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

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14,
2000)). See also http://pdos.csail.mit.edu/6.828/2007/v6.html, which
provides pointers to on-line resources for v6.

xv6 borrows code from the following sources:
    JOS (asm.h, elf.h, mmu.h, bootasm.S, ide.c, console.c, and others)
    Plan 9 (bootother.S, mp.h, mp.c, lapic.c)
    FreeBSD (ioapic.c)
    NetBSD (console.c)

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

ERROR REPORTS

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

BUILDING AND RUNNING XV6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run "make".
On non-x86 or non-ELF machines (like OS X, even on x86), you will
need to install a cross-compiler gcc suite capable of producing x86 ELF
binaries. See http://pdos.csail.mit.edu/6.828/2007/tools.html.
Then run "make TOOLPREFIX=i386-jos-elf-".

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

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

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

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

```
    swtch 2256
        0311 1928 1962 2255
        2256
```

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

```
0100 #ifndef XV6_TYPES_H_
0101 #define XV6_TYPES_H_
0102 typedef unsigned int   uint;
0103 typedef unsigned int   uint32_t;
0104 typedef unsigned short ushort;
0105 typedef unsigned short uint16_t;
0106 typedef unsigned char  uchar;
0107 typedef unsigned char  uint8_t;
0108 #endif // XV6_TYPES_H_
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 #ifndef XV6_PARAM_H_
0151 #define XV6_PARAM_H_
0152 #define NPROC        64  // maximum number of processes
0153 #define PAGE       4096  // granularity of user-space memory allocation
0154 #define KSTACKSIZE PAGE  // size of per-process kernel stack
0155 #define NCPU          1  // maximum number of CPUs
0156 #define NOFILE       16  // open files per process
0157 #define NFILE       100  // open files per system
0158 #define NBUF         10  // size of disk block cache
0159 #define NINODE       50  // maximum number of active i-nodes
0160 #define NDEV         10  // maximum major device number
0161 #define ROOTDEV       1  // device number of file system root disk
0162
0163 #endif // XV6_PARAM_H_
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 #ifndef XV6_DEFS_H_
0201 #define XV6_DEFS_H_
0202 #include "types.h"
0203 struct buf;
0204 struct context;
0205 struct file;
0206 struct inode;
0207 struct pipe;
0208 struct proc;
0209 struct spinlock;
0210 struct stat;
0211
0212 // bio.c
0213 void            binit(void);
0214 struct buf*     bread(uint, uint);
0215 void            brelse(struct buf*);
0216 void            bwrite(struct buf*);
0217
0218 // console.c
0219 void            console_init(void);
0220 void            cprintf(char*, ...);
0221 void            console_intr(int(*)(void));
0222 void            panic(char*) __attribute__((noreturn));
0223 void            printstack(void);
0224
0225 // exec.c
0226 int             exec(char*, char**);
0227
0228 // file.c
0229 struct file*    filealloc(void);
0230 void            fileclose(struct file*);
0231 struct file*    filedup(struct file*);
0232 void            fileinit(void);
0233 int             fileread(struct file*, char*, int n);
0234 int             filestat(struct file*, struct stat*);
0235 int             filewrite(struct file*, char*, int n);
0236
0237 // fs.c
0238 int             dirlink(struct inode*, char*, uint);
0239 struct inode*   dirlookup(struct inode*, char*, uint*);
0240 struct inode*   ialloc(uint, short);
0241 struct inode*   idup(struct inode*);
0242 void            iinit(void);
0243 void            ilock(struct inode*);
0244 void            iput(struct inode*);
0245 void            iunlock(struct inode*);
0246 void            iunlockput(struct inode*);
0247 void            iupdate(struct inode*);
0248 int             namecmp(const char*, const char*);
0249 struct inode*   namei(char*);
```



```
0250 struct inode*   nameiparent(char*, char*);
0251 int             readi(struct inode*, char*, uint, uint);
0252 void            stati(struct inode*, struct stat*);
0253 int             writei(struct inode*, char*, uint, uint);
0254
0255 // ide.c
0256 void            ide_init(void);
0257 void            ide_intr(void);
0258 void            ide_rw(struct buf *);
0259
0260 // ioapic.c
0261 void            ioapic_enable(int irq, int cpu);
0262 extern uchar    ioapic_id;
0263 void            ioapic_init(void);
0264
0265 // kalloc.c
0266 char*           kalloc(int);
0267 void            kfree(char*, int);
0268 void*           kmalloc(int);
0269 void            kmfree(void*, int);
0270 void            kinit(void);
0271
0272 // kbd.c
0273 void            kbd_intr(void);
0274
0275 // lapic.c
0276 int             cpu(void);
0277 extern volatile uint*    lapic;
0278 void            lapic_disableintr(void);
0279 void            lapic_enableintr(void);
0280 void            lapic_eoi(void);
0281 void            lapic_init(int);
0282 void            lapic_startap(uchar, uint);
0283 void            lapic_timerinit(void);
0284 void            lapic_timerintr(void);
0285
0286 // mp.c
0287 extern int      ismp;
0288 int             mp_bcpu(void);
0289 void            mp_init(void);
0290 void            mp_startthem(void);
0291
0292 // picirq.c
0293 void            pic_enable(int);
0294 void            pic_init(void);
0295
0296
0297
0298
0299
```

```
0300 // pipe.c
0301 int             pipealloc(struct file**, struct file**);
0302 void            pipeclose(struct pipe*, int);
0303 int             piperead(struct pipe*, char*, int);
0304 int             pipewrite(struct pipe*, char*, int);
0305
0306 // proc.c
0307 struct proc*    copyproc(struct proc*);
0308 struct proc*    allocproc(void);
0309 void            exit(void);
0310 int             growproc(int);
0311 int             kill(int);
0312 void            pinit(void);
0313 void            procdump(void);
0314 void            scheduler(void) __attribute__((noreturn));
0315 void            setupsegs(struct proc*);
0316 void            sleep(void*, struct spinlock*);
0317 int             msleep_spin(void*, struct spinlock*, int);
0318 void            userinit(void);
0319 int             wait(void);
0320 void            wakeup(void*);
0321 void            wakeup_one(void*);
0322 void            yield(void);
0323
0324 // swtch.S
0325 void            swtch(struct context*, struct context*);
0326
0327 // spinlock.c
0328 void            acquire(struct spinlock*);
0329 void            getcallerpcs(void*, uint*);
0330 int             holding(struct spinlock*);
0331 void            initlock(struct spinlock*, char*);
0332 void            release(struct spinlock*);
0333
0334 // string.c
0335 int             memcmp(const void*, const void*, uint);
0336 void*           memmove(void*, const void*, uint);
0337 void*           memset(void*, int, uint);
0338 char*           safestrcpy(char*, const char*, int);
0339 int             strlen(const char*);
0340 int             strncmp(const char*, const char*, uint);
0341 char*           strncpy(char*, const char*, int);
0342 void*           memcpy(void *dst, const void *src, uint n);
0343
0344 // syscall.c
0345 int             argint(int, int*);
0346 int             argptr(int, char**, int);
0347 int             argstr(int, char**);
0348 int             fetchint(struct proc*, uint, int*);
0349 int             fetchstr(struct proc*, uint, char**);
```



```
0350 void            syscall(void);
0351
0352 // timer.c
0353 void            timer_init(void);
0354 int             millitime(void);
0355
0356 // trap.c
0357 void            idtinit(void);
0358 extern int      ticks;
0359 void            tvinit(void);
0360 extern struct spinlock tickslock;
0361
0362 // number of elements in fixed-size array
0363 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
0364
0365 #ifndef NULL
0366 #define NULL    0
0367 #endif
0368
0369 #endif // XV6_DEFS_H_
0370
0371
0372
0373
0374
0375
0376
0377
0378
0379
0380
0381
0382
0383
0384
0385
0386
0387
0388
0389
0390
0391
0392
0393
0394
0395
0396
0397
0398
0399
```

```
0400 #ifndef XV6_X86_H_
0401 #define XV6_X86_H_
0402 // Routines to let C code use special x86 instructions.
0403
0404 static inline uchar
0405 inb(ushort port)
0406 {
0407   uchar data;
0408
0409   asm volatile("in %1,%0" : "=a" (data) : "d" (port));
0410   return data;
0411 }
0412
0413 static inline uint
0414 inl(ushort port)
0415 {
0416     uint data;
0417
0418     asm volatile("in %1,%0" : "=a" (data) : "d" (port));
0419     return data;
0420 }
0421
0422 static inline ushort
0423 inw(ushort port)
0424 {
0425     ushort data;
0426
0427     asm volatile("in %1,%0" : "=a" (data) : "d" (port));
0428     return data;
0429 }
0430
0431 static inline void
0432 insl(int port, void *addr, int cnt)
0433 {
0434   asm volatile("cld\n\trepne\n\tinsl"     :
0435                    "=D" (addr), "=c" (cnt)    :
0436                    "d" (port), "0" (addr), "1" (cnt)  :
0437                    "memory", "cc");
0438 }
0439
0440 static inline void
0441 outb(ushort port, uchar data)
0442 {
0443   asm volatile("out %0,%1" : : "a" (data), "d" (port));
0444 }
0445
0446
0447
0448
0449
```



```
0450 static inline void
0451 outw(ushort port, ushort data)
0452 {
0453   asm volatile("out %0,%1" : : "a" (data), "d" (port));
0454 }
0455
0456 static inline void
0457 outl(ushort port, uint data)
0458 {
0459     asm volatile("out %0,%1" : : "a" (data), "d" (port));
0460 }
0461
0462 static inline void
0463 outsl(int port, const void *addr, int cnt)
0464 {
0465   asm volatile("cld\n\trepne\n\toutsl"    :
0466                    "=S" (addr), "=c" (cnt)    :
0467                    "d" (port), "0" (addr), "1" (cnt)  :
0468                    "cc");
0469 }
0470
0471 struct segdesc;
0472
0473 static inline void
0474 lgdt(struct segdesc *p, int size)
0475 {
0476   volatile ushort pd[3];
0477
0478   pd[0] = size-1;
0479   pd[1] = (uint)p;
0480   pd[2] = (uint)p >> 16;
0481
0482   asm volatile("lgdt (%0)" : : "r" (pd));
0483 }
0484
0485 struct gatedesc;
0486
0487 static inline void
0488 lidt(struct gatedesc *p, int size)
0489 {
0490   volatile ushort pd[3];
0491
0492   pd[0] = size-1;
0493   pd[1] = (uint)p;
0494   pd[2] = (uint)p >> 16;
0495
0496   asm volatile("lidt (%0)" : : "r" (pd));
0497 }
0498
0499
```

```
0500 static inline void
0501 ltr(ushort sel)
0502 {
0503   asm volatile("ltr %0" : : "r" (sel));
0504 }
0505
0506 static inline uint
0507 read_eflags(void)
0508 {
0509   uint eflags;
0510   asm volatile("pushfl; popl %0" : "=r" (eflags));
0511   return eflags;
0512 }
0513
0514 static inline void
0515 write_eflags(uint eflags)
0516 {
0517   asm volatile("pushl %0; popfl" : : "r" (eflags));
0518 }
0519
0520 static inline void
0521 cpuid(uint info, uint *eaxp, uint *ebxp, uint *ecxp, uint *edxp)
0522 {
0523   uint eax, ebx, ecx, edx;
0524
0525   asm volatile("cpuid" :
0526                "=a" (eax), "=b" (ebx), "=c" (ecx), "=d" (edx) :
0527                "a" (info));
0528   if(eaxp)
0529     *eaxp = eax;
0530   if(ebxp)
0531     *ebxp = ebx;
0532   if(ecxp)
0533     *ecxp = ecx;
0534   if(edxp)
0535     *edxp = edx;
0536 }
0537
0538 static inline uint
0539 cmpxchg(uint oldval, uint newval, volatile uint* lock_addr)
0540 {
0541   uint result;
0542
0543   // The + in "+m" denotes a read-modify-write operand.
0544   asm volatile("lock; cmpxchgl %2, %0" :
0545                        "+m" (*lock_addr), "=a" (result) :
0546                        "r"(newval), "1"(oldval) :
0547                        "cc");
0548   return result;
0549 }
```



```
0550 static inline void
0551 cli(void)
0552 {
0553   asm volatile("cli");
0554 }
0555
0556 static inline void
0557 sti(void)
0558 {
0559   asm volatile("sti");
0560 }
0561
0562 // Layout of the trap frame built on the stack by the
0563 // hardware and by trapasm.S, and passed to trap().
0564 struct trapframe {
0565   // registers as pushed by pusha
0566   uint edi;
0567   uint esi;
0568   uint ebp;
0569   uint oesp;      // useless & ignored
0570   uint ebx;
0571   uint edx;
0572   uint ecx;
0573   uint eax;
0574
0575   // rest of trap frame
0576   ushort es;
0577   ushort padding1;
0578   ushort ds;
0579   ushort padding2;
0580   uint trapno;
0581
0582   // below here defined by x86 hardware
0583   uint err;
0584   uint eip;
0585   ushort cs;
0586   ushort padding3;
0587   uint eflags;
0588
0589   // below here only when crossing rings, such as from user to kernel
0590   uint esp;
0591   ushort ss;
0592   ushort padding4;
0593 };
0594 #endif // XV6_X86_H_
0595
0596
0597
0598
0599
```

```
0600 //
0601 // assembler macros to create x86 segments
0602 //
0603
0604 #define SEG_NULLASM                                             \
0605         .word 0, 0;                                             \
0606         .byte 0, 0, 0, 0
0607
0608 #define SEG_ASM(type,base,lim)                                  \
0609         .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
0610         .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
0611                 (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
0612
0613 #define STA_X     0x8       // Executable segment
0614 #define STA_E     0x4       // Expand down (non-executable segments)
0615 #define STA_C     0x4       // Conforming code segment (executable only)
0616 #define STA_W     0x2       // Writeable (non-executable segments)
0617 #define STA_R     0x2       // Readable (executable segments)
0618 #define STA_A     0x1       // Accessed
0619
0620
0621
0622
0623
0624
0625
0626
0627
0628
0629
0630
0631
0632
0633
0634
0635
0636
0637
0638
0639
0640
0641
0642
0643
0644
0645
0646
0647
0648
0649
```



```
0650 #ifndef XV6_MMU_H_
0651 #define XV6_MMU_H_
0652 // This file contains definitions for the
0653 // x86 memory management unit (MMU).
0654
0655 // Eflags register
0656 #define FL_CF           0x00000001      // Carry Flag
0657 #define FL_PF           0x00000004      // Parity Flag
0658 #define FL_AF           0x00000010      // Auxiliary carry Flag
0659 #define FL_ZF           0x00000040      // Zero Flag
0660 #define FL_SF           0x00000080      // Sign Flag
0661 #define FL_TF           0x00000100      // Trap Flag
0662 #define FL_IF           0x00000200      // Interrupt Enable
0663 #define FL_DF           0x00000400      // Direction Flag
0664 #define FL_OF           0x00000800      // Overflow Flag
0665 #define FL_IOPL_MASK    0x00003000      // I/O Privilege Level bitmask
0666 #define FL_IOPL_0       0x00000000      //   IOPL == 0
0667 #define FL_IOPL_1       0x00001000      //   IOPL == 1
0668 #define FL_IOPL_2       0x00002000      //   IOPL == 2
0669 #define FL_IOPL_3       0x00003000      //   IOPL == 3
0670 #define FL_NT           0x00004000      // Nested Task
0671 #define FL_RF           0x00010000      // Resume Flag
0672 #define FL_VM           0x00020000      // Virtual 8086 mode
0673 #define FL_AC           0x00040000      // Alignment Check
0674 #define FL_VIF          0x00080000      // Virtual Interrupt Flag
0675 #define FL_VIP          0x00100000      // Virtual Interrupt Pending
0676 #define FL_ID           0x00200000      // ID flag
0677
0678 // Segment Descriptor
0679 struct segdesc {
0680   uint lim_15_0 : 16;  // Low bits of segment limit
0681   uint base_15_0 : 16; // Low bits of segment base address
0682   uint base_23_16 : 8; // Middle bits of segment base address
0683   uint type : 4;       // Segment type (see STS_ constants)
0684   uint s : 1;          // 0 = system, 1 = application
0685   uint dpl : 2;        // Descriptor Privilege Level
0686   uint p : 1;          // Present
0687   uint lim_19_16 : 4;  // High bits of segment limit
0688   uint avl : 1;        // Unused (available for software use)
0689   uint rsv1 : 1;       // Reserved
0690   uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
0691   uint g : 1;          // Granularity: limit scaled by 4K when set
0692   uint base_31_24 : 8; // High bits of segment base address
0693 };
0694
0695
0696
0697
0698
0699
```

```
0700 // Null segment
0701 #define SEG_NULL        (struct segdesc){ 0,0,0,0,0,0,0,0,0,0,0,0,0,0 }
0702
0703 // Normal segment
0704 #define SEG(type, base, lim, dpl) (struct segdesc)                    \
0705 { ((lim) >> 12) & 0xffff, (base) & 0xffff, ((base) >> 16) & 0xff,       \
0706     type, 1, dpl, 1, (uint) (lim) >> 28, 0, 0, 1, 1,                    \
0707     (uint) (base) >> 24 }
0708
0709 #define SEG16(type, base, lim, dpl) (struct segdesc)                   \
0710 { (lim) & 0xffff, (base) & 0xffff, ((base) >> 16) & 0xff,              \
0711     type, 1, dpl, 1, (uint) (lim) >> 16, 0, 0, 1, 0,                   \
0712     (uint) (base) >> 24 }
0713
0714 #define DPL_USER    0x3     // User DPL
0715
0716 // Application segment type bits
0717 #define STA_X       0x8     // Executable segment
0718 #define STA_E       0x4     // Expand down (non-executable segments)
0719 #define STA_C       0x4     // Conforming code segment (executable only)
0720 #define STA_W       0x2     // Writeable (non-executable segments)
0721 #define STA_R       0x2     // Readable (executable segments)
0722 #define STA_A       0x1     // Accessed
0723
0724 // System segment type bits
0725 #define STS_T16A    0x1     // Available 16-bit TSS
0726 #define STS_LDT     0x2     // Local Descriptor Table
0727 #define STS_T16B    0x3     // Busy 16-bit TSS
0728 #define STS_CG16    0x4     // 16-bit Call Gate
0729 #define STS_TG      0x5     // Task Gate / Coum Transmitions
0730 #define STS_IG16    0x6     // 16-bit Interrupt Gate
0731 #define STS_TG16    0x7     // 16-bit Trap Gate
0732 #define STS_T32A    0x9     // Available 32-bit TSS
0733 #define STS_T32B    0xB     // Busy 32-bit TSS
0734 #define STS_CG32    0xC     // 32-bit Call Gate
0735 #define STS_IG32    0xE     // 32-bit Interrupt Gate
0736 #define STS_TG32    0xF     // 32-bit Trap Gate
0737
0738 // Task state segment format
0739 struct taskstate {
0740   uint link;         // Old ts selector
0741   uint esp0;         // Stack pointers and segment selectors
0742   ushort ss0;        //   after an increase in privilege level
0743   ushort padding1;
0744   uint *esp1;
0745   ushort ss1;
0746   ushort padding2;
0747   uint *esp2;
0748   ushort ss2;
0749   ushort padding3;
```



```
0750   void *cr3;         // Page directory base
0751   uint *eip;         // Saved state from last task switch
0752   uint eflags;
0753   uint eax;          // More saved state (registers)
0754   uint ecx;
0755   uint edx;
0756   uint ebx;
0757   uint *esp;
0758   uint *ebp;
0759   uint esi;
0760   uint edi;
0761   ushort es;         // Even more saved state (segment selectors)
0762   ushort padding4;
0763   ushort cs;
0764   ushort padding5;
0765   ushort ss;
0766   ushort padding6;
0767   ushort ds;
0768   ushort padding7;
0769   ushort fs;
0770   ushort padding8;
0771   ushort gs;
0772   ushort padding9;
0773   ushort ldt;
0774   ushort padding10;
0775   ushort t;          // Trap on task switch
0776   ushort iomb;       // I/O map base address
0777 };
0778
0779 // Gate descriptors for interrupts and traps
0780 struct gatedesc {
0781   uint off_15_0 : 16;   // low 16 bits of offset in segment
0782   uint ss : 16;         // segment selector
0783   uint args : 5;        // # args, 0 for interrupt/trap gates
0784   uint rsv1 : 3;        // reserved(should be zero I guess)
0785   uint type : 4;        // type(STS_{TG,IG32,TG32})
0786   uint s : 1;           // must be 0 (system)
0787   uint dpl : 2;         // descriptor(meaning new) privilege level
0788   uint p : 1;           // Present
0789   uint off_31_16 : 16;  // high bits of offset in segment
0790 };
0791
0792
0793
0794
0795
0796
0797
0798
0799
```

```
0800 // Set up a normal interrupt/trap gate descriptor.
0801 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
0802 //   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
0803 // - sel: Code segment selector for interrupt/trap handler
0804 // - off: Offset in code segment for interrupt/trap handler
0805 // - dpl: Descriptor Privilege Level -
0806 //        the privilege level required for software to invoke
0807 //        this interrupt/trap gate explicitly using an int instruction.
0808 #define SETGATE(gate, istrap, sel, off, d)                \
0809 {                                                         \
0810   (gate).off_15_0 = (uint) (off) & 0xffff;                \
0811   (gate).ss = (sel);                                      \
0812   (gate).args = 0;                                        \
0813   (gate).rsv1 = 0;                                        \
0814   (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
0815   (gate).s = 0;                                           \
0816   (gate).dpl = (d);                                       \
0817   (gate).p = 1;                                           \
0818   (gate).off_31_16 = (uint) (off) >> 16;                  \
0819 }
0820
0821 #endif // XV6_MMU_H_
0822
0823
0824
0825
0826
0827
0828
0829
0830
0831
0832
0833
0834
0835
0836
0837
0838
0839
0840
0841
0842
0843
0844
0845
0846
0847
0848
0849
```



```
0850 // Format of an ELF executable file
0851
0852 #define ELF_MAGIC 0x464C457FU  // "\x7FELF" in little endian
0853
0854 // File header
0855 struct elfhdr {
0856   uint magic;  // must equal ELF_MAGIC
0857   uchar elf[12];
0858   ushort type;
0859   ushort machine;
0860   uint version;
0861   uint entry;
0862   uint phoff;
0863   uint shoff;
0864   uint flags;
0865   ushort ehsize;
0866   ushort phentsize;
0867   ushort phnum;
0868   ushort shentsize;
0869   ushort shnum;
0870   ushort shstrndx;
0871 };
0872
0873 // Program section header
0874 struct proghdr {
0875   uint type;
0876   uint offset;
0877   uint va;
0878   uint pa;
0879   uint filesz;
0880   uint memsz;
0881   uint flags;
0882   uint align;
0883 };
0884
0885 // Values for Proghdr type
0886 #define ELF_PROG_LOAD           1
0887
0888 // Flag bits for Proghdr flags
0889 #define ELF_PROG_FLAG_EXEC      1
0890 #define ELF_PROG_FLAG_WRITE     2
0891 #define ELF_PROG_FLAG_READ      4
0892
0893
0894
0895
0896
0897
0898
0899
```

```
0900 // Blank page.
0901
0902
0903
0904
0905
0906
0907
0908
0909
0910
0911
0912
0913
0914
0915
0916
0917
0918
0919
0920
0921
0922
0923
0924
0925
0926
0927
0928
0929
0930
0931
0932
0933
0934
0935
0936
0937
0938
0939
0940
0941
0942
0943
0944
0945
0946
0947
0948
0949
```



```
0950 #include "asm.h"
0951
0952 # Start the first CPU: switch to 32-bit protected mode, jump into C.
0953 # The BIOS loads this code from the first sector of the hard disk into
0954 # memory at physical address 0x7c00 and starts executing in real mode
0955 # with %cs=0 %ip=7c00.
0956
0957 .set PROT_MODE_CSEG, 0x8         # kernel code segment selector
0958 .set PROT_MODE_DSEG, 0x10        # kernel data segment selector
0959 .set CR0_PE_ON,      0x1         # protected mode enable flag
0960
0961 .globl start
0962 start:
0963   .code16                     # Assemble for 16-bit mode
0964   cli                         # Disable interrupts
0965   cld                         # String operations increment
0966
0967   # Set up the important data segment registers (DS, ES, SS).
0968   xorw    %ax,%ax             # Segment number zero
0969   movw    %ax,%ds             # -> Data Segment
0970   movw    %ax,%es             # -> Extra Segment
0971   movw    %ax,%ss             # -> Stack Segment
0972
0973   # Enable A20:
0974   #   For backwards compatibility with the earliest PCs, physical
0975   #   address line 20 is tied low, so that addresses higher than
0976   #   1MB wrap around to zero by default.  This code undoes this.
0977 seta20.1:
0978   inb     $0x64,%al               # Wait for not busy
0979   testb   $0x2,%al
0980   jnz     seta20.1
0981
0982   movb    $0xd1,%al               # 0xd1 -> port 0x64
0983   outb    %al,$0x64
0984
0985 seta20.2:
0986   inb     $0x64,%al               # Wait for not busy
0987   testb   $0x2,%al
0988   jnz     seta20.2
0989
0990   movb    $0xdf,%al               # 0xdf -> port 0x60
0991   outb    %al,$0x60
0992
0993   # Switch from real to protected mode, using a bootstrap GDT
0994   # and segment translation that makes virtual addresses
0995   # identical to their physical addresses, so that the
0996   # effective memory map does not change during the switch.
0997   lgdt    gdtdesc
0998   movl    %cr0, %eax
0999   orl     $CR0_PE_ON, %eax
```

```
1000   movl    %eax, %cr0
1001
1002   # Jump to next instruction, but in 32-bit code segment.
1003   # Switches processor into 32-bit mode.
1004   ljmp    $PROT_MODE_CSEG, $protcseg
1005
1006   .code32                     # Assemble for 32-bit mode
1007 protcseg:
1008   # Set up the protected-mode data segment registers
1009   movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
1010   movw    %ax, %ds                # -> DS: Data Segment
1011   movw    %ax, %es                # -> ES: Extra Segment
1012   movw    %ax, %fs                # -> FS
1013   movw    %ax, %gs                # -> GS
1014   movw    %ax, %ss                # -> SS: Stack Segment
1015
1016   # Set up the stack pointer and call into C.
1017   movl    $start, %esp
1018   call    bootmain
1019
1020   # If bootmain returns (it shouldn't), loop.
1021 spin:
1022   jmp     spin
1023
1024 # Bootstrap GDT
1025 .p2align 2                                # force 4 byte alignment
1026 gdt:
1027   SEG_NULLASM                             # null seg
1028   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
1029   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
1030
1031 gdtdesc:
1032   .word   0x17                            # sizeof(gdt) - 1
1033   .long   gdt                             # address gdt
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
```



```
1050 #include "asm.h"
1051
1052 # Start an Application Processor. This must be placed on a 4KB boundary
1053 # somewhere in the 1st MB of conventional memory (APBOOTSTRAP). However,
1054 # due to some shortcuts below it's restricted further to within the 1st
1055 # 64KB. The AP starts in real-mode, with
1056 #   CS selector set to the startup memory address/16;
1057 #   CS base set to startup memory address;
1058 #   CS limit set to 64KB;
1059 #   CPL and IP set to 0.
1060 #
1061 # Bootothers (in main.c) starts each non-boot CPU in turn.
1062 # It puts the correct %esp in start-4,
1063 # and the place to jump to in start-8.
1064 #
1065 # This code is identical to bootasm.S except:
1066 #   - it does not need to enable A20
1067 #   - it uses the address at start-4 for the %esp
1068 #   - it jumps to the address at start-8 instead of calling bootmain
1069
1070 .set PROT_MODE_CSEG, 0x8         # kernel code segment selector
1071 .set PROT_MODE_DSEG, 0x10        # kernel data segment selector
1072 .set CR0_PE_ON,      0x1         # protected mode enable flag
1073
1074 .globl start
1075 start:
1076   .code16                     # Assemble for 16-bit mode
1077   cli                         # Disable interrupts
1078   cld                         # String operations increment
1079
1080   # Set up the important data segment registers (DS, ES, SS).
1081   xorw    %ax,%ax             # Segment number zero
1082   movw    %ax,%ds             # -> Data Segment
1083   movw    %ax,%es             # -> Extra Segment
1084   movw    %ax,%ss             # -> Stack Segment
1085
1086   # Switch from real to protected mode, using a bootstrap GDT
1087   # and segment translation that makes virtual addresses
1088   # identical to their physical addresses, so that the
1089   # effective memory map does not change during the switch.
1090   lgdt    gdtdesc
1091   movl    %cr0, %eax
1092   orl     $CR0_PE_ON, %eax
1093   movl    %eax, %cr0
1094
1095   # Jump to next instruction, but in 32-bit code segment.
1096   # Switches processor into 32-bit mode.
1097   ljmp    $PROT_MODE_CSEG, $protcseg
1098
1099
```

```
1100   .code32                     # Assemble for 32-bit mode
1101 protcseg:
1102   # Set up the protected-mode data segment registers
1103   movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
1104   movw    %ax, %ds                # -> DS: Data Segment
1105   movw    %ax, %es                # -> ES: Extra Segment
1106   movw    %ax, %fs                # -> FS
1107   movw    %ax, %gs                # -> GS
1108   movw    %ax, %ss                # -> SS: Stack Segment
1109
1110   movl    start-4, %esp
1111   movl    start-8, %eax
1112   jmp     *%eax
1113
1114 # Bootstrap GDT
1115 .p2align 2                                # force 4 byte alignment
1116 gdt:
1117   SEG_NULLASM                             # null seg
1118   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
1119   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
1120
1121 gdtdesc:
1122   .word   0x17                            # sizeof(gdt) - 1
1123   .long   gdt                             # address gdt
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
```



```
1150 // Boot loader.
1151 //
1152 // The BIOS loads boot sector (bootasm.S) from sector 0 of the disk
1153 // into memory and executes it.  The boot sector puts the processor
1154 // in 32-bit mode and calls bootmain below, which loads an ELF kernel
1155 // image from the disk starting at sector 1 and then jumps to the
1156 // kernel entry routine.
1157
1158 #include "types.h"
1159 #include "elf.h"
1160 #include "x86.h"
1161
1162 #define SECTSIZE  512
1163
1164 void readseg(uint, uint, uint);
1165
1166 void
1167 bootmain(void)
1168 {
1169   struct elfhdr *elf;
1170   struct proghdr *ph, *eph;
1171   void (*entry)(void);
1172
1173   elf = (struct elfhdr*)0x10000;  // scratch space
1174
1175   // Read 1st page off disk
1176   readseg((uint)elf, SECTSIZE*8, 0);
1177
1178   // Is this an ELF executable?
1179   if(elf->magic != ELF_MAGIC)
1180     goto bad;
1181
1182   // Load each program segment (ignores ph flags).
1183   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
1184   eph = ph + elf->phnum;
1185   for(; ph < eph; ph++)
1186     readseg(ph->va, ph->memsz, ph->offset);
1187
1188   // Call the entry point from the ELF header.
1189   // Does not return!
1190   entry = (void(*)(void))(elf->entry & 0xFFFFFF);
1191   entry();
1192
1193 bad:
1194   outw(0x8A00, 0x8A00);
1195   outw(0x8A00, 0x8E00);
1196   for(;;)
1197     ;
1198 }
1199
```

```
1200 void
1201 waitdisk(void)
1202 {
1203   // Wait for disk ready.
1204   while((inb(0x1F7) & 0xC0) != 0x40)
1205     ;
1206 }
1207
1208 // Read a single sector at offset into dst.
1209 void
1210 readsect(void *dst, uint offset)
1211 {
1212   // Issue command.
1213   waitdisk();
1214   outb(0x1F2, 1);   // count = 1
1215   outb(0x1F3, offset);
1216   outb(0x1F4, offset >> 8);
1217   outb(0x1F5, offset >> 16);
1218   outb(0x1F6, (offset >> 24) | 0xE0);
1219   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
1220
1221   // Read data.
1222   waitdisk();
1223   insl(0x1F0, dst, SECTSIZE/4);
1224 }
1225
1226 // Read 'count' bytes at 'offset' from kernel into virtual address 'va'.
1227 // Might copy more than asked.
1228 void
1229 readseg(uint va, uint count, uint offset)
1230 {
1231   uint eva;
1232
1233   va &= 0xFFFFFF;
1234   eva = va + count;
1235
1236   // Round down to sector boundary.
1237   va &= ~(SECTSIZE - 1);
1238
1239   // Translate from bytes to sectors; kernel starts at sector 1.
1240   offset = (offset / SECTSIZE) + 1;
1241
1242   // If this is too slow, we could read lots of sectors at a time.
1243   // We'd write more to memory than asked, but it doesn't matter --
1244   // we load in increasing order.
1245   for(; va < eva; va += SECTSIZE, offset++)
1246     readsect((uchar*)va, offset);
1247 }
1248
1249
```



```
1250 #include "types.h"
1251 #include "defs.h"
1252 #include "param.h"
1253 #include "mmu.h"
1254 #include "proc.h"
1255 #include "x86.h"
1256 #include "pci.h"
1257
1258 static void bootothers(void);
1259
1260 // Bootstrap processor starts running C code here.
1261 int
1262 main(void)
1263 {
1264   int i;
1265   static volatile int bcpu;  // cannot be on stack
1266   extern char edata[], end[];
1267
1268   // clear BSS
1269   memset(edata, 0, end - edata);
1270
1271   // Prevent release() from enabling interrupts.
1272   for(i=0; i<NCPU; i++)
1273     cpus[i].nlock = 1;
1274
1275 //  mp_init(); // collect info about this machine
1276   bcpu = mp_bcpu();
1277
1278   // Switch to bootstrap processor's stack
1279   asm volatile("movl %0, %%esp" : : "r" (cpus[bcpu].mpstack+MPSTACK-32));
1280   asm volatile("movl %0, %%ebp" : : "r" (cpus[bcpu].mpstack+MPSTACK));
1281
1282   lapic_init(bcpu);
1283   cprintf("\ncpu%d: starting xv6\n\n", cpu());
1284
1285   pinit();         // process table
1286   binit();         // buffer cache
1287   pic_init();      // interrupt controller
1288   ioapic_init();   // another interrupt controller
1289   kinit();         // physical memory allocator
1290   tvinit();        // trap vectors
1291   idtinit();       // interrupt descriptor table
1292   fileinit();      // file table
1293   iinit();         // inode cache
1294   setupsegs(0);    // segments & TSS
1295   console_init();  // I/O devices & their interrupts
1296   ide_init();      // disk
1297
1298
1299
```

```
1300   // lwIP init
1301   memp_init();
1302   mem_init();
1303   netif_init();
1304   pbuf_init();
1305   tcpip_init(0, 0);
1306
1307   pci_init();      // PCI
1308   bootothers();    // boot other CPUs
1309   if(!ismp)
1310     timer_init(); // uniprocessor timer
1311   userinit();      // first user process
1312
1313   // enable interrupts on this processor.
1314   cpus[cpu()].nlock--;
1315   if (cpus[cpu()].nlock == 0)
1316     sti();
1317
1318   scheduler();
1319 }
1320
1321 // Additional processors start here.
1322 static void
1323 mpmain(void)
1324 {
1325   cprintf("cpu%d: starting\n", cpu());
1326   idtinit();
1327   lapic_init(cpu());
1328   setupsegs(0);
1329   cpuid(0, 0, 0, 0, 0);  // memory barrier
1330   cpus[cpu()].booted = 1;
1331
1332   // Enable interrupts on this processor.
1333   cpus[cpu()].nlock--;
1334   if (cpus[cpu()].nlock == 0)
1335     sti();
1336
1337   scheduler();
1338 }
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
```



```
1350 static void
1351 bootothers(void)
1352 {
1353   extern uchar _binary_bootother_start[], _binary_bootother_size[];
1354   uchar *code;
1355   struct cpu *c;
1356
1357   // Write bootstrap code to unused memory at 0x7000.
1358   code = (uchar*)0x7000;
1359   memmove(code, _binary_bootother_start, (uint)_binary_bootother_size);
1360
1361   for(c = cpus; c < cpus+ncpu; c++){
1362     if(c == cpus+cpu())  // We've started already.
1363       continue;
1364
1365     // Fill in %esp, %eip and start code on cpu.
1366     *(void**)(code-4) = c->mpstack + MPSTACK;
1367     *(void**)(code-8) = mpmain;
1368     lapic_startap(c->apicid, (uint)code);
1369
1370     // Wait for cpu to get through bootstrap.
1371     while(c->booted == 0)
1372       ;
1373   }
1374 }
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
```

```
#ifndef XV6_SPINLOCK_H_
#define XV6_SPINLOCK_H_
// Mutual exclusion lock.
struct spinlock {
  uint locked;   // Is the lock held?

  // For debugging:
  char *name;    // Name of lock.
  int  cpu;      // The number of the cpu holding the lock.
  uint pcs[10];  // The call stack (an array of program counters)
                 // that locked the lock.
};
#endif // XV6_SPINLOCK_H_
```



```
// Blank page.
```

```
1500 // Mutual exclusion spin locks.
1501
1502 #include "types.h"
1503 #include "defs.h"
1504 #include "param.h"
1505 #include "x86.h"
1506 #include "mmu.h"
1507 #include "proc.h"
1508 #include "spinlock.h"
1509
1510 extern int use_console_lock;
1511
1512 void
1513 initlock(struct spinlock *lock, char *name)
1514 {
1515   lock->name = name;
1516   lock->locked = 0;
1517   lock->cpu = 0xffffffff;
1518 }
1519
1520 int int_enabled;
1521 // Acquire the lock.
1522 // Loops (spins) until the lock is acquired.
1523 // Holding a lock for a long time may cause
1524 // other CPUs to waste time spinning to acquire it.
1525 void
1526 acquire(struct spinlock *lock)
1527 {
1528   if(holding(lock))
1529   {
1530     cprintf("lock already acquired by:\n");
1531     int i;
1532     for (i=0; i<10; i++)
1533     {
1534         if ((lock->pcs[i] == 0) || (lock->pcs[i] == 0xffffffff))
1535             break;
1536         cprintf("0x%08x ", lock->pcs[i]);
1537     }
1538     cprintf("\n");
1539     panic("acquire");
1540   }
1541
1542 //  if(cpus[cpu()].nlock == 0)
1543 //  {
1544 //    int_enabled = 0;
1545     cli();
1546 //    cprintf("!D");
1547 // }
1548   cpus[cpu()].nlock++;
1549   cpus[cpu()].locks[cpus[cpu()].nlock] = lock;
```



```
1550   while(cmpxchg(0, 1, &lock->locked) == 1)
1551     ;
1552
1553   // Serialize instructions: now that lock is acquired, make sure
1554   // we wait for all pending writes from other processors.
1555   cpuid(0, 0, 0, 0, 0);  // memory barrier (see Ch 7, IA-32 manual vol 3)
1556
1557   // Record info about lock acquisition for debugging.
1558   // The +10 is only so that we can tell the difference
1559   // between forgetting to initialize lock->cpu
1560   // and holding a lock on cpu 0.
1561   lock->cpu = cpu() + 10;
1562   getcallerpcs(&lock, lock->pcs);
1563 }
1564
1565 // Release the lock.
1566 void
1567 release(struct spinlock *lock)
1568 {
1569   if(!holding(lock))
1570     panic("release");
1571
1572   lock->pcs[0] = 0;
1573   lock->cpu = 0xffffffff;
1574
1575   // Serialize instructions: before unlocking the lock, make sure
1576   // to flush any pending memory writes from this processor.
1577   cpuid(0, 0, 0, 0, 0);  // memory barrier (see Ch 7, IA-32 manual vol 3)
1578
1579   if (int_enabled)
1580       panic("release when interrupts enabled");
1581   lock->locked = 0;
1582   if(--cpus[cpu()].nlock == 0)
1583   {
1584 //      int_enabled = 1;
1585     sti();
1586  //   cprintf("!E");
1587   }
1588 }
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
```

```
1600 // Record the current call stack in pcs[] by following the %ebp chain.
1601 void
1602 getcallerpcs(void *v, uint pcs[])
1603 {
1604   uint *ebp;
1605   int i;
1606
1607   ebp = (uint*)v - 2;
1608   for(i = 0; i < 10; i++){
1609     if(ebp == 0 || ebp == (uint*)0xffffffff)
1610       break;
1611     pcs[i] = ebp[1];     // saved %eip
1612     ebp = (uint*)ebp[0]; // saved %ebp
1613   }
1614   for(; i < 10; i++)
1615     pcs[i] = 0;
1616 }
1617
1618 // Check whether this cpu is holding the lock.
1619 int
1620 holding(struct spinlock *lock)
1621 {
1622   return lock->locked && lock->cpu == cpu() + 10;
1623 }
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
```



```
1650 #ifndef XV6_SEM_H_
1651 #define XV6_SEM_H_
1652
1653 struct sem;
1654 typedef struct sem sem_t;
1655
1656 int sem_init(sem_t *sem, unsigned int value);
1657 int sem_destroy(sem_t *sem);
1658 void sem_post(sem_t *sem);
1659 void sem_wait(sem_t *sem);
1660 int sem_timedwait(sem_t *sem, int timo);
1661 int sem_trywait(sem_t *sem);
1662 int sem_value(sem_t *sem);
1663 int sem_size();
1664
1665 #endif // XV6_SEM_H_
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
```

```
1700 #include "assert.h"
1701 #include "types.h"
1702 #include "spinlock.h"
1703 #include "defs.h"
1704 #include "sem.h"
1705
1706 struct sem {
1707     struct spinlock lock;
1708     int val;
1709     int waiters;
1710 };
1711
1712 int sem_init(sem_t *sem, unsigned int value)
1713 {
1714     assert(value >= 0);
1715     initlock(&sem->lock, "sem lock");
1716     sem->val = value;
1717     sem->waiters = 0;
1718     return 0;
1719 }
1720
1721 int sem_destroy(sem_t *sem)
1722 {
1723     assert(sem->waiters == 0);
1724     return 0;
1725 }
1726
1727 void sem_post(sem_t *sem)
1728 {
1729     acquire(&sem->lock);
1730     sem->val++;
1731     if ((sem->waiters) && (sem->val > 0))
1732     {
1733         wakeup_one(sem); // XXX maybe wakeup?
1734     }
1735     release(&sem->lock);
1736 }
1737
1738 void sem_wait(sem_t *sem)
1739 {
1740     acquire(&sem->lock);
1741     while (sem->val == 0)
1742     {
1743         sem->waiters++;
1744         sleep(sem, &sem->lock);
1745         sem->waiters--;
1746     }
1747     sem->val--;
1748     release(&sem->lock);
1749 }
```



```
1750 int sem_timedwait(sem_t *sem, int timo)
1751 {
1752     int ret;
1753
1754     acquire(&sem->lock);
1755     for (ret = 0; sem->val == 0 && ret == 0;)
1756     {
1757         sem->waiters++;
1758         ret = msleep_spin(sem, &sem->lock, timo);
1759         sem->waiters--;
1760     }
1761     if (sem->val > 0)
1762     {
1763         sem->val--;
1764         ret = 0;
1765     }
1766     release(&sem->lock);
1767
1768     return ret;
1769 }
1770
1771 int sem_trywait(sem_t *sem)
1772 {
1773     int ret;
1774
1775     acquire(&sem->lock);
1776     if (sem->val > 0)
1777     {
1778         sem->val--;
1779         ret = 1;
1780     } else {
1781         ret = 0;
1782     }
1783     release(&sem->lock);
1784     return ret;
1785 }
1786
1787 int sem_value(sem_t *sem)
1788 {
1789     int ret;
1790
1791     acquire(&sem->lock);
1792     ret = sem->val;
1793     release(&sem->lock);
1794     return ret;
1795 }
1796
1797
1798
1799
```

```
1800 int sem_size()
1801 {
1802     return sizeof(struct sem);
1803 }
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
```



```
1850 #ifndef XV6_ASSERT_H_
1851 #define XV6_ASSERT_H_
1852
1853 #define assert(expr) \
1854     ((expr) ? (void)0 : __assert(__func__, __FILE__, \
1855         __LINE__, #expr))
1856
1857 void __assert(const char *, const char *, int, const char *);
1858
1859 #endif // XV6_ASSERT_H_
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
```

