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
({kaashoek,rtm}@csail.mit.edu). 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.

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.

```
0100 typedef unsigned int uint;
0101 typedef unsigned short ushort;
0102 typedef unsigned char uchar;
0103
0104
0105
0106
0107
0108
0109
0110
0111
0112
0113
0114
0115
0116
0117
0118
0119
0120
0121
0122
0123
0124
0125
0126
0127
0128
0129
0130
0131
0132
0133
0134
0135
0136
0137
0138
0139
0140
0141
0142
0143
0144
0145
0146
0147
0148
0149
```



```
0150 #define NPROC        64  // maximum number of processes
0151 #define PAGE       4096  // granularity of user-space memory allocation
0152 #define KSTACKSIZE PAGE  // size of per-process kernel stack
0153 #define NCPU          8  // maximum number of CPUs
0154 #define NOFILE       16  // open files per process
0155 #define NFILE       100  // open files per system
0156 #define NREQUEST    100  // outstanding disk requests
0157 #define NBUF         10  // size of disk block cache
0158 #define NINODE      100  // maximum number of active i-nodes
0159 #define NDEV         10  // maximum major device number
0160
0161
0162
0163
0164
0165
0166
0167
0168
0169
0170
0171
0172
0173
0174
0175
0176
0177
0178
0179
0180
0181
0182
0183
0184
0185
0186
0187
0188
0189
0190
0191
0192
0193
0194
0195
0196
0197
0198
0199
```

```
0200 // kalloc.c
0201 char* kalloc(int);
0202 void kfree(char*, int);
0203 void kinit(void);
0204
0205 // console.c
0206 void console_init(void);
0207 void cprintf(char*, ...);
0208 void panic(char*);
0209 void kbd_intr(void);
0210
0211 // proc.c
0212 void pinit(void);
0213 struct proc;
0214 void setupsegs(struct proc*);
0215 struct proc* copyproc(struct proc*);
0216 struct spinlock;
0217 int growproc(int);
0218 void sleep(void*, struct spinlock*);
0219 void wakeup(void*);
0220 void scheduler(void);
0221 void proc_exit(void);
0222 int proc_kill(int);
0223 int proc_wait(void);
0224 void yield(void);
0225 void procdump(void);
0226
0227 // setjmp.S
0228 struct jmpbuf;
0229 int setjmp(struct jmpbuf*);
0230 void longjmp(struct jmpbuf*);
0231
0232 // trap.c
0233 void tvinit(void);
0234 void idtinit(void);
0235
0236 // string.c
0237 void* memset(void*, int, uint);
0238 int memcmp(const void*, const void*, uint);
0239 void* memmove(void*, const void*, uint);
0240 int strncmp(const char*, const char*, uint);
0241
0242 // syscall.c
0243 void syscall(void);
0244 int fetchint(struct proc*, uint, int*);
0245 int fetchstr(struct proc*, uint, char**);
0246 int argint(int, int*);
0247 int argptr(int, char**, int);
0248 int argstr(int, char**);
0249
```



```
0250 // picirq.c
0251 void pic_init(void);
0252 void irq_enable(int);
0253
0254 // 8253pit.c
0255 void pit8253_timerinit(void);
0256
0257 // mp.c
0258 extern int ismp;
0259 void mp_init(void);
0260 void mp_startthem(void);
0261 int mp_bcpu(void);
0262
0263 // lapic.c
0264 extern uint *lapicaddr;
0265 void lapic_init(int);
0266 void lapic_startap(uchar, int);
0267 void lapic_timerinit(void);
0268 void lapic_timerintr(void);
0269 void lapic_enableintr(void);
0270 void lapic_disableintr(void);
0271 void lapic_eoi(void);
0272 int cpu(void);
0273
0274 // ioapic.c
0275 extern uchar ioapic_id;
0276 void ioapic_init(void);
0277 void ioapic_enable(int irq, int cpu);
0278
0279 // spinlock.c
0280 struct spinlock;
0281 void initlock(struct spinlock*, char*);
0282 void acquire(struct spinlock*);
0283 void release(struct spinlock*);
0284 int holding(struct spinlock*);
0285 void getcallerpcs(void*, uint*);
0286
0287 // main.c
0288 void load_icode(struct proc*, uchar*, uint);
0289
0290 // pipe.c
0291 struct pipe;
0292 struct file;
0293 int pipe_alloc(struct file**, struct file**);
0294 void pipe_close(struct pipe*, int);
0295 int pipe_write(struct pipe*, char*, int);
0296 int pipe_read(struct pipe*, char*, int);
0297
0298
0299
```

```
0300 // file.c
0301 struct stat;
0302 void fileinit(void);
0303 struct file* filealloc(void);
0304 void fileclose(struct file*);
0305 int fileread(struct file*, char*, int n);
0306 int filewrite(struct file*, char*, int n);
0307 int filestat(struct file*, struct stat*);
0308 void fileincref(struct file*);
0309
0310 // ide.c
0311 void ide_init(void);
0312 void ide_intr(void);
0313 void ide_rw(int, uint, void*, uint, int);
0314
0315 // bio.c
0316 void binit(void);
0317 struct buf;
0318 struct buf* getblk(uint dev, uint sector);
0319 struct buf* bread(uint, uint);
0320 void bwrite(struct buf*, uint);
0321 void brelse(struct buf*);
0322
0323 // fs.c
0324 extern uint rootdev;
0325 void iinit(void);
0326 struct inode* iget(uint, uint);
0327 void ilock(struct inode*);
0328 void iunlock(struct inode*);
0329 void itrunc(struct inode*);
0330 void idecref(struct inode*);
0331 void iincref(struct inode*);
0332 void iput(struct inode*);
0333 struct inode* namei(char*, int, uint*, char**, struct inode**);
0334 void stati(struct inode*, struct stat*);
0335 int readi(struct inode*, char*, uint, uint);
0336 int writei(struct inode*, char*, uint, uint);
0337 struct inode* mknod(char*, short, short, short);
0338 struct inode* mknod1(struct inode*, char*, short, short, short);
0339 int unlink(char*);
0340 void iupdate(struct inode*);
0341 int link(char*, char*);
0342
0343
0344
0345
0346
0347
0348
0349
```



```
0350 // Special assembly routines to access x86-specific
0351 // hardware instructions.
0352
0353 static __inline uchar
0354 inb(int port)
0355 {
0356   uchar data;
0357   __asm __volatile("inb %w1,%0" : "=a" (data) : "d" (port));
0358   return data;
0359 }
0360
0361 static __inline void
0362 insl(int port, void *addr, int cnt)
0363 {
0364   __asm __volatile("cld\n\trepne\n\tinsl"     :
0365                    "=D" (addr), "=c" (cnt)    :
0366                    "d" (port), "0" (addr), "1" (cnt)  :
0367                    "memory", "cc");
0368 }
0369
0370 static __inline void
0371 outb(int port, uchar data)
0372 {
0373   __asm __volatile("outb %0,%w1" : : "a" (data), "d" (port));
0374 }
0375
0376 static __inline void
0377 outw(int port, ushort data)
0378 {
0379   __asm __volatile("outw %0,%w1" : : "a" (data), "d" (port));
0380 }
0381
0382 static __inline void
0383 outsl(int port, const void *addr, int cnt)
0384 {
0385   __asm __volatile("cld\n\trepne\n\toutsl"    :
0386                    "=S" (addr), "=c" (cnt)    :
0387                    "d" (port), "0" (addr), "1" (cnt)  :
0388                    "cc");
0389 }
0390
0391
0392
0393
0394
0395
0396
0397
0398
0399
```

```
0400 struct segdesc;
0401
0402 static __inline void
0403 lgdt(struct segdesc *p, int size)
0404 {
0405   volatile ushort pd[3];
0406
0407   pd[0] = size-1;
0408   pd[1] = (uint)p;
0409   pd[2] = (uint)p >> 16;
0410
0411   asm volatile("lgdt (%0)" : : "g" (pd));
0412 }
0413
0414 struct gatedesc;
0415
0416 static __inline void
0417 lidt(struct gatedesc *p, int size)
0418 {
0419   volatile ushort pd[3];
0420
0421   pd[0] = size-1;
0422   pd[1] = (uint)p;
0423   pd[2] = (uint)p >> 16;
0424
0425   asm volatile("lidt (%0)" : : "g" (pd));
0426 }
0427
0428 static __inline void
0429 ltr(ushort sel)
0430 {
0431   __asm __volatile("ltr %0" : : "r" (sel));
0432 }
0433
0434 static __inline uint
0435 read_eflags(void)
0436 {
0437   uint eflags;
0438   __asm __volatile("pushfl; popl %0" : "=r" (eflags));
0439   return eflags;
0440 }
0441
0442 static __inline void
0443 write_eflags(uint eflags)
0444 {
0445   __asm __volatile("pushl %0; popfl" : : "r" (eflags));
0446 }
0447
0448
0449
```



```
0450 static __inline void
0451 cpuid(uint info, uint *eaxp, uint *ebxp, uint *ecxp, uint *edxp)
0452 {
0453   uint eax, ebx, ecx, edx;
0454   asm volatile("cpuid" :
0455                "=a" (eax), "=b" (ebx), "=c" (ecx), "=d" (edx) :
0456                "a" (info));
0457   if(eaxp)
0458     *eaxp = eax;
0459   if(ebxp)
0460     *ebxp = ebx;
0461   if(ecxp)
0462     *ecxp = ecx;
0463   if(edxp)
0464     *edxp = edx;
0465 }
0466
0467 static __inline uint
0468 cmpxchg(uint oldval, uint newval, volatile uint* lock_addr)
0469 {
0470   uint result;
0471   __asm__ __volatile__("lock; cmpxchgl %2, %0" :
0472                        "+m" (*lock_addr), "=a" (result) :
0473                        "r"(newval), "1"(oldval) :
0474                        "cc");
0475   return result;
0476 }
0477
0478 static __inline void
0479 cli(void)
0480 {
0481   __asm__ volatile("cli");
0482 }
0483
0484 static __inline void
0485 sti(void)
0486 {
0487   __asm__ volatile("sti");
0488 }
0489
0490
0491
0492
0493
0494
0495
0496
0497
0498
0499
```

```
0500 // Layout of the trap frame on the stack upon entry to trap.
0501 struct trapframe {
0502   // registers as pushed by pusha
0503   uint edi;
0504   uint esi;
0505   uint ebp;
0506   uint oesp;      // useless & ignored
0507   uint ebx;
0508   uint edx;
0509   uint ecx;
0510   uint eax;
0511
0512   // rest of trap frame
0513   ushort es;
0514   ushort padding1;
0515   ushort ds;
0516   ushort padding2;
0517   uint trapno;
0518
0519   // below here defined by x86 hardware
0520   uint err;
0521   uint eip;
0522   ushort cs;
0523   ushort padding3;
0524   uint eflags;
0525
0526   // below here only when crossing rings, such as from user to kernel
0527   uint esp;
0528   ushort ss;
0529   ushort padding4;
0530 };
0531
0532
0533
0534
0535
0536
0537
0538
0539
0540
0541
0542
0543
0544
0545
0546
0547
0548
0549
```



```
0550 //
0551 // macros to create x86 segments from assembler
0552 //
0553
0554 #define SEG_NULLASM                                             \
0555         .word 0, 0;                                             \
0556         .byte 0, 0, 0, 0
0557
0558 #define SEG_ASM(type,base,lim)                                  \
0559         .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
0560         .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
0561                 (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
0562
0563 #define STA_X     0x8       // Executable segment
0564 #define STA_E     0x4       // Expand down (non-executable segments)
0565 #define STA_C     0x4       // Conforming code segment (executable only)
0566 #define STA_W     0x2       // Writeable (non-executable segments)
0567 #define STA_R     0x2       // Readable (executable segments)
0568 #define STA_A     0x1       // Accessed
0569
0570
0571
0572
0573
0574
0575
0576
0577
0578
0579
0580
0581
0582
0583
0584
0585
0586
0587
0588
0589
0590
0591
0592
0593
0594
0595
0596
0597
0598
0599
```

```
0600 // This file contains definitions for the
0601 // x86 memory management unit (MMU).
0602
0603 // Eflags register
0604 #define FL_CF           0x00000001      // Carry Flag
0605 #define FL_PF           0x00000004      // Parity Flag
0606 #define FL_AF           0x00000010      // Auxiliary carry Flag
0607 #define FL_ZF           0x00000040      // Zero Flag
0608 #define FL_SF           0x00000080      // Sign Flag
0609 #define FL_TF           0x00000100      // Trap Flag
0610 #define FL_IF           0x00000200      // Interrupt Enable
0611 #define FL_DF           0x00000400      // Direction Flag
0612 #define FL_OF           0x00000800      // Overflow Flag
0613 #define FL_IOPL_MASK    0x00003000      // I/O Privilege Level bitmask
0614 #define FL_IOPL_0       0x00000000      //   IOPL == 0
0615 #define FL_IOPL_1       0x00001000      //   IOPL == 1
0616 #define FL_IOPL_2       0x00002000      //   IOPL == 2
0617 #define FL_IOPL_3       0x00003000      //   IOPL == 3
0618 #define FL_NT           0x00004000      // Nested Task
0619 #define FL_RF           0x00010000      // Resume Flag
0620 #define FL_VM           0x00020000      // Virtual 8086 mode
0621 #define FL_AC           0x00040000      // Alignment Check
0622 #define FL_VIF          0x00080000      // Virtual Interrupt Flag
0623 #define FL_VIP          0x00100000      // Virtual Interrupt Pending
0624 #define FL_ID           0x00200000      // ID flag
0625
0626 // Segment Descriptor
0627 struct segdesc {
0628   uint lim_15_0 : 16;  // Low bits of segment limit
0629   uint base_15_0 : 16; // Low bits of segment base address
0630   uint base_23_16 : 8; // Middle bits of segment base address
0631   uint type : 4;       // Segment type (see STS_ constants)
0632   uint s : 1;          // 0 = system, 1 = application
0633   uint dpl : 2;        // Descriptor Privilege Level
0634   uint p : 1;          // Present
0635   uint lim_19_16 : 4;  // High bits of segment limit
0636   uint avl : 1;        // Unused (available for software use)
0637   uint rsv1 : 1;       // Reserved
0638   uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
0639   uint g : 1;          // Granularity: limit scaled by 4K when set
0640   uint base_31_24 : 8; // High bits of segment base address
0641 };
0642
0643
0644
0645
0646
0647
0648
0649
```



```
0650 // Null segment
0651 #define SEG_NULL        (struct segdesc){ 0,0,0,0,0,0,0,0,0,0,0,0,0,0 }
0652
0653 // Normal segment
0654 #define SEG(type, base, lim, dpl) (struct segdesc)                      \
0655 { ((lim) >> 12) & 0xffff, (base) & 0xffff, ((base) >> 16) & 0xff,       \
0656     type, 1, dpl, 1, (uint) (lim) >> 28, 0, 0, 1, 1,            \
0657     (uint) (base) >> 24 }
0658
0659 #define SEG16(type, base, lim, dpl) (struct segdesc)                    \
0660 { (lim) & 0xffff, (base) & 0xffff, ((base) >> 16) & 0xff,               \
0661     type, 1, dpl, 1, (uint) (lim) >> 16, 0, 0, 1, 0,            \
0662     (uint) (base) >> 24 }
0663
0664 // Application segment type bits
0665 #define STA_X       0x8     // Executable segment
0666 #define STA_E       0x4     // Expand down (non-executable segments)
0667 #define STA_C       0x4     // Conforming code segment (executable only)
0668 #define STA_W       0x2     // Writeable (non-executable segments)
0669 #define STA_R       0x2     // Readable (executable segments)
0670 #define STA_A       0x1     // Accessed
0671
0672 // System segment type bits
0673 #define STS_T16A    0x1     // Available 16-bit TSS
0674 #define STS_LDT     0x2     // Local Descriptor Table
0675 #define STS_T16B    0x3     // Busy 16-bit TSS
0676 #define STS_CG16    0x4     // 16-bit Call Gate
0677 #define STS_TG      0x5     // Task Gate / Coum Transmitions
0678 #define STS_IG16    0x6     // 16-bit Interrupt Gate
0679 #define STS_TG16    0x7     // 16-bit Trap Gate
0680 #define STS_T32A    0x9     // Available 32-bit TSS
0681 #define STS_T32B    0xB     // Busy 32-bit TSS
0682 #define STS_CG32    0xC     // 32-bit Call Gate
0683 #define STS_IG32    0xE     // 32-bit Interrupt Gate
0684 #define STS_TG32    0xF     // 32-bit Trap Gate
0685
0686 // Task state segment format
0687 struct taskstate {
0688   uint link;      // Old ts selector
0689   uint esp0;      // Stack pointers and segment selectors
0690   ushort ss0;     //   after an increase in privilege level
0691   ushort padding1;
0692   uint *esp1;
0693   ushort ss1;
0694   ushort padding2;
0695   uint *esp2;
0696   ushort ss2;
0697   ushort padding3;
0698   void *cr3;     // Page directory base
0699   uint *eip;     // Saved state from last task switch
```

```
0700   uint eflags;
0701   uint eax;       // More saved state (registers)
0702   uint ecx;
0703   uint edx;
0704   uint ebx;
0705   uint *esp;
0706   uint *ebp;
0707   uint esi;
0708   uint edi;
0709   ushort es;              // Even more saved state (segment selectors)
0710   ushort padding4;
0711   ushort cs;
0712   ushort padding5;
0713   ushort ss;
0714   ushort padding6;
0715   ushort ds;
0716   ushort padding7;
0717   ushort fs;
0718   ushort padding8;
0719   ushort gs;
0720   ushort padding9;
0721   ushort ldt;
0722   ushort padding10;
0723   ushort t;               // Trap on task switch
0724   ushort iomb;    // I/O map base address
0725 };
0726
0727 // Gate descriptors for interrupts and traps
0728 struct gatedesc {
0729   uint off_15_0 : 16;   // low 16 bits of offset in segment
0730   uint ss : 16;         // segment selector
0731   uint args : 5;        // # args, 0 for interrupt/trap gates
0732   uint rsv1 : 3;        // reserved(should be zero I guess)
0733   uint type : 4;        // type(STS_{TG,IG32,TG32})
0734   uint s : 1;           // must be 0 (system)
0735   uint dpl : 2;         // descriptor(meaning new) privilege level
0736   uint p : 1;           // Present
0737   uint off_31_16 : 16;  // high bits of offset in segment
0738 };
0739
0740
0741
0742
0743
0744
0745
0746
0747
0748
0749
```



```
0750 // Set up a normal interrupt/trap gate descriptor.
0751 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
0752 //   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
0753 // - sel: Code segment selector for interrupt/trap handler
0754 // - off: Offset in code segment for interrupt/trap handler
0755 // - dpl: Descriptor Privilege Level -
0756 //        the privilege level required for software to invoke
0757 //        this interrupt/trap gate explicitly using an int instruction.
0758 #define SETGATE(gate, istrap, sel, off, d)                      \
0759 {                                                               \
0760   (gate).off_15_0 = (uint) (off) & 0xffff;                \
0761   (gate).ss = (sel);                                      \
0762   (gate).args = 0;                                        \
0763   (gate).rsv1 = 0;                                        \
0764   (gate).type = (istrap) ? STS_TG32 : STS_IG32;   \
0765   (gate).s = 0;                                           \
0766   (gate).dpl = (d);                                       \
0767   (gate).p = 1;                                           \
0768   (gate).off_31_16 = (uint) (off) >> 16;          \
0769 }
0770
0771
0772
0773
0774
0775
0776
0777
0778
0779
0780
0781
0782
0783
0784
0785
0786
0787
0788
0789
0790
0791
0792
0793
0794
0795
0796
0797
0798
0799
```

```
0800 // Format of an ELF executable file
0801
0802 #define ELF_MAGIC 0x464C457FU  // "\x7FELF" in little endian
0803
0804 // File header
0805 struct elfhdr {
0806   uint magic;  // must equal ELF_MAGIC
0807   uchar elf[12];
0808   ushort type;
0809   ushort machine;
0810   uint version;
0811   uint entry;
0812   uint phoff;
0813   uint shoff;
0814   uint flags;
0815   ushort ehsize;
0816   ushort phentsize;
0817   ushort phnum;
0818   ushort shentsize;
0819   ushort shnum;
0820   ushort shstrndx;
0821 };
0822
0823 // Program section header
0824 struct proghdr {
0825   uint type;
0826   uint offset;
0827   uint va;
0828   uint pa;
0829   uint filesz;
0830   uint memsz;
0831   uint flags;
0832   uint align;
0833 };
0834
0835 // Values for Proghdr type
0836 #define ELF_PROG_LOAD           1
0837
0838 // Flag bits for Proghdr flags
0839 #define ELF_PROG_FLAG_EXEC      1
0840 #define ELF_PROG_FLAG_WRITE     2
0841 #define ELF_PROG_FLAG_READ      4
0842
0843
0844
0845
0846
0847
0848
0849
```



```
0850 // See MultiProcessor Specification Version 1.[14].
0851
0852 struct mp {             // floating pointer
0853   uchar signature[4];           // "_MP_"
0854   void *physaddr;               // phys addr of MP config table
0855   uchar length;                 // 1
0856   uchar specrev;                // [14]
0857   uchar checksum;               // all bytes must add up to 0
0858   uchar type;                   // MP system config type
0859   uchar imcrp;
0860   uchar reserved[3];
0861 };
0862
0863 struct mpctb {          // configuration table header
0864   uchar signature[4];           // "PCMP"
0865   ushort length;                // total table length
0866   uchar version;                // [14]
0867   uchar checksum;               // all bytes must add up to 0
0868   uchar product[20];            // product id
0869   uint *oemtable;               // OEM table pointer
0870   ushort oemlength;             // OEM table length
0871   ushort entry;                 // entry count
0872   uint *lapicaddr;              // address of local APIC
0873   ushort xlength;               // extended table length
0874   uchar xchecksum;              // extended table checksum
0875   uchar reserved;
0876 };
0877
0878 struct mppe {           // processor table entry
0879   uchar type;                   // entry type (0)
0880   uchar apicid;                 // local APIC id
0881   uchar version;                // local APIC verison
0882   uchar flags;                  // CPU flags
0883   uchar signature[4];           // CPU signature
0884   uint feature;                 // feature flags from CPUID instruction
0885   uchar reserved[8];
0886 };
0887
0888 struct mpbe {           // bus table entry
0889   uchar type;                   // entry type (1)
0890   uchar busno;                  // bus id
0891   char string[6];               // bus type string
0892 };
0893
0894
0895
0896
0897
0898
0899
```

```
0900 struct mpioapic {       // I/O APIC table entry
0901   uchar type;                   // entry type (2)
0902   uchar apicno;                 // I/O APIC id
0903   uchar version;                // I/O APIC version
0904   uchar flags;                  // I/O APIC flags
0905   uint *addr;                  // I/O APIC address
0906 };
0907
0908 struct mpie {           // interrupt table entry
0909   uchar type;                   // entry type ([34])
0910   uchar intr;                   // interrupt type
0911   ushort flags;                 // interrupt flag
0912   uchar busno;                  // source bus id
0913   uchar irq;                    // source bus irq
0914   uchar apicno;                 // destination APIC id
0915   uchar intin;                  // destination APIC [L]INTIN#
0916 };
0917
0918 enum {                  // table entry types
0919   MPPROCESSOR   = 0x00,         // one entry per processor
0920   MPBUS = 0x01,                 // one entry per bus
0921   MPIOAPIC = 0x02,              // one entry per I/O APIC
0922   MPIOINTR = 0x03,              // one entry per bus interrupt source
0923   MPLINTR = 0x04,               // one entry per system interrupt source
0924
0925   MPSASM = 0x80,
0926   MPHIERARCHY   = 0x81,
0927   MPCBASM = 0x82,
0928
0929                         // PCMPprocessor and PCMPioapic flags
0930   MPEN = 0x01,                  // enabled
0931   MPBP = 0x02,                  // bootstrap processor
0932
0933                         // PCMPiointr and PCMPlintr flags
0934   MPPOMASK = 0x03,              // polarity conforms to bus specs
0935   MPHIGH = 0x01,                // active high
0936   MPLOW = 0x03,                 // active low
0937   MPELMASK = 0x0C,              // trigger mode of APIC input signals
0938   MPEDGE = 0x04,                // edge-triggered
0939   MPLEVEL = 0x0C,               // level-triggered
0940
0941                         // PCMPiointr and PCMPlintr interrupt type
0942   MPINT = 0x00,                 // vectored interrupt from APIC Rdt
0943   MPNMI = 0x01,                 // non-maskable interrupt
0944   MPSMI = 0x02,                 // system management interrupt
0945   MPExtINT = 0x03,              // vectored interrupt from external PIC
0946 };
0947
0948
0949
```



```
0950 // Common bits for
0951 //      I/O APIC Redirection Table Entry;
0952 //      Local APIC Local Interrupt Vector Table;
0953 //      Local APIC Inter-Processor Interrupt;
0954 //      Local APIC Timer Vector Table.
0955 enum {
0956   APIC_FIXED     = 0x00000000,  // [10:8] Delivery Mode
0957   APIC_LOWEST    = 0x00000100,  // Lowest priority
0958   APIC_SMI       = 0x00000200,  // System Management Interrupt
0959   APIC_RR        = 0x00000300,  // Remote Read
0960   APIC_NMI       = 0x00000400,
0961   APIC_INIT      = 0x00000500,  // INIT/RESET
0962   APIC_STARTUP   = 0x00000600,  // Startup IPI
0963   APIC_EXTINT    = 0x00000700,
0964
0965   APIC_PHYSICAL  = 0x00000000,  // [11] Destination Mode (RW)
0966   APIC_LOGICAL   = 0x00000800,
0967
0968   APIC_DELIVS    = 0x00001000,  // [12] Delivery Status (RO)
0969   APIC_HIGH      = 0x00000000,  // [13] Interrupt Input Pin Polarity (RW)
0970   APIC_LOW       = 0x00002000,
0971   APIC_REMOTEIRR = 0x00004000,  // [14] Remote IRR (RO)
0972   APIC_EDGE      = 0x00000000,  // [15] Trigger Mode (RW)
0973   APIC_LEVEL     = 0x00008000,
0974   APIC_IMASK     = 0x00010000,  // [16] Interrupt Mask
0975 };
0976
0977
0978
0979
0980
0981
0982
0983
0984
0985
0986
0987
0988
0989
0990
0991
0992
0993
0994
0995
0996
0997
0998
0999
```

```
#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
```

```
1200 #include "types.h"
1201 #include "param.h"
1202 #include "mmu.h"
1203 #include "proc.h"
1204 #include "defs.h"
1205 #include "x86.h"
1206 #include "traps.h"
1207 #include "syscall.h"
1208 #include "elf.h"
1209 #include "param.h"
1210 #include "spinlock.h"
1211
1212 extern char edata[], end[];
1213 extern uchar _binary__init_start[], _binary__init_size[];
1214
1215 void process0();
1216
1217 // Bootstrap processor starts running C code here.
1218 // This is called main0 not main so that it can have
1219 // a void return type.  Gcc can't handle functions named
1220 // main that don't return int.  Really.
1221 void
1222 main0(void)
1223 {
1224   int i;
1225   int bcpu;
1226   struct proc *p;
1227
1228   // clear BSS
1229   memset(edata, 0, end - edata);
1230
1231   // Prevent release() from enabling interrupts.
1232   for(i=0; i<NCPU; i++)
1233     cpus[i].nlock = 1;
1234
1235   mp_init(); // collect info about this machine
1236   bcpu = mp_bcpu();
1237
1238   // switch to bootstrap processor's stack
1239   asm volatile("movl %0, %%esp" : : "r" (cpus[0].mpstack + MPSTACK - 32));
1240   asm volatile("movl %0, %%ebp" : : "r" (cpus[0].mpstack + MPSTACK));
1241
1242   lapic_init(bcpu);
1243
1244   cprintf("\ncpu%d: starting xv6\n\n", cpu());
1245
1246
1247
1248
1249
```



```
1250   pinit(); // process table
1251   binit(); // buffer cache
1252   pic_init();
1253   ioapic_init();
1254   kinit(); // physical memory allocator
1255   tvinit(); // trap vectors
1256   idtinit(); // this CPU's interrupt descriptor table
1257   fileinit();
1258   iinit(); // i-node table
1259
1260   // initialize process 0
1261   p = &proc[0];
1262   p->state = RUNNABLE;
1263   p->kstack = kalloc(KSTACKSIZE);
1264
1265   // cause proc[0] to start in kernel at process0
1266   p->jmpbuf.eip = (uint) process0;
1267   p->jmpbuf.esp = (uint) (p->kstack + KSTACKSIZE - 4);
1268
1269   // make sure there's a TSS
1270   setupsegs(0);
1271
1272   // initialize I/O devices, let them enable interrupts
1273   console_init();
1274   ide_init();
1275
1276   // start other CPUs
1277   mp_startthem();
1278
1279   // turn on timer
1280   if(ismp)
1281     lapic_timerinit();
1282   else
1283     pit8253_timerinit();
1284
1285   // enable interrupts on the local APIC
1286   lapic_enableintr();
1287
1288   // enable interrupts on this processor.
1289   cpus[cpu()].nlock--;
1290   sti();
1291
1292   scheduler();
1293 }
1294
1295
1296
1297
1298
1299
```

```
1300 // Additional processors start here.
1301 void
1302 mpmain(void)
1303 {
1304   cprintf("cpu%d: starting\n", cpu());
1305   idtinit(); // CPU's idt
1306   if(cpu() == 0)
1307     panic("mpmain on cpu 0");
1308   lapic_init(cpu());
1309   lapic_timerinit();
1310   lapic_enableintr();
1311
1312   // make sure there's a TSS
1313   setupsegs(0);
1314
1315   cpuid(0, 0, 0, 0, 0);  // memory barrier
1316   cpus[cpu()].booted = 1;
1317
1318   // Enable interrupts on this processor.
1319   cpus[cpu()].nlock--;
1320   sti();
1321
1322   scheduler();
1323 }
1324
1325 // proc[0] starts here, called by scheduler() in the ordinary way.
1326 void
1327 process0()
1328 {
1329   struct proc *p0 = &proc[0];
1330   struct proc *p1;
1331   extern struct spinlock proc_table_lock;
1332   struct trapframe tf;
1333
1334   release(&proc_table_lock);
1335
1336   p0->cwd = iget(rootdev, 1);
1337   iunlock(p0->cwd);
1338
1339   // dummy user memory to make copyproc() happy.
1340   // must be big enough to hold the init binary.
1341   p0->sz = PAGE;
1342   p0->mem = kalloc(p0->sz);
1343
1344   // fake a trap frame as if a user process had made a system
1345   // call, so that copyproc will have a place for the new
1346   // process to return to.
1347   p0->tf = &tf;
1348   memset(p0->tf, 0, sizeof(struct trapframe));
1349   p0->tf->es = p0->tf->ds = p0->tf->ss = (SEG_UDATA << 3) | 3;
```



```
1350   p0->tf->cs = (SEG_UCODE << 3) | 3;
1351   p0->tf->eflags = FL_IF;
1352   p0->tf->esp = p0->sz;
1353
1354   p1 = copyproc(p0);
1355
1356   load_icode(p1, _binary__init_start, (uint) _binary__init_size);
1357   p1->state = RUNNABLE;
1358
1359   proc_wait();
1360   panic("init exited");
1361 }
1362
1363 void
1364 load_icode(struct proc *p, uchar *binary, uint size)
1365 {
1366   int i;
1367   struct elfhdr *elf;
1368   struct proghdr *ph;
1369
1370   elf = (struct elfhdr*) binary;
1371   if(elf->magic != ELF_MAGIC)
1372     panic("load_icode: not an ELF binary");
1373
1374   p->tf->eip = elf->entry;
1375
1376   // Map and load segments as directed.
1377   ph = (struct proghdr*) (binary + elf->phoff);
1378   for(i = 0; i < elf->phnum; i++, ph++) {
1379     if(ph->type != ELF_PROG_LOAD)
1380       continue;
1381     if(ph->va + ph->memsz < ph->va)
1382       panic("load_icode: overflow in proghdr");
1383     if(ph->va + ph->memsz >= p->sz)
1384       panic("load_icode: icode too large");
1385
1386     // Load/clear the segment
