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

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14,
2000)). See also http://pdos.csail.mit.edu/6.828/2014/xv6.html, which
provides pointers to on-line resources for v6.

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

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

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

ERROR REPORTS

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

BUILDING AND RUNNING XV6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run "make".
On non-x86 or non-ELF machines (like OS X, even on x86), you will
need to install a cross-compiler gcc suite capable of producing x86 ELF
binaries. See http://pdos.csail.mit.edu/6.828/2014/tools.html.
Then run "make TOOLPREFIX=i386-jos-elf-".

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

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

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

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

    swtch 2658
        0374 2428 2466 2657 2658

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

```
    3857 4017 8962
sys_exit 4166
    3983 4012 4166
SYS_exit 3852
    3852 4012 8967
sys_fork 4160
    3984 4011 4160
SYS_fork 3851
    3851 4011
sys_fstat 6551
    3985 4018 6551
SYS_fstat 3858
    3858 4018
sys_getgid 4273
    4003 4035 4273
SYS_getgid 3875
    3875 4035
sys_getpid 4189
    3986 4021 4189
SYS_getpid 3861
    3861 4021
sys_getppid 4278
    4004 4036 4278
SYS_getppid 3876
    3876 4036
sys_getpriority 4315
    4008 4040 4315
SYS_getpriority 3880
    3880 4040
sys_getprocs 4301
    4007 4039 4301
SYS_getprocs 3879
    3879 4039
sys_getuid 4268
    4002 4034 4268
SYS_getuid 3874
    3874 4034
SYS_halt 3872
    3872 4032
sys_kill 4179
    3987 4016 4179
SYS_kill 3856
    3856 4016
sys_link 6563
    3988 4029 6563
SYS_link 3869
    3869 4029
sys_mkdir 6801
    3989 4030 6801
SYS_mkdir 3870
    3870 4030
sys_mknod 6817
    3990 4027 6817
SYS_mknod 3867
    3867 4027
sys_open 6751
    3991 4025 6751
SYS_open 3865
    3865 4025
sys_pipe 6901
    3992 4014 6901
SYS_pipe 3854
    3854 4014
sys_read 6515
    3993 4015 6515
SYS_read 3855
    3855 4015
sys_sbrk 4201
    3994 4022 4201
SYS_sbrk 3862
    3862 4022
sys_setgid 4292
    4006 4038 4292
SYS_setgid 3878
    3878 4038
sys_setpriority 4324
    4009 4041 4324
SYS_setpriority 3881
    3881 4041
sys_setuid 4283
    4005 4037 4283
SYS_setuid 3877
    3877 4037
sys_sleep 4215
    3995 4023 4215
SYS_sleep 3863
    3863 4023
sys_unlink 6651
    3996 4028 6651
SYS_unlink 3868
    3868 4028
sys_uptime 4238
    3999 4024 4238
SYS_uptime 3864
    3864 4024
sys_wait 4173
    3997 4013 4173
SYS_wait 3853
    3853 4013
sys_write 6527
```



```
    3998 4026 6527
SYS_write 3866
    3866 4026
taskstate 0851
    0851 2357
TDCR 7790
    7790 7863
T_DEV 4502
    4502 5907 5957 6828
T_DIR 4500
    4500 6016 6166 6578 6679
    6687 6735 6775 6807 6862
T_FILE 4501
    4501 6720 6764
ticks 3714
    0413 3714 3767 3768 4223
    4224 4229 4243
tickslock 3713
    0415 3713 3725 3766 3769
    4222 4226 4229 4231 4242
    4244
TICR 7788
    7788 7865
TIMER 7780
    7780 7864
TIMER_16BIT 8821
    8821 8827
TIMER_DIV 8816
    8816 8828 8829
TIMER_FREQ 8815
    8815 8816
timerinit 8824
    0409 1285 8824
TIMER_MODE 8818
    8818 8827
TIMER_RATEGEN 8820
    8820 8827
TIMER_SEL0 8819
    8819 8827
T_IRQ0 3579
    3579 3764 3773 3777 3780
    3784 3788 3789 3823 7857
    7864 7877 8117 8131 8197
    8216
TPR 7764
    7764 7893
trap 3751
    3602 3604 3672 3751 3803
    3805 3808
trapframe 0602
    0602 2410 2629 3751
trapret 3677
    2472 2634 3676 3677
T_SYSCALL 3576
    3576 3723 3753 8963 8968
    9007
tvinit 3717
    0414 1280 3717
uart 8865
    8865 8886 8905 8915
uartgetc 8913
    8913 8925
uartinit 8868
    0418 1278 8868
uartintr 8923
    0419 3785 8923
uartputc 8901
    0420 8660 8662 8897 8901
uproc 1002
    0260 0363 1002 3251 4303
    10107
userinit 2652
    0371 1288 2652 2669
uva2ka 2152
    0427 2152 2176
V2P 0217
    0217 1880 1881
V2P_WO 0220
    0220 1086 1096
VER 7763
    7763 7873
wait 2819
    0372 2819 4061 4175 9012
    9082 9199 9223 9224 9383
    10070
waitdisk 9951
    9951 9963 9972
wakeup 3166
    0373 3166 3768 4872 5091
    5316 5326 5742 5772 7166
    7169 7191 7196 7218 8707
wakeup1 3153
    2474 2795 2805 3153 3169
walkpgdir 1804
    1804 1837 1976 2042 2083
    2113 2156
write_head 5254
    5254 5273 5355 5358
writei 5952
    0303 5952 6074 6426 6685
```