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

void __assert(const char *func, const char *file,
        int line, const char *expr)
{
    cprintf("assert failed: %s at %s:%d, expression %s\n",
            func, file, line, expr);
}
```



```
#include "spinlock.h"
#include "param.h"
#include "mmu.h"
// Segments in proc->gdt
#define SEG_KCODE 1  // kernel code
#define SEG_KDATA 2  // kernel data+stack
#define SEG_UCODE 3
#define SEG_UDATA 4
#define SEG_TSS   5  // this process's task state
#define NSEGS     6

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

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

// Per-process state
struct proc {
  char *mem;                // Start of process memory (kernel address)
  uint sz;                  // Size of process memory (bytes)
  char *kstack;             // Bottom of kernel stack for this process
  enum proc_state state;    // Process state
  int pid;                  // Process ID
  struct proc *parent;      // Parent process
  void *chan;               // If non-zero, sleeping on chan
  int killed;               // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;        // Current directory
  struct context context;   // Switch here to run process
  struct trapframe *tf;     // Trap frame for current interrupt
  char name[32];            // Process name (debugging)
  struct thread *thr;
};
```

```
2000 // Process memory is laid out contiguously, low addresses first:
2001 //   text
2002 //   original data and bss
2003 //   fixed-size stack
2004 //   expandable heap
2005
2006 // Arrange that cp point to the struct proc that this
2007 // CPU is currently running.  Such preprocessor
2008 // subterfuge can be confusing, but saves a lot of typing.
2009 extern struct proc *curproc[NCPU];  // Current (running) process per CPU
2010 #define cp (curproc[cpu()])  // Current process on this CPU
2011
2012
2013 #define MPSTACK 4096
2014
2015 // Per-CPU state
2016 struct cpu {
2017   uchar apicid;               // Local APIC ID
2018   struct context context;     // Switch here to enter scheduler
2019   struct taskstate ts;        // Used by x86 to find stack for interrupt
2020   struct segdesc gdt[NSEGS];  // x86 global descriptor table
2021   char mpstack[MPSTACK];      // Per-CPU startup stack
2022   volatile int booted;        // Has the CPU started?
2023   int nlock;                  // Number of locks currently held
2024 #define MAX_LOCKS   10
2025   struct spinlock *locks[MAX_LOCKS];
2026 };
2027
2028 extern struct cpu cpus[NCPU];
2029 extern int ncpu;
2030 extern struct spinlock proc_table_lock;
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
```



```
2050 #include "types.h"
2051 #include "defs.h"
2052 #include "param.h"
2053 #include "mmu.h"
2054 #include "x86.h"
2055 #include "proc.h"
2056 #include "spinlock.h"
2057 #include "thread.h"
2058
2059 struct spinlock proc_table_lock;
2060
2061 struct proc proc[NPROC];
2062 struct proc *curproc[NCPU];
2063 struct proc *initproc;
2064
2065 int nextpid = 1;
2066 extern void forkret(void);
2067 extern void forkret1(struct trapframe*);
2068
2069 void
2070 pinit(void)
2071 {
2072   initlock(&proc_table_lock, "proc_table");
2073 }
2074
2075 // Look in the process table for an UNUSED proc.
2076 // If found, change state to EMBRYO and return it.
2077 // Otherwise return 0.
2078 struct proc*
2079 allocproc(void)
2080 {
2081   int i;
2082   struct proc *p;
2083
2084   acquire(&proc_table_lock);
2085   for(i = 0; i < NPROC; i++){
2086     p = &proc[i];
2087     if(p->state == UNUSED){
2088       p->state = EMBRYO;
2089       p->pid = nextpid++;
2090       p->thr = 0;
2091       release(&proc_table_lock);
2092       return p;
2093     }
2094   }
2095   release(&proc_table_lock);
2096   return 0;
2097 }
2098
2099
```

```
2100 // Grow current process's memory by n bytes.
2101 // Return old size on success, -1 on failure.
2102 int
2103 growproc(int n)
2104 {
2105   char *newmem, *oldmem;
2106
2107   newmem = kalloc(cp->sz + n);
2108   if(newmem == 0)
2109     return -1;
2110   memmove(newmem, cp->mem, cp->sz);
2111   memset(newmem + cp->sz, 0, n);
2112   oldmem = cp->mem;
2113   cp->mem = newmem;
2114   kfree(oldmem, cp->sz);
2115   cp->sz += n;
2116   return cp->sz - n;
2117 }
2118
2119 // Set up CPU's segment descriptors and task state for a given process.
2120 // If p==0, set up for "idle" state for when scheduler() is running.
2121 void
2122 setupsegs(struct proc *p)
2123 {
2124   struct cpu *c;
2125
2126   c = &cpus[cpu()];
2127   c->ts.ss0 = SEG_KDATA << 3;
2128   if(p)
2129     c->ts.esp0 = (uint)(p->kstack + KSTACKSIZE);
2130   else
2131     c->ts.esp0 = 0xffffffff;
2132
2133   c->gdt[0] = SEG_NULL;
2134   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0x100000 + 64*1024-1, 0);
2135   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
2136   c->gdt[SEG_TSS] = SEG16(STS_T32A, (uint)&c->ts, sizeof(c->ts)-1, 0);
2137   c->gdt[SEG_TSS].s = 0;
2138   if(p){
2139     c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, (uint)p->mem, p->sz-1, DPL_USER);
2140     c->gdt[SEG_UDATA] = SEG(STA_W, (uint)p->mem, p->sz-1, DPL_USER);
2141   } else {
2142     c->gdt[SEG_UCODE] = SEG_NULL;
2143     c->gdt[SEG_UDATA] = SEG_NULL;
2144   }
2145
2146   lgdt(c->gdt, sizeof(c->gdt));
2147   ltr(SEG_TSS << 3);
2148 }
2149
```



```
2150 // Create a new process copying p as the parent.
2151 // Sets up stack to return as if from system call.
2152 // Caller must set state of returned proc to RUNNABLE.
2153 struct proc*
2154 copyproc(struct proc *p)
2155 {
2156   int i;
2157   struct proc *np;
2158
2159   // Allocate process.
2160   if((np = allocproc()) == 0)
2161     return 0;
2162
2163   // Allocate kernel stack.
2164   if((np->kstack = kalloc(KSTACKSIZE)) == 0){
2165     np->state = UNUSED;
2166     return 0;
2167   }
2168   np->tf = (struct trapframe*)(np->kstack + KSTACKSIZE) - 1;
2169
2170   if(p){  // Copy process state from p.
2171     np->parent = p;
2172     memmove(np->tf, p->tf, sizeof(*np->tf));
2173
2174     np->sz = p->sz;
2175     if((np->mem = kalloc(np->sz)) == 0){
2176       kfree(np->kstack, KSTACKSIZE);
2177       np->kstack = 0;
2178       np->state = UNUSED;
2179       return 0;
2180     }
2181     memmove(np->mem, p->mem, np->sz);
2182
2183     for(i = 0; i < NOFILE; i++)
2184       if(p->ofile[i])
2185         np->ofile[i] = filedup(p->ofile[i]);
2186     np->cwd = idup(p->cwd);
2187     np->thr = p->thr;
2188   }
2189
2190   // Set up new context to start executing at forkret (see below).
2191   memset(&np->context, 0, sizeof(np->context));
2192   np->context.eip = (uint)forkret;
2193   np->context.esp = (uint)np->tf;
2194
2195   // Clear %eax so that fork system call returns 0 in child.
2196   np->tf->eax = 0;
2197   return np;
2198 }
2199
```

```
2200 // Set up first user process.
2201 void
2202 userinit(void)
2203 {
2204   struct proc *p;
2205   extern uchar _binary_initcode_start[], _binary_initcode_size[];
2206
2207   p = copyproc(0);
2208   p->sz = PAGE;
2209   p->mem = kalloc(p->sz);
2210   p->cwd = namei("/");
2211   memset(p->tf, 0, sizeof(*p->tf));
2212   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2213   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2214   p->tf->es = p->tf->ds;
2215   p->tf->ss = p->tf->ds;
2216   p->tf->eflags = FL_IF;
2217   p->tf->esp = p->sz;
2218
2219   // Make return address readable; needed for some gcc.
2220   p->tf->esp -= 4;
2221   *(uint*)(p->mem + p->tf->esp) = 0xefefefef;
2222
2223   // On entry to user space, start executing at beginning of initcode.S.
2224   p->tf->eip = 0;
2225   memmove(p->mem, _binary_initcode_start, (int)_binary_initcode_size);
2226   safestrcpy(p->name, "initcode", sizeof(p->name));
2227   p->state = RUNNABLE;
2228
2229   initproc = p;
2230 }
2231
2232 // Per-CPU process scheduler.
2233 // Each CPU calls scheduler() after setting itself up.
2234 // Scheduler never returns.  It loops, doing:
2235 //  - choose a process to run
2236 //  - swtch to start running that process
2237 //  - eventually that process transfers control
2238 //      via swtch back to the scheduler.
2239 void
2240 scheduler(void)
2241 {
2242   struct proc *p;
2243   int i;
2244
2245   for(;;){
2246     // Loop over process table looking for process to run.
2247     acquire(&proc_table_lock);
2248
2249
```



```
2250     for(i = 0; i < NPROC; i++){
2251       p = &proc[i];
2252       if((p->state != RUNNABLE) && (p->state != MSLEEPING))
2253         continue;
2254
2255 //      cprintf("switch to %s eip: 0x%08x...\n", p->name, p->context.eip);
2256       // Switch to chosen process.  It is the process's job
2257       // to release proc_table_lock and then reacquire it
2258       // before jumping back to us.
2259       cp = p;
2260       setupsegs(p);
2261       if (p->state == RUNNABLE)
2262         p->state = RUNNING;
2263       swtch(&cpus[cpu()].context, &p->context);
2264
2265 //      cprintf("done.\n");
2266       // Process is done running for now.
2267       // It should have changed its p->state before coming back.
2268       cp = 0;
2269       setupsegs(0);
2270     }
2271
2272     release(&proc_table_lock);
2273   }
2274 }
2275
2276 // Enter scheduler.  Must already hold proc_table_lock
2277 // and have changed curproc[cpu()]->state.
2278 void
2279 sched(void)
2280 {
2281   if(cp->state == RUNNING)
2282     panic("sched running");
2283   if(!holding(&proc_table_lock))
2284     panic("sched proc_table_lock");
2285 //  if(cpus[cpu()].nlock != 1)
2286 //    panic("sched locks");
2287
2288   swtch(&cp->context, &cpus[cpu()].context);
2289 }
2290
2291 // Give up the CPU for one scheduling round.
2292 void
2293 yield(void)
2294 {
2295   acquire(&proc_table_lock);
2296   cp->state = RUNNABLE;
2297   sched();
2298   release(&proc_table_lock);
2299 }
```

```
2300 // A fork child's very first scheduling by scheduler()
2301 // will swtch here.  "Return" to user space.
2302 void
2303 forkret(void)
2304 {
2305   // Still holding proc_table_lock from scheduler.
2306   release(&proc_table_lock);
2307
2308   // Jump into assembly, never to return.
2309   forkret1(cp->tf);
2310 }
2311
2312 int
2313 msleep_spin(void *chan, struct spinlock *lk, int timo)
2314 {
2315     uint32_t s = millitime();
2316     uint32_t p = s;
2317     int ret = 1; // Time Out
2318     s += timo;
2319
2320     if (cp == 0)
2321         panic("msleep with cp == 0");
2322     if (lk == 0)
2323         panic("msleep without lock");
2324
2325     if (lk != &proc_table_lock)
2326     {
2327         acquire(&proc_table_lock);
2328         release(lk);
2329     }
2330
2331     cp->chan = chan;
2332     cp->state = MSLEEPING;
2333     while (p < s)
2334     {
2335         sched();
2336         if (cp->state == RUNNING)
2337         {
2338             ret = 0;
2339             break;
2340         }
2341         p = millitime();
2342     }
2343     cp->chan = 0;
2344     cp->state = RUNNING;
2345
2346
2347
2348
2349
```



```
2350     if (lk != &proc_table_lock)
2351     {
2352         release(&proc_table_lock);
2353         acquire(lk);
2354     }
2355     return ret;
2356 }
2357
2358 // Atomically release lock and sleep on chan.
2359 // Reacquires lock when reawakened.
2360 void
2361 sleep(void *chan, struct spinlock *lk)
2362 {
2363   if(cp == 0)
2364     panic("sleep");
2365
2366   if(lk == 0)
2367     panic("sleep without lk");
2368
2369   // Must acquire proc_table_lock in order to
2370   // change p->state and then call sched.
2371   // Once we hold proc_table_lock, we can be
2372   // guaranteed that we won't miss any wakeup
2373   // (wakeup runs with proc_table_lock locked),
2374   // so it's okay to release lk.
2375   if(lk != &proc_table_lock){
2376     acquire(&proc_table_lock);
2377     release(lk);
2378   }
2379
2380   // Go to sleep.
2381   cp->chan = chan;
2382   cp->state = SLEEPING;
2383   sched();
2384
2385   // Tidy up.
2386   cp->chan = 0;
2387
2388   // Reacquire original lock.
2389   if(lk != &proc_table_lock){
2390     release(&proc_table_lock);
2391     acquire(lk);
2392   }
2393 }
2394
2395
2396
2397
2398
2399
```

```
2400 // Wake up all processes sleeping on chan.
2401 // Proc_table_lock must be held.
2402 static void
2403 wakeup1(void *chan)
2404 {
2405   struct proc *p;
2406
2407   for(p = proc; p < &proc[NPROC]; p++)
2408     if(((p->state == SLEEPING) ||
2409                 (p->state == MSLEEPING)) && p->chan == chan)
2410       p->state = RUNNABLE;
2411 }
2412
2413 // Wake up all processes sleeping on chan.
2414 // Proc_table_lock must be held.
2415 static void
2416 wakeup_one1(void *chan)
2417 {
2418   struct proc *p;
2419
2420   for(p = proc; p < &proc[NPROC]; p++)
2421     if(((p->state == SLEEPING) ||
2422                 (p->state == MSLEEPING)) && p->chan == chan)
2423     {
2424       p->state = RUNNABLE;
2425       break;
2426     }
2427 }
2428
2429 void
2430 wakeup_one(void *chan)
2431 {
2432   acquire(&proc_table_lock);
2433   wakeup_one1(chan);
2434   release(&proc_table_lock);
2435 }
2436
2437 // Wake up all processes sleeping on chan.
2438 // Proc_table_lock is acquired and released.
2439 void
2440 wakeup(void *chan)
2441 {
2442   acquire(&proc_table_lock);
2443   wakeup1(chan);
2444   release(&proc_table_lock);
2445 }
2446
2447
2448
2449
```



```
2450 // Kill the process with the given pid.
2451 // Process won't actually exit until it returns
2452 // to user space (see trap in trap.c).
2453 int
2454 kill(int pid)
2455 {
2456   struct proc *p;
2457
2458   acquire(&proc_table_lock);
2459   for(p = proc; p < &proc[NPROC]; p++){
2460     if(p->pid == pid){
2461       p->killed = 1;
2462       // Wake process from sleep if necessary.
2463       if(p->state == SLEEPING)
2464         p->state = RUNNABLE;
2465       release(&proc_table_lock);
2466       return 0;
2467     }
2468   }
2469   release(&proc_table_lock);
2470   return -1;
2471 }
2472
2473 // Exit the current process.  Does not return.
2474 // Exited processes remain in the zombie state
2475 // until their parent calls wait() to find out they exited.
2476 void
2477 exit(void)
2478 {
2479   struct proc *p;
2480   int fd;
2481
2482   if(cp == initproc)
2483     panic("init exiting");
2484
2485   // Close all open files.
2486   for(fd = 0; fd < NOFILE; fd++){
2487     if(cp->ofile[fd]){
2488       fileclose(cp->ofile[fd]);
2489       cp->ofile[fd] = 0;
2490     }
2491   }
2492
2493   iput(cp->cwd);
2494   cp->cwd = 0;
2495
2496   acquire(&proc_table_lock);
2497
2498   // Parent might be sleeping in proc_wait.
2499   wakeup1(cp->parent);
```

```
2500   // Pass abandoned children to init.
2501   for(p = proc; p < &proc[NPROC]; p++){
2502     if(p->parent == cp){
2503       p->parent = initproc;
2504       if(p->state == ZOMBIE)
2505         wakeup1(initproc);
2506     }
2507   }
2508
2509   if (cp->thr)
2510       kproc_free(cp->thr);
2511
2512   // Jump into the scheduler, never to return.
2513   cp->killed = 0;
2514   cp->state = ZOMBIE;
2515   cp->thr = 0;
2516   sched();
2517   panic("zombie exit");
2518 }
2519
2520 // Wait for a child process to exit and return its pid.
2521 // Return -1 if this process has no children.
2522 int
2523 wait(void)
2524 {
2525   struct proc *p;
2526   int i, havekids, pid;
2527
2528   acquire(&proc_table_lock);
2529   for(;;){
2530     // Scan through table looking for zombie children.
2531     havekids = 0;
2532     for(i = 0; i < NPROC; i++){
2533       p = &proc[i];
2534       if(p->state == UNUSED)
2535         continue;
2536       if(p->parent == cp){
2537         if(p->state == ZOMBIE){
2538           // Found one.
2539 #ifdef PROC_DEBUG
2540           cprintf("wait: freeing ZOMBIE %d %s\n", p->pid, p->name);
2541           cprintf("      freeing mem\n");
2542 #endif
2543           if (p->sz != 0) // kernel thread doesn't have mem
2544             kfree(p->mem, p->sz);
2545 #ifdef PROC_DEBUG
2546           cprintf("      freeing kstack\n");
2547 #endif
2548           kfree(p->kstack, KSTACKSIZE);
2549           pid = p->pid;
```



```
2550           p->state = UNUSED;
2551           p->pid = 0;
2552           p->parent = 0;
2553           p->name[0] = 0;
2554           release(&proc_table_lock);
2555           return pid;
2556         }
2557         havekids = 1;
2558       }
2559     }
2560
2561     // No point waiting if we don't have any children.
2562     if(!havekids || cp->killed){
2563       release(&proc_table_lock);
2564       return -1;
2565     }
2566
2567     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2568     sleep(cp, &proc_table_lock);
2569   }
2570 }
2571
2572 // Print a process listing to console.  For debugging.
2573 // Runs when user types ^P on console.
2574 // No lock to avoid wedging a stuck machine further.
2575 void
2576 procdump(void)
2577 {
2578   static char *states[] = {
2579   [UNUSED]    "unused",
2580   [EMBRYO]    "embryo",
2581   [SLEEPING]  "sleep ",
2582   [RUNNABLE]  "runble",
2583   [RUNNING]   "run   ",
2584   [ZOMBIE]    "zombie",
2585   [MSLEEPING] "msleep"
2586   };
2587   int i, j;
2588   struct proc *p;
2589   char *state;
2590   uint pc[10];
2591
2592   for(i = 0; i < NPROC; i++){
2593     p = &proc[i];
2594     if(p->state == UNUSED)
2595       continue;
2596     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
2597       state = states[p->state];
2598     else
2599       state = "???";
```

```
2600     cprintf("%d %s %s", p->pid, state, p->name);
2601     if((p->state == SLEEPING) || (p->state == MSLEEPING)){
2602       getcallerpcs((uint*)p->context.ebp+2, pc);
2603       for(j=0; j<10 && pc[j] != 0; j++)
2604         cprintf(" %p", pc[j]);
2605     }
2606     cprintf("\n");
2607   }
2608 }
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
```



```
2650 #   void swtch(struct context *old, struct context *new);
2651 #
2652 # Save current register context in old
2653 # and then load register context from new.
2654
2655 .globl swtch
2656 swtch:
2657   # Save old registers
2658   movl 4(%esp), %eax
2659
2660   popl 0(%eax)  # %eip
2661   movl %esp, 4(%eax)
2662   movl %ebx, 8(%eax)
2663   movl %ecx, 12(%eax)
2664   movl %edx, 16(%eax)
2665   movl %esi, 20(%eax)
2666   movl %edi, 24(%eax)
2667   movl %ebp, 28(%eax)
2668
2669   # Load new registers
2670   movl 4(%esp), %eax  # not 8(%esp) - popped return address above
2671
2672   movl 28(%eax), %ebp
2673   movl 24(%eax), %edi
2674   movl 20(%eax), %esi
2675   movl 16(%eax), %edx
2676   movl 12(%eax), %ecx
2677   movl 8(%eax), %ebx
2678   movl 4(%eax), %esp
2679   pushl 0(%eax)  # %eip
2680
2681   ret
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
```

```
2700 // Physical memory allocator, intended to allocate
2701 // memory for user processes. Allocates in 4096-byte "pages".
2702 // Free list is kept sorted and combines adjacent pages into
2703 // long runs, to make it easier to allocate big segments.
2704 // One reason the page size is 4k is that the x86 segment size
2705 // granularity is 4k.
2706
2707 #include "types.h"
2708 #include "defs.h"
2709 #include "param.h"
2710 #include "spinlock.h"
2711
2712 //#define MEM_DEBUG 1
2713 struct spinlock kalloc_lock;
2714
2715 struct run {
2716   struct run *next;
2717   int len; // bytes
2718 };
2719 struct run *freelist;
2720
2721 // Initialize free list of physical pages.
2722 // This code cheats by just considering one megabyte of
2723 // pages after _end.  Real systems would determine the
2724 // amount of memory available in the system and use it all.
2725 void
2726 kinit(void)
2727 {
2728   extern int end;
2729   uint mem;
2730   char *start;
2731
2732   initlock(&kalloc_lock, "kalloc");
2733   start = (char*) &end;
2734   start = (char*) (((uint)start + PAGE) & ~(PAGE-1));
2735   mem = 256; // assume computer has 256 pages of RAM
2736   cprintf("mem = %d\n", mem * PAGE);
2737   kfree(start, mem * PAGE);
2738 }
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
```



```
2750 // Free the len bytes of memory pointed at by v,
2751 // which normally should have been returned by a
2752 // call to kalloc(len).  (The exception is when
2753 // initializing the allocator; see kinit above.)
2754 void
2755 kfree(char *v, int len)
2756 {
2757   struct run *r, *rend, **rp, *p, *pend;
2758
2759 #ifdef MEM_DEBUG
2760   cprintf("kfree: %d pages\n", len / PAGE);
2761 #endif
2762   if(len <= 0 || len % PAGE)
2763   {
2764     cprintf("kfree: length = %d\n", len);
2765     panic("kfree");
2766   }
2767
2768   // Fill with junk to catch dangling refs.
2769   memset(v, 1, len);
2770
2771   acquire(&kalloc_lock);
2772   p = (struct run*)v;
2773   pend = (struct run*)(v + len);
2774   for(rp=&freelist; (r=*rp) != 0 && r <= pend; rp=&r->next){
2775     rend = (struct run*)((char*)r + r->len);
2776     if(r <= p && p < rend)
2777       panic("freeing free page");
2778     if(pend == r){  // p next to r: replace r with p
2779       p->len = len + r->len;
2780       p->next = r->next;
2781       *rp = p;
2782       goto out;
2783     }
2784     if(rend == p){  // r next to p: replace p with r
2785       r->len += len;
2786       if(r->next && r->next == pend){  // r now next to r->next?
2787         r->len += r->next->len;
2788         r->next = r->next->next;
2789       }
2790       goto out;
2791     }
2792   }
2793   // Insert p before r in list.
2794   p->len = len;
2795   p->next = r;
2796   *rp = p;
2797
2798
2799
```

```
2800  out:
2801    release(&kalloc_lock);
2802  }
2803
2804  // Allocate n bytes of physical memory.
2805  // Returns a kernel-segment pointer.
2806  // Returns 0 if the memory cannot be allocated.
2807  char*
2808  kalloc(int n)
2809  {
2810    char *p;
2811    struct run *r, **rp;
2812
2813  #ifdef MEM_DEBUG
2814    cprintf("kalloc: %d pages\n", n / PAGE);
2815    printstack();
2816  #endif
2817    if(n % PAGE || n <= 0)
2818      panic("kalloc");
2819
2820    acquire(&kalloc_lock);
2821    for(rp=&freelist; (r=*rp) != 0; rp=&r->next){
2822      if(r->len == n){
2823        *rp = r->next;
2824        release(&kalloc_lock);
2825        return (char*)r;
2826      }
2827      if(r->len > n){
2828        r->len -= n;
2829        p = (char*)r + r->len;
2830        release(&kalloc_lock);
2831        return p;
2832      }
2833    }
2834    release(&kalloc_lock);
2835
2836    cprintf("kalloc: out of memory\n");
2837    return 0;
2838  }
2839
2840  void *
2841  kmalloc(int n)
2842  {
2843  #ifdef MEM_DEBUG
2844      cprintf("kmalloc: %d bytes\n", n);
2845  #endif
2846      return kalloc(((n - 1) / PAGE + 1) * PAGE);
2847  }
2848
2849
```



```
2850  void
2851  kmfree(void *p, int n)
2852  {
2853  #ifdef MEM_DEBUG
2854      cprintf("kmfree: %d bytes\n", n);
2855  #endif
2856      return kfree(p, ((n - 1) / PAGE + 1) * PAGE);
2857  }
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
```

```
2900 #ifndef XV6_THREAD_H_
2901 #define XV6_THREAD_H_
2902 typedef struct thread * kproc_t;
2903
2904 #ifndef SYS_TIMEOUTS_DEFINED
2905 #define SYS_TIMEOUTS_DEFINED
2906
2907 struct sys_timeouts {
2908     struct sys_timeout *next;
2909 };
2910 #endif // SYS_TIMEOUTS_DEFINED
2911
2912 struct thread {
2913     struct proc *p;
2914     void *data;
2915     struct sys_timeouts timeouts;
2916 };
2917
2918 kproc_t kproc_start(void (* proc)(void *arg),
2919         void *arg, int prio, void *data, char *name);
2920 void kproc_free(kproc_t thread);
2921
2922 #endif // XV6_THREAD_H_
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
```



```
2950 #include "types.h"
2951 #include "defs.h"
2952 #include "thread.h"
2953 #include "param.h"
2954 #include "mmu.h"
2955 #include "proc.h"
2956
2957 void thread_wrap(void (* thread)(void *arg), void *arg);
2958
2959 extern struct proc *initproc;
2960
2961 kproc_t kproc_start(void (* proc)(void *arg),
2962         void *arg, int prio, void *data, char *name)
2963 {
2964     kproc_t thr = (kproc_t)kmalloc(sizeof(struct thread));
2965     if (!thr)
2966         return NULL;
2967     thr->p = allocproc();
2968     struct proc *np = thr->p;
2969     if (!np)
2970         return NULL;
2971     if((np->kstack = kmalloc(KSTACKSIZE)) == 0){
2972       np->state = UNUSED;
2973       return NULL;
2974     }
2975     np->thr = thr;
2976     np->parent = initproc;
2977     np->sz = 0;
2978     np->chan = 0;
2979     np->killed = 0;
2980     thr->data = data;
2981     memset(&np->context, 0, sizeof(np->context));
2982     thr->timeouts.next = 0;
2983     if (name == 0)
2984         safestrcpy(np->name,"[kernel thread]",sizeof(np->name));
2985     else
2986         safestrcpy(np->name, name, sizeof(np->name));
2987     np->context.eip = (uint)thread_wrap;
2988     np->context.esp = (uint)np->kstack + KSTACKSIZE - 1;
2989     *(void**)(np->context.esp+8) = arg;
2990     *(void**)(np->context.esp+4) = proc;
2991     *(void**)(np->context.esp) = exit;
2992     np->cwd = namei("/");
2993     np->state = RUNNABLE;
2994     return thr;
2995 }
2996
2997
2998
2999
```

```
3000 void kproc_free(kproc_t thread)
3001 {
3002     struct proc *p = thread->p;
3003     p->thr = 0;
3004     kmfree(thread, sizeof(struct thread));
3005 }
3006
3007 void thread_wrap(void (* thread)(void *arg), void *arg)
3008 {
3009     release(&proc_table_lock);
3010     thread(arg);
3011 }
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
```



