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

ACKNOWLEDGMENTS

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

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

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

In addition, we are grateful for the patches contributed by Greg
Price, Yandong Mao, and Hitoshi Mitake.

ERROR REPORTS

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

BUILDING AND RUNNING XV6

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

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

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 2308
        0317 2128 2166 2307 2308

indicates that swtch is defined on line 2308 and is mentioned on five lines
on sheets 03, 21, and 23.

```
    3117 3253
sys_exec 5351
    3232 3254 5351
SYS_exec 3109
    3109 3254 7211
sys_exit 3314
    3233 3255 3314
SYS_exit 3102
    3102 3255 7216
sys_fork 3308
    3234 3256 3308
SYS_fork 3101
    3101 3256
sys_fstat 5051
    3235 3257 5051
SYS_fstat 3113
    3113 3257
sys_getpid 3337
    3236 3258 3337
SYS_getpid 3118
    3118 3258
sys_kill 3327
    3237 3259 3327
SYS_kill 3108
    3108 3259
sys_link 5063
    3238 3260 5063
SYS_link 3114
    3114 3260
sys_mkdir 5290
    3239 3261 5290
SYS_mkdir 3115
    3115 3261
sys_mknod 5301
    3240 3262 5301
SYS_mknod 3111
    3111 3262
sys_open 5251
    3241 3263 5251
SYS_open 3110
    3110 3263
sys_pipe 5377
    3242 3264 5377
SYS_pipe 3104
    3104 3264
sys_read 5015
    3243 3265 5015
SYS_read 3106
    3106 3265
sys_sbrk 3351
    3244 3266 3351
SYS_sbrk 3119
    3119 3266
sys_sleep 3365
    3245 3267 3365
SYS_sleep 3120
    3120 3267
sys_unlink 5151
    3246 3268 5151
SYS_unlink 3112
    3112 3268
sys_uptime 3388
    3249 3271 3388
SYS_uptime 3121
    3121 3271
sys_wait 3321
    3247 3269 3321
SYS_wait 3103
    3103 3269
sys_write 5027
    3248 3270 5027
SYS_write 3105
    3105 3270
taskstate 0851
    0851 1664
TDCR 6138
    6138 6164
T_DEV 3502
    3502 4507 4557 5311
T_DIR 3500
    3500 4618 4765 5073 5178
    5187 5229 5267 5295 5326
T_FILE 3501
    3501 5214 5261
ticks 2963
    0352 2963 3017 3018 3373
    3374 3379 3393
tickslock 2962
    0354 2962 2974 3016 3019
    3372 3376 3379 3381 3392
    3394
TICR 6136
    6136 6166
TIMER 6128
    6128 6165
TIMER_16BIT 7071
    7071 7077
TIMER_DIV 7066
    7066 7078 7079
TIMER_FREQ 7065
```



```
    7065 7066
timerinit 7074
    0346 1362 7074
TIMER_MODE 7068
    7068 7077
TIMER_RATEGEN 7070
    7070 7077
TIMER_SEL0 7069
    7069 7077
T_IRQ0 2829
    2829 3014 3023 3027 3031
    3035 3036 3073 6158 6165
    6178 6367 6381 6447 6466
TPR 6112
    6112 6194
trap 3001
    2852 2854 2924 3001 3053
    3055 3058
trapframe 0602
    0602 1728 1880 3001
trapret 2929
    1767 1885 2928 2929
T_SYSCALL 2826
    2826 2972 3003 7212 7217
    7257
tvinit 2966
    0353 1356 2966
uart 7115
    7115 7136 7155 7165
uartgetc 7163
    7163 7175
uartinit 7118
    0357 1322 7118
uartintr 7173
    0358 3032 7173
uartputc 7151
    0359 6895 6897 7147 7151
userinit 1902
    0311 1363 1902 1910
USERTOP 2458
    2458 2592 2707 2759
uva2ka 2655
    0366 2655 5454
VER 6111
    6111 6174
vmenable 2602
    0364 1380 2602
wait 2053
    0312 2053 3323 7262 7333
    7444 7470 7471 7526
waitdisk 1251
    1251 1263 1272
wakeup 2264
    0313 2264 3018 3822 4039
    4341 4366 5616 5619 5641
    5646 5668 6942
wakeup1 2253
    1769 2026 2033 2253 2267
walkpgdir 2504
    2504 2537 2657 2687 2734
    2779
writei 4552
    0249 4552 4674 4932 5185
    5186
xchg 0529
    0529 1383 1532 1569
yield 2172
    0314 2172 3074
```

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



```
0150 #define NPROC        64  // maximum number of processes
0151 #define KSTACKSIZE 4096  // size of per-process kernel stack
0152 #define NCPU          8  // maximum number of CPUs
0153 #define NOFILE       16  // open files per process
0154 #define NFILE       100  // open files per system
0155 #define NBUF         10  // size of disk block cache
0156 #define NINODE       50  // maximum number of active i-nodes
0157 #define NDEV         10  // maximum major device number
0158 #define ROOTDEV       1  // device number of file system root disk
0159 #define PHYSTOP  0x1000000 // use phys mem up to here as free pool
0160
0161
0162
0163
0164
0165
0166
0167
0168
0169
0170
0171
0172
0173
0174
0175
0176
0177
0178
0179
0180
0181
0182
0183
0184
0185
0186
0187
0188
0189
0190
0191
0192
0193
0194
0195
0196
0197
0198
0199
```

```
0200 struct buf;
0201 struct context;
0202 struct file;
0203 struct inode;
0204 struct pipe;
0205 struct proc;
0206 struct spinlock;
0207 struct stat;
0208
0209 // bio.c
0210 void            binit(void);
0211 struct buf*     bread(uint, uint);
0212 void            brelse(struct buf*);
0213 void            bwrite(struct buf*);
0214
0215 // console.c
0216 void            consoleinit(void);
0217 void            cprintf(char*, ...);
0218 void            consoleintr(int(*)(void));
0219 void            panic(char*) __attribute__((noreturn));
0220
0221 // exec.c
0222 int             exec(char*, char**);
0223
0224 // file.c
0225 struct file*    filealloc(void);
0226 void            fileclose(struct file*);
0227 struct file*    filedup(struct file*);
0228 void            fileinit(void);
0229 int             fileread(struct file*, char*, int n);
0230 int             filestat(struct file*, struct stat*);
0231 int             filewrite(struct file*, char*, int n);
0232
0233 // fs.c
0234 int             dirlink(struct inode*, char*, uint);
0235 struct inode*   dirlookup(struct inode*, char*, uint*);
0236 struct inode*   ialloc(uint, short);
0237 struct inode*   idup(struct inode*);
0238 void            iinit(void);
0239 void            ilock(struct inode*);
0240 void            iput(struct inode*);
0241 void            iunlock(struct inode*);
0242 void            iunlockput(struct inode*);
0243 void            iupdate(struct inode*);
0244 int             namecmp(const char*, const char*);
0245 struct inode*   namei(char*);
0246 struct inode*   nameiparent(char*, char*);
0247 int             readi(struct inode*, char*, uint, uint);
0248 void            stati(struct inode*, struct stat*);
0249 int             writei(struct inode*, char*, uint, uint);
```



```
0250 // ide.c
0251 void            ideinit(void);
0252 void            ideintr(void);
0253 void            iderw(struct buf*);
0254
0255 // ioapic.c
0256 void            ioapicenable(int irq, int cpu);
0257 extern uchar    ioapicid;
0258 void            ioapicinit(void);
0259
0260 // kalloc.c
0261 char*           kalloc(void);
0262 void            kfree(char*);
0263 void            kinit();
0264
0265 // kbd.c
0266 void            kbdintr(void);
0267
0268 // lapic.c
0269 int             cpunum(void);
0270 extern volatile uint*    lapic;
0271 void            lapiceoi(void);
0272 void            lapicinit(int);
0273 void            lapicstartap(uchar, uint);
0274 void            microdelay(int);
0275
0276 // mp.c
0277 extern int      ismp;
0278 int             mpbcpu(void);
0279 void            mpinit(void);
0280 void            mpstartthem(void);
0281
0282 // picirq.c
0283 void            picenable(int);
0284 void            picinit(void);
0285
0286 // pipe.c
0287 int             pipealloc(struct file**, struct file**);
0288 void            pipeclose(struct pipe*, int);
0289 int             piperead(struct pipe*, char*, int);
0290 int             pipewrite(struct pipe*, char*, int);
0291
0292
0293
0294
0295
0296
0297
0298
0299
```

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



```
0350 // trap.c
0351 void            idtinit(void);
0352 extern uint     ticks;
0353 void            tvinit(void);
0354 extern struct spinlock tickslock;
0355
0356 // uart.c
0357 void            uartinit(void);
0358 void            uartintr(void);
0359 void            uartputc(int);
0360
0361 // vm.c
0362 void            ksegment(void);
0363 void            kvmalloc(void);
0364 void            vmenable(void);
0365 pde_t*          setupkvm(void);
0366 char*           uva2ka(pde_t*, char*);
0367 int             allocuvm(pde_t*, uint, uint);
0368 int             deallocuvm(pde_t*, uint, uint);
0369 void            freevm(pde_t*);
0370 void            inituvm(pde_t*, char*, uint);
0371 int             loaduvm(pde_t*, char*, struct inode *, uint, uint);
0372 pde_t*          copyuvm(pde_t*,uint);
0373 void            switchuvm(struct proc*);
0374 void            switchkvm();
0375
0376 // number of elements in fixed-size array
0377 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
0378
0379
0380
0381
0382
0383
0384
0385
0386
0387
0388
0389
0390
0391
0392
0393
0394
0395
0396
0397
0398
0399
```

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



```
0450 struct segdesc;
0451
0452 static inline void
0453 lgdt(struct segdesc *p, int size)
0454 {
0455   volatile ushort pd[3];
0456
0457   pd[0] = size-1;
0458   pd[1] = (uint)p;
0459   pd[2] = (uint)p >> 16;
0460
0461   asm volatile("lgdt (%0)" : : "r" (pd));
0462 }
0463
0464 struct gatedesc;
0465
0466 static inline void
0467 lidt(struct gatedesc *p, int size)
0468 {
0469   volatile ushort pd[3];
0470
0471   pd[0] = size-1;
0472   pd[1] = (uint)p;
0473   pd[2] = (uint)p >> 16;
0474
0475   asm volatile("lidt (%0)" : : "r" (pd));
0476 }
0477
0478 static inline void
0479 ltr(ushort sel)
0480 {
0481   asm volatile("ltr %0" : : "r" (sel));
0482 }
0483
0484 static inline uint
0485 readeflags(void)
0486 {
0487   uint eflags;
0488   asm volatile("pushfl; popl %0" : "=r" (eflags));
0489   return eflags;
0490 }
0491
0492 static inline void
0493 loadgs(ushort v)
0494 {
0495   asm volatile("movw %0, %%gs" : : "r" (v));
0496 }
0497
0498
0499
```

```
0500 static inline uint
0501 rebp(void)
0502 {
0503   uint val;
0504   asm volatile("movl %%ebp,%0" : "=r" (val));
0505   return val;
0506 }
0507
0508 static inline uint
0509 resp(void)
0510 {
0511   uint val;
0512   asm volatile("movl %%esp,%0" : "=r" (val));
0513   return val;
0514 }
0515
0516 static inline void
0517 cli(void)
0518 {
0519   asm volatile("cli");
0520 }
0521
0522 static inline void
0523 sti(void)
0524 {
0525   asm volatile("sti");
0526 }
0527
0528 static inline uint
0529 xchg(volatile uint *addr, uint newval)
0530 {
0531   uint result;
0532
0533   // The + in "+m" denotes a read-modify-write operand.
0534   asm volatile("lock; xchgl %0, %1" :
0535                "+m" (*addr), "=a" (result) :
0536                "1" (newval) :
0537                "cc");
0538   return result;
0539 }
0540
0541
0542
0543
0544
0545
0546
0547
0548
0549
```



```
0550 static inline void
0551 lcr0(uint val)
0552 {
0553   asm volatile("movl %0,%%cr0" : : "r" (val));
0554 }
0555
0556 static inline uint
0557 rcr0(void)
0558 {
0559   uint val;
0560   asm volatile("movl %%cr0,%0" : "=r" (val));
0561   return val;
0562 }
0563
0564 static inline uint
0565 rcr2(void)
0566 {
0567   uint val;
0568   asm volatile("movl %%cr2,%0" : "=r" (val));
0569   return val;
0570 }
0571
0572 static inline void
0573 lcr3(uint val)
0574 {
0575   asm volatile("movl %0,%%cr3" : : "r" (val));
0576 }
0577
0578 static inline uint
0579 rcr3(void)
0580 {
0581   uint val;
0582   asm volatile("movl %%cr3,%0" : "=r" (val));
0583   return val;
0584 }
0585
0586
0587
0588
0589
0590
0591
0592
0593
0594
0595
0596
0597
0598
0599
```

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



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

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



```
0750 // Segment Descriptor
0751 struct segdesc {
0752   uint lim_15_0 : 16;  // Low bits of segment limit
0753   uint base_15_0 : 16; // Low bits of segment base address
0754   uint base_23_16 : 8; // Middle bits of segment base address
0755   uint type : 4;       // Segment type (see STS_ constants)
0756   uint s : 1;          // 0 = system, 1 = application
0757   uint dpl : 2;        // Descriptor Privilege Level
0758   uint p : 1;          // Present
0759   uint lim_19_16 : 4;  // High bits of segment limit
0760   uint avl : 1;        // Unused (available for software use)
0761   uint rsv1 : 1;       // Reserved
0762   uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
0763   uint g : 1;          // Granularity: limit scaled by 4K when set
0764   uint base_31_24 : 8; // High bits of segment base address
0765 };
0766
0767 // Normal segment
0768 #define SEG(type, base, lim, dpl) (struct segdesc)    \
0769 { ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff,      \
0770   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
0771   (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
0772 #define SEG16(type, base, lim, dpl) (struct segdesc)  \
0773 { (lim) & 0xffff, (uint)(base) & 0xffff,              \
0774   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
0775   (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
0776
0777 #define DPL_USER    0x3     // User DPL
0778
0779 // Application segment type bits
0780 #define STA_X       0x8     // Executable segment
0781 #define STA_E       0x4     // Expand down (non-executable segments)
0782 #define STA_C       0x4     // Conforming code segment (executable only)
0783 #define STA_W       0x2     // Writeable (non-executable segments)
0784 #define STA_R       0x2     // Readable (executable segments)
0785 #define STA_A       0x1     // Accessed
0786
0787 // System segment type bits
0788 #define STS_T16A    0x1     // Available 16-bit TSS
0789 #define STS_LDT     0x2     // Local Descriptor Table
0790 #define STS_T16B    0x3     // Busy 16-bit TSS
0791 #define STS_CG16    0x4     // 16-bit Call Gate
0792 #define STS_TG      0x5     // Task Gate / Coum Transmitions
0793 #define STS_IG16    0x6     // 16-bit Interrupt Gate
0794 #define STS_TG16    0x7     // 16-bit Trap Gate
0795 #define STS_T32A    0x9     // Available 32-bit TSS
0796 #define STS_T32B    0xB     // Busy 32-bit TSS
0797 #define STS_CG32    0xC     // 32-bit Call Gate
0798 #define STS_IG32    0xE     // 32-bit Interrupt Gate
0799 #define STS_TG32    0xF     // 32-bit Trap Gate
```

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



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

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



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

```
1000 #include "asm.h"
1001
1002 # Start the first CPU: switch to 32-bit protected mode, jump into C.
1003 # The BIOS loads this code from the first sector of the hard disk into
1004 # memory at physical address 0x7c00 and starts executing in real mode
1005 # with %cs=0 %ip=7c00.
1006
1007 #define SEG_KCODE 1  // kernel code
1008 #define SEG_KDATA 2  // kernel data+stack
1009
1010 #define CR0_PE    1  // protected mode enable bit
1011
1012 .code16                       # Assemble for 16-bit mode
1013 .globl start
1014 start:
1015   cli                         # Disable interrupts
1016
1017   # Set up the important data segment registers (DS, ES, SS).
1018   xorw    %ax,%ax             # Segment number zero
1019   movw    %ax,%ds             # -> Data Segment
1020   movw    %ax,%es             # -> Extra Segment
1021   movw    %ax,%ss             # -> Stack Segment
1022
1023   # Enable A20:
1024   #   For backwards compatibility with the earliest PCs, physical
1025   #   address line 20 is tied low, so that addresses higher than
1026   #   1MB wrap around to zero by default.  This code undoes this.
1027 seta20.1:
1028   inb     $0x64,%al               # Wait for not busy
1029   testb   $0x2,%al
1030   jnz     seta20.1
1031
1032   movb    $0xd1,%al               # 0xd1 -> port 0x64
1033   outb    %al,$0x64
1034
1035 seta20.2:
1036   inb     $0x64,%al               # Wait for not busy
1037   testb   $0x2,%al
1038   jnz     seta20.2
1039
1040   movb    $0xdf,%al               # 0xdf -> port 0x60
1041   outb    %al,$0x60
1042
1043
1044
1045
1046
1047
1048
1049
```



```
1050   # Switch from real to protected mode, using a bootstrap GDT
1051   # and segment translation that makes virtual addresses
1052   # identical to physical addresses, so that the
1053   # effective memory map does not change during the switch.
1054   lgdt    gdtdesc
1055   movl    %cr0, %eax
1056   orl     $CR0_PE, %eax
1057   movl    %eax, %cr0
1058
1059   # This ljmp is how you load the CS (Code Segment) register.
1060   # SEG_ASM produces segment descriptors with the 32-bit mode
1061   # flag set (the D flag), so addresses and word operands will
1062   # default to 32 bits after this jump.
1063   ljmp    $(SEG_KCODE<<3), $start32
1064
1065 .code32                       # Assemble for 32-bit mode
1066 start32:
1067   # Set up the protected-mode data segment registers
1068   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
1069   movw    %ax, %ds                # -> DS: Data Segment
1070   movw    %ax, %es                # -> ES: Extra Segment
1071   movw    %ax, %ss                # -> SS: Stack Segment
1072   movw    $0, %ax                 # Zero segments not ready for use
1073   movw    %ax, %fs                # -> FS
1074   movw    %ax, %gs                # -> GS
1075
1076   # Set up the stack pointer and call into C.
1077   movl    $start, %esp
1078   call    bootmain
1079
1080   # If bootmain returns (it shouldn't), trigger a Bochs
1081   # breakpoint if running under Bochs, then loop.
1082   movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
1083   movw    %ax, %dx
1084   outw    %ax, %dx
1085   movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
1086   outw    %ax, %dx
1087 spin:
1088   jmp     spin
1089
1090 # Bootstrap GDT
1091 .p2align 2                                # force 4 byte alignment
1092 gdt:
1093   SEG_NULLASM                             # null seg
1094   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
1095   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
1096
1097 gdtdesc:
1098   .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
1099   .long   gdt                             # address gdt
```

```
1100 #include "asm.h"
1101
1102 # Each non-boot CPU ("AP") is started up in response to a STARTUP
1103 # IPI from the boot CPU.  Section B.4.2 of the Multi-Processor
1104 # Specification says that the AP will start in real mode with CS:IP
1105 # set to XY00:0000, where XY is an 8-bit value sent with the
1106 # STARTUP. Thus this code must start at a 4096-byte boundary.
1107 #
1108 # Because this code sets DS to zero, it must sit
1109 # at an address in the low 2^16 bytes.
1110 #
1111 # Bootothers (in main.c) sends the STARTUPs, one at a time.
1112 # It puts this code (start) at 0x7000.
1113 # It puts the correct %esp in start-4,
1114 # and the place to jump to in start-8.
1115 #
1116 # This code is identical to bootasm.S except:
1117 #   - it does not need to enable A20
1118 #   - it uses the address at start-4 for the %esp
1119 #   - it jumps to the address at start-8 instead of calling bootmain
1120
1121 #define SEG_KCODE 1  // kernel code
1122 #define SEG_KDATA 2  // kernel data+stack
1123
1124 #define CR0_PE    1  // protected mode enable bit
1125
1126 .code16                       # Assemble for 16-bit mode
1127 .globl start
1128 start:
1129   cli                         # Disable interrupts
1130
1131   # Set up the important data segment registers (DS, ES, SS).
1132   xorw    %ax,%ax             # Segment number zero
1133   movw    %ax,%ds             # -> Data Segment
1134   movw    %ax,%es             # -> Extra Segment
1135   movw    %ax,%ss             # -> Stack Segment
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
```



```
1150   # Switch from real to protected mode, using a bootstrap GDT
1151   # and segment translation that makes virtual addresses
1152   # identical to physical addresses, so that the
1153   # effective memory map does not change during the switch.
1154   lgdt    gdtdesc
1155   movl    %cr0, %eax
1156   orl     $CR0_PE, %eax
1157   movl    %eax, %cr0
1158
1159   # This ljmp is how you load the CS (Code Segment) register.
1160   # SEG_ASM produces segment descriptors with the 32-bit mode
1161   # flag set (the D flag), so addresses and word operands will
1162   # default to 32 bits after this jump.
1163   ljmp    $(SEG_KCODE<<3), $start32
1164
1165 .code32                       # Assemble for 32-bit mode
1166 start32:
1167   # Set up the protected-mode data segment registers
1168   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
1169   movw    %ax, %ds                # -> DS: Data Segment
1170   movw    %ax, %es                # -> ES: Extra Segment
1171   movw    %ax, %ss                # -> SS: Stack Segment
1172   movw    $0, %ax                 # Zero segments not ready for use
1173   movw    %ax, %fs                # -> FS
1174   movw    %ax, %gs                # -> GS
1175
1176   # Set up the stack pointer and call into C.
1177   movl    start-4, %esp
1178   call     *(start-8)
1179
1180   # If the call returns (it shouldn't), trigger a Bochs
1181   # breakpoint if running under Bochs, then loop.
1182   movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
1183   movw    %ax, %dx
1184   outw    %ax, %dx
1185   movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
1186   outw    %ax, %dx
1187 spin:
1188   jmp     spin
1189
1190 # Bootstrap GDT
1191 .p2align 2                                # force 4 byte alignment
1192 gdt:
1193   SEG_NULLASM                             # null seg
1194   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
1195   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
1196
1197 gdtdesc:
1198   .word   (gdtdesc - gdt - 1)                           # sizeof(gdt) - 1
1199   .long   gdt                             # address gdt
```

```
1200 // Boot loader.
1201 //
1202 // Part of the boot sector, along with bootasm.S, which calls bootmain().
1203 // bootasm.S has put the processor into protected 32-bit mode.
1204 // bootmain() loads an ELF kernel image from the disk starting at
1205 // sector 1 and then jumps to the kernel entry routine.
1206
1207 #include "types.h"
1208 #include "elf.h"
1209 #include "x86.h"
1210
1211 #define SECTSIZE  512
1212
1213 void readseg(uchar*, uint, uint);
1214
1215 void
1216 bootmain(void)
1217 {
1218   struct elfhdr *elf;
1219   struct proghdr *ph, *eph;
1220   void (*entry)(void);
1221   uchar* va;
1222
1223   elf = (struct elfhdr*)0x10000;  // scratch space
1224
1225   // Read 1st page off disk
1226   readseg((uchar*)elf, 4096, 0);
1227
1228   // Is this an ELF executable?
1229   if(elf->magic != ELF_MAGIC)
1230     return;  // let bootasm.S handle error
1231
1232   // Load each program segment (ignores ph flags).
1233   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
1234   eph = ph + elf->phnum;
1235   for(; ph < eph; ph++) {
1236     va = (uchar*)(ph->va & 0xFFFFFF);
1237     readseg(va, ph->filesz, ph->offset);
1238     if(ph->memsz > ph->filesz)
1239       stosb(va + ph->filesz, 0, ph->memsz - ph->filesz);
1240   }
1241
1242   // Call the entry point from the ELF header.
1243   // Does not return!
1244   entry = (void(*)(void))(elf->entry & 0xFFFFFF);
1245   entry();
1246 }
1247
1248
1249
```



```
1250 void
1251 waitdisk(void)
1252 {
1253   // Wait for disk ready.
1254   while((inb(0x1F7) & 0xC0) != 0x40)
1255     ;
1256 }
1257
1258 // Read a single sector at offset into dst.
1259 void
1260 readsect(void *dst, uint offset)
1261 {
1262   // Issue command.
1263   waitdisk();
1264   outb(0x1F2, 1);   // count = 1
1265   outb(0x1F3, offset);
1266   outb(0x1F4, offset >> 8);
1267   outb(0x1F5, offset >> 16);
1268   outb(0x1F6, (offset >> 24) | 0xE0);
1269   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
1270
1271   // Read data.
1272   waitdisk();
1273   insl(0x1F0, dst, SECTSIZE/4);
1274 }
1275
1276 // Read 'count' bytes at 'offset' from kernel into virtual address 'va'.
1277 // Might copy more than asked.
1278 void
1279 readseg(uchar* va, uint count, uint offset)
1280 {
1281   uchar* eva;
1282
1283   eva = va + count;
1284
1285   // Round down to sector boundary.
1286   va -= offset % SECTSIZE;
1287
1288   // Translate from bytes to sectors; kernel starts at sector 1.
1289   offset = (offset / SECTSIZE) + 1;
1290
1291   // If this is too slow, we could read lots of sectors at a time.
1292   // We'd write more to memory than asked, but it doesn't matter --
1293   // we load in increasing order.
1294   for(; va < eva; va += SECTSIZE, offset++)
1295     readsect(va, offset);
1296 }
1297
1298
1299
```

