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.

```
#include "types.h"
#include "user.h"
#include "date.h"
//#include "lapic.c"
int get_secs();

int
main (int argc, char * argv[])
{

    //error check
    if(argc < 2) {
   printf(2, "No arguments passed to time\n");
   exit();
    }

    char ** args = argv+1;

    //get start time
    int start = get_secs();

    if(fork() == 0) {
   exec(argv[1], args);
   printf(2, "exec failed running %s\n", argv[1] );
   exit();
     }
   wait();

    //get end time
    int finish = get_secs();

    int total_time = finish - start;

    printf(1, "%s ran in %d seconds \n", argv[1], total_time);



   exit();
}

int get_secs()
    {
           struct rtcdate time;
           date(&time);
   int hour = time.hour * 3600;
   int min = time.minute * 60;
   return time.second + min + hour;
    }


```



```
#include "types.h"
#include "user.h"
#include "date.h"
//#include "lapic.c"

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

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


    printf(1, "\n %d Hours, %d Minutes,  %d Seconds,  %d Months,  %d Day, %d
           r.hour, r.minute, r.second, r.month, r.day, r.year);

   exit();
}
```

```
0200 typedef unsigned int   uint;
0201 typedef unsigned short ushort;
0202 typedef unsigned char  uchar;
0203 typedef uint pde_t;
0204
0205
0206
0207
0208
0209
0210
0211
0212
0213
0214
0215
0216
0217
0218
0219
0220
0221
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 #define NPROC        64  // maximum number of processes
0251 #define KSTACKSIZE 4096  // size of per-process kernel stack
0252 #define NCPU          8  // maximum number of CPUs
0253 #define NOFILE       16  // open files per process
0254 #define NFILE       100  // open files per system
0255 #define NINODE       50  // maximum number of active i-nodes
0256 #define NDEV         10  // maximum major device number
0257 #define ROOTDEV       1  // device number of file system root disk
0258 #define MAXARG       32  // max exec arguments
0259 #define MAXOPBLOCKS  10  // max # of blocks any FS op writes
0260 #define LOGSIZE      (MAXOPBLOCKS*3)  // max data blocks in on-disk log
0261 #define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
0262 #define FSSIZE       1000  // size of file system in blocks
0263
0264
0265
0266
0267
0268
0269
0270
0271
0272
0273
0274
0275
0276
0277
0278
0279
0280
0281
0282
0283
0284
0285
0286
0287
0288
0289
0290
0291
0292
0293
0294
0295
0296
0297
0298
0299
```

```
0300 // Memory layout
0301
0302 #define EXTMEM  0x100000            // Start of extended memory
0303 #define PHYSTOP 0xE000000           // Top physical memory
0304 #define DEVSPACE 0xFE000000         // Other devices are at high addresses
0305
0306 // Key addresses for address space layout (see kmap in vm.c for layout)
0307 #define KERNBASE 0x80000000         // First kernel virtual address
0308 #define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked
0309
0310 #ifndef __ASSEMBLER__
0311
0312 static inline uint v2p(void *a) { return ((uint) (a))  - KERNBASE; }
0313 static inline void *p2v(uint a) { return (void *) ((a) + KERNBASE); }
0314
0315 #endif
0316
0317 #define V2P(a) (((uint) (a)) - KERNBASE)
0318 #define P2V(a) (((void *) (a)) + KERNBASE)
0319
0320 #define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
0321 #define P2V_WO(x) ((x) + KERNBASE)    // same as P2V, but without casts
0322
0323
0324
0325
0326
0327
0328
0329
0330
0331
0332
0333
0334
0335
0336
0337
0338
0339
0340
0341
0342
0343
0344
0345
0346
0347
0348
0349
```



```
0350 struct buf;
0351 struct context;
0352 struct file;
0353 struct inode;
0354 struct pipe;
0355 struct proc;
0356 struct rtcdate;
0357 struct spinlock;
0358 struct stat;
0359 struct superblock;
0360
0361 // bio.c
0362 void            binit(void);
0363 struct buf*     bread(uint, uint);
0364 void            brelse(struct buf*);
0365 void            bwrite(struct buf*);
0366
0367 // console.c
0368 void            consoleinit(void);
0369 void            cprintf(char*, ...);
0370 void            consoleintr(int(*)(void));
0371 void            panic(char*) __attribute__((noreturn));
0372
0373 // exec.c
0374 int             exec(char*, char**);
0375
0376 // file.c
0377 struct file*    filealloc(void);
0378 void            fileclose(struct file*);
0379 struct file*    filedup(struct file*);
0380 void            fileinit(void);
0381 int             fileread(struct file*, char*, int n);
0382 int             filestat(struct file*, struct stat*);
0383 int             filewrite(struct file*, char*, int n);
0384
0385 // fs.c
0386 void            readsb(int dev, struct superblock *sb);
0387 int             dirlink(struct inode*, char*, uint);
0388 struct inode*   dirlookup(struct inode*, char*, uint*);
0389 struct inode*   ialloc(uint, short);
0390 struct inode*   idup(struct inode*);
0391 void            iinit(int dev);
0392 void            ilock(struct inode*);
0393 void            iput(struct inode*);
0394 void            iunlock(struct inode*);
0395 void            iunlockput(struct inode*);
0396 void            iupdate(struct inode*);
0397 int             namecmp(const char*, const char*);
0398 struct inode*   namei(char*);
0399 struct inode*   nameiparent(char*, char*);
```

```
0400 int             readi(struct inode*, char*, uint, uint);
0401 void            stati(struct inode*, struct stat*);
0402 int             writei(struct inode*, char*, uint, uint);
0403
0404 // ide.c
0405 void            ideinit(void);
0406 void            ideintr(void);
0407 void            iderw(struct buf*);
0408
0409 // ioapic.c
0410 void            ioapicenable(int irq, int cpu);
0411 extern uchar    ioapicid;
0412 void            ioapicinit(void);
0413
0414 // kalloc.c
0415 char*           kalloc(void);
0416 void            kfree(char*);
0417 void            kinit1(void*, void*);
0418 void            kinit2(void*, void*);
0419
0420 // kbd.c
0421 void            kbdintr(void);
0422
0423 // lapic.c
0424 void            cmostime(struct rtcdate *r);
0425 int             cpunum(void);
0426 extern volatile uint*    lapic;
0427 void            lapiceoi(void);
0428 void            lapicinit(void);
0429 void            lapicstartap(uchar, uint);
0430 void            microdelay(int);
0431
0432 // log.c
0433 void            initlog(int dev);
0434 void            log_write(struct buf*);
0435 void            begin_op();
0436 void            end_op();
0437
0438 // mp.c
0439 extern int      ismp;
0440 int             mpbcpu(void);
0441 void            mpinit(void);
0442 void            mpstartthem(void);
0443
0444 // picirq.c
0445 void            picenable(int);
0446 void            picinit(void);
0447
0448
0449
```



```
0450 // pipe.c
0451 int             pipealloc(struct file**, struct file**);
0452 void            pipeclose(struct pipe*, int);
0453 int             piperead(struct pipe*, char*, int);
0454 int             pipewrite(struct pipe*, char*, int);
0455
0456
0457 // proc.c
0458 struct proc*    copyproc(struct proc*);
0459 void            exit(void);
0460 int             fork(void);
0461 int             growproc(int);
0462 int             kill(int);
0463 void            pinit(void);
0464 void            procdump(void);
0465 void            scheduler(void) __attribute__((noreturn));
0466 void            sched(void);
0467 void            sleep(void*, struct spinlock*);
0468 void            userinit(void);
0469 int             wait(void);
0470 void            wakeup(void*);
0471 void            yield(void);
0472
0473 // swtch.S
0474 void            swtch(struct context**, struct context*);
0475
0476 // spinlock.c
0477 void            acquire(struct spinlock*);
0478 void            getcallerpcs(void*, uint*);
0479 int             holding(struct spinlock*);
0480 void            initlock(struct spinlock*, char*);
0481 void            release(struct spinlock*);
0482 void            pushcli(void);
0483 void            popcli(void);
0484
0485 // string.c
0486 int             memcmp(const void*, const void*, uint);
0487 void*           memmove(void*, const void*, uint);
0488 void*           memset(void*, int, uint);
0489 char*           safestrcpy(char*, const char*, int);
0490 int             strlen(const char*);
0491 int             strncmp(const char*, const char*, uint);
0492 char*           strncpy(char*, const char*, int);
0493
0494 // syscall.c
0495 int             argint(int, int*);
0496 int             argptr(int, char**, int);
0497 int             argstr(int, char**);
0498 int             fetchint(uint, int*);
0499 int             fetchstr(uint, char**);
```

```
0500 void            syscall(void);
0501
0502 // timer.c
0503 void            timerinit(void);
0504
0505 // trap.c
0506 void            idtinit(void);
0507 extern uint     ticks;
0508 void            tvinit(void);
0509 extern struct spinlock tickslock;
0510
0511 // uart.c
0512 void            uartinit(void);
0513 void            uartintr(void);
0514 void            uartputc(int);
0515
0516 // vm.c
0517 void            seginit(void);
0518 void            kvmalloc(void);
0519 void            vmenable(void);
0520 pde_t*          setupkvm(void);
0521 char*           uva2ka(pde_t*, char*);
0522 int             allocuvm(pde_t*, uint, uint);
0523 int             deallocuvm(pde_t*, uint, uint);
0524 void            freevm(pde_t*);
0525 void            inituvm(pde_t*, char*, uint);
0526 int             loaduvm(pde_t*, char*, struct inode*, uint, uint);
0527 pde_t*          copyuvm(pde_t*, uint);
0528 void            switchuvm(struct proc*);
0529 void            switchkvm(void);
0530 int             copyout(pde_t*, uint, void*, uint);
0531 void            clearpteu(pde_t *pgdir, char *uva);
0532
0533 // number of elements in fixed-size array
0534 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
0535
0536
0537
0538
0539
0540
0541
0542
0543
0544
0545
0546
0547
0548
0549
```



```
0550 // Routines to let C code use special x86 instructions.
0551
0552 static inline uchar
0553 inb(ushort port)
0554 {
0555   uchar data;
0556
0557   asm volatile("in %1,%0" : "=a" (data) : "d" (port));
0558   return data;
0559 }
0560
0561 static inline void
0562 insl(int port, void *addr, int cnt)
0563 {
0564   asm volatile("cld; rep insl" :
0565                "=D" (addr), "=c" (cnt) :
0566                "d" (port), "0" (addr), "1" (cnt) :
0567                "memory", "cc");
0568 }
0569
0570 static inline void
0571 outb(ushort port, uchar data)
0572 {
0573   asm volatile("out %0,%1" : : "a" (data), "d" (port));
0574 }
0575
0576 static inline void
0577 outw(ushort port, ushort data)
0578 {
0579   asm volatile("out %0,%1" : : "a" (data), "d" (port));
0580 }
0581
0582 static inline void
0583 outsl(int port, const void *addr, int cnt)
0584 {
0585   asm volatile("cld; rep outsl" :
0586                "=S" (addr), "=c" (cnt) :
0587                "d" (port), "0" (addr), "1" (cnt) :
0588                "cc");
0589 }
0590
0591 static inline void
0592 stosb(void *addr, int data, int cnt)
0593 {
0594   asm volatile("cld; rep stosb" :
0595                "=D" (addr), "=c" (cnt) :
0596                "0" (addr), "1" (cnt), "a" (data) :
0597                "memory", "cc");
0598 }
0599
```

```
0600 static inline void
0601 stosl(void *addr, int data, int cnt)
0602 {
0603   asm volatile("cld; rep stosl" :
0604                "=D" (addr), "=c" (cnt) :
0605                "0" (addr), "1" (cnt), "a" (data) :
0606                "memory", "cc");
0607 }
0608
0609 struct segdesc;
0610
0611 static inline void
0612 lgdt(struct segdesc *p, int size)
0613 {
0614   volatile ushort pd[3];
0615
0616   pd[0] = size-1;
0617   pd[1] = (uint)p;
0618   pd[2] = (uint)p >> 16;
0619
0620   asm volatile("lgdt (%0)" : : "r" (pd));
0621 }
0622
0623 struct gatedesc;
0624
0625 static inline void
0626 lidt(struct gatedesc *p, int size)
0627 {
0628   volatile ushort pd[3];
0629
0630   pd[0] = size-1;
0631   pd[1] = (uint)p;
0632   pd[2] = (uint)p >> 16;
0633
0634   asm volatile("lidt (%0)" : : "r" (pd));
0635 }
0636
0637 static inline void
0638 ltr(ushort sel)
0639 {
0640   asm volatile("ltr %0" : : "r" (sel));
0641 }
0642
0643 static inline uint
0644 readeflags(void)
0645 {
0646   uint eflags;
0647   asm volatile("pushfl; popl %0" : "=r" (eflags));
0648   return eflags;
0649 }
```



```
0650 static inline void
0651 loadgs(ushort v)
0652 {
0653   asm volatile("movw %0, %%gs" : : "r" (v));
0654 }
0655
0656 static inline void
0657 cli(void)
0658 {
0659   asm volatile("cli");
0660 }
0661
0662 static inline void
0663 sti(void)
0664 {
0665   asm volatile("sti");
0666 }
0667
0668 static inline uint
0669 xchg(volatile uint *addr, uint newval)
0670 {
0671   uint result;
0672
0673   // The + in "+m" denotes a read-modify-write operand.
0674   asm volatile("lock; xchgl %0, %1" :
0675                "+m" (*addr), "=a" (result) :
0676                "1" (newval) :
0677                "cc");
0678   return result;
0679 }
0680
0681 static inline uint
0682 rcr2(void)
0683 {
0684   uint val;
0685   asm volatile("movl %%cr2,%0" : "=r" (val));
0686   return val;
0687 }
0688
0689 static inline void
0690 lcr3(uint val)
0691 {
0692   asm volatile("movl %0,%%cr3" : : "r" (val));
0693 }
0694
0695
0696
0697
0698
0699
```

```
0700 // Layout of the trap frame built on the stack by the
0701 // hardware and by trapasm.S, and passed to trap().
0702 struct trapframe {
0703   // registers as pushed by pusha
0704   uint edi;
0705   uint esi;
0706   uint ebp;
0707   uint oesp;      // useless & ignored
0708   uint ebx;
0709   uint edx;
0710   uint ecx;
0711   uint eax;
0712
0713   // rest of trap frame
0714   ushort gs;
0715   ushort padding1;
0716   ushort fs;
0717   ushort padding2;
0718   ushort es;
0719   ushort padding3;
0720   ushort ds;
0721   ushort padding4;
0722   uint trapno;
0723
0724   // below here defined by x86 hardware
0725   uint err;
0726   uint eip;
0727   ushort cs;
0728   ushort padding5;
0729   uint eflags;
0730
0731   // below here only when crossing rings, such as from user to kernel
0732   uint esp;
0733   ushort ss;
0734   ushort padding6;
0735 };
0736
0737
0738
0739
0740
0741
0742
0743
0744
0745
0746
0747
0748
0749
```



```
0750 //
0751 // assembler macros to create x86 segments
0752 //
0753
0754 #define SEG_NULLASM                                             \
0755         .word 0, 0;                                             \
0756         .byte 0, 0, 0, 0
0757
0758 // The 0xC0 means the limit is in 4096-byte units
0759 // and (for executable segments) 32-bit mode.
0760 #define SEG_ASM(type,base,lim)                                  \
0761         .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
0762         .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
0763                 (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
0764
0765 #define STA_X     0x8       // Executable segment
0766 #define STA_E     0x4       // Expand down (non-executable segments)
0767 #define STA_C     0x4       // Conforming code segment (executable only)
0768 #define STA_W     0x2       // Writeable (non-executable segments)
0769 #define STA_R     0x2       // Readable (executable segments)
0770 #define STA_A     0x1       // Accessed
0771
0772
0773
0774
0775
0776
0777
0778
0779
0780
0781
0782
0783
0784
0785
0786
0787
0788
0789
0790
0791
0792
0793
0794
0795
0796
0797
0798
0799
```

```
0800 // This file contains definitions for the
0801 // x86 memory management unit (MMU).
0802
0803 // Eflags register
0804 #define FL_CF           0x00000001      // Carry Flag
0805 #define FL_PF           0x00000004      // Parity Flag
0806 #define FL_AF           0x00000010      // Auxiliary carry Flag
0807 #define FL_ZF           0x00000040      // Zero Flag
0808 #define FL_SF           0x00000080      // Sign Flag
0809 #define FL_TF           0x00000100      // Trap Flag
0810 #define FL_IF           0x00000200      // Interrupt Enable
0811 #define FL_DF           0x00000400      // Direction Flag
0812 #define FL_OF           0x00000800      // Overflow Flag
0813 #define FL_IOPL_MASK    0x00003000      // I/O Privilege Level bitmask
0814 #define FL_IOPL_0       0x00000000      //   IOPL == 0
0815 #define FL_IOPL_1       0x00001000      //   IOPL == 1
0816 #define FL_IOPL_2       0x00002000      //   IOPL == 2
0817 #define FL_IOPL_3       0x00003000      //   IOPL == 3
0818 #define FL_NT           0x00004000      // Nested Task
0819 #define FL_RF           0x00010000      // Resume Flag
0820 #define FL_VM           0x00020000      // Virtual 8086 mode
0821 #define FL_AC           0x00040000      // Alignment Check
0822 #define FL_VIF          0x00080000      // Virtual Interrupt Flag
0823 #define FL_VIP          0x00100000      // Virtual Interrupt Pending
0824 #define FL_ID           0x00200000      // ID flag
0825
0826 // Control Register flags
0827 #define CR0_PE          0x00000001      // Protection Enable
0828 #define CR0_MP          0x00000002      // Monitor coProcessor
0829 #define CR0_EM          0x00000004      // Emulation
0830 #define CR0_TS          0x00000008      // Task Switched
0831 #define CR0_ET          0x00000010      // Extension Type
0832 #define CR0_NE          0x00000020      // Numeric Errror
0833 #define CR0_WP          0x00010000      // Write Protect
0834 #define CR0_AM          0x00040000      // Alignment Mask
0835 #define CR0_NW          0x20000000      // Not Writethrough
0836 #define CR0_CD          0x40000000      // Cache Disable
0837 #define CR0_PG          0x80000000      // Paging
0838
0839 #define CR4_PSE         0x00000010      // Page size extension
0840
0841 #define SEG_KCODE 1  // kernel code
0842 #define SEG_KDATA 2  // kernel data+stack
0843 #define SEG_KCPU  3  // kernel per-cpu data
0844 #define SEG_UCODE 4  // user code
0845 #define SEG_UDATA 5  // user data+stack
0846 #define SEG_TSS   6  // this process's task state
0847
0848
0849
```



```
0850 #ifndef __ASSEMBLER__
0851 // Segment Descriptor
0852 struct segdesc {
0853   uint lim_15_0 : 16;  // Low bits of segment limit
0854   uint base_15_0 : 16; // Low bits of segment base address
0855   uint base_23_16 : 8; // Middle bits of segment base address
0856   uint type : 4;       // Segment type (see STS_ constants)
0857   uint s : 1;          // 0 = system, 1 = application
0858   uint dpl : 2;        // Descriptor Privilege Level
0859   uint p : 1;          // Present
0860   uint lim_19_16 : 4;  // High bits of segment limit
0861   uint avl : 1;        // Unused (available for software use)
0862   uint rsv1 : 1;       // Reserved
0863   uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
0864   uint g : 1;          // Granularity: limit scaled by 4K when set
0865   uint base_31_24 : 8; // High bits of segment base address
0866 };
0867
0868 // Normal segment
0869 #define SEG(type, base, lim, dpl) (struct segdesc)    \
0870 { ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff,      \
0871   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
0872   (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
0873 #define SEG16(type, base, lim, dpl) (struct segdesc)  \
0874 { (lim) & 0xffff, (uint)(base) & 0xffff,              \
0875   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
0876   (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
0877 #endif
0878
0879 #define DPL_USER    0x3     // User DPL
0880
0881 // Application segment type bits
0882 #define STA_X       0x8     // Executable segment
0883 #define STA_E       0x4     // Expand down (non-executable segments)
0884 #define STA_C       0x4     // Conforming code segment (executable only)
0885 #define STA_W       0x2     // Writeable (non-executable segments)
0886 #define STA_R       0x2     // Readable (executable segments)
0887 #define STA_A       0x1     // Accessed
0888
0889 // System segment type bits
0890 #define STS_T16A    0x1     // Available 16-bit TSS
0891 #define STS_LDT     0x2     // Local Descriptor Table
0892 #define STS_T16B    0x3     // Busy 16-bit TSS
0893 #define STS_CG16    0x4     // 16-bit Call Gate
0894 #define STS_TG      0x5     // Task Gate / Coum Transmitions
0895 #define STS_IG16    0x6     // 16-bit Interrupt Gate
0896 #define STS_TG16    0x7     // 16-bit Trap Gate
0897 #define STS_T32A    0x9     // Available 32-bit TSS
0898 #define STS_T32B    0xB     // Busy 32-bit TSS
0899 #define STS_CG32    0xC     // 32-bit Call Gate
```

```
0900 #define STS_IG32    0xE     // 32-bit Interrupt Gate
0901 #define STS_TG32    0xF     // 32-bit Trap Gate
0902
0903 // A virtual address 'la' has a three-part structure as follows:
0904 //
0905 // +--------10------+-------10-------+---------12----------+
0906 // | Page Directory |   Page Table   | Offset within Page  |
0907 // |      Index     |      Index     |                     |
0908 // +----------------+----------------+---------------------+
0909 //  \--- PDX(va) --/ \--- PTX(va) --/
0910
0911 // page directory index
0912 #define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)
0913
0914 // page table index
0915 #define PTX(va)         (((uint)(va) >> PTXSHIFT) & 0x3FF)
0916
0917 // construct virtual address from indexes and offset
0918 #define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
0919
0920 // Page directory and page table constants.
0921 #define NPDENTRIES      1024    // # directory entries per page directory
0922 #define NPTENTRIES      1024    // # PTEs per page table
0923 #define PGSIZE          4096    // bytes mapped by a page
0924
0925 #define PGSHIFT         12      // log2(PGSIZE)
0926 #define PTXSHIFT        12      // offset of PTX in a linear address
0927 #define PDXSHIFT        22      // offset of PDX in a linear address
0928
0929 #define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
0930 #define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
0931
0932 // Page table/directory entry flags.
0933 #define PTE_P           0x001   // Present
0934 #define PTE_W           0x002   // Writeable
0935 #define PTE_U           0x004   // User
0936 #define PTE_PWT         0x008   // Write-Through
0937 #define PTE_PCD         0x010   // Cache-Disable
0938 #define PTE_A           0x020   // Accessed
0939 #define PTE_D           0x040   // Dirty
0940 #define PTE_PS          0x080   // Page Size
0941 #define PTE_MBZ         0x180   // Bits must be zero
0942
0943 // Address in page table or page directory entry
0944 #define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
0945 #define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF)
0946
0947 #ifndef __ASSEMBLER__
0948 typedef uint pte_t;
0949
```



```
0950 // Task state segment format
0951 struct taskstate {
0952   uint link;         // Old ts selector
0953   uint esp0;         // Stack pointers and segment selectors
0954   ushort ss0;        //   after an increase in privilege level
0955   ushort padding1;
0956   uint *esp1;
0957   ushort ss1;
0958   ushort padding2;
0959   uint *esp2;
0960   ushort ss2;
0961   ushort padding3;
0962   void *cr3;         // Page directory base
0963   uint *eip;         // Saved state from last task switch
0964   uint eflags;
0965   uint eax;          // More saved state (registers)
0966   uint ecx;
0967   uint edx;
0968   uint ebx;
0969   uint *esp;
0970   uint *ebp;
0971   uint esi;
0972   uint edi;
0973   ushort es;         // Even more saved state (segment selectors)
0974   ushort padding4;
0975   ushort cs;
0976   ushort padding5;
0977   ushort ss;
0978   ushort padding6;
0979   ushort ds;
0980   ushort padding7;
0981   ushort fs;
0982   ushort padding8;
0983   ushort gs;
0984   ushort padding9;
0985   ushort ldt;
0986   ushort padding10;
0987   ushort t;          // Trap on task switch
0988   ushort iomb;       // I/O map base address
0989 };
0990
0991
0992
0993
0994
0995
0996
0997
0998
0999
```

```
1000 // Gate descriptors for interrupts and traps
1001 struct gatedesc {
1002   uint off_15_0 : 16;   // low 16 bits of offset in segment
1003   uint cs : 16;         // code segment selector