```
3050 // x86 trap and interrupt constants.
3051
3052 // Processor-defined:
3053 #define T_DIVIDE         0      // divide error
3054 #define T_DEBUG          1      // debug exception
3055 #define T_NMI            2      // non-maskable interrupt
3056 #define T_BRKPT          3      // breakpoint
3057 #define T_OFLOW          4      // overflow
3058 #define T_BOUND          5      // bounds check
3059 #define T_ILLOP          6      // illegal opcode
3060 #define T_DEVICE         7      // device not available
3061 #define T_DBLFLT         8      // double fault
3062 // #define T_COPROC      9      // reserved (not used since 486)
3063 #define T_TSS           10      // invalid task switch segment
3064 #define T_SEGNP         11      // segment not present
3065 #define T_STACK         12      // stack exception
3066 #define T_GPFLT         13      // general protection fault
3067 #define T_PGFLT         14      // page fault
3068 // #define T_RES        15      // reserved
3069 #define T_FPERR         16      // floating point error
3070 #define T_ALIGN         17      // aligment check
3071 #define T_MCHK          18      // machine check
3072 #define T_SIMDERR       19      // SIMD floating point error
3073
3074 // These are arbitrarily chosen, but with care not to overlap
3075 // processor defined exceptions or interrupt vectors.
3076 #define T_SYSCALL       48      // system call
3077 #define T_DEFAULT      500      // catchall
3078
3079 #define IRQ_OFFSET      32      // IRQ 0 corresponds to int IRQ_OFFSET
3080
3081 #define IRQ_TIMER        0
3082 #define IRQ_KBD          1
3083 #define IRQ_IDE         14
3084 #define IRQ_IDE_2       15
3085 #define IRQ_ERROR       19
3086 #define IRQ_SPURIOUS    31
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
```

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

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

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

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

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



```
.text

.set SEG_KDATA_SEL, 0x10   # selector for SEG_KDATA

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

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

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

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

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

```
3200 #include "types.h"
3201 #include "defs.h"
3202 #include "param.h"
3203 #include "mmu.h"
3204 #include "proc.h"
3205 #include "x86.h"
3206 #include "traps.h"
3207 #include "spinlock.h"
3208 #include "picirq.h"
3209
3210 // Interrupt descriptor table (shared by all CPUs).
3211 struct gatedesc idt[256];
3212 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
3213 struct spinlock tickslock;
3214 int ticks;
3215
3216 void
3217 tvinit(void)
3218 {
3219   int i;
3220
3221   for(i = 0; i < 256; i++)
3222     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3223   SETGATE(idt[T_SYSCALL], 0, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3224
3225   initlock(&tickslock, "time");
3226 }
3227
3228 void
3229 idtinit(void)
3230 {
3231   lidt(idt, sizeof(idt));
3232 }
3233
3234 void
3235 trap(struct trapframe *tf)
3236 {
3237   if(tf->trapno == T_SYSCALL){
3238     if(cp->killed)
3239       exit();
3240     cp->tf = tf;
3241     syscall();
3242     if(cp->killed)
3243       exit();
3244     return;
3245   }
3246
3247   // Increment nlock to make sure interrupts stay off
3248   // during interrupt handler.  Decrement before returning.
3249   cpus[cpu()].nlock++;
```



```
3250   switch(tf->trapno){
3251   case IRQ_OFFSET + IRQ_TIMER:
3252     if(cpu() == 0){
3253       acquire(&tickslock);
3254       ticks++;
3255       wakeup(&ticks);
3256       release(&tickslock);
3257     }
3258     lapic_eoi();
3259     break;
3260   case IRQ_OFFSET + IRQ_IDE:
3261     ide_intr();
3262     lapic_eoi();
3263     break;
3264   case IRQ_OFFSET + IRQ_KBD:
3265     kbd_intr();
3266     lapic_eoi();
3267     break;
3268   case IRQ_OFFSET + IRQ_SPURIOUS:
3269     cprintf("spurious interrupt from cpu %d eip %x\n", cpu(), tf->eip);
3270     lapic_eoi();
3271     break;
3272
3273   default:
3274 //    if (tf->trapno <= 30)
3275 //    {
3276     if ((tf->trapno >= IRQ_OFFSET) && (tf->trapno <= IRQ_MAX))
3277         if (irq_handler[tf->trapno - IRQ_OFFSET])
3278         {
3279 //            cprintf("IRQ %d goes to handler\n", tf->trapno);
3280             irq_handler[tf->trapno - IRQ_OFFSET](tf);
3281             lapic_eoi();
3282             break;
3283         }
3284     if(cp == 0){
3285       // Otherwise it's our mistake.
3286       cprintf("unexpected trap %d from cpu %d eip %x\n",
3287               tf->trapno, cpu(), tf->eip);
3288 //      panic("trap");
3289     }
3290     else
3291     {
3292         // Assume process divided by zero or dereferenced null, etc.
3293         cprintf("pid %d %s: trap %d err %d"
3294                 "on cpu %d eip %x -- kill proc\n",
3295             cp->pid, cp->name, tf->trapno, tf->err, cpu(), tf->eip);
3296         cp->killed = 1;
3297     }
3298 //    }
3299   }
```

```
3300   cpus[cpu()].nlock--;
3301
3302   // Force process exit if it has been killed and is in user space.
3303   // (If it is still executing in the kernel, let it keep running
3304   // until it gets to the regular system call return.)
3305   if(cp && cp->killed && (tf->cs&3) == DPL_USER)
3306     exit();
3307
3308   // Force process to give up CPU on clock tick.
3309   // If interrupts were on while locks held, would need to check nlock.
3310   if(cp && cp->state == RUNNING && tf->trapno == IRQ_OFFSET+IRQ_TIMER)
3311     yield();
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 // System call numbers
3351 #define SYS_fork    1
3352 #define SYS_exit    2
3353 #define SYS_wait    3
3354 #define SYS_pipe    4
3355 #define SYS_write   5
3356 #define SYS_read    6
3357 #define SYS_close   7
3358 #define SYS_kill    8
3359 #define SYS_exec    9
3360 #define SYS_open   10
3361 #define SYS_mknod  11
3362 #define SYS_unlink 12
3363 #define SYS_fstat  13
3364 #define SYS_link   14
3365 #define SYS_mkdir  15
3366 #define SYS_chdir  16
3367 #define SYS_dup    17
3368 #define SYS_getpid 18
3369 #define SYS_sbrk   19
3370 #define SYS_sleep  20
3371 #define SYS_upmsec 21
3372 #define SYS_socket 22
3373 #define SYS_bind   23
3374 #define SYS_listen 24
3375 #define SYS_accept 25
3376 #define SYS_recv   26
3377 #define SYS_recvfrom 27
3378 #define SYS_send    28
3379 #define SYS_sendto  29
3380 #define SYS_shutdown 30
3381 #define SYS_getsockopt 31
3382 #define SYS_setsockopt 32
3383 #define SYS_sockclose 33
3384 #define SYS_connect 34
3385 #define SYS_getpeername 35
3386 #define SYS_getsockname 36
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399
```

```
3400 #include "types.h"
3401 #include "defs.h"
3402 #include "param.h"
3403 #include "mmu.h"
3404 #include "proc.h"
3405 #include "x86.h"
3406 #include "syscall.h"
3407
3408 // User code makes a system call with INT T_SYSCALL.
3409 // System call number in %eax.
3410 // Arguments on the stack, from the user call to the C
3411 // library system call function. The saved user %esp points
3412 // to a saved program counter, and then the first argument.
3413
3414 // Fetch the int at addr from process p.
3415 int
3416 fetchint(struct proc *p, uint addr, int *ip)
3417 {
3418   if(addr >= p->sz || addr+4 > p->sz)
3419     return -1;
3420   *ip = *(int*)(p->mem + addr);
3421   return 0;
3422 }
3423
3424 // Fetch the nul-terminated string at addr from process p.
3425 // Doesn't actually copy the string - just sets *pp to point at it.
3426 // Returns length of string, not including nul.
3427 int
3428 fetchstr(struct proc *p, uint addr, char **pp)
3429 {
3430   char *s, *ep;
3431
3432   if(addr >= p->sz)
3433     return -1;
3434   *pp = p->mem + addr;
3435   ep = p->mem + p->sz;
3436   for(s = *pp; s < ep; s++)
3437     if(*s == 0)
3438       return s - *pp;
3439   return -1;
3440 }
3441
3442 // Fetch the nth 32-bit system call argument.
3443 int
3444 argint(int n, int *ip)
3445 {
3446   return fetchint(cp, cp->tf->esp + 4 + 4*n, ip);
3447 }
3448
3449
```



```
3450 // Fetch the nth word-sized system call argument as a pointer
3451 // to a block of memory of size n bytes.  Check that the pointer
3452 // lies within the process address space.
3453 int
3454 argptr(int n, char **pp, int size)
3455 {
3456   int i;
3457
3458   if(argint(n, &i) < 0)
3459     return -1;
3460   if((uint)i >= cp->sz || (uint)i+size >= cp->sz)
3461     return -1;
3462   *pp = cp->mem + i;
3463   return 0;
3464 }
3465
3466 // Fetch the nth word-sized system call argument as a string pointer.
3467 // Check that the pointer is valid and the string is nul-terminated.
3468 // (There is no shared writable memory, so the string can't change
3469 // between this check and being used by the kernel.)
3470 int
3471 argstr(int n, char **pp)
3472 {
3473   int addr;
3474   if(argint(n, &addr) < 0)
3475     return -1;
3476   return fetchstr(cp, addr, pp);
3477 }
3478
3479 extern int sys_chdir(void);
3480 extern int sys_close(void);
3481 extern int sys_dup(void);
3482 extern int sys_exec(void);
3483 extern int sys_exit(void);
3484 extern int sys_fork(void);
3485 extern int sys_fstat(void);
3486 extern int sys_getpid(void);
3487 extern int sys_kill(void);
3488 extern int sys_link(void);
3489 extern int sys_mkdir(void);
3490 extern int sys_mknod(void);
3491 extern int sys_open(void);
3492 extern int sys_pipe(void);
3493 extern int sys_read(void);
3494 extern int sys_sbrk(void);
3495 extern int sys_sleep(void);
3496 extern int sys_unlink(void);
3497 extern int sys_wait(void);
3498 extern int sys_write(void);
3499 extern int sys_upmsec(void);
```

```
3500 // BSD sockets
3501 extern int sys_accept(void);
3502 extern int sys_bind(void);
3503 extern int sys_shutdown(void);
3504 extern int sys_getsockopt(void);
3505 extern int sys_setsockopt(void);
3506 extern int sys_sockclose(void);
3507 extern int sys_connect(void);
3508 extern int sys_listen(void);
3509 extern int sys_recv(void);
3510 extern int sys_recvfrom(void);
3511 extern int sys_send(void);
3512 extern int sys_sendto(void);
3513 extern int sys_socket(void);
3514 extern int sys_getpeername (void);
3515 extern int sys_getsockname (void);
3516 //int lwip_read(int s, void *mem, int len);
3517 //int lwip_write(int s, void *dataptr, int size);
3518 //int lwip_select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *ex
3519 //                struct timeval *timeout);
3520 //extern int sys_sockioctl(void);
3521
3522 static int (*syscalls[])(void) = {
3523 [SYS_chdir]   sys_chdir,
3524 [SYS_close]   sys_close,
3525 [SYS_dup]     sys_dup,
3526 [SYS_exec]    sys_exec,
3527 [SYS_exit]    sys_exit,
3528 [SYS_fork]    sys_fork,
3529 [SYS_fstat]   sys_fstat,
3530 [SYS_getpid]  sys_getpid,
3531 [SYS_kill]    sys_kill,
3532 [SYS_link]    sys_link,
3533 [SYS_mkdir]   sys_mkdir,
3534 [SYS_mknod]   sys_mknod,
3535 [SYS_open]    sys_open,
3536 [SYS_pipe]    sys_pipe,
3537 [SYS_read]    sys_read,
3538 [SYS_sbrk]    sys_sbrk,
3539 [SYS_sleep]   sys_sleep,
3540 [SYS_unlink]  sys_unlink,
3541 [SYS_wait]    sys_wait,
3542 [SYS_write]   sys_write,
3543 [SYS_upmsec]  sys_upmsec,
3544 // BSD socket
3545 [SYS_socket]  sys_socket,
3546 [SYS_bind]    sys_bind,
3547 [SYS_listen]  sys_listen,
3548 [SYS_accept]  sys_accept,
3549 [SYS_recv]    sys_recv,
```



```
3550 [SYS_recvfrom] sys_recvfrom,
3551 [SYS_send]    sys_send,
3552 [SYS_sendto]  sys_sendto,
3553 [SYS_shutdown] sys_shutdown,
3554 [SYS_getsockopt] sys_getsockopt,
3555 [SYS_setsockopt] sys_setsockopt,
3556 [SYS_sockclose] sys_sockclose,
3557 [SYS_connect] sys_connect,
3558 [SYS_getpeername] sys_getpeername,
3559 [SYS_getsockname] sys_getsockname,
3560 };
3561
3562 void
3563 syscall(void)
3564 {
3565   int num;
3566
3567   num = cp->tf->eax;
3568   if(num >= 0 && num < NELEM(syscalls) && syscalls[num])
3569     cp->tf->eax = syscalls[num]();
3570   else {
3571     cprintf("%d %s: unknown sys call %d\n",
3572             cp->pid, cp->name, num);
3573     cp->tf->eax = -1;
3574   }
3575 }
3576
3577
3578
3579
3580
3581
3582
3583
3584
3585
3586
3587
3588
3589
3590
3591
3592
3593
3594
3595
3596
3597
3598
3599
```

```
3600 #include "types.h"
3601 #include "defs.h"
3602 #include "param.h"
3603 #include "mmu.h"
3604 #include "proc.h"
3605 #include "lwip/include/lwip/sockets.h"
3606
3607 int
3608 sys_fork(void)
3609 {
3610   struct proc *np;
3611
3612   if((np = copyproc(cp)) == 0)
3613     return -1;
3614   np->state = RUNNABLE;
3615   return np->pid;
3616 }
3617
3618 int
3619 sys_exit(void)
3620 {
3621   exit();
3622   return 0;  // not reached
3623 }
3624
3625 int
3626 sys_wait(void)
3627 {
3628   return wait();
3629 }
3630
3631 int
3632 sys_kill(void)
3633 {
3634   int pid;
3635
3636   if(argint(0, &pid) < 0)
3637     return -1;
3638   return kill(pid);
3639 }
3640
3641 int
3642 sys_getpid(void)
3643 {
3644   return cp->pid;
3645 }
3646
3647
3648
3649
```



```
3650 int
3651 sys_sbrk(void)
3652 {
3653   int addr;
3654   int n;
3655
3656   if(argint(0, &n) < 0)
3657     return -1;
3658   if((addr = growproc(n)) < 0)
3659     return -1;
3660   setupsegs(cp);
3661   return addr;
3662 }
3663
3664 int
3665 sys_sleep(void)
3666 {
3667   int n, ticks0;
3668
3669   if(argint(0, &n) < 0)
3670     return -1;
3671   acquire(&tickslock);
3672   ticks0 = ticks;
3673   while(ticks - ticks0 < n){
3674     if(cp->killed){
3675       release(&tickslock);
3676       return -1;
3677     }
3678     sleep(&ticks, &tickslock);
3679   }
3680   release(&tickslock);
3681   return 0;
3682 }
3683
3684 int
3685 sys_upmsec(void)
3686 {
3687     return millitime();
3688 }
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699
```

```
3700 int
3701 sys_accept(void)
3702 {
3703     int s;
3704     struct sockaddr *addr;
3705     socklen_t *addrlen;
3706     if((argint(0, &s)<0) ||
3707             (argptr(1, &addr, sizeof(struct sockaddr))<0) ||
3708             (argptr(2, &addrlen, sizeof(socklen_t))<0))
3709         return -1;
3710     return lwip_accept(s, addr, addrlen);
3711 }
3712
3713 int
3714 sys_bind(void)
3715 {
3716     int s;
3717     struct sockaddr *name;
3718     socklen_t *namelen;
3719     if((argint(0, &s)<0) ||
3720             (argptr(1, &name, sizeof(struct sockaddr))<0) ||
3721             (argptr(2, &namelen, sizeof(socklen_t))<0))
3722         return -1;
3723     return lwip_bind(s, name, namelen);
3724 }
3725
3726 int
3727 sys_shutdown(void)
3728 {
3729     int s;
3730     int how;
3731     if ((argint(0, &s)<0) || (argint(1, &how) < 0))
3732         return -1;
3733     return lwip_shutdown(s, how);
3734 }
3735
3736
3737
3738
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3749
```



```
3750 int
3751 sys_getsockopt(void)
3752 {
3753     int s;
3754     int level;
3755     int optname;
3756     void *optval;
3757     socklen_t *optlen;
3758     if ((argint(0, &s)<0) ||
3759         (argint(1, &level)<0) ||
3760         (argint(2, &optname)<0) ||
3761         (argptr(4, &optlen, sizeof(socklen_t))<0) ||
3762         (argptr(3, &optval, 0)<0))
3763         return -1;
3764     return lwip_getsockopt(s, level, optname, optval, optlen);
3765 }
3766
3767 int sys_setsockopt(void)
3768 {
3769     int s;
3770     int level;
3771     int optname;
3772     void *optval;
3773     socklen_t *optlen;
3774     if ((argint(0, &s)<0) ||
3775         (argint(1, &level)<0) ||
3776         (argint(2, &optname)<0) ||
3777         (argptr(4, &optlen, sizeof(socklen_t))<0) ||
3778         (argptr(3, &optval, *optlen)<0))
3779         return -1;
3780     return lwip_setsockopt(s, level, optname, optval, optlen);
3781 }
3782
3783 int sys_sockclose(void)
3784 {
3785     int s;
3786     if (argint(0, &s) <0)
3787         return -1;
3788     return lwip_close(s);
3789 }
3790
3791
3792
3793
3794
3795
3796
3797
3798
3799
```

```
3800 int sys_connect(void)
3801 {
3802     int s;
3803     struct sockaddr *name;
3804     socklen_t *namelen;
3805     if((argint(0, &s)<0) ||
3806             (argptr(1, &name, sizeof(struct sockaddr))<0) ||
3807             (argptr(2, &namelen, sizeof(socklen_t))<0))
3808         return -1;
3809     return lwip_connect(s, name, namelen);
3810 }
3811
3812 int sys_listen(void)
3813 {
3814     int s;
3815     int backlog;
3816     if ((argint(0, &s)<0) ||
3817         (argint(1, &backlog)<0))
3818         return -1;
3819     return lwip_listen(s, backlog);
3820 }
3821
3822 int sys_recv(void)
3823 {
3824     int s;
3825     void *mem;
3826     int len;
3827     unsigned int flags;
3828     if ((argint(0, &s)<0) ||
3829         (argint(2, &len)<0) ||
3830         (argptr(1, &mem, len)<0) ||
3831         (argint(3, &flags)<0))
3832         return -1;
3833     return lwip_recv(s, mem, len, flags);
3834 }
3835
3836
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
```



```
3850 int sys_recvfrom(void)
3851 {
3852     int s;
3853     void *mem;
3854     int len;
3855     unsigned int flags;
3856     struct sockaddr *from;
3857     socklen_t *fromlen;
3858     if ((argint(0, &s)<0) ||
3859         (argint(2, &len)<0) ||
3860         (argptr(1, &mem, len)<0) ||
3861         (argint(3, &flags)<0) ||
3862         (argptr(4, &from, 0)<0) ||
3863         (argptr(5, &fromlen, sizeof(socklen_t))<0))
3864         return -1;
3865     return lwip_recvfrom(s, mem, len, flags, from, fromlen);
3866 }
3867
3868 int sys_send(void)
3869 {
3870     int s;
3871     void *dataptr;
3872     int size;
3873     unsigned int flags;
3874     if ((argint(0, &s)<0) ||
3875         (argint(2, &size)<0) ||
3876         (argptr(1, &dataptr, size)<0) ||
3877         (argint(3, &flags)<0))
3878         return -1;
3879     return lwip_send(s, dataptr, size, flags);
3880 }
3881
3882 int sys_sendto(void)
3883 {
3884     int s;
3885     void *dataptr;
3886     int size;
3887     unsigned int flags;
3888     struct sockaddr *to;
3889     socklen_t *tolen;
3890     if ((argint(0, &s)<0) ||
3891         (argint(2, &size)<0) ||
3892         (argptr(1, &dataptr, size)<0) ||
3893         (argint(3, &flags)<0) ||
3894         (argptr(5, &tolen, sizeof(socklen_t))<0) ||
3895         (argptr(4, &to, *tolen)<0))
3896         return -1;
3897     return lwip_send(s, dataptr, size, flags);
3898 }
3899
```

```
3900 int sys_socket(void)
3901 {
3902     int domain;
3903     int type;
3904     int protocol;
3905     if ((argint(0, &domain)<0) ||
3906         (argint(1, &type)<0) ||
3907         (argint(2, &protocol)<0))
3908         return -1;
3909     return lwip_socket(domain, type, protocol);
3910 }
3911
3912 int
3913 sys_getpeername(void)
3914 {
3915     int s;
3916     struct sockaddr *name;
3917     socklen_t *namelen;
3918     if((argint(0, &s)<0) ||
3919             (argptr(1, &name, sizeof(struct sockaddr))<0) ||
3920             (argptr(2, &namelen, sizeof(socklen_t))<0))
3921         return -1;
3922     return lwip_getpeername(s, name, namelen);
3923 }
3924
3925 int
3926 sys_getsockname(void)
3927 {
3928     int s;
3929     struct sockaddr *name;
3930     socklen_t *namelen;
3931     if((argint(0, &s)<0) ||
3932             (argptr(1, &name, sizeof(struct sockaddr))<0) ||
3933             (argptr(2, &namelen, sizeof(socklen_t))<0))
3934         return -1;
3935     return lwip_getsockname(s, name, namelen);
3936 }
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
```



```
3950 struct buf {
3951   int flags;
3952   uint dev;
3953   uint sector;
3954   struct buf *prev; // LRU cache list
3955   struct buf *next;
3956   struct buf *qnext; // disk queue
3957   uchar data[512];
3958 };
3959 #define B_BUSY  0x1  // buffer is locked by some process
3960 #define B_VALID 0x2  // buffer has been read from disk
3961 #define B_DIRTY 0x4  // buffer needs to be written to disk
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
```

```
4000 struct devsw {
4001   int (*read)(struct inode*, char*, int);
4002   int (*write)(struct inode*, char*, int);
4003 };
4004
4005 extern struct devsw devsw[];
4006
4007 #define CONSOLE 1
4008
4009
4010
4011
4012
4013
4014
4015
4016
4017
4018
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
```



```
4050 #define O_RDONLY  0x000
4051 #define O_WRONLY  0x001
4052 #define O_RDWR    0x002
4053 #define O_CREATE  0x200
4054
4055
4056
4057
4058
4059
4060
4061
4062
4063
4064
4065
4066
4067
4068
4069
4070
4071
4072
4073
4074
4075
4076
4077
4078
4079
4080
4081
4082
4083
4084
4085
4086
4087
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
```

```
4100 struct stat {
4101   int dev;     // Device number
4102   uint ino;    // Inode number on device
4103   short type;  // Type of file
4104   short nlink; // Number of links to file
4105   uint size;   // Size of file in bytes
4106 };
4107
4108
4109
4110
4111
4112
4113
4114
4115
4116
4117
4118
4119
4120
4121
4122
4123
4124
4125
4126
4127
4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```



```
4150 struct file {
4151   enum { FD_CLOSED, FD_NONE, FD_PIPE, FD_INODE } type;
4152   int ref; // reference count
4153   char readable;
4154   char writable;
4155   struct pipe *pipe;
4156   struct inode *ip;
4157   uint off;
4158 };
4159
4160
4161
4162
4163
4164
4165
4166
4167
4168
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
```

```
4200 // On-disk file system format.
4201 // Both the kernel and user programs use this header file.
4202
4203 // Block 0 is unused.
4204 // Block 1 is super block.
4205 // Inodes start at block 2.
4206
4207 #define BSIZE 512  // block size
4208
4209 // File system super block
4210 struct superblock {
4211   uint size;         // Size of file system image (blocks)
4212   uint nblocks;      // Number of data blocks
4213   uint ninodes;      // Number of inodes.
4214 };
4215
4216 #define NADDRS (NDIRECT+1)
4217 #define NDIRECT 12
4218 #define INDIRECT 12
4219 #define NINDIRECT (BSIZE / sizeof(uint))
4220 #define MAXFILE (NDIRECT  + NINDIRECT)
4221
4222 // On-disk inode structure
4223 struct dinode {
4224   short type;           // File type
4225   short major;          // Major device number (T_DEV only)
4226   short minor;          // Minor device number (T_DEV only)
4227   short nlink;          // Number of links to inode in file system
4228   uint size;            // Size of file (bytes)
4229   uint addrs[NADDRS];   // Data block addresses
4230 };
4231
4232 #define T_DIR  1   // Directory
4233 #define T_FILE 2   // File
4234 #define T_DEV  3   // Special device
4235
4236 // Inodes per block.
4237 #define IPB           (BSIZE / sizeof(struct dinode))
4238
4239 // Block containing inode i
4240 #define IBLOCK(i)     ((i) / IPB + 2)
4241
4242 // Bitmap bits per block
4243 #define BPB           (BSIZE*8)
4244
4245 // Block containing bit for block b
4246 #define BBLOCK(b, ninodes) (b/BPB + (ninodes)/IPB + 3)
4247
4248 // Directory is a file containing a sequence of dirent structures.
4249 #define DIRSIZ 14
```



```
4250 struct dirent {
4251   ushort inum;
4252   char name[DIRSIZ];
4253 };
4254
4255
4256
4257
4258
4259
4260
4261
4262
4263
4264
4265
4266
4267
4268
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
```

```
4300 // in-core file system types
4301
4302 struct inode {
4303   uint dev;           // Device number
4304   uint inum;          // Inode number
4305   int ref;            // Reference count
4306   int flags;          // I_BUSY, I_VALID
4307
4308   short type;         // copy of disk inode
4309   short major;
4310   short minor;
4311   short nlink;
4312   uint size;
4313   uint addrs[NADDRS];
4314 };
4315
4316 #define I_BUSY 0x1
4317 #define I_VALID 0x2
4318
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4330
4331
4332
4333
4334
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
```



```
4350 // Simple PIO-based (non-DMA) IDE driver code.
4351
4352 #include "types.h"
4353 #include "defs.h"
4354 #include "param.h"
4355 #include "mmu.h"
4356 #include "proc.h"
4357 #include "x86.h"
4358 #include "traps.h"
4359 #include "spinlock.h"
4360 #include "buf.h"
4361 #include "picirq.h"
4362
4363 #define IDE_BSY       0x80
4364 #define IDE_DRDY      0x40
4365 #define IDE_DF        0x20
4366 #define IDE_ERR       0x01
4367
4368 #define IDE_CMD_READ  0x20
4369 #define IDE_CMD_WRITE 0x30
4370
4371 // ide_queue points to the buf now being read/written to the disk.
4372 // ide_queue->qnext points to the next buf to be processed.
4373 // You must hold ide_lock while manipulating queue.
4374
4375 static struct spinlock ide_lock;
4376 static struct buf *ide_queue;
4377
4378 static int disk_1_present;
4379 static void ide_start_request();
4380
4381 // Wait for IDE disk to become ready.
4382 static int
4383 ide_wait_ready(int check_error)
4384 {
4385   int r;
4386
4387   while(((r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
4388     ;
4389   if(check_error && (r & (IDE_DF|IDE_ERR)) != 0)
4390     return -1;
4391   return 0;
4392 }
4393
4394 void
4395 dummy_ide_intr(struct trapframe *tf)
4396 {
4397 }
4398
4399
```

```
4400 void
4401 ide_init(void)
4402 {
4403   int i;
4404
4405   initlock(&ide_lock, "ide");
4406   pic_enable(IRQ_IDE);
4407   ioapic_enable(IRQ_IDE, ncpu - 1);
4408   ide_wait_ready(0);
4409   reg_irq_handler(IRQ_IDE_2, dummy_ide_intr);
4410
4411   // Check if disk 1 is present
4412   outb(0x1f6, 0xe0 | (1<<4));
4413   for(i=0; i<1000; i++){
4414     if(inb(0x1f7) != 0){
4415       disk_1_present = 1;
4416       break;
4417     }
4418   }
4419
4420   // Switch back to disk 0.
4421   outb(0x1f6, 0xe0 | (0<<4));
4422   outb(0x3f6, 0);  // generate interrupt
4423 }
4424
4425 // Start the request for b.  Caller must hold ide_lock.
4426 static void
4427 ide_start_request(struct buf *b)
4428 {
4429 //    cprintf("XXX ide_start_request\n");
4430   if(b == 0)
4431     panic("ide_start_request");
4432
4433   ide_wait_ready(0);
4434   outb(0x1f2, 1);  // number of sectors
4435   outb(0x1f3, b->sector & 0xff);
4436   outb(0x1f4, (b->sector >> 8) & 0xff);
4437   outb(0x1f5, (b->sector >> 16) & 0xff);
4438   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
4439   if(b->flags & B_DIRTY){
4440     outb(0x1f7, IDE_CMD_WRITE);
4441     outsl(0x1f0, b->data, 512/4);
4442   } else {
4443     outb(0x1f7, IDE_CMD_READ);
4444   }
4445 }
4446
4447
4448
4449
```



```
4450 // Interrupt handler.
4451 void
4452 ide_intr(void)
4453 {
4454   struct buf *b;
4455
4456   acquire(&ide_lock);
4457   if((b = ide_queue) == 0){
4458     release(&ide_lock);
4459     return;
4460   }
4461
4462   // Read data if needed.
4463   if(!(b->flags & B_DIRTY) && ide_wait_ready(1) >= 0)
4464     insl(0x1f0, b->data, 512/4);
4465
4466   // Wake process waiting for this buf.
4467   b->flags |= B_VALID;
4468   b->flags &= ~B_DIRTY;
4469   wakeup(b);
4470
4471   // Start disk on next buf in queue.
4472   if((ide_queue = b->qnext) != 0)
4473     ide_start_request(ide_queue);
4474
4475   release(&ide_lock);
4476 }
4477
4478 // Sync buf with disk.
4479 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
4480 // Else if B_VALID is not set, read buf from disk, set B_VALID.
4481 void
4482 ide_rw(struct buf *b)
4483 {
4484   struct buf **pp;
4485
4486   if(!(b->flags & B_BUSY))
4487     panic("ide_rw: buf not busy");
4488   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
4489     panic("ide_rw: nothing to do");
4490   if(b->dev != 0 && !disk_1_present)
4491     panic("ide disk 1 not present");
4492
4493   acquire(&ide_lock);
4494
4495   // Append b to ide_queue.
4496   b->qnext = 0;
4497   for(pp=&ide_queue; *pp; pp=&(*pp)->qnext)
4498     ;
4499   *pp = b;
```

```
4500   // Start disk if necessary.
4501   if(ide_queue == b)
4502     ide_start_request(b);
4503
4504   // Wait for request to finish.
4505   // Assuming will not sleep too long: ignore cp->killed.
4506   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
4507   {
4508     sleep(b, &ide_lock);
4509   }
4510 //  cprintf("XXX ide_rw: finished\n");
4511
4512   release(&ide_lock);
4513 }
4514
4515
4516
4517
4518
4519
4520
4521
4522
4523
4524
4525
4526
4527
4528
4529
4530
4531
4532
4533
4534
4535
4536
4537
4538
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4549
```



```
4550 // Buffer cache.
4551 //
4552 // The buffer cache is a linked list of buf structures holding
4553 // cached copies of disk block contents.  Caching disk blocks
4554 // in memory reduces the number of disk reads and also provides
4555 // a synchronization point for disk blocks used by multiple processes.
4556 //
4557 // Interface:
4558 // * To get a buffer for a particular disk block, call bread.
4559 // * After changing buffer data, call bwrite to flush it to disk.
4560 // * When done with the buffer, call brelse.
4561 // * Do not use the buffer after calling brelse.
4562 // * Only one process at a time can use a buffer,
4563 //     so do not keep them longer than necessary.
4564 //
4565 // The implementation uses three state flags internally:
4566 // * B_BUSY: the block has been returned from bread
4567 //     and has not been passed back to brelse.
4568 // * B_VALID: the buffer data has been initialized
4569 //     with the associated disk block contents.
4570 // * B_DIRTY: the buffer data has been modified
4571 //     and needs to be written to disk.
4572
4573 #include "types.h"
4574 #include "defs.h"
4575 #include "param.h"
4576 #include "spinlock.h"
4577 #include "buf.h"
4578
4579 struct buf buf[NBUF];
4580 struct spinlock buf_table_lock;
4581
4582 // Linked list of all buffers, through prev/next.
4583 // bufhead->next is most recently used.
4584 // bufhead->tail is least recently used.
4585 struct buf bufhead;
4586
4587 void
4588 binit(void)
4589 {
4590   struct buf *b;
4591
4592   initlock(&buf_table_lock, "buf_table");
4593
4594   // Create linked list of buffers
4595   bufhead.prev = &bufhead;
4596   bufhead.next = &bufhead;
4597   for(b = buf; b < buf+NBUF; b++){
4598     b->next = bufhead.next;
4599     b->prev = &bufhead;
```

```
4600     bufhead.next->prev = b;
4601     bufhead.next = b;
4602   }
4603 }
4604
4605 // Look through buffer cache for sector on device dev.
4606 // If not found, allocate fresh block.
4607 // In either case, return locked buffer.
4608 static struct buf*
4609 bget(uint dev, uint sector)
4610 {
4611   struct buf *b;
4612
4613   acquire(&buf_table_lock);
4614
4615  loop:
4616   // Try for cached block.
4617   for(b = bufhead.next; b != &bufhead; b = b->next){
4618     if((b->flags & (B_BUSY|B_VALID)) &&
4619        b->dev == dev && b->sector == sector){
4620       if(b->flags & B_BUSY){
4621         sleep(buf, &buf_table_lock);
4622         goto loop;
4623       }
4624       b->flags |= B_BUSY;
4625       release(&buf_table_lock);
4626       return b;
4627     }
4628   }
4629
4630   // Allocate fresh block.
4631   for(b = bufhead.prev; b != &bufhead; b = b->prev){
4632     if((b->flags & B_BUSY) == 0){
4633       b->flags = B_BUSY;
4634       b->dev = dev;
4635       b->sector = sector;
4636       release(&buf_table_lock);
4637       return b;
4638     }
4639   }
4640   panic("bget: no buffers");
4641 }
4642
4643
4644
4645
4646
4647
4648
4649
```



```
4650 // Return a B_BUSY buf with the contents of the indicated disk sector.
4651 struct buf*
4652 bread(uint dev, uint sector)
4653 {
4654   struct buf *b;
4655
4656   b = bget(dev, sector);
4657   if(!(b->flags & B_VALID))
4658     ide_rw(b);
4659   return b;
4660 }
4661
4662 // Write buf's contents to disk.  Must be locked.
4663 void
4664 bwrite(struct buf *b)
4665 {
4666   if((b->flags & B_BUSY) == 0)
4667     panic("bwrite");
4668   b->flags |= B_DIRTY;
4669   ide_rw(b);
4670 }
4671
4672 // Release the buffer buf.
4673 void
4674 brelse(struct buf *b)
4675 {
4676   if((b->flags & B_BUSY) == 0)
4677     panic("brelse");
4678
4679   acquire(&buf_table_lock);
4680
4681   b->next->prev = b->prev;
4682   b->prev->next = b->next;
4683   b->next = bufhead.next;
4684   b->prev = &bufhead;
4685   bufhead.next->prev = b;
4686   bufhead.next = b;
4687
4688   b->flags &= ~B_BUSY;
4689   wakeup(buf);
4690
4691   release(&buf_table_lock);
4692 }
4693
4694
4695
4696
4697
4698
4699
```

```
4700 // File system implementation.  Four layers:
4701 //   + Blocks: allocator for raw disk blocks.
4702 //   + Files: inode allocator, reading, writing, metadata.
4703 //   + Directories: inode with special contents (list of other inodes!)
4704 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
4705 //
4706 // Disk layout is: superblock, inodes, block in-use bitmap, data blocks.
4707 //
4708 // This file contains the low-level file system manipulation
4709 // routines.  The (higher-level) system call implementations
4710 // are in sysfile.c.
4711
4712 #include "types.h"
4713 #include "defs.h"
4714 #include "param.h"
4715 #include "stat.h"
4716 #include "mmu.h"
4717 #include "proc.h"
4718 #include "spinlock.h"
4719 #include "buf.h"
4720 #include "fs.h"
4721 #include "fsvar.h"
4722 #include "dev.h"
4723
4724 #define min(a, b) ((a) < (b) ? (a) : (b))
4725 static void itrunc(struct inode*);
4726
4727 // Read the super block.
4728 static void
4729 readsb(int dev, struct superblock *sb)
4730 {
4731   struct buf *bp;
4732
4733 //  cprintf("XXX %s\n",__func__);
4734   bp = bread(dev, 1);
4735   memmove(sb, bp->data, sizeof(*sb));
4736   brelse(bp);
4737 }
4738
4739
4740
4741
4742
4743
4744
4745
4746
4747
4748
4749
```



