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 int             copy(int,struct uproc*);
0373
0374 // swtch.S
0375 void            swtch(struct context**, struct context*);
0376
0377 // spinlock.c
0378 void            acquire(struct spinlock*);
0379 void            getcallerpcs(void*, uint*);
0380 int             holding(struct spinlock*);
0381 void            initlock(struct spinlock*, char*);
0382 void            release(struct spinlock*);
0383 void            pushcli(void);
0384 void            popcli(void);
0385
0386 // string.c
0387 int             memcmp(const void*, const void*, uint);
0388 void*           memmove(void*, const void*, uint);
0389 void*           memset(void*, int, uint);
0390 char*           safestrcpy(char*, const char*, int);
0391 int             strlen(const char*);
0392 int             strncmp(const char*, const char*, uint);
0393 char*           strncpy(char*, const char*, int);
0394
0395 // syscall.c
0396 int             argint(int, int*);
0397 int             argptr(int, char**, int);
0398 int             argstr(int, char**);
0399 int             fetchint(uint, int*);
```

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

```
1000 # Multiboot header, for multiboot boot loaders like GNU Grub.
1001 # http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
1002 #
1003 # Using GRUB 2, you can boot xv6 from a file stored in a
1004 # Linux file system by copying kernel or kernelmemfs to /boot
1005 # and then adding this menu entry:
1006 #
1007 # menuentry "xv6" {
1008 #  insmod ext2
1009 #  set root='(hd0,msdos1)'
1010 #  set kernel='/boot/kernel'
1011 #  echo "Loading ${kernel}..."
1012 #  multiboot ${kernel} ${kernel}
1013 #  boot
1014 # }
1015
1016 #include "asm.h"
1017 #include "memlayout.h"
1018 #include "mmu.h"
1019 #include "param.h"
1020
1021 # Multiboot header.  Data to direct multiboot loader.
1022 .p2align 2
1023 .text
1024 .globl multiboot_header
1025 multiboot_header:
1026   #define magic 0x1badb002
1027   #define flags 0
1028   .long magic
1029   .long flags
1030   .long (-magic-flags)
1031
1032 # By convention, the _start symbol specifies the ELF entry point.
1033 # Since we haven't set up virtual memory yet, our entry point is
1034 # the physical address of 'entry'.
1035 .globl _start
1036 _start = V2P_WO(entry)
1037
1038 # Entering xv6 on boot processor, with paging off.
1039 .globl entry
1040 entry:
1041   # Turn on page size extension for 4Mbyte pages
1042   movl    %cr4, %eax
1043   orl     $(CR4_PSE), %eax
1044   movl    %eax, %cr4
1045   # Set page directory
1046   movl    $(V2P_WO(entrypgdir)), %eax
1047   movl    %eax, %cr3
1048   # Turn on paging.
1049   movl    %cr0, %eax
```



```
1050   orl     $(CR0_PG|CR0_WP), %eax
1051   movl    %eax, %cr0
1052
1053   # Set up the stack pointer.
1054   movl $(stack + KSTACKSIZE), %esp
1055
1056   # Jump to main(), and switch to executing at
1057   # high addresses. The indirect call is needed because
1058   # the assembler produces a PC-relative instruction
1059   # for a direct jump.
1060   mov $main, %eax
1061   jmp *%eax
1062
1063 .comm stack, KSTACKSIZE
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
```

```
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
```

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

extern char data[];  // defined by kernel.ld
pde_t *kpgdir;  // for use in scheduler()
struct segdesc gdt[NSEGS];

// Set up CPU's kernel segment descriptors.
// Run once on entry on each CPU.
void
seginit(void)
{
  struct cpu *c;

  // Map "logical" addresses to virtual addresses using identity map.
  // Cannot share a CODE descriptor for both kernel and user
  // because it would have to have DPL_USR, but the CPU forbids
  // an interrupt from CPL=0 to DPL=3.
  c = &cpus[cpunum()];
  c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
  c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);

  // Map cpu, and curproc
  c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);

  lgdt(c->gdt, sizeof(c->gdt));
  loadgs(SEG_KCPU << 3);

  // Initialize cpu-local storage.
  cpu = c;
  proc = 0;
}
```



```
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}

// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

```
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 #define DEFAULT_uid 0
2303 #define DEFAULT_gid 0
2304
2305 // Per-CPU state
2306 struct cpu {
2307   uchar id;                    // Local APIC ID; index into cpus[] below
2308   struct context *scheduler;   // swtch() here to enter scheduler
2309   struct taskstate ts;         // Used by x86 to find stack for interrupt
2310   struct segdesc gdt[NSEGS];   // x86 global descriptor table
2311   volatile uint started;       // Has the CPU started?
2312   int ncli;                    // Depth of pushcli nesting.
2313   int intena;                  // Were interrupts enabled before pushcli?
2314
2315   // Cpu-local storage variables; see below
2316   struct cpu *cpu;
2317   struct proc *proc;           // The currently-running process.
2318 };
2319
2320 extern struct cpu cpus[NCPU];
2321 extern int ncpu;
2322
2323 // Per-CPU variables, holding pointers to the
2324 // current cpu and to the current process.
2325 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2326 // and "%gs:4" to refer to proc.  seginit sets up the
2327 // %gs segment register so that %gs refers to the memory
2328 // holding those two variables in the local cpu's struct cpu.
2329 // This is similar to how thread-local variables are implemented
2330 // in thread libraries such as Linux pthreads.
2331 extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
2332 extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc
2333
2334
2335
2336
2337
2338
2339
2340
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
2391 // Process memory is laid out contiguously, low addresses first:
2392 //   text
2393 //   original data and bss
2394 //   fixed-size stack
2395 //   expandable heap
2396
2397
2398
2399
```

```
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 struct {
2411   struct spinlock lock;
2412   struct proc proc[NPROC];
2413 } ptable;
2414
2415 static struct proc *initproc;
2416
2417 int nextpid = 1;
2418 extern void forkret(void);
2419 extern void trapret(void);
2420
2421 static void wakeup1(void *chan);
2422
2423 void
2424 pinit(void)
2425 {
2426   initlock(&ptable.lock, "ptable");
2427 }
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
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     if(p->state == UNUSED)
2463       goto found;
2464   release(&ptable.lock);
2465   return 0;
2466
2467 found:
2468   p->state = EMBRYO;
2469   p->pid = nextpid++;
2470   release(&ptable.lock);
2471
2472   // Allocate kernel stack.
2473   if((p->kstack = kalloc()) == 0){
2474     p->state = UNUSED;
2475     return 0;
2476   }
2477   sp = p->kstack + KSTACKSIZE;
2478
2479   // Leave room for trap frame.
2480   sp -= sizeof *p->tf;
2481   p->tf = (struct trapframe*)sp;
2482
2483   // Set up new context to start executing at forkret,
2484   // which returns to trapret.
2485   sp -= 4;
2486   *(uint*)sp = (uint)trapret;
2487
2488   sp -= sizeof *p->context;
2489   p->context = (struct context*)sp;
2490   memset(p->context, 0, sizeof *p->context);
2491   p->context->eip = (uint)forkret;
2492
2493   return p;
2494 }
2495
2496
2497
2498
2499
```

```
2500 // Set up first user process.
2501 void
2502 userinit(void)
2503 {
2504   struct proc *p;
2505   extern char _binary_initcode_start[], _binary_initcode_size[];
2506
2507   p = allocproc();
2508   initproc = p;
2509   if((p->pgdir = setupkvm()) == 0)
2510     panic("userinit: out of memory?");
2511   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2512   p->sz = PGSIZE;
2513   memset(p->tf, 0, sizeof(*p->tf));
2514   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2515   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2516   p->tf->es = p->tf->ds;
2517   p->tf->ss = p->tf->ds;
2518   p->tf->eflags = FL_IF;
2519   p->tf->esp = PGSIZE;
2520   p->tf->eip = 0;  // beginning of initcode.S
2521
2522   safestrcpy(p->name, "initcode", sizeof(p->name));
2523   p->cwd = namei("/");
2524
2525   p->state = RUNNABLE;
2526   p->uid = DEFAULT_uid; //set default uid for first process
2527   p->gid = DEFAULT_gid; //set default gid for first process
2528 }
2529
2530 // Grow current process's memory by n bytes.
2531 // Return 0 on success, -1 on failure.
2532 int
2533 growproc(int n)
2534 {
2535   uint sz;
2536
2537   sz = proc->sz;
2538   if(n > 0){
2539     if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2540       return -1;
2541   } else if(n < 0){
2542     if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2543       return -1;
2544   }
2545   proc->sz = sz;
2546   switchuvm(proc);
2547   return 0;
2548 }
2549
```



```
2550 // Create a new process copying p as the parent.
2551 // Sets up stack to return as if from system call.
2552 // Caller must set state of returned proc to RUNNABLE.
2553 int
2554 fork(void)
2555 {
2556   int i, pid;
2557   struct proc *np;
2558
2559   // Allocate process.
2560   if((np = allocproc()) == 0)
2561     return -1;
2562
2563   // Copy process state from p.
2564   if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2565     kfree(np->kstack);
2566     np->kstack = 0;
2567     np->state = UNUSED;
2568     return -1;
2569   }
2570   np->sz = proc->sz;
2571   np->parent = proc;
2572   *np->tf = *proc->tf;
2573
2574   // Clear %eax so that fork returns 0 in the child.
2575   np->tf->eax = 0;
2576
2577   for(i = 0; i < NOFILE; i++)
2578     if(proc->ofile[i])
2579       np->ofile[i] = filedup(proc->ofile[i]);
2580   np->cwd = idup(proc->cwd);
2581
2582   safestrcpy(np->name, proc->name, sizeof(proc->name));
2583
2584   pid = np->pid;
2585   np->uid = proc->uid; //Copy uid from parent to child
2586   np->gid = proc->gid; //Copy gid from parent to child
2587
2588   // lock to force the compiler to emit the np->state write last.
2589   acquire(&ptable.lock);
2590   np->state = RUNNABLE;
2591   release(&ptable.lock);
2592
2593   return pid;
2594 }
2595
2596
2597
2598
2599
```

```
2600 // Exit the current process.  Does not return.
2601 // An exited process remains in the zombie state
2602 // until its parent calls wait() to find out it exited.
2603 void
2604 exit(void)
2605 {
2606   struct proc *p;
2607   int fd;
2608
2609   if(proc == initproc)
2610     panic("init exiting");
2611
2612   // Close all open files.
2613   for(fd = 0; fd < NOFILE; fd++){
2614     if(proc->ofile[fd]){
2615       fileclose(proc->ofile[fd]);
2616       proc->ofile[fd] = 0;
2617     }
2618   }
2619
2620   begin_op();
2621   iput(proc->cwd);
2622   end_op();
2623   proc->cwd = 0;
2624
2625   acquire(&ptable.lock);
2626
2627   // Parent might be sleeping in wait().
2628   wakeup1(proc->parent);
2629
2630   // Pass abandoned children to init.
2631   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2632     if(p->parent == proc){
2633       p->parent = initproc;
2634       if(p->state == ZOMBIE)
2635         wakeup1(initproc);
2636     }
2637   }
2638
2639   // Jump into the scheduler, never to return.
2640   proc->state = ZOMBIE;
2641   sched();
2642   panic("zombie exit");
2643 }
2644
2645
2646
2647
2648
2649
```



```
2650 // Wait for a child process to exit and return its pid.
2651 // Return -1 if this process has no children.
2652 int
2653 wait(void)
2654 {
2655   struct proc *p;
2656   int havekids, pid;
2657
2658   acquire(&ptable.lock);
2659   for(;;){
2660     // Scan through table looking for zombie children.
2661     havekids = 0;
2662     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2663       if(p->parent != proc)
2664         continue;
2665       havekids = 1;
2666       if(p->state == ZOMBIE){
2667         // Found one.
2668         pid = p->pid;
2669         kfree(p->kstack);
2670         p->kstack = 0;
2671         freevm(p->pgdir);
2672         p->state = UNUSED;
2673         p->pid = 0;
2674         p->parent = 0;
2675         p->name[0] = 0;
2676         p->killed = 0;
2677         release(&ptable.lock);
2678         return pid;
2679       }
2680     }
2681
2682     // No point waiting if we don't have any children.
2683     if(!havekids || proc->killed){
2684       release(&ptable.lock);
2685       return -1;
2686     }
2687
2688     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2689     sleep(proc, &ptable.lock);
2690   }
2691 }
2692
2693
2694
2695
2696
2697
2698
2699
```

```
2700 // Per-CPU process scheduler.
2701 // Each CPU calls scheduler() after setting itself up.
2702 // Scheduler never returns.  It loops, doing:
2703 //  - choose a process to run
2704 //  - swtch to start running that process
2705 //  - eventually that process transfers control
2706 //      via swtch back to the scheduler.
2707 void
2708 scheduler(void)
2709 {
2710   struct proc *p;
2711
2712   for(;;){
2713     // Enable interrupts on this processor.
2714     sti();
2715
2716     // Loop over process table looking for process to run.
2717     acquire(&ptable.lock);
2718     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2719       if(p->state != RUNNABLE)
2720         continue;
2721
2722       // Switch to chosen process.  It is the process's job
2723       // to release ptable.lock and then reacquire it
2724       // before jumping back to us.
2725       proc = p;
2726       switchuvm(p);
2727       p->state = RUNNING;
2728       swtch(&cpu->scheduler, proc->context);
2729       switchkvm();
2730
2731       // Process is done running for now.
2732       // It should have changed its p->state before coming back.
2733       proc = 0;
2734     }
2735     release(&ptable.lock);
2736
2737   }
2738 }
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
```



```
2750 // Enter scheduler.  Must hold only ptable.lock
2751 // and have changed proc->state.
2752 void
2753 sched(void)
2754 {
2755   int intena;
2756
2757   if(!holding(&ptable.lock))
2758     panic("sched ptable.lock");
2759   if(cpu->ncli != 1)
2760     panic("sched locks");
2761   if(proc->state == RUNNING)
2762     panic("sched running");
2763   if(readeflags()&FL_IF)
2764     panic("sched interruptible");
2765   intena = cpu->intena;
2766   swtch(&proc->context, cpu->scheduler);
2767   cpu->intena = intena;
2768 }
2769
2770 // Give up the CPU for one scheduling round.
2771 void
2772 yield(void)
2773 {
2774   acquire(&ptable.lock);
2775   proc->state = RUNNABLE;
2776   sched();
2777   release(&ptable.lock);
2778 }
2779
2780 // A fork child's very first scheduling by scheduler()
2781 // will swtch here.  "Return" to user space.
2782 void
2783 forkret(void)
2784 {
2785   static int first = 1;
2786   // Still holding ptable.lock from scheduler.
2787   release(&ptable.lock);
2788
2789   if (first) {
2790     // Some initialization functions must be run in the context
2791     // of a regular process (e.g., they call sleep), and thus cannot
2792     // be run from main().
2793     first = 0;
2794     iinit(ROOTDEV);
2795     initlog(ROOTDEV);
2796   }
2797
2798   // Return to "caller", actually trapret (see allocproc).
2799 }
```

```
2800 // Atomically release lock and sleep on chan.
2801 // Reacquires lock when awakened.
2802 void
2803 sleep(void *chan, struct spinlock *lk)
2804 {
2805   if(proc == 0)
2806     panic("sleep");
2807
2808   if(lk == 0)
2809     panic("sleep without lk");
2810
2811   // Must acquire ptable.lock in order to
2812   // change p->state and then call sched.
2813   // Once we hold ptable.lock, we can be
2814   // guaranteed that we won't miss any wakeup
2815   // (wakeup runs with ptable.lock locked),
2816   // so it's okay to release lk.
2817   if(lk != &ptable.lock){
2818     acquire(&ptable.lock);
2819     release(lk);
2820   }
2821
2822   // Go to sleep.
2823   proc->chan = chan;
2824   proc->state = SLEEPING;
2825   sched();
2826
2827   // Tidy up.
2828   proc->chan = 0;
2829
2830   // Reacquire original lock.
2831   if(lk != &ptable.lock){
2832     release(&ptable.lock);
2833     acquire(lk);
2834   }
2835 }
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
```



```
2850 // Wake up all processes sleeping on chan.
2851 // The ptable lock must be held.
2852 static void
2853 wakeup1(void *chan)
2854 {
2855   struct proc *p;
2856
2857   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2858     if(p->state == SLEEPING && p->chan == chan)
2859       p->state = RUNNABLE;
2860 }
2861
2862 // Wake up all processes sleeping on chan.
2863 void
2864 wakeup(void *chan)
2865 {
2866   acquire(&ptable.lock);
2867   wakeup1(chan);
2868   release(&ptable.lock);
2869 }
2870
2871 // Kill the process with the given pid.
2872 // Process won't exit until it returns
2873 // to user space (see trap in trap.c).
2874 int
2875 kill(int pid)
2876 {
2877   struct proc *p;
2878
2879   acquire(&ptable.lock);
2880   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2881     if(p->pid == pid){
2882       p->killed = 1;
2883       // Wake process from sleep if necessary.
2884       if(p->state == SLEEPING)
2885         p->state = RUNNABLE;
2886       release(&ptable.lock);
2887       return 0;
2888     }
2889   }
2890   release(&ptable.lock);
2891   return -1;
2892 }
2893
2894
2895
2896
2897
2898
2899
```

```
// Print a process listing to console.  For debugging.
// Runs when user types ^P on console.
// No lock to avoid wedging a stuck machine further.
void
procdump(void)
{
  static char *states[] = {
  [UNUSED]    "unused",
  [EMBRYO]    "embryo",
  [SLEEPING]  "sleep ",
  [RUNNABLE]  "runble",
  [RUNNING]   "run   ",
  [ZOMBIE]    "zombie"
  };
  int i;
  struct proc *p;
  char *state;
  uint pc[10];

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;
    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
      state = states[p->state];
    else
      state = "???";
    cprintf("\nPID: %d  UID: %d  GID: %d  STATE: %s  NAME: %s    ", p->pid,
    if(p->state == SLEEPING){
      getcallerpcs((uint*)p->context->ebp+2, pc);
      for(i=0; i<10 && pc[i] != 0; i++)
        cprintf(" %p", pc[i]);
    }
    cprintf("\n");
  }
}
```



```
int
copy(int MAX, struct uproc *table)
{
  acquire(&ptable.lock);
  int i;

  static char *states[] = {
  [UNUSED]    "unused",
  [EMBRYO]    "embryo",
  [SLEEPING]  "sleep ",
  [RUNNABLE]  "runble",
  [RUNNING]   "run   ",
  [ZOMBIE]    "zombie"
  };

  for(i = 0; ptable.proc[i].state != UNUSED && ptable.proc[i].state != EMBRY(
  {
    table[i].pid  = ptable.proc[i].pid;
    table[i].uid  = ptable.proc[i].uid;
    table[i].gid  = ptable.proc[i].gid;
    if(i)
      table[i].ppid = ptable.proc[i].parent->pid;
    else
      table[i].ppid = 0;
    table[i].size = ptable.proc[i].sz;
    safestrcpy(table[i].STATE, states[ptable.proc[i].state], strlen(states[p
    safestrcpy(table[i].name, ptable.proc[i].name, strlen(ptable.proc[i].name
  }
  release(&ptable.lock);
  return i;
}
```

```
3000 # Context switch
3001 #
3002 #   void swtch(struct context **old, struct context *new);
3003 #
3004 # Save current register context in old
3005 # and then load register context from new.
3006
3007 .globl swtch
3008 swtch:
3009   movl 4(%esp), %eax
3010   movl 8(%esp), %edx
3011
3012   # Save old callee-save registers
3013   pushl %ebp
3014   pushl %ebx
3015   pushl %esi
3016   pushl %edi
3017
3018   # Switch stacks
3019   movl %esp, (%eax)
3020   movl %edx, %esp
3021
3022   # Load new callee-save registers
3023   popl %edi
3024   popl %esi
3025   popl %ebx
3026   popl %ebp
3027   ret
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
```



```
3050 // Physical memory allocator, intended to allocate
3051 // memory for user processes, kernel stacks, page table pages,
3052 // and pipe buffers. Allocates 4096-byte pages.
3053
3054 #include "types.h"
3055 #include "defs.h"
3056 #include "param.h"
3057 #include "memlayout.h"
3058 #include "mmu.h"
3059 #include "spinlock.h"
3060
3061 void freerange(void *vstart, void *vend);
3062 extern char end[]; // first address after kernel loaded from ELF file
3063
3064 struct run {
3065   struct run *next;
3066 };
3067
3068 struct {
3069   struct spinlock lock;
3070   int use_lock;
3071   struct run *freelist;
3072 } kmem;
3073
3074 // Initialization happens in two phases.
3075 // 1. main() calls kinit1() while still using entrypgdir to place just
3076 // the pages mapped by entrypgdir on free list.
3077 // 2. main() calls kinit2() with the rest of the physical pages
3078 // after installing a full page table that maps them on all cores.
3079 void
3080 kinit1(void *vstart, void *vend)
3081 {
3082   initlock(&kmem.lock, "kmem");
3083   kmem.use_lock = 0;
3084   freerange(vstart, vend);
3085 }
3086
3087 void
3088 kinit2(void *vstart, void *vend)
3089 {
3090   freerange(vstart, vend);
3091   kmem.use_lock = 1;
3092 }
3093
3094
3095
3096
3097
3098
3099
```