1387     memmove(p->mem + ph->va, binary + ph->offset, ph->filesz);
1388     memset(p->mem + ph->va + ph->filesz, 0, ph->memsz - ph->filesz);
1389   }
1390 }
1391
1392
1393
1394
1395
1396
1397
1398
1399
```

```
1400 #include "types.h"
1401 #include "mp.h"
1402 #include "defs.h"
1403 #include "param.h"
1404 #include "x86.h"
1405 #include "traps.h"
1406 #include "mmu.h"
1407 #include "proc.h"
1408
1409 static char *buses[] = {
1410   "CBUSI ",
1411   "CBUSII",
1412   "EISA  ",
1413   "FUTURE",
1414   "INTERN",
1415   "ISA   ",
1416   "MBI   ",
1417   "MBII  ",
1418   "MCA   ",
1419   "MPI   ",
1420   "MPSA  ",
1421   "NUBUS ",
1422   "PCI   ",
1423   "PCMCIA",
1424   "TC    ",
1425   "VL    ",
1426   "VME   ",
1427   "XPRESS",
1428   0,
1429 };
1430
1431 struct cpu cpus[NCPU];
1432 int ismp;
1433 int ncpu;
1434 uchar ioapic_id;
1435
1436 static struct cpu *bcpu;
1437 static struct mp *mp;  // The MP floating point structure
1438
1439 static struct mp*
1440 mp_scan(uchar *addr, int len)
1441 {
1442   uchar *e, *p, sum;
1443   int i;
1444
1445   e = addr+len;
1446   for(p = addr; p < e; p += sizeof(struct mp)){
1447     if(memcmp(p, "_MP_", 4))
1448       continue;
1449     sum = 0;
```



```
1450     for(i = 0; i < sizeof(struct mp); i++)
1451       sum += p[i];
1452     if(sum == 0)
1453       return (struct mp*)p;
1454   }
1455   return 0;
1456 }
1457
1458 // Search for the MP Floating Pointer Structure, which according to the
1459 // spec is in one of the following three locations:
1460 // 1) in the first KB of the EBDA;
1461 // 2) in the last KB of system base memory;
1462 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
1463 static struct mp*
1464 mp_search(void)
1465 {
1466   uchar *bda;
1467   uint p;
1468   struct mp *mp;
1469
1470   bda = (uchar*) 0x400;
1471   if((p = (bda[0x0F]<<8)|bda[0x0E])){
1472     if((mp = mp_scan((uchar*) p, 1024)))
1473       return mp;
1474   }
1475   else{
1476     p = ((bda[0x14]<<8)|bda[0x13])*1024;
1477     if((mp = mp_scan((uchar*)p-1024, 1024)))
1478       return mp;
1479   }
1480   return mp_scan((uchar*)0xF0000, 0x10000);
1481 }
1482
1483 // Search for an MP configuration table. For now,
1484 // don't accept the default configurations (physaddr == 0).
1485 // Check for correct signature, calculate the checksum and,
1486 // if correct, check the version.
1487 // To do: check extended table checksum.
1488 static int
1489 mp_detect(void)
1490 {
1491   struct mpctb *pcmp;
1492   uchar *p, sum;
1493   uint length;
1494
1495   if((mp = mp_search()) == 0 || mp->physaddr == 0)
1496     return 1;
1497
1498
1499
```

```
1500   pcmp = (struct mpctb*) mp->physaddr;
1501   if(memcmp(pcmp, "PCMP", 4))
1502     return 2;
1503
1504   length = pcmp->length;
1505   sum = 0;
1506   for(p = (uchar*)pcmp; length; length--)
1507     sum += *p++;
1508
1509   if(sum || (pcmp->version != 1 && pcmp->version != 4))
1510     return 3;
1511
1512   return 0;
1513 }
1514
1515 void
1516 mp_init(void)
1517 {
1518   int r;
1519   uchar *p, *e;
1520   struct mpctb *mpctb;
1521   struct mppe *proc;
1522   struct mpbe *bus;
1523   struct mpioapic *ioapic;
1524   struct mpie *intr;
1525   int i;
1526   uchar byte;
1527
1528   ncpu = 0;
1529   if((r = mp_detect()) != 0) {
1530     return;
1531   }
1532
1533   ismp = 1;
1534
1535   // Run through the table saving information needed for starting
1536   // application processors and initialising any I/O APICs. The table
1537   // is guaranteed to be in order such that only one pass is necessary.
1538
1539   mpctb = (struct mpctb*) mp->physaddr;
1540   lapicaddr = (uint*) mpctb->lapicaddr;
1541   p = ((uchar*)mpctb)+sizeof(struct mpctb);
1542   e = ((uchar*)mpctb)+mpctb->length;
1543
1544   while(p < e) {
1545     switch(*p){
1546     case MPPROCESSOR:
1547       proc = (struct mppe*) p;
1548       cpus[ncpu].apicid = proc->apicid;
1549       if(proc->flags & MPBP) {
```



```
1550         bcpu = &cpus[ncpu];
1551       }
1552       ncpu++;
1553       p += sizeof(struct mppe);
1554       continue;
1555     case MPBUS:
1556       bus = (struct mpbe*) p;
1557       for(i = 0; buses[i]; i++){
1558         if(strncmp(buses[i], bus->string, sizeof(bus->string)) == 0)
1559           break;
1560       }
1561       p += sizeof(struct mpbe);
1562       continue;
1563     case MPIOAPIC:
1564       ioapic = (struct mpioapic*) p;
1565       ioapic_id = ioapic->apicno;
1566       p += sizeof(struct mpioapic);
1567       continue;
1568     case MPIOINTR:
1569       intr = (struct mpie*) p;
1570       p += sizeof(struct mpie);
1571       continue;
1572     default:
1573       cprintf("mp_init: unknown PCMP type 0x%x (e-p 0x%x)\n", *p, e-p);
1574       while(p < e){
1575         cprintf("%uX ", *p);
1576         p++;
1577       }
1578       break;
1579     }
1580   }
1581
1582   if(mp->imcrp) {
1583     // It appears that Bochs doesn't support IMCR, so code won't run.
1584     outb(0x22, 0x70);   // Select IMCR
1585     byte = inb(0x23);   // Current contents
1586     byte |= 0x01;       // Mask external INTR
1587     outb(0x23, byte);   // Disconnect 8259s/NMI
1588   }
1589 }
1590
1591
1592 int
1593 mp_bcpu(void)
1594 {
1595   if(ismp)
1596     return bcpu-cpus;
1597   return 0;
1598 }
1599
```

```
1600 extern void mpmain(void);
1601
1602 // Write bootstrap code to unused memory at 0x7000.
1603 #define APBOOTCODE 0x7000
1604
1605 void
1606 mp_startthem(void)
1607 {
1608   extern uchar _binary_bootother_start[], _binary_bootother_size[];
1609   extern int main();
1610   int c;
1611
1612   memmove((void*) APBOOTCODE,_binary_bootother_start,
1613           (uint) _binary_bootother_size);
1614
1615   for(c = 0; c < ncpu; c++){
1616     // Our current cpu has already started.
1617     if(c == cpu())
1618       continue;
1619
1620     // Set target %esp
1621     *(uint*)(APBOOTCODE-4) = (uint) (cpus[c].mpstack) + MPSTACK;
1622
1623     // Set target %eip
1624     *(uint*)(APBOOTCODE-8) = (uint)mpmain;
1625
1626     // Go!
1627     lapic_startap(cpus[c].apicid, (uint) APBOOTCODE);
1628
1629     // Wait for cpu to get through bootstrap.
1630     while(cpus[c].booted == 0)
1631       ;
1632   }
1633 }
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
```



```
1650 #include "types.h"
1651 #include "stat.h"
1652 #include "user.h"
1653 #include "fs.h"
1654 #include "fcntl.h"
1655
1656 // init: The initial user-level program
1657
1658 char *sh_args[] = { "sh", 0 };
1659
1660 int
1661 main(void)
1662 {
1663   int pid;
1664
1665   if(open("console", O_RDWR) < 0){
1666     mknod("console", T_DEV, 1, 1);
1667     open("console", O_RDWR);
1668   }
1669   dup(0);  // stdout
1670   dup(0);  // stderr
1671
1672   for(;;){
1673     pid = fork();
1674     if(pid < 0){
1675       puts("init: fork failed\n");
1676       exit();
1677     }
1678     if(pid == 0){
1679       exec("sh", sh_args);
1680       puts("init: exec sh failed\n");
1681       exit();
1682     } else {
1683       wait();
1684     }
1685   }
1686 }
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
```

```
1700 // Mutual exclusion lock.
1701 struct spinlock {
1702   uint locked;   // Is the lock held?
1703
1704   // For debugging:
1705   char *name;    // Name of lock.
1706   int  cpu;      // The number of the cpu holding the lock.
1707   uint pcs[10];  // The call stack (an array of program counters)
1708                  // that locked the lock.
1709 };
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
```



```
1750 // Mutual exclusion spin locks.
1751
1752 #include "types.h"
1753 #include "defs.h"
1754 #include "x86.h"
1755 #include "mmu.h"
1756 #include "param.h"
1757 #include "proc.h"
1758 #include "spinlock.h"
1759
1760 extern int use_console_lock;
1761
1762 void
1763 initlock(struct spinlock *lock, char *name)
1764 {
1765   lock->name = name;
1766   lock->locked = 0;
1767   lock->cpu = 0xffffffff;
1768 }
1769
1770 // Record the current call stack in pcs[] by following the %ebp chain.
1771 void
1772 getcallerpcs(void *v, uint pcs[])
1773 {
1774   uint *ebp = (uint*)v - 2;
1775   int i;
1776   for(i = 0; i < 10; i++){
1777     if(ebp == 0 || ebp == (uint*)0xffffffff)
1778       break;
1779     pcs[i] = ebp[1];     // saved %eip
1780     ebp = (uint*)ebp[0]; // saved %ebp
1781   }
1782   for(; i < 10; i++)
1783     pcs[i] = 0;
1784 }
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
```

```
1800 // Acquire the lock.
1801 // Loops (spins) until the lock is acquired.
1802 // (Because contention is handled by spinning, must not
1803 // go to sleep holding any locks.)
1804 void
1805 acquire(struct spinlock *lock)
1806 {
1807   if(holding(lock))
1808     panic("acquire");
1809
1810   if(cpus[cpu()].nlock == 0)
1811     cli();
1812   cpus[cpu()].nlock++;
1813
1814   while(cmpxchg(0, 1, &lock->locked) == 1)
1815     ;
1816
1817   // Serialize instructions: now that lock is acquired, make sure
1818   // we wait for all pending writes from other processors.
1819   cpuid(0, 0, 0, 0, 0);  // memory barrier (see Ch 7, IA-32 manual vol 3)
1820
1821   // Record info about lock acquisition for debugging.
1822   // The +10 is only so that we can tell the difference
1823   // between forgetting to initialize lock->cpu
1824   // and holding a lock on cpu 0.
1825   lock->cpu = cpu() + 10;
1826   getcallerpcs(&lock, lock->pcs);
1827 }
1828
1829 // Release the lock.
1830 void
1831 release(struct spinlock *lock)
1832 {
1833   if(!holding(lock))
1834     panic("release");
1835
1836   lock->pcs[0] = 0;
1837   lock->cpu = 0xffffffff;
1838
1839   // Serialize instructions: before unlocking the lock, make sure
1840   // to flush any pending memory writes from this processor.
1841   cpuid(0, 0, 0, 0, 0);  // memory barrier (see Ch 7, IA-32 manual vol 3)
1842
1843   lock->locked = 0;
1844   if(--cpus[cpu()].nlock == 0)
1845     sti();
1846 }
1847
1848
1849
```



```
1850 // Check whether this cpu is holding the lock.
1851 int
1852 holding(struct spinlock *lock)
1853 {
1854   return lock->locked && lock->cpu == cpu() + 10;
1855 }
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
```

```
1900 // Segments in proc->gdt
1901 #define SEG_KCODE 1  // kernel code
1902 #define SEG_KDATA 2  // kernel data+stack
1903 #define SEG_UCODE 3
1904 #define SEG_UDATA 4
1905 #define SEG_TSS   5  // this process's task state
1906 #define NSEGS     6
1907
1908 // Saved registers for kernel context switches.
1909 // Don't need to save all the %fs etc. segment registers,
1910 // because they are constant across kernel contexts.
1911 // Save all the regular registers so we don't need to care
1912 // which are caller save.
1913 // Don't save %eax, because that's the return register.
1914 // The layout of jmpbuf is known to setjmp.S.
1915 struct jmpbuf {
1916   int ebx;
1917   int ecx;
1918   int edx;
1919   int esi;
1920   int edi;
1921   int esp;
1922   int ebp;
1923   int eip;
1924 };
1925
1926 enum proc_state { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
1927
1928 // Per-process state
1929 struct proc {
1930   char *mem;                // Start of process memory (kernel address)
1931   uint sz;                  // Size of process memory (bytes)
1932   char *kstack;             // Bottom of kernel stack for this process
1933   enum proc_state state;    // Process state
1934   int pid;                  // Process ID
1935   int ppid;                 // Parent pid
1936   void *chan;               // If non-zero, sleeping on chan
1937   int killed;               // If non-zero, have been killed
1938   struct file *ofile[NOFILE];  // Open files
1939   struct inode *cwd;        // Current directory
1940   struct jmpbuf jmpbuf;     // Jump here to run process
1941   struct trapframe *tf;     // Trap frame for current interrupt
1942 };
1943
1944
1945
1946
1947
1948
1949
```



```
1950 // Process memory is laid out contiguously:
1951 //   text
1952 //   original data and bss
1953 //   fixed-size stack
1954 //   expandable heap
1955
1956 extern struct proc proc[];
1957 extern struct proc *curproc[NCPU];  // Current (running) process per CPU
1958
1959 #define MPSTACK 512
1960
1961 // Per-CPU state
1962 struct cpu {
1963   uchar apicid;               // Local APIC ID
1964   struct jmpbuf jmpbuf;       // Jump here to enter scheduler
1965   struct taskstate ts;        // Used by x86 to find stack for interrupt
1966   struct segdesc gdt[NSEGS];  // x86 global descriptor table
1967   char mpstack[MPSTACK];      // Per-CPU startup stack
1968   volatile int booted;        // Has the CPU started?
1969   int nlock;                  // Number of locks currently held
1970 };
1971
1972 extern struct cpu cpus[NCPU];
1973 extern int ncpu;
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
```

```
2000 #include "types.h"
2001 #include "mmu.h"
2002 #include "x86.h"
2003 #include "param.h"
2004 #include "file.h"
2005 #include "proc.h"
2006 #include "defs.h"
2007 #include "spinlock.h"
2008
2009 struct spinlock proc_table_lock;
2010
2011 struct proc proc[NPROC];
2012 struct proc *curproc[NCPU];
2013 int next_pid = 1;
2014 extern void forkret(void);
2015 extern void forkret1(struct trapframe*);
2016
2017 void
2018 pinit(void)
2019 {
2020   initlock(&proc_table_lock, "proc_table");
2021 }
2022
2023 // Set up CPU's segment descriptors and task state for a
2024 // given process.
2025 // If p==0, set up for "idle" state for when scheduler()
2026 // is idling, not running any process.
2027 void
2028 setupsegs(struct proc *p)
2029 {
2030   struct cpu *c = &cpus[cpu()];
2031
2032   c->ts.ss0 = SEG_KDATA << 3;
2033   if(p){
2034     c->ts.esp0 = (uint)(p->kstack + KSTACKSIZE);
2035   } else {
2036     c->ts.esp0 = 0xffffffff;
2037   }
2038
2039   c->gdt[0] = SEG_NULL;
2040   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0x100000 + 64*1024, 0);
2041   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
2042   c->gdt[SEG_TSS] = SEG16(STS_T32A, (uint) &c->ts, sizeof(c->ts), 0);
2043   c->gdt[SEG_TSS].s = 0;
2044   if(p){
2045     c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, (uint)p->mem, p->sz, 3);
2046     c->gdt[SEG_UDATA] = SEG(STA_W, (uint)p->mem, p->sz, 3);
2047   } else {
2048     c->gdt[SEG_UCODE] = SEG_NULL;
2049     c->gdt[SEG_UDATA] = SEG_NULL;
```



```
2050   }
2051
2052   lgdt(c->gdt, sizeof c->gdt);
2053   ltr(SEG_TSS << 3);
2054 }
2055
2056 // Grow current process's memory by n bytes.
2057 // Return old size on success, -1 on failure.
2058 int
2059 growproc(int n)
2060 {
2061   struct proc *cp = curproc[cpu()];
2062   char *newmem, *oldmem;
2063
2064   newmem = kalloc(cp->sz + n);
2065   if(newmem == 0)
2066     return 0xffffffff;
2067   memmove(newmem, cp->mem, cp->sz);
2068   memset(newmem + cp->sz, 0, n);
2069   oldmem = cp->mem;
2070   cp->mem = newmem;
2071   kfree(oldmem, cp->sz);
2072   cp->sz += n;
2073   return cp->sz - n;
2074 }
2075
2076 // Look in the process table for an UNUSED proc.
2077 // If found, change state to EMBRYO and return it.
2078 // Otherwise return 0.
2079 struct proc*
2080 allocproc(void)
2081 {
2082   int i;
2083   struct proc *p;
2084
2085   for(i = 0; i < NPROC; i++){
2086     p = &proc[i];
2087     if(p->state == UNUSED){
2088       p->state = EMBRYO;
2089       return p;
2090     }
2091   }
2092   return 0;
2093 }
2094
2095
2096
2097
2098
2099
```

```
2100 // Create a new process copying p as the parent.
2101 // Does not copy the kernel stack.
2102 // Instead, sets up stack to return as if from system call.
2103 // Caller must arrange for process to run (set state to RUNNABLE).
2104 struct proc*
2105 copyproc(struct proc *p)
2106 {
2107   int i;
2108   struct proc *np;
2109
2110   // Allocate process.
2111   acquire(&proc_table_lock);
2112   if((np = allocproc()) == 0){
2113     release(&proc_table_lock);
2114     return 0;
2115   }
2116   np->pid = next_pid++;
2117   np->ppid = p->pid;
2118   release(&proc_table_lock);
2119
2120   // Copy user memory.
2121   np->sz = p->sz;
2122   np->mem = kalloc(np->sz);
2123   if(np->mem == 0){
2124     np->state = UNUSED;
2125     return 0;
2126   }
2127   memmove(np->mem, p->mem, np->sz);
2128
2129   // Allocate kernel stack.
2130   np->kstack = kalloc(KSTACKSIZE);
2131   if(np->kstack == 0){
2132     kfree(np->mem, np->sz);
2133     np->mem = 0;
2134     np->state = UNUSED;
2135     return 0;
2136   }
2137
2138   // Copy trapframe registers from parent.
2139   np->tf = (struct trapframe*)(np->kstack + KSTACKSIZE) - 1;
2140   memmove(np->tf, p->tf, sizeof(*np->tf));
2141
2142   // Clear %eax so that fork system call returns 0 in child.
2143   np->tf->eax = 0;
2144
2145   // Set up new jmpbuf to start executing at forkret (see below).
2146   memset(&np->jmpbuf, 0, sizeof np->jmpbuf);
2147   np->jmpbuf.eip = (uint)forkret;
2148   np->jmpbuf.esp = (uint)np->tf - 4;
2149
```



```
2150   // Copy file descriptors
2151   for(i = 0; i < NOFILE; i++){
2152     np->ofile[i] = p->ofile[i];
2153     if(np->ofile[i])
2154       fileincref(np->ofile[i]);
2155   }
2156
2157   np->cwd = p->cwd;
2158   iincref(p->cwd);
2159
2160   return np;
2161 }
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
```

```
2200 // Per-CPU process scheduler.
2201 // Each CPU calls scheduler() after setting itself up.
2202 // Scheduler never returns.  It loops, doing:
2203 //  - choose a process to run
2204 //  - longjmp to start running that process
2205 //  - eventually that process transfers control back
2206 //      via longjmp back to the top of scheduler.
2207 void
2208 scheduler(void)
2209 {
2210   struct proc *p;
2211   int i;
2212
2213   for(;;){
2214     // Loop over process table looking for process to run.
2215     acquire(&proc_table_lock);
2216
2217     for(i = 0; i < NPROC; i++){
2218       p = &proc[i];
2219       if(p->state != RUNNABLE)
2220         continue;
2221
2222       // Switch to chosen process.  It is the process's job
2223       // to release proc_table_lock and then reacquire it
2224       // before jumping back to us.
2225
2226       setupsegs(p);
2227       curproc[cpu()] = p;
2228       p->state = RUNNING;
2229       if(setjmp(&cpus[cpu()].jmpbuf) == 0)
2230         longjmp(&p->jmpbuf);
2231
2232       // Process is done running for now.
2233       // It should have changed its p->state before coming back.
2234       curproc[cpu()] = 0;
2235
2236       setupsegs(0);
2237     }
2238
2239     release(&proc_table_lock);
2240   }
2241 }
2242
2243
2244
2245
2246
2247
2248
2249
```



```
2250 // Enter scheduler.  Must already hold proc_table_lock
2251 // and have changed curproc[cpu()]->state.
2252 void
2253 sched(void)
2254 {
2255   struct proc *p = curproc[cpu()];
2256
2257   if(!holding(&proc_table_lock))
2258     panic("sched");
2259   if(cpus[cpu()].nlock != 1)
2260     panic("sched locks");
2261
2262   if(setjmp(&p->jmpbuf) == 0)
2263     longjmp(&cpus[cpu()].jmpbuf);
2264 }
2265
2266 // Give up the CPU for one scheduling round.
2267 void
2268 yield(void)
2269 {
2270   struct proc *p = curproc[cpu()];
2271
2272   acquire(&proc_table_lock);
2273   p->state = RUNNABLE;
2274   sched();
2275   release(&proc_table_lock);
2276 }
2277
2278 // A fork child's very first scheduling by scheduler()
2279 // will longjmp here. "return" to user space.
2280 void
2281 forkret(void)
2282 {
2283   // Still holding proc_table_lock from scheduler.
2284   release(&proc_table_lock);
2285
2286   // Jump into assembly, never to return.
2287   forkret1(curproc[cpu()]->tf);
2288 }
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
```

```
2300 // Atomically release lock and sleep on chan.
2301 // Reacquires lock when reawakened.
2302 void
2303 sleep(void *chan, struct spinlock *lk)
2304 {
2305   struct proc *p = curproc[cpu()];
2306
2307   if(p == 0)
2308     panic("sleep");
2309
2310   if(lk == 0)
2311     panic("sleep without lk");
2312
2313   // Must acquire proc_table_lock in order to
2314   // change p->state and then call sched.
2315   // Once we hold proc_table_lock, we can be
2316   // guaranteed that we won't miss any wakeup
2317   // (wakeup runs with proc_table_lock locked),
2318   // so it's okay to release lk.
2319   if(lk != &proc_table_lock){
2320     acquire(&proc_table_lock);
2321     release(lk);
2322   }
2323
2324   // Go to sleep.
2325   p->chan = chan;
2326   p->state = SLEEPING;
2327   sched();
2328
2329   // Tidy up.
2330   p->chan = 0;
2331
2332   // Reacquire original lock.
2333   if(lk != &proc_table_lock){
2334     release(&proc_table_lock);
2335     acquire(lk);
2336   }
2337 }
2338
2339 // Wake up all processes sleeping on chan.
2340 // Proc_table_lock must be held.
2341 void
2342 wakeup1(void *chan)
2343 {
2344   struct proc *p;
2345
2346   for(p = proc; p < &proc[NPROC]; p++)
2347     if(p->state == SLEEPING && p->chan == chan)
2348       p->state = RUNNABLE;
2349 }
```



```
2350 // Wake up all processes sleeping on chan.
2351 // Proc_table_lock is acquired and released.
2352 void
2353 wakeup(void *chan)
2354 {
2355   acquire(&proc_table_lock);
2356   wakeup1(chan);
2357   release(&proc_table_lock);
2358 }
2359
2360 // Kill the process with the given pid.
2361 // Process won't actually exit until it returns
2362 // to user space (see trap in trap.c).
2363 int
2364 proc_kill(int pid)
2365 {
2366   struct proc *p;
2367
2368   acquire(&proc_table_lock);
2369   for(p = proc; p < &proc[NPROC]; p++){
2370     if(p->pid == pid){
2371       p->killed = 1;
2372       // Wake process from sleep if necessary.
2373       if(p->state == SLEEPING)
2374         p->state = RUNNABLE;
2375       release(&proc_table_lock);
2376       return 0;
2377     }
2378   }
2379   release(&proc_table_lock);
2380   return -1;
2381 }
2382
2383 // Exit the current process.  Does not return.
2384 // Exited processes remain in the zombie state
2385 // until their parent calls wait() to find out they exited.
2386 void
2387 proc_exit(void)
2388 {
2389   struct proc *p;
2390   struct proc *cp = curproc[cpu()];
2391   int fd;
2392
2393   // Close all open files.
2394   for(fd = 0; fd < NOFILE; fd++){
2395     if(cp->ofile[fd]){
2396       fileclose(cp->ofile[fd]);
2397       cp->ofile[fd] = 0;
2398     }
2399   }
```

```
2400   idecref(cp->cwd);
2401   cp->cwd = 0;
2402
2403   acquire(&proc_table_lock);
2404
2405   // Wake up our parent.
2406   for(p = proc; p < &proc[NPROC]; p++)
2407     if(p->pid == cp->ppid)
2408       wakeup1(p);
2409
2410   // Reparent our children to process 1.
2411   for(p = proc; p < &proc[NPROC]; p++)
2412     if(p->ppid == cp->pid)
2413       p->ppid = 1;
2414
2415   // Jump into the scheduler, never to return.
2416   cp->killed = 0;
2417   cp->state = ZOMBIE;
2418   sched();
2419   panic("zombie exit");
2420 }
2421
2422 // Wait for a child process to exit and return its pid.
2423 // Return -1 if this process has no children.
2424 int
2425 proc_wait(void)
2426 {
2427   struct proc *p;
2428   struct proc *cp = curproc[cpu()];
2429   int i, havekids, pid;
2430
2431   acquire(&proc_table_lock);
2432   for(;;){
2433     // Scan through table looking for zombie children.
2434     havekids = 0;
2435     for(i = 0; i < NPROC; i++){
2436       p = &proc[i];
2437       if(p->state == UNUSED)
2438         continue;
2439       if(p->ppid == cp->pid){
2440         if(p->state == ZOMBIE){
2441           // Found one.
2442           kfree(p->mem, p->sz);
2443           kfree(p->kstack, KSTACKSIZE);
2444           pid = p->pid;
2445           p->state = UNUSED;
2446           p->pid = 0;
2447           p->ppid = 0;
2448           release(&proc_table_lock);
2449           return pid;
```



```
2450         }
2451         havekids = 1;
2452       }
2453     }
2454
2455     // No point waiting if we don't have any children.
2456     if(!havekids){
2457       release(&proc_table_lock);
2458       return -1;
2459     }
2460
2461     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2462     sleep(cp, &proc_table_lock);
2463   }
2464 }
2465
2466 // Print a process listing to console.  For debugging.
2467 // Runs when user types ^P on console.
2468 // No lock to avoid wedging a stuck machine further.
2469 void
2470 procdump(void)
2471 {
2472   int i;
2473   struct proc *p;
2474
2475   for(i = 0; i < NPROC; i++) {
2476     p = &proc[i];
2477     if(p->state == UNUSED)
2478       continue;
2479     cprintf("%d %d %p\n", p->pid, p->state);
2480   }
2481 }
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
```

```
2500 #   int  setjmp(struct jmpbuf *jmp);
2501 #   void longjmp(struct jmpbuf *jmp);
2502 #
2503 # Setjmp saves its stack environment in jmp for later use by longjmp.
2504 # It returns 0.
2505 #
2506 # Longjmp restores the environment saved by the last call of setjmp.
2507 # It then causes execution to continue as if the call of setjmp
2508 # had just returned 1.
2509 #
2510 # The caller of setjmp must not itself have returned in the interim.
2511 # All accessible data have values as of the time longjmp was called.
2512 #
2513 #    [Description, but not code, borrowed from Plan 9.]
2514
2515 .globl setjmp
2516 setjmp:
2517   movl 4(%esp), %eax
2518
2519   movl %ebx, 0(%eax)
2520   movl %ecx, 4(%eax)
2521   movl %edx, 8(%eax)
2522   movl %esi, 12(%eax)
2523   movl %edi, 16(%eax)
2524   movl %esp, 20(%eax)
2525   movl %ebp, 24(%eax)
2526   pushl 0(%esp)   # %eip
2527   popl 28(%eax)
2528
2529   movl $0, %eax   # return value
2530   ret
2531
2532 .globl longjmp
2533 longjmp:
2534   movl 4(%esp), %eax
2535
2536   movl 0(%eax), %ebx
2537   movl 4(%eax), %ecx
2538   movl 8(%eax), %edx
2539   movl 12(%eax), %esi
2540   movl 16(%eax), %edi
2541   movl 20(%eax), %esp
2542   movl 24(%eax), %ebp
2543
2544   addl $4, %esp   # pop and discard %eip
2545   pushl 28(%eax)  # push new %eip
2546
2547   movl $1, %eax   # return value (appears to come from setjmp!)
2548   ret
2549
```



```
2550 // Physical memory allocator, intended to allocate
2551 // memory for user processes. Allocates in 4096-byte "pages".
2552 // Free list is kept sorted and combines adjacent pages into
2553 // long runs, to make it easier to allocate big segments.
2554 // One reason the page size is 4k is that the x86 segment size
2555 // granularity is 4k.
2556
2557 #include "param.h"
2558 #include "types.h"
2559 #include "defs.h"
2560 #include "param.h"
2561 #include "mmu.h"
2562 #include "proc.h"
2563 #include "spinlock.h"
2564
2565 struct spinlock kalloc_lock;
2566
2567 struct run {
2568   struct run *next;
2569   int len; // bytes
2570 };
2571 struct run *freelist;
2572
2573 // Initialize free list of physical pages.
2574 // This code cheats by just considering one megabyte of
2575 // pages after _end.  Real systems would determine the
2576 // amount of memory available in the system and use it all.
2577 void
2578 kinit(void)
2579 {
2580   extern int end;
2581   uint mem;
2582   char *start;
2583
2584   initlock(&kalloc_lock, "kalloc");
2585   start = (char*) &end;
2586   start = (char*) (((uint)start + PAGE) & ~(PAGE-1));
2587   mem = 256; // assume computer has 256 pages of RAM
2588   cprintf("mem = %d\n", mem * PAGE);
2589   kfree(start, mem * PAGE);
2590 }
2591
2592
2593
2594
2595
2596
2597
2598
2599
```

```
2600 // Free the len bytes of memory pointed at by cp,
2601 // which normally should have been returned by a
2602 // call to kalloc(cp).  (The exception is when
2603 // initializing the allocator; see kinit above.)
2604 void
2605 kfree(char *cp, int len)
2606 {
2607   struct run **rr;
2608   struct run *p = (struct run*) cp;
2609   struct run *pend = (struct run*) (cp + len);
2610   int i;
2611
2612   if(len % PAGE)
2613     panic("kfree");
2614
2615   // Fill with junk to catch dangling refs.
2616   for(i = 0; i < len; i++)
2617     cp[i] = 1;
2618
2619   acquire(&kalloc_lock);
2620
2621   rr = &freelist;
2622   while(*rr){
2623     struct run *rend = (struct run*) ((char*)(*rr) + (*rr)->len);
2624     if(p >= *rr && p < rend)
2625       panic("freeing free page");
2626     if(pend == *rr){
2627       p->len = len + (*rr)->len;
2628       p->next = (*rr)->next;
2629       *rr = p;
2630       goto out;
2631     }
2632     if(pend < *rr){
2633       p->len = len;
2634       p->next = *rr;
2635       *rr = p;
2636       goto out;
2637     }
2638     if(p == rend){
2639       (*rr)->len += len;
2640       if((*rr)->next && (*rr)->next == pend){
2641         (*rr)->len += (*rr)->next->len;
2642         (*rr)->next = (*rr)->next->next;
2643       }
2644       goto out;
2645     }
2646     rr = &((*rr)->next);
2647   }
2648   p->len = len;
2649   p->next = 0;
```



```
2650   *rr = p;
2651
2652  out:
2653   release(&kalloc_lock);
2654 }
2655
2656 // Allocate n bytes of physical memory.
2657 // Returns a kernel-segment pointer.
2658 // Returns 0 if the memory cannot be allocated.
2659 char*
2660 kalloc(int n)
2661 {
2662   struct run **rr;
2663
2664   if(n % PAGE)
2665     panic("kalloc");
2666
2667   acquire(&kalloc_lock);
2668
2669   rr = &freelist;
2670   while(*rr){
2671     struct run *r = *rr;
2672     if(r->len == n){
2673       *rr = r->next;
2674       release(&kalloc_lock);
2675       return (char*) r;
2676     }
2677     if(r->len > n){
2678       char *p = (char*)r + (r->len - n);
2679       r->len -= n;
2680       release(&kalloc_lock);
2681       return p;
2682     }
2683     rr = &(*rr)->next;
2684   }
