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.

ACKNOWLEDGEMENTS

xv6 is inspired by John Lions' 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/6.828/2006/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, ioapic.h, 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
This version is the very first one, so don't be surprised if there are errors or the code is unclear.

BUIDLING 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/6.828/2006/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 PC

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:

```
    namei 4610
        0333 4610 4709 4758
        4808 4857 4866 5264
        5277 5362 5410 5490
```

indicates that namei is defined on line 4610 and is mentioned on twelve lines on sheets 03, 46, 47, 48, 52, 53, and 54.

```
pipe_alloc 5621
    0293 5176 5621
pipe_close 5661
    0294 5017 5661
pipe_read 5706
    0296 4979 5706
PIPESIZE 5609
    5609 5617 5687 5696
    5724
pipe_write 5680
    0295 4957 5680
pit8253_timerinit 6887
    0255 1283 6887
printint 6473
    6473 6522 6525
proc 1929
    0211 0213 0214 0215
    0244 0245 0288 1203
    1226 1261 1265 1325
    1329 1330 1364 1407
    1521 1547 1548 1549
    1757 1900 1929 1956
    1957 2005 2011 2012
    2028 2061 2076 2079
    2083 2086 2104 2105
    2108 2210 2218 2255
    2270 2305 2344 2346
    2366 2369 2389 2390
    2406 2411 2427 2428
    2436 2473 2476 2562
    2853 2883 2938 3004
    3025 3037 3054 3066
    3123 3204 3220 3261
    3705 3930 4055 4614
    4905 5104 5124 5156
    5172 5403 5477 5604
    5957
procdump 2470
    0225 2470 6790
process0 1327
    1215 1265 1266 1327
proc_exit 2387
    0221 2387 2461 2887
    2891 2913 2940 3231
    5597
proc_kill 2364
    0222 2364 3248
proc_table_lock 2009
    1331 1334 2009 2020
    2111 2113 2118 2215
```

```
    2223 2239 2250 2257
    2272 2275 2283 2284
    2313 2315 2317 2319
    2320 2333 2334 2355
    2357 2368 2375 2379
    2403 2431 2448 2457
    2462
proc_wait 2425
    0223 1359 2425 3238
proghdr 0824
    0824 1368 1377 1382
    5482
read_eflags 0435
    0435
readi 4483
    0335 4483 4723 4814
    4982 5494 5502 5569
    5576
release 1831
    0283 1231 1334 1831
    1834 2113 2118 2223
    2239 2275 2284 2300
    2318 2321 2334 2357
    2375 2379 2448 2457
    2653 2674 2680 2685
    3793 3867 3920 3983
    3992 4047 4206 4221
    4333 4364 4448 4935
    4939 5014 5024 5058
    5673 5689 5700 5714
    5727 6551 6584 6804
    6828
run 2567
    1583 1940 2103 2203
    2214 2567 2568 2571
    2607 2608 2609 2623
    2662 2671 2915
RUNNING 1926
    1926 2228 2916
sched 2253
    2253 2258 2260 2274
    2314 2327 2418
scheduler 2208
    0220 1292 1322 1325
    1964 2025 2200 2201
    2206 2208 2250 2278
    2283 2415
segdesc 0627
    0400 0403 0627 0651
    0654 0659 1966
```

```
SEG_KCODE 1901
    1901 2040 2869 2870
SEG_KDATA 1902
    1902 2032 2041 2760
SEG_NULLASM 0554
    0554 1094 1172
SEG_TSS 1905
    1905 2042 2043 2053
SEG_UCODE 1903
    1350 1903 2045 2048
SEG_UDATA 1904
    1349 1904 2046 2049
setupsegs 2028
    0214 1270 1313 2028
    2226 2236 3267 5585
shift 6740
    6740 6756 6760 6761
    6763 6766 6769 6770
    6772 6773
sleep 2303
    0218 1803 2300 2303
    2308 2311 2324 2372
    2462 3837 3851 3907
    3979 4076 4197 4330
    5693 5717 6815
spinlock 1701
    0216 0218 0279 0280
    0281 0282 0283 0284
    1210 1331 1701 1758
    1763 1805 1831 1852
    2007 2009 2303 2563
    2565 3009 3209 3709
    3735 3932 3936 4057
    4079 4908 4913 5109
    5607 5616 6404 6409
    6739
STA_A 0568 0670
    0568 0670
STA_C 0565 0667
    0565 0667
STA_E 0564 0666
    0564 0666
STA_R 0567 0669
    0567 0669 1095 1173
    2040 2045
stat 3450
    0301 0307 0334 1651
    3001 3201 3450 4051
    4468 4470 4901 5030
    5101 5443
```

```
stati 4470
    0334 4470 5034
STA_W 0566 0668
    0566 0668 1096 1174
    2041 2046
STA_X 0563 0665
    0563 0665 1095 1173
    2040 2045
sti 0485
    0485 0487 1290 1320
    1845
strncmp 5801
    0240 1558 5801
STS_CG16 0676
    0676
STS_CG32 0682
    0682
STS_IG16 0678
    0678
STS_IG32 0683
    0683 0764
STS_LDT 0674
    0674
STS_T16A 0673
    0673
STS_T16B 0675
    0675
STS_T32A 0680
    0680 2042
STS_T32B 0681
    0681
STS_TG 0677
    0677
STS_TG16 0679
    0679
STS_TG32 0684
    0684 0764
superblock 3560
    3560 4106 4113 4155
    4161 4277 4283
syscall 3121
    0242 0243 1207 2857
    2889 3008 3121 3208
    5108
SYS_chdir 2716
    2716 3173
SYS_close 2707
    2707 3146
SYS_dup 2717
    2717 3176
```

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

```c
#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 NREQUEST    100  // outstanding disk requests
#define NBUF         10  // size of disk block cache
#define NINODE      100  // maximum number of active i-nodes
#define NDEV         10  // maximum major device number
```

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

// console.c
void console_init(void);
void cprintf(char*, ...);
void panic(char*);
void kbd_intr(void);

// proc.c
void pinit(void);
struct proc;
void setupsegs(struct proc*);
struct proc* copyproc(struct proc*);
struct spinlock;
int growproc(int);
void sleep(void*, struct spinlock*);
void wakeup(void*);
void scheduler(void);
void proc_exit(void);
int proc_kill(int);
int proc_wait(void);
void yield(void);
void procdump(void);

// setjmp.S
struct jmpbuf;
int setjmp(struct jmpbuf*);
void longjmp(struct jmpbuf*);

// trap.c
void tvinit(void);
void idtinit(void);

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

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

```

```
// picirq.c
void pic_init(void);
void irq_enable(int);

// 8253pit.c
void pit8253_timerinit(void);

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

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

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

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

// main.c
void load_icode(struct proc*, uchar*, uint);

// pipe.c
struct pipe;
struct file;
int pipe_alloc(struct file**, struct file**);
void pipe_close(struct pipe*, int);
int pipe_write(struct pipe*, char*, int);
int pipe_read(struct pipe*, char*, int);



```

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

// ide.c
void ide_init(void);
void ide_intr(void);
void ide_rw(int, uint, void*, uint, int);

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

// fs.c
extern uint rootdev;
void iinit(void);
struct inode* iget(uint, uint);
void ilock(struct inode*);
void iunlock(struct inode*);
void itrunc(struct inode*);
void idecref(struct inode*);
void iincref(struct inode*);
void iput(struct inode*);
struct inode* namei(char*, int, uint*, char**, struct inode**);
void stati(struct inode*, struct stat*);
int readi(struct inode*, char*, uint, uint);
int writei(struct inode*, char*, uint, uint);
struct inode* mknod(char*, short, short, short);
struct inode* mknod1(struct inode*, char*, short, short, short);
int unlink(char*);
void iupdate(struct inode*);
int link(char*, char*);
```

```c
// Special assembly routines to access x86-specific
// hardware instructions.

static __inline uchar
inb(int port)
{
  uchar data;
  __asm __volatile("inb %w1,%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(int port, uchar data)
{
  __asm __volatile("outb %0,%w1" : : "a" (data), "d" (port));
}

static __inline void
outw(int port, ushort data)
{
  __asm __volatile("outw %0,%w1" : : "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)" : : "g" (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)" : : "g" (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;
  __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
```

```
// 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
};
```

```
// 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 }

// 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
```

```
// 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 mpctb {          // 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 mppe {           // processor table entry
  uchar type;                   // entry type (0)
  uchar apicid;                 // local APIC id
  uchar version;                // local APIC verison
  uchar flags;                  // CPU flags
  uchar signature[4];           // CPU signature
  uint feature;                 // feature flags from CPUID instruction
  uchar reserved[8];
};

struct mpbe {           // bus table entry
  uchar type;                   // entry type (1)
  uchar busno;                  // bus id
  char string[6];               // bus type string
};
```

```
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
};

struct mpie {           // interrupt table entry
  uchar type;                   // entry type ([34])
  uchar intr;                   // interrupt type
  ushort flags;                 // interrupt flag
  uchar busno;                  // source bus id
  uchar irq;                    // source bus irq
  uchar apicno;                 // destination APIC id
  uchar intin;                  // destination APIC [L]INTIN#
};

enum {                  // table entry types
  MPPROCESSOR   = 0x00,         // one entry per processor
  MPBUS = 0x01,                 // one entry per bus
  MPIOAPIC = 0x02,              // one entry per I/O APIC
  MPIOINTR = 0x03,              // one entry per bus interrupt source
  MPLINTR = 0x04,               // one entry per system interrupt source

  MPSASM = 0x80,
  MPHIERARCHY   = 0x81,
  MPCBASM = 0x82,

                        // PCMPprocessor and PCMPioapic flags
  MPEN = 0x01,                  // enabled
  MPBP = 0x02,                  // bootstrap processor

                        // PCMPiointr and PCMPlintr flags
  MPPOMASK = 0x03,              // polarity conforms to bus specs
  MPHIGH = 0x01,                // active high
  MPLOW = 0x03,                 // active low
  MPELMASK = 0x0C,              // trigger mode of APIC input signals
  MPEDGE = 0x04,                // edge-triggered
  MPLEVEL = 0x0C,               // level-triggered

                        // PCMPiointr and PCMPlintr interrupt type
  MPINT = 0x00,                 // vectored interrupt from APIC Rdt
  MPNMI = 0x01,                 // non-maskable interrupt
  MPSMI = 0x02,                 // system management interrupt
  MPExtINT = 0x03,              // vectored interrupt from external PIC
};
```

```
// Common bits for
//      I/O APIC Redirection Table Entry;
//      Local APIC Local Interrupt Vector Table;
//      Local APIC Inter-Processor Interrupt;
//      Local APIC Timer Vector Table.
enum {
  APIC_FIXED     = 0x00000000,  // [10:8] Delivery Mode
  APIC_LOWEST    = 0x00000100,  // Lowest priority
  APIC_SMI       = 0x00000200,  // System Management Interrupt
  APIC_RR        = 0x00000300,  // Remote Read
  APIC_NMI       = 0x00000400,
  APIC_INIT      = 0x00000500,  // INIT/RESET
  APIC_STARTUP   = 0x00000600,  // Startup IPI
  APIC_EXTINT    = 0x00000700,

  APIC_PHYSICAL  = 0x00000000,  // [11] Destination Mode (RW)
  APIC_LOGICAL   = 0x00000800,

  APIC_DELIVS    = 0x00001000,  // [12] Delivery Status (RO)
  APIC_HIGH      = 0x00000000,  // [13] Interrupt Input Pin Polarity (RW)
  APIC_LOW       = 0x00002000,
  APIC_REMOTEIRR = 0x00004000,  // [14] Remote IRR (RO)
  APIC_EDGE      = 0x00000000,  // [15] Trigger Mode (RW)
  APIC_LEVEL     = 0x00008000,
  APIC_IMASK     = 0x00010000,  // [16] Interrupt Mask
};
```

```
#include "asm.h"

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

#########################################################################
# ENTRY POINT for the bootstrap processor
#   This code should be stored in the first sector of the hard disk.
#   After the BIOS initializes the hardware on startup or system reset,
#   it loads this code at physical address 0x7c00 - 0x7d00 (512 bytes).
#   Then the BIOS jumps to the beginning of it, address 0x7c00,
#   while running in 16-bit real-mode (8086 compatibility mode).
#   The Code Segment register (CS) is initially zero on entry.
#
# This code switches into 32-bit protected mode so that all of
# memory can accessed, then calls into C.
#########################################################################

.globl start                      # Entry point
start:
.code16                           # This runs in real 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

  # Set up the stack pointer, growing downward from 0x7c00.
  movw    $start,%sp              # Stack Pointer

  # Enable A20:
  #   For fascinating historical reasons (related to the fact that
  #   the earliest 8086-based PCs could only address 1MB of physical
  #   memory and subsequent 80286-based PCs wanted to retain maximum
  #   compatibility), physical address line 20 is tied to low when the
  #   machine boots.  Obviously this a bit of a drag for us, especially
  #   when trying to address memory above 1MB.  This code undoes this.

seta20.1:
  inb     $0x64,%al               # Get status
  testb   $0x2,%al                # Busy?
  jnz     seta20.1                # Yes
  movb    $0xd1,%al               # Command: Write
  outb    %al,$0x64               #  output port