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



```
0150 #define NPROC        64  // maximum number of processes
0151 #define KSTACKSIZE 4096  // size of per-process kernel stack
0152 #define NCPU          8  // maximum number of CPUs
0153 #define NOFILE       16  // open files per process
0154 #define NFILE       100  // open files per system
0155 #define NINODE       50  // maximum number of active i-nodes
0156 #define NDEV         10  // maximum major device number
0157 #define ROOTDEV       1  // device number of file system root disk
0158 #define MAXARG       32  // max exec arguments
0159 #define MAXOPBLOCKS  10  // max # of blocks any FS op writes
0160 #define LOGSIZE      (MAXOPBLOCKS*3)  // max data blocks in on-disk log
0161 #define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
0162 #define FSSIZE       1000  // size of file system in blocks
0163 #define MaxTimeToReset 99999999
0164
0165
0166
0167
0168
0169
0170
0171
0172
0173
0174
0175
0176
0177
0178
0179
0180
0181
0182
0183
0184
0185
0186
0187
0188
0189
0190
0191
0192
0193
0194
0195
0196
0197
0198
0199
```

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

```
1000 #include "types.h"
1001
1002 struct uproc {
1003   int pid;
1004   int uid;
1005   int gid;
1006   int ppid;
1007   int state;
1008   int priority;
1009   uint size;
1010   char name[16];
1011 };
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
```



```
1050 # Multiboot header, for multiboot boot loaders like GNU Grub.
1051 # http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
1052 #
1053 # Using GRUB 2, you can boot xv6 from a file stored in a
1054 # Linux file system by copying kernel or kernelmemfs to /boot
1055 # and then adding this menu entry:
1056 #
1057 # menuentry "xv6" {
1058 #  insmod ext2
1059 #  set root='(hd0,msdos1)'
1060 #  set kernel='/boot/kernel'
1061 #  echo "Loading ${kernel}..."
1062 #  multiboot ${kernel} ${kernel}
1063 #  boot
1064 # }
1065
1066 #include "asm.h"
1067 #include "memlayout.h"
1068 #include "mmu.h"
1069 #include "param.h"
1070
1071 # Multiboot header.  Data to direct multiboot loader.
1072 .p2align 2
1073 .text
1074 .globl multiboot_header
1075 multiboot_header:
1076   #define magic 0x1badb002
1077   #define flags 0
1078   .long magic
1079   .long flags
1080   .long (-magic-flags)
1081
1082 # By convention, the _start symbol specifies the ELF entry point.
1083 # Since we haven't set up virtual memory yet, our entry point is
1084 # the physical address of 'entry'.
1085 .globl _start
1086 _start = V2P_WO(entry)
1087
1088 # Entering xv6 on boot processor, with paging off.
1089 .globl entry
1090 entry:
1091   # Turn on page size extension for 4Mbyte pages
1092   movl    %cr4, %eax
1093   orl     $(CR4_PSE), %eax
1094   movl    %eax, %cr4
1095   # Set page directory
1096   movl    $(V2P_WO(entrypgdir)), %eax
1097   movl    %eax, %cr3
1098   # Turn on paging.
1099   movl    %cr0, %eax
```

```
1100   orl     $(CR0_PG|CR0_WP), %eax
1101   movl    %eax, %cr0
1102
1103   # Set up the stack pointer.
1104   movl $(stack + KSTACKSIZE), %esp
1105
1106   # Jump to main(), and switch to executing at
1107   # high addresses. The indirect call is needed because
1108   # the assembler produces a PC-relative instruction
1109   # for a direct jump.
1110   mov $main, %eax
1111   jmp *%eax
1112
1113 .comm stack, KSTACKSIZE
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
```



```
1150 #include "asm.h"
1151 #include "memlayout.h"
1152 #include "mmu.h"
1153
1154 # Each non-boot CPU ("AP") is started up in response to a STARTUP
1155 # IPI from the boot CPU.  Section B.4.2 of the Multi-Processor
1156 # Specification says that the AP will start in real mode with CS:IP
1157 # set to XY00:0000, where XY is an 8-bit value sent with the
1158 # STARTUP. Thus this code must start at a 4096-byte boundary.
1159 #
1160 # Because this code sets DS to zero, it must sit
1161 # at an address in the low 2^16 bytes.
1162 #
1163 # Startothers (in main.c) sends the STARTUPs one at a time.
1164 # It copies this code (start) at 0x7000.  It puts the address of
1165 # a newly allocated per-core stack in start-4,the address of the
1166 # place to jump to (mpenter) in start-8, and the physical address
1167 # of entrypgdir in start-12.
1168 #
1169 # This code is identical to bootasm.S except:
1170 #   - it does not need to enable A20
1171 #   - it uses the address at start-4, start-8, and start-12
1172
1173 .code16
1174 .globl start
1175 start:
1176   cli
1177
1178   xorw    %ax,%ax
1179   movw    %ax,%ds
1180   movw    %ax,%es
1181   movw    %ax,%ss
1182
1183   lgdt    gdtdesc
1184   movl    %cr0, %eax
1185   orl     $CR0_PE, %eax
1186   movl    %eax, %cr0
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
```

```
1200   ljmpl    $(SEG_KCODE<<3), $(start32)
1201
1202 .code32
1203 start32:
1204   movw    $(SEG_KDATA<<3), %ax
1205   movw    %ax, %ds
1206   movw    %ax, %es