2685   release(&kalloc_lock);
2686   cprintf("kalloc: out of memory\n");
2687   return 0;
2688 }
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
```

```
2700 // System call numbers
2701 #define SYS_fork    1
2702 #define SYS_exit    2
2703 #define SYS_wait    3
2704 #define SYS_pipe    4
2705 #define SYS_write   5
2706 #define SYS_read    6
2707 #define SYS_close   7
2708 #define SYS_kill    8
2709 #define SYS_exec    9
2710 #define SYS_open   10
2711 #define SYS_mknod  11
2712 #define SYS_unlink 12
2713 #define SYS_fstat  13
2714 #define SYS_link   14
2715 #define SYS_mkdir  15
2716 #define SYS_chdir  16
2717 #define SYS_dup    17
2718 #define SYS_getpid 18
2719 #define SYS_sbrk   19
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
```



```
2750 .text
2751 .globl trap
2752 .globl trapret1
2753
2754 .globl alltraps
2755 alltraps:
2756   /* vectors.S sends all traps here */
2757   pushl   %ds      # build
2758   pushl   %es      #  trap
2759   pushal           #  frame
2760   movl $16,%eax    # SEG_KDATA << 3
2761   movw %ax,%ds     #  kernel
2762   movw %ax,%es     #  segments
2763   pushl %esp       # pass pointer to this trapframe
2764   call    trap     #  and call trap()
2765   addl $4, %esp
2766   # return falls through to trapret...
2767
2768   /*
2769    * a forked process RETs here
2770    * expects ESP to point to a Trapframe
2771    */
2772 .globl trapret
2773 trapret:
2774   popal
2775   popl %es
2776   popl %ds
2777   addl $0x8, %esp /* trapno and errcode */
2778   iret
2779
2780 .globl forkret1
2781 forkret1:
2782   movl 4(%esp), %esp
2783   jmp trapret
2784
2785 .globl  acpu
2786 acpu:
2787   .long 0
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
```

```
2800 // x86 trap and interrupt constants.
2801
2802 // Processor-defined:
2803 #define T_DIVIDE         0      // divide error
2804 #define T_DEBUG          1      // debug exception
2805 #define T_NMI            2      // non-maskable interrupt
2806 #define T_BRKPT          3      // breakpoint
2807 #define T_OFLOW          4      // overflow
2808 #define T_BOUND          5      // bounds check
2809 #define T_ILLOP          6      // illegal opcode
2810 #define T_DEVICE         7      // device not available
2811 #define T_DBLFLT         8      // double fault
2812 // #define T_COPROC      9      // reserved (not used since 486)
2813 #define T_TSS           10      // invalid task switch segment
2814 #define T_SEGNP         11      // segment not present
2815 #define T_STACK         12      // stack exception
2816 #define T_GPFLT         13      // genernal protection fault
2817 #define T_PGFLT         14      // page fault
2818 // #define T_RES        15      // reserved
2819 #define T_FPERR         16      // floating point error
2820 #define T_ALIGN         17      // aligment check
2821 #define T_MCHK          18      // machine check
2822 #define T_SIMDERR       19      // SIMD floating point error
2823
2824 // These are arbitrarily chosen, but with care not to overlap
2825 // processor defined exceptions or interrupt vectors.
2826 #define T_SYSCALL       48      // system call
2827 #define T_DEFAULT      500      // catchall
2828
2829 #define IRQ_OFFSET      32      // IRQ 0 corresponds to int IRQ_OFFSET
2830
2831 #define IRQ_TIMER        0
2832 #define IRQ_KBD          1
2833 #define IRQ_IDE         14
2834 #define IRQ_ERROR       19
2835 #define IRQ_SPURIOUS    31
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
```



```
2850 #include "types.h"
2851 #include "param.h"
2852 #include "mmu.h"
2853 #include "proc.h"
2854 #include "defs.h"
2855 #include "x86.h"
2856 #include "traps.h"
2857 #include "syscall.h"
2858
2859 // Interrupt descriptor table (shared by all CPUs).
2860 struct gatedesc idt[256];
2861 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
2862
2863 void
2864 tvinit(void)
2865 {
2866   int i;
2867
2868   for(i = 0; i < 256; i++)
2869     SETGATE(idt[i], 0, SEG_KCODE << 3, vectors[i], 0);
2870   SETGATE(idt[T_SYSCALL], 0, SEG_KCODE << 3, vectors[T_SYSCALL], 3);
2871 }
2872
2873 void
2874 idtinit(void)
2875 {
2876   lidt(idt, sizeof idt);
2877 }
2878
2879 void
2880 trap(struct trapframe *tf)
2881 {
2882   int v = tf->trapno;
2883   struct proc *cp = curproc[cpu()];
2884
2885   if(v == T_SYSCALL){
2886     if(cp->killed)
2887       proc_exit();
2888     cp->tf = tf;
2889     syscall();
2890     if(cp->killed)
2891       proc_exit();
2892     return;
2893   }
2894
2895
2896
2897
2898
2899
```

```
2900   // Increment nlock to make sure interrupts stay off
2901   // during interrupt handler.  Decrement before returning.
2902   cpus[cpu()].nlock++;
2903
2904   switch(v){
2905   case IRQ_OFFSET + IRQ_TIMER:
2906     lapic_timerintr();
2907     cpus[cpu()].nlock--;
2908     if(cp){
2909       // Force process exit if it has been killed and is in user space.
2910       // (If it is still executing in the kernel, let it keep running
2911       // until it gets to the regular system call return.)
2912       if((tf->cs&3) == 3 && cp->killed)
2913         proc_exit();
2914
2915       // Force process to give up CPU and let others run.
2916       if(cp->state == RUNNING)
2917         yield();
2918     }
2919     return;
2920
2921   case IRQ_OFFSET + IRQ_IDE:
2922     ide_intr();
2923     lapic_eoi();
2924     break;
2925
2926   case IRQ_OFFSET + IRQ_KBD:
2927     kbd_intr();
2928     lapic_eoi();
2929     break;
2930
2931   case IRQ_OFFSET + IRQ_SPURIOUS:
2932     cprintf("spurious interrupt from cpu %d eip %x\n", cpu(), tf->eip);
2933     break;
2934
2935   default:
2936     if(curproc[cpu()]) {
2937       // Assume process divided by zero or dereferenced null, etc.
2938       cprintf("pid %d: unhandled trap %d on cpu %d eip %x -- kill proc\n",
2939               curproc[cpu()]->pid, v, cpu(), tf->eip);
2940       proc_exit();
2941     }
2942
2943     // Otherwise it's our mistake.
2944     cprintf("unexpected trap %d from cpu %d eip %x\n", v, cpu(), tf->eip);
2945     panic("trap");
2946   }
2947
2948   cpus[cpu()].nlock--;
2949 }
```



```
2950 #!/usr/bin/perl -w
2951
2952 # Generate vectors.S, the trap/interrupt entry points.
2953 # There has to be one entry point per interrupt number
2954 # since otherwise there's no way for trap() to discover
2955 # the interrupt number.
2956
2957 print "# generated by vectors.pl - do not edit\n";
2958 print "# handlers\n";
2959 print ".text\n";
2960 print ".globl alltraps\n";
2961 for(my $i = 0; $i < 256; $i++){
2962     print ".globl vector$i\n";
2963     print "vector$i:\n";
2964     if(($i < 8 || $i > 14) && $i != 17){
2965         print "  pushl \$0\n";
2966     }
2967     print "  pushl \$$i\n";
2968     print "  jmp alltraps\n";
2969 }
2970
2971 print "\n# vector table\n";
2972 print ".data\n";
2973 print ".globl vectors\n";
2974 print "vectors:\n";
2975 for(my $i = 0; $i < 256; $i++){
2976     print "  .long vector$i\n";
2977 }
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
```

```
3000 #include "types.h"
3001 #include "stat.h"
3002 #include "param.h"
3003 #include "mmu.h"
3004 #include "proc.h"
3005 #include "defs.h"
3006 #include "x86.h"
3007 #include "traps.h"
3008 #include "syscall.h"
3009 #include "spinlock.h"
3010 #include "buf.h"
3011 #include "fs.h"
3012 #include "fsvar.h"
3013 #include "elf.h"
3014 #include "file.h"
3015 #include "fcntl.h"
3016
3017 // User code makes a system call with INT T_SYSCALL.
3018 // System call number in %eax.
3019 // Arguments on the stack, from the user call to the C
3020 // library system call function. The saved user %esp points
3021 // to a saved program counter, and then the first argument.
3022
3023 // Fetch the int at addr from process p.
3024 int
3025 fetchint(struct proc *p, uint addr, int *ip)
3026 {
3027   if(addr >= p->sz || addr+4 > p->sz)
3028     return -1;
3029   *ip = *(int*)(p->mem + addr);
3030   return 0;
3031 }
3032
3033 // Fetch the nul-terminated string at addr from process p.
3034 // Doesn't actually copy the string - just sets *pp to point at it.
3035 // Returns length of string, not including nul.
3036 int
3037 fetchstr(struct proc *p, uint addr, char **pp)
3038 {
3039   char *cp, *ep;
3040
3041   if(addr >= p->sz)
3042     return -1;
3043   *pp = p->mem + addr;
3044   ep = p->mem + p->sz;
3045   for(cp = *pp; cp < ep; cp++)
3046     if(*cp == 0)
3047       return cp - *pp;
3048   return -1;
3049 }
```



```
3050 // Fetch the argno'th word-sized system call argument as an integer.
3051 int
3052 argint(int argno, int *ip)
3053 {
3054   struct proc *p = curproc[cpu()];
3055
3056   return fetchint(p, p->tf->esp + 4 + 4*argno, ip);
3057 }
3058
3059 // Fetch the nth word-sized system call argument as a pointer
3060 // to a block of memory of size n bytes.  Check that the pointer
3061 // lies within the process address space.
3062 int
3063 argptr(int argno, char **pp, int size)
3064 {
3065   int i;
3066   struct proc *p = curproc[cpu()];
3067
3068   if(argint(argno, &i) < 0)
3069     return -1;
3070   if((uint)i >= p->sz || (uint)i+size >= p->sz)
3071     return -1;
3072   *pp = p->mem + i;
3073   return 0;
3074 }
3075
3076 // Fetch the nth word-sized system call argument as a string pointer.
3077 // Check that the pointer is valid and the string is nul-terminated.
3078 // (There is no shared writable memory, so the string can't change
3079 // between this check and being used by the kernel.)
3080 int
3081 argstr(int argno, char **pp)
3082 {
3083   int addr;
3084   if(argint(argno, &addr) < 0)
3085     return -1;
3086   return fetchstr(curproc[cpu()], addr, pp);
3087 }
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
```

```
3100 extern int sys_chdir(void);
3101 extern int sys_close(void);
3102 extern int sys_dup(void);
3103 extern int sys_exec(void);
3104 extern int sys_exit(void);
3105 extern int sys_fork(void);
3106 extern int sys_fstat(void);
3107 extern int sys_getpid(void);
3108 extern int sys_kill(void);
3109 extern int sys_link(void);
3110 extern int sys_mkdir(void);
3111 extern int sys_mknod(void);
3112 extern int sys_open(void);
3113 extern int sys_pipe(void);
3114 extern int sys_read(void);
3115 extern int sys_sbrk(void);
3116 extern int sys_unlink(void);
3117 extern int sys_wait(void);
3118 extern int sys_write(void);
3119
3120 void
3121 syscall(void)
3122 {
3123   struct proc *cp = curproc[cpu()];
3124   int num = cp->tf->eax;
3125   int ret = -1;
3126
3127   switch(num){
3128   case SYS_fork:
3129     ret = sys_fork();
3130     break;
3131   case SYS_exit:
3132     ret = sys_exit();
3133     break;
3134   case SYS_wait:
3135     ret = sys_wait();
3136     break;
3137   case SYS_pipe:
3138     ret = sys_pipe();
3139     break;
3140   case SYS_write:
3141     ret = sys_write();
3142     break;
3143   case SYS_read:
3144     ret = sys_read();
3145     break;
3146   case SYS_close:
3147     ret = sys_close();
3148     break;
3149   case SYS_kill:
```



```
3150     ret = sys_kill();
3151     break;
3152   case SYS_exec:
3153     ret = sys_exec();
3154     break;
3155   case SYS_open:
3156     ret = sys_open();
3157     break;
3158   case SYS_mknod:
3159     ret = sys_mknod();
3160     break;
3161   case SYS_unlink:
3162     ret = sys_unlink();
3163     break;
3164   case SYS_fstat:
3165     ret = sys_fstat();
3166     break;
3167   case SYS_link:
3168     ret = sys_link();
3169     break;
3170   case SYS_mkdir:
3171     ret = sys_mkdir();
3172     break;
3173   case SYS_chdir:
3174     ret = sys_chdir();
3175     break;
3176   case SYS_dup:
3177     ret = sys_dup();
3178     break;
3179   case SYS_getpid:
3180     ret = sys_getpid();
3181     break;
3182   case SYS_sbrk:
3183     ret = sys_sbrk();
3184     break;
3185   default:
3186     cprintf("unknown sys call %d\n", num);
3187     // Maybe kill the process?
3188     break;
3189   }
3190   cp->tf->eax = ret;
3191 }
3192
3193
3194
3195
3196
3197
3198
3199
```

```
3200 #include "types.h"
3201 #include "stat.h"
3202 #include "param.h"
3203 #include "mmu.h"
3204 #include "proc.h"
3205 #include "defs.h"
3206 #include "x86.h"
3207 #include "traps.h"
3208 #include "syscall.h"
3209 #include "spinlock.h"
3210 #include "buf.h"
3211 #include "fs.h"
3212 #include "fsvar.h"
3213 #include "elf.h"
3214 #include "file.h"
3215 #include "fcntl.h"
3216
3217 int
3218 sys_fork(void)
3219 {
3220   struct proc *np;
3221
3222   if((np = copyproc(curproc[cpu()])) == 0)
3223     return -1;
3224   np->state = RUNNABLE;
3225   return np->pid;
3226 }
3227
3228 int
3229 sys_exit(void)
3230 {
3231   proc_exit();
3232   return 0;  // not reached
3233 }
3234
3235 int
3236 sys_wait(void)
3237 {
3238   return proc_wait();
3239 }
3240
3241 int
3242 sys_kill(void)
3243 {
3244   int pid;
3245
3246   if(argint(0, &pid) < 0)
3247     return -1;
3248   return proc_kill(pid);
3249 }
```



```
3250 int
3251 sys_getpid(void)
3252 {
3253   return curproc[cpu()]->pid;
3254 }
3255
3256 int
3257 sys_sbrk(void)
3258 {
3259   int addr;
3260   int n;
3261   struct proc *cp = curproc[cpu()];
3262
3263   if(argint(0, &n) < 0)
3264     return -1;
3265   if((addr = growproc(n)) < 0)
3266     return -1;
3267   setupsegs(cp);
3268   return addr;
3269 }
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
```

```
3300 struct buf {
3301   int flags;
3302   uint dev;
3303   uint sector;
3304   struct buf *prev;
3305   struct buf *next;
3306   uchar data[512];
3307 };
3308 #define B_BUSY 0x1  // buffer is locked by some process
3309 #define B_VALID 0x2 // buffer contains the data of the sector
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
```



```
3350 struct devsw {
3351   int (*read)(int, char*, int);
3352   int (*write)(int, char*, int);
3353 };
3354
3355 extern struct devsw devsw[];
3356
3357 #define CONSOLE 1
3358
3359
3360
3361
3362
3363
3364
3365
3366
3367
3368
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3379
3380
3381
3382
3383
3384
3385
3386
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399
```

```
3400 #define O_CREATE  0x200
3401 #define O_RDONLY  0x000
3402 #define O_WRONLY  0x001
3403 #define O_RDWR    0x002
3404
3405
3406
3407
3408
3409
3410
3411
3412
3413
3414
3415
3416
3417
3418
3419
3420
3421
3422
3423
3424
3425
3426
3427
3428
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
```



```
3450 struct stat {
3451   int dev;     // Device number
3452   uint ino;    // Inode number on device
3453   short type;  // Type of file
3454   short nlink; // Number of links to file
3455   uint size;   // Size of file in bytes
3456 };
3457
3458
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3479
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
```

```
3500 struct file {
3501   enum { FD_CLOSED, FD_NONE, FD_PIPE, FD_FILE } type;
3502   int ref; // reference count
3503   char readable;
3504   char writable;
3505   struct pipe *pipe;
3506   struct inode *ip;
3507   uint off;
3508 };
3509
3510
3511
3512
3513
3514
3515
3516
3517
3518
3519
3520
3521
3522
3523
3524
3525
3526
3527
3528
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
```



```
3550 // On-disk file system format.
3551 // This header is shared between kernel and user space.
3552
3553 // Block 0 is unused.
3554 // Block 1 is super block.
3555 // Inodes start at block 2.
3556
3557 #define BSIZE 512  // block size
3558
3559 // File system super block
3560 struct superblock {
3561   uint size;         // Size of file system image (blocks)
3562   uint nblocks;      // Number of data blocks
3563   uint ninodes;      // Number of inodes.
3564 };
3565
3566 #define NADDRS (NDIRECT+1)
3567 #define NDIRECT 12
3568 #define INDIRECT 12
3569 #define NINDIRECT (BSIZE / sizeof(uint))
3570 #define MAXFILE (NDIRECT  + NINDIRECT)
3571
3572 // On-disk inode structure
3573 struct dinode {
3574   short type;           // File type
3575   short major;          // Major device number (T_DEV only)
3576   short minor;          // Minor device number (T_DEV only)
3577   short nlink;          // Number of links to inode in file system
3578   uint size;            // Size of file (bytes)
3579   uint addrs[NADDRS];   // Data block addresses
3580 };
3581
3582 #define T_DIR  1   // Directory
3583 #define T_FILE 2   // File
3584 #define T_DEV  3   // Special device
3585
3586 // Inodes per block.
3587 #define IPB           (BSIZE / sizeof(struct dinode))
3588
3589 // Block containing inode i
3590 #define IBLOCK(i)     ((i) / IPB + 2)
3591
3592 // Bitmap bits per block
3593 #define BPB           (BSIZE*8)
3594
3595 // Block containing bit for block b
3596 #define BBLOCK(b, ninodes) (b/BPB + (ninodes)/IPB + 3)
3597
3598
3599
```

```
3600 // Directory is a file containing a sequence of dirent structures.
3601 #define DIRSIZ 14
3602
3603 struct dirent {
3604   ushort inum;
3605   char name[DIRSIZ];
3606 };
3607
3608 extern uint rootdev;  // Device number of root file system
3609
3610
3611
3612
3613
3614
3615
3616
3617
3618
3619
3620
3621
3622
3623
3624
3625
3626
3627
3628
3629
3630
3631
3632
3633
3634
3635
3636
3637
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649
```



```
3650 // in-core file system types
3651
3652 struct inode {
3653   uint dev;           // Device number
3654   uint inum;          // Inode number
3655   int ref;            // Reference count
3656   int busy;           // Is the inode "locked"?
3657
3658   short type;         // copy of disk inode
3659   short major;
3660   short minor;
3661   short nlink;
3662   uint size;
3663   uint addrs[NADDRS];
3664 };
3665
3666 extern uint rootdev;
3667
3668 #define NAMEI_LOOKUP 1
3669 #define NAMEI_CREATE 2
3670 #define NAMEI_DELETE 3
3671
3672
3673
3674
3675
3676
3677
3678
3679
3680
3681
3682
3683
3684
3685
3686
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699
```

```
3700 // Simple PIO-based (non-DMA) IDE driver code.
3701
3702 #include "types.h"
3703 #include "param.h"
3704 #include "mmu.h"
3705 #include "proc.h"
3706 #include "defs.h"
3707 #include "x86.h"
3708 #include "traps.h"
3709 #include "spinlock.h"
3710
3711 #define IDE_BSY       0x80
3712 #define IDE_DRDY      0x40
3713 #define IDE_DF        0x20
3714 #define IDE_ERR       0x01
3715
3716 #define IDE_CMD_READ  0x20
3717 #define IDE_CMD_WRITE 0x30
3718
3719 // IDE request queue.
3720 // The next request will be stored in request[head],
3721 // and the request currently being served by the disk
3722 // is request[tail].
3723 // Must hold ide_lock while manipulating queue.
3724
3725 struct ide_request {
3726   int diskno;
3727   uint secno;
3728   void *addr;
3729   uint nsecs;
3730   uint read;
3731 };
3732
3733 static struct ide_request request[NREQUEST];
3734 static int head, tail;
3735 static struct spinlock ide_lock;
3736
3737 static int disk_1_present;
3738 static int disk_queue;
3739
3740 static int ide_probe_disk1(void);
3741
3742 // Wait for IDE disk to become ready.
3743 static int
3744 ide_wait_ready(int check_error)
3745 {
3746   int r;
3747
3748   while(((r = inb(0x1F7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
3749     ;
```



```
3750   if(check_error && (r & (IDE_DF|IDE_ERR)) != 0)
3751     return -1;
3752   return 0;
3753 }
3754
3755 void
3756 ide_init(void)
3757 {
3758   initlock(&ide_lock, "ide");
3759   irq_enable(IRQ_IDE);
3760   ioapic_enable(IRQ_IDE, ncpu - 1);
3761   ide_wait_ready(0);
3762   disk_1_present = ide_probe_disk1();
3763 }
3764
3765 // Probe to see if disk 1 exists (we assume disk 0 exists).
3766 static int
3767 ide_probe_disk1(void)
3768 {
3769   int r, x;
3770
3771   // wait for Device 0 to be ready
3772   ide_wait_ready(0);
3773
3774   // switch to Device 1
3775   outb(0x1F6, 0xE0 | (1<<4));
3776
3777   // check for Device 1 to be ready for a while
3778   for(x = 0; x < 1000 && (r = inb(0x1F7)) == 0; x++)
3779     ;
3780
3781   // switch back to Device 0
3782   outb(0x1F6, 0xE0 | (0<<4));
3783
3784   return x < 1000;
3785 }
3786
3787 // Interrupt handler - wake up the request that just finished.
3788 void
3789 ide_intr(void)
3790 {
3791   acquire(&ide_lock);
3792   wakeup(&request[tail]);
3793   release(&ide_lock);
3794 }
3795
3796
3797
3798
3799
```

```
3800 // Start the next request in the queue.
3801 static void
3802 ide_start_request (void)
3803 {
3804   struct ide_request *r;
3805
3806   if(head != tail) {
3807     r = &request[tail];
3808     ide_wait_ready(0);
3809     outb(0x3f6, 0);  // generate interrupt
3810     outb(0x1F2, r->nsecs);
3811     outb(0x1F3, r->secno & 0xFF);
3812     outb(0x1F4, (r->secno >> 8) & 0xFF);
3813     outb(0x1F5, (r->secno >> 16) & 0xFF);
3814     outb(0x1F6, 0xE0 | ((r->diskno&1)<<4) | ((r->secno>>24)&0x0F));
3815     if(r->read)
3816       outb(0x1F7, IDE_CMD_READ);
3817     else {
3818       outb(0x1F7, IDE_CMD_WRITE);
3819       outsl(0x1F0, r->addr, 512/4);
3820     }
3821   }
3822 }
3823
3824 // Run an entire disk operation.
3825 void
3826 ide_rw(int diskno, uint secno, void *addr, uint nsecs, int read)
3827 {
3828   struct ide_request *r;
3829
3830   if(diskno && !disk_1_present)
3831     panic("ide disk 1 not present");
3832
3833   acquire(&ide_lock);
3834
3835   // Add request to queue.
3836   while((head + 1) % NREQUEST == tail)
3837     sleep(&disk_queue, &ide_lock);
3838
3839   r = &request[head];
3840   r->secno = secno;
3841   r->addr = addr;
3842   r->nsecs = nsecs;
3843   r->diskno = diskno;
3844   r->read = read;
3845   head = (head + 1) % NREQUEST;
3846
3847   // Start request if necessary.
3848   ide_start_request();
3849
```



```
3850   // Wait for request to finish.
3851   sleep(r, &ide_lock);
3852
3853   // Finish request.
3854   if(read){
3855     if(ide_wait_ready(1) >= 0)
3856       insl(0x1F0, addr, 512/4);
3857   }
3858
3859   // Remove request from queue.
3860   if((head + 1) % NREQUEST == tail)
3861     wakeup(&disk_queue);
3862   tail = (tail + 1) % NREQUEST;
3863
3864   // Start next request in queue, if any.
3865   ide_start_request();
3866
3867   release(&ide_lock);
3868 }
3869
3870 // Synchronous disk write.
3871 int
3872 ide_write(int diskno, uint secno, const void *src, uint nsecs)
3873 {
3874   int r;
3875
3876   if(nsecs > 256)
3877     panic("ide_write");
3878
3879   ide_wait_ready(0);
3880
3881   outb(0x1F2, nsecs);
3882   outb(0x1F3, secno & 0xFF);
3883   outb(0x1F4, (secno >> 8) & 0xFF);
3884   outb(0x1F5, (secno >> 16) & 0xFF);
3885   outb(0x1F6, 0xE0 | ((diskno&1)<<4) | ((secno>>24)&0x0F));
3886   outb(0x1F7, 0x30);    // CMD 0x30 means write sector
3887
3888   for(; nsecs > 0; nsecs--, src += 512) {
3889     if((r = ide_wait_ready(1)) < 0)
3890       return r;
3891     outsl(0x1F0, src, 512/4);
3892   }
3893
3894   return 0;
3895 }
3896
3897
3898
3899
```

```
3900 // Buffer cache.
3901 //
3902 // The buffer cache is a linked list of buf structures
3903 // holding cached copies of disk block contents.
3904 // Each buf has two state bits B_BUSY and B_VALID.
3905 // If B_BUSY is set, it means that some code is currently
3906 // editing buf, so other code is not allowed to look at it.
3907 // To wait for a buffer that is B_BUSY, sleep on buf.
3908 // (See bget below.)
3909 //
3910 // If B_VALID is set, it means that the memory contents
3911 // have been initialized by reading them off the disk.
3912 // (Conversely, if B_VALID is not set, the memory contents
3913 // of buf must be initialized, often by calling bread,
3914 // before being used.)
3915 //
3916 // After making changes to a buf's memory, call bwrite to flush
3917 // the changes out to disk, to keep the disk and memory copies
3918 // in sync.
3919 //
3920 // When finished with a buffer, call brelse to release the buffer
3921 // (i.e., clear B_BUSY), so that others can access it.
3922 //
3923 // Bufs that are not B_BUSY are fair game for reuse for other
3924 // disk blocks.  It is not allowed to use a buf after calling brelse.
3925
3926 #include "types.h"
3927 #include "param.h"
3928 #include "x86.h"
3929 #include "mmu.h"
3930 #include "proc.h"
3931 #include "defs.h"
3932 #include "spinlock.h"
3933 #include "buf.h"
3934
3935 struct buf buf[NBUF];
3936 struct spinlock buf_table_lock;
3937
3938 // Linked list of all buffers, through prev/next.
3939 // bufhead->next is most recently used.
3940 // bufhead->tail is least recently used.
3941 struct buf bufhead;
3942
3943 void
3944 binit(void)
3945 {
3946   struct buf *b;
3947
3948   initlock(&buf_table_lock, "buf_table");
3949
```



```
3950   // Create linked list of buffers
3951   bufhead.prev = &bufhead;
3952   bufhead.next = &bufhead;
3953   for(b = buf; b < buf+NBUF; b++){
3954     b->next = bufhead.next;
3955     b->prev = &bufhead;
3956     bufhead.next->prev = b;
3957     bufhead.next = b;
3958   }
3959 }
3960
3961 // Look through buffer cache for block n on device dev.
3962 // If not found, allocate fresh block.
3963 // In either case, return locked buffer.
3964 static struct buf*
3965 bget(uint dev, uint sector)
3966 {
3967   struct buf *b;
3968
3969   acquire(&buf_table_lock);
3970
3971   for(;;){
3972     for(b = bufhead.next; b != &bufhead; b = b->next)
3973       if((b->flags & (B_BUSY|B_VALID)) &&
3974          b->dev == dev && b->sector == sector)
3975         break;
3976
3977     if(b != &bufhead){
3978       if(b->flags & B_BUSY){
3979         sleep(buf, &buf_table_lock);
3980       } else {
3981         b->flags |= B_BUSY;
3982         // b->flags &= ~B_VALID; // Force reread from disk
3983         release(&buf_table_lock);
3984         return b;
3985       }
3986     } else {
3987       for(b = bufhead.prev; b != &bufhead; b = b->prev){
3988         if((b->flags & B_BUSY) == 0){
3989           b->flags = B_BUSY;
3990           b->dev = dev;
3991           b->sector = sector;
3992           release(&buf_table_lock);
3993           return b;
3994         }
3995       }
3996       panic("bget: no buffers");
3997     }
3998   }
3999 }
```

```
4000 // Read buf's contents from disk.
4001 struct buf*
4002 bread(uint dev, uint sector)
4003 {
4004   struct buf *b;
4005
4006   b = bget(dev, sector);
4007   if(b->flags & B_VALID)
4008     return b;
4009
4010   ide_rw(dev & 0xff, sector, b->data, 1, 1);
4011   b->flags |= B_VALID;
4012
4013   return b;
4014 }
4015
4016 // Write buf's contents to disk.
4017 // Must be locked.
4018 void
4019 bwrite(struct buf *b, uint sector)
4020 {
4021   if((b->flags & B_BUSY) == 0)
4022     panic("bwrite");
4023
4024   ide_rw(b->dev & 0xff, sector, b->data, 1, 0);
4025   b->flags |= B_VALID;
4026 }
4027
4028 // Release the buffer buf.
4029 void
4030 brelse(struct buf *b)
4031 {
4032   if((b->flags & B_BUSY) == 0)
4033     panic("brelse");
4034
4035   acquire(&buf_table_lock);
4036
4037   b->next->prev = b->prev;
4038   b->prev->next = b->next;
4039   b->next = bufhead.next;
4040   b->prev = &bufhead;
4041   bufhead.next->prev = b;
4042   bufhead.next = b;
4043
4044   b->flags &= ~B_BUSY;
4045   wakeup(buf);
4046
4047   release(&buf_table_lock);
4048 }
4049
```



```
4050 #include "types.h"
4051 #include "stat.h"
4052 #include "param.h"
4053 #include "x86.h"
4054 #include "mmu.h"
4055 #include "proc.h"
4056 #include "defs.h"
4057 #include "spinlock.h"
4058 #include "buf.h"
4059 #include "fs.h"
4060 #include "fsvar.h"
4061 #include "dev.h"
4062
4063 // Inode table.  The inode table is an in-memory cache of the
4064 // on-disk inode structures.  If an inode in the table has a non-zero
4065 // reference count, then some open files refer to it and it must stay
4066 // in memory.  If an inode has a zero reference count, it is only in
4067 // memory as a cache in hopes of being used again (avoiding a disk read).
4068 // Any inode with reference count zero can be evicted from the table.
4069 //
4070 // In addition to having a reference count, inodes can be marked busy
4071 // (just like bufs), meaning that some code has logically locked the
4072 // inode, and others are not allowed to look at it.
4073 // This locking can last for a long
4074 // time (for example, if the inode is busy during a disk access),
4075 // so we don't use spin locks.  Instead, if a process wants to use
4076 // a particular inode, it must sleep(ip) to wait for it to be not busy.
4077 // See iget below.
4078 struct inode inode[NINODE];
4079 struct spinlock inode_table_lock;
4080
4081 uint rootdev = 1;
4082
4083 void
4084 iinit(void)
4085 {
4086   initlock(&inode_table_lock, "inode_table");
4087 }
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
```

```
4100 // Allocate a disk block.
4101 static uint
4102 balloc(uint dev)
4103 {
4104   int b;
4105   struct buf *bp;
4106   struct superblock *sb;
4107   int bi = 0;
4108   int size;
4109   int ninodes;
4110   uchar m;
4111
4112   bp = bread(dev, 1);
4113   sb = (struct superblock*) bp->data;
4114   size = sb->size;
4115   ninodes = sb->ninodes;
4116
4117   for(b = 0; b < size; b++) {
4118     if(b % BPB == 0) {
4119       brelse(bp);
4120       bp = bread(dev, BBLOCK(b, ninodes));
4121     }
4122     bi = b % BPB;
4123     m = 0x1 << (bi % 8);
4124     if((bp->data[bi/8] & m) == 0) {  // is block free?
4125       break;
4126     }
4127   }
4128   if(b >= size)
4129     panic("balloc: out of blocks");
4130
4131   bp->data[bi/8] |= 0x1 << (bi % 8);
4132   bwrite(bp, BBLOCK(b, ninodes));  // mark it allocated on disk
4133   brelse(bp);
4134   return b;
4135 }
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```