```

```
seta20.2:
  inb     $0x64,%al               # Get status
  testb   $0x2,%al                # Busy?
  jnz     seta20.2                # Yes
  movb    $0xdf,%al               # Enable
  outb    %al,$0x60               #  A20

# Switch from real to protected mode
#  The descriptors in our GDT allow all physical memory to be accessed.
#  Furthermore, the descriptors have base addresses of 0, so that the
#  segment translation is a NOP, ie. virtual addresses are identical to
#  their physical addresses.  With this setup, immediately after
#  enabling protected mode it will still appear to this code
#  that it is running directly on physical memory with no translation.
#  This initial NOP-translation setup is required by the processor
#  to ensure that the transition to protected mode occurs smoothly.
real_to_prot:
  cli                         # Mandatory since we dont set up an IDT
  lgdt    gdtdesc             # load GDT -- mandatory in protected mode
  movl    %cr0, %eax          # turn on protected mode
  orl     $CR0_PE_ON, %eax    #
  movl    %eax, %cr0          #
  ### CPU magic: jump to relocation, flush prefetch queue, and reload %cs
  ### Has the effect of just jmp to the next instruction, but simultaneous
  ### loads CS with $PROT_MODE_CSEG.
  ljmp    $PROT_MODE_CSEG, $protcseg

#### we are in 32-bit protected mode (hence the .code32)
.code32
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
  call cmain                      # finish the boot load from C.
                                  # cmain() should not return
spin:
  jmp spin                        # ..but in case it does, spin

.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.
#
# mp.c causes each non-boot CPU in turn to jump to start.
# mp.c puts the correct %esp in start-4, and the place to jump
# to in start-8.

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

.globl start
start:
  .code16                     # This runs in real 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

  # Set up the stack pointer, growing downward from 0x7000-8.
  movw    $start-8,%sp        # Stack Pointer

  # Switch from real to protected mode
  #  The descriptors in our GDT allow all physical memory to be accessed.
  #  Furthermore, the descriptors have base addresses of 0, so that the
  #  segment translation is a NOP, ie. virtual addresses are identical to
  #  their physical addresses.  With this setup, immediately after
  #  enabling protected mode it will still appear to this code
  #  that it is running directly on physical memory with no translation.
  #  This initial NOP-translation setup is required by the processor
  #  to ensure that the transition to protected mode occurs smoothly.

  lgdt    gdtdesc             # load GDT -- mandatory in protected mode
  movl    %cr0, %eax          # turn on protected mode
  orl     $CR0_PE_ON, %eax    #
  movl    %eax, %cr0          #
```

```
  # CPU magic: jump to relocation, flush prefetch queue, and reload %cs
  # Has the effect of just jmp to the next instruction, but simultaneous
  # loads CS with $PROT_MODE_CSEG.
  ljmp    $PROT_MODE_CSEG, $protcseg

# We are now in 32-bit protected mode (hence the .code32)
.code32
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-8, %eax
  movl    start-4, %esp
  jmp     *%eax

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

extern char edata[], end[];
extern uchar _binary__init_start[], _binary__init_size[];

void process0();

// Bootstrap processor starts running C code here.
// This is called main0 not main so that it can have
// a void return type.  Gcc can't handle functions named
// main that don't return int.  Really.
void
main0(void)
{
  int i;
  int bcpu;
  struct proc *p;

  // 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[0].mpstack + MPSTACK - 32));
  asm volatile("movl %0, %%ebp" : : "r" (cpus[0].mpstack + MPSTACK));

  lapic_init(bcpu);

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

```c
  pinit(); // process table
  binit(); // buffer cache
  pic_init();
  ioapic_init();
  kinit(); // physical memory allocator
  tvinit(); // trap vectors
  idtinit(); // this CPU's interrupt descriptor table
  fileinit();
  iinit(); // i-node table

  // initialize process 0
  p = &proc[0];
  p->state = RUNNABLE;
  p->kstack = kalloc(KSTACKSIZE);

  // cause proc[0] to start in kernel at process0
  p->jmpbuf.eip = (uint) process0;
  p->jmpbuf.esp = (uint) (p->kstack + KSTACKSIZE - 4);

  // make sure there's a TSS
  setupsegs(0);

  // initialize I/O devices, let them enable interrupts
  console_init();
  ide_init();

  // start other CPUs
  mp_startthem();

  // turn on timer
  if(ismp)
    lapic_timerinit();
  else
    pit8253_timerinit();

  // enable interrupts on the local APIC
  lapic_enableintr();

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

  scheduler();
}
```

```
// Additional processors start here.
void
mpmain(void)
{
  cprintf("cpu%d: starting\n", cpu());
  idtinit(); // CPU's idt
  if(cpu() == 0)
    panic("mpmain on cpu 0");
  lapic_init(cpu());
  lapic_timerinit();
  lapic_enableintr();

  // make sure there's a TSS
  setupsegs(0);

  cpuid(0, 0, 0, 0, 0);  // memory barrier
  cpus[cpu()].booted = 1;

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

  scheduler();
}

// proc[0] starts here, called by scheduler() in the ordinary way.
void
process0()
{
  struct proc *p0 = &proc[0];
  struct proc *p1;
  extern struct spinlock proc_table_lock;
  struct trapframe tf;

  release(&proc_table_lock);

  p0->cwd = iget(rootdev, 1);
  iunlock(p0->cwd);

  // dummy user memory to make copyproc() happy.
  // must be big enough to hold the init binary.
  p0->sz = PAGE;
  p0->mem = kalloc(p0->sz);

  // fake a trap frame as if a user process had made a system
  // call, so that copyproc will have a place for the new
  // process to return to.
  p0->tf = &tf;
  memset(p0->tf, 0, sizeof(struct trapframe));
  p0->tf->es = p0->tf->ds = p0->tf->ss = (SEG_UDATA << 3) | 3;
```

```
  p0->tf->cs = (SEG_UCODE << 3) | 3;
  p0->tf->eflags = FL_IF;
  p0->tf->esp = p0->sz;

  p1 = copyproc(p0);

  load_icode(p1, _binary__init_start, (uint) _binary__init_size);
  p1->state = RUNNABLE;

  proc_wait();
  panic("init exited");
}

void
load_icode(struct proc *p, uchar *binary, uint size)
{
  int i;
  struct elfhdr *elf;
  struct proghdr *ph;

  elf = (struct elfhdr*) binary;
  if(elf->magic != ELF_MAGIC)
    panic("load_icode: not an ELF binary");

  p->tf->eip = elf->entry;

  // Map and load segments as directed.
  ph = (struct proghdr*) (binary + elf->phoff);
  for(i = 0; i < elf->phnum; i++, ph++) {
    if(ph->type != ELF_PROG_LOAD)
      continue;
    if(ph->va + ph->memsz < ph->va)
      panic("load_icode: overflow in proghdr");
    if(ph->va + ph->memsz >= p->sz)
      panic("load_icode: icode too large");

    // Load/clear the segment
    memmove(p->mem + ph->va, binary + ph->offset, ph->filesz);
    memset(p->mem + ph->va + ph->filesz, 0, ph->memsz - ph->filesz);
  }
}
```

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

static char *buses[] = {
  "CBUSI ",
  "CBUSII",
  "EISA  ",
  "FUTURE",
  "INTERN",
  "ISA   ",
  "MBI   ",
  "MBII  ",
  "MCA   ",
  "MPI   ",
  "MPSA  ",
  "NUBUS ",
  "PCI   ",
  "PCMCIA",
  "TC    ",
  "VL    ",
  "VME   ",
  "XPRESS",
  0,
};

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

static struct cpu *bcpu;
static struct mp *mp;  // The MP floating point structure

static struct mp*
mp_scan(uchar *addr, int len)
{
  uchar *e, *p, sum;
  int i;

  e = addr+len;
  for(p = addr; p < e; p += sizeof(struct mp)){
    if(memcmp(p, "_MP_", 4))
      continue;
    sum = 0;
```

```c
    for(i = 0; i < sizeof(struct mp); i++)
      sum += p[i];
    if(sum == 0)
      return (struct mp*)p;
  }
  return 0;
}

// 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_scan((uchar*) p, 1024)))
      return mp;
  }
  else{
    p = ((bda[0x14]<<8)|bda[0x13])*1024;
    if((mp = mp_scan((uchar*)p-1024, 1024)))
      return mp;
  }
  return mp_scan((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 int
mp_detect(void)
{
  struct mpctb *pcmp;
  uchar *p, sum;
  uint length;

  if((mp = mp_search()) == 0 || mp->physaddr == 0)
    return 1;
  pcmp = (struct mpctb*) mp->physaddr;
  if(memcmp(pcmp, "PCMP", 4))
    return 2;

  length = pcmp->length;
  sum = 0;
  for(p = (uchar*)pcmp; length; length--)
    sum += *p++;

  if(sum || (pcmp->version != 1 && pcmp->version != 4))
    return 3;

  return 0;
}

void
mp_init(void)
{
  int r;
  uchar *p, *e;
  struct mpctb *mpctb;
  struct mppe *proc;
  struct mpbe *bus;
  struct mpioapic *ioapic;
  struct mpie *intr;
  int i;
  uchar byte;

  ncpu = 0;
  if((r = mp_detect()) != 0) {
    return;
  }

  ismp = 1;

  // Run through the table saving information needed for starting
  // application processors and initialising any I/O APICs. The table
  // is guaranteed to be in order such that only one pass is necessary.

  mpctb = (struct mpctb*) mp->physaddr;
  lapicaddr = (uint*) mpctb->lapicaddr;
  p = ((uchar*)mpctb)+sizeof(struct mpctb);
  e = ((uchar*)mpctb)+mpctb->length;

  while(p < e) {
    switch(*p){
    case MPPROCESSOR:
      proc = (struct mppe*) p;
      cpus[ncpu].apicid = proc->apicid;
      if(proc->flags & MPBP) {
```

```
        bcpu = &cpus[ncpu];
      }
      ncpu++;
      p += sizeof(struct mppe);
      continue;
    case MPBUS:
      bus = (struct mpbe*) p;
      for(i = 0; buses[i]; i++){
        if(strncmp(buses[i], bus->string, sizeof(bus->string)) == 0)
          break;
      }
      p += sizeof(struct mpbe);
      continue;
    case MPIOAPIC:
      ioapic = (struct mpioapic*) p;
      ioapic_id = ioapic->apicno;
      p += sizeof(struct mpioapic);
      continue;
    case MPIOINTR:
      intr = (struct mpie*) p;
      p += sizeof(struct mpie);
      continue;
    default:
      cprintf("mp_init: unknown PCMP type 0x%x (e-p 0x%x)\n", *p, e-p);
      while(p < e){
        cprintf("%uX ", *p);
        p++;
      }
      break;
    }
  }

  if(mp->imcrp) {
    // It appears that Bochs doesn't support IMCR, so code won't run.
    outb(0x22, 0x70);   // Select IMCR
    byte = inb(0x23);   // Current contents
    byte |= 0x01;       // Mask external INTR
    outb(0x23, byte);   // Disconnect 8259s/NMI
  }
}


int
mp_bcpu(void)
{
  if(ismp)
    return bcpu-cpus;
  return 0;
}

```

```
extern void mpmain(void);

// Write bootstrap code to unused memory at 0x7000.
#define APBOOTCODE 0x7000

void
mp_startthem(void)
{
  extern uchar _binary_bootother_start[], _binary_bootother_size[];
  extern int main();
  int c;

  memmove((void*) APBOOTCODE,_binary_bootother_start,
          (uint) _binary_bootother_size);

  for(c = 0; c < ncpu; c++){
    // Our current cpu has already started.
    if(c == cpu())
      continue;

    // Set target %esp
    *(uint*)(APBOOTCODE-4) = (uint) (cpus[c].mpstack) + MPSTACK;

    // Set target %eip
    *(uint*)(APBOOTCODE-8) = (uint)mpmain;

    // Go!
    lapic_startap(cpus[c].apicid, (uint) APBOOTCODE);

    // Wait for cpu to get through bootstrap.
    while(cpus[c].booted == 0)
      ;
  }
}
```

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

// init: The initial user-level program

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

int
main(void)
{
  int pid;

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

  for(;;){
    pid = fork();
    if(pid < 0){
      puts("init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", sh_args);
      puts("init: exec sh failed\n");
      exit();
    } else {
      wait();
    }
  }
}
```

```
// 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.
};
```

```
// Mutual exclusion spin locks.