1207   movw    %ax, %ss
1208   movw    $0, %ax
1209   movw    %ax, %fs
1210   movw    %ax, %gs
1211
1212   # Turn on page size extension for 4Mbyte pages
1213   movl    %cr4, %eax
1214   orl     $(CR4_PSE), %eax
1215   movl    %eax, %cr4
1216   # Use enterpgdir as our initial page table
1217   movl    (start-12), %eax
1218   movl    %eax, %cr3
1219   # Turn on paging.
1220   movl    %cr0, %eax
1221   orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
1222   movl    %eax, %cr0
1223
1224   # Switch to the stack allocated by startothers()
1225   movl    (start-4), %esp
1226   # Call mpenter()
1227   call      *(start-8)
1228
1229   movw    $0x8a00, %ax
1230   movw    %ax, %dx
1231   outw    %ax, %dx
1232   movw    $0x8ae0, %ax
1233   outw    %ax, %dx
1234 spin:
1235   jmp     spin
1236
1237 .p2align 2
1238 gdt:
1239   SEG_NULLASM
1240   SEG_ASM(STA_X|STA_R, 0, 0xffffffff)
1241   SEG_ASM(STA_W, 0, 0xffffffff)
1242
1243
1244 gdtdesc:
1245   .word   (gdtdesc - gdt - 1)
1246   .long   gdt
1247
1248
1249
```



```
1250 #include "types.h"
1251 #include "defs.h"
1252 #include "param.h"
1253 #include "memlayout.h"
1254 #include "mmu.h"
1255 #include "proc.h"
1256 #include "x86.h"
1257
1258 static void startothers(void);
1259 static void mpmain(void)  __attribute__((noreturn));
1260 extern pde_t *kpgdir;
1261 extern char end[]; // first address after kernel loaded from ELF file
1262
1263 // Bootstrap processor starts running C code here.
1264 // Allocate a real stack and switch to it, first
1265 // doing some setup required for memory allocator to work.
1266 int
1267 main(void)
1268 {
1269   kinit1(end, P2V(4*1024*1024)); // phys page allocator
1270   kvmalloc();      // kernel page table
1271   mpinit();        // collect info about this machine
1272   lapicinit();
1273   seginit();       // set up segments
1274   cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
1275   picinit();       // interrupt controller
1276   ioapicinit();    // another interrupt controller
1277   consoleinit();   // I/O devices & their interrupts
1278   uartinit();      // serial port
1279   pinit();         // process table
1280   tvinit();        // trap vectors
1281   binit();         // buffer cache
1282   fileinit();      // file table
1283   ideinit();       // disk
1284   if(!ismp)
1285     timerinit();   // uniprocessor timer
1286   startothers();   // start other processors
1287   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1288   userinit();      // first user process
1289   // Finish setting up this processor in mpmain.
1290   mpmain();
1291 }
1292
1293
1294
1295
1296
1297
1298
1299
```

```
1300 // Other CPUs jump here from entryother.S.
1301 static void
1302 mpenter(void)
1303 {
1304   switchkvm();
1305   seginit();
1306   lapicinit();
1307   mpmain();
1308 }
1309
1310 // Common CPU setup code.
1311 static void
1312 mpmain(void)
1313 {
1314   cprintf("cpu%d: starting\n", cpu->id);
1315   idtinit();       // load idt register
1316   xchg(&cpu->started, 1); // tell startothers() we're up
1317   scheduler();     // start running processes
1318 }
1319
1320 pde_t entrypgdir[];  // For entry.S
1321
1322 // Start the non-boot (AP) processors.
1323 static void
1324 startothers(void)
1325 {
1326   extern uchar _binary_entryother_start[], _binary_entryother_size[];
1327   uchar *code;
1328   struct cpu *c;
1329   char *stack;
1330
1331   // Write entry code to unused memory at 0x7000.
1332   // The linker has placed the image of entryother.S in
1333   // _binary_entryother_start.
1334   code = p2v(0x7000);
1335   memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
1336
1337   for(c = cpus; c < cpus+ncpu; c++){
1338     if(c == cpus+cpunum())  // We've started already.
1339       continue;
1340
1341     // Tell entryother.S what stack to use, where to enter, and what
1342     // pgdir to use. We cannot use kpgdir yet, because the AP processor
1343     // is running in low  memory, so we use entrypgdir for the APs too.
1344     stack = kalloc();
1345     *(void**)(code-4) = stack + KSTACKSIZE;
1346     *(void**)(code-8) = mpenter;
1347     *(int**)(code-12) = (void *) v2p(entrypgdir);
1348
1349     lapicstartap(c->id, v2p(code));
```



```
1350     // wait for cpu to finish mpmain()
1351     while(c->started == 0)
1352       ;
1353   }
1354 }
1355
1356 // Boot page table used in entry.S and entryother.S.
1357 // Page directories (and page tables), must start on a page boundary,
1358 // hence the "__aligned__" attribute.
1359 // Use PTE_PS in page directory entry to enable 4Mbyte pages.
1360 __attribute__((__aligned__(PGSIZE)))
1361 pde_t entrypgdir[NPDENTRIES] = {
1362   // Map VA's [0, 4MB) to PA's [0, 4MB)
1363   [0] = (0) | PTE_P | PTE_W | PTE_PS,
1364   // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1365   [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1366 };
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
```

