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/6.828/2014/xv6.html, which
provides pointers to on-line resources for v6.

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

The following people have made contributions:
    Russ Cox (context switching, locking)
    Cliff Frey (MP)
    Xiao Yu (MP)
    Nickolai Zeldovich
    Austin Clements

In addition, we are grateful for the bug reports and patches contributed by
Silas Boyd-Wickizer, Peter Froehlich, Shivam Handa, Anders Kaseorg, Eddie
Kohler, Yandong Mao, Hitoshi Mitake, Carmi Merimovich, Joel Nider, Greg Price,
Eldar Sehayek, Yongming Shen, Stephen Tu, and Zouchangwei.

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/6.828/2014/tools.html.
Then run "make TOOLPREFIX=i386-jos-elf-".

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

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

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

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

    swtch 2658
        0374 2428 2466 2657 2658

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

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



```
0150 #define NPROC        64  // maximum number of processes
0151 #define KSTACKSIZE 4096  // size of per-process kernel stack
0152 #define NCPU          8  // maximum number of CPUs
0153 #define NOFILE       16  // open files per process
0154 #define NFILE       100  // open files per system
0155 #define NINODE       50  // maximum number of active i-nodes
0156 #define NDEV         10  // maximum major device number
0157 #define ROOTDEV       1  // device number of file system root disk
0158 #define MAXARG       32  // max exec arguments
0159 #define MAXOPBLOCKS  10  // max # of blocks any FS op writes
0160 #define LOGSIZE      (MAXOPBLOCKS*3)  // max data blocks in on-disk log
0161 #define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
0162 #define FSSIZE       1000  // size of file system in blocks
0163
0164
0165
0166
0167
0168
0169
0170
0171
0172
0173
0174
0175
0176
0177
0178
0179
0180
0181
0182
0183
0184
0185
0186
0187
0188
0189
0190
0191
0192
0193
0194
0195
0196
0197
0198
0199
```

```
0200 // Memory layout
0201
0202 #define EXTMEM  0x100000            // Start of extended memory
0203 #define PHYSTOP 0xE000000           // Top physical memory
0204 #define DEVSPACE 0xFE000000         // Other devices are at high addresses
0205
0206 // Key addresses for address space layout (see kmap in vm.c for layout)
0207 #define KERNBASE 0x80000000         // First kernel virtual address
0208 #define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked
0209
0210 #ifndef __ASSEMBLER__
0211
0212 static inline uint v2p(void *a) { return ((uint) (a))  - KERNBASE; }
0213 static inline void *p2v(uint a) { return (void *) ((a) + KERNBASE); }
0214
0215 #endif
0216
0217 #define V2P(a) (((uint) (a)) - KERNBASE)
0218 #define P2V(a) (((void *) (a)) + KERNBASE)
0219
0220 #define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
0221 #define P2V_WO(x) ((x) + KERNBASE)    // same as P2V, but without casts
0222
0223
0224
0225
0226
0227
0228
0229
0230
0231
0232
0233
0234
0235
0236
0237
0238
0239
0240
0241
0242
0243
0244
0245
0246
0247
0248
0249
```



```
0250 struct buf;
0251 struct context;
0252 struct file;
0253 struct inode;
0254 struct pipe;
0255 struct proc;
0256 struct rtcdate;
0257 struct spinlock;
0258 struct stat;
0259 struct superblock;
0260 struct uproc;
0261
0262 // bio.c
0263 void            binit(void);
0264 struct buf*     bread(uint, uint);
0265 void            brelse(struct buf*);
0266 void            bwrite(struct buf*);
0267
0268 // console.c
0269 void            consoleinit(void);
0270 void            cprintf(char*, ...);
0271 void            consoleintr(int(*)(void));
0272 void            panic(char*) __attribute__((noreturn));
0273
0274 // exec.c
0275 int             exec(char*, char**);
0276
0277 // file.c
0278 struct file*    filealloc(void);
0279 void            fileclose(struct file*);
0280 struct file*    filedup(struct file*);
0281 void            fileinit(void);
0282 int             fileread(struct file*, char*, int n);
0283 int             filestat(struct file*, struct stat*);
0284 int             filewrite(struct file*, char*, int n);
0285
0286 // fs.c
0287 void            readsb(int dev, struct superblock *sb);
0288 int             dirlink(struct inode*, char*, uint);
0289 struct inode*   dirlookup(struct inode*, char*, uint*);
0290 struct inode*   ialloc(uint, short);
0291 struct inode*   idup(struct inode*);
0292 void            iinit(int dev);
0293 void            ilock(struct inode*);
0294 void            iput(struct inode*);
0295 void            iunlock(struct inode*);
0296 void            iunlockput(struct inode*);
0297 void            iupdate(struct inode*);
0298 int             namecmp(const char*, const char*);
0299 struct inode*   namei(char*);
```

```
0300 struct inode*   nameiparent(char*, char*);
0301 int             readi(struct inode*, char*, uint, uint);
0302 void            stati(struct inode*, struct stat*);
0303 int             writei(struct inode*, char*, uint, uint);
0304
0305 // ide.c
0306 void            ideinit(void);
0307 void            ideintr(void);
0308 void            iderw(struct buf*);
0309
0310 // ioapic.c
0311 void            ioapicenable(int irq, int cpu);
0312 extern uchar    ioapicid;
0313 void            ioapicinit(void);
0314
0315 // kalloc.c
0316 char*           kalloc(void);
0317 void            kfree(char*);
0318 void            kinit1(void*, void*);
0319 void            kinit2(void*, void*);
0320
0321 // kbd.c
0322 void            kbdintr(void);
0323
0324 // lapic.c
0325 void            cmostime(struct rtcdate *r);
0326 int             cpunum(void);
0327 extern volatile uint*    lapic;
0328 void            lapiceoi(void);
0329 void            lapicinit(void);
0330 void            lapicstartap(uchar, uint);
0331 void            microdelay(int);
0332
0333 // log.c
0334 void            initlog(int dev);
0335 void            log_write(struct buf*);
0336 void            begin_op();
0337 void            end_op();
0338
0339 // mp.c
0340 extern int      ismp;
0341 int             mpbcpu(void);
0342 void            mpinit(void);
0343 void            mpstartthem(void);
0344
0345 // picirq.c
0346 void            picenable(int);
0347 void            picinit(void);
0348
0349
```



```
0350 // pipe.c
0351 int             pipealloc(struct file**, struct file**);
0352 void            pipeclose(struct pipe*, int);
0353 int             piperead(struct pipe*, char*, int);
0354 int             pipewrite(struct pipe*, char*, int);
0355
0356
0357 // proc.c
0358 struct proc*    copyproc(struct proc*);
0359 void            exit(void);
0360 int             fork(void);
0361 int             growproc(int);
0362 int             kill(int);
0363 void            pinit(void);
0364 void            procdump(void);
0365 void            scheduler(void) __attribute__((noreturn));
0366 void            sched(void);
0367 void            sleep(void*, struct spinlock*);
0368 void            userinit(void);
0369 int             wait(void);
0370 void            wakeup(void*);
0371 void            yield(void);
0372
0373 // PROJECT 3
0374 int             copy(int,struct uproc*);
0375
0376 // PROJECT 4
0377 int                                                     setPriority(uint,
0378 void                                            enqueue(struct proc*, uin
0379 void                                            insert(struct proc*);
0380 void                                            reset(void);
0381
0382 // swtch.S
0383 void            swtch(struct context**, struct context*);
0384
0385 // spinlock.c
0386 void            acquire(struct spinlock*);
0387 void            getcallerpcs(void*, uint*);
0388 int             holding(struct spinlock*);
0389 void            initlock(struct spinlock*, char*);
0390 void            release(struct spinlock*);
0391 void            pushcli(void);
0392 void            popcli(void);
0393
0394 // string.c
0395 int             memcmp(const void*, const void*, uint);
0396 void*           memmove(void*, const void*, uint);
0397 void*           memset(void*, int, uint);
0398 char*           safestrcpy(char*, const char*, int);
0399 int             strlen(const char*);
```

```
0400 int             strncmp(const char*, const char*, uint);
0401 char*           strncpy(char*, const char*, int);
0402
0403 // syscall.c
0404 int             argint(int, int*);
0405 int             argptr(int, char**, int);
0406 int             argstr(int, char**);
0407 int             fetchint(uint, int*);
0408 int             fetchstr(uint, char**);
0409 void            syscall(void);
0410
0411 // timer.c
0412 void            timerinit(void);
0413
0414 // trap.c
0415 void            idtinit(void);
0416 extern uint     ticks;
0417 void            tvinit(void);
0418 extern struct spinlock tickslock;
0419
0420 // uart.c
0421 void            uartinit(void);
0422 void            uartintr(void);
0423 void            uartputc(int);
0424
0425 // vm.c
0426 void            seginit(void);
0427 void            kvmalloc(void);
0428 void            vmenable(void);
0429 pde_t*          setupkvm(void);
0430 char*           uva2ka(pde_t*, char*);
0431 int             allocuvm(pde_t*, uint, uint);
0432 int             deallocuvm(pde_t*, uint, uint);
0433 void            freevm(pde_t*);
0434 void            inituvm(pde_t*, char*, uint);
0435 int             loaduvm(pde_t*, char*, struct inode*, uint, uint);
0436 pde_t*          copyuvm(pde_t*, uint);
0437 void            switchuvm(struct proc*);
0438 void            switchkvm(void);
0439 int             copyout(pde_t*, uint, void*, uint);
0440 void            clearpteu(pde_t *pgdir, char *uva);
0441
0442 // number of elements in fixed-size array
0443 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
0444
0445
0446
0447
0448
0449
```



```
0450 // Routines to let C code use special x86 instructions.
0451
0452 static inline uchar
0453 inb(ushort port)
0454 {
0455   uchar data;
0456
0457   asm volatile("in %1,%0" : "=a" (data) : "d" (port));
0458   return data;
0459 }
0460
0461 static inline void
0462 insl(int port, void *addr, int cnt)
0463 {
0464   asm volatile("cld; rep insl" :
0465                "=D" (addr), "=c" (cnt) :
0466                "d" (port), "0" (addr), "1" (cnt) :
0467                "memory", "cc");
0468 }
0469
0470 static inline void
0471 outb(ushort port, uchar data)
0472 {
0473   asm volatile("out %0,%1" : : "a" (data), "d" (port));
0474 }
0475
0476 static inline void
0477 outw(ushort port, ushort data)
0478 {
0479   asm volatile("out %0,%1" : : "a" (data), "d" (port));
0480 }
0481
0482 static inline void
0483 outsl(int port, const void *addr, int cnt)
0484 {
0485   asm volatile("cld; rep outsl" :
0486                "=S" (addr), "=c" (cnt) :
0487                "d" (port), "0" (addr), "1" (cnt) :
0488                "cc");
0489 }
0490
0491 static inline void
0492 stosb(void *addr, int data, int cnt)
0493 {
0494   asm volatile("cld; rep stosb" :
0495                "=D" (addr), "=c" (cnt) :
0496                "0" (addr), "1" (cnt), "a" (data) :
0497                "memory", "cc");
0498 }
0499
```

```
0500 static inline void
0501 stosl(void *addr, int data, int cnt)
0502 {
0503   asm volatile("cld; rep stosl" :
0504                "=D" (addr), "=c" (cnt) :
0505                "0" (addr), "1" (cnt), "a" (data) :
0506                "memory", "cc");
0507 }
0508
0509 struct segdesc;
0510
0511 static inline void
0512 lgdt(struct segdesc *p, int size)
0513 {
0514   volatile ushort pd[3];
0515
0516   pd[0] = size-1;
0517   pd[1] = (uint)p;
0518   pd[2] = (uint)p >> 16;
0519
0520   asm volatile("lgdt (%0)" : : "r" (pd));
0521 }
0522
0523 struct gatedesc;
0524
0525 static inline void
0526 lidt(struct gatedesc *p, int size)
0527 {
0528   volatile ushort pd[3];
0529
0530   pd[0] = size-1;
0531   pd[1] = (uint)p;
0532   pd[2] = (uint)p >> 16;
0533
0534   asm volatile("lidt (%0)" : : "r" (pd));
0535 }
0536
0537 static inline void
0538 ltr(ushort sel)
0539 {
0540   asm volatile("ltr %0" : : "r" (sel));
0541 }
0542
0543 static inline uint
0544 readeflags(void)
0545 {
0546   uint eflags;
0547   asm volatile("pushfl; popl %0" : "=r" (eflags));
0548   return eflags;
0549 }
```



```
0550 static inline void
0551 loadgs(ushort v)
0552 {
0553   asm volatile("movw %0, %%gs" : : "r" (v));
0554 }
0555
0556 static inline void
0557 cli(void)
0558 {
0559   asm volatile("cli");
0560 }
0561
0562 static inline void
0563 sti(void)
0564 {
0565   asm volatile("sti");
0566 }
0567
0568 static inline uint
0569 xchg(volatile uint *addr, uint newval)
0570 {
0571   uint result;
0572
0573   // The + in "+m" denotes a read-modify-write operand.
0574   asm volatile("lock; xchgl %0, %1" :
0575                "+m" (*addr), "=a" (result) :
0576                "1" (newval) :
0577                "cc");
0578   return result;
0579 }
0580
0581 static inline uint
0582 rcr2(void)
0583 {
0584   uint val;
0585   asm volatile("movl %%cr2,%0" : "=r" (val));
0586   return val;
0587 }
0588
0589 static inline void
0590 lcr3(uint val)
0591 {
0592   asm volatile("movl %0,%%cr3" : : "r" (val));
0593 }
0594
0595
0596
0597
0598
0599
```

```
0600 // Layout of the trap frame built on the stack by the
0601 // hardware and by trapasm.S, and passed to trap().
0602 struct trapframe {
0603   // registers as pushed by pusha
0604   uint edi;
0605   uint esi;
0606   uint ebp;
0607   uint oesp;      // useless & ignored
0608   uint ebx;
0609   uint edx;
0610   uint ecx;
0611   uint eax;
0612
0613   // rest of trap frame
0614   ushort gs;
0615   ushort padding1;
0616   ushort fs;
0617   ushort padding2;
0618   ushort es;
0619   ushort padding3;
0620   ushort ds;
0621   ushort padding4;
0622   uint trapno;
0623
0624   // below here defined by x86 hardware
0625   uint err;
0626   uint eip;
0627   ushort cs;
0628   ushort padding5;
0629   uint eflags;
0630
0631   // below here only when crossing rings, such as from user to kernel
0632   uint esp;
0633   ushort ss;
0634   ushort padding6;
0635 };
0636
0637
0638
0639
0640
0641
0642
0643
0644
0645
0646
0647
0648
0649
```



```
0650 //
0651 // assembler macros to create x86 segments
0652 //
0653
0654 #define SEG_NULLASM                                             \
0655         .word 0, 0;                                             \
0656         .byte 0, 0, 0, 0
0657
0658 // The 0xC0 means the limit is in 4096-byte units
0659 // and (for executable segments) 32-bit mode.
0660 #define SEG_ASM(type,base,lim)                                  \
0661         .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
0662         .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
0663                 (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
0664
0665 #define STA_X     0x8       // Executable segment
0666 #define STA_E     0x4       // Expand down (non-executable segments)
0667 #define STA_C     0x4       // Conforming code segment (executable only)
0668 #define STA_W     0x2       // Writeable (non-executable segments)
0669 #define STA_R     0x2       // Readable (executable segments)
0670 #define STA_A     0x1       // Accessed
0671
0672
0673
0674
0675
0676
0677
0678
0679
0680
0681
0682
0683
0684
0685
0686
0687
0688
0689
0690
0691
0692
0693
0694
0695
0696
0697
0698
0699
```

```
0700 // This file contains definitions for the
0701 // x86 memory management unit (MMU).
0702
0703 // Eflags register
0704 #define FL_CF           0x00000001      // Carry Flag
0705 #define FL_PF           0x00000004      // Parity Flag
0706 #define FL_AF           0x00000010      // Auxiliary carry Flag
0707 #define FL_ZF           0x00000040      // Zero Flag
0708 #define FL_SF           0x00000080      // Sign Flag
0709 #define FL_TF           0x00000100      // Trap Flag
0710 #define FL_IF           0x00000200      // Interrupt Enable
0711 #define FL_DF           0x00000400      // Direction Flag
0712 #define FL_OF           0x00000800      // Overflow Flag
0713 #define FL_IOPL_MASK    0x00003000      // I/O Privilege Level bitmask
0714 #define FL_IOPL_0       0x00000000      //   IOPL == 0
0715 #define FL_IOPL_1       0x00001000      //   IOPL == 1
0716 #define FL_IOPL_2       0x00002000      //   IOPL == 2
0717 #define FL_IOPL_3       0x00003000      //   IOPL == 3
0718 #define FL_NT           0x00004000      // Nested Task
0719 #define FL_RF           0x00010000      // Resume Flag
0720 #define FL_VM           0x00020000      // Virtual 8086 mode
0721 #define FL_AC           0x00040000      // Alignment Check
0722 #define FL_VIF          0x00080000      // Virtual Interrupt Flag
0723 #define FL_VIP          0x00100000      // Virtual Interrupt Pending
0724 #define FL_ID           0x00200000      // ID flag
0725
0726 // Control Register flags
0727 #define CR0_PE          0x00000001      // Protection Enable
0728 #define CR0_MP          0x00000002      // Monitor coProcessor
0729 #define CR0_EM          0x00000004      // Emulation
0730 #define CR0_TS          0x00000008      // Task Switched
0731 #define CR0_ET          0x00000010      // Extension Type
0732 #define CR0_NE          0x00000020      // Numeric Errror
0733 #define CR0_WP          0x00010000      // Write Protect
0734 #define CR0_AM          0x00040000      // Alignment Mask
0735 #define CR0_NW          0x20000000      // Not Writethrough
0736 #define CR0_CD          0x40000000      // Cache Disable
0737 #define CR0_PG          0x80000000      // Paging
0738
0739 #define CR4_PSE         0x00000010      // Page size extension
0740
0741 #define SEG_KCODE 1  // kernel code
0742 #define SEG_KDATA 2  // kernel data+stack
0743 #define SEG_KCPU  3  // kernel per-cpu data
0744 #define SEG_UCODE 4  // user code
0745 #define SEG_UDATA 5  // user data+stack
0746 #define SEG_TSS   6  // this process's task state
0747
0748
0749
```



```
0750 #ifndef __ASSEMBLER__
0751 // Segment Descriptor
0752 struct segdesc {
0753   uint lim_15_0 : 16;  // Low bits of segment limit
0754   uint base_15_0 : 16; // Low bits of segment base address
0755   uint base_23_16 : 8; // Middle bits of segment base address
0756   uint type : 4;       // Segment type (see STS_ constants)
0757   uint s : 1;          // 0 = system, 1 = application
0758   uint dpl : 2;        // Descriptor Privilege Level
0759   uint p : 1;          // Present
0760   uint lim_19_16 : 4;  // High bits of segment limit
0761   uint avl : 1;        // Unused (available for software use)
0762   uint rsv1 : 1;       // Reserved
0763   uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
0764   uint g : 1;          // Granularity: limit scaled by 4K when set
0765   uint base_31_24 : 8; // High bits of segment base address
0766 };
0767
0768 // Normal segment
0769 #define SEG(type, base, lim, dpl) (struct segdesc)    \
0770 { ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff,      \
0771   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
0772   (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
0773 #define SEG16(type, base, lim, dpl) (struct segdesc)  \
0774 { (lim) & 0xffff, (uint)(base) & 0xffff,              \
0775   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
0776   (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
0777 #endif
0778
0779 #define DPL_USER    0x3     // User DPL
0780
0781 // Application segment type bits
0782 #define STA_X       0x8     // Executable segment
0783 #define STA_E       0x4     // Expand down (non-executable segments)
0784 #define STA_C       0x4     // Conforming code segment (executable only)
0785 #define STA_W       0x2     // Writeable (non-executable segments)
0786 #define STA_R       0x2     // Readable (executable segments)
0787 #define STA_A       0x1     // Accessed
0788
0789 // System segment type bits
0790 #define STS_T16A    0x1     // Available 16-bit TSS
0791 #define STS_LDT     0x2     // Local Descriptor Table
0792 #define STS_T16B    0x3     // Busy 16-bit TSS
0793 #define STS_CG16    0x4     // 16-bit Call Gate
0794 #define STS_TG      0x5     // Task Gate / Coum Transmitions
0795 #define STS_IG16    0x6     // 16-bit Interrupt Gate
0796 #define STS_TG16    0x7     // 16-bit Trap Gate
0797 #define STS_T32A    0x9     // Available 32-bit TSS
0798 #define STS_T32B    0xB     // Busy 32-bit TSS
0799 #define STS_CG32    0xC     // 32-bit Call Gate
```

```
0800 #define STS_IG32    0xE     // 32-bit Interrupt Gate
0801 #define STS_TG32    0xF     // 32-bit Trap Gate
0802
0803 // A virtual address 'la' has a three-part structure as follows:
0804 //
0805 // +--------10------+-------10-------+---------12----------+
0806 // | Page Directory |   Page Table   | Offset within Page  |
0807 // |      Index     |      Index     |                     |
0808 // +----------------+----------------+---------------------+
0809 //  \--- PDX(va) --/ \--- PTX(va) --/
0810
0811 // page directory index
0812 #define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)
0813
0814 // page table index
0815 #define PTX(va)         (((uint)(va) >> PTXSHIFT) & 0x3FF)
0816
0817 // construct virtual address from indexes and offset
0818 #define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
0819
0820 // Page directory and page table constants.
0821 #define NPDENTRIES      1024    // # directory entries per page directory
0822 #define NPTENTRIES      1024    // # PTEs per page table
0823 #define PGSIZE          4096    // bytes mapped by a page
0824
0825 #define PGSHIFT         12      // log2(PGSIZE)
0826 #define PTXSHIFT        12      // offset of PTX in a linear address
0827 #define PDXSHIFT        22      // offset of PDX in a linear address
0828
0829 #define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
0830 #define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
0831
0832 // Page table/directory entry flags.
0833 #define PTE_P           0x001   // Present
0834 #define PTE_W           0x002   // Writeable
0835 #define PTE_U           0x004   // User
0836 #define PTE_PWT         0x008   // Write-Through
0837 #define PTE_PCD         0x010   // Cache-Disable
0838 #define PTE_A           0x020   // Accessed
0839 #define PTE_D           0x040   // Dirty
0840 #define PTE_PS          0x080   // Page Size
0841 #define PTE_MBZ         0x180   // Bits must be zero
0842
0843 // Address in page table or page directory entry
0844 #define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
0845 #define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF)
0846
0847 #ifndef __ASSEMBLER__
0848 typedef uint pte_t;
0849
```



```
0850 // Task state segment format
0851 struct taskstate {
0852   uint link;         // Old ts selector
0853   uint esp0;         // Stack pointers and segment selectors
0854   ushort ss0;        //   after an increase in privilege level
0855   ushort padding1;
0856   uint *esp1;
0857   ushort ss1;
0858   ushort padding2;
0859   uint *esp2;
0860   ushort ss2;
0861   ushort padding3;
0862   void *cr3;         // Page directory base
0863   uint *eip;         // Saved state from last task switch
0864   uint eflags;
0865   uint eax;          // More saved state (registers)
0866   uint ecx;
0867   uint edx;
0868   uint ebx;
0869   uint *esp;
0870   uint *ebp;
0871   uint esi;
0872   uint edi;
0873   ushort es;         // Even more saved state (segment selectors)
0874   ushort padding4;
0875   ushort cs;
0876   ushort padding5;
0877   ushort ss;
0878   ushort padding6;
0879   ushort ds;
0880   ushort padding7;
0881   ushort fs;
0882   ushort padding8;
0883   ushort gs;
0884   ushort padding9;
0885   ushort ldt;
0886   ushort padding10;
0887   ushort t;          // Trap on task switch
0888   ushort iomb;       // I/O map base address
0889 };
0890
0891
0892
0893
0894
0895
0896
0897
0898
0899
```

```
0900 // Gate descriptors for interrupts and traps
0901 struct gatedesc {
0902   uint off_15_0 : 16;   // low 16 bits of offset in segment
0903   uint cs : 16;         // code segment selector
0904   uint args : 5;        // # args, 0 for interrupt/trap gates
0905   uint rsv1 : 3;        // reserved(should be zero I guess)
0906   uint type : 4;        // type(STS_{TG,IG32,TG32})
0907   uint s : 1;           // must be 0 (system)
0908   uint dpl : 2;         // descriptor(meaning new) privilege level
0909   uint p : 1;           // Present
0910   uint off_31_16 : 16;  // high bits of offset in segment
0911 };
0912
0913 // Set up a normal interrupt/trap gate descriptor.
0914 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
0915 //   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
0916 // - sel: Code segment selector for interrupt/trap handler
0917 // - off: Offset in code segment for interrupt/trap handler
0918 // - dpl: Descriptor Privilege Level -
0919 //        the privilege level required for software to invoke
0920 //        this interrupt/trap gate explicitly using an int instruction.
0921 #define SETGATE(gate, istrap, sel, off, d)                \
0922 {                                                         \
0923   (gate).off_15_0 = (uint)(off) & 0xffff;                \
0924   (gate).cs = (sel);                                      \
0925   (gate).args = 0;                                        \
0926   (gate).rsv1 = 0;                                        \
0927   (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
0928   (gate).s = 0;                                           \
0929   (gate).dpl = (d);                                       \
0930   (gate).p = 1;                                           \
0931   (gate).off_31_16 = (uint)(off) >> 16;                  \
0932 }
0933
0934 #endif
0935
0936
0937
0938
0939
0940
0941
0942
0943
0944
0945
0946
0947
0948
0949
```



```
0950 // Format of an ELF executable file
0951
0952 #define ELF_MAGIC 0x464C457FU  // "\x7FELF" in little endian
0953
0954 // File header
0955 struct elfhdr {
0956   uint magic;  // must equal ELF_MAGIC
0957   uchar elf[12];
0958   ushort type;
0959   ushort machine;
0960   uint version;
0961   uint entry;
0962   uint phoff;
0963   uint shoff;
0964   uint flags;
0965   ushort ehsize;
0966   ushort phentsize;
0967   ushort phnum;
0968   ushort shentsize;
0969   ushort shnum;
0970   ushort shstrndx;
0971 };
0972
0973 // Program section header
0974 struct proghdr {
0975   uint type;
0976   uint off;
0977   uint vaddr;
0978   uint paddr;
0979   uint filesz;
0980   uint memsz;
0981   uint flags;
0982   uint align;
0983 };
0984
0985 // Values for Proghdr type
0986 #define ELF_PROG_LOAD           1
0987
0988 // Flag bits for Proghdr flags
0989 #define ELF_PROG_FLAG_EXEC      1
0990 #define ELF_PROG_FLAG_WRITE     2
0991 #define ELF_PROG_FLAG_READ      4
0992
0993
0994
0995
0996
0997
0998
0999
```

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

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

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

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

# Entering xv6 on boot processor, with paging off.
.globl entry
entry:
  # Turn on page size extension for 4Mbyte pages
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  # Set page directory
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  # Turn on paging.
  movl    %cr0, %eax
```



```
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0

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

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

.comm stack, KSTACKSIZE
```

```
1100 #include "asm.h"
1101 #include "memlayout.h"
1102 #include "mmu.h"
1103
1104 # Each non-boot CPU ("AP") is started up in response to a STARTUP
1105 # IPI from the boot CPU.  Section B.4.2 of the Multi-Processor
1106 # Specification says that the AP will start in real mode with CS:IP
1107 # set to XY00:0000, where XY is an 8-bit value sent with the
1108 # STARTUP. Thus this code must start at a 4096-byte boundary.
1109 #
1110 # Because this code sets DS to zero, it must sit
1111 # at an address in the low 2^16 bytes.
1112 #
1113 # Startothers (in main.c) sends the STARTUPs one at a time.
1114 # It copies this code (start) at 0x7000.  It puts the address of
1115 # a newly allocated per-core stack in start-4,the address of the
1116 # place to jump to (mpenter) in start-8, and the physical address
1117 # of entrypgdir in start-12.
1118 #
1119 # This code is identical to bootasm.S except:
1120 #   - it does not need to enable A20
1121 #   - it uses the address at start-4, start-8, and start-12
1122
1123 .code16
1124 .globl start
1125 start:
1126   cli
1127
1128   xorw    %ax,%ax
1129   movw    %ax,%ds
1130   movw    %ax,%es
1131   movw    %ax,%ss
1132
1133   lgdt    gdtdesc
1134   movl    %cr0, %eax
1135   orl     $CR0_PE, %eax
1136   movl    %eax, %cr0
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
```