#include "types.h"
#include "defs.h"
#include "x86.h"
#include "mmu.h"
#include "param.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;
}

// Record the current call stack in pcs[] by following the %ebp chain.
void
getcallerpcs(void *v, uint pcs[])
{
  uint *ebp = (uint*)v - 2;
  int i;
  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;
}
```

```c
// 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);
}

// 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();
}
```

```c
// 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.
// Don't save %eax, because that's the return register.
// The layout of jmpbuf is known to setjmp.S.
struct jmpbuf {
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int esp;
  int ebp;
  int eip;
};

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
  int ppid;                 // Parent pid
  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 jmpbuf jmpbuf;     // Jump here to run process
  struct trapframe *tf;     // Trap frame for current interrupt
};
```

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

extern struct proc proc[];
extern struct proc *curproc[NCPU];  // Current (running) process per CPU

#define MPSTACK 512

// Per-CPU state
struct cpu {
  uchar apicid;               // Local APIC ID
  struct jmpbuf jmpbuf;       // Jump 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 "mmu.h"
#include "x86.h"
#include "param.h"
#include "file.h"
#include "proc.h"
#include "defs.h"
#include "spinlock.h"

struct spinlock proc_table_lock;

struct proc proc[NPROC];
struct proc *curproc[NCPU];
int next_pid = 1;
extern void forkret(void);
extern void forkret1(struct trapframe*);

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

// 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 idling, not running any process.
void
setupsegs(struct proc *p)
{
  struct cpu *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, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c->gdt[SEG_TSS] = SEG16(STS_T32A, (uint) &c->ts, sizeof(c->ts), 0);
  c->gdt[SEG_TSS].s = 0;
  if(p){
    c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, (uint)p->mem, p->sz, 3);
    c->gdt[SEG_UDATA] = SEG(STA_W, (uint)p->mem, p->sz, 3);
  } else {
    c->gdt[SEG_UCODE] = SEG_NULL;
    c->gdt[SEG_UDATA] = SEG_NULL;
  }

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

// Grow current process's memory by n bytes.
// Return old size on success, -1 on failure.
int
growproc(int n)
{
  struct proc *cp = curproc[cpu()];
  char *newmem, *oldmem;

  newmem = kalloc(cp->sz + n);
  if(newmem == 0)
    return 0xffffffff;
  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;
}

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

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

```c
// Create a new process copying p as the parent.
// Does not copy the kernel stack.
// Instead, sets up stack to return as if from system call.
// Caller must arrange for process to run (set state to RUNNABLE).
struct proc*
copyproc(struct proc *p)
{
  int i;
  struct proc *np;

  // Allocate process.
  acquire(&proc_table_lock);
  if((np = allocproc()) == 0){
    release(&proc_table_lock);
    return 0;
  }
  np->pid = next_pid++;
  np->ppid = p->pid;
  release(&proc_table_lock);

  // Copy user memory.
  np->sz = p->sz;
  np->mem = kalloc(np->sz);
  if(np->mem == 0){
    np->state = UNUSED;
    return 0;
  }
  memmove(np->mem, p->mem, np->sz);

  // Allocate kernel stack.
  np->kstack = kalloc(KSTACKSIZE);
  if(np->kstack == 0){
    kfree(np->mem, np->sz);
    np->mem = 0;
    np->state = UNUSED;
    return 0;
  }

  // Copy trapframe registers from parent.
  np->tf = (struct trapframe*)(np->kstack + KSTACKSIZE) - 1;
  memmove(np->tf, p->tf, sizeof(*np->tf));

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

  // Set up new jmpbuf to start executing at forkret (see below).
  memset(&np->jmpbuf, 0, sizeof np->jmpbuf);
  np->jmpbuf.eip = (uint)forkret;
  np->jmpbuf.esp = (uint)np->tf - 4;

```

```c
  // Copy file descriptors
  for(i = 0; i < NOFILE; i++){
    np->ofile[i] = p->ofile[i];
    if(np->ofile[i])
      fileincref(np->ofile[i]);
  }

  np->cwd = p->cwd;
  iincref(p->cwd);

  return np;
}
```

```
// 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 top of 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.

      setupsegs(p);
      curproc[cpu()] = p;
      p->state = RUNNING;
      if(setjmp(&cpus[cpu()].jmpbuf) == 0)
        longjmp(&p->jmpbuf);

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      curproc[cpu()] = 0;

      setupsegs(0);
    }

    release(&proc_table_lock);
  }
}
```

```
// Enter scheduler.  Must already hold proc_table_lock
// and have changed curproc[cpu()]->state.
void
sched(void)
{
  struct proc *p = curproc[cpu()];

  if(!holding(&proc_table_lock))
    panic("sched");
  if(cpus[cpu()].nlock != 1)
    panic("sched locks");

  if(setjmp(&p->jmpbuf) == 0)
    longjmp(&cpus[cpu()].jmpbuf);
}

// Give up the CPU for one scheduling round.
void
yield(void)
{
  struct proc *p = curproc[cpu()];

  acquire(&proc_table_lock);
  p->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(curproc[cpu()]->tf);
}
```

```
// Atomically release lock and sleep on chan.
// Reacquires lock when reawakened.
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = curproc[cpu()];

  if(p == 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.
  p->chan = chan;
  p->state = SLEEPING;
  sched();

  // Tidy up.
  p->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.
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
proc_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
proc_exit(void)
{
  struct proc *p;
  struct proc *cp = curproc[cpu()];
  int fd;

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

  acquire(&proc_table_lock);

  // Wake up our parent.
  for(p = proc; p < &proc[NPROC]; p++)
    if(p->pid == cp->ppid)
      wakeup1(p);

  // Reparent our children to process 1.
  for(p = proc; p < &proc[NPROC]; p++)
    if(p->ppid == cp->pid)
      p->ppid = 1;

  // 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
proc_wait(void)
{
  struct proc *p;
  struct proc *cp = curproc[cpu()];
  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->ppid == cp->pid){
        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->ppid = 0;
          release(&proc_table_lock);
          return pid;
```

```
        }
        havekids = 1;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids){
      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)
{
  int i;
  struct proc *p;

  for(i = 0; i < NPROC; i++) {
    p = &proc[i];
    if(p->state == UNUSED)
      continue;
    cprintf("%d %d %p\n", p->pid, p->state);
  }
}
```

```
#   int  setjmp(struct jmpbuf *jmp);
#   void longjmp(struct jmpbuf *jmp);
#
# Setjmp saves its stack environment in jmp for later use by longjmp.
# It returns 0.
#
# Longjmp restores the environment saved by the last call of setjmp.
# It then causes execution to continue as if the call of setjmp
# had just returned 1.
#
# The caller of setjmp must not itself have returned in the interim.
# All accessible data have values as of the time longjmp was called.
#
#    [Description, but not code, borrowed from Plan 9.]

.globl setjmp
setjmp:
  movl 4(%esp), %eax

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

  movl $0, %eax   # return value
  ret

.globl longjmp
longjmp:
  movl 4(%esp), %eax

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

  addl $4, %esp   # pop and discard %eip
  pushl 28(%eax)  # push new %eip

  movl $1, %eax   # return value (appears to come from setjmp!)
  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 "param.h"
#include "types.h"
#include "defs.h"
#include "param.h"
#include "mmu.h"
#include "proc.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 cp,
// which normally should have been returned by a
// call to kalloc(cp).  (The exception is when
// initializing the allocator; see kinit above.)
void
kfree(char *cp, int len)
{
  struct run **rr;
  struct run *p = (struct run*) cp;
  struct run *pend = (struct run*) (cp + len);
  int i;

  if(len % PAGE)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  for(i = 0; i < len; i++)
    cp[i] = 1;

  acquire(&kalloc_lock);

  rr = &freelist;
  while(*rr){
    struct run *rend = (struct run*) ((char*)(*rr) + (*rr)->len);
    if(p >= *rr && p < rend)
      panic("freeing free page");
    if(pend == *rr){
      p->len = len + (*rr)->len;
      p->next = (*rr)->next;
      *rr = p;
      goto out;
    }
    if(pend < *rr){
      p->len = len;
      p->next = *rr;
      *rr = p;
      goto out;
    }
    if(p == rend){
      (*rr)->len += len;
      if((*rr)->next && (*rr)->next == pend){
        (*rr)->len += (*rr)->next->len;
        (*rr)->next = (*rr)->next->next;
      }
      goto out;
    }
    rr = &((*rr)->next);
  }
  p->len = len;
  p->next = 0;
```

```
  *rr = 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)
{
  struct run **rr;

  if(n % PAGE)
    panic("kalloc");

  acquire(&kalloc_lock);

  rr = &freelist;
  while(*rr){
    struct run *r = *rr;
    if(r->len == n){
      *rr = r->next;
      release(&kalloc_lock);
      return (char*) r;
    }
    if(r->len > n){
      char *p = (char*)r + (r->len - n);
      r->len -= n;
      release(&kalloc_lock);
      return p;
    }
    rr = &(*rr)->next;
  }
  release(&kalloc_lock);
  cprintf("kalloc: out of memory\n");
  return 0;
}
```

```
// 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
```

```
.text
.globl trap
.globl trapret1

.globl alltraps
alltraps:
  /* vectors.S sends all traps here */
  pushl   %ds      # build
  pushl   %es      #  trap
  pushal           #  frame
  movl $16,%eax    # SEG_KDATA << 3
  movw %ax,%ds     #  kernel
  movw %ax,%es     #  segments
  pushl %esp       # pass pointer to this trapframe
  call    trap     #  and call trap()
  addl $4, %esp
  # return falls through to trapret...

  /*
   * a forked process RETs here
   * expects ESP to point to a Trapframe
   */
.globl trapret
trapret:
  popal
  popl %es
  popl %ds
  addl $0x8, %esp /* trapno and errcode */
  iret