```
4750 // Zero a block.
4751 static void
4752 bzero(int dev, int bno)
4753 {
4754   struct buf *bp;
4755
4756 //  cprintf("XXX %s\n",__func__);
4757   bp = bread(dev, bno);
4758   memset(bp->data, 0, BSIZE);
4759   bwrite(bp);
4760   brelse(bp);
4761 }
4762
4763 // Blocks.
4764
4765 // Allocate a disk block.
4766 static uint
4767 balloc(uint dev)
4768 {
4769   int b, bi, m;
4770   struct buf *bp;
4771   struct superblock sb;
4772
4773 //  cprintf("XXX %s\n",__func__);
4774   bp = 0;
4775   readsb(dev, &sb);
4776   for(b = 0; b < sb.size; b += BPB){
4777     bp = bread(dev, BBLOCK(b, sb.ninodes));
4778     for(bi = 0; bi < BPB; bi++){
4779       m = 1 << (bi % 8);
4780       if((bp->data[bi/8] & m) == 0){  // Is block free?
4781         bp->data[bi/8] |= m;  // Mark block in use on disk.
4782         bwrite(bp);
4783         brelse(bp);
4784         return b + bi;
4785       }
4786     }
4787     brelse(bp);
4788   }
4789   panic("balloc: out of blocks");
4790 }
4791
4792
4793
4794
4795
4796
4797
4798
4799
```

```
4800 // Free a disk block.
4801 static void
4802 bfree(int dev, uint b)
4803 {
4804   struct buf *bp;
4805   struct superblock sb;
4806   int bi, m;
4807
4808   bzero(dev, b);
4809
4810 //  cprintf("XXX %s\n",__func__);
4811   readsb(dev, &sb);
4812   bp = bread(dev, BBLOCK(b, sb.ninodes));
4813   bi = b % BPB;
4814   m = 1 << (bi % 8);
4815   if((bp->data[bi/8] & m) == 0)
4816     panic("freeing free block");
4817   bp->data[bi/8] &= ~m;  // Mark block free on disk.
4818   bwrite(bp);
4819   brelse(bp);
4820 }
4821
4822 // Inodes.
4823 //
4824 // An inode is a single, unnamed file in the file system.
4825 // The inode disk structure holds metadata (the type, device numbers,
4826 // and data size) along with a list of blocks where the associated
4827 // data can be found.
4828 //
4829 // The inodes are laid out sequentially on disk immediately after
4830 // the superblock.  The kernel keeps a cache of the in-use
4831 // on-disk structures to provide a place for synchronizing access
4832 // to inodes shared between multiple processes.
4833 //
4834 // ip->ref counts the number of pointer references to this cached
4835 // inode; references are typically kept in struct file and in cp->cwd.
4836 // When ip->ref falls to zero, the inode is no longer cached.
4837 // It is an error to use an inode without holding a reference to it.
4838 //
4839 // Processes are only allowed to read and write inode
4840 // metadata and contents when holding the inode's lock,
4841 // represented by the I_BUSY flag in the in-memory copy.
4842 // Because inode locks are held during disk accesses,
4843 // they are implemented using a flag rather than with
4844 // spin locks.  Callers are responsible for locking
4845 // inodes before passing them to routines in this file; leaving
4846 // this responsibility with the caller makes it possible for them
4847 // to create arbitrarily-sized atomic operations.
4848 //
4849 // To give maximum control over locking to the callers,
```



```
4850 // the routines in this file that return inode pointers
4851 // return pointers to *unlocked* inodes.  It is the callers'
4852 // responsibility to lock them before using them.  A non-zero
4853 // ip->ref keeps these unlocked inodes in the cache.
4854
4855 struct {
4856   struct spinlock lock;
4857   struct inode inode[NINODE];
4858 } icache;
4859
4860 void
4861 iinit(void)
4862 {
4863   initlock(&icache.lock, "icache.lock");
4864 }
4865
4866 // Find the inode with number inum on device dev
4867 // and return the in-memory copy.
4868 static struct inode*
4869 iget(uint dev, uint inum)
4870 {
4871   struct inode *ip, *empty;
4872
4873   acquire(&icache.lock);
4874
4875   // Try for cached inode.
4876   empty = 0;
4877   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
4878     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
4879       ip->ref++;
4880       release(&icache.lock);
4881 //      cprintf("XXX iget: found in cache\n");
4882       return ip;
4883     }
4884     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
4885       empty = ip;
4886   }
4887
4888   // Allocate fresh inode.
4889   if(empty == 0)
4890     panic("iget: no inodes");
4891
4892 //  cprintf("XXX iget: allocating new\n");
4893   ip = empty;
4894   ip->dev = dev;
4895   ip->inum = inum;
4896   ip->ref = 1;
4897   ip->flags = 0;
4898   release(&icache.lock);
4899
```

```
4900   return ip;
4901 }
4902
4903 // Increment reference count for ip.
4904 // Returns ip to enable ip = idup(ip1) idiom.
4905 struct inode*
4906 idup(struct inode *ip)
4907 {
4908   acquire(&icache.lock);
4909   ip->ref++;
4910   release(&icache.lock);
4911   return ip;
4912 }
4913
4914 // Lock the given inode.
4915 void
4916 ilock(struct inode *ip)
4917 {
4918   struct buf *bp;
4919   struct dinode *dip;
4920
4921   if(ip == 0 || ip->ref < 1)
4922     panic("ilock");
4923
4924   acquire(&icache.lock);
4925   while(ip->flags & I_BUSY)
4926     sleep(ip, &icache.lock);
4927   ip->flags |= I_BUSY;
4928   release(&icache.lock);
4929
4930   if(!(ip->flags & I_VALID)){
4931     bp = bread(ip->dev, IBLOCK(ip->inum));
4932 //  cprintf("XXX %s:after bread \n",__func__);
4933     dip = (struct dinode*)bp->data + ip->inum%IPB;
4934     ip->type = dip->type;
4935     ip->major = dip->major;
4936     ip->minor = dip->minor;
4937     ip->nlink = dip->nlink;
4938     ip->size = dip->size;
4939     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
4940     brelse(bp);
4941     ip->flags |= I_VALID;
4942     if(ip->type == 0)
4943       panic("ilock: no type");
4944   }
4945 }
4946
4947
4948
4949
```



```
4950 // Unlock the given inode.
4951 void
4952 iunlock(struct inode *ip)
4953 {
4954   if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
4955     panic("iunlock");
4956
4957   acquire(&icache.lock);
4958   ip->flags &= ~I_BUSY;
4959   wakeup(ip);
4960   release(&icache.lock);
4961 }
4962
4963 // Caller holds reference to unlocked ip.  Drop reference.
4964 void
4965 iput(struct inode *ip)
4966 {
4967   acquire(&icache.lock);
4968   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
4969     // inode is no longer used: truncate and free inode.
4970     if(ip->flags & I_BUSY)
4971       panic("iput busy");
4972     ip->flags |= I_BUSY;
4973     release(&icache.lock);
4974     itrunc(ip);
4975     ip->type = 0;
4976     iupdate(ip);
4977     acquire(&icache.lock);
4978     ip->flags &= ~I_BUSY;
4979     wakeup(ip);
4980   }
4981   ip->ref--;
4982   release(&icache.lock);
4983 }
4984
4985 // Common idiom: unlock, then put.
4986 void
4987 iunlockput(struct inode *ip)
4988 {
4989   iunlock(ip);
4990   iput(ip);
4991 }
4992
4993
4994
4995
4996
4997
4998
4999
```

```
5000 // Allocate a new inode with the given type on device dev.
5001 struct inode*
5002 ialloc(uint dev, short type)
5003 {
5004   int inum;
5005   struct buf *bp;
5006   struct dinode *dip;
5007   struct superblock sb;
5008
5009 //  cprintf("XXX %s\n",__func__);
5010   readsb(dev, &sb);
5011   for(inum = 1; inum < sb.ninodes; inum++){  // loop over inode blocks
5012     bp = bread(dev, IBLOCK(inum));
5013     dip = (struct dinode*)bp->data + inum%IPB;
5014     if(dip->type == 0){  // a free inode
5015       memset(dip, 0, sizeof(*dip));
5016       dip->type = type;
5017       bwrite(bp);   // mark it allocated on the disk
5018       brelse(bp);
5019       return iget(dev, inum);
5020     }
5021     brelse(bp);
5022   }
5023   panic("ialloc: no inodes");
5024 }
5025
5026 // Copy inode, which has changed, from memory to disk.
5027 void
5028 iupdate(struct inode *ip)
5029 {
5030   struct buf *bp;
5031   struct dinode *dip;
5032
5033   bp = bread(ip->dev, IBLOCK(ip->inum));
5034   dip = (struct dinode*)bp->data + ip->inum%IPB;
5035   dip->type = ip->type;
5036   dip->major = ip->major;
5037   dip->minor = ip->minor;
5038   dip->nlink = ip->nlink;
5039   dip->size = ip->size;
5040   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5041   bwrite(bp);
5042   brelse(bp);
5043 }
5044
5045
5046
5047
5048
5049
```



```
5050 // Inode contents
5051 //
5052 // The contents (data) associated with each inode is stored
5053 // in a sequence of blocks on the disk.  The first NDIRECT blocks
5054 // are listed in ip->addrs[].  The next NINDIRECT blocks are
5055 // listed in the block ip->addrs[INDIRECT].
5056
5057 // Return the disk block address of the nth block in inode ip.
5058 // If there is no such block, alloc controls whether one is allocated.
5059 static uint
5060 bmap(struct inode *ip, uint bn, int alloc)
5061 {
5062   uint addr, *a;
5063   struct buf *bp;
5064
5065   if(bn < NDIRECT){
5066     if((addr = ip->addrs[bn]) == 0){
5067       if(!alloc)
5068         return -1;
5069       ip->addrs[bn] = addr = balloc(ip->dev);
5070     }
5071     return addr;
5072   }
5073   bn -= NDIRECT;
5074
5075   if(bn < NINDIRECT){
5076     // Load indirect block, allocating if necessary.
5077     if((addr = ip->addrs[INDIRECT]) == 0){
5078       if(!alloc)
5079         return -1;
5080       ip->addrs[INDIRECT] = addr = balloc(ip->dev);
5081     }
5082     bp = bread(ip->dev, addr);
5083     a = (uint*)bp->data;
5084
5085     if((addr = a[bn]) == 0){
5086       if(!alloc){
5087         brelse(bp);
5088         return -1;
5089       }
5090       a[bn] = addr = balloc(ip->dev);
5091       bwrite(bp);
5092     }
5093     brelse(bp);
5094     return addr;
5095   }
5096
5097   panic("bmap: out of range");
5098 }
5099
```

```
5100 // Truncate inode (discard contents).
5101 static void
5102 itrunc(struct inode *ip)
5103 {
5104   int i, j;
5105   struct buf *bp;
5106   uint *a;
5107
5108   for(i = 0; i < NDIRECT; i++){
5109     if(ip->addrs[i]){
5110       bfree(ip->dev, ip->addrs[i]);
5111       ip->addrs[i] = 0;
5112     }
5113   }
5114
5115   if(ip->addrs[INDIRECT]){
5116     bp = bread(ip->dev, ip->addrs[INDIRECT]);
5117     a = (uint*)bp->data;
5118     for(j = 0; j < NINDIRECT; j++){
5119       if(a[j])
5120         bfree(ip->dev, a[j]);
5121     }
5122     brelse(bp);
5123     ip->addrs[INDIRECT] = 0;
5124   }
5125
5126   ip->size = 0;
5127   iupdate(ip);
5128 }
5129
5130 // Copy stat information from inode.
5131 void
5132 stati(struct inode *ip, struct stat *st)
5133 {
5134   st->dev = ip->dev;
5135   st->ino = ip->inum;
5136   st->type = ip->type;
5137   st->nlink = ip->nlink;
5138   st->size = ip->size;
5139 }
5140
5141
5142
5143
5144
5145
5146
5147
5148
5149
```



```
5150 // Read data from inode.
5151 int
5152 readi(struct inode *ip, char *dst, uint off, uint n)
5153 {
5154   uint tot, m;
5155   struct buf *bp;
5156
5157 //  cprintf("XXX readi %d.%d\n", ip->dev, ip->inum);
5158   if(ip->type == T_DEV){
5159     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
5160       return -1;
5161     return devsw[ip->major].read(ip, dst, n);
5162   }
5163
5164   if(off > ip->size || off + n < off)
5165     return -1;
5166   if(off + n > ip->size)
5167     n = ip->size - off;
5168
5169   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5170     bp = bread(ip->dev, bmap(ip, off/BSIZE, 0));
5171     m = min(n - tot, BSIZE - off%BSIZE);
5172     memmove(dst, bp->data + off%BSIZE, m);
5173     brelse(bp);
5174   }
5175   return n;
5176 }
5177
5178 // Write data to inode.
5179 int
5180 writei(struct inode *ip, char *src, uint off, uint n)
5181 {
5182   uint tot, m;
5183   struct buf *bp;
5184
5185   if(ip->type == T_DEV){
5186     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
5187       return -1;
5188     return devsw[ip->major].write(ip, src, n);
5189   }
5190
5191   if(off + n < off)
5192     return -1;
5193   if(off + n > MAXFILE*BSIZE)
5194     n = MAXFILE*BSIZE - off;
5195
5196
5197
5198
5199
```

```
5200   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
5201     bp = bread(ip->dev, bmap(ip, off/BSIZE, 1));
5202     m = min(n - tot, BSIZE - off%BSIZE);
5203     memmove(bp->data + off%BSIZE, src, m);
5204     bwrite(bp);
5205     brelse(bp);
5206   }
5207
5208   if(n > 0 && off > ip->size){
5209     ip->size = off;
5210     iupdate(ip);
5211   }
5212   return n;
5213 }
5214
5215 // Directories
5216
5217 int
5218 namecmp(const char *s, const char *t)
5219 {
5220   return strncmp(s, t, DIRSIZ);
5221 }
5222
5223 // Look for a directory entry in a directory.
5224 // If found, set *poff to byte offset of entry.
5225 // Caller must have already locked dp.
5226 struct inode*
5227 dirlookup(struct inode *dp, char *name, uint *poff)
5228 {
5229   uint off, inum;
5230   struct buf *bp;
5231   struct dirent *de;
5232
5233   if(dp->type != T_DIR)
5234     panic("dirlookup not DIR");
5235
5236   for(off = 0; off < dp->size; off += BSIZE){
5237     bp = bread(dp->dev, bmap(dp, off / BSIZE, 0));
5238     for(de = (struct dirent*)bp->data;
5239         de < (struct dirent*)(bp->data + BSIZE);
5240         de++){
5241       if(de->inum == 0)
5242         continue;
5243       if(namecmp(name, de->name) == 0){
5244         // entry matches path element
5245         if(poff)
5246           *poff = off + (uchar*)de - bp->data;
5247         inum = de->inum;
5248         brelse(bp);
5249         return iget(dp->dev, inum);
```



```
5250       }
5251     }
5252     brelse(bp);
5253   }
5254   return 0;
5255 }
5256
5257 // Write a new directory entry (name, ino) into the directory dp.
5258 int
5259 dirlink(struct inode *dp, char *name, uint ino)
5260 {
5261   int off;
5262   struct dirent de;
5263   struct inode *ip;
5264
5265   // Check that name is not present.
5266   if((ip = dirlookup(dp, name, 0)) != 0){
5267     iput(ip);
5268     return -1;
5269   }
5270
5271   // Look for an empty dirent.
5272   for(off = 0; off < dp->size; off += sizeof(de)){
5273     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5274       panic("dirlink read");
5275     if(de.inum == 0)
5276       break;
5277   }
5278
5279   strncpy(de.name, name, DIRSIZ);
5280   de.inum = ino;
5281   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5282     panic("dirlink");
5283
5284   return 0;
5285 }
5286
5287
5288
5289
5290
5291
5292
5293
5294
5295
5296
5297
5298
5299
```

```
5300 // Paths
5301
5302 // Copy the next path element from path into name.
5303 // Return a pointer to the element following the copied one.
5304 // The returned path has no leading slashes,
5305 // so the caller can check *path=='\0' to see if the name is the last one.
5306 // If no name to remove, return 0.
5307 //
5308 // Examples:
5309 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5310 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5311 //   skipelem("", name) = skipelem("////", name) = 0
5312 //
5313 static char*
5314 skipelem(char *path, char *name)
5315 {
5316   char *s;
5317   int len;
5318
5319   while(*path == '/')
5320     path++;
5321   if(*path == 0)
5322     return 0;
5323   s = path;
5324   while(*path != '/' && *path != 0)
5325     path++;
5326   len = path - s;
5327   if(len >= DIRSIZ)
5328     memmove(name, s, DIRSIZ);
5329   else {
5330     memmove(name, s, len);
5331     name[len] = 0;
5332   }
5333   while(*path == '/')
5334     path++;
5335   return path;
5336 }
5337
5338
5339
5340
5341
5342
5343
5344
5345
5346
5347
5348
5349
```



```
5350 // Look up and return the inode for a path name.
5351 // If parent != 0, return the inode for the parent and copy the final
5352 // path element into name, which must have room for DIRSIZ bytes.
5353 static struct inode*
5354 _namei(char *path, int parent, char *name)
5355 {
5356   struct inode *ip, *next;
5357
5358 //  cprintf("XXX _namei %s\n", path);
5359   if(*path == '/')
5360     ip = iget(ROOTDEV, 1);
5361   else
5362     ip = idup(cp->cwd);
5363
5364   while((path = skipelem(path, name)) != 0){
5365 //    cprintf("XXX ready to ilock\n");
5366     ilock(ip);
5367 //    cprintf("XXX here %d\n", ip->type);
5368     if(ip->type != T_DIR){
5369       iunlockput(ip);
5370       return 0;
5371     }
5372     if(parent && *path == '\0'){
5373       // Stop one level early.
5374       iunlock(ip);
5375       return ip;
5376     }
5377     if((next = dirlookup(ip, name, 0)) == 0){
5378       iunlockput(ip);
5379       return 0;
5380     }
5381     iunlockput(ip);
5382     ip = next;
5383   }
5384   if(parent){
5385     iput(ip);
5386     return 0;
5387   }
5388   return ip;
5389 }
5390
5391 struct inode*
5392 namei(char *path)
5393 {
5394   char name[DIRSIZ];
5395   return _namei(path, 0, name);
5396 }
5397
5398
5399
```

```
5400 struct inode*
5401 nameiparent(char *path, char *name)
5402 {
5403   return _namei(path, 1, name);
5404 }
5405
5406
5407
5408
5409
5410
5411
5412
5413
5414
5415
5416
5417
5418
5419
5420
5421
5422
5423
5424
5425
5426
5427
5428
5429
5430
5431
5432
5433
5434
5435
5436
5437
5438
5439
5440
5441
5442
5443
5444
5445
5446
5447
5448
5449
```



```
5450 #include "types.h"
5451 #include "defs.h"
5452 #include "param.h"
5453 #include "file.h"
5454 #include "spinlock.h"
5455 #include "dev.h"
5456
5457 struct devsw devsw[NDEV];
5458 struct spinlock file_table_lock;
5459 struct file file[NFILE];
5460
5461 void
5462 fileinit(void)
5463 {
5464   initlock(&file_table_lock, "file_table");
5465 }
5466
5467 // Allocate a file structure.
5468 struct file*
5469 filealloc(void)
5470 {
5471   int i;
5472
5473   acquire(&file_table_lock);
5474   for(i = 0; i < NFILE; i++){
5475     if(file[i].type == FD_CLOSED){
5476       file[i].type = FD_NONE;
5477       file[i].ref = 1;
5478       release(&file_table_lock);
5479       return file + i;
5480     }
5481   }
5482   release(&file_table_lock);
5483   return 0;
5484 }
5485
5486 // Increment ref count for file f.
5487 struct file*
5488 filedup(struct file *f)
5489 {
5490   acquire(&file_table_lock);
5491   if(f->ref < 1 || f->type == FD_CLOSED)
5492     panic("filedup");
5493   f->ref++;
5494   release(&file_table_lock);
5495   return f;
5496 }
5497
5498
5499
```

```
5500 // Close file f.  (Decrement ref count, close when reaches 0.)
5501 void
5502 fileclose(struct file *f)
5503 {
5504   struct file ff;
5505
5506   acquire(&file_table_lock);
5507   if(f->ref < 1 || f->type == FD_CLOSED)
5508   {
5509       cprintf("ref: %d type: %d\n", f->ref, f->type);
5510       panic("fileclose: file closed");
5511   }
5512   if(--f->ref > 0){
5513     release(&file_table_lock);
5514     return;
5515   }
5516   ff = *f;
5517   f->ref = 0;
5518   f->type = FD_CLOSED;
5519   release(&file_table_lock);
5520
5521   if(ff.type == FD_PIPE)
5522     pipeclose(ff.pipe, ff.writable);
5523   else if(ff.type == FD_INODE)
5524     iput(ff.ip);
5525   else
5526   {
5527       cprintf("type: %d\n");
5528       panic("fileclose: file type error");
5529   }
5530 }
5531
5532 // Get metadata about file f.
5533 int
5534 filestat(struct file *f, struct stat *st)
5535 {
5536   if(f->type == FD_INODE){
5537     ilock(f->ip);
5538     stati(f->ip, st);
5539     iunlock(f->ip);
5540     return 0;
5541   }
5542   return -1;
5543 }
5544
5545
5546
5547
5548
5549
```



```
5550 // Read from file f.  Addr is kernel address.
5551 int
5552 fileread(struct file *f, char *addr, int n)
5553 {
5554   int r;
5555
5556   if(f->readable == 0)
5557     return -1;
5558   if(f->type == FD_PIPE)
5559     return piperead(f->pipe, addr, n);
5560   if(f->type == FD_INODE){
5561     ilock(f->ip);
5562     if((r = readi(f->ip, addr, f->off, n)) > 0)
5563       f->off += r;
5564     iunlock(f->ip);
5565     return r;
5566   }
5567   panic("fileread");
5568 }
5569
5570 // Write to file f.  Addr is kernel address.
5571 int
5572 filewrite(struct file *f, char *addr, int n)
5573 {
5574   int r;
5575
5576   if(f->writable == 0)
5577     return -1;
5578   if(f->type == FD_PIPE)
5579     return pipewrite(f->pipe, addr, n);
5580   if(f->type == FD_INODE){
5581     ilock(f->ip);
5582     if((r = writei(f->ip, addr, f->off, n)) > 0)
5583       f->off += r;
5584     iunlock(f->ip);
5585     return r;
5586   }
5587   panic("filewrite");
5588 }
5589
5590
5591
5592
5593
5594
5595
5596
5597
5598
5599
```

```
5600 #include "types.h"
5601 #include "defs.h"
5602 #include "param.h"
5603 #include "stat.h"
5604 #include "mmu.h"
5605 #include "proc.h"
5606 #include "fs.h"
5607 #include "fsvar.h"
5608 #include "file.h"
5609 #include "fcntl.h"
5610
5611 // Fetch the nth word-sized system call argument as a file descriptor
5612 // and return both the descriptor and the corresponding struct file.
5613 static int
5614 argfd(int n, int *pfd, struct file **pf)
5615 {
5616   int fd;
5617   struct file *f;
5618
5619   if(argint(n, &fd) < 0)
5620     return -1;
5621   if(fd < 0 || fd >= NOFILE || (f=cp->ofile[fd]) == 0)
5622     return -1;
5623   if(pfd)
5624     *pfd = fd;
5625   if(pf)
5626     *pf = f;
5627   return 0;
5628 }
5629
5630 // Allocate a file descriptor for the given file.
5631 // Takes over file reference from caller on success.
5632 static int
5633 fdalloc(struct file *f)
5634 {
5635   int fd;
5636
5637   for(fd = 0; fd < NOFILE; fd++){
5638     if(cp->ofile[fd] == 0){
5639       cp->ofile[fd] = f;
5640       return fd;
5641     }
5642   }
5643   return -1;
5644 }
5645
5646
5647
5648
5649
```



```
5650 int
5651 sys_read(void)
5652 {
5653   struct file *f;
5654   int n;
5655   char *cp;
5656
5657   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &cp, n) < 0)
5658     return -1;
5659   return fileread(f, cp, n);
5660 }
5661
5662 int
5663 sys_write(void)
5664 {
5665   struct file *f;
5666   int n;
5667   char *cp;
5668
5669   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &cp, n) < 0)
5670     return -1;
5671   return filewrite(f, cp, n);
5672 }
5673
5674 int
5675 sys_dup(void)
5676 {
5677   struct file *f;
5678   int fd;
5679
5680   if(argfd(0, 0, &f) < 0)
5681     return -1;
5682   if((fd=fdalloc(f)) < 0)
5683     return -1;
5684   filedup(f);
5685   return fd;
5686 }
5687
5688 int
5689 sys_close(void)
5690 {
5691   int fd;
5692   struct file *f;
5693
5694   if(argfd(0, &fd, &f) < 0)
5695     return -1;
5696   cp->ofile[fd] = 0;
5697   fileclose(f);
5698   return 0;
5699 }
```

```
5700 int
5701 sys_fstat(void)
5702 {
5703   struct file *f;
5704   struct stat *st;
5705
5706   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
5707     return -1;
5708   return filestat(f, st);
5709 }
5710
5711 // Create the path new as a link to the same inode as old.
5712 int
5713 sys_link(void)
5714 {
5715   char name[DIRSIZ], *new, *old;
5716   struct inode *dp, *ip;
5717
5718   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
5719     return -1;
5720   if((ip = namei(old)) == 0)
5721     return -1;
5722   ilock(ip);
5723   if(ip->type == T_DIR){
5724     iunlockput(ip);
5725     return -1;
5726   }
5727   ip->nlink++;
5728   iupdate(ip);
5729   iunlock(ip);
5730
5731   if((dp = nameiparent(new, name)) == 0)
5732     goto  bad;
5733   ilock(dp);
5734   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0)
5735     goto bad;
5736   iunlockput(dp);
5737   iput(ip);
5738   return 0;
5739
5740 bad:
5741   if(dp)
5742     iunlockput(dp);
5743   ilock(ip);
5744   ip->nlink--;
5745   iupdate(ip);
5746   iunlockput(ip);
5747   return -1;
5748 }
5749
```



```
5750 // Is the directory dp empty except for "." and ".." ?
5751 static int
5752 isdirempty(struct inode *dp)
5753 {
5754   int off;
5755   struct dirent de;
5756
5757   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
5758     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5759       panic("isdirempty: readi");
5760     if(de.inum != 0)
5761       return 0;
5762   }
5763   return 1;
5764 }
5765
5766 int
5767 sys_unlink(void)
5768 {
5769   struct inode *ip, *dp;
5770   struct dirent de;
5771   char name[DIRSIZ], *path;
5772   uint off;
5773
5774   if(argstr(0, &path) < 0)
5775     return -1;
5776   if((dp = nameiparent(path, name)) == 0)
5777     return -1;
5778   ilock(dp);
5779
5780   // Cannot unlink "." or "..".
5781   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0){
5782     iunlockput(dp);
5783     return -1;
5784   }
5785
5786   if((ip = dirlookup(dp, name, &off)) == 0){
5787     iunlockput(dp);
5788     return -1;
5789   }
5790   ilock(ip);
5791
5792   if(ip->nlink < 1)
5793     panic("unlink: nlink < 1");
5794   if(ip->type == T_DIR && !isdirempty(ip)){
5795     iunlockput(ip);
5796     iunlockput(dp);
5797     return -1;
5798   }
5799
```

```
5800   memset(&de, 0, sizeof(de));
5801   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5802     panic("unlink: writei");
5803   iunlockput(dp);
5804
5805   ip->nlink--;
5806   iupdate(ip);
5807   iunlockput(ip);
5808   return 0;
5809 }
5810
5811 static struct inode*
5812 create(char *path, int canexist, short type, short major, short minor)
5813 {
5814   uint off;
5815   struct inode *ip, *dp;
5816   char name[DIRSIZ];
5817
5818   if((dp = nameiparent(path, name)) == 0)
5819     return 0;
5820   ilock(dp);
5821
5822   if(canexist && (ip = dirlookup(dp, name, &off)) != 0){
5823     iunlockput(dp);
5824     ilock(ip);
5825     if(ip->type != type || ip->major != major || ip->minor != minor){
5826       iunlockput(ip);
5827       return 0;
5828     }
5829     return ip;
5830   }
5831
5832   if((ip = ialloc(dp->dev, type)) == 0){
5833     iunlockput(dp);
5834     return 0;
5835   }
5836   ilock(ip);
5837   ip->major = major;
5838   ip->minor = minor;
5839   ip->nlink = 1;
5840   iupdate(ip);
5841
5842   if(dirlink(dp, name, ip->inum) < 0){
5843     ip->nlink = 0;
5844     iunlockput(ip);
5845     iunlockput(dp);
5846     return 0;
5847   }
5848
5849
```



```
5850   if(type == T_DIR){  // Create . and .. entries.
5851     dp->nlink++;  // for ".."
5852     iupdate(dp);
5853     // No ip->nlink++ for ".": avoid cyclic ref count.
5854     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
5855       panic("create dots");
5856   }
5857   iunlockput(dp);
5858   return ip;
5859 }
5860
5861 int
5862 sys_open(void)
5863 {
5864   char *path;
5865   int fd, omode;
5866   struct file *f;
5867   struct inode *ip;
5868
5869   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
5870     return -1;
5871
5872   if(omode & O_CREATE){
5873     if((ip = create(path, 1, T_FILE, 0, 0)) == 0)
5874       return -1;
5875   } else {
5876     if((ip = namei(path)) == 0)
5877       return -1;
5878     ilock(ip);
5879     if(ip->type == T_DIR && (omode & (O_RDWR|O_WRONLY))){
5880       iunlockput(ip);
5881       return -1;
5882     }
5883   }
5884
5885   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
5886     if(f)
5887       fileclose(f);
5888     iunlockput(ip);
5889     return -1;
5890   }
5891   iunlock(ip);
5892
5893   f->type = FD_INODE;
5894   f->ip = ip;
5895   f->off = 0;
5896   f->readable = !(omode & O_WRONLY);
5897   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
5898
5899
```

```
5900   return fd;
5901 }
5902
5903 int
5904 sys_mknod(void)
5905 {
5906   struct inode *ip;
5907   char *path;
5908   int len;
5909   int major, minor;
5910
5911   if((len=argstr(0, &path)) < 0 ||
5912      argint(1, &major) < 0 ||
5913      argint(2, &minor) < 0 ||
5914      (ip = create(path, 0, T_DEV, major, minor)) == 0)
5915     return -1;
5916   iunlockput(ip);
5917   return 0;
5918 }
5919
5920 int
5921 sys_mkdir(void)
5922 {
5923   char *path;
5924   struct inode *ip;
5925
5926   if(argstr(0, &path) < 0 || (ip = create(path, 0, T_DIR, 0, 0)) == 0)
5927     return -1;
5928   iunlockput(ip);
5929   return 0;
5930 }
5931
5932 int
5933 sys_chdir(void)
5934 {
5935   char *path;
5936   struct inode *ip;
5937
5938   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0)
5939     return -1;
5940   ilock(ip);
5941   if(ip->type != T_DIR){
5942     iunlockput(ip);
5943     return -1;
5944   }
5945   iunlock(ip);
5946   iput(cp->cwd);
5947   cp->cwd = ip;
5948   return 0;
5949 }
```



```
5950 int
5951 sys_exec(void)
5952 {
5953   char *path, *argv[20];
5954   int i;
5955   uint uargv, uarg;
5956
5957   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0)
5958     return -1;
5959   memset(argv, 0, sizeof(argv));
5960   for(i=0;; i++){
5961     if(i >= NELEM(argv))
5962       return -1;
5963     if(fetchint(cp, uargv+4*i, (int*)&uarg) < 0)
5964       return -1;
5965     if(uarg == 0){
5966       argv[i] = 0;
5967       break;
5968     }
5969     if(fetchstr(cp, uarg, &argv[i]) < 0)
5970       return -1;
5971   }
5972   return exec(path, argv);
5973 }
5974
5975 int
5976 sys_pipe(void)
5977 {
5978   int *fd;
5979   struct file *rf, *wf;
5980   int fd0, fd1;
5981
5982   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
5983     return -1;
5984   if(pipealloc(&rf, &wf) < 0)
5985     return -1;
5986   fd0 = -1;
5987   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
5988     if(fd0 >= 0)
5989       cp->ofile[fd0] = 0;
5990     fileclose(rf);
5991     fileclose(wf);
5992     return -1;
5993   }
5994   fd[0] = fd0;
5995   fd[1] = fd1;
5996   return 0;
5997 }
5998
5999
```

```
6000 #include "types.h"
6001 #include "param.h"
6002 #include "mmu.h"
6003 #include "proc.h"
6004 #include "defs.h"
6005 #include "x86.h"
6006 #include "elf.h"
6007
6008 int
6009 exec(char *path, char **argv)
6010 {
6011   char *mem, *s, *last;
6012   int i, argc, arglen, len, off;
6013   uint sz, sp, argp;
6014   struct elfhdr elf;
6015   struct inode *ip;
6016   struct proghdr ph;
6017
6018   if((ip = namei(path)) == 0)
6019     return -1;
6020 //  cprintf("XXX exec\n");
6021   ilock(ip);
6022
6023   // Compute memory size of new process.
6024   mem = 0;
6025   sz = 0;
6026
6027   // Program segments.
6028   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
6029     goto bad;
6030   if(elf.magic != ELF_MAGIC)
6031     goto bad;
6032   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6033     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6034       goto bad;
6035     if(ph.type != ELF_PROG_LOAD)
6036       continue;
6037     if(ph.memsz < ph.filesz)
6038       goto bad;
6039     sz += ph.memsz;
6040   }
6041
6042   // Arguments.
6043   arglen = 0;
6044   for(argc=0; argv[argc]; argc++)
6045     arglen += strlen(argv[argc]) + 1;
6046   arglen = (arglen+3) & ~3;
6047   sz += arglen + 4*(argc+1);
6048
6049
```



```
6050   // Stack.
6051   sz += PAGE;
6052
6053   // Allocate program memory.
6054   sz = (sz+PAGE-1) & ~(PAGE-1);
6055   mem = kalloc(sz);
6056   if(mem == 0)
6057     goto bad;
6058   memset(mem, 0, sz);
6059
6060   // Load program into memory.
6061   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6062     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6063       goto bad;
6064     if(ph.type != ELF_PROG_LOAD)
6065       continue;
6066     if(ph.va + ph.memsz > sz)
6067       goto bad;
6068     if(readi(ip, mem + ph.va, ph.offset, ph.filesz) != ph.filesz)
6069       goto bad;
6070     memset(mem + ph.va + ph.filesz, 0, ph.memsz - ph.filesz);
6071   }
6072   iunlockput(ip);
6073
6074   // Initialize stack.
6075   sp = sz;
6076   argp = sz - arglen - 4*(argc+1);
6077
6078   // Copy argv strings and pointers to stack.
6079   *(uint*)(mem+argp + 4*argc) = 0;  // argv[argc]
6080   for(i=argc-1; i>=0; i--){
6081     len = strlen(argv[i]) + 1;
6082     sp -= len;
6083     memmove(mem+sp, argv[i], len);
6084     *(uint*)(mem+argp + 4*i) = sp;  // argv[i]
6085   }
6086
6087   // Stack frame for main(argc, argv), below arguments.
6088   sp = argp;
6089   sp -= 4;
6090   *(uint*)(mem+sp) = argp;
6091   sp -= 4;
6092   *(uint*)(mem+sp) = argc;
6093   sp -= 4;
6094   *(uint*)(mem+sp) = 0xffffffff;   // fake return pc
6095
6096
6097
6098
6099
```