```
3100 void
3101 freerange(void *vstart, void *vend)
3102 {
3103   char *p;
3104   p = (char*)PGROUNDUP((uint)vstart);
3105   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3106     kfree(p);
3107 }
3108
3109
3110 // Free the page of physical memory pointed at by v,
3111 // which normally should have been returned by a
3112 // call to kalloc().  (The exception is when
3113 // initializing the allocator; see kinit above.)
3114 void
3115 kfree(char *v)
3116 {
3117   struct run *r;
3118
3119   if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
3120     panic("kfree");
3121
3122   // Fill with junk to catch dangling refs.
3123   memset(v, 1, PGSIZE);
3124
3125   if(kmem.use_lock)
3126     acquire(&kmem.lock);
3127   r = (struct run*)v;
3128   r->next = kmem.freelist;
3129   kmem.freelist = r;
3130   if(kmem.use_lock)
3131     release(&kmem.lock);
3132 }
3133
3134 // Allocate one 4096-byte page of physical memory.
3135 // Returns a pointer that the kernel can use.
3136 // Returns 0 if the memory cannot be allocated.
3137 char*
3138 kalloc(void)
3139 {
3140   struct run *r;
3141
3142   if(kmem.use_lock)
3143     acquire(&kmem.lock);
3144   r = kmem.freelist;
3145   if(r)
3146     kmem.freelist = r->next;
3147   if(kmem.use_lock)
3148     release(&kmem.lock);
3149   return (char*)r;
```



```
3150 }
3151
3152
3153
3154
3155
3156
3157
3158
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3180
3181
3182
3183
3184
3185
3186
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
```

```
3200 // x86 trap and interrupt constants.
3201
3202 // Processor-defined:
3203 #define T_DIVIDE         0      // divide error
3204 #define T_DEBUG          1      // debug exception
3205 #define T_NMI            2      // non-maskable interrupt
3206 #define T_BRKPT          3      // breakpoint
3207 #define T_OFLOW          4      // overflow
3208 #define T_BOUND          5      // bounds check
3209 #define T_ILLOP          6      // illegal opcode
3210 #define T_DEVICE         7      // device not available
3211 #define T_DBLFLT         8      // double fault
3212 // #define T_COPROC      9      // reserved (not used since 486)
3213 #define T_TSS           10      // invalid task switch segment
3214 #define T_SEGNP         11      // segment not present
3215 #define T_STACK         12      // stack exception
3216 #define T_GPFLT         13      // general protection fault
3217 #define T_PGFLT         14      // page fault
3218 // #define T_RES        15      // reserved
3219 #define T_FPERR         16      // floating point error
3220 #define T_ALIGN         17      // aligment check
3221 #define T_MCHK          18      // machine check
3222 #define T_SIMDERR       19      // SIMD floating point error
3223
3224 // These are arbitrarily chosen, but with care not to overlap
3225 // processor defined exceptions or interrupt vectors.
3226 #define T_SYSCALL       64      // system call
3227 #define T_DEFAULT      500      // catchall
3228
3229 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
3230
3231 #define IRQ_TIMER        0
3232 #define IRQ_KBD          1
3233 #define IRQ_COM1         4
3234 #define IRQ_IDE         14
3235 #define IRQ_ERROR       19
3236 #define IRQ_SPURIOUS    31
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
```



```
3250 #!/usr/bin/perl -w
3251
3252 # Generate vectors.S, the trap/interrupt entry points.
3253 # There has to be one entry point per interrupt number
3254 # since otherwise there's no way for trap() to discover
3255 # the interrupt number.
3256
3257 print "# generated by vectors.pl - do not edit\n";
3258 print "# handlers\n";
3259 print ".globl alltraps\n";
3260 for(my $i = 0; $i < 256; $i++){
3261     print ".globl vector$i\n";
3262     print "vector$i:\n";
3263     if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
3264         print "  pushl \$0\n";
3265     }
3266     print "  pushl \$$i\n";
3267     print "  jmp alltraps\n";
3268 }
3269
3270 print "\n# vector table\n";
3271 print ".data\n";
3272 print ".globl vectors\n";
3273 print "vectors:\n";
3274 for(my $i = 0; $i < 256; $i++){
3275     print "  .long vector$i\n";
3276 }
3277
3278 # sample output:
3279 #   # handlers
3280 #   .globl alltraps
3281 #   .globl vector0
3282 #   vector0:
3283 #     pushl $0
3284 #     pushl $0
3285 #     jmp alltraps
3286 #   ...
3287 #
3288 #   # vector table
3289 #   .data
3290 #   .globl vectors
3291 #   vectors:
3292 #     .long vector0
3293 #     .long vector1
3294 #     .long vector2
3295 #   ...
3296
3297
3298
3299
```

```
#include "mmu.h"

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

  # Set up data and per-cpu segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es
  movw $(SEG_KCPU<<3), %ax
  movw %ax, %fs
  movw %ax, %gs

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

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



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

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

void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}

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

```
3400 void
3401 trap(struct trapframe *tf)
3402 {
3403   if(tf->trapno == T_SYSCALL){
3404     if(proc->killed)
3405       exit();
3406     proc->tf = tf;
3407     syscall();
3408     if(proc->killed)
3409       exit();
3410     return;
3411   }
3412
3413   switch(tf->trapno){
3414   case T_IRQ0 + IRQ_TIMER:
3415     if(cpu->id == 0){
3416       acquire(&tickslock);
3417       ticks++;
3418       wakeup(&ticks);
3419       release(&tickslock);
3420     }
3421     lapiceoi();
3422     break;
3423   case T_IRQ0 + IRQ_IDE:
3424     ideintr();
3425     lapiceoi();
3426     break;
3427   case T_IRQ0 + IRQ_IDE+1:
3428     // Bochs generates spurious IDE1 interrupts.
3429     break;
3430   case T_IRQ0 + IRQ_KBD:
3431     kbdintr();
3432     lapiceoi();
3433     break;
3434   case T_IRQ0 + IRQ_COM1:
3435     uartintr();
3436     lapiceoi();
3437     break;
3438   case T_IRQ0 + 7:
3439   case T_IRQ0 + IRQ_SPURIOUS:
3440     cprintf("cpu%d: spurious interrupt at %x:%x\n",
3441             cpu->id, tf->cs, tf->eip);
3442     lapiceoi();
3443     break;
3444
3445
3446
3447
3448
3449
```



```
3450   default:
3451     if(proc == 0 || (tf->cs&3) == 0){
3452       // In kernel, it must be our mistake.
3453       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3454               tf->trapno, cpu->id, tf->eip, rcr2());
3455       panic("trap");
3456     }
3457     // In user space, assume process misbehaved.
3458     cprintf("pid %d %s: trap %d err %d on cpu %d "
3459             "eip 0x%x addr 0x%x--kill proc\n",
3460             proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
3461             rcr2());
3462     proc->killed = 1;
3463   }
3464
3465   // Force process exit if it has been killed and is in user space.
3466   // (If it is still executing in the kernel, let it keep running
3467   // until it gets to the regular system call return.)
3468   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3469     exit();
3470
3471   // Force process to give up CPU on clock tick.
3472   // If interrupts were on while locks held, would need to check nlock.
3473   if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3474     yield();
3475
3476   // Check if the process has been killed since we yielded
3477   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3478     exit();
3479 }
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
```

```
3500 // System call numbers
3501 #define SYS_fork     1
3502 #define SYS_exit     2
3503 #define SYS_wait     3
3504 #define SYS_pipe     4
3505 #define SYS_read     5
3506 #define SYS_kill     6
3507 #define SYS_exec     7
3508 #define SYS_fstat    8
3509 #define SYS_chdir           9
3510 #define SYS_dup             10
3511 #define SYS_getpid          11
3512 #define SYS_sbrk            12
3513 #define SYS_sleep           13
3514 #define SYS_uptime          14
3515 #define SYS_open            15
3516 #define SYS_write           16
3517 #define SYS_mknod           17
3518 #define SYS_unlink          18
3519 #define SYS_link            19
3520 #define SYS_mkdir           20
3521 #define SYS_close           21
3522 #define SYS_halt            22
3523 //Added to define position of the system call vector that connects
3524 //to implementation
3525 #define SYS_date            23
3526
3527 //PROJECT 3
3528 #define SYS_getuid  24
3529 #define SYS_getgid  25
3530 #define SYS_getppid         26
3531 #define SYS_setuid  27
3532 #define SYS_setgid  28
3533 #define SYS_getprocs 29
3534
3535
3536
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
```



```
3550 #include "types.h"
3551 #include "defs.h"
3552 #include "param.h"
3553 #include "memlayout.h"
3554 #include "mmu.h"
3555 #include "proc.h"
3556 #include "x86.h"
3557 #include "syscall.h"
3558
3559 // User code makes a system call with INT T_SYSCALL.
3560 // System call number in %eax.
3561 // Arguments on the stack, from the user call to the C
3562 // library system call function. The saved user %esp points
3563 // to a saved program counter, and then the first argument.
3564
3565 // Fetch the int at addr from the current process.
3566 int
3567 fetchint(uint addr, int *ip)
3568 {
3569   if(addr >= proc->sz || addr+4 > proc->sz)
3570     return -1;
3571   *ip = *(int*)(addr);
3572   return 0;
3573 }
3574
3575 // Fetch the nul-terminated string at addr from the current process.
3576 // Doesn't actually copy the string - just sets *pp to point at it.
3577 // Returns length of string, not including nul.
3578 int
3579 fetchstr(uint addr, char **pp)
3580 {
3581   char *s, *ep;
3582
3583   if(addr >= proc->sz)
3584     return -1;
3585   *pp = (char*)addr;
3586   ep = (char*)proc->sz;
3587   for(s = *pp; s < ep; s++)
3588     if(*s == 0)
3589       return s - *pp;
3590   return -1;
3591 }
3592
3593 // Fetch the nth 32-bit system call argument.
3594 int
3595 argint(int n, int *ip)
3596 {
3597   return fetchint(proc->tf->esp + 4 + 4*n, ip);
3598 }
3599
```

```
3600 // Fetch the nth word-sized system call argument as a pointer
3601 // to a block of memory of size n bytes.  Check that the pointer
3602 // lies within the process address space.
3603 int
3604 argptr(int n, char **pp, int size)
3605 {
3606   int i;
3607
3608   if(argint(n, &i) < 0)
3609     return -1;
3610   if((uint)i >= proc->sz || (uint)i+size > proc->sz)
3611     return -1;
3612   *pp = (char*)i;
3613   return 0;
3614 }
3615
3616 // Fetch the nth word-sized system call argument as a string pointer.
3617 // Check that the pointer is valid and the string is nul-terminated.
3618 // (There is no shared writable memory, so the string can't change
3619 // between this check and being used by the kernel.)
3620 int
3621 argstr(int n, char **pp)
3622 {
3623   int addr;
3624   if(argint(n, &addr) < 0)
3625     return -1;
3626   return fetchstr(addr, pp);
3627 }
3628
3629 extern int sys_chdir(void);
3630 extern int sys_close(void);
3631 extern int sys_dup(void);
3632 extern int sys_exec(void);
3633 extern int sys_exit(void);
3634 extern int sys_fork(void);
3635 extern int sys_fstat(void);
3636 extern int sys_getpid(void);
3637 extern int sys_kill(void);
3638 extern int sys_link(void);
3639 extern int sys_mkdir(void);
3640 extern int sys_mknod(void);
3641 extern int sys_open(void);
3642 extern int sys_pipe(void);
3643 extern int sys_read(void);
3644 extern int sys_sbrk(void);
3645 extern int sys_sleep(void);
3646 extern int sys_unlink(void);
3647 extern int sys_wait(void);
3648 extern int sys_write(void);
3649 extern int sys_uptime(void);
```



```
3650 extern int sys_halt(void);
3651 //Added sys_date to allow routine to be available to other parts of kernel
3652 extern int sys_date(void);
3653
3654 //Project 3
3655 extern int sys_getuid(void);
3656 extern int sys_getgid(void);
3657 extern int sys_getppid(void);
3658 extern int sys_setuid(void);
3659 extern int sys_setgid(void);
3660 extern int sys_getprocs(void);
3661
3662 static int (*syscalls[])(void) = {
3663 [SYS_fork]    sys_fork,
3664 [SYS_exit]    sys_exit,
3665 [SYS_wait]    sys_wait,
3666 [SYS_pipe]    sys_pipe,
3667 [SYS_read]    sys_read,
3668 [SYS_kill]    sys_kill,
3669 [SYS_exec]    sys_exec,
3670 [SYS_fstat]   sys_fstat,
3671 [SYS_chdir]   sys_chdir,
3672 [SYS_dup]     sys_dup,
3673 [SYS_getpid]  sys_getpid,
3674 [SYS_sbrk]    sys_sbrk,
3675 [SYS_sleep]   sys_sleep,
3676 [SYS_uptime]  sys_uptime,
3677 [SYS_open]    sys_open,
3678 [SYS_write]   sys_write,
3679 [SYS_mknod]   sys_mknod,
3680 [SYS_unlink]  sys_unlink,
3681 [SYS_link]    sys_link,
3682 [SYS_mkdir]   sys_mkdir,
3683 [SYS_close]   sys_close,
3684 [SYS_halt]    sys_halt,
3685
3686 [SYS_date]    sys_date,
3687
3688 [SYS_getuid]       sys_getuid,
3689 [SYS_getgid]       sys_getgid,
3690 [SYS_getppid]      sys_getppid,
3691 [SYS_setuid]       sys_setuid,
3692 [SYS_setgid]       sys_setgid,
3693 [SYS_getprocs] sys_getprocs,
3694 };
3695
3696
3697
3698
3699
```

```
3700 void
3701 syscall(void)
3702 {
3703 /*
3704     char * syscallnames[] = {
3705     [SYS_fork]    "fork",
3706     [SYS_exit]    "sys_exit",
3707     [SYS_wait]    "sys_wait",
3708     [SYS_pipe]    "sys_pipe",
3709     [SYS_read]    "sys_read",
3710     [SYS_kill]    "sys_kill",
3711     [SYS_exec]    "sys_exec",
3712     [SYS_fstat]   "sys_fstat",
3713     [SYS_chdir]   "sys_chdir",
3714     [SYS_dup]     "sys_dup",
3715     [SYS_getpid]  "sys_getpid",
3716     [SYS_sbrk]    "sys_sbrk",
3717     [SYS_sleep]   "sys_sleep",
3718     [SYS_uptime]  "sys_uptime",
3719     [SYS_open]    "sys_open",
3720     [SYS_write]   "sys_write",
3721     [SYS_mknod]   "sys_mknod",
3722     [SYS_unlink]  "sys_unlink",
3723     [SYS_link]    "sys_link",
3724     [SYS_mkdir]   "sys_mkdir",
3725     [SYS_close]   "sys_close",
3726     [SYS_halt]    "sys_halt",
3727     };
3728 */
3729   int num;
3730   num = proc->tf->eax;
3731
3732   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3733     proc->tf->eax = syscalls[num]();
3734       //cprintf("%s -> %d\n", syscallnames[num], proc->tf->eax);
3735   } else {
3736     cprintf("%d %s: unknown sys call %d\n",
3737             proc->pid, proc->name, num);
3738     proc->tf->eax = -1;
3739   }
3740 }
3741
3742
3743
3744
3745
3746
3747
3748
3749
```



```
3750 #include "types.h"
3751 #include "x86.h"
3752 #include "defs.h"
3753 #include "date.h"
3754 #include "param.h"
3755 #include "memlayout.h"
3756 #include "mmu.h"
3757 #include "proc.h"
3758 #include "uproc.h"
3759
3760 int
3761 sys_fork(void)
3762 {
3763   return fork();
3764 }
3765
3766 int
3767 sys_exit(void)
3768 {
3769   exit();
3770   return 0;  // not reached
3771 }
3772
3773 int
3774 sys_wait(void)
3775 {
3776   return wait();
3777 }
3778
3779 int
3780 sys_kill(void)
3781 {
3782   int pid;
3783
3784   if(argint(0, &pid) < 0)
3785     return -1;
3786   return kill(pid);
3787 }
3788
3789 int
3790 sys_getpid(void)
3791 {
3792   return proc->pid;
3793 }
3794
3795
3796
3797
3798
3799
```

```
3800 int
3801 sys_sbrk(void)
3802 {
3803   int addr;
3804   int n;
3805
3806   if(argint(0, &n) < 0)
3807     return -1;
3808   addr = proc->sz;
3809   if(growproc(n) < 0)
3810     return -1;
3811   return addr;
3812 }
3813
3814 int
3815 sys_sleep(void)
3816 {
3817   int n;
3818   uint ticks0;
3819
3820   if(argint(0, &n) < 0)
3821     return -1;
3822   acquire(&tickslock);
3823   ticks0 = ticks;
3824   while(ticks - ticks0 < n){
3825     if(proc->killed){
3826       release(&tickslock);
3827       return -1;
3828     }
3829     sleep(&ticks, &tickslock);
3830   }
3831   release(&tickslock);
3832   return 0;
3833 }
3834
3835 // return how many clock tick interrupts have occurred
3836 // since start.
3837 int
3838 sys_uptime(void)
3839 {
3840   uint xticks;
3841
3842   acquire(&tickslock);
3843   xticks = ticks;
3844   release(&tickslock);
3845   return xticks;
3846 }
3847
3848
3849
```



```
3850 //Turn of the computer
3851 int sys_halt(void){
3852   cprintf("Shutting down ...\n");
3853   outw (0xB004, 0x0 | 0x2000);
3854   return 0;
3855 }
3856
3857 //Implemented date and time
3858 int
3859 sys_date(void)
3860 {
3861     struct rtcdate *d;
3862
3863     if(argptr(0, (void*)&d, sizeof(*d)) < 0)
3864         return -1;
3865
3866     cmostime(d);
3867     return 0;
3868 }
3869
3870 int
3871 sys_getuid(void)
3872 {
3873   return proc->uid;
3874 }
3875
3876 int
3877 sys_getgid(void)
3878 {
3879   return proc->gid;
3880 }
3881
3882 int
3883 sys_getppid(void)
3884 {
3885   if(proc->parent)
3886     return proc->parent->pid;
3887   return proc->pid;
3888 }
3889
3890 int
3891 sys_setuid(void)
3892 {
3893   int uid;
3894
3895   if(argint(0, &uid) < 0)
3896     return -1;
3897   proc->uid = uid;
3898   return 0;
3899 }
```

```
3900 int
3901 sys_setgid(void)
3902 {
3903   int gid;
3904
3905   if(argint(0, &gid) < 0)
3906     return -1;
3907   proc->gid = gid;
3908   return 0;
3909 }
3910
3911 int
3912 sys_getprocs(void)
3913 {
3914   struct uproc * table;
3915   int MAX;
3916
3917   if(argptr(1, (void*)&table, sizeof(*table)) < 0)
3918     return -1;
3919
3920   if(argint(0, &MAX) < 0)
3921     return -1;
3922
3923   return copy(MAX, table);
3924 }
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
```



```
3950 // halt the system.
3951 #include "types.h"
3952 #include "user.h"
3953
3954 int
3955 main(void) {
3956   halt();
3957   return 0;
3958 }
3959
3960
3961
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
```

```
4000 struct buf {
4001   int flags;
4002   uint dev;
4003   uint blockno;
4004   struct buf *prev; // LRU cache list
4005   struct buf *next;
4006   struct buf *qnext; // disk queue
4007   uchar data[BSIZE];
4008 };
4009 #define B_BUSY  0x1  // buffer is locked by some process
4010 #define B_VALID 0x2  // buffer has been read from disk
4011 #define B_DIRTY 0x4  // buffer needs to be written to disk
4012
4013
4014
4015
4016
4017
4018
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
```



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

```
4100 #define T_DIR  1   // Directory
4101 #define T_FILE 2   // File
4102 #define T_DEV  3   // Device
4103
4104 struct stat {
4105   short type;  // Type of file
4106   int dev;     // File system's disk device
4107   uint ino;    // Inode number
4108   short nlink; // Number of links to file
4109   uint size;   // Size of file in bytes
4110 };
4111
4112
4113
4114
4115
4116
4117
4118
4119
4120
4121
4122
4123
4124
4125
4126
4127
4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```



```
4150 // On-disk file system format.
4151 // Both the kernel and user programs use this header file.
4152
4153
4154 #define ROOTINO 1  // root i-number
4155 #define BSIZE 512  // block size
4156
4157 // Disk layout:
4158 // [ boot block | super block | log | inode blocks | free bit map | data bloc
4159 //
4160 // mkfs computes the super block and builds an initial file system. The super
4161 // the disk layout:
4162 struct superblock {
4163   uint size;         // Size of file system image (blocks)
4164   uint nblocks;      // Number of data blocks
4165   uint ninodes;      // Number of inodes.
4166   uint nlog;         // Number of log blocks
4167   uint logstart;     // Block number of first log block
4168   uint inodestart;   // Block number of first inode block
4169   uint bmapstart;    // Block number of first free map block
4170 };
4171
4172 #define NDIRECT 12
4173 #define NINDIRECT (BSIZE / sizeof(uint))
4174 #define MAXFILE (NDIRECT + NINDIRECT)
4175
4176 // On-disk inode structure
4177 struct dinode {
4178   short type;           // File type
4179   short major;          // Major device number (T_DEV only)
4180   short minor;          // Minor device number (T_DEV only)
4181   short nlink;          // Number of links to inode in file system
4182   uint size;            // Size of file (bytes)
4183   uint addrs[NDIRECT+1];   // Data block addresses
4184 };
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
```