.globl forkret1
forkret1:
  movl 4(%esp), %esp
  jmp trapret

.globl  acpu
acpu:
  .long 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      // genernal 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
```

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

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

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], 3);
}

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

void
trap(struct trapframe *tf)
{
  int v = tf->trapno;
  struct proc *cp = curproc[cpu()];

  if(v == T_SYSCALL){
    if(cp->killed)
      proc_exit();
    cp->tf = tf;
    syscall();
    if(cp->killed)
      proc_exit();
    return;
  }
  // Increment nlock to make sure interrupts stay off
  // during interrupt handler.  Decrement before returning.
  cpus[cpu()].nlock++;

  switch(v){
  case IRQ_OFFSET + IRQ_TIMER:
    lapic_timerintr();
    cpus[cpu()].nlock--;
    if(cp){
      // 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((tf->cs&3) == 3 && cp->killed)
        proc_exit();

      // Force process to give up CPU and let others run.
      if(cp->state == RUNNING)
        yield();
    }
    return;

  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);
    break;

  default:
    if(curproc[cpu()]) {
      // Assume process divided by zero or dereferenced null, etc.
      cprintf("pid %d: unhandled trap %d on cpu %d eip %x -- kill proc\n",
              curproc[cpu()]->pid, v, cpu(), tf->eip);
      proc_exit();
    }

    // Otherwise it's our mistake.
    cprintf("unexpected trap %d from cpu %d eip %x\n", v, cpu(), tf->eip);
    panic("trap");
  }

  cpus[cpu()].nlock--;
}
```

```
#!/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";
}
```

```c
#include "types.h"
#include "stat.h"
#include "param.h"
#include "mmu.h"
#include "proc.h"
#include "defs.h"
#include "x86.h"
#include "traps.h"
#include "syscall.h"
#include "spinlock.h"
#include "buf.h"
#include "fs.h"
#include "fsvar.h"
#include "elf.h"
#include "file.h"
#include "fcntl.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 *cp, *ep;

  if(addr >= p->sz)
    return -1;
  *pp = p->mem + addr;
  ep = p->mem + p->sz;
  for(cp = *pp; cp < ep; cp++)
    if(*cp == 0)
      return cp - *pp;
  return -1;
}
```

```c
// Fetch the argno’th word-sized system call argument as an integer.
int
argint(int argno, int *ip)
{
  struct proc *p = curproc[cpu()];

  return fetchint(p, p->tf->esp + 4 + 4*argno, 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 argno, char **pp, int size)
{
  int i;
  struct proc *p = curproc[cpu()];

  if(argint(argno, &i) < 0)
    return -1;
  if((uint)i >= p->sz || (uint)i+size >= p->sz)
    return -1;
  *pp = p->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 argno, char **pp)
{
  int addr;
  if(argint(argno, &addr) < 0)
    return -1;
  return fetchstr(curproc[cpu()], addr, pp);
}
```

```c
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_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);

void
syscall(void)
{
  struct proc *cp = curproc[cpu()];
  int num = cp->tf->eax;
  int ret = -1;

  switch(num){
  case SYS_fork:
    ret = sys_fork();
    break;
  case SYS_exit:
    ret = sys_exit();
    break;
  case SYS_wait:
    ret = sys_wait();
    break;
  case SYS_pipe:
    ret = sys_pipe();
    break;
  case SYS_write:
    ret = sys_write();
    break;
  case SYS_read:
    ret = sys_read();
    break;
  case SYS_close:
    ret = sys_close();
    break;
  case SYS_kill:
    ret = sys_kill();
    break;
  case SYS_exec:
    ret = sys_exec();
    break;
  case SYS_open:
    ret = sys_open();
    break;
  case SYS_mknod:
    ret = sys_mknod();
    break;
  case SYS_unlink:
    ret = sys_unlink();
    break;
  case SYS_fstat:
    ret = sys_fstat();
    break;
  case SYS_link:
    ret = sys_link();
    break;
  case SYS_mkdir:
    ret = sys_mkdir();
    break;
  case SYS_chdir:
    ret = sys_chdir();
    break;
  case SYS_dup:
    ret = sys_dup();
    break;
  case SYS_getpid:
    ret = sys_getpid();
    break;
  case SYS_sbrk:
    ret = sys_sbrk();
    break;
  default:
    cprintf("unknown sys call %d\n", num);
    // Maybe kill the process?
    break;
  }
  cp->tf->eax = ret;
}
```

```
#include "types.h"
#include "stat.h"
#include "param.h"
#include "mmu.h"
#include "proc.h"
#include "defs.h"
#include "x86.h"
#include "traps.h"
#include "syscall.h"
#include "spinlock.h"
#include "buf.h"
#include "fs.h"
#include "fsvar.h"
#include "elf.h"
#include "file.h"
#include "fcntl.h"

int
sys_fork(void)
{
  struct proc *np;

  if((np = copyproc(curproc[cpu()])) == 0)
    return -1;
  np->state = RUNNABLE;
  return np->pid;
}

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

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

int
sys_kill(void)
{
  int pid;

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

```
int
sys_getpid(void)
{
  return curproc[cpu()]->pid;
}

int
sys_sbrk(void)
{
  int addr;
  int n;
  struct proc *cp = curproc[cpu()];

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

```
struct buf {
  int flags;
  uint dev;
  uint sector;
  struct buf *prev;
  struct buf *next;
  uchar data[512];
};
#define B_BUSY 0x1  // buffer is locked by some process
#define B_VALID 0x2 // buffer contains the data of the sector
```

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

extern struct devsw devsw[];

#define CONSOLE 1
```

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

```
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_FILE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

```
// On-disk file system format.
// This header is shared between kernel and user space.

// 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];
};

extern uint rootdev;  // Device number of root file system
```

```
// in-core file system types

struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  int busy;           // Is the inode "locked"?

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NADDRS];
};

extern uint rootdev;

#define NAMEI_LOOKUP 1
#define NAMEI_CREATE 2
#define NAMEI_DELETE 3
```

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

#include "types.h"
#include "param.h"
#include "mmu.h"
#include "proc.h"
#include "defs.h"
#include "x86.h"
#include "traps.h"
#include "spinlock.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 request queue.
// The next request will be stored in request[head],
// and the request currently being served by the disk
// is request[tail].
// Must hold ide_lock while manipulating queue.

struct ide_request {
  int diskno;
  uint secno;
  void *addr;
  uint nsecs;
  uint read;
};

static struct ide_request request[NREQUEST];
static int head, tail;
static struct spinlock ide_lock;

static int disk_1_present;
static int disk_queue;

static int ide_probe_disk1(void);

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

  while(((r = inb(0x1F7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
    ;
```

```
  if(check_error && (r & (IDE_DF|IDE_ERR)) != 0)
    return -1;
  return 0;
}

void
ide_init(void)
{
  initlock(&ide_lock, "ide");
  irq_enable(IRQ_IDE);
  ioapic_enable(IRQ_IDE, ncpu - 1);
  ide_wait_ready(0);
  disk_1_present = ide_probe_disk1();
}

// Probe to see if disk 1 exists (we assume disk 0 exists).
static int
ide_probe_disk1(void)
{
  int r, x;

  // wait for Device 0 to be ready
  ide_wait_ready(0);

  // switch to Device 1
  outb(0x1F6, 0xE0 | (1<<4));

  // check for Device 1 to be ready for a while
  for(x = 0; x < 1000 && (r = inb(0x1F7)) == 0; x++)
    ;

  // switch back to Device 0
  outb(0x1F6, 0xE0 | (0<<4));

  return x < 1000;
}

// Interrupt handler - wake up the request that just finished.
void
ide_intr(void)
{
  acquire(&ide_lock);
  wakeup(&request[tail]);
  release(&ide_lock);
}
```

```
// Start the next request in the queue.
static void
ide_start_request (void)
{
  struct ide_request *r;

  if(head != tail) {
    r = &request[tail];
    ide_wait_ready(0);
    outb(0x3f6, 0);  // generate interrupt
    outb(0x1F2, r->nsecs);
    outb(0x1F3, r->secno & 0xFF);
    outb(0x1F4, (r->secno >> 8) & 0xFF);
    outb(0x1F5, (r->secno >> 16) & 0xFF);
    outb(0x1F6, 0xE0 | ((r->diskno&1)<<4) | ((r->secno>>24)&0x0F));
    if(r->read)
      outb(0x1F7, IDE_CMD_READ);
    else {
      outb(0x1F7, IDE_CMD_WRITE);
      outsl(0x1F0, r->addr, 512/4);
    }
  }
}

// Run an entire disk operation.
void
ide_rw(int diskno, uint secno, void *addr, uint nsecs, int read)
{
  struct ide_request *r;

  if(diskno && !disk_1_present)
    panic("ide disk 1 not present");

  acquire(&ide_lock);

  // Add request to queue.
  while((head + 1) % NREQUEST == tail)
    sleep(&disk_queue, &ide_lock);

  r = &request[head];
  r->secno = secno;
  r->addr = addr;
  r->nsecs = nsecs;
  r->diskno = diskno;
  r->read = read;
  head = (head + 1) % NREQUEST;

  // Start request if necessary.
  ide_start_request();

```

```
  // Wait for request to finish.
  sleep(r, &ide_lock);

  // Finish request.
  if(read){
    if(ide_wait_ready(1) >= 0)
      insl(0x1F0, addr, 512/4);
  }

  // Remove request from queue.
  if((head + 1) % NREQUEST == tail)
    wakeup(&disk_queue);
  tail = (tail + 1) % NREQUEST;

  // Start next request in queue, if any.
  ide_start_request();

  release(&ide_lock);
}

// Synchronous disk write.
int
ide_write(int diskno, uint secno, const void *src, uint nsecs)
{
  int r;

  if(nsecs > 256)
    panic("ide_write");

  ide_wait_ready(0);

  outb(0x1F2, nsecs);
  outb(0x1F3, secno & 0xFF);
  outb(0x1F4, (secno >> 8) & 0xFF);
  outb(0x1F5, (secno >> 16) & 0xFF);
  outb(0x1F6, 0xE0 | ((diskno&1)<<4) | ((secno>>24)&0x0F));
  outb(0x1F7, 0x30);    // CMD 0x30 means write sector

  for(; nsecs > 0; nsecs--, src += 512) {
    if((r = ide_wait_ready(1)) < 0)
      return r;
    outsl(0x1F0, src, 512/4);
  }

  return 0;
}
```

```
// Buffer cache.
//
// The buffer cache is a linked list of buf structures
// holding cached copies of disk block contents.
// Each buf has two state bits B_BUSY and B_VALID.
// If B_BUSY is set, it means that some code is currently
// editing buf, so other code is not allowed to look at it.
// To wait for a buffer that is B_BUSY, sleep on buf.
// (See bget below.)
//
// If B_VALID is set, it means that the memory contents
// have been initialized by reading them off the disk.
// (Conversely, if B_VALID is not set, the memory contents
// of buf must be initialized, often by calling bread,
// before being used.)
//
// After making changes to a buf’s memory, call bwrite to flush
// the changes out to disk, to keep the disk and memory copies
// in sync.
//
// When finished with a buffer, call brelse to release the buffer
// (i.e., clear B_BUSY), so that others can access it.
//
// Bufs that are not B_BUSY are fair game for reuse for other
// disk blocks.  It is not allowed to use a buf after calling brelse.

#include "types.h"
#include "param.h"
#include "x86.h"
#include "mmu.h"
#include "proc.h"
#include "defs.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 block n 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);

  for(;;){
    for(b = bufhead.next; b != &bufhead; b = b->next)
      if((b->flags & (B_BUSY|B_VALID)) &&
         b->dev == dev && b->sector == sector)
        break;

    if(b != &bufhead){
      if(b->flags & B_BUSY){
        sleep(buf, &buf_table_lock);
      } else {
        b->flags |= B_BUSY;
        // b->flags &= ~B_VALID; // Force reread from disk
        release(&buf_table_lock);
        return b;
      }
    } else {
      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");
    }
  }
}
```

```
// Read buf’s contents from disk.
struct buf*
bread(uint dev, uint sector)
{
  struct buf *b;

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

  ide_rw(dev & 0xff, sector, b->data, 1, 1);
  b->flags |= B_VALID;

  return b;
}

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

  ide_rw(b->dev & 0xff, sector, b->data, 1, 0);
  b->flags |= B_VALID;
}

// 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);
}
```

```
#include "types.h"
#include "stat.h"
#include "param.h"
#include "x86.h"
#include "mmu.h"
#include "proc.h"
#include "defs.h"
#include "spinlock.h"
#include "buf.h"
#include "fs.h"
#include "fsvar.h"
#include "dev.h"