```
1150   ljmpl   $(SEG_KCODE<<3), $(start32)
1151
1152 .code32
1153 start32:
1154   movw    $(SEG_KDATA<<3), %ax
1155   movw    %ax, %ds
1156   movw    %ax, %es
1157   movw    %ax, %ss
1158   movw    $0, %ax
1159   movw    %ax, %fs
1160   movw    %ax, %gs
1161
1162   # Turn on page size extension for 4Mbyte pages
1163   movl    %cr4, %eax
1164   orl     $(CR4_PSE), %eax
1165   movl    %eax, %cr4
1166   # Use enterpgdir as our initial page table
1167   movl    (start-12), %eax
1168   movl    %eax, %cr3
1169   # Turn on paging.
1170   movl    %cr0, %eax
1171   orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
1172   movl    %eax, %cr0
1173
1174   # Switch to the stack allocated by startothers()
1175   movl    (start-4), %esp
1176   # Call mpenter()
1177   call      *(start-8)
1178
1179   movw    $0x8a00, %ax
1180   movw    %ax, %dx
1181   outw    %ax, %dx
1182   movw    $0x8ae0, %ax
1183   outw    %ax, %dx
1184 spin:
1185   jmp     spin
1186
1187 .p2align 2
1188 gdt:
1189   SEG_NULLASM
1190   SEG_ASM(STA_X|STA_R, 0, 0xffffffff)
1191   SEG_ASM(STA_W, 0, 0xffffffff)
1192
1193
1194 gdtdesc:
1195   .word   (gdtdesc - gdt - 1)
1196   .long   gdt
1197
1198
1199
```

```
1200 #include "types.h"
1201 #include "defs.h"
1202 #include "param.h"
1203 #include "memlayout.h"
1204 #include "mmu.h"
1205 #include "proc.h"
1206 #include "x86.h"
1207
1208 static void startothers(void);
1209 static void mpmain(void)  __attribute__((noreturn));
1210 extern pde_t *kpgdir;
1211 extern char end[]; // first address after kernel loaded from ELF file
1212
1213 // Bootstrap processor starts running C code here.
1214 // Allocate a real stack and switch to it, first
1215 // doing some setup required for memory allocator to work.
1216 int
1217 main(void)
1218 {
1219   kinit1(end, P2V(4*1024*1024)); // phys page allocator
1220   kvmalloc();      // kernel page table
1221   mpinit();        // collect info about this machine
1222   lapicinit();
1223   seginit();       // set up segments
1224   cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
1225   picinit();       // interrupt controller
1226   ioapicinit();    // another interrupt controller
1227   consoleinit();   // I/O devices & their interrupts
1228   uartinit();      // serial port
1229   pinit();         // process table
1230   tvinit();        // trap vectors
1231   binit();         // buffer cache
1232   fileinit();      // file table
1233   ideinit();       // disk
1234   if(!ismp)
1235     timerinit();   // uniprocessor timer
1236   startothers();   // start other processors
1237   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1238   userinit();      // first user process
1239   // Finish setting up this processor in mpmain.
1240   mpmain();
1241 }
1242
1243
1244
1245
1246
1247
1248
1249
```



```
1250 // Other CPUs jump here from entryother.S.
1251 static void
1252 mpenter(void)
1253 {
1254   switchkvm();
1255   seginit();
1256   lapicinit();
1257   mpmain();
1258 }
1259
1260 // Common CPU setup code.
1261 static void
1262 mpmain(void)
1263 {
1264   cprintf("cpu%d: starting\n", cpu->id);
1265   idtinit();       // load idt register
1266   xchg(&cpu->started, 1); // tell startothers() we're up
1267   scheduler();     // start running processes
1268 }
1269
1270 pde_t entrypgdir[];  // For entry.S
1271
1272 // Start the non-boot (AP) processors.
1273 static void
1274 startothers(void)
1275 {
1276   extern uchar _binary_entryother_start[], _binary_entryother_size[];
1277   uchar *code;
1278   struct cpu *c;
1279   char *stack;
1280
1281   // Write entry code to unused memory at 0x7000.
1282   // The linker has placed the image of entryother.S in
1283   // _binary_entryother_start.
1284   code = p2v(0x7000);
1285   memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
1286
1287   for(c = cpus; c < cpus+ncpu; c++){
1288     if(c == cpus+cpunum())  // We've started already.
1289       continue;
1290
1291     // Tell entryother.S what stack to use, where to enter, and what
1292     // pgdir to use. We cannot use kpgdir yet, because the AP processor
1293     // is running in low  memory, so we use entrypgdir for the APs too.
1294     stack = kalloc();
1295     *(void**)(code-4) = stack + KSTACKSIZE;
1296     *(void**)(code-8) = mpenter;
1297     *(int**)(code-12) = (void *) v2p(entrypgdir);
1298
1299     lapicstartap(c->id, v2p(code));
```

```
1300     // wait for cpu to finish mpmain()
1301     while(c->started == 0)
1302       ;
1303   }
1304 }
1305
1306 // Boot page table used in entry.S and entryother.S.
1307 // Page directories (and page tables), must start on a page boundary,
1308 // hence the "__aligned__" attribute.
1309 // Use PTE_PS in page directory entry to enable 4Mbyte pages.
1310 __attribute__((__aligned__(PGSIZE)))
1311 pde_t entrypgdir[NPDENTRIES] = {
1312   // Map VA's [0, 4MB) to PA's [0, 4MB)
1313   [0] = (0) | PTE_P | PTE_W | PTE_PS,
1314   // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1315   [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1316 };
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
```



```
1350 // Blank page.
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
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
```

```
1400 // Blank page.
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
```



```
1450 // Blank page.
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
```

```
1500 // Mutual exclusion lock.
1501 struct spinlock {
1502   uint locked;       // Is the lock held?
1503
1504   // For debugging:
1505   char *name;        // Name of lock.
1506   struct cpu *cpu;   // The cpu holding the lock.
1507   uint pcs[10];      // The call stack (an array of program counters)
1508                      // that locked the lock.
1509 };
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
```



```
1550 // Mutual exclusion spin locks.
1551
1552 #include "types.h"
1553 #include "defs.h"
1554 #include "param.h"
1555 #include "x86.h"
1556 #include "memlayout.h"
1557 #include "mmu.h"
1558 #include "proc.h"
1559 #include "spinlock.h"
1560
1561 void
1562 initlock(struct spinlock *lk, char *name)
1563 {
1564   lk->name = name;
1565   lk->locked = 0;
1566   lk->cpu = 0;
1567 }
1568
1569 // Acquire the lock.
1570 // Loops (spins) until the lock is acquired.
1571 // Holding a lock for a long time may cause
1572 // other CPUs to waste time spinning to acquire it.
1573 void
1574 acquire(struct spinlock *lk)
1575 {
1576   pushcli(); // disable interrupts to avoid deadlock.
1577   if(holding(lk))
1578     panic("acquire");
1579
1580   // The xchg is atomic.
1581   // It also serializes, so that reads after acquire are not
1582   // reordered before it.
1583   while(xchg(&lk->locked, 1) != 0)
1584     ;
1585
1586   // Record info about lock acquisition for debugging.
1587   lk->cpu = cpu;
1588   getcallerpcs(&lk, lk->pcs);
1589 }
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
```

```
1600 // Release the lock.
1601 void
1602 release(struct spinlock *lk)
1603 {
1604   if(!holding(lk))
1605     panic("release");
1606
1607   lk->pcs[0] = 0;
1608   lk->cpu = 0;
1609
1610   // The xchg serializes, so that reads before release are
1611   // not reordered after it.  The 1996 PentiumPro manual (Volume 3,
1612   // 7.2) says reads can be carried out speculatively and in
1613   // any order, which implies we need to serialize here.
1614   // But the 2007 Intel 64 Architecture Memory Ordering White
1615   // Paper says that Intel 64 and IA-32 will not move a load
1616   // after a store. So lock->locked = 0 would work here.
1617   // The xchg being asm volatile ensures gcc emits it after
1618   // the above assignments (and after the critical section).
1619   xchg(&lk->locked, 0);
1620
1621   popcli();
1622 }
1623
1624 // Record the current call stack in pcs[] by following the %ebp chain.
1625 void
1626 getcallerpcs(void *v, uint pcs[])
1627 {
1628   uint *ebp;
1629   int i;
1630
1631   ebp = (uint*)v - 2;
1632   for(i = 0; i < 10; i++){
1633     if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
1634       break;
1635     pcs[i] = ebp[1];     // saved %eip
1636     ebp = (uint*)ebp[0]; // saved %ebp
1637   }
1638   for(; i < 10; i++)
1639     pcs[i] = 0;
1640 }
1641
1642 // Check whether this cpu is holding the lock.
1643 int
1644 holding(struct spinlock *lock)
1645 {
1646   return lock->locked && lock->cpu == cpu;
1647 }
1648
1649
```



```
1650 // Pushcli/popcli are like cli/sti except that they are matched:
1651 // it takes two popcli to undo two pushcli.  Also, if interrupts
1652 // are off, then pushcli, popcli leaves them off.
1653
1654 void
1655 pushcli(void)
1656 {
1657   int eflags;
1658
1659   eflags = readeflags();
1660   cli();
1661   if(cpu->ncli++ == 0)
1662     cpu->intena = eflags & FL_IF;
1663 }
1664
1665 void
1666 popcli(void)
1667 {
1668   if(readeflags()&FL_IF)
1669     panic("popcli - interruptible");
1670   if(--cpu->ncli < 0)
1671     panic("popcli");
1672   if(cpu->ncli == 0 && cpu->intena)
1673     sti();
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 "param.h"
1701 #include "types.h"
1702 #include "defs.h"
1703 #include "x86.h"
1704 #include "memlayout.h"
1705 #include "mmu.h"
1706 #include "proc.h"
1707 #include "elf.h"
1708
1709 extern char data[];  // defined by kernel.ld
1710 pde_t *kpgdir;  // for use in scheduler()
1711 struct segdesc gdt[NSEGS];
1712
1713 // Set up CPU's kernel segment descriptors.
1714 // Run once on entry on each CPU.
1715 void
1716 seginit(void)
1717 {
1718   struct cpu *c;
1719
1720   // Map "logical" addresses to virtual addresses using identity map.
1721   // Cannot share a CODE descriptor for both kernel and user
1722   // because it would have to have DPL_USR, but the CPU forbids
1723   // an interrupt from CPL=0 to DPL=3.
1724   c = &cpus[cpunum()];
1725   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
1726   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1727   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
1728   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1729
1730   // Map cpu, and curproc
1731   c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
1732
1733   lgdt(c->gdt, sizeof(c->gdt));
1734   loadgs(SEG_KCPU << 3);
1735
1736   // Initialize cpu-local storage.
1737   cpu = c;
1738   proc = 0;
1739 }
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
```



```
1750 // Return the address of the PTE in page table pgdir
1751 // that corresponds to virtual address va.  If alloc!=0,
1752 // create any required page table pages.
1753 static pte_t *
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756   pde_t *pde;
1757   pte_t *pgtab;
1758
1759   pde = &pgdir[PDX(va)];
1760   if(*pde & PTE_P){
1761     pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
1762   } else {
1763     if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1764       return 0;
1765     // Make sure all those PTE_P bits are zero.
1766     memset(pgtab, 0, PGSIZE);
1767     // The permissions here are overly generous, but they can
1768     // be further restricted by the permissions in the page table
1769     // entries, if necessary.
1770     *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
1771   }
1772   return &pgtab[PTX(va)];
1773 }
1774
1775 // Create PTEs for virtual addresses starting at va that refer to
1776 // physical addresses starting at pa. va and size might not
1777 // be page-aligned.
1778 static int
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return -1;
1789     if(*pte & PTE_P)
1790       panic("remap");
1791     *pte = pa | perm | PTE_P;
1792     if(a == last)
1793       break;
1794     a += PGSIZE;
1795     pa += PGSIZE;
1796   }
1797   return 0;
1798 }
1799
```

```
1800 // There is one page table per process, plus one that's used when
1801 // a CPU is not running any process (kpgdir). The kernel uses the
1802 // current process's page table during system calls and interrupts;
1803 // page protection bits prevent user code from using the kernel's
1804 // mappings.
1805 //
1806 // setupkvm() and exec() set up every page table like this:
1807 //
1808 //   0..KERNBASE: user memory (text+data+stack+heap), mapped to
1809 //                phys memory allocated by the kernel
1810 //   KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
1811 //   KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
1812 //                for the kernel's instructions and r/o data
1813 //   data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
1814 //                                  rw data + free physical memory
1815 //   0xfe000000..0: mapped direct (devices such as ioapic)
1816 //
1817 // The kernel allocates physical memory for its heap and for user memory
1818 // between V2P(end) and the end of physical memory (PHYSTOP)
1819 // (directly addressable from end..P2V(PHYSTOP)).
1820
1821 // This table defines the kernel's mappings, which are present in
1822 // every process's page table.
1823 static struct kmap {
1824   void *virt;
1825   uint phys_start;
1826   uint phys_end;
1827   int perm;
1828 } kmap[] = {
1829  { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
1830  { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
1831  { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
1832  { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
1833 };
1834
1835 // Set up kernel part of a page table.
1836 pde_t*
1837 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
1845   if (p2v(PHYSTOP) > (void*)DEVSPACE)
1846     panic("PHYSTOP too high");
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                 (uint)k->phys_start, k->perm) < 0)
```



```
1850       return 0;
1851   return pgdir;
1852 }
1853
1854 // Allocate one page table for the machine for the kernel address
1855 // space for scheduler processes.
1856 void
1857 kvmalloc(void)
1858 {
1859   kpgdir = setupkvm();
1860   switchkvm();
1861 }
1862
1863 // Switch h/w page table register to the kernel-only page table,
1864 // for when no process is running.
1865 void
1866 switchkvm(void)
1867 {
1868   lcr3(v2p(kpgdir));   // switch to the kernel page table
1869 }
1870
1871 // Switch TSS and h/w page table to correspond to process p.
1872 void
1873 switchuvm(struct proc *p)
1874 {
1875   pushcli();
1876   cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
1877   cpu->gdt[SEG_TSS].s = 0;
1878   cpu->ts.ss0 = SEG_KDATA << 3;
1879   cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
1880   ltr(SEG_TSS << 3);
1881   if(p->pgdir == 0)
1882     panic("switchuvm: no pgdir");
1883   lcr3(v2p(p->pgdir));  // switch to new address space
1884   popcli();
1885 }
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
```

```
1900 // Load the initcode into address 0 of pgdir.
1901 // sz must be less than a page.
1902 void
1903 inituvm(pde_t *pgdir, char *init, uint sz)
1904 {
1905   char *mem;
1906
1907   if(sz >= PGSIZE)
1908     panic("inituvm: more than a page");
1909   mem = kalloc();
1910   memset(mem, 0, PGSIZE);
1911   mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
1912   memmove(mem, init, sz);
1913 }
1914
1915 // Load a program segment into pgdir.  addr must be page-aligned
1916 // and the pages from addr to addr+sz must already be mapped.
1917 int
1918 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
1919 {
1920   uint i, pa, n;
1921   pte_t *pte;
1922
1923   if((uint) addr % PGSIZE != 0)
1924     panic("loaduvm: addr must be page aligned");
1925   for(i = 0; i < sz; i += PGSIZE){
1926     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1927       panic("loaduvm: address should exist");
1928     pa = PTE_ADDR(*pte);
1929     if(sz - i < PGSIZE)
1930       n = sz - i;
1931     else
1932       n = PGSIZE;
1933     if(readi(ip, p2v(pa), offset+i, n) != n)
1934       return -1;
1935   }
1936   return 0;
1937 }
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
```



```
1950 // Allocate page tables and physical memory to grow process from oldsz to
1951 // newsz, which need not be page aligned.  Returns new size or 0 on error.
1952 int
1953 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1954 {
1955   char *mem;
1956   uint a;
1957
1958   if(newsz >= KERNBASE)
1959     return 0;
1960   if(newsz < oldsz)
1961     return oldsz;
1962
1963   a = PGROUNDUP(oldsz);
1964   for(; a < newsz; a += PGSIZE){
1965     mem = kalloc();
1966     if(mem == 0){
1967       cprintf("allocuvm out of memory\n");
1968       deallocuvm(pgdir, newsz, oldsz);
1969       return 0;
1970     }
1971     memset(mem, 0, PGSIZE);
1972     mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U);
1973   }
1974   return newsz;
1975 }
1976
1977 // Deallocate user pages to bring the process size from oldsz to
1978 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
1979 // need to be less than oldsz.  oldsz can be larger than the actual
1980 // process size.  Returns the new process size.
1981 int
1982 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1983 {
1984   pte_t *pte;
1985   uint a, pa;
1986
1987   if(newsz >= oldsz)
1988     return oldsz;
1989
1990   a = PGROUNDUP(newsz);
1991   for(; a  < oldsz; a += PGSIZE){
1992     pte = walkpgdir(pgdir, (char*)a, 0);
1993     if(!pte)
1994       a += (NPTENTRIES - 1) * PGSIZE;
1995     else if((*pte & PTE_P) != 0){
1996       pa = PTE_ADDR(*pte);
1997       if(pa == 0)
1998         panic("kfree");
1999       char *v = p2v(pa);
```

```
2000       kfree(v);
2001       *pte = 0;
2002     }
2003   }
2004   return newsz;
2005 }
2006
2007 // Free a page table and all the physical memory pages
2008 // in the user part.
2009 void
2010 freevm(pde_t *pgdir)
2011 {
2012   uint i;
2013
2014   if(pgdir == 0)
2015     panic("freevm: no pgdir");
2016   deallocuvm(pgdir, KERNBASE, 0);
2017   for(i = 0; i < NPDENTRIES; i++){
2018     if(pgdir[i] & PTE_P){
2019       char * v = p2v(PTE_ADDR(pgdir[i]));
2020       kfree(v);
2021     }
2022   }
2023   kfree((char*)pgdir);
2024 }
2025
2026 // Clear PTE_U on a page. Used to create an inaccessible
2027 // page beneath the user stack.
2028 void
2029 clearpteu(pde_t *pgdir, char *uva)
2030 {
2031   pte_t *pte;
2032
2033   pte = walkpgdir(pgdir, uva, 0);
2034   if(pte == 0)
2035     panic("clearpteu");
2036   *pte &= ~PTE_U;
2037 }
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
```



```
2050 // Given a parent process's page table, create a copy
2051 // of it for a child.
2052 pde_t*
2053 copyuvm(pde_t *pgdir, uint sz)
2054 {
2055   pde_t *d;
2056   pte_t *pte;
2057   uint pa, i, flags;
2058   char *mem;
2059
2060   if((d = setupkvm()) == 0)
2061     return 0;
2062   for(i = 0; i < sz; i += PGSIZE){
2063     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
2064       panic("copyuvm: pte should exist");
2065     if(!(*pte & PTE_P))
2066       panic("copyuvm: page not present");
2067     pa = PTE_ADDR(*pte);
2068     flags = PTE_FLAGS(*pte);
2069     if((mem = kalloc()) == 0)
2070       goto bad;
2071     memmove(mem, (char*)p2v(pa), PGSIZE);
2072     if(mappages(d, (void*)i, PGSIZE, v2p(mem), flags) < 0)
2073       goto bad;
2074   }
2075   return d;
2076
2077 bad:
2078   freevm(d);
2079   return 0;
2080 }
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
```

```
2100 // Map user virtual address to kernel address.
2101 char*
2102 uva2ka(pde_t *pgdir, char *uva)
2103 {
2104   pte_t *pte;
2105
2106   pte = walkpgdir(pgdir, uva, 0);
2107   if((*pte & PTE_P) == 0)
2108     return 0;
2109   if((*pte & PTE_U) == 0)
2110     return 0;
2111   return (char*)p2v(PTE_ADDR(*pte));
2112 }
2113
2114 // Copy len bytes from p to user address va in page table pgdir.
2115 // Most useful when pgdir is not the current page table.
2116 // uva2ka ensures this only works for PTE_U pages.
2117 int
2118 copyout(pde_t *pgdir, uint va, void *p, uint len)
2119 {
2120   char *buf, *pa0;
2121   uint n, va0;
2122
2123   buf = (char*)p;
2124   while(len > 0){
2125     va0 = (uint)PGROUNDDOWN(va);
2126     pa0 = uva2ka(pgdir, (char*)va0);
2127     if(pa0 == 0)
2128       return -1;
2129     n = PGSIZE - (va - va0);
2130     if(n > len)
2131       n = len;
2132     memmove(pa0 + (va - va0), buf, n);
2133     len -= n;
2134     buf += n;
2135     va = va0 + PGSIZE;
2136   }
2137   return 0;
2138 }
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
```



```
2150 // Blank page.
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
```

```
2200 // Blank page.
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
```



```
2250 // Blank page.
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
```

```
2300 // Segments in proc->gdt.
2301 #define NSEGS       7
2302
2303 // Project 3
2304 #define DEFAULT_uid 0 // default uid
2305 #define DEFAULT_gid 0 // default gid
2306
2307 // Project 4
2308 #define LOW_priority     0  // lowest priority
2309 #define DEFAULT_priority 1  // default priority
2310 #define HIGH_priority    2  // highest priority
2311 #define QUEUE            3  // number of queues
2312
2313 // Per-CPU state
2314 struct cpu {
2315   uchar id;                    // Local APIC ID; index into cpus[] below
2316   struct context *scheduler;   // swtch() here to enter scheduler
2317   struct taskstate ts;         // Used by x86 to find stack for interrupt
2318   struct segdesc gdt[NSEGS];   // x86 global descriptor table
2319   volatile uint started;       // Has the CPU started?
2320   int ncli;                    // Depth of pushcli nesting.
2321   int intena;                  // Were interrupts enabled before pushcli?
2322
2323   // Cpu-local storage variables; see below
2324   struct cpu *cpu;
2325   struct proc *proc;           // The currently-running process.
2326 };
2327
2328 extern struct cpu cpus[NCPU];
2329 extern int ncpu;
2330
2331 // Per-CPU variables, holding pointers to the
2332 // current cpu and to the current process.
2333 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2334 // and "%gs:4" to refer to proc.  seginit sets up the
2335 // %gs segment register so that %gs refers to the memory
2336 // holding those two variables in the local cpu's struct cpu.
2337 // This is similar to how thread-local variables are implemented
2338 // in thread libraries such as Linux pthreads.
2339 extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
2340 extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc
2341
2342
2343
2344
2345
2346
2347
2348
2349
```



```
2350 // Saved registers for kernel context switches.
2351 // Don't need to save all the segment registers (%cs, etc),
2352 // because they are constant across kernel contexts.
2353 // Don't need to save %eax, %ecx, %edx, because the
2354 // x86 convention is that the caller has saved them.
2355 // Contexts are stored at the bottom of the stack they
2356 // describe; the stack pointer is the address of the context.
2357 // The layout of the context matches the layout of the stack in swtch.S
2358 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2359 // but it is on the stack and allocproc() manipulates it.
2360 struct context {
2361   uint edi;
2362   uint esi;
2363   uint ebx;
2364   uint ebp;
2365   uint eip;
2366 };
2367
2368 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2369
2370 // Per-process state
2371 struct proc {
2372   uint sz;                     // Size of process memory (bytes)
2373   pde_t* pgdir;                // Page table
2374   char *kstack;                // Bottom of kernel stack for this process
2375   enum procstate state;        // Process state
2376   int pid;                     // Process ID
2377   struct proc *parent;         // Parent process
2378   struct trapframe *tf;        // Trap frame for current syscall
2379   struct context *context;     // swtch() here to run process
2380   void *chan;                  // If non-zero, sleeping on chan
2381   int killed;                  // If non-zero, have been killed
2382   struct file *ofile[NOFILE];  // Open files
2383   struct inode *cwd;           // Current directory
2384   char name[16];               // Process name (debugging)
2385
2386   // Project 3
2387   int uid;                     // User ID
2388   int gid;                     // Group ID
2389
2390   // Project 4
2391   struct proc *next;           //Points to the next item in the ready or fre
2392   int priority;                //Priority for the queue ranging from 0-2 (0
2393 };
2394
2395 // Process memory is laid out contiguously, low addresses first:
2396 //   text
2397 //   original data and bss
2398 //   fixed-size stack
2399 //   expandable heap
```

```
2400 #include "types.h"
2401 #include "defs.h"
2402 #include "param.h"
2403 #include "memlayout.h"
2404 #include "mmu.h"
2405 #include "x86.h"
2406 #include "proc.h"
2407 #include "spinlock.h"
2408 #include "uproc.h"
2409
2410 // Project 4
2411 #define PROJECT4
2412 #ifdef  PROJECT4
2413 int CountToReset = 1000; //any input
2414 #endif
2415
2416 struct {
2417   struct spinlock lock;
2418   struct proc proc[NPROC];
2419
2420
2421 #ifdef PROJECT4
2422   // Project 4
2423   struct proc *pReadyList[QUEUE]; // Points to first process in ready list
2424                                   // Processes are RUNNABLE state (able to b
2425
2426   struct proc *pFreeList;         // Points to first process in free list
2427                                   // Processes are UNUSED state (can create
2428   uint TimeToReset;               // Count to decrement before reset
2429 #endif
2430
2431 } ptable;
2432
2433 static struct proc *initproc;
2434
2435 int nextpid = 1;
2436 extern void forkret(void);
2437 extern void trapret(void);
2438
2439 static void wakeup1(void *chan);
2440
2441 void
2442 pinit(void)
2443 {
2444   initlock(&ptable.lock, "ptable");
2445 }
2446
2447
2448
2449
```



```
2450 // Look in the process table for an UNUSED proc.
2451 // If found, change state to EMBRYO and initialize
2452 // state required to run in the kernel.
2453 // Otherwise return 0.
2454 static struct proc*
2455 allocproc(void)
2456 {
2457   struct proc *p;
2458   char *sp;
2459
2460   acquire(&ptable.lock);
2461   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2462   {
2463     if(p->state == UNUSED)
2464     {
2465       #ifdef PROJECT4
2466       insert(p);
2467       #endif
2468       goto found;
2469     }
2470   }
2471   release(&ptable.lock);
2472   return 0;
2473
2474 found:
2475   p->state = EMBRYO;
2476   p->pid = nextpid++;
2477   release(&ptable.lock);
2478
2479   // Allocate kernel stack.
2480   if((p->kstack = kalloc()) == 0){
2481     p->state = UNUSED;
2482     #ifdef PROJECT4
2483     insert(p);
2484     #endif
2485     return 0;
2486   }
2487   sp = p->kstack + KSTACKSIZE;
2488
2489   // Leave room for trap frame.
2490   sp -= sizeof *p->tf;
2491   p->tf = (struct trapframe*)sp;
2492
2493   // Set up new context to start executing at forkret,
2494   // which returns to trapret.
2495   sp -= 4;
2496   *(uint*)sp = (uint)trapret;
2497
2498
2499
```

```
2500   sp -= sizeof *p->context;
2501   p->context = (struct context*)sp;
2502   memset(p->context, 0, sizeof *p->context);
2503   p->context->eip = (uint)forkret;
2504
2505   return p;
2506 }
2507
2508
2509 // Set up first user process.
2510
2511 void
2512 userinit(void)
2513 {
2514   struct proc *p;
2515
2516 // Project 4
2517 #ifdef PROJECT4
2518   for(p = ptable.proc; p<&ptable.proc[NPROC]; p++)
2519   {
2520     if(!ptable.pFreeList)   // Initialize free list before allocproc
2521       ptable.pFreeList = p; // Which removes processes from free list and al
2522     else
2523     {
2524       p->next = ptable.pFreeList;
2525       ptable.pFreeList = p;
2526     }
2527   }
2528 #endif
2529
2530   extern char _binary_initcode_start[], _binary_initcode_size[];
2531
2532   p = allocproc();
2533   initproc = p;
2534   if((p->pgdir = setupkvm()) == 0)
2535     panic("userinit: out of memory?");
2536   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2537   p->sz = PGSIZE;
2538   memset(p->tf, 0, sizeof(*p->tf));
2539   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2540   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2541   p->tf->es = p->tf->ds;
2542   p->tf->ss = p->tf->ds;
2543   p->tf->eflags = FL_IF;
2544   p->tf->esp = PGSIZE;
2545   p->tf->eip = 0;  // beginning of initcode.S
2546
2547   safestrcpy(p->name, "initcode", sizeof(p->name));
2548   p->cwd = namei("/");
2549
```



```
2550   p->state = RUNNABLE;
2551
2552   // Project 3
2553   p->uid = DEFAULT_uid;   // Set default uid for first process
2554   p->gid = DEFAULT_gid;   // Set default gid for first process
2555
2556
2557
2558 #ifdef PROJECT4
2559   // Project 4
2560   ptable.pReadyList[DEFAULT_priority] = p;  // Set first process as ready li
2561   p->priority = DEFAULT_priority;           // Set priority to default (1);
2562   p->next = 0;                              // Set first process next pointe
2563
2564   ptable.TimeToReset = CountToReset;
2565 #endif
2566 }
2567
2568 // Grow current process's memory by n bytes.
2569 // Return 0 on success, -1 on failure.
2570 int
2571 growproc(int n)
2572 {
2573   uint sz;
2574
2575   sz = proc->sz;
2576   if(n > 0){
2577     if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2578       return -1;
2579   } else if(n < 0){
2580     if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2581       return -1;
2582   }
2583   proc->sz = sz;
2584   switchuvm(proc);
2585   return 0;
2586 }
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
```