```
6100   // Save program name for debugging.
6101   for(last=s=path; *s; s++)
6102     if(*s == '/')
6103       last = s+1;
6104   safestrcpy(cp->name, last, sizeof(cp->name));
6105
6106   // Commit to the new image.
6107   kfree(cp->mem, cp->sz);
6108   cp->mem = mem;
6109   cp->sz = sz;
6110   cp->tf->eip = elf.entry;  // main
6111   cp->tf->esp = sp;
6112   setupsegs(cp);
6113   return 0;
6114
6115  bad:
6116   if(mem)
6117     kfree(mem, sz);
6118   iunlockput(ip);
6119   return -1;
6120 }
6121
6122
6123
6124
6125
6126
6127
6128
6129
6130
6131
6132
6133
6134
6135
6136
6137
6138
6139
6140
6141
6142
6143
6144
6145
6146
6147
6148
6149
```



```
6150 #include "types.h"
6151 #include "defs.h"
6152 #include "param.h"
6153 #include "mmu.h"
6154 #include "proc.h"
6155 #include "file.h"
6156 #include "spinlock.h"
6157
6158 #define PIPESIZE 512
6159
6160 struct pipe {
6161   int readopen;   // read fd is still open
6162   int writeopen;  // write fd is still open
6163   int writep;     // next index to write
6164   int readp;      // next index to read
6165   struct spinlock lock;
6166   char data[PIPESIZE];
6167 };
6168
6169 int
6170 pipealloc(struct file **f0, struct file **f1)
6171 {
6172   struct pipe *p;
6173
6174   p = 0;
6175   *f0 = *f1 = 0;
6176   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
6177     goto bad;
6178   if((p = (struct pipe*)kalloc(PAGE)) == 0)
6179     goto bad;
6180   p->readopen = 1;
6181   p->writeopen = 1;
6182   p->writep = 0;
6183   p->readp = 0;
6184   initlock(&p->lock, "pipe");
6185   (*f0)->type = FD_PIPE;
6186   (*f0)->readable = 1;
6187   (*f0)->writable = 0;
6188   (*f0)->pipe = p;
6189   (*f1)->type = FD_PIPE;
6190   (*f1)->readable = 0;
6191   (*f1)->writable = 1;
6192   (*f1)->pipe = p;
6193   return 0;
6194
6195  bad:
6196   if(p)
6197     kfree((char*)p, PAGE);
6198   if(*f0){
6199     (*f0)->type = FD_NONE;
```

```
6200     fileclose(*f0);
6201   }
6202   if(*f1){
6203     (*f1)->type = FD_NONE;
6204     fileclose(*f1);
6205   }
6206   return -1;
6207 }
6208
6209 void
6210 pipeclose(struct pipe *p, int writable)
6211 {
6212   acquire(&p->lock);
6213   if(writable){
6214     p->writeopen = 0;
6215     wakeup(&p->readp);
6216   } else {
6217     p->readopen = 0;
6218     wakeup(&p->writep);
6219   }
6220   release(&p->lock);
6221
6222   if(p->readopen == 0 && p->writeopen == 0)
6223     kfree((char*)p, PAGE);
6224 }
6225
6226 int
6227 pipewrite(struct pipe *p, char *addr, int n)
6228 {
6229   int i;
6230
6231   acquire(&p->lock);
6232   for(i = 0; i < n; i++){
6233     while(((p->writep + 1) % PIPESIZE) == p->readp){
6234       if(p->readopen == 0 || cp->killed){
6235         release(&p->lock);
6236         return -1;
6237       }
6238       wakeup(&p->readp);
6239       sleep(&p->writep, &p->lock);
6240     }
6241     p->data[p->writep] = addr[i];
6242     p->writep = (p->writep + 1) % PIPESIZE;
6243   }
6244   wakeup(&p->readp);
6245   release(&p->lock);
6246   return i;
6247 }
6248
6249
```



```
6250 int
6251 piperead(struct pipe *p, char *addr, int n)
6252 {
6253   int i;
6254
6255   acquire(&p->lock);
6256   while(p->readp == p->writep && p->writeopen){
6257     if(cp->killed){
6258       release(&p->lock);
6259       return -1;
6260     }
6261     sleep(&p->readp, &p->lock);
6262   }
6263   for(i = 0; i < n; i++){
6264     if(p->readp == p->writep)
6265       break;
6266     addr[i] = p->data[p->readp];
6267     p->readp = (p->readp + 1) % PIPESIZE;
6268   }
6269   wakeup(&p->writep);
6270   release(&p->lock);
6271   return i;
6272 }
6273
6274
6275
6276
6277
6278
6279
6280
6281
6282
6283
6284
6285
6286
6287
6288
6289
6290
6291
6292
6293
6294
6295
6296
6297
6298
6299
```

```
#include "types.h"

void*
memcpy(void *dst, const void *src, uint n)
{
    const uchar *s1;
    uchar *s2;

    s1 = src;
    s2 = dst;
    while(n-- > 0){
        *s2 = *s1;
        s1++;
        s2++;
    }

    return 0;
}

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

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

  return dst;
}

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

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

  return 0;
}
```



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

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

  return dst;
}

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

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

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

```
6400 // Like strncpy but guaranteed to NUL-terminate.
6401 char*
6402 safestrcpy(char *s, const char *t, int n)
6403 {
6404   char *os;
6405
6406   os = s;
6407   if(n <= 0)
6408     return os;
6409   while(--n > 0 && (*s++ = *t++) != 0)
6410     ;
6411   *s = 0;
6412   return os;
6413 }
6414
6415 int
6416 strlen(const char *s)
6417 {
6418   int n;
6419
6420   for(n = 0; s[n]; n++)
6421     ;
6422   return n;
6423 }
6424
6425
6426
6427
6428
6429
6430
6431
6432
6433
6434
6435
6436
6437
6438
6439
6440
6441
6442
6443
6444
6445
6446
6447
6448
6449
```



```
6450 // See MultiProcessor Specification Version 1.[14]
6451
6452 struct mp {             // floating pointer
6453   uchar signature[4];           // "_MP_"
6454   void *physaddr;               // phys addr of MP config table
6455   uchar length;                 // 1
6456   uchar specrev;                // [14]
6457   uchar checksum;               // all bytes must add up to 0
6458   uchar type;                   // MP system config type
6459   uchar imcrp;
6460   uchar reserved[3];
6461 };
6462
6463 struct mpconf {         // configuration table header
6464   uchar signature[4];           // "PCMP"
6465   ushort length;                // total table length
6466   uchar version;                // [14]
6467   uchar checksum;               // all bytes must add up to 0
6468   uchar product[20];            // product id
6469   uint *oemtable;               // OEM table pointer
6470   ushort oemlength;             // OEM table length
6471   ushort entry;                 // entry count
6472   uint *lapicaddr;              // address of local APIC
6473   ushort xlength;               // extended table length
6474   uchar xchecksum;              // extended table checksum
6475   uchar reserved;
6476 };
6477
6478 struct mpproc {         // processor table entry
6479   uchar type;                   // entry type (0)
6480   uchar apicid;                 // local APIC id
6481   uchar version;                // local APIC verison
6482   uchar flags;                  // CPU flags
6483     #define MPBOOT 0x02           // This proc is the bootstrap processor.
6484   uchar signature[4];           // CPU signature
6485   uint feature;                 // feature flags from CPUID instruction
6486   uchar reserved[8];
6487 };
6488
6489 struct mpioapic {       // I/O APIC table entry
6490   uchar type;                   // entry type (2)
6491   uchar apicno;                 // I/O APIC id
6492   uchar version;                // I/O APIC version
6493   uchar flags;                  // I/O APIC flags
6494   uint *addr;                  // I/O APIC address
6495 };
6496
6497
6498
6499
```

```
6500 // Table entry types
6501 #define MPPROC    0x00  // One per processor
6502 #define MPBUS     0x01  // One per bus
6503 #define MPIOAPIC  0x02  // One per I/O APIC
6504 #define MPIOINTR  0x03  // One per bus interrupt source
6505 #define MPLINTR   0x04  // One per system interrupt source
6506
6507
6508
6509
6510
6511
6512
6513
6514
6515
6516
6517
6518
6519
6520
6521
6522
6523
6524
6525
6526
6527
6528
6529
6530
6531
6532
6533
6534
6535
6536
6537
6538
6539
6540
6541
6542
6543
6544
6545
6546
6547
6548
6549
```



```
6550 // Multiprocessor bootstrap.
6551 // Search memory for MP description structures.
6552 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
6553
6554 #include "types.h"
6555 #include "defs.h"
6556 #include "param.h"
6557 #include "mp.h"
6558 #include "x86.h"
6559 #include "mmu.h"
6560 #include "proc.h"
6561
6562 struct cpu cpus[NCPU];
6563 static struct cpu *bcpu;
6564 int ismp;
6565 int ncpu;
6566 uchar ioapic_id;
6567
6568 int
6569 mp_bcpu(void)
6570 {
6571   return bcpu-cpus;
6572 }
6573
6574 static uchar
6575 sum(uchar *addr, int len)
6576 {
6577   int i, sum;
6578
6579   sum = 0;
6580   for(i=0; i<len; i++)
6581     sum += addr[i];
6582   return sum;
6583 }
6584
6585 // Look for an MP structure in the len bytes at addr.
6586 static struct mp*
6587 mp_search1(uchar *addr, int len)
6588 {
6589   uchar *e, *p;
6590
6591   e = addr+len;
6592   for(p = addr; p < e; p += sizeof(struct mp))
6593     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
6594       return (struct mp*)p;
6595   return 0;
6596 }
6597
6598
6599
```

```
6600 // Search for the MP Floating Pointer Structure, which according to the
6601 // spec is in one of the following three locations:
6602 // 1) in the first KB of the EBDA;
6603 // 2) in the last KB of system base memory;
6604 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
6605 static struct mp*
6606 mp_search(void)
6607 {
6608   uchar *bda;
6609   uint p;
6610   struct mp *mp;
6611
6612   bda = (uchar*)0x400;
6613   if((p = (bda[0x0F]<<8)|bda[0x0E])){
6614     if((mp = mp_search1((uchar*)p, 1024)))
6615       return mp;
6616   } else {
6617     p = ((bda[0x14]<<8)|bda[0x13])*1024;
6618     if((mp = mp_search1((uchar*)p-1024, 1024)))
6619       return mp;
6620   }
6621   return mp_search1((uchar*)0xF0000, 0x10000);
6622 }
6623
6624 // Search for an MP configuration table.  For now,
6625 // don't accept the default configurations (physaddr == 0).
6626 // Check for correct signature, calculate the checksum and,
6627 // if correct, check the version.
6628 // To do: check extended table checksum.
6629 static struct mpconf*
6630 mp_config(struct mp **pmp)
6631 {
6632   struct mpconf *conf;
6633   struct mp *mp;
6634
6635   if((mp = mp_search()) == 0 || mp->physaddr == 0)
6636     return 0;
6637   conf = (struct mpconf*)mp->physaddr;
6638   if(memcmp(conf, "PCMP", 4) != 0)
6639     return 0;
6640   if(conf->version != 1 && conf->version != 4)
6641     return 0;
6642   if(sum((uchar*)conf, conf->length) != 0)
6643     return 0;
6644   *pmp = mp;
6645   return conf;
6646 }
6647
6648
6649
```



```
6650 void
6651 mp_init(void)
6652 {
6653   uchar *p, *e;
6654   struct mp *mp;
6655   struct mpconf *conf;
6656   struct mpproc *proc;
6657   struct mpioapic *ioapic;
6658
6659   bcpu = &cpus[ncpu];
6660   if((conf = mp_config(&mp)) == 0)
6661     return;
6662
6663   ismp = 1;
6664   lapic = (uint*)conf->lapicaddr;
6665
6666   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
6667     switch(*p){
6668     case MPPROC:
6669       proc = (struct mpproc*)p;
6670       cpus[ncpu].apicid = proc->apicid;
6671       if(proc->flags & MPBOOT)
6672         bcpu = &cpus[ncpu];
6673       ncpu++;
6674       p += sizeof(struct mpproc);
6675       continue;
6676     case MPIOAPIC:
6677       ioapic = (struct mpioapic*)p;
6678       ioapic_id = ioapic->apicno;
6679       p += sizeof(struct mpioapic);
6680       continue;
6681     case MPBUS:
6682     case MPIOINTR:
6683     case MPLINTR:
6684       p += 8;
6685       continue;
6686     default:
6687       cprintf("mp_init: unknown config type %x\n", *p);
6688       panic("mp_init");
6689     }
6690   }
6691
6692   if(mp->imcrp){
6693     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
6694     // But it would on real hardware.
6695     outb(0x22, 0x70);   // Select IMCR
6696     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
6697   }
6698 }
6699
```

```
6700 // The local APIC manages internal (non-I/O) interrupts.
6701 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
6702
6703 #include "types.h"
6704 #include "traps.h"
6705
6706 // Local APIC registers, divided by 4 for use as uint[] indices.
6707 #define ID      (0x0020/4)   // ID
6708 #define VER     (0x0030/4)   // Version
6709 #define TPR     (0x0080/4)   // Task Priority
6710 #define EOI     (0x00B0/4)   // EOI
6711 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
6712   #define ENABLE     0x00000100   // Unit Enable
6713 #define ESR     (0x0280/4)   // Error Status
6714 #define ICRLO   (0x0300/4)   // Interrupt Command
6715   #define INIT       0x00000500   // INIT/RESET
6716   #define STARTUP    0x00000600   // Startup IPI
6717   #define DELIVS     0x00001000   // Delivery status
6718   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
6719   #define LEVEL      0x00008000   // Level triggered
6720   #define BCAST      0x00080000   // Send to all APICs, including self.
6721 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
6722 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
6723   #define X1         0x0000000B   // divide counts by 1
6724   #define PERIODIC   0x00020000   // Periodic
6725 #define PCINT   (0x0340/4)   // Performance Counter LVT
6726 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
6727 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
6728 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
6729   #define MASKED     0x00010000   // Interrupt masked
6730 #define TICR    (0x0380/4)   // Timer Initial Count
6731 #define TCCR    (0x0390/4)   // Timer Current Count
6732 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
6733
6734 volatile uint *lapic;  // Initialized in mp.c
6735
6736 void
6737 lapic_init(int c)
6738 {
6739   if(!lapic)
6740     return;
6741
6742   // Enable local APIC; set spurious interrupt vector.
6743   lapic[SVR] = ENABLE | (IRQ_OFFSET+IRQ_SPURIOUS);
6744
6745   // The timer repeatedly counts down at bus frequency
6746   // from lapic[TICR] and then issues an interrupt.
6747   // Lapic[TCCR] is the current counter value.
6748   // If xv6 cared more about precise timekeeping, the
6749   // values of TICR and TCCR would be calibrated using
```



```
6750   // an external time source.
6751   lapic[TDCR] = X1;
6752   lapic[TICR] = 10000000;
6753   lapic[TCCR] = 10000000;
6754   lapic[TIMER] = PERIODIC | (IRQ_OFFSET + IRQ_TIMER);
6755
6756   // Disable logical interrupt lines.
6757   lapic[LINT0] = MASKED;
6758   lapic[LINT1] = MASKED;
6759
6760   // Disable performance counter overflow interrupts
6761   // on machines that provide that interrupt entry.
6762   if(((lapic[VER]>>16) & 0xFF) >= 4)
6763     lapic[PCINT] = MASKED;
6764
6765   // Map error interrupt to IRQ_ERROR.
6766   lapic[ERROR] = IRQ_OFFSET+IRQ_ERROR;
6767
6768   // Clear error status register (requires back-to-back writes).
6769   lapic[ESR] = 0;
6770   lapic[ESR] = 0;
6771
6772   // Ack any outstanding interrupts.
6773   lapic[EOI] = 0;
6774
6775   // Send an Init Level De-Assert to synchronise arbitration ID's.
6776   lapic[ICRHI] = 0;
6777   lapic[ICRLO] = BCAST | INIT | LEVEL;
6778   while(lapic[ICRLO] & DELIVS)
6779     ;
6780
6781   // Enable interrupts on the APIC (but not on the processor).
6782   lapic[TPR] = 0;
6783 }
6784
6785 int
6786 cpu(void)
6787 {
6788   if(lapic)
6789     return lapic[ID]>>24;
6790   return 0;
6791 }
6792
6793 // Acknowledge interrupt.
6794 void
6795 lapic_eoi(void)
6796 {
6797   if(lapic)
6798     lapic[EOI] = 0;
6799 }
```

```
6800 // Spin for a given number of microseconds.
6801 // On real hardware would want to tune this dynamically.
6802 static void
6803 microdelay(int us)
6804 {
6805   volatile int j = 0;
6806
6807   while(us-- > 0)
6808     for(j=0; j<10000; j++);
6809 }
6810
6811 // Start additional processor running bootstrap code at addr.
6812 // See Appendix B of MultiProcessor Specification.
6813 void
6814 lapic_startap(uchar apicid, uint addr)
6815 {
6816   int i;
6817   volatile int j = 0;
6818
6819   // Send INIT interrupt to reset other CPU.
6820   lapic[ICRHI] = apicid<<24;
6821   lapic[ICRLO] = INIT | LEVEL;
6822   microdelay(10);
6823
6824   // Send startup IPI (twice!) to enter bootstrap code.
6825   for(i = 0; i < 2; i++){
6826     lapic[ICRHI] = apicid<<24;
6827     lapic[ICRLO] = STARTUP | (addr>>12);
6828     for(j=0; j<10000; j++);  // 200us
6829   }
6830 }
6831
6832
6833
6834
6835
6836
6837
6838
6839
6840
6841
6842
6843
6844
6845
6846
6847
6848
6849
```



```
6850 // The I/O APIC manages hardware interrupts for an SMP system.
6851 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
6852 // See also picirq.c.
6853
6854 #include "types.h"
6855 #include "defs.h"
6856 #include "traps.h"
6857
6858 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
6859
6860 #define REG_ID     0x00  // Register index: ID
6861 #define REG_VER    0x01  // Register index: version
6862 #define REG_TABLE  0x10  // Redirection table base
6863
6864 // The redirection table starts at REG_TABLE and uses
6865 // two registers to configure each interrupt.
6866 // The first (low) register in a pair contains configuration bits.
6867 // The second (high) register contains a bitmask telling which
6868 // CPUs can serve that interrupt.
6869 #define INT_DISABLED   0x00100000  // Interrupt disabled
6870 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
6871 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
6872 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
6873
6874 volatile struct ioapic *ioapic;
6875
6876 // IO APIC MMIO structure: write reg, then read or write data.
6877 struct ioapic {
6878   uint reg;
6879   uint pad[3];
6880   uint data;
6881 };
6882
6883 static uint
6884 ioapic_read(int reg)
6885 {
6886   ioapic->reg = reg;
6887   return ioapic->data;
6888 }
6889
6890 static void
6891 ioapic_write(int reg, uint data)
6892 {
6893   ioapic->reg = reg;
6894   ioapic->data = data;
6895 }
6896
6897
6898
6899
```

```
6900 void
6901 ioapic_init(void)
6902 {
6903   int i, id, maxintr;
6904
6905   if(!ismp)
6906     return;
6907
6908   ioapic = (volatile struct ioapic*)IOAPIC;
6909   maxintr = (ioapic_read(REG_VER) >> 16) & 0xFF;
6910   id = ioapic_read(REG_ID) >> 24;
6911   if(id != ioapic_id)
6912     cprintf("ioapic_init: id isn't equal to ioapic_id; not a MP\n");
6913
6914   // Mark all interrupts edge-triggered, active high, disabled,
6915   // and not routed to any CPUs.
6916   for(i = 0; i <= maxintr; i++){
6917     ioapic_write(REG_TABLE+2*i, INT_DISABLED | (IRQ_OFFSET + i));
6918     ioapic_write(REG_TABLE+2*i+1, 0);
6919   }
6920 }
6921
6922 void
6923 ioapic_enable(int irq, int cpunum)
6924 {
6925   if(!ismp)
6926     return;
6927
6928   // Mark interrupt edge-triggered, active high,
6929   // enabled, and routed to the given cpunum,
6930   // which happens to be that cpu's APIC ID.
6931   ioapic_write(REG_TABLE+2*irq, IRQ_OFFSET + irq);
6932   ioapic_write(REG_TABLE+2*irq+1, cpunum << 24);
6933 }
6934
6935
6936
6937
6938
6939
6940
6941
6942
6943
6944
6945
6946
6947
6948
6949
```



```
6950 #ifndef XV6_PICIRQ_H_
6951 #define XV6_PICIRQ_H_
6952 #include "x86.h"
6953
6954 typedef void (*irq_handler_t)(struct trapframe *);
6955 extern irq_handler_t irq_handler[];
6956 void reg_irq_handler(int irq_num, irq_handler_t handler);
6957 #define IRQ_MAX         256
6958
6959 #endif // XV6_PICIRQ_H_
6960
6961
6962
6963
6964
6965
6966
6967
6968
6969
6970
6971
6972
6973
6974
6975
6976
6977
6978
6979
6980
6981
6982
6983
6984
6985
6986
6987
6988
6989
6990
6991
6992
6993
6994
6995
6996
6997
6998
6999
```

```
7000 // Intel 8259A programmable interrupt controllers.
7001
7002 #include "types.h"
7003 #include "x86.h"
7004 #include "picirq.h"
7005 #include "traps.h"
7006 #include "defs.h"
7007
7008 irq_handler_t irq_handler[IRQ_MAX];
7009 // I/O Addresses of the two programmable interrupt controllers
7010 #define IO_PIC1         0x20    // Master (IRQs 0-7)
7011 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
7012
7013 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
7014
7015 // Current IRQ mask.
7016 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
7017 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
7018
7019 void
7020 reg_irq_handler(int irq_num, irq_handler_t handler)
7021 {
7022     irq_handler[irq_num] = handler;
7023 }
7024
7025 static void
7026 pic_setmask(ushort mask)
7027 {
7028   irqmask = mask;
7029   outb(IO_PIC1+1, mask);
7030   outb(IO_PIC2+1, mask >> 8);
7031 }
7032
7033 void
7034 pic_enable(int irq)
7035 {
7036   pic_setmask(irqmask & ~(1<<irq));
7037 }
7038
7039
7040
7041
7042
7043
7044
7045
7046
7047
7048
7049
```



```
7050 // Initialize the 8259A interrupt controllers.
7051 void
7052 pic_init(void)
7053 {
7054   // mask all interrupts
7055   outb(IO_PIC1+1, 0xFF);
7056   outb(IO_PIC2+1, 0xFF);
7057
7058   // Set up master (8259A-1)
7059
7060   // ICW1:  0001g0hi
7061   //    g:  0 = edge triggering, 1 = level triggering
7062   //    h:  0 = cascaded PICs, 1 = master only
7063   //    i:  0 = no ICW4, 1 = ICW4 required
7064   outb(IO_PIC1, 0x11);
7065
7066   // ICW2:  Vector offset
7067   outb(IO_PIC1+1, IRQ_OFFSET);
7068
7069   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
7070   //        (slave PIC) 3-bit # of slave's connection to master
7071   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
7072
7073   // ICW4:  000nbmap
7074   //    n:  1 = special fully nested mode
7075   //    b:  1 = buffered mode
7076   //    m:  0 = slave PIC, 1 = master PIC
7077   //      (ignored when b is 0, as the master/slave role
7078   //      can be hardwired).
7079   //    a:  1 = Automatic EOI mode
7080   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
7081   outb(IO_PIC1+1, 0x3);
7082
7083   // Set up slave (8259A-2)
7084   outb(IO_PIC2, 0x11);                  // ICW1
7085   outb(IO_PIC2+1, IRQ_OFFSET + 8);      // ICW2
7086   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
7087   // NB Automatic EOI mode doesn't tend to work on the slave.
7088   // Linux source code says it's "to be investigated".
7089   outb(IO_PIC2+1, 0x3);                 // ICW4
7090
7091   // OCW3:  0ef01prs
7092   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
7093   //    p:  0 = no polling, 1 = polling mode
7094   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
7095   outb(IO_PIC1, 0x68);             // clear specific mask
7096   outb(IO_PIC1, 0x0a);             // read IRR by default
7097
7098   outb(IO_PIC2, 0x68);             // OCW3
7099   outb(IO_PIC2, 0x0a);             // OCW3
```

```
7100   if(irqmask != 0xFFFF)
7101     pic_setmask(irqmask);
7102   memset(irq_handler, 0, sizeof(irq_handler));
7103 }
7104
7105
7106
7107
7108
7109
7110
7111
7112
7113
7114
7115
7116
7117
7118
7119
7120
7121
7122
7123
7124
7125
7126
7127
7128
7129
7130
7131
7132
7133
7134
7135
7136
7137
7138
7139
7140
7141
7142
7143
7144
7145
7146
7147
7148
7149
```



```
7150 // PC keyboard interface constants
7151
7152 #define KBSTATP         0x64    // kbd controller status port(I)
7153 #define KBS_DIB         0x01    // kbd data in buffer
7154 #define KBDATAP         0x60    // kbd data port(I)
7155
7156 #define NO              0
7157
7158 #define SHIFT           (1<<0)
7159 #define CTL             (1<<1)
7160 #define ALT             (1<<2)
7161
7162 #define CAPSLOCK        (1<<3)
7163 #define NUMLOCK         (1<<4)
7164 #define SCROLLLOCK      (1<<5)
7165
7166 #define E0ESC           (1<<6)
7167
7168 // Special keycodes
7169 #define KEY_HOME        0xE0
7170 #define KEY_END         0xE1
7171 #define KEY_UP          0xE2
7172 #define KEY_DN          0xE3
7173 #define KEY_LF          0xE4
7174 #define KEY_RT          0xE5
7175 #define KEY_PGUP        0xE6
7176 #define KEY_PGDN        0xE7
7177 #define KEY_INS         0xE8
7178 #define KEY_DEL         0xE9
7179
7180 // C('A') == Control-A
7181 #define C(x) (x - '@')
7182
7183 static uchar shiftcode[256] =
7184 {
7185   [0x1D] CTL,
7186   [0x2A] SHIFT,
7187   [0x36] SHIFT,
7188   [0x38] ALT,
7189   [0x9D] CTL,
7190   [0xB8] ALT
7191 };
7192
7193 static uchar togglecode[256] =
7194 {
7195   [0x3A] CAPSLOCK,
7196   [0x45] NUMLOCK,
7197   [0x46] SCROLLLOCK
7198 };
7199
```

```
7200 static uchar normalmap[256] =
7201 {
7202   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
7203   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
7204   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
7205   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
7206   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
7207   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
7208   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
7209   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7210   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7211   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7212   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7213   [0x9C] '\n',      // KP_Enter
7214   [0xB5] '/',       // KP_Div
7215   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7216   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7217   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7218   [0x97] KEY_HOME,  [0xCF] KEY_END,
7219   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7220 };
7221
7222 static uchar shiftmap[256] =
7223 {
7224   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
7225   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
7226   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
7227   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
7228   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
7229   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
7230   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
7231   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7232   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7233   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7234   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7235   [0x9C] '\n',      // KP_Enter
7236   [0xB5] '/',       // KP_Div
7237   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7238   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7239   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7240   [0x97] KEY_HOME,  [0xCF] KEY_END,
7241   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7242 };
7243
7244
7245
7246
7247
7248
7249
```



```
7250 static uchar ctlmap[256] =
7251 {
7252   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7253   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7254   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
7255   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
7256   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
7257   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
7258   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
7259   [0x9C] '\r',      // KP_Enter
7260   [0xB5] C('/'),    // KP_Div
7261   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7262   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7263   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7264   [0x97] KEY_HOME,  [0xCF] KEY_END,
7265   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7266 };
7267
7268
7269
7270
7271
7272
7273
7274
7275
7276
7277
7278
7279
7280
7281
7282
7283
7284
7285
7286
7287
7288
7289
7290
7291
7292
7293
7294
7295
7296
7297
7298
7299
```

```
7300 #include "types.h"
7301 #include "x86.h"
7302 #include "defs.h"
7303 #include "kbd.h"
7304
7305 int
7306 kbd_getc(void)
7307 {
7308   static uint shift;
7309   static uchar *charcode[4] = {
7310     normalmap, shiftmap, ctlmap, ctlmap
7311   };
7312   uint st, data, c;
7313
7314   st = inb(KBSTATP);
7315   if((st & KBS_DIB) == 0)
7316     return -1;
7317   data = inb(KBDATAP);
7318
7319   if(data == 0xE0){
7320     shift |= E0ESC;
7321     return 0;
7322   } else if(data & 0x80){
7323     // Key released
7324     data = (shift & E0ESC ? data : data & 0x7F);
7325     shift &= ~(shiftcode[data] | E0ESC);
7326     return 0;
7327   } else if(shift & E0ESC){
7328     // Last character was an E0 escape; or with 0x80
7329     data |= 0x80;
7330     shift &= ~E0ESC;
7331   }
7332
7333   shift |= shiftcode[data];
7334   shift ^= togglecode[data];
7335   c = charcode[shift & (CTL | SHIFT)][data];
7336   if(shift & CAPSLOCK){
7337     if('a' <= c && c <= 'z')
7338       c += 'A' - 'a';
7339     else if('A' <= c && c <= 'Z')
7340       c += 'a' - 'A';
7341   }
7342   return c;
7343 }
7344
7345 void
7346 kbd_intr(void)
7347 {
7348   console_intr(kbd_getc);
7349 }
```



```
7350 // Console input and output.
7351 // Input is from the keyboard only.
7352 // Output is written to the screen and the printer port.
7353
7354 #include "types.h"
7355 #include "defs.h"
7356 #include "param.h"
7357 #include "traps.h"
7358 #include "spinlock.h"
7359 #include "dev.h"
7360 #include "mmu.h"
7361 #include "proc.h"
7362 #include "x86.h"
7363 #include "thread.h"
7364 #include "lwip/sockets.h"
7365
7366 #define CRTPORT 0x3d4
7367 #define LPTPORT 0x378
7368 #define BACKSPACE 0x100
7369
7370 static ushort *crt = (ushort*)0xb8000;  // CGA memory
7371
7372 static struct spinlock console_lock;
7373 int panicked = 0;
7374 int use_console_lock = 0;
7375
7376 // Copy console output to parallel port, which you can tell
7377 // .bochsrc to copy to the stdout:
7378 //   parport1: enabled=1, file="/dev/stdout"
7379 static void
7380 lpt_putc(int c)
7381 {
7382   int i;
7383
7384   for(i = 0; !(inb(LPTPORT+1) & 0x80) && i < 12800; i++)
7385     ;
7386   if(c == BACKSPACE)
7387     c = '\b';
7388   outb(LPTPORT+0, c);
7389   outb(LPTPORT+2, 0x08|0x04|0x01);
7390   outb(LPTPORT+2, 0x08);
7391 }
7392
7393
7394
7395
7396
7397
7398
7399
```

```
7400 static void
7401 cga_putc(int c)
7402 {
7403   int pos;
7404
7405   // Cursor position: col + 80*row.
7406   outb(CRTPORT, 14);
7407   pos = inb(CRTPORT+1) << 8;
7408   outb(CRTPORT, 15);
7409   pos |= inb(CRTPORT+1);
7410
7411   if(c == '\n')
7412     pos += 80 - pos%80;
7413   else if(c == BACKSPACE){
7414     if(pos > 0)
7415       crt[--pos] = ' ' | 0x0700;
7416   } else
7417     crt[pos++] = (c&0xff) | 0x0700;  // black on white
7418
7419   if((pos/80) >= 24){  // Scroll up.
7420     memmove(crt, crt+80, sizeof(crt[0])*23*80);
7421     pos -= 80;
7422     memset(crt + pos, 0, sizeof(crt[0])*80);
7423   }
7424
7425   outb(CRTPORT, 14);
7426   outb(CRTPORT+1, pos>>8);
7427   outb(CRTPORT, 15);
7428   outb(CRTPORT+1, pos);
7429   crt[pos] = ' ' | 0x0700;
7430 }
7431
7432 void
7433 cons_putc(int c)
7434 {
7435   if(panicked){
7436     cli();
7437     for(;;)
7438       ;
7439   }
7440
7441   lpt_putc(c);
7442   cga_putc(c);
7443 }
7444
7445
7446
7447
7448
7449
```



```
7450 void
7451 printintlen(int xx, int base, int sgn, int len, char fill)
7452 {
7453   static char digits[] = "0123456789ABCDEF";
7454   char buf[16];
7455   int i = 0, neg = 0, j = 0;
7456   uint x;
7457
7458   if(sgn && xx < 0){
7459     neg = 1;
7460     x = 0 - xx;
7461   } else {
7462     x = xx;
7463   }
7464
7465   do{
7466     buf[i++] = digits[x % base];
7467   }while((x /= base) != 0);
7468   if(neg)
7469     buf[i++] = '-';
7470
7471   if (i < len)
7472   {
7473       if (neg)
7474       {
7475           cons_putc('-');
7476           i--;
7477       }
7478       j = len - i;
7479       while (j-- > 0)
7480           cons_putc(fill);
7481   }
7482   while(--i >= 0)
7483     cons_putc(buf[i]);
7484 }
7485
7486
7487
7488
7489
7490
7491
7492
7493
7494
7495
7496
7497
7498
7499
```

```
7500 void
7501 printint(int xx, int base, int sgn)
7502 {
7503   static char digits[] = "0123456789ABCDEF";
7504   char buf[16];
7505   int i = 0, neg = 0;
7506   uint x;
7507
7508   if(sgn && xx < 0){
7509     neg = 1;
7510     x = 0 - xx;
7511   } else {
7512     x = xx;
7513   }
7514
7515   do{
7516     buf[i++] = digits[x % base];
7517   }while((x /= base) != 0);
7518   if(neg)
7519     buf[i++] = '-';
7520
7521   while(--i >= 0)
7522     cons_putc(buf[i]);
7523 }
7524
7525 enum fmt_types {
7526     CHAR,
7527     SHORT,
7528     LONG,
7529     LONGLONG,
7530 };
7531
7532 // Print to the console. only understands %d, %x, %p, %s.
7533 void
7534 cprintf(char *fmt, ...)
7535 {
7536   int i, c, state, locking, len = 0;
7537   uint *argp;
7538   char *s, last, fill;
7539   enum fmt_types type;
7540
7541   locking = use_console_lock;
7542   if(locking)
7543     acquire(&console_lock);
7544
7545   argp = (uint*)(void*)&fmt + 1;
7546   state = 0;
7547   c = 0;
7548   fill = ' ';
7549   len = -1;
```



```
7550   for(i = 0; fmt[i]; i++){
7551     last = c;
7552     c = fmt[i] & 0xff;
7553     switch(state){
7554     case 0:
7555       if(c == '%')
7556         state = '%';
7557       else
7558         cons_putc(c);
7559       break;
7560
7561     case '%':
7562       switch(c){
7563       case 'l':
7564         if (last == 'l')
7565         {
7566             // ll: long long
7567             type = LONGLONG;
7568         } else {
7569             type = LONG;
7570         }
7571         break;
7572       case 'h':
7573         if (last == 'h')
7574         {
7575             // hh: char
7576             type = CHAR;
7577         } else {
7578             type = SHORT;
7579         }
7580         break;
7581       case 'c':
7582         cons_putc(*argp++);
7583         len = -1;
7584         state = 0;
7585         break;
7586       case 'd':
7587       case 'i':
7588         if (len != -1)
7589             printintlen(*argp++, 10, 1, len, fill);
7590         else
7591             printint(*argp++, 10, 1);
7592         fill = ' ';
7593         len = -1;
7594         state = 0;
7595         break;
7596       case 'u':
7597         if (len != -1)
7598             printintlen(*argp++, 10, 0, len, fill);
7599         else
```

```
7600            printint(*argp++, 10, 0);
7601        fill = ' ';
7602        len = -1;
7603        state = 0;
7604        break;
7605      case 'x':
7606      case 'p':
7607        if (len != -1)
7608            printintlen(*argp++, 16, 0, len, fill);
7609        else
7610            printint(*argp++, 16, 0);
7611        fill = ' ';
7612        len = -1;
7613        state = 0;
7614        break;
7615      case 's':
7616        s = (char*)*argp++;
7617        if(s == 0)
7618          s = "(null)";
7619        for(; *s; s++)
7620          cons_putc(*s);
7621        len = -1;
7622        state = 0;
7623        break;
7624      case '%':
7625        cons_putc('%');
7626        state = 0;
7627        break;
7628      case '0':
7629        if (len == -1)
7630        {
7631            len = 0;
7632            fill = '0';
7633        } else {
7634            len = len * 10;
7635        }
7636        break;
7637      case '1':
7638      case '2':
7639      case '3':
7640      case '4':
7641      case '5':
7642      case '6':
7643      case '7':
7644      case '8':
7645      case '9':
7646        if (len == -1)
7647            len = c - '0';
7648        else
7649            len = len * 10 + c - '0';
```