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



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

```
1500 // Blank page.
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
```



```
1550 // Mutual exclusion lock.
1551 struct spinlock {
1552   uint locked;       // Is the lock held?
1553
1554   // For debugging:
1555   char *name;        // Name of lock.
1556   struct cpu *cpu;   // The cpu holding the lock.
1557   uint pcs[10];      // The call stack (an array of program counters)
1558                      // that locked the lock.
1559 };
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
```

```
1600 // Mutual exclusion spin locks.
1601
1602 #include "types.h"
1603 #include "defs.h"
1604 #include "param.h"
1605 #include "x86.h"
1606 #include "memlayout.h"
1607 #include "mmu.h"
1608 #include "proc.h"
1609 #include "spinlock.h"
1610
1611 void
1612 initlock(struct spinlock *lk, char *name)
1613 {
1614   lk->name = name;
1615   lk->locked = 0;
1616   lk->cpu = 0;
1617 }
1618
1619 // Acquire the lock.
1620 // Loops (spins) until the lock is acquired.
1621 // Holding a lock for a long time may cause
1622 // other CPUs to waste time spinning to acquire it.
1623 void
1624 acquire(struct spinlock *lk)
1625 {
1626   pushcli(); // disable interrupts to avoid deadlock.
1627   if(holding(lk))
1628     panic("acquire");
1629
1630   // The xchg is atomic.
1631   // It also serializes, so that reads after acquire are not
1632   // reordered before it.
1633   while(xchg(&lk->locked, 1) != 0)
1634     ;
1635
1636   // Record info about lock acquisition for debugging.
1637   lk->cpu = cpu;
1638   getcallerpcs(&lk, lk->pcs);
1639 }
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
```



```
1650 // Release the lock.
1651 void
1652 release(struct spinlock *lk)
1653 {
1654   if(!holding(lk))
1655     panic("release");
1656
1657   lk->pcs[0] = 0;
1658   lk->cpu = 0;
1659
1660   // The xchg serializes, so that reads before release are
1661   // not reordered after it.  The 1996 PentiumPro manual (Volume 3,
1662   // 7.2) says reads can be carried out speculatively and in
1663   // any order, which implies we need to serialize here.
1664   // But the 2007 Intel 64 Architecture Memory Ordering White
1665   // Paper says that Intel 64 and IA-32 will not move a load
1666   // after a store. So lock->locked = 0 would work here.
1667   // The xchg being asm volatile ensures gcc emits it after
1668   // the above assignments (and after the critical section).
1669   xchg(&lk->locked, 0);
1670
1671   popcli();
1672 }
1673
1674 // Record the current call stack in pcs[] by following the %ebp chain.
1675 void
1676 getcallerpcs(void *v, uint pcs[])
1677 {
1678   uint *ebp;
1679   int i;
1680
1681   ebp = (uint*)v - 2;
1682   for(i = 0; i < 10; i++){
1683     if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
1684       break;
1685     pcs[i] = ebp[1];     // saved %eip
1686     ebp = (uint*)ebp[0]; // saved %ebp
1687   }
1688   for(; i < 10; i++)
1689     pcs[i] = 0;
1690 }
1691
1692 // Check whether this cpu is holding the lock.
1693 int
1694 holding(struct spinlock *lock)
1695 {
1696   return lock->locked && lock->cpu == cpu;
1697 }
1698
1699
```

```
1700 // Pushcli/popcli are like cli/sti except that they are matched:
1701 // it takes two popcli to undo two pushcli.  Also, if interrupts
1702 // are off, then pushcli, popcli leaves them off.
1703
1704 void
1705 pushcli(void)
1706 {
1707   int eflags;
1708
1709   eflags = readeflags();
1710   cli();
1711   if(cpu->ncli++ == 0)
1712     cpu->intena = eflags & FL_IF;
1713 }
1714
1715 void
1716 popcli(void)
1717 {
1718   if(readeflags()&FL_IF)
1719     panic("popcli - interruptible");
1720   if(--cpu->ncli < 0)
1721     panic("popcli");
1722   if(cpu->ncli == 0 && cpu->intena)
1723     sti();
1724 }
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
```



```