1004   uint args : 5;        // # args, 0 for interrupt/trap gates
1005   uint rsv1 : 3;        // reserved(should be zero I guess)
1006   uint type : 4;        // type(STS_{TG,IG32,TG32})
1007   uint s : 1;           // must be 0 (system)
1008   uint dpl : 2;         // descriptor(meaning new) privilege level
1009   uint p : 1;           // Present
1010   uint off_31_16 : 16;  // high bits of offset in segment
1011 };
1012
1013 // Set up a normal interrupt/trap gate descriptor.
1014 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
1015 //   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
1016 // - sel: Code segment selector for interrupt/trap handler
1017 // - off: Offset in code segment for interrupt/trap handler
1018 // - dpl: Descriptor Privilege Level -
1019 //        the privilege level required for software to invoke
1020 //        this interrupt/trap gate explicitly using an int instruction.
1021 #define SETGATE(gate, istrap, sel, off, d)                \
1022 {                                                         \
1023   (gate).off_15_0 = (uint)(off) & 0xffff;                \
1024   (gate).cs = (sel);                                      \
1025   (gate).args = 0;                                        \
1026   (gate).rsv1 = 0;                                        \
1027   (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
1028   (gate).s = 0;                                           \
1029   (gate).dpl = (d);                                       \
1030   (gate).p = 1;                                           \
1031   (gate).off_31_16 = (uint)(off) >> 16;                  \
1032 }
1033
1034 #endif
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
```



```
1050 // Format of an ELF executable file
1051
1052 #define ELF_MAGIC 0x464C457FU  // "\x7FELF" in little endian
1053
1054 // File header
1055 struct elfhdr {
1056   uint magic;  // must equal ELF_MAGIC
1057   uchar elf[12];
1058   ushort type;
1059   ushort machine;
1060   uint version;
1061   uint entry;
1062   uint phoff;
1063   uint shoff;
1064   uint flags;
1065   ushort ehsize;
1066   ushort phentsize;
1067   ushort phnum;
1068   ushort shentsize;
1069   ushort shnum;
1070   ushort shstrndx;
1071 };
1072
1073 // Program section header
1074 struct proghdr {
1075   uint type;
1076   uint off;
1077   uint vaddr;
1078   uint paddr;
1079   uint filesz;
1080   uint memsz;
1081   uint flags;
1082   uint align;
1083 };
1084
1085 // Values for Proghdr type
1086 #define ELF_PROG_LOAD           1
1087
1088 // Flag bits for Proghdr flags
1089 #define ELF_PROG_FLAG_EXEC      1
1090 #define ELF_PROG_FLAG_WRITE     2
1091 #define ELF_PROG_FLAG_READ      4
1092
1093
1094
1095
1096
1097
1098
1099
```

```
1100 # Multiboot header, for multiboot boot loaders like GNU Grub.
1101 # http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
1102 #
1103 # Using GRUB 2, you can boot xv6 from a file stored in a
1104 # Linux file system by copying kernel or kernelmemfs to /boot
1105 # and then adding this menu entry:
1106 #
1107 # menuentry "xv6" {
1108 #  insmod ext2
1109 #  set root='(hd0,msdos1)'
1110 #  set kernel='/boot/kernel'
1111 #  echo "Loading ${kernel}..."
1112 #  multiboot ${kernel} ${kernel}
1113 #  boot
1114 # }
1115
1116 #include "asm.h"
1117 #include "memlayout.h"
1118 #include "mmu.h"
1119 #include "param.h"
1120
1121 # Multiboot header.  Data to direct multiboot loader.
1122 .p2align 2
1123 .text
1124 .globl multiboot_header
1125 multiboot_header:
1126   #define magic 0x1badb002
1127   #define flags 0
1128   .long magic
1129   .long flags
1130   .long (-magic-flags)
1131
1132 # By convention, the _start symbol specifies the ELF entry point.
1133 # Since we haven't set up virtual memory yet, our entry point is
1134 # the physical address of 'entry'.
1135 .globl _start
1136 _start = V2P_WO(entry)
1137
1138 # Entering xv6 on boot processor, with paging off.
1139 .globl entry
1140 entry:
1141   # Turn on page size extension for 4Mbyte pages
1142   movl    %cr4, %eax
1143   orl     $(CR4_PSE), %eax
1144   movl    %eax, %cr4
1145   # Set page directory
1146   movl    $(V2P_WO(entrypgdir)), %eax
1147   movl    %eax, %cr3
1148   # Turn on paging.
1149   movl    %cr0, %eax
```



```
1150   orl     $(CR0_PG|CR0_WP), %eax
1151   movl    %eax, %cr0
1152
1153   # Set up the stack pointer.
1154   movl $(stack + KSTACKSIZE), %esp
1155
1156   # Jump to main(), and switch to executing at
1157   # high addresses. The indirect call is needed because
1158   # the assembler produces a PC-relative instruction
1159   # for a direct jump.
1160   mov $main, %eax
1161   jmp *%eax
1162
1163 .comm stack, KSTACKSIZE
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
```

```
1200 #include "asm.h"
1201 #include "memlayout.h"
1202 #include "mmu.h"
1203
1204 # Each non-boot CPU ("AP") is started up in response to a STARTUP
1205 # IPI from the boot CPU.  Section B.4.2 of the Multi-Processor
1206 # Specification says that the AP will start in real mode with CS:IP
1207 # set to XY00:0000, where XY is an 8-bit value sent with the
1208 # STARTUP. Thus this code must start at a 4096-byte boundary.
1209 #
1210 # Because this code sets DS to zero, it must sit
1211 # at an address in the low 2^16 bytes.
1212 #
1213 # Startothers (in main.c) sends the STARTUPs one at a time.
1214 # It copies this code (start) at 0x7000.  It puts the address of
1215 # a newly allocated per-core stack in start-4,the address of the
1216 # place to jump to (mpenter) in start-8, and the physical address
1217 # of entrypgdir in start-12.
1218 #
1219 # This code is identical to bootasm.S except:
1220 #   - it does not need to enable A20
1221 #   - it uses the address at start-4, start-8, and start-12
1222
1223 .code16
1224 .globl start
1225 start:
1226   cli
1227
1228   xorw    %ax,%ax
1229   movw    %ax,%ds
1230   movw    %ax,%es
1231   movw    %ax,%ss
1232
1233   lgdt    gdtdesc
1234   movl    %cr0, %eax
1235   orl     $CR0_PE, %eax
1236   movl    %eax, %cr0
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
```



```
1250   ljmpl   $(SEG_KCODE<<3), $(start32)
1251
1252 .code32
1253 start32:
1254   movw    $(SEG_KDATA<<3), %ax
1255   movw    %ax, %ds
1256   movw    %ax, %es
1257   movw    %ax, %ss
1258   movw    $0, %ax
1259   movw    %ax, %fs
1260   movw    %ax, %gs
1261
1262   # Turn on page size extension for 4Mbyte pages
1263   movl    %cr4, %eax
1264   orl     $(CR4_PSE), %eax
1265   movl    %eax, %cr4
1266   # Use enterpgdir as our initial page table
1267   movl    (start-12), %eax
1268   movl    %eax, %cr3
1269   # Turn on paging.
1270   movl    %cr0, %eax
1271   orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
1272   movl    %eax, %cr0
1273
1274   # Switch to the stack allocated by startothers()
1275   movl    (start-4), %esp
1276   # Call mpenter()
1277   call      *(start-8)
1278
1279   movw    $0x8a00, %ax
1280   movw    %ax, %dx
1281   outw    %ax, %dx
1282   movw    $0x8ae0, %ax
1283   outw    %ax, %dx
1284 spin:
1285   jmp     spin
1286
1287 .p2align 2
1288 gdt:
1289   SEG_NULLASM
1290   SEG_ASM(STA_X|STA_R, 0, 0xffffffff)
1291   SEG_ASM(STA_W, 0, 0xffffffff)
1292
1293
1294 gdtdesc:
1295   .word   (gdtdesc - gdt - 1)
1296   .long   gdt
1297
1298
1299
```

```
1300 #include "types.h"
1301 #include "defs.h"
1302 #include "param.h"
1303 #include "memlayout.h"
1304 #include "mmu.h"
1305 #include "proc.h"
1306 #include "x86.h"
1307
1308 static void startothers(void);
1309 static void mpmain(void)  __attribute__((noreturn));
1310 extern pde_t *kpgdir;
1311 extern char end[]; // first address after kernel loaded from ELF file
1312
1313 // Bootstrap processor starts running C code here.
1314 // Allocate a real stack and switch to it, first
1315 // doing some setup required for memory allocator to work.
1316 int
1317 main(void)
1318 {
1319   kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320   kvmalloc();      // kernel page table
1321   mpinit();        // collect info about this machine
1322   lapicinit();
1323   seginit();       // set up segments
1324   cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
1325   picinit();       // interrupt controller
1326   ioapicinit();    // another interrupt controller
1327   consoleinit();   // I/O devices & their interrupts
1328   uartinit();      // serial port
1329   pinit();         // process table
1330   tvinit();        // trap vectors
1331   binit();         // buffer cache
1332   fileinit();      // file table
1333   ideinit();       // disk
1334   if(!ismp)
1335     timerinit();   // uniprocessor timer
1336   startothers();   // start other processors
1337   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1338   userinit();      // first user process
1339   // Finish setting up this processor in mpmain.
1340   mpmain();
1341 }
1342
1343
1344
1345
1346
1347
1348
1349
```



```
1350 // Other CPUs jump here from entryother.S.
1351 static void
1352 mpenter(void)
1353 {
1354   switchkvm();
1355   seginit();
1356   lapicinit();
1357   mpmain();
1358 }
1359
1360 // Common CPU setup code.
1361 static void
1362 mpmain(void)
1363 {
1364   cprintf("cpu%d: starting\n", cpu->id);
1365   idtinit();       // load idt register
1366   xchg(&cpu->started, 1); // tell startothers() we're up
1367   scheduler();     // start running processes
1368 }
1369
1370 pde_t entrypgdir[];  // For entry.S
1371
1372 // Start the non-boot (AP) processors.
1373 static void
1374 startothers(void)
1375 {
1376   extern uchar _binary_entryother_start[], _binary_entryother_size[];
1377   uchar *code;
1378   struct cpu *c;
1379   char *stack;
1380
1381   // Write entry code to unused memory at 0x7000.
1382   // The linker has placed the image of entryother.S in
1383   // _binary_entryother_start.
1384   code = p2v(0x7000);
1385   memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
1386
1387   for(c = cpus; c < cpus+ncpu; c++){
1388     if(c == cpus+cpunum())  // We've started already.
1389       continue;
1390
1391     // Tell entryother.S what stack to use, where to enter, and what
1392     // pgdir to use. We cannot use kpgdir yet, because the AP processor
1393     // is running in low  memory, so we use entrypgdir for the APs too.
1394     stack = kalloc();
1395     *(void**)(code-4) = stack + KSTACKSIZE;
1396     *(void**)(code-8) = mpenter;
1397     *(int**)(code-12) = (void *) v2p(entrypgdir);
1398
1399     lapicstartap(c->id, v2p(code));
```

```
1400     // wait for cpu to finish mpmain()
1401     while(c->started == 0)
1402       ;
1403   }
1404 }
1405
1406 // Boot page table used in entry.S and entryother.S.
1407 // Page directories (and page tables), must start on a page boundary,
1408 // hence the "__aligned__" attribute.
1409 // Use PTE_PS in page directory entry to enable 4Mbyte pages.
1410 __attribute__((__aligned__(PGSIZE)))
1411 pde_t entrypgdir[NPDENTRIES] = {
1412   // Map VA's [0, 4MB) to PA's [0, 4MB)
1413   [0] = (0) | PTE_P | PTE_W | PTE_PS,
1414   // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1415   [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
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 // Blank page.
1501
1502
1503
1504
1505
1506
1507
1508
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 // Blank page.
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
```

```
1600 // Mutual exclusion lock.
1601 struct spinlock {
1602   uint locked;       // Is the lock held?
1603
1604   // For debugging:
1605   char *name;        // Name of lock.
1606   struct cpu *cpu;   // The cpu holding the lock.
1607   uint pcs[10];      // The call stack (an array of program counters)
1608                      // that locked the lock.
1609 };
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
```



```
1650 // Mutual exclusion spin locks.
1651
1652 #include "types.h"
1653 #include "defs.h"
1654 #include "param.h"
1655 #include "x86.h"
1656 #include "memlayout.h"
1657 #include "mmu.h"
1658 #include "proc.h"
1659 #include "spinlock.h"
1660
1661 void
1662 initlock(struct spinlock *lk, char *name)
1663 {
1664   lk->name = name;
1665   lk->locked = 0;
1666   lk->cpu = 0;
1667 }
1668
1669 // Acquire the lock.
1670 // Loops (spins) until the lock is acquired.
1671 // Holding a lock for a long time may cause
1672 // other CPUs to waste time spinning to acquire it.
1673 void
1674 acquire(struct spinlock *lk)
1675 {
1676   pushcli(); // disable interrupts to avoid deadlock.
1677   if(holding(lk))
1678     panic("acquire");
1679
1680   // The xchg is atomic.
1681   // It also serializes, so that reads after acquire are not
1682   // reordered before it.
1683   while(xchg(&lk->locked, 1) != 0)
1684     ;
1685
1686   // Record info about lock acquisition for debugging.
1687   lk->cpu = cpu;
1688   getcallerpcs(&lk, lk->pcs);
1689 }
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
```

```
1700 // Release the lock.
1701 void
1702 release(struct spinlock *lk)
1703 {
1704   if(!holding(lk))
1705     panic("release");
1706
1707   lk->pcs[0] = 0;
1708   lk->cpu = 0;
1709
1710   // The xchg serializes, so that reads before release are
1711   // not reordered after it.  The 1996 PentiumPro manual (Volume 3,
1712   // 7.2) says reads can be carried out speculatively and in
1713   // any order, which implies we need to serialize here.
1714   // But the 2007 Intel 64 Architecture Memory Ordering White
1715   // Paper says that Intel 64 and IA-32 will not move a load
1716   // after a store. So lock->locked = 0 would work here.
1717   // The xchg being asm volatile ensures gcc emits it after
1718   // the above assignments (and after the critical section).
1719   xchg(&lk->locked, 0);
1720
1721   popcli();
1722 }
1723
1724 // Record the current call stack in pcs[] by following the %ebp chain.
1725 void
1726 getcallerpcs(void *v, uint pcs[])
1727 {
1728   uint *ebp;
1729   int i;
1730
1731   ebp = (uint*)v - 2;
1732   for(i = 0; i < 10; i++){
1733     if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
1734       break;
1735     pcs[i] = ebp[1];     // saved %eip
1736     ebp = (uint*)ebp[0]; // saved %ebp
1737   }
1738   for(; i < 10; i++)
1739     pcs[i] = 0;
1740 }
1741
1742 // Check whether this cpu is holding the lock.
1743 int
1744 holding(struct spinlock *lock)
1745 {
1746   return lock->locked && lock->cpu == cpu;
1747 }
1748
1749
```



```
1750 // Pushcli/popcli are like cli/sti except that they are matched:
1751 // it takes two popcli to undo two pushcli.  Also, if interrupts
1752 // are off, then pushcli, popcli leaves them off.
1753
1754 void
1755 pushcli(void)
1756 {
1757   int eflags;
1758
1759   eflags = readeflags();
1760   cli();
1761   if(cpu->ncli++ == 0)
1762     cpu->intena = eflags & FL_IF;
1763 }
1764
1765 void
1766 popcli(void)
1767 {
1768   if(readeflags()&FL_IF)
1769     panic("popcli - interruptible");
1770   if(--cpu->ncli < 0)
1771     panic("popcli");
1772   if(cpu->ncli == 0 && cpu->intena)
1773     sti();
1774 }
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
```

```
1800 #include "param.h"
1801 #include "types.h"
1802 #include "defs.h"
1803 #include "x86.h"
1804 #include "memlayout.h"
1805 #include "mmu.h"
1806 #include "proc.h"
1807 #include "elf.h"
1808
1809 extern char data[];  // defined by kernel.ld
1810 pde_t *kpgdir;  // for use in scheduler()
1811 struct segdesc gdt[NSEGS];
1812
1813 // Set up CPU's kernel segment descriptors.
1814 // Run once on entry on each CPU.
1815 void
1816 seginit(void)
1817 {
1818   struct cpu *c;
1819
1820   // Map "logical" addresses to virtual addresses using identity map.
1821   // Cannot share a CODE descriptor for both kernel and user
1822   // because it would have to have DPL_USR, but the CPU forbids
1823   // an interrupt from CPL=0 to DPL=3.
1824   c = &cpus[cpunum()];
1825   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
1826   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1827   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
1828   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1829
1830   // Map cpu, and curproc
1831   c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
1832
1833   lgdt(c->gdt, sizeof(c->gdt));
1834   loadgs(SEG_KCPU << 3);
1835
1836   // Initialize cpu-local storage.
1837   cpu = c;
1838   proc = 0;
1839 }
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
```



```
1850 // Return the address of the PTE in page table pgdir
1851 // that corresponds to virtual address va.  If alloc!=0,
1852 // create any required page table pages.
1853 static pte_t *
1854 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1855 {
1856   pde_t *pde;
1857   pte_t *pgtab;
1858
1859   pde = &pgdir[PDX(va)];
1860   if(*pde & PTE_P){
1861     pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
1862   } else {
1863     if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1864       return 0;
1865     // Make sure all those PTE_P bits are zero.
1866     memset(pgtab, 0, PGSIZE);
1867     // The permissions here are overly generous, but they can
1868     // be further restricted by the permissions in the page table
1869     // entries, if necessary.
1870     *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
1871   }
1872   return &pgtab[PTX(va)];
1873 }
1874
1875 // Create PTEs for virtual addresses starting at va that refer to
1876 // physical addresses starting at pa. va and size might not
1877 // be page-aligned.
1878 static int
1879 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1880 {
1881   char *a, *last;
1882   pte_t *pte;
1883
1884   a = (char*)PGROUNDDOWN((uint)va);
1885   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1886   for(;;){
1887     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1888       return -1;
1889     if(*pte & PTE_P)
1890       panic("remap");
1891     *pte = pa | perm | PTE_P;
1892     if(a == last)
1893       break;
1894     a += PGSIZE;
1895     pa += PGSIZE;
1896   }
1897   return 0;
1898 }
1899
```

```
1900 // There is one page table per process, plus one that's used when
1901 // a CPU is not running any process (kpgdir). The kernel uses the
1902 // current process's page table during system calls and interrupts;
1903 // page protection bits prevent user code from using the kernel's
1904 // mappings.
1905 //
1906 // setupkvm() and exec() set up every page table like this:
1907 //
1908 //   0..KERNBASE: user memory (text+data+stack+heap), mapped to
1909 //                phys memory allocated by the kernel
1910 //   KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
1911 //   KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
1912 //                for the kernel's instructions and r/o data
1913 //   data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
1914 //                                  rw data + free physical memory
1915 //   0xfe000000..0: mapped direct (devices such as ioapic)
1916 //
1917 // The kernel allocates physical memory for its heap and for user memory
1918 // between V2P(end) and the end of physical memory (PHYSTOP)
1919 // (directly addressable from end..P2V(PHYSTOP)).
1920
1921 // This table defines the kernel's mappings, which are present in
1922 // every process's page table.
1923 static struct kmap {
1924   void *virt;
1925   uint phys_start;
1926   uint phys_end;
1927   int perm;
1928 } kmap[] = {
1929  { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
1930  { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
1931  { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
1932  { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
1933 };
1934
1935 // Set up kernel part of a page table.
1936 pde_t*
1937 setupkvm(void)
1938 {
1939   pde_t *pgdir;
1940   struct kmap *k;
1941
1942   if((pgdir = (pde_t*)kalloc()) == 0)
1943     return 0;
1944   memset(pgdir, 0, PGSIZE);
1945   if (p2v(PHYSTOP) > (void*)DEVSPACE)
1946     panic("PHYSTOP too high");
1947   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1948     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1949                 (uint)k->phys_start, k->perm) < 0)
```



```
1950       return 0;
1951   return pgdir;
1952 }
1953
1954 // Allocate one page table for the machine for the kernel address
1955 // space for scheduler processes.
1956 void
1957 kvmalloc(void)
1958 {
1959   kpgdir = setupkvm();
1960   switchkvm();
1961 }
1962
1963 // Switch h/w page table register to the kernel-only page table,
1964 // for when no process is running.
1965 void
1966 switchkvm(void)
1967 {
1968   lcr3(v2p(kpgdir));   // switch to the kernel page table
1969 }
1970
1971 // Switch TSS and h/w page table to correspond to process p.
1972 void
1973 switchuvm(struct proc *p)
1974 {
1975   pushcli();
1976   cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
1977   cpu->gdt[SEG_TSS].s = 0;
1978   cpu->ts.ss0 = SEG_KDATA << 3;
1979   cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
1980   ltr(SEG_TSS << 3);
1981   if(p->pgdir == 0)
1982     panic("switchuvm: no pgdir");
1983   lcr3(v2p(p->pgdir));  // switch to new address space
1984   popcli();
1985 }
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
```

```
2000 // Load the initcode into address 0 of pgdir.
2001 // sz must be less than a page.
2002 void
2003 inituvm(pde_t *pgdir, char *init, uint sz)
2004 {
2005   char *mem;
2006
2007   if(sz >= PGSIZE)
2008     panic("inituvm: more than a page");
2009   mem = kalloc();
2010   memset(mem, 0, PGSIZE);
2011   mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
2012   memmove(mem, init, sz);
2013 }
2014
2015 // Load a program segment into pgdir.  addr must be page-aligned
2016 // and the pages from addr to addr+sz must already be mapped.
2017 int
2018 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
2019 {
2020   uint i, pa, n;
2021   pte_t *pte;
2022
2023   if((uint) addr % PGSIZE != 0)
2024     panic("loaduvm: addr must be page aligned");
2025   for(i = 0; i < sz; i += PGSIZE){
2026     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
2027       panic("loaduvm: address should exist");
2028     pa = PTE_ADDR(*pte);
2029     if(sz - i < PGSIZE)
2030       n = sz - i;
2031     else
2032       n = PGSIZE;
2033     if(readi(ip, p2v(pa), offset+i, n) != n)
2034       return -1;
2035   }
2036   return 0;
2037 }
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
```



```
2050 // Allocate page tables and physical memory to grow process from oldsz to
2051 // newsz, which need not be page aligned.  Returns new size or 0 on error.
2052 int
2053 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
2054 {
2055   char *mem;
2056   uint a;
2057
2058   if(newsz >= KERNBASE)
2059     return 0;
2060   if(newsz < oldsz)
2061     return oldsz;
2062
2063   a = PGROUNDUP(oldsz);
2064   for(; a < newsz; a += PGSIZE){
2065     mem = kalloc();
2066     if(mem == 0){
2067       cprintf("allocuvm out of memory\n");
2068       deallocuvm(pgdir, newsz, oldsz);
2069       return 0;
2070     }
2071     memset(mem, 0, PGSIZE);
2072     mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U);
2073   }
2074   return newsz;
2075 }
2076
2077 // Deallocate user pages to bring the process size from oldsz to
2078 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
2079 // need to be less than oldsz.  oldsz can be larger than the actual
2080 // process size.  Returns the new process size.
2081 int
2082 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
2083 {
2084   pte_t *pte;
2085   uint a, pa;
2086
2087   if(newsz >= oldsz)
2088     return oldsz;
2089
2090   a = PGROUNDUP(newsz);
2091   for(; a  < oldsz; a += PGSIZE){
2092     pte = walkpgdir(pgdir, (char*)a, 0);
2093     if(!pte)
2094       a += (NPTENTRIES - 1) * PGSIZE;
2095     else if((*pte & PTE_P) != 0){
2096       pa = PTE_ADDR(*pte);
2097       if(pa == 0)
2098         panic("kfree");
2099       char *v = p2v(pa);
```

```
2100       kfree(v);
2101       *pte = 0;
2102     }
2103   }
2104   return newsz;
2105 }
2106
2107 // Free a page table and all the physical memory pages