```
2600 // Create a new process copying p as the parent.
2601 // Sets up stack to return as if from system call.
2602 // Caller must set state of returned proc to RUNNABLE.
2603 int
2604 fork(void)
2605 {
2606   int i, pid;
2607   struct proc *np;
2608
2609   // Allocate process.
2610   if((np = allocproc()) == 0)
2611     return -1;
2612
2613   // Copy process state from p.
2614   if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2615     kfree(np->kstack);
2616     np->kstack = 0;
2617     np->state = UNUSED;
2618     #ifdef PROJECT4
2619     insert(np);
2620     #endif
2621     return -1;
2622   }
2623   np->sz = proc->sz;
2624   np->parent = proc;
2625   *np->tf = *proc->tf;
2626
2627   // Clear %eax so that fork returns 0 in the child.
2628   np->tf->eax = 0;
2629
2630   for(i = 0; i < NOFILE; i++)
2631     if(proc->ofile[i])
2632       np->ofile[i] = filedup(proc->ofile[i]);
2633   np->cwd = idup(proc->cwd);
2634
2635   safestrcpy(np->name, proc->name, sizeof(proc->name));
2636
2637   pid = np->pid;
2638   np->uid = proc->uid; //Copy uid from parent to child
2639   np->gid = proc->gid; //Copy gid from parent to child
2640
2641   // lock to force the compiler to emit the np->state write last.
2642   acquire(&ptable.lock);
2643   np->state = RUNNABLE;
2644
2645 #ifdef PROJECT4
2646   // Project 4
2647   np->priority = DEFAULT_priority;
2648   enqueue(np, DEFAULT_priority);
2649 #endif
```



```
2650   release(&ptable.lock);
2651
2652   return pid;
2653 }
2654
2655 // Exit the current process.  Does not return.
2656 // An exited process remains in the zombie state
2657 // until its parent calls wait() to find out it exited.
2658 void
2659 exit(void)
2660 {
2661   struct proc *p;
2662   int fd;
2663
2664   if(proc == initproc)
2665     panic("init exiting");
2666
2667   // Close all open files.
2668   for(fd = 0; fd < NOFILE; fd++){
2669     if(proc->ofile[fd]){
2670       fileclose(proc->ofile[fd]);
2671       proc->ofile[fd] = 0;
2672     }
2673   }
2674
2675   begin_op();
2676   iput(proc->cwd);
2677   end_op();
2678   proc->cwd = 0;
2679
2680   acquire(&ptable.lock);
2681
2682   // Parent might be sleeping in wait().
2683   wakeup1(proc->parent);
2684
2685   // Pass abandoned children to init.
2686   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2687     if(p->parent == proc){
2688       p->parent = initproc;
2689       if(p->state == ZOMBIE)
2690         wakeup1(initproc);
2691     }
2692   }
2693
2694   // Jump into the scheduler, never to return.
2695   proc->state = ZOMBIE;
2696   sched();
2697   panic("zombie exit");
2698 }
2699
```

```
2700 // Wait for a child process to exit and return its pid.
2701 // Return -1 if this process has no children.
2702 int
2703 wait(void)
2704 {
2705   struct proc *p;
2706   int havekids, pid;
2707
2708   acquire(&ptable.lock);
2709   for(;;){
2710     // Scan through table looking for zombie children.
2711     havekids = 0;
2712     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2713       if(p->parent != proc)
2714         continue;
2715       havekids = 1;
2716       if(p->state == ZOMBIE){
2717         // Found one.
2718         pid = p->pid;
2719         kfree(p->kstack);
2720         p->kstack = 0;
2721         freevm(p->pgdir);
2722         p->state = UNUSED;
2723         #ifdef PROJECT4
2724         insert(p);
2725         #endif
2726         p->pid = 0;
2727         p->parent = 0;
2728         p->name[0] = 0;
2729         p->killed = 0;
2730         release(&ptable.lock);
2731         return pid;
2732       }
2733     }
2734
2735     // No point waiting if we don't have any children.
2736     if(!havekids || proc->killed){
2737       release(&ptable.lock);
2738       return -1;
2739     }
2740
2741     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2742     sleep(proc, &ptable.lock);
2743   }
2744 }
2745
2746
2747
2748
2749
```



```
2750 // Per-CPU process scheduler.
2751 // Each CPU calls scheduler() after setting itself up.
2752 // Scheduler never returns.  It loops, doing:
2753 //  - choose a process to run
2754 //  - swtch to start running that process
2755 //  - eventually that process transfers control
2756 //      via swtch back to the scheduler.
2757
2758
2759 #ifdef PROJECT4
2760 void
2761 scheduler(void)
2762 {
2763   struct proc *p;
2764
2765   for(;;)
2766   {
2767     // Enable interrupts on this processor.
2768     sti();
2769
2770     // Loop over process table looking for process to run.
2771     acquire(&ptable.lock);
2772
2773     if(ptable.pReadyList[HIGH_priority] || ptable.pReadyList[DEFAULT_priorit
2774     {
2775       if(ptable.pReadyList[HIGH_priority])
2776       {
2777         p = ptable.pReadyList[HIGH_priority];
2778         ptable.pReadyList[HIGH_priority] = ptable.pReadyList[HIGH_priority]-
2779       }
2780       else if(ptable.pReadyList[DEFAULT_priority])
2781       {
2782         p = ptable.pReadyList[DEFAULT_priority];
2783         ptable.pReadyList[DEFAULT_priority] = ptable.pReadyList[DEFAULT_prio
2784       }
2785       else if(ptable.pReadyList[LOW_priority])
2786       {
2787         p = ptable.pReadyList[LOW_priority];
2788         ptable.pReadyList[LOW_priority] = ptable.pReadyList[LOW_priority]->n
2789       }
2790
2791       // Switch to chosen process.  It is the process's job
2792       // to release ptable.lock and then reacquire it
2793       // before jumping back to us.
2794       proc = p;
2795       switchuvm(p);
2796       p->state = RUNNING;
2797       swtch(&cpu->scheduler, proc->context);
2798       switchkvm();
2799
```

```
2800       // Process is done running for now.
2801       // It should have changed its p->state before coming back.
2802       proc = 0;
2803       ptable.TimeToReset--;
2804       if(!ptable.TimeToReset)
2805         reset();
2806     }
2807     release(&ptable.lock);
2808   }
2809 }
2810
2811 #else
2812 void
2813 scheduler(void)
2814 {
2815   struct proc *p;
2816
2817   for(;;){
2818     // Enable interrupts on this processor.
2819     sti();
2820
2821     // Loop over process table looking for process to run.
2822     acquire(&ptable.lock);
2823     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2824       if(p->state != RUNNABLE)
2825         continue;
2826
2827       // Switch to chosen process.  It is the process's job
2828       // to release ptable.lock and then reacquire it
2829       // before jumping back to us.
2830       proc = p;
2831       switchuvm(p);
2832       p->state = RUNNING;
2833       swtch(&cpu->scheduler, proc->context);
2834       switchkvm();
2835
2836       // Process is done running for now.
2837       // It should have changed its p->state before coming back.
2838       proc = 0;
2839     }
2840     release(&ptable.lock);
2841
2842   }
2843 }
2844
2845
2846
2847
2848
2849
```



```
2850 #endif
2851
2852 // Enter scheduler.  Must hold only ptable.lock
2853 // and have changed proc->state.
2854 void
2855 sched(void)
2856 {
2857   int intena;
2858
2859   if(!holding(&ptable.lock))
2860     panic("sched ptable.lock");
2861   if(cpu->ncli != 1)
2862     panic("sched locks");
2863   if(proc->state == RUNNING)
2864     panic("sched running");
2865   if(readeflags()&FL_IF)
2866     panic("sched interruptible");
2867   intena = cpu->intena;
2868   swtch(&proc->context, cpu->scheduler);
2869   cpu->intena = intena;
2870 }
2871 // Give up the CPU for one scheduling round.
2872 void
2873 yield(void)
2874 {
2875   acquire(&ptable.lock);
2876   proc->state = RUNNABLE;
2877
2878 #ifdef PROJECT4
2879   enqueue(proc, proc->priority);
2880 #endif
2881
2882   sched();
2883   release(&ptable.lock);
2884 }
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
```

```
2900 // A fork child's very first scheduling by scheduler()
2901 // will swtch here.  "Return" to user space.
2902 void
2903 forkret(void)
2904 {
2905   static int first = 1;
2906   // Still holding ptable.lock from scheduler.
2907   release(&ptable.lock);
2908
2909   if (first) {
2910     // Some initialization functions must be run in the context
2911     // of a regular process (e.g., they call sleep), and thus cannot
2912     // be run from main().
2913     first = 0;
2914     iinit(ROOTDEV);
2915     initlog(ROOTDEV);
2916   }
2917
2918   // Return to "caller", actually trapret (see allocproc).
2919 }
2920
2921 // Atomically release lock and sleep on chan.
2922 // Reacquires lock when awakened.
2923 void
2924 sleep(void *chan, struct spinlock *lk)
2925 {
2926   if(proc == 0)
2927     panic("sleep");
2928
2929   if(lk == 0)
2930     panic("sleep without lk");
2931
2932   // Must acquire ptable.lock in order to
2933   // change p->state and then call sched.
2934   // Once we hold ptable.lock, we can be
2935   // guaranteed that we won't miss any wakeup
2936   // (wakeup runs with ptable.lock locked),
2937   // so it's okay to release lk.
2938   if(lk != &ptable.lock){
2939     acquire(&ptable.lock);
2940     release(lk);
2941   }
2942
2943   // Go to sleep.
2944   proc->chan = chan;
2945   proc->state = SLEEPING;
2946   sched();
2947
2948   // Tidy up.
2949   proc->chan = 0;
```



```
2950   // Reacquire original lock.
2951   if(lk != &ptable.lock){
2952     release(&ptable.lock);
2953     acquire(lk);
2954   }
2955 }
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
```

```
3000 // Wake up all processes sleeping on chan.
3001 // The ptable lock must be held.
3002 static void
3003 wakeup1(void *chan)
3004 {
3005   struct proc *p;
3006
3007   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
3008     if(p->state == SLEEPING && p->chan == chan)
3009     {
3010       p->state = RUNNABLE;
3011       #ifdef PROJECT4
3012       enqueue(p, p->priority);
3013       #endif
3014     }
3015 }
3016
3017 // Wake up all processes sleeping on chan.
3018 void
3019 wakeup(void *chan)
3020 {
3021   acquire(&ptable.lock);
3022   wakeup1(chan);
3023   release(&ptable.lock);
3024 }
3025
3026 // Kill the process with the given pid.
3027 // Process won't exit until it returns
3028 // to user space (see trap in trap.c).
3029 int
3030 kill(int pid)
3031 {
3032   struct proc *p;
3033
3034   acquire(&ptable.lock);
3035   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3036     if(p->pid == pid){
3037       p->killed = 1;
3038       // Wake process from sleep if necessary.
3039       if(p->state == SLEEPING)
3040       {
3041         p->state = RUNNABLE;
3042         #ifdef PROJECT4
3043         enqueue(p, p->priority);
3044         #endif
3045       }
3046       release(&ptable.lock);
3047       return 0;
3048     }
3049   }
```



```
3050   release(&ptable.lock);
3051   return -1;
3052 }
3053
3054
3055 // Print a process listing to console.  For debugging.
3056 // Runs when user types ^P on console.
3057 // No lock to avoid wedging a stuck machine further.
3058
3059 #ifdef PROJECT4
3060 void
3061 procdump(void)
3062 {
3063   static char *states[] = {
3064   [UNUSED]    "unused",
3065   [EMBRYO]    "embryo",
3066   [SLEEPING]  "sleep ",
3067   [RUNNABLE]  "runble",
3068   [RUNNING]   "run   ",
3069   [ZOMBIE]    "zombie"
3070   };
3071
3072   struct proc *p;
3073   char *state;
3074   int i;
3075
3076   // Project 4 Test Header
3077   cprintf("\n************************PROCESS LIST*************************"
3078   cprintf("\nPID    UID    GID    PPID    STATE    SIZE    PRIORITY    NAME\n
3079
3080   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
3081   {
3082     if(p->state == UNUSED)
3083       continue;
3084     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
3085       state = states[p->state];
3086     else
3087       state = "???";
3088
3089     // Project 4 Test
3090     if(p->parent)
3091       cprintf("%d      %d      %d      %d       %s   %d   %d           %s",
3092       p->pid, p->uid, p->gid, p->parent->pid, state, p->sz, p->priority, p->
3093     else
3094       cprintf("%d      %d      %d      %d       %s   %d   %d           %s",
3095       p->pid, p->uid, p->gid, 0, state, p->sz, p->priority, p->name);
3096
3097     cprintf("\n");
3098   }
3099
```

```
3100   // Project 4 Test: Print Ready List
3101   cprintf("\n************************READY LIST**************************"
3102   cprintf("\nPID    UID    GID    PPID    STATE    SIZE    PRIORITY    NAME\n
3103   for(i = 0; i < QUEUE; i++)
3104   {
3105     for(p = ptable.pReadyList[i]; p; p = p->next)
3106     {
3107       if(p->parent)
3108         cprintf("%d      %d      %d      %d       %s   %d   %d           %s\n
3109         p->pid, p->uid, p->gid, p->parent->pid, state, p->sz, p->priority, p->
3110       else
3111         cprintf("%d      %d      %d      %d       %s   %d   %d           %s\n
3112         p->pid, p->uid, p->gid, 0, state, p->sz, p->priority, p->name);
3113     }
3114   }
3115 }
3116 #else
3117 void
3118 procdump(void)
3119 {
3120   static char *states[] = {
3121   [UNUSED]    "unused",
3122   [EMBRYO]    "embryo",
3123   [SLEEPING]  "sleep ",
3124   [RUNNABLE]  "runble",
3125   [RUNNING]   "run   ",
3126   [ZOMBIE]    "zombie"
3127   };
3128   int i;
3129   struct proc *p;
3130   char *state;
3131   uint pc[10];
3132
3133   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3134     if(p->state == UNUSED)
3135       continue;
3136     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
3137       state = states[p->state];
3138     else
3139       state = "???";
3140
3141     // Project 3
3142     cprintf("\nPID: %d  UID: %d  GID: %d  STATE: %s  NAME: %s    ",
3143     p->pid, p->uid, p->gid, state, p->name);
3144
3145     if(p->state == SLEEPING){
3146       getcallerpcs((uint*)p->context->ebp+2, pc);
3147       for(i=0; i<10 && pc[i] != 0; i++)
3148         cprintf(" %p", pc[i]);
3149     }
```



```
3150     cprintf("\n");
3151   }
3152 }
3153 #endif
3154 // Project 3
3155 int
3156 copy(int MAX, struct uproc *table)
3157 {
3158   acquire(&ptable.lock);
3159   int i;
3160
3161   static char *states[] = {
3162   [UNUSED]    "unused",
3163   [EMBRYO]    "embryo",
3164   [SLEEPING]  "sleep ",
3165   [RUNNABLE]  "runble",
3166   [RUNNING]   "run   ",
3167   [ZOMBIE]    "zombie"
3168   };
3169
3170   for(i = 0; ptable.proc[i].state != UNUSED && ptable.proc[i].state != EMBRYO
3171   {
3172     table[i].pid      = ptable.proc[i].pid;
3173     table[i].uid      = ptable.proc[i].uid;
3174     table[i].gid      = ptable.proc[i].gid;
3175     #ifdef PROJECT4
3176     table[i].priority = ptable.proc[i].priority;
3177     #endif
3178     if(i)
3179       table[i].ppid = ptable.proc[i].parent->pid;
3180     else
3181       table[i].ppid = 0;
3182     table[i].size = ptable.proc[i].sz;
3183     safestrcpy(table[i].STATE, states[ptable.proc[i].state], strlen(states[pt
3184     safestrcpy(table[i].name, ptable.proc[i].name, strlen(ptable.proc[i].name
3185   }
3186   release(&ptable.lock);
3187   return i;
3188 }
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
```

```
3200 #ifdef PROJECT4
3201 // Project 4
3202 int
3203 setPriority(uint pid, uint priority)
3204 {
3205   struct proc *p;
3206
3207   acquire(&ptable.lock);
3208   for(p = ptable.proc; p<&ptable.proc[NPROC]; p++)
3209   {
3210     if(p->pid == pid)
3211     {
3212       if(p->state == RUNNABLE)
3213       {
3214         enqueue(p, priority);
3215       }
3216
3217       p->priority = priority;
3218       release(&ptable.lock);
3219       return 0;
3220     }
3221   }
3222   release(&ptable.lock);
3223   return -1;
3224 }
3225
3226 void
3227 enqueue(struct proc *p, uint priority)
3228 {
3229   if(!holding(&ptable.lock))
3230     panic("enqueue ptable.lock");
3231
3232   struct proc *current;
3233   if(!ptable.pReadyList[priority])
3234     ptable.pReadyList[priority] = p;
3235   else
3236   {
3237     // Traverse to end of list to insert
3238     for(current = ptable.pReadyList[priority]; current->next; current = curr
3239     current->next = p;
3240   }
3241   p->next = 0;
3242 }
3243
3244
3245
3246
3247
3248
3249
```



```
3250 void
3251 reset(void)
3252 {
3253
3254   struct proc *current;
3255
3256   for(current = ptable.proc; current < &ptable.proc[NPROC]; current++)
3257     current->priority = DEFAULT_priority;
3258
3259   if(ptable.pReadyList[HIGH_priority])
3260   {
3261     enqueue(ptable.pReadyList[HIGH_priority], DEFAULT_priority);
3262     ptable.pReadyList[HIGH_priority] = 0;
3263   }
3264
3265   if(ptable.pReadyList[LOW_priority])
3266   {
3267     enqueue(ptable.pReadyList[LOW_priority], DEFAULT_priority);
3268     ptable.pReadyList[LOW_priority] = 0;
3269   }
3270
3271   ptable.TimeToReset = CountToReset;
3272   cprintf("\n\nRESET\n\n");
3273 }
3274
3275
3276 void
3277 insert(struct proc *p)
3278 {
3279   if(!holding(&ptable.lock))
3280     panic("insert ptable.lock");
3281
3282   p->next = ptable.pFreeList;
3283   ptable.pFreeList = p;
3284 }
3285 #endif
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
```

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

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

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

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

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



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

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

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

struct run {
  struct run *next;
};

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

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

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

```
3400 void
3401 freerange(void *vstart, void *vend)
3402 {
3403   char *p;
3404   p = (char*)PGROUNDUP((uint)vstart);
3405   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3406     kfree(p);
3407 }
3408
3409
3410 // Free the page of physical memory pointed at by v,
3411 // which normally should have been returned by a
3412 // call to kalloc().  (The exception is when
3413 // initializing the allocator; see kinit above.)
3414 void
3415 kfree(char *v)
3416 {
3417   struct run *r;
3418
3419   if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
3420     panic("kfree");
3421
3422   // Fill with junk to catch dangling refs.
3423   memset(v, 1, PGSIZE);
3424
3425   if(kmem.use_lock)
3426     acquire(&kmem.lock);
3427   r = (struct run*)v;
3428   r->next = kmem.freelist;
3429   kmem.freelist = r;
3430   if(kmem.use_lock)
3431     release(&kmem.lock);
3432 }
3433
3434 // Allocate one 4096-byte page of physical memory.
3435 // Returns a pointer that the kernel can use.
3436 // Returns 0 if the memory cannot be allocated.
3437 char*
3438 kalloc(void)
3439 {
3440   struct run *r;
3441
3442   if(kmem.use_lock)
3443     acquire(&kmem.lock);
3444   r = kmem.freelist;
3445   if(r)
3446     kmem.freelist = r->next;
3447   if(kmem.use_lock)
3448     release(&kmem.lock);
3449   return (char*)r;
```



```
3450 }
3451
3452
3453
3454
3455
3456
3457
3458
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3479
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
```

```
3500 // x86 trap and interrupt constants.
3501
3502 // Processor-defined:
3503 #define T_DIVIDE         0      // divide error
3504 #define T_DEBUG          1      // debug exception
3505 #define T_NMI            2      // non-maskable interrupt
3506 #define T_BRKPT          3      // breakpoint
3507 #define T_OFLOW          4      // overflow
3508 #define T_BOUND          5      // bounds check
3509 #define T_ILLOP          6      // illegal opcode
3510 #define T_DEVICE         7      // device not available
3511 #define T_DBLFLT         8      // double fault
3512 // #define T_COPROC      9      // reserved (not used since 486)
3513 #define T_TSS           10      // invalid task switch segment
3514 #define T_SEGNP         11      // segment not present
3515 #define T_STACK         12      // stack exception
3516 #define T_GPFLT         13      // general protection fault
3517 #define T_PGFLT         14      // page fault
3518 // #define T_RES        15      // reserved
3519 #define T_FPERR         16      // floating point error
3520 #define T_ALIGN         17      // aligment check
3521 #define T_MCHK          18      // machine check
3522 #define T_SIMDERR       19      // SIMD floating point error
3523
3524 // These are arbitrarily chosen, but with care not to overlap
3525 // processor defined exceptions or interrupt vectors.
3526 #define T_SYSCALL       64      // system call
3527 #define T_DEFAULT      500      // catchall
3528
3529 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
3530
3531 #define IRQ_TIMER        0
3532 #define IRQ_KBD          1
3533 #define IRQ_COM1         4
3534 #define IRQ_IDE         14
3535 #define IRQ_ERROR       19
3536 #define IRQ_SPURIOUS    31
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
```



```
3550 #!/usr/bin/perl -w
3551
3552 # Generate vectors.S, the trap/interrupt entry points.
3553 # There has to be one entry point per interrupt number
3554 # since otherwise there's no way for trap() to discover
3555 # the interrupt number.
3556
3557 print "# generated by vectors.pl - do not edit\n";
3558 print "# handlers\n";
3559 print ".globl alltraps\n";
3560 for(my $i = 0; $i < 256; $i++){
3561     print ".globl vector$i\n";
3562     print "vector$i:\n";
3563     if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
3564         print "  pushl \$0\n";
3565     }
3566     print "  pushl \$$i\n";
3567     print "  jmp alltraps\n";
3568 }
3569
3570 print "\n# vector table\n";
3571 print ".data\n";
3572 print ".globl vectors\n";
3573 print "vectors:\n";
3574 for(my $i = 0; $i < 256; $i++){
3575     print "  .long vector$i\n";
3576 }
3577
3578 # sample output:
3579 #   # handlers
3580 #   .globl alltraps
3581 #   .globl vector0
3582 #   vector0:
3583 #     pushl $0
3584 #     pushl $0
3585 #     jmp alltraps
3586 #   ...
3587 #
3588 #   # vector table
3589 #   .data
3590 #   .globl vectors
3591 #   vectors:
3592 #     .long vector0
3593 #     .long vector1
3594 #     .long vector2
3595 #   ...
3596
3597
3598
3599
```

```
3600 #include "mmu.h"
3601
3602   # vectors.S sends all traps here.
3603 .globl alltraps
3604 alltraps:
3605   # Build trap frame.
3606   pushl %ds
3607   pushl %es
3608   pushl %fs
3609   pushl %gs
3610   pushal
3611
3612   # Set up data and per-cpu segments.
3613   movw $(SEG_KDATA<<3), %ax
3614   movw %ax, %ds
3615   movw %ax, %es
3616   movw $(SEG_KCPU<<3), %ax
3617   movw %ax, %fs
3618   movw %ax, %gs
3619
3620   # Call trap(tf), where tf=%esp
3621   pushl %esp
3622   call trap
3623   addl $4, %esp
3624
3625   # Return falls through to trapret...
3626 .globl trapret
3627 trapret:
3628   popal
3629   popl %gs
3630   popl %fs
3631   popl %es
3632   popl %ds
3633   addl $0x8, %esp  # trapno and errcode
3634   iret
3635
3636
3637
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649
```



```
3650 #include "types.h"
3651 #include "defs.h"
3652 #include "param.h"
3653 #include "memlayout.h"
3654 #include "mmu.h"
3655 #include "proc.h"
3656 #include "x86.h"
3657 #include "traps.h"
3658 #include "spinlock.h"
3659
3660 // Interrupt descriptor table (shared by all CPUs).
3661 struct gatedesc idt[256];
3662 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
3663 struct spinlock tickslock;
3664 uint ticks;
3665
3666 void
3667 tvinit(void)
3668 {
3669   int i;
3670
3671   for(i = 0; i < 256; i++)
3672     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3673   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3674
3675   initlock(&tickslock, "time");
3676 }
3677
3678 void
3679 idtinit(void)
3680 {
3681   lidt(idt, sizeof(idt));
3682 }
3683
3684
3685
3686
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699
```

```
3700 void
3701 trap(struct trapframe *tf)
3702 {
3703   if(tf->trapno == T_SYSCALL){
3704     if(proc->killed)
3705       exit();
3706     proc->tf = tf;
3707     syscall();
3708     if(proc->killed)
3709       exit();
3710     return;
3711   }
3712
3713   switch(tf->trapno){
3714   case T_IRQ0 + IRQ_TIMER:
3715     if(cpu->id == 0){
3716       acquire(&tickslock);
3717       ticks++;
3718       wakeup(&ticks);
3719       release(&tickslock);
3720     }
3721     lapiceoi();
3722     break;
3723   case T_IRQ0 + IRQ_IDE:
3724     ideintr();
3725     lapiceoi();
3726     break;
3727   case T_IRQ0 + IRQ_IDE+1:
3728     // Bochs generates spurious IDE1 interrupts.
3729     break;
3730   case T_IRQ0 + IRQ_KBD:
3731     kbdintr();
3732     lapiceoi();
3733     break;
3734   case T_IRQ0 + IRQ_COM1:
3735     uartintr();
3736     lapiceoi();
3737     break;
3738   case T_IRQ0 + 7:
3739   case T_IRQ0 + IRQ_SPURIOUS:
3740     cprintf("cpu%d: spurious interrupt at %x:%x\n",
3741             cpu->id, tf->cs, tf->eip);
3742     lapiceoi();
3743     break;
3744
3745
3746
3747
3748
3749
```



```
3750   default:
3751     if(proc == 0 || (tf->cs&3) == 0){
3752       // In kernel, it must be our mistake.
3753       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3754               tf->trapno, cpu->id, tf->eip, rcr2());
3755       panic("trap");
3756     }
3757     // In user space, assume process misbehaved.
3758     cprintf("pid %d %s: trap %d err %d on cpu %d "
3759             "eip 0x%x addr 0x%x--kill proc\n",
3760             proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
3761             rcr2());
3762     proc->killed = 1;
3763   }
3764
3765   // Force process exit if it has been killed and is in user space.
3766   // (If it is still executing in the kernel, let it keep running
3767   // until it gets to the regular system call return.)
3768   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3769     exit();
3770
3771   // Force process to give up CPU on clock tick.
3772   // If interrupts were on while locks held, would need to check nlock.
3773   if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3774     yield();
3775
3776   // Check if the process has been killed since we yielded
3777   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3778     exit();
3779 }
3780
3781
3782
3783
3784
3785
3786
3787
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3799
```

```
3800 // System call numbers
3801 #define SYS_fork        1
3802 #define SYS_exit        2
3803 #define SYS_wait        3
3804 #define SYS_pipe        4
3805 #define SYS_read        5
3806 #define SYS_kill        6
3807 #define SYS_exec        7
3808 #define SYS_fstat       8
3809 #define SYS_chdir       9
3810 #define SYS_dup         10
3811 #define SYS_getpid      11
3812 #define SYS_sbrk        12
3813 #define SYS_sleep       13
3814 #define SYS_uptime      14
3815 #define SYS_open        15
3816 #define SYS_write       16
3817 #define SYS_mknod       17
3818 #define SYS_unlink      18
3819 #define SYS_link        19
3820 #define SYS_mkdir       20
3821 #define SYS_close       21
3822 #define SYS_halt        22
3823
3824 // PROJECT 2
3825 #define SYS_date        23
3826
3827 // PROJECT 3
3828 #define SYS_getuid      24
3829 #define SYS_getgid      25
3830 #define SYS_getppid     26
3831 #define SYS_setuid      27
3832 #define SYS_setgid      28
3833 #define SYS_getprocs    29
3834
3835 // PROJECT 4
3836 #define SYS_setPriority 30
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
```



```
3850 #include "types.h"
3851 #include "defs.h"
3852 #include "param.h"
3853 #include "memlayout.h"
3854 #include "mmu.h"
3855 #include "proc.h"
3856 #include "x86.h"
3857 #include "syscall.h"
3858
3859 // User code makes a system call with INT T_SYSCALL.
3860 // System call number in %eax.
3861 // Arguments on the stack, from the user call to the C
3862 // library system call function. The saved user %esp points
3863 // to a saved program counter, and then the first argument.
3864
3865 // Fetch the int at addr from the current process.
3866 int
3867 fetchint(uint addr, int *ip)
3868 {
3869   if(addr >= proc->sz || addr+4 > proc->sz)
3870     return -1;
3871   *ip = *(int*)(addr);
3872   return 0;
3873 }
3874
3875 // Fetch the nul-terminated string at addr from the current process.
3876 // Doesn't actually copy the string - just sets *pp to point at it.
3877 // Returns length of string, not including nul.
3878 int
3879 fetchstr(uint addr, char **pp)
3880 {
3881   char *s, *ep;
3882
3883   if(addr >= proc->sz)
3884     return -1;
3885   *pp = (char*)addr;
3886   ep = (char*)proc->sz;
3887   for(s = *pp; s < ep; s++)
3888     if(*s == 0)
3889       return s - *pp;
3890   return -1;
3891 }
3892
3893 // Fetch the nth 32-bit system call argument.
3894 int
3895 argint(int n, int *ip)
3896 {
3897   return fetchint(proc->tf->esp + 4 + 4*n, ip);
3898 }
3899
```