```
4200 // Inodes per block.
4201 #define IPB           (BSIZE / sizeof(struct dinode))
4202
4203 // Block containing inode i
4204 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
4205
4206 // Bitmap bits per block
4207 #define BPB           (BSIZE*8)
4208
4209 // Block of free map containing bit for block b
4210 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
4211
4212 // Directory is a file containing a sequence of dirent structures.
4213 #define DIRSIZ 14
4214
4215 struct dirent {
4216   ushort inum;
4217   char name[DIRSIZ];
4218 };
4219
4220
4221
4222
4223
4224
4225
4226
4227
4228
4229
4230
4231
4232
4233
4234
4235
4236
4237
4238
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
```



```
4250 struct file {
4251   enum { FD_NONE, FD_PIPE, FD_INODE } type;
4252   int ref; // reference count
4253   char readable;
4254   char writable;
4255   struct pipe *pipe;
4256   struct inode *ip;
4257   uint off;
4258 };
4259
4260
4261 // in-memory copy of an inode
4262 struct inode {
4263   uint dev;           // Device number
4264   uint inum;          // Inode number
4265   int ref;            // Reference count
4266   int flags;          // I_BUSY, I_VALID
4267
4268   short type;         // copy of disk inode
4269   short major;
4270   short minor;
4271   short nlink;
4272   uint size;
4273   uint addrs[NDIRECT+1];
4274 };
4275 #define I_BUSY 0x1
4276 #define I_VALID 0x2
4277
4278 // table mapping major device number to
4279 // device functions
4280 struct devsw {
4281   int (*read)(struct inode*, char*, int);
4282   int (*write)(struct inode*, char*, int);
4283 };
4284
4285 extern struct devsw devsw[];
4286
4287 #define CONSOLE 1
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
```

```
4300 // Blank page.
4301
4302
4303
4304
4305
4306
4307
4308
4309
4310
4311
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 // Simple PIO-based (non-DMA) IDE driver code.
4351
4352 #include "types.h"
4353 #include "defs.h"
4354 #include "param.h"
4355 #include "memlayout.h"
4356 #include "mmu.h"
4357 #include "proc.h"
4358 #include "x86.h"
4359 #include "traps.h"
4360 #include "spinlock.h"
4361 #include "fs.h"
4362 #include "buf.h"
4363
4364 #define SECTOR_SIZE   512
4365 #define IDE_BSY       0x80
4366 #define IDE_DRDY      0x40
4367 #define IDE_DF        0x20
4368 #define IDE_ERR       0x01
4369
4370 #define IDE_CMD_READ  0x20
4371 #define IDE_CMD_WRITE 0x30
4372
4373 // idequeue points to the buf now being read/written to the disk.
4374 // idequeue->qnext points to the next buf to be processed.
4375 // You must hold idelock while manipulating queue.
4376
4377 static struct spinlock idelock;
4378 static struct buf *idequeue;
4379
4380 static int havedisk1;
4381 static void idestart(struct buf*);
4382
4383 // Wait for IDE disk to become ready.
4384 static int
4385 idewait(int checkerr)
4386 {
4387   int r;
4388
4389   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4390     ;
4391   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4392     return -1;
4393   return 0;
4394 }
4395
4396
4397
4398
4399
```

```
4400 void
4401 ideinit(void)
4402 {
4403   int i;
4404
4405   initlock(&idelock, "ide");
4406   picenable(IRQ_IDE);
4407   ioapicenable(IRQ_IDE, ncpu - 1);
4408   idewait(0);
4409
4410   // Check if disk 1 is present
4411   outb(0x1f6, 0xe0 | (1<<4));
4412   for(i=0; i<1000; i++){
4413     if(inb(0x1f7) != 0){
4414       havedisk1 = 1;
4415       break;
4416     }
4417   }
4418
4419   // Switch back to disk 0.
4420   outb(0x1f6, 0xe0 | (0<<4));
4421 }
4422
4423 // Start the request for b.  Caller must hold idelock.
4424 static void
4425 idestart(struct buf *b)
4426 {
4427   if(b == 0)
4428     panic("idestart");
4429   if(b->blockno >= FSSIZE)
4430     panic("incorrect blockno");
4431   int sector_per_block =  BSIZE/SECTOR_SIZE;
4432   int sector = b->blockno * sector_per_block;
4433
4434   if (sector_per_block > 7) panic("idestart");
4435
4436   idewait(0);
4437   outb(0x3f6, 0);  // generate interrupt
4438   outb(0x1f2, sector_per_block);  // number of sectors
4439   outb(0x1f3, sector & 0xff);
4440   outb(0x1f4, (sector >> 8) & 0xff);
4441   outb(0x1f5, (sector >> 16) & 0xff);
4442   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4443   if(b->flags & B_DIRTY){
4444     outb(0x1f7, IDE_CMD_WRITE);
4445     outsl(0x1f0, b->data, BSIZE/4);
4446   } else {
4447     outb(0x1f7, IDE_CMD_READ);
4448   }
4449 }
```



```
4450 // Interrupt handler.
4451 void
4452 ideintr(void)
4453 {
4454   struct buf *b;
4455
4456   // First queued buffer is the active request.
4457   acquire(&idelock);
4458   if((b = idequeue) == 0){
4459     release(&idelock);
4460     // cprintf("spurious IDE interrupt\n");
4461     return;
4462   }
4463   idequeue = b->qnext;
4464
4465   // Read data if needed.
4466   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4467     insl(0x1f0, b->data, BSIZE/4);
4468
4469   // Wake process waiting for this buf.
4470   b->flags |= B_VALID;
4471   b->flags &= ~B_DIRTY;
4472   wakeup(b);
4473
4474   // Start disk on next buf in queue.
4475   if(idequeue != 0)
4476     idestart(idequeue);
4477
4478   release(&idelock);
4479 }
4480
4481
4482
4483
4484
4485
4486
4487
4488
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499
```

```
4500 // Sync buf with disk.
4501 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
4502 // Else if B_VALID is not set, read buf from disk, set B_VALID.
4503 void
4504 iderw(struct buf *b)
4505 {
4506   struct buf **pp;
4507
4508   if(!(b->flags & B_BUSY))
4509     panic("iderw: buf not busy");
4510   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
4511     panic("iderw: nothing to do");
4512   if(b->dev != 0 && !havedisk1)
4513     panic("iderw: ide disk 1 not present");
4514
4515   acquire(&idelock);
4516
4517   // Append b to idequeue.
4518   b->qnext = 0;
4519   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
4520     ;
4521   *pp = b;
4522
4523   // Start disk if necessary.
4524   if(idequeue == b)
4525     idestart(b);
4526
4527   // Wait for request to finish.
4528   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
4529     sleep(b, &idelock);
4530   }
4531
4532   release(&idelock);
4533 }
4534
4535
4536
4537
4538
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4549
```



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

```
4600   // Create linked list of buffers
4601   bcache.head.prev = &bcache.head;
4602   bcache.head.next = &bcache.head;
4603   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4604     b->next = bcache.head.next;
4605     b->prev = &bcache.head;
4606     b->dev = -1;
4607     bcache.head.next->prev = b;
4608     bcache.head.next = b;
4609   }
4610 }
4611
4612 // Look through buffer cache for block on device dev.
4613 // If not found, allocate a buffer.
4614 // In either case, return B_BUSY buffer.
4615 static struct buf*
4616 bget(uint dev, uint blockno)
4617 {
4618   struct buf *b;
4619
4620   acquire(&bcache.lock);
4621
4622  loop:
4623   // Is the block already cached?
4624   for(b = bcache.head.next; b != &bcache.head; b = b->next){
4625     if(b->dev == dev && b->blockno == blockno){
4626       if(!(b->flags & B_BUSY)){
4627         b->flags |= B_BUSY;
4628         release(&bcache.lock);
4629         return b;
4630       }
4631       sleep(b, &bcache.lock);
4632       goto loop;
4633     }
4634   }
4635
4636   // Not cached; recycle some non-busy and clean buffer.
4637   // "clean" because B_DIRTY and !B_BUSY means log.c
4638   // hasn't yet committed the changes to the buffer.
4639   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4640     if((b->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
4641       b->dev = dev;
4642       b->blockno = blockno;
4643       b->flags = B_BUSY;
4644       release(&bcache.lock);
4645       return b;
4646     }
4647   }
4648   panic("bget: no buffers");
4649 }
```



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

```
4700 // Blank page.
4701
4702
4703
4704
4705
4706
4707
4708
4709
4710
4711
4712
4713
4714
4715
4716
4717
4718
4719
4720
4721
4722
4723
4724
4725
4726
4727
4728
4729
4730
4731
4732
4733
4734
4735
4736
4737
4738
4739
4740
4741
4742
4743
4744
4745
4746
4747
4748
4749
```



```
4750 #include "types.h"
4751 #include "defs.h"
4752 #include "param.h"
4753 #include "spinlock.h"
4754 #include "fs.h"
4755 #include "buf.h"
4756
4757 // Simple logging that allows concurrent FS system calls.
4758 //
4759 // A log transaction contains the updates of multiple FS system
4760 // calls. The logging system only commits when there are
4761 // no FS system calls active. Thus there is never
4762 // any reasoning required about whether a commit might
4763 // write an uncommitted system call's updates to disk.
4764 //
4765 // A system call should call begin_op()/end_op() to mark
4766 // its start and end. Usually begin_op() just increments
4767 // the count of in-progress FS system calls and returns.
4768 // But if it thinks the log is close to running out, it
4769 // sleeps until the last outstanding end_op() commits.
4770 //
4771 // The log is a physical re-do log containing disk blocks.
4772 // The on-disk log format:
4773 //   header block, containing block #s for block A, B, C, ...
4774 //   block A
4775 //   block B
4776 //   block C
4777 //   ...
4778 // Log appends are synchronous.
4779
4780 // Contents of the header block, used for both the on-disk header block
4781 // and to keep track in memory of logged block# before commit.
4782 struct logheader {
4783   int n;
4784   int block[LOGSIZE];
4785 };
4786
4787 struct log {
4788   struct spinlock lock;
4789   int start;
4790   int size;
4791   int outstanding; // how many FS sys calls are executing.
4792   int committing;  // in commit(), please wait.
4793   int dev;
4794   struct logheader lh;
4795 };
4796
4797
4798
4799
```

```
4800 struct log log;
4801
4802 static void recover_from_log(void);
4803 static void commit();
4804
4805 void
4806 initlog(int dev)
4807 {
4808   if (sizeof(struct logheader) >= BSIZE)
4809     panic("initlog: too big logheader");
4810
4811   struct superblock sb;
4812   initlock(&log.lock, "log");
4813   readsb(dev, &sb);
4814   log.start = sb.logstart;
4815   log.size = sb.nlog;
4816   log.dev = dev;
4817   recover_from_log();
4818 }
4819
4820 // Copy committed blocks from log to their home location
4821 static void
4822 install_trans(void)
4823 {
4824   int tail;
4825
4826   for (tail = 0; tail < log.lh.n; tail++) {
4827     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
4828     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
4829     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
4830     bwrite(dbuf);  // write dst to disk
4831     brelse(lbuf);
4832     brelse(dbuf);
4833   }
4834 }
4835
4836 // Read the log header from disk into the in-memory log header
4837 static void
4838 read_head(void)
4839 {
4840   struct buf *buf = bread(log.dev, log.start);
4841   struct logheader *lh = (struct logheader *) (buf->data);
4842   int i;
4843   log.lh.n = lh->n;
4844   for (i = 0; i < log.lh.n; i++) {
4845     log.lh.block[i] = lh->block[i];
4846   }
4847   brelse(buf);
4848 }
4849
```



```
4850 // Write in-memory log header to disk.
4851 // This is the true point at which the
4852 // current transaction commits.
4853 static void
4854 write_head(void)
4855 {
4856   struct buf *buf = bread(log.dev, log.start);
4857   struct logheader *hb = (struct logheader *) (buf->data);
4858   int i;
4859   hb->n = log.lh.n;
4860   for (i = 0; i < log.lh.n; i++) {
4861     hb->block[i] = log.lh.block[i];
4862   }
4863   bwrite(buf);
4864   brelse(buf);
4865 }
4866
4867 static void
4868 recover_from_log(void)
4869 {
4870   read_head();
4871   install_trans(); // if committed, copy from log to disk
4872   log.lh.n = 0;
4873   write_head(); // clear the log
4874 }
4875
4876 // called at the start of each FS system call.
4877 void
4878 begin_op(void)
4879 {
4880   acquire(&log.lock);
4881   while(1){
4882     if(log.committing){
4883       sleep(&log, &log.lock);
4884     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
4885       // this op might exhaust log space; wait for commit.
4886       sleep(&log, &log.lock);
4887     } else {
4888       log.outstanding += 1;
4889       release(&log.lock);
4890       break;
4891     }
4892   }
4893 }
4894
4895
4896
4897
4898
4899
```

```
4900 // called at the end of each FS system call.
4901 // commits if this was the last outstanding operation.
4902 void
4903 end_op(void)
4904 {
4905   int do_commit = 0;
4906
4907   acquire(&log.lock);
4908   log.outstanding -= 1;
4909   if(log.committing)
4910     panic("log.committing");
4911   if(log.outstanding == 0){
4912     do_commit = 1;
4913     log.committing = 1;
4914   } else {
4915     // begin_op() may be waiting for log space.
4916     wakeup(&log);
4917   }
4918   release(&log.lock);
4919
4920   if(do_commit){
4921     // call commit w/o holding locks, since not allowed
4922     // to sleep with locks.
4923     commit();
4924     acquire(&log.lock);
4925     log.committing = 0;
4926     wakeup(&log);
4927     release(&log.lock);
4928   }
4929 }
4930
4931 // Copy modified blocks from cache to log.
4932 static void
4933 write_log(void)
4934 {
4935   int tail;
4936
4937   for (tail = 0; tail < log.lh.n; tail++) {
4938     struct buf *to = bread(log.dev, log.start+tail+1); // log block
4939     struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
4940     memmove(to->data, from->data, BSIZE);
4941     bwrite(to);  // write the log
4942     brelse(from);
4943     brelse(to);
4944   }
4945 }
4946
4947
4948
4949
```



```
4950 static void
4951 commit()
4952 {
4953   if (log.lh.n > 0) {
4954     write_log();     // Write modified blocks from cache to log
4955     write_head();    // Write header to disk -- the real commit
4956     install_trans(); // Now install writes to home locations
4957     log.lh.n = 0;
4958     write_head();    // Erase the transaction from the log
4959   }
4960 }
4961
4962 // Caller has modified b->data and is done with the buffer.
4963 // Record the block number and pin in the cache with B_DIRTY.
4964 // commit()/write_log() will do the disk write.
4965 //
4966 // log_write() replaces bwrite(); a typical use is:
4967 //   bp = bread(...)
4968 //   modify bp->data[]
4969 //   log_write(bp)
4970 //   brelse(bp)
4971 void
4972 log_write(struct buf *b)
4973 {
4974   int i;
4975
4976   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
4977     panic("too big a transaction");
4978   if (log.outstanding < 1)
4979     panic("log_write outside of trans");
4980
4981   acquire(&log.lock);
4982   for (i = 0; i < log.lh.n; i++) {
4983     if (log.lh.block[i] == b->blockno)   // log absorbtion
4984       break;
4985   }
4986   log.lh.block[i] = b->blockno;
4987   if (i == log.lh.n)
4988     log.lh.n++;
4989   b->flags |= B_DIRTY; // prevent eviction
4990   release(&log.lock);
4991 }
4992
4993
4994
4995
4996
4997
4998
4999
```

```
5000 // File system implementation.  Five layers:
5001 //   + Blocks: allocator for raw disk blocks.
5002 //   + Log: crash recovery for multi-step updates.
5003 //   + Files: inode allocator, reading, writing, metadata.
5004 //   + Directories: inode with special contents (list of other inodes!)
5005 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
5006 //
5007 // This file contains the low-level file system manipulation
5008 // routines.  The (higher-level) system call implementations
5009 // are in sysfile.c.
5010
5011 #include "types.h"
5012 #include "defs.h"
5013 #include "param.h"
5014 #include "stat.h"
5015 #include "mmu.h"
5016 #include "proc.h"
5017 #include "spinlock.h"
5018 #include "fs.h"
5019 #include "buf.h"
5020 #include "file.h"
5021
5022 #define min(a, b) ((a) < (b) ? (a) : (b))
5023 static void itrunc(struct inode*);
5024 struct superblock sb;   // there should be one per dev, but we run with one d
5025
5026 // Read the super block.
5027 void
5028 readsb(int dev, struct superblock *sb)
5029 {
5030   struct buf *bp;
5031
5032   bp = bread(dev, 1);
5033   memmove(sb, bp->data, sizeof(*sb));
5034   brelse(bp);
5035 }
5036
5037 // Zero a block.
5038 static void
5039 bzero(int dev, int bno)
5040 {
5041   struct buf *bp;
5042
5043   bp = bread(dev, bno);
5044   memset(bp->data, 0, BSIZE);
5045   log_write(bp);
5046   brelse(bp);
5047 }
5048
5049
```



```
5050 // Blocks.
5051
5052 // Allocate a zeroed disk block.
5053 static uint
5054 balloc(uint dev)
5055 {
5056   int b, bi, m;
5057   struct buf *bp;
5058
5059   bp = 0;
5060   for(b = 0; b < sb.size; b += BPB){
5061     bp = bread(dev, BBLOCK(b, sb));
5062     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
5063       m = 1 << (bi % 8);
5064       if((bp->data[bi/8] & m) == 0){  // Is block free?
5065         bp->data[bi/8] |= m;  // Mark block in use.
5066         log_write(bp);
5067         brelse(bp);
5068         bzero(dev, b + bi);
5069         return b + bi;
5070       }
5071     }
5072     brelse(bp);
5073   }
5074   panic("balloc: out of blocks");
5075 }
5076
5077 // Free a disk block.
5078 static void
5079 bfree(int dev, uint b)
5080 {
5081   struct buf *bp;
5082   int bi, m;
5083
5084   readsb(dev, &sb);
5085   bp = bread(dev, BBLOCK(b, sb));
5086   bi = b % BPB;
5087   m = 1 << (bi % 8);
5088   if((bp->data[bi/8] & m) == 0)
5089     panic("freeing free block");
5090   bp->data[bi/8] &= ~m;
5091   log_write(bp);
5092   brelse(bp);
5093 }
5094
5095
5096
5097
5098
5099
```

```
5100 // Inodes.
5101 //
5102 // An inode describes a single unnamed file.
5103 // The inode disk structure holds metadata: the file's type,
5104 // its size, the number of links referring to it, and the
5105 // list of blocks holding the file's content.
5106 //
5107 // The inodes are laid out sequentially on disk at
5108 // sb.startinode. Each inode has a number, indicating its
5109 // position on the disk.
5110 //
5111 // The kernel keeps a cache of in-use inodes in memory
5112 // to provide a place for synchronizing access
5113 // to inodes used by multiple processes. The cached
5114 // inodes include book-keeping information that is
5115 // not stored on disk: ip->ref and ip->flags.
5116 //
5117 // An inode and its in-memory represtative go through a
5118 // sequence of states before they can be used by the
5119 // rest of the file system code.
5120 //
5121 // * Allocation: an inode is allocated if its type (on disk)
5122 //   is non-zero. ialloc() allocates, iput() frees if
5123 //   the link count has fallen to zero.
5124 //
5125 // * Referencing in cache: an entry in the inode cache
5126 //   is free if ip->ref is zero. Otherwise ip->ref tracks
5127 //   the number of in-memory pointers to the entry (open
5128 //   files and current directories). iget() to find or
5129 //   create a cache entry and increment its ref, iput()
5130 //   to decrement ref.
5131 //
5132 // * Valid: the information (type, size, &c) in an inode
5133 //   cache entry is only correct when the I_VALID bit
5134 //   is set in ip->flags. ilock() reads the inode from
5135 //   the disk and sets I_VALID, while iput() clears
5136 //   I_VALID if ip->ref has fallen to zero.
5137 //
5138 // * Locked: file system code may only examine and modify
5139 //   the information in an inode and its content if it
5140 //   has first locked the inode. The I_BUSY flag indicates
5141 //   that the inode is locked. ilock() sets I_BUSY,
5142 //   while iunlock clears it.
5143 //
5144 // Thus a typical sequence is:
5145 //   ip = iget(dev, inum)
5146 //   ilock(ip)
5147 //   ... examine and modify ip->xxx ...
5148 //   iunlock(ip)
5149 //   iput(ip)
```



```
5150 //
5151 // ilock() is separate from iget() so that system calls can
5152 // get a long-term reference to an inode (as for an open file)
5153 // and only lock it for short periods (e.g., in read()).
5154 // The separation also helps avoid deadlock and races during
5155 // pathname lookup. iget() increments ip->ref so that the inode
5156 // stays cached and pointers to it remain valid.
5157 //
5158 // Many internal file system functions expect the caller to
5159 // have locked the inodes involved; this lets callers create
5160 // multi-step atomic operations.
5161
5162 struct {
5163   struct spinlock lock;
5164   struct inode inode[NINODE];
5165 } icache;
5166
5167 void
5168 iinit(int dev)
5169 {
5170   initlock(&icache.lock, "icache");
5171   readsb(dev, &sb);
5172   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d inodestart %
5173           sb.nblocks, sb.ninodes, sb.nlog, sb.logstart, sb.inodestart, sb.bma
5174 }
5175
5176 static struct inode* iget(uint dev, uint inum);
5177
5178
5179
5180
5181
5182
5183
5184
5185
5186
5187
5188
5189
5190
5191
5192
5193
5194
5195
5196
5197
5198
5199
```