1750 #include "param.h"
1751 #include "types.h"
1752 #include "defs.h"
1753 #include "x86.h"
1754 #include "memlayout.h"
1755 #include "mmu.h"
1756 #include "proc.h"
1757 #include "elf.h"
1758
1759 extern char data[];  // defined by kernel.ld
1760 pde_t *kpgdir;  // for use in scheduler()
1761 struct segdesc gdt[NSEGS];
1762
1763 // Set up CPU's kernel segment descriptors.
1764 // Run once on entry on each CPU.
1765 void
1766 seginit(void)
1767 {
1768   struct cpu *c;
1769
1770   // Map "logical" addresses to virtual addresses using identity map.
1771   // Cannot share a CODE descriptor for both kernel and user
1772   // because it would have to have DPL_USR, but the CPU forbids
1773   // an interrupt from CPL=0 to DPL=3.
1774   c = &cpus[cpunum()];
1775   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
1776   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1777   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
1778   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1779
1780   // Map cpu, and curproc
1781   c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
1782
1783   lgdt(c->gdt, sizeof(c->gdt));
1784   loadgs(SEG_KCPU << 3);
1785
1786   // Initialize cpu-local storage.
1787   cpu = c;
1788   proc = 0;
1789 }
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
```

```
1800 // Return the address of the PTE in page table pgdir
1801 // that corresponds to virtual address va.  If alloc!=0,
1802 // create any required page table pages.
1803 static pte_t *
1804 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1805 {
1806   pde_t *pde;
1807   pte_t *pgtab;
1808
1809   pde = &pgdir[PDX(va)];
1810   if(*pde & PTE_P){
1811     pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
1812   } else {
1813     if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1814       return 0;
1815     // Make sure all those PTE_P bits are zero.
1816     memset(pgtab, 0, PGSIZE);
1817     // The permissions here are overly generous, but they can
1818     // be further restricted by the permissions in the page table
1819     // entries, if necessary.
1820     *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
1821   }
1822   return &pgtab[PTX(va)];
1823 }
1824
1825 // Create PTEs for virtual addresses starting at va that refer to
1826 // physical addresses starting at pa. va and size might not
1827 // be page-aligned.
1828 static int
1829 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1830 {
1831   char *a, *last;
1832   pte_t *pte;
1833
1834   a = (char*)PGROUNDDOWN((uint)va);
1835   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1836   for(;;){
1837     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1838       return -1;
1839     if(*pte & PTE_P)
1840       panic("remap");
1841     *pte = pa | perm | PTE_P;
1842     if(a == last)
1843       break;
1844     a += PGSIZE;
1845     pa += PGSIZE;
1846   }
1847   return 0;
1848 }
1849
```



```
1850 // There is one page table per process, plus one that's used when
1851 // a CPU is not running any process (kpgdir). The kernel uses the
1852 // current process's page table during system calls and interrupts;
1853 // page protection bits prevent user code from using the kernel's
1854 // mappings.
1855 //
1856 // setupkvm() and exec() set up every page table like this:
1857 //
1858 //   0..KERNBASE: user memory (text+data+stack+heap), mapped to
1859 //                phys memory allocated by the kernel
1860 //   KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
1861 //   KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
1862 //                for the kernel's instructions and r/o data
1863 //   data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
1864 //                                     rw data + free physical memory
1865 //   0xfe000000..0: mapped direct (devices such as ioapic)
1866 //
1867 // The kernel allocates physical memory for its heap and for user memory