```
3900 // Fetch the nth word-sized system call argument as a pointer
3901 // to a block of memory of size n bytes.  Check that the pointer
3902 // lies within the process address space.
3903 int
3904 argptr(int n, char **pp, int size)
3905 {
3906   int i;
3907
3908   if(argint(n, &i) < 0)
3909     return -1;
3910   if((uint)i >= proc->sz || (uint)i+size > proc->sz)
3911     return -1;
3912   *pp = (char*)i;
3913   return 0;
3914 }
3915
3916 // Fetch the nth word-sized system call argument as a string pointer.
3917 // Check that the pointer is valid and the string is nul-terminated.
3918 // (There is no shared writable memory, so the string can't change
3919 // between this check and being used by the kernel.)
3920 int
3921 argstr(int n, char **pp)
3922 {
3923   int addr;
3924   if(argint(n, &addr) < 0)
3925     return -1;
3926   return fetchstr(addr, pp);
3927 }
3928
3929 extern int sys_chdir(void);
3930 extern int sys_close(void);
3931 extern int sys_dup(void);
3932 extern int sys_exec(void);
3933 extern int sys_exit(void);
3934 extern int sys_fork(void);
3935 extern int sys_fstat(void);
3936 extern int sys_getpid(void);
3937 extern int sys_kill(void);
3938 extern int sys_link(void);
3939 extern int sys_mkdir(void);
3940 extern int sys_mknod(void);
3941 extern int sys_open(void);
3942 extern int sys_pipe(void);
3943 extern int sys_read(void);
3944 extern int sys_sbrk(void);
3945 extern int sys_sleep(void);
3946 extern int sys_unlink(void);
3947 extern int sys_wait(void);
3948 extern int sys_write(void);
3949 extern int sys_uptime(void);
```



```
3950 extern int sys_halt(void);
3951
3952 // PROJECT 2
3953 extern int sys_date(void);
3954
3955 // PROJECT 3
3956 extern int sys_getuid(void);
3957 extern int sys_getgid(void);
3958 extern int sys_getppid(void);
3959 extern int sys_setuid(void);
3960 extern int sys_setgid(void);
3961 extern int sys_getprocs(void);
3962
3963 // PROJECT 4
3964 extern int sys_setPriority(void);
3965
3966 static int (*syscalls[])(void) = {
3967 [SYS_fork]        sys_fork,
3968 [SYS_exit]        sys_exit,
3969 [SYS_wait]        sys_wait,
3970 [SYS_pipe]        sys_pipe,
3971 [SYS_read]        sys_read,
3972 [SYS_kill]        sys_kill,
3973 [SYS_exec]        sys_exec,
3974 [SYS_fstat]       sys_fstat,
3975 [SYS_chdir]       sys_chdir,
3976 [SYS_dup]         sys_dup,
3977 [SYS_getpid]      sys_getpid,
3978 [SYS_sbrk]        sys_sbrk,
3979 [SYS_sleep]       sys_sleep,
3980 [SYS_uptime]      sys_uptime,
3981 [SYS_open]        sys_open,
3982 [SYS_write]       sys_write,
3983 [SYS_mknod]       sys_mknod,
3984 [SYS_unlink]      sys_unlink,
3985 [SYS_link]        sys_link,
3986 [SYS_mkdir]       sys_mkdir,
3987 [SYS_close]       sys_close,
3988 [SYS_halt]        sys_halt,
3989
3990 [SYS_date]        sys_date,
3991
3992 [SYS_getuid]      sys_getuid,
3993 [SYS_getgid]      sys_getgid,
3994 [SYS_getppid]     sys_getppid,
3995 [SYS_setuid]      sys_setuid,
3996 [SYS_setgid]      sys_setgid,
3997 [SYS_getprocs]    sys_getprocs,
3998
3999
```

```
4000 [SYS_setPriority] sys_setPriority,
4001 };
4002
4003 void
4004 syscall(void)
4005 {
4006 /*
4007     char * syscallnames[] = {
4008     [SYS_fork]    "fork",
4009     [SYS_exit]    "sys_exit",
4010     [SYS_wait]    "sys_wait",
4011     [SYS_pipe]    "sys_pipe",
4012     [SYS_read]    "sys_read",
4013     [SYS_kill]    "sys_kill",
4014     [SYS_exec]    "sys_exec",
4015     [SYS_fstat]   "sys_fstat",
4016     [SYS_chdir]   "sys_chdir",
4017     [SYS_dup]     "sys_dup",
4018     [SYS_getpid]  "sys_getpid",
4019     [SYS_sbrk]    "sys_sbrk",
4020     [SYS_sleep]   "sys_sleep",
4021     [SYS_uptime]  "sys_uptime",
4022     [SYS_open]    "sys_open",
4023     [SYS_write]   "sys_write",
4024     [SYS_mknod]   "sys_mknod",
4025     [SYS_unlink]  "sys_unlink",
4026     [SYS_link]    "sys_link",
4027     [SYS_mkdir]   "sys_mkdir",
4028     [SYS_close]   "sys_close",
4029     [SYS_halt]    "sys_halt",
4030     };
4031 */
4032   int num;
4033   num = proc->tf->eax;
4034
4035   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
4036     proc->tf->eax = syscalls[num]();
4037       //cprintf("%s -> %d\n", syscallnames[num], proc->tf->eax);
4038   } else {
4039     cprintf("%d %s: unknown sys call %d\n",
4040             proc->pid, proc->name, num);
4041     proc->tf->eax = -1;
4042   }
4043 }
4044
4045
4046
4047
4048
4049
```



```
4050 #include "types.h"
4051 #include "x86.h"
4052 #include "defs.h"
4053 #include "date.h"
4054 #include "param.h"
4055 #include "memlayout.h"
4056 #include "mmu.h"
4057 #include "proc.h"
4058 #include "uproc.h"
4059
4060 int
4061 sys_fork(void)
4062 {
4063   return fork();
4064 }
4065
4066 int
4067 sys_exit(void)
4068 {
4069   exit();
4070   return 0;  // not reached
4071 }
4072
4073 int
4074 sys_wait(void)
4075 {
4076   return wait();
4077 }
4078
4079 int
4080 sys_kill(void)
4081 {
4082   int pid;
4083
4084   if(argint(0, &pid) < 0)
4085     return -1;
4086   return kill(pid);
4087 }
4088
4089 int
4090 sys_getpid(void)
4091 {
4092   return proc->pid;
4093 }
4094
4095
4096
4097
4098
4099
```

```
4100 int
4101 sys_sbrk(void)
4102 {
4103   int addr;
4104   int n;
4105
4106   if(argint(0, &n) < 0)
4107     return -1;
4108   addr = proc->sz;
4109   if(growproc(n) < 0)
4110     return -1;
4111   return addr;
4112 }
4113
4114 int
4115 sys_sleep(void)
4116 {
4117   int n;
4118   uint ticks0;
4119
4120   if(argint(0, &n) < 0)
4121     return -1;
4122   acquire(&tickslock);
4123   ticks0 = ticks;
4124   while(ticks - ticks0 < n){
4125     if(proc->killed){
4126       release(&tickslock);
4127       return -1;
4128     }
4129     sleep(&ticks, &tickslock);
4130   }
4131   release(&tickslock);
4132   return 0;
4133 }
4134
4135 // return how many clock tick interrupts have occurred
4136 // since start.
4137 int
4138 sys_uptime(void)
4139 {
4140   uint xticks;
4141
4142   acquire(&tickslock);
4143   xticks = ticks;
4144   release(&tickslock);
4145   return xticks;
4146 }
4147
4148
4149
```



```
4150 //Turn of the computer
4151 int sys_halt(void){
4152   cprintf("Shutting down ...\n");
4153   outw (0xB004, 0x0 | 0x2000);
4154   return 0;
4155 }
4156
4157 //Implemented date and time
4158 int
4159 sys_date(void)
4160 {
4161     struct rtcdate *d;
4162
4163     if(argptr(0, (void*)&d, sizeof(*d)) < 0)
4164         return -1;
4165
4166     cmostime(d);
4167     return 0;
4168 }
4169
4170 int
4171 sys_getuid(void)
4172 {
4173   return proc->uid;
4174 }
4175
4176 int
4177 sys_getgid(void)
4178 {
4179   return proc->gid;
4180 }
4181
4182 int
4183 sys_getppid(void)
4184 {
4185   if(proc->parent)
4186     return proc->parent->pid;
4187   return proc->pid;
4188 }
4189
4190 int
4191 sys_setuid(void)
4192 {
4193   int uid;
4194
4195   if(argint(0, &uid) < 0)
4196     return -1;
4197   proc->uid = uid;
4198   return 0;
4199 }
```

```
4200 int
4201 sys_setgid(void)
4202 {
4203   int gid;
4204
4205   if(argint(0, &gid) < 0)
4206     return -1;
4207   proc->gid = gid;
4208   return 0;
4209 }
4210
4211 int
4212 sys_getprocs(void)
4213 {
4214   struct uproc * table;
4215   int MAX;
4216
4217   if(argptr(1, (void*)&table, sizeof(*table)) < 0)
4218     return -1;
4219
4220   if(argint(0, &MAX) < 0)
4221     return -1;
4222
4223   return copy(MAX, table);
4224 }
4225
4226
4227 // PROJECT 4
4228 int
4229 sys_setPriority(void)
4230 {
4231   int pid;
4232   int priority;
4233
4234   if(argint(1, &priority) < 0)
4235     return -1;
4236
4237   if(argint(0, &pid) < 0)
4238     return -1;
4239
4240   return setPriority(pid, priority);
4241 }
4242
4243
4244
4245
4246
4247
4248
4249
```



```
4250 // halt the system.
4251 #include "types.h"
4252 #include "user.h"
4253
4254 int
4255 main(void) {
4256   halt();
4257   return 0;
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 struct buf {
4301   int flags;
4302   uint dev;
4303   uint blockno;
4304   struct buf *prev; // LRU cache list
4305   struct buf *next;
4306   struct buf *qnext; // disk queue
4307   uchar data[BSIZE];
4308 };
4309 #define B_BUSY  0x1  // buffer is locked by some process
4310 #define B_VALID 0x2  // buffer has been read from disk
4311 #define B_DIRTY 0x4  // buffer needs to be written to disk
4312
4313
4314
4315
4316
4317
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 #define O_RDONLY  0x000
4351 #define O_WRONLY  0x001
4352 #define O_RDWR    0x002
4353 #define O_CREATE  0x200
4354
4355
4356
4357
4358
4359
4360
4361
4362
4363
4364
4365
4366
4367
4368
4369
4370
4371
4372
4373
4374
4375
4376
4377
4378
4379
4380
4381
4382
4383
4384
4385
4386
4387
4388
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
```

```
4400 #define T_DIR  1   // Directory
4401 #define T_FILE 2   // File
4402 #define T_DEV  3   // Device
4403
4404 struct stat {
4405   short type;  // Type of file
4406   int dev;     // File system's disk device
4407   uint ino;    // Inode number
4408   short nlink; // Number of links to file
4409   uint size;   // Size of file in bytes
4410 };
4411
4412
4413
4414
4415
4416
4417
4418
4419
4420
4421
4422
4423
4424
4425
4426
4427
4428
4429
4430
4431
4432
4433
4434
4435
4436
4437
4438
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
```



```
4450 // On-disk file system format.
4451 // Both the kernel and user programs use this header file.
4452
4453
4454 #define ROOTINO 1  // root i-number
4455 #define BSIZE 512  // block size
4456
4457 // Disk layout:
4458 // [ boot block | super block | log | inode blocks | free bit map | data bloc
4459 //
4460 // mkfs computes the super block and builds an initial file system. The super
4461 // the disk layout:
4462 struct superblock {
4463   uint size;         // Size of file system image (blocks)
4464   uint nblocks;      // Number of data blocks
4465   uint ninodes;      // Number of inodes.
4466   uint nlog;         // Number of log blocks
4467   uint logstart;     // Block number of first log block
4468   uint inodestart;   // Block number of first inode block
4469   uint bmapstart;    // Block number of first free map block
4470 };
4471
4472 #define NDIRECT 12
4473 #define NINDIRECT (BSIZE / sizeof(uint))
4474 #define MAXFILE (NDIRECT + NINDIRECT)
4475
4476 // On-disk inode structure
4477 struct dinode {
4478   short type;           // File type
4479   short major;          // Major device number (T_DEV only)
4480   short minor;          // Minor device number (T_DEV only)
4481   short nlink;          // Number of links to inode in file system
4482   uint size;            // Size of file (bytes)
4483   uint addrs[NDIRECT+1];   // Data block addresses
4484 };
4485
4486
4487
4488
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499
```

```
4500 // Inodes per block.
4501 #define IPB           (BSIZE / sizeof(struct dinode))
4502
4503 // Block containing inode i
4504 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
4505
4506 // Bitmap bits per block
4507 #define BPB           (BSIZE*8)
4508
4509 // Block of free map containing bit for block b
4510 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
4511
4512 // Directory is a file containing a sequence of dirent structures.
4513 #define DIRSIZ 14
4514
4515 struct dirent {
4516   ushort inum;
4517   char name[DIRSIZ];
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 struct file {
4551   enum { FD_NONE, FD_PIPE, FD_INODE } type;
4552   int ref; // reference count
4553   char readable;
4554   char writable;
4555   struct pipe *pipe;
4556   struct inode *ip;
4557   uint off;
4558 };
4559
4560
4561 // in-memory copy of an inode
4562 struct inode {
4563   uint dev;           // Device number
4564   uint inum;          // Inode number
4565   int ref;            // Reference count
4566   int flags;          // I_BUSY, I_VALID
4567
4568   short type;         // copy of disk inode
4569   short major;
4570   short minor;
4571   short nlink;
4572   uint size;
4573   uint addrs[NDIRECT+1];
4574 };
4575 #define I_BUSY 0x1
4576 #define I_VALID 0x2
4577
4578 // table mapping major device number to
4579 // device functions
4580 struct devsw {
4581   int (*read)(struct inode*, char*, int);
4582   int (*write)(struct inode*, char*, int);
4583 };
4584
4585 extern struct devsw devsw[];
4586
4587 #define CONSOLE 1
4588
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4599
```

```
4600 // Blank page.
4601
4602
4603
4604
4605
4606
4607
4608
4609
4610
4611
4612
4613
4614
4615
4616
4617
4618
4619
4620
4621
4622
4623
4624
4625
4626
4627
4628
4629
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649
```



```
4650 // Simple PIO-based (non-DMA) IDE driver code.
4651
4652 #include "types.h"
4653 #include "defs.h"
4654 #include "param.h"
4655 #include "memlayout.h"
4656 #include "mmu.h"
4657 #include "proc.h"
4658 #include "x86.h"
4659 #include "traps.h"
4660 #include "spinlock.h"
4661 #include "fs.h"
4662 #include "buf.h"
4663
4664 #define SECTOR_SIZE   512
4665 #define IDE_BSY       0x80
4666 #define IDE_DRDY      0x40
4667 #define IDE_DF        0x20
4668 #define IDE_ERR       0x01
4669
4670 #define IDE_CMD_READ  0x20
4671 #define IDE_CMD_WRITE 0x30
4672
4673 // idequeue points to the buf now being read/written to the disk.
4674 // idequeue->qnext points to the next buf to be processed.
4675 // You must hold idelock while manipulating queue.
4676
4677 static struct spinlock idelock;
4678 static struct buf *idequeue;
4679
4680 static int havedisk1;
4681 static void idestart(struct buf*);
4682
4683 // Wait for IDE disk to become ready.
4684 static int
4685 idewait(int checkerr)
4686 {
4687   int r;
4688
4689   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4690     ;
4691   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4692     return -1;
4693   return 0;
4694 }
4695
4696
4697
4698
4699
```

```
4700 void
4701 ideinit(void)
4702 {
4703   int i;
4704
4705   initlock(&idelock, "ide");
4706   picenable(IRQ_IDE);
4707   ioapicenable(IRQ_IDE, ncpu - 1);
4708   idewait(0);
4709
4710   // Check if disk 1 is present
4711   outb(0x1f6, 0xe0 | (1<<4));
4712   for(i=0; i<1000; i++){
4713     if(inb(0x1f7) != 0){
4714       havedisk1 = 1;
4715       break;
4716     }
4717   }
4718
4719   // Switch back to disk 0.
4720   outb(0x1f6, 0xe0 | (0<<4));
4721 }
4722
4723 // Start the request for b.  Caller must hold idelock.
4724 static void
4725 idestart(struct buf *b)
4726 {
4727   if(b == 0)
4728     panic("idestart");
4729   if(b->blockno >= FSSIZE)
4730     panic("incorrect blockno");
4731   int sector_per_block =  BSIZE/SECTOR_SIZE;
4732   int sector = b->blockno * sector_per_block;
4733
4734   if (sector_per_block > 7) panic("idestart");
4735
4736   idewait(0);
4737   outb(0x3f6, 0);  // generate interrupt
4738   outb(0x1f2, sector_per_block);  // number of sectors
4739   outb(0x1f3, sector & 0xff);
4740   outb(0x1f4, (sector >> 8) & 0xff);
4741   outb(0x1f5, (sector >> 16) & 0xff);
4742   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4743   if(b->flags & B_DIRTY){
4744     outb(0x1f7, IDE_CMD_WRITE);
4745     outsl(0x1f0, b->data, BSIZE/4);
4746   } else {
4747     outb(0x1f7, IDE_CMD_READ);
4748   }
4749 }
```



```
4750 // Interrupt handler.
4751 void
4752 ideintr(void)
4753 {
4754   struct buf *b;
4755
4756   // First queued buffer is the active request.
4757   acquire(&idelock);
4758   if((b = idequeue) == 0){
4759     release(&idelock);
4760     // cprintf("spurious IDE interrupt\n");
4761     return;
4762   }
4763   idequeue = b->qnext;
4764
4765   // Read data if needed.
4766   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4767     insl(0x1f0, b->data, BSIZE/4);
4768
4769   // Wake process waiting for this buf.
4770   b->flags |= B_VALID;
4771   b->flags &= ~B_DIRTY;
4772   wakeup(b);
4773
4774   // Start disk on next buf in queue.
4775   if(idequeue != 0)
4776     idestart(idequeue);
4777
4778   release(&idelock);
4779 }
4780
4781
4782
4783
4784
4785
4786
4787
4788
4789
4790
4791
4792
4793
4794
4795
4796
4797
4798
4799
```

```
4800 // Sync buf with disk.
4801 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
4802 // Else if B_VALID is not set, read buf from disk, set B_VALID.
4803 void
4804 iderw(struct buf *b)
4805 {
4806   struct buf **pp;
4807
4808   if(!(b->flags & B_BUSY))
4809     panic("iderw: buf not busy");
4810   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
4811     panic("iderw: nothing to do");
4812   if(b->dev != 0 && !havedisk1)
4813     panic("iderw: ide disk 1 not present");
4814
4815   acquire(&idelock);
4816
4817   // Append b to idequeue.
4818   b->qnext = 0;
4819   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
4820     ;
4821   *pp = b;
4822
4823   // Start disk if necessary.
4824   if(idequeue == b)
4825     idestart(b);
4826
4827   // Wait for request to finish.
4828   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
4829     sleep(b, &idelock);
4830   }
4831
4832   release(&idelock);
4833 }
4834
4835
4836
4837
4838
4839
4840
4841
4842
4843
4844
4845
4846
4847
4848
4849
```



```
4850 // Buffer cache.
4851 //
4852 // The buffer cache is a linked list of buf structures holding
4853 // cached copies of disk block contents.  Caching disk blocks
4854 // in memory reduces the number of disk reads and also provides
4855 // a synchronization point for disk blocks used by multiple processes.
4856 //
4857 // Interface:
4858 // * To get a buffer for a particular disk block, call bread.
4859 // * After changing buffer data, call bwrite to write it to disk.
4860 // * When done with the buffer, call brelse.
4861 // * Do not use the buffer after calling brelse.
4862 // * Only one process at a time can use a buffer,
4863 //     so do not keep them longer than necessary.
4864 //
4865 // The implementation uses three state flags internally:
4866 // * B_BUSY: the block has been returned from bread
4867 //     and has not been passed back to brelse.
4868 // * B_VALID: the buffer data has been read from the disk.
4869 // * B_DIRTY: the buffer data has been modified
4870 //     and needs to be written to disk.
4871
4872 #include "types.h"
4873 #include "defs.h"
4874 #include "param.h"
4875 #include "spinlock.h"
4876 #include "fs.h"
4877 #include "buf.h"
4878
4879 struct {
4880   struct spinlock lock;
4881   struct buf buf[NBUF];
4882
4883   // Linked list of all buffers, through prev/next.
4884   // head.next is most recently used.
4885   struct buf head;
4886 } bcache;
4887
4888 void
4889 binit(void)
4890 {
4891   struct buf *b;
4892
4893   initlock(&bcache.lock, "bcache");
4894
4895
4896
4897
4898
4899
```

```
4900   // Create linked list of buffers
4901   bcache.head.prev = &bcache.head;
4902   bcache.head.next = &bcache.head;
4903   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4904     b->next = bcache.head.next;
4905     b->prev = &bcache.head;
4906     b->dev = -1;
4907     bcache.head.next->prev = b;
4908     bcache.head.next = b;
4909   }
4910 }
4911
4912 // Look through buffer cache for block on device dev.
4913 // If not found, allocate a buffer.
4914 // In either case, return B_BUSY buffer.
4915 static struct buf*
4916 bget(uint dev, uint blockno)
4917 {
4918   struct buf *b;
4919
4920   acquire(&bcache.lock);
4921
4922  loop:
4923   // Is the block already cached?
4924   for(b = bcache.head.next; b != &bcache.head; b = b->next){
4925     if(b->dev == dev && b->blockno == blockno){
4926       if(!(b->flags & B_BUSY)){
4927         b->flags |= B_BUSY;
4928         release(&bcache.lock);
4929         return b;
4930       }
4931       sleep(b, &bcache.lock);
4932       goto loop;
4933     }
4934   }
4935
4936   // Not cached; recycle some non-busy and clean buffer.
4937   // "clean" because B_DIRTY and !B_BUSY means log.c
4938   // hasn't yet committed the changes to the buffer.
4939   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4940     if((b->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
4941       b->dev = dev;
4942       b->blockno = blockno;
4943       b->flags = B_BUSY;
4944       release(&bcache.lock);
4945       return b;
4946     }
4947   }
4948   panic("bget: no buffers");
4949 }
```



```
4950 // Return a B_BUSY buf with the contents of the indicated block.
4951 struct buf*
4952 bread(uint dev, uint blockno)
4953 {
4954   struct buf *b;
4955
4956   b = bget(dev, blockno);
4957   if(!(b->flags & B_VALID)) {
4958     iderw(b);
4959   }
4960   return b;
4961 }
4962
4963 // Write b's contents to disk.  Must be B_BUSY.
4964 void
4965 bwrite(struct buf *b)
4966 {
4967   if((b->flags & B_BUSY) == 0)
4968     panic("bwrite");
4969   b->flags |= B_DIRTY;
4970   iderw(b);
4971 }
4972
4973 // Release a B_BUSY buffer.
4974 // Move to the head of the MRU list.
4975 void
4976 brelse(struct buf *b)
4977 {
4978   if((b->flags & B_BUSY) == 0)
4979     panic("brelse");
4980
4981   acquire(&bcache.lock);
4982
4983   b->next->prev = b->prev;
4984   b->prev->next = b->next;
4985   b->next = bcache.head.next;
4986   b->prev = &bcache.head;
4987   bcache.head.next->prev = b;
4988   bcache.head.next = b;
4989
4990   b->flags &= ~B_BUSY;
4991   wakeup(b);
4992
4993   release(&bcache.lock);
4994 }
4995
4996
4997
4998
4999
```

```
5000 // Blank page.
5001
5002
5003
5004
5005
5006
5007
5008
5009
5010
5011
5012
5013
5014
5015
5016
5017
5018
5019
5020
5021
5022
5023
5024
5025
5026
5027
5028
5029
5030
5031
5032
5033
5034
5035
5036
5037
5038
5039
5040
5041
5042
5043
5044
5045
5046
5047
5048
5049
```



```
5050 #include "types.h"
5051 #include "defs.h"
5052 #include "param.h"
5053 #include "spinlock.h"
5054 #include "fs.h"
5055 #include "buf.h"
5056
5057 // Simple logging that allows concurrent FS system calls.
5058 //
5059 // A log transaction contains the updates of multiple FS system
5060 // calls. The logging system only commits when there are
5061 // no FS system calls active. Thus there is never
5062 // any reasoning required about whether a commit might
5063 // write an uncommitted system call's updates to disk.
5064 //
5065 // A system call should call begin_op()/end_op() to mark
5066 // its start and end. Usually begin_op() just increments
5067 // the count of in-progress FS system calls and returns.
5068 // But if it thinks the log is close to running out, it
5069 // sleeps until the last outstanding end_op() commits.
5070 //
5071 // The log is a physical re-do log containing disk blocks.
5072 // The on-disk log format:
5073 //   header block, containing block #s for block A, B, C, ...
5074 //   block A
5075 //   block B
5076 //   block C
5077 //   ...
5078 // Log appends are synchronous.
5079
5080 // Contents of the header block, used for both the on-disk header block
5081 // and to keep track in memory of logged block# before commit.
5082 struct logheader {
5083   int n;
5084   int block[LOGSIZE];
5085 };
5086
5087 struct log {
5088   struct spinlock lock;
5089   int start;
5090   int size;
5091   int outstanding; // how many FS sys calls are executing.
5092   int committing;  // in commit(), please wait.
5093   int dev;
5094   struct logheader lh;
5095 };
5096
5097
5098
5099
```

```
5100 struct log log;
5101
5102 static void recover_from_log(void);
5103 static void commit();
5104
5105 void
5106 initlog(int dev)
5107 {
5108   if (sizeof(struct logheader) >= BSIZE)
5109     panic("initlog: too big logheader");
5110
5111   struct superblock sb;
5112   initlock(&log.lock, "log");
5113   readsb(dev, &sb);
5114   log.start = sb.logstart;
5115   log.size = sb.nlog;
5116   log.dev = dev;
5117   recover_from_log();
5118 }
5119
5120 // Copy committed blocks from log to their home location
5121 static void
5122 install_trans(void)
5123 {
5124   int tail;
5125
5126   for (tail = 0; tail < log.lh.n; tail++) {
5127     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
5128     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
5129     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
5130     bwrite(dbuf);  // write dst to disk
5131     brelse(lbuf);
5132     brelse(dbuf);
5133   }
5134 }
5135
5136 // Read the log header from disk into the in-memory log header
5137 static void
5138 read_head(void)
5139 {
5140   struct buf *buf = bread(log.dev, log.start);
5141   struct logheader *lh = (struct logheader *) (buf->data);
5142   int i;
5143   log.lh.n = lh->n;
5144   for (i = 0; i < log.lh.n; i++) {
5145     log.lh.block[i] = lh->block[i];
5146   }
5147   brelse(buf);
5148 }
5149
```



```
5150 // Write in-memory log header to disk.
5151 // This is the true point at which the
5152 // current transaction commits.
5153 static void
5154 write_head(void)
5155 {
5156   struct buf *buf = bread(log.dev, log.start);
5157   struct logheader *hb = (struct logheader *) (buf->data);
5158   int i;
5159   hb->n = log.lh.n;
5160   for (i = 0; i < log.lh.n; i++) {
5161     hb->block[i] = log.lh.block[i];
5162   }
5163   bwrite(buf);
5164   brelse(buf);
5165 }
5166
5167 static void
5168 recover_from_log(void)
5169 {
5170   read_head();
5171   install_trans(); // if committed, copy from log to disk
5172   log.lh.n = 0;
5173   write_head(); // clear the log
5174 }
5175
5176 // called at the start of each FS system call.
5177 void
5178 begin_op(void)
5179 {
5180   acquire(&log.lock);
5181   while(1){
5182     if(log.committing){
5183       sleep(&log, &log.lock);
5184     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
5185       // this op might exhaust log space; wait for commit.
5186       sleep(&log, &log.lock);
5187     } else {
5188       log.outstanding += 1;
5189       release(&log.lock);
5190       break;
5191     }
5192   }
5193 }
5194
5195
5196
5197
5198
5199
```