```
5200 // Allocate a new inode with the given type on device dev.
5201 // A free inode has a type of zero.
5202 struct inode*
5203 ialloc(uint dev, short type)
5204 {
5205   int inum;
5206   struct buf *bp;
5207   struct dinode *dip;
5208
5209   for(inum = 1; inum < sb.ninodes; inum++){
5210     bp = bread(dev, IBLOCK(inum, sb));
5211     dip = (struct dinode*)bp->data + inum%IPB;
5212     if(dip->type == 0){  // a free inode
5213       memset(dip, 0, sizeof(*dip));
5214       dip->type = type;
5215       log_write(bp);   // mark it allocated on the disk
5216       brelse(bp);
5217       return iget(dev, inum);
5218     }
5219     brelse(bp);
5220   }
5221   panic("ialloc: no inodes");
5222 }
5223
5224 // Copy a modified in-memory inode to disk.
5225 void
5226 iupdate(struct inode *ip)
5227 {
5228   struct buf *bp;
5229   struct dinode *dip;
5230
5231   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5232   dip = (struct dinode*)bp->data + ip->inum%IPB;
5233   dip->type = ip->type;
5234   dip->major = ip->major;
5235   dip->minor = ip->minor;
5236   dip->nlink = ip->nlink;
5237   dip->size = ip->size;
5238   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5239   log_write(bp);
5240   brelse(bp);
5241 }
5242
5243
5244
5245
5246
5247
5248
5249
```



```
5250 // Find the inode with number inum on device dev
5251 // and return the in-memory copy. Does not lock
5252 // the inode and does not read it from disk.
5253 static struct inode*
5254 iget(uint dev, uint inum)
5255 {
5256   struct inode *ip, *empty;
5257
5258   acquire(&icache.lock);
5259
5260   // Is the inode already cached?
5261   empty = 0;
5262   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5263     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5264       ip->ref++;
5265       release(&icache.lock);
5266       return ip;
5267     }
5268     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
5269       empty = ip;
5270   }
5271
5272   // Recycle an inode cache entry.
5273   if(empty == 0)
5274     panic("iget: no inodes");
5275
5276   ip = empty;
5277   ip->dev = dev;
5278   ip->inum = inum;
5279   ip->ref = 1;
5280   ip->flags = 0;
5281   release(&icache.lock);
5282
5283   return ip;
5284 }
5285
5286 // Increment reference count for ip.
5287 // Returns ip to enable ip = idup(ip1) idiom.
5288 struct inode*
5289 idup(struct inode *ip)
5290 {
5291   acquire(&icache.lock);
5292   ip->ref++;
5293   release(&icache.lock);
5294   return ip;
5295 }
5296
5297
5298
5299
```

```
5300 // Lock the given inode.
5301 // Reads the inode from disk if necessary.
5302 void
5303 ilock(struct inode *ip)
5304 {
5305   struct buf *bp;
5306   struct dinode *dip;
5307
5308   if(ip == 0 || ip->ref < 1)
5309     panic("ilock");
5310
5311   acquire(&icache.lock);
5312   while(ip->flags & I_BUSY)
5313     sleep(ip, &icache.lock);
5314   ip->flags |= I_BUSY;
5315   release(&icache.lock);
5316
5317   if(!(ip->flags & I_VALID)){
5318     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5319     dip = (struct dinode*)bp->data + ip->inum%IPB;
5320     ip->type = dip->type;
5321     ip->major = dip->major;
5322     ip->minor = dip->minor;
5323     ip->nlink = dip->nlink;
5324     ip->size = dip->size;
5325     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5326     brelse(bp);
5327     ip->flags |= I_VALID;
5328     if(ip->type == 0)
5329       panic("ilock: no type");
5330   }
5331 }
5332
5333 // Unlock the given inode.
5334 void
5335 iunlock(struct inode *ip)
5336 {
5337   if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
5338     panic("iunlock");
5339
5340   acquire(&icache.lock);
5341   ip->flags &= ~I_BUSY;
5342   wakeup(ip);
5343   release(&icache.lock);
5344 }
5345
5346
5347
5348
5349
```



```
5350 // Drop a reference to an in-memory inode.
5351 // If that was the last reference, the inode cache entry can
5352 // be recycled.
5353 // If that was the last reference and the inode has no links
5354 // to it, free the inode (and its content) on disk.
5355 // All calls to iput() must be inside a transaction in
5356 // case it has to free the inode.
5357 void
5358 iput(struct inode *ip)
5359 {
5360   acquire(&icache.lock);
5361   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
5362     // inode has no links and no other references: truncate and free.
5363     if(ip->flags & I_BUSY)
5364       panic("iput busy");
5365     ip->flags |= I_BUSY;
5366     release(&icache.lock);
5367     itrunc(ip);
5368     ip->type = 0;
5369     iupdate(ip);
5370     acquire(&icache.lock);
5371     ip->flags = 0;
5372     wakeup(ip);
5373   }
5374   ip->ref--;
5375   release(&icache.lock);
5376 }
5377
5378 // Common idiom: unlock, then put.
5379 void
5380 iunlockput(struct inode *ip)
5381 {
5382   iunlock(ip);
5383   iput(ip);
5384 }
5385
5386
5387
5388
5389
5390
5391
5392
5393
5394
5395
5396
5397
5398
5399
```

```
5400 // Inode content
5401 //
5402 // The content (data) associated with each inode is stored
5403 // in blocks on the disk. The first NDIRECT block numbers
5404 // are listed in ip->addrs[].  The next NINDIRECT blocks are
5405 // listed in block ip->addrs[NDIRECT].
5406
5407 // Return the disk block address of the nth block in inode ip.
5408 // If there is no such block, bmap allocates one.
5409 static uint
5410 bmap(struct inode *ip, uint bn)
5411 {
5412   uint addr, *a;
5413   struct buf *bp;
5414
5415   if(bn < NDIRECT){
5416     if((addr = ip->addrs[bn]) == 0)
5417       ip->addrs[bn] = addr = balloc(ip->dev);
5418     return addr;
5419   }
5420   bn -= NDIRECT;
5421
5422   if(bn < NINDIRECT){
5423     // Load indirect block, allocating if necessary.
5424     if((addr = ip->addrs[NDIRECT]) == 0)
5425       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5426     bp = bread(ip->dev, addr);
5427     a = (uint*)bp->data;
5428     if((addr = a[bn]) == 0){
5429       a[bn] = addr = balloc(ip->dev);
5430       log_write(bp);
5431     }
5432     brelse(bp);
5433     return addr;
5434   }
5435
5436   panic("bmap: out of range");
5437 }
5438
5439
5440
5441
5442
5443
5444
5445
5446
5447
5448
5449
```



```
5450 // Truncate inode (discard contents).
5451 // Only called when the inode has no links
5452 // to it (no directory entries referring to it)
5453 // and has no in-memory reference to it (is
5454 // not an open file or current directory).
5455 static void
5456 itrunc(struct inode *ip)
5457 {
5458   int i, j;
5459   struct buf *bp;
5460   uint *a;
5461
5462   for(i = 0; i < NDIRECT; i++){
5463     if(ip->addrs[i]){
5464       bfree(ip->dev, ip->addrs[i]);
5465       ip->addrs[i] = 0;
5466     }
5467   }
5468
5469   if(ip->addrs[NDIRECT]){
5470     bp = bread(ip->dev, ip->addrs[NDIRECT]);
5471     a = (uint*)bp->data;
5472     for(j = 0; j < NINDIRECT; j++){
5473       if(a[j])
5474         bfree(ip->dev, a[j]);
5475     }
5476     brelse(bp);
5477     bfree(ip->dev, ip->addrs[NDIRECT]);
5478     ip->addrs[NDIRECT] = 0;
5479   }
5480
5481   ip->size = 0;
5482   iupdate(ip);
5483 }
5484
5485 // Copy stat information from inode.
5486 void
5487 stati(struct inode *ip, struct stat *st)
5488 {
5489   st->dev = ip->dev;
5490   st->ino = ip->inum;
5491   st->type = ip->type;
5492   st->nlink = ip->nlink;
5493   st->size = ip->size;
5494 }
5495
5496
5497
5498
5499
```

```
5500 // Read data from inode.
5501 int
5502 readi(struct inode *ip, char *dst, uint off, uint n)
5503 {
5504   uint tot, m;
5505   struct buf *bp;
5506
5507   if(ip->type == T_DEV){
5508     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
5509       return -1;
5510     return devsw[ip->major].read(ip, dst, n);
5511   }
5512
5513   if(off > ip->size || off + n < off)
5514     return -1;
5515   if(off + n > ip->size)
5516     n = ip->size - off;
5517
5518   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5519     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5520     m = min(n - tot, BSIZE - off%BSIZE);
5521     memmove(dst, bp->data + off%BSIZE, m);
5522     brelse(bp);
5523   }
5524   return n;
5525 }
5526
5527
5528
5529
5530
5531
5532
5533
5534
5535
5536
5537
5538
5539
5540
5541
5542
5543
5544
5545
5546
5547
5548
5549
```



```
5550 // Write data to inode.
5551 int
5552 writei(struct inode *ip, char *src, uint off, uint n)
5553 {
5554   uint tot, m;
5555   struct buf *bp;
5556
5557   if(ip->type == T_DEV){
5558     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
5559       return -1;
5560     return devsw[ip->major].write(ip, src, n);
5561   }
5562
5563   if(off > ip->size || off + n < off)
5564     return -1;
5565   if(off + n > MAXFILE*BSIZE)
5566     return -1;
5567
5568   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
5569     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5570     m = min(n - tot, BSIZE - off%BSIZE);
5571     memmove(bp->data + off%BSIZE, src, m);
5572     log_write(bp);
5573     brelse(bp);
5574   }
5575
5576   if(n > 0 && off > ip->size){
5577     ip->size = off;
5578     iupdate(ip);
5579   }
5580   return n;
5581 }
5582
5583
5584
5585
5586
5587
5588
5589
5590
5591
5592
5593
5594
5595
5596
5597
5598
5599
```

```
5600 // Directories
5601
5602 int
5603 namecmp(const char *s, const char *t)
5604 {
5605   return strncmp(s, t, DIRSIZ);
5606 }
5607
5608 // Look for a directory entry in a directory.
5609 // If found, set *poff to byte offset of entry.
5610 struct inode*
5611 dirlookup(struct inode *dp, char *name, uint *poff)
5612 {
5613   uint off, inum;
5614   struct dirent de;
5615
5616   if(dp->type != T_DIR)
5617     panic("dirlookup not DIR");
5618
5619   for(off = 0; off < dp->size; off += sizeof(de)){
5620     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5621       panic("dirlink read");
5622     if(de.inum == 0)
5623       continue;
5624     if(namecmp(name, de.name) == 0){
5625       // entry matches path element
5626       if(poff)
5627         *poff = off;
5628       inum = de.inum;
5629       return iget(dp->dev, inum);
5630     }
5631   }
5632
5633   return 0;
5634 }
5635
5636
5637
5638
5639
5640
5641
5642
5643
5644
5645
5646
5647
5648
5649
```



```
5650 // Write a new directory entry (name, inum) into the directory dp.
5651 int
5652 dirlink(struct inode *dp, char *name, uint inum)
5653 {
5654   int off;
5655   struct dirent de;
5656   struct inode *ip;
5657
5658   // Check that name is not present.
5659   if((ip = dirlookup(dp, name, 0)) != 0){
5660     iput(ip);
5661     return -1;
5662   }
5663
5664   // Look for an empty dirent.
5665   for(off = 0; off < dp->size; off += sizeof(de)){
5666     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5667       panic("dirlink read");
5668     if(de.inum == 0)
5669       break;
5670   }
5671
5672   strncpy(de.name, name, DIRSIZ);
5673   de.inum = inum;
5674   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5675     panic("dirlink");
5676
5677   return 0;
5678 }
5679
5680
5681
5682
5683
5684
5685
5686
5687
5688
5689
5690
5691
5692
5693
5694
5695
5696
5697
5698
5699
```

```
5700 // Paths
5701
5702 // Copy the next path element from path into name.
5703 // Return a pointer to the element following the copied one.
5704 // The returned path has no leading slashes,
5705 // so the caller can check *path=='\0' to see if the name is the last one.
5706 // If no name to remove, return 0.
5707 //
5708 // Examples:
5709 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5710 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5711 //   skipelem("a", name) = "", setting name = "a"
5712 //   skipelem("", name) = skipelem("////", name) = 0
5713 //
5714 static char*
5715 skipelem(char *path, char *name)
5716 {
5717   char *s;
5718   int len;
5719
5720   while(*path == '/')
5721     path++;
5722   if(*path == 0)
5723     return 0;
5724   s = path;
5725   while(*path != '/' && *path != 0)
5726     path++;
5727   len = path - s;
5728   if(len >= DIRSIZ)
5729     memmove(name, s, DIRSIZ);
5730   else {
5731     memmove(name, s, len);
5732     name[len] = 0;
5733   }
5734   while(*path == '/')
5735     path++;
5736   return path;
5737 }
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749
```



```
5750 // Look up and return the inode for a path name.
5751 // If parent != 0, return the inode for the parent and copy the final
5752 // path element into name, which must have room for DIRSIZ bytes.
5753 // Must be called inside a transaction since it calls iput().
5754 static struct inode*
5755 namex(char *path, int nameiparent, char *name)
5756 {
5757   struct inode *ip, *next;
5758
5759   if(*path == '/')
5760     ip = iget(ROOTDEV, ROOTINO);
5761   else
5762     ip = idup(proc->cwd);
5763
5764   while((path = skipelem(path, name)) != 0){
5765     ilock(ip);
5766     if(ip->type != T_DIR){
5767       iunlockput(ip);
5768       return 0;
5769     }
5770     if(nameiparent && *path == '\0'){
5771       // Stop one level early.
5772       iunlock(ip);
5773       return ip;
5774     }
5775     if((next = dirlookup(ip, name, 0)) == 0){
5776       iunlockput(ip);
5777       return 0;
5778     }
5779     iunlockput(ip);
5780     ip = next;
5781   }
5782   if(nameiparent){
5783     iput(ip);
5784     return 0;
5785   }
5786   return ip;
5787 }
5788
5789 struct inode*
5790 namei(char *path)
5791 {
5792   char name[DIRSIZ];
5793   return namex(path, 0, name);
5794 }
5795
5796
5797
5798
5799
```