1868 // between V2P(end) and the end of physical memory (PHYSTOP)
1869 // (directly addressable from end..P2V(PHYSTOP)).
1870
1871 // This table defines the kernel's mappings, which are present in
1872 // every process's page table.
1873 static struct kmap {
1874   void *virt;
1875   uint phys_start;
1876   uint phys_end;
1877   int perm;
1878 } kmap[] = {
1879  { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
1880  { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
1881  { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
1882  { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
1883 };
1884
1885 // Set up kernel part of a page table.
1886 pde_t*
1887 setupkvm(void)
1888 {
1889   pde_t *pgdir;
1890   struct kmap *k;
1891
1892   if((pgdir = (pde_t*)kalloc()) == 0)
1893     return 0;
1894   memset(pgdir, 0, PGSIZE);
1895   if (p2v(PHYSTOP) > (void*)DEVSPACE)
1896     panic("PHYSTOP too high");
1897   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1898     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1899                 (uint)k->phys_start, k->perm) < 0)
```

```
1900       return 0;
1901   return pgdir;
1902 }
1903
1904 // Allocate one page table for the machine for the kernel address
1905 // space for scheduler processes.
1906 void
1907 kvmalloc(void)
1908 {
1909   kpgdir = setupkvm();
1910   switchkvm();
1911 }
1912
1913 // Switch h/w page table register to the kernel-only page table,
1914 // for when no process is running.
1915 void
1916 switchkvm(void)
1917 {
1918   lcr3(v2p(kpgdir));   // switch to the kernel page table
1919 }
1920
1921 // Switch TSS and h/w page table to correspond to process p.
1922 void
1923 switchuvm(struct proc *p)
1924 {
1925   pushcli();
1926   cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
1927   cpu->gdt[SEG_TSS].s = 0;
1928   cpu->ts.ss0 = SEG_KDATA << 3;
1929   cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
1930   ltr(SEG_TSS << 3);
1931   if(p->pgdir == 0)
1932     panic("switchuvm: no pgdir");
1933   lcr3(v2p(p->pgdir));  // switch to new address space
1934   popcli();
1935 }
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
```



```
1950 // Load the initcode into address 0 of pgdir.
1951 // sz must be less than a page.
1952 void
1953 inituvm(pde_t *pgdir, char *init, uint sz)
1954 {
1955   char *mem;
1956
1957   if(sz >= PGSIZE)
1958     panic("inituvm: more than a page");
1959   mem = kalloc();
1960   memset(mem, 0, PGSIZE);
1961   mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
1962   memmove(mem, init, sz);
1963 }
1964
1965 // Load a program segment into pgdir.  addr must be page-aligned
1966 // and the pages from addr to addr+sz must already be mapped.
1967 int
1968 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
1969 {
1970   uint i, pa, n;
1971   pte_t *pte;
1972
1973   if((uint) addr % PGSIZE != 0)
1974     panic("loaduvm: addr must be page aligned");
1975   for(i = 0; i < sz; i += PGSIZE){
1976     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1977       panic("loaduvm: address should exist");
1978     pa = PTE_ADDR(*pte);
1979     if(sz - i < PGSIZE)
1980       n = sz - i;
1981     else
1982       n = PGSIZE;
1983     if(readi(ip, p2v(pa), offset+i, n) != n)
1984       return -1;
1985   }
1986   return 0;
1987 }
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
```

```
2000 // Allocate page tables and physical memory to grow process from oldsz to
2001 // newsz, which need not be page aligned.  Returns new size or 0 on error.
2002 int
2003 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
2004 {
2005   char *mem;
2006   uint a;
2007
2008   if(newsz >= KERNBASE)
2009     return 0;
2010   if(newsz < oldsz)
2011     return oldsz;
2012
2013   a = PGROUNDUP(oldsz);
2014   for(; a < newsz; a += PGSIZE){
2015     mem = kalloc();
2016     if(mem == 0){
2017       cprintf("allocuvm out of memory\n");