```
5200 // called at the end of each FS system call.
5201 // commits if this was the last outstanding operation.
5202 void
5203 end_op(void)
5204 {
5205   int do_commit = 0;
5206
5207   acquire(&log.lock);
5208   log.outstanding -= 1;
5209   if(log.committing)
5210     panic("log.committing");
5211   if(log.outstanding == 0){
5212     do_commit = 1;
5213     log.committing = 1;
5214   } else {
5215     // begin_op() may be waiting for log space.
5216     wakeup(&log);
5217   }
5218   release(&log.lock);
5219
5220   if(do_commit){
5221     // call commit w/o holding locks, since not allowed
5222     // to sleep with locks.
5223     commit();
5224     acquire(&log.lock);
5225     log.committing = 0;
5226     wakeup(&log);
5227     release(&log.lock);
5228   }
5229 }
5230
5231 // Copy modified blocks from cache to log.
5232 static void
5233 write_log(void)
5234 {
5235   int tail;
5236
5237   for (tail = 0; tail < log.lh.n; tail++) {
5238     struct buf *to = bread(log.dev, log.start+tail+1); // log block
5239     struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
5240     memmove(to->data, from->data, BSIZE);
5241     bwrite(to);  // write the log
5242     brelse(from);
5243     brelse(to);
5244   }
5245 }
5246
5247
5248
5249
```



```
5250 static void
5251 commit()
5252 {
5253   if (log.lh.n > 0) {
5254     write_log();     // Write modified blocks from cache to log
5255     write_head();    // Write header to disk -- the real commit
5256     install_trans(); // Now install writes to home locations
5257     log.lh.n = 0;
5258     write_head();    // Erase the transaction from the log
5259   }
5260 }
5261
5262 // Caller has modified b->data and is done with the buffer.
5263 // Record the block number and pin in the cache with B_DIRTY.
5264 // commit()/write_log() will do the disk write.
5265 //
5266 // log_write() replaces bwrite(); a typical use is:
5267 //   bp = bread(...)
5268 //   modify bp->data[]
5269 //   log_write(bp)
5270 //   brelse(bp)
5271 void
5272 log_write(struct buf *b)
5273 {
5274   int i;
5275
5276   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
5277     panic("too big a transaction");
5278   if (log.outstanding < 1)
5279     panic("log_write outside of trans");
5280
5281   acquire(&log.lock);
5282   for (i = 0; i < log.lh.n; i++) {
5283     if (log.lh.block[i] == b->blockno)   // log absorbtion
5284       break;
5285   }
5286   log.lh.block[i] = b->blockno;
5287   if (i == log.lh.n)
5288     log.lh.n++;
5289   b->flags |= B_DIRTY; // prevent eviction
5290   release(&log.lock);
5291 }
5292
5293
5294
5295
5296
5297
5298
5299
```

```
5300 // File system implementation.  Five layers:
5301 //   + Blocks: allocator for raw disk blocks.
5302 //   + Log: crash recovery for multi-step updates.
5303 //   + Files: inode allocator, reading, writing, metadata.
5304 //   + Directories: inode with special contents (list of other inodes!)
5305 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
5306 //
5307 // This file contains the low-level file system manipulation
5308 // routines.  The (higher-level) system call implementations
5309 // are in sysfile.c.
5310
5311 #include "types.h"
5312 #include "defs.h"
5313 #include "param.h"
5314 #include "stat.h"
5315 #include "mmu.h"
5316 #include "proc.h"
5317 #include "spinlock.h"
5318 #include "fs.h"
5319 #include "buf.h"
5320 #include "file.h"
5321
5322 #define min(a, b) ((a) < (b) ? (a) : (b))
5323 static void itrunc(struct inode*);
5324 struct superblock sb;   // there should be one per dev, but we run with one d
5325
5326 // Read the super block.
5327 void
5328 readsb(int dev, struct superblock *sb)
5329 {
5330   struct buf *bp;
5331
5332   bp = bread(dev, 1);
5333   memmove(sb, bp->data, sizeof(*sb));
5334   brelse(bp);
5335 }
5336
5337 // Zero a block.
5338 static void
5339 bzero(int dev, int bno)
5340 {
5341   struct buf *bp;
5342
5343   bp = bread(dev, bno);
5344   memset(bp->data, 0, BSIZE);
5345   log_write(bp);
5346   brelse(bp);
5347 }
5348
5349
```



```
5350 // Blocks.
5351
5352 // Allocate a zeroed disk block.
5353 static uint
5354 balloc(uint dev)
5355 {
5356   int b, bi, m;
5357   struct buf *bp;
5358
5359   bp = 0;
5360   for(b = 0; b < sb.size; b += BPB){
5361     bp = bread(dev, BBLOCK(b, sb));
5362     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
5363       m = 1 << (bi % 8);
5364       if((bp->data[bi/8] & m) == 0){  // Is block free?
5365         bp->data[bi/8] |= m;  // Mark block in use.
5366         log_write(bp);
5367         brelse(bp);
5368         bzero(dev, b + bi);
5369         return b + bi;
5370       }
5371     }
5372     brelse(bp);
5373   }
5374   panic("balloc: out of blocks");
5375 }
5376
5377 // Free a disk block.
5378 static void
5379 bfree(int dev, uint b)
5380 {
5381   struct buf *bp;
5382   int bi, m;
5383
5384   readsb(dev, &sb);
5385   bp = bread(dev, BBLOCK(b, sb));
5386   bi = b % BPB;
5387   m = 1 << (bi % 8);
5388   if((bp->data[bi/8] & m) == 0)
5389     panic("freeing free block");
5390   bp->data[bi/8] &= ~m;
5391   log_write(bp);
5392   brelse(bp);
5393 }
5394
5395
5396
5397
5398
5399
```

```
5400 // Inodes.
5401 //
5402 // An inode describes a single unnamed file.
5403 // The inode disk structure holds metadata: the file's type,
5404 // its size, the number of links referring to it, and the
5405 // list of blocks holding the file's content.
5406 //
5407 // The inodes are laid out sequentially on disk at
5408 // sb.startinode. Each inode has a number, indicating its
5409 // position on the disk.
5410 //
5411 // The kernel keeps a cache of in-use inodes in memory
5412 // to provide a place for synchronizing access
5413 // to inodes used by multiple processes. The cached
5414 // inodes include book-keeping information that is
5415 // not stored on disk: ip->ref and ip->flags.
5416 //
5417 // An inode and its in-memory represtative go through a
5418 // sequence of states before they can be used by the
5419 // rest of the file system code.
5420 //
5421 // * Allocation: an inode is allocated if its type (on disk)
5422 //   is non-zero. ialloc() allocates, iput() frees if
5423 //   the link count has fallen to zero.
5424 //
5425 // * Referencing in cache: an entry in the inode cache
5426 //   is free if ip->ref is zero. Otherwise ip->ref tracks
5427 //   the number of in-memory pointers to the entry (open
5428 //   files and current directories). iget() to find or
5429 //   create a cache entry and increment its ref, iput()
5430 //   to decrement ref.
5431 //
5432 // * Valid: the information (type, size, &c) in an inode
5433 //   cache entry is only correct when the I_VALID bit
5434 //   is set in ip->flags. ilock() reads the inode from
5435 //   the disk and sets I_VALID, while iput() clears
5436 //   I_VALID if ip->ref has fallen to zero.
5437 //
5438 // * Locked: file system code may only examine and modify
5439 //   the information in an inode and its content if it
5440 //   has first locked the inode. The I_BUSY flag indicates
5441 //   that the inode is locked. ilock() sets I_BUSY,
5442 //   while iunlock clears it.
5443 //
5444 // Thus a typical sequence is:
5445 //   ip = iget(dev, inum)
5446 //   ilock(ip)
5447 //   ... examine and modify ip->xxx ...
5448 //   iunlock(ip)
5449 //   iput(ip)
```



```
5450 //
5451 // ilock() is separate from iget() so that system calls can
5452 // get a long-term reference to an inode (as for an open file)
5453 // and only lock it for short periods (e.g., in read()).
5454 // The separation also helps avoid deadlock and races during
5455 // pathname lookup. iget() increments ip->ref so that the inode
5456 // stays cached and pointers to it remain valid.
5457 //
5458 // Many internal file system functions expect the caller to
5459 // have locked the inodes involved; this lets callers create
5460 // multi-step atomic operations.
5461
5462 struct {
5463   struct spinlock lock;
5464   struct inode inode[NINODE];
5465 } icache;
5466
5467 void
5468 iinit(int dev)
5469 {
5470   initlock(&icache.lock, "icache");
5471   readsb(dev, &sb);
5472   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d inodestart %
5473           sb.nblocks, sb.ninodes, sb.nlog, sb.logstart, sb.inodestart, sb.bma
5474 }
5475
5476 static struct inode* iget(uint dev, uint inum);
5477
5478
5479
5480
5481
5482
5483
5484
5485
5486
5487
5488
5489
5490
5491
5492
5493
5494
5495
5496
5497
5498
5499
```

```
5500 // Allocate a new inode with the given type on device dev.
5501 // A free inode has a type of zero.
5502 struct inode*
5503 ialloc(uint dev, short type)
5504 {
5505   int inum;
5506   struct buf *bp;
5507   struct dinode *dip;
5508
5509   for(inum = 1; inum < sb.ninodes; inum++){
5510     bp = bread(dev, IBLOCK(inum, sb));
5511     dip = (struct dinode*)bp->data + inum%IPB;
5512     if(dip->type == 0){  // a free inode
5513       memset(dip, 0, sizeof(*dip));
5514       dip->type = type;
5515       log_write(bp);   // mark it allocated on the disk
5516       brelse(bp);
5517       return iget(dev, inum);
5518     }
5519     brelse(bp);
5520   }
5521   panic("ialloc: no inodes");
5522 }
5523
5524 // Copy a modified in-memory inode to disk.
5525 void
5526 iupdate(struct inode *ip)
5527 {
5528   struct buf *bp;
5529   struct dinode *dip;
5530
5531   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5532   dip = (struct dinode*)bp->data + ip->inum%IPB;
5533   dip->type = ip->type;
5534   dip->major = ip->major;
5535   dip->minor = ip->minor;
5536   dip->nlink = ip->nlink;
5537   dip->size = ip->size;
5538   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5539   log_write(bp);
5540   brelse(bp);
5541 }
5542
5543
5544
5545
5546
5547
5548
5549
```



```
5550 // Find the inode with number inum on device dev
5551 // and return the in-memory copy. Does not lock
5552 // the inode and does not read it from disk.
5553 static struct inode*
5554 iget(uint dev, uint inum)
5555 {
5556   struct inode *ip, *empty;
5557
5558   acquire(&icache.lock);
5559
5560   // Is the inode already cached?
5561   empty = 0;
5562   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5563     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5564       ip->ref++;
5565       release(&icache.lock);
5566       return ip;
5567     }
5568     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
5569       empty = ip;
5570   }
5571
5572   // Recycle an inode cache entry.
5573   if(empty == 0)
5574     panic("iget: no inodes");
5575
5576   ip = empty;
5577   ip->dev = dev;
5578   ip->inum = inum;
5579   ip->ref = 1;
5580   ip->flags = 0;
5581   release(&icache.lock);
5582
5583   return ip;
5584 }
5585
5586 // Increment reference count for ip.
5587 // Returns ip to enable ip = idup(ip1) idiom.
5588 struct inode*
5589 idup(struct inode *ip)
5590 {
5591   acquire(&icache.lock);
5592   ip->ref++;
5593   release(&icache.lock);
5594   return ip;
5595 }
5596
5597
5598
5599
```

```
5600 // Lock the given inode.
5601 // Reads the inode from disk if necessary.
5602 void
5603 ilock(struct inode *ip)
5604 {
5605   struct buf *bp;
5606   struct dinode *dip;
5607
5608   if(ip == 0 || ip->ref < 1)
5609     panic("ilock");
5610
5611   acquire(&icache.lock);
5612   while(ip->flags & I_BUSY)
5613     sleep(ip, &icache.lock);
5614   ip->flags |= I_BUSY;
5615   release(&icache.lock);
5616
5617   if(!(ip->flags & I_VALID)){
5618     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5619     dip = (struct dinode*)bp->data + ip->inum%IPB;
5620     ip->type = dip->type;
5621     ip->major = dip->major;
5622     ip->minor = dip->minor;
5623     ip->nlink = dip->nlink;
5624     ip->size = dip->size;
5625     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5626     brelse(bp);
5627     ip->flags |= I_VALID;
5628     if(ip->type == 0)
5629       panic("ilock: no type");
5630   }
5631 }
5632
5633 // Unlock the given inode.
5634 void
5635 iunlock(struct inode *ip)
5636 {
5637   if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
5638     panic("iunlock");
5639
5640   acquire(&icache.lock);
5641   ip->flags &= ~I_BUSY;
5642   wakeup(ip);
5643   release(&icache.lock);
5644 }
5645
5646
5647
5648
5649
```



```
5650 // Drop a reference to an in-memory inode.
5651 // If that was the last reference, the inode cache entry can
5652 // be recycled.
5653 // If that was the last reference and the inode has no links
5654 // to it, free the inode (and its content) on disk.
5655 // All calls to iput() must be inside a transaction in
5656 // case it has to free the inode.
5657 void
5658 iput(struct inode *ip)
5659 {
5660   acquire(&icache.lock);
5661   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
5662     // inode has no links and no other references: truncate and free.
5663     if(ip->flags & I_BUSY)
5664       panic("iput busy");
5665     ip->flags |= I_BUSY;
5666     release(&icache.lock);
5667     itrunc(ip);
5668     ip->type = 0;
5669     iupdate(ip);
5670     acquire(&icache.lock);
5671     ip->flags = 0;
5672     wakeup(ip);
5673   }
5674   ip->ref--;
5675   release(&icache.lock);
5676 }
5677
5678 // Common idiom: unlock, then put.
5679 void
5680 iunlockput(struct inode *ip)
5681 {
5682   iunlock(ip);
5683   iput(ip);
5684 }
5685
5686
5687
5688
5689
5690
5691
5692
5693
5694
5695
5696
5697
5698
5699
```

```
5700 // Inode content
5701 //
5702 // The content (data) associated with each inode is stored
5703 // in blocks on the disk. The first NDIRECT block numbers
5704 // are listed in ip->addrs[].  The next NINDIRECT blocks are
5705 // listed in block ip->addrs[NDIRECT].
5706
5707 // Return the disk block address of the nth block in inode ip.
5708 // If there is no such block, bmap allocates one.
5709 static uint
5710 bmap(struct inode *ip, uint bn)
5711 {
5712   uint addr, *a;
5713   struct buf *bp;
5714
5715   if(bn < NDIRECT){
5716     if((addr = ip->addrs[bn]) == 0)
5717       ip->addrs[bn] = addr = balloc(ip->dev);
5718     return addr;
5719   }
5720   bn -= NDIRECT;
5721
5722   if(bn < NINDIRECT){
5723     // Load indirect block, allocating if necessary.
5724     if((addr = ip->addrs[NDIRECT]) == 0)
5725       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5726     bp = bread(ip->dev, addr);
5727     a = (uint*)bp->data;
5728     if((addr = a[bn]) == 0){
5729       a[bn] = addr = balloc(ip->dev);
5730       log_write(bp);
5731     }
5732     brelse(bp);
5733     return addr;
5734   }
5735
5736   panic("bmap: out of range");
5737 }
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749
```



```
5750 // Truncate inode (discard contents).
5751 // Only called when the inode has no links
5752 // to it (no directory entries referring to it)
5753 // and has no in-memory reference to it (is
5754 // not an open file or current directory).
5755 static void
5756 itrunc(struct inode *ip)
5757 {
5758   int i, j;
5759   struct buf *bp;
5760   uint *a;
5761
5762   for(i = 0; i < NDIRECT; i++){
5763     if(ip->addrs[i]){
5764       bfree(ip->dev, ip->addrs[i]);
5765       ip->addrs[i] = 0;
5766     }
5767   }
5768
5769   if(ip->addrs[NDIRECT]){
5770     bp = bread(ip->dev, ip->addrs[NDIRECT]);
5771     a = (uint*)bp->data;
5772     for(j = 0; j < NINDIRECT; j++){
5773       if(a[j])
5774         bfree(ip->dev, a[j]);
5775     }
5776     brelse(bp);
5777     bfree(ip->dev, ip->addrs[NDIRECT]);
5778     ip->addrs[NDIRECT] = 0;
5779   }
5780
5781   ip->size = 0;
5782   iupdate(ip);
5783 }
5784
5785 // Copy stat information from inode.
5786 void
5787 stati(struct inode *ip, struct stat *st)
5788 {
5789   st->dev = ip->dev;
5790   st->ino = ip->inum;
5791   st->type = ip->type;
5792   st->nlink = ip->nlink;
5793   st->size = ip->size;
5794 }
5795
5796
5797
5798
5799
```