```
4150 // Free a disk block.
4151 static void
4152 bfree(int dev, uint b)
4153 {
4154   struct buf *bp;
4155   struct superblock *sb;
4156   int bi;
4157   int ninodes;
4158   uchar m;
4159
4160   bp = bread(dev, 1);
4161   sb = (struct superblock*) bp->data;
4162   ninodes = sb->ninodes;
4163   brelse(bp);
4164
4165   bp = bread(dev, b);
4166   memset(bp->data, 0, BSIZE);
4167   bwrite(bp, b);
4168   brelse(bp);
4169
4170   bp = bread(dev, BBLOCK(b, ninodes));
4171   bi = b % BPB;
4172   m = ~(0x1 << (bi %8));
4173   bp->data[bi/8] &= m;
4174   bwrite(bp, BBLOCK(b, ninodes));  // mark it free on disk
4175   brelse(bp);
4176 }
4177
4178 // Find the inode with number inum on device dev
4179 // and return an in-memory copy.  Loads the inode
4180 // from disk into the in-core table if necessary.
4181 // The returned inode has busy set and has its ref count incremented.
4182 // Caller must iput the return value when done with it.
4183 struct inode*
4184 iget(uint dev, uint inum)
4185 {
4186   struct inode *ip, *nip;
4187   struct dinode *dip;
4188   struct buf *bp;
4189
4190   acquire(&inode_table_lock);
4191
4192  loop:
4193   nip = 0;
4194   for(ip = &inode[0]; ip < &inode[NINODE]; ip++){
4195     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
4196       if(ip->busy){
4197         sleep(ip, &inode_table_lock);
4198         // Since we droped inode_table_lock, ip might have been reused
4199         // for some other inode entirely.  Must start the scan over,
```