2108 // in the user part.
2109 void
2110 freevm(pde_t *pgdir)
2111 {
2112   uint i;
2113
2114   if(pgdir == 0)
2115     panic("freevm: no pgdir");
2116   deallocuvm(pgdir, KERNBASE, 0);
2117   for(i = 0; i < NPDENTRIES; i++){
2118     if(pgdir[i] & PTE_P){
2119       char * v = p2v(PTE_ADDR(pgdir[i]));
2120       kfree(v);
2121     }
2122   }
2123   kfree((char*)pgdir);
2124 }
2125
2126 // Clear PTE_U on a page. Used to create an inaccessible
2127 // page beneath the user stack.
2128 void
2129 clearpteu(pde_t *pgdir, char *uva)
2130 {
2131   pte_t *pte;
2132
2133   pte = walkpgdir(pgdir, uva, 0);
2134   if(pte == 0)
2135     panic("clearpteu");
2136   *pte &= ~PTE_U;
2137 }
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
```



```
2150 // Given a parent process's page table, create a copy
2151 // of it for a child.
2152 pde_t*
2153 copyuvm(pde_t *pgdir, uint sz)
2154 {
2155   pde_t *d;
2156   pte_t *pte;
2157   uint pa, i, flags;
2158   char *mem;
2159
2160   if((d = setupkvm()) == 0)
2161     return 0;
2162   for(i = 0; i < sz; i += PGSIZE){
2163     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
2164       panic("copyuvm: pte should exist");
2165     if(!(*pte & PTE_P))
2166       panic("copyuvm: page not present");
2167     pa = PTE_ADDR(*pte);
2168     flags = PTE_FLAGS(*pte);
2169     if((mem = kalloc()) == 0)
2170       goto bad;
2171     memmove(mem, (char*)p2v(pa), PGSIZE);
2172     if(mappages(d, (void*)i, PGSIZE, v2p(mem), flags) < 0)
2173       goto bad;
2174   }
2175   return d;
2176
2177 bad:
2178   freevm(d);
2179   return 0;
2180 }
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
```

```
2200 // Map user virtual address to kernel address.
2201 char*
2202 uva2ka(pde_t *pgdir, char *uva)
2203 {
2204   pte_t *pte;
2205
2206   pte = walkpgdir(pgdir, uva, 0);
2207   if((*pte & PTE_P) == 0)
2208     return 0;
2209   if((*pte & PTE_U) == 0)
2210     return 0;
2211   return (char*)p2v(PTE_ADDR(*pte));
2212 }
2213
2214 // Copy len bytes from p to user address va in page table pgdir.
2215 // Most useful when pgdir is not the current page table.
2216 // uva2ka ensures this only works for PTE_U pages.
2217 int
2218 copyout(pde_t *pgdir, uint va, void *p, uint len)
2219 {
2220   char *buf, *pa0;
2221   uint n, va0;
2222
2223   buf = (char*)p;
2224   while(len > 0){
2225     va0 = (uint)PGROUNDDOWN(va);
2226     pa0 = uva2ka(pgdir, (char*)va0);
2227     if(pa0 == 0)
2228       return -1;
2229     n = PGSIZE - (va - va0);
2230     if(n > len)
2231       n = len;
2232     memmove(pa0 + (va - va0), buf, n);
2233     len -= n;
2234     buf += n;
2235     va = va0 + PGSIZE;
2236   }
2237   return 0;
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 // Blank page.
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
```



```
2350 // Blank page.
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
```

```
2400 // Segments in proc->gdt.
2401 #define NSEGS     7
2402
2403 // Per-CPU state
2404 struct cpu {
2405   uchar id;                    // Local APIC ID; index into cpus[] below
2406   struct context *scheduler;   // swtch() here to enter scheduler
2407   struct taskstate ts;         // Used by x86 to find stack for interrupt
2408   struct segdesc gdt[NSEGS];   // x86 global descriptor table
2409   volatile uint started;       // Has the CPU started?
2410   int ncli;                    // Depth of pushcli nesting.
2411   int intena;                  // Were interrupts enabled before pushcli?
2412
2413   // Cpu-local storage variables; see below
2414   struct cpu *cpu;
2415   struct proc *proc;           // The currently-running process.
2416 };
2417
2418 extern struct cpu cpus[NCPU];
2419 extern int ncpu;
2420
2421 // Per-CPU variables, holding pointers to the
2422 // current cpu and to the current process.
2423 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2424 // and "%gs:4" to refer to proc.  seginit sets up the
2425 // %gs segment register so that %gs refers to the memory
2426 // holding those two variables in the local cpu's struct cpu.
2427 // This is similar to how thread-local variables are implemented
2428 // in thread libraries such as Linux pthreads.
2429 extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
2430 extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc
2431
2432
2433 // Saved registers for kernel context switches.
2434 // Don't need to save all the segment registers (%cs, etc),
2435 // because they are constant across kernel contexts.
2436 // Don't need to save %eax, %ecx, %edx, because the
2437 // x86 convention is that the caller has saved them.
2438 // Contexts are stored at the bottom of the stack they
2439 // describe; the stack pointer is the address of the context.
2440 // The layout of the context matches the layout of the stack in swtch.S
2441 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2442 // but it is on the stack and allocproc() manipulates it.
2443 struct context {
2444   uint edi;
2445   uint esi;
2446   uint ebx;
2447   uint ebp;
2448   uint eip;
2449 };
```



```
2450 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2451
2452 // Per-process state
2453 struct proc {
2454   uint sz;                     // Size of process memory (bytes)
2455   pde_t* pgdir;                // Page table
2456   char *kstack;                // Bottom of kernel stack for this process
2457   enum procstate state;        // Process state
2458   int pid;                     // Process ID
2459   struct proc *parent;         // Parent process
2460   struct trapframe *tf;        // Trap frame for current syscall
2461   struct context *context;     // swtch() here to run process
2462   void *chan;                  // If non-zero, sleeping on chan
2463   int killed;                  // If non-zero, have been killed
2464   struct file *ofile[NOFILE];  // Open files
2465   struct inode *cwd;           // Current directory
2466   char name[16];               // Process name (debugging)
2467 };
2468
2469 // Process memory is laid out contiguously, low addresses first:
2470 //   text
2471 //   original data and bss
2472 //   fixed-size stack
2473 //   expandable heap
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
```

```
2500 #include "types.h"
2501 #include "defs.h"
2502 #include "param.h"
2503 #include "memlayout.h"
2504 #include "mmu.h"
2505 #include "x86.h"
2506 #include "proc.h"
2507 #include "spinlock.h"
2508
2509 struct {
2510   struct spinlock lock;
2511   struct proc proc[NPROC];
2512 } ptable;
2513
2514 static struct proc *initproc;
2515
2516 int nextpid = 1;
2517 extern void forkret(void);
2518 extern void trapret(void);
2519
2520 static void wakeup1(void *chan);
2521
2522 void
2523 pinit(void)
2524 {
2525   initlock(&ptable.lock, "ptable");
2526 }
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
```



```
2550 // Look in the process table for an UNUSED proc.
2551 // If found, change state to EMBRYO and initialize
2552 // state required to run in the kernel.
2553 // Otherwise return 0.
2554 static struct proc*
2555 allocproc(void)
2556 {
2557   struct proc *p;
2558   char *sp;
2559
2560   acquire(&ptable.lock);
2561   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2562     if(p->state == UNUSED)
2563       goto found;
2564   release(&ptable.lock);
2565   return 0;
2566
2567 found:
2568   p->state = EMBRYO;
2569   p->pid = nextpid++;
2570   release(&ptable.lock);
2571
2572   // Allocate kernel stack.
2573   if((p->kstack = kalloc()) == 0){
2574     p->state = UNUSED;
2575     return 0;
2576   }
2577   sp = p->kstack + KSTACKSIZE;
2578
2579   // Leave room for trap frame.
2580   sp -= sizeof *p->tf;
2581   p->tf = (struct trapframe*)sp;
2582
2583   // Set up new context to start executing at forkret,
2584   // which returns to trapret.
2585   sp -= 4;
2586   *(uint*)sp = (uint)trapret;
2587
2588   sp -= sizeof *p->context;
2589   p->context = (struct context*)sp;
2590   memset(p->context, 0, sizeof *p->context);
2591   p->context->eip = (uint)forkret;
2592
2593   return p;
2594 }
2595
2596
2597
2598
2599
```

```
2600 // Set up first user process.
2601 void
2602 userinit(void)
2603 {
2604   struct proc *p;
2605   extern char _binary_initcode_start[], _binary_initcode_size[];
2606
2607   p = allocproc();
2608   initproc = p;
2609   if((p->pgdir = setupkvm()) == 0)
2610     panic("userinit: out of memory?");
2611   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2612   p->sz = PGSIZE;
2613   memset(p->tf, 0, sizeof(*p->tf));
2614   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2615   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2616   p->tf->es = p->tf->ds;
2617   p->tf->ss = p->tf->ds;
2618   p->tf->eflags = FL_IF;
2619   p->tf->esp = PGSIZE;
2620   p->tf->eip = 0;  // beginning of initcode.S
2621
2622   safestrcpy(p->name, "initcode", sizeof(p->name));
2623   p->cwd = namei("/");
2624
2625   p->state = RUNNABLE;
2626 }
2627
2628 // Grow current process's memory by n bytes.
2629 // Return 0 on success, -1 on failure.
2630 int
2631 growproc(int n)
2632 {
2633   uint sz;
2634
2635   sz = proc->sz;
2636   if(n > 0){
2637     if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2638       return -1;
2639   } else if(n < 0){
2640     if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2641       return -1;
2642   }
2643   proc->sz = sz;
2644   switchuvm(proc);
2645   return 0;
2646 }
2647
2648
2649
```



```
2650 // Create a new process copying p as the parent.
2651 // Sets up stack to return as if from system call.
2652 // Caller must set state of returned proc to RUNNABLE.
2653 int
2654 fork(void)
2655 {
2656   int i, pid;
2657   struct proc *np;
2658
2659   // Allocate process.
2660   if((np = allocproc()) == 0)
2661     return -1;
2662
2663   // Copy process state from p.
2664   if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2665     kfree(np->kstack);
2666     np->kstack = 0;
2667     np->state = UNUSED;
2668     return -1;
2669   }
2670   np->sz = proc->sz;
2671   np->parent = proc;
2672   *np->tf = *proc->tf;
2673
2674   // Clear %eax so that fork returns 0 in the child.
2675   np->tf->eax = 0;
2676
2677   for(i = 0; i < NOFILE; i++)
2678     if(proc->ofile[i])
2679       np->ofile[i] = filedup(proc->ofile[i]);
2680   np->cwd = idup(proc->cwd);
2681
2682   safestrcpy(np->name, proc->name, sizeof(proc->name));
2683
2684   pid = np->pid;
2685
2686   // lock to force the compiler to emit the np->state write last.
2687   acquire(&ptable.lock);
2688   np->state = RUNNABLE;
2689   release(&ptable.lock);
2690
2691   return pid;
2692 }
2693
2694
2695
2696
2697
2698
2699
```

```
2700 // Exit the current process.  Does not return.
2701 // An exited process remains in the zombie state
2702 // until its parent calls wait() to find out it exited.
2703 void
2704 exit(void)
2705 {
2706   struct proc *p;
2707   int fd;
2708
2709   if(proc == initproc)
2710     panic("init exiting");
2711
2712   // Close all open files.
2713   for(fd = 0; fd < NOFILE; fd++){
2714     if(proc->ofile[fd]){
2715       fileclose(proc->ofile[fd]);
2716       proc->ofile[fd] = 0;
2717     }
2718   }
2719
2720   begin_op();
2721   iput(proc->cwd);
2722   end_op();
2723   proc->cwd = 0;
2724
2725   acquire(&ptable.lock);
2726
2727   // Parent might be sleeping in wait().
2728   wakeup1(proc->parent);
2729
2730   // Pass abandoned children to init.
2731   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2732     if(p->parent == proc){
2733       p->parent = initproc;
2734       if(p->state == ZOMBIE)
2735         wakeup1(initproc);
2736     }
2737   }
2738
2739   // Jump into the scheduler, never to return.
2740   proc->state = ZOMBIE;
2741   sched();
2742   panic("zombie exit");
2743 }
2744
2745
2746
2747
2748
2749
```



```
2750 // Wait for a child process to exit and return its pid.
2751 // Return -1 if this process has no children.
2752 int
2753 wait(void)
2754 {
2755   struct proc *p;
2756   int havekids, pid;
2757
2758   acquire(&ptable.lock);
2759   for(;;){
2760     // Scan through table looking for zombie children.
2761     havekids = 0;
2762     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2763       if(p->parent != proc)
2764         continue;
2765       havekids = 1;
2766       if(p->state == ZOMBIE){
2767         // Found one.
2768         pid = p->pid;
2769         kfree(p->kstack);
2770         p->kstack = 0;
2771         freevm(p->pgdir);
2772         p->state = UNUSED;
2773         p->pid = 0;
2774         p->parent = 0;
2775         p->name[0] = 0;
2776         p->killed = 0;
2777         release(&ptable.lock);
2778         return pid;
2779       }
2780     }
2781
2782     // No point waiting if we don't have any children.
2783     if(!havekids || proc->killed){
2784       release(&ptable.lock);
2785       return -1;
2786     }
2787
2788     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2789     sleep(proc, &ptable.lock);
2790   }
2791 }
2792
2793
2794
2795
2796
2797
2798
2799
```

```
2800 // Per-CPU process scheduler.
2801 // Each CPU calls scheduler() after setting itself up.
2802 // Scheduler never returns.  It loops, doing:
2803 //  - choose a process to run
2804 //  - swtch to start running that process
2805 //  - eventually that process transfers control
2806 //      via swtch back to the scheduler.
2807 void
2808 scheduler(void)
2809 {
2810   struct proc *p;
2811
2812   for(;;){
2813     // Enable interrupts on this processor.
2814     sti();
2815
2816     // Loop over process table looking for process to run.
2817     acquire(&ptable.lock);
2818     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2819       if(p->state != RUNNABLE)
2820         continue;
2821
2822       // Switch to chosen process.  It is the process's job
2823       // to release ptable.lock and then reacquire it
2824       // before jumping back to us.
2825       proc = p;
2826       switchuvm(p);
2827       p->state = RUNNING;
2828       swtch(&cpu->scheduler, proc->context);
2829       switchkvm();
2830
2831       // Process is done running for now.
2832       // It should have changed its p->state before coming back.
2833       proc = 0;
2834     }
2835     release(&ptable.lock);
2836
2837   }
2838 }
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
```



```
2850 // Enter scheduler.  Must hold only ptable.lock
2851 // and have changed proc->state.
2852 void
2853 sched(void)
2854 {
2855   int intena;
2856
2857   if(!holding(&ptable.lock))
2858     panic("sched ptable.lock");
2859   if(cpu->ncli != 1)
2860     panic("sched locks");
2861   if(proc->state == RUNNING)
2862     panic("sched running");
2863   if(readeflags()&FL_IF)
2864     panic("sched interruptible");
2865   intena = cpu->intena;
2866   swtch(&proc->context, cpu->scheduler);
2867   cpu->intena = intena;
2868 }
2869
2870 // Give up the CPU for one scheduling round.
2871 void
2872 yield(void)
2873 {
2874   acquire(&ptable.lock);
2875   proc->state = RUNNABLE;
2876   sched();
2877   release(&ptable.lock);
2878 }
2879
2880 // A fork child's very first scheduling by scheduler()
2881 // will swtch here.  "Return" to user space.
2882 void
2883 forkret(void)
2884 {
2885   static int first = 1;
2886   // Still holding ptable.lock from scheduler.
2887   release(&ptable.lock);
2888
2889   if (first) {
2890     // Some initialization functions must be run in the context
2891     // of a regular process (e.g., they call sleep), and thus cannot
2892     // be run from main().
2893     first = 0;
2894     iinit(ROOTDEV);
2895     initlog(ROOTDEV);
2896   }
2897
2898   // Return to "caller", actually trapret (see allocproc).
2899 }
```

```
2900 // Atomically release lock and sleep on chan.
2901 // Reacquires lock when awakened.
2902 void
2903 sleep(void *chan, struct spinlock *lk)
2904 {
2905   if(proc == 0)
2906     panic("sleep");
2907
2908   if(lk == 0)
2909     panic("sleep without lk");
2910
2911   // Must acquire ptable.lock in order to
2912   // change p->state and then call sched.
2913   // Once we hold ptable.lock, we can be
2914   // guaranteed that we won't miss any wakeup
2915   // (wakeup runs with ptable.lock locked),
2916   // so it's okay to release lk.
2917   if(lk != &ptable.lock){
2918     acquire(&ptable.lock);
2919     release(lk);
2920   }
2921
2922   // Go to sleep.
2923   proc->chan = chan;
2924   proc->state = SLEEPING;
2925   sched();
2926
2927   // Tidy up.
2928   proc->chan = 0;
2929
2930   // Reacquire original lock.
2931   if(lk != &ptable.lock){
2932     release(&ptable.lock);
2933     acquire(lk);
2934   }
2935 }
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
```



```
2950 // Wake up all processes sleeping on chan.
2951 // The ptable lock must be held.
2952 static void
2953 wakeup1(void *chan)
2954 {
2955   struct proc *p;
2956
2957   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2958     if(p->state == SLEEPING && p->chan == chan)
2959       p->state = RUNNABLE;
2960 }
2961
2962 // Wake up all processes sleeping on chan.
2963 void
2964 wakeup(void *chan)
2965 {
2966   acquire(&ptable.lock);
2967   wakeup1(chan);
2968   release(&ptable.lock);
2969 }
2970
2971 // Kill the process with the given pid.
2972 // Process won't exit until it returns
2973 // to user space (see trap in trap.c).
2974 int
2975 kill(int pid)
2976 {
2977   struct proc *p;
2978
2979   acquire(&ptable.lock);
2980   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2981     if(p->pid == pid){
2982       p->killed = 1;
2983       // Wake process from sleep if necessary.
2984       if(p->state == SLEEPING)
2985         p->state = RUNNABLE;
2986       release(&ptable.lock);
2987       return 0;
2988     }
2989   }
2990   release(&ptable.lock);
2991   return -1;
2992 }
2993
2994
2995
2996
2997
2998
2999
```

```
3000 // Print a process listing to console.  For debugging.
3001 // Runs when user types ^P on console.
3002 // No lock to avoid wedging a stuck machine further.
3003 void
3004 procdump(void)
3005 {
3006   static char *states[] = {
3007   [UNUSED]    "unused",
3008   [EMBRYO]    "embryo",
3009   [SLEEPING]  "sleep ",
3010   [RUNNABLE]  "runble",
3011   [RUNNING]   "run   ",
3012   [ZOMBIE]    "zombie"
3013   };
3014   int i;
3015   struct proc *p;
3016   char *state;
3017   uint pc[10];
3018
3019   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3020     if(p->state == UNUSED)
3021       continue;
3022     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
3023       state = states[p->state];
3024     else
3025       state = "???";
3026     cprintf("%d %s %s", p->pid, state, p->name);
3027     if(p->state == SLEEPING){
3028       getcallerpcs((uint*)p->context->ebp+2, pc);
3029       for(i=0; i<10 && pc[i] != 0; i++)
3030         cprintf(" %p", pc[i]);
3031     }
3032     cprintf("\n");
3033   }
3034 }
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
```



```
3050 # Context switch
3051 #
3052 #   void swtch(struct context **old, struct context *new);
3053 #
3054 # Save current register context in old
3055 # and then load register context from new.
3056
3057 .globl swtch
3058 swtch:
3059   movl 4(%esp), %eax
3060   movl 8(%esp), %edx
3061
3062   # Save old callee-save registers
3063   pushl %ebp
3064   pushl %ebx
3065   pushl %esi
3066   pushl %edi
3067
3068   # Switch stacks
3069   movl %esp, (%eax)
3070   movl %edx, %esp
3071
3072   # Load new callee-save registers
3073   popl %edi
3074   popl %esi
3075   popl %ebx
3076   popl %ebp
3077   ret
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
```

```
3100 // Physical memory allocator, intended to allocate
3101 // memory for user processes, kernel stacks, page table pages,
3102 // and pipe buffers. Allocates 4096-byte pages.
3103
3104 #include "types.h"
3105 #include "defs.h"
3106 #include "param.h"
3107 #include "memlayout.h"
3108 #include "mmu.h"
3109 #include "spinlock.h"
3110
3111 void freerange(void *vstart, void *vend);
3112 extern char end[]; // first address after kernel loaded from ELF file
3113
3114 struct run {
3115   struct run *next;
3116 };
3117
3118 struct {
3119   struct spinlock lock;
3120   int use_lock;
3121   struct run *freelist;
3122 } kmem;
3123
3124 // Initialization happens in two phases.
3125 // 1. main() calls kinit1() while still using entrypgdir to place just
3126 // the pages mapped by entrypgdir on free list.
3127 // 2. main() calls kinit2() with the rest of the physical pages
3128 // after installing a full page table that maps them on all cores.
3129 void
3130 kinit1(void *vstart, void *vend)
3131 {
3132   initlock(&kmem.lock, "kmem");
3133   kmem.use_lock = 0;
3134   freerange(vstart, vend);
3135 }
3136
3137 void
3138 kinit2(void *vstart, void *vend)
3139 {
3140   freerange(vstart, vend);
3141   kmem.use_lock = 1;
3142 }
3143
3144
3145
3146
3147
3148
3149
```



```
3150 void
3151 freerange(void *vstart, void *vend)
3152 {
3153   char *p;
3154   p = (char*)PGROUNDUP((uint)vstart);
3155   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3156     kfree(p);
3157 }
3158
3159
3160 // Free the page of physical memory pointed at by v,
3161 // which normally should have been returned by a
3162 // call to kalloc().  (The exception is when
3163 // initializing the allocator; see kinit above.)
3164 void
3165 kfree(char *v)
3166 {
3167   struct run *r;
3168
3169   if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
3170     panic("kfree");
3171
3172   // Fill with junk to catch dangling refs.
3173   memset(v, 1, PGSIZE);
3174
3175   if(kmem.use_lock)
3176     acquire(&kmem.lock);
3177   r = (struct run*)v;
3178   r->next = kmem.freelist;
3179   kmem.freelist = r;
3180   if(kmem.use_lock)
3181     release(&kmem.lock);
3182 }
3183
3184 // Allocate one 4096-byte page of physical memory.
3185 // Returns a pointer that the kernel can use.
3186 // Returns 0 if the memory cannot be allocated.
3187 char*
3188 kalloc(void)
3189 {
3190   struct run *r;
3191
3192   if(kmem.use_lock)
3193     acquire(&kmem.lock);
3194   r = kmem.freelist;
3195   if(r)
3196     kmem.freelist = r->next;
3197   if(kmem.use_lock)
3198     release(&kmem.lock);
3199   return (char*)r;
```

```
3200 }
3201
3202
3203
3204
3205
3206
3207
3208
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
```



```
3250 // x86 trap and interrupt constants.
3251
3252 // Processor-defined:
3253 #define T_DIVIDE         0      // divide error
3254 #define T_DEBUG          1      // debug exception
3255 #define T_NMI            2      // non-maskable interrupt
3256 #define T_BRKPT          3      // breakpoint
3257 #define T_OFLOW          4      // overflow
3258 #define T_BOUND          5      // bounds check
3259 #define T_ILLOP          6      // illegal opcode
3260 #define T_DEVICE         7      // device not available
3261 #define T_DBLFLT         8      // double fault
3262 // #define T_COPROC      9      // reserved (not used since 486)
3263 #define T_TSS           10      // invalid task switch segment
3264 #define T_SEGNP         11      // segment not present
3265 #define T_STACK         12      // stack exception
3266 #define T_GPFLT         13      // general protection fault
3267 #define T_PGFLT         14      // page fault
3268 // #define T_RES        15      // reserved
3269 #define T_FPERR         16      // floating point error
3270 #define T_ALIGN         17      // aligment check
3271 #define T_MCHK          18      // machine check
3272 #define T_SIMDERR       19      // SIMD floating point error
3273
3274 // These are arbitrarily chosen, but with care not to overlap
3275 // processor defined exceptions or interrupt vectors.
3276 #define T_SYSCALL       64      // system call
3277 #define T_DEFAULT      500      // catchall
3278
3279 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
3280
3281 #define IRQ_TIMER        0
3282 #define IRQ_KBD          1
3283 #define IRQ_COM1         4
3284 #define IRQ_IDE         14
3285 #define IRQ_ERROR       19
3286 #define IRQ_SPURIOUS    31
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
```