// Inode table.  The inode table is an in-memory cache of the
// on-disk inode structures.  If an inode in the table has a non-zero
// reference count, then some open files refer to it and it must stay
// in memory.  If an inode has a zero reference count, it is only in
// memory as a cache in hopes of being used again (avoiding a disk read).
// Any inode with reference count zero can be evicted from the table.
//
// In addition to having a reference count, inodes can be marked busy
// (just like bufs), meaning that some code has logically locked the
// inode, and others are not allowed to look at it.
// This locking can last for a long
// time (for example, if the inode is busy during a disk access),
// so we don’t use spin locks.  Instead, if a process wants to use
// a particular inode, it must sleep(ip) to wait for it to be not busy.
// See iget below.
struct inode inode[NINODE];
struct spinlock inode_table_lock;

uint rootdev = 1;

void
iinit(void)
{
  initlock(&inode_table_lock, "inode_table");
}
```

```c
// Allocate a disk block.
static uint
balloc(uint dev)
{
  int b;
  struct buf *bp;
  struct superblock *sb;
  int bi = 0;
  int size;
  int ninodes;
  uchar m;

  bp = bread(dev, 1);
  sb = (struct superblock*) bp->data;
  size = sb->size;
  ninodes = sb->ninodes;

  for(b = 0; b < size; b++) {
    if(b % BPB == 0) {
      brelse(bp);
      bp = bread(dev, BBLOCK(b, ninodes));
    }
    bi = b % BPB;
    m = 0x1 << (bi % 8);
    if((bp->data[bi/8] & m) == 0) {  // is block free?
      break;
    }
  }
  if(b >= size)
    panic("balloc: out of blocks");

  bp->data[bi/8] |= 0x1 << (bi % 8);
  bwrite(bp, BBLOCK(b, ninodes));  // mark it allocated on disk
  brelse(bp);
  return b;
}
```

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

  bp = bread(dev, 1);
  sb = (struct superblock*) bp->data;
  ninodes = sb->ninodes;
  brelse(bp);

  bp = bread(dev, b);
  memset(bp->data, 0, BSIZE);
  bwrite(bp, b);
  brelse(bp);

  bp = bread(dev, BBLOCK(b, ninodes));
  bi = b % BPB;
  m = ~(0x1 << (bi %8));
  bp->data[bi/8] &= m;
  bwrite(bp, BBLOCK(b, ninodes));  // mark it free on disk
  brelse(bp);
}

// Find the inode with number inum on device dev
// and return an in-memory copy.  Loads the inode
// from disk into the in-core table if necessary.
// The returned inode has busy set and has its ref count incremented.
// Caller must iput the return value when done with it.
struct inode*
iget(uint dev, uint inum)
{
  struct inode *ip, *nip;
  struct dinode *dip;
  struct buf *bp;

  acquire(&inode_table_lock);

 loop:
  nip = 0;
  for(ip = &inode[0]; ip < &inode[NINODE]; ip++){
    if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
      if(ip->busy){
        sleep(ip, &inode_table_lock);
        // Since we droped inode_table_lock, ip might have been reused
        // for some other inode entirely.  Must start the scan over,
        // and hopefully this time we will find the inode we want
        // and it will not be busy.
        goto loop;
      }
      ip->ref++;
      ip->busy = 1;
      release(&inode_table_lock);
      return ip;
    }
    if(nip == 0 && ip->ref == 0)
      nip = ip;
  }

  if(nip == 0)
    panic("out of inodes");

  nip->dev = dev;
  nip->inum = inum;
  nip->ref = 1;
  nip->busy = 1;

  release(&inode_table_lock);

  bp = bread(dev, IBLOCK(inum));
  dip = &((struct dinode*)(bp->data))[inum % IPB];
  nip->type = dip->type;
  nip->major = dip->major;
  nip->minor = dip->minor;
  nip->nlink = dip->nlink;
  nip->size = dip->size;
  memmove(nip->addrs, dip->addrs, sizeof(nip->addrs));
  brelse(bp);

  return nip;
}
```

```
// Copy inode in memory, which has changed, to disk.
// Caller must have locked ip.
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, IBLOCK(ip->inum));   // mark it allocated on the disk
  brelse(bp);
}

// Allocate a new inode with the given type
// from the file system on device dev.
struct inode*
ialloc(uint dev, short type)
{
  struct inode *ip;
  struct dinode *dip = 0;
  struct superblock *sb;
  int ninodes;
  int inum;
  struct buf *bp;

  bp = bread(dev, 1);
  sb = (struct superblock*) bp->data;
  ninodes = sb->ninodes;
  brelse(bp);

  for(inum = 1; inum < 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
      break;
    }
    brelse(bp);
  }

  if(inum >= ninodes)
    panic("ialloc: no inodes left");
```

```c
  memset(dip, 0, sizeof(*dip));
  dip->type = type;
  bwrite(bp, IBLOCK(inum));   // mark it allocated on the disk
  brelse(bp);
  ip = iget(dev, inum);
  return ip;
}

// Free the given inode from its file system.
static void
ifree(struct inode *ip)
{
  ip->type = 0;
  iupdate(ip);
}

// Lock the given inode (wait for it to be not busy,
// and then ip->busy).
// Caller must already hold a reference to ip.
// Otherwise, if all the references to ip go away,
// it might be reused underfoot.
void
ilock(struct inode *ip)
{
  if(ip->ref < 1)
    panic("ilock");

  acquire(&inode_table_lock);

  while(ip->busy)
    sleep(ip, &inode_table_lock);
  ip->busy = 1;

  release(&inode_table_lock);
}
```

```c
// Caller holds reference to ip and has locked it.
// Caller no longer needs to examine / change it.
// Unlock it, but keep the reference.
void
iunlock(struct inode *ip)
{
  if(ip->busy != 1 || ip->ref < 1)
    panic("iunlock");

  acquire(&inode_table_lock);

  ip->busy = 0;
  wakeup(ip);

  release(&inode_table_lock);
}

// Return the disk block address of the nth block in inode ip.
uint
bmap(struct inode *ip, uint bn)
{
  unsigned x;
  uint *a;
  struct buf *inbp;

  if(bn >= MAXFILE)
    panic("bmap 1");
  if(bn < NDIRECT) {
    x = ip->addrs[bn];
    if(x == 0)
      panic("bmap 2");
  } else {
    if(ip->addrs[INDIRECT] == 0)
      panic("bmap 3");
    inbp = bread(ip->dev, ip->addrs[INDIRECT]);
    a = (uint*) inbp->data;
    x = a[bn - NDIRECT];
    brelse(inbp);
    if(x == 0)
      panic("bmap 4");
  }
  return x;
}
```

```
// Truncate the inode ip, discarding all its data blocks.
void
itrunc(struct inode *ip)
{
  int i, j;
  struct buf *inbp;

  for(i = 0; i < NADDRS; i++) {
    if(ip->addrs[i] != 0) {
      if(i == INDIRECT) {
        inbp = bread(ip->dev, ip->addrs[INDIRECT]);
        uint *a = (uint*) inbp->data;
        for(j = 0; j < NINDIRECT; j++) {
          if(a[j] != 0) {
            bfree(ip->dev, a[j]);
            a[j] = 0;
          }
        }
        brelse(inbp);
      }
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0;
    }
  }
  ip->size = 0;
  iupdate(ip);
}

// Caller holds reference to ip and has locked it,
// possibly editing it.
// Release lock and drop the reference.
void
iput(struct inode *ip)
{
  if(ip->ref < 1 || ip->busy != 1)
    panic("iput");

  if((ip->ref == 1) && (ip->nlink == 0)) {
    itrunc(ip);
    ifree(ip);
  }

  acquire(&inode_table_lock);

  ip->ref -= 1;
  ip->busy = 0;
  wakeup(ip);

  release(&inode_table_lock);
}
```

```
// Caller holds reference to ip but not lock.
// Drop reference.
void
idecref(struct inode *ip)
{
  ilock(ip);
  iput(ip);
}

// Increment reference count for ip.
void
iincref(struct inode *ip)
{
  ilock(ip);
  ip->ref++;
  iunlock(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;
}

#define min(a, b) ((a) < (b) ? (a) : (b))

// Read data from inode.
int
readi(struct inode *ip, char *dst, uint off, uint n)
{
  uint target = n, n1;
  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->minor, dst, n);
  }

  while(n > 0 && off < ip->size){
    bp = bread(ip->dev, bmap(ip, off / BSIZE));
    n1 = min(n, ip->size - off);
    n1 = min(n1, BSIZE - (off % BSIZE));
    memmove(dst, bp->data + (off % BSIZE), n1);
    n -= n1;
    off += n1;
    dst += n1;
    brelse(bp);
  }

  return target - n;
}

// Allocate the nth block in inode ip if necessary.
static int
newblock(struct inode *ip, uint lbn)
{
  struct buf *inbp;
  uint *inaddrs;
  uint b;

  if(lbn < NDIRECT) {
    if(ip->addrs[lbn] == 0) {
      b = balloc(ip->dev);
      if(b <= 0)
        return -1;
      ip->addrs[lbn] = b;
    }
  } else {
    if(ip->addrs[INDIRECT] == 0) {
      b = balloc(ip->dev);
      if(b <= 0)
        return -1;
      ip->addrs[INDIRECT] = b;
    }
    inbp = bread(ip->dev, ip->addrs[INDIRECT]);
    inaddrs = (uint*) inbp->data;
    if(inaddrs[lbn - NDIRECT] == 0) {
      b = balloc(ip->dev);
      if(b <= 0)
        return -1;
      inaddrs[lbn - NDIRECT] = b;
      bwrite(inbp, ip->addrs[INDIRECT]);
    }
    brelse(inbp);
  }
  return 0;
}
```

```
// Write data to inode.
int
writei(struct inode *ip, char *addr, uint off, uint n)
{
  if(ip->type == T_DEV) {
    if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
      return -1;
    return devsw[ip->major].write(ip->minor, addr, n);
  } else if(ip->type == T_FILE || ip->type == T_DIR) {
    struct buf *bp;
    int r = 0;
    int m;
    int lbn;
    while(r < n) {
      lbn = off / BSIZE;
      if(lbn >= MAXFILE)
        return r;
      if(newblock(ip, lbn) < 0) {
        cprintf("newblock failed\n");
        return r;
      }
      m = min(BSIZE - off % BSIZE, n-r);
      bp = bread(ip->dev, bmap(ip, lbn));
      memmove(bp->data + off % BSIZE, addr, m);
      bwrite(bp, bmap(ip, lbn));
      brelse(bp);
      r += m;
      off += m;
    }
    if(r > 0) {
      if(off > ip->size) {
        if(ip->type == T_DIR)
          ip->size = ((off / BSIZE) + 1) * BSIZE;
        else
          ip->size = off;
      }
      iupdate(ip);
    }
    return r;
  } else {
    panic("writei: unknown type");
    return 0;
  }
}
```

```
// look up a path name, in one of three modes.
// NAMEI_LOOKUP: return locked target inode.
// NAMEI_CREATE: return locked parent inode.
//   return 0 if name does exist.
//   *ret_last points to last path component (i.e. new file name).
//   *ret_ip points to the the name that did exist, if it did.
//   *ret_ip and *ret_last may be zero even if return value is zero.
// NAMEI_DELETE: return locked parent inode, offset of dirent in *ret_off.
//   return 0 if name doesn't exist.
struct inode*
namei(char *path, int mode, uint *ret_off,
      char **ret_last, struct inode **ret_ip)
{
  struct inode *dp;
  struct proc *p = curproc[cpu()];
  char *cp = path, *cp1;
  uint off, dev;
  struct buf *bp;
  struct dirent *ep;
  int i, l, atend;
  uint ninum;

  if(ret_off)
    *ret_off = 0xffffffff;
  if(ret_last)
    *ret_last = 0;
  if(ret_ip)
    *ret_ip = 0;

  if(*cp == '/')
    dp = iget(rootdev, 1);
  else {
    dp = p->cwd;
    iincref(dp);
    ilock(dp);
  }

  for(;;){
    while(*cp == '/')
      cp++;

    if(*cp == '\0'){
      if(mode == NAMEI_LOOKUP)
        return dp;
      if(mode == NAMEI_CREATE && ret_ip){
        *ret_ip = dp;
        return 0;
      }
      iput(dp);
      return 0;
```

```
    }

    if(dp->type != T_DIR){
      iput(dp);
      return 0;
    }

    for(i = 0; cp[i] != 0 && cp[i] != '/'; i++)
      ;
    l = i;
    if(i > DIRSIZ)
      l = DIRSIZ;

    for(off = 0; off < dp->size; off += BSIZE){
      bp = bread(dp->dev, bmap(dp, off / BSIZE));
      for(ep = (struct dirent*) bp->data;
          ep < (struct dirent*) (bp->data + BSIZE);
          ep++){
        if(ep->inum == 0)
          continue;
        if(memcmp(cp, ep->name, l) == 0 &&
           (l == DIRSIZ || ep->name[l]== 0)){
          // entry matches path element
          off += (uchar*)ep - bp->data;
          ninum = ep->inum;
          brelse(bp);
          cp += i;
          goto found;
        }
      }
      brelse(bp);
    }
    atend = 1;
    for(cp1 = cp; *cp1; cp1++)
      if(*cp1 == '/')
        atend = 0;
    if(mode == NAMEI_CREATE && atend){
      if(*cp == '\0'){
        iput(dp);
        return 0;
      }
      *ret_last = cp;
      return dp;
    }

    iput(dp);
    return 0;
```

```
  found:
    if(mode == NAMEI_DELETE && *cp == '\0'){
      *ret_off = off;
      return dp;
    }
    dev = dp->dev;
    iput(dp);
    dp = iget(dev, ninum);
    if(dp->type == 0 || dp->nlink < 1)
      panic("namei");
  }
}

// Write a new directory entry (name, ino) into the directory dp.
// Caller must have locked dp.
void
wdir(struct inode *dp, char *name, uint ino)
{
  uint off;
  struct dirent de;
  int i;

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

  de.inum = ino;
  for(i = 0; i < DIRSIZ && name[i]; i++)
    de.name[i] = name[i];
  for( ; i < DIRSIZ; i++)
    de.name[i] = '\0';

  if(writei(dp, (char*) &de, off, sizeof(de)) != sizeof(de))
    panic("wdir write");
}
```

```
// Create the path cp and return its locked inode structure.
// If cp already exists, return 0.
struct inode*
mknod(char *cp, short type, short major, short minor)
{
  struct inode *ip, *dp;
  char *last;

  if((dp = namei(cp, NAMEI_CREATE, 0, &last, 0)) == 0)
    return 0;

  ip = mknod1(dp, last, type, major, minor);

  iput(dp);

  return ip;
}

// Create a new inode named name inside dp
// and return its locked inode structure.
// If name already exists, return 0.
struct inode*
mknod1(struct inode *dp, char *name, short type, short major, short minor)
{
  struct inode *ip;

  ip = ialloc(dp->dev, type);
  if(ip == 0)
    return 0;
  ip->major = major;
  ip->minor = minor;
  ip->size = 0;
  ip->nlink = 1;

  iupdate(ip);  // write new inode to disk

  wdir(dp, name, ip->inum);

  return ip;
}
```

```
// Unlink the inode named cp.
int
unlink(char *cp)
{
  struct inode *ip, *dp;
  struct dirent de;
  uint off, inum, dev;

  dp = namei(cp, NAMEI_DELETE, &off, 0, 0);
  if(dp == 0)
    return -1;

  dev = dp->dev;

  if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de) || de.inum == 0)
    panic("unlink no entry");

  inum = de.inum;

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

  iupdate(dp);
  iput(dp);

  ip = iget(dev, inum);

  if(ip->nlink < 1)
    panic("unlink nlink < 1");

  ip->nlink--;

  iupdate(ip);
  iput(ip);

  return 0;
}
```

```
// Create the path new as a link to the same inode as old.
int
link(char *name1, char *name2)
{
  struct inode *ip, *dp;
  char *last;

  if((ip = namei(name1, NAMEI_LOOKUP, 0, 0, 0)) == 0)
    return -1;
  if(ip->type == T_DIR){
    iput(ip);
    return -1;
  }

  iunlock(ip);

  if((dp = namei(name2, NAMEI_CREATE, 0, &last, 0)) == 0) {
    idecref(ip);
    return -1;
  }
  if(dp->dev != ip->dev){
    idecref(ip);
    iput(dp);
    return -1;
  }

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

  wdir(dp, last, ip->inum);
  iput(dp);
  iput(ip);

  return 0;
}
```

```
#include "types.h"
#include "stat.h"
#include "param.h"
#include "x86.h"
#include "mmu.h"
#include "proc.h"
#include "defs.h"
#include "file.h"
#include "spinlock.h"
#include "dev.h"
#include "fs.h"
#include "fsvar.h"

struct spinlock file_table_lock;
struct devsw devsw[NDEV];

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;
}
```

```
// Write to file f.  Addr is kernel address.
int
filewrite(struct file *f, char *addr, int n)
{
  if(f->writable == 0)
    return -1;
  if(f->type == FD_PIPE){
    return pipe_write(f->pipe, addr, n);
  } else if(f->type == FD_FILE) {
    ilock(f->ip);
    int r = writei(f->ip, addr, f->off, n);
    if(r > 0) {
      f->off += r;
    }
    iunlock(f->ip);
    return r;
  } else {
    panic("filewrite");
    return -1;
  }
}

// Read from file f.  Addr is kernel address.
int
fileread(struct file *f, char *addr, int n)
{
  if(f->readable == 0)
    return -1;
  if(f->type == FD_PIPE){
    return pipe_read(f->pipe, addr, n);
  } else if(f->type == FD_FILE){
    ilock(f->ip);
    int cc = readi(f->ip, addr, f->off, n);
    if(cc > 0)
      f->off += cc;
    iunlock(f->ip);
    return cc;
  } else {
    panic("fileread");
    return -1;
  }
}
```