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

static void bootothers(void);
static void mpmain(void);
void jkstack(void)  __attribute__((noreturn));
void mainc(void);

// Bootstrap processor starts running C code here.
int
main(void)
{
  mpinit();        // collect info about this machine
  lapicinit(mpbcpu());
  ksegment();      // set up segments
  picinit();       // interrupt controller
  ioapicinit();    // another interrupt controller
  consoleinit();   // I/O devices & their interrupts
  uartinit();      // serial port
  kinit();         // initialize memory allocator
  jkstack();       // call mainc() on a properly-allocated stack
}

void
jkstack(void)
{
  char *kstack = kalloc();
  if(!kstack)
    panic("jkstack\n");
  char *top = kstack + PGSIZE;
  asm volatile("movl %0,%%esp" : : "r" (top));
  asm volatile("call mainc");
  panic("jkstack");
}
```



```
void
mainc(void)
{
  cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
  kvmalloc();      // initialize the kernel page table
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  iinit();         // inode cache
  ideinit();       // disk
  if(!ismp)
    timerinit();   // uniprocessor timer
  userinit();      // first user process
  bootothers();    // start other processors

  // Finish setting up this processor in mpmain.
  mpmain();
}

// Common CPU setup code.
// Bootstrap CPU comes here from mainc().
// Other CPUs jump here from bootother.S.
static void
mpmain(void)
{
  if(cpunum() != mpbcpu()) {
    ksegment();
    lapicinit(cpunum());
  }
  vmenable();        // turn on paging
  cprintf("cpu%d: starting\n", cpu->id);
  idtinit();       // load idt register
  xchg(&cpu->booted, 1);
  scheduler();     // start running processes
}
```

```
1400 static void
1401 bootothers(void)
1402 {
1403   extern uchar _binary_bootother_start[], _binary_bootother_size[];
1404   uchar *code;
1405   struct cpu *c;
1406   char *stack;
1407
1408   // Write bootstrap code to unused memory at 0x7000.  The linker has
1409   // placed the start of bootother.S there.
1410   code = (uchar *) 0x7000;
1411   memmove(code, _binary_bootother_start, (uint)_binary_bootother_size);
1412
1413   for(c = cpus; c < cpus+ncpu; c++){
1414     if(c == cpus+cpunum())  // We've started already.
1415       continue;
1416
1417     // Fill in %esp, %eip and start code on cpu.
1418     stack = kalloc();
1419     *(void**)(code-4) = stack + KSTACKSIZE;
1420     *(void**)(code-8) = mpmain;
1421     lapicstartap(c->id, (uint)code);
1422
1423     // Wait for cpu to finish mpmain()
1424     while(c->booted == 0)
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 // Mutual exclusion lock.
1451 struct spinlock {
1452   uint locked;       // Is the lock held?
1453
1454   // For debugging:
1455   char *name;        // Name of lock.
1456   struct cpu *cpu;   // The cpu holding the lock.
1457   uint pcs[10];      // The call stack (an array of program counters)
1458                      // that locked the lock.
1459 };
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
```

```
1500 // Mutual exclusion spin locks.
1501
1502 #include "types.h"
1503 #include "defs.h"
1504 #include "param.h"
1505 #include "x86.h"
1506 #include "mmu.h"
1507 #include "proc.h"
1508 #include "spinlock.h"
1509
1510 void
1511 initlock(struct spinlock *lk, char *name)
1512 {
1513   lk->name = name;
1514   lk->locked = 0;
1515   lk->cpu = 0;
1516 }
1517
1518 // Acquire the lock.
1519 // Loops (spins) until the lock is acquired.
1520 // Holding a lock for a long time may cause
1521 // other CPUs to waste time spinning to acquire it.
1522 void
1523 acquire(struct spinlock *lk)
1524 {
1525   pushcli();
1526   if(holding(lk))
1527     panic("acquire");
1528
1529   // The xchg is atomic.
1530   // It also serializes, so that reads after acquire are not
1531   // reordered before it.
1532   while(xchg(&lk->locked, 1) != 0)
1533     ;
1534
1535   // Record info about lock acquisition for debugging.
1536   lk->cpu = cpu;
1537   getcallerpcs(&lk, lk->pcs);
1538 }
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
```



```
1550 // Release the lock.
1551 void
1552 release(struct spinlock *lk)
1553 {
1554   if(!holding(lk))
1555     panic("release");
1556
1557   lk->pcs[0] = 0;
1558   lk->cpu = 0;
1559
1560   // The xchg serializes, so that reads before release are
1561   // not reordered after it.  The 1996 PentiumPro manual (Volume 3,
1562   // 7.2) says reads can be carried out speculatively and in
1563   // any order, which implies we need to serialize here.
1564   // But the 2007 Intel 64 Architecture Memory Ordering White
1565   // Paper says that Intel 64 and IA-32 will not move a load
1566   // after a store. So lock->locked = 0 would work here.
1567   // The xchg being asm volatile ensures gcc emits it after
1568   // the above assignments (and after the critical section).
1569   xchg(&lk->locked, 0);
1570
1571   popcli();
1572 }
1573
1574 // Record the current call stack in pcs[] by following the %ebp chain.
1575 void
1576 getcallerpcs(void *v, uint pcs[])
1577 {
1578   uint *ebp;
1579   int i;
1580
1581   ebp = (uint*)v - 2;
1582   for(i = 0; i < 10; i++){
1583     if(ebp == 0 || ebp < (uint *) 0x100000 || ebp == (uint*)0xffffffff)
1584       break;
1585     pcs[i] = ebp[1];     // saved %eip
1586     ebp = (uint*)ebp[0]; // saved %ebp
1587   }
1588   for(; i < 10; i++)
1589     pcs[i] = 0;
1590 }
1591
1592 // Check whether this cpu is holding the lock.
1593 int
1594 holding(struct spinlock *lock)
1595 {
1596   return lock->locked && lock->cpu == cpu;
1597 }
1598
1599
```

```
1600 // Pushcli/popcli are like cli/sti except that they are matched:
1601 // it takes two popcli to undo two pushcli.  Also, if interrupts
1602 // are off, then pushcli, popcli leaves them off.
1603
1604 void
1605 pushcli(void)
1606 {
1607   int eflags;
1608
1609   eflags = readeflags();
1610   cli();
1611   if(cpu->ncli++ == 0)
1612     cpu->intena = eflags & FL_IF;
1613 }
1614
1615 void
1616 popcli(void)
1617 {
1618   if(readeflags()&FL_IF)
1619     panic("popcli - interruptible");
1620   if(--cpu->ncli < 0)
1621     panic("popcli");
1622   if(cpu->ncli == 0 && cpu->intena)
1623     sti();
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 // Segments in proc->gdt.
1651 // Also known to bootasm.S and trapasm.S
1652 #define SEG_KCODE 1  // kernel code
1653 #define SEG_KDATA 2  // kernel data+stack
1654 #define SEG_KCPU  3  // kernel per-cpu data
1655 #define SEG_UCODE 4  // user code
1656 #define SEG_UDATA 5  // user data+stack
1657 #define SEG_TSS   6  // this process's task state
1658 #define NSEGS     7
1659
1660 // Per-CPU state
1661 struct cpu {
1662   uchar id;                    // Local APIC ID; index into cpus[] below
1663   struct context *scheduler;   // Switch here to enter scheduler
1664   struct taskstate ts;         // Used by x86 to find stack for interrupt
1665   struct segdesc gdt[NSEGS];   // x86 global descriptor table
1666   volatile uint booted;        // Has the CPU started?
1667   int ncli;                    // Depth of pushcli nesting.
1668   int intena;                  // Were interrupts enabled before pushcli?
1669
1670   // Cpu-local storage variables; see below
1671   struct cpu *cpu;
1672   struct proc *proc;
1673 };
1674
1675 extern struct cpu cpus[NCPU];
1676 extern int ncpu;
1677
1678 // Per-CPU variables, holding pointers to the
1679 // current cpu and to the current process.
1680 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
1681 // and "%gs:4" to refer to proc.  ksegment sets up the
1682 // %gs segment register so that %gs refers to the memory
1683 // holding those two variables in the local cpu's struct cpu.
1684 // This is similar to how thread-local variables are implemented
1685 // in thread libraries such as Linux pthreads.
1686 extern struct cpu *cpu asm("%gs:0");       // This cpu.
1687 extern struct proc *proc asm("%gs:4");     // Current proc on this cpu.
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
```

```
1700 // Saved registers for kernel context switches.
1701 // Don't need to save all the segment registers (%cs, etc),
1702 // because they are constant across kernel contexts.
1703 // Don't need to save %eax, %ecx, %edx, because the
1704 // x86 convention is that the caller has saved them.
1705 // Contexts are stored at the bottom of the stack they
1706 // describe; the stack pointer is the address of the context.
1707 // The layout of the context matches the layout of the stack in swtch.S
1708 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
1709 // but it is on the stack and allocproc() manipulates it.
1710 struct context {
1711   uint edi;
1712   uint esi;
1713   uint ebx;
1714   uint ebp;
1715   uint eip;
1716 };
1717
1718 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
1719
1720 // Per-process state
1721 struct proc {
1722   uint sz;                     // Size of process memory (bytes)
1723   pde_t* pgdir;                // Linear address of proc's pgdir
1724   char *kstack;                // Bottom of kernel stack for this process
1725   enum procstate state;        // Process state
1726   volatile int pid;            // Process ID
1727   struct proc *parent;         // Parent process
1728   struct trapframe *tf;        // Trap frame for current syscall
1729   struct context *context;     // Switch here to run process
1730   void *chan;                  // If non-zero, sleeping on chan
1731   int killed;                  // If non-zero, have been killed
1732   struct file *ofile[NOFILE];  // Open files
1733   struct inode *cwd;           // Current directory
1734   char name[16];               // Process name (debugging)
1735 };
1736
1737 // Process memory is laid out contiguously, low addresses first:
1738 //   text
1739 //   original data and bss
1740 //   fixed-size stack
1741 //   expandable heap
1742
1743
1744
1745
1746
1747
1748
1749
```



```
1750 #include "types.h"
1751 #include "defs.h"
1752 #include "param.h"
1753 #include "mmu.h"
1754 #include "x86.h"
1755 #include "proc.h"
1756 #include "spinlock.h"
1757
1758 struct {
1759   struct spinlock lock;
1760   struct proc proc[NPROC];
1761 } ptable;
1762
1763 static struct proc *initproc;
1764
1765 int nextpid = 1;
1766 extern void forkret(void);
1767 extern void trapret(void);
1768
1769 static void wakeup1(void *chan);
1770
1771 void
1772 pinit(void)
1773 {
1774   initlock(&ptable.lock, "ptable");
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 // Print a process listing to console.  For debugging.
1801 // Runs when user types ^P on console.
1802 // No lock to avoid wedging a stuck machine further.
1803 void
1804 procdump(void)
1805 {
1806   static char *states[] = {
1807   [UNUSED]    "unused",
1808   [EMBRYO]    "embryo",
1809   [SLEEPING]  "sleep ",
1810   [RUNNABLE]  "runble",
1811   [RUNNING]   "run   ",
1812   [ZOMBIE]    "zombie"
1813   };
1814   int i;
1815   struct proc *p;
1816   char *state;
1817   uint pc[10];
1818
1819   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
1820     if(p->state == UNUSED)
1821       continue;
1822     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
1823       state = states[p->state];
1824     else
1825       state = "???";
1826     cprintf("%d %s %s", p->pid, state, p->name);
1827     if(p->state == SLEEPING){
1828       getcallerpcs((uint*)p->context->ebp+2, pc);
1829       for(i=0; i<10 && pc[i] != 0; i++)
1830         cprintf(" %p", pc[i]);
1831     }
1832     cprintf("\n");
1833   }
1834 }
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
```



```
1850 // Look in the process table for an UNUSED proc.
1851 // If found, change state to EMBRYO and return it.
1852 // Otherwise return 0.
1853 static struct proc*
1854 allocproc(void)
1855 {
1856   struct proc *p;
1857   char *sp;
1858
1859   acquire(&ptable.lock);
1860   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
1861     if(p->state == UNUSED)
1862       goto found;
1863   release(&ptable.lock);
1864   return 0;
1865
1866 found:
1867   p->state = EMBRYO;
1868   p->pid = nextpid++;
1869   release(&ptable.lock);
1870
1871   // Allocate kernel stack if possible.
1872   if((p->kstack = kalloc()) == 0){
1873     p->state = UNUSED;
1874     return 0;
1875   }
1876   sp = p->kstack + KSTACKSIZE;
1877
1878   // Leave room for trap frame.
1879   sp -= sizeof *p->tf;
1880   p->tf = (struct trapframe*)sp;
1881
1882   // Set up new context to start executing at forkret,
1883   // which returns to trapret (see below).
1884   sp -= 4;
1885   *(uint*)sp = (uint)trapret;
1886
1887   sp -= sizeof *p->context;
1888   p->context = (struct context*)sp;
1889   memset(p->context, 0, sizeof *p->context);
1890   p->context->eip = (uint)forkret;
1891   return p;
1892 }
1893
1894
1895
1896
1897
1898
1899
```

```
1900 // Set up first user process.
1901 void
1902 userinit(void)
1903 {
1904   struct proc *p;
1905   extern char _binary_initcode_start[], _binary_initcode_size[];
1906
1907   p = allocproc();
1908   initproc = p;
1909   if(!(p->pgdir = setupkvm()))
1910     panic("userinit: out of memory?");
1911   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
1912   p->sz = PGSIZE;
1913   memset(p->tf, 0, sizeof(*p->tf));
1914   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
1915   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
1916   p->tf->es = p->tf->ds;
1917   p->tf->ss = p->tf->ds;
1918   p->tf->eflags = FL_IF;
1919   p->tf->esp = PGSIZE;
1920   p->tf->eip = 0;  // beginning of initcode.S
1921
1922   safestrcpy(p->name, "initcode", sizeof(p->name));
1923   p->cwd = namei("/");
1924
1925   p->state = RUNNABLE;
1926 }
1927
1928 // Grow current process's memory by n bytes.
1929 // Return 0 on success, -1 on failure.
1930 int
1931 growproc(int n)
1932 {
1933   uint sz = proc->sz;
1934   if(n > 0){
1935     if(!(sz = allocuvm(proc->pgdir, sz, sz + n)))
1936       return -1;
1937   } else if(n < 0){
1938     if(!(sz = deallocuvm(proc->pgdir, sz, sz + n)))
1939       return -1;
1940   }
1941   proc->sz = sz;
1942   switchuvm(proc);
1943   return 0;
1944 }
1945
1946
1947
1948
1949
```



```
1950 // Create a new process copying p as the parent.
1951 // Sets up stack to return as if from system call.
1952 // Caller must set state of returned proc to RUNNABLE.
1953 int
1954 fork(void)
1955 {
1956   int i, pid;
1957   struct proc *np;
1958
1959   // Allocate process.
1960   if((np = allocproc()) == 0)
1961     return -1;
1962
1963   // Copy process state from p.
1964   if(!(np->pgdir = copyuvm(proc->pgdir, proc->sz))){
1965     kfree(np->kstack);
1966     np->kstack = 0;
1967     np->state = UNUSED;
1968     return -1;
1969   }
1970   np->sz = proc->sz;
1971   np->parent = proc;
1972   *np->tf = *proc->tf;
1973
1974   // Clear %eax so that fork returns 0 in the child.
1975   np->tf->eax = 0;
1976
1977   for(i = 0; i < NOFILE; i++)
1978     if(proc->ofile[i])
1979       np->ofile[i] = filedup(proc->ofile[i]);
1980   np->cwd = idup(proc->cwd);
1981
1982   pid = np->pid;
1983   np->state = RUNNABLE;
1984   safestrcpy(np->name, proc->name, sizeof(proc->name));
1985   return pid;
1986 }
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
```

```
2000 // Exit the current process.  Does not return.
2001 // An exited process remains in the zombie state
2002 // until its parent calls wait() to find out it exited.
2003 void
2004 exit(void)
2005 {
2006   struct proc *p;
2007   int fd;
2008
2009   if(proc == initproc)
2010     panic("init exiting");
2011
2012   // Close all open files.
2013   for(fd = 0; fd < NOFILE; fd++){
2014     if(proc->ofile[fd]){
2015       fileclose(proc->ofile[fd]);
2016       proc->ofile[fd] = 0;
2017     }
2018   }
2019
2020   iput(proc->cwd);
2021   proc->cwd = 0;
2022
2023   acquire(&ptable.lock);
2024
2025   // Parent might be sleeping in wait().
2026   wakeup1(proc->parent);
2027
2028   // Pass abandoned children to init.
2029   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2030     if(p->parent == proc){
2031       p->parent = initproc;
2032       if(p->state == ZOMBIE)
2033         wakeup1(initproc);
2034     }
2035   }
2036
2037   // Jump into the scheduler, never to return.
2038   proc->state = ZOMBIE;
2039   sched();
2040   panic("zombie exit");
2041 }
2042
2043
2044
2045
2046
2047
2048
2049
```



```
2050 // Wait for a child process to exit and return its pid.
2051 // Return -1 if this process has no children.
2052 int
2053 wait(void)
2054 {
2055   struct proc *p;
2056   int havekids, pid;
2057
2058   acquire(&ptable.lock);
2059   for(;;){
2060     // Scan through table looking for zombie children.
2061     havekids = 0;
2062     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2063       if(p->parent != proc)
2064         continue;
2065       havekids = 1;
2066       if(p->state == ZOMBIE){
2067         // Found one.
2068         pid = p->pid;
2069         kfree(p->kstack);
2070         p->kstack = 0;
2071         freevm(p->pgdir);
2072         p->state = UNUSED;
2073         p->pid = 0;
2074         p->parent = 0;
2075         p->name[0] = 0;
2076         p->killed = 0;
2077         release(&ptable.lock);
2078         return pid;
2079       }
2080     }
2081
2082     // No point waiting if we don't have any children.
2083     if(!havekids || proc->killed){
2084       release(&ptable.lock);
2085       return -1;
2086     }
2087
2088     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2089     sleep(proc, &ptable.lock);
2090   }
2091 }
2092
2093
2094
2095
2096
2097
2098
2099
```