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

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

print "# generated by vectors.pl - do not edit\n";
print "# handlers\n";
print ".globl alltraps\n";
for(my $i = 0; $i < 256; $i++){
    print ".globl vector$i\n";
    print "vector$i:\n";
    if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
        print "  pushl \$0\n";
    }
    print "  pushl \$$i\n";
    print "  jmp alltraps\n";
}

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

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



```
#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
```

```
3400 #include "types.h"
3401 #include "defs.h"
3402 #include "param.h"
3403 #include "memlayout.h"
3404 #include "mmu.h"
3405 #include "proc.h"
3406 #include "x86.h"
3407 #include "traps.h"
3408 #include "spinlock.h"
3409
3410 // Interrupt descriptor table (shared by all CPUs).
3411 struct gatedesc idt[256];
3412 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
3413 struct spinlock tickslock;
3414 uint ticks;
3415
3416 void
3417 tvinit(void)
3418 {
3419   int i;
3420
3421   for(i = 0; i < 256; i++)
3422     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3423   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3424
3425   initlock(&tickslock, "time");
3426 }
3427
3428 void
3429 idtinit(void)
3430 {
3431   lidt(idt, sizeof(idt));
3432 }
3433
3434
3435
3436
3437
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
```



```
3450 void
3451 trap(struct trapframe *tf)
3452 {
3453   if(tf->trapno == T_SYSCALL){
3454     if(proc->killed)
3455       exit();
3456     proc->tf = tf;
3457     syscall();
3458     if(proc->killed)
3459       exit();
3460     return;
3461   }
3462
3463   switch(tf->trapno){
3464   case T_IRQ0 + IRQ_TIMER:
3465     if(cpu->id == 0){
3466       acquire(&tickslock);
3467       ticks++;
3468       wakeup(&ticks);
3469       release(&tickslock);
3470     }
3471     lapiceoi();
3472     break;
3473   case T_IRQ0 + IRQ_IDE:
3474     ideintr();
3475     lapiceoi();
3476     break;
3477   case T_IRQ0 + IRQ_IDE+1:
3478     // Bochs generates spurious IDE1 interrupts.
3479     break;
3480   case T_IRQ0 + IRQ_KBD:
3481     kbdintr();
3482     lapiceoi();
3483     break;
3484   case T_IRQ0 + IRQ_COM1:
3485     uartintr();
3486     lapiceoi();
3487     break;
3488   case T_IRQ0 + 7:
3489   case T_IRQ0 + IRQ_SPURIOUS:
3490     cprintf("cpu%d: spurious interrupt at %x:%x\n",
3491             cpu->id, tf->cs, tf->eip);
3492     lapiceoi();
3493     break;
3494
3495
3496
3497
3498
3499
```

```
3500   default:
3501     if(proc == 0 || (tf->cs&3) == 0){
3502       // In kernel, it must be our mistake.
3503       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3504               tf->trapno, cpu->id, tf->eip, rcr2());
3505       panic("trap");
3506     }
3507     // In user space, assume process misbehaved.
3508     cprintf("pid %d %s: trap %d err %d on cpu %d "
3509             "eip 0x%x addr 0x%x--kill proc\n",
3510             proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
3511             rcr2());
3512     proc->killed = 1;
3513   }
3514
3515   // Force process exit if it has been killed and is in user space.
3516   // (If it is still executing in the kernel, let it keep running
3517   // until it gets to the regular system call return.)
3518   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3519     exit();
3520
3521   // Force process to give up CPU on clock tick.
3522   // If interrupts were on while locks held, would need to check nlock.
3523   if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3524     yield();
3525
3526   // Check if the process has been killed since we yielded
3527   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3528     exit();
3529 }
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
```



```
3550 // System call numbers
3551 #define SYS_fork    1
3552 #define SYS_exit    2
3553 #define SYS_wait    3
3554 #define SYS_pipe    4
3555 #define SYS_read    5
3556 #define SYS_kill    6
3557 #define SYS_exec    7
3558 #define SYS_fstat   8
3559 #define SYS_chdir   9
3560 #define SYS_dup    10
3561 #define SYS_getpid 11
3562 #define SYS_sbrk   12
3563 #define SYS_sleep  13
3564 #define SYS_uptime 14
3565 #define SYS_open   15
3566 #define SYS_write  16
3567 #define SYS_mknod  17
3568 #define SYS_unlink 18
3569 #define SYS_link   19
3570 #define SYS_mkdir  20
3571 #define SYS_close  21
3572 #define SYS_halt   22
3573 #define SYS_date   23
3574
3575
3576
3577
3578
3579
3580
3581
3582
3583
3584
3585
3586
3587
3588
3589
3590
3591
3592
3593
3594
3595
3596
3597
3598
3599
```

```
3600 #include "types.h"
3601 #include "defs.h"
3602 #include "param.h"
3603 #include "memlayout.h"
3604 #include "mmu.h"
3605 #include "proc.h"
3606 #include "x86.h"
3607 #include "syscall.h"
3608
3609 // User code makes a system call with INT T_SYSCALL.
3610 // System call number in %eax.
3611 // Arguments on the stack, from the user call to the C
3612 // library system call function. The saved user %esp points
3613 // to a saved program counter, and then the first argument.
3614
3615 // Fetch the int at addr from the current process.
3616 int
3617 fetchint(uint addr, int *ip)
3618 {
3619   if(addr >= proc->sz || addr+4 > proc->sz)
3620     return -1;
3621   *ip = *(int*)(addr);
3622   return 0;
3623 }
3624
3625 // Fetch the nul-terminated string at addr from the current process.
3626 // Doesn't actually copy the string - just sets *pp to point at it.
3627 // Returns length of string, not including nul.
3628 int
3629 fetchstr(uint addr, char **pp)
3630 {
3631   char *s, *ep;
3632
3633   if(addr >= proc->sz)
3634     return -1;
3635   *pp = (char*)addr;
3636   ep = (char*)proc->sz;
3637   for(s = *pp; s < ep; s++)
3638     if(*s == 0)
3639       return s - *pp;
3640   return -1;
3641 }
3642
3643 // Fetch the nth 32-bit system call argument.
3644 int
3645 argint(int n, int *ip)
3646 {
3647   return fetchint(proc->tf->esp + 4 + 4*n, ip);
3648 }
3649
```



```
3650 // Fetch the nth word-sized system call argument as a pointer
3651 // to a block of memory of size n bytes.  Check that the pointer
3652 // lies within the process address space.
3653 int
3654 argptr(int n, char **pp, int size)
3655 {
3656   int i;
3657
3658   if(argint(n, &i) < 0)
3659     return -1;
3660   if((uint)i >= proc->sz || (uint)i+size > proc->sz)
3661     return -1;
3662   *pp = (char*)i;
3663   return 0;
3664 }
3665
3666 // Fetch the nth word-sized system call argument as a string pointer.
3667 // Check that the pointer is valid and the string is nul-terminated.
3668 // (There is no shared writable memory, so the string can't change
3669 // between this check and being used by the kernel.)
3670 int
3671 argstr(int n, char **pp)
3672 {
3673   int addr;
3674   if(argint(n, &addr) < 0)
3675     return -1;
3676   return fetchstr(addr, pp);
3677 }
3678
3679 extern int sys_chdir(void);
3680 extern int sys_close(void);
3681 extern int sys_dup(void);
3682 extern int sys_exec(void);
3683 extern int sys_exit(void);
3684 extern int sys_fork(void);
3685 extern int sys_fstat(void);
3686 extern int sys_getpid(void);
3687 extern int sys_kill(void);
3688 extern int sys_link(void);
3689 extern int sys_mkdir(void);
3690 extern int sys_mknod(void);
3691 extern int sys_open(void);
3692 extern int sys_pipe(void);
3693 extern int sys_read(void);
3694 extern int sys_sbrk(void);
3695 extern int sys_sleep(void);
3696 extern int sys_unlink(void);
3697 extern int sys_wait(void);
3698 extern int sys_write(void);
3699 extern int sys_uptime(void);
```

```
3700 extern int sys_halt(void);
3701 extern int sys_date(void);
3702
3703 static int (*syscalls[])(void) = {
3704 [SYS_fork]    sys_fork,
3705 [SYS_exit]    sys_exit,
3706 [SYS_wait]    sys_wait,
3707 [SYS_pipe]    sys_pipe,
3708 [SYS_read]    sys_read,
3709 [SYS_kill]    sys_kill,
3710 [SYS_exec]    sys_exec,
3711 [SYS_fstat]   sys_fstat,
3712 [SYS_chdir]   sys_chdir,
3713 [SYS_dup]     sys_dup,
3714 [SYS_getpid]  sys_getpid,
3715 [SYS_sbrk]    sys_sbrk,
3716 [SYS_sleep]   sys_sleep,
3717 [SYS_uptime]  sys_uptime,
3718 [SYS_open]    sys_open,
3719 [SYS_write]   sys_write,
3720 [SYS_mknod]   sys_mknod,
3721 [SYS_unlink]  sys_unlink,
3722 [SYS_link]    sys_link,
3723 [SYS_mkdir]   sys_mkdir,
3724 [SYS_close]   sys_close,
3725 [SYS_halt]    sys_halt,
3726 [SYS_date]    sys_date,
3727 };
3728
3729 char * sysCallNames[] = {
3730 [SYS_fork]    "fork",
3731 [SYS_exit]    "exit",
3732 [SYS_wait]    "wait",
3733 [SYS_pipe]    "pipe",
3734 [SYS_read]    "read",
3735 [SYS_kill]    "kill",
3736 [SYS_exec]    "exec",
3737 [SYS_fstat]   "fstat",
3738 [SYS_chdir]   "chdir",
3739 [SYS_dup]     "dup",
3740 [SYS_getpid]  "getpid",
3741 [SYS_sbrk]    "sbrk",
3742 [SYS_sleep]   "sleep",
3743 [SYS_uptime]  "uptime",
3744 [SYS_open]    "open",
3745 [SYS_write]   "write",
3746 [SYS_mknod]   "mknod",
3747 [SYS_unlink]  "unlink",
3748 [SYS_link]    "link",
3749 [SYS_mkdir]   "mkdir",
```



```
3750 [SYS_close]   "close",
3751 [SYS_halt]    "halt",
3752 [SYS_date]    "date",
3753 };
3754
3755 void
3756 syscall(void)
3757 {
3758   int num;
3759
3760   num = proc->tf->eax;
3761   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3762     proc->tf->eax = syscalls[num]();
3763     //int returnValue = proc->tf->eax;
3764     //Omits printing the 'write' system call (16) for readability
3765     //if(num != 16) { cprintf("%s -> %d\n", sysCallNames[num], returnValue);
3766   } else {
3767     cprintf("%d %s: unknown sys call %d\n",
3768             proc->pid, proc->name, num);
3769     proc->tf->eax = -1;
3770   }
3771 }
3772
3773
3774
3775
3776
3777
3778
3779
3780
3781
3782
3783
3784
3785
3786
3787
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3799
```

```
3800 #include "types.h"
3801 #include "x86.h"
3802 #include "defs.h"
3803 #include "date.h"
3804 #include "param.h"
3805 #include "memlayout.h"
3806 #include "mmu.h"
3807 #include "proc.h"
3808
3809 int
3810 sys_fork(void)
3811 {
3812   return fork();
3813 }
3814
3815 int
3816 sys_exit(void)
3817 {
3818   exit();
3819   return 0;  // not reached
3820 }
3821
3822 int
3823 sys_wait(void)
3824 {
3825   return wait();
3826 }
3827
3828 int
3829 sys_kill(void)
3830 {
3831   int pid;
3832
3833   if(argint(0, &pid) < 0)
3834     return -1;
3835   return kill(pid);
3836 }
3837
3838 int
3839 sys_getpid(void)
3840 {
3841   return proc->pid;
3842 }
3843
3844
3845
3846
3847
3848
3849
```



```
3850 int
3851 sys_sbrk(void)
3852 {
3853   int addr;
3854   int n;
3855
3856   if(argint(0, &n) < 0)
3857     return -1;
3858   addr = proc->sz;
3859   if(growproc(n) < 0)
3860     return -1;
3861   return addr;
3862 }
3863
3864 int
3865 sys_sleep(void)
3866 {
3867   int n;
3868   uint ticks0;
3869
3870   if(argint(0, &n) < 0)
3871     return -1;
3872   acquire(&tickslock);
3873   ticks0 = ticks;
3874   while(ticks - ticks0 < n){
3875     if(proc->killed){
3876       release(&tickslock);
3877       return -1;
3878     }
3879     sleep(&ticks, &tickslock);
3880   }
3881   release(&tickslock);
3882   return 0;
3883 }
3884
3885 // return how many clock tick interrupts have occurred
3886 // since start.
3887 int
3888 sys_uptime(void)
3889 {
3890   uint xticks;
3891
3892   acquire(&tickslock);
3893   xticks = ticks;
3894   release(&tickslock);
3895   return xticks;
3896 }
3897
3898
3899
```

```
3900 //Turn of the computer
3901 int sys_halt(void){
3902   cprintf("Shutting down ...\n");
3903   outw (0xB004, 0x0 | 0x2000);
3904   return 0;
3905 }
3906
3907 int
3908 sys_date(void)
3909 {
3910     struct rtcdate * d;
3911
3912     if(argptr(0, (void*)&d, sizeof(*d)) < 0)
3913    return -1;
3914
3915    cmostime(d);
3916
3917    return 0;
3918 }
3919
3920
3921
3922
3923
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 struct buf {
3951   int flags;
3952   uint dev;
3953   uint blockno;
3954   struct buf *prev; // LRU cache list
3955   struct buf *next;
3956   struct buf *qnext; // disk queue
3957   uchar data[BSIZE];
3958 };
3959 #define B_BUSY  0x1  // buffer is locked by some process
3960 #define B_VALID 0x2  // buffer has been read from disk
3961 #define B_DIRTY 0x4  // buffer needs to be written to disk
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
```

```
4000 #define O_RDONLY  0x000
4001 #define O_WRONLY  0x001
4002 #define O_RDWR    0x002
4003 #define O_CREATE  0x200
4004
4005
4006
4007
4008
4009
4010
4011
4012
4013
4014
4015
4016
4017
4018
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
```



```
4050 #define T_DIR  1   // Directory
4051 #define T_FILE 2   // File
4052 #define T_DEV  3   // Device
4053
4054 struct stat {
4055   short type;  // Type of file
4056   int dev;     // File system's disk device
4057   uint ino;    // Inode number
4058   short nlink; // Number of links to file
4059   uint size;   // Size of file in bytes
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 // On-disk file system format.
4101 // Both the kernel and user programs use this header file.
4102
4103
4104 #define ROOTINO 1  // root i-number
4105 #define BSIZE 512  // block size
4106
4107 // Disk layout:
4108 // [ boot block | super block | log | inode blocks | free bit map | data blo
4109 //
4110 // mkfs computes the super block and builds an initial file system. The supe
4111 // the disk layout:
4112 struct superblock {
4113   uint size;         // Size of file system image (blocks)
4114   uint nblocks;      // Number of data blocks
4115   uint ninodes;      // Number of inodes.
4116   uint nlog;         // Number of log blocks
4117   uint logstart;     // Block number of first log block
4118   uint inodestart;   // Block number of first inode block
4119   uint bmapstart;    // Block number of first free map block
4120 };
4121
4122 #define NDIRECT 12
4123 #define NINDIRECT (BSIZE / sizeof(uint))
4124 #define MAXFILE (NDIRECT + NINDIRECT)
4125
4126 // On-disk inode structure
4127 struct dinode {
4128   short type;           // File type
4129   short major;          // Major device number (T_DEV only)
4130   short minor;          // Minor device number (T_DEV only)
4131   short nlink;          // Number of links to inode in file system
4132   uint size;            // Size of file (bytes)
4133   uint addrs[NDIRECT+1];   // Data block addresses
4134 };
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```



```
4150 // Inodes per block.
4151 #define IPB           (BSIZE / sizeof(struct dinode))
4152
4153 // Block containing inode i
4154 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
4155
4156 // Bitmap bits per block
4157 #define BPB           (BSIZE*8)
4158
4159 // Block of free map containing bit for block b
4160 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
4161
4162 // Directory is a file containing a sequence of dirent structures.
4163 #define DIRSIZ 14
4164
4165 struct dirent {
4166   ushort inum;
4167   char name[DIRSIZ];
4168 };
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
```

```
4200 struct file {
4201   enum { FD_NONE, FD_PIPE, FD_INODE } type;
4202   int ref; // reference count
4203   char readable;
4204   char writable;
4205   struct pipe *pipe;
4206   struct inode *ip;
4207   uint off;
4208 };
4209
4210
4211 // in-memory copy of an inode
4212 struct inode {
4213   uint dev;           // Device number
4214   uint inum;          // Inode number
4215   int ref;            // Reference count
4216   int flags;          // I_BUSY, I_VALID
4217
4218   short type;         // copy of disk inode
4219   short major;
4220   short minor;
4221   short nlink;
4222   uint size;
4223   uint addrs[NDIRECT+1];
4224 };
4225 #define I_BUSY 0x1
4226 #define I_VALID 0x2
4227
4228 // table mapping major device number to
4229 // device functions
4230 struct devsw {
4231   int (*read)(struct inode*, char*, int);
4232   int (*write)(struct inode*, char*, int);
4233 };
4234
4235 extern struct devsw devsw[];
4236
4237 #define CONSOLE 1
4238
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
```



```
4250 // Blank page.
4251
4252
4253
4254
4255
4256
4257
4258
4259
4260
4261
4262
4263
4264
4265
4266
4267
4268
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
```

```
4300 // Simple PIO-based (non-DMA) IDE driver code.
4301
4302 #include "types.h"
4303 #include "defs.h"
4304 #include "param.h"
4305 #include "memlayout.h"
4306 #include "mmu.h"
4307 #include "proc.h"
4308 #include "x86.h"
4309 #include "traps.h"
4310 #include "spinlock.h"
4311 #include "fs.h"
4312 #include "buf.h"
4313
4314 #define SECTOR_SIZE   512
4315 #define IDE_BSY       0x80
4316 #define IDE_DRDY      0x40
4317 #define IDE_DF        0x20
4318 #define IDE_ERR       0x01
4319
4320 #define IDE_CMD_READ  0x20
4321 #define IDE_CMD_WRITE 0x30
4322
4323 // idequeue points to the buf now being read/written to the disk.
4324 // idequeue->qnext points to the next buf to be processed.
4325 // You must hold idelock while manipulating queue.
4326
4327 static struct spinlock idelock;
4328 static struct buf *idequeue;
4329
4330 static int havedisk1;
4331 static void idestart(struct buf*);
4332
4333 // Wait for IDE disk to become ready.
4334 static int
4335 idewait(int checkerr)
4336 {
4337   int r;
4338
4339   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4340     ;
4341   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4342     return -1;
4343   return 0;
4344 }
4345
4346
4347
4348
4349
```



```
4350 void
4351 ideinit(void)
4352 {
4353   int i;
4354
4355   initlock(&idelock, "ide");
4356   picenable(IRQ_IDE);
4357   ioapicenable(IRQ_IDE, ncpu - 1);
4358   idewait(0);
4359
4360   // Check if disk 1 is present
4361   outb(0x1f6, 0xe0 | (1<<4));
4362   for(i=0; i<1000; i++){
4363     if(inb(0x1f7) != 0){
4364       havedisk1 = 1;
4365       break;
4366     }
4367   }
4368
4369   // Switch back to disk 0.
4370   outb(0x1f6, 0xe0 | (0<<4));
4371 }
4372
4373 // Start the request for b.  Caller must hold idelock.
4374 static void
4375 idestart(struct buf *b)
4376 {
4377   if(b == 0)
4378     panic("idestart");
4379   if(b->blockno >= FSSIZE)
4380     panic("incorrect blockno");
4381   int sector_per_block =  BSIZE/SECTOR_SIZE;
4382   int sector = b->blockno * sector_per_block;
4383
4384   if (sector_per_block > 7) panic("idestart");
4385
4386   idewait(0);
4387   outb(0x3f6, 0);  // generate interrupt
4388   outb(0x1f2, sector_per_block);  // number of sectors
4389   outb(0x1f3, sector & 0xff);
4390   outb(0x1f4, (sector >> 8) & 0xff);
4391   outb(0x1f5, (sector >> 16) & 0xff);
4392   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4393   if(b->flags & B_DIRTY){
4394     outb(0x1f7, IDE_CMD_WRITE);
4395     outsl(0x1f0, b->data, BSIZE/4);
4396   } else {
4397     outb(0x1f7, IDE_CMD_READ);
4398   }
4399 }
```

```
4400 // Interrupt handler.
4401 void
4402 ideintr(void)
4403 {
4404   struct buf *b;
4405
4406   // First queued buffer is the active request.
4407   acquire(&idelock);
4408   if((b = idequeue) == 0){
4409     release(&idelock);
4410     // cprintf("spurious IDE interrupt\n");
4411     return;
4412   }
4413   idequeue = b->qnext;
4414
4415   // Read data if needed.
4416   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4417     insl(0x1f0, b->data, BSIZE/4);
4418
4419   // Wake process waiting for this buf.
4420   b->flags |= B_VALID;
4421   b->flags &= ~B_DIRTY;
4422   wakeup(b);
4423
4424   // Start disk on next buf in queue.
4425   if(idequeue != 0)
4426     idestart(idequeue);
4427
4428   release(&idelock);
4429 }
4430
4431
4432
4433
4434
4435
4436
4437
4438
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
```



```
4450 // Sync buf with disk.
4451 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
4452 // Else if B_VALID is not set, read buf from disk, set B_VALID.
4453 void
4454 iderw(struct buf *b)
4455 {
4456   struct buf **pp;
4457
4458   if(!(b->flags & B_BUSY))
4459     panic("iderw: buf not busy");
4460   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
4461     panic("iderw: nothing to do");
4462   if(b->dev != 0 && !havedisk1)
4463     panic("iderw: ide disk 1 not present");
4464
4465   acquire(&idelock);
4466
4467   // Append b to idequeue.
4468   b->qnext = 0;
4469   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
4470     ;
4471   *pp = b;
4472
4473   // Start disk if necessary.
4474   if(idequeue == b)
4475     idestart(b);
4476
4477   // Wait for request to finish.
4478   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
4479     sleep(b, &idelock);
4480   }
4481
4482   release(&idelock);
4483 }
4484
4485
4486
4487
4488
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499
```

```
4500 // Buffer cache.
4501 //
4502 // The buffer cache is a linked list of buf structures holding
4503 // cached copies of disk block contents.  Caching disk blocks
4504 // in memory reduces the number of disk reads and also provides
4505 // a synchronization point for disk blocks used by multiple processes.
4506 //
4507 // Interface:
4508 // * To get a buffer for a particular disk block, call bread.
4509 // * After changing buffer data, call bwrite to write it to disk.
4510 // * When done with the buffer, call brelse.
4511 // * Do not use the buffer after calling brelse.
4512 // * Only one process at a time can use a buffer,
4513 //     so do not keep them longer than necessary.
4514 //
4515 // The implementation uses three state flags internally:
4516 // * B_BUSY: the block has been returned from bread
4517 //     and has not been passed back to brelse.
4518 // * B_VALID: the buffer data has been read from the disk.
4519 // * B_DIRTY: the buffer data has been modified
4520 //     and needs to be written to disk.
4521
4522 #include "types.h"
4523 #include "defs.h"
4524 #include "param.h"
4525 #include "spinlock.h"
4526 #include "fs.h"
4527 #include "buf.h"
4528
4529 struct {
4530   struct spinlock lock;
4531   struct buf buf[NBUF];
4532
4533   // Linked list of all buffers, through prev/next.
4534   // head.next is most recently used.
4535   struct buf head;
4536 } bcache;
4537
4538 void
4539 binit(void)
4540 {
4541   struct buf *b;
4542
4543   initlock(&bcache.lock, "bcache");
4544
4545
4546
4547
4548
4549
```



```
4550   // Create linked list of buffers
4551   bcache.head.prev = &bcache.head;
4552   bcache.head.next = &bcache.head;
4553   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4554     b->next = bcache.head.next;
4555     b->prev = &bcache.head;
4556     b->dev = -1;
4557     bcache.head.next->prev = b;
4558     bcache.head.next = b;
4559   }
4560 }
4561
4562 // Look through buffer cache for block on device dev.
4563 // If not found, allocate a buffer.
4564 // In either case, return B_BUSY buffer.
4565 static struct buf*
4566 bget(uint dev, uint blockno)
4567 {
4568   struct buf *b;
4569
4570   acquire(&bcache.lock);
4571
4572  loop:
4573   // Is the block already cached?
4574   for(b = bcache.head.next; b != &bcache.head; b = b->next){
4575     if(b->dev == dev && b->blockno == blockno){
4576       if(!(b->flags & B_BUSY)){
4577         b->flags |= B_BUSY;
4578         release(&bcache.lock);
4579         return b;
4580       }
4581       sleep(b, &bcache.lock);
4582       goto loop;
4583     }
4584   }
4585
4586   // Not cached; recycle some non-busy and clean buffer.
4587   // "clean" because B_DIRTY and !B_BUSY means log.c
4588   // hasn't yet committed the changes to the buffer.
4589   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4590     if((b->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
4591       b->dev = dev;
4592       b->blockno = blockno;
4593       b->flags = B_BUSY;
4594       release(&bcache.lock);
4595       return b;
4596     }
4597   }
4598   panic("bget: no buffers");
4599 }
```

```
4600 // Return a B_BUSY buf with the contents of the indicated block.
4601 struct buf*
4602 bread(uint dev, uint blockno)
4603 {
4604   struct buf *b;
4605
4606   b = bget(dev, blockno);
4607   if(!(b->flags & B_VALID)) {
4608     iderw(b);
4609   }
4610   return b;
4611 }
4612
4613 // Write b's contents to disk.  Must be B_BUSY.
4614 void
4615 bwrite(struct buf *b)
4616 {
4617   if((b->flags & B_BUSY) == 0)
4618     panic("bwrite");
4619   b->flags |= B_DIRTY;
4620   iderw(b);
4621 }
4622
4623 // Release a B_BUSY buffer.
4624 // Move to the head of the MRU list.
4625 void
4626 brelse(struct buf *b)
4627 {
4628   if((b->flags & B_BUSY) == 0)
4629     panic("brelse");
4630
4631   acquire(&bcache.lock);
4632
4633   b->next->prev = b->prev;
4634   b->prev->next = b->next;
4635   b->next = bcache.head.next;
4636   b->prev = &bcache.head;
4637   bcache.head.next->prev = b;
4638   bcache.head.next = b;
4639
4640   b->flags &= ~B_BUSY;
4641   wakeup(b);
4642
4643   release(&bcache.lock);
4644 }
4645
4646
4647
4648
4649
```



```
4650 // Blank page.
4651
4652
4653
4654
4655
4656
4657
4658
4659
4660
4661
4662
4663
4664
4665
4666
4667
4668
4669
4670
4671
4672
4673
4674
4675
4676
4677
4678
4679
4680
4681
4682
4683
4684
4685
4686
4687
4688
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4699
```

```
4700 #include "types.h"
4701 #include "defs.h"
4702 #include "param.h"
4703 #include "spinlock.h"
4704 #include "fs.h"
4705 #include "buf.h"
4706
4707 // Simple logging that allows concurrent FS system calls.
4708 //
4709 // A log transaction contains the updates of multiple FS system
4710 // calls. The logging system only commits when there are
4711 // no FS system calls active. Thus there is never
4712 // any reasoning required about whether a commit might
4713 // write an uncommitted system call's updates to disk.
4714 //
4715 // A system call should call begin_op()/end_op() to mark
4716 // its start and end. Usually begin_op() just increments
4717 // the count of in-progress FS system calls and returns.
4718 // But if it thinks the log is close to running out, it
4719 // sleeps until the last outstanding end_op() commits.
4720 //
4721 // The log is a physical re-do log containing disk blocks.
4722 // The on-disk log format:
4723 //   header block, containing block #s for block A, B, C, ...
4724 //   block A
4725 //   block B
4726 //   block C
4727 //   ...
4728 // Log appends are synchronous.
4729
4730 // Contents of the header block, used for both the on-disk header block
4731 // and to keep track in memory of logged block# before commit.
4732 struct logheader {
4733   int n;
4734   int block[LOGSIZE];
4735 };
4736
4737 struct log {
4738   struct spinlock lock;
4739   int start;
4740   int size;
4741   int outstanding; // how many FS sys calls are executing.
4742   int committing;  // in commit(), please wait.
4743   int dev;
4744   struct logheader lh;
4745 };
4746
4747
4748
4749
```



```
4750 struct log log;
4751
4752 static void recover_from_log(void);
4753 static void commit();
4754
4755 void
4756 initlog(int dev)
4757 {
4758   if (sizeof(struct logheader) >= BSIZE)
4759     panic("initlog: too big logheader");
4760
4761   struct superblock sb;
4762   initlock(&log.lock, "log");
4763   readsb(dev, &sb);
4764   log.start = sb.logstart;
4765   log.size = sb.nlog;
4766   log.dev = dev;
4767   recover_from_log();
4768 }
4769
4770 // Copy committed blocks from log to their home location
4771 static void
4772 install_trans(void)
4773 {
4774   int tail;
4775
4776   for (tail = 0; tail < log.lh.n; tail++) {
4777     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
4778     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
4779     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
4780     bwrite(dbuf);  // write dst to disk
4781     brelse(lbuf);
4782     brelse(dbuf);
4783   }
4784 }
4785
4786 // Read the log header from disk into the in-memory log header
4787 static void
4788 read_head(void)
4789 {
4790   struct buf *buf = bread(log.dev, log.start);
4791   struct logheader *lh = (struct logheader *) (buf->data);
4792   int i;
4793   log.lh.n = lh->n;
4794   for (i = 0; i < log.lh.n; i++) {
4795     log.lh.block[i] = lh->block[i];
4796   }
4797   brelse(buf);
4798 }
4799
```

```
4800 // Write in-memory log header to disk.
4801 // This is the true point at which the
4802 // current transaction commits.
4803 static void
4804 write_head(void)
4805 {
4806   struct buf *buf = bread(log.dev, log.start);
4807   struct logheader *hb = (struct logheader *) (buf->data);
4808   int i;
4809   hb->n = log.lh.n;
4810   for (i = 0; i < log.lh.n; i++) {
4811     hb->block[i] = log.lh.block[i];
4812   }
4813   bwrite(buf);
4814   brelse(buf);
4815 }
4816
4817 static void
4818 recover_from_log(void)
4819 {
4820   read_head();
4821   install_trans(); // if committed, copy from log to disk
4822   log.lh.n = 0;
4823   write_head(); // clear the log
4824 }
4825
4826 // called at the start of each FS system call.
4827 void
4828 begin_op(void)
4829 {
4830   acquire(&log.lock);
4831   while(1){
4832     if(log.committing){
4833       sleep(&log, &log.lock);
4834     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
4835       // this op might exhaust log space; wait for commit.
4836       sleep(&log, &log.lock);
4837     } else {
4838       log.outstanding += 1;
4839       release(&log.lock);
4840       break;
4841     }
4842   }
4843 }
4844
4845
4846
4847
4848
4849
```



```
4850 // called at the end of each FS system call.
4851 // commits if this was the last outstanding operation.
4852 void
4853 end_op(void)
4854 {
4855   int do_commit = 0;
4856
4857   acquire(&log.lock);
4858   log.outstanding -= 1;
4859   if(log.committing)
4860     panic("log.committing");
4861   if(log.outstanding == 0){
4862     do_commit = 1;
4863     log.committing = 1;
4864   } else {
4865     // begin_op() may be waiting for log space.
4866     wakeup(&log);
4867   }
4868   release(&log.lock);
4869
4870   if(do_commit){
4871     // call commit w/o holding locks, since not allowed
4872     // to sleep with locks.
4873     commit();
4874     acquire(&log.lock);
4875     log.committing = 0;
4876     wakeup(&log);
4877     release(&log.lock);
4878   }
4879 }
4880
4881 // Copy modified blocks from cache to log.
4882 static void
4883 write_log(void)
4884 {
4885   int tail;
4886
4887   for (tail = 0; tail < log.lh.n; tail++) {
4888     struct buf *to = bread(log.dev, log.start+tail+1); // log block
4889     struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
4890     memmove(to->data, from->data, BSIZE);
4891     bwrite(to);  // write the log
4892     brelse(from);
4893     brelse(to);
4894   }
4895 }
4896
4897
4898
4899
```