```
5800 // Read data from inode.
5801 int
5802 readi(struct inode *ip, char *dst, uint off, uint n)
5803 {
5804   uint tot, m;
5805   struct buf *bp;
5806
5807   if(ip->type == T_DEV){
5808     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
5809       return -1;
5810     return devsw[ip->major].read(ip, dst, n);
5811   }
5812
5813   if(off > ip->size || off + n < off)
5814     return -1;
5815   if(off + n > ip->size)
5816     n = ip->size - off;
5817
5818   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5819     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5820     m = min(n - tot, BSIZE - off%BSIZE);
5821     memmove(dst, bp->data + off%BSIZE, m);
5822     brelse(bp);
5823   }
5824   return n;
5825 }
5826
5827
5828
5829
5830
5831
5832
5833
5834
5835
5836
5837
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```



```
5850 // Write data to inode.
5851 int
5852 writei(struct inode *ip, char *src, uint off, uint n)
5853 {
5854   uint tot, m;
5855   struct buf *bp;
5856
5857   if(ip->type == T_DEV){
5858     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
5859       return -1;
5860     return devsw[ip->major].write(ip, src, n);
5861   }
5862
5863   if(off > ip->size || off + n < off)
5864     return -1;
5865   if(off + n > MAXFILE*BSIZE)
5866     return -1;
5867
5868   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
5869     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5870     m = min(n - tot, BSIZE - off%BSIZE);
5871     memmove(bp->data + off%BSIZE, src, m);
5872     log_write(bp);
5873     brelse(bp);
5874   }
5875
5876   if(n > 0 && off > ip->size){
5877     ip->size = off;
5878     iupdate(ip);
5879   }
5880   return n;
5881 }
5882
5883
5884
5885
5886
5887
5888
5889
5890
5891
5892
5893
5894
5895
5896
5897
5898
5899
```

```
5900 // Directories
5901
5902 int
5903 namecmp(const char *s, const char *t)
5904 {
5905   return strncmp(s, t, DIRSIZ);
5906 }
5907
5908 // Look for a directory entry in a directory.
5909 // If found, set *poff to byte offset of entry.
5910 struct inode*
5911 dirlookup(struct inode *dp, char *name, uint *poff)
5912 {
5913   uint off, inum;
5914   struct dirent de;
5915
5916   if(dp->type != T_DIR)
5917     panic("dirlookup not DIR");
5918
5919   for(off = 0; off < dp->size; off += sizeof(de)){
5920     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5921       panic("dirlink read");
5922     if(de.inum == 0)
5923       continue;
5924     if(namecmp(name, de.name) == 0){
5925       // entry matches path element
5926       if(poff)
5927         *poff = off;
5928       inum = de.inum;
5929       return iget(dp->dev, inum);
5930     }
5931   }
5932
5933   return 0;
5934 }
5935
5936
5937
5938
5939
5940
5941
5942
5943
5944
5945
5946
5947
5948
5949
```



```
5950 // Write a new directory entry (name, inum) into the directory dp.
5951 int
5952 dirlink(struct inode *dp, char *name, uint inum)
5953 {
5954   int off;
5955   struct dirent de;
5956   struct inode *ip;
5957
5958   // Check that name is not present.
5959   if((ip = dirlookup(dp, name, 0)) != 0){
5960     iput(ip);
5961     return -1;
5962   }
5963
5964   // Look for an empty dirent.
5965   for(off = 0; off < dp->size; off += sizeof(de)){
5966     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5967       panic("dirlink read");
5968     if(de.inum == 0)
5969       break;
5970   }
5971
5972   strncpy(de.name, name, DIRSIZ);
5973   de.inum = inum;
5974   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5975     panic("dirlink");
5976
5977   return 0;
5978 }
5979
5980
5981
5982
5983
5984
5985
5986
5987
5988
5989
5990
5991
5992
5993
5994
5995
5996
5997
5998
5999
```

```
6000 // Paths
6001
6002 // Copy the next path element from path into name.
6003 // Return a pointer to the element following the copied one.
6004 // The returned path has no leading slashes,
6005 // so the caller can check *path=='\0' to see if the name is the last one.
6006 // If no name to remove, return 0.
6007 //
6008 // Examples:
6009 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
6010 //   skipelem("///a//bb", name) = "bb", setting name = "a"
6011 //   skipelem("a", name) = "", setting name = "a"
6012 //   skipelem("", name) = skipelem("////", name) = 0
6013 //
6014 static char*
6015 skipelem(char *path, char *name)
6016 {
6017   char *s;
6018   int len;
6019
6020   while(*path == '/')
6021     path++;
6022   if(*path == 0)
6023     return 0;
6024   s = path;
6025   while(*path != '/' && *path != 0)
6026     path++;
6027   len = path - s;
6028   if(len >= DIRSIZ)
6029     memmove(name, s, DIRSIZ);
6030   else {
6031     memmove(name, s, len);
6032     name[len] = 0;
6033   }
6034   while(*path == '/')
6035     path++;
6036   return path;
6037 }
6038
6039
6040
6041
6042
6043
6044
6045
6046
6047
6048
6049
```



```
6050 // Look up and return the inode for a path name.
6051 // If parent != 0, return the inode for the parent and copy the final
6052 // path element into name, which must have room for DIRSIZ bytes.
6053 // Must be called inside a transaction since it calls iput().
6054 static struct inode*
6055 namex(char *path, int nameiparent, char *name)
6056 {
6057   struct inode *ip, *next;
6058
6059   if(*path == '/')
6060     ip = iget(ROOTDEV, ROOTINO);
6061   else
6062     ip = idup(proc->cwd);
6063
6064   while((path = skipelem(path, name)) != 0){
6065     ilock(ip);
6066     if(ip->type != T_DIR){
6067       iunlockput(ip);
6068       return 0;
6069     }
6070     if(nameiparent && *path == '\0'){
6071       // Stop one level early.
6072       iunlock(ip);
6073       return ip;
6074     }
6075     if((next = dirlookup(ip, name, 0)) == 0){
6076       iunlockput(ip);
6077       return 0;
6078     }
6079     iunlockput(ip);
6080     ip = next;
6081   }
6082   if(nameiparent){
6083     iput(ip);
6084     return 0;
6085   }
6086   return ip;
6087 }
6088
6089 struct inode*
6090 namei(char *path)
6091 {
6092   char name[DIRSIZ];
6093   return namex(path, 0, name);
6094 }
6095
6096
6097
6098
6099
```

```
6100 struct inode*
6101 nameiparent(char *path, char *name)
6102 {
6103   return namex(path, 1, name);
6104 }
6105
6106
6107
6108
6109
6110
6111
6112
6113
6114
6115
6116
6117
6118
6119
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 //
6151 // File descriptors
6152 //
6153
6154 #include "types.h"
6155 #include "defs.h"
6156 #include "param.h"
6157 #include "fs.h"
6158 #include "file.h"
6159 #include "spinlock.h"
6160
6161 struct devsw devsw[NDEV];
6162 struct {
6163   struct spinlock lock;
6164   struct file file[NFILE];
6165 } ftable;
6166
6167 void
6168 fileinit(void)
6169 {
6170   initlock(&ftable.lock, "ftable");
6171 }
6172
6173 // Allocate a file structure.
6174 struct file*
6175 filealloc(void)
6176 {
6177   struct file *f;
6178
6179   acquire(&ftable.lock);
6180   for(f = ftable.file; f < ftable.file + NFILE; f++){
6181     if(f->ref == 0){
6182       f->ref = 1;
6183       release(&ftable.lock);
6184       return f;
6185     }
6186   }
6187   release(&ftable.lock);
6188   return 0;
6189 }
6190
6191
6192
6193
6194
6195
6196
6197
6198
6199
```

```
6200 // Increment ref count for file f.
6201 struct file*
6202 filedup(struct file *f)
6203 {
6204   acquire(&ftable.lock);
6205   if(f->ref < 1)
6206     panic("filedup");
6207   f->ref++;
6208   release(&ftable.lock);
6209   return f;
6210 }
6211
6212 // Close file f.  (Decrement ref count, close when reaches 0.)
6213 void
6214 fileclose(struct file *f)
6215 {
6216   struct file ff;
6217
6218   acquire(&ftable.lock);
6219   if(f->ref < 1)
6220     panic("fileclose");
6221   if(--f->ref > 0){
6222     release(&ftable.lock);
6223     return;
6224   }
6225   ff = *f;
6226   f->ref = 0;
6227   f->type = FD_NONE;
6228   release(&ftable.lock);
6229
6230   if(ff.type == FD_PIPE)
6231     pipeclose(ff.pipe, ff.writable);
6232   else if(ff.type == FD_INODE){
6233     begin_op();
6234     iput(ff.ip);
6235     end_op();
6236   }
6237 }
6238
6239
6240
6241
6242
6243
6244
6245
6246
6247
6248
6249
```



```
6250 // Get metadata about file f.
6251 int
6252 filestat(struct file *f, struct stat *st)
6253 {
6254   if(f->type == FD_INODE){
6255     ilock(f->ip);
6256     stati(f->ip, st);
6257     iunlock(f->ip);
6258     return 0;
6259   }
6260   return -1;
6261 }
6262
6263 // Read from file f.
6264 int
6265 fileread(struct file *f, char *addr, int n)
6266 {
6267   int r;
6268
6269   if(f->readable == 0)
6270     return -1;
6271   if(f->type == FD_PIPE)
6272     return piperead(f->pipe, addr, n);
6273   if(f->type == FD_INODE){
6274     ilock(f->ip);
6275     if((r = readi(f->ip, addr, f->off, n)) > 0)
6276       f->off += r;
6277     iunlock(f->ip);
6278     return r;
6279   }
6280   panic("fileread");
6281 }
6282
6283
6284
6285
6286
6287
6288
6289
6290
6291
6292
6293
6294
6295
6296
6297
6298
6299
```

```
6300 // Write to file f.
6301 int
6302 filewrite(struct file *f, char *addr, int n)
6303 {
6304   int r;
6305
6306   if(f->writable == 0)
6307     return -1;
6308   if(f->type == FD_PIPE)
6309     return pipewrite(f->pipe, addr, n);
6310   if(f->type == FD_INODE){
6311     // write a few blocks at a time to avoid exceeding
6312     // the maximum log transaction size, including
6313     // i-node, indirect block, allocation blocks,
6314     // and 2 blocks of slop for non-aligned writes.
6315     // this really belongs lower down, since writei()
6316     // might be writing a device like the console.
6317     int max = ((LOGSIZE-1-1-2) / 2) * 512;
6318     int i = 0;
6319     while(i < n){
6320       int n1 = n - i;
6321       if(n1 > max)
6322         n1 = max;
6323
6324       begin_op();
6325       ilock(f->ip);
6326       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
6327         f->off += r;
6328       iunlock(f->ip);
6329       end_op();
6330
6331       if(r < 0)
6332         break;
6333       if(r != n1)
6334         panic("short filewrite");
6335       i += r;
6336     }
6337     return i == n ? n : -1;
6338   }
6339   panic("filewrite");
6340 }
6341
6342
6343
6344
6345
6346
6347
6348
6349
```



```
6350 //
6351 // File-system system calls.
6352 // Mostly argument checking, since we don't trust
6353 // user code, and calls into file.c and fs.c.
6354 //
6355
6356 #include "types.h"
6357 #include "defs.h"
6358 #include "param.h"
6359 #include "stat.h"
6360 #include "mmu.h"
6361 #include "proc.h"
6362 #include "fs.h"
6363 #include "file.h"
6364 #include "fcntl.h"
6365
6366 // Fetch the nth word-sized system call argument as a file descriptor
6367 // and return both the descriptor and the corresponding struct file.
6368 static int
6369 argfd(int n, int *pfd, struct file **pf)
6370 {
6371   int fd;
6372   struct file *f;
6373
6374   if(argint(n, &fd) < 0)
6375     return -1;
6376   if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
6377     return -1;
6378   if(pfd)
6379     *pfd = fd;
6380   if(pf)
6381     *pf = f;
6382   return 0;
6383 }
6384
6385 // Allocate a file descriptor for the given file.
6386 // Takes over file reference from caller on success.
6387 static int
6388 fdalloc(struct file *f)
6389 {
6390   int fd;
6391
6392   for(fd = 0; fd < NOFILE; fd++){
6393     if(proc->ofile[fd] == 0){
6394       proc->ofile[fd] = f;
6395       return fd;
6396     }
6397   }
6398   return -1;
6399 }
```

```
6400 int
6401 sys_dup(void)
6402 {
6403   struct file *f;
6404   int fd;
6405
6406   if(argfd(0, 0, &f) < 0)
6407     return -1;
6408   if((fd=fdalloc(f)) < 0)
6409     return -1;
6410   filedup(f);
6411   return fd;
6412 }
6413
6414 int
6415 sys_read(void)
6416 {
6417   struct file *f;
6418   int n;
6419   char *p;
6420
6421   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6422     return -1;
6423   return fileread(f, p, n);
6424 }
6425
6426 int
6427 sys_write(void)
6428 {
6429   struct file *f;
6430   int n;
6431   char *p;
6432
6433   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6434     return -1;
6435   return filewrite(f, p, n);
6436 }
6437
6438 int
6439 sys_close(void)
6440 {
6441   int fd;
6442   struct file *f;
6443
6444   if(argfd(0, &fd, &f) < 0)
6445     return -1;
6446   proc->ofile[fd] = 0;
6447   fileclose(f);
6448   return 0;
6449 }
```



```
6450 int
6451 sys_fstat(void)
6452 {
6453   struct file *f;
6454   struct stat *st;
6455
6456   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
6457     return -1;
6458   return filestat(f, st);
6459 }
6460
6461 // Create the path new as a link to the same inode as old.
6462 int
6463 sys_link(void)
6464 {
6465   char name[DIRSIZ], *new, *old;
6466   struct inode *dp, *ip;
6467
6468   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
6469     return -1;
6470
6471   begin_op();
6472   if((ip = namei(old)) == 0){
6473     end_op();
6474     return -1;
6475   }
6476
6477   ilock(ip);
6478   if(ip->type == T_DIR){
6479     iunlockput(ip);
6480     end_op();
6481     return -1;
6482   }
6483
6484   ip->nlink++;
6485   iupdate(ip);
6486   iunlock(ip);
6487
6488   if((dp = nameiparent(new, name)) == 0)
6489     goto bad;
6490   ilock(dp);
6491   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
6492     iunlockput(dp);
6493     goto bad;
6494   }
6495   iunlockput(dp);
6496   iput(ip);
6497
6498   end_op();
6499
```

```
6500   return 0;
6501
6502 bad:
6503   ilock(ip);
6504   ip->nlink--;
6505   iupdate(ip);
6506   iunlockput(ip);
6507   end_op();
6508   return -1;
6509 }
6510
6511 // Is the directory dp empty except for "." and ".." ?
6512 static int
6513 isdirempty(struct inode *dp)
6514 {
6515   int off;
6516   struct dirent de;
6517
6518   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
6519     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6520       panic("isdirempty: readi");
6521     if(de.inum != 0)
6522       return 0;
6523   }
6524   return 1;
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 int
6551 sys_unlink(void)
6552 {
6553   struct inode *ip, *dp;
6554   struct dirent de;
6555   char name[DIRSIZ], *path;
6556   uint off;
6557
6558   if(argstr(0, &path) < 0)
6559     return -1;
6560
6561   begin_op();
6562   if((dp = nameiparent(path, name)) == 0){
6563     end_op();
6564     return -1;
6565   }
6566
6567   ilock(dp);
6568
6569   // Cannot unlink "." or "..".
6570   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6571     goto bad;
6572
6573   if((ip = dirlookup(dp, name, &off)) == 0)
6574     goto bad;
6575   ilock(ip);
6576
6577   if(ip->nlink < 1)
6578     panic("unlink: nlink < 1");
6579   if(ip->type == T_DIR && !isdirempty(ip)){
6580     iunlockput(ip);
6581     goto bad;
6582   }
6583
6584   memset(&de, 0, sizeof(de));
6585   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6586     panic("unlink: writei");
6587   if(ip->type == T_DIR){
6588     dp->nlink--;
6589     iupdate(dp);
6590   }
6591   iunlockput(dp);
6592
6593   ip->nlink--;
6594   iupdate(ip);
6595   iunlockput(ip);
6596
6597   end_op();
6598
6599   return 0;
```

```
6600 bad:
6601   iunlockput(dp);
6602   end_op();
6603   return -1;
6604 }
6605
6606 static struct inode*
6607 create(char *path, short type, short major, short minor)
6608 {
6609   uint off;
6610   struct inode *ip, *dp;
6611   char name[DIRSIZ];
6612
6613   if((dp = nameiparent(path, name)) == 0)
6614     return 0;
6615   ilock(dp);
6616
6617   if((ip = dirlookup(dp, name, &off)) != 0){
6618     iunlockput(dp);
6619     ilock(ip);
6620     if(type == T_FILE && ip->type == T_FILE)
6621       return ip;
6622     iunlockput(ip);
6623     return 0;
6624   }
6625
6626   if((ip = ialloc(dp->dev, type)) == 0)
6627     panic("create: ialloc");
6628
6629   ilock(ip);
6630   ip->major = major;
6631   ip->minor = minor;
6632   ip->nlink = 1;
6633   iupdate(ip);
6634
6635   if(type == T_DIR){  // Create . and .. entries.
6636     dp->nlink++;  // for ".."
6637     iupdate(dp);
6638     // No ip->nlink++ for ".": avoid cyclic ref count.
6639     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6640       panic("create dots");
6641   }
6642
6643   if(dirlink(dp, name, ip->inum) < 0)
6644     panic("create: dirlink");
6645
6646   iunlockput(dp);
6647
6648   return ip;
6649 }
```



```
6650 int
6651 sys_open(void)
6652 {
6653   char *path;
6654   int fd, omode;
6655   struct file *f;
6656   struct inode *ip;
6657
6658   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6659     return -1;
6660
6661   begin_op();
6662
6663   if(omode & O_CREATE){
6664     ip = create(path, T_FILE, 0, 0);
6665     if(ip == 0){
6666       end_op();
6667       return -1;
6668     }
6669   } else {
6670     if((ip = namei(path)) == 0){
6671       end_op();
6672       return -1;
6673     }
6674     ilock(ip);
6675     if(ip->type == T_DIR && omode != O_RDONLY){
6676       iunlockput(ip);
6677       end_op();
6678       return -1;
6679     }
6680   }
6681
6682   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6683     if(f)
6684       fileclose(f);
6685     iunlockput(ip);
6686     end_op();
6687     return -1;
6688   }
6689   iunlock(ip);
6690   end_op();
6691
6692   f->type = FD_INODE;
6693   f->ip = ip;
6694   f->off = 0;
6695   f->readable = !(omode & O_WRONLY);
6696   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6697   return fd;
6698 }
6699
```

```
6700 int
6701 sys_mkdir(void)
6702 {
6703   char *path;
6704   struct inode *ip;
6705
6706   begin_op();
6707   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
6708     end_op();
6709     return -1;
6710   }
6711   iunlockput(ip);
6712   end_op();
6713   return 0;
6714 }
6715
6716 int
6717 sys_mknod(void)
6718 {
6719   struct inode *ip;
6720   char *path;
6721   int len;
6722   int major, minor;
6723
6724   begin_op();
6725   if((len=argstr(0, &path)) < 0 ||
6726      argint(1, &major) < 0 ||
6727      argint(2, &minor) < 0 ||
6728      (ip = create(path, T_DEV, major, minor)) == 0){
6729     end_op();
6730     return -1;
6731   }
6732   iunlockput(ip);
6733   end_op();
6734   return 0;
6735 }
6736
6737
6738
6739
6740
6741
6742
6743
6744
6745
6746
6747
6748
6749
```



```
6750 int
6751 sys_chdir(void)
6752 {
6753   char *path;
6754   struct inode *ip;
6755
6756   begin_op();
6757   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
6758     end_op();
6759     return -1;
6760   }
6761   ilock(ip);
6762   if(ip->type != T_DIR){
6763     iunlockput(ip);
6764     end_op();
6765     return -1;
6766   }
6767   iunlock(ip);
6768   iput(proc->cwd);
6769   end_op();
6770   proc->cwd = ip;
6771   return 0;
6772 }
6773
6774 int
6775 sys_exec(void)
6776 {
6777   char *path, *argv[MAXARG];
6778   int i;
6779   uint uargv, uarg;
6780
6781   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6782     return -1;
6783   }
6784   memset(argv, 0, sizeof(argv));
6785   for(i=0;; i++){
6786     if(i >= NELEM(argv))
6787       return -1;
6788     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6789       return -1;
6790     if(uarg == 0){
6791       argv[i] = 0;
6792       break;
6793     }
6794     if(fetchstr(uarg, &argv[i]) < 0)
6795       return -1;
6796   }
6797   return exec(path, argv);
6798 }
6799
```

```
6800 int
6801 sys_pipe(void)
6802 {
6803   int *fd;
6804   struct file *rf, *wf;
6805   int fd0, fd1;
6806
6807   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6808     return -1;
6809   if(pipealloc(&rf, &wf) < 0)
6810     return -1;
6811   fd0 = -1;
6812   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6813     if(fd0 >= 0)
6814       proc->ofile[fd0] = 0;
6815     fileclose(rf);
6816     fileclose(wf);
6817     return -1;
6818   }
6819   fd[0] = fd0;
6820   fd[1] = fd1;
6821   return 0;
6822 }
6823
6824
6825
6826
6827
6828
6829
6830
6831
6832
6833
6834
6835
6836
6837
6838
6839
6840
6841
6842
6843
6844
6845
6846
6847
6848
6849
```



```
6850 #include "types.h"
6851 #include "param.h"
6852 #include "memlayout.h"
6853 #include "mmu.h"
6854 #include "proc.h"
6855 #include "defs.h"
6856 #include "x86.h"
6857 #include "elf.h"
6858
6859 int
6860 exec(char *path, char **argv)
6861 {
6862   char *s, *last;
6863   int i, off;
6864   uint argc, sz, sp, ustack[3+MAXARG+1];
6865   struct elfhdr elf;
6866   struct inode *ip;
6867   struct proghdr ph;
6868   pde_t *pgdir, *oldpgdir;
6869
6870   begin_op();
6871   if((ip = namei(path)) == 0){
6872     end_op();
6873     return -1;
6874   }
6875   ilock(ip);
6876   pgdir = 0;
6877
6878   // Check ELF header
6879   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
6880     goto bad;
6881   if(elf.magic != ELF_MAGIC)
6882     goto bad;
6883
6884   if((pgdir = setupkvm()) == 0)
6885     goto bad;
6886
6887   // Load program into memory.
6888   sz = 0;
6889   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6890     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6891       goto bad;
6892     if(ph.type != ELF_PROG_LOAD)
6893       continue;
6894     if(ph.memsz < ph.filesz)
6895       goto bad;
6896     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6897       goto bad;
6898     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6899       goto bad;
```

```
6900   }
6901   iunlockput(ip);
6902   end_op();
6903   ip = 0;
6904
6905   // Allocate two pages at the next page boundary.
6906   // Make the first inaccessible.  Use the second as the user stack.
6907   sz = PGROUNDUP(sz);
6908   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
6909     goto bad;
6910   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6911   sp = sz;
6912
6913   // Push argument strings, prepare rest of stack in ustack.
6914   for(argc = 0; argv[argc]; argc++) {
6915     if(argc >= MAXARG)
6916       goto bad;
6917     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6918     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6919       goto bad;
6920     ustack[3+argc] = sp;
6921   }
6922   ustack[3+argc] = 0;
6923
6924   ustack[0] = 0xffffffff;  // fake return PC
6925   ustack[1] = argc;
6926   ustack[2] = sp - (argc+1)*4;  // argv pointer
6927
6928   sp -= (3+argc+1) * 4;
6929   if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6930     goto bad;
6931
6932   // Save program name for debugging.
6933   for(last=s=path; *s; s++)
6934     if(*s == '/')
6935       last = s+1;
6936   safestrcpy(proc->name, last, sizeof(proc->name));
6937
6938   // Commit to the user image.
6939   oldpgdir = proc->pgdir;
6940   proc->pgdir = pgdir;
6941   proc->sz = sz;
6942   proc->tf->eip = elf.entry;  // main
6943   proc->tf->esp = sp;
6944   switchuvm(proc);
6945   freevm(oldpgdir);
6946   return 0;
6947
6948
6949
```



```
6950  bad:
6951    if(pgdir)
6952      freevm(pgdir);
6953    if(ip){
6954      iunlockput(ip);
6955      end_op();
6956    }
6957    return -1;
6958 }
6959
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 #include "types.h"
7001 #include "defs.h"
7002 #include "param.h"
7003 #include "mmu.h"
7004 #include "proc.h"
7005 #include "fs.h"
7006 #include "file.h"
7007 #include "spinlock.h"
7008
7009 #define PIPESIZE 512
7010
7011 struct pipe {
7012   struct spinlock lock;
7013   char data[PIPESIZE];
7014   uint nread;     // number of bytes read
7015   uint nwrite;    // number of bytes written
7016   int readopen;   // read fd is still open
7017   int writeopen;  // write fd is still open
7018 };
7019
7020 int
7021 pipealloc(struct file **f0, struct file **f1)
7022 {
7023   struct pipe *p;
7024
7025   p = 0;
7026   *f0 = *f1 = 0;
7027   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
7028     goto bad;
7029   if((p = (struct pipe*)kalloc()) == 0)
7030     goto bad;
7031   p->readopen = 1;
7032   p->writeopen = 1;
7033   p->nwrite = 0;
7034   p->nread = 0;
7035   initlock(&p->lock, "pipe");
7036   (*f0)->type = FD_PIPE;
7037   (*f0)->readable = 1;
7038   (*f0)->writable = 0;
7039   (*f0)->pipe = p;
7040   (*f1)->type = FD_PIPE;
7041   (*f1)->readable = 0;
7042   (*f1)->writable = 1;
7043   (*f1)->pipe = p;
7044   return 0;
7045
7046
7047
7048
7049
```



```
7050  bad:
7051   if(p)
7052     kfree((char*)p);
7053   if(*f0)
7054     fileclose(*f0);
7055   if(*f1)
7056     fileclose(*f1);
7057   return -1;
7058 }
7059
7060 void
7061 pipeclose(struct pipe *p, int writable)
7062 {
7063   acquire(&p->lock);
7064   if(writable){
7065     p->writeopen = 0;
7066     wakeup(&p->nread);
7067   } else {
7068     p->readopen = 0;
7069     wakeup(&p->nwrite);
7070   }
7071   if(p->readopen == 0 && p->writeopen == 0){
7072     release(&p->lock);
7073     kfree((char*)p);
7074   } else
7075     release(&p->lock);
7076 }
7077
7078
7079 int
7080 pipewrite(struct pipe *p, char *addr, int n)
7081 {
7082   int i;
7083
7084   acquire(&p->lock);
7085   for(i = 0; i < n; i++){
7086     while(p->nwrite == p->nread + PIPESIZE){
7087       if(p->readopen == 0 || proc->killed){
7088         release(&p->lock);
7089         return -1;
7090       }
7091       wakeup(&p->nread);
7092       sleep(&p->nwrite, &p->lock);
7093     }
7094     p->data[p->nwrite++ % PIPESIZE] = addr[i];
7095   }
7096   wakeup(&p->nread);
7097   release(&p->lock);
7098   return n;
7099 }
```

```
7100 int
7101 piperead(struct pipe *p, char *addr, int n)
7102 {
7103   int i;
7104
7105   acquire(&p->lock);
7106   while(p->nread == p->nwrite && p->writeopen){
7107     if(proc->killed){
7108       release(&p->lock);
7109       return -1;
7110     }
7111     sleep(&p->nread, &p->lock);
7112   }
7113   for(i = 0; i < n; i++){
7114     if(p->nread == p->nwrite)
7115       break;
7116     addr[i] = p->data[p->nread++ % PIPESIZE];
7117   }
7118   wakeup(&p->nwrite);
7119   release(&p->lock);
7120   return i;
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 #include "types.h"
7151 #include "x86.h"
7152
7153 void*
7154 memset(void *dst, int c, uint n)
7155 {
7156   if ((int)dst%4 == 0 && n%4 == 0){
7157     c &= 0xFF;
7158     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
7159   } else
7160     stosb(dst, c, n);
7161   return dst;
7162 }
7163
7164 int
7165 memcmp(const void *v1, const void *v2, uint n)
7166 {
7167   const uchar *s1, *s2;
7168
7169   s1 = v1;
7170   s2 = v2;
7171   while(n-- > 0){
7172     if(*s1 != *s2)
7173       return *s1 - *s2;
7174     s1++, s2++;
7175   }
7176
7177   return 0;
7178 }
7179
7180 void*
7181 memmove(void *dst, const void *src, uint n)
7182 {
7183   const char *s;
7184   char *d;
7185
7186   s = src;
7187   d = dst;
7188   if(s < d && s + n > d){
7189     s += n;
7190     d += n;
7191     while(n-- > 0)
7192       *--d = *--s;
7193   } else
7194     while(n-- > 0)
7195       *d++ = *s++;
7196
7197   return dst;
7198 }
7199
```

```
7200 // memcpy exists to placate GCC.  Use memmove.
7201 void*
7202 memcpy(void *dst, const void *src, uint n)
7203 {
7204   return memmove(dst, src, n);
7205 }
7206
7207 int
7208 strncmp(const char *p, const char *q, uint n)
7209 {
7210   while(n > 0 && *p && *p == *q)
7211     n--, p++, q++;
7212   if(n == 0)
7213     return 0;
7214   return (uchar)*p - (uchar)*q;
7215 }
7216
7217 char*
7218 strncpy(char *s, const char *t, int n)
7219 {
7220   char *os;
7221
7222   os = s;
7223   while(n-- > 0 && (*s++ = *t++) != 0)
7224     ;
7225   while(n-- > 0)
7226     *s++ = 0;
7227   return os;
7228 }
7229
7230 // Like strncpy but guaranteed to NUL-terminate.
7231 char*
7232 safestrcpy(char *s, const char *t, int n)
7233 {
7234   char *os;
7235
7236   os = s;
7237   if(n <= 0)
7238     return os;
7239   while(--n > 0 && (*s++ = *t++) != 0)
7240     ;
7241   *s = 0;
7242   return os;
7243 }
7244
7245
7246
7247
7248
7249
```



```
7250 int
7251 strlen(const char *s)
7252 {
7253   int n;
7254
7255   for(n = 0; s[n]; n++)
7256     ;
7257   return n;
7258 }
7259
7260
7261
7262
7263
7264
7265
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
```