2018       deallocuvm(pgdir, newsz, oldsz);
2019       return 0;
2020     }
2021     memset(mem, 0, PGSIZE);
2022     mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U);
2023   }
2024   return newsz;
2025 }
2026
2027 // Deallocate user pages to bring the process size from oldsz to
2028 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
2029 // need to be less than oldsz.  oldsz can be larger than the actual
2030 // process size.  Returns the new process size.
2031 int
2032 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
2033 {
2034   pte_t *pte;
2035   uint a, pa;
2036
2037   if(newsz >= oldsz)
2038     return oldsz;
2039
2040   a = PGROUNDUP(newsz);
2041   for(; a  < oldsz; a += PGSIZE){
2042     pte = walkpgdir(pgdir, (char*)a, 0);
2043     if(!pte)
2044       a += (NPTENTRIES - 1) * PGSIZE;
2045     else if((*pte & PTE_P) != 0){
2046       pa = PTE_ADDR(*pte);
2047       if(pa == 0)
2048         panic("kfree");
2049       char *v = p2v(pa);
```



```
2050       kfree(v);
2051       *pte = 0;
2052     }
2053   }
2054   return newsz;
2055 }
2056
2057 // Free a page table and all the physical memory pages
2058 // in the user part.
2059 void
2060 freevm(pde_t *pgdir)
2061 {
2062   uint i;
2063
2064   if(pgdir == 0)
2065     panic("freevm: no pgdir");
2066   deallocuvm(pgdir, KERNBASE, 0);
2067   for(i = 0; i < NPDENTRIES; i++){
2068     if(pgdir[i] & PTE_P){
2069       char * v = p2v(PTE_ADDR(pgdir[i]));
2070       kfree(v);
2071     }
2072   }
2073   kfree((char*)pgdir);
2074 }
2075
2076 // Clear PTE_U on a page. Used to create an inaccessible
2077 // page beneath the user stack.
2078 void
2079 clearpteu(pde_t *pgdir, char *uva)
2080 {
2081   pte_t *pte;
2082
2083   pte = walkpgdir(pgdir, uva, 0);
2084   if(pte == 0)
2085     panic("clearpteu");
2086   *pte &= ~PTE_U;
2087 }
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
```

```
2100 // Given a parent process's page table, create a copy
2101 // of it for a child.
2102 pde_t*
2103 copyuvm(pde_t *pgdir, uint sz)
2104 {
2105   pde_t *d;
2106   pte_t *pte;
2107   uint pa, i, flags;
2108   char *mem;
2109
2110   if((d = setupkvm()) == 0)
2111     return 0;
2112   for(i = 0; i < sz; i += PGSIZE){
2113     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
2114       panic("copyuvm: pte should exist");
2115     if(!(*pte & PTE_P))
2116       panic("copyuvm: page not present");
2117     pa = PTE_ADDR(*pte);
2118     flags = PTE_FLAGS(*pte);
2119     if((mem = kalloc()) == 0)
2120       goto bad;
2121     memmove(mem, (char*)p2v(pa), PGSIZE);
2122     if(mappages(d, (void*)i, PGSIZE, v2p(mem), flags) < 0)
2123       goto bad;
2124   }
2125   return d;
2126
2127 bad:
2128   freevm(d);
2129   return 0;
2130 }
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
```



```
2150 // Map user virtual address to kernel address.
2151 char*
2152 uva2ka(pde_t *pgdir, char *uva)
2153 {
2154   pte_t *pte;
2155
2156   pte = walkpgdir(pgdir, uva, 0);
2157   if((*pte & PTE_P) == 0)
2158     return 0;
2159   if((*pte & PTE_U) == 0)
2160     return 0;
2161   return (char*)p2v(PTE_ADDR(*pte));
2162 }
2163
2164 // Copy len bytes from p to user address va in page table pgdir.
2165 // Most useful when pgdir is not the current page table.
2166 // uva2ka ensures this only works for PTE_U pages.
2167 int
2168 copyout(pde_t *pgdir, uint va, void *p, uint len)
2169 {
2170   char *buf, *pa0;
2171   uint n, va0;
2172
2173   buf = (char*)p;
2174   while(len > 0){
2175     va0 = (uint)PGROUNDDOWN(va);
2176     pa0 = uva2ka(pgdir, (char*)va0);
2177     if(pa0 == 0)
2178       return -1;
2179     n = PGSIZE - (va - va0);
2180     if(n > len)
2181       n = len;
2182     memmove(pa0 + (va - va0), buf, n);
2183     len -= n;
2184     buf += n;
2185     va = va0 + PGSIZE;
2186   }
2187   return 0;
2188 }
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
```