```
4900 static void
4901 commit()
4902 {
4903   if (log.lh.n > 0) {
4904     write_log();     // Write modified blocks from cache to log
4905     write_head();    // Write header to disk -- the real commit
4906     install_trans(); // Now install writes to home locations
4907     log.lh.n = 0;
4908     write_head();    // Erase the transaction from the log
4909   }
4910 }
4911
4912 // Caller has modified b->data and is done with the buffer.
4913 // Record the block number and pin in the cache with B_DIRTY.
4914 // commit()/write_log() will do the disk write.
4915 //
4916 // log_write() replaces bwrite(); a typical use is:
4917 //   bp = bread(...)
4918 //   modify bp->data[]
4919 //   log_write(bp)
4920 //   brelse(bp)
4921 void
4922 log_write(struct buf *b)
4923 {
4924   int i;
4925
4926   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
4927     panic("too big a transaction");
4928   if (log.outstanding < 1)
4929     panic("log_write outside of trans");
4930
4931   acquire(&log.lock);
4932   for (i = 0; i < log.lh.n; i++) {
4933     if (log.lh.block[i] == b->blockno)   // log absorbtion
4934       break;
4935   }
4936   log.lh.block[i] = b->blockno;
4937   if (i == log.lh.n)
4938     log.lh.n++;
4939   b->flags |= B_DIRTY; // prevent eviction
4940   release(&log.lock);
4941 }
4942
4943
4944
4945
4946
4947
4948
4949
```



```
4950 // File system implementation.  Five layers:
4951 //   + Blocks: allocator for raw disk blocks.
4952 //   + Log: crash recovery for multi-step updates.
4953 //   + Files: inode allocator, reading, writing, metadata.
4954 //   + Directories: inode with special contents (list of other inodes!)
4955 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
4956 //
4957 // This file contains the low-level file system manipulation
4958 // routines.  The (higher-level) system call implementations
4959 // are in sysfile.c.
4960
4961 #include "types.h"
4962 #include "defs.h"
4963 #include "param.h"
4964 #include "stat.h"
4965 #include "mmu.h"
4966 #include "proc.h"
4967 #include "spinlock.h"
4968 #include "fs.h"
4969 #include "buf.h"
4970 #include "file.h"
4971
4972 #define min(a, b) ((a) < (b) ? (a) : (b))
4973 static void itrunc(struct inode*);
4974 struct superblock sb;   // there should be one per dev, but we run with one d
4975
4976 // Read the super block.
4977 void
4978 readsb(int dev, struct superblock *sb)
4979 {
4980   struct buf *bp;
4981
4982   bp = bread(dev, 1);
4983   memmove(sb, bp->data, sizeof(*sb));
4984   brelse(bp);
4985 }
4986
4987 // Zero a block.
4988 static void
4989 bzero(int dev, int bno)
4990 {
4991   struct buf *bp;
4992
4993   bp = bread(dev, bno);
4994   memset(bp->data, 0, BSIZE);
4995   log_write(bp);
4996   brelse(bp);
4997 }
4998
4999
```

```
5000 // Blocks.
5001
5002 // Allocate a zeroed disk block.
5003 static uint
5004 balloc(uint dev)
5005 {
5006   int b, bi, m;
5007   struct buf *bp;
5008
5009   bp = 0;
5010   for(b = 0; b < sb.size; b += BPB){
5011     bp = bread(dev, BBLOCK(b, sb));
5012     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
5013       m = 1 << (bi % 8);
5014       if((bp->data[bi/8] & m) == 0){  // Is block free?
5015         bp->data[bi/8] |= m;  // Mark block in use.
5016         log_write(bp);
5017         brelse(bp);
5018         bzero(dev, b + bi);
5019         return b + bi;
5020       }
5021     }
5022     brelse(bp);
5023   }
5024   panic("balloc: out of blocks");
5025 }
5026
5027 // Free a disk block.
5028 static void
5029 bfree(int dev, uint b)
5030 {
5031   struct buf *bp;
5032   int bi, m;
5033
5034   readsb(dev, &sb);
5035   bp = bread(dev, BBLOCK(b, sb));
5036   bi = b % BPB;
5037   m = 1 << (bi % 8);
5038   if((bp->data[bi/8] & m) == 0)
5039     panic("freeing free block");
5040   bp->data[bi/8] &= ~m;
5041   log_write(bp);
5042   brelse(bp);
5043 }
5044
5045
5046
5047
5048
5049
```



```
5050 // Inodes.
5051 //
5052 // An inode describes a single unnamed file.
5053 // The inode disk structure holds metadata: the file's type,
5054 // its size, the number of links referring to it, and the
5055 // list of blocks holding the file's content.
5056 //
5057 // The inodes are laid out sequentially on disk at
5058 // sb.startinode. Each inode has a number, indicating its
5059 // position on the disk.
5060 //
5061 // The kernel keeps a cache of in-use inodes in memory
5062 // to provide a place for synchronizing access
5063 // to inodes used by multiple processes. The cached
5064 // inodes include book-keeping information that is
5065 // not stored on disk: ip->ref and ip->flags.
5066 //
5067 // An inode and its in-memory represtative go through a
5068 // sequence of states before they can be used by the
5069 // rest of the file system code.
5070 //
5071 // * Allocation: an inode is allocated if its type (on disk)
5072 //   is non-zero. ialloc() allocates, iput() frees if
5073 //   the link count has fallen to zero.
5074 //
5075 // * Referencing in cache: an entry in the inode cache
5076 //   is free if ip->ref is zero. Otherwise ip->ref tracks
5077 //   the number of in-memory pointers to the entry (open
5078 //   files and current directories). iget() to find or
5079 //   create a cache entry and increment its ref, iput()
5080 //   to decrement ref.
5081 //
5082 // * Valid: the information (type, size, &c) in an inode
5083 //   cache entry is only correct when the I_VALID bit
5084 //   is set in ip->flags. ilock() reads the inode from
5085 //   the disk and sets I_VALID, while iput() clears
5086 //   I_VALID if ip->ref has fallen to zero.
5087 //
5088 // * Locked: file system code may only examine and modify
5089 //   the information in an inode and its content if it
5090 //   has first locked the inode. The I_BUSY flag indicates
5091 //   that the inode is locked. ilock() sets I_BUSY,
5092 //   while iunlock clears it.
5093 //
5094 // Thus a typical sequence is:
5095 //   ip = iget(dev, inum)
5096 //   ilock(ip)
5097 //   ... examine and modify ip->xxx ...
5098 //   iunlock(ip)
5099 //   iput(ip)
```

```
5100 //
5101 // ilock() is separate from iget() so that system calls can
5102 // get a long-term reference to an inode (as for an open file)
5103 // and only lock it for short periods (e.g., in read()).
5104 // The separation also helps avoid deadlock and races during
5105 // pathname lookup. iget() increments ip->ref so that the inode
5106 // stays cached and pointers to it remain valid.
5107 //
5108 // Many internal file system functions expect the caller to
5109 // have locked the inodes involved; this lets callers create
5110 // multi-step atomic operations.
5111
5112 struct {
5113   struct spinlock lock;
5114   struct inode inode[NINODE];
5115 } icache;
5116
5117 void
5118 iinit(int dev)
5119 {
5120   initlock(&icache.lock, "icache");
5121   readsb(dev, &sb);
5122   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d inodestart %
5123           sb.nblocks, sb.ninodes, sb.nlog, sb.logstart, sb.inodestart, sb.bma
5124 }
5125
5126 static struct inode* iget(uint dev, uint inum);
5127
5128
5129
5130
5131
5132
5133
5134
5135
5136
5137
5138
5139
5140
5141
5142
5143
5144
5145
5146
5147
5148
5149
```



```
5150 // Allocate a new inode with the given type on device dev.
5151 // A free inode has a type of zero.
5152 struct inode*
5153 ialloc(uint dev, short type)
5154 {
5155   int inum;
5156   struct buf *bp;
5157   struct dinode *dip;
5158
5159   for(inum = 1; inum < sb.ninodes; inum++){
5160     bp = bread(dev, IBLOCK(inum, sb));
5161     dip = (struct dinode*)bp->data + inum%IPB;
5162     if(dip->type == 0){  // a free inode
5163       memset(dip, 0, sizeof(*dip));
5164       dip->type = type;
5165       log_write(bp);   // mark it allocated on the disk
5166       brelse(bp);
5167       return iget(dev, inum);
5168     }
5169     brelse(bp);
5170   }
5171   panic("ialloc: no inodes");
5172 }
5173
5174 // Copy a modified in-memory inode to disk.
5175 void
5176 iupdate(struct inode *ip)
5177 {
5178   struct buf *bp;
5179   struct dinode *dip;
5180
5181   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5182   dip = (struct dinode*)bp->data + ip->inum%IPB;
5183   dip->type = ip->type;
5184   dip->major = ip->major;
5185   dip->minor = ip->minor;
5186   dip->nlink = ip->nlink;
5187   dip->size = ip->size;
5188   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5189   log_write(bp);
5190   brelse(bp);
5191 }
5192
5193
5194
5195
5196
5197
5198
5199
```

```
5200 // Find the inode with number inum on device dev
5201 // and return the in-memory copy. Does not lock
5202 // the inode and does not read it from disk.
5203 static struct inode*
5204 iget(uint dev, uint inum)
5205 {
5206   struct inode *ip, *empty;
5207
5208   acquire(&icache.lock);
5209
5210   // Is the inode already cached?
5211   empty = 0;
5212   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5213     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5214       ip->ref++;
5215       release(&icache.lock);
5216       return ip;
5217     }
5218     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
5219       empty = ip;
5220   }
5221
5222   // Recycle an inode cache entry.
5223   if(empty == 0)
5224     panic("iget: no inodes");
5225
5226   ip = empty;
5227   ip->dev = dev;
5228   ip->inum = inum;
5229   ip->ref = 1;
5230   ip->flags = 0;
5231   release(&icache.lock);
5232
5233   return ip;
5234 }
5235
5236 // Increment reference count for ip.
5237 // Returns ip to enable ip = idup(ip1) idiom.
5238 struct inode*
5239 idup(struct inode *ip)
5240 {
5241   acquire(&icache.lock);
5242   ip->ref++;
5243   release(&icache.lock);
5244   return ip;
5245 }
5246
5247
5248
5249
```



```
5250 // Lock the given inode.
5251 // Reads the inode from disk if necessary.
5252 void
5253 ilock(struct inode *ip)
5254 {
5255   struct buf *bp;
5256   struct dinode *dip;
5257
5258   if(ip == 0 || ip->ref < 1)
5259     panic("ilock");
5260
5261   acquire(&icache.lock);
5262   while(ip->flags & I_BUSY)
5263     sleep(ip, &icache.lock);
5264   ip->flags |= I_BUSY;
5265   release(&icache.lock);
5266
5267   if(!(ip->flags & I_VALID)){
5268     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5269     dip = (struct dinode*)bp->data + ip->inum%IPB;
5270     ip->type = dip->type;
5271     ip->major = dip->major;
5272     ip->minor = dip->minor;
5273     ip->nlink = dip->nlink;
5274     ip->size = dip->size;
5275     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5276     brelse(bp);
5277     ip->flags |= I_VALID;
5278     if(ip->type == 0)
5279       panic("ilock: no type");
5280   }
5281 }
5282
5283 // Unlock the given inode.
5284 void
5285 iunlock(struct inode *ip)
5286 {
5287   if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
5288     panic("iunlock");
5289
5290   acquire(&icache.lock);
5291   ip->flags &= ~I_BUSY;
5292   wakeup(ip);
5293   release(&icache.lock);
5294 }
5295
5296
5297
5298
5299
```

```
5300 // Drop a reference to an in-memory inode.
5301 // If that was the last reference, the inode cache entry can
5302 // be recycled.
5303 // If that was the last reference and the inode has no links
5304 // to it, free the inode (and its content) on disk.
5305 // All calls to iput() must be inside a transaction in
5306 // case it has to free the inode.
5307 void
5308 iput(struct inode *ip)
5309 {
5310   acquire(&icache.lock);
5311   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
5312     // inode has no links and no other references: truncate and free.
5313     if(ip->flags & I_BUSY)
5314       panic("iput busy");
5315     ip->flags |= I_BUSY;
5316     release(&icache.lock);
5317     itrunc(ip);
5318     ip->type = 0;
5319     iupdate(ip);
5320     acquire(&icache.lock);
5321     ip->flags = 0;
5322     wakeup(ip);
5323   }
5324   ip->ref--;
5325   release(&icache.lock);
5326 }
5327
5328 // Common idiom: unlock, then put.
5329 void
5330 iunlockput(struct inode *ip)
5331 {
5332   iunlock(ip);
5333   iput(ip);
5334 }
5335
5336
5337
5338
5339
5340
5341
5342
5343
5344
5345
5346
5347
5348
5349
```



```
5350 // Inode content
5351 //
5352 // The content (data) associated with each inode is stored
5353 // in blocks on the disk. The first NDIRECT block numbers
5354 // are listed in ip->addrs[].  The next NINDIRECT blocks are
5355 // listed in block ip->addrs[NDIRECT].
5356
5357 // Return the disk block address of the nth block in inode ip.
5358 // If there is no such block, bmap allocates one.
5359 static uint
5360 bmap(struct inode *ip, uint bn)
5361 {
5362   uint addr, *a;
5363   struct buf *bp;
5364
5365   if(bn < NDIRECT){
5366     if((addr = ip->addrs[bn]) == 0)
5367       ip->addrs[bn] = addr = balloc(ip->dev);
5368     return addr;
5369   }
5370   bn -= NDIRECT;
5371
5372   if(bn < NINDIRECT){
5373     // Load indirect block, allocating if necessary.
5374     if((addr = ip->addrs[NDIRECT]) == 0)
5375       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5376     bp = bread(ip->dev, addr);
5377     a = (uint*)bp->data;
5378     if((addr = a[bn]) == 0){
5379       a[bn] = addr = balloc(ip->dev);
5380       log_write(bp);
5381     }
5382     brelse(bp);
5383     return addr;
5384   }
5385
5386   panic("bmap: out of range");
5387 }
5388
5389
5390
5391
5392
5393
5394
5395
5396
5397
5398
5399
```

```
5400 // Truncate inode (discard contents).
5401 // Only called when the inode has no links
5402 // to it (no directory entries referring to it)
5403 // and has no in-memory reference to it (is
5404 // not an open file or current directory).
5405 static void
5406 itrunc(struct inode *ip)
5407 {
5408   int i, j;
5409   struct buf *bp;
5410   uint *a;
5411
5412   for(i = 0; i < NDIRECT; i++){
5413     if(ip->addrs[i]){
5414       bfree(ip->dev, ip->addrs[i]);
5415       ip->addrs[i] = 0;
5416     }
5417   }
5418
5419   if(ip->addrs[NDIRECT]){
5420     bp = bread(ip->dev, ip->addrs[NDIRECT]);
5421     a = (uint*)bp->data;
5422     for(j = 0; j < NINDIRECT; j++){
5423       if(a[j])
5424         bfree(ip->dev, a[j]);
5425     }
5426     brelse(bp);
5427     bfree(ip->dev, ip->addrs[NDIRECT]);
5428     ip->addrs[NDIRECT] = 0;
5429   }
5430
5431   ip->size = 0;
5432   iupdate(ip);
5433 }
5434
5435 // Copy stat information from inode.
5436 void
5437 stati(struct inode *ip, struct stat *st)
5438 {
5439   st->dev = ip->dev;
5440   st->ino = ip->inum;
5441   st->type = ip->type;
5442   st->nlink = ip->nlink;
5443   st->size = ip->size;
5444 }
5445
5446
5447
5448
5449
```



```
5450 // Read data from inode.
5451 int
5452 readi(struct inode *ip, char *dst, uint off, uint n)
5453 {
5454   uint tot, m;
5455   struct buf *bp;
5456
5457   if(ip->type == T_DEV){
5458     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
5459       return -1;
5460     return devsw[ip->major].read(ip, dst, n);
5461   }
5462
5463   if(off > ip->size || off + n < off)
5464     return -1;
5465   if(off + n > ip->size)
5466     n = ip->size - off;
5467
5468   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5469     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5470     m = min(n - tot, BSIZE - off%BSIZE);
5471     memmove(dst, bp->data + off%BSIZE, m);
5472     brelse(bp);
5473   }
5474   return n;
5475 }
5476
5477
5478
5479
5480
5481
5482
5483
5484
5485
5486
5487
5488
5489
5490
5491
5492
5493
5494
5495
5496
5497
5498
5499
```

```
5500 // Write data to inode.
5501 int
5502 writei(struct inode *ip, char *src, 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].write)
5509       return -1;
5510     return devsw[ip->major].write(ip, src, n);
5511   }
5512
5513   if(off > ip->size || off + n < off)
5514     return -1;
5515   if(off + n > MAXFILE*BSIZE)
5516     return -1;
5517
5518   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
5519     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5520     m = min(n - tot, BSIZE - off%BSIZE);
5521     memmove(bp->data + off%BSIZE, src, m);
5522     log_write(bp);
5523     brelse(bp);
5524   }
5525
5526   if(n > 0 && off > ip->size){
5527     ip->size = off;
5528     iupdate(ip);
5529   }
5530   return n;
5531 }
5532
5533
5534
5535
5536
5537
5538
5539
5540
5541
5542
5543
5544
5545
5546
5547
5548
5549
```



```
5550 // Directories
5551
5552 int
5553 namecmp(const char *s, const char *t)
5554 {
5555   return strncmp(s, t, DIRSIZ);
5556 }
5557
5558 // Look for a directory entry in a directory.
5559 // If found, set *poff to byte offset of entry.
5560 struct inode*
5561 dirlookup(struct inode *dp, char *name, uint *poff)
5562 {
5563   uint off, inum;
5564   struct dirent de;
5565
5566   if(dp->type != T_DIR)
5567     panic("dirlookup not DIR");
5568
5569   for(off = 0; off < dp->size; off += sizeof(de)){
5570     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5571       panic("dirlink read");
5572     if(de.inum == 0)
5573       continue;
5574     if(namecmp(name, de.name) == 0){
5575       // entry matches path element
5576       if(poff)
5577         *poff = off;
5578       inum = de.inum;
5579       return iget(dp->dev, inum);
5580     }
5581   }
5582
5583   return 0;
5584 }
5585
5586
5587
5588
5589
5590
5591
5592
5593
5594
5595
5596
5597
5598
5599
```

```
5600 // Write a new directory entry (name, inum) into the directory dp.
5601 int
5602 dirlink(struct inode *dp, char *name, uint inum)
5603 {
5604   int off;
5605   struct dirent de;
5606   struct inode *ip;
5607
5608   // Check that name is not present.
5609   if((ip = dirlookup(dp, name, 0)) != 0){
5610     iput(ip);
5611     return -1;
5612   }
5613
5614   // Look for an empty dirent.
5615   for(off = 0; off < dp->size; off += sizeof(de)){
5616     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5617       panic("dirlink read");
5618     if(de.inum == 0)
5619       break;
5620   }
5621
5622   strncpy(de.name, name, DIRSIZ);
5623   de.inum = inum;
5624   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5625     panic("dirlink");
5626
5627   return 0;
5628 }
5629
5630
5631
5632
5633
5634
5635
5636
5637
5638
5639
5640
5641
5642
5643
5644
5645
5646
5647
5648
5649
```



```
5650 // Paths
5651
5652 // Copy the next path element from path into name.
5653 // Return a pointer to the element following the copied one.
5654 // The returned path has no leading slashes,
5655 // so the caller can check *path=='\0' to see if the name is the last one.
5656 // If no name to remove, return 0.
5657 //
5658 // Examples:
5659 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5660 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5661 //   skipelem("a", name) = "", setting name = "a"
5662 //   skipelem("", name) = skipelem("////", name) = 0
5663 //
5664 static char*
5665 skipelem(char *path, char *name)
5666 {
5667   char *s;
5668   int len;
5669
5670   while(*path == '/')
5671     path++;
5672   if(*path == 0)
5673     return 0;
5674   s = path;
5675   while(*path != '/' && *path != 0)
5676     path++;
5677   len = path - s;
5678   if(len >= DIRSIZ)
5679     memmove(name, s, DIRSIZ);
5680   else {
5681     memmove(name, s, len);
5682     name[len] = 0;
5683   }
5684   while(*path == '/')
5685     path++;
5686   return path;
5687 }
5688
5689
5690
5691
5692
5693
5694
5695
5696
5697
5698
5699
```

```
5700 // Look up and return the inode for a path name.
5701 // If parent != 0, return the inode for the parent and copy the final
5702 // path element into name, which must have room for DIRSIZ bytes.
5703 // Must be called inside a transaction since it calls iput().
5704 static struct inode*
5705 namex(char *path, int nameiparent, char *name)
5706 {
5707   struct inode *ip, *next;
5708
5709   if(*path == '/')
5710     ip = iget(ROOTDEV, ROOTINO);
5711   else
5712     ip = idup(proc->cwd);
5713
5714   while((path = skipelem(path, name)) != 0){
5715     ilock(ip);
5716     if(ip->type != T_DIR){
5717       iunlockput(ip);
5718       return 0;
5719     }
5720     if(nameiparent && *path == '\0'){
5721       // Stop one level early.
5722       iunlock(ip);
5723       return ip;
5724     }
5725     if((next = dirlookup(ip, name, 0)) == 0){
5726       iunlockput(ip);
5727       return 0;
5728     }
5729     iunlockput(ip);
5730     ip = next;
5731   }
5732   if(nameiparent){
5733     iput(ip);
5734     return 0;
5735   }
5736   return ip;
5737 }
5738
5739 struct inode*
5740 namei(char *path)
5741 {
5742   char name[DIRSIZ];
5743   return namex(path, 0, name);
5744 }
5745
5746
5747
5748
5749
```



```
5750 struct inode*
5751 nameiparent(char *path, char *name)
5752 {
5753   return namex(path, 1, name);
5754 }
5755
5756
5757
5758
5759
5760
5761
5762
5763
5764
5765
5766
5767
5768
5769
5770
5771
5772
5773
5774
5775
5776
5777
5778
5779
5780
5781
5782
5783
5784
5785
5786
5787
5788
5789
5790
5791
5792
5793
5794
5795
5796
5797
5798
5799
```

