[ESR] = 0;
  lapic[ESR] = 0;

  // Ack any outstanding interrupts.
  lapic[EOI] = 0;

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

  // Enable interrupts on the APIC (but not on the processor).
  lapic[TPR] = 0;
}
```

```c
int
cpu(void)
{
  if(lapic)
    return lapic[ID]>>24;
  return 0;
}

// Acknowledge interrupt.
void
lapic_eoi(void)
{
  if(lapic)
    lapic[EOI] = 0;
}

// Spin for a given number of microseconds.
// On real hardware would want to tune this dynamically.
static void
microdelay(int us)
{
  volatile int j = 0;

  while(us-- > 0)
    for(j=0; j<10000; j++);
}

// Start additional processor running bootstrap code at addr.
// See Appendix B of MultiProcessor Specification.
void
lapic_startap(uchar apicid, uint addr)
{
  int i;
  volatile int j = 0;

  // Send INIT interrupt to reset other CPU.
  lapic[ICRHI] = apicid<<24;
  lapic[ICRLO] = INIT | LEVEL;
  microdelay(10);

  // Send startup IPI (twice!) to enter bootstrap code.
  for(i = 0; i < 2; i++){
    lapic[ICRHI] = apicid<<24;
    lapic[ICRLO] = STARTUP | (addr>>12);
    for(j=0; j<10000; j++);  // 200us
  }
}
```

```c
// 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   0x00100000  // 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
ioapic_read(int reg)
{
  ioapic->reg = reg;
  return ioapic->data;
}

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

```c
void
ioapic_init(void)
{
  int i, id, maxintr;

  if(!ismp)
    return;

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

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

void
ioapic_enable(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.
  ioapic_write(REG_TABLE+2*irq, IRQ_OFFSET + irq);
  ioapic_write(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
pic_setmask(ushort mask)
{
  irqmask = mask;
  outb(IO_PIC1+1, mask);
  outb(IO_PIC2+1, mask >> 8);
}

void
pic_enable(int irq)
{
  pic_setmask(irqmask & ~(1<<irq));
}

// Initialize the 8259A interrupt controllers.
void
pic_init(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, IRQ_OFFSET);
```

```
  // 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, IRQ_OFFSET + 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)
    pic_setmask(irqmask);
}
```

```
// 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
kbd_getc(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
kbd_intr(void)
{
  console_intr(kbd_getc);
}
```

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

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

#define CRTPORT 0x3d4
#define LPTPORT 0x378
#define BACKSPACE 0x100

static ushort *crt = (ushort*)0xb8000;  // CGA memory

static struct spinlock console_lock;
int panicked = 0;
int use_console_lock = 0;

// Copy console output to parallel port, which you can tell
// .bochsrc to copy to the stdout:
//   parport1: enabled=1, file="/dev/stdout"
static void
lpt_putc(int c)
{
  int i;

  for(i = 0; !(inb(LPTPORT+1) & 0x80) && i < 12800; i++)
    ;
  if(c == BACKSPACE)
    c = ’\b’;
  outb(LPTPORT+0, c);
  outb(LPTPORT+2, 0x08|0x04|0x01);
  outb(LPTPORT+2, 0x08);
}
```

```
static void
cga_putc(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)
      crt[--pos] = ’ ’ | 0x0700;
  } else
    crt[pos++] = (c&0xff) | 0x0700;  // black on white

  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;
}

static void
cons_putc(int c)
{
  if(panicked){
    cli();
    for(;;)
      ;
  }

  lpt_putc(c);
  cga_putc(c);
}
```

```
void
printint(int xx, int base, int sgn)
{
  static char digits[] = "0123456789ABCDEF";
  char buf[16];
  int i = 0, neg = 0;
  uint x;

  if(sgn && xx < 0){
    neg = 1;
    x = 0 - xx;
  } else {
    x = xx;
  }

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

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

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

  locking = use_console_lock;
  if(locking)
    acquire(&console_lock);

  argp = (uint*)(void*)&fmt + 1;
  state = 0;
  for(i = 0; fmt[i]; i++){
    c = fmt[i] & 0xff;
    switch(state){
    case 0:
      if(c == ’%’)
        state = ’%’;
      else
        cons_putc(c);
      break;


    case ’%’:
      switch(c){
      case ’d’:
        printint(*argp++, 10, 1);
        break;
      case ’x’:
      case ’p’:
        printint(*argp++, 16, 0);
        break;
      case ’s’:
        s = (char*)*argp++;
        if(s == 0)
          s = "(null)";
        for(; *s; s++)
          cons_putc(*s);
        break;
      case ’%’:
        cons_putc(’%’);
        break;
      default:
        // Print unknown % sequence to draw attention.
        cons_putc(’%’);
        cons_putc(c);
        break;
      }
      state = 0;
      break;
    }
  }

  if(locking)
    release(&console_lock);
}

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

  iunlock(ip);
  acquire(&console_lock);
  for(i = 0; i < n; i++)
    cons_putc(buf[i] & 0xff);
  release(&console_lock);
  ilock(ip);

  return n;
}
```

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

#define C(x)  ((x)-’@’)  // Control-x

void
console_intr(int (*getc)(void))
{
  int c;

  acquire(&input.lock);
  while((c = getc()) >= 0){
    switch(c){
    case C(’P’):  // Process listing.
      procdump();
      break;
    case C(’U’):  // Kill line.
      while(input.e > input.w &&
            input.buf[(input.e-1) % INPUT_BUF] != ’\n’){
        input.e--;
        cons_putc(BACKSPACE);
      }
      break;
    case C(’H’):  // Backspace
      if(input.e > input.w){
        input.e--;
        cons_putc(BACKSPACE);
      }
      break;
    default:
      if(c != 0 && input.e < input.r+INPUT_BUF){
        input.buf[input.e++] = c;
        cons_putc(c);
        if(c == ’\n’ || c == C(’D’) || input.e == input.r+INPUT_BUF){
          input.w = input.e;
          wakeup(&input.r);
        }
      }
      break;
    }
  }
  release(&input.lock);
}
```

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

  iunlock(ip);
  target = n;
  acquire(&input.lock);
  while(n > 0){
    while(input.r == input.w){
      if(cp->killed){
        release(&input.lock);
        ilock(ip);
        return -1;
      }
      sleep(&input.r, &input.lock);
    }
    c = input.buf[input.r++];
    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;
    if(input.r >= INPUT_BUF)
      input.r = 0;
  }
  release(&input.lock);
  ilock(ip);

  return target - n;
}
```

```
void
console_init(void)
{
  initlock(&console_lock, "console");
  initlock(&input.lock, "console input");

  devsw[CONSOLE].write = console_write;
  devsw[CONSOLE].read = console_read;
  //use_console_lock = 1;

  pic_enable(IRQ_KBD);
  ioapic_enable(IRQ_KBD, 0);
}

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

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

```
// 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
timer_init(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);
  pic_enable(IRQ_TIMER);
}
```

```
# Initial process execs /init.

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

# exec(init, argv)
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0
  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
```

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

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

char *sh_args[] = { "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", sh_args);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

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

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

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

```
// 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;
};
```

```c
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;
```

```c
  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;

  // Assumes three file descriptors 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(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;
}
```

```c
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;
}
```

```c
// 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;
}
```