```
5800 struct inode*
5801 nameiparent(char *path, char *name)
5802 {
5803   return namex(path, 1, name);
5804 }
5805
5806
5807
5808
5809
5810
5811
5812
5813
5814
5815
5816
5817
5818
5819
5820
5821
5822
5823
5824
5825
5826
5827
5828
5829
5830
5831
5832
5833
5834
5835
5836
5837
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```



```
5850 //
5851 // File descriptors
5852 //
5853
5854 #include "types.h"
5855 #include "defs.h"
5856 #include "param.h"
5857 #include "fs.h"
5858 #include "file.h"
5859 #include "spinlock.h"
5860
5861 struct devsw devsw[NDEV];
5862 struct {
5863   struct spinlock lock;
5864   struct file file[NFILE];
5865 } ftable;
5866
5867 void
5868 fileinit(void)
5869 {
5870   initlock(&ftable.lock, "ftable");
5871 }
5872
5873 // Allocate a file structure.
5874 struct file*
5875 filealloc(void)
5876 {
5877   struct file *f;
5878
5879   acquire(&ftable.lock);
5880   for(f = ftable.file; f < ftable.file + NFILE; f++){
5881     if(f->ref == 0){
5882       f->ref = 1;
5883       release(&ftable.lock);
5884       return f;
5885     }
5886   }
5887   release(&ftable.lock);
5888   return 0;
5889 }
5890
5891
5892
5893
5894
5895
5896
5897
5898
5899
```

```
5900 // Increment ref count for file f.
5901 struct file*
5902 filedup(struct file *f)
5903 {
5904   acquire(&ftable.lock);
5905   if(f->ref < 1)
5906     panic("filedup");
5907   f->ref++;
5908   release(&ftable.lock);
5909   return f;
5910 }
5911
5912 // Close file f.  (Decrement ref count, close when reaches 0.)
5913 void
5914 fileclose(struct file *f)
5915 {
5916   struct file ff;
5917
5918   acquire(&ftable.lock);
5919   if(f->ref < 1)
5920     panic("fileclose");
5921   if(--f->ref > 0){
5922     release(&ftable.lock);
5923     return;
5924   }
5925   ff = *f;
5926   f->ref = 0;
5927   f->type = FD_NONE;
5928   release(&ftable.lock);
5929
5930   if(ff.type == FD_PIPE)
5931     pipeclose(ff.pipe, ff.writable);
5932   else if(ff.type == FD_INODE){
5933     begin_op();
5934     iput(ff.ip);
5935     end_op();
5936   }
5937 }
5938
5939
5940
5941
5942
5943
5944
5945
5946
5947
5948
5949
```



```
5950 // Get metadata about file f.
5951 int
5952 filestat(struct file *f, struct stat *st)
5953 {
5954   if(f->type == FD_INODE){
5955     ilock(f->ip);
5956     stati(f->ip, st);
5957     iunlock(f->ip);
5958     return 0;
5959   }
5960   return -1;
5961 }
5962
5963 // Read from file f.
5964 int
5965 fileread(struct file *f, char *addr, int n)
5966 {
5967   int r;
5968
5969   if(f->readable == 0)
5970     return -1;
5971   if(f->type == FD_PIPE)
5972     return piperead(f->pipe, addr, n);
5973   if(f->type == FD_INODE){
5974     ilock(f->ip);
5975     if((r = readi(f->ip, addr, f->off, n)) > 0)
5976       f->off += r;
5977     iunlock(f->ip);
5978     return r;
5979   }
5980   panic("fileread");
5981 }
5982
5983
5984
5985
5986
5987
5988
5989
5990
5991
5992
5993
5994
5995
5996
5997
5998
5999
```

```
6000 // Write to file f.
6001 int
6002 filewrite(struct file *f, char *addr, int n)
6003 {
6004   int r;
6005
6006   if(f->writable == 0)
6007     return -1;
6008   if(f->type == FD_PIPE)
6009     return pipewrite(f->pipe, addr, n);
6010   if(f->type == FD_INODE){
6011     // write a few blocks at a time to avoid exceeding
6012     // the maximum log transaction size, including
6013     // i-node, indirect block, allocation blocks,
6014     // and 2 blocks of slop for non-aligned writes.
6015     // this really belongs lower down, since writei()
6016     // might be writing a device like the console.
6017     int max = ((LOGSIZE-1-1-2) / 2) * 512;
6018     int i = 0;
6019     while(i < n){
6020       int n1 = n - i;
6021       if(n1 > max)
6022         n1 = max;
6023
6024       begin_op();
6025       ilock(f->ip);
6026       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
6027         f->off += r;
6028       iunlock(f->ip);
6029       end_op();
6030
6031       if(r < 0)
6032         break;
6033       if(r != n1)
6034         panic("short filewrite");
6035       i += r;
6036     }
6037     return i == n ? n : -1;
6038   }
6039   panic("filewrite");
6040 }
6041
6042
6043
6044
6045
6046
6047
6048
6049
```



```
6050 //
6051 // File-system system calls.
6052 // Mostly argument checking, since we don't trust
6053 // user code, and calls into file.c and fs.c.
6054 //
6055
6056 #include "types.h"
6057 #include "defs.h"
6058 #include "param.h"
6059 #include "stat.h"
6060 #include "mmu.h"
6061 #include "proc.h"
6062 #include "fs.h"
6063 #include "file.h"
6064 #include "fcntl.h"
6065
6066 // Fetch the nth word-sized system call argument as a file descriptor
6067 // and return both the descriptor and the corresponding struct file.
6068 static int
6069 argfd(int n, int *pfd, struct file **pf)
6070 {
6071   int fd;
6072   struct file *f;
6073
6074   if(argint(n, &fd) < 0)
6075     return -1;
6076   if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
6077     return -1;
6078   if(pfd)
6079     *pfd = fd;
6080   if(pf)
6081     *pf = f;
6082   return 0;
6083 }
6084
6085 // Allocate a file descriptor for the given file.
6086 // Takes over file reference from caller on success.
6087 static int
6088 fdalloc(struct file *f)
6089 {
6090   int fd;
6091
6092   for(fd = 0; fd < NOFILE; fd++){
6093     if(proc->ofile[fd] == 0){
6094       proc->ofile[fd] = f;
6095       return fd;
6096     }
6097   }
6098   return -1;
6099 }
```

```
6100 int
6101 sys_dup(void)
6102 {
6103   struct file *f;
6104   int fd;
6105
6106   if(argfd(0, 0, &f) < 0)
6107     return -1;
6108   if((fd=fdalloc(f)) < 0)
6109     return -1;
6110   filedup(f);
6111   return fd;
6112 }
6113
6114 int
6115 sys_read(void)
6116 {
6117   struct file *f;
6118   int n;
6119   char *p;
6120
6121   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6122     return -1;
6123   return fileread(f, p, n);
6124 }
6125
6126 int
6127 sys_write(void)
6128 {
6129   struct file *f;
6130   int n;
6131   char *p;
6132
6133   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6134     return -1;
6135   return filewrite(f, p, n);
6136 }
6137
6138 int
6139 sys_close(void)
6140 {
6141   int fd;
6142   struct file *f;
6143
6144   if(argfd(0, &fd, &f) < 0)
6145     return -1;
6146   proc->ofile[fd] = 0;
6147   fileclose(f);
6148   return 0;
6149 }
```



```
6150 int
6151 sys_fstat(void)
6152 {
6153   struct file *f;
6154   struct stat *st;
6155
6156   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
6157     return -1;
6158   return filestat(f, st);
6159 }
6160
6161 // Create the path new as a link to the same inode as old.
6162 int
6163 sys_link(void)
6164 {
6165   char name[DIRSIZ], *new, *old;
6166   struct inode *dp, *ip;
6167
6168   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
6169     return -1;
6170
6171   begin_op();
6172   if((ip = namei(old)) == 0){
6173     end_op();
6174     return -1;
6175   }
6176
6177   ilock(ip);
6178   if(ip->type == T_DIR){
6179     iunlockput(ip);
6180     end_op();
6181     return -1;
6182   }
6183
6184   ip->nlink++;
6185   iupdate(ip);
6186   iunlock(ip);
6187
6188   if((dp = nameiparent(new, name)) == 0)
6189     goto bad;
6190   ilock(dp);
6191   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
6192     iunlockput(dp);
6193     goto bad;
6194   }
6195   iunlockput(dp);
6196   iput(ip);
6197
6198   end_op();
6199
```

```
6200   return 0;
6201
6202 bad:
6203   ilock(ip);
6204   ip->nlink--;
6205   iupdate(ip);
6206   iunlockput(ip);
6207   end_op();
6208   return -1;
6209 }
6210
6211 // Is the directory dp empty except for "." and ".." ?
6212 static int
6213 isdirempty(struct inode *dp)
6214 {
6215   int off;
6216   struct dirent de;
6217
6218   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
6219     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6220       panic("isdirempty: readi");
6221     if(de.inum != 0)
6222       return 0;
6223   }
6224   return 1;
6225 }
6226
6227
6228
6229
6230
6231
6232
6233
6234
6235
6236
6237
6238
6239
6240
6241
6242
6243
6244
6245
6246
6247
6248
6249
```



```
6250 int
6251 sys_unlink(void)
6252 {
6253   struct inode *ip, *dp;
6254   struct dirent de;
6255   char name[DIRSIZ], *path;
6256   uint off;
6257
6258   if(argstr(0, &path) < 0)
6259     return -1;
6260
6261   begin_op();
6262   if((dp = nameiparent(path, name)) == 0){
6263     end_op();
6264     return -1;
6265   }
6266
6267   ilock(dp);
6268
6269   // Cannot unlink "." or "..".
6270   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6271     goto bad;
6272
6273   if((ip = dirlookup(dp, name, &off)) == 0)
6274     goto bad;
6275   ilock(ip);
6276
6277   if(ip->nlink < 1)
6278     panic("unlink: nlink < 1");
6279   if(ip->type == T_DIR && !isdirempty(ip)){
6280     iunlockput(ip);
6281     goto bad;
6282   }
6283
6284   memset(&de, 0, sizeof(de));
6285   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6286     panic("unlink: writei");
6287   if(ip->type == T_DIR){
6288     dp->nlink--;
6289     iupdate(dp);
6290   }
6291   iunlockput(dp);
6292
6293   ip->nlink--;
6294   iupdate(ip);
6295   iunlockput(ip);
6296
6297   end_op();
6298
6299   return 0;
```

```
6300 bad:
6301   iunlockput(dp);
6302   end_op();
6303   return -1;
6304 }
6305
6306 static struct inode*
6307 create(char *path, short type, short major, short minor)
6308 {
6309   uint off;
6310   struct inode *ip, *dp;
6311   char name[DIRSIZ];
6312
6313   if((dp = nameiparent(path, name)) == 0)
6314     return 0;
6315   ilock(dp);
6316
6317   if((ip = dirlookup(dp, name, &off)) != 0){
6318     iunlockput(dp);
6319     ilock(ip);
6320     if(type == T_FILE && ip->type == T_FILE)
6321       return ip;
6322     iunlockput(ip);
6323     return 0;
6324   }
6325
6326   if((ip = ialloc(dp->dev, type)) == 0)
6327     panic("create: ialloc");
6328
6329   ilock(ip);
6330   ip->major = major;
6331   ip->minor = minor;
6332   ip->nlink = 1;
6333   iupdate(ip);
6334
6335   if(type == T_DIR){  // Create . and .. entries.
6336     dp->nlink++;  // for ".."
6337     iupdate(dp);
6338     // No ip->nlink++ for ".": avoid cyclic ref count.
6339     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6340       panic("create dots");
6341   }
6342
6343   if(dirlink(dp, name, ip->inum) < 0)
6344     panic("create: dirlink");
6345
6346   iunlockput(dp);
6347
6348   return ip;
6349 }
```



```
6350 int
6351 sys_open(void)
6352 {
6353   char *path;
6354   int fd, omode;
6355   struct file *f;
6356   struct inode *ip;
6357
6358   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6359     return -1;
6360
6361   begin_op();
6362
6363   if(omode & O_CREATE){
6364     ip = create(path, T_FILE, 0, 0);
6365     if(ip == 0){
6366       end_op();
6367       return -1;
6368     }
6369   } else {
6370     if((ip = namei(path)) == 0){
6371       end_op();
6372       return -1;
6373     }
6374     ilock(ip);
6375     if(ip->type == T_DIR && omode != O_RDONLY){
6376       iunlockput(ip);
6377       end_op();
6378       return -1;
6379     }
6380   }
6381
6382   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6383     if(f)
6384       fileclose(f);
6385     iunlockput(ip);
6386     end_op();
6387     return -1;
6388   }
6389   iunlock(ip);
6390   end_op();
6391
6392   f->type = FD_INODE;
6393   f->ip = ip;
6394   f->off = 0;
6395   f->readable = !(omode & O_WRONLY);
6396   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6397   return fd;
6398 }
6399
```

```
6400 int
6401 sys_mkdir(void)
6402 {
6403   char *path;
6404   struct inode *ip;
6405
6406   begin_op();
6407   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
6408     end_op();
6409     return -1;
6410   }
6411   iunlockput(ip);
6412   end_op();
6413   return 0;
6414 }
6415
6416 int
6417 sys_mknod(void)
6418 {
6419   struct inode *ip;
6420   char *path;
6421   int len;
6422   int major, minor;
6423
6424   begin_op();
6425   if((len=argstr(0, &path)) < 0 ||
6426      argint(1, &major) < 0 ||
6427      argint(2, &minor) < 0 ||
6428      (ip = create(path, T_DEV, major, minor)) == 0){
6429     end_op();
6430     return -1;
6431   }
6432   iunlockput(ip);
6433   end_op();
6434   return 0;
6435 }
6436
6437
6438
6439
6440
6441
6442
6443
6444
6445
6446
6447
6448
6449
```



```
6450 int
6451 sys_chdir(void)
6452 {
6453   char *path;
6454   struct inode *ip;
6455
6456   begin_op();
6457   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
6458     end_op();
6459     return -1;
6460   }
6461   ilock(ip);
6462   if(ip->type != T_DIR){
6463     iunlockput(ip);
6464     end_op();
6465     return -1;
6466   }
6467   iunlock(ip);
6468   iput(proc->cwd);
6469   end_op();
6470   proc->cwd = ip;
6471   return 0;
6472 }
6473
6474 int
6475 sys_exec(void)
6476 {
6477   char *path, *argv[MAXARG];
6478   int i;
6479   uint uargv, uarg;
6480
6481   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6482     return -1;
6483   }
6484   memset(argv, 0, sizeof(argv));
6485   for(i=0;; i++){
6486     if(i >= NELEM(argv))
6487       return -1;
6488     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6489       return -1;
6490     if(uarg == 0){
6491       argv[i] = 0;
6492       break;
6493     }
6494     if(fetchstr(uarg, &argv[i]) < 0)
6495       return -1;
6496   }
6497   return exec(path, argv);
6498 }
6499
```

```
6500 int
6501 sys_pipe(void)
6502 {
6503   int *fd;
6504   struct file *rf, *wf;
6505   int fd0, fd1;
6506
6507   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6508     return -1;
6509   if(pipealloc(&rf, &wf) < 0)
6510     return -1;
6511   fd0 = -1;
6512   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6513     if(fd0 >= 0)
6514       proc->ofile[fd0] = 0;
6515     fileclose(rf);
6516     fileclose(wf);
6517     return -1;
6518   }
6519   fd[0] = fd0;
6520   fd[1] = fd1;
6521   return 0;
6522 }
6523
6524
6525
6526
6527
6528
6529
6530
6531
6532
6533
6534
6535
6536
6537
6538
6539
6540
6541
6542
6543
6544
6545
6546
6547
6548
6549
```



```
6550 #include "types.h"
6551 #include "param.h"
6552 #include "memlayout.h"
6553 #include "mmu.h"
6554 #include "proc.h"
6555 #include "defs.h"
6556 #include "x86.h"
6557 #include "elf.h"
6558
6559 int
6560 exec(char *path, char **argv)
6561 {
6562   char *s, *last;
6563   int i, off;
6564   uint argc, sz, sp, ustack[3+MAXARG+1];
6565   struct elfhdr elf;
6566   struct inode *ip;
6567   struct proghdr ph;
6568   pde_t *pgdir, *oldpgdir;
6569
6570   begin_op();
6571   if((ip = namei(path)) == 0){
6572     end_op();
6573     return -1;
6574   }
6575   ilock(ip);
6576   pgdir = 0;
6577
6578   // Check ELF header
6579   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
6580     goto bad;
6581   if(elf.magic != ELF_MAGIC)
6582     goto bad;
6583
6584   if((pgdir = setupkvm()) == 0)
6585     goto bad;
6586
6587   // Load program into memory.
6588   sz = 0;
6589   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6590     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6591       goto bad;
6592     if(ph.type != ELF_PROG_LOAD)
6593       continue;
6594     if(ph.memsz < ph.filesz)
6595       goto bad;
6596     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6597       goto bad;
6598     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6599       goto bad;
```

```
  }
  iunlockput(ip);
  end_op();
  ip = 0;

  // Allocate two pages at the next page boundary.
  // Make the first inaccessible.  Use the second as the user stack.
  sz = PGROUNDUP(sz);
  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
  sp = sz;

  // Push argument strings, prepare rest of stack in ustack.
  for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
      goto bad;
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
      goto bad;
    ustack[3+argc] = sp;
  }
  ustack[3+argc] = 0;

  ustack[0] = 0xffffffff;  // fake return PC
  ustack[1] = argc;
  ustack[2] = sp - (argc+1)*4;  // argv pointer

  sp -= (3+argc+1) * 4;
  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;

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

  // Commit to the user image.
  oldpgdir = proc->pgdir;
  proc->pgdir = pgdir;
  proc->sz = sz;
  proc->tf->eip = elf.entry;  // main
  proc->tf->esp = sp;
  switchuvm(proc);
  freevm(oldpgdir);
  return 0;


```



```
 bad:
  if(pgdir)
    freevm(pgdir);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
}