```
4200        // and hopefully this time we will find the inode we want
4201        // and it will not be busy.
4202        goto loop;
4203      }
4204      ip->ref++;
4205      ip->busy = 1;
4206      release(&inode_table_lock);
4207      return ip;
4208    }
4209    if(nip == 0 && ip->ref == 0)
4210      nip = ip;
4211  }
4212
4213  if(nip == 0)
4214    panic("out of inodes");
4215
4216  nip->dev = dev;
4217  nip->inum = inum;
4218  nip->ref = 1;
4219  nip->busy = 1;
4220
4221  release(&inode_table_lock);
4222
4223  bp = bread(dev, IBLOCK(inum));
4224  dip = &((struct dinode*)(bp->data))[inum % IPB];
4225  nip->type = dip->type;
4226  nip->major = dip->major;
4227  nip->minor = dip->minor;
4228  nip->nlink = dip->nlink;
4229  nip->size = dip->size;
4230  memmove(nip->addrs, dip->addrs, sizeof(nip->addrs));
4231  brelse(bp);
4232
4233  return nip;
4234 }
4235
4236
4237
4238
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
```



```
4250 // Copy inode in memory, which has changed, to disk.
4251 // Caller must have locked ip.
4252 void
4253 iupdate(struct inode *ip)
4254 {
4255   struct buf *bp;
4256   struct dinode *dip;
4257
4258   bp = bread(ip->dev, IBLOCK(ip->inum));
4259   dip = &((struct dinode*)(bp->data))[ip->inum % IPB];
4260   dip->type = ip->type;
4261   dip->major = ip->major;
4262   dip->minor = ip->minor;
4263   dip->nlink = ip->nlink;
4264   dip->size = ip->size;
4265   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
4266   bwrite(bp, IBLOCK(ip->inum));   // mark it allocated on the disk
4267   brelse(bp);
4268 }
4269
4270 // Allocate a new inode with the given type
4271 // from the file system on device dev.
4272 struct inode*
4273 ialloc(uint dev, short type)
4274 {
4275   struct inode *ip;
4276   struct dinode *dip = 0;
4277   struct superblock *sb;
4278   int ninodes;
4279   int inum;
4280   struct buf *bp;
4281
4282   bp = bread(dev, 1);
4283   sb = (struct superblock*) bp->data;
4284   ninodes = sb->ninodes;
4285   brelse(bp);
4286
4287   for(inum = 1; inum < ninodes; inum++) {  // loop over inode blocks
4288     bp = bread(dev, IBLOCK(inum));
4289     dip = &((struct dinode*)(bp->data))[inum % IPB];
4290     if(dip->type == 0) {  // a free inode
4291       break;
4292     }
4293     brelse(bp);
4294   }
4295
4296   if(inum >= ninodes)
4297     panic("ialloc: no inodes left");
4298
4299
```

```
4300   memset(dip, 0, sizeof(*dip));
4301   dip->type = type;
4302   bwrite(bp, IBLOCK(inum));   // mark it allocated on the disk
4303   brelse(bp);
4304   ip = iget(dev, inum);
4305   return ip;
4306 }
4307
4308 // Free the given inode from its file system.
4309 static void
4310 ifree(struct inode *ip)
4311 {
4312   ip->type = 0;
4313   iupdate(ip);
4314 }
4315
4316 // Lock the given inode (wait for it to be not busy,
4317 // and then ip->busy).
4318 // Caller must already hold a reference to ip.
4319 // Otherwise, if all the references to ip go away,
4320 // it might be reused underfoot.
4321 void
4322 ilock(struct inode *ip)
4323 {
4324   if(ip->ref < 1)
4325     panic("ilock");
4326
4327   acquire(&inode_table_lock);
4328
4329   while(ip->busy)
4330     sleep(ip, &inode_table_lock);
4331   ip->busy = 1;
4332
4333   release(&inode_table_lock);
4334 }
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
```



```
4350 // Caller holds reference to ip and has locked it.
4351 // Caller no longer needs to examine / change it.
4352 // Unlock it, but keep the reference.
4353 void
4354 iunlock(struct inode *ip)
4355 {
4356   if(ip->busy != 1 || ip->ref < 1)
4357     panic("iunlock");
4358
4359   acquire(&inode_table_lock);
4360
4361   ip->busy = 0;
4362   wakeup(ip);
4363
4364   release(&inode_table_lock);
4365 }
4366
4367 // Return the disk block address of the nth block in inode ip.
4368 uint
4369 bmap(struct inode *ip, uint bn)
4370 {
4371   unsigned x;
4372   uint *a;
4373   struct buf *inbp;
4374
4375   if(bn >= MAXFILE)
4376     panic("bmap 1");
4377   if(bn < NDIRECT) {
4378     x = ip->addrs[bn];
4379     if(x == 0)
4380       panic("bmap 2");
4381   } else {
4382     if(ip->addrs[INDIRECT] == 0)
4383       panic("bmap 3");
4384     inbp = bread(ip->dev, ip->addrs[INDIRECT]);
4385     a = (uint*) inbp->data;
4386     x = a[bn - NDIRECT];
4387     brelse(inbp);
4388     if(x == 0)
4389       panic("bmap 4");
4390   }
4391   return x;
4392 }
4393
4394
4395
4396
4397
4398
4399
```

```
4400 // Truncate the inode ip, discarding all its data blocks.
4401 void
4402 itrunc(struct inode *ip)
4403 {
4404   int i, j;
4405   struct buf *inbp;
4406
4407   for(i = 0; i < NADDRS; i++) {
4408     if(ip->addrs[i] != 0) {
4409       if(i == INDIRECT) {
4410         inbp = bread(ip->dev, ip->addrs[INDIRECT]);
4411         uint *a = (uint*) inbp->data;
4412         for(j = 0; j < NINDIRECT; j++) {
4413           if(a[j] != 0) {
4414             bfree(ip->dev, a[j]);
4415             a[j] = 0;
4416           }
4417         }
4418         brelse(inbp);
4419       }
4420       bfree(ip->dev, ip->addrs[i]);
4421       ip->addrs[i] = 0;
4422     }
4423   }
4424   ip->size = 0;
4425   iupdate(ip);
4426 }
4427
4428 // Caller holds reference to ip and has locked it,
4429 // possibly editing it.
4430 // Release lock and drop the reference.
4431 void
4432 iput(struct inode *ip)
4433 {
4434   if(ip->ref < 1 || ip->busy != 1)
4435     panic("iput");
4436
4437   if((ip->ref == 1) && (ip->nlink == 0)) {
4438     itrunc(ip);
4439     ifree(ip);
4440   }
4441
4442   acquire(&inode_table_lock);
4443
4444   ip->ref -= 1;
4445   ip->busy = 0;
4446   wakeup(ip);
4447
4448   release(&inode_table_lock);
4449 }
```



```
4450 // Caller holds reference to ip but not lock.
4451 // Drop reference.
4452 void
4453 idecref(struct inode *ip)
4454 {
4455   ilock(ip);
4456   iput(ip);
4457 }
4458
4459 // Increment reference count for ip.
4460 void
4461 iincref(struct inode *ip)
4462 {
4463   ilock(ip);
4464   ip->ref++;
4465   iunlock(ip);
4466 }
4467
4468 // Copy stat information from inode.
4469 void
4470 stati(struct inode *ip, struct stat *st)
4471 {
4472   st->dev = ip->dev;
4473   st->ino = ip->inum;
4474   st->type = ip->type;
4475   st->nlink = ip->nlink;
4476   st->size = ip->size;
4477 }
4478
4479 #define min(a, b) ((a) < (b) ? (a) : (b))
4480
4481 // Read data from inode.
4482 int
4483 readi(struct inode *ip, char *dst, uint off, uint n)
4484 {
4485   uint target = n, n1;
4486   struct buf *bp;
4487
4488   if(ip->type == T_DEV) {
4489     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
4490       return -1;
4491     return devsw[ip->major].read(ip->minor, dst, n);
4492   }
4493
4494   while(n > 0 && off < ip->size){
4495     bp = bread(ip->dev, bmap(ip, off / BSIZE));
4496     n1 = min(n, ip->size - off);
4497     n1 = min(n1, BSIZE - (off % BSIZE));
4498     memmove(dst, bp->data + (off % BSIZE), n1);
4499     n -= n1;
```