```
2100 // Per-CPU process scheduler.
2101 // Each CPU calls scheduler() after setting itself up.
2102 // Scheduler never returns.  It loops, doing:
2103 //  - choose a process to run
2104 //  - swtch to start running that process
2105 //  - eventually that process transfers control
2106 //      via swtch back to the scheduler.
2107 void
2108 scheduler(void)
2109 {
2110   struct proc *p;
2111
2112   for(;;){
2113     // Enable interrupts on this processor.
2114     sti();
2115
2116     // Loop over process table looking for process to run.
2117     acquire(&ptable.lock);
2118     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2119       if(p->state != RUNNABLE)
2120         continue;
2121
2122       // Switch to chosen process.  It is the process's job
2123       // to release ptable.lock and then reacquire it
2124       // before jumping back to us.
2125       proc = p;
2126       switchuvm(p);
2127       p->state = RUNNING;
2128       swtch(&cpu->scheduler, proc->context);
2129       switchkvm();
2130
2131       // Process is done running for now.
2132       // It should have changed its p->state before coming back.
2133       proc = 0;
2134     }
2135     release(&ptable.lock);
2136
2137   }
2138 }
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
```



```
2150 // Enter scheduler.  Must hold only ptable.lock
2151 // and have changed proc->state.
2152 void
2153 sched(void)
2154 {
2155   int intena;
2156
2157   if(!holding(&ptable.lock))
2158     panic("sched ptable.lock");
2159   if(cpu->ncli != 1)
2160     panic("sched locks");
2161   if(proc->state == RUNNING)
2162     panic("sched running");
2163   if(readeflags()&FL_IF)
2164     panic("sched interruptible");
2165   intena = cpu->intena;
2166   swtch(&proc->context, cpu->scheduler);
2167   cpu->intena = intena;
2168 }
2169
2170 // Give up the CPU for one scheduling round.
2171 void
2172 yield(void)
2173 {
2174   acquire(&ptable.lock);
2175   proc->state = RUNNABLE;
2176   sched();
2177   release(&ptable.lock);
2178 }
2179
2180 // A fork child's very first scheduling by scheduler()
2181 // will swtch here.  "Return" to user space.
2182 void
2183 forkret(void)
2184 {
2185   // Still holding ptable.lock from scheduler.
2186   release(&ptable.lock);
2187
2188   // Return to "caller", actually trapret (see allocproc).
2189 }
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
```

```
2200 // Atomically release lock and sleep on chan.
2201 // Reacquires lock when awakened.
2202 void
2203 sleep(void *chan, struct spinlock *lk)
2204 {
2205   if(proc == 0)
2206     panic("sleep");
2207
2208   if(lk == 0)
2209     panic("sleep without lk");
2210
2211   // Must acquire ptable.lock in order to
2212   // change p->state and then call sched.
2213   // Once we hold ptable.lock, we can be
2214   // guaranteed that we won't miss any wakeup
2215   // (wakeup runs with ptable.lock locked),
2216   // so it's okay to release lk.
2217   if(lk != &ptable.lock){
2218     acquire(&ptable.lock);
2219     release(lk);
2220   }
2221
2222   // Go to sleep.
2223   proc->chan = chan;
2224   proc->state = SLEEPING;
2225   sched();
2226
2227   // Tidy up.
2228   proc->chan = 0;
2229
2230   // Reacquire original lock.
2231   if(lk != &ptable.lock){
2232     release(&ptable.lock);
2233     acquire(lk);
2234   }
2235 }
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
```



```
2250 // Wake up all processes sleeping on chan.
2251 // The ptable lock must be held.
2252 static void
2253 wakeup1(void *chan)
2254 {
2255   struct proc *p;
2256
2257   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2258     if(p->state == SLEEPING && p->chan == chan)
2259       p->state = RUNNABLE;
2260 }
2261
2262 // Wake up all processes sleeping on chan.
2263 void
2264 wakeup(void *chan)
2265 {
2266   acquire(&ptable.lock);
2267   wakeup1(chan);
2268   release(&ptable.lock);
2269 }
2270
2271 // Kill the process with the given pid.
2272 // Process won't exit until it returns
2273 // to user space (see trap in trap.c).
2274 int
2275 kill(int pid)
2276 {
2277   struct proc *p;
2278
2279   acquire(&ptable.lock);
2280   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2281     if(p->pid == pid){
2282       p->killed = 1;
2283       // Wake process from sleep if necessary.
2284       if(p->state == SLEEPING)
2285         p->state = RUNNABLE;
2286       release(&ptable.lock);
2287       return 0;
2288     }
2289   }
2290   release(&ptable.lock);
2291   return -1;
2292 }
2293
2294
2295
2296
2297
2298
2299
```

```
2300 # Context switch
2301 #
2302 #   void swtch(struct context **old, struct context *new);
2303 #
2304 # Save current register context in old
2305 # and then load register context from new.
2306
2307 .globl swtch
2308 swtch:
2309   movl 4(%esp), %eax
2310   movl 8(%esp), %edx
2311
2312   # Save old callee-save registers
2313   pushl %ebp
2314   pushl %ebx
2315   pushl %esi
2316   pushl %edi
2317
2318   # Switch stacks
2319   movl %esp, (%eax)
2320   movl %edx, %esp
2321
2322   # Load new callee-save registers
2323   popl %edi
2324   popl %esi
2325   popl %ebx
2326   popl %ebp
2327   ret
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
```



```
2350 // Physical memory allocator, intended to allocate
2351 // memory for user processes, kernel stacks, page table pages,
2352 // and pipe buffers. Allocates 4096-byte pages.
2353
2354 #include "types.h"
2355 #include "defs.h"
2356 #include "param.h"
2357 #include "mmu.h"
2358 #include "spinlock.h"
2359
2360 struct run {
2361   struct run *next;
2362 };
2363
2364 struct {
2365   struct spinlock lock;
2366   struct run *freelist;
2367 } kmem;
2368
2369 // Initialize free list of physical pages.
2370 void
2371 kinit(void)
2372 {
2373   extern char end[];
2374
2375   initlock(&kmem.lock, "kmem");
2376   char *p = (char*)PGROUNDUP((uint)end);
2377   for( ; p + PGSIZE - 1 < (char*) PHYSTOP; p += PGSIZE)
2378     kfree(p);
2379 }
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
```

```
2400 // Free the page of physical memory pointed at by v,
2401 // which normally should have been returned by a
2402 // call to kalloc().  (The exception is when
2403 // initializing the allocator; see kinit above.)
2404 void
2405 kfree(char *v)
2406 {
2407   struct run *r;
2408
2409   if(((uint) v) % PGSIZE || (uint)v < 1024*1024 || (uint)v >= PHYSTOP)
2410     panic("kfree");
2411
2412   // Fill with junk to catch dangling refs.
2413   memset(v, 1, PGSIZE);
2414
2415   acquire(&kmem.lock);
2416   r = (struct run *) v;
2417   r->next = kmem.freelist;
2418   kmem.freelist = r;
2419   release(&kmem.lock);
2420 }
2421
2422 // Allocate one 4096-byte page of physical memory.
2423 // Returns a pointer that the kernel can use.
2424 // Returns 0 if the memory cannot be allocated.
2425 char*
2426 kalloc()
2427 {
2428   struct run *r;
2429
2430   acquire(&kmem.lock);
2431   r = kmem.freelist;
2432   if(r)
2433     kmem.freelist = r->next;
2434   release(&kmem.lock);
2435   return (char*) r;
2436 }
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
```



```
2450 #include "param.h"
2451 #include "types.h"
2452 #include "defs.h"
2453 #include "x86.h"
2454 #include "mmu.h"
2455 #include "proc.h"
2456 #include "elf.h"
2457
2458 #define USERTOP  0xA0000
2459
2460 static pde_t *kpgdir;  // for use in scheduler()
2461
2462 // Set up CPU's kernel segment descriptors.
2463 // Run once at boot time on each CPU.
2464 void
2465 ksegment(void)
2466 {
2467   struct cpu *c;
2468
2469   // Map virtual addresses to linear addresses using identity map.
2470   // Cannot share a CODE descriptor for both kernel and user
2471   // because it would have to have DPL_USR, but the CPU forbids
2472   // an interrupt from CPL=0 to DPL=3.
2473   c = &cpus[cpunum()];
2474   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
2475   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
2476   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
2477   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
2478
2479   // Map cpu, and curproc
2480   c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
2481
2482   lgdt(c->gdt, sizeof(c->gdt));
2483   loadgs(SEG_KCPU << 3);
2484
2485   // Initialize cpu-local storage.
2486   cpu = c;
2487   proc = 0;
2488 }
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
```

```
2500 // Return the address of the PTE in page table pgdir
2501 // that corresponds to linear address va.  If create!=0,
2502 // create any required page table pages.
2503 static pte_t *
2504 walkpgdir(pde_t *pgdir, const void *va, int create)
2505 {
2506   uint r;
2507   pde_t *pde;
2508   pte_t *pgtab;
2509
2510   pde = &pgdir[PDX(va)];
2511   if(*pde & PTE_P){
2512     pgtab = (pte_t*) PTE_ADDR(*pde);
2513   } else if(!create || !(r = (uint) kalloc()))
2514     return 0;
2515   else {
2516     pgtab = (pte_t*) r;
2517     // Make sure all those PTE_P bits are zero.
2518     memset(pgtab, 0, PGSIZE);
2519     // The permissions here are overly generous, but they can
2520     // be further restricted by the permissions in the page table
2521     // entries, if necessary.
2522     *pde = PADDR(r) | PTE_P | PTE_W | PTE_U;
2523   }
2524   return &pgtab[PTX(va)];
2525 }
2526
2527 // Create PTEs for linear addresses starting at la that refer to
2528 // physical addresses starting at pa. la and size might not
2529 // be page-aligned.
2530 static int
2531 mappages(pde_t *pgdir, void *la, uint size, uint pa, int perm)
2532 {
2533   char *a = PGROUNDDOWN(la);
2534   char *last = PGROUNDDOWN(la + size - 1);
2535
2536   while(1){
2537     pte_t *pte = walkpgdir(pgdir, a, 1);
2538     if(pte == 0)
2539       return 0;
2540     if(*pte & PTE_P)
2541       panic("remap");
2542     *pte = pa | perm | PTE_P;
2543     if(a == last)
2544       break;
2545     a += PGSIZE;
2546     pa += PGSIZE;
2547   }
2548   return 1;
2549 }
```



```
2550 // The mappings from logical to linear are one to one (i.e.,
2551 // segmentation doesn't do anything).
2552 // There is one page table per process, plus one that's used
2553 // when a CPU is not running any process (kpgdir).
2554 // A user process uses the same page table as the kernel; the
2555 // page protection bits prevent it from using anything other
2556 // than its memory.
2557 //
2558 // setupkvm() and exec() set up every page table like this:
2559 //   0..640K          : user memory (text, data, stack, heap)
2560 //   640K..1M         : mapped direct (for IO space)
2561 //   1M..end          : mapped direct (for the kernel's text and data)
2562 //   end..PHYSTOP     : mapped direct (kernel heap and user pages)
2563 //   0xfe000000..0    : mapped direct (devices such as ioapic)
2564 //
2565 // The kernel allocates memory for its heap and for user memory
2566 // between kernend and the end of physical memory (PHYSTOP).
2567 // The virtual address space of each user program includes the kernel
2568 // (which is inaccessible in user mode).  The user program addresses
2569 // range from 0 till 640KB (USERTOP), which where the I/O hole starts
2570 // (both in physical memory and in the kernel's virtual address
2571 // space).
2572
2573 // Allocate one page table for the machine for the kernel address
2574 // space for scheduler processes.
2575 void
2576 kvmalloc(void)
2577 {
2578   kpgdir = setupkvm();
2579 }
2580
2581 // Set up kernel part of a page table.
2582 pde_t*
2583 setupkvm(void)
2584 {
2585   pde_t *pgdir;
2586
2587   // Allocate page directory
2588   if(!(pgdir = (pde_t *) kalloc()))
2589     return 0;
2590   memset(pgdir, 0, PGSIZE);
2591   if(// Map IO space from 640K to 1Mbyte
2592      !mappages(pgdir, (void *)USERTOP, 0x60000, USERTOP, PTE_W) ||
2593      // Map kernel and free memory pool
2594      !mappages(pgdir, (void *)0x100000, PHYSTOP-0x100000, 0x100000, PTE_W) |
2595      // Map devices such as ioapic, lapic, ...
2596      !mappages(pgdir, (void *)0xFE000000, 0x2000000, 0xFE000000, PTE_W))
2597     return 0;
2598   return pgdir;
2599 }
```

```
2600 // Turn on paging.
2601 void
2602 vmenable(void)
2603 {
2604   uint cr0;
2605
2606   switchkvm(); // load kpgdir into cr3
2607   cr0 = rcr0();
2608   cr0 |= CR0_PG;
2609   lcr0(cr0);
2610 }
2611
2612 // Switch h/w page table register to the kernel-only page table,
2613 // for when no process is running.
2614 void
2615 switchkvm()
2616 {
2617   lcr3(PADDR(kpgdir));   // switch to the kernel page table
2618 }
2619
2620 // Switch h/w page table and TSS registers to point to process p.
2621 void
2622 switchuvm(struct proc *p)
2623 {
2624   pushcli();
2625
2626   // Setup TSS
2627   cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
2628   cpu->gdt[SEG_TSS].s = 0;
2629   cpu->ts.ss0 = SEG_KDATA << 3;
2630   cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
2631   ltr(SEG_TSS << 3);
2632
2633   if(p->pgdir == 0)
2634     panic("switchuvm: no pgdir\n");
2635
2636   lcr3(PADDR(p->pgdir));  // switch to new address space
2637   popcli();
2638 }
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
```



```
2650 // Return the physical address that a given user address
2651 // maps to.  The result is also a kernel logical address,
2652 // since the kernel maps the physical memory allocated to user
2653 // processes directly.
2654 char*
2655 uva2ka(pde_t *pgdir, char *uva)
2656 {
2657   pte_t *pte = walkpgdir(pgdir, uva, 0);
2658   if(pte == 0) return 0;
2659   uint pa = PTE_ADDR(*pte);
2660   return (char *)pa;
2661 }
2662
2663 // Load the initcode into address 0 of pgdir.
2664 // sz must be less than a page.
2665 void
2666 inituvm(pde_t *pgdir, char *init, uint sz)
2667 {
2668   char *mem = kalloc();
2669   if (sz >= PGSIZE)
2670     panic("inituvm: more than a page");
2671   memset(mem, 0, PGSIZE);
2672   mappages(pgdir, 0, PGSIZE, PADDR(mem), PTE_W|PTE_U);
2673   memmove(mem, init, sz);
2674 }
2675
2676 // Load a program segment into pgdir.  addr must be page-aligned
2677 // and the pages from addr to addr+sz must already be mapped.
2678 int
2679 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
2680 {
2681   uint i, pa, n;
2682   pte_t *pte;
2683
2684   if((uint)addr % PGSIZE != 0)
2685     panic("loaduvm: addr must be page aligned\n");
2686   for(i = 0; i < sz; i += PGSIZE){
2687     if(!(pte = walkpgdir(pgdir, addr+i, 0)))
2688       panic("loaduvm: address should exist\n");
2689     pa = PTE_ADDR(*pte);
2690     if(sz - i < PGSIZE) n = sz - i;
2691     else n = PGSIZE;
2692     if(readi(ip, (char *)pa, offset+i, n) != n)
2693       return 0;
2694   }
2695   return 1;
2696 }
2697
2698
2699
```

```
2700 // Allocate memory to the process to bring its size from oldsz to
2701 // newsz. Allocates physical memory and page table entries. oldsz and
2702 // newsz need not be page-aligned, nor does newsz have to be larger
2703 // than oldsz.  Returns the new process size or 0 on error.
2704 int
2705 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
2706 {
2707   if(newsz > USERTOP)
2708     return 0;
2709   char *a = (char *)PGROUNDUP(oldsz);
2710   char *last = PGROUNDDOWN(newsz - 1);
2711   for (; a <= last; a += PGSIZE){
2712     char *mem = kalloc();
2713     if(mem == 0){
2714       cprintf("allocuvm out of memory\n");
2715       deallocuvm(pgdir, newsz, oldsz);
2716       return 0;
2717     }
2718     memset(mem, 0, PGSIZE);
2719     mappages(pgdir, a, PGSIZE, PADDR(mem), PTE_W|PTE_U);
2720   }
2721   return newsz > oldsz ? newsz : oldsz;
2722 }
2723
2724 // Deallocate user pages to bring the process size from oldsz to
2725 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
2726 // need to be less than oldsz.  oldsz can be larger than the actual
2727 // process size.  Returns the new process size.
2728 int
2729 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
2730 {
2731   char *a = (char *)PGROUNDUP(newsz);
2732   char *last = PGROUNDDOWN(oldsz - 1);
2733   for(; a <= last; a += PGSIZE){
2734     pte_t *pte = walkpgdir(pgdir, a, 0);
2735     if(pte && (*pte & PTE_P) != 0){
2736       uint pa = PTE_ADDR(*pte);
2737       if(pa == 0)
2738         panic("kfree");
2739       kfree((void *) pa);
2740       *pte = 0;
2741     }
2742   }
2743   return newsz < oldsz ? newsz : oldsz;
2744 }
2745
2746
2747
2748
2749
```



```
2750 // Free a page table and all the physical memory pages
2751 // in the user part.
2752 void
2753 freevm(pde_t *pgdir)
2754 {
2755   uint i;
2756
2757   if(!pgdir)
2758     panic("freevm: no pgdir");
2759   deallocuvm(pgdir, USERTOP, 0);
2760   for(i = 0; i < NPDENTRIES; i++){
2761     if(pgdir[i] & PTE_P)
2762       kfree((void *) PTE_ADDR(pgdir[i]));
2763   }
2764   kfree((void *) pgdir);
2765 }
2766
2767 // Given a parent process's page table, create a copy
2768 // of it for a child.
2769 pde_t*
2770 copyuvm(pde_t *pgdir, uint sz)
2771 {
2772   pde_t *d = setupkvm();
2773   pte_t *pte;
2774   uint pa, i;
2775   char *mem;
2776
2777   if(!d) return 0;
2778   for(i = 0; i < sz; i += PGSIZE){
2779     if(!(pte = walkpgdir(pgdir, (void *)i, 0)))
2780       panic("copyuvm: pte should exist\n");
2781     if(!(*pte & PTE_P))
2782       panic("copyuvm: page not present\n");
2783     pa = PTE_ADDR(*pte);
2784     if(!(mem = kalloc()))
2785       goto bad;
2786     memmove(mem, (char *)pa, PGSIZE);
2787     if(!mappages(d, (void *)i, PGSIZE, PADDR(mem), PTE_W|PTE_U))
2788       goto bad;
2789   }
2790   return d;
2791
2792 bad:
2793   freevm(d);
2794   return 0;
2795 }
2796
2797
2798
2799
```

```
2800 // x86 trap and interrupt constants.
2801
2802 // Processor-defined:
2803 #define T_DIVIDE         0      // divide error
2804 #define T_DEBUG          1      // debug exception
2805 #define T_NMI            2      // non-maskable interrupt
2806 #define T_BRKPT          3      // breakpoint
2807 #define T_OFLOW          4      // overflow
2808 #define T_BOUND          5      // bounds check
2809 #define T_ILLOP          6      // illegal opcode
2810 #define T_DEVICE         7      // device not available
2811 #define T_DBLFLT         8      // double fault
2812 // #define T_COPROC      9      // reserved (not used since 486)
2813 #define T_TSS           10      // invalid task switch segment
2814 #define T_SEGNP         11      // segment not present
2815 #define T_STACK         12      // stack exception
2816 #define T_GPFLT         13      // general protection fault
2817 #define T_PGFLT         14      // page fault
2818 // #define T_RES        15      // reserved
2819 #define T_FPERR         16      // floating point error
2820 #define T_ALIGN         17      // aligment check
2821 #define T_MCHK          18      // machine check
2822 #define T_SIMDERR       19      // SIMD floating point error
2823
2824 // These are arbitrarily chosen, but with care not to overlap
2825 // processor defined exceptions or interrupt vectors.
2826 #define T_SYSCALL       64      // system call
2827 #define T_DEFAULT      500      // catchall
2828
2829 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
2830
2831 #define IRQ_TIMER        0
2832 #define IRQ_KBD          1
2833 #define IRQ_COM1         4
2834 #define IRQ_IDE         14
2835 #define IRQ_ERROR       19
2836 #define IRQ_SPURIOUS    31
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
```



```
2850 #!/usr/bin/perl -w
2851
2852 # Generate vectors.S, the trap/interrupt entry points.
2853 # There has to be one entry point per interrupt number
2854 # since otherwise there's no way for trap() to discover
2855 # the interrupt number.
2856
2857 print "# generated by vectors.pl - do not edit\n";
2858 print "# handlers\n";
2859 print ".globl alltraps\n";
2860 for(my $i = 0; $i < 256; $i++){
2861     print ".globl vector$i\n";
2862     print "vector$i:\n";
2863     if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
2864         print "  pushl \$0\n";
2865     }
2866     print "  pushl \$$i\n";
2867     print "  jmp alltraps\n";
2868 }
2869
2870 print "\n# vector table\n";
2871 print ".data\n";
2872 print ".globl vectors\n";
2873 print "vectors:\n";
2874 for(my $i = 0; $i < 256; $i++){
2875     print "  .long vector$i\n";
2876 }
2877
2878 # sample output:
2879 #   # handlers
2880 #   .globl alltraps
2881 #   .globl vector0
2882 #   vector0:
2883 #     pushl $0
2884 #     pushl $0
2885 #     jmp alltraps
2886 #   ...
2887 #
2888 #   # vector table
2889 #   .data
2890 #   .globl vectors
2891 #   vectors:
2892 #     .long vector0
2893 #     .long vector1
2894 #     .long vector2
2895 #   ...
2896
2897
2898
2899
```

```
#define SEG_KCODE 1  // kernel code
#define SEG_KDATA 2  // kernel data+stack
#define SEG_KCPU  3  // kernel per-cpu data

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

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

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

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



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

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

void
tvinit(void)
{
  int i;