```

```
6700 #include "types.h"
6701 #include "defs.h"
6702 #include "param.h"
6703 #include "mmu.h"
6704 #include "proc.h"
6705 #include "fs.h"
6706 #include "file.h"
6707 #include "spinlock.h"
6708
6709 #define PIPESIZE 512
6710
6711 struct pipe {
6712   struct spinlock lock;
6713   char data[PIPESIZE];
6714   uint nread;     // number of bytes read
6715   uint nwrite;    // number of bytes written
6716   int readopen;   // read fd is still open
6717   int writeopen;  // write fd is still open
6718 };
6719
6720 int
6721 pipealloc(struct file **f0, struct file **f1)
6722 {
6723   struct pipe *p;
6724
6725   p = 0;
6726   *f0 = *f1 = 0;
6727   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
6728     goto bad;
6729   if((p = (struct pipe*)kalloc()) == 0)
6730     goto bad;
6731   p->readopen = 1;
6732   p->writeopen = 1;
6733   p->nwrite = 0;
6734   p->nread = 0;
6735   initlock(&p->lock, "pipe");
6736   (*f0)->type = FD_PIPE;
6737   (*f0)->readable = 1;
6738   (*f0)->writable = 0;
6739   (*f0)->pipe = p;
6740   (*f1)->type = FD_PIPE;
6741   (*f1)->readable = 0;
6742   (*f1)->writable = 1;
6743   (*f1)->pipe = p;
6744   return 0;
6745
6746
6747
6748
6749
```



```
6750  bad:
6751   if(p)
6752     kfree((char*)p);
6753   if(*f0)
6754     fileclose(*f0);
6755   if(*f1)
6756     fileclose(*f1);
6757   return -1;
6758 }
6759
6760 void
6761 pipeclose(struct pipe *p, int writable)
6762 {
6763   acquire(&p->lock);
6764   if(writable){
6765     p->writeopen = 0;
6766     wakeup(&p->nread);
6767   } else {
6768     p->readopen = 0;
6769     wakeup(&p->nwrite);
6770   }
6771   if(p->readopen == 0 && p->writeopen == 0){
6772     release(&p->lock);
6773     kfree((char*)p);
6774   } else
6775     release(&p->lock);
6776 }
6777
6778
6779 int
6780 pipewrite(struct pipe *p, char *addr, int n)
6781 {
6782   int i;
6783
6784   acquire(&p->lock);
6785   for(i = 0; i < n; i++){
6786     while(p->nwrite == p->nread + PIPESIZE){
6787       if(p->readopen == 0 || proc->killed){
6788         release(&p->lock);
6789         return -1;
6790       }
6791       wakeup(&p->nread);
6792       sleep(&p->nwrite, &p->lock);
6793     }
6794     p->data[p->nwrite++ % PIPESIZE] = addr[i];
6795   }
6796   wakeup(&p->nread);
6797   release(&p->lock);
6798   return n;
6799 }
```

```
6800 int
6801 piperead(struct pipe *p, char *addr, int n)
6802 {
6803   int i;
6804
6805   acquire(&p->lock);
6806   while(p->nread == p->nwrite && p->writeopen){
6807     if(proc->killed){
6808       release(&p->lock);
6809       return -1;
6810     }
6811     sleep(&p->nread, &p->lock);
6812   }
6813   for(i = 0; i < n; i++){
6814     if(p->nread == p->nwrite)
6815       break;
6816     addr[i] = p->data[p->nread++ % PIPESIZE];
6817   }
6818   wakeup(&p->nwrite);
6819   release(&p->lock);
6820   return i;
6821 }
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 "x86.h"
6852
6853 void*
6854 memset(void *dst, int c, uint n)
6855 {
6856   if ((int)dst%4 == 0 && n%4 == 0){
6857     c &= 0xFF;
6858     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
6859   } else
6860     stosb(dst, c, n);
6861   return dst;
6862 }
6863
6864 int
6865 memcmp(const void *v1, const void *v2, uint n)
6866 {
6867   const uchar *s1, *s2;
6868
6869   s1 = v1;
6870   s2 = v2;
6871   while(n-- > 0){
6872     if(*s1 != *s2)
6873       return *s1 - *s2;
6874     s1++, s2++;
6875   }
6876
6877   return 0;
6878 }
6879
6880 void*
6881 memmove(void *dst, const void *src, uint n)
6882 {
6883   const char *s;
6884   char *d;
6885
6886   s = src;
6887   d = dst;
6888   if(s < d && s + n > d){
6889     s += n;
6890     d += n;
6891     while(n-- > 0)
6892       *--d = *--s;
6893   } else
6894     while(n-- > 0)
6895       *d++ = *s++;
6896
6897   return dst;
6898 }
6899
```

```
6900 // memcpy exists to placate GCC.  Use memmove.
6901 void*
6902 memcpy(void *dst, const void *src, uint n)
6903 {
6904   return memmove(dst, src, n);
6905 }
6906
6907 int
6908 strncmp(const char *p, const char *q, uint n)
6909 {
6910   while(n > 0 && *p && *p == *q)
6911     n--, p++, q++;
6912   if(n == 0)
6913     return 0;
6914   return (uchar)*p - (uchar)*q;
6915 }
6916
6917 char*
6918 strncpy(char *s, const char *t, int n)
6919 {
6920   char *os;
6921
6922   os = s;
6923   while(n-- > 0 && (*s++ = *t++) != 0)
6924     ;
6925   while(n-- > 0)
6926     *s++ = 0;
6927   return os;
6928 }
6929
6930 // Like strncpy but guaranteed to NUL-terminate.
6931 char*
6932 safestrcpy(char *s, const char *t, int n)
6933 {
6934   char *os;
6935
6936   os = s;
6937   if(n <= 0)
6938     return os;
6939   while(--n > 0 && (*s++ = *t++) != 0)
6940     ;
6941   *s = 0;
6942   return os;
6943 }
6944
6945
6946
6947
6948
6949
```



```
6950 int
6951 strlen(const char *s)
6952 {
6953   int n;
6954
6955   for(n = 0; s[n]; n++)
6956     ;
6957   return n;
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 // See MultiProcessor Specification Version 1.[14]
7001
7002 struct mp {             // floating pointer
7003   uchar signature[4];           // "_MP_"
7004   void *physaddr;               // phys addr of MP config table
7005   uchar length;                 // 1
7006   uchar specrev;                // [14]
7007   uchar checksum;               // all bytes must add up to 0
7008   uchar type;                   // MP system config type
7009   uchar imcrp;
7010   uchar reserved[3];
7011 };
7012
7013 struct mpconf {         // configuration table header
7014   uchar signature[4];           // "PCMP"
7015   ushort length;                // total table length
7016   uchar version;                // [14]
7017   uchar checksum;               // all bytes must add up to 0
7018   uchar product[20];            // product id
7019   uint *oemtable;               // OEM table pointer
7020   ushort oemlength;             // OEM table length
7021   ushort entry;                 // entry count
7022   uint *lapicaddr;              // address of local APIC
7023   ushort xlength;               // extended table length
7024   uchar xchecksum;              // extended table checksum
7025   uchar reserved;
7026 };
7027
7028 struct mpproc {         // processor table entry
7029   uchar type;                   // entry type (0)
7030   uchar apicid;                 // local APIC id
7031   uchar version;                // local APIC verison
7032   uchar flags;                  // CPU flags
7033     #define MPBOOT 0x02           // This proc is the bootstrap processor.
7034   uchar signature[4];           // CPU signature
7035   uint feature;                 // feature flags from CPUID instruction
7036   uchar reserved[8];
7037 };
7038
7039 struct mpioapic {       // I/O APIC table entry
7040   uchar type;                   // entry type (2)
7041   uchar apicno;                 // I/O APIC id
7042   uchar version;                // I/O APIC version
7043   uchar flags;                  // I/O APIC flags
7044   uint *addr;                  // I/O APIC address
7045 };
7046
7047
7048
7049
```



```
7050 // Table entry types
7051 #define MPPROC    0x00  // One per processor
7052 #define MPBUS     0x01  // One per bus
7053 #define MPIOAPIC  0x02  // One per I/O APIC
7054 #define MPIOINTR  0x03  // One per bus interrupt source
7055 #define MPLINTR   0x04  // One per system interrupt source
7056
7057
7058
7059
7060
7061
7062
7063
7064
7065
7066
7067
7068
7069
7070
7071
7072
7073
7074
7075
7076
7077
7078
7079
7080
7081
7082
7083
7084
7085
7086
7087
7088
7089
7090
7091
7092
7093
7094
7095
7096
7097
7098
7099
```

```
7100 // Blank page.
7101
7102
7103
7104
7105
7106
7107
7108
7109
7110
7111
7112
7113
7114
7115
7116
7117
7118
7119
7120
7121
7122
7123
7124
7125
7126
7127
7128
7129
7130
7131
7132
7133
7134
7135
7136
7137
7138
7139
7140
7141
7142
7143
7144
7145
7146
7147
7148
7149
```



```
7150 // Multiprocessor support
7151 // Search memory for MP description structures.
7152 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
7153
7154 #include "types.h"
7155 #include "defs.h"
7156 #include "param.h"
7157 #include "memlayout.h"
7158 #include "mp.h"
7159 #include "x86.h"
7160 #include "mmu.h"
7161 #include "proc.h"
7162
7163 struct cpu cpus[NCPU];
7164 static struct cpu *bcpu;
7165 int ismp;
7166 int ncpu;
7167 uchar ioapicid;
7168
7169 int
7170 mpbcpu(void)
7171 {
7172   return bcpu-cpus;
7173 }
7174
7175 static uchar
7176 sum(uchar *addr, int len)
7177 {
7178   int i, sum;
7179
7180   sum = 0;
7181   for(i=0; i<len; i++)
7182     sum += addr[i];
7183   return sum;
7184 }
7185
7186 // Look for an MP structure in the len bytes at addr.
7187 static struct mp*
7188 mpsearch1(uint a, int len)
7189 {
7190   uchar *e, *p, *addr;
7191
7192   addr = p2v(a);
7193   e = addr+len;
7194   for(p = addr; p < e; p += sizeof(struct mp))
7195     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
7196       return (struct mp*)p;
7197   return 0;
7198 }
7199
```

```
7200 // Search for the MP Floating Pointer Structure, which according to the
7201 // spec is in one of the following three locations:
7202 // 1) in the first KB of the EBDA;
7203 // 2) in the last KB of system base memory;
7204 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
7205 static struct mp*
7206 mpsearch(void)
7207 {
7208   uchar *bda;
7209   uint p;
7210   struct mp *mp;
7211
7212   bda = (uchar *) P2V(0x400);
7213   if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
7214     if((mp = mpsearch1(p, 1024)))
7215       return mp;
7216   } else {
7217     p = ((bda[0x14]<<8)|bda[0x13])*1024;
7218     if((mp = mpsearch1(p-1024, 1024)))
7219       return mp;
7220   }
7221   return mpsearch1(0xF0000, 0x10000);
7222 }
7223
7224 // Search for an MP configuration table.  For now,
7225 // don't accept the default configurations (physaddr == 0).
7226 // Check for correct signature, calculate the checksum and,
7227 // if correct, check the version.
7228 // To do: check extended table checksum.
7229 static struct mpconf*
7230 mpconfig(struct mp **pmp)
7231 {
7232   struct mpconf *conf;
7233   struct mp *mp;
7234
7235   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
7236     return 0;
7237   conf = (struct mpconf*) p2v((uint) mp->physaddr);
7238   if(memcmp(conf, "PCMP", 4) != 0)
7239     return 0;
7240   if(conf->version != 1 && conf->version != 4)
7241     return 0;
7242   if(sum((uchar*)conf, conf->length) != 0)
7243     return 0;
7244   *pmp = mp;
7245   return conf;
7246 }
7247
7248
7249
```



```
7250 void
7251 mpinit(void)
7252 {
7253   uchar *p, *e;
7254   struct mp *mp;
7255   struct mpconf *conf;
7256   struct mpproc *proc;
7257   struct mpioapic *ioapic;
7258
7259   bcpu = &cpus[0];
7260   if((conf = mpconfig(&mp)) == 0)
7261     return;
7262   ismp = 1;
7263   lapic = (uint*)conf->lapicaddr;
7264   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
7265     switch(*p){
7266     case MPPROC:
7267       proc = (struct mpproc*)p;
7268       if(ncpu != proc->apicid){
7269         cprintf("mpinit: ncpu=%d apicid=%d\n", ncpu, proc->apicid);
7270         ismp = 0;
7271       }
7272       if(proc->flags & MPBOOT)
7273         bcpu = &cpus[ncpu];
7274       cpus[ncpu].id = ncpu;
7275       ncpu++;
7276       p += sizeof(struct mpproc);
7277       continue;
7278     case MPIOAPIC:
7279       ioapic = (struct mpioapic*)p;
7280       ioapicid = ioapic->apicno;
7281       p += sizeof(struct mpioapic);
7282       continue;
7283     case MPBUS:
7284     case MPIOINTR:
7285     case MPLINTR:
7286       p += 8;
7287       continue;
7288     default:
7289       cprintf("mpinit: unknown config type %x\n", *p);
7290       ismp = 0;
7291     }
7292   }
7293   if(!ismp){
7294     // Didn't like what we found; fall back to no MP.
7295     ncpu = 1;
7296     lapic = 0;
7297     ioapicid = 0;
7298     return;
7299   }
```

```
7300   if(mp->imcrp){
7301     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
7302     // But it would on real hardware.
7303     outb(0x22, 0x70);   // Select IMCR
7304     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
7305   }
7306 }
7307
7308
7309
7310
7311
7312
7313
7314
7315
7316
7317
7318
7319
7320
7321
7322
7323
7324
7325
7326
7327
7328
7329
7330
7331
7332
7333
7334
7335
7336
7337
7338
7339
7340
7341
7342
7343
7344
7345
7346
7347
7348
7349
```



```
7350 // The local APIC manages internal (non-I/O) interrupts.
7351 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
7352
7353 #include "types.h"
7354 #include "defs.h"
7355 #include "date.h"
7356 #include "memlayout.h"
7357 #include "traps.h"
7358 #include "mmu.h"
7359 #include "x86.h"
7360
7361 // Local APIC registers, divided by 4 for use as uint[] indices.
7362 #define ID      (0x0020/4)   // ID
7363 #define VER     (0x0030/4)   // Version
7364 #define TPR     (0x0080/4)   // Task Priority
7365 #define EOI     (0x00B0/4)   // EOI
7366 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
7367   #define ENABLE     0x00000100   // Unit Enable
7368 #define ESR     (0x0280/4)   // Error Status
7369 #define ICRLO   (0x0300/4)   // Interrupt Command
7370   #define INIT       0x00000500   // INIT/RESET
7371   #define STARTUP    0x00000600   // Startup IPI
7372   #define DELIVS     0x00001000   // Delivery status
7373   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
7374   #define DEASSERT   0x00000000
7375   #define LEVEL      0x00008000   // Level triggered
7376   #define BCAST      0x00080000   // Send to all APICs, including self.
7377   #define BUSY       0x00001000
7378   #define FIXED      0x00000000
7379 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
7380 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
7381   #define X1         0x0000000B   // divide counts by 1
7382   #define PERIODIC   0x00020000   // Periodic
7383 #define PCINT   (0x0340/4)   // Performance Counter LVT
7384 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
7385 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
7386 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
7387   #define MASKED     0x00010000   // Interrupt masked
7388 #define TICR    (0x0380/4)   // Timer Initial Count
7389 #define TCCR    (0x0390/4)   // Timer Current Count
7390 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
7391
7392 volatile uint *lapic;  // Initialized in mp.c
7393
7394 static void
7395 lapicw(int index, int value)
7396 {
7397   lapic[index] = value;
7398   lapic[ID];  // wait for write to finish, by reading
7399 }
```

```
7450 void
7451 lapicinit(void)
7452 {
7453   if(!lapic)
7454     return;
7455
7456   // Enable local APIC; set spurious interrupt vector.
7457   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
7458
7459   // The timer repeatedly counts down at bus frequency
7460   // from lapic[TICR] and then issues an interrupt.
7461   // If xv6 cared more about precise timekeeping,
7462   // TICR would be calibrated using an external time source.
7463   lapicw(TDCR, X1);
7464   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
7465   lapicw(TICR, 10000000);
7466
7467   // Disable logical interrupt lines.
7468   lapicw(LINT0, MASKED);
7469   lapicw(LINT1, MASKED);
7470
7471   // Disable performance counter overflow interrupts
7472   // on machines that provide that interrupt entry.
7473   if(((lapic[VER]>>16) & 0xFF) >= 4)
7474     lapicw(PCINT, MASKED);
7475
7476   // Map error interrupt to IRQ_ERROR.
7477   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
7478
7479   // Clear error status register (requires back-to-back writes).
7480   lapicw(ESR, 0);
7481   lapicw(ESR, 0);
7482
7483   // Ack any outstanding interrupts.
7484   lapicw(EOI, 0);
7485
7486   // Send an Init Level De-Assert to synchronise arbitration ID's.
7487   lapicw(ICRHI, 0);
7488   lapicw(ICRLO, BCAST | INIT | LEVEL);
7489   while(lapic[ICRLO] & DELIVS)
7490     ;
7491
7492   // Enable interrupts on the APIC (but not on the processor).
7493   lapicw(TPR, 0);
7494 }
7495
7496
7497
7498
7499
```

```
7500 int
7501 cpunum(void)
7502 {
7503   // Cannot call cpu when interrupts are enabled:
7504   // result not guaranteed to last long enough to be used!
7505   // Would prefer to panic but even printing is chancy here:
7506   // almost everything, including cprintf and panic, calls cpu,
7507   // often indirectly through acquire and release.
7508   if(readeflags()&FL_IF){
7509     static int n;
7510     if(n++ == 0)
7511       cprintf("cpu called from %x with interrupts enabled\n",
7512         __builtin_return_address(0));
7513   }
7514
7515   if(lapic)
7516     return lapic[ID]>>24;
7517   return 0;
7518 }
7519
7520 // Acknowledge interrupt.
7521 void
7522 lapiceoi(void)
7523 {
7524   if(lapic)
7525     lapicw(EOI, 0);
7526 }
7527
7528 // Spin for a given number of microseconds.
7529 // On real hardware would want to tune this dynamically.
7530 void
7531 microdelay(int us)
7532 {
7533 }
7534
7535 #define CMOS_PORT    0x70
7536 #define CMOS_RETURN  0x71
7537
7538 // Start additional processor running entry code at addr.
7539 // See Appendix B of MultiProcessor Specification.
7540 void
7541 lapicstartap(uchar apicid, uint addr)
7542 {
7543   int i;
7544   ushort *wrv;
7545
7546   // "The BSP must initialize CMOS shutdown code to 0AH
7547   // and the warm reset vector (DWORD based at 40:67) to point at
7548   // the AP startup code prior to the [universal startup algorithm]."
7549   outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
```



```
7550   outb(CMOS_PORT+1, 0x0A);
7551   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
7552   wrv[0] = 0;
7553   wrv[1] = addr >> 4;
7554
7555   // "Universal startup algorithm."
7556   // Send INIT (level-triggered) interrupt to reset other CPU.
7557   lapicw(ICRHI, apicid<<24);
7558   lapicw(ICRLO, INIT | LEVEL | ASSERT);
7559   microdelay(200);
7560   lapicw(ICRLO, INIT | LEVEL);
7561   microdelay(100);    // should be 10ms, but too slow in Bochs!
7562
7563   // Send startup IPI (twice!) to enter code.
7564   // Regular hardware is supposed to only accept a STARTUP
7565   // when it is in the halted state due to an INIT.  So the second
7566   // should be ignored, but it is part of the official Intel algorithm.
7567   // Bochs complains about the second one.  Too bad for Bochs.
7568   for(i = 0; i < 2; i++){
7569     lapicw(ICRHI, apicid<<24);
7570     lapicw(ICRLO, STARTUP | (addr>>12));
7571     microdelay(200);
7572   }
7573 }
7574
7575 #define CMOS_STATA   0x0a
7576 #define CMOS_STATB   0x0b
7577 #define CMOS_UIP    (1 << 7)        // RTC update in progress
7578
7579 #define SECS    0x00
7580 #define MINS    0x02
7581 #define HOURS   0x04
7582 #define DAY     0x07
7583 #define MONTH   0x08
7584 #define YEAR    0x09
7585
7586 static uint cmos_read(uint reg)
7587 {
7588   outb(CMOS_PORT,  reg);
7589   microdelay(200);
7590
7591   return inb(CMOS_RETURN);
7592 }
7593
7594
7595
7596
7597
7598
7599
```

```
7600 static void fill_rtcdate(struct rtcdate *r)
7601 {
7602   r->second = cmos_read(SECS);
7603   r->minute = cmos_read(MINS);
7604   r->hour   = cmos_read(HOURS);
7605   r->day    = cmos_read(DAY);
7606   r->month  = cmos_read(MONTH);
7607   r->year   = cmos_read(YEAR);
7608 }
7609
7610 // qemu seems to use 24-hour GWT and the values are BCD encoded
7611 void cmostime(struct rtcdate *r)
7612 {
7613   struct rtcdate t1, t2;
7614   int sb, bcd;
7615
7616   sb = cmos_read(CMOS_STATB);
7617
7618   bcd = (sb & (1 << 2)) == 0;
7619
7620   // make sure CMOS doesn't modify time while we read it
7621   for (;;) {
7622     fill_rtcdate(&t1);
7623     if (cmos_read(CMOS_STATA) & CMOS_UIP)
7624         continue;
7625     fill_rtcdate(&t2);
7626     if (memcmp(&t1, &t2, sizeof(t1)) == 0)
7627       break;
7628   }
7629
7630   // convert
7631   if (bcd) {
7632 #define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
7633     CONV(second);
7634     CONV(minute);
7635     CONV(hour  );
7636     CONV(day   );
7637     CONV(month );
7638     CONV(year  );
7639 #undef     CONV
7640   }
7641
7642   *r = t1;
7643   r->year += 2000;
7644 }
7645
7646
7647
7648
7649
```



```
7650 // The I/O APIC manages hardware interrupts for an SMP system.
7651 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
7652 // See also picirq.c.
7653
7654 #include "types.h"
7655 #include "defs.h"
7656 #include "traps.h"
7657
7658 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
7659
7660 #define REG_ID     0x00  // Register index: ID
7661 #define REG_VER    0x01  // Register index: version
7662 #define REG_TABLE  0x10  // Redirection table base
7663
7664 // The redirection table starts at REG_TABLE and uses
7665 // two registers to configure each interrupt.
7666 // The first (low) register in a pair contains configuration bits.
7667 // The second (high) register contains a bitmask telling which
7668 // CPUs can serve that interrupt.
7669 #define INT_DISABLED   0x00010000  // Interrupt disabled
7670 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
7671 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
7672 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
7673
7674 volatile struct ioapic *ioapic;
7675
7676 // IO APIC MMIO structure: write reg, then read or write data.
7677 struct ioapic {
7678   uint reg;
7679   uint pad[3];
7680   uint data;
7681 };
7682
7683 static uint
7684 ioapicread(int reg)
7685 {
7686   ioapic->reg = reg;
7687   return ioapic->data;
7688 }
7689
7690 static void
7691 ioapicwrite(int reg, uint data)
7692 {
7693   ioapic->reg = reg;
7694   ioapic->data = data;
7695 }
7696
7697
7698
7699
```

```
7700 void
7701 ioapicinit(void)
7702 {
7703   int i, id, maxintr;
7704
7705   if(!ismp)
7706     return;
7707
7708   ioapic = (volatile struct ioapic*)IOAPIC;
7709   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
7710   id = ioapicread(REG_ID) >> 24;
7711   if(id != ioapicid)
7712     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
7713
7714   // Mark all interrupts edge-triggered, active high, disabled,
7715   // and not routed to any CPUs.
7716   for(i = 0; i <= maxintr; i++){
7717     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
7718     ioapicwrite(REG_TABLE+2*i+1, 0);
7719   }
7720 }
7721
7722 void
7723 ioapicenable(int irq, int cpunum)
7724 {
7725   if(!ismp)
7726     return;
7727
7728   // Mark interrupt edge-triggered, active high,
7729   // enabled, and routed to the given cpunum,
7730   // which happens to be that cpu's APIC ID.