```
4500     off += n1;
4501     dst += n1;
4502     brelse(bp);
4503   }
4504
4505   return target - n;
4506 }
4507
4508 // Allocate the nth block in inode ip if necessary.
4509 static int
4510 newblock(struct inode *ip, uint lbn)
4511 {
4512   struct buf *inbp;
4513   uint *inaddrs;
4514   uint b;
4515
4516   if(lbn < NDIRECT) {
4517     if(ip->addrs[lbn] == 0) {
4518       b = balloc(ip->dev);
4519       if(b <= 0)
4520         return -1;
4521       ip->addrs[lbn] = b;
4522     }
4523   } else {
4524     if(ip->addrs[INDIRECT] == 0) {
4525       b = balloc(ip->dev);
4526       if(b <= 0)
4527         return -1;
4528       ip->addrs[INDIRECT] = b;
4529     }
4530     inbp = bread(ip->dev, ip->addrs[INDIRECT]);
4531     inaddrs = (uint*) inbp->data;
4532     if(inaddrs[lbn - NDIRECT] == 0) {
4533       b = balloc(ip->dev);
4534       if(b <= 0)
4535         return -1;
4536       inaddrs[lbn - NDIRECT] = b;
4537       bwrite(inbp, ip->addrs[INDIRECT]);
4538     }
4539     brelse(inbp);
4540   }
4541   return 0;
4542 }
4543
4544
4545
4546
4547
4548
4549
```



```
4550 // Write data to inode.
4551 int
4552 writei(struct inode *ip, char *addr, uint off, uint n)
4553 {
4554   if(ip->type == T_DEV) {
4555     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
4556       return -1;
4557     return devsw[ip->major].write(ip->minor, addr, n);
4558   } else if(ip->type == T_FILE || ip->type == T_DIR) {
4559     struct buf *bp;
4560     int r = 0;
4561     int m;
4562     int lbn;
4563     while(r < n) {
4564       lbn = off / BSIZE;
4565       if(lbn >= MAXFILE)
4566         return r;
4567       if(newblock(ip, lbn) < 0) {
4568         cprintf("newblock failed\n");
4569         return r;
4570       }
4571       m = min(BSIZE - off % BSIZE, n-r);
4572       bp = bread(ip->dev, bmap(ip, lbn));
4573       memmove(bp->data + off % BSIZE, addr, m);
4574       bwrite(bp, bmap(ip, lbn));
4575       brelse(bp);
4576       r += m;
4577       off += m;
4578     }
4579     if(r > 0) {
4580       if(off > ip->size) {
4581         if(ip->type == T_DIR)
4582           ip->size = ((off / BSIZE) + 1) * BSIZE;
4583         else
4584           ip->size = off;
4585       }
4586       iupdate(ip);
4587     }
4588     return r;
4589   } else {
4590     panic("writei: unknown type");
4591     return 0;
4592   }
4593 }
4594
4595
4596
4597
4598
4599
```

```
4600 // look up a path name, in one of three modes.
4601 // NAMEI_LOOKUP: return locked target inode.
4602 // NAMEI_CREATE: return locked parent inode.
4603 //   return 0 if name does exist.
4604 //   *ret_last points to last path component (i.e. new file name).
4605 //   *ret_ip points to the the name that did exist, if it did.
4606 //   *ret_ip and *ret_last may be zero even if return value is zero.
4607 // NAMEI_DELETE: return locked parent inode, offset of dirent in *ret_off.
4608 //   return 0 if name doesn't exist.
4609 struct inode*
4610 namei(char *path, int mode, uint *ret_off,
4611       char **ret_last, struct inode **ret_ip)
4612 {
4613   struct inode *dp;
4614   struct proc *p = curproc[cpu()];
4615   char *cp = path, *cp1;
4616   uint off, dev;
4617   struct buf *bp;
4618   struct dirent *ep;
4619   int i, l, atend;
4620   uint ninum;
4621
4622   if(ret_off)
4623     *ret_off = 0xffffffff;
4624   if(ret_last)
4625     *ret_last = 0;
4626   if(ret_ip)
4627     *ret_ip = 0;
4628
4629   if(*cp == '/')
4630     dp = iget(rootdev, 1);
4631   else {
4632     dp = p->cwd;
4633     iincref(dp);
4634     ilock(dp);
4635   }
4636
4637   for(;;){
4638     while(*cp == '/')
4639       cp++;
4640
4641     if(*cp == '\0'){
4642       if(mode == NAMEI_LOOKUP)
4643         return dp;
4644       if(mode == NAMEI_CREATE && ret_ip){
4645         *ret_ip = dp;
4646         return 0;
4647       }
4648       iput(dp);
4649       return 0;
```



```
4650     }
4651
4652     if(dp->type != T_DIR){
4653       iput(dp);
4654       return 0;
4655     }
4656
4657     for(i = 0; cp[i] != 0 && cp[i] != '/'; i++)
4658       ;
4659     l = i;
4660     if(i > DIRSIZ)
4661       l = DIRSIZ;
4662
4663     for(off = 0; off < dp->size; off += BSIZE){
4664       bp = bread(dp->dev, bmap(dp, off / BSIZE));
4665       for(ep = (struct dirent*) bp->data;
4666           ep < (struct dirent*) (bp->data + BSIZE);
4667           ep++){
4668         if(ep->inum == 0)
4669           continue;
4670         if(memcmp(cp, ep->name, l) == 0 &&
4671            (l == DIRSIZ || ep->name[l]== 0)){
4672           // entry matches path element
4673           off += (uchar*)ep - bp->data;
4674           ninum = ep->inum;
4675           brelse(bp);
4676           cp += i;
4677           goto found;
4678         }
4679       }
4680       brelse(bp);
4681     }
4682     atend = 1;
4683     for(cp1 = cp; *cp1; cp1++)
4684       if(*cp1 == '/')
4685         atend = 0;
4686     if(mode == NAMEI_CREATE && atend){
4687       if(*cp == '\0'){
4688         iput(dp);
4689         return 0;
4690       }
4691       *ret_last = cp;
4692       return dp;
4693     }
4694
4695     iput(dp);
4696     return 0;
4697
4698
4699
```

```
4700   found:
4701     if(mode == NAMEI_DELETE && *cp == '\0'){
4702       *ret_off = off;
4703       return dp;
4704     }
4705     dev = dp->dev;
4706     iput(dp);
4707     dp = iget(dev, inum);
4708     if(dp->type == 0 || dp->nlink < 1)
4709       panic("namei");
4710   }
4711 }
4712
4713 // Write a new directory entry (name, ino) into the directory dp.
4714 // Caller must have locked dp.
4715 void
4716 wdir(struct inode *dp, char *name, uint ino)
4717 {
4718   uint off;
4719   struct dirent de;
4720   int i;
4721
4722   for(off = 0; off < dp->size; off += sizeof(de)){
4723     if(readi(dp, (char*) &de, off, sizeof(de)) != sizeof(de))
4724       panic("wdir read");
4725     if(de.inum == 0)
4726       break;
4727   }
4728
4729   de.inum = ino;
4730   for(i = 0; i < DIRSIZ && name[i]; i++)
4731     de.name[i] = name[i];
4732   for( ; i < DIRSIZ; i++)
4733     de.name[i] = '\0';
4734
4735   if(writei(dp, (char*) &de, off, sizeof(de)) != sizeof(de))
4736     panic("wdir write");
4737 }
4738
4739
4740
4741
4742
4743
4744
4745
4746
4747
4748
4749
```



```
4750 // Create the path cp and return its locked inode structure.
4751 // If cp already exists, return 0.
4752 struct inode*
4753 mknod(char *cp, short type, short major, short minor)
4754 {
4755   struct inode *ip, *dp;
4756   char *last;
4757
4758   if((dp = namei(cp, NAMEI_CREATE, 0, &last, 0)) == 0)
4759     return 0;
4760
4761   ip = mknod1(dp, last, type, major, minor);
4762
4763   iput(dp);
4764
4765   return ip;
4766 }
4767
4768 // Create a new inode named name inside dp
4769 // and return its locked inode structure.
4770 // If name already exists, return 0.
4771 struct inode*
4772 mknod1(struct inode *dp, char *name, short type, short major, short minor)
4773 {
4774   struct inode *ip;
4775
4776   ip = ialloc(dp->dev, type);
4777   if(ip == 0)
4778     return 0;
4779   ip->major = major;
4780   ip->minor = minor;
4781   ip->size = 0;
4782   ip->nlink = 1;
4783
4784   iupdate(ip);  // write new inode to disk
4785
4786   wdir(dp, name, ip->inum);
4787
4788   return ip;
4789 }
4790
4791
4792
4793
4794
4795
4796
4797
4798
4799
```

```
4800 // Unlink the inode named cp.
4801 int
4802 unlink(char *cp)
4803 {
4804   struct inode *ip, *dp;
4805   struct dirent de;
4806   uint off, inum, dev;
4807
4808   dp = namei(cp, NAMEI_DELETE, &off, 0, 0);
4809   if(dp == 0)
4810     return -1;
4811
4812   dev = dp->dev;
4813
4814   if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de) || de.inum == 0)
4815     panic("unlink no entry");
4816
4817   inum = de.inum;
4818
4819   memset(&de, 0, sizeof(de));
4820   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4821     panic("unlink dir write");
4822
4823   iupdate(dp);
4824   iput(dp);
4825
4826   ip = iget(dev, inum);
4827
4828   if(ip->nlink < 1)
4829     panic("unlink nlink < 1");
4830
4831   ip->nlink--;
4832
4833   iupdate(ip);
4834   iput(ip);
4835
4836   return 0;
4837 }
4838
4839
4840
4841
4842
4843
4844
4845
4846
4847
4848
4849
```



```
4850 // Create the path new as a link to the same inode as old.
4851 int
4852 link(char *name1, char *name2)
4853 {
4854   struct inode *ip, *dp;
4855   char *last;
4856
4857   if((ip = namei(name1, NAMEI_LOOKUP, 0, 0, 0)) == 0)
4858     return -1;
4859   if(ip->type == T_DIR){
4860     iput(ip);
4861     return -1;
4862   }
4863
4864   iunlock(ip);
4865
4866   if((dp = namei(name2, NAMEI_CREATE, 0, &last, 0)) == 0) {
4867     idecref(ip);
4868     return -1;
4869   }
4870   if(dp->dev != ip->dev){
4871     idecref(ip);
4872     iput(dp);
4873     return -1;
4874   }
4875
4876   ilock(ip);
4877   ip->nlink++;
4878   iupdate(ip);
4879
4880   wdir(dp, last, ip->inum);
4881   iput(dp);
4882   iput(ip);
4883
4884   return 0;
4885 }
4886
4887
4888
4889
4890
4891
4892
4893
4894
4895
4896
4897
4898
4899
```

```
4900 #include "types.h"
4901 #include "stat.h"
4902 #include "param.h"
4903 #include "x86.h"
4904 #include "mmu.h"
4905 #include "proc.h"
4906 #include "defs.h"
4907 #include "file.h"
4908 #include "spinlock.h"
4909 #include "dev.h"
4910 #include "fs.h"
4911 #include "fsvar.h"
4912
4913 struct spinlock file_table_lock;
4914 struct devsw devsw[NDEV];
4915
4916 struct file file[NFILE];
4917
4918 void
4919 fileinit(void)
4920 {
4921   initlock(&file_table_lock, "file_table");
4922 }
4923
4924 // Allocate a file structure
4925 struct file*
4926 filealloc(void)
4927 {
4928   int i;
4929
4930   acquire(&file_table_lock);
4931   for(i = 0; i < NFILE; i++){
4932     if(file[i].type == FD_CLOSED){
4933       file[i].type = FD_NONE;
4934       file[i].ref = 1;
4935       release(&file_table_lock);
4936       return file + i;
4937     }
4938   }
4939   release(&file_table_lock);
4940   return 0;
4941 }
4942
4943
4944
4945
4946
4947
4948
4949
```



```
4950 // Write to file f.  Addr is kernel address.
4951 int
4952 filewrite(struct file *f, char *addr, int n)
4953 {
4954   if(f->writable == 0)
4955     return -1;
4956   if(f->type == FD_PIPE){
4957     return pipe_write(f->pipe, addr, n);
4958   } else if(f->type == FD_FILE) {
4959     ilock(f->ip);
4960     int r = writei(f->ip, addr, f->off, n);
4961     if(r > 0) {
4962       f->off += r;
4963     }
4964     iunlock(f->ip);
4965     return r;
4966   } else {
4967     panic("filewrite");
4968     return -1;
4969   }
4970 }
4971
4972 // Read from file f.  Addr is kernel address.
4973 int
4974 fileread(struct file *f, char *addr, int n)
4975 {
4976   if(f->readable == 0)
4977     return -1;
4978   if(f->type == FD_PIPE){
4979     return pipe_read(f->pipe, addr, n);
4980   } else if(f->type == FD_FILE){
4981     ilock(f->ip);
4982     int cc = readi(f->ip, addr, f->off, n);
4983     if(cc > 0)
4984       f->off += cc;
4985     iunlock(f->ip);
4986     return cc;
4987   } else {
4988     panic("fileread");
4989     return -1;
4990   }
4991 }
4992
4993
4994
4995
4996
4997
4998
4999
```

```
5000 // Close file f.  (Decrement ref count, close when reaches 0.)
5001 void
5002 fileclose(struct file *f)
5003 {
5004   acquire(&file_table_lock);
5005
5006   if(f->ref < 1 || f->type == FD_CLOSED)
5007     panic("fileclose");
5008
5009   if(--f->ref == 0){
5010     struct file dummy = *f;
5011
5012     f->ref = 0;
5013     f->type = FD_CLOSED;
5014     release(&file_table_lock);
5015
5016     if(dummy.type == FD_PIPE){
5017       pipe_close(dummy.pipe, dummy.writable);
5018     } else if(dummy.type == FD_FILE){
5019       idecref(dummy.ip);
5020     } else {
5021       panic("fileclose");
5022     }
5023   } else {
5024     release(&file_table_lock);
5025   }
5026 }
5027
5028 // Get metadata about file f.
5029 int
5030 filestat(struct file *f, struct stat *st)
5031 {
5032   if(f->type == FD_FILE){
5033     ilock(f->ip);
5034     stati(f->ip, st);
5035     iunlock(f->ip);
5036     return 0;
5037   } else
5038     return -1;
5039 }
5040
5041
5042
5043
5044
5045
5046
5047
5048
5049
```



```
5050 // Increment ref count for file f.
5051 void
5052 fileincref(struct file *f)
5053 {
5054   acquire(&file_table_lock);
5055   if(f->ref < 1 || f->type == FD_CLOSED)
5056     panic("fileincref");
5057   f->ref++;
5058   release(&file_table_lock);
5059 }
5060
5061
5062
5063
5064
5065
5066
5067
5068
5069
5070
5071
5072
5073
5074
5075
5076
5077
5078
5079
5080
5081
5082
5083
5084
5085
5086
5087
5088
5089
5090
5091
5092
5093
5094
5095
5096
5097
5098
5099
```

```
5100 #include "types.h"
5101 #include "stat.h"
5102 #include "param.h"
5103 #include "mmu.h"
5104 #include "proc.h"
5105 #include "defs.h"
5106 #include "x86.h"
5107 #include "traps.h"
5108 #include "syscall.h"
5109 #include "spinlock.h"
5110 #include "buf.h"
5111 #include "fs.h"
5112 #include "fsvar.h"
5113 #include "elf.h"
5114 #include "file.h"
5115 #include "fcntl.h"
5116
5117 // Fetch the nth word-sized system call argument as a file descriptor
5118 // and return both the descriptor and the corresponding struct file.
5119 static int
5120 argfd(int argno, int *pfd, struct file **pf)
5121 {
5122   int fd;
5123   struct file *f;
5124   struct proc *p = curproc[cpu()];
5125
5126   if(argint(argno, &fd) < 0)
5127     return -1;
5128   if(fd < 0 || fd >= NOFILE || (f=p->ofile[fd]) == 0)
5129     return -1;
5130   if(pfd)
5131     *pfd = fd;
5132   if(pf)
5133     *pf = f;
5134   return 0;
5135 }
5136
5137
5138
5139
5140
5141
5142
5143
5144
5145
5146
5147
5148
5149
```



```
5150 // Allocate a file descriptor for the given file.
5151 // Takes over file reference from caller on success.
5152 static int
5153 fdalloc(struct file *f)
5154 {
5155   int fd;
5156   struct proc *p = curproc[cpu()];
5157   for(fd = 0; fd < NOFILE; fd++){
5158     if(p->ofile[fd] == 0){
5159       p->ofile[fd] = f;
5160       return fd;
5161     }
5162   }
5163   return -1;
5164 }
5165
5166 int
5167 sys_pipe(void)
5168 {
5169   int *fd;
5170   struct file *rf = 0, *wf = 0;
5171   int fd0, fd1;
5172   struct proc *p = curproc[cpu()];
5173
5174   if(argptr(0, (void*)&fd, 2*sizeof fd[0]) < 0)
5175     return -1;
5176   if(pipe_alloc(&rf, &wf) < 0)
5177     return -1;
5178   fd0 = -1;
5179   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
5180     if(fd0 >= 0)
5181       p->ofile[fd0] = 0;
5182     fileclose(rf);
5183     fileclose(wf);
5184     return -1;
5185   }
5186   fd[0] = fd0;
5187   fd[1] = fd1;
5188   return 0;
5189 }
5190
5191
5192
5193
5194
5195
5196
5197
5198
5199
```

```
5200 int
5201 sys_write(void)
5202 {
5203   struct file *f;
5204   int n;
5205   char *cp;
5206
5207   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &cp, n) < 0)
5208     return -1;
5209   return filewrite(f, cp, n);
5210 }
5211
5212 int
5213 sys_read(void)
5214 {
5215   struct file *f;
5216   int n;
5217   char *cp;
5218
5219   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &cp, n) < 0)
5220     return -1;
5221   return fileread(f, cp, n);
5222 }
5223
5224 int
5225 sys_close(void)
5226 {
5227   int fd;
5228   struct file *f;
5229
5230   if(argfd(0, &fd, &f) < 0)
5231     return -1;
5232   curproc[cpu()]->ofile[fd] = 0;
5233   fileclose(f);
5234   return 0;
5235 }
5236
5237
5238
5239
5240
5241
5242
5243
5244
5245
5246
5247
5248
5249
```



```
5250 int
5251 sys_open(void)
5252 {
5253   struct inode *ip, *dp;
5254   char *path;
5255   int omode;
5256   int fd;
5257   struct file *f;
5258   char *last;
5259
5260   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
5261     return -1;
5262
5263   if(omode & O_CREATE){
5264     dp = namei(path, NAMEI_CREATE, 0, &last, &ip);
5265     if(dp){
5266       ip = mknod1(dp, last, T_FILE, 0, 0);
5267       iput(dp);
5268       if(ip == 0)
5269         return -1;
5270     } else if(ip == 0){
5271       return -1;
5272     } else if(ip->type == T_DIR){
5273       iput(ip);
5274       return -1;
5275     }
5276   } else {
5277     ip = namei(path, NAMEI_LOOKUP, 0, 0, 0);
5278     if(ip == 0)
5279       return -1;
5280   }
5281   if(ip->type == T_DIR && ((omode & O_RDWR) || (omode & O_WRONLY))){
5282     iput(ip);
5283     return -1;
5284   }
5285
5286   if((f = filealloc()) == 0){
5287     iput(ip);
5288     return -1;
5289   }
5290   if((fd = fdalloc(f)) < 0){
5291     iput(ip);
5292     fileclose(f);
5293     return -1;
5294   }
5295
5296
5297
5298
5299
```

```
5300   iunlock(ip);
5301   f->type = FD_FILE;
5302   if(omode & O_RDWR) {
5303     f->readable = 1;
5304     f->writable = 1;
5305   } else if(omode & O_WRONLY) {
5306     f->readable = 0;
5307     f->writable = 1;
5308   } else {
5309     f->readable = 1;
5310     f->writable = 0;
5311   }
5312   f->ip = ip;
5313   f->off = 0;
5314
5315   return fd;
5316 }
5317
5318 int
5319 sys_mknod(void)
5320 {
5321   struct inode *nip;
5322   char *path;
5323   int len;
5324   int type, major, minor;
5325
5326   if((len=argstr(0, &path)) < 0 || argint(1, &type) < 0 ||
5327      argint(2, &major) < 0 || argint(3, &minor) < 0)
5328     return -1;
5329
5330   if(len >= DIRSIZ)
5331     return -1;
5332
5333   if((nip = mknod(path, type, major, minor)) == 0)
5334     return -1;
5335   iput(nip);
5336   return 0;
5337 }
5338
5339
5340
5341
5342
5343
5344
5345
5346
5347
5348
5349
```



```
5350 int
5351 sys_mkdir(void)
5352 {
5353   struct inode *nip;
5354   struct inode *dp;
5355   char *path;
5356   struct dirent de;
5357   char *last;
5358
5359   if(argstr(0, &path) < 0)
5360     return -1;
5361
5362   dp = namei(path, NAMEI_CREATE, 0, &last, 0);
5363   if(dp == 0)
5364     return -1;
5365
5366   nip = mknod1(dp, last, T_DIR, 0, 0);
5367   if(nip == 0){
5368     iput(dp);
5369     return -1;
5370   }
5371
5372   dp->nlink++;
5373   iupdate(dp);
5374
5375   memset(de.name, '\0', DIRSIZ);
5376   de.name[0] = '.';
5377   de.inum = nip->inum;
5378   writei(nip, (char*) &de, 0, sizeof(de));
5379
5380   de.inum = dp->inum;
5381   de.name[1] = '.';
5382   writei(nip, (char*) &de, sizeof(de), sizeof(de));
5383
5384   iput(dp);
5385   iput(nip);
5386
5387   return 0;
5388 }
5389
5390
5391
5392
5393
5394
5395
5396
5397
5398
5399
```