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

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

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

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

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



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

```
7400 // Blank page.
7401
7402
7403
7404
7405
7406
7407
7408
7409
7410
7411
7412
7413
7414
7415
7416
7417
7418
7419
7420
7421
7422
7423
7424
7425
7426
7427
7428
7429
7430
7431
7432
7433
7434
7435
7436
7437
7438
7439
7440
7441
7442
7443
7444
7445
7446
7447
7448
7449
```



```
7450 // Multiprocessor support
7451 // Search memory for MP description structures.
7452 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
7453
7454 #include "types.h"
7455 #include "defs.h"
7456 #include "param.h"
7457 #include "memlayout.h"
7458 #include "mp.h"
7459 #include "x86.h"
7460 #include "mmu.h"
7461 #include "proc.h"
7462
7463 struct cpu cpus[NCPU];
7464 static struct cpu *bcpu;
7465 int ismp;
7466 int ncpu;
7467 uchar ioapicid;
7468
7469 int
7470 mpbcpu(void)
7471 {
7472   return bcpu-cpus;
7473 }
7474
7475 static uchar
7476 sum(uchar *addr, int len)
7477 {
7478   int i, sum;
7479
7480   sum = 0;
7481   for(i=0; i<len; i++)
7482     sum += addr[i];
7483   return sum;
7484 }
7485
7486 // Look for an MP structure in the len bytes at addr.
7487 static struct mp*
7488 mpsearch1(uint a, int len)
7489 {
7490   uchar *e, *p, *addr;
7491
7492   addr = p2v(a);
7493   e = addr+len;
7494   for(p = addr; p < e; p += sizeof(struct mp))
7495     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
7496       return (struct mp*)p;
7497   return 0;
7498 }
7499
```

```
7500 // Search for the MP Floating Pointer Structure, which according to the
7501 // spec is in one of the following three locations:
7502 // 1) in the first KB of the EBDA;
7503 // 2) in the last KB of system base memory;
7504 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
7505 static struct mp*
7506 mpsearch(void)
7507 {
7508   uchar *bda;
7509   uint p;
7510   struct mp *mp;
7511
7512   bda = (uchar *) P2V(0x400);
7513   if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
7514     if((mp = mpsearch1(p, 1024)))
7515       return mp;
7516   } else {
7517     p = ((bda[0x14]<<8)|bda[0x13])*1024;
7518     if((mp = mpsearch1(p-1024, 1024)))
7519       return mp;
7520   }
7521   return mpsearch1(0xF0000, 0x10000);
7522 }
7523
7524 // Search for an MP configuration table.  For now,
7525 // don't accept the default configurations (physaddr == 0).
7526 // Check for correct signature, calculate the checksum and,
7527 // if correct, check the version.
7528 // To do: check extended table checksum.
7529 static struct mpconf*
7530 mpconfig(struct mp **pmp)
7531 {
7532   struct mpconf *conf;
7533   struct mp *mp;
7534
7535   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
7536     return 0;
7537   conf = (struct mpconf*) p2v((uint) mp->physaddr);
7538   if(memcmp(conf, "PCMP", 4) != 0)
7539     return 0;
7540   if(conf->version != 1 && conf->version != 4)
7541     return 0;
7542   if(sum((uchar*)conf, conf->length) != 0)
7543     return 0;
7544   *pmp = mp;
7545   return conf;
7546 }
7547
7548
7549
```



```
7550 void
7551 mpinit(void)
7552 {
7553   uchar *p, *e;
7554   struct mp *mp;
7555   struct mpconf *conf;
7556   struct mpproc *proc;
7557   struct mpioapic *ioapic;
7558
7559   bcpu = &cpus[0];
7560   if((conf = mpconfig(&mp)) == 0)
7561     return;
7562   ismp = 1;
7563   lapic = (uint*)conf->lapicaddr;
7564   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
7565     switch(*p){
7566     case MPPROC:
7567       proc = (struct mpproc*)p;
7568       if(ncpu != proc->apicid){
7569         cprintf("mpinit: ncpu=%d apicid=%d\n", ncpu, proc->apicid);
7570         ismp = 0;
7571       }
7572       if(proc->flags & MPBOOT)
7573         bcpu = &cpus[ncpu];
7574       cpus[ncpu].id = ncpu;
7575       ncpu++;
7576       p += sizeof(struct mpproc);
7577       continue;
7578     case MPIOAPIC:
7579       ioapic = (struct mpioapic*)p;
7580       ioapicid = ioapic->apicno;
7581       p += sizeof(struct mpioapic);
7582       continue;
7583     case MPBUS:
7584     case MPIOINTR:
7585     case MPLINTR:
7586       p += 8;
7587       continue;
7588     default:
7589       cprintf("mpinit: unknown config type %x\n", *p);
7590       ismp = 0;
7591     }
7592   }
7593   if(!ismp){
7594     // Didn't like what we found; fall back to no MP.
7595     ncpu = 1;
7596     lapic = 0;
7597     ioapicid = 0;
7598     return;
7599   }
```

```
7600   if(mp->imcrp){
7601     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
7602     // But it would on real hardware.
7603     outb(0x22, 0x70);   // Select IMCR
7604     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
7605   }
7606 }
7607
7608
7609
7610
7611
7612
7613
7614
7615
7616
7617
7618
7619
7620
7621
7622
7623
7624
7625
7626
7627
7628
7629
7630
7631
7632
7633
7634
7635
7636
7637
7638
7639
7640
7641
7642
7643
7644
7645
7646
7647
7648
7649
```



```
7650 // The local APIC manages internal (non-I/O) interrupts.
7651 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
7652
7653 #include "types.h"
7654 #include "defs.h"
7655 #include "date.h"
7656 #include "memlayout.h"
7657 #include "traps.h"
7658 #include "mmu.h"
7659 #include "x86.h"
7660
7661 // Local APIC registers, divided by 4 for use as uint[] indices.
7662 #define ID      (0x0020/4)   // ID
7663 #define VER     (0x0030/4)   // Version
7664 #define TPR     (0x0080/4)   // Task Priority
7665 #define EOI     (0x00B0/4)   // EOI
7666 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
7667   #define ENABLE     0x00000100   // Unit Enable
7668 #define ESR     (0x0280/4)   // Error Status
7669 #define ICRLO   (0x0300/4)   // Interrupt Command
7670   #define INIT       0x00000500   // INIT/RESET
7671   #define STARTUP    0x00000600   // Startup IPI
7672   #define DELIVS     0x00001000   // Delivery status
7673   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
7674   #define DEASSERT   0x00000000
7675   #define LEVEL      0x00008000   // Level triggered
7676   #define BCAST      0x00080000   // Send to all APICs, including self.
7677   #define BUSY       0x00001000
7678   #define FIXED      0x00000000
7679 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
7680 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
7681   #define X1         0x0000000B   // divide counts by 1
7682   #define PERIODIC   0x00020000   // Periodic
7683 #define PCINT   (0x0340/4)   // Performance Counter LVT
7684 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
7685 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
7686 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
7687   #define MASKED     0x00010000   // Interrupt masked
7688 #define TICR    (0x0380/4)   // Timer Initial Count
7689 #define TCCR    (0x0390/4)   // Timer Current Count
7690 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
7691
7692 volatile uint *lapic;  // Initialized in mp.c
7693
7694 static void
7695 lapicw(int index, int value)
7696 {
7697   lapic[index] = value;
7698   lapic[ID];  // wait for write to finish, by reading
7699 }
```

```
7750 void
7751 lapicinit(void)
7752 {
7753   if(!lapic)
7754     return;
7755
7756   // Enable local APIC; set spurious interrupt vector.
7757   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
7758
7759   // The timer repeatedly counts down at bus frequency
7760   // from lapic[TICR] and then issues an interrupt.
7761   // If xv6 cared more about precise timekeeping,
7762   // TICR would be calibrated using an external time source.
7763   lapicw(TDCR, X1);
7764   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
7765   lapicw(TICR, 10000000);
7766
7767   // Disable logical interrupt lines.
7768   lapicw(LINT0, MASKED);
7769   lapicw(LINT1, MASKED);
7770
7771   // Disable performance counter overflow interrupts
7772   // on machines that provide that interrupt entry.
7773   if(((lapic[VER]>>16) & 0xFF) >= 4)
7774     lapicw(PCINT, MASKED);
7775
7776   // Map error interrupt to IRQ_ERROR.
7777   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
7778
7779   // Clear error status register (requires back-to-back writes).
7780   lapicw(ESR, 0);
7781   lapicw(ESR, 0);
7782
7783   // Ack any outstanding interrupts.
7784   lapicw(EOI, 0);
7785
7786   // Send an Init Level De-Assert to synchronise arbitration ID's.
7787   lapicw(ICRHI, 0);
7788   lapicw(ICRLO, BCAST | INIT | LEVEL);
7789   while(lapic[ICRLO] & DELIVS)
7790     ;
7791
7792   // Enable interrupts on the APIC (but not on the processor).
7793   lapicw(TPR, 0);
7794 }
7795
7796
7797
7798
7799
```

```
7800 int
7801 cpunum(void)
7802 {
7803   // Cannot call cpu when interrupts are enabled:
7804   // result not guaranteed to last long enough to be used!
7805   // Would prefer to panic but even printing is chancy here:
7806   // almost everything, including cprintf and panic, calls cpu,
7807   // often indirectly through acquire and release.
7808   if(readeflags()&FL_IF){
7809     static int n;
7810     if(n++ == 0)
7811       cprintf("cpu called from %x with interrupts enabled\n",
7812         __builtin_return_address(0));
7813   }
7814
7815   if(lapic)
7816     return lapic[ID]>>24;
7817   return 0;
7818 }
7819
7820 // Acknowledge interrupt.
7821 void
7822 lapiceoi(void)
7823 {
7824   if(lapic)
7825     lapicw(EOI, 0);
7826 }
7827
7828 // Spin for a given number of microseconds.
7829 // On real hardware would want to tune this dynamically.
7830 void
7831 microdelay(int us)
7832 {
7833 }
7834
7835 #define CMOS_PORT    0x70
7836 #define CMOS_RETURN  0x71
7837
7838 // Start additional processor running entry code at addr.
7839 // See Appendix B of MultiProcessor Specification.
7840 void
7841 lapicstartap(uchar apicid, uint addr)
7842 {
7843   int i;
7844   ushort *wrv;
7845
7846   // "The BSP must initialize CMOS shutdown code to 0AH
7847   // and the warm reset vector (DWORD based at 40:67) to point at
7848   // the AP startup code prior to the [universal startup algorithm]."
7849   outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
```



```
7850   outb(CMOS_PORT+1, 0x0A);
7851   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
7852   wrv[0] = 0;
7853   wrv[1] = addr >> 4;
7854
7855   // "Universal startup algorithm."
7856   // Send INIT (level-triggered) interrupt to reset other CPU.
7857   lapicw(ICRHI, apicid<<24);
7858   lapicw(ICRLO, INIT | LEVEL | ASSERT);
7859   microdelay(200);
7860   lapicw(ICRLO, INIT | LEVEL);
7861   microdelay(100);    // should be 10ms, but too slow in Bochs!
7862
7863   // Send startup IPI (twice!) to enter code.
7864   // Regular hardware is supposed to only accept a STARTUP
7865   // when it is in the halted state due to an INIT.  So the second
7866   // should be ignored, but it is part of the official Intel algorithm.
7867   // Bochs complains about the second one.  Too bad for Bochs.
7868   for(i = 0; i < 2; i++){
7869     lapicw(ICRHI, apicid<<24);
7870     lapicw(ICRLO, STARTUP | (addr>>12));
7871     microdelay(200);
7872   }
7873 }
7874
7875 #define CMOS_STATA   0x0a
7876 #define CMOS_STATB   0x0b
7877 #define CMOS_UIP    (1 << 7)        // RTC update in progress
7878
7879 #define SECS    0x00
7880 #define MINS    0x02
7881 #define HOURS   0x04
7882 #define DAY     0x07
7883 #define MONTH   0x08
7884 #define YEAR    0x09
7885
7886 static uint cmos_read(uint reg)
7887 {
7888   outb(CMOS_PORT,  reg);
7889   microdelay(200);
7890
7891   return inb(CMOS_RETURN);
7892 }
7893
7894
7895
7896
7897
7898
7899
```

```
7900 static void fill_rtcdate(struct rtcdate *r)
7901 {
7902   r->second = cmos_read(SECS);
7903   r->minute = cmos_read(MINS);
7904   r->hour   = cmos_read(HOURS);
7905   r->day    = cmos_read(DAY);
7906   r->month  = cmos_read(MONTH);
7907   r->year   = cmos_read(YEAR);
7908 }
7909
7910 // qemu seems to use 24-hour GWT and the values are BCD encoded
7911 void cmostime(struct rtcdate *r)
7912 {
7913   struct rtcdate t1, t2;
7914   int sb, bcd;
7915
7916   sb = cmos_read(CMOS_STATB);
7917
7918   bcd = (sb & (1 << 2)) == 0;
7919
7920   // make sure CMOS doesn't modify time while we read it
7921   for (;;) {
7922     fill_rtcdate(&t1);
7923     if (cmos_read(CMOS_STATA) & CMOS_UIP)
7924         continue;
7925     fill_rtcdate(&t2);
7926     if (memcmp(&t1, &t2, sizeof(t1)) == 0)
7927       break;
7928   }
7929
7930   // convert
7931   if (bcd) {
7932 #define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
7933     CONV(second);
7934     CONV(minute);
7935     CONV(hour  );
7936     CONV(day   );
7937     CONV(month );
7938     CONV(year  );
7939 #undef     CONV
7940   }
7941
7942   *r = t1;
7943   r->year += 2000;
7944 }
7945
7946
7947
7948
7949
```



```
7950 // The I/O APIC manages hardware interrupts for an SMP system.
7951 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
7952 // See also picirq.c.
7953
7954 #include "types.h"
7955 #include "defs.h"
7956 #include "traps.h"
7957
7958 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
7959
7960 #define REG_ID     0x00  // Register index: ID
7961 #define REG_VER    0x01  // Register index: version
7962 #define REG_TABLE  0x10  // Redirection table base
7963
7964 // The redirection table starts at REG_TABLE and uses
7965 // two registers to configure each interrupt.
7966 // The first (low) register in a pair contains configuration bits.
7967 // The second (high) register contains a bitmask telling which
7968 // CPUs can serve that interrupt.
7969 #define INT_DISABLED   0x00010000  // Interrupt disabled
7970 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
7971 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
7972 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
7973
7974 volatile struct ioapic *ioapic;
7975
7976 // IO APIC MMIO structure: write reg, then read or write data.
7977 struct ioapic {
7978   uint reg;
7979   uint pad[3];
7980   uint data;
7981 };
7982
7983 static uint
7984 ioapicread(int reg)
7985 {
7986   ioapic->reg = reg;
7987   return ioapic->data;
7988 }
7989
7990 static void
7991 ioapicwrite(int reg, uint data)
7992 {
7993   ioapic->reg = reg;
7994   ioapic->data = data;
7995 }
7996
7997
7998
7999
```

```
8000 void
8001 ioapicinit(void)
8002 {
8003   int i, id, maxintr;
8004
8005   if(!ismp)
8006     return;
8007
8008   ioapic = (volatile struct ioapic*)IOAPIC;
8009   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
8010   id = ioapicread(REG_ID) >> 24;
8011   if(id != ioapicid)
8012     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
8013
8014   // Mark all interrupts edge-triggered, active high, disabled,
8015   // and not routed to any CPUs.
8016   for(i = 0; i <= maxintr; i++){
8017     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
8018     ioapicwrite(REG_TABLE+2*i+1, 0);
8019   }
8020 }
8021
8022 void
8023 ioapicenable(int irq, int cpunum)
8024 {
8025   if(!ismp)
8026     return;
8027
8028   // Mark interrupt edge-triggered, active high,
8029   // enabled, and routed to the given cpunum,
8030   // which happens to be that cpu's APIC ID.
8031   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
8032   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
8033 }
8034
8035
8036
8037
8038
8039
8040
8041
8042
8043
8044
8045
8046
8047
8048
8049
```



```
8050 // Intel 8259A programmable interrupt controllers.
8051
8052 #include "types.h"
8053 #include "x86.h"
8054 #include "traps.h"
8055
8056 // I/O Addresses of the two programmable interrupt controllers
8057 #define IO_PIC1         0x20    // Master (IRQs 0-7)
8058 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
8059
8060 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
8061
8062 // Current IRQ mask.
8063 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
8064 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
8065
8066 static void
8067 picsetmask(ushort mask)
8068 {
8069   irqmask = mask;
8070   outb(IO_PIC1+1, mask);
8071   outb(IO_PIC2+1, mask >> 8);
8072 }
8073
8074 void
8075 picenable(int irq)
8076 {
8077   picsetmask(irqmask & ~(1<<irq));
8078 }
8079
8080 // Initialize the 8259A interrupt controllers.
8081 void
8082 picinit(void)
8083 {
8084   // mask all interrupts
8085   outb(IO_PIC1+1, 0xFF);
8086   outb(IO_PIC2+1, 0xFF);
8087
8088   // Set up master (8259A-1)
8089
8090   // ICW1:  0001g0hi
8091   //    g:  0 = edge triggering, 1 = level triggering
8092   //    h:  0 = cascaded PICs, 1 = master only
8093   //    i:  0 = no ICW4, 1 = ICW4 required
8094   outb(IO_PIC1, 0x11);
8095
8096   // ICW2:  Vector offset
8097   outb(IO_PIC1+1, T_IRQ0);
8098
8099
```

```
8100   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
8101   //        (slave PIC) 3-bit # of slave's connection to master
8102   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
8103
8104   // ICW4:  000nbmap
8105   //    n:  1 = special fully nested mode
8106   //    b:  1 = buffered mode
8107   //    m:  0 = slave PIC, 1 = master PIC
8108   //      (ignored when b is 0, as the master/slave role
8109   //      can be hardwired).
8110   //    a:  1 = Automatic EOI mode
8111   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
8112   outb(IO_PIC1+1, 0x3);
8113
8114   // Set up slave (8259A-2)
8115   outb(IO_PIC2, 0x11);                  // ICW1
8116   outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
8117   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
8118   // NB Automatic EOI mode doesn't tend to work on the slave.
8119   // Linux source code says it's "to be investigated".
8120   outb(IO_PIC2+1, 0x3);                 // ICW4
8121
8122   // OCW3:  0ef01prs
8123   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
8124   //    p:  0 = no polling, 1 = polling mode
8125   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
8126   outb(IO_PIC1, 0x68);             // clear specific mask
8127   outb(IO_PIC1, 0x0a);             // read IRR by default
8128
8129   outb(IO_PIC2, 0x68);             // OCW3
8130   outb(IO_PIC2, 0x0a);             // OCW3
8131
8132   if(irqmask != 0xFFFF)
8133     picsetmask(irqmask);
8134 }
8135
8136
8137
8138
8139
8140
8141
8142
8143
8144
8145
8146
8147
8148
8149
```



```
8150 // PC keyboard interface constants
8151
8152 #define KBSTATP         0x64    // kbd controller status port(I)
8153 #define KBS_DIB         0x01    // kbd data in buffer
8154 #define KBDATAP         0x60    // kbd data port(I)
8155
8156 #define NO              0
8157
8158 #define SHIFT           (1<<0)
8159 #define CTL             (1<<1)
8160 #define ALT             (1<<2)
8161
8162 #define CAPSLOCK        (1<<3)
8163 #define NUMLOCK         (1<<4)
8164 #define SCROLLLOCK      (1<<5)
8165
8166 #define E0ESC           (1<<6)
8167
8168 // Special keycodes
8169 #define KEY_HOME        0xE0
8170 #define KEY_END         0xE1
8171 #define KEY_UP          0xE2
8172 #define KEY_DN          0xE3
8173 #define KEY_LF          0xE4
8174 #define KEY_RT          0xE5
8175 #define KEY_PGUP        0xE6
8176 #define KEY_PGDN        0xE7
8177 #define KEY_INS         0xE8
8178 #define KEY_DEL         0xE9
8179
8180 // C('A') == Control-A
8181 #define C(x) (x - '@')
8182
8183 static uchar shiftcode[256] =
8184 {
8185   [0x1D] CTL,
8186   [0x2A] SHIFT,
8187   [0x36] SHIFT,
8188   [0x38] ALT,
8189   [0x9D] CTL,
8190   [0xB8] ALT
8191 };
8192
8193 static uchar togglecode[256] =
8194 {
8195   [0x3A] CAPSLOCK,
8196   [0x45] NUMLOCK,
8197   [0x46] SCROLLLOCK
8198 };
8199
```

```
8200 static uchar normalmap[256] =
8201 {
8202   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
8203   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
8204   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
8205   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
8206   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
8207   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
8208   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
8209   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
8210   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
8211   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
8212   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
8213   [0x9C] '\n',      // KP_Enter
8214   [0xB5] '/',       // KP_Div
8215   [0xC8] KEY_UP,    [0xD0] KEY_DN,
8216   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
8217   [0xCB] KEY_LF,    [0xCD] KEY_RT,
8218   [0x97] KEY_HOME,  [0xCF] KEY_END,
8219   [0xD2] KEY_INS,   [0xD3] KEY_DEL
8220 };
8221
8222 static uchar shiftmap[256] =
8223 {
8224   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
8225   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
8226   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
8227   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
8228   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
8229   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
8230   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
8231   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
8232   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
8233   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
8234   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
8235   [0x9C] '\n',      // KP_Enter
8236   [0xB5] '/',       // KP_Div
8237   [0xC8] KEY_UP,    [0xD0] KEY_DN,
8238   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
8239   [0xCB] KEY_LF,    [0xCD] KEY_RT,
8240   [0x97] KEY_HOME,  [0xCF] KEY_END,
8241   [0xD2] KEY_INS,   [0xD3] KEY_DEL
8242 };
8243
8244
8245
8246
8247
8248
8249
```



```
8250 static uchar ctlmap[256] =
8251 {
8252   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
8253   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
8254   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
8255   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
8256   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
8257   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
8258   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
8259   [0x9C] '\r',      // KP_Enter
8260   [0xB5] C('/'),    // KP_Div
8261   [0xC8] KEY_UP,    [0xD0] KEY_DN,
8262   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
8263   [0xCB] KEY_LF,    [0xCD] KEY_RT,
8264   [0x97] KEY_HOME,  [0xCF] KEY_END,
8265   [0xD2] KEY_INS,   [0xD3] KEY_DEL
8266 };
8267
8268
8269
8270
8271
8272
8273
8274
8275
8276
8277
8278
8279
8280
8281
8282
8283
8284
8285
8286
8287
8288
8289
8290
8291
8292
8293
8294
8295
8296
8297
8298
8299
```

```
8300 #include "types.h"
8301 #include "x86.h"
8302 #include "defs.h"
8303 #include "kbd.h"
8304
8305 int
8306 kbdgetc(void)
8307 {
8308   static uint shift;
8309   static uchar *charcode[4] = {
8310     normalmap, shiftmap, ctlmap, ctlmap
8311   };
8312   uint st, data, c;
8313
8314   st = inb(KBSTATP);
8315   if((st & KBS_DIB) == 0)
8316     return -1;
8317   data = inb(KBDATAP);
8318
8319   if(data == 0xE0){
8320     shift |= E0ESC;
8321     return 0;
8322   } else if(data & 0x80){
8323     // Key released
8324     data = (shift & E0ESC ? data : data & 0x7F);
8325     shift &= ~(shiftcode[data] | E0ESC);
8326     return 0;
8327   } else if(shift & E0ESC){
8328     // Last character was an E0 escape; or with 0x80
8329     data |= 0x80;
8330     shift &= ~E0ESC;
8331   }
8332
8333   shift |= shiftcode[data];
8334   shift ^= togglecode[data];
8335   c = charcode[shift & (CTL | SHIFT)][data];
8336   if(shift & CAPSLOCK){
8337     if('a' <= c && c <= 'z')
8338       c += 'A' - 'a';
8339     else if('A' <= c && c <= 'Z')
8340       c += 'a' - 'A';
8341   }
8342   return c;
8343 }
8344
8345 void
8346 kbdintr(void)
8347 {
8348   consoleintr(kbdgetc);
8349 }
```



```
8350 // Console input and output.
8351 // Input is from the keyboard or serial port.
8352 // Output is written to the screen and serial port.
8353
8354 #include "types.h"
8355 #include "defs.h"
8356 #include "param.h"
8357 #include "traps.h"
8358 #include "spinlock.h"
8359 #include "fs.h"
8360 #include "file.h"
8361 #include "memlayout.h"
8362 #include "mmu.h"
8363 #include "proc.h"
8364 #include "x86.h"
8365
8366 static void consputc(int);
8367
8368 static int panicked = 0;
8369
8370 static struct {
8371   struct spinlock lock;
8372   int locking;
8373 } cons;
8374
8375 static void
8376 printint(int xx, int base, int sign)
8377 {
8378   static char digits[] = "0123456789abcdef";
8379   char buf[16];
8380   int i;
8381   uint x;
8382
8383   if(sign && (sign = xx < 0))
8384     x = -xx;
8385   else
8386     x = xx;
8387
8388   i = 0;
8389   do{
8390     buf[i++] = digits[x % base];
8391   }while((x /= base) != 0);
8392
8393   if(sign)
8394     buf[i++] = '-';
8395
8396   while(--i >= 0)
8397     consputc(buf[i]);
8398 }
8399
```

```
8400 // Print to the console. only understands %d, %x, %p, %s.
8401 void
8402 cprintf(char *fmt, ...)
8403 {
8404   int i, c, locking;
8405   uint *argp;
8406   char *s;
8407
8408   locking = cons.locking;
8409   if(locking)
8410     acquire(&cons.lock);
8411
8412   if (fmt == 0)
8413     panic("null fmt");
8414
8415   argp = (uint*)(void*)(&fmt + 1);
8416   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
8417     if(c != '%'){
8418       consputc(c);
8419       continue;
8420     }
8421     c = fmt[++i] & 0xff;
8422     if(c == 0)
8423       break;
8424     switch(c){
8425     case 'd':
8426       printint(*argp++, 10, 1);
8427       break;
8428     case 'x':
8429     case 'p':
8430       printint(*argp++, 16, 0);
8431       break;
8432     case 's':
8433       if((s = (char*)*argp++) == 0)
8434         s = "(null)";
8435       for(; *s; s++)
8436         consputc(*s);
8437       break;
8438     case '%':
8439       consputc('%');
8440       break;
8441     default:
8442       // Print unknown % sequence to draw attention.
8443       consputc('%');
8444       consputc(c);
8445       break;
8446     }
8447   }
8448
8449
```



```
8450   if(locking)
8451     release(&cons.lock);
8452 }
8453
8454 void
8455 panic(char *s)
8456 {
8457   int i;
8458   uint pcs[10];
8459
8460   cli();
8461   cons.locking = 0;
8462   cprintf("cpu%d: panic: ", cpu->id);
8463   cprintf(s);
8464   cprintf("\n");
8465   getcallerpcs(&s, pcs);
8466   for(i=0; i<10; i++)
8467     cprintf(" %p", pcs[i]);
8468   panicked = 1; // freeze other CPU
8469   for(;;)
8470     ;
8471 }
8472
8473
8474
8475
8476
8477
8478
8479
8480
8481
8482
8483
8484
8485
8486
8487
8488
8489
8490
8491
8492
8493
8494
8495
8496
8497
8498
8499
```

```
8500 #define BACKSPACE 0x100
8501 #define CRTPORT 0x3d4
8502 static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory
8503
8504 static void
8505 cgaputc(int c)
8506 {
8507   int pos;
8508
8509   // Cursor position: col + 80*row.
8510   outb(CRTPORT, 14);
8511   pos = inb(CRTPORT+1) << 8;
8512   outb(CRTPORT, 15);
8513   pos |= inb(CRTPORT+1);
8514
8515   if(c == '\n')
8516     pos += 80 - pos%80;
8517   else if(c == BACKSPACE){
8518     if(pos > 0) --pos;
8519   } else
8520     crt[pos++] = (c&0xff) | 0x0700;  // black on white
8521
8522   if(pos < 0 || pos > 25*80)
8523     panic("pos under/overflow");
8524
8525   if((pos/80) >= 24){  // Scroll up.
8526     memmove(crt, crt+80, sizeof(crt[0])*23*80);
8527     pos -= 80;
8528     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
8529   }
8530
8531   outb(CRTPORT, 14);
8532   outb(CRTPORT+1, pos>>8);
8533   outb(CRTPORT, 15);
8534   outb(CRTPORT+1, pos);
8535   crt[pos] = ' ' | 0x0700;
8536 }
8537
8538
8539
8540
8541
8542
8543
8544
8545
8546
8547
8548
8549
```



```
8550 void
8551 consputc(int c)
8552 {
8553   if(panicked){
8554     cli();
8555     for(;;)
8556       ;
8557   }
8558
8559   if(c == BACKSPACE){
8560     uartputc('\b'); uartputc(' '); uartputc('\b');
8561   } else
8562     uartputc(c);
8563   cgaputc(c);
8564 }
8565
8566 #define INPUT_BUF 128
8567 struct {
8568   char buf[INPUT_BUF];
8569   uint r;  // Read index
8570   uint w;  // Write index
8571   uint e;  // Edit index
8572 } input;
8573
8574 #define C(x)  ((x)-'@')  // Control-x
8575
8576 void
8577 consoleintr(int (*getc)(void))
8578 {
8579   int c, doprocdump = 0;
8580
8581   acquire(&cons.lock);
8582   while((c = getc()) >= 0){
8583     switch(c){
8584     case C('P'):  // Process listing.
8585       doprocdump = 1;   // procdump() locks cons.lock indirectly; invoke lat
8586       break;
8587     case C('U'):  // Kill line.
8588       while(input.e != input.w &&
8589             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
8590         input.e--;
8591         consputc(BACKSPACE);
8592       }
8593       break;
8594     case C('H'): case '\x7f':  // Backspace
8595       if(input.e != input.w){
8596         input.e--;
8597         consputc(BACKSPACE);
8598       }
8599       break;
```

```
8600     default:
8601       if(c != 0 && input.e-input.r < INPUT_BUF){
8602         c = (c == '\r') ? '\n' : c;
8603         input.buf[input.e++ % INPUT_BUF] = c;
8604         consputc(c);
8605         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
8606           input.w = input.e;
8607           wakeup(&input.r);
8608         }
8609       }
8610       break;
8611     }
8612   }
8613   release(&cons.lock);
8614   if(doprocdump) {
8615     procdump();  // now call procdump() wo. cons.lock held
8616   }
8617 }
8618
8619 int
8620 consoleread(struct inode *ip, char *dst, int n)
8621 {
8622   uint target;
8623   int c;
8624
8625   iunlock(ip);
8626   target = n;
8627   acquire(&cons.lock);
8628   while(n > 0){
8629     while(input.r == input.w){
8630       if(proc->killed){
8631         release(&cons.lock);
8632         ilock(ip);
8633         return -1;
8634       }
8635       sleep(&input.r, &cons.lock);
8636     }
8637     c = input.buf[input.r++ % INPUT_BUF];
8638     if(c == C('D')){  // EOF
8639       if(n < target){
8640         // Save ^D for next time, to make sure
8641         // caller gets a 0-byte result.
8642         input.r--;
8643       }
8644       break;
8645     }
8646     *dst++ = c;
8647     --n;
8648     if(c == '\n')
8649       break;
```



```
8650   }
8651   release(&cons.lock);
8652   ilock(ip);
8653
8654   return target - n;
8655 }
8656
8657 int
8658 consolewrite(struct inode *ip, char *buf, int n)
8659 {
8660   int i;
8661
8662   iunlock(ip);
8663   acquire(&cons.lock);
8664   for(i = 0; i < n; i++)
8665     consputc(buf[i] & 0xff);
8666   release(&cons.lock);
8667   ilock(ip);
8668
8669   return n;
8670 }
8671
8672 void
8673 consoleinit(void)
8674 {
8675   initlock(&cons.lock, "console");
8676
8677   devsw[CONSOLE].write = consolewrite;
8678   devsw[CONSOLE].read = consoleread;
8679   cons.locking = 1;
8680
8681   picenable(IRQ_KBD);
8682   ioapicenable(IRQ_KBD, 0);
8683 }
8684
8685
8686
8687
8688
8689
8690
8691
8692
8693
8694
8695
8696
8697
8698
8699
```

```
8700 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
8701 // Only used on uniprocessors;
8702 // SMP machines use the local APIC timer.
8703
8704 #include "types.h"
8705 #include "defs.h"
8706 #include "traps.h"
8707 #include "x86.h"
8708
8709 #define IO_TIMER1       0x040           // 8253 Timer #1
8710
8711 // Frequency of all three count-down timers;
8712 // (TIMER_FREQ/freq) is the appropriate count
8713 // to generate a frequency of freq Hz.
8714
8715 #define TIMER_FREQ      1193182
8716 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
8717
8718 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
8719 #define TIMER_SEL0      0x00    // select counter 0
8720 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
8721 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