```
7650        break;
7651      default:
7652        // Print unknown % sequence to draw attention.
7653        cons_putc('%');
7654        cons_putc(c);
7655        break;
7656      }
7657      break;
7658    }
7659  }
7660
7661  if(locking)
7662    release(&console_lock);
7663 }
7664
7665 int
7666 console_write(struct inode *ip, char *buf, int n)
7667 {
7668   int i;
7669
7670   iunlock(ip);
7671   acquire(&console_lock);
7672   for(i = 0; i < n; i++)
7673     cons_putc(buf[i] & 0xff);
7674   release(&console_lock);
7675   ilock(ip);
7676
7677   return n;
7678 }
7679
7680 #define INPUT_BUF 128
7681 struct {
7682   struct spinlock lock;
7683   char buf[INPUT_BUF];
7684   int r;  // Read index
7685   int w;  // Write index
7686   int e;  // Edit index
7687 } input;
7688
7689 #define C(x)  ((x)-'@')  // Control-x
7690
7691 void
7692 console_intr(int (*getc)(void))
7693 {
7694   int c;
7695
7696   acquire(&input.lock);
7697   while((c = getc()) >= 0){
7698     switch(c){
7699     case C('P'):  // Process listing.
```

```
7700       procdump();
7701       break;
7702     case C('U'):  // Kill line.
7703       while(input.e > input.w &&
7704             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
7705         input.e--;
7706         cons_putc(BACKSPACE);
7707       }
7708       break;
7709     case C('H'):  // Backspace
7710       if(input.e > input.w){
7711         input.e--;
7712         cons_putc(BACKSPACE);
7713       }
7714       break;
7715     default:
7716       if(c != 0 && input.e < input.r+INPUT_BUF){
7717         input.buf[input.e++ % INPUT_BUF] = c;
7718         cons_putc(c);
7719         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
7720           input.w = input.e;
7721           wakeup(&input.r);
7722         }
7723       }
7724       break;
7725     }
7726   }
7727   release(&input.lock);
7728 }
7729
7730 void
7731 thread_stub(void * arg)
7732 {
7733 /*    static int id = 0;
7734     id++;
7735 //    int myid = id;
7736     cprintf("thread_stub started! arg: %d\n", arg);
7737     int i;
7738     for (i=0; i<100000000; i++)
7739     {
7740         if (i % 100000 == 0)
7741         cprintf("%d:", myid);
7742     }*/
7743   static int in = 0;
7744   if (in)
7745       return;
7746   in = 1;
7747   unsigned char data[512];
7748   int s;
7749   int len;
```



```
7750   s = lwip_socket(PF_INET, SOCK_STREAM, 0);
7751   struct sockaddr_in sa;
7752   sa.sin_family = AF_INET;
7753   sa.sin_port = htons(80);
7754   sa.sin_addr.s_addr = inet_addr("192.168.1.1");
7755   len = 1;
7756   lwip_setsockopt(s, SOL_SOCKET, SO_REUSEPORT, &len, sizeof(int));
7757   lwip_bind(s, &sa, sizeof(sa));
7758   int addrlen = sizeof(sa);
7759 //  len = lwip_recvfrom(s, data, sizeof(data), 0, &sa, &addrlen);
7760 //  lwip_sendto(s, data, len, 0, &sa, addrlen);
7761   int client;
7762   lwip_listen(s, 1);
7763   while ((client = lwip_accept(s, &sa, &addrlen)) > 0)
7764   {
7765       do {
7766         len = lwip_read(client, data, sizeof(data));
7767 //        cprintf("received %d bytes\n", len);
7768         lwip_send(client, data, len, 0);
7769         if (data[0] == '!')
7770           len = -1;
7771       } while (len > 0);
7772       lwip_close(client);
7773   }
7774   lwip_close(s);
7775   in = 0;
7776 }
7777
7778
7779 int
7780 console_read(struct inode *ip, char *dst, int n)
7781 {
7782   uint target;
7783   int c;
7784   int ret;
7785 //  unsigned char data[100];
7786   int mark;
7787
7788 //  kproc_start(thread_stub, 0, 0, 0, "[stub thread]");
7789 //  for (c=0; c<10; c++)
7790 //      data[c] = 0xda;
7791   iunlock(ip);
7792   target = n;
7793   acquire(&input.lock);
7794   while(n > 0){
7795     while(input.r == input.w){
7796       if(cp->killed){
7797         release(&input.lock);
7798         ilock(ip);
7799         return -1;
```

```
7800      }
7801      mark = 0;
7802      sleep(&input.r, &input.lock);
7803 /*     do {
7804        ret = msleep_spin(&input.r, &input.lock, 1000);
7805       if (ret)
7806        {*/
7807 //           if ((mark = lwip_recv(s, data,
7808 //           sizeof(data), MSG_DONTWAIT)) > 0)
7809 //               cprintf("received from 192.168.1.1:80: %d\n", mark);
7810 /*           if (mark == 0)
7811            {
7812                for ( ; mark < 64; mark ++)
7813                    e100_send(data, sizeof(data));
7814            }
7815            int len = e100_receive(data, sizeof(data));
7816            cprintf("received: %d bytes\n", len);
7817            int i;
7818            for (i=0; i<len; i++)
7819            {
7820                cprintf("%02x ", data[i]);
7821                if ((i+1) % 10 == 0)
7822                    cprintf("\n");
7823            }
7824            cprintf("\n");*/
7825 //           kproc_start(thread_stub, (void *)100, 0, 0);
7826 //           cprintf("XXX faster!!\n");
7827 /*       }
7828      } while (ret);*/
7829    }
7830    c = input.buf[input.r++ % INPUT_BUF];
7831    if(c == C('D')){  // EOF
7832      if(n < target){
7833        // Save ^D for next time, to make sure
7834        // caller gets a 0-byte result.
7835        input.r--;
7836      }
7837      break;
7838    }
7839    *dst++ = c;
7840    --n;
7841    if(c == '\n')
7842      break;
7843  }
7844  release(&input.lock);
7845  ilock(ip);
7846
7847  return target - n;
7848 }
7849
```



```
7850 void
7851 console_init(void)
7852 {
7853   initlock(&console_lock, "console");
7854   initlock(&input.lock, "console input");
7855
7856   devsw[CONSOLE].write = console_write;
7857   devsw[CONSOLE].read = console_read;
7858   //use_console_lock = 1;
7859
7860   pic_enable(IRQ_KBD);
7861   ioapic_enable(IRQ_KBD, 0);
7862 }
7863
7864 void
7865 printstack()
7866 {
7867   int i;
7868   uint ebp;
7869   uint pcs[10];
7870   cprintf("Stack trace:\n");
7871   asm("movl %%ebp, %0" : "=r"(ebp) : );
7872   getcallerpcs((void*)(ebp+8), pcs);
7873   for(i=0; i<10; i++)
7874     cprintf("0x%08x ", pcs[i]);
7875   cprintf("\n");
7876 }
7877
7878
7879 void
7880 panic(char *s)
7881 {
7882   int i;
7883   uint pcs[10];
7884
7885   __asm __volatile("cli");
7886   use_console_lock = 0;
7887   cprintf("panic (%d): ", cpu());
7888   cprintf(s, 0);
7889   cprintf("\n", 0);
7890   getcallerpcs(&s, pcs);
7891   for(i=0; i<10; i++)
7892     cprintf(" %p", pcs[i]);
7893   panicked = 1; // freeze other CPU
7894   for(;;)
7895     ;
7896 }
7897
7898
7899
```

```
7900 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
7901 // Only used on uniprocessors;
7902 // SMP machines use the local APIC timer.
7903
7904 #include "types.h"
7905 #include "defs.h"
7906 #include "traps.h"
7907 #include "x86.h"
7908
7909 #define IO_TIMER1       0x040           // 8253 Timer #1
7910
7911 // Frequency of all three count-down timers;
7912 // (TIMER_FREQ/freq) is the appropriate count
7913 // to generate a frequency of freq Hz.
7914
7915 #define TIMER_FREQ      1193182
7916 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
7917
7918 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
7919 #define TIMER_SEL0      0x00    // select counter 0
7920 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
7921 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
7922
7923 void
7924 timer_init(void)
7925 {
7926   // Interrupt 100 times/sec.
7927   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
7928   outb(IO_TIMER1, TIMER_DIV(100) % 256);
7929   outb(IO_TIMER1, TIMER_DIV(100) / 256);
7930   pic_enable(IRQ_TIMER);
7931 }
7932
7933 int
7934 millitime(void)
7935 {
7936     return ticks * 10;
7937 }
7938
7939
7940
7941
7942
7943
7944
7945
7946
7947
7948
7949
```



```
7950 #ifndef XV6_PCI_H_
7951 #define XV6_PCI_H_
7952
7953 #include "types.h"
7954
7955 // PCI subsystem interface
7956 enum { pci_res_bus, pci_res_mem, pci_res_io, pci_res_max };
7957
7958 struct pci_bus;
7959
7960 struct pci_func {
7961     struct pci_bus *bus;   // Primary bus for bridges
7962
7963     uint32_t dev;
7964     uint32_t func;
7965
7966     uint32_t dev_id;
7967     uint32_t dev_class;
7968
7969     uint32_t reg_base[6];
7970     uint32_t reg_size[6];
7971     uint8_t irq_line;
7972 };
7973
7974 struct pci_bus {
7975     struct pci_func *parent_bridge;
7976     uint32_t busno;
7977 };
7978
7979 int pci_init(void);
7980 void pci_func_enable(struct pci_func *f);
7981
7982 int ether_send(void *buffer, int len);
7983 int ether_receive(void *buffer, int len);
7984
7985 #endif
7986
7987
7988
7989
7990
7991
7992
7993
7994
7995
7996
7997
7998
7999
```

```
8000 /* $NetBSD: pcireg.h,v 1.45 2004/02/04 06:58:24 soren Exp $       */
8001
8002 /*
8003  * Copyright (c) 1995, 1996, 1999, 2000
8004  *     Christopher G. Demetriou.  All rights reserved.
8005  * Copyright (c) 1994, 1996 Charles M. Hannum.  All rights reserved.
8006  *
8007  * Redistribution and use in source and binary forms, with or without
8008  * modification, are permitted provided that the following conditions
8009  * are met:
8010  * 1. Redistributions of source code must retain the above copyright
8011  *    notice, this list of conditions and the following disclaimer.
8012  * 2. Redistributions in binary form must reproduce the above copyright
8013  *    notice, this list of conditions and the following disclaimer in the
8014  *    documentation and/or other materials provided with the distribution.
8015  * 3. All advertising materials mentioning features or use of this software
8016  *    must display the following acknowledgement:
8017  * This product includes software developed by Charles M. Hannum.
8018  * 4. The name of the author may not be used to endorse or promote products
8019  *    derived from this software without specific prior written permission.
8020  *
8021  * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
8022  * IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
8023  * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
8024  * IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
8025  * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
8026  * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
8027  * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
8028  * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
8029  * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
8030  * THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
8031  */
8032
8033 #ifndef _DEV_PCI_PCIREG_H_
8034 #define    _DEV_PCI_PCIREG_H_
8035
8036 /*
8037  * Standardized PCI configuration information
8038  *
8039  * XXX This is not complete.
8040  */
8041
8042 #include "types.h"
8043
8044 /*
8045  * Device identification register; contains a vendor ID and a device ID.
8046  */
8047 #define    PCI_ID_REG                      0x00
8048
8049
```



```
8050 typedef uint16_t pci_vendor_id_t;
8051 typedef uint16_t pci_product_id_t;
8052
8053 #define    PCI_VENDOR_SHIFT                        0
8054 #define    PCI_VENDOR_MASK                         0xffff
8055 #define    PCI_VENDOR(id) \
8056        (((id) >> PCI_VENDOR_SHIFT) & PCI_VENDOR_MASK)
8057
8058 #define    PCI_PRODUCT_SHIFT                       16
8059 #define    PCI_PRODUCT_MASK                        0xffff
8060 #define    PCI_PRODUCT(id) \
8061        (((id) >> PCI_PRODUCT_SHIFT) & PCI_PRODUCT_MASK)
8062
8063 #define PCI_ID_CODE(vid,pid)                                        \
8064    ((((vid) & PCI_VENDOR_MASK) << PCI_VENDOR_SHIFT) |      \
8065     (((pid) & PCI_PRODUCT_MASK) << PCI_PRODUCT_SHIFT))     \
8066
8067 /*
8068  * Command and status register.
8069  */
8070 #define    PCI_COMMAND_STATUS_REG                  0x04
8071 #define    PCI_COMMAND_SHIFT                       0
8072 #define    PCI_COMMAND_MASK                        0xffff
8073 #define    PCI_STATUS_SHIFT                        16
8074 #define    PCI_STATUS_MASK                         0xffff
8075
8076 #define PCI_COMMAND_STATUS_CODE(cmd,stat)                   \
8077    ((((cmd) & PCI_COMMAND_MASK) >> PCI_COMMAND_SHIFT) |    \
8078     (((stat) & PCI_STATUS_MASK) >> PCI_STATUS_SHIFT))      \
8079
8080 #define    PCI_COMMAND_IO_ENABLE                   0x00000001
8081 #define    PCI_COMMAND_MEM_ENABLE                  0x00000002
8082 #define    PCI_COMMAND_MASTER_ENABLE               0x00000004
8083 #define    PCI_COMMAND_SPECIAL_ENABLE              0x00000008
8084 #define    PCI_COMMAND_INVALIDATE_ENABLE           0x00000010
8085 #define    PCI_COMMAND_PALETTE_ENABLE              0x00000020
8086 #define    PCI_COMMAND_PARITY_ENABLE               0x00000040
8087 #define    PCI_COMMAND_STEPPING_ENABLE             0x00000080
8088 #define    PCI_COMMAND_SERR_ENABLE                 0x00000100
8089 #define    PCI_COMMAND_BACKTOBACK_ENABLE           0x00000200
8090
8091 #define    PCI_STATUS_CAPLIST_SUPPORT              0x00100000
8092 #define    PCI_STATUS_66MHZ_SUPPORT                0x00200000
8093 #define    PCI_STATUS_UDF_SUPPORT                  0x00400000
8094 #define    PCI_STATUS_BACKTOBACK_SUPPORT           0x00800000
8095 #define    PCI_STATUS_PARITY_ERROR                 0x01000000
8096 #define    PCI_STATUS_DEVSEL_FAST                  0x00000000
8097 #define    PCI_STATUS_DEVSEL_MEDIUM                0x02000000
8098 #define    PCI_STATUS_DEVSEL_SLOW                  0x04000000
8099 #define    PCI_STATUS_DEVSEL_MASK                  0x06000000
```

```
8100 #define    PCI_STATUS_TARGET_TARGET_ABORT          0x08000000
8101 #define    PCI_STATUS_MASTER_TARGET_ABORT          0x10000000
8102 #define    PCI_STATUS_MASTER_ABORT                 0x20000000
8103 #define    PCI_STATUS_SPECIAL_ERROR                0x40000000
8104 #define    PCI_STATUS_PARITY_DETECT                0x80000000
8105
8106 /*
8107  * PCI Class and Revision Register; defines type and revision of device.
8108  */
8109 #define    PCI_CLASS_REG                   0x08
8110
8111 typedef uint8_t pci_class_t;
8112 typedef uint8_t pci_subclass_t;
8113 typedef uint8_t pci_interface_t;
8114 typedef uint8_t pci_revision_t;
8115
8116 #define    PCI_CLASS_SHIFT                         24
8117 #define    PCI_CLASS_MASK                          0xff
8118 #define    PCI_CLASS(cr) \
8119        (((cr) >> PCI_CLASS_SHIFT) & PCI_CLASS_MASK)
8120
8121 #define    PCI_SUBCLASS_SHIFT                      16
8122 #define    PCI_SUBCLASS_MASK                       0xff
8123 #define    PCI_SUBCLASS(cr) \
8124        (((cr) >> PCI_SUBCLASS_SHIFT) & PCI_SUBCLASS_MASK)
8125
8126 #define    PCI_INTERFACE_SHIFT                     8
8127 #define    PCI_INTERFACE_MASK                      0xff
8128 #define    PCI_INTERFACE(cr) \
8129        (((cr) >> PCI_INTERFACE_SHIFT) & PCI_INTERFACE_MASK)
8130
8131 #define    PCI_REVISION_SHIFT                      0
8132 #define    PCI_REVISION_MASK                       0xff
8133 #define    PCI_REVISION(cr) \
8134        (((cr) >> PCI_REVISION_SHIFT) & PCI_REVISION_MASK)
8135
8136 #define    PCI_CLASS_CODE(mainclass, subclass, interface) \
8137        ((((mainclass) & PCI_CLASS_MASK) << PCI_CLASS_SHIFT) | \
8138         (((subclass) & PCI_SUBCLASS_MASK) << PCI_SUBCLASS_SHIFT) | \
8139         (((interface) & PCI_INTERFACE_MASK) << PCI_INTERFACE_SHIFT))
8140
8141 /* base classes */
8142 #define    PCI_CLASS_PREHISTORIC                   0x00
8143 #define    PCI_CLASS_MASS_STORAGE                  0x01
8144 #define    PCI_CLASS_NETWORK                       0x02
8145 #define    PCI_CLASS_DISPLAY                       0x03
8146 #define    PCI_CLASS_MULTIMEDIA                    0x04
8147 #define    PCI_CLASS_MEMORY                        0x05
8148 #define    PCI_CLASS_BRIDGE                        0x06
8149 #define    PCI_CLASS_COMMUNICATIONS                0x07
```



```
8150 #define    PCI_CLASS_SYSTEM                        0x08
8151 #define    PCI_CLASS_INPUT                         0x09
8152 #define    PCI_CLASS_DOCK                          0x0a
8153 #define    PCI_CLASS_PROCESSOR                     0x0b
8154 #define    PCI_CLASS_SERIALBUS                     0x0c
8155 #define    PCI_CLASS_WIRELESS                      0x0d
8156 #define    PCI_CLASS_I2O                           0x0e
8157 #define    PCI_CLASS_SATCOM                        0x0f
8158 #define    PCI_CLASS_CRYPTO                        0x10
8159 #define    PCI_CLASS_DASP                          0x11
8160 #define    PCI_CLASS_UNDEFINED                     0xff
8161
8162 /* 0x00 prehistoric subclasses */
8163 #define    PCI_SUBCLASS_PREHISTORIC_MISC           0x00
8164 #define    PCI_SUBCLASS_PREHISTORIC_VGA            0x01
8165
8166 /* 0x01 mass storage subclasses */
8167 #define    PCI_SUBCLASS_MASS_STORAGE_SCSI          0x00
8168 #define    PCI_SUBCLASS_MASS_STORAGE_IDE           0x01
8169 #define    PCI_SUBCLASS_MASS_STORAGE_FLOPPY        0x02
8170 #define    PCI_SUBCLASS_MASS_STORAGE_IPI           0x03
8171 #define    PCI_SUBCLASS_MASS_STORAGE_RAID          0x04
8172 #define    PCI_SUBCLASS_MASS_STORAGE_ATA           0x05
8173 #define    PCI_SUBCLASS_MASS_STORAGE_SATA          0x06
8174 #define    PCI_SUBCLASS_MASS_STORAGE_MISC          0x80
8175
8176 /* 0x02 network subclasses */
8177 #define    PCI_SUBCLASS_NETWORK_ETHERNET           0x00
8178 #define    PCI_SUBCLASS_NETWORK_TOKENRING          0x01
8179 #define    PCI_SUBCLASS_NETWORK_FDDI               0x02
8180 #define    PCI_SUBCLASS_NETWORK_ATM                0x03
8181 #define    PCI_SUBCLASS_NETWORK_ISDN               0x04
8182 #define    PCI_SUBCLASS_NETWORK_WORLDFIP           0x05
8183 #define    PCI_SUBCLASS_NETWORK_PCIMGMULTICOMP     0x06
8184 #define    PCI_SUBCLASS_NETWORK_MISC               0x80
8185
8186 /* 0x03 display subclasses */
8187 #define    PCI_SUBCLASS_DISPLAY_VGA                0x00
8188 #define    PCI_SUBCLASS_DISPLAY_XGA                0x01
8189 #define    PCI_SUBCLASS_DISPLAY_3D                 0x02
8190 #define    PCI_SUBCLASS_DISPLAY_MISC               0x80
8191
8192 /* 0x04 multimedia subclasses */
8193 #define    PCI_SUBCLASS_MULTIMEDIA_VIDEO           0x00
8194 #define    PCI_SUBCLASS_MULTIMEDIA_AUDIO           0x01
8195 #define    PCI_SUBCLASS_MULTIMEDIA_TELEPHONY       0x02
8196 #define    PCI_SUBCLASS_MULTIMEDIA_MISC            0x80
8197
8198
8199
```

```
8200 /* 0x05 memory subclasses */
8201 #define    PCI_SUBCLASS_MEMORY_RAM                0x00
8202 #define    PCI_SUBCLASS_MEMORY_FLASH              0x01
8203 #define    PCI_SUBCLASS_MEMORY_MISC               0x80
8204
8205 /* 0x06 bridge subclasses */
8206 #define    PCI_SUBCLASS_BRIDGE_HOST               0x00
8207 #define    PCI_SUBCLASS_BRIDGE_ISA                0x01
8208 #define    PCI_SUBCLASS_BRIDGE_EISA               0x02
8209 #define    PCI_SUBCLASS_BRIDGE_MC                 0x03    /* XXX _MCA? */
8210 #define    PCI_SUBCLASS_BRIDGE_PCI                0x04
8211 #define    PCI_SUBCLASS_BRIDGE_PCMCIA             0x05
8212 #define    PCI_SUBCLASS_BRIDGE_NUBUS              0x06
8213 #define    PCI_SUBCLASS_BRIDGE_CARDBUS            0x07
8214 #define    PCI_SUBCLASS_BRIDGE_RACEWAY            0x08
8215 #define    PCI_SUBCLASS_BRIDGE_STPCI              0x09
8216 #define    PCI_SUBCLASS_BRIDGE_INFINIBAND         0x0a
8217 #define    PCI_SUBCLASS_BRIDGE_MISC               0x80
8218
8219 /* 0x07 communications subclasses */
8220 #define    PCI_SUBCLASS_COMMUNICATIONS_SERIAL     0x00
8221 #define    PCI_SUBCLASS_COMMUNICATIONS_PARALLEL   0x01
8222 #define    PCI_SUBCLASS_COMMUNICATIONS_MPSERIAL   0x02
8223 #define    PCI_SUBCLASS_COMMUNICATIONS_MODEM      0x03
8224 #define    PCI_SUBCLASS_COMMUNICATIONS_GPIB       0x04
8225 #define    PCI_SUBCLASS_COMMUNICATIONS_SMARTCARD  0x05
8226 #define    PCI_SUBCLASS_COMMUNICATIONS_MISC       0x80
8227
8228 /* 0x08 system subclasses */
8229 #define    PCI_SUBCLASS_SYSTEM_PIC                0x00
8230 #define    PCI_SUBCLASS_SYSTEM_DMA                0x01
8231 #define    PCI_SUBCLASS_SYSTEM_TIMER              0x02
8232 #define    PCI_SUBCLASS_SYSTEM_RTC                0x03
8233 #define    PCI_SUBCLASS_SYSTEM_PCIHOTPLUG         0x04
8234 #define    PCI_SUBCLASS_SYSTEM_MISC               0x80
8235
8236 /* 0x09 input subclasses */
8237 #define    PCI_SUBCLASS_INPUT_KEYBOARD            0x00
8238 #define    PCI_SUBCLASS_INPUT_DIGITIZER           0x01
8239 #define    PCI_SUBCLASS_INPUT_MOUSE               0x02
8240 #define    PCI_SUBCLASS_INPUT_SCANNER             0x03
8241 #define    PCI_SUBCLASS_INPUT_GAMEPORT            0x04
8242 #define    PCI_SUBCLASS_INPUT_MISC                0x80
8243
8244 /* 0x0a dock subclasses */
8245 #define    PCI_SUBCLASS_DOCK_GENERIC              0x00
8246 #define    PCI_SUBCLASS_DOCK_MISC                 0x80
8247
8248
8249
```



```
8250 /* 0x0b processor subclasses */
8251 #define    PCI_SUBCLASS_PROCESSOR_386             0x00
8252 #define    PCI_SUBCLASS_PROCESSOR_486             0x01
8253 #define    PCI_SUBCLASS_PROCESSOR_PENTIUM         0x02
8254 #define    PCI_SUBCLASS_PROCESSOR_ALPHA           0x10
8255 #define    PCI_SUBCLASS_PROCESSOR_POWERPC         0x20
8256 #define    PCI_SUBCLASS_PROCESSOR_MIPS            0x30
8257 #define    PCI_SUBCLASS_PROCESSOR_COPROC          0x40
8258
8259 /* 0x0c serial bus subclasses */
8260 #define    PCI_SUBCLASS_SERIALBUS_FIREWIRE        0x00
8261 #define    PCI_SUBCLASS_SERIALBUS_ACCESS          0x01
8262 #define    PCI_SUBCLASS_SERIALBUS_SSA             0x02
8263 #define    PCI_SUBCLASS_SERIALBUS_USB             0x03
8264 #define    PCI_SUBCLASS_SERIALBUS_FIBER           0x04    /* XXX _FIBRECHANI
8265 #define    PCI_SUBCLASS_SERIALBUS_SMBUS           0x05
8266 #define    PCI_SUBCLASS_SERIALBUS_INFINIBAND      0x06
8267 #define    PCI_SUBCLASS_SERIALBUS_IPMI            0x07
8268 #define    PCI_SUBCLASS_SERIALBUS_SERCOS          0x08
8269 #define    PCI_SUBCLASS_SERIALBUS_CANBUS          0x09
8270
8271 /* 0x0d wireless subclasses */
8272 #define    PCI_SUBCLASS_WIRELESS_IRDA             0x00
8273 #define    PCI_SUBCLASS_WIRELESS_CONSUMERIR       0x01
8274 #define    PCI_SUBCLASS_WIRELESS_RF               0x10
8275 #define    PCI_SUBCLASS_WIRELESS_BLUETOOTH        0x11
8276 #define    PCI_SUBCLASS_WIRELESS_BROADBAND        0x12
8277 #define    PCI_SUBCLASS_WIRELESS_802_11A          0x20
8278 #define    PCI_SUBCLASS_WIRELESS_802_11B          0x21
8279 #define    PCI_SUBCLASS_WIRELESS_MISC             0x80
8280
8281 /* 0x0e I2O (Intelligent I/O) subclasses */
8282 #define    PCI_SUBCLASS_I2O_STANDARD              0x00
8283
8284 /* 0x0f satellite communication subclasses */
8285 /* PCI_SUBCLASS_SATCOM_???                0x00    / * XXX ??? */
8286 #define    PCI_SUBCLASS_SATCOM_TV                 0x01
8287 #define    PCI_SUBCLASS_SATCOM_AUDIO              0x02
8288 #define    PCI_SUBCLASS_SATCOM_VOICE              0x03
8289 #define    PCI_SUBCLASS_SATCOM_DATA               0x04
8290
8291 /* 0x10 encryption/decryption subclasses */
8292 #define    PCI_SUBCLASS_CRYPTO_NETCOMP            0x00
8293 #define    PCI_SUBCLASS_CRYPTO_ENTERTAINMENT      0x10
8294 #define    PCI_SUBCLASS_CRYPTO_MISC               0x80
8295
8296
8297
8298
8299
```

```
8300 /* 0x11 data acquisition and signal processing subclasses */
8301 #define    PCI_SUBCLASS_DASP_DPIO                  0x00
8302 #define    PCI_SUBCLASS_DASP_TIMEFREQ              0x01
8303 #define    PCI_SUBCLASS_DASP_SYNC                  0x10
8304 #define    PCI_SUBCLASS_DASP_MGMT                  0x20
8305 #define    PCI_SUBCLASS_DASP_MISC                  0x80
8306
8307 /*
8308  * PCI BIST/Header Type/Latency Timer/Cache Line Size Register.
8309  */
8310 #define    PCI_BHLC_REG                    0x0c
8311
8312 #define    PCI_BIST_SHIFT                          24
8313 #define    PCI_BIST_MASK                           0xff
8314 #define    PCI_BIST(bhlcr) \
8315        (((bhlcr) >> PCI_BIST_SHIFT) & PCI_BIST_MASK)
8316
8317 #define    PCI_HDRTYPE_SHIFT                       16
8318 #define    PCI_HDRTYPE_MASK                        0xff
8319 #define    PCI_HDRTYPE(bhlcr) \
8320        (((bhlcr) >> PCI_HDRTYPE_SHIFT) & PCI_HDRTYPE_MASK)
8321
8322 #define    PCI_HDRTYPE_TYPE(bhlcr) \
8323        (PCI_HDRTYPE(bhlcr) & 0x7f)
8324 #define    PCI_HDRTYPE_MULTIFN(bhlcr) \
8325        ((PCI_HDRTYPE(bhlcr) & 0x80) != 0)
8326
8327 #define    PCI_LATTIMER_SHIFT                      8
8328 #define    PCI_LATTIMER_MASK                       0xff
8329 #define    PCI_LATTIMER(bhlcr) \
8330        (((bhlcr) >> PCI_LATTIMER_SHIFT) & PCI_LATTIMER_MASK)
8331
8332 #define    PCI_CACHELINE_SHIFT                     0
8333 #define    PCI_CACHELINE_MASK                      0xff
8334 #define    PCI_CACHELINE(bhlcr) \
8335        (((bhlcr) >> PCI_CACHELINE_SHIFT) & PCI_CACHELINE_MASK)
8336
8337 #define PCI_BHLC_CODE(bist,type,multi,latency,cacheline)          \
8338        ((((bist) & PCI_BIST_MASK) << PCI_BIST_SHIFT) |            \
8339         (((type) & PCI_HDRTYPE_MASK) << PCI_HDRTYPE_SHIFT) |      \
8340         (((multi)?0x80:0) << PCI_HDRTYPE_SHIFT) |                 \
8341         (((latency) & PCI_LATTIMER_MASK) << PCI_LATTIMER_SHIFT) |  \
8342         (((cacheline) & PCI_CACHELINE_MASK) << PCI_CACHELINE_SHIFT))
8343
8344 /*
8345  * PCI header type
8346  */
8347 #define PCI_HDRTYPE_DEVICE 0
8348 #define PCI_HDRTYPE_PPB           1
8349 #define PCI_HDRTYPE_PCB           2
```



```
8350 /*
8351  * Mapping registers
8352  */
8353 #define    PCI_MAPREG_START                0x10
8354 #define    PCI_MAPREG_END                  0x28
8355 #define    PCI_MAPREG_ROM                  0x30
8356 #define    PCI_MAPREG_PPB_END              0x18
8357 #define    PCI_MAPREG_PCB_END              0x14
8358
8359 #define    PCI_MAPREG_TYPE(mr)                                    \
8360        ((mr) & PCI_MAPREG_TYPE_MASK)
8361 #define    PCI_MAPREG_TYPE_MASK                    0x00000001
8362
8363 #define    PCI_MAPREG_TYPE_MEM                     0x00000000
8364 #define    PCI_MAPREG_TYPE_IO                      0x00000001
8365 #define    PCI_MAPREG_ROM_ENABLE                   0x00000001
8366
8367 #define    PCI_MAPREG_MEM_TYPE(mr)                                \
8368        ((mr) & PCI_MAPREG_MEM_TYPE_MASK)
8369 #define    PCI_MAPREG_MEM_TYPE_MASK                0x00000006
8370
8371 #define    PCI_MAPREG_MEM_TYPE_32BIT               0x00000000
8372 #define    PCI_MAPREG_MEM_TYPE_32BIT_1M            0x00000002
8373 #define    PCI_MAPREG_MEM_TYPE_64BIT               0x00000004
8374
8375 #define    PCI_MAPREG_MEM_PREFETCHABLE(mr)                        \
8376        (((mr) & PCI_MAPREG_MEM_PREFETCHABLE_MASK) != 0)
8377 #define    PCI_MAPREG_MEM_PREFETCHABLE_MASK        0x00000008
8378
8379 #define    PCI_MAPREG_MEM_ADDR(mr)                                \
8380        ((mr) & PCI_MAPREG_MEM_ADDR_MASK)
8381 #define    PCI_MAPREG_MEM_SIZE(mr)                                \
8382        (PCI_MAPREG_MEM_ADDR(mr) & -PCI_MAPREG_MEM_ADDR(mr))
8383 #define    PCI_MAPREG_MEM_ADDR_MASK                0xfffffff0
8384
8385 #define    PCI_MAPREG_MEM64_ADDR(mr)                              \
8386        ((mr) & PCI_MAPREG_MEM64_ADDR_MASK)
8387 #define    PCI_MAPREG_MEM64_SIZE(mr)                              \
8388        (PCI_MAPREG_MEM64_ADDR(mr) & -PCI_MAPREG_MEM64_ADDR(mr))
8389 #define    PCI_MAPREG_MEM64_ADDR_MASK              0xfffffffffffffff0ULL
8390
8391 #define    PCI_MAPREG_IO_ADDR(mr)                                 \
8392        ((mr) & PCI_MAPREG_IO_ADDR_MASK)
8393 #define    PCI_MAPREG_IO_SIZE(mr)                                 \
8394        (PCI_MAPREG_IO_ADDR(mr) & -PCI_MAPREG_IO_ADDR(mr))
8395 #define    PCI_MAPREG_IO_ADDR_MASK                 0xfffffffc
8396
8397 #define PCI_MAPREG_SIZE_TO_MASK(size)                             \
8398        (-(size))
8399
```

```
8400 #define PCI_MAPREG_NUM(offset)                                          \
8401         (((unsigned)(offset)-PCI_MAPREG_START)/4)
8402
8403
8404 /*
8405  * Cardbus CIS pointer (PCI rev. 2.1)
8406  */
8407 #define PCI_CARDBUS_CIS_REG 0x28
8408
8409 /*
8410  * Subsystem identification register; contains a vendor ID and a device ID.
8411  * Types/macros for PCI_ID_REG apply.
8412  * (PCI rev. 2.1)
8413  */
8414 #define PCI_SUBSYS_ID_REG 0x2c
8415
8416 /*
8417  * Capabilities link list (PCI rev. 2.2)
8418  */
8419 #define    PCI_CAPLISTPTR_REG              0x34    /* header type 0 */
8420 #define    PCI_CARDBUS_CAPLISTPTR_REG      0x14    /* header type 2 */
8421 #define    PCI_CAPLIST_PTR(cpr)    ((cpr) & 0xff)
8422 #define    PCI_CAPLIST_NEXT(cr)    (((cr) >> 8) & 0xff)
8423 #define    PCI_CAPLIST_CAP(cr)     ((cr) & 0xff)
8424
8425 #define    PCI_CAP_RESERVED0       0x00
8426 #define    PCI_CAP_PWRMGMT         0x01
8427 #define    PCI_CAP_AGP             0x02
8428 #define PCI_CAP_AGP_MAJOR(cr)      (((cr) >> 20) & 0xf)
8429 #define PCI_CAP_AGP_MINOR(cr)      (((cr) >> 16) & 0xf)
8430 #define    PCI_CAP_VPD             0x03
8431 #define    PCI_CAP_SLOTID          0x04
8432 #define    PCI_CAP_MSI             0x05
8433 #define    PCI_CAP_CPCI_HOTSWAP    0x06
8434 #define    PCI_CAP_PCIX            0x07
8435 #define    PCI_CAP_LDT             0x08
8436 #define    PCI_CAP_VENDSPEC        0x09
8437 #define    PCI_CAP_DEBUGPORT       0x0a
8438 #define    PCI_CAP_CPCI_RSRCCTL    0x0b
8439 #define    PCI_CAP_HOTPLUG         0x0c
8440 #define    PCI_CAP_AGP8            0x0e
8441 #define    PCI_CAP_SECURE          0x0f
8442 #define    PCI_CAP_PCIEXPRESS      0x10
8443 #define    PCI_CAP_MSIX            0x11
8444
8445 /*
8446  * Vital Product Data; access via capability pointer (PCI rev 2.2).
8447  */
8448 #define    PCI_VPD_ADDRESS_MASK    0x7fff
8449 #define    PCI_VPD_ADDRESS_SHIFT   16
```