7731   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
7732   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
7733 }
7734
7735
7736
7737
7738
7739
7740
7741
7742
7743
7744
7745
7746
7747
7748
7749
```



```
7750 // Intel 8259A programmable interrupt controllers.
7751
7752 #include "types.h"
7753 #include "x86.h"
7754 #include "traps.h"
7755
7756 // I/O Addresses of the two programmable interrupt controllers
7757 #define IO_PIC1         0x20    // Master (IRQs 0-7)
7758 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
7759
7760 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
7761
7762 // Current IRQ mask.
7763 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
7764 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
7765
7766 static void
7767 picsetmask(ushort mask)
7768 {
7769   irqmask = mask;
7770   outb(IO_PIC1+1, mask);
7771   outb(IO_PIC2+1, mask >> 8);
7772 }
7773
7774 void
7775 picenable(int irq)
7776 {
7777   picsetmask(irqmask & ~(1<<irq));
7778 }
7779
7780 // Initialize the 8259A interrupt controllers.
7781 void
7782 picinit(void)
7783 {
7784   // mask all interrupts
7785   outb(IO_PIC1+1, 0xFF);
7786   outb(IO_PIC2+1, 0xFF);
7787
7788   // Set up master (8259A-1)
7789
7790   // ICW1:  0001g0hi
7791   //    g:  0 = edge triggering, 1 = level triggering
7792   //    h:  0 = cascaded PICs, 1 = master only
7793   //    i:  0 = no ICW4, 1 = ICW4 required
7794   outb(IO_PIC1, 0x11);
7795
7796   // ICW2:  Vector offset
7797   outb(IO_PIC1+1, T_IRQ0);
7798
7799
```

```
7800   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
7801   //        (slave PIC) 3-bit # of slave's connection to master
7802   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
7803
7804   // ICW4:  000nbmap
7805   //    n:  1 = special fully nested mode
7806   //    b:  1 = buffered mode
7807   //    m:  0 = slave PIC, 1 = master PIC
7808   //      (ignored when b is 0, as the master/slave role
7809   //      can be hardwired).
7810   //    a:  1 = Automatic EOI mode
7811   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
7812   outb(IO_PIC1+1, 0x3);
7813
7814   // Set up slave (8259A-2)
7815   outb(IO_PIC2, 0x11);                  // ICW1
7816   outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
7817   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
7818   // NB Automatic EOI mode doesn't tend to work on the slave.
7819   // Linux source code says it's "to be investigated".
7820   outb(IO_PIC2+1, 0x3);                 // ICW4
7821
7822   // OCW3:  0ef01prs
7823   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
7824   //    p:  0 = no polling, 1 = polling mode
7825   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
7826   outb(IO_PIC1, 0x68);             // clear specific mask
7827   outb(IO_PIC1, 0x0a);             // read IRR by default
7828
7829   outb(IO_PIC2, 0x68);             // OCW3
7830   outb(IO_PIC2, 0x0a);             // OCW3
7831
7832   if(irqmask != 0xFFFF)
7833     picsetmask(irqmask);
7834 }
7835
7836
7837
7838
7839
7840
7841
7842
7843
7844
7845
7846
7847
7848
7849
```



```
7850 // PC keyboard interface constants
7851
7852 #define KBSTATP         0x64    // kbd controller status port(I)
7853 #define KBS_DIB         0x01    // kbd data in buffer
7854 #define KBDATAP         0x60    // kbd data port(I)
7855
7856 #define NO              0
7857
7858 #define SHIFT           (1<<0)
7859 #define CTL             (1<<1)
7860 #define ALT             (1<<2)
7861
7862 #define CAPSLOCK        (1<<3)
7863 #define NUMLOCK         (1<<4)
7864 #define SCROLLLOCK      (1<<5)
7865
7866 #define E0ESC           (1<<6)
7867
7868 // Special keycodes
7869 #define KEY_HOME        0xE0
7870 #define KEY_END         0xE1
7871 #define KEY_UP          0xE2
7872 #define KEY_DN          0xE3
7873 #define KEY_LF          0xE4
7874 #define KEY_RT          0xE5
7875 #define KEY_PGUP        0xE6
7876 #define KEY_PGDN        0xE7
7877 #define KEY_INS         0xE8
7878 #define KEY_DEL         0xE9
7879
7880 // C('A') == Control-A
7881 #define C(x) (x - '@')
7882
7883 static uchar shiftcode[256] =
7884 {
7885   [0x1D] CTL,
7886   [0x2A] SHIFT,
7887   [0x36] SHIFT,
7888   [0x38] ALT,
7889   [0x9D] CTL,
7890   [0xB8] ALT
7891 };
7892
7893 static uchar togglecode[256] =
7894 {
7895   [0x3A] CAPSLOCK,
7896   [0x45] NUMLOCK,
7897   [0x46] SCROLLLOCK
7898 };
7899
```

```
7900 static uchar normalmap[256] =
7901 {
7902   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
7903   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
7904   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
7905   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
7906   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
7907   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
7908   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
7909   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7910   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7911   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7912   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7913   [0x9C] '\n',      // KP_Enter
7914   [0xB5] '/',       // KP_Div
7915   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7916   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7917   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7918   [0x97] KEY_HOME,  [0xCF] KEY_END,
7919   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7920 };
7921
7922 static uchar shiftmap[256] =
7923 {
7924   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
7925   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
7926   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
7927   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
7928   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
7929   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
7930   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
7931   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7932   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7933   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7934   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7935   [0x9C] '\n',      // KP_Enter
7936   [0xB5] '/',       // KP_Div
7937   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7938   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7939   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7940   [0x97] KEY_HOME,  [0xCF] KEY_END,
7941   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7942 };
7943
7944
7945
7946
7947
7948
7949
```



```
7950 static uchar ctlmap[256] =
7951 {
7952   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7953   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7954   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
7955   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
7956   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
7957   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
7958   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
7959   [0x9C] '\r',      // KP_Enter
7960   [0xB5] C('/'),    // KP_Div
7961   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7962   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7963   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7964   [0x97] KEY_HOME,  [0xCF] KEY_END,
7965   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7966 };
7967
7968
7969
7970
7971
7972
7973
7974
7975
7976
7977
7978
7979
7980
7981
7982
7983
7984
7985
7986
7987
7988
7989
7990
7991
7992
7993
7994
7995
7996
7997
7998
7999
```

```
8000 #include "types.h"
8001 #include "x86.h"
8002 #include "defs.h"
8003 #include "kbd.h"
8004
8005 int
8006 kbdgetc(void)
8007 {
8008   static uint shift;
8009   static uchar *charcode[4] = {
8010     normalmap, shiftmap, ctlmap, ctlmap
8011   };
8012   uint st, data, c;
8013
8014   st = inb(KBSTATP);
8015   if((st & KBS_DIB) == 0)
8016     return -1;
8017   data = inb(KBDATAP);
8018
8019   if(data == 0xE0){
8020     shift |= E0ESC;
8021     return 0;
8022   } else if(data & 0x80){
8023     // Key released
8024     data = (shift & E0ESC ? data : data & 0x7F);
8025     shift &= ~(shiftcode[data] | E0ESC);
8026     return 0;
8027   } else if(shift & E0ESC){
8028     // Last character was an E0 escape; or with 0x80
8029     data |= 0x80;
8030     shift &= ~E0ESC;
8031   }
8032
8033   shift |= shiftcode[data];
8034   shift ^= togglecode[data];
8035   c = charcode[shift & (CTL | SHIFT)][data];
8036   if(shift & CAPSLOCK){
8037     if('a' <= c && c <= 'z')
8038       c += 'A' - 'a';
8039     else if('A' <= c && c <= 'Z')
8040       c += 'a' - 'A';
8041   }
8042   return c;
8043 }
8044
8045 void
8046 kbdintr(void)
8047 {
8048   consoleintr(kbdgetc);
8049 }
```



```
8050 // Console input and output.
8051 // Input is from the keyboard or serial port.
8052 // Output is written to the screen and serial port.
8053
8054 #include "types.h"
8055 #include "defs.h"
8056 #include "param.h"
8057 #include "traps.h"
8058 #include "spinlock.h"
8059 #include "fs.h"
8060 #include "file.h"
8061 #include "memlayout.h"
8062 #include "mmu.h"
8063 #include "proc.h"
8064 #include "x86.h"
8065
8066 static void consputc(int);
8067
8068 static int panicked = 0;
8069
8070 static struct {
8071   struct spinlock lock;
8072   int locking;
8073 } cons;
8074
8075 static void
8076 printint(int xx, int base, int sign)
8077 {
8078   static char digits[] = "0123456789abcdef";
8079   char buf[16];
8080   int i;
8081   uint x;
8082
8083   if(sign && (sign = xx < 0))
8084     x = -xx;
8085   else
8086     x = xx;
8087
8088   i = 0;
8089   do{
8090     buf[i++] = digits[x % base];
8091   }while((x /= base) != 0);
8092
8093   if(sign)
8094     buf[i++] = '-';
8095
8096   while(--i >= 0)
8097     consputc(buf[i]);
8098 }
8099
```

```
8100 // Print to the console. only understands %d, %x, %p, %s.
8101 void
8102 cprintf(char *fmt, ...)
8103 {
8104   int i, c, locking;
8105   uint *argp;
8106   char *s;
8107
8108   locking = cons.locking;
8109   if(locking)
8110     acquire(&cons.lock);
8111
8112   if (fmt == 0)
8113     panic("null fmt");
8114
8115   argp = (uint*)(void*)(&fmt + 1);
8116   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
8117     if(c != '%'){
8118       consputc(c);
8119       continue;
8120     }
8121     c = fmt[++i] & 0xff;
8122     if(c == 0)
8123       break;
8124     switch(c){
8125     case 'd':
8126       printint(*argp++, 10, 1);
8127       break;
8128     case 'x':
8129     case 'p':
8130       printint(*argp++, 16, 0);
8131       break;
8132     case 's':
8133       if((s = (char*)*argp++) == 0)
8134         s = "(null)";
8135       for(; *s; s++)
8136         consputc(*s);
8137       break;
8138     case '%':
8139       consputc('%');
8140       break;
8141     default:
8142       // Print unknown % sequence to draw attention.
8143       consputc('%');
8144       consputc(c);
8145       break;
8146     }
8147   }
8148
8149
```



```
8150   if(locking)
8151     release(&cons.lock);
8152 }
8153
8154 void
8155 panic(char *s)
8156 {
8157   int i;
8158   uint pcs[10];
8159
8160   cli();
8161   cons.locking = 0;
8162   cprintf("cpu%d: panic: ", cpu->id);
8163   cprintf(s);
8164   cprintf("\n");
8165   getcallerpcs(&s, pcs);
8166   for(i=0; i<10; i++)
8167     cprintf(" %p", pcs[i]);
8168   panicked = 1; // freeze other CPU
8169   for(;;)
8170     ;
8171 }
8172
8173
8174
8175
8176
8177
8178
8179
8180
8181
8182
8183
8184
8185
8186
8187
8188
8189
8190
8191
8192
8193
8194
8195
8196
8197
8198
8199
```

```
8200 #define BACKSPACE 0x100
8201 #define CRTPORT 0x3d4
8202 static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory
8203
8204 static void
8205 cgaputc(int c)
8206 {
8207   int pos;
8208
8209   // Cursor position: col + 80*row.
8210   outb(CRTPORT, 14);
8211   pos = inb(CRTPORT+1) << 8;
8212   outb(CRTPORT, 15);
8213   pos |= inb(CRTPORT+1);
8214
8215   if(c == '\n')
8216     pos += 80 - pos%80;
8217   else if(c == BACKSPACE){
8218     if(pos > 0) --pos;
8219   } else
8220     crt[pos++] = (c&0xff) | 0x0700;  // black on white
8221
8222   if(pos < 0 || pos > 25*80)
8223     panic("pos under/overflow");
8224
8225   if((pos/80) >= 24){  // Scroll up.
8226     memmove(crt, crt+80, sizeof(crt[0])*23*80);
8227     pos -= 80;
8228     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
8229   }
8230
8231   outb(CRTPORT, 14);
8232   outb(CRTPORT+1, pos>>8);
8233   outb(CRTPORT, 15);
8234   outb(CRTPORT+1, pos);
8235   crt[pos] = ' ' | 0x0700;
8236 }
8237
8238
8239
8240
8241
8242
8243
8244
8245
8246
8247
8248
8249
```



```
8250 void
8251 consputc(int c)
8252 {
8253   if(panicked){
8254     cli();
8255     for(;;)
8256       ;
8257   }
8258
8259   if(c == BACKSPACE){
8260     uartputc('\b'); uartputc(' '); uartputc('\b');
8261   } else
8262     uartputc(c);
8263   cgaputc(c);
8264 }
8265
8266 #define INPUT_BUF 128
8267 struct {
8268   char buf[INPUT_BUF];
8269   uint r;  // Read index
8270   uint w;  // Write index
8271   uint e;  // Edit index
8272 } input;
8273
8274 #define C(x)  ((x)-'@')  // Control-x
8275
8276 void
8277 consoleintr(int (*getc)(void))
8278 {
8279   int c, doprocdump = 0;
8280
8281   acquire(&cons.lock);
8282   while((c = getc()) >= 0){
8283     switch(c){
8284     case C('P'):  // Process listing.
8285       doprocdump = 1;   // procdump() locks cons.lock indirectly; invoke lat
8286       break;
8287     case C('U'):  // Kill line.
8288       while(input.e != input.w &&
8289             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
8290         input.e--;
8291         consputc(BACKSPACE);
8292       }
8293       break;
8294     case C('H'): case '\x7f':  // Backspace
8295       if(input.e != input.w){
8296         input.e--;
8297         consputc(BACKSPACE);
8298       }
8299       break;
```

```
8300     default:
8301       if(c != 0 && input.e-input.r < INPUT_BUF){
8302         c = (c == '\r') ? '\n' : c;
8303         input.buf[input.e++ % INPUT_BUF] = c;
8304         consputc(c);
8305         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
8306           input.w = input.e;
8307           wakeup(&input.r);
8308         }
8309       }
8310       break;
8311     }
8312   }
8313   release(&cons.lock);
8314   if(doprocdump) {
8315     procdump();  // now call procdump() wo. cons.lock held
8316   }
8317 }
8318
8319 int
8320 consoleread(struct inode *ip, char *dst, int n)
8321 {
8322   uint target;
8323   int c;
8324
8325   iunlock(ip);
8326   target = n;
8327   acquire(&cons.lock);
8328   while(n > 0){
8329     while(input.r == input.w){
8330       if(proc->killed){
8331         release(&cons.lock);
8332         ilock(ip);
8333         return -1;
8334       }
8335       sleep(&input.r, &cons.lock);
8336     }
8337     c = input.buf[input.r++ % INPUT_BUF];
8338     if(c == C('D')){  // EOF
8339       if(n < target){
8340         // Save ^D for next time, to make sure
8341         // caller gets a 0-byte result.
8342         input.r--;
8343       }
8344       break;
8345     }
8346     *dst++ = c;
8347     --n;
8348     if(c == '\n')
8349       break;
```



```
8350   }
8351   release(&cons.lock);
8352   ilock(ip);
8353
8354   return target - n;
8355 }
8356
8357 int
8358 consolewrite(struct inode *ip, char *buf, int n)
8359 {
8360   int i;
8361
8362   iunlock(ip);
8363   acquire(&cons.lock);
8364   for(i = 0; i < n; i++)
8365     consputc(buf[i] & 0xff);
8366   release(&cons.lock);
8367   ilock(ip);
8368
8369   return n;
8370 }
8371
8372 void
8373 consoleinit(void)
8374 {
8375   initlock(&cons.lock, "console");
8376
8377   devsw[CONSOLE].write = consolewrite;
8378   devsw[CONSOLE].read = consoleread;
8379   cons.locking = 1;
8380
8381   picenable(IRQ_KBD);
8382   ioapicenable(IRQ_KBD, 0);
8383 }
8384
8385
8386
8387
8388
8389
8390
8391
8392
8393
8394
8395
8396
8397
8398
8399
```

```
8400 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
8401 // Only used on uniprocessors;
8402 // SMP machines use the local APIC timer.
8403
8404 #include "types.h"
8405 #include "defs.h"
8406 #include "traps.h"
8407 #include "x86.h"
8408
8409 #define IO_TIMER1       0x040           // 8253 Timer #1
8410
8411 // Frequency of all three count-down timers;
8412 // (TIMER_FREQ/freq) is the appropriate count
8413 // to generate a frequency of freq Hz.
8414
8415 #define TIMER_FREQ      1193182
8416 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
8417
8418 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
8419 #define TIMER_SEL0      0x00    // select counter 0
8420 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
8421 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
8422
8423 void
8424 timerinit(void)
8425 {
8426   // Interrupt 100 times/sec.
8427   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
8428   outb(IO_TIMER1, TIMER_DIV(100) % 256);
8429   outb(IO_TIMER1, TIMER_DIV(100) / 256);
8430   picenable(IRQ_TIMER);
8431 }
8432
8433
8434
8435
8436
8437
8438
8439
8440
8441
8442
8443
8444
8445
8446
8447
8448
8449
```



```
8450 // Intel 8250 serial port (UART).
8451
8452 #include "types.h"
8453 #include "defs.h"
8454 #include "param.h"
8455 #include "traps.h"
8456 #include "spinlock.h"
8457 #include "fs.h"
8458 #include "file.h"
8459 #include "mmu.h"
8460 #include "proc.h"
8461 #include "x86.h"
8462
8463 #define COM1    0x3f8
8464
8465 static int uart;    // is there a uart?
8466
8467 void
8468 uartinit(void)
8469 {
8470   char *p;
8471
8472   // Turn off the FIFO
8473   outb(COM1+2, 0);
8474
8475   // 9600 baud, 8 data bits, 1 stop bit, parity off.
8476   outb(COM1+3, 0x80);    // Unlock divisor
8477   outb(COM1+0, 115200/9600);
8478   outb(COM1+1, 0);
8479   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
8480   outb(COM1+4, 0);
8481   outb(COM1+1, 0x01);    // Enable receive interrupts.
8482
8483   // If status is 0xFF, no serial port.
8484   if(inb(COM1+5) == 0xFF)
8485     return;
8486   uart = 1;
8487
8488   // Acknowledge pre-existing interrupt conditions;
8489   // enable interrupts.
8490   inb(COM1+2);
8491   inb(COM1+0);
8492   picenable(IRQ_COM1);
8493   ioapicenable(IRQ_COM1, 0);
8494
8495   // Announce that we're here.
8496   for(p="xv6...\n"; *p; p++)
8497     uartputc(*p);
8498 }
8499
```

```
8500 void
8501 uartputc(int c)
8502 {
8503   int i;
8504
8505   if(!uart)
8506     return;
8507   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
8508     microdelay(10);
8509   outb(COM1+0, c);
8510 }
8511
8512 static int
8513 uartgetc(void)
8514 {
8515   if(!uart)
8516     return -1;
8517   if(!(inb(COM1+5) & 0x01))
8518     return -1;
8519   return inb(COM1+0);
8520 }
8521
8522 void
8523 uartintr(void)
8524 {
8525   consoleintr(uartgetc);
8526 }
8527
8528
8529
8530
8531
8532
8533
8534
8535
8536
8537
8538
8539
8540
8541
8542
8543
8544
8545
8546
8547
8548
8549
```



```
8550 # Initial process execs /init.
8551
8552 #include "syscall.h"
8553 #include "traps.h"
8554
8555
8556 # exec(init, argv)
8557 .globl start
8558 start:
8559   pushl $argv
8560   pushl $init
8561   pushl $0  // where caller pc would be
8562   movl $SYS_exec, %eax
8563   int $T_SYSCALL
8564
8565 # for(;;) exit();
8566 exit:
8567   movl $SYS_exit, %eax
8568   int $T_SYSCALL
8569   jmp exit
8570
8571 # char init[] = "/init\0";
8572 init:
8573   .string "/init\0"
8574
8575 # char *argv[] = { init, 0 };
8576 .p2align 2
8577 argv:
8578   .long init
8579   .long 0
8580
8581
8582
8583
8584
8585
8586
8587
8588
8589
8590
8591
8592
8593
8594
8595
8596
8597
8598
8599
```

```
8600 #include "syscall.h"
8601 #include "traps.h"
8602
8603 #define SYSCALL(name) \
8604   .globl name; \
8605   name: \
8606     movl $SYS_ ## name, %eax; \
8607     int $T_SYSCALL; \
8608     ret
8609
8610 SYSCALL(fork)
8611 SYSCALL(exit)
8612 SYSCALL(wait)
8613 SYSCALL(pipe)
8614 SYSCALL(read)
8615 SYSCALL(write)
8616 SYSCALL(close)
8617 SYSCALL(kill)
8618 SYSCALL(exec)
8619 SYSCALL(open)
8620 SYSCALL(mknod)
8621 SYSCALL(unlink)
8622 SYSCALL(fstat)
8623 SYSCALL(link)
8624 SYSCALL(mkdir)
8625 SYSCALL(chdir)
8626 SYSCALL(dup)
8627 SYSCALL(getpid)
8628 SYSCALL(sbrk)
8629 SYSCALL(sleep)
8630 SYSCALL(uptime)
8631 SYSCALL(halt)
8632 SYSCALL(date)
8633
8634 SYSCALL(getuid)
8635 SYSCALL(getgid)
8636 SYSCALL(getppid)
8637 SYSCALL(setuid)
8638 SYSCALL(setgid)
8639 SYSCALL(getprocs)
8640
8641
8642
8643
8644
8645
8646
8647
8648
8649
```



```
8650 // init: The initial user-level program
8651
8652 #include "types.h"
8653 #include "stat.h"
8654 #include "user.h"
8655 #include "fcntl.h"
8656
8657 char *argv[] = { "sh", 0 };
8658
8659 int
8660 main(void)
8661 {
8662   int pid, wpid;
8663
8664   if(open("console", O_RDWR) < 0){
8665     mknod("console", 1, 1);
8666     open("console", O_RDWR);
8667   }
8668   dup(0);  // stdout
8669   dup(0);  // stderr
8670
8671   for(;;){
8672     printf(1, "init: starting sh\n");
8673     pid = fork();
8674     if(pid < 0){
8675       printf(1, "init: fork failed\n");
8676       exit();
8677     }
8678     if(pid == 0){
8679       exec("sh", argv);
8680       printf(1, "init: exec sh failed\n");
8681       exit();
8682     }
8683     while((wpid=wait()) >= 0 && wpid != pid)
8684       printf(1, "zombie!\n");
8685   }
8686 }
8687
8688
8689
8690
8691
8692
8693
8694
8695
8696
8697
8698
8699
```

```
8700 // Shell.
8701 // 2015-12-21. Added very simple processing for builtin commands
8702
8703 #include "types.h"
8704 #include "user.h"
8705 #include "fcntl.h"
8706
8707 // Parsed command representation
8708 #define EXEC  1
8709 #define REDIR 2
8710 #define PIPE  3
8711 #define LIST  4
8712 #define BACK  5
8713
8714 #define MAXARGS 10
8715
8716 struct cmd {
8717   int type;
8718 };
8719
8720 struct execcmd {
8721   int type;
8722   char *argv[MAXARGS];
8723   char *eargv[MAXARGS];
8724 };
8725
8726 struct redircmd {
8727   int type;
8728   struct cmd *cmd;
8729   char *file;
8730   char *efile;
8731   int mode;
8732   int fd;
8733 };
8734
8735 struct pipecmd {
8736   int type;
8737   struct cmd *left;
8738   struct cmd *right;
8739 };
8740
8741 struct listcmd {
8742   int type;
8743   struct cmd *left;
8744   struct cmd *right;
8745 };
8746
8747
8748
8749
```



```
8750 struct backcmd {
8751   int type;
8752   struct cmd *cmd;
8753 };
8754
8755 int fork1(void);  // Fork but panics on failure.
8756 void panic(char*);
8757 struct cmd *parsecmd(char*);
8758
8759 // Execute cmd.  Never returns.