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

  initlock(&tickslock, "time");
}

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

```
3000 void
3001 trap(struct trapframe *tf)
3002 {
3003   if(tf->trapno == T_SYSCALL){
3004     if(proc->killed)
3005       exit();
3006     proc->tf = tf;
3007     syscall();
3008     if(proc->killed)
3009       exit();
3010     return;
3011   }
3012
3013   switch(tf->trapno){
3014   case T_IRQ0 + IRQ_TIMER:
3015     if(cpu->id == 0){
3016       acquire(&tickslock);
3017       ticks++;
3018       wakeup(&ticks);
3019       release(&tickslock);
3020     }
3021     lapiceoi();
3022     break;
3023   case T_IRQ0 + IRQ_IDE:
3024     ideintr();
3025     lapiceoi();
3026     break;
3027   case T_IRQ0 + IRQ_KBD:
3028     kbdintr();
3029     lapiceoi();
3030     break;
3031   case T_IRQ0 + IRQ_COM1:
3032     uartintr();
3033     lapiceoi();
3034     break;
3035   case T_IRQ0 + 7:
3036   case T_IRQ0 + IRQ_SPURIOUS:
3037     cprintf("cpu%d: spurious interrupt at %x:%x\n",
3038             cpu->id, tf->cs, tf->eip);
3039     lapiceoi();
3040     break;
3041
3042
3043
3044
3045
3046
3047
3048
3049
```



```
3050   default:
3051     if(proc == 0 || (tf->cs&3) == 0){
3052       // In kernel, it must be our mistake.
3053       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3054               tf->trapno, cpu->id, tf->eip, rcr2());
3055       panic("trap");
3056     }
3057     // In user space, assume process misbehaved.
3058     cprintf("pid %d %s: trap %d err %d on cpu %d "
3059             "eip 0x%x addr 0x%x--kill proc\n",
3060             proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
3061             rcr2());
3062     proc->killed = 1;
3063   }
3064
3065   // Force process exit if it has been killed and is in user space.
3066   // (If it is still executing in the kernel, let it keep running
3067   // until it gets to the regular system call return.)
3068   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3069     exit();
3070
3071   // Force process to give up CPU on clock tick.
3072   // If interrupts were on while locks held, would need to check nlock.
3073   if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3074     yield();
3075
3076   // Check if the process has been killed since we yielded
3077   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3078     exit();
3079 }
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
```

```
3100 // System call numbers
3101 #define SYS_fork    1
3102 #define SYS_exit    2
3103 #define SYS_wait    3
3104 #define SYS_pipe    4
3105 #define SYS_write   5
3106 #define SYS_read    6
3107 #define SYS_close   7
3108 #define SYS_kill    8
3109 #define SYS_exec    9
3110 #define SYS_open   10
3111 #define SYS_mknod  11
3112 #define SYS_unlink 12
3113 #define SYS_fstat  13
3114 #define SYS_link   14
3115 #define SYS_mkdir  15
3116 #define SYS_chdir  16
3117 #define SYS_dup    17
3118 #define SYS_getpid 18
3119 #define SYS_sbrk   19
3120 #define SYS_sleep  20
3121 #define SYS_uptime 21
3122
3123
3124
3125
3126
3127
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
```



```
3150 #include "types.h"
3151 #include "defs.h"
3152 #include "param.h"
3153 #include "mmu.h"
3154 #include "proc.h"
3155 #include "x86.h"
3156 #include "syscall.h"
3157
3158 // User code makes a system call with INT T_SYSCALL.
3159 // System call number in %eax.
3160 // Arguments on the stack, from the user call to the C
3161 // library system call function. The saved user %esp points
3162 // to a saved program counter, and then the first argument.
3163
3164 // Fetch the int at addr from process p.
3165 int
3166 fetchint(struct proc *p, uint addr, int *ip)
3167 {
3168   if(addr >= p->sz || addr+4 > p->sz)
3169     return -1;
3170   *ip = *(int*)(addr);
3171   return 0;
3172 }
3173
3174 // Fetch the nul-terminated string at addr from process p.
3175 // Doesn't actually copy the string - just sets *pp to point at it.
3176 // Returns length of string, not including nul.
3177 int
3178 fetchstr(struct proc *p, uint addr, char **pp)
3179 {
3180   char *s, *ep;
3181
3182   if(addr >= p->sz)
3183     return -1;
3184   *pp = (char *) addr;
3185   ep = (char *) p->sz;
3186   for(s = *pp; s < ep; s++)
3187     if(*s == 0)
3188       return s - *pp;
3189   return -1;
3190 }
3191
3192 // Fetch the nth 32-bit system call argument.
3193 int
3194 argint(int n, int *ip)
3195 {
3196   int x = fetchint(proc, proc->tf->esp + 4 + 4*n, ip);
3197   return x;
3198 }
3199
```

```
3200 // Fetch the nth word-sized system call argument as a pointer
3201 // to a block of memory of size n bytes.  Check that the pointer
3202 // lies within the process address space.
3203 int
3204 argptr(int n, char **pp, int size)
3205 {
3206   int i;
3207
3208   if(argint(n, &i) < 0)
3209     return -1;
3210   if((uint)i >= proc->sz || (uint)i+size >= proc->sz)
3211     return -1;
3212   *pp = (char *) i;
3213   return 0;
3214 }
3215
3216 // Fetch the nth word-sized system call argument as a string pointer.
3217 // Check that the pointer is valid and the string is nul-terminated.
3218 // (There is no shared writable memory, so the string can't change
3219 // between this check and being used by the kernel.)
3220 int
3221 argstr(int n, char **pp)
3222 {
3223   int addr;
3224   if(argint(n, &addr) < 0)
3225     return -1;
3226   return fetchstr(proc, addr, pp);
3227 }
3228
3229 extern int sys_chdir(void);
3230 extern int sys_close(void);
3231 extern int sys_dup(void);
3232 extern int sys_exec(void);
3233 extern int sys_exit(void);
3234 extern int sys_fork(void);
3235 extern int sys_fstat(void);
3236 extern int sys_getpid(void);
3237 extern int sys_kill(void);
3238 extern int sys_link(void);
3239 extern int sys_mkdir(void);
3240 extern int sys_mknod(void);
3241 extern int sys_open(void);
3242 extern int sys_pipe(void);
3243 extern int sys_read(void);
3244 extern int sys_sbrk(void);
3245 extern int sys_sleep(void);
3246 extern int sys_unlink(void);
3247 extern int sys_wait(void);
3248 extern int sys_write(void);
3249 extern int sys_uptime(void);
```



```
3250 static int (*syscalls[])(void) = {
3251 [SYS_chdir]   sys_chdir,
3252 [SYS_close]   sys_close,
3253 [SYS_dup]     sys_dup,
3254 [SYS_exec]    sys_exec,
3255 [SYS_exit]    sys_exit,
3256 [SYS_fork]    sys_fork,
3257 [SYS_fstat]   sys_fstat,
3258 [SYS_getpid]  sys_getpid,
3259 [SYS_kill]    sys_kill,
3260 [SYS_link]    sys_link,
3261 [SYS_mkdir]   sys_mkdir,
3262 [SYS_mknod]   sys_mknod,
3263 [SYS_open]    sys_open,
3264 [SYS_pipe]    sys_pipe,
3265 [SYS_read]    sys_read,
3266 [SYS_sbrk]    sys_sbrk,
3267 [SYS_sleep]   sys_sleep,
3268 [SYS_unlink]  sys_unlink,
3269 [SYS_wait]    sys_wait,
3270 [SYS_write]   sys_write,
3271 [SYS_uptime]  sys_uptime,
3272 };
3273
3274 void
3275 syscall(void)
3276 {
3277   int num;
3278
3279   num = proc->tf->eax;
3280   if(num >= 0 && num < NELEM(syscalls) && syscalls[num])
3281     proc->tf->eax = syscalls[num]();
3282   else {
3283     cprintf("%d %s: unknown sys call %d\n",
3284             proc->pid, proc->name, num);
3285     proc->tf->eax = -1;
3286   }
3287 }
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
```

```
3300 #include "types.h"
3301 #include "x86.h"
3302 #include "defs.h"
3303 #include "param.h"
3304 #include "mmu.h"
3305 #include "proc.h"
3306
3307 int
3308 sys_fork(void)
3309 {
3310   return fork();
3311 }
3312
3313 int
3314 sys_exit(void)
3315 {
3316   exit();
3317   return 0;  // not reached
3318 }
3319
3320 int
3321 sys_wait(void)
3322 {
3323   return wait();
3324 }
3325
3326 int
3327 sys_kill(void)
3328 {
3329   int pid;
3330
3331   if(argint(0, &pid) < 0)
3332     return -1;
3333   return kill(pid);
3334 }
3335
3336 int
3337 sys_getpid(void)
3338 {
3339   return proc->pid;
3340 }
3341
3342
3343
3344
3345
3346
3347
3348
3349
```



```
3350 int
3351 sys_sbrk(void)
3352 {
3353   int addr;
3354   int n;
3355
3356   if(argint(0, &n) < 0)
3357     return -1;
3358   addr = proc->sz;
3359   if(growproc(n) < 0)
3360     return -1;
3361   return addr;
3362 }
3363
3364 int
3365 sys_sleep(void)
3366 {
3367   int n;
3368   uint ticks0;
3369
3370   if(argint(0, &n) < 0)
3371     return -1;
3372   acquire(&tickslock);
3373   ticks0 = ticks;
3374   while(ticks - ticks0 < n){
3375     if(proc->killed){
3376       release(&tickslock);
3377       return -1;
3378     }
3379     sleep(&ticks, &tickslock);
3380   }
3381   release(&tickslock);
3382   return 0;
3383 }
3384
3385 // return how many clock tick interrupts have occurred
3386 // since boot.
3387 int
3388 sys_uptime(void)
3389 {
3390   uint xticks;
3391
3392   acquire(&tickslock);
3393   xticks = ticks;
3394   release(&tickslock);
3395   return xticks;
3396 }
3397
3398
3399
```

```
3400 struct buf {
3401   int flags;
3402   uint dev;
3403   uint sector;
3404   struct buf *prev; // LRU cache list
3405   struct buf *next;
3406   struct buf *qnext; // disk queue
3407   uchar data[512];
3408 };
3409 #define B_BUSY  0x1  // buffer is locked by some process
3410 #define B_VALID 0x2  // buffer has been read from disk
3411 #define B_DIRTY 0x4  // buffer needs to be written to disk
3412
3413
3414
3415
3416
3417
3418
3419
3420
3421
3422
3423
3424
3425
3426
3427
3428
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
```



```
3450 #define O_RDONLY  0x000
3451 #define O_WRONLY  0x001
3452 #define O_RDWR    0x002
3453 #define O_CREATE  0x200
3454
3455
3456
3457
3458
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3479
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
```

```
3500 #define T_DIR  1   // Directory
3501 #define T_FILE 2   // File
3502 #define T_DEV  3   // Special device
3503
3504 struct stat {
3505   short type;  // Type of file
3506   int dev;     // Device number
3507   uint ino;    // Inode number on device
3508   short nlink; // Number of links to file
3509   uint size;   // Size of file in bytes
3510 };
3511
3512
3513
3514
3515
3516
3517
3518
3519
3520
3521
3522
3523
3524
3525
3526
3527
3528
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
```



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

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



```
3650 struct file {
3651   enum { FD_NONE, FD_PIPE, FD_INODE } type;
3652   int ref; // reference count
3653   char readable;
3654   char writable;
3655   struct pipe *pipe;
3656   struct inode *ip;
3657   uint off;
3658 };
3659
3660
3661 // in-core file system types
3662
3663 struct inode {
3664   uint dev;           // Device number
3665   uint inum;          // Inode number
3666   int ref;            // Reference count
3667   int flags;          // I_BUSY, I_VALID
3668
3669   short type;         // copy of disk inode
3670   short major;
3671   short minor;
3672   short nlink;
3673   uint size;
3674   uint addrs[NDIRECT+1];
3675 };
3676
3677 #define I_BUSY 0x1
3678 #define I_VALID 0x2
3679
3680
3681 // device implementations
3682
3683 struct devsw {
3684   int (*read)(struct inode*, char*, int);
3685   int (*write)(struct inode*, char*, int);
3686 };
3687
3688 extern struct devsw devsw[];
3689
3690 #define CONSOLE 1
3691
3692
3693
3694
3695
3696
3697
3698
3699
```

```
3700 // Simple PIO-based (non-DMA) IDE driver code.
3701
3702 #include "types.h"
3703 #include "defs.h"
3704 #include "param.h"
3705 #include "mmu.h"
3706 #include "proc.h"
3707 #include "x86.h"
3708 #include "traps.h"
3709 #include "spinlock.h"
3710 #include "buf.h"
3711
3712 #define IDE_BSY       0x80
3713 #define IDE_DRDY      0x40
3714 #define IDE_DF        0x20
3715 #define IDE_ERR       0x01
3716
3717 #define IDE_CMD_READ  0x20
3718 #define IDE_CMD_WRITE 0x30
3719
3720 // idequeue points to the buf now being read/written to the disk.
3721 // idequeue->qnext points to the next buf to be processed.
3722 // You must hold idelock while manipulating queue.
3723
3724 static struct spinlock idelock;
3725 static struct buf *idequeue;
3726
3727 static int havedisk1;
3728 static void idestart(struct buf*);
3729
3730 // Wait for IDE disk to become ready.
3731 static int
3732 idewait(int checkerr)
3733 {
3734   int r;
3735
3736   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
3737     ;
3738   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
3739     return -1;
3740   return 0;
3741 }
3742
3743
3744
3745
3746
3747
3748
3749
```



```
3750 void
3751 ideinit(void)
3752 {
3753   int i;
3754
3755   initlock(&idelock, "ide");
3756   picenable(IRQ_IDE);
3757   ioapicenable(IRQ_IDE, ncpu - 1);
3758   idewait(0);
3759
3760   // Check if disk 1 is present
3761   outb(0x1f6, 0xe0 | (1<<4));
3762   for(i=0; i<1000; i++){
3763     if(inb(0x1f7) != 0){
3764       havedisk1 = 1;
3765       break;
3766     }
3767   }
3768
3769   // Switch back to disk 0.
3770   outb(0x1f6, 0xe0 | (0<<4));
3771 }
3772
3773 // Start the request for b.  Caller must hold idelock.
3774 static void
3775 idestart(struct buf *b)
3776 {
3777   if(b == 0)
3778     panic("idestart");
3779
3780   idewait(0);
3781   outb(0x3f6, 0);  // generate interrupt
3782   outb(0x1f2, 1);  // number of sectors
3783   outb(0x1f3, b->sector & 0xff);
3784   outb(0x1f4, (b->sector >> 8) & 0xff);
3785   outb(0x1f5, (b->sector >> 16) & 0xff);
3786   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
3787   if(b->flags & B_DIRTY){
3788     outb(0x1f7, IDE_CMD_WRITE);
3789     outsl(0x1f0, b->data, 512/4);
3790   } else {
3791     outb(0x1f7, IDE_CMD_READ);
3792   }
3793 }
3794
3795
3796
3797
3798
3799
```

```
3800 // Interrupt handler.
3801 void
3802 ideintr(void)
3803 {
3804   struct buf *b;
3805
3806   // Take first buffer off queue.
3807   acquire(&idelock);
3808   if((b = idequeue) == 0){
3809     release(&idelock);
3810     cprintf("Spurious IDE interrupt.\n");
3811     return;
3812   }
3813   idequeue = b->qnext;
3814
3815   // Read data if needed.
3816   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
3817     insl(0x1f0, b->data, 512/4);
3818
3819   // Wake process waiting for this buf.
3820   b->flags |= B_VALID;
3821   b->flags &= ~B_DIRTY;
3822   wakeup(b);
3823
3824   // Start disk on next buf in queue.
3825   if(idequeue != 0)
3826     idestart(idequeue);
3827
3828   release(&idelock);
3829 }
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
```



```
3850 // Sync buf with disk.
3851 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
3852 // Else if B_VALID is not set, read buf from disk, set B_VALID.
3853 void
3854 iderw(struct buf *b)
3855 {
3856   struct buf **pp;
3857
3858   if(!(b->flags & B_BUSY))
3859     panic("iderw: buf not busy");
3860   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
3861     panic("iderw: nothing to do");
3862   if(b->dev != 0 && !havedisk1)
3863     panic("idrw: ide disk 1 not present");
3864
3865   acquire(&idelock);
3866
3867   // Append b to idequeue.
3868   b->qnext = 0;
3869   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
3870     ;
3871   *pp = b;
3872
3873   // Start disk if necessary.
3874   if(idequeue == b)
3875     idestart(b);
3876
3877   // Wait for request to finish.
3878   // Assuming will not sleep too long: ignore proc->killed.
3879   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID) {
3880     sleep(b, &idelock);
3881   }
3882
3883   release(&idelock);
3884 }
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
```

```
3900 // Buffer cache.
3901 //
3902 // The buffer cache is a linked list of buf structures holding
3903 // cached copies of disk block contents.  Caching disk blocks
3904 // in memory reduces the number of disk reads and also provides
3905 // a synchronization point for disk blocks used by multiple processes.
3906 //
3907 // Interface:
3908 // * To get a buffer for a particular disk block, call bread.
3909 // * After changing buffer data, call bwrite to flush it to disk.
3910 // * When done with the buffer, call brelse.
3911 // * Do not use the buffer after calling brelse.
3912 // * Only one process at a time can use a buffer,
3913 //     so do not keep them longer than necessary.
3914 //
3915 // The implementation uses three state flags internally:
3916 // * B_BUSY: the block has been returned from bread
3917 //     and has not been passed back to brelse.
3918 // * B_VALID: the buffer data has been initialized
3919 //     with the associated disk block contents.
3920 // * B_DIRTY: the buffer data has been modified
3921 //     and needs to be written to disk.
3922
3923 #include "types.h"
3924 #include "defs.h"
3925 #include "param.h"
3926 #include "spinlock.h"
3927 #include "buf.h"
3928
3929 struct {
3930   struct spinlock lock;
3931   struct buf buf[NBUF];
3932
3933   // Linked list of all buffers, through prev/next.
3934   // head.next is most recently used.
3935   struct buf head;
3936 } bcache;
3937
3938 void
3939 binit(void)
3940 {
3941   struct buf *b;
3942
3943   initlock(&bcache.lock, "bcache");
3944
3945
3946
3947
3948
3949
```



```
3950   // Create linked list of buffers
3951   bcache.head.prev = &bcache.head;
3952   bcache.head.next = &bcache.head;
3953   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
3954     b->next = bcache.head.next;
3955     b->prev = &bcache.head;
3956     b->dev = -1;
3957     bcache.head.next->prev = b;
3958     bcache.head.next = b;
3959   }
3960 }
3961
3962 // Look through buffer cache for sector on device dev.
3963 // If not found, allocate fresh block.
3964 // In either case, return locked buffer.
3965 static struct buf*
3966 bget(uint dev, uint sector)
3967 {
3968   struct buf *b;
3969
3970   acquire(&bcache.lock);
3971
3972  loop:
3973   // Try for cached block.
3974   for(b = bcache.head.next; b != &bcache.head; b = b->next){
3975     if(b->dev == dev && b->sector == sector){
3976       if(!(b->flags & B_BUSY)){
3977         b->flags |= B_BUSY;
3978         release(&bcache.lock);
3979         return b;
3980       }
3981       sleep(b, &bcache.lock);
3982       goto loop;
3983     }
3984   }
3985
3986   // Allocate fresh block.
3987   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
3988     if((b->flags & B_BUSY) == 0){
3989       b->dev = dev;
3990       b->sector = sector;
3991       b->flags = B_BUSY;
3992       release(&bcache.lock);
3993       return b;
3994     }
3995   }
3996   panic("bget: no buffers");
3997 }
3998
3999
```

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



```
4050 // File system implementation.  Four layers:
4051 //   + Blocks: allocator for raw disk blocks.
4052 //   + Files: inode allocator, reading, writing, metadata.
4053 //   + Directories: inode with special contents (list of other inodes!)
4054 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
4055 //
4056 // Disk layout is: superblock, inodes, block in-use bitmap, data blocks.
4057 //
4058 // This file contains the low-level file system manipulation
4059 // routines.  The (higher-level) system call implementations
4060 // are in sysfile.c.
4061
4062 #include "types.h"
4063 #include "defs.h"
4064 #include "param.h"
4065 #include "stat.h"
4066 #include "mmu.h"
4067 #include "proc.h"
4068 #include "spinlock.h"
4069 #include "buf.h"
4070 #include "fs.h"
4071 #include "file.h"
4072
4073 #define min(a, b) ((a) < (b) ? (a) : (b))
4074 static void itrunc(struct inode*);
4075
4076 // Read the super block.
4077 static void
4078 readsb(int dev, struct superblock *sb)
4079 {
4080   struct buf *bp;
4081
4082   bp = bread(dev, 1);
4083   memmove(sb, bp->data, sizeof(*sb));
4084   brelse(bp);
4085 }
4086
4087 // Zero a block.
4088 static void
4089 bzero(int dev, int bno)
4090 {
4091   struct buf *bp;
4092
4093   bp = bread(dev, bno);
4094   memset(bp->data, 0, BSIZE);
4095   bwrite(bp);
4096   brelse(bp);
4097 }
4098
4099
```