```
8450 #define    PCI_VPD_ADDRESS(ofs)    \
8451    (((ofs) & PCI_VPD_ADDRESS_MASK) << PCI_VPD_ADDRESS_SHIFT)
8452 #define    PCI_VPD_DATAREG(ofs)    ((ofs) + 4)
8453 #define    PCI_VPD_OPFLAG          0x80000000
8454
8455 /*
8456  * Power Management Capability; access via capability pointer.
8457  */
8458
8459 /* Power Management Capability Register */
8460 #define PCI_PMCR           0x02
8461 #define PCI_PMCR_D1SUPP            0x0200
8462 #define PCI_PMCR_D2SUPP            0x0400
8463 /* Power Management Control Status Register */
8464 #define PCI_PMCSR          0x04
8465 #define PCI_PMCSR_STATE_MASK       0x03
8466 #define PCI_PMCSR_STATE_D0      0x00
8467 #define PCI_PMCSR_STATE_D1      0x01
8468 #define PCI_PMCSR_STATE_D2      0x02
8469 #define PCI_PMCSR_STATE_D3      0x03
8470
8471 /*
8472  * PCI-X capability.
8473  */
8474
8475 /*
8476  * Command. 16 bits at offset 2 (e.g. upper 16 bits of the first 32-bit
8477  * word at the capability; the lower 16 bits are the capability ID and
8478  * next capability pointer).
8479  *
8480  * Since we always read PCI config space in 32-bit words, we define these
8481  * as 32-bit values, offset and shifted appropriately.  Make sure you perfor
8482  * the appropriate R/M/W cycles!
8483  */
8484 #define PCI_PCIX_CMD                       0x00
8485 #define PCI_PCIX_CMD_PERR_RECOVER  0x00010000
8486 #define PCI_PCIX_CMD_RELAXED_ORDER 0x00020000
8487 #define PCI_PCIX_CMD_BYTECNT_MASK  0x000c0000
8488 #define    PCI_PCIX_CMD_BYTECNT_SHIFT      18
8489 #define            PCI_PCIX_CMD_BCNT_512           0x00000000
8490 #define            PCI_PCIX_CMD_BCNT_1024          0x00040000
8491 #define            PCI_PCIX_CMD_BCNT_2048          0x00080000
8492 #define            PCI_PCIX_CMD_BCNT_4096          0x000c0000
8493 #define PCI_PCIX_CMD_SPLTRANS_MASK 0x00700000
8494 #define            PCI_PCIX_CMD_SPLTRANS_1         0x00000000
8495 #define            PCI_PCIX_CMD_SPLTRANS_2         0x00100000
8496 #define            PCI_PCIX_CMD_SPLTRANS_3         0x00200000
8497 #define            PCI_PCIX_CMD_SPLTRANS_4         0x00300000
8498 #define            PCI_PCIX_CMD_SPLTRANS_8         0x00400000
8499 #define            PCI_PCIX_CMD_SPLTRANS_12        0x00500000
```

```
8500 #define            PCI_PCIX_CMD_SPLTRANS_16        0x00600000
8501 #define            PCI_PCIX_CMD_SPLTRANS_32        0x00700000
8502
8503 /*
8504  * Status. 32 bits at offset 4.
8505  */
8506 #define PCI_PCIX_STATUS                    0x04
8507 #define PCI_PCIX_STATUS_FN_MASK            0x00000007
8508 #define PCI_PCIX_STATUS_DEV_MASK   0x000000f8
8509 #define PCI_PCIX_STATUS_BUS_MASK   0x0000ff00
8510 #define PCI_PCIX_STATUS_64BIT              0x00010000
8511 #define PCI_PCIX_STATUS_133                0x00020000
8512 #define PCI_PCIX_STATUS_SPLDISC            0x00040000
8513 #define PCI_PCIX_STATUS_SPLUNEX            0x00080000
8514 #define PCI_PCIX_STATUS_DEVCPLX            0x00100000
8515 #define PCI_PCIX_STATUS_MAXB_MASK  0x00600000
8516 #define    PCI_PCIX_STATUS_MAXB_SHIFT      21
8517 #define            PCI_PCIX_STATUS_MAXB_512        0x00000000
8518 #define            PCI_PCIX_STATUS_MAXB_1024       0x00200000
8519 #define            PCI_PCIX_STATUS_MAXB_2048       0x00400000
8520 #define            PCI_PCIX_STATUS_MAXB_4096       0x00600000
8521 #define PCI_PCIX_STATUS_MAXST_MASK 0x03800000
8522 #define            PCI_PCIX_STATUS_MAXST_1         0x00000000
8523 #define            PCI_PCIX_STATUS_MAXST_2         0x00800000
8524 #define            PCI_PCIX_STATUS_MAXST_3         0x01000000
8525 #define            PCI_PCIX_STATUS_MAXST_4         0x01800000
8526 #define            PCI_PCIX_STATUS_MAXST_8         0x02000000
8527 #define            PCI_PCIX_STATUS_MAXST_12        0x02800000
8528 #define            PCI_PCIX_STATUS_MAXST_16        0x03000000
8529 #define            PCI_PCIX_STATUS_MAXST_32        0x03800000
8530 #define PCI_PCIX_STATUS_MAXRS_MASK 0x1c000000
8531 #define            PCI_PCIX_STATUS_MAXRS_1K        0x00000000
8532 #define            PCI_PCIX_STATUS_MAXRS_2K        0x04000000
8533 #define            PCI_PCIX_STATUS_MAXRS_4K        0x08000000
8534 #define            PCI_PCIX_STATUS_MAXRS_8K        0x0c000000
8535 #define            PCI_PCIX_STATUS_MAXRS_16K       0x10000000
8536 #define            PCI_PCIX_STATUS_MAXRS_32K       0x14000000
8537 #define            PCI_PCIX_STATUS_MAXRS_64K       0x18000000
8538 #define            PCI_PCIX_STATUS_MAXRS_128K      0x1c000000
8539 #define PCI_PCIX_STATUS_SCERR                      0x20000000
8540
8541
8542 /*
8543  * Interrupt Configuration Register; contains interrupt pin and line.
8544  */
8545 #define    PCI_INTERRUPT_REG               0x3c
8546
8547
8548
8549
```



```
8550 typedef uint8_t pci_intr_latency_t;
8551 typedef uint8_t pci_intr_grant_t;
8552 typedef uint8_t pci_intr_pin_t;
8553 typedef uint8_t pci_intr_line_t;
8554
8555 #define PCI_MAX_LAT_SHIFT                  24
8556 #define    PCI_MAX_LAT_MASK                        0xff
8557 #define    PCI_MAX_LAT(icr) \
8558        (((icr) >> PCI_MAX_LAT_SHIFT) & PCI_MAX_LAT_MASK)
8559
8560 #define PCI_MIN_GNT_SHIFT                  16
8561 #define    PCI_MIN_GNT_MASK                        0xff
8562 #define    PCI_MIN_GNT(icr) \
8563        (((icr) >> PCI_MIN_GNT_SHIFT) & PCI_MIN_GNT_MASK)
8564
8565 #define    PCI_INTERRUPT_GRANT_SHIFT               24
8566 #define    PCI_INTERRUPT_GRANT_MASK                0xff
8567 #define    PCI_INTERRUPT_GRANT(icr) \
8568        (((icr) >> PCI_INTERRUPT_GRANT_SHIFT) & PCI_INTERRUPT_GRANT_MASK)
8569
8570 #define    PCI_INTERRUPT_LATENCY_SHIFT             16
8571 #define    PCI_INTERRUPT_LATENCY_MASK              0xff
8572 #define    PCI_INTERRUPT_LATENCY(icr) \
8573        (((icr) >> PCI_INTERRUPT_LATENCY_SHIFT) & PCI_INTERRUPT_LATENCY_MASK)
8574
8575 #define    PCI_INTERRUPT_PIN_SHIFT                 8
8576 #define    PCI_INTERRUPT_PIN_MASK                  0xff
8577 #define    PCI_INTERRUPT_PIN(icr) \
8578        (((icr) >> PCI_INTERRUPT_PIN_SHIFT) & PCI_INTERRUPT_PIN_MASK)
8579
8580 #define    PCI_INTERRUPT_LINE_SHIFT                0
8581 #define    PCI_INTERRUPT_LINE_MASK                 0xff
8582 #define    PCI_INTERRUPT_LINE(icr) \
8583        (((icr) >> PCI_INTERRUPT_LINE_SHIFT) & PCI_INTERRUPT_LINE_MASK)
8584
8585 #define PCI_INTERRUPT_CODE(lat,gnt,pin,line)               \
8586      ((((lat)&PCI_INTERRUPT_LATENCY_MASK)<<PCI_INTERRUPT_LATENCY_SHIFT)| \
8587       (((gnt)&PCI_INTERRUPT_GRANT_MASK)  <<PCI_INTERRUPT_GRANT_SHIFT)  | \
8588       (((pin)&PCI_INTERRUPT_PIN_MASK)    <<PCI_INTERRUPT_PIN_SHIFT)    | \
8589       (((line)&PCI_INTERRUPT_LINE_MASK)  <<PCI_INTERRUPT_LINE_SHIFT))
8590
8591 #define    PCI_INTERRUPT_PIN_NONE                  0x00
8592 #define    PCI_INTERRUPT_PIN_A                     0x01
8593 #define    PCI_INTERRUPT_PIN_B                     0x02
8594 #define    PCI_INTERRUPT_PIN_C                     0x03
8595 #define    PCI_INTERRUPT_PIN_D                     0x04
8596 #define    PCI_INTERRUPT_PIN_MAX                   0x04
8597
8598
8599
```

```
8600 /* Header Type 1 (Bridge) configuration registers */
8601 #define PCI_BRIDGE_BUS_REG         0x18
8602 #define   PCI_BRIDGE_BUS_PRIMARY_SHIFT             0
8603 #define   PCI_BRIDGE_BUS_SECONDARY_SHIFT   8
8604 #define   PCI_BRIDGE_BUS_SUBORDINATE_SHIFT 16
8605
8606 #define PCI_BRIDGE_STATIO_REG              0x1C
8607 #define      PCI_BRIDGE_STATIO_IOBASE_SHIFT       0
8608 #define      PCI_BRIDGE_STATIO_IOLIMIT_SHIFT      8
8609 #define      PCI_BRIDGE_STATIO_STATUS_SHIFT       16
8610 #define      PCI_BRIDGE_STATIO_IOBASE_MASK        0xf0
8611 #define      PCI_BRIDGE_STATIO_IOLIMIT_MASK       0xf0
8612 #define      PCI_BRIDGE_STATIO_STATUS_MASK        0xffff
8613 #define      PCI_BRIDGE_IO_32BITS(reg)            (((reg) & 0xf) == 1)
8614
8615 #define PCI_BRIDGE_MEMORY_REG              0x20
8616 #define      PCI_BRIDGE_MEMORY_BASE_SHIFT         4
8617 #define      PCI_BRIDGE_MEMORY_LIMIT_SHIFT        20
8618 #define      PCI_BRIDGE_MEMORY_BASE_MASK          0xffff
8619 #define      PCI_BRIDGE_MEMORY_LIMIT_MASK         0xffff
8620
8621 #define PCI_BRIDGE_PREFETCHMEM_REG 0x24
8622 #define      PCI_BRIDGE_PREFETCHMEM_BASE_SHIFT    4
8623 #define      PCI_BRIDGE_PREFETCHMEM_LIMIT_SHIFT   20
8624 #define      PCI_BRIDGE_PREFETCHMEM_BASE_MASK     0xffff
8625 #define      PCI_BRIDGE_PREFETCHMEM_LIMIT_MASK    0xffff
8626 #define      PCI_BRIDGE_PREFETCHMEM_64BITS(reg)   ((reg) & 0xf)
8627
8628 #define PCI_BRIDGE_PREFETCHBASE32_REG      0x28
8629 #define PCI_BRIDGE_PREFETCHLIMIT32_REG     0x2C
8630
8631 #define PCI_BRIDGE_IOHIGH_REG              0x30
8632 #define      PCI_BRIDGE_IOHIGH_BASE_SHIFT         0
8633 #define      PCI_BRIDGE_IOHIGH_LIMIT_SHIFT        16
8634 #define      PCI_BRIDGE_IOHIGH_BASE_MASK          0xffff
8635 #define      PCI_BRIDGE_IOHIGH_LIMIT_MASK         0xffff
8636
8637 #define PCI_BRIDGE_CONTROL_REG             0x3C
8638 #define      PCI_BRIDGE_CONTROL_SHIFT             16
8639 #define      PCI_BRIDGE_CONTROL_MASK              0xffff
8640 #define   PCI_BRIDGE_CONTROL_PERE          (1 <<  0)
8641 #define   PCI_BRIDGE_CONTROL_SERR          (1 <<  1)
8642 #define   PCI_BRIDGE_CONTROL_ISA           (1 <<  2)
8643 #define   PCI_BRIDGE_CONTROL_VGA           (1 <<  3)
8644 /* Reserved                                 (1 <<  4) */
8645 #define   PCI_BRIDGE_CONTROL_MABRT         (1 <<  5)
8646 #define   PCI_BRIDGE_CONTROL_SECBR         (1 <<  6)
8647 #define   PCI_BRIDGE_CONTROL_SECFASTB2B            (1 <<  7)
8648 #define   PCI_BRIDGE_CONTROL_PRI_DISC_TIMER        (1 <<  8)
8649 #define   PCI_BRIDGE_CONTROL_SEC_DISC_TIMER        (1 <<  9)
```



```
8650 #define   PCI_BRIDGE_CONTROL_DISC_TIMER_STAT       (1 << 10)
8651 #define   PCI_BRIDGE_CONTROL_DISC_TIMER_SERR       (1 << 11)
8652 /* Reserved                                 (1 << 12) - (1 << 15) */
8653
8654 /*
8655  * Vital Product Data resource tags.
8656  */
8657 struct pci_vpd_smallres {
8658    uint8_t         vpdres_byte0;           /* length of data + tag */
8659    /* Actual data. */
8660 } __attribute__((__packed__));
8661
8662 struct pci_vpd_largeres {
8663    uint8_t         vpdres_byte0;
8664    uint8_t         vpdres_len_lsb;         /* length of data only */
8665    uint8_t         vpdres_len_msb;
8666    /* Actual data. */
8667 } __attribute__((__packed__));
8668
8669 #define    PCI_VPDRES_ISLARGE(x)                   ((x) & 0x80)
8670
8671 #define    PCI_VPDRES_SMALL_LENGTH(x)              ((x) & 0x7)
8672 #define    PCI_VPDRES_SMALL_NAME(x)                (((x) >> 3) & 0xf)
8673
8674 #define    PCI_VPDRES_LARGE_NAME(x)                ((x) & 0x7f)
8675
8676 #define    PCI_VPDRES_TYPE_COMPATIBLE_DEVICE_ID    0x3     /* small */
8677 #define    PCI_VPDRES_TYPE_VENDOR_DEFINED          0xe     /* small */
8678 #define    PCI_VPDRES_TYPE_END_TAG                 0xf     /* small */
8679
8680 #define    PCI_VPDRES_TYPE_IDENTIFIER_STRING       0x02    /* large */
8681 #define    PCI_VPDRES_TYPE_VPD                     0x10    /* large */
8682
8683 struct pci_vpd {
8684    uint8_t         vpd_key0;
8685    uint8_t         vpd_key1;
8686    uint8_t         vpd_len;                /* length of data only */
8687    /* Actual data. */
8688 } __attribute__((__packed__));
8689
8690 /*
8691  * Recommended VPD fields:
8692  *
8693  * PN              Part number of assembly
8694  * FN              FRU part number
8695  * EC              EC level of assembly
8696  * MN              Manufacture ID
8697  * SN              Serial Number
8698  *
8699  * Conditionally recommended VPD fields:
```

```
 *
 * LI              Load ID
 * RL              ROM Level
 * RM              Alterable ROM Level
 * NA              Network Address
 * DD              Device Driver Level
 * DG              Diagnostic Level
 * LL              Loadable Microcode Level
 * VI              Vendor ID/Device ID
 * FU              Function Number
 * SI              Subsystem Vendor ID/Subsystem ID
 *
 * Additional VPD fields:
 *
 * Z0-ZZ           User/Product Specific
 */

/*
 * Threshold below which 32bit PCI DMA needs bouncing.
 */
#define PCI32_DMA_BOUNCE_THRESHOLD 0x100000000ULL

#endif /* _DEV_PCI_PCIREG_H_ */
```



```
#include "types.h"
#include "x86.h"
#include "defs.h"
#include "pci.h"
#include "pcireg.h"
#include "assert.h"
#include "e100.h"

// Flag to do "lspci" at bootup
static int pci_show_devs = 1;
static int pci_show_addrs = 1;

// PCI "configuration mechanism one"
static uint32_t pci_conf1_addr_ioport = 0x0cf8;
static uint32_t pci_conf1_data_ioport = 0x0cfc;

// Forward declarations
static int pci_bridge_attach(struct pci_func *pcif);
static int pci_bridge_pci_attach(struct pci_func *pcif);
static int pci_display_attach(struct pci_func *pcif);
static int pci_net_ether_attach(struct pci_func *pcif);

// PCI driver table
struct pci_driver {
    uint32_t key1, key2;
    int (*attachfn) (struct pci_func *pcif);
};

#define PCI_SUBCLASS_ANY 0xffff

struct pci_driver pci_attach_class[] = {
    { PCI_CLASS_BRIDGE, PCI_SUBCLASS_ANY, &pci_bridge_attach },
    { PCI_CLASS_BRIDGE, PCI_SUBCLASS_BRIDGE_PCI, &pci_bridge_pci_attach },
    { PCI_CLASS_DISPLAY, PCI_SUBCLASS_ANY, &pci_display_attach },
    { PCI_CLASS_NETWORK, PCI_SUBCLASS_NETWORK_ETHERNET,
        &pci_net_ether_attach },
    { 0, 0, 0 },
};

struct pci_driver pci_attach_vendor[] = {
    { PCI_VENDOR_INTEL, PCI_PRODUCT_E100, &ether_e100_attach },
    { 0, 0, 0 },
};
```

```
8800 static void
8801 pci_conf1_set_addr(uint32_t bus,
8802               uint32_t dev,
8803               uint32_t func,
8804               uint32_t offset)
8805 {
8806     assert(bus < 256);
8807     assert(dev < 32);
8808     assert(func < 8);
8809     assert(offset < 256);
8810     assert((offset & 0x3) == 0);
8811
8812     uint32_t v = (1 << 31) |               // config-space
8813             (bus << 16) | (dev << 11) | (func << 8) | (offset);
8814     outl(pci_conf1_addr_ioport, v);
8815 }
8816
8817 static uint32_t
8818 pci_conf_read(struct pci_func *f, uint32_t off)
8819 {
8820     pci_conf1_set_addr(f->bus->busno, f->dev, f->func, off);
8821     return inl(pci_conf1_data_ioport);
8822 }
8823
8824 static void
8825 pci_conf_write(struct pci_func *f, uint32_t off, uint32_t v)
8826 {
8827     pci_conf1_set_addr(f->bus->busno, f->dev, f->func, off);
8828     outl(pci_conf1_data_ioport, v);
8829 }
8830
8831 static int __attribute__((warn_unused_result))
8832 pci_attach_match(uint32_t key1, uint32_t key2,
8833             struct pci_driver *list, struct pci_func *pcif)
8834 {
8835     uint32_t i;
8836
8837     for (i = 0; list[i].attachfn; i++) {
8838    if (list[i].key1 == key1 )
8839         if ((list[i].key2 == key2) || (list[i].key2 == 0xffff)) {
8840            int r = list[i].attachfn(pcif);
8841            if (r > 0)
8842                    return r;
8843            if (r < 0)
8844            ///cprintf("pci_attach_match: attaching %x.%x (%p): %s\n",
8845            //      key1, key2, list[i].attachfn, e2s(r));
8846                    cprintf("pci_attach_match: attaching %x.%x (%p): SOME ERR
8847                            key1, key2, list[i].attachfn);
8848        }
8849     }
```



```
8850     return 0;
8851 }
8852
8853 static int
8854 pci_attach(struct pci_func *f)
8855 {
8856     return
8857    pci_attach_match(PCI_CLASS(f->dev_class), PCI_SUBCLASS(f->dev_class),
8858                     &pci_attach_class[0], f) ||
8859    pci_attach_match(PCI_VENDOR(f->dev_id), PCI_PRODUCT(f->dev_id),
8860                     &pci_attach_vendor[0], f);
8861 }
8862
8863 static int
8864 pci_scan_bus(struct pci_bus *bus)
8865 {
8866     int totaldev = 0;
8867     struct pci_func df;
8868     memset(&df, 0, sizeof(df));
8869     df.bus = bus;
8870
8871     for (df.dev = 0; df.dev < 32; df.dev++) {
8872    uint32_t bhlc = pci_conf_read(&df, PCI_BHLC_REG);
8873    if (PCI_HDRTYPE_TYPE(bhlc) > 1)     // Unsupported or no device
8874        continue;
8875
8876    totaldev++;
8877
8878    struct pci_func f = df;
8879    for (f.func = 0; f.func < (PCI_HDRTYPE_MULTIFN(bhlc) ? 8 : 1);
8880                     f.func++) {
8881        struct pci_func af = f;
8882
8883        af.dev_id = pci_conf_read(&f, PCI_ID_REG);
8884        if (PCI_VENDOR(af.dev_id) == 0xffff)
8885            continue;
8886
8887        uint32_t intr = pci_conf_read(&af, PCI_INTERRUPT_REG);
8888        af.irq_line = PCI_INTERRUPT_LINE(intr);
8889
8890        af.dev_class = pci_conf_read(&af, PCI_CLASS_REG);
8891        if (pci_show_devs)
8892            cprintf("PCI: %02x:%02x.%d: %04x:%04x: class %x.%x irq %d\n",
8893                    af.bus->busno, af.dev, af.func,
8894                    PCI_VENDOR(af.dev_id), PCI_PRODUCT(af.dev_id),
8895                    PCI_CLASS(af.dev_class), PCI_SUBCLASS(af.dev_class),
8896                    af.irq_line);
8897
8898
8899
```

```
8900        pci_attach(&af);
8901    }
8902     }
8903
8904     return totaldev;
8905 }
8906
8907 static int
8908 pci_net_ether_attach(struct pci_func *pcif)
8909 {
8910     cprintf("PCI: %02x:%02x.%d: Network (ethernet) %04x:%04x\n",
8911            pcif->bus->busno, pcif->dev, pcif->func,
8912         PCI_VENDOR(pcif->dev_id), PCI_PRODUCT(pcif->dev_id));
8913     return 0;
8914 }
8915
8916 static int
8917 pci_bridge_attach(struct pci_func *pcif)
8918 {
8919     cprintf("PCI: %02x:%02x.%d: Bridge %04x:%04x\n",
8920            pcif->bus->busno, pcif->dev, pcif->func,
8921         PCI_VENDOR(pcif->dev_id), PCI_PRODUCT(pcif->dev_id));
8922     return 0;
8923 }
8924
8925 static int
8926 pci_bridge_pci_attach(struct pci_func *pcif)
8927 {
8928     uint32_t ioreg  = pci_conf_read(pcif, PCI_BRIDGE_STATIO_REG);
8929     uint32_t busreg = pci_conf_read(pcif, PCI_BRIDGE_BUS_REG);
8930
8931     if (PCI_BRIDGE_IO_32BITS(ioreg)) {
8932    cprintf("PCI: %02x:%02x.%d: 32-bit bridge IO not supported.\n",
8933            pcif->bus->busno, pcif->dev, pcif->func);
8934    return 0;
8935     }
8936
8937     struct pci_bus nbus;
8938     memset(&nbus, 0, sizeof(nbus));
8939     nbus.parent_bridge = pcif;
8940     nbus.busno = (busreg >> PCI_BRIDGE_BUS_SECONDARY_SHIFT) & 0xff;
8941
8942     if (pci_show_devs)
8943    cprintf("PCI: %02x:%02x.%d: bridge to PCI bus %d--%d\n",
8944            pcif->bus->busno, pcif->dev, pcif->func,
8945            nbus.busno,
8946            (busreg >> PCI_BRIDGE_BUS_SUBORDINATE_SHIFT) & 0xff);
8947
8948
8949
```



```
8950     pci_scan_bus(&nbus);
8951     return 1;
8952 }
8953
8954 static int
8955 pci_display_attach(struct pci_func *pcif)
8956 {
8957     cprintf("PCI: %02x:%02x.%d: Display %04x:%04x\n",
8958            pcif->bus->busno, pcif->dev, pcif->func,
8959          PCI_VENDOR(pcif->dev_id), PCI_PRODUCT(pcif->dev_id));
8960     return 0;
8961 }
8962
8963 // External PCI subsystem interface
8964
8965 void
8966 pci_func_enable(struct pci_func *f)
8967 {
8968     pci_conf_write(f, PCI_COMMAND_STATUS_REG,
8969                PCI_COMMAND_IO_ENABLE |
8970                PCI_COMMAND_MEM_ENABLE |
8971                PCI_COMMAND_MASTER_ENABLE);
8972
8973     uint32_t bar_width;
8974     uint32_t bar;
8975     for (bar = PCI_MAPREG_START; bar < PCI_MAPREG_END;
8976     bar += bar_width)
8977     {
8978    uint32_t oldv = pci_conf_read(f, bar);
8979
8980    bar_width = 4;
8981    pci_conf_write(f, bar, 0xffffffff);
8982    uint32_t rv = pci_conf_read(f, bar);
8983
8984    if (rv == 0)
8985        continue;
8986
8987    int regnum = PCI_MAPREG_NUM(bar);
8988    uint32_t base, size;
8989    if (PCI_MAPREG_TYPE(rv) == PCI_MAPREG_TYPE_MEM) {
8990        if (PCI_MAPREG_MEM_TYPE(rv) == PCI_MAPREG_MEM_TYPE_64BIT)
8991            bar_width = 8;
8992
8993        size = PCI_MAPREG_MEM_SIZE(rv);
8994        base = PCI_MAPREG_MEM_ADDR(oldv);
8995        if (pci_show_addrs)
8996            cprintf("  mem region %d: %d bytes at 0x%x\n",
8997                    regnum, size, base);
8998    } else {
8999        size = PCI_MAPREG_IO_SIZE(rv);
```

```
9000        base = PCI_MAPREG_IO_ADDR(oldv);
9001        if (pci_show_addrs)
9002            cprintf("  io region %d: %d bytes at 0x%x\n",
9003                    regnum, size, base);
9004    }
9005
9006    pci_conf_write(f, bar, oldv);
9007    f->reg_base[regnum] = base;
9008    f->reg_size[regnum] = size;
9009
9010    cprintf("  -> reg_base[%d] = %08x\n", regnum, base);
9011    cprintf("  -> reg_size[%d] = %08x\n", regnum, size);
9012
9013    if (size && !base)
9014        cprintf("PCI device %02x:%02x.%d (%04x:%04x) may be misconfigured: "
9015                "region %d: base 0x%x, size %d\n",
9016                f->bus->busno, f->dev, f->func,
9017                PCI_VENDOR(f->dev_id), PCI_PRODUCT(f->dev_id),
9018                regnum, base, size);
9019     }
9020 }
9021
9022 int ether_receive(void *buffer, int len)
9023 {
9024     if (len == 0)
9025         return 0;
9026     int ret;
9027     ret = e100_receive(buffer, len);
9028     if (ret > 0)
9029         return ret;
9030     return -2;
9031 }
9032
9033 int ether_send(void *buffer, int len)
9034 {
9035     if (len == 0)
9036         return 0;
9037     int ret;
9038     ret = e100_send(buffer, len);
9039     if (ret > 0)
9040         return ret;
9041     return -2;
9042 }
9043
9044
9045
9046
9047
9048
9049
```



```
9050 int
9051 pci_init(void)
9052 {
9053     static struct pci_bus root_bus;
9054     memset(&root_bus, 0, sizeof(root_bus));
9055
9056     return pci_scan_bus(&root_bus);
9057 }
9058
9059
9060
9061
9062
9063
9064
9065
9066
9067
9068
9069
9070
9071
9072
9073
9074
9075
9076
9077
9078
9079
9080
9081
9082
9083
9084
9085
9086
9087
9088
9089
9090
9091
9092
9093
9094
9095
9096
9097
9098
9099
```

```
#ifndef XV6_E100_H_
#define XV6_E100_H_
#include "pci.h"

int ether_e100_attach(struct pci_func *pcif);
int e100_send(void *buffer, uint32_t len);
int e100_receive(void *buffer, uint32_t len);

#define PCI_VENDOR_INTEL    0x8086
#define PCI_PRODUCT_E100    0x1209

#define E100_MAX_DEVS       10

#define E100_IOPORT_SIZE    64
#define E100_CU_RING_SIZE   32
#define E100_RU_RING_SIZE   32
#define E100_CB_SIZE       2048
#define E100_RFD_SIZE      2048

// Control / Status Register
#define SCB_STATUS          0x0 // Status
#define SCB_COMMAND         0x2 // Command
#define SCB_GENPTR          0x4 // General Pointer
#define SCB_PORT            0x8 // PORT
#define SCB_EEPROM_CTL      0xE // EEPROM Control
#define SCB_MDI_CTL         0x10    // MDI Control
#define SCB_RX_COUNT        0x14    // RX DMA Byte Count
#define SCB_FLOW_CTL        0x19    // Flow Control
#define SCB_PMDR            0x21    // PMDR
#define SCB_GEN_CTL         0x1C    // General Control
#define SCB_GEN_STATUS      0x1D    // General Status
#define SCB_FUNC_EVT        0x30    // Function Event
#define SCB_FUN_MASK        0x34    // Function Event Mask
#define SCB_FUNC_STATE      0x38    // Function Present State
#define SCB_FORCE_EVT       0x3C    // Force Event

// CU Command
#define CUC_NOP             0x0
#define CUC_START           0x1
#define CUC_RESUME          0x2
#define CUC_LOAD_DC_ADDR    0x4
#define CUC_DUMP            0x5
#define CUC_LOAD_BASE       0x6
#define CUC_DUMP_RESET      0x7
#define CUC_STAT_RESUME     0xA
```



```
// RU Command
#define RUC_NOP             0x0
#define RUC_START           0x1
#define RUC_RESUME          0x2
#define RUC_RCV_DMA         0x3
#define RUC_ABORT           0x4
#define RUC_LOAD_HDS        0x5
#define RUC_LOAD_BASE       0x6

// PORT Function
#define PORT_SOFT_RESET     0
#define PORT_SELF_TEST      1
#define PORT_SElECT_TEST    2
#define PORT_DUMP           3
#define PORT_DUMP_WAKE      7

// Operation Codes
#define OP_NOP              0
#define OP_ADDR_SETUP       1
#define OP_CONFIG           2
#define OP_MUL_ADDR_SETUP   3
#define OP_TRANSMIT         4
#define OP_LOAD_MC          5
#define OP_DUMP             6
#define OP_DIAG             7

// CU Status
#define CUS_IDLE            0
#define CUS_SUSPENDED       1
#define CUS_LPQ             2
#define CUS_HQP             3

struct scb_status_word {
    char zero : 2;
    char rus  : 4;
    char cus  : 2;
    // STAT / ACK
    char fcp  : 1;
    char rsv  : 1;
    char swi  : 1;
    char mdi  : 1;
    char rnr  : 1;
    char cna  : 1;
    char fr   : 1;
    char cx   : 1;
};
```

```
9200 typedef union scb_status_word_tag {
9201     uint16_t word;
9202     struct scb_status_word status;
9203 } scb_status_word;
9204
9205 struct scb_command_word {
9206     char ru_command : 3;
9207     char res : 1;
9208     char cu_command : 4;
9209     // Interrupt Masks
9210     char m : 1;
9211     char si : 1;
9212     char fcp_mask : 1;
9213     char er_mask : 1;
9214     char rnr_mask : 1;
9215     char cna_mask : 1;
9216     char fr_mask : 1;
9217     char cx_mask : 1;
9218 };
9219
9220 typedef union scb_command_word_tag {
9221     uint16_t word;
9222     struct scb_command_word cmd;
9223 } scb_command_word;
9224
9225 typedef struct command_block_tag {
9226     volatile union {
9227         uint16_t status_word;
9228         struct {
9229             short stat :    13;
9230             char ok :       1;    // No Error
9231             char x :        1;
9232             char c :        1;     // Completed
9233         };
9234     };
9235     union {
9236         uint16_t cmd_word;
9237         struct {
9238             char cmd :  3;
9239             short res : 10;
9240             char i :    1;     // Interrupt after finish
9241             char s :    1;     // Suspend after complete
9242             char el :   1;    // Last one
9243         };
9244     };
9245     struct command_block_tag * link;
9246 } command_block;
9247
9248
9249
```



```
9250 typedef struct op_transmit_cmd_cmd_tag {
9251     union {
9252         uint16_t word;
9253         struct {
9254             char cmd    : 3;
9255             char sf     : 1;
9256             char nc     : 1;
9257             char res    : 3;
9258             char cid    : 5;
9259             char i      : 1;
9260             char s      : 1;
9261             char el     : 1;
9262         };
9263     };
9264 } op_transmit_cmd_cmd;
9265
9266 typedef struct op_transmit_cmd {
9267     command_block base_cmd;
9268     uint32_t tbd_addr;
9269     struct {
9270         int byte_count :14;
9271         char res :      1;
9272         char eof :      1;
9273         uint8_t trans_thres;
9274         uint8_t tbd_num;
9275     };
9276 } op_transmit_cmd;
9277
9278 typedef struct rf_desc {
9279     volatile command_block head;
9280     uint32_t res;
9281     volatile struct {
9282         uint16_t count :    14;
9283         char f :            1;
9284         char eof :          1;
9285         uint16_t size :     14;
9286         char res2 :         2;
9287     };
9288 } rf_desc;
9289
9290 typedef struct rf_status {
9291     char tco :      1;
9292     char ia :       1;
9293     char nomatch :  1;
9294     char res :      1;
9295     char rcv_err :  1;
9296     char type :     1;
9297     char res2 :     1;
9298     char tooshort : 1;
9299     char dma_err :  1;
```

```
9300     char no_buf :   1;
9301     char align_err :1;
9302     char crc_err :  1;
9303     char res3 :     1;
9304 } rf_status;
9305
9306 #define TBD_DATA_LIMIT  1600
9307
9308
9309 #endif     // XV6_E100_H_
9310
9311
9312
9313
9314
9315
9316
9317
9318
9319
9320
9321
9322
9323
9324
9325
9326
9327
9328
9329
9330
9331
9332
9333
9334
9335
9336
9337
9338
9339
9340
9341
9342
9343
9344
9345
9346
9347
9348
9349
```



```
9350 #include "types.h"
9351 #include "x86.h"
9352 #include "pci.h"
9353 #include "pcireg.h"
9354 #include "defs.h"
9355 #include "e100.h"
9356 #include "proc.h"
9357 #include "picirq.h"
9358 #include "traps.h"
9359 #include "lwip/include/ipv4/lwip/ip_addr.h"
9360 #include "lwip/include/ipv4/lwip/ip.h"
9361 #include "lwip/include/lwip/netif.h"
9362 #include "lwip/include/netif/ethernetif.h"
9363 #include "thread.h"
9364
9365 static int dev_count = 0;
9366
9367 typedef struct e100_devinfo {
9368     int irq;
9369     uint32_t regbase[6], regsize[6];
9370     uint32_t iobase;
9371     struct netif netif;
9372     struct spinlock culock;
9373     uint32_t cubase;
9374     uint32_t cusize;
9375 //    uint32_t cuhead;
9376 //    uint32_t cuend;
9377     uint32_t cucount;
9378     int cufirst;
9379     int cuidle;
9380     command_block *cu_last_pkt;
9381     command_block *cu_next_pkt;
9382     struct spinlock rulock;
9383     uint32_t rubase;
9384     uint32_t rusize;
9385     uint32_t rucount;
9386     rf_desc *ru_first;
9387     rf_desc *ru_last;
9388     rf_desc *ru_prev;
9389     int ru_full;
9390     struct spinlock rxlock;
9391 } e100_dev;
9392
9393 static e100_dev e100_devs[E100_MAX_DEVS];
9394 int e100_reset(e100_dev *dev);
9395 scb_status_word e100_print_state(e100_dev *dev);
9396 void e100_put_state(e100_dev *dev, scb_status_word stat);
9397 uint8_t e100_read_scb_command(e100_dev *dev);
9398 void e100_cu_command(e100_dev *dev, int command, void *cmd_addr);
9399 void e100_ru_command(e100_dev *dev, int command, void *cmd_addr);
```