8760 void
8761 runcmd(struct cmd *cmd)
8762 {
8763   int p[2];
8764   struct backcmd *bcmd;
8765   struct execcmd *ecmd;
8766   struct listcmd *lcmd;
8767   struct pipecmd *pcmd;
8768   struct redircmd *rcmd;
8769
8770   if(cmd == 0)
8771     exit();
8772
8773   switch(cmd->type){
8774   default:
8775     panic("runcmd");
8776
8777   case EXEC:
8778     ecmd = (struct execcmd*)cmd;
8779     if(ecmd->argv[0] == 0)
8780       exit();
8781     exec(ecmd->argv[0], ecmd->argv);
8782     printf(2, "exec %s failed\n", ecmd->argv[0]);
8783     break;
8784
8785   case REDIR:
8786     rcmd = (struct redircmd*)cmd;
8787     close(rcmd->fd);
8788     if(open(rcmd->file, rcmd->mode) < 0){
8789       printf(2, "open %s failed\n", rcmd->file);
8790       exit();
8791     }
8792     runcmd(rcmd->cmd);
8793     break;
8794
8795   case LIST:
8796     lcmd = (struct listcmd*)cmd;
8797     if(fork1() == 0)
8798       runcmd(lcmd->left);
8799     wait();
```

```
8800     runcmd(lcmd->right);
8801     break;
8802
8803   case PIPE:
8804     pcmd = (struct pipecmd*)cmd;
8805     if(pipe(p) < 0)
8806       panic("pipe");
8807     if(fork1() == 0){
8808       close(1);
8809       dup(p[1]);
8810       close(p[0]);
8811       close(p[1]);
8812       runcmd(pcmd->left);
8813     }
8814     if(fork1() == 0){
8815       close(0);
8816       dup(p[0]);
8817       close(p[0]);
8818       close(p[1]);
8819       runcmd(pcmd->right);
8820     }
8821     close(p[0]);
8822     close(p[1]);
8823     wait();
8824     wait();
8825     break;
8826
8827   case BACK:
8828     bcmd = (struct backcmd*)cmd;
8829     if(fork1() == 0)
8830       runcmd(bcmd->cmd);
8831     break;
8832   }
8833   exit();
8834 }
8835
8836 int
8837 getcmd(char *buf, int nbuf)
8838 {
8839   printf(2, "$ ");
8840   memset(buf, 0, nbuf);
8841   gets(buf, nbuf);
8842   if(buf[0] == 0) // EOF
8843     return -1;
8844   return 0;
8845 }
8846
8847
8848
8849
```



```
8850 // ***** processing for shell builtins begins here *****
8851
8852 int
8853 strncmp(const char *p, const char *q, uint n)
8854 {
8855     while(n > 0 && *p && *p == *q)
8856       n--, p++, q++;
8857     if(n == 0)
8858       return 0;
8859     return (uchar)*p - (uchar)*q;
8860 }
8861
8862 int
8863 makeint(char *p)
8864 {
8865   int val = 0;
8866
8867   while ((*p >= '0') && (*p <= '9')) {
8868     val = 10*val + (*p-'0');
8869     ++p;
8870   }
8871   return val;
8872 }
8873
8874 int
8875 setbuiltin(char *p)
8876 {
8877   int i;
8878
8879   p += strlen("_set");
8880   while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8881   if (strncmp("uid", p, 3) == 0) {
8882     p += strlen("uid");
8883     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8884     i = makeint(p); // ugly
8885     return (setuid(i));
8886   } else
8887   if (strncmp("gid", p, 3) == 0) {
8888     p += strlen("gid");
8889     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8890     i = makeint(p); // ugly
8891     return (setgid(i));
8892   }
8893   printf(2, "Invalid _set parameter\n");
8894   return -1;
8895 }
8896
8897
8898
8899
```

```
8900 int
8901 getbuiltin(char *p)
8902 {
8903   p += strlen("_get");
8904   while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
8905   if (strncmp("uid", p, 3) == 0) {
8906     printf(2, "%d\n", getuid());
8907     return 0;
8908   }
8909   if (strncmp("gid", p, 3) == 0) {
8910     printf(2, "%d\n", getgid());
8911     return 0;
8912   }
8913   printf(2, "Invalid _get parameter\n");
8914   return -1;
8915 }
8916
8917 typedef int funcPtr_t(char *);
8918 typedef struct {
8919   char       *cmd;
8920   funcPtr_t  *name;
8921 } dispatchTableEntry_t;
8922
8923 // Use a simple function dispatch table (FDT) to process builtin commands
8924 dispatchTableEntry_t fdt[] = {
8925   {"_set", setbuiltin},
8926   {"_get", getbuiltin}
8927 };
8928 int FDTcount = sizeof(fdt) / sizeof(fdt[0]); // # entris in FDT
8929
8930 void
8931 dobuiltin(char *cmd) {
8932   int i;
8933
8934   for (i=0; i<FDTcount; i++)
8935     if (strncmp(cmd, fdt[i].cmd, strlen(fdt[i].cmd)) == 0)
8936      (*fdt[i].name)(cmd);
8937 }
8938
8939
8940
8941
8942
8943
8944
8945
8946
8947
8948
8949
```



```
8950 // ***** processing for shell builtins ends here *****
8951
8952 int
8953 main(void)
8954 {
8955   static char buf[100];
8956   int fd;
8957
8958   // Assumes three file descriptors open.
8959   while((fd = open("console", O_RDWR)) >= 0){
8960     if(fd >= 3){
8961       close(fd);
8962       break;
8963     }
8964   }
8965
8966   // Read and run input commands.
8967   while(getcmd(buf, sizeof(buf)) >= 0){
8968 // add support for built-ins here. cd is a built-in
8969     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
8970       // Clumsy but will have to do for now.
8971       // Chdir has no effect on the parent if run in the child.
8972       buf[strlen(buf)-1] = 0;  // chop \n
8973       if(chdir(buf+3) < 0)
8974         printf(2, "cannot cd %s\n", buf+3);
8975       continue;
8976     }
8977     if (buf[0]=='_') {     // assume it is a builtin command
8978       dobuiltin(buf);
8979       continue;
8980     }
8981     if(fork1() == 0)
8982       runcmd(parsecmd(buf));
8983     wait();
8984   }
8985   exit();
8986 }
8987
8988 void
8989 panic(char *s)
8990 {
8991   printf(2, "%s\n", s);
8992   exit();
8993 }
8994
8995
8996
8997
8998
8999
```

```
int
fork1(void)
{
  int pid;

  pid = fork();
  if(pid == -1)
    panic("fork");
  return pid;
}
```



```
// Constructors

struct cmd*
execcmd(void)
{
  struct execcmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = EXEC;
  return (struct cmd*)cmd;
}

struct cmd*
redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
{
  struct redircmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = REDIR;
  cmd->cmd = subcmd;
  cmd->file = file;
  cmd->efile = efile;
  cmd->mode = mode;
  cmd->fd = fd;
  return (struct cmd*)cmd;
}

struct cmd*
pipecmd(struct cmd *left, struct cmd *right)
{
  struct pipecmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = PIPE;
  cmd->left = left;
  cmd->right = right;
  return (struct cmd*)cmd;
}
```

```
9100 struct cmd*
9101 listcmd(struct cmd *left, struct cmd *right)
9102 {
9103   struct listcmd *cmd;
9104
9105   cmd = malloc(sizeof(*cmd));
9106   memset(cmd, 0, sizeof(*cmd));
9107   cmd->type = LIST;
9108   cmd->left = left;
9109   cmd->right = right;
9110   return (struct cmd*)cmd;
9111 }
9112
9113 struct cmd*
9114 backcmd(struct cmd *subcmd)
9115 {
9116   struct backcmd *cmd;
9117
9118   cmd = malloc(sizeof(*cmd));
9119   memset(cmd, 0, sizeof(*cmd));
9120   cmd->type = BACK;
9121   cmd->cmd = subcmd;
9122   return (struct cmd*)cmd;
9123 }
9124
9125
9126
9127
9128
9129
9130
9131
9132
9133
9134
9135
9136
9137
9138
9139
9140
9141
9142
9143
9144
9145
9146
9147
9148
9149
```



```
9150 // Parsing
9151
9152 char whitespace[] = " \t\r\n\v";
9153 char symbols[] = "<|>&;()";
9154
9155 int
9156 gettoken(char **ps, char *es, char **q, char **eq)
9157 {
9158   char *s;
9159   int ret;
9160
9161   s = *ps;
9162   while(s < es && strchr(whitespace, *s))
9163     s++;
9164   if(q)
9165     *q = s;
9166   ret = *s;
9167   switch(*s){
9168   case 0:
9169     break;
9170   case '|':
9171   case '(':
9172   case ')':
9173   case ';':
9174   case '&':
9175   case '<':
9176     s++;
9177     break;
9178   case '>':
9179     s++;
9180     if(*s == '>'){
9181       ret = '+';
9182       s++;
9183     }
9184     break;
9185   default:
9186     ret = 'a';
9187     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
9188       s++;
9189     break;
9190   }
9191   if(eq)
9192     *eq = s;
9193
9194   while(s < es && strchr(whitespace, *s))
9195     s++;
9196   *ps = s;
9197   return ret;
9198 }
9199
```

```
9200 int
9201 peek(char **ps, char *es, char *toks)
9202 {
9203   char *s;
9204
9205   s = *ps;
9206   while(s < es && strchr(whitespace, *s))
9207     s++;
9208   *ps = s;
9209   return *s && strchr(toks, *s);
9210 }
9211
9212 struct cmd *parseline(char**, char*);
9213 struct cmd *parsepipe(char**, char*);
9214 struct cmd *parseexec(char**, char*);
9215 struct cmd *nulterminate(struct cmd*);
9216
9217 struct cmd*
9218 parsecmd(char *s)
9219 {
9220   char *es;
9221   struct cmd *cmd;
9222
9223   es = s + strlen(s);
9224   cmd = parseline(&s, es);
9225   peek(&s, es, "");
9226   if(s != es){
9227     printf(2, "leftovers: %s\n", s);
9228     panic("syntax");
9229   }
9230   nulterminate(cmd);
9231   return cmd;
9232 }
9233
9234 struct cmd*
9235 parseline(char **ps, char *es)
9236 {
9237   struct cmd *cmd;
9238
9239   cmd = parsepipe(ps, es);
9240   while(peek(ps, es, "&")){
9241     gettoken(ps, es, 0, 0);
9242     cmd = backcmd(cmd);
9243   }
9244   if(peek(ps, es, ";")){
9245     gettoken(ps, es, 0, 0);
9246     cmd = listcmd(cmd, parseline(ps, es));
9247   }
9248   return cmd;
9249 }
```



```
9250 struct cmd*
9251 parsepipe(char **ps, char *es)
9252 {
9253   struct cmd *cmd;
9254
9255   cmd = parseexec(ps, es);
9256   if(peek(ps, es, "|")){
9257     gettoken(ps, es, 0, 0);
9258     cmd = pipecmd(cmd, parsepipe(ps, es));
9259   }
9260   return cmd;
9261 }
9262
9263 struct cmd*
9264 parseredirs(struct cmd *cmd, char **ps, char *es)
9265 {
9266   int tok;
9267   char *q, *eq;
9268
9269   while(peek(ps, es, "<>")){
9270     tok = gettoken(ps, es, 0, 0);
9271     if(gettoken(ps, es, &q, &eq) != 'a')
9272       panic("missing file for redirection");
9273     switch(tok){
9274     case '<':
9275       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
9276       break;
9277     case '>':
9278       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9279       break;
9280     case '+':  // >>
9281       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9282       break;
9283     }
9284   }
9285   return cmd;
9286 }
9287
9288
9289
9290
9291
9292
9293
9294
9295
9296
9297
9298
9299
```

```
9300 struct cmd*
9301 parseblock(char **ps, char *es)
9302 {
9303   struct cmd *cmd;
9304
9305   if(!peek(ps, es, "("))
9306     panic("parseblock");
9307   gettoken(ps, es, 0, 0);
9308   cmd = parseline(ps, es);
9309   if(!peek(ps, es, ")"))
9310     panic("syntax - missing )");
9311   gettoken(ps, es, 0, 0);
9312   cmd = parseredirs(cmd, ps, es);
9313   return cmd;
9314 }
9315
9316 struct cmd*
9317 parseexec(char **ps, char *es)
9318 {
9319   char *q, *eq;
9320   int tok, argc;
9321   struct execcmd *cmd;
9322   struct cmd *ret;
9323
9324   if(peek(ps, es, "("))
9325     return parseblock(ps, es);
9326
9327   ret = execcmd();
9328   cmd = (struct execcmd*)ret;
9329
9330   argc = 0;
9331   ret = parseredirs(ret, ps, es);
9332   while(!peek(ps, es, "|)&;")){
9333     if((tok=gettoken(ps, es, &q, &eq)) == 0)
9334       break;
9335     if(tok != 'a')
9336       panic("syntax");
9337     cmd->argv[argc] = q;
9338     cmd->eargv[argc] = eq;
9339     argc++;
9340     if(argc >= MAXARGS)
9341       panic("too many args");
9342     ret = parseredirs(ret, ps, es);
9343   }
9344   cmd->argv[argc] = 0;
9345   cmd->eargv[argc] = 0;
9346   return ret;
9347 }
9348
9349
```



```
9350 // NUL-terminate all the counted strings.
9351 struct cmd*
9352 nulterminate(struct cmd *cmd)
9353 {
9354   int i;
9355   struct backcmd *bcmd;
9356   struct execcmd *ecmd;
9357   struct listcmd *lcmd;
9358   struct pipecmd *pcmd;
9359   struct redircmd *rcmd;
9360
9361   if(cmd == 0)
9362     return 0;
9363
9364   switch(cmd->type){
9365   case EXEC:
9366     ecmd = (struct execcmd*)cmd;
9367     for(i=0; ecmd->argv[i]; i++)
9368       *ecmd->eargv[i] = 0;
9369     break;
9370
9371   case REDIR:
9372     rcmd = (struct redircmd*)cmd;
9373     nulterminate(rcmd->cmd);
9374     *rcmd->efile = 0;
9375     break;
9376
9377   case PIPE:
9378     pcmd = (struct pipecmd*)cmd;
9379     nulterminate(pcmd->left);
9380     nulterminate(pcmd->right);
9381     break;
9382
9383   case LIST:
9384     lcmd = (struct listcmd*)cmd;
9385     nulterminate(lcmd->left);
9386     nulterminate(lcmd->right);
9387     break;
9388
9389   case BACK:
9390     bcmd = (struct backcmd*)cmd;
9391     nulterminate(bcmd->cmd);
9392     break;
9393   }
9394   return cmd;
9395 }
9396
9397
9398
9399
```

```
9400 #include "asm.h"
9401 #include "memlayout.h"
9402 #include "mmu.h"
9403
9404 # Start the first CPU: switch to 32-bit protected mode, jump into C.
9405 # The BIOS loads this code from the first sector of the hard disk into
9406 # memory at physical address 0x7c00 and starts executing in real mode
9407 # with %cs=0 %ip=7c00.
9408
9409 .code16                       # Assemble for 16-bit mode
9410 .globl start
9411 start:
9412   cli                         # BIOS enabled interrupts; disable
9413
9414   # Zero data segment registers DS, ES, and SS.
9415   xorw    %ax,%ax             # Set %ax to zero
9416   movw    %ax,%ds             # -> Data Segment
9417   movw    %ax,%es             # -> Extra Segment
9418   movw    %ax,%ss             # -> Stack Segment
9419
9420   # Physical address line A20 is tied to zero so that the first PCs
9421   # with 2 MB would run software that assumed 1 MB.  Undo that.
9422 seta20.1:
9423   inb     $0x64,%al               # Wait for not busy
9424   testb   $0x2,%al
9425   jnz     seta20.1
9426
9427   movb    $0xd1,%al               # 0xd1 -> port 0x64
9428   outb    %al,$0x64
9429
9430 seta20.2:
9431   inb     $0x64,%al               # Wait for not busy
9432   testb   $0x2,%al
9433   jnz     seta20.2
9434
9435   movb    $0xdf,%al               # 0xdf -> port 0x60
9436   outb    %al,$0x60
9437
9438   # Switch from real to protected mode.  Use a bootstrap GDT that makes
9439   # virtual addresses map directly to physical addresses so that the
9440   # effective memory map doesn't change during the transition.
9441   lgdt    gdtdesc
9442   movl    %cr0, %eax
9443   orl     $CR0_PE, %eax
9444   movl    %eax, %cr0
9445
9446
9447
9448
9449
```



```
9450   # Complete transition to 32-bit protected mode by using long jmp
9451   # to reload %cs and %eip.  The segment descriptors are set up with no
9452   # translation, so that the mapping is still the identity mapping.
9453   ljmp    $(SEG_KCODE<<3), $start32
9454
9455 .code32  # Tell assembler to generate 32-bit code now.
9456 start32:
9457   # Set up the protected-mode data segment registers
9458   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
9459   movw    %ax, %ds                # -> DS: Data Segment
9460   movw    %ax, %es                # -> ES: Extra Segment
9461   movw    %ax, %ss                # -> SS: Stack Segment
9462   movw    $0, %ax                 # Zero segments not ready for use
9463   movw    %ax, %fs                # -> FS
9464   movw    %ax, %gs                # -> GS
9465
9466   # Set up the stack pointer and call into C.
9467   movl    $start, %esp
9468   call    bootmain
9469
9470   # If bootmain returns (it shouldn't), trigger a Bochs
9471   # breakpoint if running under Bochs, then loop.
9472   movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
9473   movw    %ax, %dx
9474   outw    %ax, %dx
9475   movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
9476   outw    %ax, %dx
9477 spin:
9478   jmp     spin
9479
9480 # Bootstrap GDT
9481 .p2align 2                                # force 4 byte alignment
9482 gdt:
9483   SEG_NULLASM                             # null seg
9484   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
9485   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
9486
9487 gdtdesc:
9488   .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
9489   .long   gdt                             # address gdt
9490
9491
9492
9493
9494
9495
9496
9497
9498
9499
```

```
9500 // Boot loader.
9501 //
9502 // Part of the boot block, along with bootasm.S, which calls bootmain().
9503 // bootasm.S has put the processor into protected 32-bit mode.
9504 // bootmain() loads an ELF kernel image from the disk starting at
9505 // sector 1 and then jumps to the kernel entry routine.
9506
9507 #include "types.h"
9508 #include "elf.h"
9509 #include "x86.h"
9510 #include "memlayout.h"
9511
9512 #define SECTSIZE  512
9513
9514 void readseg(uchar*, uint, uint);
9515
9516 void
9517 bootmain(void)
9518 {
9519   struct elfhdr *elf;
9520   struct proghdr *ph, *eph;
9521   void (*entry)(void);
9522   uchar* pa;
9523
9524   elf = (struct elfhdr*)0x10000;  // scratch space
9525
9526   // Read 1st page off disk
9527   readseg((uchar*)elf, 4096, 0);
9528
9529   // Is this an ELF executable?
9530   if(elf->magic != ELF_MAGIC)
9531     return;  // let bootasm.S handle error
9532
9533   // Load each program segment (ignores ph flags).
9534   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
9535   eph = ph + elf->phnum;
9536   for(; ph < eph; ph++){
9537     pa = (uchar*)ph->paddr;
9538     readseg(pa, ph->filesz, ph->off);
9539     if(ph->memsz > ph->filesz)
9540       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
9541   }
9542
9543   // Call the entry point from the ELF header.
9544   // Does not return!
9545   entry = (void(*)(void))(elf->entry);
9546   entry();
9547 }
9548
9549
```



```
9550 void
9551 waitdisk(void)
9552 {
9553   // Wait for disk ready.
9554   while((inb(0x1F7) & 0xC0) != 0x40)
9555     ;
9556 }
9557
9558 // Read a single sector at offset into dst.
9559 void
9560 readsect(void *dst, uint offset)
9561 {
9562   // Issue command.
9563   waitdisk();
9564   outb(0x1F2, 1);   // count = 1
9565   outb(0x1F3, offset);
9566   outb(0x1F4, offset >> 8);
9567   outb(0x1F5, offset >> 16);
9568   outb(0x1F6, (offset >> 24) | 0xE0);
9569   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
9570
9571   // Read data.
9572   waitdisk();
9573   insl(0x1F0, dst, SECTSIZE/4);
9574 }
9575
9576 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
9577 // Might copy more than asked.
9578 void
9579 readseg(uchar* pa, uint count, uint offset)
9580 {
9581   uchar* epa;
9582
9583   epa = pa + count;
9584
9585   // Round down to sector boundary.
9586   pa -= offset % SECTSIZE;
9587
9588   // Translate from bytes to sectors; kernel starts at sector 1.
9589   offset = (offset / SECTSIZE) + 1;
9590
9591   // If this is too slow, we could read lots of sectors at a time.
9592   // We'd write more to memory than asked, but it doesn't matter --
9593   // we load in increasing order.
9594   for(; pa < epa; pa += SECTSIZE, offset++)
9595     readsect(pa, offset);
9596 }
9597
9598
9599
```

```
#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 ex
    {
        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[1
        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();
}
```

```
9800 struct stat;
9801 struct rtcdate;
9802 struct uproc;
9803
9804 // system calls
9805 int fork(void);
9806 int exit(void) __attribute__((noreturn));
9807 int wait(void);
9808 int pipe(int*);
9809 int write(int, void*, int);
9810 int read(int, void*, int);
9811 int close(int);
9812 int kill(int);
9813 int exec(char*, char**);
9814 int open(char*, int);
9815 int mknod(char*, short, short);
9816 int unlink(char*);
9817 int fstat(int fd, struct stat*);
9818 int link(char*, char*);
9819 int mkdir(char*);
9820 int chdir(char*);
9821 int dup(int);
9822 int getpid(void);
9823 char* sbrk(int);
9824 int sleep(int);
9825 int uptime(void);
9826 int halt(void);
9827 //Defined date function that allows user to call through shell
9828 int date(struct rtcdate*);
9829
9830 //Project 3
9831 int getuid(void);    // UID of the current process
9832 int getgid (void); // GID of the current process
9833 int getppid(void); // process ID of the parent process
9834
9835 int setuid(unsigned int);  // set UID to unsigned int
9836 int setgid(unsigned int);  // set GID to unsigned int
9837 int getprocs(int, struct uproc*);
9838
9839 // ulib.c
9840 int stat(char*, struct stat*);
9841 char* strcpy(char*, char*);
9842 void *memmove(void*, void*, int);
9843 char* strchr(const char*, char c);
9844 int strcmp(const char*, const char*);
9845 void printf(int, char*, ...);
9846 char* gets(char*, int max);
9847 uint strlen(char*);
9848 void* memset(void*, int, uint);
9849 void* malloc(uint);
```



```
9850 void free(void*);
9851 int atoi(const char*);
9852
9853
9854
9855
9856
9857
9858
9859
9860
9861
9862
9863
9864
9865
9866
9867
9868
9869
9870
9871
9872
9873
9874
9875
9876
9877
9878
9879
9880
9881
9882
9883
9884
9885
9886
9887
9888
9889
9890
9891
9892
9893
9894
9895
9896
9897
9898
9899
```

```
9900 // Project 3
9901 struct uproc{
9902    int pid;
9903    int uid;
9904    int gid;
9905    int ppid;
9906
9907    char STATE[16];
9908    int size;
9909    char name[16];
9910 };
9911
9912
9913
9914
9915
9916
9917
9918
9919
9920
9921
9922
9923
9924
9925
9926
9927
9928
9929
9930
9931
9932
9933
9934
9935
9936
9937
9938
9939
9940
9941
9942
9943
9944
9945
9946
9947
9948
9949
```



```
9950 #include "types.h"
9951 #include "user.h"
9952 #include "uproc.h"
9953
9954 int main (int argc, char *argv[])
9955 {
9956
9957   int i;
9958   int count = 0;
9959   int MAX = 65;
9960   struct uproc table[MAX];
9961   count = getprocs(MAX,table);
9962
9963   if(count < 0)
9964   {
9965     printf(2, "getprocs failed \n" );
9966     exit();
9967   }
9968
9969   for(i = 0; i < count; ++i)
9970   {
9971     printf(0, "PID:%d  UID:%d  GID:%d  PPID:%d  STATE:%s  SIZE:%d  NAME:%s\n"
9972            table[i].pid, table[i].uid, table[i].gid, table[i].ppid, table[i]
9973   }
9974
9975   exit();
9976 }
9977
9978
9979
9980
9981
9982
9983
9984
9985
9986
9987
9988
9989
9990
9991
9992
9993
9994
9995
9996
9997
9998
9999
```

```
10000 #include "types.h"
10001 #include "user.h"
10002
10003 int
10004 testuidgid (void)
10005 {
10006   int uid , gid , ppid;
10007   uid = getuid ();
10008   printf(2, "Current UID is: %d\n", uid);
10009   printf(2, "Setting UID to 100\n");
10010   setuid (100);
10011   uid = getuid ();
10012   printf(2, "Current UID is: %d\n", uid);
10013
10014   gid = getgid ();
10015   printf(2, "Current GID is: %d\n", gid);
10016   printf(2, "Setting GID to 100\n");
10017    setgid (100);
10018   gid = getgid ();
10019   printf(2, "Current GID is: %d\n", uid);
10020
10021   ppid = getppid ();
10022   printf(2, "My parent process is: %d\n", ppid);
10023   printf(2, "Done!\n");
10024
10025   exit();
10026 }
10027
10028
10029
10030
10031
10032
10033
10034
10035
10036
10037
10038
10039
10040
10041
10042
10043
10044
10045
10046
10047
10048
10049
```