```
5400 int
5401 sys_chdir(void)
5402 {
5403   struct proc *p = curproc[cpu()];
5404   struct inode *ip;
5405   char *path;
5406
5407   if(argstr(0, &path) < 0)
5408     return -1;
5409
5410   if((ip = namei(path, NAMEI_LOOKUP, 0, 0, 0)) == 0)
5411     return -1;
5412
5413   if(ip == p->cwd) {
5414     iput(ip);
5415     return 0;
5416   }
5417
5418   if(ip->type != T_DIR) {
5419     iput(ip);
5420     return -1;
5421   }
5422
5423   idecref(p->cwd);
5424   p->cwd = ip;
5425   iunlock(p->cwd);
5426   return 0;
5427 }
5428
5429 int
5430 sys_unlink(void)
5431 {
5432   char *path;
5433
5434   if(argstr(0, &path) < 0)
5435     return -1;
5436   return unlink(path);
5437 }
5438
5439 int
5440 sys_fstat(void)
5441 {
5442   struct file *f;
5443   struct stat *st;
5444
5445   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof *st) < 0)
5446     return -1;
5447   return filestat(f, st);
5448 }
5449
```



```
5450 int
5451 sys_dup(void)
5452 {
5453   struct file *f;
5454   int fd;
5455
5456   if(argfd(0, 0, &f) < 0)
5457     return -1;
5458   if((fd=fdalloc(f)) < 0)
5459     return -1;
5460   fileincref(f);
5461   return fd;
5462 }
5463
5464 int
5465 sys_link(void)
5466 {
5467   char *old, *new;
5468
5469   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
5470     return -1;
5471   return link(old, new);
5472 }
5473
5474 int
5475 sys_exec(void)
5476 {
5477   struct proc *cp = curproc[cpu()];
5478   uint sz=0, ap, sp, p1, p2;
5479   int i, nargs, argbytes, len;
5480   struct inode *ip;
5481   struct elfhdr elf;
5482   struct proghdr ph;
5483   char *mem = 0;
5484   char *path, *s;
5485   uint argv;
5486
5487   if(argstr(0, &path) < 0 || argint(1, (int*)&argv) < 0)
5488     return -1;
5489
5490   ip = namei(path, NAMEI_LOOKUP, 0, 0, 0);
5491   if(ip == 0)
5492     return -1;
5493
5494   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
5495     goto bad;
5496
5497   if(elf.magic != ELF_MAGIC)
5498     goto bad;
5499
```

```
5500   sz = 0;
5501   for(i = 0; i < elf.phnum; i++){
5502     if(readi(ip, (char*)&ph, elf.phoff + i * sizeof(ph),
5503              sizeof(ph)) != sizeof(ph))
5504       goto bad;
5505     if(ph.type != ELF_PROG_LOAD)
5506       continue;
5507     if(ph.memsz < ph.filesz)
5508       goto bad;
5509     sz += ph.memsz;
5510   }
5511
5512   sz += 4096 - (sz % 4096);
5513   sz += 4096;
5514
5515   mem = kalloc(sz);
5516   if(mem == 0)
5517     goto bad;
5518   memset(mem, 0, sz);
5519
5520   nargs = 0;
5521   argbytes = 0;
5522   for(i = 0;; i++){
5523     if(fetchint(cp, argv + 4*i, (int*)&ap) < 0)
5524       goto bad;
5525     if(ap == 0)
5526       break;
5527     len = fetchstr(cp, ap, &s);
5528     if(len < 0)
5529       goto bad;
5530     nargs++;
5531     argbytes += len + 1;
5532   }
5533
5534   // argn\0
5535   // ...
5536   // arg0\0
5537   // 0
5538   // ptr to argn
5539   // ...
5540   // 12: ptr to arg0
5541   //  8: argv (points to ptr to arg0)
5542   //  4: argc
5543   //  0: fake return pc
5544   sp = sz - argbytes - (nargs+1)*4 - 4 - 4 - 4;
5545   *(uint*)(mem + sp) = 0xffffffff;
5546   *(uint*)(mem + sp + 4) = nargs;
5547   *(uint*)(mem + sp + 8) = (uint)(sp + 12);
5548
5549
```



```
5550   p1 = sp + 12;
5551   p2 = sp + 12 + (nargs + 1) * 4;
5552   for(i = 0; i < nargs; i++){
5553     fetchint(cp, argv + 4*i, (int*)&ap);
5554     len = fetchstr(cp, ap, &s);
5555     memmove(mem + p2, s, len + 1);
5556     *(uint*)(mem + p1) = p2;
5557     p1 += 4;
5558     p2 += len + 1;
5559   }
5560   *(uint*)(mem + p1) = 0;
5561
5562   // commit to the new image.
5563   kfree(cp->mem, cp->sz);
5564   cp->sz = sz;
5565   cp->mem = mem;
5566   mem = 0;
5567
5568   for(i = 0; i < elf.phnum; i++){
5569     if(readi(ip, (char*)&ph, elf.phoff + i * sizeof(ph),
5570              sizeof(ph)) != sizeof(ph))
5571       goto bad2;
5572     if(ph.type != ELF_PROG_LOAD)
5573       continue;
5574     if(ph.va + ph.memsz > sz)
5575       goto bad2;
5576     if(readi(ip, cp->mem + ph.va, ph.offset, ph.filesz) != ph.filesz)
5577       goto bad2;
5578     memset(cp->mem + ph.va + ph.filesz, 0, ph.memsz - ph.filesz);
5579   }
5580
5581   iput(ip);
5582
5583   cp->tf->eip = elf.entry;
5584   cp->tf->esp = sp;
5585   setupsegs(cp);
5586
5587   return 0;
5588
5589 bad:
5590   if(mem)
5591     kfree(mem, sz);
5592   iput(ip);
5593   return -1;
5594
5595 bad2:
5596   iput(ip);
5597   proc_exit();
5598   return 0;
5599 }
```

```
5600 #include "types.h"
5601 #include "param.h"
5602 #include "x86.h"
5603 #include "mmu.h"
5604 #include "proc.h"
5605 #include "defs.h"
5606 #include "file.h"
5607 #include "spinlock.h"
5608
5609 #define PIPESIZE 512
5610
5611 struct pipe {
5612   int readopen; // read fd is still open
5613   int writeopen; // write fd is still open
5614   int writep; // next index to write
5615   int readp;  // next index to read
5616   struct spinlock lock;
5617   char data[PIPESIZE];
5618 };
5619
5620 int
5621 pipe_alloc(struct file **f0, struct file **f1)
5622 {
5623   *f0 = *f1 = 0;
5624   struct pipe *p = 0;
5625
5626   if((*f0 = filealloc()) == 0)
5627     goto oops;
5628   if((*f1 = filealloc()) == 0)
5629     goto oops;
5630   if((p = (struct pipe*) kalloc(PAGE)) == 0)
5631     goto oops;
5632   p->readopen = 1;
5633   p->writeopen = 1;
5634   p->writep = 0;
5635   p->readp = 0;
5636   initlock(&p->lock, "pipe");
5637   (*f0)->type = FD_PIPE;
5638   (*f0)->readable = 1;
5639   (*f0)->writable = 0;
5640   (*f0)->pipe = p;
5641   (*f1)->type = FD_PIPE;
5642   (*f1)->readable = 0;
5643   (*f1)->writable = 1;
5644   (*f1)->pipe = p;
5645   return 0;
5646  oops:
5647   if(p)
5648     kfree((char*) p, PAGE);
5649   if(*f0){
```



```
5650     (*f0)->type = FD_NONE;
5651     fileclose(*f0);
5652   }
5653   if(*f1){
5654     (*f1)->type = FD_NONE;
5655     fileclose(*f1);
5656   }
5657   return -1;
5658 }
5659
5660 void
5661 pipe_close(struct pipe *p, int writable)
5662 {
5663   acquire(&p->lock);
5664
5665   if(writable){
5666     p->writeopen = 0;
5667     wakeup(&p->readp);
5668   } else {
5669     p->readopen = 0;
5670     wakeup(&p->writep);
5671   }
5672
5673   release(&p->lock);
5674
5675   if(p->readopen == 0 && p->writeopen == 0)
5676     kfree((char*) p, PAGE);
5677 }
5678
5679 int
5680 pipe_write(struct pipe *p, char *addr, int n)
5681 {
5682   int i;
5683
5684   acquire(&p->lock);
5685
5686   for(i = 0; i < n; i++){
5687     while(((p->writep + 1) % PIPESIZE) == p->readp){
5688       if(p->readopen == 0){
5689         release(&p->lock);
5690         return -1;
5691       }
5692       wakeup(&p->readp);
5693       sleep(&p->writep, &p->lock);
5694     }
5695     p->data[p->writep] = addr[i];
5696     p->writep = (p->writep + 1) % PIPESIZE;
5697   }
5698
5699
```

```
5700   release(&p->lock);
5701   wakeup(&p->readp);
5702   return i;
5703 }
5704
5705 int
5706 pipe_read(struct pipe *p, char *addr, int n)
5707 {
5708   int i;
5709
5710   acquire(&p->lock);
5711
5712   while(p->readp == p->writep){
5713     if(p->writeopen == 0){
5714       release(&p->lock);
5715       return 0;
5716     }
5717     sleep(&p->readp, &p->lock);
5718   }
5719
5720   for(i = 0; i < n; i++){
5721     if(p->readp == p->writep)
5722       break;
5723     addr[i] = p->data[p->readp];
5724     p->readp = (p->readp + 1) % PIPESIZE;
5725   }
5726
5727   release(&p->lock);
5728   wakeup(&p->writep);
5729   return i;
5730 }
5731
5732
5733
5734
5735
5736
5737
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749
```



```
5750 #include "types.h"
5751 #include "defs.h"
5752
5753 void*
5754 memset(void *dst, int c, uint n)
5755 {
5756   char *d = (char*) dst;
5757
5758   while(n-- > 0)
5759     *d++ = c;
5760
5761   return dst;
5762 }
5763
5764 int
5765 memcmp(const void *v1, const void *v2, uint n)
5766 {
5767   const uchar *s1 = (const uchar*) v1;
5768   const uchar *s2 = (const uchar*) v2;
5769
5770   while(n-- > 0) {
5771     if(*s1 != *s2)
5772       return (int) *s1 - (int) *s2;
5773     s1++, s2++;
5774   }
5775
5776   return 0;
5777 }
5778
5779 void*
5780 memmove(void *dst, const void *src, uint n)
5781 {
5782   const char *s;
5783   char *d;
5784
5785   s = src;
5786   d = dst;
5787   if(s < d && s + n > d) {
5788     s += n;
5789     d += n;
5790     while(n-- > 0)
5791       *--d = *--s;
5792   } else
5793     while(n-- > 0)
5794       *d++ = *s++;
5795
5796   return dst;
5797 }
5798
5799
```