8722
8723 void
8724 timerinit(void)
8725 {
8726   // Interrupt 100 times/sec.
8727   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
8728   outb(IO_TIMER1, TIMER_DIV(100) % 256);
8729   outb(IO_TIMER1, TIMER_DIV(100) / 256);
8730   picenable(IRQ_TIMER);
8731 }
8732
8733
8734
8735
8736
8737
8738
8739
8740
8741
8742
8743
8744
8745
8746
8747
8748
8749
```



```
8750 // Intel 8250 serial port (UART).
8751
8752 #include "types.h"
8753 #include "defs.h"
8754 #include "param.h"
8755 #include "traps.h"
8756 #include "spinlock.h"
8757 #include "fs.h"
8758 #include "file.h"
8759 #include "mmu.h"
8760 #include "proc.h"
8761 #include "x86.h"
8762
8763 #define COM1    0x3f8
8764
8765 static int uart;    // is there a uart?
8766
8767 void
8768 uartinit(void)
8769 {
8770   char *p;
8771
8772   // Turn off the FIFO
8773   outb(COM1+2, 0);
8774
8775   // 9600 baud, 8 data bits, 1 stop bit, parity off.
8776   outb(COM1+3, 0x80);    // Unlock divisor
8777   outb(COM1+0, 115200/9600);
8778   outb(COM1+1, 0);
8779   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
8780   outb(COM1+4, 0);
8781   outb(COM1+1, 0x01);    // Enable receive interrupts.
8782
8783   // If status is 0xFF, no serial port.
8784   if(inb(COM1+5) == 0xFF)
8785     return;
8786   uart = 1;
8787
8788   // Acknowledge pre-existing interrupt conditions;
8789   // enable interrupts.
8790   inb(COM1+2);
8791   inb(COM1+0);
8792   picenable(IRQ_COM1);
8793   ioapicenable(IRQ_COM1, 0);
8794
8795   // Announce that we're here.
8796   for(p="xv6...\n"; *p; p++)
8797     uartputc(*p);
8798 }
8799
```

```
8800 void
8801 uartputc(int c)
8802 {
8803   int i;
8804
8805   if(!uart)
8806     return;
8807   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
8808     microdelay(10);
8809   outb(COM1+0, c);
8810 }
8811
8812 static int
8813 uartgetc(void)
8814 {
8815   if(!uart)
8816     return -1;
8817   if(!(inb(COM1+5) & 0x01))
8818     return -1;
8819   return inb(COM1+0);
8820 }
8821
8822 void
8823 uartintr(void)
8824 {
8825   consoleintr(uartgetc);
8826 }
8827
8828
8829
8830
8831
8832
8833
8834
8835
8836
8837
8838
8839
8840
8841
8842
8843
8844
8845
8846
8847
8848
8849
```



```
8850 # Initial process execs /init.
8851
8852 #include "syscall.h"
8853 #include "traps.h"
8854
8855
8856 # exec(init, argv)
8857 .globl start
8858 start:
8859   pushl $argv
8860   pushl $init
8861   pushl $0  // where caller pc would be
8862   movl $SYS_exec, %eax
8863   int $T_SYSCALL
8864
8865 # for(;;) exit();
8866 exit:
8867   movl $SYS_exit, %eax
8868   int $T_SYSCALL
8869   jmp exit
8870
8871 # char init[] = "/init\0";
8872 init:
8873   .string "/init\0"
8874
8875 # char *argv[] = { init, 0 };
8876 .p2align 2
8877 argv:
8878   .long init
8879   .long 0
8880
8881
8882
8883
8884
8885
8886
8887
8888
8889
8890
8891
8892
8893
8894
8895
8896
8897
8898
8899
```

```
8900 #include "syscall.h"
8901 #include "traps.h"
8902
8903 #define SYSCALL(name) \
8904   .globl name; \
8905   name: \
8906     movl $SYS_ ## name, %eax; \
8907     int $T_SYSCALL; \
8908     ret
8909
8910 SYSCALL(fork)
8911 SYSCALL(exit)
8912 SYSCALL(wait)
8913 SYSCALL(pipe)
8914 SYSCALL(read)
8915 SYSCALL(write)
8916 SYSCALL(close)
8917 SYSCALL(kill)
8918 SYSCALL(exec)
8919 SYSCALL(open)
8920 SYSCALL(mknod)
8921 SYSCALL(unlink)
8922 SYSCALL(fstat)
8923 SYSCALL(link)
8924 SYSCALL(mkdir)
8925 SYSCALL(chdir)
8926 SYSCALL(dup)
8927 SYSCALL(getpid)
8928 SYSCALL(sbrk)
8929 SYSCALL(sleep)
8930 SYSCALL(uptime)
8931 SYSCALL(halt)
8932 SYSCALL(date)
8933
8934 # Project 3
8935 SYSCALL(getuid)
8936 SYSCALL(getgid)
8937 SYSCALL(getppid)
8938 SYSCALL(setuid)
8939 SYSCALL(setgid)
8940 SYSCALL(getprocs)
8941
8942 # Project 4
8943 SYSCALL(setPriority)
8944
8945
8946
8947
8948
8949
```



```
8950 // init: The initial user-level program
8951
8952 #include "types.h"
8953 #include "stat.h"
8954 #include "user.h"
8955 #include "fcntl.h"
8956
8957 char *argv[] = { "sh", 0 };
8958
8959 int
8960 main(void)
8961 {
8962   int pid, wpid;
8963
8964   if(open("console", O_RDWR) < 0){
8965     mknod("console", 1, 1);
8966     open("console", O_RDWR);
8967   }
8968   dup(0);  // stdout
8969   dup(0);  // stderr
8970
8971   for(;;){
8972     printf(1, "init: starting sh\n");
8973     pid = fork();
8974     if(pid < 0){
8975       printf(1, "init: fork failed\n");
8976       exit();
8977     }
8978     if(pid == 0){
8979       exec("sh", argv);
8980       printf(1, "init: exec sh failed\n");
8981       exit();
8982     }
8983     while((wpid=wait()) >= 0 && wpid != pid)
8984       printf(1, "zombie!\n");
8985   }
8986 }
8987
8988
8989
8990
8991
8992
8993
8994
8995
8996
8997
8998
8999
```

```
9000 // Shell.
9001 // 2015-12-21. Added very simple processing for builtin commands
9002
9003 #include "types.h"
9004 #include "user.h"
9005 #include "fcntl.h"
9006
9007 // Parsed command representation
9008 #define EXEC  1
9009 #define REDIR 2
9010 #define PIPE  3
9011 #define LIST  4
9012 #define BACK  5
9013
9014 #define MAXARGS 10
9015
9016 struct cmd {
9017   int type;
9018 };
9019
9020 struct execcmd {
9021   int type;
9022   char *argv[MAXARGS];
9023   char *eargv[MAXARGS];
9024 };
9025
9026 struct redircmd {
9027   int type;
9028   struct cmd *cmd;
9029   char *file;
9030   char *efile;
9031   int mode;
9032   int fd;
9033 };
9034
9035 struct pipecmd {
9036   int type;
9037   struct cmd *left;
9038   struct cmd *right;
9039 };
9040
9041 struct listcmd {
9042   int type;
9043   struct cmd *left;
9044   struct cmd *right;
9045 };
9046
9047
9048
9049
```



```
9050 struct backcmd {
9051   int type;
9052   struct cmd *cmd;
9053 };
9054
9055 int fork1(void);  // Fork but panics on failure.
9056 void panic(char*);
9057 struct cmd *parsecmd(char*);
9058
9059 // Execute cmd.  Never returns.
9060 void
9061 runcmd(struct cmd *cmd)
9062 {
9063   int p[2];
9064   struct backcmd *bcmd;
9065   struct execcmd *ecmd;
9066   struct listcmd *lcmd;
9067   struct pipecmd *pcmd;
9068   struct redircmd *rcmd;
9069
9070   if(cmd == 0)
9071     exit();
9072
9073   switch(cmd->type){
9074   default:
9075     panic("runcmd");
9076
9077   case EXEC:
9078     ecmd = (struct execcmd*)cmd;
9079     if(ecmd->argv[0] == 0)
9080       exit();
9081     exec(ecmd->argv[0], ecmd->argv);
9082     printf(2, "exec %s failed\n", ecmd->argv[0]);
9083     break;
9084
9085   case REDIR:
9086     rcmd = (struct redircmd*)cmd;
9087     close(rcmd->fd);
9088     if(open(rcmd->file, rcmd->mode) < 0){
9089       printf(2, "open %s failed\n", rcmd->file);
9090       exit();
9091     }
9092     runcmd(rcmd->cmd);
9093     break;
9094
9095   case LIST:
9096     lcmd = (struct listcmd*)cmd;
9097     if(fork1() == 0)
9098       runcmd(lcmd->left);
9099     wait();
```

```
9100     runcmd(lcmd->right);
9101     break;
9102
9103   case PIPE:
9104     pcmd = (struct pipecmd*)cmd;
9105     if(pipe(p) < 0)
9106       panic("pipe");
9107     if(fork1() == 0){
9108       close(1);
9109       dup(p[1]);
9110       close(p[0]);
9111       close(p[1]);
9112       runcmd(pcmd->left);
9113     }
9114     if(fork1() == 0){
9115       close(0);
9116       dup(p[0]);
9117       close(p[0]);
9118       close(p[1]);
9119       runcmd(pcmd->right);
9120     }
9121     close(p[0]);
9122     close(p[1]);
9123     wait();
9124     wait();
9125     break;
9126
9127   case BACK:
9128     bcmd = (struct backcmd*)cmd;
9129     if(fork1() == 0)
9130       runcmd(bcmd->cmd);
9131     break;
9132   }
9133   exit();
9134 }
9135
9136 int
9137 getcmd(char *buf, int nbuf)
9138 {
9139   printf(2, "$ ");
9140   memset(buf, 0, nbuf);
9141   gets(buf, nbuf);
9142   if(buf[0] == 0) // EOF
9143     return -1;
9144   return 0;
9145 }
9146
9147
9148
9149
```



```
9150 // ***** processing for shell builtins begins here *****
9151
9152 int
9153 strncmp(const char *p, const char *q, uint n)
9154 {
9155     while(n > 0 && *p && *p == *q)
9156       n--, p++, q++;
9157     if(n == 0)
9158       return 0;
9159     return (uchar)*p - (uchar)*q;
9160 }
9161
9162 int
9163 makeint(char *p)
9164 {
9165   int val = 0;
9166
9167   while ((*p >= '0') && (*p <= '9')) {
9168     val = 10*val + (*p-'0');
9169     ++p;
9170   }
9171   return val;
9172 }
9173
9174 int
9175 setbuiltin(char *p)
9176 {
9177   int i;
9178
9179   p += strlen("_set");
9180   while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
9181   if (strncmp("uid", p, 3) == 0) {
9182     p += strlen("uid");
9183     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
9184     i = makeint(p); // ugly
9185     return (setuid(i));
9186   } else
9187   if (strncmp("gid", p, 3) == 0) {
9188     p += strlen("gid");
9189     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
9190     i = makeint(p); // ugly
9191     return (setgid(i));
9192   }
9193   printf(2, "Invalid _set parameter\n");
9194   return -1;
9195 }
9196
9197
9198
9199
```

```
9200 int
9201 getbuiltin(char *p)
9202 {
9203   p += strlen("_get");
9204   while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
9205   if (strncmp("uid", p, 3) == 0) {
9206     printf(2, "%d\n", getuid());
9207     return 0;
9208   }
9209   if (strncmp("gid", p, 3) == 0) {
9210     printf(2, "%d\n", getgid());
9211     return 0;
9212   }
9213   printf(2, "Invalid _get parameter\n");
9214   return -1;
9215 }
9216
9217 typedef int funcPtr_t(char *);
9218 typedef struct {
9219   char       *cmd;
9220   funcPtr_t  *name;
9221 } dispatchTableEntry_t;
9222
9223 // Use a simple function dispatch table (FDT) to process builtin commands
9224 dispatchTableEntry_t fdt[] = {
9225   {"_set", setbuiltin},
9226   {"_get", getbuiltin}
9227 };
9228 int FDTcount = sizeof(fdt) / sizeof(fdt[0]); // # entris in FDT
9229
9230 void
9231 dobuiltin(char *cmd) {
9232   int i;
9233
9234   for (i=0; i<FDTcount; i++)
9235     if (strncmp(cmd, fdt[i].cmd, strlen(fdt[i].cmd)) == 0)
9236      (*fdt[i].name)(cmd);
9237 }
9238
9239
9240
9241
9242
9243
9244
9245
9246
9247
9248
9249
```



```
9250 // ***** processing for shell builtins ends here *****
9251
9252 int
9253 main(void)
9254 {
9255   static char buf[100];
9256   int fd;
9257
9258   // Assumes three file descriptors open.
9259   while((fd = open("console", O_RDWR)) >= 0){
9260     if(fd >= 3){
9261       close(fd);
9262       break;
9263     }
9264   }
9265
9266   // Read and run input commands.
9267   while(getcmd(buf, sizeof(buf)) >= 0){
9268 // add support for built-ins here. cd is a built-in
9269     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
9270       // Clumsy but will have to do for now.
9271       // Chdir has no effect on the parent if run in the child.
9272       buf[strlen(buf)-1] = 0;  // chop \n
9273       if(chdir(buf+3) < 0)
9274         printf(2, "cannot cd %s\n", buf+3);
9275       continue;
9276     }
9277     if (buf[0]=='_') {     // assume it is a builtin command
9278       dobuiltin(buf);
9279       continue;
9280     }
9281     if(fork1() == 0)
9282       runcmd(parsecmd(buf));
9283     wait();
9284   }
9285   exit();
9286 }
9287
9288 void
9289 panic(char *s)
9290 {
9291   printf(2, "%s\n", s);
9292   exit();
9293 }
9294
9295
9296
9297
9298
9299
```

```
9300 int
9301 fork1(void)
9302 {
9303   int pid;
9304
9305   pid = fork();
9306   if(pid == -1)
9307     panic("fork");
9308   return pid;
9309 }
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 // Constructors
9351
9352 struct cmd*
9353 execcmd(void)
9354 {
9355   struct execcmd *cmd;
9356
9357   cmd = malloc(sizeof(*cmd));
9358   memset(cmd, 0, sizeof(*cmd));
9359   cmd->type = EXEC;
9360   return (struct cmd*)cmd;
9361 }
9362
9363 struct cmd*
9364 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
9365 {
9366   struct redircmd *cmd;
9367
9368   cmd = malloc(sizeof(*cmd));
9369   memset(cmd, 0, sizeof(*cmd));
9370   cmd->type = REDIR;
9371   cmd->cmd = subcmd;
9372   cmd->file = file;
9373   cmd->efile = efile;
9374   cmd->mode = mode;
9375   cmd->fd = fd;
9376   return (struct cmd*)cmd;
9377 }
9378
9379 struct cmd*
9380 pipecmd(struct cmd *left, struct cmd *right)
9381 {
9382   struct pipecmd *cmd;
9383
9384   cmd = malloc(sizeof(*cmd));
9385   memset(cmd, 0, sizeof(*cmd));
9386   cmd->type = PIPE;
9387   cmd->left = left;
9388   cmd->right = right;
9389   return (struct cmd*)cmd;
9390 }
9391
9392
9393
9394
9395
9396
9397
9398
9399
```

```
9400 struct cmd*
9401 listcmd(struct cmd *left, struct cmd *right)
9402 {
9403   struct listcmd *cmd;
9404
9405   cmd = malloc(sizeof(*cmd));
9406   memset(cmd, 0, sizeof(*cmd));
9407   cmd->type = LIST;
9408   cmd->left = left;
9409   cmd->right = right;
9410   return (struct cmd*)cmd;
9411 }
9412
9413 struct cmd*
9414 backcmd(struct cmd *subcmd)
9415 {
9416   struct backcmd *cmd;
9417
9418   cmd = malloc(sizeof(*cmd));
9419   memset(cmd, 0, sizeof(*cmd));
9420   cmd->type = BACK;
9421   cmd->cmd = subcmd;
9422   return (struct cmd*)cmd;
9423 }
9424
9425
9426
9427
9428
9429
9430
9431
9432
9433
9434
9435
9436
9437
9438
9439
9440
9441
9442
9443
9444
9445
9446
9447
9448
9449
```



```
9450 // Parsing
9451
9452 char whitespace[] = " \t\r\n\v";
9453 char symbols[] = "<|>&;()";
9454
9455 int
9456 gettoken(char **ps, char *es, char **q, char **eq)
9457 {
9458   char *s;
9459   int ret;
9460
9461   s = *ps;
9462   while(s < es && strchr(whitespace, *s))
9463     s++;
9464   if(q)
9465     *q = s;
9466   ret = *s;
9467   switch(*s){
9468   case 0:
9469     break;
9470   case '|':
9471   case '(':
9472   case ')':
9473   case ';':
9474   case '&':
9475   case '<':
9476     s++;
9477     break;
9478   case '>':
9479     s++;
9480     if(*s == '>'){
9481       ret = '+';
9482       s++;
9483     }
9484     break;
9485   default:
9486     ret = 'a';
9487     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
9488       s++;
9489     break;
9490   }
9491   if(eq)
9492     *eq = s;
9493
9494   while(s < es && strchr(whitespace, *s))
9495     s++;
9496   *ps = s;
9497   return ret;
9498 }
9499
```

```
9500 int
9501 peek(char **ps, char *es, char *toks)
9502 {
9503   char *s;
9504
9505   s = *ps;
9506   while(s < es && strchr(whitespace, *s))
9507     s++;
9508   *ps = s;
9509   return *s && strchr(toks, *s);
9510 }
9511
9512 struct cmd *parseline(char**, char*);
9513 struct cmd *parsepipe(char**, char*);
9514 struct cmd *parseexec(char**, char*);
9515 struct cmd *nulterminate(struct cmd*);
9516
9517 struct cmd*
9518 parsecmd(char *s)
9519 {
9520   char *es;
9521   struct cmd *cmd;
9522
9523   es = s + strlen(s);
9524   cmd = parseline(&s, es);
9525   peek(&s, es, "");
9526   if(s != es){
9527     printf(2, "leftovers: %s\n", s);
9528     panic("syntax");
9529   }
9530   nulterminate(cmd);
9531   return cmd;
9532 }
9533
9534 struct cmd*
9535 parseline(char **ps, char *es)
9536 {
9537   struct cmd *cmd;
9538
9539   cmd = parsepipe(ps, es);
9540   while(peek(ps, es, "&")){
9541     gettoken(ps, es, 0, 0);
9542     cmd = backcmd(cmd);
9543   }
9544   if(peek(ps, es, ";")){
9545     gettoken(ps, es, 0, 0);
9546     cmd = listcmd(cmd, parseline(ps, es));
9547   }
9548   return cmd;
9549 }
```



```
9550 struct cmd*
9551 parsepipe(char **ps, char *es)
9552 {
9553   struct cmd *cmd;
9554
9555   cmd = parseexec(ps, es);
9556   if(peek(ps, es, "|")){
9557     gettoken(ps, es, 0, 0);
9558     cmd = pipecmd(cmd, parsepipe(ps, es));
9559   }
9560   return cmd;
9561 }
9562
9563 struct cmd*
9564 parseredirs(struct cmd *cmd, char **ps, char *es)
9565 {
9566   int tok;
9567   char *q, *eq;
9568
9569   while(peek(ps, es, "<>")){
9570     tok = gettoken(ps, es, 0, 0);
9571     if(gettoken(ps, es, &q, &eq) != 'a')
9572       panic("missing file for redirection");
9573     switch(tok){
9574     case '<':
9575       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
9576       break;
9577     case '>':
9578       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9579       break;
9580     case '+':  // >>
9581       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9582       break;
9583     }
9584   }
9585   return cmd;
9586 }
9587
9588
9589
9590
9591
9592
9593
9594
9595
9596
9597
9598
9599
```

```
9600 struct cmd*
9601 parseblock(char **ps, char *es)
9602 {
9603   struct cmd *cmd;
9604
9605   if(!peek(ps, es, "("))
9606     panic("parseblock");
9607   gettoken(ps, es, 0, 0);
9608   cmd = parseline(ps, es);
9609   if(!peek(ps, es, ")"))
9610     panic("syntax - missing )");
9611   gettoken(ps, es, 0, 0);
9612   cmd = parseredirs(cmd, ps, es);
9613   return cmd;
9614 }
9615
9616 struct cmd*
9617 parseexec(char **ps, char *es)
9618 {
9619   char *q, *eq;
9620   int tok, argc;
9621   struct execcmd *cmd;
9622   struct cmd *ret;
9623
9624   if(peek(ps, es, "("))
9625     return parseblock(ps, es);
9626
9627   ret = execcmd();
9628   cmd = (struct execcmd*)ret;
9629
9630   argc = 0;
9631   ret = parseredirs(ret, ps, es);
9632   while(!peek(ps, es, "|)&;")){
9633     if((tok=gettoken(ps, es, &q, &eq)) == 0)
9634       break;
9635     if(tok != 'a')
9636       panic("syntax");
9637     cmd->argv[argc] = q;
9638     cmd->eargv[argc] = eq;
9639     argc++;
9640     if(argc >= MAXARGS)
9641       panic("too many args");
9642     ret = parseredirs(ret, ps, es);
9643   }
9644   cmd->argv[argc] = 0;
9645   cmd->eargv[argc] = 0;
9646   return ret;
9647 }
9648
9649
```



```
9650 // NUL-terminate all the counted strings.
9651 struct cmd*
9652 nulterminate(struct cmd *cmd)
9653 {
9654   int i;
9655   struct backcmd *bcmd;
9656   struct execcmd *ecmd;
9657   struct listcmd *lcmd;
9658   struct pipecmd *pcmd;
9659   struct redircmd *rcmd;
9660
9661   if(cmd == 0)
9662     return 0;
9663
9664   switch(cmd->type){
9665   case EXEC:
9666     ecmd = (struct execcmd*)cmd;
9667     for(i=0; ecmd->argv[i]; i++)
9668       *ecmd->eargv[i] = 0;
9669     break;
9670
9671   case REDIR:
9672     rcmd = (struct redircmd*)cmd;
9673     nulterminate(rcmd->cmd);
9674     *rcmd->efile = 0;
9675     break;
9676
9677   case PIPE:
9678     pcmd = (struct pipecmd*)cmd;
9679     nulterminate(pcmd->left);
9680     nulterminate(pcmd->right);
9681     break;
9682
9683   case LIST:
9684     lcmd = (struct listcmd*)cmd;
9685     nulterminate(lcmd->left);
9686     nulterminate(lcmd->right);
9687     break;
9688
9689   case BACK:
9690     bcmd = (struct backcmd*)cmd;
9691     nulterminate(bcmd->cmd);
9692     break;
9693   }
9694   return cmd;
9695 }
9696
9697
9698
9699
```

```
9700 #include "asm.h"
9701 #include "memlayout.h"
9702 #include "mmu.h"
9703
9704 # Start the first CPU: switch to 32-bit protected mode, jump into C.
9705 # The BIOS loads this code from the first sector of the hard disk into
9706 # memory at physical address 0x7c00 and starts executing in real mode
9707 # with %cs=0 %ip=7c00.
9708
9709 .code16                       # Assemble for 16-bit mode
9710 .globl start
9711 start:
9712   cli                         # BIOS enabled interrupts; disable
9713
9714   # Zero data segment registers DS, ES, and SS.
9715   xorw    %ax,%ax             # Set %ax to zero
9716   movw    %ax,%ds             # -> Data Segment
9717   movw    %ax,%es             # -> Extra Segment
9718   movw    %ax,%ss             # -> Stack Segment
9719
9720   # Physical address line A20 is tied to zero so that the first PCs
9721   # with 2 MB would run software that assumed 1 MB.  Undo that.
9722 seta20.1:
9723   inb     $0x64,%al               # Wait for not busy
9724   testb   $0x2,%al
9725   jnz     seta20.1
9726
9727   movb    $0xd1,%al               # 0xd1 -> port 0x64
9728   outb    %al,$0x64
9729
9730 seta20.2:
9731   inb     $0x64,%al               # Wait for not busy
9732   testb   $0x2,%al
9733   jnz     seta20.2
9734
9735   movb    $0xdf,%al               # 0xdf -> port 0x60
9736   outb    %al,$0x60
9737
9738   # Switch from real to protected mode.  Use a bootstrap GDT that makes
9739   # virtual addresses map directly to physical addresses so that the
9740   # effective memory map doesn't change during the transition.
9741   lgdt    gdtdesc
9742   movl    %cr0, %eax
9743   orl     $CR0_PE, %eax
9744   movl    %eax, %cr0
9745
9746
9747
9748
9749
```



```
9750   # Complete transition to 32-bit protected mode by using long jmp
9751   # to reload %cs and %eip.  The segment descriptors are set up with no
9752   # translation, so that the mapping is still the identity mapping.
9753   ljmp    $(SEG_KCODE<<3), $start32
9754
9755 .code32  # Tell assembler to generate 32-bit code now.
9756 start32:
9757   # Set up the protected-mode data segment registers
9758   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
9759   movw    %ax, %ds                # -> DS: Data Segment
9760   movw    %ax, %es                # -> ES: Extra Segment
9761   movw    %ax, %ss                # -> SS: Stack Segment
9762   movw    $0, %ax                 # Zero segments not ready for use
9763   movw    %ax, %fs                # -> FS
9764   movw    %ax, %gs                # -> GS
9765
9766   # Set up the stack pointer and call into C.
9767   movl    $start, %esp
9768   call    bootmain
9769
9770   # If bootmain returns (it shouldn't), trigger a Bochs
9771   # breakpoint if running under Bochs, then loop.
9772   movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
9773   movw    %ax, %dx
9774   outw    %ax, %dx
9775   movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
9776   outw    %ax, %dx
9777 spin:
9778   jmp     spin
9779
9780 # Bootstrap GDT
9781 .p2align 2                                # force 4 byte alignment
9782 gdt:
9783   SEG_NULLASM                             # null seg
9784   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
9785   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
9786
9787 gdtdesc:
9788   .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
9789   .long   gdt                             # address gdt
9790
9791
9792
9793
9794
9795
9796
9797
9798
9799
```

```
9800 // Boot loader.
9801 //
9802 // Part of the boot block, along with bootasm.S, which calls bootmain().
9803 // bootasm.S has put the processor into protected 32-bit mode.
9804 // bootmain() loads an ELF kernel image from the disk starting at
9805 // sector 1 and then jumps to the kernel entry routine.
9806
9807 #include "types.h"
9808 #include "elf.h"
9809 #include "x86.h"
9810 #include "memlayout.h"
9811
9812 #define SECTSIZE  512
9813
9814 void readseg(uchar*, uint, uint);
9815
9816 void
9817 bootmain(void)
9818 {
9819   struct elfhdr *elf;
9820   struct proghdr *ph, *eph;
9821   void (*entry)(void);
9822   uchar* pa;
9823
9824   elf = (struct elfhdr*)0x10000;  // scratch space
9825
9826   // Read 1st page off disk
9827   readseg((uchar*)elf, 4096, 0);
9828
9829   // Is this an ELF executable?
9830   if(elf->magic != ELF_MAGIC)
9831     return;  // let bootasm.S handle error
9832
9833   // Load each program segment (ignores ph flags).
9834   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
9835   eph = ph + elf->phnum;
9836   for(; ph < eph; ph++){
9837     pa = (uchar*)ph->paddr;
9838     readseg(pa, ph->filesz, ph->off);
9839     if(ph->memsz > ph->filesz)
9840       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
9841   }
9842
9843   // Call the entry point from the ELF header.
9844   // Does not return!
9845   entry = (void(*)(void))(elf->entry);
9846   entry();
9847 }
9848
9849
```



```
9850 void
9851 waitdisk(void)
9852 {
9853   // Wait for disk ready.
9854   while((inb(0x1F7) & 0xC0) != 0x40)
9855     ;
9856 }
9857
9858 // Read a single sector at offset into dst.
9859 void
9860 readsect(void *dst, uint offset)
9861 {
9862   // Issue command.
9863   waitdisk();
9864   outb(0x1F2, 1);   // count = 1
9865   outb(0x1F3, offset);
9866   outb(0x1F4, offset >> 8);
9867   outb(0x1F5, offset >> 16);
9868   outb(0x1F6, (offset >> 24) | 0xE0);
9869   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
9870
9871   // Read data.
9872   waitdisk();
9873   insl(0x1F0, dst, SECTSIZE/4);
9874 }
9875
9876 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
9877 // Might copy more than asked.
9878 void
9879 readseg(uchar* pa, uint count, uint offset)
9880 {
9881   uchar* epa;
9882
9883   epa = pa + count;
9884
9885   // Round down to sector boundary.
9886   pa -= offset % SECTSIZE;
9887
9888   // Translate from bytes to sectors; kernel starts at sector 1.
9889   offset = (offset / SECTSIZE) + 1;
9890
9891   // If this is too slow, we could read lots of sectors at a time.
9892   // We'd write more to memory than asked, but it doesn't matter --
9893   // we load in increasing order.
9894   for(; pa < epa; pa += SECTSIZE, offset++)
9895     readsect(pa, offset);
9896 }
9897
9898
9899
```

```
#include "types.h"
#include "user.h"
#include "date.h"

int main (int argc, char *argv[])
{
    struct rtcdate r;

    if(date(&r))
    {
        printf(2, "date failed \n" ) ;
        exit();
    }

    //CODE to print time in any format
    printf(1, "%d:%d:%d    %d/%d/%d\n", r.hour, r.minute, r.second, r.month,
    exit();
}
```



```
struct rtcdate {
  uint second;
  uint minute;
  uint hour;
  uint day;
  uint month;
  uint year;
};
```

```
#include "types.h"
#include "user.h"
#include "date.h"

int main (int argc, char *argv[])
{
    struct rtcdate r1;
    struct rtcdate r2;
    int pid = 0;
    int hour = 0;
    int minute = 0;
    int second = 0;

    if(date(&r1))           //Get time start
    {
        printf(2, "date failed \n" ) ;
        exit();
    }

    pid = fork();
    if(pid > 0)             //parent exits and waits for child process to e
    {
        pid = wait();
        if(date(&r2))       //Get time finish
        {
            printf(2, "date failed \n" ) ;
            exit();
        }
    }
    else if(pid == 0)   //child exits
    {
        exec(argv[1], argv+2); //run the process with name located in argv[
        if(date(&r2))       //Get time finish
        {
            printf(2, "date failed \n" ) ;
            exit();
        }
    }
    else
    {
        printf(0, "fork error\n");
    }

    hour = r2.hour - r1.hour;
    minute = r2.minute - r1.minute;

    if(r2.second > r1.second)
        second = r2.second - r1.second;
    else
        second = r1.second - r2.second;
```



```
    //Print elapsed time
    printf(1, "Elapsed Time: %d hours %d minutes %d seconds\n", hour, minute
    exit();
}
```

```
10100 struct stat;
10101 struct rtcdate;
10102 struct uproc;
10103
10104 // system calls
10105 int fork(void);
10106 int exit(void) __attribute__((noreturn));
10107 int wait(void);
10108 int pipe(int*);
10109 int write(int, void*, int);
10110 int read(int, void*, int);
10111 int close(int);
10112 int kill(int);
10113 int exec(char*, char**);
10114 int open(char*, int);
10115 int mknod(char*, short, short);
10116 int unlink(char*);
10117 int fstat(int fd, struct stat*);
10118 int link(char*, char*);
10119 int mkdir(char*);
10120 int chdir(char*);
10121 int dup(int);
10122 int getpid(void);
10123 char* sbrk(int);
10124 int sleep(int);
10125 int uptime(void);
10126 int halt(void);
10127 //Defined date function that allows user to call through shell
10128 int date(struct rtcdate*);
10129
10130 // Project 3
10131 int getuid(void);   // UID of the current process
10132 int getgid (void);  // GID of the current process
10133 int getppid(void);  // process ID of the parent process
10134
10135 int setuid(unsigned int); // set UID to unsigned int
10136 int setgid(unsigned int); // set GID to unsigned int
10137 int getprocs(int, struct uproc*);
10138
10139 // Project 4
10140 int setPriority(unsigned int, unsigned int);
10141
10142 // ulib.c
10143 int stat(char*, struct stat*);
10144 char* strcpy(char*, char*);
10145 void *memmove(void*, void*, int);
10146 char* strchr(const char*, char c);
10147 int strcmp(const char*, const char*);
10148 void printf(int, char*, ...);
10149 char* gets(char*, int max);
```



```
10150 uint strlen(char*);
10151 void* memset(void*, int, uint);
10152 void* malloc(uint);
10153 void free(void*);
10154 int atoi(const char*);
10155
10156
10157
10158
10159
10160
10161
10162
10163
10164
10165
10166
10167
10168
10169
10170
10171
10172
10173
10174
10175
10176
10177
10178
10179
10180
10181
10182
10183
10184
10185
10186
10187
10188
10189
10190
10191
10192
10193
10194
10195
10196
10197
10198
10199
```

```
10200 // Project 3
10201 struct uproc{
10202   int pid;
10203   int uid;
10204   int gid;
10205   int ppid;
10206
10207   char STATE[16];
10208   int size;
10209   char name[16];
10210
10211   int priority;   // added for project 4
10212 };
10213
10214
10215
10216
10217
10218
10219
10220
10221
10222
10223
10224
10225
10226
10227
10228
10229
10230
10231
10232
10233
10234
10235
10236
10237
10238
10239
10240
10241
10242
10243
10244
10245
10246
10247
10248
10249
```



```
10250 #include "types.h"
10251 #include "user.h"
10252 #include "uproc.h"
10253
10254 int main (int argc, char *argv[])
10255 {
10256
10257   int i;
10258   int count = 0;
10259   int MAX = 65;
10260   struct uproc table[MAX];
10261
10262   if(setPriority(1,2) < 0)
10263   {
10264     printf(2, "setPriority failed \n");
10265     exit();
10266   }
10267
10268   count = getprocs(MAX,table);
10269
10270   if(count < 0)
10271   {
10272     printf(2, "getprocs failed \n" );
10273     exit();
10274   }
10275
10276   printf(0, "PID    UID    GID    PPID    STATE    SIZE    PRIORITY    NAME`
10277   printf(0, "---    ---    ---    ----    -----    ----    --------    ----`
10278   for(i = 0; i < count; ++i)
10279   {
10280     printf(0, "%d      %d      %d      %d       %s   %d   %d           %s\n`
10281     table[i].pid, table[i].uid, table[i].gid, table[i].ppid, table[i].STATE,
10282     table[i].size, table[i].priority, table[i].name);
10283   }
10284
10285   exit();
10286 }
10287
10288
10289
10290
10291
10292
10293
10294
10295
10296
10297
10298
10299
```

```
10300 #include "types.h"
10301 #include "user.h"
10302
10303 int
10304 testuidgid (void)
10305 {
10306   int uid , gid , ppid;
10307   uid = getuid ();
10308   printf(2, "Current UID is: %d\n", uid);
10309   printf(2, "Setting UID to 100\n");
10310   setuid (100);
10311   uid = getuid ();
10312   printf(2, "Current UID is: %d\n", uid);
10313
10314   gid = getgid ();
10315   printf(2, "Current GID is: %d\n", gid);
10316   printf(2, "Setting GID to 100\n");
10317    setgid (100);
10318   gid = getgid ();
10319   printf(2, "Current GID is: %d\n", uid);
10320
10321   ppid = getppid ();
10322   printf(2, "My parent process is: %d\n", ppid);
10323   printf(2, "Done!\n");
10324
10325   exit();
10326 }
10327
10328
10329
10330
10331
10332
10333
10334
10335
10336
10337
10338
10339
10340
10341
10342
10343
10344
10345
10346
10347
10348
10349
```



```
10350 // Test program for CS333 scheduler, project 4.
10351
10352 #include "types.h"
10353 #include "user.h"
10354
10355 // We currently have 3 priority levels
10356 #define PrioCount 3
10357 #define numChildren 10
10358
10359 void
10360 countForever(int p)
10361 {
10362   int j;
10363   unsigned long count = 0;
10364
10365   j = getpid();
10366   p = p%PrioCount;
10367   setPriority(j, p);
10368   printf(1, "%d: start prio %d\n", j, p);
10369
10370   while (1) {
10371     count++;
10372     if ((count & 0xFFFFFFF) == 0) {
10373       p = (p+1) % PrioCount;
10374       setPriority(j, p);
10375       printf(1, "%d: new prio %d\n", j, p);
10376     }
10377   }
10378 }
10379
10380 int
10381 main(void)
10382 {
10383   int i, rc;
10384
10385   for (i=0; i<numChildren; i++) {
10386     rc = fork();
10387     if (!rc) { // child
10388       countForever(i);
10389     }
10390   }
10391   // what the heck, let's have the parent waste time as well!
10392   countForever(1);
10393   exit();
10394 }
10395
10396
10397
10398
10399
```