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



```
4150 // Inodes.
4151 //
4152 // An inode is a single, unnamed file in the file system.
4153 // The inode disk structure holds metadata (the type, device numbers,
4154 // and data size) along with a list of blocks where the associated
4155 // data can be found.
4156 //
4157 // The inodes are laid out sequentially on disk immediately after
4158 // the superblock.  The kernel keeps a cache of the in-use
4159 // on-disk structures to provide a place for synchronizing access
4160 // to inodes shared between multiple processes.
4161 //
4162 // ip->ref counts the number of pointer references to this cached
4163 // inode; references are typically kept in struct file and in proc->cwd.
4164 // When ip->ref falls to zero, the inode is no longer cached.
4165 // It is an error to use an inode without holding a reference to it.
4166 //
4167 // Processes are only allowed to read and write inode
4168 // metadata and contents when holding the inode's lock,
4169 // represented by the I_BUSY flag in the in-memory copy.
4170 // Because inode locks are held during disk accesses,
4171 // they are implemented using a flag rather than with
4172 // spin locks.  Callers are responsible for locking
4173 // inodes before passing them to routines in this file; leaving
4174 // this responsibility with the caller makes it possible for them
4175 // to create arbitrarily-sized atomic operations.
4176 //
4177 // To give maximum control over locking to the callers,
4178 // the routines in this file that return inode pointers
4179 // return pointers to *unlocked* inodes.  It is the callers'
4180 // responsibility to lock them before using them.  A non-zero
4181 // ip->ref keeps these unlocked inodes in the cache.
4182
4183 struct {
4184   struct spinlock lock;
4185   struct inode inode[NINODE];
4186 } icache;
4187
4188 void
4189 iinit(void)
4190 {
4191   initlock(&icache.lock, "icache");
4192 }
4193
4194 static struct inode* iget(uint dev, uint inum);
4195
4196
4197
4198
4199
```

```
4200 // Allocate a new inode with the given type on device dev.
4201 struct inode*
4202 ialloc(uint dev, short type)
4203 {
4204   int inum;
4205   struct buf *bp;
4206   struct dinode *dip;
4207   struct superblock sb;
4208
4209   readsb(dev, &sb);
4210   for(inum = 1; inum < sb.ninodes; inum++){  // loop over inode blocks
4211     bp = bread(dev, IBLOCK(inum));
4212     dip = (struct dinode*)bp->data + inum%IPB;
4213     if(dip->type == 0){  // a free inode
4214       memset(dip, 0, sizeof(*dip));
4215       dip->type = type;
4216       bwrite(bp);   // mark it allocated on the disk
4217       brelse(bp);
4218       return iget(dev, inum);
4219     }
4220     brelse(bp);
4221   }
4222   panic("ialloc: no inodes");
4223 }
4224
4225 // Copy inode, which has changed, from memory to disk.
4226 void
4227 iupdate(struct inode *ip)
4228 {
4229   struct buf *bp;
4230   struct dinode *dip;
4231
4232   bp = bread(ip->dev, IBLOCK(ip->inum));
4233   dip = (struct dinode*)bp->data + ip->inum%IPB;
4234   dip->type = ip->type;
4235   dip->major = ip->major;
4236   dip->minor = ip->minor;
4237   dip->nlink = ip->nlink;
4238   dip->size = ip->size;
4239   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
4240   bwrite(bp);
4241   brelse(bp);
4242 }
4243
4244
4245
4246
4247
4248
4249
```



```
4250 // Find the inode with number inum on device dev
4251 // and return the in-memory copy.
4252 static struct inode*
4253 iget(uint dev, uint inum)
4254 {
4255   struct inode *ip, *empty;
4256
4257   acquire(&icache.lock);
4258
4259   // Try for cached inode.
4260   empty = 0;
4261   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
4262     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
4263       ip->ref++;
4264       release(&icache.lock);
4265       return ip;
4266     }
4267     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
4268       empty = ip;
4269   }
4270
4271   // Allocate fresh inode.
4272   if(empty == 0)
4273     panic("iget: no inodes");
4274
4275   ip = empty;
4276   ip->dev = dev;
4277   ip->inum = inum;
4278   ip->ref = 1;
4279   ip->flags = 0;
4280   release(&icache.lock);
4281
4282   return ip;
4283 }
4284
4285 // Increment reference count for ip.
4286 // Returns ip to enable ip = idup(ip1) idiom.
4287 struct inode*
4288 idup(struct inode *ip)
4289 {
4290   acquire(&icache.lock);
4291   ip->ref++;
4292   release(&icache.lock);
4293   return ip;
4294 }
4295
4296
4297
4298
4299
```

```
4300 // Lock the given inode.
4301 void
4302 ilock(struct inode *ip)
4303 {
4304   struct buf *bp;
4305   struct dinode *dip;
4306
4307   if(ip == 0 || ip->ref < 1)
4308     panic("ilock");
4309
4310   acquire(&icache.lock);
4311   while(ip->flags & I_BUSY)
4312     sleep(ip, &icache.lock);
4313   ip->flags |= I_BUSY;
4314   release(&icache.lock);
4315
4316   if(!(ip->flags & I_VALID)){
4317     bp = bread(ip->dev, IBLOCK(ip->inum));
4318     dip = (struct dinode*)bp->data + ip->inum%IPB;
4319     ip->type = dip->type;
4320     ip->major = dip->major;
4321     ip->minor = dip->minor;
4322     ip->nlink = dip->nlink;
4323     ip->size = dip->size;
4324     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
4325     brelse(bp);
4326     ip->flags |= I_VALID;
4327     if(ip->type == 0)
4328       panic("ilock: no type");
4329   }
4330 }
4331
4332 // Unlock the given inode.
4333 void
4334 iunlock(struct inode *ip)
4335 {
4336   if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
4337     panic("iunlock");
4338
4339   acquire(&icache.lock);
4340   ip->flags &= ~I_BUSY;
4341   wakeup(ip);
4342   release(&icache.lock);
4343 }
4344
4345
4346
4347
4348
4349
```



```
4350 // Caller holds reference to unlocked ip.  Drop reference.
4351 void
4352 iput(struct inode *ip)
4353 {
4354   acquire(&icache.lock);
4355   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
4356     // inode is no longer used: truncate and free inode.
4357     if(ip->flags & I_BUSY)
4358       panic("iput busy");
4359     ip->flags |= I_BUSY;
4360     release(&icache.lock);
4361     itrunc(ip);
4362     ip->type = 0;
4363     iupdate(ip);
4364     acquire(&icache.lock);
4365     ip->flags = 0;
4366     wakeup(ip);
4367   }
4368   ip->ref--;
4369   release(&icache.lock);
4370 }
4371
4372 // Common idiom: unlock, then put.
4373 void
4374 iunlockput(struct inode *ip)
4375 {
4376   iunlock(ip);
4377   iput(ip);
4378 }
4379
4380
4381
4382
4383
4384
4385
4386
4387
4388
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
```

```
4400 // Inode contents
4401 //
4402 // The contents (data) associated with each inode is stored
4403 // in a sequence of blocks on the disk.  The first NDIRECT blocks
4404 // are listed in ip->addrs[].  The next NINDIRECT blocks are
4405 // listed in the block ip->addrs[NDIRECT].
4406
4407 // Return the disk block address of the nth block in inode ip.
4408 // If there is no such block, bmap allocates one.
4409 static uint
4410 bmap(struct inode *ip, uint bn)
4411 {
4412   uint addr, *a;
4413   struct buf *bp;
4414
4415   if(bn < NDIRECT){
4416     if((addr = ip->addrs[bn]) == 0)
4417       ip->addrs[bn] = addr = balloc(ip->dev);
4418     return addr;
4419   }
4420   bn -= NDIRECT;
4421
4422   if(bn < NINDIRECT){
4423     // Load indirect block, allocating if necessary.
4424     if((addr = ip->addrs[NDIRECT]) == 0)
4425       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
4426     bp = bread(ip->dev, addr);
4427     a = (uint*)bp->data;
4428     if((addr = a[bn]) == 0){
4429       a[bn] = addr = balloc(ip->dev);
4430       bwrite(bp);
4431     }
4432     brelse(bp);
4433     return addr;
4434   }
4435
4436   panic("bmap: out of range");
4437 }
4438
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
```



```
4450 // Truncate inode (discard contents).
4451 // Only called after the last dirent referring
4452 // to this inode has been erased on disk.
4453 static void
4454 itrunc(struct inode *ip)
4455 {
4456   int i, j;
4457   struct buf *bp;
4458   uint *a;
4459
4460   for(i = 0; i < NDIRECT; i++){
4461     if(ip->addrs[i]){
4462       bfree(ip->dev, ip->addrs[i]);
4463       ip->addrs[i] = 0;
4464     }
4465   }
4466
4467   if(ip->addrs[NDIRECT]){
4468     bp = bread(ip->dev, ip->addrs[NDIRECT]);
4469     a = (uint*)bp->data;
4470     for(j = 0; j < NINDIRECT; j++){
4471       if(a[j])
4472         bfree(ip->dev, a[j]);
4473     }
4474     brelse(bp);
4475     bfree(ip->dev, ip->addrs[NDIRECT]);
4476     ip->addrs[NDIRECT] = 0;
4477   }
4478
4479   ip->size = 0;
4480   iupdate(ip);
4481 }
4482
4483 // Copy stat information from inode.
4484 void
4485 stati(struct inode *ip, struct stat *st)
4486 {
4487   st->dev = ip->dev;
4488   st->ino = ip->inum;
4489   st->type = ip->type;
4490   st->nlink = ip->nlink;
4491   st->size = ip->size;
4492 }
4493
4494
4495
4496
4497
4498
4499
```

```
4500 // Read data from inode.
4501 int
4502 readi(struct inode *ip, char *dst, uint off, uint n)
4503 {
4504   uint tot, m;
4505   struct buf *bp;
4506
4507   if(ip->type == T_DEV){
4508     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
4509       return -1;
4510     return devsw[ip->major].read(ip, dst, n);
4511   }
4512
4513   if(off > ip->size || off + n < off)
4514     return -1;
4515   if(off + n > ip->size)
4516     n = ip->size - off;
4517
4518   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
4519     bp = bread(ip->dev, bmap(ip, off/BSIZE));
4520     m = min(n - tot, BSIZE - off%BSIZE);
4521     memmove(dst, bp->data + off%BSIZE, m);
4522     brelse(bp);
4523   }
4524   return n;
4525 }
4526
4527
4528
4529
4530
4531
4532
4533
4534
4535
4536
4537
4538
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4549
```



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

```
4600 // Directories
4601
4602 int
4603 namecmp(const char *s, const char *t)
4604 {
4605   return strncmp(s, t, DIRSIZ);
4606 }
4607
4608 // Look for a directory entry in a directory.
4609 // If found, set *poff to byte offset of entry.
4610 // Caller must have already locked dp.
4611 struct inode*
4612 dirlookup(struct inode *dp, char *name, uint *poff)
4613 {
4614   uint off, inum;
4615   struct buf *bp;
4616   struct dirent *de;
4617
4618   if(dp->type != T_DIR)
4619     panic("dirlookup not DIR");
4620
4621   for(off = 0; off < dp->size; off += BSIZE){
4622     bp = bread(dp->dev, bmap(dp, off / BSIZE));
4623     for(de = (struct dirent*)bp->data;
4624         de < (struct dirent*)(bp->data + BSIZE);
4625         de++){
4626       if(de->inum == 0)
4627         continue;
4628       if(namecmp(name, de->name) == 0){
4629         // entry matches path element
4630         if(poff)
4631           *poff = off + (uchar*)de - bp->data;
4632         inum = de->inum;
4633         brelse(bp);
4634         return iget(dp->dev, inum);
4635       }
4636     }
4637     brelse(bp);
4638   }
4639   return 0;
4640 }
4641
4642
4643
4644
4645
4646
4647
4648
4649
```



```
4650 // Write a new directory entry (name, inum) into the directory dp.
4651 int
4652 dirlink(struct inode *dp, char *name, uint inum)
4653 {
4654   int off;
4655   struct dirent de;
4656   struct inode *ip;
4657
4658   // Check that name is not present.
4659   if((ip = dirlookup(dp, name, 0)) != 0){
4660     iput(ip);
4661     return -1;
4662   }
4663
4664   // Look for an empty dirent.
4665   for(off = 0; off < dp->size; off += sizeof(de)){
4666     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4667       panic("dirlink read");
4668     if(de.inum == 0)
4669       break;
4670   }
4671
4672   strncpy(de.name, name, DIRSIZ);
4673   de.inum = inum;
4674   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4675     panic("dirlink");
4676
4677   return 0;
4678 }
4679
4680
4681
4682
4683
4684
4685
4686
4687
4688
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4699
```

```
4700 // Paths
4701
4702 // Copy the next path element from path into name.
4703 // Return a pointer to the element following the copied one.
4704 // The returned path has no leading slashes,
4705 // so the caller can check *path=='\0' to see if the name is the last one.
4706 // If no name to remove, return 0.
4707 //
4708 // Examples:
4709 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
4710 //   skipelem("///a//bb", name) = "bb", setting name = "a"
4711 //   skipelem("a", name) = "", setting name = "a"
4712 //   skipelem("", name) = skipelem("////", name) = 0
4713 //
4714 static char*
4715 skipelem(char *path, char *name)
4716 {
4717   char *s;
4718   int len;
4719
4720   while(*path == '/')
4721     path++;
4722   if(*path == 0)
4723     return 0;
4724   s = path;
4725   while(*path != '/' && *path != 0)
4726     path++;
4727   len = path - s;
4728   if(len >= DIRSIZ)
4729     memmove(name, s, DIRSIZ);
4730   else {
4731     memmove(name, s, len);
4732     name[len] = 0;
4733   }
4734   while(*path == '/')
4735     path++;
4736   return path;
4737 }
4738
4739
4740
4741
4742
4743
4744
4745
4746
4747
4748
4749
```



```
4750 // Look up and return the inode for a path name.
4751 // If parent != 0, return the inode for the parent and copy the final
4752 // path element into name, which must have room for DIRSIZ bytes.
4753 static struct inode*
4754 namex(char *path, int nameiparent, char *name)
4755 {
4756   struct inode *ip, *next;
4757
4758   if(*path == '/')
4759     ip = iget(ROOTDEV, ROOTINO);
4760   else
4761     ip = idup(proc->cwd);
4762
4763   while((path = skipelem(path, name)) != 0){
4764     ilock(ip);
4765     if(ip->type != T_DIR){
4766       iunlockput(ip);
4767       return 0;
4768     }
4769     if(nameiparent && *path == '\0'){
4770       // Stop one level early.
4771       iunlock(ip);
4772       return ip;
4773     }
4774     if((next = dirlookup(ip, name, 0)) == 0){
4775       iunlockput(ip);
4776       return 0;
4777     }
4778     iunlockput(ip);
4779     ip = next;
4780   }
4781   if(nameiparent){
4782     iput(ip);
4783     return 0;
4784   }
4785   return ip;
4786 }
4787
4788 struct inode*
4789 namei(char *path)
4790 {
4791   char name[DIRSIZ];
4792   return namex(path, 0, name);
4793 }
4794
4795 struct inode*
4796 nameiparent(char *path, char *name)
4797 {
4798   return namex(path, 1, name);
4799 }
```

```
4800 #include "types.h"
4801 #include "defs.h"
4802 #include "param.h"
4803 #include "fs.h"
4804 #include "file.h"
4805 #include "spinlock.h"
4806
4807 struct devsw devsw[NDEV];
4808 struct {
4809   struct spinlock lock;
4810   struct file file[NFILE];
4811 } ftable;
4812
4813 void
4814 fileinit(void)
4815 {
4816   initlock(&ftable.lock, "ftable");
4817 }
4818
4819 // Allocate a file structure.
4820 struct file*
4821 filealloc(void)
4822 {
4823   struct file *f;
4824
4825   acquire(&ftable.lock);
4826   for(f = ftable.file; f < ftable.file + NFILE; f++){
4827     if(f->ref == 0){
4828       f->ref = 1;
4829       release(&ftable.lock);
4830       return f;
4831     }
4832   }
4833   release(&ftable.lock);
4834   return 0;
4835 }
4836
4837 // Increment ref count for file f.
4838 struct file*
4839 filedup(struct file *f)
4840 {
4841   acquire(&ftable.lock);
4842   if(f->ref < 1)
4843     panic("filedup");
4844   f->ref++;
4845   release(&ftable.lock);
4846   return f;
4847 }
4848
4849
```



```
4850 // Close file f.  (Decrement ref count, close when reaches 0.)
4851 void
4852 fileclose(struct file *f)
4853 {
4854   struct file ff;
4855
4856   acquire(&ftable.lock);
4857   if(f->ref < 1)
4858     panic("fileclose");
4859   if(--f->ref > 0){
4860     release(&ftable.lock);
4861     return;
4862   }
4863   ff = *f;
4864   f->ref = 0;
4865   f->type = FD_NONE;
4866   release(&ftable.lock);
4867
4868   if(ff.type == FD_PIPE)
4869     pipeclose(ff.pipe, ff.writable);
4870   else if(ff.type == FD_INODE)
4871     iput(ff.ip);
4872 }
4873
4874 // Get metadata about file f.
4875 int
4876 filestat(struct file *f, struct stat *st)
4877 {
4878   if(f->type == FD_INODE){
4879     ilock(f->ip);
4880     stati(f->ip, st);
4881     iunlock(f->ip);
4882     return 0;
4883   }
4884   return -1;
4885 }
4886
4887
4888
4889
4890
4891
4892
4893
4894
4895
4896
4897
4898
4899
```

```
4900 // Read from file f.  Addr is kernel address.
4901 int
4902 fileread(struct file *f, char *addr, int n)
4903 {
4904   int r;
4905
4906   if(f->readable == 0)
4907     return -1;
4908   if(f->type == FD_PIPE)
4909     return piperead(f->pipe, addr, n);
4910   if(f->type == FD_INODE){
4911     ilock(f->ip);
4912     if((r = readi(f->ip, addr, f->off, n)) > 0)
4913       f->off += r;
4914     iunlock(f->ip);
4915     return r;
4916   }
4917   panic("fileread");
4918 }
4919
4920 // Write to file f.  Addr is kernel address.
4921 int
4922 filewrite(struct file *f, char *addr, int n)
4923 {
4924   int r;
4925
4926   if(f->writable == 0)
4927     return -1;
4928   if(f->type == FD_PIPE)
4929     return pipewrite(f->pipe, addr, n);
4930   if(f->type == FD_INODE){
4931     ilock(f->ip);
4932     if((r = writei(f->ip, addr, f->off, n)) > 0)
4933       f->off += r;
4934     iunlock(f->ip);
4935     return r;
4936   }
4937   panic("filewrite");
4938 }
4939
4940
4941
4942
4943
4944
4945
4946
4947
4948
4949
```



```
4950 #include "types.h"
4951 #include "defs.h"
4952 #include "param.h"
4953 #include "stat.h"
4954 #include "mmu.h"
4955 #include "proc.h"
4956 #include "fs.h"
4957 #include "file.h"
4958 #include "fcntl.h"
4959
4960 // Fetch the nth word-sized system call argument as a file descriptor
4961 // and return both the descriptor and the corresponding struct file.
4962 static int
4963 argfd(int n, int *pfd, struct file **pf)
4964 {
4965   int fd;
4966   struct file *f;
4967
4968   if(argint(n, &fd) < 0)
4969     return -1;
4970   if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
4971     return -1;
4972   if(pfd)
4973     *pfd = fd;
4974   if(pf)
4975     *pf = f;
4976   return 0;
4977 }
4978
4979 // Allocate a file descriptor for the given file.
4980 // Takes over file reference from caller on success.
4981 static int
4982 fdalloc(struct file *f)
4983 {
4984   int fd;
4985
4986   for(fd = 0; fd < NOFILE; fd++){
4987     if(proc->ofile[fd] == 0){
4988       proc->ofile[fd] = f;
4989       return fd;
4990     }
4991   }
4992   return -1;
4993 }
4994
4995
4996
4997
4998
4999
```

```
5000 int
5001 sys_dup(void)
5002 {
5003   struct file *f;
5004   int fd;
5005
5006   if(argfd(0, 0, &f) < 0)
5007     return -1;
5008   if((fd=fdalloc(f)) < 0)
5009     return -1;
5010   filedup(f);
5011   return fd;
5012 }
5013
5014 int
5015 sys_read(void)
5016 {
5017   struct file *f;
5018   int n;
5019   char *p;
5020
5021   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5022     return -1;
5023   return fileread(f, p, n);
5024 }
5025
5026 int
5027 sys_write(void)
5028 {
5029   struct file *f;
5030   int n;
5031   char *p;
5032
5033   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5034     return -1;
5035   return filewrite(f, p, n);
5036 }
5037
5038 int
5039 sys_close(void)
5040 {
5041   int fd;
5042   struct file *f;
5043
5044   if(argfd(0, &fd, &f) < 0)
5045     return -1;
5046   proc->ofile[fd] = 0;
5047   fileclose(f);
5048   return 0;
5049 }
```



```
5050 int
5051 sys_fstat(void)
5052 {
5053   struct file *f;
5054   struct stat *st;
5055
5056   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
5057     return -1;
5058   return filestat(f, st);
5059 }
5060
5061 // Create the path new as a link to the same inode as old.
5062 int
5063 sys_link(void)
5064 {
5065   char name[DIRSIZ], *new, *old;
5066   struct inode *dp, *ip;
5067
5068   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
5069     return -1;
5070   if((ip = namei(old)) == 0)
5071     return -1;
5072   ilock(ip);
5073   if(ip->type == T_DIR){
5074     iunlockput(ip);
5075     return -1;
5076   }
5077   ip->nlink++;
5078   iupdate(ip);
5079   iunlock(ip);
5080
5081   if((dp = nameiparent(new, name)) == 0)
5082     goto bad;
5083   ilock(dp);
5084   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
5085     iunlockput(dp);
5086     goto bad;
5087   }
5088   iunlockput(dp);
5089   iput(ip);
5090   return 0;
5091
5092 bad:
5093   ilock(ip);
5094   ip->nlink--;
5095   iupdate(ip);
5096   iunlockput(ip);
5097   return -1;
5098 }
5099
```

```
5100 // Is the directory dp empty except for "." and ".." ?
5101 static int
5102 isdirempty(struct inode *dp)
5103 {
5104   int off;
5105   struct dirent de;
5106
5107   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
5108     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5109       panic("isdirempty: readi");
5110     if(de.inum != 0)
5111       return 0;
5112   }
5113   return 1;
5114 }
5115
5116
5117
5118
5119
5120
5121
5122
5123
5124
5125
5126
5127
5128
5129
5130
5131
5132
5133
5134
5135
5136
5137
5138
5139
5140
5141
5142
5143
5144
5145
5146
5147
5148
5149
```



```
5150 int
5151 sys_unlink(void)
5152 {
5153   struct inode *ip, *dp;
5154   struct dirent de;
5155   char name[DIRSIZ], *path;
5156   uint off;
5157
5158   if(argstr(0, &path) < 0)
5159     return -1;
5160   if((dp = nameiparent(path, name)) == 0)
5161     return -1;
5162   ilock(dp);
5163
5164   // Cannot unlink "." or "..".
5165   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0){
5166     iunlockput(dp);
5167     return -1;
5168   }
5169
5170   if((ip = dirlookup(dp, name, &off)) == 0){
5171     iunlockput(dp);
5172     return -1;
5173   }
5174   ilock(ip);
5175
5176   if(ip->nlink < 1)
5177     panic("unlink: nlink < 1");
5178   if(ip->type == T_DIR && !isdirempty(ip)){
5179     iunlockput(ip);
5180     iunlockput(dp);
5181     return -1;
5182   }
5183
5184   memset(&de, 0, sizeof(de));
5185   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5186     panic("unlink: writei");
5187   if(ip->type == T_DIR){
5188     dp->nlink--;
5189     iupdate(dp);
5190   }
5191   iunlockput(dp);
5192
5193   ip->nlink--;
5194   iupdate(ip);
5195   iunlockput(ip);
5196   return 0;
5197 }
5198
5199
```

```
5200 static struct inode*
5201 create(char *path, short type, short major, short minor)
5202 {
5203   uint off;
5204   struct inode *ip, *dp;
5205   char name[DIRSIZ];
5206
5207   if((dp = nameiparent(path, name)) == 0)
5208     return 0;
5209   ilock(dp);
5210
5211   if((ip = dirlookup(dp, name, &off)) != 0){
5212     iunlockput(dp);
5213     ilock(ip);
5214     if(type == T_FILE && ip->type == T_FILE)
5215       return ip;
5216     iunlockput(ip);
5217     return 0;
5218   }
5219
5220   if((ip = ialloc(dp->dev, type)) == 0)
5221     panic("create: ialloc");
5222
5223   ilock(ip);
5224   ip->major = major;
5225   ip->minor = minor;
5226   ip->nlink = 1;
5227   iupdate(ip);
5228
5229   if(type == T_DIR){  // Create . and .. entries.
5230     dp->nlink++;  // for ".."
5231     iupdate(dp);
5232     // No ip->nlink++ for ".": avoid cyclic ref count.
5233     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
5234       panic("create dots");
5235   }
5236
5237   if(dirlink(dp, name, ip->inum) < 0)
5238     panic("create: dirlink");
5239
5240   iunlockput(dp);
5241   return ip;
5242 }
5243
5244
5245
5246
5247
5248
5249
```