```
5800 //
5801 // File descriptors
5802 //
5803
5804 #include "types.h"
5805 #include "defs.h"
5806 #include "param.h"
5807 #include "fs.h"
5808 #include "file.h"
5809 #include "spinlock.h"
5810
5811 struct devsw devsw[NDEV];
5812 struct {
5813   struct spinlock lock;
5814   struct file file[NFILE];
5815 } ftable;
5816
5817 void
5818 fileinit(void)
5819 {
5820   initlock(&ftable.lock, "ftable");
5821 }
5822
5823 // Allocate a file structure.
5824 struct file*
5825 filealloc(void)
5826 {
5827   struct file *f;
5828
5829   acquire(&ftable.lock);
5830   for(f = ftable.file; f < ftable.file + NFILE; f++){
5831     if(f->ref == 0){
5832       f->ref = 1;
5833       release(&ftable.lock);
5834       return f;
5835     }
5836   }
5837   release(&ftable.lock);
5838   return 0;
5839 }
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```



```
5850 // Increment ref count for file f.
5851 struct file*
5852 filedup(struct file *f)
5853 {
5854   acquire(&ftable.lock);
5855   if(f->ref < 1)
5856     panic("filedup");
5857   f->ref++;
5858   release(&ftable.lock);
5859   return f;
5860 }
5861
5862 // Close file f.  (Decrement ref count, close when reaches 0.)
5863 void
5864 fileclose(struct file *f)
5865 {
5866   struct file ff;
5867
5868   acquire(&ftable.lock);
5869   if(f->ref < 1)
5870     panic("fileclose");
5871   if(--f->ref > 0){
5872     release(&ftable.lock);
5873     return;
5874   }
5875   ff = *f;
5876   f->ref = 0;
5877   f->type = FD_NONE;
5878   release(&ftable.lock);
5879
5880   if(ff.type == FD_PIPE)
5881     pipeclose(ff.pipe, ff.writable);
5882   else if(ff.type == FD_INODE){
5883     begin_op();
5884     iput(ff.ip);
5885     end_op();
5886   }
5887 }
5888
5889
5890
5891
5892
5893
5894
5895
5896
5897
5898
5899
```

```
5900 // Get metadata about file f.
5901 int
5902 filestat(struct file *f, struct stat *st)
5903 {
5904   if(f->type == FD_INODE){
5905     ilock(f->ip);
5906     stati(f->ip, st);
5907     iunlock(f->ip);
5908     return 0;
5909   }
5910   return -1;
5911 }
5912
5913 // Read from file f.
5914 int
5915 fileread(struct file *f, char *addr, int n)
5916 {
5917   int r;
5918
5919   if(f->readable == 0)
5920     return -1;
5921   if(f->type == FD_PIPE)
5922     return piperead(f->pipe, addr, n);
5923   if(f->type == FD_INODE){
5924     ilock(f->ip);
5925     if((r = readi(f->ip, addr, f->off, n)) > 0)
5926       f->off += r;
5927     iunlock(f->ip);
5928     return r;
5929   }
5930   panic("fileread");
5931 }
5932
5933
5934
5935
5936
5937
5938
5939
5940
5941
5942
5943
5944
5945
5946
5947
5948
5949
```



```
5950 // Write to file f.
5951 int
5952 filewrite(struct file *f, char *addr, int n)
5953 {
5954   int r;
5955
5956   if(f->writable == 0)
5957     return -1;
5958   if(f->type == FD_PIPE)
5959     return pipewrite(f->pipe, addr, n);
5960   if(f->type == FD_INODE){
5961     // write a few blocks at a time to avoid exceeding
5962     // the maximum log transaction size, including
5963     // i-node, indirect block, allocation blocks,
5964     // and 2 blocks of slop for non-aligned writes.
5965     // this really belongs lower down, since writei()
5966     // might be writing a device like the console.
5967     int max = ((LOGSIZE-1-1-2) / 2) * 512;
5968     int i = 0;
5969     while(i < n){
5970       int n1 = n - i;
5971       if(n1 > max)
5972         n1 = max;
5973
5974       begin_op();
5975       ilock(f->ip);
5976       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
5977         f->off += r;
5978       iunlock(f->ip);
5979       end_op();
5980
5981       if(r < 0)
5982         break;
5983       if(r != n1)
5984         panic("short filewrite");
5985       i += r;
5986     }
5987     return i == n ? n : -1;
5988   }
5989   panic("filewrite");
5990 }
5991
5992
5993
5994
5995
5996
5997
5998
5999
```

```
6000 //
6001 // File-system system calls.
6002 // Mostly argument checking, since we don't trust
6003 // user code, and calls into file.c and fs.c.
6004 //
6005
6006 #include "types.h"
6007 #include "defs.h"
6008 #include "param.h"
6009 #include "stat.h"
6010 #include "mmu.h"
6011 #include "proc.h"
6012 #include "fs.h"
6013 #include "file.h"
6014 #include "fcntl.h"
6015
6016 // Fetch the nth word-sized system call argument as a file descriptor
6017 // and return both the descriptor and the corresponding struct file.
6018 static int
6019 argfd(int n, int *pfd, struct file **pf)
6020 {
6021   int fd;
6022   struct file *f;
6023
6024   if(argint(n, &fd) < 0)
6025     return -1;
6026   if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
6027     return -1;
6028   if(pfd)
6029     *pfd = fd;
6030   if(pf)
6031     *pf = f;
6032   return 0;
6033 }
6034
6035 // Allocate a file descriptor for the given file.
6036 // Takes over file reference from caller on success.
6037 static int
6038 fdalloc(struct file *f)
6039 {
6040   int fd;
6041
6042   for(fd = 0; fd < NOFILE; fd++){
6043     if(proc->ofile[fd] == 0){
6044       proc->ofile[fd] = f;
6045       return fd;
6046     }
6047   }
6048   return -1;
6049 }
```



```
6050 int
6051 sys_dup(void)
6052 {
6053   struct file *f;
6054   int fd;
6055
6056   if(argfd(0, 0, &f) < 0)
6057     return -1;
6058   if((fd=fdalloc(f)) < 0)
6059     return -1;
6060   filedup(f);
6061   return fd;
6062 }
6063
6064 int
6065 sys_read(void)
6066 {
6067   struct file *f;
6068   int n;
6069   char *p;
6070
6071   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6072     return -1;
6073   return fileread(f, p, n);
6074 }
6075
6076 int
6077 sys_write(void)
6078 {
6079   struct file *f;
6080   int n;
6081   char *p;
6082
6083   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6084     return -1;
6085   return filewrite(f, p, n);
6086 }
6087
6088 int
6089 sys_close(void)
6090 {
6091   int fd;
6092   struct file *f;
6093
6094   if(argfd(0, &fd, &f) < 0)
6095     return -1;
6096   proc->ofile[fd] = 0;
6097   fileclose(f);
6098   return 0;
6099 }
```

```
6100 int
6101 sys_fstat(void)
6102 {
6103   struct file *f;
6104   struct stat *st;
6105
6106   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
6107     return -1;
6108   return filestat(f, st);
6109 }
6110
6111 // Create the path new as a link to the same inode as old.
6112 int
6113 sys_link(void)
6114 {
6115   char name[DIRSIZ], *new, *old;
6116   struct inode *dp, *ip;
6117
6118   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
6119     return -1;
6120
6121   begin_op();
6122   if((ip = namei(old)) == 0){
6123     end_op();
6124     return -1;
6125   }
6126
6127   ilock(ip);
6128   if(ip->type == T_DIR){
6129     iunlockput(ip);
6130     end_op();
6131     return -1;
6132   }
6133
6134   ip->nlink++;
6135   iupdate(ip);
6136   iunlock(ip);
6137
6138   if((dp = nameiparent(new, name)) == 0)
6139     goto bad;
6140   ilock(dp);
6141   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
6142     iunlockput(dp);
6143     goto bad;
6144   }
6145   iunlockput(dp);
6146   iput(ip);
6147
6148   end_op();
6149
```



```
6150   return 0;
6151
6152 bad:
6153   ilock(ip);
6154   ip->nlink--;
6155   iupdate(ip);
6156   iunlockput(ip);
6157   end_op();
6158   return -1;
6159 }
6160
6161 // Is the directory dp empty except for "." and ".." ?
6162 static int
6163 isdirempty(struct inode *dp)
6164 {
6165   int off;
6166   struct dirent de;
6167
6168   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
6169     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6170       panic("isdirempty: readi");
6171     if(de.inum != 0)
6172       return 0;
6173   }
6174   return 1;
6175 }
6176
6177
6178
6179
6180
6181
6182
6183
6184
6185
6186
6187
6188
6189
6190
6191
6192
6193
6194
6195
6196
6197
6198
6199
```

```
6200 int
6201 sys_unlink(void)
6202 {
6203   struct inode *ip, *dp;
6204   struct dirent de;
6205   char name[DIRSIZ], *path;
6206   uint off;
6207
6208   if(argstr(0, &path) < 0)
6209     return -1;
6210
6211   begin_op();
6212   if((dp = nameiparent(path, name)) == 0){
6213     end_op();
6214     return -1;
6215   }
6216
6217   ilock(dp);
6218
6219   // Cannot unlink "." or "..".
6220   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6221     goto bad;
6222
6223   if((ip = dirlookup(dp, name, &off)) == 0)
6224     goto bad;
6225   ilock(ip);
6226
6227   if(ip->nlink < 1)
6228     panic("unlink: nlink < 1");
6229   if(ip->type == T_DIR && !isdirempty(ip)){
6230     iunlockput(ip);
6231     goto bad;
6232   }
6233
6234   memset(&de, 0, sizeof(de));
6235   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6236     panic("unlink: writei");
6237   if(ip->type == T_DIR){
6238     dp->nlink--;
6239     iupdate(dp);
6240   }
6241   iunlockput(dp);
6242
6243   ip->nlink--;
6244   iupdate(ip);
6245   iunlockput(ip);
6246
6247   end_op();
6248
6249   return 0;
```



```
6250 bad:
6251   iunlockput(dp);
6252   end_op();
6253   return -1;
6254 }
6255
6256 static struct inode*
6257 create(char *path, short type, short major, short minor)
6258 {
6259   uint off;
6260   struct inode *ip, *dp;
6261   char name[DIRSIZ];
6262
6263   if((dp = nameiparent(path, name)) == 0)
6264     return 0;
6265   ilock(dp);
6266
6267   if((ip = dirlookup(dp, name, &off)) != 0){
6268     iunlockput(dp);
6269     ilock(ip);
6270     if(type == T_FILE && ip->type == T_FILE)
6271       return ip;
6272     iunlockput(ip);
6273     return 0;
6274   }
6275
6276   if((ip = ialloc(dp->dev, type)) == 0)
6277     panic("create: ialloc");
6278
6279   ilock(ip);
6280   ip->major = major;
6281   ip->minor = minor;
6282   ip->nlink = 1;
6283   iupdate(ip);
6284
6285   if(type == T_DIR){  // Create . and .. entries.
6286     dp->nlink++;  // for ".."
6287     iupdate(dp);
6288     // No ip->nlink++ for ".": avoid cyclic ref count.
6289     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6290       panic("create dots");
6291   }
6292
6293   if(dirlink(dp, name, ip->inum) < 0)
6294     panic("create: dirlink");
6295
6296   iunlockput(dp);
6297
6298   return ip;
6299 }
```

```
6300 int
6301 sys_open(void)
6302 {
6303   char *path;
6304   int fd, omode;
6305   struct file *f;
6306   struct inode *ip;
6307
6308   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6309     return -1;
6310
6311   begin_op();
6312
6313   if(omode & O_CREATE){
6314     ip = create(path, T_FILE, 0, 0);
6315     if(ip == 0){
6316       end_op();
6317       return -1;
6318     }
6319   } else {
6320     if((ip = namei(path)) == 0){
6321       end_op();
6322       return -1;
6323     }
6324     ilock(ip);
6325     if(ip->type == T_DIR && omode != O_RDONLY){
6326       iunlockput(ip);
6327       end_op();
6328       return -1;
6329     }
6330   }
6331
6332   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6333     if(f)
6334       fileclose(f);
6335     iunlockput(ip);
6336     end_op();
6337     return -1;
6338   }
6339   iunlock(ip);
6340   end_op();
6341
6342   f->type = FD_INODE;
6343   f->ip = ip;
6344   f->off = 0;
6345   f->readable = !(omode & O_WRONLY);
6346   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6347   return fd;
6348 }
6349
```



```
6350 int
6351 sys_mkdir(void)
6352 {
6353   char *path;
6354   struct inode *ip;
6355
6356   begin_op();
6357   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
6358     end_op();
6359     return -1;
6360   }
6361   iunlockput(ip);
6362   end_op();
6363   return 0;
6364 }
6365
6366 int
6367 sys_mknod(void)
6368 {
6369   struct inode *ip;
6370   char *path;
6371   int len;
6372   int major, minor;
6373
6374   begin_op();
6375   if((len=argstr(0, &path)) < 0 ||
6376      argint(1, &major) < 0 ||
6377      argint(2, &minor) < 0 ||
6378      (ip = create(path, T_DEV, major, minor)) == 0){
6379     end_op();
6380     return -1;
6381   }
6382   iunlockput(ip);
6383   end_op();
6384   return 0;
6385 }
6386
6387
6388
6389
6390
6391
6392
6393
6394
6395
6396
6397
6398
6399
```

```
6400 int
6401 sys_chdir(void)
6402 {
6403   char *path;
6404   struct inode *ip;
6405
6406   begin_op();
6407   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
6408     end_op();
6409     return -1;
6410   }
6411   ilock(ip);
6412   if(ip->type != T_DIR){
6413     iunlockput(ip);
6414     end_op();
6415     return -1;
6416   }
6417   iunlock(ip);
6418   iput(proc->cwd);
6419   end_op();
6420   proc->cwd = ip;
6421   return 0;
6422 }
6423
6424 int
6425 sys_exec(void)
6426 {
6427   char *path, *argv[MAXARG];
6428   int i;
6429   uint uargv, uarg;
6430
6431   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6432     return -1;
6433   }
6434   memset(argv, 0, sizeof(argv));
6435   for(i=0;; i++){
6436     if(i >= NELEM(argv))
6437       return -1;
6438     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6439       return -1;
6440     if(uarg == 0){
6441       argv[i] = 0;
6442       break;
6443     }
6444     if(fetchstr(uarg, &argv[i]) < 0)
6445       return -1;
6446   }
6447   return exec(path, argv);
6448 }
6449
```



```
6450 int
6451 sys_pipe(void)
6452 {
6453   int *fd;
6454   struct file *rf, *wf;
6455   int fd0, fd1;
6456
6457   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6458     return -1;
6459   if(pipealloc(&rf, &wf) < 0)
6460     return -1;
6461   fd0 = -1;
6462   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6463     if(fd0 >= 0)
6464       proc->ofile[fd0] = 0;
6465     fileclose(rf);
6466     fileclose(wf);
6467     return -1;
6468   }
6469   fd[0] = fd0;
6470   fd[1] = fd1;
6471   return 0;
6472 }
6473
6474
6475
6476
6477
6478
6479
6480
6481
6482
6483
6484
6485
6486
6487
6488
6489
6490
6491
6492
6493
6494
6495
6496
6497
6498
6499
```

```
6500 #include "types.h"
6501 #include "param.h"
6502 #include "memlayout.h"
6503 #include "mmu.h"
6504 #include "proc.h"
6505 #include "defs.h"
6506 #include "x86.h"
6507 #include "elf.h"
6508
6509 int
6510 exec(char *path, char **argv)
6511 {
6512   char *s, *last;
6513   int i, off;
6514   uint argc, sz, sp, ustack[3+MAXARG+1];
6515   struct elfhdr elf;
6516   struct inode *ip;
6517   struct proghdr ph;
6518   pde_t *pgdir, *oldpgdir;
6519
6520   begin_op();
6521   if((ip = namei(path)) == 0){
6522     end_op();
6523     return -1;
6524   }
6525   ilock(ip);
6526   pgdir = 0;
6527
6528   // Check ELF header
6529   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
6530     goto bad;
6531   if(elf.magic != ELF_MAGIC)
6532     goto bad;
6533
6534   if((pgdir = setupkvm()) == 0)
6535     goto bad;
6536
6537   // Load program into memory.
6538   sz = 0;
6539   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6540     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6541       goto bad;
6542     if(ph.type != ELF_PROG_LOAD)
6543       continue;
6544     if(ph.memsz < ph.filesz)
6545       goto bad;
6546     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6547       goto bad;
6548     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6549       goto bad;
```



```
6550   }
6551   iunlockput(ip);
6552   end_op();
6553   ip = 0;
6554
6555   // Allocate two pages at the next page boundary.
6556   // Make the first inaccessible.  Use the second as the user stack.
6557   sz = PGROUNDUP(sz);
6558   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
6559     goto bad;
6560   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6561   sp = sz;
6562
6563   // Push argument strings, prepare rest of stack in ustack.
6564   for(argc = 0; argv[argc]; argc++) {
6565     if(argc >= MAXARG)
6566       goto bad;
6567     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6568     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6569       goto bad;
6570     ustack[3+argc] = sp;
6571   }
6572   ustack[3+argc] = 0;
6573
6574   ustack[0] = 0xffffffff;  // fake return PC
6575   ustack[1] = argc;
6576   ustack[2] = sp - (argc+1)*4;  // argv pointer
6577
6578   sp -= (3+argc+1) * 4;
6579   if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6580     goto bad;
6581
6582   // Save program name for debugging.
6583   for(last=s=path; *s; s++)
6584     if(*s == '/')
6585       last = s+1;
6586   safestrcpy(proc->name, last, sizeof(proc->name));
6587
6588   // Commit to the user image.
6589   oldpgdir = proc->pgdir;
6590   proc->pgdir = pgdir;
6591   proc->sz = sz;
6592   proc->tf->eip = elf.entry;  // main
6593   proc->tf->esp = sp;
6594   switchuvm(proc);
6595   freevm(oldpgdir);
6596   return 0;
6597
6598
6599
```

```
6600  bad:
6601    if(pgdir)
6602      freevm(pgdir);
6603    if(ip){
6604      iunlockput(ip);
6605      end_op();
6606    }
6607    return -1;
6608 }
6609
6610
6611
6612
6613
6614
6615
6616
6617
6618
6619
6620
6621
6622
6623
6624
6625
6626
6627
6628
6629
6630
6631
6632
6633
6634
6635
6636
6637
6638
6639
6640
6641
6642
6643
6644
6645
6646
6647
6648
6649
```



```
6650 #include "types.h"
6651 #include "defs.h"
6652 #include "param.h"
6653 #include "mmu.h"
6654 #include "proc.h"
6655 #include "fs.h"
6656 #include "file.h"
6657 #include "spinlock.h"
6658
6659 #define PIPESIZE 512
6660
6661 struct pipe {
6662   struct spinlock lock;
6663   char data[PIPESIZE];
6664   uint nread;     // number of bytes read
6665   uint nwrite;    // number of bytes written
6666   int readopen;   // read fd is still open
6667   int writeopen;  // write fd is still open
6668 };
6669
6670 int
6671 pipealloc(struct file **f0, struct file **f1)
6672 {
6673   struct pipe *p;
6674
6675   p = 0;
6676   *f0 = *f1 = 0;
6677   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
6678     goto bad;
6679   if((p = (struct pipe*)kalloc()) == 0)
6680     goto bad;
6681   p->readopen = 1;
6682   p->writeopen = 1;
6683   p->nwrite = 0;
6684   p->nread = 0;
6685   initlock(&p->lock, "pipe");
6686   (*f0)->type = FD_PIPE;
6687   (*f0)->readable = 1;
6688   (*f0)->writable = 0;
6689   (*f0)->pipe = p;
6690   (*f1)->type = FD_PIPE;
6691   (*f1)->readable = 0;
6692   (*f1)->writable = 1;
6693   (*f1)->pipe = p;
6694   return 0;
6695
6696
6697
6698
6699
```

```
 bad:
  if(p)
    kfree((char*)p);
  if(*f0)
    fileclose(*f0);
  if(*f1)
    fileclose(*f1);
  return -1;
}

void
pipeclose(struct pipe *p, int writable)
{
  acquire(&p->lock);
  if(writable){
    p->writeopen = 0;
    wakeup(&p->nread);
  } else {
    p->readopen = 0;
    wakeup(&p->nwrite);
  }
  if(p->readopen == 0 && p->writeopen == 0){
    release(&p->lock);
    kfree((char*)p);
  } else
    release(&p->lock);
}


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

  acquire(&p->lock);
  for(i = 0; i < n; i++){
    while(p->nwrite == p->nread + PIPESIZE){
      if(p->readopen == 0 || proc->killed){
        release(&p->lock);
        return -1;
      }
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
  wakeup(&p->nread);
  release(&p->lock);
  return n;
}
```



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

  acquire(&p->lock);
  while(p->nread == p->nwrite && p->writeopen){
    if(proc->killed){
      release(&p->lock);
      return -1;
    }
    sleep(&p->nread, &p->lock);
  }
  for(i = 0; i < n; i++){
    if(p->nread == p->nwrite)
      break;
    addr[i] = p->data[p->nread++ % PIPESIZE];
  }
  wakeup(&p->nwrite);
  release(&p->lock);
  return i;
}
```

```
6800 #include "types.h"
6801 #include "x86.h"
6802
6803 void*
6804 memset(void *dst, int c, uint n)
6805 {
6806   if ((int)dst%4 == 0 && n%4 == 0){
6807     c &= 0xFF;
6808     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
6809   } else
6810     stosb(dst, c, n);
6811   return dst;
6812 }
6813
6814 int
6815 memcmp(const void *v1, const void *v2, uint n)
6816 {
6817   const uchar *s1, *s2;
6818
6819   s1 = v1;
6820   s2 = v2;
6821   while(n-- > 0){
6822     if(*s1 != *s2)
6823       return *s1 - *s2;
6824     s1++, s2++;
6825   }
6826
6827   return 0;
6828 }
6829
6830 void*
6831 memmove(void *dst, const void *src, uint n)
6832 {
6833   const char *s;
6834   char *d;
6835
6836   s = src;
6837   d = dst;
6838   if(s < d && s + n > d){
6839     s += n;
6840     d += n;
6841     while(n-- > 0)
6842       *--d = *--s;
6843   } else
6844     while(n-- > 0)
6845       *d++ = *s++;
6846
6847   return dst;
6848 }
6849
```



```
6850 // memcpy exists to placate GCC.  Use memmove.
6851 void*
6852 memcpy(void *dst, const void *src, uint n)
6853 {
6854   return memmove(dst, src, n);
6855 }
6856
6857 int
6858 strncmp(const char *p, const char *q, uint n)
6859 {
6860   while(n > 0 && *p && *p == *q)
6861     n--, p++, q++;
6862   if(n == 0)
6863     return 0;
6864   return (uchar)*p - (uchar)*q;
6865 }
6866
6867 char*
6868 strncpy(char *s, const char *t, int n)
6869 {
6870   char *os;
6871
6872   os = s;
6873   while(n-- > 0 && (*s++ = *t++) != 0)
6874     ;
6875   while(n-- > 0)
6876     *s++ = 0;
6877   return os;
6878 }
6879
6880 // Like strncpy but guaranteed to NUL-terminate.
6881 char*
6882 safestrcpy(char *s, const char *t, int n)
6883 {
6884   char *os;
6885
6886   os = s;
6887   if(n <= 0)
6888     return os;
6889   while(--n > 0 && (*s++ = *t++) != 0)
6890     ;
6891   *s = 0;
6892   return os;
6893 }
6894
6895
6896
6897
6898
6899
```

```
6900 int
6901 strlen(const char *s)
6902 {
6903   int n;
6904
6905   for(n = 0; s[n]; n++)
6906     ;
6907   return n;
6908 }
6909
6910
6911
6912
6913
6914
6915
6916
6917
6918
6919
6920
6921
6922
6923
6924
6925
6926
6927
6928
6929
6930
6931
6932
6933
6934
6935
6936
6937
6938
6939
6940
6941
6942
6943
6944
6945
6946
6947
6948
6949
```



```
6950 // See MultiProcessor Specification Version 1.[14]
6951
6952 struct mp {             // floating pointer
6953   uchar signature[4];           // "_MP_"
6954   void *physaddr;               // phys addr of MP config table
6955   uchar length;                 // 1
6956   uchar specrev;                // [14]
6957   uchar checksum;               // all bytes must add up to 0
6958   uchar type;                   // MP system config type
6959   uchar imcrp;
6960   uchar reserved[3];
6961 };
6962
6963 struct mpconf {         // configuration table header
6964   uchar signature[4];           // "PCMP"
6965   ushort length;                // total table length
6966   uchar version;                // [14]
6967   uchar checksum;               // all bytes must add up to 0
6968   uchar product[20];            // product id
6969   uint *oemtable;               // OEM table pointer
6970   ushort oemlength;             // OEM table length
6971   ushort entry;                 // entry count
6972   uint *lapicaddr;              // address of local APIC
6973   ushort xlength;               // extended table length
6974   uchar xchecksum;              // extended table checksum
6975   uchar reserved;
6976 };
6977
6978 struct mpproc {         // processor table entry
6979   uchar type;                   // entry type (0)
6980   uchar apicid;                 // local APIC id
6981   uchar version;                // local APIC verison
6982   uchar flags;                  // CPU flags
6983     #define MPBOOT 0x02           // This proc is the bootstrap processor.
6984   uchar signature[4];           // CPU signature
6985   uint feature;                 // feature flags from CPUID instruction
6986   uchar reserved[8];
6987 };
6988
6989 struct mpioapic {       // I/O APIC table entry
6990   uchar type;                   // entry type (2)
6991   uchar apicno;                 // I/O APIC id
6992   uchar version;                // I/O APIC version
6993   uchar flags;                  // I/O APIC flags
6994   uint *addr;                  // I/O APIC address
6995 };
6996
6997
6998
6999
```

```
7000 // Table entry types
7001 #define MPPROC    0x00  // One per processor
7002 #define MPBUS     0x01  // One per bus
7003 #define MPIOAPIC  0x02  // One per I/O APIC
7004 #define MPIOINTR  0x03  // One per bus interrupt source
7005 #define MPLINTR   0x04  // One per system interrupt source
7006
7007
7008
7009
7010
7011
7012
7013
7014
7015
7016
7017
7018
7019
7020
7021
7022
7023
7024
7025
7026
7027
7028
7029
7030
7031
7032
7033
7034
7035
7036
7037
7038
7039
7040
7041
7042
7043
7044
7045
7046
7047
7048
7049
```



```
7050 // Blank page.
7051
7052
7053
7054
7055
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 // Multiprocessor support
7101 // Search memory for MP description structures.
7102 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
7103
7104 #include "types.h"
7105 #include "defs.h"
7106 #include "param.h"
7107 #include "memlayout.h"
7108 #include "mp.h"
7109 #include "x86.h"
7110 #include "mmu.h"
7111 #include "proc.h"
7112
7113 struct cpu cpus[NCPU];
7114 static struct cpu *bcpu;
7115 int ismp;
7116 int ncpu;
7117 uchar ioapicid;
7118
7119 int
7120 mpbcpu(void)
7121 {
7122   return bcpu-cpus;
7123 }
7124
7125 static uchar
7126 sum(uchar *addr, int len)
7127 {
7128   int i, sum;
7129
7130   sum = 0;
7131   for(i=0; i<len; i++)
7132     sum += addr[i];
7133   return sum;
7134 }
7135
7136 // Look for an MP structure in the len bytes at addr.
7137 static struct mp*
7138 mpsearch1(uint a, int len)
7139 {
7140   uchar *e, *p, *addr;
7141
7142   addr = p2v(a);
7143   e = addr+len;
7144   for(p = addr; p < e; p += sizeof(struct mp))
7145     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
7146       return (struct mp*)p;
7147   return 0;
7148 }
7149
```



```
7150 // Search for the MP Floating Pointer Structure, which according to the
7151 // spec is in one of the following three locations:
7152 // 1) in the first KB of the EBDA;
7153 // 2) in the last KB of system base memory;
7154 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
7155 static struct mp*
7156 mpsearch(void)
7157 {
7158   uchar *bda;
7159   uint p;
7160   struct mp *mp;
7161
7162   bda = (uchar *) P2V(0x400);
7163   if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
7164     if((mp = mpsearch1(p, 1024)))
7165       return mp;
7166   } else {
7167     p = ((bda[0x14]<<8)|bda[0x13])*1024;
7168     if((mp = mpsearch1(p-1024, 1024)))
7169       return mp;
7170   }
7171   return mpsearch1(0xF0000, 0x10000);
7172 }
7173
7174 // Search for an MP configuration table.  For now,
7175 // don't accept the default configurations (physaddr == 0).
7176 // Check for correct signature, calculate the checksum and,
7177 // if correct, check the version.
7178 // To do: check extended table checksum.
7179 static struct mpconf*
7180 mpconfig(struct mp **pmp)
7181 {
7182   struct mpconf *conf;
7183   struct mp *mp;
7184
7185   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
7186     return 0;
7187   conf = (struct mpconf*) p2v((uint) mp->physaddr);
7188   if(memcmp(conf, "PCMP", 4) != 0)
7189     return 0;
7190   if(conf->version != 1 && conf->version != 4)
7191     return 0;
7192   if(sum((uchar*)conf, conf->length) != 0)
7193     return 0;
7194   *pmp = mp;
7195   return conf;
7196 }
7197
7198
7199
```