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

  if(f->ref < 1 || f->type == FD_CLOSED)
    panic("fileclose");

  if(--f->ref == 0){
    struct file dummy = *f;

    f->ref = 0;
    f->type = FD_CLOSED;
    release(&file_table_lock);

    if(dummy.type == FD_PIPE){
      pipe_close(dummy.pipe, dummy.writable);
    } else if(dummy.type == FD_FILE){
      idecref(dummy.ip);
    } else {
      panic("fileclose");
    }
  } else {
    release(&file_table_lock);
  }
}

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

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

```c
#include "types.h"
#include "stat.h"
#include "param.h"
#include "mmu.h"
#include "proc.h"
#include "defs.h"
#include "x86.h"
#include "traps.h"
#include "syscall.h"
#include "spinlock.h"
#include "buf.h"
#include "fs.h"
#include "fsvar.h"
#include "elf.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 argno, int *pfd, struct file **pf)
{
  int fd;
  struct file *f;
  struct proc *p = curproc[cpu()];

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

```c
// Allocate a file descriptor for the given file.
// Takes over file reference from caller on success.
static int
fdalloc(struct file *f)
{
  int fd;
  struct proc *p = curproc[cpu()];
  for(fd = 0; fd < NOFILE; fd++){
    if(p->ofile[fd] == 0){
      p->ofile[fd] = f;
      return fd;
    }
  }
  return -1;
}

int
sys_pipe(void)
{
  int *fd;
  struct file *rf = 0, *wf = 0;
  int fd0, fd1;
  struct proc *p = curproc[cpu()];

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

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

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

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

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

  if(omode & O_CREATE){
    dp = namei(path, NAMEI_CREATE, 0, &last, &ip);
    if(dp){
      ip = mknod1(dp, last, T_FILE, 0, 0);
      iput(dp);
      if(ip == 0)
        return -1;
    } else if(ip == 0){
      return -1;
    } else if(ip->type == T_DIR){
      iput(ip);
      return -1;
    }
  } else {
    ip = namei(path, NAMEI_LOOKUP, 0, 0, 0);
    if(ip == 0)
      return -1;
  }
  if(ip->type == T_DIR && ((omode & O_RDWR) || (omode & O_WRONLY))){
    iput(ip);
    return -1;
  }

  if((f = filealloc()) == 0){
    iput(ip);
    return -1;
  }
  if((fd = fdalloc(f)) < 0){
    iput(ip);
    fileclose(f);
    return -1;
  }
  iunlock(ip);
  f->type = FD_FILE;
  if(omode & O_RDWR) {
    f->readable = 1;
    f->writable = 1;
  } else if(omode & O_WRONLY) {
    f->readable = 0;
    f->writable = 1;
  } else {
    f->readable = 1;
    f->writable = 0;
  }
  f->ip = ip;
  f->off = 0;

  return fd;
}

int
sys_mknod(void)
{
  struct inode *nip;
  char *path;
  int len;
  int type, major, minor;

  if((len=argstr(0, &path)) < 0 || argint(1, &type) < 0 ||
     argint(2, &major) < 0 || argint(3, &minor) < 0)
    return -1;

  if(len >= DIRSIZ)
    return -1;

  if((nip = mknod(path, type, major, minor)) == 0)
    return -1;
  iput(nip);
  return 0;
}
```

```
int
sys_mkdir(void)
{
  struct inode *nip;
  struct inode *dp;
  char *path;
  struct dirent de;
  char *last;

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

  dp = namei(path, NAMEI_CREATE, 0, &last, 0);
  if(dp == 0)
    return -1;

  nip = mknod1(dp, last, T_DIR, 0, 0);
  if(nip == 0){
    iput(dp);
    return -1;
  }

  dp->nlink++;
  iupdate(dp);

  memset(de.name, '\0', DIRSIZ);
  de.name[0] = '.';
  de.inum = nip->inum;
  writei(nip, (char*) &de, 0, sizeof(de));

  de.inum = dp->inum;
  de.name[1] = '.';
  writei(nip, (char*) &de, sizeof(de), sizeof(de));

  iput(dp);
  iput(nip);

  return 0;
}
```

```
int
sys_chdir(void)
{
  struct proc *p = curproc[cpu()];
  struct inode *ip;
  char *path;

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

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

  if(ip == p->cwd) {
    iput(ip);
    return 0;
  }

  if(ip->type != T_DIR) {
    iput(ip);
    return -1;
  }

  idecref(p->cwd);
  p->cwd = ip;
  iunlock(p->cwd);
  return 0;
}

int
sys_unlink(void)
{
  char *path;

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

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);
}

```

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

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

int
sys_link(void)
{
  char *old, *new;

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

int
sys_exec(void)
{
  struct proc *cp = curproc[cpu()];
  uint sz=0, ap, sp, p1, p2;
  int i, nargs, argbytes, len;
  struct inode *ip;
  struct elfhdr elf;
  struct proghdr ph;
  char *mem = 0;
  char *path, *s;
  uint argv;

  if(argstr(0, &path) < 0 || argint(1, (int*)&argv) < 0)
    return -1;

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

  if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
    goto bad;

  if(elf.magic != ELF_MAGIC)
    goto bad;

  sz = 0;
  for(i = 0; i < elf.phnum; i++){
    if(readi(ip, (char*)&ph, elf.phoff + i * sizeof(ph),
             sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    sz += ph.memsz;
  }

  sz += 4096 - (sz % 4096);
  sz += 4096;

  mem = kalloc(sz);
  if(mem == 0)
    goto bad;
  memset(mem, 0, sz);

  nargs = 0;
  argbytes = 0;
  for(i = 0;; i++){
    if(fetchint(cp, argv + 4*i, (int*)&ap) < 0)
      goto bad;
    if(ap == 0)
      break;
    len = fetchstr(cp, ap, &s);
    if(len < 0)
      goto bad;
    nargs++;
    argbytes += len + 1;
  }

  // argn\0
  // ...
  // arg0\0
  // 0
  // ptr to argn
  // ...
  // 12: ptr to arg0
  //  8: argv (points to ptr to arg0)
  //  4: argc
  //  0: fake return pc
  sp = sz - argbytes - (nargs+1)*4 - 4 - 4 - 4;
  *(uint*)(mem + sp) = 0xffffffff;
  *(uint*)(mem + sp + 4) = nargs;
  *(uint*)(mem + sp + 8) = (uint)(sp + 12);


```

```
  p1 = sp + 12;
  p2 = sp + 12 + (nargs + 1) * 4;
  for(i = 0; i < nargs; i++){
    fetchint(cp, argv + 4*i, (int*)&ap);
    len = fetchstr(cp, ap, &s);
    memmove(mem + p2, s, len + 1);
    *(uint*)(mem + p1) = p2;
    p1 += 4;
    p2 += len + 1;
  }
  *(uint*)(mem + p1) = 0;

  // commit to the new image.
  kfree(cp->mem, cp->sz);
  cp->sz = sz;
  cp->mem = mem;
  mem = 0;

  for(i = 0; i < elf.phnum; i++){
    if(readi(ip, (char*)&ph, elf.phoff + i * sizeof(ph),
             sizeof(ph)) != sizeof(ph))
      goto bad2;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.va + ph.memsz > sz)
      goto bad2;
    if(readi(ip, cp->mem + ph.va, ph.offset, ph.filesz) != ph.filesz)
      goto bad2;
    memset(cp->mem + ph.va + ph.filesz, 0, ph.memsz - ph.filesz);
  }

  iput(ip);

  cp->tf->eip = elf.entry;
  cp->tf->esp = sp;
  setupsegs(cp);

  return 0;

 bad:
  if(mem)
    kfree(mem, sz);
  iput(ip);
  return -1;

 bad2:
  iput(ip);
  proc_exit();
  return 0;
}
```

```
#include "types.h"
#include "param.h"
#include "x86.h"
#include "mmu.h"
#include "proc.h"
#include "defs.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
pipe_alloc(struct file **f0, struct file **f1)
{
  *f0 = *f1 = 0;
  struct pipe *p = 0;

  if((*f0 = filealloc()) == 0)
    goto oops;
  if((*f1 = filealloc()) == 0)
    goto oops;
  if((p = (struct pipe*) kalloc(PAGE)) == 0)
    goto oops;
  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;
 oops:
  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
pipe_close(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
pipe_write(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){
        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;
  }
```

```c
  release(&p->lock);
  wakeup(&p->readp);
  return i;
}

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

  acquire(&p->lock);

  while(p->readp == p->writep){
    if(p->writeopen == 0){
      release(&p->lock);
      return 0;
    }
    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;
  }

  release(&p->lock);
  wakeup(&p->writep);
  return i;
}
```