```
5800 int
5801 strncmp(const char *p, const char *q, uint n)
5802 {
5803   while(n > 0 && *p && *p == *q)
5804     n--, p++, q++;
5805   if(n == 0)
5806     return 0;
5807   else
5808     return (int) ((uchar) *p - (uchar) *q);
5809 }
5810
5811
5812
5813
5814
5815
5816
5817
5818
5819
5820
5821
5822
5823
5824
5825
5826
5827
5828
5829
5830
5831
5832
5833
5834
5835
5836
5837
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```



```
5850 #define IO_APIC_BASE   0xFEC00000   // Default phys addr of IO APIC
5851 #define IOAPIC_WINDOW        0x10   // Window register offset
5852
5853 // Constants relating to APIC ID registers
5854 #define APIC_ID_MASK            0xff000000
5855 #define APIC_ID_SHIFT           24
5856 #define APIC_ID_CLUSTER         0xf0
5857 #define APIC_ID_CLUSTER_ID      0x0f
5858 #define APIC_MAX_CLUSTER        0xe
5859 #define APIC_MAX_INTRACLUSTER_ID 3
5860 #define APIC_ID_CLUSTER_SHIFT   4
5861
5862 // Fields in VER
5863 #define APIC_VER_VERSION        0x000000ff
5864 #define APIC_VER_MAXLVT         0x00ff0000
5865 #define MAXLVTSHIFT             16
5866
5867 // Indexes into IO APIC
5868 #define IOAPIC_ID               0x00
5869 #define IOAPIC_VER              0x01
5870 #define IOAPIC_ARB              0x02
5871 #define IOAPIC_REDTBL           0x10
5872 #define IOAPIC_REDTBL0          IOAPIC_REDTBL
5873 #define IOAPIC_REDTBL1          (IOAPIC_REDTBL+0x02)
5874 #define IOAPIC_REDTBL2          (IOAPIC_REDTBL+0x04)
5875 #define IOAPIC_REDTBL3          (IOAPIC_REDTBL+0x06)
5876 #define IOAPIC_REDTBL4          (IOAPIC_REDTBL+0x08)
5877 #define IOAPIC_REDTBL5          (IOAPIC_REDTBL+0x0a)
5878 #define IOAPIC_REDTBL6          (IOAPIC_REDTBL+0x0c)
5879 #define IOAPIC_REDTBL7          (IOAPIC_REDTBL+0x0e)
5880 #define IOAPIC_REDTBL8          (IOAPIC_REDTBL+0x10)
5881 #define IOAPIC_REDTBL9          (IOAPIC_REDTBL+0x12)
5882 #define IOAPIC_REDTBL10         (IOAPIC_REDTBL+0x14)
5883 #define IOAPIC_REDTBL11         (IOAPIC_REDTBL+0x16)
5884 #define IOAPIC_REDTBL12         (IOAPIC_REDTBL+0x18)
5885 #define IOAPIC_REDTBL13         (IOAPIC_REDTBL+0x1a)
5886 #define IOAPIC_REDTBL14         (IOAPIC_REDTBL+0x1c)
5887 #define IOAPIC_REDTBL15         (IOAPIC_REDTBL+0x1e)
5888 #define IOAPIC_REDTBL16         (IOAPIC_REDTBL+0x20)
5889 #define IOAPIC_REDTBL17         (IOAPIC_REDTBL+0x22)
5890 #define IOAPIC_REDTBL18         (IOAPIC_REDTBL+0x24)
5891 #define IOAPIC_REDTBL19         (IOAPIC_REDTBL+0x26)
5892 #define IOAPIC_REDTBL20         (IOAPIC_REDTBL+0x28)
5893 #define IOAPIC_REDTBL21         (IOAPIC_REDTBL+0x2a)
5894 #define IOAPIC_REDTBL22         (IOAPIC_REDTBL+0x2c)
5895 #define IOAPIC_REDTBL23         (IOAPIC_REDTBL+0x2e)
5896
5897 // Fields in the IO APIC's redirection table entries
5898 #define IOART_DEST      APIC_ID_MASK    // broadcast addr: all APICs
5899
```

```
#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
};
```

```
6000 enum {  // LAPIC_ICRLO
6001   // [14] IPI Trigger Mode Level (RW)
6002   LAPIC_DEASSERT = 0x00000000,  // Deassert level-sensitive interrupt
6003   LAPIC_ASSERT  = 0x00004000,   // Assert level-sensitive interrupt
6004
6005   // [17:16] Remote Read Status
6006   LAPIC_INVALID = 0x00000000,   // Invalid
6007   LAPIC_WAIT    = 0x00010000,   // In-Progress
6008   LAPIC_VALID   = 0x00020000,   // Valid
6009
6010   // [19:18] Destination Shorthand
6011   LAPIC_FIELD   = 0x00000000,   // No shorthand
6012   LAPIC_SELF    = 0x00040000,   // Self is single destination
6013   LAPIC_ALLINC  = 0x00080000,   // All including self
6014   LAPIC_ALLEXC  = 0x000C0000,   // All Excluding self
6015 };
6016
6017 enum {  // LAPIC_ESR
6018   LAPIC_SENDCS  = 0x00000001,     // Send CS Error
6019   LAPIC_RCVCS   = 0x00000002,     // Receive CS Error
6020   LAPIC_SENDACCEPT = 0x00000004,  // Send Accept Error
6021   LAPIC_RCVACCEPT = 0x00000008,   // Receive Accept Error
6022   LAPIC_SENDVECTOR = 0x00000020,  // Send Illegal Vector
6023   LAPIC_RCVVECTOR = 0x00000040,   // Receive Illegal Vector
6024   LAPIC_REGISTER = 0x00000080,    // Illegal Register Address
6025 };
6026
6027 enum {  // LAPIC_TIMER
6028   // [17] Timer Mode (RW)
6029   LAPIC_ONESHOT = 0x00000000,   // One-shot
6030   LAPIC_PERIODIC = 0x00020000,  // Periodic
6031
6032   // [19:18] Timer Base (RW)
6033   LAPIC_CLKIN   = 0x00000000,   // use CLKIN as input
6034   LAPIC_TMBASE  = 0x00040000,   // use TMBASE
6035   LAPIC_DIVIDER = 0x00080000,   // use output of the divider
6036 };
6037
6038 enum {  // LAPIC_TDCR
6039   LAPIC_X2 = 0x00000000,        // divide by 2
6040   LAPIC_X4 = 0x00000001,        // divide by 4
6041   LAPIC_X8 = 0x00000002,        // divide by 8
6042   LAPIC_X16 = 0x00000003,       // divide by 16
6043   LAPIC_X32 = 0x00000008,       // divide by 32
6044   LAPIC_X64 = 0x00000009,       // divide by 64
6045   LAPIC_X128 = 0x0000000A,      // divide by 128
6046   LAPIC_X1 = 0x0000000B,        // divide by 1
6047 };
6048
6049
```



```
6050 uint *lapicaddr;
6051
6052 static int
6053 lapic_read(int r)
6054 {
6055   return *(lapicaddr+(r/sizeof(*lapicaddr)));
6056 }
6057
6058 static void
6059 lapic_write(int r, int data)
6060 {
6061   *(lapicaddr+(r/sizeof(*lapicaddr))) = data;
6062 }
6063
6064
6065 void
6066 lapic_timerinit(void)
6067 {
6068   if(!lapicaddr)
6069     return;
6070
6071   lapic_write(LAPIC_TDCR, LAPIC_X1);
6072   lapic_write(LAPIC_TIMER, LAPIC_CLKIN | LAPIC_PERIODIC |
6073          (IRQ_OFFSET + IRQ_TIMER));
6074   lapic_write(LAPIC_TCCR, 10000000);
6075   lapic_write(LAPIC_TICR, 10000000);
6076 }
6077
6078 void
6079 lapic_timerintr(void)
6080 {
6081   if(lapicaddr)
6082     lapic_write(LAPIC_EOI, 0);
6083 }
6084
6085 void
6086 lapic_init(int c)
6087 {
6088   uint r, lvt;
6089
6090   if(!lapicaddr)
6091     return;
6092
6093   lapic_write(LAPIC_DFR, 0xFFFFFFFF);    // Set dst format register
6094   r = (lapic_read(LAPIC_ID)>>24) & 0xFF; // Read APIC ID
6095   lapic_write(LAPIC_LDR, (1<<r)<<24);    // Set logical dst register to r
6096   lapic_write(LAPIC_TPR, 0xFF);          // No interrupts for now
6097
6098   // Enable APIC
6099   lapic_write(LAPIC_SVR, LAPIC_ENABLE|(IRQ_OFFSET+IRQ_SPURIOUS));
```

```
6100   // In virtual wire mode, set up the LINT0 and LINT1 as follows:
6101   lapic_write(LAPIC_LINT0, APIC_IMASK | APIC_EXTINT);
6102   lapic_write(LAPIC_LINT1, APIC_IMASK | APIC_NMI);
6103
6104   lapic_write(LAPIC_EOI, 0); // Ack any outstanding interrupts.
6105
6106   lvt = (lapic_read(LAPIC_VER)>>16) & 0xFF;
6107   if(lvt >= 4)
6108     lapic_write(LAPIC_PCINT, APIC_IMASK);
6109   lapic_write(LAPIC_ERROR, IRQ_OFFSET+IRQ_ERROR);
6110   lapic_write(LAPIC_ESR, 0);
6111   lapic_read(LAPIC_ESR);
6112
6113   // Issue an INIT Level De-Assert to synchronise arbitration ID's.
6114   lapic_write(LAPIC_ICRHI, 0);
6115   lapic_write(LAPIC_ICRLO, LAPIC_ALLINC|APIC_LEVEL|
6116                            LAPIC_DEASSERT|APIC_INIT);
6117   while(lapic_read(LAPIC_ICRLO) & APIC_DELIVS)
6118     ;
6119 }
6120
6121 void
6122 lapic_enableintr(void)
6123 {
6124   if(lapicaddr)
6125     lapic_write(LAPIC_TPR, 0);
6126 }
6127
6128 void
6129 lapic_disableintr(void)
6130 {
6131   if(lapicaddr)
6132     lapic_write(LAPIC_TPR, 0xFF);
6133 }
6134
6135 void
6136 lapic_eoi(void)
6137 {
6138   if(lapicaddr)
6139     lapic_write(LAPIC_EOI, 0);
6140 }
6141
6142
6143
6144
6145
6146
6147
6148
6149
```



```
6150 int
6151 cpu(void)
6152 {
6153   int x;
6154   if(lapicaddr)
6155     x = (lapic_read(LAPIC_ID)>>24) & 0xFF;
6156   else
6157     x = 0;
6158   return x;
6159 }
6160
6161 void
6162 lapic_startap(uchar apicid, int v)
6163 {
6164   int crhi, i;
6165   volatile int j = 0;
6166
6167   crhi = apicid<<24;
6168   lapic_write(LAPIC_ICRHI, crhi);
6169   lapic_write(LAPIC_ICRLO, LAPIC_FIELD|APIC_LEVEL|
6170                            LAPIC_ASSERT|APIC_INIT);
6171
6172   while(j++ < 10000) {;}
6173   lapic_write(LAPIC_ICRLO, LAPIC_FIELD|APIC_LEVEL|
6174                            LAPIC_DEASSERT|APIC_INIT);
6175
6176   while(j++ < 1000000) {;}
6177
6178   // in p9 code, this was i < 2, which is what the spec says on page B-3
6179   for(i = 0; i < 1; i++){
6180     lapic_write(LAPIC_ICRHI, crhi);
6181     lapic_write(LAPIC_ICRLO, LAPIC_FIELD|APIC_EDGE|APIC_STARTUP|(v/4096));
6182     while(j++ < 100000) {;}
6183   }
6184 }
6185
6186
6187
6188
6189
6190
6191
6192
6193
6194
6195
6196
6197
6198
6199
```

```
6200 #include "types.h"
6201 #include "mp.h"
6202 #include "defs.h"
6203 #include "x86.h"
6204 #include "traps.h"
6205 #include "ioapic.h"
6206
6207 struct ioapic {
6208   uint ioregsel;  uint p01; uint p02; uint p03;
6209   uint iowin;     uint p11; uint p12; uint p13;
6210 };
6211
6212
6213 #define IOAPIC_REDTBL_LO(i)  (IOAPIC_REDTBL + (i) * 2)
6214 #define IOAPIC_REDTBL_HI(i)  (IOAPIC_REDTBL_LO(i) + 1)
6215
6216 static uint
6217 ioapic_read(struct ioapic *io, int reg)
6218 {
6219   io->ioregsel = reg;
6220   return io->iowin;
6221 }
6222
6223 static void
6224 ioapic_write(struct ioapic *io, int reg, uint val)
6225 {
6226   io->ioregsel = reg;
6227   io->iowin = val;
6228 }
6229
6230 void
6231 ioapic_init(void)
6232 {
6233   struct ioapic *io;
6234   uint l, h;
6235   int nintr;
6236   uchar id;
6237   int i;
6238
6239   if(!ismp)
6240     return;
6241
6242   io = (struct ioapic*) IO_APIC_BASE;
6243   l = ioapic_read(io, IOAPIC_VER);
6244   nintr =  ((l & IOART_VER_MAXREDIR) >> MAXREDIRSHIFT) + 1;
6245   id = ioapic_read(io, IOAPIC_ID) >> APIC_ID_SHIFT;
6246   if(id != ioapic_id)
6247     cprintf("ioapic_init: id isn't equal to ioapic_id; not a MP\n");
6248   for(i = 0; i < nintr; i++) {
6249     // active-hi and edge-triggered for ISA interrupts
```



```
6250     // Assume that pin 0 on the first I/O APIC is an ExtINT pin.
6251     // Assume that pins 1-15 are ISA interrupts
6252     l = ioapic_read(io, IOAPIC_REDTBL_LO(i));
6253     l = l & ~IOART_INTMASK;  // allow INTs
6254     l |= IOART_INTMSET;
6255     l = l & ~IOART_INTPOL;   // active hi
6256     l = l & ~IOART_TRGRMOD;  // edgee triggered
6257     l = l & ~IOART_DELMOD;   // fixed
6258     l = l & ~IOART_DESTMOD;  // physical mode
6259     l = l | (IRQ_OFFSET + i); // vector
6260     ioapic_write(io, IOAPIC_REDTBL_LO(i), l);
6261     h = ioapic_read(io, IOAPIC_REDTBL_HI(i));
6262     h &= ~IOART_DEST;
6263     ioapic_write(io, IOAPIC_REDTBL_HI(i), h);
6264   }
6265 }
6266
6267 void
6268 ioapic_enable (int irq, int cpunum)
6269 {
6270   uint l, h;
6271   struct ioapic *io;
6272
6273   if(!ismp)
6274     return;
6275
6276   io = (struct ioapic*) IO_APIC_BASE;
6277   l = ioapic_read(io, IOAPIC_REDTBL_LO(irq));
6278   l = l & ~IOART_INTMASK;  // allow INTs
6279   ioapic_write(io, IOAPIC_REDTBL_LO(irq), l);
6280   h = ioapic_read(io, IOAPIC_REDTBL_HI(irq));
6281   h &= ~IOART_DEST;
6282   h |= (cpunum << APIC_ID_SHIFT);
6283   ioapic_write(io, IOAPIC_REDTBL_HI(irq), h);
6284 }
6285
6286
6287
6288
6289
6290
6291
6292
6293
6294
6295
6296
6297
6298
6299
```

```
6300 #include "types.h"
6301 #include "x86.h"
6302 #include "traps.h"
6303 #include "defs.h"
6304
6305 // I/O Addresses of the two 8259A programmable interrupt controllers
6306 #define IO_PIC1         0x20    // Master (IRQs 0-7)
6307 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
6308
6309 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
6310
6311 // Current IRQ mask.
6312 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
6313 static ushort irq_mask_8259A = 0xFFFF & ~(1<<IRQ_SLAVE);
6314
6315 static void
6316 irq_setmask_8259A(ushort mask)
6317 {
6318   irq_mask_8259A = mask;
6319
6320   outb(IO_PIC1+1, (char)mask);
6321   outb(IO_PIC2+1, (char)(mask >> 8));
6322 }
6323
6324 void
6325 irq_enable(int irq)
6326 {
6327   irq_setmask_8259A(irq_mask_8259A & ~(1<<irq));
6328 }
6329
6330 // Initialize the 8259A interrupt controllers.
6331 void
6332 pic_init(void)
6333 {
6334   // mask all interrupts
6335   outb(IO_PIC1+1, 0xFF);
6336   outb(IO_PIC2+1, 0xFF);
6337
6338   // Set up master (8259A-1)
6339
6340   // ICW1:  0001g0hi
6341   //    g:  0 = edge triggering, 1 = level triggering
6342   //    h:  0 = cascaded PICs, 1 = master only
6343   //    i:  0 = no ICW4, 1 = ICW4 required
6344   outb(IO_PIC1, 0x11);
6345
6346   // ICW2:  Vector offset
6347   outb(IO_PIC1+1, IRQ_OFFSET);
6348
6349
```



```
6350   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
6351   //        (slave PIC) 3-bit # of slave's connection to master
6352   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
6353
6354   // ICW4:  000nbmap
6355   //    n:  1 = special fully nested mode
6356   //    b:  1 = buffered mode
6357   //    m:  0 = slave PIC, 1 = master PIC
6358   //      (ignored when b is 0, as the master/slave role
6359   //      can be hardwired).
6360   //    a:  1 = Automatic EOI mode
6361   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
6362   outb(IO_PIC1+1, 0x3);
6363
6364   // Set up slave (8259A-2)
6365   outb(IO_PIC2, 0x11);                  // ICW1
6366   outb(IO_PIC2+1, IRQ_OFFSET + 8);      // ICW2
6367   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
6368   // NB Automatic EOI mode doesn't tend to work on the slave.
6369   // Linux source code says it's "to be investigated".
6370   outb(IO_PIC2+1, 0x3);                 // ICW4
6371
6372   // OCW3:  0ef01prs
6373   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
6374   //    p:  0 = no polling, 1 = polling mode
6375   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
6376   outb(IO_PIC1, 0x68);             // clear specific mask
6377   outb(IO_PIC1, 0x0a);             // read IRR by default
6378
6379   outb(IO_PIC2, 0x68);             // OCW3
6380   outb(IO_PIC2, 0x0a);             // OCW3
6381
6382   if(irq_mask_8259A != 0xFFFF)
6383     irq_setmask_8259A(irq_mask_8259A);
6384 }
6385
6386
6387
6388
6389
6390
6391
6392
6393
6394
6395
6396
6397
6398
6399
```

```
6400 #include "types.h"
6401 #include "x86.h"
6402 #include "traps.h"
6403 #include "defs.h"
6404 #include "spinlock.h"
6405 #include "dev.h"
6406 #include "param.h"
6407 #include "mmu.h"
6408
6409 struct spinlock console_lock;
6410 int panicked = 0;
6411 int use_console_lock = 0;
6412
6413 // Copy console output to parallel port, which you can tell
6414 // .bochsrc to copy to the stdout:
6415 //   parport1: enabled=1, file="/dev/stdout"
6416 static void
6417 lpt_putc(int c)
6418 {
6419   int i;
6420
6421   for(i = 0; !(inb(0x378+1) & 0x80) && i < 12800; i++)
6422     ;
6423   outb(0x378+0, c);
6424   outb(0x378+2, 0x08|0x04|0x01);
6425   outb(0x378+2, 0x08);
6426 }
6427
6428 static void
6429 cons_putc(int c)
6430 {
6431   int crtport = 0x3d4; // io port of CGA
6432   ushort *crt = (ushort*) 0xB8000; // base of CGA memory
6433   int ind;
6434
6435   if(panicked){
6436     cli();
6437     for(;;)
6438       ;
6439   }
6440
6441   lpt_putc(c);
6442
6443   // cursor position, 16 bits, col + 80*row
6444   outb(crtport, 14);
6445   ind = inb(crtport + 1) << 8;
6446   outb(crtport, 15);
6447   ind |= inb(crtport + 1);
6448
6449   c &= 0xff;
```



```
6450   if(c == '\n'){
6451     ind -= (ind % 80);
6452     ind += 80;
6453   } else {
6454     c |= 0x0700; // black on white
6455     crt[ind] = c;
6456     ind++;
6457   }
6458
6459   if((ind / 80) >= 24){
6460     // scroll up
6461     memmove(crt, crt + 80, sizeof(crt[0]) * (23 * 80));
6462     ind -= 80;
6463     memset(crt + ind, 0, sizeof(crt[0]) * ((24 * 80) - ind));
6464   }
6465
6466   outb(crtport, 14);
6467   outb(crtport + 1, ind >> 8);
6468   outb(crtport, 15);
6469   outb(crtport + 1, ind);
6470 }
6471
6472 void
6473 printint(int xx, int base, int sgn)
6474 {
6475   char buf[16];
6476   char digits[] = "0123456789ABCDEF";
6477   int i = 0, neg = 0;
6478   uint x;
6479
6480   if(sgn && xx < 0){
6481     neg = 1;
6482     x = 0 - xx;
6483   } else {
6484     x = xx;
6485   }
6486
6487   do {
6488     buf[i++] = digits[x % base];
6489   } while((x /= base) != 0);
6490   if(neg)
6491     buf[i++] = '-';
6492
6493   while(--i >= 0)
6494     cons_putc(buf[i]);
6495 }
6496
6497
6498
6499
```

```
6500 // Print to the console. only understands %d, %x, %p, %s.
6501 void
6502 cprintf(char *fmt, ...)
6503 {
6504   int i, state = 0, c, locking = 0;
6505   uint *ap = (uint*)(void*)&fmt + 1;
6506
6507   if(use_console_lock){
6508     locking = 1;
6509     acquire(&console_lock);
6510   }
6511
6512   for(i = 0; fmt[i]; i++){
6513     c = fmt[i] & 0xff;
6514     if(state == 0){
6515       if(c == '%'){
6516         state = '%';
6517       } else {
6518         cons_putc(c);
6519       }
6520     } else if(state == '%'){
6521       if(c == 'd'){
6522         printint(*ap, 10, 1);
6523         ap++;
6524       } else if(c == 'x' || c == 'p'){
6525         printint(*ap, 16, 0);
6526         ap++;
6527       } else if(c == 's'){
6528         char *s = (char*)*ap;
6529         ap++;
6530         if(s == 0){
6531           cons_putc('0');
6532         }else{
6533           while(*s != 0){
6534             cons_putc(*s);
6535             s++;
6536           }
6537         }
6538       } else if(c == '%'){
6539         cons_putc(c);
6540       } else {
6541         // Unknown % sequence.  Print it to draw attention.
6542         cons_putc('%');
6543         cons_putc(c);
6544       }
6545       state = 0;
6546     }
6547   }
6548
6549
```



```
6550   if(locking)
6551     release(&console_lock);
6552 }
6553
6554 void
6555 panic(char *s)
6556 {
6557   int i;
6558   uint pcs[10];
6559
6560   __asm __volatile("cli");
6561   use_console_lock = 0;
6562   cprintf("panic (%d): ", cpu());
6563   cprintf(s, 0);
6564   cprintf("\n", 0);
6565   getcallerpcs(&s, pcs);
6566   for(i=0; i<10; i++)
6567     cprintf(" %p", pcs[i]);
6568   panicked = 1; // freeze other CPU
6569   for(;;)
6570     ;
6571 }
6572
6573 int
6574 console_write(int minor, char *buf, int n)
6575 {
6576   int i;
6577
6578   acquire(&console_lock);
6579
6580   for(i = 0; i < n; i++) {
6581     cons_putc(buf[i] & 0xff);
6582   }
6583
6584   release(&console_lock);
6585
6586   return n;
6587 }
6588
6589
6590
6591
6592
6593
6594
6595
6596
6597
6598
6599
```

```
6600 #define KBSTATP         0x64    // kbd controller status port(I)
6601 #define KBS_DIB         0x01    // kbd data in buffer
6602 #define KBDATAP         0x60    // kbd data port(I)
6603
6604 #define NO              0
6605
6606 #define SHIFT           (1<<0)
6607 #define CTL             (1<<1)
6608 #define ALT             (1<<2)
6609
6610 #define CAPSLOCK        (1<<3)
6611 #define NUMLOCK         (1<<4)
6612 #define SCROLLLOCK      (1<<5)
6613
6614 #define E0ESC           (1<<6)
6615
6616 // Special keycodes
6617 #define KEY_HOME        0xE0
6618 #define KEY_END         0xE1
6619 #define KEY_UP          0xE2
6620 #define KEY_DN          0xE3
6621 #define KEY_LF          0xE4
6622 #define KEY_RT          0xE5
6623 #define KEY_PGUP        0xE6
6624 #define KEY_PGDN        0xE7
6625 #define KEY_INS         0xE8
6626 #define KEY_DEL         0xE9
6627
6628 static uchar shiftcode[256] =
6629 {
6630   [0x1D] CTL,
6631   [0x2A] SHIFT,
6632   [0x36] SHIFT,
6633   [0x38] ALT,
6634   [0x9D] CTL,
6635   [0xB8] ALT
6636 };
6637
6638 static uchar togglecode[256] =
6639 {
6640   [0x3A] CAPSLOCK,
6641   [0x45] NUMLOCK,
6642   [0x46] SCROLLLOCK
6643 };
6644
6645
6646
6647
6648
6649
```



```
6650 static uchar normalmap[256] =
6651 {
6652   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
6653   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
6654   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
6655   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
6656   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
6657   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
6658   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
6659   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
6660   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
6661   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
6662   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
6663   [0x97] KEY_HOME,
6664   [0x9C] '\n',      // KP_Enter
6665   [0xB5] '/',       // KP_Div
6666   [0xC8] KEY_UP,
6667   [0xC9] KEY_PGUP,
6668   [0xCB] KEY_LF,
6669   [0xCD] KEY_RT,
6670   [0xCF] KEY_END,
6671   [0xD0] KEY_DN,
6672   [0xD1] KEY_PGDN,
6673   [0xD2] KEY_INS,
6674   [0xD3] KEY_DEL
6675 };
6676
6677 static uchar shiftmap[256] =
6678 {
6679   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
6680   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
6681   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
6682   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
6683   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
6684   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
6685   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
6686   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
6687   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
6688   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
6689   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
6690   [0x97] KEY_HOME,
6691   [0x9C] '\n',      // KP_Enter
6692   [0xB5] '/',       // KP_Div
6693   [0xC8] KEY_UP,
6694   [0xC9] KEY_PGUP,
6695   [0xCB] KEY_LF,
6696   [0xCD] KEY_RT,
6697   [0xCF] KEY_END,
6698   [0xD0] KEY_DN,
6699   [0xD1] KEY_PGDN,
```

```
6700   [0xD2] KEY_INS,
6701   [0xD3] KEY_DEL
6702 };
6703
6704 #define C(x) (x - '@')
6705
6706 static uchar ctlmap[256] =
6707 {
6708   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
6709   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
6710   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
6711   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
6712   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
6713   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
6714   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
6715   [0x97] KEY_HOME,
6716   [0xB5] C('/'),    // KP_Div
6717   [0xC8] KEY_UP,
6718   [0xC9] KEY_PGUP,
6719   [0xCB] KEY_LF,
6720   [0xCD] KEY_RT,
6721   [0xCF] KEY_END,
6722   [0xD0] KEY_DN,
6723   [0xD1] KEY_PGDN,
6724   [0xD2] KEY_INS,
6725   [0xD3] KEY_DEL
6726 };
6727
6728 static uchar *charcode[4] = {
6729   normalmap,
6730   shiftmap,
6731   ctlmap,
6732   ctlmap
6733 };
6734
6735 #define KBD_BUF 64
6736 char kbd_buf[KBD_BUF];
6737 int kbd_r;
6738 int kbd_w;
6739 struct spinlock kbd_lock;
6740 static uint shift;
6741
6742 void
6743 kbd_intr()
6744 {
6745   uint st, data, c;
6746
6747   acquire(&kbd_lock);
6748
6749
```



```
6750   st = inb(KBSTATP);
6751   if((st & KBS_DIB) == 0)
6752     goto out;
6753   data = inb(KBDATAP);
6754
6755   if(data == 0xE0) {
6756     shift |= E0ESC;
6757     goto out;
6758   } else if(data & 0x80) {
6759     // Key released
6760     data = (shift & E0ESC ? data : data & 0x7F);
6761     shift &= ~(shiftcode[data] | E0ESC);
6762     goto out;
6763   } else if(shift & E0ESC) {
6764     // Last character was an E0 escape; or with 0x80
6765     data |= 0x80;
6766     shift &= ~E0ESC;
6767   }
6768
6769   shift |= shiftcode[data];
6770   shift ^= togglecode[data];
6771
6772   c = charcode[shift & (CTL | SHIFT)][data];
6773   if(shift & CAPSLOCK) {
6774     if('a' <= c && c <= 'z')
6775       c += 'A' - 'a';
6776     else if('A' <= c && c <= 'Z')
6777       c += 'a' - 'A';
6778   }
6779
6780   switch(c){
6781   case 0:
6782     // Ignore unknown keystrokes.
6783     break;
6784
6785   case C('T'):
6786     cprintf("#");  // Let user know we're still alive.
6787     break;
6788
6789   case C('P'):
6790     procdump();
6791     break;
6792
6793   default:
6794     if(((kbd_w + 1) % KBD_BUF) != kbd_r){
6795       kbd_buf[kbd_w++] = c;
6796       if(kbd_w >= KBD_BUF)
6797         kbd_w = 0;
6798       wakeup(&kbd_r);
6799     }
```

```
6800     break;
6801   }
6802
6803 out:
6804   release(&kbd_lock);
6805 }
6806
6807 int
6808 console_read(int minor, char *dst, int n)
6809 {
6810   uint target = n;
6811
6812   acquire(&kbd_lock);
6813
6814   while(kbd_w == kbd_r) {
6815     sleep(&kbd_r, &kbd_lock);
6816   }
6817
6818   while(n > 0 && kbd_w != kbd_r){
6819     *dst = (kbd_buf[kbd_r]) & 0xff;
6820     cons_putc(*dst & 0xff);
6821     dst++;
6822     --n;
6823     kbd_r++;
6824     if(kbd_r >= KBD_BUF)
6825       kbd_r = 0;
6826   }
6827
6828   release(&kbd_lock);
6829
6830   return target - n;
6831 }
6832
6833 void
6834 console_init()
6835 {
6836   initlock(&console_lock, "console");
6837   initlock(&kbd_lock, "kbd");
6838
6839   devsw[CONSOLE].write = console_write;
6840   devsw[CONSOLE].read = console_read;
6841
6842   irq_enable(IRQ_KBD);
6843   ioapic_enable(IRQ_KBD, 0);
6844
6845   use_console_lock = 1;
6846 }
6847
6848
6849
```



```
6850 #include "types.h"
6851 #include "x86.h"
6852 #include "defs.h"
6853 #include "traps.h"
6854
6855 // Register definitions for the Intel
6856 // 8253/8254/82C54 Programmable Interval Timer (PIT).
6857
6858 #define IO_TIMER1       0x040           // 8253 Timer #1
6859 #define IO_TIMER2       0x048           // 8253 Timer #2 (EISA only)
6860
6861 // Frequency of all three count-down timers; (TIMER_FREQ/freq) is the
6862 // appropriate count to generate a frequency of freq hz.
6863
6864 #define TIMER_FREQ      1193182
6865 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
6866
6867 #define TIMER_CNTR0     (IO_TIMER1 + 0) // timer 0 counter port
6868 #define TIMER_CNTR1     (IO_TIMER1 + 1) // timer 1 counter port
6869 #define TIMER_CNTR2     (IO_TIMER1 + 2) // timer 2 counter port
6870 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
6871 #define TIMER_SEL0      0x00    // select counter 0
6872 #define TIMER_SEL1      0x40    // select counter 1
6873 #define TIMER_SEL2      0x80    // select counter 2
6874 #define TIMER_INTTC     0x00    // mode 0, intr on terminal cnt
6875 #define TIMER_ONESHOT   0x02    // mode 1, one shot
6876 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
6877 #define TIMER_SQWAVE    0x06    // mode 3, square wave
6878 #define TIMER_SWSTROBE  0x08    // mode 4, s/w triggered strobe
6879 #define TIMER_HWSTROBE  0x0a    // mode 5, h/w triggered strobe
6880 #define TIMER_LATCH     0x00    // latch counter for reading
6881 #define TIMER_LSB       0x10    // r/w counter LSB
6882 #define TIMER_MSB       0x20    // r/w counter MSB
6883 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
6884 #define TIMER_BCD       0x01    // count in BCD
6885
6886 void
6887 pit8253_timerinit(void)
6888 {
6889   // initialize 8253 clock to interrupt 100 times/sec
6890   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
6891   outb(IO_TIMER1, TIMER_DIV(100) % 256);
6892   outb(IO_TIMER1, TIMER_DIV(100) / 256);
6893   irq_enable(IRQ_TIMER);
6894 }
6895
6896
6897
6898
6899
```