```
7200 void
7201 mpinit(void)
7202 {
7203   uchar *p, *e;
7204   struct mp *mp;
7205   struct mpconf *conf;
7206   struct mpproc *proc;
7207   struct mpioapic *ioapic;
7208
7209   bcpu = &cpus[0];
7210   if((conf = mpconfig(&mp)) == 0)
7211     return;
7212   ismp = 1;
7213   lapic = (uint*)conf->lapicaddr;
7214   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
7215     switch(*p){
7216     case MPPROC:
7217       proc = (struct mpproc*)p;
7218       if(ncpu != proc->apicid){
7219         cprintf("mpinit: ncpu=%d apicid=%d\n", ncpu, proc->apicid);
7220         ismp = 0;
7221       }
7222       if(proc->flags & MPBOOT)
7223         bcpu = &cpus[ncpu];
7224       cpus[ncpu].id = ncpu;
7225       ncpu++;
7226       p += sizeof(struct mpproc);
7227       continue;
7228     case MPIOAPIC:
7229       ioapic = (struct mpioapic*)p;
7230       ioapicid = ioapic->apicno;
7231       p += sizeof(struct mpioapic);
7232       continue;
7233     case MPBUS:
7234     case MPIOINTR:
7235     case MPLINTR:
7236       p += 8;
7237       continue;
7238     default:
7239       cprintf("mpinit: unknown config type %x\n", *p);
7240       ismp = 0;
7241     }
7242   }
7243   if(!ismp){
7244     // Didn't like what we found; fall back to no MP.
7245     ncpu = 1;
7246     lapic = 0;
7247     ioapicid = 0;
7248     return;
7249   }
```



```
7250   if(mp->imcrp){
7251     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
7252     // But it would on real hardware.
7253     outb(0x22, 0x70);   // Select IMCR
7254     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
7255   }
7256 }
7257
7258
7259
7260
7261
7262
7263
7264
7265
7266
7267
7268
7269
7270
7271
7272
7273
7274
7275
7276
7277
7278
7279
7280
7281
7282
7283
7284
7285
7286
7287
7288
7289
7290
7291
7292
7293
7294
7295
7296
7297
7298
7299
```

Apr 21 15:28 2016 xv6/lapic.c Page 2

```
7300 // The local APIC manages internal (non-I/O) interrupts.
7301 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
7302
7303 #include "types.h"
7304 #include "defs.h"
7305 #include "date.h"
7306 #include "memlayout.h"
7307 #include "traps.h"
7308 #include "mmu.h"
7309 #include "x86.h"
7310
7311 // Local APIC registers, divided by 4 for use as uint[] indices.
7312 #define ID      (0x0020/4)   // ID
7313 #define VER     (0x0030/4)   // Version
7314 #define TPR     (0x0080/4)   // Task Priority
7315 #define EOI     (0x00B0/4)   // EOI
7316 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
7317   #define ENABLE     0x00000100   // Unit Enable
7318 #define ESR     (0x0280/4)   // Error Status
7319 #define ICRLO   (0x0300/4)   // Interrupt Command
7320   #define INIT       0x00000500   // INIT/RESET
7321   #define STARTUP    0x00000600   // Startup IPI
7322   #define DELIVS     0x00001000   // Delivery status
7323   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
7324   #define DEASSERT   0x00000000
7325   #define LEVEL      0x00008000   // Level triggered
7326   #define BCAST      0x00080000   // Send to all APICs, including self.
7327   #define BUSY       0x00001000
7328   #define FIXED      0x00000000
7329 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
7330 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
7331   #define X1         0x0000000B   // divide counts by 1
7332   #define PERIODIC   0x00020000   // Periodic
7333 #define PCINT   (0x0340/4)   // Performance Counter LVT
7334 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
7335 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
7336 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
7337   #define MASKED     0x00010000   // Interrupt masked
7338 #define TICR    (0x0380/4)   // Timer Initial Count
7339 #define TCCR    (0x0390/4)   // Timer Current Count
7340 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
7341
7342 volatile uint *lapic;  // Initialized in mp.c
7343
7344 static void
7345 lapicw(int index, int value)
7346 {
7347   lapic[index] = value;
7348   lapic[ID];  // wait for write to finish, by reading
7349 }
```


Sheet 73

```
7400 void
7401 lapicinit(void)
7402 {
7403   if(!lapic)
7404     return;
7405
7406   // Enable local APIC; set spurious interrupt vector.
7407   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
7408
7409   // The timer repeatedly counts down at bus frequency
7410   // from lapic[TICR] and then issues an interrupt.
7411   // If xv6 cared more about precise timekeeping,
7412   // TICR would be calibrated using an external time source.
7413   lapicw(TDCR, X1);
7414   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
7415   lapicw(TICR, 10000000);
7416
7417   // Disable logical interrupt lines.
7418   lapicw(LINT0, MASKED);
7419   lapicw(LINT1, MASKED);
7420
7421   // Disable performance counter overflow interrupts
7422   // on machines that provide that interrupt entry.
7423   if(((lapic[VER]>>16) & 0xFF) >= 4)
7424     lapicw(PCINT, MASKED);
7425
7426   // Map error interrupt to IRQ_ERROR.
7427   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
7428
7429   // Clear error status register (requires back-to-back writes).
7430   lapicw(ESR, 0);
7431   lapicw(ESR, 0);
7432
7433   // Ack any outstanding interrupts.
7434   lapicw(EOI, 0);
7435
7436   // Send an Init Level De-Assert to synchronise arbitration ID's.
7437   lapicw(ICRHI, 0);
7438   lapicw(ICRLO, BCAST | INIT | LEVEL);
7439   while(lapic[ICRLO] & DELIVS)
7440     ;
7441
7442   // Enable interrupts on the APIC (but not on the processor).
7443   lapicw(TPR, 0);
7444 }
7445
7446
7447
7448
7449
```



```
7450 int
7451 cpunum(void)
7452 {
7453   // Cannot call cpu when interrupts are enabled:
7454   // result not guaranteed to last long enough to be used!
7455   // Would prefer to panic but even printing is chancy here:
7456   // almost everything, including cprintf and panic, calls cpu,
7457   // often indirectly through acquire and release.
7458   if(readeflags()&FL_IF){
7459     static int n;
7460     if(n++ == 0)
7461       cprintf("cpu called from %x with interrupts enabled\n",
7462         __builtin_return_address(0));
7463   }
7464
7465   if(lapic)
7466     return lapic[ID]>>24;
7467   return 0;
7468 }
7469
7470 // Acknowledge interrupt.
7471 void
7472 lapiceoi(void)
7473 {
7474   if(lapic)
7475     lapicw(EOI, 0);
7476 }
7477
7478 // Spin for a given number of microseconds.
7479 // On real hardware would want to tune this dynamically.
7480 void
7481 microdelay(int us)
7482 {
7483 }
7484
7485 #define CMOS_PORT    0x70
7486 #define CMOS_RETURN  0x71
7487
7488 // Start additional processor running entry code at addr.
7489 // See Appendix B of MultiProcessor Specification.
7490 void
7491 lapicstartap(uchar apicid, uint addr)
7492 {
7493   int i;
7494   ushort *wrv;
7495
7496   // "The BSP must initialize CMOS shutdown code to 0AH
7497   // and the warm reset vector (DWORD based at 40:67) to point at
7498   // the AP startup code prior to the [universal startup algorithm]."
7499   outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
```

```
7500   outb(CMOS_PORT+1, 0x0A);
7501   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
7502   wrv[0] = 0;
7503   wrv[1] = addr >> 4;
7504
7505   // "Universal startup algorithm."
7506   // Send INIT (level-triggered) interrupt to reset other CPU.
7507   lapicw(ICRHI, apicid<<24);
7508   lapicw(ICRLO, INIT | LEVEL | ASSERT);
7509   microdelay(200);
7510   lapicw(ICRLO, INIT | LEVEL);
7511   microdelay(100);    // should be 10ms, but too slow in Bochs!
7512
7513   // Send startup IPI (twice!) to enter code.
7514   // Regular hardware is supposed to only accept a STARTUP
7515   // when it is in the halted state due to an INIT.  So the second
7516   // should be ignored, but it is part of the official Intel algorithm.
7517   // Bochs complains about the second one.  Too bad for Bochs.
7518   for(i = 0; i < 2; i++){
7519     lapicw(ICRHI, apicid<<24);
7520     lapicw(ICRLO, STARTUP | (addr>>12));
7521     microdelay(200);
7522   }
7523 }
7524
7525 #define CMOS_STATA   0x0a
7526 #define CMOS_STATB   0x0b
7527 #define CMOS_UIP    (1 << 7)        // RTC update in progress
7528
7529 #define SECS    0x00
7530 #define MINS    0x02
7531 #define HOURS   0x04
7532 #define DAY     0x07
7533 #define MONTH   0x08
7534 #define YEAR    0x09
7535
7536 static uint cmos_read(uint reg)
7537 {
7538   outb(CMOS_PORT,  reg);
7539   microdelay(200);
7540
7541   return inb(CMOS_RETURN);
7542 }
7543
7544
7545
7546
7547
7548
7549
```



```
7550 static void fill_rtcdate(struct rtcdate *r)
7551 {
7552   r->second = cmos_read(SECS);
7553   r->minute = cmos_read(MINS);
7554   r->hour   = cmos_read(HOURS);
7555   r->day    = cmos_read(DAY);
7556   r->month  = cmos_read(MONTH);
7557   r->year   = cmos_read(YEAR);
7558 }
7559
7560 // qemu seems to use 24-hour GWT and the values are BCD encoded
7561 void cmostime(struct rtcdate *r)
7562 {
7563   struct rtcdate t1, t2;
7564   int sb, bcd;
7565
7566   sb = cmos_read(CMOS_STATB);
7567
7568   bcd = (sb & (1 << 2)) == 0;
7569
7570   // make sure CMOS doesn't modify time while we read it
7571   for (;;) {
7572     fill_rtcdate(&t1);
7573     if (cmos_read(CMOS_STATA) & CMOS_UIP)
7574         continue;
7575     fill_rtcdate(&t2);
7576     if (memcmp(&t1, &t2, sizeof(t1)) == 0)
7577       break;
7578   }
7579
7580   // convert
7581   if (bcd) {
7582 #define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
7583     CONV(second);
7584     CONV(minute);
7585     CONV(hour  );
7586     CONV(day   );
7587     CONV(month );
7588     CONV(year  );
7589 #undef     CONV
7590   }
7591
7592   *r = t1;
7593   r->year += 2000;
7594 }
7595
7596
7597
7598
7599
```

```
7600 // The I/O APIC manages hardware interrupts for an SMP system.
7601 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
7602 // See also picirq.c.
7603
7604 #include "types.h"
7605 #include "defs.h"
7606 #include "traps.h"
7607
7608 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
7609
7610 #define REG_ID     0x00  // Register index: ID
7611 #define REG_VER    0x01  // Register index: version
7612 #define REG_TABLE  0x10  // Redirection table base
7613
7614 // The redirection table starts at REG_TABLE and uses
7615 // two registers to configure each interrupt.
7616 // The first (low) register in a pair contains configuration bits.
7617 // The second (high) register contains a bitmask telling which
7618 // CPUs can serve that interrupt.
7619 #define INT_DISABLED   0x00010000  // Interrupt disabled
7620 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
7621 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
7622 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
7623
7624 volatile struct ioapic *ioapic;
7625
7626 // IO APIC MMIO structure: write reg, then read or write data.
7627 struct ioapic {
7628   uint reg;
7629   uint pad[3];
7630   uint data;
7631 };
7632
7633 static uint
7634 ioapicread(int reg)
7635 {
7636   ioapic->reg = reg;
7637   return ioapic->data;
7638 }
7639
7640 static void
7641 ioapicwrite(int reg, uint data)
7642 {
7643   ioapic->reg = reg;
7644   ioapic->data = data;
7645 }
7646
7647
7648
7649
```



```
7650 void
7651 ioapicinit(void)
7652 {
7653   int i, id, maxintr;
7654
7655   if(!ismp)
7656     return;
7657
7658   ioapic = (volatile struct ioapic*)IOAPIC;
7659   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
7660   id = ioapicread(REG_ID) >> 24;
7661   if(id != ioapicid)
7662     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
7663
7664   // Mark all interrupts edge-triggered, active high, disabled,
7665   // and not routed to any CPUs.
7666   for(i = 0; i <= maxintr; i++){
7667     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
7668     ioapicwrite(REG_TABLE+2*i+1, 0);
7669   }
7670 }
7671
7672 void
7673 ioapicenable(int irq, int cpunum)
7674 {
7675   if(!ismp)
7676     return;
7677
7678   // Mark interrupt edge-triggered, active high,
7679   // enabled, and routed to the given cpunum,
7680   // which happens to be that cpu's APIC ID.
7681   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
7682   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
7683 }
7684
7685
7686
7687
7688
7689
7690
7691
7692
7693
7694
7695
7696
7697
7698
7699
```

```
7700 // Intel 8259A programmable interrupt controllers.
7701
7702 #include "types.h"
7703 #include "x86.h"
7704 #include "traps.h"
7705
7706 // I/O Addresses of the two programmable interrupt controllers
7707 #define IO_PIC1         0x20    // Master (IRQs 0-7)
7708 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
7709
7710 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
7711
7712 // Current IRQ mask.
7713 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
7714 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
7715
7716 static void
7717 picsetmask(ushort mask)
7718 {
7719   irqmask = mask;
7720   outb(IO_PIC1+1, mask);
7721   outb(IO_PIC2+1, mask >> 8);
7722 }
7723
7724 void
7725 picenable(int irq)
7726 {
7727   picsetmask(irqmask & ~(1<<irq));
7728 }
7729
7730 // Initialize the 8259A interrupt controllers.
7731 void
7732 picinit(void)
7733 {
7734   // mask all interrupts
7735   outb(IO_PIC1+1, 0xFF);
7736   outb(IO_PIC2+1, 0xFF);
7737
7738   // Set up master (8259A-1)
7739
7740   // ICW1:  0001g0hi
7741   //    g:  0 = edge triggering, 1 = level triggering
7742   //    h:  0 = cascaded PICs, 1 = master only
7743   //    i:  0 = no ICW4, 1 = ICW4 required
7744   outb(IO_PIC1, 0x11);
7745
7746   // ICW2:  Vector offset
7747   outb(IO_PIC1+1, T_IRQ0);
7748
7749
```



```
7750   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
7751   //        (slave PIC) 3-bit # of slave's connection to master
7752   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
7753
7754   // ICW4:  000nbmap
7755   //    n:  1 = special fully nested mode
7756   //    b:  1 = buffered mode
7757   //    m:  0 = slave PIC, 1 = master PIC
7758   //      (ignored when b is 0, as the master/slave role
7759   //      can be hardwired).
7760   //    a:  1 = Automatic EOI mode
7761   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
7762   outb(IO_PIC1+1, 0x3);
7763
7764   // Set up slave (8259A-2)
7765   outb(IO_PIC2, 0x11);                  // ICW1
7766   outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
7767   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
7768   // NB Automatic EOI mode doesn't tend to work on the slave.
7769   // Linux source code says it's "to be investigated".
7770   outb(IO_PIC2+1, 0x3);                 // ICW4
7771
7772   // OCW3:  0ef01prs
7773   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
7774   //    p:  0 = no polling, 1 = polling mode
7775   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
7776   outb(IO_PIC1, 0x68);             // clear specific mask
7777   outb(IO_PIC1, 0x0a);             // read IRR by default
7778
7779   outb(IO_PIC2, 0x68);             // OCW3
7780   outb(IO_PIC2, 0x0a);             // OCW3
7781
7782   if(irqmask != 0xFFFF)
7783     picsetmask(irqmask);
7784 }
7785
7786
7787
7788
7789
7790
7791
7792
7793
7794
7795
7796
7797
7798
7799
```

```
7800 // PC keyboard interface constants
7801
7802 #define KBSTATP         0x64    // kbd controller status port(I)
7803 #define KBS_DIB         0x01    // kbd data in buffer
7804 #define KBDATAP         0x60    // kbd data port(I)
7805
7806 #define NO              0
7807
7808 #define SHIFT           (1<<0)
7809 #define CTL             (1<<1)
7810 #define ALT             (1<<2)
7811
7812 #define CAPSLOCK        (1<<3)
7813 #define NUMLOCK         (1<<4)
7814 #define SCROLLLOCK      (1<<5)
7815
7816 #define E0ESC           (1<<6)
7817
7818 // Special keycodes
7819 #define KEY_HOME        0xE0
7820 #define KEY_END         0xE1
7821 #define KEY_UP          0xE2
7822 #define KEY_DN          0xE3
7823 #define KEY_LF          0xE4
7824 #define KEY_RT          0xE5
7825 #define KEY_PGUP        0xE6
7826 #define KEY_PGDN        0xE7
7827 #define KEY_INS         0xE8
7828 #define KEY_DEL         0xE9
7829
7830 // C('A') == Control-A
7831 #define C(x) (x - '@')
7832
7833 static uchar shiftcode[256] =
7834 {
7835   [0x1D] CTL,
7836   [0x2A] SHIFT,
7837   [0x36] SHIFT,
7838   [0x38] ALT,
7839   [0x9D] CTL,
7840   [0xB8] ALT
7841 };
7842
7843 static uchar togglecode[256] =
7844 {
7845   [0x3A] CAPSLOCK,
7846   [0x45] NUMLOCK,
7847   [0x46] SCROLLLOCK
7848 };
7849
```



```
7850 static uchar normalmap[256] =
7851 {
7852   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
7853   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
7854   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
7855   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
7856   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
7857   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
7858   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
7859   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7860   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7861   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7862   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7863   [0x9C] '\n',      // KP_Enter
7864   [0xB5] '/',       // KP_Div
7865   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7866   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7867   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7868   [0x97] KEY_HOME,  [0xCF] KEY_END,
7869   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7870 };
7871
7872 static uchar shiftmap[256] =
7873 {
7874   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
7875   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
7876   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
7877   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
7878   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
7879   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
7880   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
7881   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
7882   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
7883   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
7884   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
7885   [0x9C] '\n',      // KP_Enter
7886   [0xB5] '/',       // KP_Div
7887   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7888   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7889   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7890   [0x97] KEY_HOME,  [0xCF] KEY_END,
7891   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7892 };
7893
7894
7895
7896
7897
7898
7899
```

```
7900 static uchar ctlmap[256] =
7901 {
7902   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7903   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7904   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
7905   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
7906   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
7907   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
7908   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
7909   [0x9C] '\r',      // KP_Enter
7910   [0xB5] C('/'),    // KP_Div
7911   [0xC8] KEY_UP,    [0xD0] KEY_DN,
7912   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7913   [0xCB] KEY_LF,    [0xCD] KEY_RT,
7914   [0x97] KEY_HOME,  [0xCF] KEY_END,
7915   [0xD2] KEY_INS,   [0xD3] KEY_DEL
7916 };
7917
7918
7919
7920
7921
7922
7923
7924
7925
7926
7927
7928
7929
7930
7931
7932
7933
7934
7935
7936
7937
7938
7939
7940
7941
7942
7943
7944
7945
7946
7947
7948
7949
```



```
7950 #include "types.h"
7951 #include "x86.h"
7952 #include "defs.h"
7953 #include "kbd.h"
7954
7955 int
7956 kbdgetc(void)
7957 {
7958   static uint shift;
7959   static uchar *charcode[4] = {
7960     normalmap, shiftmap, ctlmap, ctlmap
7961   };
7962   uint st, data, c;
7963
7964   st = inb(KBSTATP);
7965   if((st & KBS_DIB) == 0)
7966     return -1;
7967   data = inb(KBDATAP);
7968
7969   if(data == 0xE0){
7970     shift |= E0ESC;
7971     return 0;
7972   } else if(data & 0x80){
7973     // Key released
7974     data = (shift & E0ESC ? data : data & 0x7F);
7975     shift &= ~(shiftcode[data] | E0ESC);
7976     return 0;
7977   } else if(shift & E0ESC){
7978     // Last character was an E0 escape; or with 0x80
7979     data |= 0x80;
7980     shift &= ~E0ESC;
7981   }
7982
7983   shift |= shiftcode[data];
7984   shift ^= togglecode[data];
7985   c = charcode[shift & (CTL | SHIFT)][data];
7986   if(shift & CAPSLOCK){
7987     if('a' <= c && c <= 'z')
7988       c += 'A' - 'a';
7989     else if('A' <= c && c <= 'Z')
7990       c += 'a' - 'A';
7991   }
7992   return c;
7993 }
7994
7995 void
7996 kbdintr(void)
7997 {
7998   consoleintr(kbdgetc);
7999 }
```

```
8000 // Console input and output.
8001 // Input is from the keyboard or serial port.
8002 // Output is written to the screen and serial port.
8003
8004 #include "types.h"
8005 #include "defs.h"
8006 #include "param.h"
8007 #include "traps.h"
8008 #include "spinlock.h"
8009 #include "fs.h"
8010 #include "file.h"
8011 #include "memlayout.h"
8012 #include "mmu.h"
8013 #include "proc.h"
8014 #include "x86.h"
8015
8016 static void consputc(int);
8017
8018 static int panicked = 0;
8019
8020 static struct {
8021   struct spinlock lock;
8022   int locking;
8023 } cons;
8024
8025 static void
8026 printint(int xx, int base, int sign)
8027 {
8028   static char digits[] = "0123456789abcdef";
8029   char buf[16];
8030   int i;
8031   uint x;
8032
8033   if(sign && (sign = xx < 0))
8034     x = -xx;
8035   else
8036     x = xx;
8037
8038   i = 0;
8039   do{
8040     buf[i++] = digits[x % base];
8041   }while((x /= base) != 0);
8042
8043   if(sign)
8044     buf[i++] = '-';
8045
8046   while(--i >= 0)
8047     consputc(buf[i]);
8048 }
8049
```



```
8050 // Print to the console. only understands %d, %x, %p, %s.
8051 void
8052 cprintf(char *fmt, ...)
8053 {
8054   int i, c, locking;
8055   uint *argp;
8056   char *s;
8057
8058   locking = cons.locking;
8059   if(locking)
8060     acquire(&cons.lock);
8061
8062   if (fmt == 0)
8063     panic("null fmt");
8064
8065   argp = (uint*)(void*)(&fmt + 1);
8066   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
8067     if(c != '%'){
8068       consputc(c);
8069       continue;
8070     }
8071     c = fmt[++i] & 0xff;
8072     if(c == 0)
8073       break;
8074     switch(c){
8075     case 'd':
8076       printint(*argp++, 10, 1);
8077       break;
8078     case 'x':
8079     case 'p':
8080       printint(*argp++, 16, 0);
8081       break;
8082     case 's':
8083       if((s = (char*)*argp++) == 0)
8084         s = "(null)";
8085       for(; *s; s++)
8086         consputc(*s);
8087       break;
8088     case '%':
8089       consputc('%');
8090       break;
8091     default:
8092       // Print unknown % sequence to draw attention.
8093       consputc('%');
8094       consputc(c);
8095       break;
8096     }
8097   }
8098
8099
```

```
8100   if(locking)
8101     release(&cons.lock);
8102 }
8103
8104 void
8105 panic(char *s)
8106 {
8107   int i;
8108   uint pcs[10];
8109
8110   cli();
8111   cons.locking = 0;
8112   cprintf("cpu%d: panic: ", cpu->id);
8113   cprintf(s);
8114   cprintf("\n");
8115   getcallerpcs(&s, pcs);
8116   for(i=0; i<10; i++)
8117     cprintf(" %p", pcs[i]);
8118   panicked = 1; // freeze other CPU
8119   for(;;)
8120     ;
8121 }
8122
8123
8124
8125
8126
8127
8128
8129
8130
8131
8132
8133
8134
8135
8136
8137
8138
8139
8140
8141
8142
8143
8144
8145
8146
8147
8148
8149
```



```
8150 #define BACKSPACE 0x100
8151 #define CRTPORT 0x3d4
8152 static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory
8153
8154 static void
8155 cgaputc(int c)
8156 {
8157   int pos;
8158
8159   // Cursor position: col + 80*row.
8160   outb(CRTPORT, 14);
8161   pos = inb(CRTPORT+1) << 8;
8162   outb(CRTPORT, 15);
8163   pos |= inb(CRTPORT+1);
8164
8165   if(c == '\n')
8166     pos += 80 - pos%80;
8167   else if(c == BACKSPACE){
8168     if(pos > 0) --pos;
8169   } else
8170     crt[pos++] = (c&0xff) | 0x0700;  // black on white
8171
8172   if(pos < 0 || pos > 25*80)
8173     panic("pos under/overflow");
8174
8175   if((pos/80) >= 24){  // Scroll up.
8176     memmove(crt, crt+80, sizeof(crt[0])*23*80);
8177     pos -= 80;
8178     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
8179   }
8180
8181   outb(CRTPORT, 14);
8182   outb(CRTPORT+1, pos>>8);
8183   outb(CRTPORT, 15);
8184   outb(CRTPORT+1, pos);
8185   crt[pos] = ' ' | 0x0700;
8186 }
8187
8188
8189
8190
8191
8192
8193
8194
8195
8196
8197
8198
8199
```

```
8200 void
8201 consputc(int c)
8202 {
8203   if(panicked){
8204     cli();
8205     for(;;)
8206       ;
8207   }
8208
8209   if(c == BACKSPACE){
8210     uartputc('\b'); uartputc(' '); uartputc('\b');
8211   } else
8212     uartputc(c);
8213   cgaputc(c);
8214 }
8215
8216 #define INPUT_BUF 128
8217 struct {
8218   char buf[INPUT_BUF];
8219   uint r;  // Read index
8220   uint w;  // Write index
8221   uint e;  // Edit index
8222 } input;
8223
8224 #define C(x)  ((x)-'@')  // Control-x
8225
8226 void
8227 consoleintr(int (*getc)(void))
8228 {
8229   int c, doprocdump = 0;
8230
8231   acquire(&cons.lock);
8232   while((c = getc()) >= 0){
8233     switch(c){
8234     case C('P'):  // Process listing.
8235       doprocdump = 1;   // procdump() locks cons.lock indirectly; invoke lat
8236       break;
8237     case C('U'):  // Kill line.
8238       while(input.e != input.w &&
8239             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
8240         input.e--;
8241         consputc(BACKSPACE);
8242       }
8243       break;
8244     case C('H'): case '\x7f':  // Backspace
8245       if(input.e != input.w){
8246         input.e--;
8247         consputc(BACKSPACE);
8248       }
8249       break;
```