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

```
5300 int
5301 sys_mknod(void)
5302 {
5303   struct inode *ip;
5304   char *path;
5305   int len;
5306   int major, minor;
5307
5308   if((len=argstr(0, &path)) < 0 ||
5309      argint(1, &major) < 0 ||
5310      argint(2, &minor) < 0 ||
5311      (ip = create(path, T_DEV, major, minor)) == 0)
5312     return -1;
5313   iunlockput(ip);
5314   return 0;
5315 }
5316
5317 int
5318 sys_chdir(void)
5319 {
5320   char *path;
5321   struct inode *ip;
5322
5323   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0)
5324     return -1;
5325   ilock(ip);
5326   if(ip->type != T_DIR){
5327     iunlockput(ip);
5328     return -1;
5329   }
5330   iunlock(ip);
5331   iput(proc->cwd);
5332   proc->cwd = ip;
5333   return 0;
5334 }
5335
5336
5337
5338
5339
5340
5341
5342
5343
5344
5345
5346
5347
5348
5349
```



```
5350 int
5351 sys_exec(void)
5352 {
5353   char *path, *argv[20];
5354   int i;
5355   uint uargv, uarg;
5356
5357   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0) {
5358     return -1;
5359   }
5360   memset(argv, 0, sizeof(argv));
5361   for(i=0;; i++){
5362     if(i >= NELEM(argv))
5363       return -1;
5364     if(fetchint(proc, uargv+4*i, (int*)&uarg) < 0)
5365       return -1;
5366     if(uarg == 0){
5367       argv[i] = 0;
5368       break;
5369     }
5370     if(fetchstr(proc, uarg, &argv[i]) < 0)
5371       return -1;
5372   }
5373   return exec(path, argv);
5374 }
5375
5376 int
5377 sys_pipe(void)
5378 {
5379   int *fd;
5380   struct file *rf, *wf;
5381   int fd0, fd1;
5382
5383   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
5384     return -1;
5385   if(pipealloc(&rf, &wf) < 0)
5386     return -1;
5387   fd0 = -1;
5388   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
5389     if(fd0 >= 0)
5390       proc->ofile[fd0] = 0;
5391     fileclose(rf);
5392     fileclose(wf);
5393     return -1;
5394   }
5395   fd[0] = fd0;
5396   fd[1] = fd1;
5397   return 0;
5398 }
5399
```

```
5400 #include "types.h"
5401 #include "param.h"
5402 #include "mmu.h"
5403 #include "proc.h"
5404 #include "defs.h"
5405 #include "x86.h"
5406 #include "elf.h"
5407
5408 int
5409 exec(char *path, char **argv)
5410 {
5411   char *mem, *s, *last;
5412   int i, argc, arglen, len, off;
5413   uint sz, sp, spbottom, argp;
5414   struct elfhdr elf;
5415   struct inode *ip;
5416   struct proghdr ph;
5417   pde_t *pgdir, *oldpgdir;
5418
5419   pgdir = 0;
5420   sz = 0;
5421
5422   if((ip = namei(path)) == 0)
5423     return -1;
5424   ilock(ip);
5425
5426   // Check ELF header
5427   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
5428     goto bad;
5429   if(elf.magic != ELF_MAGIC)
5430     goto bad;
5431
5432   if(!(pgdir = setupkvm()))
5433     goto bad;
5434
5435   // Load program into memory.
5436   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
5437     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
5438       goto bad;
5439     if(ph.type != ELF_PROG_LOAD)
5440       continue;
5441     if(ph.memsz < ph.filesz)
5442       goto bad;
5443     if(!(sz = allocuvm(pgdir, sz, ph.va + ph.memsz)))
5444       goto bad;
5445     if(!loaduvm(pgdir, (char *)ph.va, ip, ph.offset, ph.filesz))
5446       goto bad;
5447   }
5448   iunlockput(ip);
5449
```



```
5450   // Allocate and initialize stack at sz
5451   sz = spbottom = PGROUNDUP(sz);
5452   if(!(sz = allocuvm(pgdir, sz, sz + PGSIZE)))
5453     goto bad;
5454   mem = uva2ka(pgdir, (char *)spbottom);
5455
5456   arglen = 0;
5457   for(argc=0; argv[argc]; argc++)
5458     arglen += strlen(argv[argc]) + 1;
5459   arglen = (arglen+3) & ~3;
5460
5461   sp = sz;
5462   argp = sz - arglen - 4*(argc+1);
5463
5464   // Copy argv strings and pointers to stack.
5465   *(uint*)(mem+argp-spbottom + 4*argc) = 0;  // argv[argc]
5466   for(i=argc-1; i>=0; i--){
5467     len = strlen(argv[i]) + 1;
5468     sp -= len;
5469     memmove(mem+sp-spbottom, argv[i], len);
5470     *(uint*)(mem+argp-spbottom + 4*i) = sp;  // argv[i]
5471   }
5472
5473   // Stack frame for main(argc, argv), below arguments.
5474   sp = argp;
5475   sp -= 4;
5476   *(uint*)(mem+sp-spbottom) = argp;
5477   sp -= 4;
5478   *(uint*)(mem+sp-spbottom) = argc;
5479   sp -= 4;
5480   *(uint*)(mem+sp-spbottom) = 0xffffffff;   // fake return pc
5481
5482   // Save program name for debugging.
5483   for(last=s=path; *s; s++)
5484     if(*s == '/')
5485       last = s+1;
5486   safestrcpy(proc->name, last, sizeof(proc->name));
5487
5488   // Commit to the user image.
5489   oldpgdir = proc->pgdir;
5490   proc->pgdir = pgdir;
5491   proc->sz = sz;
5492   proc->tf->eip = elf.entry;  // main
5493   proc->tf->esp = sp;
5494
5495   switchuvm(proc);
5496
5497   freevm(oldpgdir);
5498
5499   return 0;
```

```
5500  bad:
5501    if(pgdir) freevm(pgdir);
5502    iunlockput(ip);
5503    return -1;
5504  }
5505
5506
5507
5508
5509
5510
5511
5512
5513
5514
5515
5516
5517
5518
5519
5520
5521
5522
5523
5524
5525
5526
5527
5528
5529
5530
5531
5532
5533
5534
5535
5536
5537
5538
5539
5540
5541
5542
5543
5544
5545
5546
5547
5548
5549
```



```
5550 #include "types.h"
5551 #include "defs.h"
5552 #include "param.h"
5553 #include "mmu.h"
5554 #include "proc.h"
5555 #include "fs.h"
5556 #include "file.h"
5557 #include "spinlock.h"
5558
5559 #define PIPESIZE 512
5560
5561 struct pipe {
5562   struct spinlock lock;
5563   char data[PIPESIZE];
5564   uint nread;     // number of bytes read
5565   uint nwrite;    // number of bytes written
5566   int readopen;   // read fd is still open
5567   int writeopen;  // write fd is still open
5568 };
5569
5570 int
5571 pipealloc(struct file **f0, struct file **f1)
5572 {
5573   struct pipe *p;
5574
5575   p = 0;
5576   *f0 = *f1 = 0;
5577   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
5578     goto bad;
5579   if((p = (struct pipe*)kalloc()) == 0)
5580     goto bad;
5581   p->readopen = 1;
5582   p->writeopen = 1;
5583   p->nwrite = 0;
5584   p->nread = 0;
5585   initlock(&p->lock, "pipe");
5586   (*f0)->type = FD_PIPE;
5587   (*f0)->readable = 1;
5588   (*f0)->writable = 0;
5589   (*f0)->pipe = p;
5590   (*f1)->type = FD_PIPE;
5591   (*f1)->readable = 0;
5592   (*f1)->writable = 1;
5593   (*f1)->pipe = p;
5594   return 0;
5595
5596
5597
5598
5599
```

```
5600  bad:
5601    if(p)
5602      kfree((char*)p);
5603    if(*f0)
5604      fileclose(*f0);
5605    if(*f1)
5606      fileclose(*f1);
5607    return -1;
5608  }
5609
5610  void
5611  pipeclose(struct pipe *p, int writable)
5612  {
5613    acquire(&p->lock);
5614    if(writable){
5615      p->writeopen = 0;
5616      wakeup(&p->nread);
5617    } else {
5618      p->readopen = 0;
5619      wakeup(&p->nwrite);
5620    }
5621    if(p->readopen == 0 && p->writeopen == 0) {
5622      release(&p->lock);
5623      kfree((char*)p);
5624    } else
5625      release(&p->lock);
5626  }
5627
5628
5629  int
5630  pipewrite(struct pipe *p, char *addr, int n)
5631  {
5632    int i;
5633
5634    acquire(&p->lock);
5635    for(i = 0; i < n; i++){
5636      while(p->nwrite == p->nread + PIPESIZE) {
5637        if(p->readopen == 0 || proc->killed){
5638          release(&p->lock);
5639          return -1;
5640        }
5641        wakeup(&p->nread);
5642        sleep(&p->nwrite, &p->lock);
5643      }
5644      p->data[p->nwrite++ % PIPESIZE] = addr[i];
5645    }
5646    wakeup(&p->nread);
5647    release(&p->lock);
5648    return n;
5649  }
```



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

```
5700 #include "types.h"
5701 #include "x86.h"
5702
5703 void*
5704 memset(void *dst, int c, uint n)
5705 {
5706   stosb(dst, c, n);
5707   return dst;
5708 }
5709
5710 int
5711 memcmp(const void *v1, const void *v2, uint n)
5712 {
5713   const uchar *s1, *s2;
5714
5715   s1 = v1;
5716   s2 = v2;
5717   while(n-- > 0){
5718     if(*s1 != *s2)
5719       return *s1 - *s2;
5720     s1++, s2++;
5721   }
5722
5723   return 0;
5724 }
5725
5726 void*
5727 memmove(void *dst, const void *src, uint n)
5728 {
5729   const char *s;
5730   char *d;
5731
5732   s = src;
5733   d = dst;
5734   if(s < d && s + n > d){
5735     s += n;
5736     d += n;
5737     while(n-- > 0)
5738       *--d = *--s;
5739   } else
5740     while(n-- > 0)
5741       *d++ = *s++;
5742
5743   return dst;
5744 }
5745
5746
5747
5748
5749
```



```
5750 // memcpy exists to placate GCC.  Use memmove.
5751 void*
5752 memcpy(void *dst, const void *src, uint n)
5753 {
5754   return memmove(dst, src, n);
5755 }
5756
5757 int
5758 strncmp(const char *p, const char *q, uint n)
5759 {
5760   while(n > 0 && *p && *p == *q)
5761     n--, p++, q++;
5762   if(n == 0)
5763     return 0;
5764   return (uchar)*p - (uchar)*q;
5765 }
5766
5767 char*
5768 strncpy(char *s, const char *t, int n)
5769 {
5770   char *os;
5771
5772   os = s;
5773   while(n-- > 0 && (*s++ = *t++) != 0)
5774     ;
5775   while(n-- > 0)
5776     *s++ = 0;
5777   return os;
5778 }
5779
5780 // Like strncpy but guaranteed to NUL-terminate.
5781 char*
5782 safestrcpy(char *s, const char *t, int n)
5783 {
5784   char *os;
5785
5786   os = s;
5787   if(n <= 0)
5788     return os;
5789   while(--n > 0 && (*s++ = *t++) != 0)
5790     ;
5791   *s = 0;
5792   return os;
5793 }
5794
5795
5796
5797
5798
5799
```

```
5800 int
5801 strlen(const char *s)
5802 {
5803   int n;
5804
5805   for(n = 0; s[n]; n++)
5806     ;
5807   return n;
5808 }
5809
5810
5811
5812
5813
5814
5815
5816
5817
5818
5819
5820
5821
5822
5823
5824
5825
5826
5827
5828
5829
5830
5831
5832
5833
5834
5835
5836
5837
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```



```
5850 // See MultiProcessor Specification Version 1.[14]
5851
5852 struct mp {             // floating pointer
5853   uchar signature[4];           // "_MP_"
5854   void *physaddr;               // phys addr of MP config table
5855   uchar length;                 // 1
5856   uchar specrev;                // [14]
5857   uchar checksum;               // all bytes must add up to 0
5858   uchar type;                   // MP system config type
5859   uchar imcrp;
5860   uchar reserved[3];
5861 };
5862
5863 struct mpconf {         // configuration table header
5864   uchar signature[4];           // "PCMP"
5865   ushort length;                // total table length
5866   uchar version;                // [14]
5867   uchar checksum;               // all bytes must add up to 0
5868   uchar product[20];            // product id
5869   uint *oemtable;               // OEM table pointer
5870   ushort oemlength;             // OEM table length
5871   ushort entry;                 // entry count
5872   uint *lapicaddr;              // address of local APIC
5873   ushort xlength;               // extended table length
5874   uchar xchecksum;              // extended table checksum
5875   uchar reserved;
5876 };
5877
5878 struct mpproc {         // processor table entry
5879   uchar type;                   // entry type (0)
5880   uchar apicid;                 // local APIC id
5881   uchar version;                // local APIC verison
5882   uchar flags;                  // CPU flags
5883     #define MPBOOT 0x02           // This proc is the bootstrap processor.
5884   uchar signature[4];           // CPU signature
5885   uint feature;                 // feature flags from CPUID instruction
5886   uchar reserved[8];
5887 };
5888
5889 struct mpioapic {       // I/O APIC table entry
5890   uchar type;                   // entry type (2)
5891   uchar apicno;                 // I/O APIC id
5892   uchar version;                // I/O APIC version
5893   uchar flags;                  // I/O APIC flags
5894   uint *addr;                  // I/O APIC address
5895 };
5896
5897
5898
5899
```

```
5900 // Table entry types
5901 #define MPPROC    0x00  // One per processor
5902 #define MPBUS     0x01  // One per bus
5903 #define MPIOAPIC  0x02  // One per I/O APIC
5904 #define MPIOINTR  0x03  // One per bus interrupt source
5905 #define MPLINTR   0x04  // One per system interrupt source
5906
5907
5908
5909
5910
5911
5912
5913
5914
5915
5916
5917
5918
5919
5920
5921
5922
5923
5924
5925
5926
5927
5928
5929
5930
5931
5932
5933
5934
5935
5936
5937
5938
5939
5940
5941
5942
5943
5944
5945
5946
5947
5948
5949
```



```
5950 // Multiprocessor bootstrap.
5951 // Search memory for MP description structures.
5952 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
5953
5954 #include "types.h"
5955 #include "defs.h"
5956 #include "param.h"
5957 #include "mp.h"
5958 #include "x86.h"
5959 #include "mmu.h"
5960 #include "proc.h"
5961
5962 struct cpu cpus[NCPU];
5963 static struct cpu *bcpu;
5964 int ismp;
5965 int ncpu;
5966 uchar ioapicid;
5967
5968 int
5969 mpbcpu(void)
5970 {
5971   return bcpu-cpus;
5972 }
5973
5974 static uchar
5975 sum(uchar *addr, int len)
5976 {
5977   int i, sum;
5978
5979   sum = 0;
5980   for(i=0; i<len; i++)
5981     sum += addr[i];
5982   return sum;
5983 }
5984
5985 // Look for an MP structure in the len bytes at addr.
5986 static struct mp*
5987 mpsearch1(uchar *addr, int len)
5988 {
5989   uchar *e, *p;
5990
5991   cprintf("mpsearch1 0x%x %d\n", addr, len);
5992   e = addr+len;
5993   for(p = addr; p < e; p += sizeof(struct mp))
5994     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
5995       return (struct mp*)p;
5996   return 0;
5997 }
5998
5999
```

```
6000 // Search for the MP Floating Pointer Structure, which according to the
6001 // spec is in one of the following three locations:
6002 // 1) in the first KB of the EBDA;
6003 // 2) in the last KB of system base memory;
6004 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
6005 static struct mp*
6006 mpsearch(void)
6007 {
6008   uchar *bda;
6009   uint p;
6010   struct mp *mp;
6011
6012   bda = (uchar*)0x400;
6013   if((p = ((bda[0x0F]<<8)|bda[0x0E]) << 4)){
6014     if((mp = mpsearch1((uchar*)p, 1024)))
6015       return mp;
6016   } else {
6017     p = ((bda[0x14]<<8)|bda[0x13])*1024;
6018     if((mp = mpsearch1((uchar*)p-1024, 1024)))
6019       return mp;
6020   }
6021   return mpsearch1((uchar*)0xF0000, 0x10000);
6022 }
6023
6024 // Search for an MP configuration table.  For now,
6025 // don't accept the default configurations (physaddr == 0).
6026 // Check for correct signature, calculate the checksum and,
6027 // if correct, check the version.
6028 // To do: check extended table checksum.
6029 static struct mpconf*
6030 mpconfig(struct mp **pmp)
6031 {
6032   struct mpconf *conf;
6033   struct mp *mp;
6034
6035   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
6036     return 0;
6037   conf = (struct mpconf*)mp->physaddr;
6038   if(memcmp(conf, "PCMP", 4) != 0)
6039     return 0;
6040   if(conf->version != 1 && conf->version != 4)
6041     return 0;
6042   if(sum((uchar*)conf, conf->length) != 0)
6043     return 0;
6044   *pmp = mp;
6045   return conf;
6046 }
6047
6048
6049
```



```
6050 void
6051 mpinit(void)
6052 {
6053   uchar *p, *e;
6054   struct mp *mp;
6055   struct mpconf *conf;
6056   struct mpproc *proc;
6057   struct mpioapic *ioapic;
6058
6059   bcpu = &cpus[0];
6060   if((conf = mpconfig(&mp)) == 0)
6061     return;
6062   ismp = 1;
6063   lapic = (uint*)conf->lapicaddr;
6064   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
6065     switch(*p){
6066     case MPPROC:
6067       proc = (struct mpproc*)p;
6068       if(ncpu != proc->apicid) {
6069         cprintf("mpinit: ncpu=%d apicpid=%d", ncpu, proc->apicid);
6070         panic("mpinit");
6071       }
6072       if(proc->flags & MPBOOT)
6073         bcpu = &cpus[ncpu];
6074       cpus[ncpu].id = ncpu;
6075       ncpu++;
6076       p += sizeof(struct mpproc);
6077       continue;
6078     case MPIOAPIC:
6079       ioapic = (struct mpioapic*)p;
6080       ioapicid = ioapic->apicno;
6081       p += sizeof(struct mpioapic);
6082       continue;
6083     case MPBUS:
6084     case MPIOINTR:
6085     case MPLINTR:
6086       p += 8;
6087       continue;
6088     default:
6089       cprintf("mpinit: unknown config type %x\n", *p);
6090       panic("mpinit");
6091     }
6092   }
6093   if(mp->imcrp){
6094     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
6095     // But it would on real hardware.
6096     outb(0x22, 0x70);   // Select IMCR
6097     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
6098   }
6099 }
```

```
6100 // The local APIC manages internal (non-I/O) interrupts.
6101 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
6102
6103 #include "types.h"
6104 #include "defs.h"
6105 #include "traps.h"
6106 #include "mmu.h"
6107 #include "x86.h"
6108
6109 // Local APIC registers, divided by 4 for use as uint[] indices.
6110 #define ID      (0x0020/4)   // ID
6111 #define VER     (0x0030/4)   // Version
6112 #define TPR     (0x0080/4)   // Task Priority
6113 #define EOI     (0x00B0/4)   // EOI
6114 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
6115   #define ENABLE     0x00000100   // Unit Enable
6116 #define ESR     (0x0280/4)   // Error Status
6117 #define ICRLO   (0x0300/4)   // Interrupt Command
6118   #define INIT       0x00000500   // INIT/RESET
6119   #define STARTUP    0x00000600   // Startup IPI
6120   #define DELIVS     0x00001000   // Delivery status
6121   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
6122   #define DEASSERT   0x00000000
6123   #define LEVEL      0x00008000   // Level triggered
6124   #define BCAST      0x00080000   // Send to all APICs, including self.
6125   #define BUSY       0x00001000
6126   #define FIXED      0x00000000
6127 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
6128 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
6129   #define X1         0x0000000B   // divide counts by 1
6130   #define PERIODIC   0x00020000   // Periodic
6131 #define PCINT   (0x0340/4)   // Performance Counter LVT
6132 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
6133 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
6134 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
6135   #define MASKED     0x00010000   // Interrupt masked
6136 #define TICR    (0x0380/4)   // Timer Initial Count
6137 #define TCCR    (0x0390/4)   // Timer Current Count
6138 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
6139
6140 volatile uint *lapic;  // Initialized in mp.c
6141
6142 static void
6143 lapicw(int index, int value)
6144 {
6145   lapic[index] = value;
6146   lapic[ID];  // wait for write to finish, by reading
6147 }
6148
6149
```