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

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

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

  return dst;
}

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

  while(n-- > 0) {
    if(*s1 != *s2)
      return (int) *s1 - (int) *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;
}
```

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

```
#define IO_APIC_BASE   0xFEC00000   // Default phys addr of IO APIC
#define IOAPIC_WINDOW        0x10   // Window register offset

// Constants relating to APIC ID registers
#define APIC_ID_MASK            0xff000000
#define APIC_ID_SHIFT           24
#define APIC_ID_CLUSTER         0xf0
#define APIC_ID_CLUSTER_ID      0x0f
#define APIC_MAX_CLUSTER        0xe
#define APIC_MAX_INTRACLUSTER_ID 3
#define APIC_ID_CLUSTER_SHIFT    4

// Fields in VER
#define APIC_VER_VERSION        0x000000ff
#define APIC_VER_MAXLVT         0x00ff0000
#define MAXLVTSHIFT             16

// Indexes into IO APIC
#define IOAPIC_ID               0x00
#define IOAPIC_VER              0x01
#define IOAPIC_ARB              0x02
#define IOAPIC_REDTBL           0x10
#define IOAPIC_REDTBL0          IOAPIC_REDTBL
#define IOAPIC_REDTBL1          (IOAPIC_REDTBL+0x02)
#define IOAPIC_REDTBL2          (IOAPIC_REDTBL+0x04)
#define IOAPIC_REDTBL3          (IOAPIC_REDTBL+0x06)
#define IOAPIC_REDTBL4          (IOAPIC_REDTBL+0x08)
#define IOAPIC_REDTBL5          (IOAPIC_REDTBL+0x0a)
#define IOAPIC_REDTBL6          (IOAPIC_REDTBL+0x0c)
#define IOAPIC_REDTBL7          (IOAPIC_REDTBL+0x0e)
#define IOAPIC_REDTBL8          (IOAPIC_REDTBL+0x10)
#define IOAPIC_REDTBL9          (IOAPIC_REDTBL+0x12)
#define IOAPIC_REDTBL10         (IOAPIC_REDTBL+0x14)
#define IOAPIC_REDTBL11         (IOAPIC_REDTBL+0x16)
#define IOAPIC_REDTBL12         (IOAPIC_REDTBL+0x18)
#define IOAPIC_REDTBL13         (IOAPIC_REDTBL+0x1a)
#define IOAPIC_REDTBL14         (IOAPIC_REDTBL+0x1c)
#define IOAPIC_REDTBL15         (IOAPIC_REDTBL+0x1e)
#define IOAPIC_REDTBL16         (IOAPIC_REDTBL+0x20)
#define IOAPIC_REDTBL17         (IOAPIC_REDTBL+0x22)
#define IOAPIC_REDTBL18         (IOAPIC_REDTBL+0x24)
#define IOAPIC_REDTBL19         (IOAPIC_REDTBL+0x26)
#define IOAPIC_REDTBL20         (IOAPIC_REDTBL+0x28)
#define IOAPIC_REDTBL21         (IOAPIC_REDTBL+0x2a)
#define IOAPIC_REDTBL22         (IOAPIC_REDTBL+0x2c)
#define IOAPIC_REDTBL23         (IOAPIC_REDTBL+0x2e)

// Fields in the IO APIC's redirection table entries
#define IOART_DEST      APIC_ID_MASK    // broadcast addr: all APICs
```

```
#define IOART_RESV      0x00fe0000      // reserved

#define IOART_INTMASK   0x00010000      // R/W: INTerrupt mask
#define IOART_INTMCLR   0x00000000      //       clear, allow INTs
#define IOART_INTMSET   0x00010000      //       set, inhibit INTs

#define IOART_TRGRMOD   0x00008000      // R/W: trigger mode
#define IOART_TRGREDG   0x00000000      //       edge
#define IOART_TRGRLVL   0x00008000      //       level

#define IOART_REM_IRR   0x00004000      // RO: remote IRR

#define IOART_INTPOL    0x00002000      // R/W: INT input pin polarity
#define IOART_INTAHI    0x00000000      //      active high
#define IOART_INTALO    0x00002000      //      active low

#define IOART_DELIVS    0x00001000      // RO: delivery status

#define IOART_DESTMOD   0x00000800      // R/W: destination mode
#define IOART_DESTPHY   0x00000000      //      physical
#define IOART_DESTLOG   0x00000800      //      logical

#define IOART_DELMOD    0x00000700      // R/W: delivery mode
#define IOART_DELFIXED  0x00000000      //       fixed
#define IOART_DELLOPRI  0x00000100      //       lowest priority
#define IOART_DELSMI    0x00000200      //       System Management INT
#define IOART_DELRSV1   0x00000300      //       reserved
#define IOART_DELNMI    0x00000400      //       NMI signal
#define IOART_DELINIT   0x00000500      //       INIT signal
#define IOART_DELRSV2   0x00000600      //       reserved
#define IOART_DELEXINT  0x00000700      //       External INTerrupt

#define IOART_INTVEC    0x000000ff      // R/W: INTerrupt vector field

// Fields in VER
#define IOART_VER_VERSION       0x000000ff
#define IOART_VER_MAXREDIR      0x00ff0000
#define MAXREDIRSHIFT           16
```

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

enum {  // Local APIC registers
  LAPIC_ID  = 0x0020,   // ID
  LAPIC_VER = 0x0030,   // Version
  LAPIC_TPR = 0x0080,   // Task Priority
  LAPIC_APR = 0x0090,   // Arbitration Priority
  LAPIC_PPR = 0x00A0,   // Processor Priority
  LAPIC_EOI = 0x00B0,   // EOI
  LAPIC_LDR = 0x00D0,   // Logical Destination
  LAPIC_DFR = 0x00E0,   // Destination Format
  LAPIC_SVR = 0x00F0,   // Spurious Interrupt Vector
  LAPIC_ISR = 0x0100,   // Interrupt Status (8 registers)
  LAPIC_TMR = 0x0180,   // Trigger Mode (8 registers)
  LAPIC_IRR = 0x0200,   // Interrupt Request (8 registers)
  LAPIC_ESR = 0x0280,   // Error Status
  LAPIC_ICRLO = 0x0300, // Interrupt Command
  LAPIC_ICRHI = 0x0310, // Interrupt Command [63:32]
  LAPIC_TIMER = 0x0320, // Local Vector Table 0 (TIMER)
  LAPIC_PCINT = 0x0340, // Performance Counter LVT
  LAPIC_LINT0 = 0x0350, // Local Vector Table 1 (LINT0)
  LAPIC_LINT1 = 0x0360, // Local Vector Table 2 (LINT1)
  LAPIC_ERROR = 0x0370, // Local Vector Table 3 (ERROR)
  LAPIC_TICR = 0x0380,  // Timer Initial Count
  LAPIC_TCCR = 0x0390,  // Timer Current Count
  LAPIC_TDCR = 0x03E0,  // Timer Divide Configuration
};

enum {  // LAPIC_SVR
  LAPIC_ENABLE  = 0x00000100,   // Unit Enable
  LAPIC_FOCUS   = 0x00000200,   // Focus Processor Checking Disable
};
```

```c
enum {  // LAPIC_ICRLO
  // [14] IPI Trigger Mode Level (RW)
  LAPIC_DEASSERT = 0x00000000,  // Deassert level-sensitive interrupt
  LAPIC_ASSERT  = 0x00004000,   // Assert level-sensitive interrupt

  // [17:16] Remote Read Status
  LAPIC_INVALID = 0x00000000,   // Invalid
  LAPIC_WAIT    = 0x00010000,   // In-Progress
  LAPIC_VALID   = 0x00020000,   // Valid

  // [19:18] Destination Shorthand
  LAPIC_FIELD   = 0x00000000,   // No shorthand
  LAPIC_SELF    = 0x00040000,   // Self is single destination
  LAPIC_ALLINC  = 0x00080000,   // All including self
  LAPIC_ALLEXC  = 0x000C0000,   // All Excluding self
};

enum {  // LAPIC_ESR
  LAPIC_SENDCS  = 0x00000001,     // Send CS Error
  LAPIC_RCVCS   = 0x00000002,     // Receive CS Error
  LAPIC_SENDACCEPT = 0x00000004,  // Send Accept Error
  LAPIC_RCVACCEPT = 0x00000008,   // Receive Accept Error
  LAPIC_SENDVECTOR = 0x00000020,  // Send Illegal Vector
  LAPIC_RCVVECTOR = 0x00000040,   // Receive Illegal Vector
  LAPIC_REGISTER = 0x00000080,    // Illegal Register Address
};

enum {  // LAPIC_TIMER
  // [17] Timer Mode (RW)
  LAPIC_ONESHOT = 0x00000000,   // One-shot
  LAPIC_PERIODIC = 0x00020000,  // Periodic

  // [19:18] Timer Base (RW)
  LAPIC_CLKIN   = 0x00000000,   // use CLKIN as input
  LAPIC_TMBASE  = 0x00040000,   // use TMBASE
  LAPIC_DIVIDER = 0x00080000,   // use output of the divider
};

enum {  // LAPIC_TDCR
  LAPIC_X2 = 0x00000000,        // divide by 2
  LAPIC_X4 = 0x00000001,        // divide by 4
  LAPIC_X8 = 0x00000002,        // divide by 8
  LAPIC_X16 = 0x00000003,       // divide by 16
  LAPIC_X32 = 0x00000008,       // divide by 32
  LAPIC_X64 = 0x00000009,       // divide by 64
  LAPIC_X128 = 0x0000000A,      // divide by 128
  LAPIC_X1 = 0x0000000B,        // divide by 1
};
```

```c
uint *lapicaddr;

static int
lapic_read(int r)
{
  return *(lapicaddr+(r/sizeof(*lapicaddr)));
}

static void
lapic_write(int r, int data)
{
  *(lapicaddr+(r/sizeof(*lapicaddr))) = data;
}


void
lapic_timerinit(void)
{
  if(!lapicaddr)
    return;

  lapic_write(LAPIC_TDCR, LAPIC_X1);
  lapic_write(LAPIC_TIMER, LAPIC_CLKIN | LAPIC_PERIODIC |
         (IRQ_OFFSET + IRQ_TIMER));
  lapic_write(LAPIC_TCCR, 10000000);
  lapic_write(LAPIC_TICR, 10000000);
}

void
lapic_timerintr(void)
{
  if(lapicaddr)
    lapic_write(LAPIC_EOI, 0);
}

void
lapic_init(int c)
{
  uint r, lvt;

  if(!lapicaddr)
    return;

  lapic_write(LAPIC_DFR, 0xFFFFFFFF);    // Set dst format register
  r = (lapic_read(LAPIC_ID)>>24) & 0xFF; // Read APIC ID
  lapic_write(LAPIC_LDR, (1<<r)<<24);    // Set logical dst register to r
  lapic_write(LAPIC_TPR, 0xFF);          // No interrupts for now

  // Enable APIC
  lapic_write(LAPIC_SVR, LAPIC_ENABLE|(IRQ_OFFSET+IRQ_SPURIOUS));
  // In virtual wire mode, set up the LINT0 and LINT1 as follows:
  lapic_write(LAPIC_LINT0, APIC_IMASK | APIC_EXTINT);
  lapic_write(LAPIC_LINT1, APIC_IMASK | APIC_NMI);

  lapic_write(LAPIC_EOI, 0); // Ack any outstanding interrupts.

  lvt = (lapic_read(LAPIC_VER)>>16) & 0xFF;
  if(lvt >= 4)
    lapic_write(LAPIC_PCINT, APIC_IMASK);
  lapic_write(LAPIC_ERROR, IRQ_OFFSET+IRQ_ERROR);
  lapic_write(LAPIC_ESR, 0);
  lapic_read(LAPIC_ESR);

  // Issue an INIT Level De-Assert to synchronise arbitration ID's.
  lapic_write(LAPIC_ICRHI, 0);
  lapic_write(LAPIC_ICRLO, LAPIC_ALLINC|APIC_LEVEL|
                           LAPIC_DEASSERT|APIC_INIT);
  while(lapic_read(LAPIC_ICRLO) & APIC_DELIVS)
    ;
}

void
lapic_enableintr(void)
{
  if(lapicaddr)
    lapic_write(LAPIC_TPR, 0);
}

void
lapic_disableintr(void)
{
  if(lapicaddr)
    lapic_write(LAPIC_TPR, 0xFF);
}

void
lapic_eoi(void)
{
  if(lapicaddr)
    lapic_write(LAPIC_EOI, 0);
}
```

```c
int
cpu(void)
{
  int x;
  if(lapicaddr)
    x = (lapic_read(LAPIC_ID)>>24) & 0xFF;
  else
    x = 0;
  return x;
}

void
lapic_startap(uchar apicid, int v)
{
  int crhi, i;
  volatile int j = 0;

  crhi = apicid<<24;
  lapic_write(LAPIC_ICRHI, crhi);
  lapic_write(LAPIC_ICRLO, LAPIC_FIELD|APIC_LEVEL|
                           LAPIC_ASSERT|APIC_INIT);

  while(j++ < 10000) {;}
  lapic_write(LAPIC_ICRLO, LAPIC_FIELD|APIC_LEVEL|
                           LAPIC_DEASSERT|APIC_INIT);

  while(j++ < 1000000) {;}

  // in p9 code, this was i < 2, which is what the spec says on page B-3
  for(i = 0; i < 1; i++){
    lapic_write(LAPIC_ICRHI, crhi);
    lapic_write(LAPIC_ICRLO, LAPIC_FIELD|APIC_EDGE|APIC_STARTUP|(v/4096));
    while(j++ < 100000) {;}
  }
}
```

```
#include "types.h"
#include "mp.h"
#include "defs.h"
#include "x86.h"
#include "traps.h"
#include "ioapic.h"

struct ioapic {
  uint ioregsel;  uint p01; uint p02; uint p03;
  uint iowin;     uint p11; uint p12; uint p13;
};


#define IOAPIC_REDTBL_LO(i)  (IOAPIC_REDTBL + (i) * 2)
#define IOAPIC_REDTBL_HI(i)  (IOAPIC_REDTBL_LO(i) + 1)

static uint
ioapic_read(struct ioapic *io, int reg)
{
  io->ioregsel = reg;
  return io->iowin;
}

static void
ioapic_write(struct ioapic *io, int reg, uint val)
{
  io->ioregsel = reg;
  io->iowin = val;
}

void
ioapic_init(void)
{
  struct ioapic *io;
  uint l, h;
  int nintr;
  uchar id;
  int i;

  if(!ismp)
    return;

  io = (struct ioapic*) IO_APIC_BASE;
  l = ioapic_read(io, IOAPIC_VER);
  nintr =  ((l & IOART_VER_MAXREDIR) >> MAXREDIRSHIFT) + 1;
  id = ioapic_read(io, IOAPIC_ID) >> APIC_ID_SHIFT;
  if(id != ioapic_id)
    cprintf("ioapic_init: id isn't equal to ioapic_id; not a MP\n");
  for(i = 0; i < nintr; i++) {
    // active-hi and edge-triggered for ISA interrupts
```

```
    // Assume that pin 0 on the first I/O APIC is an ExtINT pin.
    // Assume that pins 1-15 are ISA interrupts
    l = ioapic_read(io, IOAPIC_REDTBL_LO(i));
    l = l & ~IOART_INTMASK;  // allow INTs
    l |= IOART_INTMSET;
    l = l & ~IOART_INTPOL;   // active hi
    l = l & ~IOART_TRGRMOD;  // edgee triggered
    l = l & ~IOART_DELMOD;   // fixed
    l = l & ~IOART_DESTMOD;  // physical mode
    l = l | (IRQ_OFFSET + i); // vector
    ioapic_write(io, IOAPIC_REDTBL_LO(i), l);
    h = ioapic_read(io, IOAPIC_REDTBL_HI(i));
    h &= ~IOART_DEST;
    ioapic_write(io, IOAPIC_REDTBL_HI(i), h);
  }
}

void
ioapic_enable (int irq, int cpunum)
{
  uint l, h;
  struct ioapic *io;

  if(!ismp)
    return;

  io = (struct ioapic*) IO_APIC_BASE;
  l = ioapic_read(io, IOAPIC_REDTBL_LO(irq));
  l = l & ~IOART_INTMASK;  // allow INTs
  ioapic_write(io, IOAPIC_REDTBL_LO(irq), l);
  h = ioapic_read(io, IOAPIC_REDTBL_HI(irq));
  h &= ~IOART_DEST;
  h |= (cpunum << APIC_ID_SHIFT);
  ioapic_write(io, IOAPIC_REDTBL_HI(irq), h);
}
```