```
8250     default:
8251       if(c != 0 && input.e-input.r < INPUT_BUF){
8252         c = (c == '\r') ? '\n' : c;
8253         input.buf[input.e++ % INPUT_BUF] = c;
8254         consputc(c);
8255         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
8256           input.w = input.e;
8257           wakeup(&input.r);
8258         }
8259       }
8260       break;
8261     }
8262   }
8263   release(&cons.lock);
8264   if(doprocdump) {
8265     procdump();  // now call procdump() wo. cons.lock held
8266   }
8267 }
8268
8269 int
8270 consoleread(struct inode *ip, char *dst, int n)
8271 {
8272   uint target;
8273   int c;
8274
8275   iunlock(ip);
8276   target = n;
8277   acquire(&cons.lock);
8278   while(n > 0){
8279     while(input.r == input.w){
8280       if(proc->killed){
8281         release(&cons.lock);
8282         ilock(ip);
8283         return -1;
8284       }
8285       sleep(&input.r, &cons.lock);
8286     }
8287     c = input.buf[input.r++ % INPUT_BUF];
8288     if(c == C('D')){  // EOF
8289       if(n < target){
8290         // Save ^D for next time, to make sure
8291         // caller gets a 0-byte result.
8292         input.r--;
8293       }
8294       break;
8295     }
8296     *dst++ = c;
8297     --n;
8298     if(c == '\n')
8299       break;
```

```
8300   }
8301   release(&cons.lock);
8302   ilock(ip);
8303
8304   return target - n;
8305 }
8306
8307 int
8308 consolewrite(struct inode *ip, char *buf, int n)
8309 {
8310   int i;
8311
8312   iunlock(ip);
8313   acquire(&cons.lock);
8314   for(i = 0; i < n; i++)
8315     consputc(buf[i] & 0xff);
8316   release(&cons.lock);
8317   ilock(ip);
8318
8319   return n;
8320 }
8321
8322 void
8323 consoleinit(void)
8324 {
8325   initlock(&cons.lock, "console");
8326
8327   devsw[CONSOLE].write = consolewrite;
8328   devsw[CONSOLE].read = consoleread;
8329   cons.locking = 1;
8330
8331   picenable(IRQ_KBD);
8332   ioapicenable(IRQ_KBD, 0);
8333 }
8334
8335
8336
8337
8338
8339
8340
8341
8342
8343
8344
8345
8346
8347
8348
8349
```



```
8350 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
8351 // Only used on uniprocessors;
8352 // SMP machines use the local APIC timer.
8353
8354 #include "types.h"
8355 #include "defs.h"
8356 #include "traps.h"
8357 #include "x86.h"
8358
8359 #define IO_TIMER1       0x040           // 8253 Timer #1
8360
8361 // Frequency of all three count-down timers;
8362 // (TIMER_FREQ/freq) is the appropriate count
8363 // to generate a frequency of freq Hz.
8364
8365 #define TIMER_FREQ      1193182
8366 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
8367
8368 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
8369 #define TIMER_SEL0      0x00    // select counter 0
8370 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
8371 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
8372
8373 void
8374 timerinit(void)
8375 {
8376   // Interrupt 100 times/sec.
8377   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
8378   outb(IO_TIMER1, TIMER_DIV(100) % 256);
8379   outb(IO_TIMER1, TIMER_DIV(100) / 256);
8380   picenable(IRQ_TIMER);
8381 }
8382
8383
8384
8385
8386
8387
8388
8389
8390
8391
8392
8393
8394
8395
8396
8397
8398
8399
```

```
8400 // Intel 8250 serial port (UART).
8401
8402 #include "types.h"
8403 #include "defs.h"
8404 #include "param.h"
8405 #include "traps.h"
8406 #include "spinlock.h"
8407 #include "fs.h"
8408 #include "file.h"
8409 #include "mmu.h"
8410 #include "proc.h"
8411 #include "x86.h"
8412
8413 #define COM1    0x3f8
8414
8415 static int uart;    // is there a uart?
8416
8417 void
8418 uartinit(void)
8419 {
8420   char *p;
8421
8422   // Turn off the FIFO
8423   outb(COM1+2, 0);
8424
8425   // 9600 baud, 8 data bits, 1 stop bit, parity off.
8426   outb(COM1+3, 0x80);    // Unlock divisor
8427   outb(COM1+0, 115200/9600);
8428   outb(COM1+1, 0);
8429   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
8430   outb(COM1+4, 0);
8431   outb(COM1+1, 0x01);    // Enable receive interrupts.
8432
8433   // If status is 0xFF, no serial port.
8434   if(inb(COM1+5) == 0xFF)
8435     return;
8436   uart = 1;
8437
8438   // Acknowledge pre-existing interrupt conditions;
8439   // enable interrupts.
8440   inb(COM1+2);
8441   inb(COM1+0);
8442   picenable(IRQ_COM1);
8443   ioapicenable(IRQ_COM1, 0);
8444
8445   // Announce that we're here.
8446   for(p="xv6...\n"; *p; p++)
8447     uartputc(*p);
8448 }
8449
```



```
8450 void
8451 uartputc(int c)
8452 {
8453   int i;
8454
8455   if(!uart)
8456     return;
8457   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
8458     microdelay(10);
8459   outb(COM1+0, c);
8460 }
8461
8462 static int
8463 uartgetc(void)
8464 {
8465   if(!uart)
8466     return -1;
8467   if(!(inb(COM1+5) & 0x01))
8468     return -1;
8469   return inb(COM1+0);
8470 }
8471
8472 void
8473 uartintr(void)
8474 {
8475   consoleintr(uartgetc);
8476 }
8477
8478
8479
8480
8481
8482
8483
8484
8485
8486
8487
8488
8489
8490
8491
8492
8493
8494
8495
8496
8497
8498
8499
```

```
8500 # Initial process execs /init.
8501
8502 #include "syscall.h"
8503 #include "traps.h"
8504
8505
8506 # exec(init, argv)
8507 .globl start
8508 start:
8509   pushl $argv
8510   pushl $init
8511   pushl $0  // where caller pc would be
8512   movl $SYS_exec, %eax
8513   int $T_SYSCALL
8514
8515 # for(;;) exit();
8516 exit:
8517   movl $SYS_exit, %eax
8518   int $T_SYSCALL
8519   jmp exit
8520
8521 # char init[] = "/init\0";
8522 init:
8523   .string "/init\0"
8524
8525 # char *argv[] = { init, 0 };
8526 .p2align 2
8527 argv:
8528   .long init
8529   .long 0
8530
8531
8532
8533
8534
8535
8536
8537
8538
8539
8540
8541
8542
8543
8544
8545
8546
8547
8548
8549
```



```
8550 #include "syscall.h"
8551 #include "traps.h"
8552
8553 #define SYSCALL(name) \
8554   .globl name; \
8555   name: \
8556     movl $SYS_ ## name, %eax; \
8557     int $T_SYSCALL; \
8558     ret
8559
8560 SYSCALL(fork)
8561 SYSCALL(exit)
8562 SYSCALL(wait)
8563 SYSCALL(pipe)
8564 SYSCALL(read)
8565 SYSCALL(write)
8566 SYSCALL(close)
8567 SYSCALL(kill)
8568 SYSCALL(exec)
8569 SYSCALL(open)
8570 SYSCALL(mknod)
8571 SYSCALL(unlink)
8572 SYSCALL(fstat)
8573 SYSCALL(link)
8574 SYSCALL(mkdir)
8575 SYSCALL(chdir)
8576 SYSCALL(dup)
8577 SYSCALL(getpid)
8578 SYSCALL(sbrk)
8579 SYSCALL(sleep)
8580 SYSCALL(uptime)
8581 SYSCALL(halt)
8582 SYSCALL(date)
8583
8584
8585
8586
8587
8588
8589
8590
8591
8592
8593
8594
8595
8596
8597
8598
8599
```

```
8600 // init: The initial user-level program
8601
8602 #include "types.h"
8603 #include "stat.h"
8604 #include "user.h"
8605 #include "fcntl.h"
8606
8607 char *argv[] = { "sh", 0 };
8608
8609 int
8610 main(void)
8611 {
8612   int pid, wpid;
8613
8614   if(open("console", O_RDWR) < 0){
8615     mknod("console", 1, 1);
8616     open("console", O_RDWR);
8617   }
8618   dup(0);  // stdout
8619   dup(0);  // stderr
8620
8621   for(;;){
8622     printf(1, "init: starting sh\n");
8623     pid = fork();
8624     if(pid < 0){
8625       printf(1, "init: fork failed\n");
8626       exit();
8627     }
8628     if(pid == 0){
8629       exec("sh", argv);
8630       printf(1, "init: exec sh failed\n");
8631       exit();
8632     }
8633     while((wpid=wait()) >= 0 && wpid != pid)
8634       printf(1, "zombie!\n");
8635   }
8636 }
8637
8638
8639
8640
8641
8642
8643
8644
8645
8646
8647
8648
8649
```



```
8650 // Shell.
8651
8652 #include "types.h"
8653 #include "user.h"
8654 #include "fcntl.h"
8655
8656 // Parsed command representation
8657 #define EXEC  1
8658 #define REDIR 2
8659 #define PIPE  3
8660 #define LIST  4
8661 #define BACK  5
8662
8663 #define MAXARGS 10
8664
8665 struct cmd {
8666   int type;
8667 };
8668
8669 struct execcmd {
8670   int type;
8671   char *argv[MAXARGS];
8672   char *eargv[MAXARGS];
8673 };
8674
8675 struct redircmd {
8676   int type;
8677   struct cmd *cmd;
8678   char *file;
8679   char *efile;
8680   int mode;
8681   int fd;
8682 };
8683
8684 struct pipecmd {
8685   int type;
8686   struct cmd *left;
8687   struct cmd *right;
8688 };
8689
8690 struct listcmd {
8691   int type;
8692   struct cmd *left;
8693   struct cmd *right;
8694 };
8695
8696 struct backcmd {
8697   int type;
8698   struct cmd *cmd;
8699 };
```

```
8700 int fork1(void);  // Fork but panics on failure.
8701 void panic(char*);
8702 struct cmd *parsecmd(char*);
8703
8704 // Execute cmd.  Never returns.
8705 void
8706 runcmd(struct cmd *cmd)
8707 {
8708   int p[2];
8709   struct backcmd *bcmd;
8710   struct execcmd *ecmd;
8711   struct listcmd *lcmd;
8712   struct pipecmd *pcmd;
8713   struct redircmd *rcmd;
8714
8715   if(cmd == 0)
8716     exit();
8717
8718   switch(cmd->type){
8719   default:
8720     panic("runcmd");
8721
8722   case EXEC:
8723     ecmd = (struct execcmd*)cmd;
8724     if(ecmd->argv[0] == 0)
8725       exit();
8726     exec(ecmd->argv[0], ecmd->argv);
8727     printf(2, "exec %s failed\n", ecmd->argv[0]);
8728     break;
8729
8730   case REDIR:
8731     rcmd = (struct redircmd*)cmd;
8732     close(rcmd->fd);
8733     if(open(rcmd->file, rcmd->mode) < 0){
8734       printf(2, "open %s failed\n", rcmd->file);
8735       exit();
8736     }
8737     runcmd(rcmd->cmd);
8738     break;
8739
8740   case LIST:
8741     lcmd = (struct listcmd*)cmd;
8742     if(fork1() == 0)
8743       runcmd(lcmd->left);
8744     wait();
8745     runcmd(lcmd->right);
8746     break;
8747
8748
8749
```



```
8750   case PIPE:
8751     pcmd = (struct pipecmd*)cmd;
8752     if(pipe(p) < 0)
8753       panic("pipe");
8754     if(fork1() == 0){
8755       close(1);
8756       dup(p[1]);
8757       close(p[0]);
8758       close(p[1]);
8759       runcmd(pcmd->left);
8760     }
8761     if(fork1() == 0){
8762       close(0);
8763       dup(p[0]);
8764       close(p[0]);
8765       close(p[1]);
8766       runcmd(pcmd->right);
8767     }
8768     close(p[0]);
8769     close(p[1]);
8770     wait();
8771     wait();
8772     break;
8773
8774   case BACK:
8775     bcmd = (struct backcmd*)cmd;
8776     if(fork1() == 0)
8777       runcmd(bcmd->cmd);
8778     break;
8779   }
8780   exit();
8781 }
8782
8783 int
8784 getcmd(char *buf, int nbuf)
8785 {
8786   printf(2, "$ ");
8787   memset(buf, 0, nbuf);
8788   gets(buf, nbuf);
8789   if(buf[0] == 0) // EOF
8790     return -1;
8791   return 0;
8792 }
8793
8794
8795
8796
8797
8798
8799
```

```
8800 int
8801 main(void)
8802 {
8803   static char buf[100];
8804   int fd;
8805
8806   // Assumes three file descriptors open.
8807   while((fd = open("console", O_RDWR)) >= 0){
8808     if(fd >= 3){
8809       close(fd);
8810       break;
8811     }
8812   }
8813
8814   // Read and run input commands.
8815   while(getcmd(buf, sizeof(buf)) >= 0){
8816     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
8817       // Clumsy but will have to do for now.
8818       // Chdir has no effect on the parent if run in the child.
8819       buf[strlen(buf)-1] = 0;  // chop \n
8820       if(chdir(buf+3) < 0)
8821         printf(2, "cannot cd %s\n", buf+3);
8822       continue;
8823     }
8824     if(fork1() == 0)
8825       runcmd(parsecmd(buf));
8826     wait();
8827   }
8828   exit();
8829 }
8830
8831 void
8832 panic(char *s)
8833 {
8834   printf(2, "%s\n", s);
8835   exit();
8836 }
8837
8838 int
8839 fork1(void)
8840 {
8841   int pid;
8842
8843   pid = fork();
8844   if(pid == -1)
8845     panic("fork");
8846   return pid;
8847 }
8848
8849
```



```
8850 // Constructors
8851
8852 struct cmd*
8853 execcmd(void)
8854 {
8855   struct execcmd *cmd;
8856
8857   cmd = malloc(sizeof(*cmd));
8858   memset(cmd, 0, sizeof(*cmd));
8859   cmd->type = EXEC;
8860   return (struct cmd*)cmd;
8861 }
8862
8863 struct cmd*
8864 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
8865 {
8866   struct redircmd *cmd;
8867
8868   cmd = malloc(sizeof(*cmd));
8869   memset(cmd, 0, sizeof(*cmd));
8870   cmd->type = REDIR;
8871   cmd->cmd = subcmd;
8872   cmd->file = file;
8873   cmd->efile = efile;
8874   cmd->mode = mode;
8875   cmd->fd = fd;
8876   return (struct cmd*)cmd;
8877 }
8878
8879 struct cmd*
8880 pipecmd(struct cmd *left, struct cmd *right)
8881 {
8882   struct pipecmd *cmd;
8883
8884   cmd = malloc(sizeof(*cmd));
8885   memset(cmd, 0, sizeof(*cmd));
8886   cmd->type = PIPE;
8887   cmd->left = left;
8888   cmd->right = right;
8889   return (struct cmd*)cmd;
8890 }
8891
8892
8893
8894
8895
8896
8897
8898
8899
```

```
8900 struct cmd*
8901 listcmd(struct cmd *left, struct cmd *right)
8902 {
8903   struct listcmd *cmd;
8904
8905   cmd = malloc(sizeof(*cmd));
8906   memset(cmd, 0, sizeof(*cmd));
8907   cmd->type = LIST;
8908   cmd->left = left;
8909   cmd->right = right;
8910   return (struct cmd*)cmd;
8911 }
8912
8913 struct cmd*
8914 backcmd(struct cmd *subcmd)
8915 {
8916   struct backcmd *cmd;
8917
8918   cmd = malloc(sizeof(*cmd));
8919   memset(cmd, 0, sizeof(*cmd));
8920   cmd->type = BACK;
8921   cmd->cmd = subcmd;
8922   return (struct cmd*)cmd;
8923 }
8924
8925
8926
8927
8928
8929
8930
8931
8932
8933
8934
8935
8936
8937
8938
8939
8940
8941
8942
8943
8944
8945
8946
8947
8948
8949
```



```
8950 // Parsing
8951
8952 char whitespace[] = " \t\r\n\v";
8953 char symbols[] = "<|>&;()";
8954
8955 int
8956 gettoken(char **ps, char *es, char **q, char **eq)
8957 {
8958   char *s;
8959   int ret;
8960
8961   s = *ps;
8962   while(s < es && strchr(whitespace, *s))
8963     s++;
8964   if(q)
8965     *q = s;
8966   ret = *s;
8967   switch(*s){
8968   case 0:
8969     break;
8970   case '|':
8971   case '(':
8972   case ')':
8973   case ';':
8974   case '&':
8975   case '<':
8976     s++;
8977     break;
8978   case '>':
8979     s++;
8980     if(*s == '>'){
8981       ret = '+';
8982       s++;
8983     }
8984     break;
8985   default:
8986     ret = 'a';
8987     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
8988       s++;
8989     break;
8990   }
8991   if(eq)
8992     *eq = s;
8993
8994   while(s < es && strchr(whitespace, *s))
8995     s++;
8996   *ps = s;
8997   return ret;
8998 }
8999
```

```
9000 int
9001 peek(char **ps, char *es, char *toks)
9002 {
9003   char *s;
9004
9005   s = *ps;
9006   while(s < es && strchr(whitespace, *s))
9007     s++;
9008   *ps = s;
9009   return *s && strchr(toks, *s);
9010 }
9011
9012 struct cmd *parseline(char**, char*);
9013 struct cmd *parsepipe(char**, char*);
9014 struct cmd *parseexec(char**, char*);
9015 struct cmd *nulterminate(struct cmd*);
9016
9017 struct cmd*
9018 parsecmd(char *s)
9019 {
9020   char *es;
9021   struct cmd *cmd;
9022
9023   es = s + strlen(s);
9024   cmd = parseline(&s, es);
9025   peek(&s, es, "");
9026   if(s != es){
9027     printf(2, "leftovers: %s\n", s);
9028     panic("syntax");
9029   }
9030   nulterminate(cmd);
9031   return cmd;
9032 }
9033
9034 struct cmd*
9035 parseline(char **ps, char *es)
9036 {
9037   struct cmd *cmd;
9038
9039   cmd = parsepipe(ps, es);
9040   while(peek(ps, es, "&")){
9041     gettoken(ps, es, 0, 0);
9042     cmd = backcmd(cmd);
9043   }
9044   if(peek(ps, es, ";")){
9045     gettoken(ps, es, 0, 0);
9046     cmd = listcmd(cmd, parseline(ps, es));
9047   }
9048   return cmd;
9049 }
```



```
9050 struct cmd*
9051 parsepipe(char **ps, char *es)
9052 {
9053   struct cmd *cmd;
9054
9055   cmd = parseexec(ps, es);
9056   if(peek(ps, es, "|")){
9057     gettoken(ps, es, 0, 0);
9058     cmd = pipecmd(cmd, parsepipe(ps, es));
9059   }
9060   return cmd;
9061 }
9062
9063 struct cmd*
9064 parseredirs(struct cmd *cmd, char **ps, char *es)
9065 {
9066   int tok;
9067   char *q, *eq;
9068
9069   while(peek(ps, es, "<>")){
9070     tok = gettoken(ps, es, 0, 0);
9071     if(gettoken(ps, es, &q, &eq) != 'a')
9072       panic("missing file for redirection");
9073     switch(tok){
9074     case '<':
9075       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
9076       break;
9077     case '>':
9078       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9079       break;
9080     case '+':  // >>
9081       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9082       break;
9083     }
9084   }
9085   return cmd;
9086 }
9087
9088
9089
9090
9091
9092
9093
9094
9095
9096
9097
9098
9099
```

```
9100 struct cmd*
9101 parseblock(char **ps, char *es)
9102 {
9103   struct cmd *cmd;
9104
9105   if(!peek(ps, es, "("))
9106     panic("parseblock");
9107   gettoken(ps, es, 0, 0);
9108   cmd = parseline(ps, es);
9109   if(!peek(ps, es, ")"))
9110     panic("syntax - missing )");
9111   gettoken(ps, es, 0, 0);
9112   cmd = parseredirs(cmd, ps, es);
9113   return cmd;
9114 }
9115
9116 struct cmd*
9117 parseexec(char **ps, char *es)
9118 {
9119   char *q, *eq;
9120   int tok, argc;
9121   struct execcmd *cmd;
9122   struct cmd *ret;
9123
9124   if(peek(ps, es, "("))
9125     return parseblock(ps, es);
9126
9127   ret = execcmd();
9128   cmd = (struct execcmd*)ret;
9129
9130   argc = 0;
9131   ret = parseredirs(ret, ps, es);
9132   while(!peek(ps, es, "|)&;")){
9133     if((tok=gettoken(ps, es, &q, &eq)) == 0)
9134       break;
9135     if(tok != 'a')
9136       panic("syntax");
9137     cmd->argv[argc] = q;
9138     cmd->eargv[argc] = eq;
9139     argc++;
9140     if(argc >= MAXARGS)
9141       panic("too many args");
9142     ret = parseredirs(ret, ps, es);
9143   }
9144   cmd->argv[argc] = 0;
9145   cmd->eargv[argc] = 0;
9146   return ret;
9147 }
9148
9149
```



```
9150 // NUL-terminate all the counted strings.
9151 struct cmd*
9152 nulterminate(struct cmd *cmd)
9153 {
9154   int i;
9155   struct backcmd *bcmd;
9156   struct execcmd *ecmd;
9157   struct listcmd *lcmd;
9158   struct pipecmd *pcmd;
9159   struct redircmd *rcmd;
9160
9161   if(cmd == 0)
9162     return 0;
9163
9164   switch(cmd->type){
9165   case EXEC:
9166     ecmd = (struct execcmd*)cmd;
9167     for(i=0; ecmd->argv[i]; i++)
9168       *ecmd->eargv[i] = 0;
9169     break;
9170
9171   case REDIR:
9172     rcmd = (struct redircmd*)cmd;
9173     nulterminate(rcmd->cmd);
9174     *rcmd->efile = 0;
9175     break;
9176
9177   case PIPE:
9178     pcmd = (struct pipecmd*)cmd;
9179     nulterminate(pcmd->left);
9180     nulterminate(pcmd->right);
9181     break;
9182
9183   case LIST:
9184     lcmd = (struct listcmd*)cmd;
9185     nulterminate(lcmd->left);
9186     nulterminate(lcmd->right);
9187     break;
9188
9189   case BACK:
9190     bcmd = (struct backcmd*)cmd;
9191     nulterminate(bcmd->cmd);
9192     break;
9193   }
9194   return cmd;
9195 }
9196
9197
9198
9199
```

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

# Start the first CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.code16                       # Assemble for 16-bit mode
.globl start
start:
  cli                         # BIOS enabled interrupts; disable

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax             # Set %ax to zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment

  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60

  # Switch from real to protected mode.  Use a bootstrap GDT that makes
  # virtual addresses map directly to physical addresses so that the
  # effective memory map doesn't change during the transition.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
  movl    %eax, %cr0
```



```
  # Complete transition to 32-bit protected mode by using long jmp
  # to reload %cs and %eip.  The segment descriptors are set up with no
  # translation, so that the mapping is still the identity mapping.
  ljmp    $(SEG_KCODE<<3), $start32

.code32  # Tell assembler to generate 32-bit code now.
start32:
  # Set up the protected-mode data segment registers
  movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %ss                # -> SS: Stack Segment
  movw    $0, %ax                 # Zero segments not ready for use
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS

  # Set up the stack pointer and call into C.
  movl    $start, %esp
  call    bootmain

  # If bootmain returns (it shouldn't), trigger a Bochs
  # breakpoint if running under Bochs, then loop.
  movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
  movw    %ax, %dx
  outw    %ax, %dx
  movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
  outw    %ax, %dx
spin:
  jmp     spin

# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULLASM                             # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg

gdtdesc:
  .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
  .long   gdt                             # address gdt
```

```
9300 // Boot loader.
9301 //
9302 // Part of the boot block, along with bootasm.S, which calls bootmain().
9303 // bootasm.S has put the processor into protected 32-bit mode.
9304 // bootmain() loads an ELF kernel image from the disk starting at
9305 // sector 1 and then jumps to the kernel entry routine.
9306
9307 #include "types.h"
9308 #include "elf.h"
9309 #include "x86.h"
9310 #include "memlayout.h"
9311
9312 #define SECTSIZE  512
9313
9314 void readseg(uchar*, uint, uint);
9315
9316 void
9317 bootmain(void)
9318 {
9319   struct elfhdr *elf;
9320   struct proghdr *ph, *eph;
9321   void (*entry)(void);
9322   uchar* pa;
9323
9324   elf = (struct elfhdr*)0x10000;  // scratch space
9325
9326   // Read 1st page off disk
9327   readseg((uchar*)elf, 4096, 0);
9328
9329   // Is this an ELF executable?
9330   if(elf->magic != ELF_MAGIC)
9331     return;  // let bootasm.S handle error
9332
9333   // Load each program segment (ignores ph flags).
9334   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
9335   eph = ph + elf->phnum;
9336   for(; ph < eph; ph++){
9337     pa = (uchar*)ph->paddr;
9338     readseg(pa, ph->filesz, ph->off);
9339     if(ph->memsz > ph->filesz)
9340       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
9341   }
9342
9343   // Call the entry point from the ELF header.
9344   // Does not return!
9345   entry = (void(*)(void))(elf->entry);
9346   entry();
9347 }
9348
9349
```



```
9350 void
9351 waitdisk(void)
9352 {
9353   // Wait for disk ready.
9354   while((inb(0x1F7) & 0xC0) != 0x40)
9355     ;
9356 }
9357
9358 // Read a single sector at offset into dst.
9359 void
9360 readsect(void *dst, uint offset)
9361 {
9362   // Issue command.
9363   waitdisk();
9364   outb(0x1F2, 1);   // count = 1
9365   outb(0x1F3, offset);
9366   outb(0x1F4, offset >> 8);
9367   outb(0x1F5, offset >> 16);
9368   outb(0x1F6, (offset >> 24) | 0xE0);
9369   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
9370
9371   // Read data.
9372   waitdisk();
9373   insl(0x1F0, dst, SECTSIZE/4);
9374 }
9375
9376 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
9377 // Might copy more than asked.
9378 void
9379 readseg(uchar* pa, uint count, uint offset)
9380 {
9381   uchar* epa;
9382
9383   epa = pa + count;
9384
9385   // Round down to sector boundary.
9386   pa -= offset % SECTSIZE;
9387
9388   // Translate from bytes to sectors; kernel starts at sector 1.
9389   offset = (offset / SECTSIZE) + 1;
9390
9391   // If this is too slow, we could read lots of sectors at a time.
9392   // We'd write more to memory than asked, but it doesn't matter --
9393   // we load in increasing order.
9394   for(; pa < epa; pa += SECTSIZE, offset++)
9395     readsect(pa, offset);
9396 }
9397
9398
9399
```