```
6150 void
6151 lapicinit(int c)
6152 {
6153   cprintf("lapicinit: %d 0x%x\n", c, lapic);
6154   if(!lapic)
6155     return;
6156
6157   // Enable local APIC; set spurious interrupt vector.
6158   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
6159
6160   // The timer repeatedly counts down at bus frequency
6161   // from lapic[TICR] and then issues an interrupt.
6162   // If xv6 cared more about precise timekeeping,
6163   // TICR would be calibrated using an external time source.
6164   lapicw(TDCR, X1);
6165   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
6166   lapicw(TICR, 10000000);
6167
6168   // Disable logical interrupt lines.
6169   lapicw(LINT0, MASKED);
6170   lapicw(LINT1, MASKED);
6171
6172   // Disable performance counter overflow interrupts
6173   // on machines that provide that interrupt entry.
6174   if(((lapic[VER]>>16) & 0xFF) >= 4)
6175     lapicw(PCINT, MASKED);
6176
6177   // Map error interrupt to IRQ_ERROR.
6178   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
6179
6180   // Clear error status register (requires back-to-back writes).
6181   lapicw(ESR, 0);
6182   lapicw(ESR, 0);
6183
6184   // Ack any outstanding interrupts.
6185   lapicw(EOI, 0);
6186
6187   // Send an Init Level De-Assert to synchronise arbitration ID's.
6188   lapicw(ICRHI, 0);
6189   lapicw(ICRLO, BCAST | INIT | LEVEL);
6190   while(lapic[ICRLO] & DELIVS)
6191     ;
6192
6193   // Enable interrupts on the APIC (but not on the processor).
6194   lapicw(TPR, 0);
6195 }
6196
6197
6198
6199
```

```
6200 int
6201 cpunum(void)
6202 {
6203   // Cannot call cpu when interrupts are enabled:
6204   // result not guaranteed to last long enough to be used!
6205   // Would prefer to panic but even printing is chancy here:
6206   // almost everything, including cprintf and panic, calls cpu,
6207   // often indirectly through acquire and release.
6208   if(readeflags()&FL_IF){
6209     static int n;
6210     if(n++ == 0)
6211       cprintf("cpu called from %x with interrupts enabled\n",
6212         __builtin_return_address(0));
6213   }
6214
6215   if(lapic)
6216     return lapic[ID]>>24;
6217   return 0;
6218 }
6219
6220 // Acknowledge interrupt.
6221 void
6222 lapiceoi(void)
6223 {
6224   if(lapic)
6225     lapicw(EOI, 0);
6226 }
6227
6228 // Spin for a given number of microseconds.
6229 // On real hardware would want to tune this dynamically.
6230 void
6231 microdelay(int us)
6232 {
6233 }
6234
6235 #define IO_RTC  0x70
6236
6237 // Start additional processor running bootstrap code at addr.
6238 // See Appendix B of MultiProcessor Specification.
6239 void
6240 lapicstartap(uchar apicid, uint addr)
6241 {
6242   int i;
6243   ushort *wrv;
6244
6245   // "The BSP must initialize CMOS shutdown code to 0AH
6246   // and the warm reset vector (DWORD based at 40:67) to point at
6247   // the AP startup code prior to the [universal startup algorithm]."
6248   outb(IO_RTC, 0xF);  // offset 0xF is shutdown code
6249   outb(IO_RTC+1, 0x0A);
```



```
6250   wrv = (ushort*)(0x40<<4 | 0x67);  // Warm reset vector
6251   wrv[0] = 0;
6252   wrv[1] = addr >> 4;
6253
6254   // "Universal startup algorithm."
6255   // Send INIT (level-triggered) interrupt to reset other CPU.
6256   lapicw(ICRHI, apicid<<24);
6257   lapicw(ICRLO, INIT | LEVEL | ASSERT);
6258   microdelay(200);
6259   lapicw(ICRLO, INIT | LEVEL);
6260   microdelay(100);    // should be 10ms, but too slow in Bochs!
6261
6262   // Send startup IPI (twice!) to enter bootstrap code.
6263   // Regular hardware is supposed to only accept a STARTUP
6264   // when it is in the halted state due to an INIT.  So the second
6265   // should be ignored, but it is part of the official Intel algorithm.
6266   // Bochs complains about the second one.  Too bad for Bochs.
6267   for(i = 0; i < 2; i++){
6268     lapicw(ICRHI, apicid<<24);
6269     lapicw(ICRLO, STARTUP | (addr>>12));
6270     microdelay(200);
6271   }
6272 }
6273
6274
6275
6276
6277
6278
6279
6280
6281
6282
6283
6284
6285
6286
6287
6288
6289
6290
6291
6292
6293
6294
6295
6296
6297
6298
6299
```

```
6300 // The I/O APIC manages hardware interrupts for an SMP system.
6301 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
6302 // See also picirq.c.
6303
6304 #include "types.h"
6305 #include "defs.h"
6306 #include "traps.h"
6307
6308 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
6309
6310 #define REG_ID     0x00  // Register index: ID
6311 #define REG_VER    0x01  // Register index: version
6312 #define REG_TABLE  0x10  // Redirection table base
6313
6314 // The redirection table starts at REG_TABLE and uses
6315 // two registers to configure each interrupt.
6316 // The first (low) register in a pair contains configuration bits.
6317 // The second (high) register contains a bitmask telling which
6318 // CPUs can serve that interrupt.
6319 #define INT_DISABLED   0x00010000  // Interrupt disabled
6320 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
6321 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
6322 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
6323
6324 volatile struct ioapic *ioapic;
6325
6326 // IO APIC MMIO structure: write reg, then read or write data.
6327 struct ioapic {
6328   uint reg;
6329   uint pad[3];
6330   uint data;
6331 };
6332
6333 static uint
6334 ioapicread(int reg)
6335 {
6336   ioapic->reg = reg;
6337   return ioapic->data;
6338 }
6339
6340 static void
6341 ioapicwrite(int reg, uint data)
6342 {
6343   ioapic->reg = reg;
6344   ioapic->data = data;
6345 }
6346
6347
6348
6349
```



```
6350 void
6351 ioapicinit(void)
6352 {
6353   int i, id, maxintr;
6354
6355   if(!ismp)
6356     return;
6357
6358   ioapic = (volatile struct ioapic*)IOAPIC;
6359   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
6360   id = ioapicread(REG_ID) >> 24;
6361   if(id != ioapicid)
6362     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
6363
6364   // Mark all interrupts edge-triggered, active high, disabled,
6365   // and not routed to any CPUs.
6366   for(i = 0; i <= maxintr; i++){
6367     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
6368     ioapicwrite(REG_TABLE+2*i+1, 0);
6369   }
6370 }
6371
6372 void
6373 ioapicenable(int irq, int cpunum)
6374 {
6375   if(!ismp)
6376     return;
6377
6378   // Mark interrupt edge-triggered, active high,
6379   // enabled, and routed to the given cpunum,
6380   // which happens to be that cpu's APIC ID.
6381   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
6382   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
6383 }
6384
6385
6386
6387
6388
6389
6390
6391
6392
6393
6394
6395
6396
6397
6398
6399
```

```
6400 // Intel 8259A programmable interrupt controllers.
6401
6402 #include "types.h"
6403 #include "x86.h"
6404 #include "traps.h"
6405
6406 // I/O Addresses of the two programmable interrupt controllers
6407 #define IO_PIC1         0x20    // Master (IRQs 0-7)
6408 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
6409
6410 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
6411
6412 // Current IRQ mask.
6413 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
6414 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
6415
6416 static void
6417 picsetmask(ushort mask)
6418 {
6419   irqmask = mask;
6420   outb(IO_PIC1+1, mask);
6421   outb(IO_PIC2+1, mask >> 8);
6422 }
6423
6424 void
6425 picenable(int irq)
6426 {
6427   picsetmask(irqmask & ~(1<<irq));
6428 }
6429
6430 // Initialize the 8259A interrupt controllers.
6431 void
6432 picinit(void)
6433 {
6434   // mask all interrupts
6435   outb(IO_PIC1+1, 0xFF);
6436   outb(IO_PIC2+1, 0xFF);
6437
6438   // Set up master (8259A-1)
6439
6440   // ICW1:  0001g0hi
6441   //    g:  0 = edge triggering, 1 = level triggering
6442   //    h:  0 = cascaded PICs, 1 = master only
6443   //    i:  0 = no ICW4, 1 = ICW4 required
6444   outb(IO_PIC1, 0x11);
6445
6446   // ICW2:  Vector offset
6447   outb(IO_PIC1+1, T_IRQ0);
6448
6449
```



```
6450   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
6451   //        (slave PIC) 3-bit # of slave's connection to master
6452   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
6453
6454   // ICW4:  000nbmap
6455   //    n:  1 = special fully nested mode
6456   //    b:  1 = buffered mode
6457   //    m:  0 = slave PIC, 1 = master PIC
6458   //      (ignored when b is 0, as the master/slave role
6459   //      can be hardwired).
6460   //    a:  1 = Automatic EOI mode
6461   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
6462   outb(IO_PIC1+1, 0x3);
6463
6464   // Set up slave (8259A-2)
6465   outb(IO_PIC2, 0x11);                  // ICW1
6466   outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
6467   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
6468   // NB Automatic EOI mode doesn't tend to work on the slave.
6469   // Linux source code says it's "to be investigated".
6470   outb(IO_PIC2+1, 0x3);                 // ICW4
6471
6472   // OCW3:  0ef01prs
6473   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
6474   //    p:  0 = no polling, 1 = polling mode
6475   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
6476   outb(IO_PIC1, 0x68);             // clear specific mask
6477   outb(IO_PIC1, 0x0a);             // read IRR by default
6478
6479   outb(IO_PIC2, 0x68);             // OCW3
6480   outb(IO_PIC2, 0x0a);             // OCW3
6481
6482   if(irqmask != 0xFFFF)
6483     picsetmask(irqmask);
6484 }
6485
6486
6487
6488
6489
6490
6491
6492
6493
6494
6495
6496
6497
6498
6499
```

```
6500 // PC keyboard interface constants
6501
6502 #define KBSTATP         0x64    // kbd controller status port(I)
6503 #define KBS_DIB         0x01    // kbd data in buffer
6504 #define KBDATAP         0x60    // kbd data port(I)
6505
6506 #define NO              0
6507
6508 #define SHIFT           (1<<0)
6509 #define CTL             (1<<1)
6510 #define ALT             (1<<2)
6511
6512 #define CAPSLOCK        (1<<3)
6513 #define NUMLOCK         (1<<4)
6514 #define SCROLLLOCK      (1<<5)
6515
6516 #define E0ESC           (1<<6)
6517
6518 // Special keycodes
6519 #define KEY_HOME        0xE0
6520 #define KEY_END         0xE1
6521 #define KEY_UP          0xE2
6522 #define KEY_DN          0xE3
6523 #define KEY_LF          0xE4
6524 #define KEY_RT          0xE5
6525 #define KEY_PGUP        0xE6
6526 #define KEY_PGDN        0xE7
6527 #define KEY_INS         0xE8
6528 #define KEY_DEL         0xE9
6529
6530 // C('A') == Control-A
6531 #define C(x) (x - '@')
6532
6533 static uchar shiftcode[256] =
6534 {
6535   [0x1D] CTL,
6536   [0x2A] SHIFT,
6537   [0x36] SHIFT,
6538   [0x38] ALT,
6539   [0x9D] CTL,
6540   [0xB8] ALT
6541 };
6542
6543 static uchar togglecode[256] =
6544 {
6545   [0x3A] CAPSLOCK,
6546   [0x45] NUMLOCK,
6547   [0x46] SCROLLLOCK
6548 };
6549
```



```
6550 static uchar normalmap[256] =
6551 {
6552   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
6553   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
6554   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
6555   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
6556   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
6557   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
6558   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
6559   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
6560   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
6561   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
6562   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
6563   [0x9C] '\n',      // KP_Enter
6564   [0xB5] '/',       // KP_Div
6565   [0xC8] KEY_UP,    [0xD0] KEY_DN,
6566   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
6567   [0xCB] KEY_LF,    [0xCD] KEY_RT,
6568   [0x97] KEY_HOME,  [0xCF] KEY_END,
6569   [0xD2] KEY_INS,   [0xD3] KEY_DEL
6570 };
6571
6572 static uchar shiftmap[256] =
6573 {
6574   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
6575   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
6576   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
6577   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
6578   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
6579   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
6580   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
6581   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
6582   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
6583   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
6584   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
6585   [0x9C] '\n',      // KP_Enter
6586   [0xB5] '/',       // KP_Div
6587   [0xC8] KEY_UP,    [0xD0] KEY_DN,
6588   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
6589   [0xCB] KEY_LF,    [0xCD] KEY_RT,
6590   [0x97] KEY_HOME,  [0xCF] KEY_END,
6591   [0xD2] KEY_INS,   [0xD3] KEY_DEL
6592 };
6593
6594
6595
6596
6597
6598
6599
```

```
6600 static uchar ctlmap[256] =
6601 {
6602   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
6603   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
6604   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
6605   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
6606   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
6607   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
6608   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
6609   [0x9C] '\r',      // KP_Enter
6610   [0xB5] C('/'),    // KP_Div
6611   [0xC8] KEY_UP,    [0xD0] KEY_DN,
6612   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
6613   [0xCB] KEY_LF,    [0xCD] KEY_RT,
6614   [0x97] KEY_HOME,  [0xCF] KEY_END,
6615   [0xD2] KEY_INS,   [0xD3] KEY_DEL
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 "x86.h"
6652 #include "defs.h"
6653 #include "kbd.h"
6654
6655 int
6656 kbdgetc(void)
6657 {
6658   static uint shift;
6659   static uchar *charcode[4] = {
6660     normalmap, shiftmap, ctlmap, ctlmap
6661   };
6662   uint st, data, c;
6663
6664   st = inb(KBSTATP);
6665   if((st & KBS_DIB) == 0)
6666     return -1;
6667   data = inb(KBDATAP);
6668
6669   if(data == 0xE0){
6670     shift |= E0ESC;
6671     return 0;
6672   } else if(data & 0x80){
6673     // Key released
6674     data = (shift & E0ESC ? data : data & 0x7F);
6675     shift &= ~(shiftcode[data] | E0ESC);
6676     return 0;
6677   } else if(shift & E0ESC){
6678     // Last character was an E0 escape; or with 0x80
6679     data |= 0x80;
6680     shift &= ~E0ESC;
6681   }
6682
6683   shift |= shiftcode[data];
6684   shift ^= togglecode[data];
6685   c = charcode[shift & (CTL | SHIFT)][data];
6686   if(shift & CAPSLOCK){
6687     if('a' <= c && c <= 'z')
6688       c += 'A' - 'a';
6689     else if('A' <= c && c <= 'Z')
6690       c += 'a' - 'A';
6691   }
6692   return c;
6693 }
6694
6695 void
6696 kbdintr(void)
6697 {
6698   consoleintr(kbdgetc);
6699 }
```

```
6700 // Console input and output.
6701 // Input is from the keyboard or serial port.
6702 // Output is written to the screen and serial port.
6703
6704 #include "types.h"
6705 #include "defs.h"
6706 #include "param.h"
6707 #include "traps.h"
6708 #include "spinlock.h"
6709 #include "fs.h"
6710 #include "file.h"
6711 #include "mmu.h"
6712 #include "proc.h"
6713 #include "x86.h"
6714
6715 static void consputc(int);
6716
6717 static int panicked = 0;
6718
6719 static struct {
6720   struct spinlock lock;
6721   int locking;
6722 } cons;
6723
6724 static void
6725 printint(int xx, int base, int sgn)
6726 {
6727   static char digits[] = "0123456789abcdef";
6728   char buf[16];
6729   int i = 0, neg = 0;
6730   uint x;
6731
6732   if(sgn && xx < 0){
6733     neg = 1;
6734     x = -xx;
6735   } else
6736     x = xx;
6737
6738   do{
6739     buf[i++] = digits[x % base];
6740   }while((x /= base) != 0);
6741   if(neg)
6742     buf[i++] = '-';
6743
6744   while(--i >= 0)
6745     consputc(buf[i]);
6746 }
6747
6748
6749
```



```
6750 // Print to the console. only understands %d, %x, %p, %s.
6751 void
6752 cprintf(char *fmt, ...)
6753 {
6754   int i, c, state, locking;
6755   uint *argp;
6756   char *s;
6757
6758   locking = cons.locking;
6759   if(locking)
6760     acquire(&cons.lock);
6761
6762   argp = (uint*)(void*)(&fmt + 1);
6763   state = 0;
6764   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
6765     if(c != '%'){
6766       consputc(c);
6767       continue;
6768     }
6769     c = fmt[++i] & 0xff;
6770     if(c == 0)
6771       break;
6772     switch(c){
6773     case 'd':
6774       printint(*argp++, 10, 1);
6775       break;
6776     case 'x':
6777     case 'p':
6778       printint(*argp++, 16, 0);
6779       break;
6780     case 's':
6781       if((s = (char*)*argp++) == 0)
6782         s = "(null)";
6783       for(; *s; s++)
6784         consputc(*s);
6785       break;
6786     case '%':
6787       consputc('%');
6788       break;
6789     default:
6790       // Print unknown % sequence to draw attention.
6791       consputc('%');
6792       consputc(c);
6793       break;
6794     }
6795   }
6796
6797   if(locking)
6798     release(&cons.lock);
6799 }
```

```
6800 void
6801 panic(char *s)
6802 {
6803   int i;
6804   uint pcs[10];
6805
6806   cli();
6807   cons.locking = 0;
6808   cprintf("cpu%d: panic: ", cpu->id);
6809   cprintf(s);
6810   cprintf("\n");
6811   getcallerpcs(&s, pcs);
6812   for(i=0; i<10; i++)
6813     cprintf(" %p", pcs[i]);
6814   panicked = 1; // freeze other CPU
6815   for(;;)
6816     ;
6817 }
6818
6819
6820
6821
6822
6823
6824
6825
6826
6827
6828
6829
6830
6831
6832
6833
6834
6835
6836
6837
6838
6839
6840
6841
6842
6843
6844
6845
6846
6847
6848
6849
```



```
6850 #define BACKSPACE 0x100
6851 #define CRTPORT 0x3d4
6852 static ushort *crt = (ushort*)0xb8000;  // CGA memory
6853
6854 static void
6855 cgaputc(int c)
6856 {
6857   int pos;
6858
6859   // Cursor position: col + 80*row.
6860   outb(CRTPORT, 14);
6861   pos = inb(CRTPORT+1) << 8;
6862   outb(CRTPORT, 15);
6863   pos |= inb(CRTPORT+1);
6864
6865   if(c == '\n')
6866     pos += 80 - pos%80;
6867   else if(c == BACKSPACE){
6868     if(pos > 0) --pos;
6869   } else
6870     crt[pos++] = (c&0xff) | 0x0700;  // black on white
6871
6872   if((pos/80) >= 24){  // Scroll up.
6873     memmove(crt, crt+80, sizeof(crt[0])*23*80);
6874     pos -= 80;
6875     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
6876   }
6877
6878   outb(CRTPORT, 14);
6879   outb(CRTPORT+1, pos>>8);
6880   outb(CRTPORT, 15);
6881   outb(CRTPORT+1, pos);
6882   crt[pos] = ' ' | 0x0700;
6883 }
6884
6885 void
6886 consputc(int c)
6887 {
6888   if(panicked){
6889     cli();
6890     for(;;)
6891       ;
6892   }
6893
6894   if(c == BACKSPACE){
6895     uartputc('\b'); uartputc(' '); uartputc('\b');
6896   } else
6897     uartputc(c);
6898   cgaputc(c);
6899 }
```

```
6900 #define INPUT_BUF 128
6901 struct {
6902   struct spinlock lock;
6903   char buf[INPUT_BUF];
6904   uint r;  // Read index
6905   uint w;  // Write index
6906   uint e;  // Edit index
6907 } input;
6908
6909 #define C(x)  ((x)-'@')  // Control-x
6910
6911 void
6912 consoleintr(int (*getc)(void))
6913 {
6914   int c;
6915
6916   acquire(&input.lock);
6917   while((c = getc()) >= 0){
6918     switch(c){
6919     case C('P'):  // Process listing.
6920       procdump();
6921       break;
6922     case C('U'):  // Kill line.
6923       while(input.e != input.w &&
6924             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
6925         input.e--;
6926         consputc(BACKSPACE);
6927       }
6928       break;
6929     case C('H'): case '\x7f':  // Backspace
6930       if(input.e != input.w){
6931         input.e--;
6932         consputc(BACKSPACE);
6933       }
6934       break;
6935     default:
6936       if(c != 0 && input.e-input.r < INPUT_BUF){
6937         c = (c == '\r') ? '\n' : c;
6938         input.buf[input.e++ % INPUT_BUF] = c;
6939         consputc(c);
6940         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
6941           input.w = input.e;
6942           wakeup(&input.r);
6943         }
6944       }
6945       break;
6946     }
6947   }
6948   release(&input.lock);
6949 }
```