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

// I/O Addresses of the two 8259A 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 irq_mask_8259A = 0xFFFF & ~(1<<IRQ_SLAVE);

static void
irq_setmask_8259A(ushort mask)
{
  irq_mask_8259A = mask;

  outb(IO_PIC1+1, (char)mask);
  outb(IO_PIC2+1, (char)(mask >> 8));
}

void
irq_enable(int irq)
{
  irq_setmask_8259A(irq_mask_8259A & ~(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(irq_mask_8259A != 0xFFFF)
    irq_setmask_8259A(irq_mask_8259A);
}
```

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

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(0x378+1) & 0x80) && i < 12800; i++)
    ;
  outb(0x378+0, c);
  outb(0x378+2, 0x08|0x04|0x01);
  outb(0x378+2, 0x08);
}

static void
cons_putc(int c)
{
  int crtport = 0x3d4; // io port of CGA
  ushort *crt = (ushort*) 0xB8000; // base of CGA memory
  int ind;

  if(panicked){
    cli();
    for(;;)
      ;
  }

  lpt_putc(c);

  // cursor position, 16 bits, col + 80*row
  outb(crtport, 14);
  ind = inb(crtport + 1) << 8;
  outb(crtport, 15);
  ind |= inb(crtport + 1);

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

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

  outb(crtport, 14);
  outb(crtport + 1, ind >> 8);
  outb(crtport, 15);
  outb(crtport + 1, ind);
}

void
printint(int xx, int base, int sgn)
{
  char buf[16];
  char digits[] = "0123456789ABCDEF";
  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, state = 0, c, locking = 0;
  uint *ap = (uint*)(void*)&fmt + 1;

  if(use_console_lock){
    locking = 1;
    acquire(&console_lock);
  }

  for(i = 0; fmt[i]; i++){
    c = fmt[i] & 0xff;
    if(state == 0){
      if(c == ’%’){
        state = ’%’;
      } else {
        cons_putc(c);
      }
    } else if(state == ’%’){
      if(c == ’d’){
        printint(*ap, 10, 1);
        ap++;
      } else if(c == ’x’ || c == ’p’){
        printint(*ap, 16, 0);
        ap++;
      } else if(c == ’s’){
        char *s = (char*)*ap;
        ap++;
        if(s == 0){
          cons_putc(’0’);
        }else{
          while(*s != 0){
            cons_putc(*s);
            s++;
          }
        }
      } else if(c == ’%’){
        cons_putc(c);
      } else {
        // Unknown % sequence.  Print it to draw attention.
        cons_putc(’%’);
        cons_putc(c);
      }
      state = 0;
    }
  }


```

```
  if(locking)
    release(&console_lock);
}

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(;;)
    ;
}

int
console_write(int minor, char *buf, int n)
{
  int i;

  acquire(&console_lock);

  for(i = 0; i < n; i++) {
    cons_putc(buf[i] & 0xff);
  }

  release(&console_lock);

  return n;
}
```

```
#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

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
  [0x97] KEY_HOME,
  [0x9C] '\n',      // KP_Enter
  [0xB5] '/',       // KP_Div
  [0xC8] KEY_UP,
  [0xC9] KEY_PGUP,
  [0xCB] KEY_LF,
  [0xCD] KEY_RT,
  [0xCF] KEY_END,
  [0xD0] KEY_DN,
  [0xD1] KEY_PGDN,
  [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
  [0x97] KEY_HOME,
  [0x9C] '\n',      // KP_Enter
  [0xB5] '/',       // KP_Div
  [0xC8] KEY_UP,
  [0xC9] KEY_PGUP,
  [0xCB] KEY_LF,
  [0xCD] KEY_RT,
  [0xCF] KEY_END,
  [0xD0] KEY_DN,
  [0xD1] KEY_PGDN,
  [0xD2] KEY_INS,
  [0xD3] KEY_DEL
};

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

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,
  [0x97] KEY_HOME,
  [0xB5] C('/'),    // KP_Div
  [0xC8] KEY_UP,
  [0xC9] KEY_PGUP,
  [0xCB] KEY_LF,
  [0xCD] KEY_RT,
  [0xCF] KEY_END,
  [0xD0] KEY_DN,
  [0xD1] KEY_PGDN,
  [0xD2] KEY_INS,
  [0xD3] KEY_DEL
};

static uchar *charcode[4] = {
  normalmap,
  shiftmap,
  ctlmap,
  ctlmap
};

#define KBD_BUF 64
char kbd_buf[KBD_BUF];
int kbd_r;
int kbd_w;
struct spinlock kbd_lock;
static uint shift;

void
kbd_intr()
{
  uint st, data, c;

  acquire(&kbd_lock);
```

```
  st = inb(KBSTATP);
  if((st & KBS_DIB) == 0)
    goto out;
  data = inb(KBDATAP);

  if(data == 0xE0) {
    shift |= E0ESC;
    goto out;
  } else if(data & 0x80) {
    // Key released
    data = (shift & E0ESC ? data : data & 0x7F);
    shift &= ~(shiftcode[data] | E0ESC);
    goto out;
  } 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';
  }

  switch(c){
  case 0:
    // Ignore unknown keystrokes.
    break;

  case C('T'):
    cprintf("#");  // Let user know we're still alive.
    break;

  case C('P'):
    procdump();
    break;

  default:
    if(((kbd_w + 1) % KBD_BUF) != kbd_r){
      kbd_buf[kbd_w++] = c;
      if(kbd_w >= KBD_BUF)
        kbd_w = 0;
      wakeup(&kbd_r);
    }
    break;
  }

out:
  release(&kbd_lock);
}

int
console_read(int minor, char *dst, int n)
{
  uint target = n;

  acquire(&kbd_lock);

  while(kbd_w == kbd_r) {
    sleep(&kbd_r, &kbd_lock);
  }

  while(n > 0 && kbd_w != kbd_r){
    *dst = (kbd_buf[kbd_r]) & 0xff;
    cons_putc(*dst & 0xff);
    dst++;
    --n;
    kbd_r++;
    if(kbd_r >= KBD_BUF)
      kbd_r = 0;
  }

  release(&kbd_lock);

  return target - n;
}

void
console_init()
{
  initlock(&console_lock, "console");
  initlock(&kbd_lock, "kbd");

  devsw[CONSOLE].write = console_write;
  devsw[CONSOLE].read = console_read;

  irq_enable(IRQ_KBD);
  ioapic_enable(IRQ_KBD, 0);

  use_console_lock = 1;
}
```

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

// Register definitions for the Intel
// 8253/8254/82C54 Programmable Interval Timer (PIT).

#define IO_TIMER1       0x040           // 8253 Timer #1
#define IO_TIMER2       0x048           // 8253 Timer #2 (EISA only)

// 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_CNTR0     (IO_TIMER1 + 0) // timer 0 counter port
#define TIMER_CNTR1     (IO_TIMER1 + 1) // timer 1 counter port
#define TIMER_CNTR2     (IO_TIMER1 + 2) // timer 2 counter port
#define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
#define TIMER_SEL0      0x00    // select counter 0
#define TIMER_SEL1      0x40    // select counter 1
#define TIMER_SEL2      0x80    // select counter 2
#define TIMER_INTTC     0x00    // mode 0, intr on terminal cnt
#define TIMER_ONESHOT   0x02    // mode 1, one shot
#define TIMER_RATEGEN   0x04    // mode 2, rate generator
#define TIMER_SQWAVE    0x06    // mode 3, square wave
#define TIMER_SWSTROBE  0x08    // mode 4, s/w triggered strobe
#define TIMER_HWSTROBE  0x0a    // mode 5, h/w triggered strobe
#define TIMER_LATCH     0x00    // latch counter for reading
#define TIMER_LSB       0x10    // r/w counter LSB
#define TIMER_MSB       0x20    // r/w counter MSB
#define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
#define TIMER_BCD       0x01    // count in BCD

void
pit8253_timerinit(void)
{
  // initialize 8253 clock to 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);
  irq_enable(IRQ_TIMER);
}
```