```
9400 void e100_intr(struct trapframe *tf);
9401 void e100_ru_start(e100_dev *dev);
9402 void ring_printinfo(e100_dev *dev);
9403 void ring_init(e100_dev *dev);
9404 int e100_receive_dev(e100_dev *dev, void *buffer, int len);
9405 void e100_rx_thread(void *arg);
9406
9407 char *cu_state_name[];
9408 char *ru_state_name[];
9409
9410 int
9411 ether_e100_attach(struct pci_func *pcif)
9412 {
9413     cprintf("PCI: %02x:%02x.%d: "
9414             "Intel 82559ER Fast Ethernet PCI Controller %04x:%04x\n",
9415            pcif->bus->busno, pcif->dev, pcif->func,
9416         PCI_VENDOR(pcif->dev_id), PCI_PRODUCT(pcif->dev_id));
9417     dev_count++;
9418     int index = dev_count-1;
9419     e100_dev *dev = &e100_devs[index];
9420
9421     dev->cubase = (uint32_t)kalloc(E100_CU_RING_SIZE * PAGE); // 128K
9422     if (!dev->cubase)
9423     {
9424         cprintf("   Failed to allocate CU ring\n");
9425         dev_count--;
9426         return -1;
9427     }
9428     dev->rubase = (uint32_t)kalloc(E100_RU_RING_SIZE * PAGE); // 128K
9429     if (!dev->rubase)
9430     {
9431         cprintf("   Failed to allocate RU ring\n");
9432         dev_count--;
9433         return -1;
9434     }
9435     dev->cusize = E100_CU_RING_SIZE * PAGE;
9436     dev->rusize = E100_RU_RING_SIZE * PAGE;
9437     ring_init(dev);
9438 //    dev->cuhead = 0;
9439 //    dev->cuend = 0;
9440
9441     pci_func_enable(pcif);
9442     initlock(&dev->culock, "e100 CU lock");
9443     initlock(&dev->rulock, "e100 RU lock");
9444     initlock(&dev->rxlock, "e100 RX lock");
9445     dev->irq = pcif->irq_line;
9446     int i;
9447     for (i=0; i<6; i++)
9448     {
9449         dev->regbase[i] = pcif->reg_base[i];
```



```
9450         dev->regsize[i] = pcif->reg_size[i];
9451         if (dev->regsize[i] == E100_IOPORT_SIZE)
9452             dev->iobase = dev->regbase[i];
9453     }
9454     e100_reset(dev);
9455     dev->cufirst = 1;
9456     dev->cuidle = 1;
9457     e100_print_state(dev);
9458     struct ip_addr ipaddr;
9459     IP4_ADDR(&ipaddr, 192, 168, 1, 1);
9460     struct ip_addr netmask;
9461     IP4_ADDR(&netmask, 255, 255, 255, 0);
9462     struct ip_addr gw;
9463     IP4_ADDR(&gw, 192, 168, 1, 2);
9464     netif_add(&dev->netif, &ipaddr, &netmask, &gw, 0, ethernetif_init, ip_in
9465     reg_irq_handler(dev->irq, e100_intr);
9466     pic_enable(dev->irq);
9467     ioapic_enable(dev->irq, ncpu-1);
9468     kproc_start(e100_rx_thread, dev, 0, 0, "[e100 rx thread]");
9469     e100_ru_start(dev);
9470     return 1;
9471 }
9472
9473 static void
9474 delay(int n)
9475 {
9476     volatile int i;
9477     int j;
9478     for (j=0; j<n; j++)
9479     {
9480     for (i=0; i<1000; i++)
9481         ;
9482     }
9483 }
9484
9485 int
9486 e100_reset(e100_dev *dev)
9487 {
9488     outl(dev->iobase + SCB_PORT, PORT_SOFT_RESET);
9489     delay(10);
9490     scb_command_word cmd;
9491     cmd.word = 0;
9492     cmd.cmd.cu_command = CUC_DUMP_RESET;
9493     outw(dev->iobase + SCB_COMMAND, cmd.word);
9494     return 0;
9495 }
9496
9497
9498
9499
```

```
9500 scb_status_word
9501 e100_get_state(e100_dev *dev)
9502 {
9503     scb_status_word st;
9504     st.word = inw(dev->iobase + SCB_STATUS);
9505     return st;
9506 }
9507
9508 void
9509 e100_put_state(e100_dev *dev, scb_status_word stat)
9510 {
9511     outw(dev->iobase + SCB_STATUS, stat.word);
9512 }
9513
9514 scb_status_word
9515 e100_print_state(e100_dev *dev)
9516 {
9517     scb_status_word state = e100_get_state(dev);
9518     cprintf("   RU Status: %s\n", ru_state_name[state.status.rus]);
9519     cprintf("   CU Status: %s\n", cu_state_name[state.status.cus]);
9520
9521     struct scb_status_word stat = state.status;
9522     if (stat.cx) cprintf("   CU executed\n");
9523     if (stat.fr)
9524         cprintf("   RU received\n");
9525     if (stat.cna)
9526         cprintf("   CU state change\n");
9527     if (stat.rnr)
9528         cprintf("   RU not ready\n");
9529     if (stat.mdi)
9530         cprintf("   MDI operation completed\n");
9531     if (stat.swi)
9532         cprintf("   Software Interrupt\n");
9533     if (stat.fcp)
9534         cprintf("   Flow Control Pause\n");
9535
9536     return state;
9537 }
9538
9539 char *cu_state_name[4] = {
9540     "Idle",
9541     "Suspended",
9542     "LPQ Active",
9543     "HQP Active",
9544 };
9545
9546
9547
9548
9549
```



```
9550 char *ru_state_name[16] = {
9551     "Idle",
9552     "Suspended",
9553     "No resources",
9554     "Reserved",
9555     "Ready",
9556     "Reserved",
9557     "Reserved",
9558     "Reserved",
9559     "Reserved",
9560     "Reserved",
9561     "Reserved",
9562     "Reserved",
9563     "Reserved",
9564     "Reserved",
9565     "Reserved",
9566     "Reserved",
9567 };
9568
9569 uint8_t
9570 e100_read_scb_command(e100_dev *dev)
9571 {
9572     return inb(dev->iobase + SCB_COMMAND + 1);
9573 }
9574
9575 void
9576 e100_set_gp(e100_dev *dev, void *addr)
9577 {
9578     outl(dev->iobase + SCB_GENPTR, (uint32_t)addr);
9579 }
9580
9581 void
9582 e100_cu_command(e100_dev *dev, int command, void *cmd_addr)
9583 {
9584     scb_command_word cmd;
9585     cmd.word = 0;
9586     cmd.cmd.cu_command = command;
9587     e100_set_gp(dev, cmd_addr);
9588     outw(dev->iobase + SCB_COMMAND, cmd.word);
9589     while (e100_read_scb_command(dev)) ;
9590 }
9591
9592
9593
9594
9595
9596
9597
9598
9599
```

```
9600 void
9601 e100_ru_command(e100_dev *dev, int command, void *cmd_addr)
9602 {
9603     scb_command_word cmd;
9604     cmd.word = 0;
9605     cmd.cmd.ru_command = command;
9606     e100_set_gp(dev, cmd_addr);
9607     outw(dev->iobase + SCB_COMMAND, cmd.word);
9608     while (e100_read_scb_command(dev)) ;
9609 }
9610
9611 void
9612 e100_rx_thread(void *arg)
9613 {
9614     e100_dev *dev = arg;
9615     acquire(&dev->rxlock);
9616     while(1)
9617     {
9618         sleep(&dev->rxlock, &dev->rxlock);
9619         while (dev->ru_last->eof)
9620         {
9621             ethernetif_input(&dev->netif);
9622             dev->ru_last = (void*)dev->ru_last->head.link;
9623         }
9624     }
9625     release(&dev->rxlock);
9626 }
9627
9628 void
9629 e100_intr(struct trapframe *tf)
9630 {
9631 //    cprintf("e100_intr\n");
9632     int i;
9633     int index = -1;
9634     for (i=0; i< dev_count; i++)
9635     {
9636         if (tf->trapno == IRQ_OFFSET + e100_devs[i].irq)
9637         {
9638             index = i;
9639             break;
9640         }
9641     }
9642     if (index == -1)
9643     {
9644         cprintf("e100_intr: can't find corresponding device\n");
9645         return;
9646     }
9647     e100_dev *dev = &e100_devs[index];
9648 //    e100_print_state(dev);
9649     scb_status_word state = e100_get_state(dev);
```



```
9650     struct scb_status_word stat = state.status;
9651     state.word |= 0xff00;
9652     e100_put_state(dev, state);
9653
9654     if (stat.cx)
9655     {
9656 //        cprintf("   CU executed\n");
9657     }
9658     if (stat.fr)
9659     {
9660         if (!dev->ru_first)
9661         {
9662             dev->ru_first = dev->ru_last;
9663             wakeup(&dev->ru_first);
9664         }
9665         wakeup(&dev->rxlock);
9666 /*        while (dev->ru_last->eof)
9667         {
9668             ethernetif_input(&dev->netif);
9669             dev->ru_last = (void*)dev->ru_last->head.link;
9670         }*/
9671     }
9672     if (stat.cna)
9673     {
9674 /*        if (dev->cu_next_pkt != 0)
9675         {
9676             cprintf("   finished 0x%08x\n", dev->cu_next_pkt);
9677             if (dev->cu_next_pkt->link != 0)
9678             {
9679                 cprintf("   starting next packet: "
9680                         "0x%08x\n", dev->cu_next_pkt->link);
9681                 e100_cu_command(dev, CUC_START,
9682                         (void*)dev->cu_next_pkt->link);
9683                 dev->cuhead = dev->cu_next_pkt->link - dev->cubase;
9684                 dev->cu_next_pkt = (command_block *)dev->cu_next_pkt->link;
9685             }
9686             else
9687             {
9688                 dev->cuhead = dev->cuend;
9689                 dev->cu_last_pkt = 0;
9690                 dev->cu_next_pkt = 0;
9691             }
9692             ring_printinfo(dev);
9693         }*/
9694         if (dev->cu_last_pkt)
9695         {
9696             if (dev->cu_last_pkt->c)
9697             {
9698                 dev->cuidle = 1;
9699 //                dev->cuhead = dev->cuend;
```

```
9700                dev->cu_next_pkt = 0;
9701 //               dev->cu_last_pkt = 0;
9702            } else {
9703                dev->cuidle = 0;
9704                while (dev->cu_next_pkt->c)
9705                {
9706                    if (!dev->cu_next_pkt->ok)
9707                        cprintf("send error 0x%08x\n", dev->cu_next_pkt);
9708                    dev->cu_next_pkt = (void*)dev->cu_next_pkt->link;
9709                }
9710                e100_cu_command(dev, CUC_RESUME, 0);
9711 //               dev->cuhead = (uint32_t)dev->cu_next_pkt - dev->cubase;
9712            }
9713        }
9714 //       ring_printinfo(dev);
9715    }
9716    if (stat.rnr)
9717    {
9718        cprintf("e100: RU overrun!\n");
9719        dev->ru_full = 1;
9720    }
9721    if (stat.mdi)
9722        cprintf("   MDI operation completed\n");
9723    if (stat.swi)
9724        cprintf("   Software Interrupt\n");
9725    if (stat.fcp)
9726        cprintf("   Flow Control Pause\n");
9727
9728 }
9729
9730 void
9731 ring_printinfo(e100_dev *dev)
9732 {
9733    cprintf("Ring Info:   ");
9734    cprintf("head: 0x%08x  ", dev->cu_next_pkt);
9735    cprintf("end:  0x%08x\n", dev->cu_last_pkt);
9736 }
9737
9738
9739
9740
9741
9742
9743
9744
9745
9746
9747
9748
9749
```



```
9750 void *
9751 ring_alloc(e100_dev *dev, uint32_t len)
9752 {
9753    if (len > E100_CB_SIZE)
9754        return (void*) 0;
9755    if (dev->cu_next_pkt)
9756    {
9757        if ((void*)dev->cu_last_pkt->link == dev->cu_next_pkt)
9758            return (void*) 0;
9759        return (void*)dev->cu_last_pkt->link;
9760    } else {
9761        return (void*)dev->cu_last_pkt->link;
9762    }
9763 /*
9764    len = E100_CB_SIZE;
9765    if (dev->cuhead <= dev->cuend)
9766    {
9767        uint32_t newend = dev->cuend + len;
9768        uint32_t oldend = dev->cuend;
9769        if (newend >= dev->cusize)
9770        {
9771            newend = len; // Start from base
9772            if (newend < dev->cuhead)
9773            {
9774                dev->cuend = newend;
9775                return (void*)dev->cubase;
9776            }
9777            else
9778                return (void*)0;
9779        }
9780        else
9781        {
9782            dev->cuend = newend;
9783            return (void*)(oldend + dev->cubase);
9784        }
9785    }
9786    else
9787    {
9788        uint32_t newend = dev->cuend + len;
9789        uint32_t oldend = dev->cuend;
9790        if (newend < dev->cuhead)
9791        {
9792            dev->cuend = newend;
9793            return (void*)(oldend + dev->cubase);
9794        }
9795        else
9796            return (void*)0;
9797    }*/
9798 }
9799
```

```
int
e100_send_dev(e100_dev *dev, void *buffer, uint32_t len)
{
//    cprintf("e100_send_dev\n");
    if (len > TBD_DATA_LIMIT)
        return -1; // ETOOBIG
    acquire(&dev->culock);
    op_transmit_cmd *start =
        ring_alloc(dev, sizeof(op_transmit_cmd) + len);
//    cprintf("   start: 0x%08x\n", start);
    if (!start)
    {
        release(&dev->culock);
        return -1; // No space in buffer
    }
    op_transmit_cmd_cmd cmd_word;
    cmd_word.word = 0;
    cmd_word.cmd = OP_TRANSMIT;
//    cmd_word.el = 0;
    cmd_word.cid = 0xe;
    cmd_word.i = 1;
    cmd_word.s = 1;
    if (!dev->cuidle)
        dev->cu_last_pkt->s = 0;
    start->base_cmd.cmd_word = cmd_word.word;
    start->base_cmd.status_word = 0;
//    start->base_cmd.link = 0;
    start->tbd_addr = 0xffffffff;
    start->byte_count = len;
    start->eof = 1;
    start->trans_thres = 0xE0;
    start->tbd_num = 0;
    memcpy((void*)(start+1), buffer, len);
//    e100_print_state(dev);
//    if (e100_get_state(dev).status.cus == CUS_IDLE)
//    if (dev->cu_next_pkt == 0)
//    This test would fail in real i82559er
//    But it passed in qemu
/*    if ((!dev->cufirst) && (!dev->cuidle))
        if (start->base_cmd.link == dev->cu_next_pkt)
            e100_cu_command(dev, CUC_RESUME, start);*/
    if (dev->cufirst)
    {
        dev->cufirst = 0;
        dev->cu_next_pkt = &start->base_cmd;
        e100_cu_command(dev, CUC_START, start);
    }
    else if (dev->cuidle)
    {
        dev->cuidle = 0;
```



```
        dev->cu_next_pkt = &start->base_cmd;
        e100_cu_command(dev, CUC_RESUME, start); // the GENPTR is nonsense
    }
    dev->cu_last_pkt = &start->base_cmd;
    release(&dev->culock);
    return len;
}

int
e100_send(void *buffer, uint32_t len)
{
    if (dev_count == 0)
        return -2; // Device not found
    return e100_send_dev(&e100_devs[0], buffer,len);
}

void
ring_init(e100_dev *dev)
{
    int i;
    dev->cucount = dev->cusize / E100_CB_SIZE;
    dev->cu_next_pkt = 0;
    for (i=0; i<dev->cucount ; i++)
    {
        command_block *cmd = (void*)(dev->cubase + i * E100_CB_SIZE);
        cmd->cmd_word = 0;
        cmd->status_word = 0;
        cmd->c = 1;
        cmd->el = 1;
        if (i == dev->cucount - 1)
        {
            cmd->link = (void*)dev->cubase;
            dev->cu_last_pkt = cmd;
        }
        else
        {
            cmd->link = (void*)((uint32_t)cmd + E100_CB_SIZE);
        }
//        cprintf("CB %d: start 0x%08x next 0x%08x\n", i, cmd, cmd->link);
    }
    dev->rucount = dev->rusize / E100_RFD_SIZE;
    for (i=0; i<dev->rucount; i++)
    {
        rf_desc *rfd = (void*)(dev->rubase + i * E100_RFD_SIZE);
        rfd->head.cmd_word = 0;
        rfd->head.status_word = 0;
        rfd->size = E100_RFD_SIZE - sizeof(rf_desc);
        if (rfd->size & 1)
            rfd->size--;
        rfd->eof = 0;
```

```
9900         rfd->f = 0;
9901         if (i == dev->rucount - 1)
9902         {
9903             rfd->head.link = (void*)dev->rubase;
9904             rfd->head.el = 1;
9905             dev->ru_prev = rfd;
9906         }
9907         else
9908         {
9909             rfd->head.link = (void*)((uint32_t)rfd + E100_RFD_SIZE);
9910         }
9911     }
9912     dev->ru_first = 0;
9913     dev->ru_last = (void*)dev->rubase;
9914
9915 }
9916
9917 void
9918 e100_ru_start(e100_dev *dev)
9919 {
9920     dev->ru_full = 0;
9921     e100_ru_command(dev, RUC_START, (void*)dev->rubase);
9922 }
9923
9924 int
9925 e100_receive(void *buffer, uint32_t len)
9926 {
9927     if (dev_count == 0)
9928         return -2; // Device not found
9929     return e100_receive_dev(&e100_devs[0], buffer, len);
9930 }
9931
9932 int
9933 e100_receive_dev(e100_dev *dev, void *buffer, int len)
9934 {
9935     int count;
9936 //    cprintf("e100_receive_dev\n");
9937     acquire(&dev->rulock);
9938     while (dev->ru_first == 0)
9939         sleep(&dev->rulock, &dev->rulock);
9940     if (len < dev->ru_first->count)
9941     {
9942         release(&dev->rulock);
9943         return -1; // ETOOBIG
9944     }
9945     memcpy(buffer, dev->ru_first + 1, dev->ru_first->count);
9946 //    cprintf("received %d bytes into 0x%08x\n", dev->ru_first->count, buffe
9947     count = dev->ru_first->count;
9948     dev->ru_first->eof = 0;
9949     dev->ru_first->f = 0;
```



```
9950     dev->ru_prev->head.el = 0;
9951     dev->ru_first->head.el = 1;
9952     dev->ru_prev = dev->ru_first;
9953     dev->ru_first = (void*)dev->ru_first->head.link;
9954     if (dev->ru_full)
9955     {
9956         // RU full, in No Resource state
9957         dev->ru_full = 0;
9958         e100_ru_command(dev, RUC_START, dev->ru_prev);
9959     }
9960     if (dev->ru_first == dev->ru_last)
9961         dev->ru_first = 0;
9962     release(&dev->rulock);
9963     return count;
9964 }
9965
9966
9967
9968
9969
9970
9971
9972
9973
9974
9975
9976
9977
9978
9979
9980
9981
9982
9983
9984
9985
9986
9987
9988
9989
9990
9991
9992
9993
9994
9995
9996
9997
9998
9999
```

```
10000 # Initial process execs /init.
10001
10002 #include "syscall.h"
10003 #include "traps.h"
10004
10005 # exec(init, argv)
10006 .globl start
10007 start:
10008   pushl $argv
10009   pushl $init
10010   pushl $0
10011   movl $SYS_exec, %eax
10012   int $T_SYSCALL
10013
10014 # for(;;) exit();
10015 exit:
10016   movl $SYS_exit, %eax
10017   int $T_SYSCALL
10018   jmp exit
10019
10020 # char init[] = "/init\0";
10021 init:
10022   .string "/init\0"
10023
10024 # char *argv[] = { init, 0 };
10025 .p2align 2
10026 argv:
10027   .long init
10028   .long 0
10029
10030
10031
10032
10033
10034
10035
10036
10037
10038
10039
10040
10041
10042
10043
10044
10045
10046
10047
10048
10049
```



```
10050 // init: The initial user-level program
10051
10052 #include "types.h"
10053 #include "stat.h"
10054 #include "user.h"
10055 #include "fcntl.h"
10056
10057 char *sh_args[] = { "sh", 0 };
10058
10059 int
10060 main(void)
10061 {
10062   int pid, wpid;
10063
10064   if(open("console", O_RDWR) < 0){
10065     mknod("console", 1, 1);
10066     open("console", O_RDWR);
10067   }
10068   dup(0);  // stdout
10069   dup(0);  // stderr
10070
10071   for(;;){
10072     printf(1, "init: starting sh\n");
10073     pid = fork();
10074     if(pid < 0){
10075       printf(1, "init: fork failed\n");
10076       exit();
10077     }
10078     if(pid == 0){
10079       exec("sh", sh_args);
10080       printf(1, "init: exec sh failed\n");
10081       exit();
10082     }
10083     while((wpid=wait()) >= 0 && wpid != pid)
10084     {
10085 #if 0
10086       printf(1, "zombie!\n");
10087 #endif
10088     }
10089   }
10090 }
10091
10092
10093
10094
10095
10096
10097
10098
10099
```

```
10100 #include "syscall.h"
10101 #include "traps.h"
10102
10103 #define STUB(name) \
10104   .globl name; \
10105   name: \
10106     movl $SYS_ ## name, %eax; \
10107     int $T_SYSCALL; \
10108     ret
10109
10110 STUB(fork)
10111 STUB(exit)
10112 STUB(wait)
10113 STUB(pipe)
10114 STUB(read)
10115 STUB(write)
10116 STUB(close)
10117 STUB(kill)
10118 STUB(exec)
10119 STUB(open)
10120 STUB(mknod)
10121 STUB(unlink)
10122 STUB(fstat)
10123 STUB(link)
10124 STUB(mkdir)
10125 STUB(chdir)
10126 STUB(dup)
10127 STUB(getpid)
10128 STUB(sbrk)
10129 STUB(sleep)
10130 STUB(upmsec)
10131 STUB(socket)
10132 STUB(bind)
10133 STUB(listen)
10134 STUB(accept)
10135 STUB(recv)
10136 STUB(recvfrom)
10137 STUB(send)
10138 STUB(sendto)
10139 STUB(shutdown)
10140 STUB(getsockopt)
10141 STUB(setsockopt)
10142 STUB(sockclose)
10143 STUB(connect)
10144 STUB(getpeername)
10145 STUB(getsockname)
10146
10147
10148
10149
```



```
10150 // Shell.
10151
10152 #include "types.h"
10153 #include "user.h"
10154 #include "fcntl.h"
10155
10156 // Parsed command representation
10157 #define EXEC  1
10158 #define REDIR 2
10159 #define PIPE  3
10160 #define LIST  4
10161 #define BACK  5
10162
10163 #define MAXARGS 10
10164
10165 struct cmd {
10166   int type;
10167 };
10168
10169 struct execcmd {
10170   int type;
10171   char *argv[MAXARGS];
10172   char *eargv[MAXARGS];
10173 };
10174
10175 struct redircmd {
10176   int type;
10177   struct cmd *cmd;
10178   char *file;
10179   char *efile;
10180   int mode;
10181   int fd;
10182 };
10183
10184 struct pipecmd {
10185   int type;
10186   struct cmd *left;
10187   struct cmd *right;
10188 };
10189
10190 struct listcmd {
10191   int type;
10192   struct cmd *left;
10193   struct cmd *right;
10194 };
10195
10196 struct backcmd {
10197   int type;
10198   struct cmd *cmd;
10199 };
```

```
10200 int fork1(void);  // Fork but panics on failure.
10201 void panic(char*);
10202 struct cmd *parsecmd(char*);
10203
10204 // Execute cmd.  Never returns.
10205 void
10206 runcmd(struct cmd *cmd)
10207 {
10208   int p[2];
10209   struct backcmd *bcmd;
10210   struct execcmd *ecmd;
10211   struct listcmd *lcmd;
10212   struct pipecmd *pcmd;
10213   struct redircmd *rcmd;
10214
10215   if(cmd == 0)
10216     exit();
10217
10218   switch(cmd->type){
10219   default:
10220     panic("runcmd");
10221
10222   case EXEC:
10223     ecmd = (struct execcmd*)cmd;
10224     if(ecmd->argv[0] == 0)
10225       exit();
10226     exec(ecmd->argv[0], ecmd->argv);
10227     printf(2, "exec %s failed\n", ecmd->argv[0]);
10228     break;
10229
10230   case REDIR:
10231     rcmd = (struct redircmd*)cmd;
10232     close(rcmd->fd);
10233     if(open(rcmd->file, rcmd->mode) < 0){
10234       printf(2, "open %s failed\n", rcmd->file);
10235       exit();
10236     }
10237     runcmd(rcmd->cmd);
10238     break;
10239
10240   case LIST:
10241     lcmd = (struct listcmd*)cmd;
10242     if(fork1() == 0)
10243       runcmd(lcmd->left);
10244     wait();
10245     runcmd(lcmd->right);
10246     break;
10247
10248
10249
```



```
10250   case PIPE:
10251     pcmd = (struct pipecmd*)cmd;
10252     if(pipe(p) < 0)
10253       panic("pipe");
10254     if(fork1() == 0){
10255       close(1);
10256       dup(p[1]);
10257       close(p[0]);
10258       close(p[1]);
10259       runcmd(pcmd->left);
10260     }
10261     if(fork1() == 0){
10262       close(0);
10263       dup(p[0]);
10264       close(p[0]);
10265       close(p[1]);
10266       runcmd(pcmd->right);
10267     }
10268     close(p[0]);
10269     close(p[1]);
10270     wait();
10271     wait();
10272     break;
10273
10274   case BACK:
10275     bcmd = (struct backcmd*)cmd;
10276     if(fork1() == 0)
10277       runcmd(bcmd->cmd);
10278     break;
10279   }
10280   exit();
10281 }
10282
10283 int
10284 getcmd(char *buf, int nbuf)
10285 {
10286   printf(2, "$ ");
10287   memset(buf, 0, nbuf);
10288   gets(buf, nbuf);
10289   if(buf[0] == 0) // EOF
10290     return -1;
10291   return 0;
10292 }
10293
10294
10295
10296
10297
10298
10299
```

```
10300 int
10301 main(void)
10302 {
10303   static char buf[100];
10304   int fd;
10305
10306   // Assumes three file descriptors open.
10307   while((fd = open("console", O_RDWR)) >= 0){
10308     if(fd >= 3){
10309       close(fd);
10310       break;
10311     }
10312   }
10313
10314   // Read and run input commands.
10315   while(getcmd(buf, sizeof(buf)) >= 0){
10316     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
10317       // Clumsy but will have to do for now.
10318       // Chdir has no effect on the parent if run in the child.
10319       buf[strlen(buf)-1] = 0;  // chop \n
10320       if(chdir(buf+3) < 0)
10321         printf(2, "cannot cd %s\n", buf+3);
10322       continue;
10323     }
10324     if(fork1() == 0)
10325       runcmd(parsecmd(buf));
10326     wait();
10327   }
10328   exit();
10329 }
10330
10331 void
10332 panic(char *s)
10333 {
10334   printf(2, "%s\n", s);
10335   exit();
10336 }
10337
10338 int
10339 fork1(void)
10340 {
10341   int pid;
10342
10343   pid = fork();
10344   if(pid == -1)
10345     panic("fork");
10346   return pid;
10347 }
10348
10349
```



```
10350 // Constructors
10351
10352 struct cmd*
10353 execcmd(void)
10354 {
10355   struct execcmd *cmd;
10356
10357   cmd = malloc(sizeof(*cmd));
10358   memset(cmd, 0, sizeof(*cmd));
10359   cmd->type = EXEC;
10360   return (struct cmd*)cmd;
10361 }
10362
10363 struct cmd*
10364 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
10365 {
10366   struct redircmd *cmd;
10367
10368   cmd = malloc(sizeof(*cmd));
10369   memset(cmd, 0, sizeof(*cmd));
10370   cmd->type = REDIR;
10371   cmd->cmd = subcmd;
10372   cmd->file = file;
10373   cmd->efile = efile;
10374   cmd->mode = mode;
10375   cmd->fd = fd;
10376   return (struct cmd*)cmd;
10377 }
10378
10379 struct cmd*
10380 pipecmd(struct cmd *left, struct cmd *right)
10381 {
10382   struct pipecmd *cmd;
10383
10384   cmd = malloc(sizeof(*cmd));
10385   memset(cmd, 0, sizeof(*cmd));
10386   cmd->type = PIPE;
10387   cmd->left = left;
10388   cmd->right = right;
10389   return (struct cmd*)cmd;
10390 }
10391
10392
10393
10394
10395
10396
10397
10398
10399
```

```
10400 struct cmd*
10401 listcmd(struct cmd *left, struct cmd *right)
10402 {
10403   struct listcmd *cmd;
10404
10405   cmd = malloc(sizeof(*cmd));
10406   memset(cmd, 0, sizeof(*cmd));
10407   cmd->type = LIST;
10408   cmd->left = left;
10409   cmd->right = right;
10410   return (struct cmd*)cmd;
10411 }
10412
10413 struct cmd*
10414 backcmd(struct cmd *subcmd)
10415 {
10416   struct backcmd *cmd;
10417
10418   cmd = malloc(sizeof(*cmd));
10419   memset(cmd, 0, sizeof(*cmd));
10420   cmd->type = BACK;
10421   cmd->cmd = subcmd;
10422   return (struct cmd*)cmd;
10423 }
10424 // Parsing
10425
10426 char whitespace[] = " \t\r\n\v";
10427 char symbols[] = "<|>&;()";
10428
10429 int
10430 gettoken(char **ps, char *es, char **q, char **eq)
10431 {
10432   char *s;
10433   int ret;
10434
10435   s = *ps;
10436   while(s < es && strchr(whitespace, *s))
10437     s++;
10438   if(q)
10439     *q = s;
10440   ret = *s;
10441   switch(*s){
10442   case 0:
10443     break;
10444   case '|':
10445   case '(':
10446   case ')':
10447   case ';':
10448   case '&':
10449   case '<':
```



```
10450     s++;
10451     break;
10452   case '>':
10453     s++;
10454     if(*s == '>'){
10455       ret = '+';
10456       s++;
10457     }
10458     break;
10459   default:
10460     ret = 'a';
10461     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
10462       s++;
10463     break;
10464   }
10465   if(eq)
10466     *eq = s;
10467
10468   while(s < es && strchr(whitespace, *s))
10469     s++;
10470   *ps = s;
10471   return ret;
10472 }
10473
10474 int
10475 peek(char **ps, char *es, char *toks)
10476 {
10477   char *s;
10478
10479   s = *ps;
10480   while(s < es && strchr(whitespace, *s))
10481     s++;
10482   *ps = s;
10483   return *s && strchr(toks, *s);
10484 }
10485
10486
10487
10488
10489
10490
10491
10492
10493
10494
10495
10496
10497
10498
10499
```

```
10500 struct cmd *parseline(char**, char*);
10501 struct cmd *parsepipe(char**, char*);
10502 struct cmd *parseexec(char**, char*);
10503 struct cmd *nulterminate(struct cmd*);
10504
10505 struct cmd*
10506 parsecmd(char *s)
10507 {
10508   char *es;
10509   struct cmd *cmd;
10510
10511   es = s + strlen(s);
10512   cmd = parseline(&s, es);
10513   peek(&s, es, "");
10514   if(s != es){
10515     printf(2, "leftovers: %s\n", s);
10516     panic("syntax");
10517   }
10518   nulterminate(cmd);
10519   return cmd;
10520 }
10521
10522 struct cmd*
10523 parseline(char **ps, char *es)
10524 {
10525   struct cmd *cmd;
10526
10527   cmd = parsepipe(ps, es);
10528   while(peek(ps, es, "&")){
10529     gettoken(ps, es, 0, 0);
10530     cmd = backcmd(cmd);
10531   }
10532   if(peek(ps, es, ";")){
10533     gettoken(ps, es, 0, 0);
10534     cmd = listcmd(cmd, parseline(ps, es));
10535   }
10536   return cmd;
10537 }
10538
10539
10540
10541
10542
10543
10544
10545
10546
10547
10548
10549
```



```
10550 struct cmd*
10551 parsepipe(char **ps, char *es)
10552 {
10553   struct cmd *cmd;
10554
10555   cmd = parseexec(ps, es);
10556   if(peek(ps, es, "|")){
10557     gettoken(ps, es, 0, 0);
10558     cmd = pipecmd(cmd, parsepipe(ps, es));
10559   }
10560   return cmd;
10561 }
10562
10563 struct cmd*
10564 parseredirs(struct cmd *cmd, char **ps, char *es)
10565 {
10566   int tok;
10567   char *q, *eq;
10568
10569   while(peek(ps, es, "<>")){
10570     tok = gettoken(ps, es, 0, 0);
10571     if(gettoken(ps, es, &q, &eq) != 'a')
10572       panic("missing file for redirection");
10573     switch(tok){
10574     case '<':
10575       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
10576       break;
10577     case '>':
10578       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
10579       break;
10580     case '+':  // >>
10581       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
10582       break;
10583     }
10584   }
10585   return cmd;
10586 }
10587
10588
10589
10590
10591
10592
10593
10594
10595
10596
10597
10598
10599
```

```
10600 struct cmd*
10601 parseblock(char **ps, char *es)
10602 {
10603   struct cmd *cmd;
10604
10605   if(!peek(ps, es, "("))
10606     panic("parseblock");
10607   gettoken(ps, es, 0, 0);
10608   cmd = parseline(ps, es);
10609   if(!peek(ps, es, ")"))
10610     panic("syntax - missing )");
10611   gettoken(ps, es, 0, 0);
10612   cmd = parseredirs(cmd, ps, es);
10613   return cmd;
10614 }
10615
10616 struct cmd*
10617 parseexec(char **ps, char *es)
10618 {
10619   char *q, *eq;
10620   int tok, argc;
10621   struct execcmd *cmd;
10622   struct cmd *ret;
10623
10624   if(peek(ps, es, "("))
10625     return parseblock(ps, es);
10626
10627   ret = execcmd();
10628   cmd = (struct execcmd*)ret;
10629
10630   argc = 0;
10631   ret = parseredirs(ret, ps, es);
10632   while(!peek(ps, es, "|)&;")){
10633     if((tok=gettoken(ps, es, &q, &eq)) == 0)
10634       break;
10635     if(tok != 'a')
10636       panic("syntax");
10637     cmd->argv[argc] = q;
10638     cmd->eargv[argc] = eq;
10639     argc++;
10640     if(argc >= MAXARGS)
10641       panic("too many args");
10642     ret = parseredirs(ret, ps, es);
10643   }
10644   cmd->argv[argc] = 0;
10645   cmd->eargv[argc] = 0;
10646   return ret;
10647 }
10648
10649
```



```
10650 // NUL-terminate all the counted strings.
10651 struct cmd*
10652 nulterminate(struct cmd *cmd)
10653 {
10654   int i;
10655   struct backcmd *bcmd;
10656   struct execcmd *ecmd;
10657   struct listcmd *lcmd;
10658   struct pipecmd *pcmd;
10659   struct redircmd *rcmd;
10660
10661   if(cmd == 0)
10662     return 0;
10663
10664   switch(cmd->type){
10665   case EXEC:
10666     ecmd = (struct execcmd*)cmd;
10667     for(i=0; ecmd->argv[i]; i++)
10668       *ecmd->eargv[i] = 0;
10669     break;
10670
10671   case REDIR:
10672     rcmd = (struct redircmd*)cmd;
10673     nulterminate(rcmd->cmd);
10674     *rcmd->efile = 0;
10675     break;
10676
10677   case PIPE:
10678     pcmd = (struct pipecmd*)cmd;
10679     nulterminate(pcmd->left);
10680     nulterminate(pcmd->right);
10681     break;
10682
10683   case LIST:
10684     lcmd = (struct listcmd*)cmd;
10685     nulterminate(lcmd->left);
10686     nulterminate(lcmd->right);
10687     break;
10688
10689   case BACK:
10690     bcmd = (struct backcmd*)cmd;
10691     nulterminate(bcmd->cmd);
10692     break;
10693   }
10694   return cmd;
10695 }
10696
10697
10698
10699
```