```
6950 int
6951 consoleread(struct inode *ip, char *dst, int n)
6952 {
6953   uint target;
6954   int c;
6955
6956   iunlock(ip);
6957   target = n;
6958   acquire(&input.lock);
6959   while(n > 0){
6960     while(input.r == input.w){
6961       if(proc->killed){
6962         release(&input.lock);
6963         ilock(ip);
6964         return -1;
6965       }
6966       sleep(&input.r, &input.lock);
6967     }
6968     c = input.buf[input.r++ % INPUT_BUF];
6969     if(c == C('D')){  // EOF
6970       if(n < target){
6971         // Save ^D for next time, to make sure
6972         // caller gets a 0-byte result.
6973         input.r--;
6974       }
6975       break;
6976     }
6977     *dst++ = c;
6978     --n;
6979     if(c == '\n')
6980       break;
6981   }
6982   release(&input.lock);
6983   ilock(ip);
6984
6985   return target - n;
6986 }
6987
6988
6989
6990
6991
6992
6993
6994
6995
6996
6997
6998
6999
```

```
7000 int
7001 consolewrite(struct inode *ip, char *buf, int n)
7002 {
7003   int i;
7004
7005   iunlock(ip);
7006   acquire(&cons.lock);
7007   for(i = 0; i < n; i++)
7008     consputc(buf[i] & 0xff);
7009   release(&cons.lock);
7010   ilock(ip);
7011
7012   return n;
7013 }
7014
7015 void
7016 consoleinit(void)
7017 {
7018   initlock(&cons.lock, "console");
7019   initlock(&input.lock, "input");
7020
7021   devsw[CONSOLE].write = consolewrite;
7022   devsw[CONSOLE].read = consoleread;
7023   cons.locking = 1;
7024
7025   picenable(IRQ_KBD);
7026   ioapicenable(IRQ_KBD, 0);
7027 }
7028
7029
7030
7031
7032
7033
7034
7035
7036
7037
7038
7039
7040
7041
7042
7043
7044
7045
7046
7047
7048
7049
```



```
7050 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
7051 // Only used on uniprocessors;
7052 // SMP machines use the local APIC timer.
7053
7054 #include "types.h"
7055 #include "defs.h"
7056 #include "traps.h"
7057 #include "x86.h"
7058
7059 #define IO_TIMER1       0x040           // 8253 Timer #1
7060
7061 // Frequency of all three count-down timers;
7062 // (TIMER_FREQ/freq) is the appropriate count
7063 // to generate a frequency of freq Hz.
7064
7065 #define TIMER_FREQ      1193182
7066 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
7067
7068 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
7069 #define TIMER_SEL0      0x00    // select counter 0
7070 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
7071 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
7072
7073 void
7074 timerinit(void)
7075 {
7076   // Interrupt 100 times/sec.
7077   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
7078   outb(IO_TIMER1, TIMER_DIV(100) % 256);
7079   outb(IO_TIMER1, TIMER_DIV(100) / 256);
7080   picenable(IRQ_TIMER);
7081 }
7082
7083
7084
7085
7086
7087
7088
7089
7090
7091
7092
7093
7094
7095
7096
7097
7098
7099
```

```
7100 // Intel 8250 serial port (UART).
7101
7102 #include "types.h"
7103 #include "defs.h"
7104 #include "param.h"
7105 #include "traps.h"
7106 #include "spinlock.h"
7107 #include "fs.h"
7108 #include "file.h"
7109 #include "mmu.h"
7110 #include "proc.h"
7111 #include "x86.h"
7112
7113 #define COM1    0x3f8
7114
7115 static int uart;    // is there a uart?
7116
7117 void
7118 uartinit(void)
7119 {
7120   char *p;
7121
7122   // Turn off the FIFO
7123   outb(COM1+2, 0);
7124
7125   // 9600 baud, 8 data bits, 1 stop bit, parity off.
7126   outb(COM1+3, 0x80);    // Unlock divisor
7127   outb(COM1+0, 115200/9600);
7128   outb(COM1+1, 0);
7129   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
7130   outb(COM1+4, 0);
7131   outb(COM1+1, 0x01);    // Enable receive interrupts.
7132
7133   // If status is 0xFF, no serial port.
7134   if(inb(COM1+5) == 0xFF)
7135     return;
7136   uart = 1;
7137
7138   // Acknowledge pre-existing interrupt conditions;
7139   // enable interrupts.
7140   inb(COM1+2);
7141   inb(COM1+0);
7142   picenable(IRQ_COM1);
7143   ioapicenable(IRQ_COM1, 0);
7144
7145   // Announce that we're here.
7146   for(p="xv6...\n"; *p; p++)
7147     uartputc(*p);
7148 }
7149
```



```
7150 void
7151 uartputc(int c)
7152 {
7153   int i;
7154
7155   if(!uart)
7156     return;
7157   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
7158     microdelay(10);
7159   outb(COM1+0, c);
7160 }
7161
7162 static int
7163 uartgetc(void)
7164 {
7165   if(!uart)
7166     return -1;
7167   if(!(inb(COM1+5) & 0x01))
7168     return -1;
7169   return inb(COM1+0);
7170 }
7171
7172 void
7173 uartintr(void)
7174 {
7175   consoleintr(uartgetc);
7176 }
7177
7178
7179
7180
7181
7182
7183
7184
7185
7186
7187
7188
7189
7190
7191
7192
7193
7194
7195
7196
7197
7198
7199
```

```
7200 # Initial process execs /init.
7201
7202 #include "syscall.h"
7203 #include "traps.h"
7204
7205 # exec(init, argv)
7206 .globl start
7207 start:
7208   pushl $argv
7209   pushl $init
7210   pushl $0  // where caller pc would be
7211   movl $SYS_exec, %eax
7212   int $T_SYSCALL
7213
7214 # for(;;) exit();
7215 exit:
7216   movl $SYS_exit, %eax
7217   int $T_SYSCALL
7218   jmp exit
7219
7220 # char init[] = "/init\0";
7221 init:
7222   .string "/init\0"
7223
7224 # char *argv[] = { init, 0 };
7225 .p2align 2
7226 argv:
7227   .long init
7228   .long 0
7229
7230
7231
7232
7233
7234
7235
7236
7237
7238
7239
7240
7241
7242
7243
7244
7245
7246
7247
7248
7249
```



```
7250 #include "syscall.h"
7251 #include "traps.h"
7252
7253 #define SYSCALL(name) \
7254   .globl name; \
7255   name: \
7256     movl $SYS_ ## name, %eax; \
7257     int $T_SYSCALL; \
7258     ret
7259
7260 SYSCALL(fork)
7261 SYSCALL(exit)
7262 SYSCALL(wait)
7263 SYSCALL(pipe)
7264 SYSCALL(read)
7265 SYSCALL(write)
7266 SYSCALL(close)
7267 SYSCALL(kill)
7268 SYSCALL(exec)
7269 SYSCALL(open)
7270 SYSCALL(mknod)
7271 SYSCALL(unlink)
7272 SYSCALL(fstat)
7273 SYSCALL(link)
7274 SYSCALL(mkdir)
7275 SYSCALL(chdir)
7276 SYSCALL(dup)
7277 SYSCALL(getpid)
7278 SYSCALL(sbrk)
7279 SYSCALL(sleep)
7280 SYSCALL(uptime)
7281
7282
7283
7284
7285
7286
7287
7288
7289
7290
7291
7292
7293
7294
7295
7296
7297
7298
7299
```

```
7300 // init: The initial user-level program
7301
7302 #include "types.h"
7303 #include "stat.h"
7304 #include "user.h"
7305 #include "fcntl.h"
7306
7307 char *argv[] = { "sh", 0 };
7308
7309 int
7310 main(void)
7311 {
7312   int pid, wpid;
7313
7314   if(open("console", O_RDWR) < 0){
7315     mknod("console", 1, 1);
7316     open("console", O_RDWR);
7317   }
7318   dup(0);  // stdout
7319   dup(0);  // stderr
7320
7321   for(;;){
7322     printf(1, "init: starting sh\n");
7323     pid = fork();
7324     if(pid < 0){
7325       printf(1, "init: fork failed\n");
7326       exit();
7327     }
7328     if(pid == 0){
7329       exec("sh", argv);
7330       printf(1, "init: exec sh failed\n");
7331       exit();
7332     }
7333     while((wpid=wait()) >= 0 && wpid != pid)
7334       printf(1, "zombie!\n");
7335   }
7336 }
7337
7338
7339
7340
7341
7342
7343
7344
7345
7346
7347
7348
7349
```



```
7350 // Shell.
7351
7352 #include "types.h"
7353 #include "user.h"
7354 #include "fcntl.h"
7355
7356 // Parsed command representation
7357 #define EXEC  1
7358 #define REDIR 2
7359 #define PIPE  3
7360 #define LIST  4
7361 #define BACK  5
7362
7363 #define MAXARGS 10
7364
7365 struct cmd {
7366   int type;
7367 };
7368
7369 struct execcmd {
7370   int type;
7371   char *argv[MAXARGS];
7372   char *eargv[MAXARGS];
7373 };
7374
7375 struct redircmd {
7376   int type;
7377   struct cmd *cmd;
7378   char *file;
7379   char *efile;
7380   int mode;
7381   int fd;
7382 };
7383
7384 struct pipecmd {
7385   int type;
7386   struct cmd *left;
7387   struct cmd *right;
7388 };
7389
7390 struct listcmd {
7391   int type;
7392   struct cmd *left;
7393   struct cmd *right;
7394 };
7395
7396 struct backcmd {
7397   int type;
7398   struct cmd *cmd;
7399 };
```

```
7400 int fork1(void);  // Fork but panics on failure.
7401 void panic(char*);
7402 struct cmd *parsecmd(char*);
7403
7404 // Execute cmd.  Never returns.
7405 void
7406 runcmd(struct cmd *cmd)
7407 {
7408   int p[2];
7409   struct backcmd *bcmd;
7410   struct execcmd *ecmd;
7411   struct listcmd *lcmd;
7412   struct pipecmd *pcmd;
7413   struct redircmd *rcmd;
7414
7415   if(cmd == 0)
7416     exit();
7417
7418   switch(cmd->type){
7419   default:
7420     panic("runcmd");
7421
7422   case EXEC:
7423     ecmd = (struct execcmd*)cmd;
7424     if(ecmd->argv[0] == 0)
7425       exit();
7426     exec(ecmd->argv[0], ecmd->argv);
7427     printf(2, "exec %s failed\n", ecmd->argv[0]);
7428     break;
7429
7430   case REDIR:
7431     rcmd = (struct redircmd*)cmd;
7432     close(rcmd->fd);
7433     if(open(rcmd->file, rcmd->mode) < 0){
7434       printf(2, "open %s failed\n", rcmd->file);
7435       exit();
7436     }
7437     runcmd(rcmd->cmd);
7438     break;
7439
7440   case LIST:
7441     lcmd = (struct listcmd*)cmd;
7442     if(fork1() == 0)
7443       runcmd(lcmd->left);
7444     wait();
7445     runcmd(lcmd->right);
7446     break;
7447
7448
7449
```



```
7450   case PIPE:
7451     pcmd = (struct pipecmd*)cmd;
7452     if(pipe(p) < 0)
7453       panic("pipe");
7454     if(fork1() == 0){
7455       close(1);
7456       dup(p[1]);
7457       close(p[0]);
7458       close(p[1]);
7459       runcmd(pcmd->left);
7460     }
7461     if(fork1() == 0){
7462       close(0);
7463       dup(p[0]);
7464       close(p[0]);
7465       close(p[1]);
7466       runcmd(pcmd->right);
7467     }
7468     close(p[0]);
7469     close(p[1]);
7470     wait();
7471     wait();
7472     break;
7473
7474   case BACK:
7475     bcmd = (struct backcmd*)cmd;
7476     if(fork1() == 0)
7477       runcmd(bcmd->cmd);
7478     break;
7479   }
7480   exit();
7481 }
7482
7483 int
7484 getcmd(char *buf, int nbuf)
7485 {
7486   printf(2, "$ ");
7487   memset(buf, 0, nbuf);
7488   gets(buf, nbuf);
7489   if(buf[0] == 0) // EOF
7490     return -1;
7491   return 0;
7492 }
7493
7494
7495
7496
7497
7498
7499
```

```
7500 int
7501 main(void)
7502 {
7503   static char buf[100];
7504   int fd;
7505
7506   // Assumes three file descriptors open.
7507   while((fd = open("console", O_RDWR)) >= 0){
7508     if(fd >= 3){
7509       close(fd);
7510       break;
7511     }
7512   }
7513
7514   // Read and run input commands.
7515   while(getcmd(buf, sizeof(buf)) >= 0){
7516     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
7517       // Clumsy but will have to do for now.
7518       // Chdir has no effect on the parent if run in the child.
7519       buf[strlen(buf)-1] = 0;  // chop \n
7520       if(chdir(buf+3) < 0)
7521         printf(2, "cannot cd %s\n", buf+3);
7522       continue;
7523     }
7524     if(fork1() == 0)
7525       runcmd(parsecmd(buf));
7526     wait();
7527   }
7528   exit();
7529 }
7530
7531 void
7532 panic(char *s)
7533 {
7534   printf(2, "%s\n", s);
7535   exit();
7536 }
7537
7538 int
7539 fork1(void)
7540 {
7541   int pid;
7542
7543   pid = fork();
7544   if(pid == -1)
7545     panic("fork");
7546   return pid;
7547 }
7548
7549
```



```
7550 // Constructors
7551
7552 struct cmd*
7553 execcmd(void)
7554 {
7555   struct execcmd *cmd;
7556
7557   cmd = malloc(sizeof(*cmd));
7558   memset(cmd, 0, sizeof(*cmd));
7559   cmd->type = EXEC;
7560   return (struct cmd*)cmd;
7561 }
7562
7563 struct cmd*
7564 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
7565 {
7566   struct redircmd *cmd;
7567
7568   cmd = malloc(sizeof(*cmd));
7569   memset(cmd, 0, sizeof(*cmd));
7570   cmd->type = REDIR;
7571   cmd->cmd = subcmd;
7572   cmd->file = file;
7573   cmd->efile = efile;
7574   cmd->mode = mode;
7575   cmd->fd = fd;
7576   return (struct cmd*)cmd;
7577 }
7578
7579 struct cmd*
7580 pipecmd(struct cmd *left, struct cmd *right)
7581 {
7582   struct pipecmd *cmd;
7583
7584   cmd = malloc(sizeof(*cmd));
7585   memset(cmd, 0, sizeof(*cmd));
7586   cmd->type = PIPE;
7587   cmd->left = left;
7588   cmd->right = right;
7589   return (struct cmd*)cmd;
7590 }
7591
7592
7593
7594
7595
7596
7597
7598
7599
```

```
7600 struct cmd*
7601 listcmd(struct cmd *left, struct cmd *right)
7602 {
7603   struct listcmd *cmd;
7604
7605   cmd = malloc(sizeof(*cmd));
7606   memset(cmd, 0, sizeof(*cmd));
7607   cmd->type = LIST;
7608   cmd->left = left;
7609   cmd->right = right;
7610   return (struct cmd*)cmd;
7611 }
7612
7613 struct cmd*
7614 backcmd(struct cmd *subcmd)
7615 {
7616   struct backcmd *cmd;
7617
7618   cmd = malloc(sizeof(*cmd));
7619   memset(cmd, 0, sizeof(*cmd));
7620   cmd->type = BACK;
7621   cmd->cmd = subcmd;
7622   return (struct cmd*)cmd;
7623 }
7624
7625
7626
7627
7628
7629
7630
7631
7632
7633
7634
7635
7636
7637
7638
7639
7640
7641
7642
7643
7644
7645
7646
7647
7648
7649
```



```
7650 // Parsing
7651
7652 char whitespace[] = " \t\r\n\v";
7653 char symbols[] = "<|>&;()";
7654
7655 int
7656 gettoken(char **ps, char *es, char **q, char **eq)
7657 {
7658   char *s;
7659   int ret;
7660
7661   s = *ps;
7662   while(s < es && strchr(whitespace, *s))
7663     s++;
7664   if(q)
7665     *q = s;
7666   ret = *s;
7667   switch(*s){
7668   case 0:
7669     break;
7670   case '|':
7671   case '(':
7672   case ')':
7673   case ';':
7674   case '&':
7675   case '<':
7676     s++;
7677     break;
7678   case '>':
7679     s++;
7680     if(*s == '>'){
7681       ret = '+';
7682       s++;
7683     }
7684     break;
7685   default:
7686     ret = 'a';
7687     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
7688       s++;
7689     break;
7690   }
7691   if(eq)
7692     *eq = s;
7693
7694   while(s < es && strchr(whitespace, *s))
7695     s++;
7696   *ps = s;
7697   return ret;
7698 }
7699
```

```
7700 int
7701 peek(char **ps, char *es, char *toks)
7702 {
7703   char *s;
7704
7705   s = *ps;
7706   while(s < es && strchr(whitespace, *s))
7707     s++;
7708   *ps = s;
7709   return *s && strchr(toks, *s);
7710 }
7711
7712 struct cmd *parseline(char**, char*);
7713 struct cmd *parsepipe(char**, char*);
7714 struct cmd *parseexec(char**, char*);
7715 struct cmd *nulterminate(struct cmd*);
7716
7717 struct cmd*
7718 parsecmd(char *s)
7719 {
7720   char *es;
7721   struct cmd *cmd;
7722
7723   es = s + strlen(s);
7724   cmd = parseline(&s, es);
7725   peek(&s, es, "");
7726   if(s != es){
7727     printf(2, "leftovers: %s\n", s);
7728     panic("syntax");
7729   }
7730   nulterminate(cmd);
7731   return cmd;
7732 }
7733
7734 struct cmd*
7735 parseline(char **ps, char *es)
7736 {
7737   struct cmd *cmd;
7738
7739   cmd = parsepipe(ps, es);
7740   while(peek(ps, es, "&")){
7741     gettoken(ps, es, 0, 0);
7742     cmd = backcmd(cmd);
7743   }
7744   if(peek(ps, es, ";")){
7745     gettoken(ps, es, 0, 0);
7746     cmd = listcmd(cmd, parseline(ps, es));
7747   }
7748   return cmd;
7749 }
```



```
7750 struct cmd*
7751 parsepipe(char **ps, char *es)
7752 {
7753   struct cmd *cmd;
7754
7755   cmd = parseexec(ps, es);
7756   if(peek(ps, es, "|")){
7757     gettoken(ps, es, 0, 0);
7758     cmd = pipecmd(cmd, parsepipe(ps, es));
7759   }
7760   return cmd;
7761 }
7762
7763 struct cmd*
7764 parseredirs(struct cmd *cmd, char **ps, char *es)
7765 {
7766   int tok;
7767   char *q, *eq;
7768
7769   while(peek(ps, es, "<>")){
7770     tok = gettoken(ps, es, 0, 0);
7771     if(gettoken(ps, es, &q, &eq) != 'a')
7772       panic("missing file for redirection");
7773     switch(tok){
7774     case '<':
7775       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
7776       break;
7777     case '>':
7778       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
7779       break;
7780     case '+':  // >>
7781       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
7782       break;
7783     }
7784   }
7785   return cmd;
7786 }
7787
7788
7789
7790
7791
7792
7793
7794
7795
7796
7797
7798
7799
```

```
7800 struct cmd*
7801 parseblock(char **ps, char *es)
7802 {
7803   struct cmd *cmd;
7804
7805   if(!peek(ps, es, "("))
7806     panic("parseblock");
7807   gettoken(ps, es, 0, 0);
7808   cmd = parseline(ps, es);
7809   if(!peek(ps, es, ")"))
7810     panic("syntax - missing )");
7811   gettoken(ps, es, 0, 0);
7812   cmd = parseredirs(cmd, ps, es);
7813   return cmd;
7814 }
7815
7816 struct cmd*
7817 parseexec(char **ps, char *es)
7818 {
7819   char *q, *eq;
7820   int tok, argc;
7821   struct execcmd *cmd;
7822   struct cmd *ret;
7823
7824   if(peek(ps, es, "("))
7825     return parseblock(ps, es);
7826
7827   ret = execcmd();
7828   cmd = (struct execcmd*)ret;
7829
7830   argc = 0;
7831   ret = parseredirs(ret, ps, es);
7832   while(!peek(ps, es, "|)&;")){
7833     if((tok=gettoken(ps, es, &q, &eq)) == 0)
7834       break;
7835     if(tok != 'a')
7836       panic("syntax");
7837     cmd->argv[argc] = q;
7838     cmd->eargv[argc] = eq;
7839     argc++;
7840     if(argc >= MAXARGS)
7841       panic("too many args");
7842     ret = parseredirs(ret, ps, es);
7843   }
7844   cmd->argv[argc] = 0;
7845   cmd->eargv[argc] = 0;
7846   return ret;
7847 }
7848
7849
```



```
7850 // NUL-terminate all the counted strings.
7851 struct cmd*
7852 nulterminate(struct cmd *cmd)
7853 {
7854   int i;
7855   struct backcmd *bcmd;
7856   struct execcmd *ecmd;
7857   struct listcmd *lcmd;
7858   struct pipecmd *pcmd;
7859   struct redircmd *rcmd;
7860
7861   if(cmd == 0)
7862     return 0;
7863
7864   switch(cmd->type){
7865   case EXEC:
7866     ecmd = (struct execcmd*)cmd;
7867     for(i=0; ecmd->argv[i]; i++)
7868       *ecmd->eargv[i] = 0;
7869     break;
7870
7871   case REDIR:
7872     rcmd = (struct redircmd*)cmd;
7873     nulterminate(rcmd->cmd);
7874     *rcmd->efile = 0;
7875     break;
7876
7877   case PIPE:
7878     pcmd = (struct pipecmd*)cmd;
7879     nulterminate(pcmd->left);
7880     nulterminate(pcmd->right);
7881     break;
7882
7883   case LIST:
7884     lcmd = (struct listcmd*)cmd;
7885     nulterminate(lcmd->left);
7886     nulterminate(lcmd->right);
7887     break;
7888
7889   case BACK:
7890     bcmd = (struct backcmd*)cmd;
7891     nulterminate(bcmd->cmd);
7892     break;
7893   }
7894   return cmd;
7895 }
7896
7897
7898
7899
```