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



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

```
2300 // Blank page.
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
```



```
2350 // Segments in proc->gdt.
2351 #define NSEGS     7
2352
2353 // Per-CPU state
2354 struct cpu {
2355   uchar id;                    // Local APIC ID; index into cpus[] below
2356   struct context *scheduler;   // swtch() here to enter scheduler
2357   struct taskstate ts;         // Used by x86 to find stack for interrupt
2358   struct segdesc gdt[NSEGS];   // x86 global descriptor table
2359   volatile uint started;       // Has the CPU started?
2360   int ncli;                    // Depth of pushcli nesting.
2361   int intena;                  // Were interrupts enabled before pushcli?
2362
2363   // Cpu-local storage variables; see below
2364   struct cpu *cpu;
2365   struct proc *proc;           // The currently-running process.
2366 };
2367
2368 extern struct cpu cpus[NCPU];
2369 extern int ncpu;
2370
2371 // Per-CPU variables, holding pointers to the
2372 // current cpu and to the current process.
2373 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2374 // and "%gs:4" to refer to proc.  seginit sets up the
2375 // %gs segment register so that %gs refers to the memory
2376 // holding those two variables in the local cpu's struct cpu.
2377 // This is similar to how thread-local variables are implemented
2378 // in thread libraries such as Linux pthreads.
2379 extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
2380 extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc
2381
2382
2383 // Saved registers for kernel context switches.
2384 // Don't need to save all the segment registers (%cs, etc),
2385 // because they are constant across kernel contexts.
2386 // Don't need to save %eax, %ecx, %edx, because the
2387 // x86 convention is that the caller has saved them.
2388 // Contexts are stored at the bottom of the stack they
2389 // describe; the stack pointer is the address of the context.
2390 // The layout of the context matches the layout of the stack in swtch.S
2391 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2392 // but it is on the stack and allocproc() manipulates it.
2393 struct context {
2394   uint edi;
2395   uint esi;
2396   uint ebx;
2397   uint ebp;
2398   uint eip;
2399 };
```

```
2400 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2401
2402 // Per-process state
2403 struct proc {
2404   uint sz;                     // Size of process memory (bytes)
2405   pde_t* pgdir;                // Page table
2406   char *kstack;                // Bottom of kernel stack for this process
2407   enum procstate state;        // Process state
2408   int pid;                     // Process ID
2409   struct proc *parent;         // Parent process
2410   struct trapframe *tf;        // Trap frame for current syscall
2411   struct context *context;     // swtch() here to run process
2412   void *chan;                  // If non-zero, sleeping on chan
2413   int killed;                  // If non-zero, have been killed
2414   struct file *ofile[NOFILE];  // Open files
2415   struct inode *cwd;           // Current directory
2416   char name[16];               // Process name (debugging)
2417   int uid;                     // User ID  */
2418   int gid;                /* Group ID */
2419   int ppid;                       /* Parent ID*/
2420   int priority;                   /*0-2*/
2421   struct proc * next;
2422 };
2423
2424 // Process memory is laid out contiguously, low addresses first:
2425 //   text
2426 //   original data and bss
2427 //   fixed-size stack
2428 //   expandable heap
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
```



```
2450 #include "types.h"
2451 #include "defs.h"
2452 #include "param.h"
2453 #include "memlayout.h"
2454 #include "mmu.h"
2455 #include "x86.h"
2456 #include "proc.h"
2457 #include "spinlock.h"
2458 #include "ps.h"
2459
2460 struct {
2461   struct spinlock lock;
2462   struct proc proc[NPROC];
2463   struct proc *pReadyList[3];/*when !p->, the end has been hit*/
2464   struct proc *pFreeList;//three priority levels
2465   uint TimeToReset;
2466 } ptable;
2467
2468 static struct proc *initproc;
2469
2470 int nextpid = 1;
2471 extern void forkret(void);
2472 extern void trapret(void);
2473
2474 static void wakeup1(void *chan);
2475
2476 void
2477 pinit(void)
2478 {
2479   initlock(&ptable.lock, "ptable");
2480 }
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
```

```
2500 // Look in the process table for an UNUSED proc.
2501 // If found, change state to EMBRYO and initialize
2502 // state required to run in the kernel.
2503 // Otherwise return 0.
2504 void
2505 printlists(void)
2506 {
2507   struct proc * p;
2508   struct proc * tmp;
2509   int i,j;
2510   //show ready list
2511   for(i=0;i<3;++i){
2512      p = ptable.pReadyList[i];
2513      j=0;
2514      tmp = p;
2515      if(!tmp){
2516        //cprintf("list %d empty\n",i);
2517        continue;
2518      }
2519      while (tmp){
2520        cprintf("ready list #%d item: %d: ", i,j);
2521        cprintf("%d %d %d %s\n", tmp->pid, tmp->gid, tmp->uid,  tmp->name);
2522        tmp = tmp->next;/*last item should point to 0*/
2523        ++j;
2524      }
2525   }
2526 }
2527
2528
2529 int
2530 addtoready(struct proc *toadd) //add to end of ready list
2531 {
2532   if(!toadd)
2533     panic("tried to add nothing");
2534
2535   int val = toadd->priority;
2536   struct proc *p;
2537   p= ptable.pReadyList[val];
2538   if (!p){/*empty list*/
2539     toadd->next = 0; //just in case
2540     ptable.pReadyList[val] = toadd;
2541     return 0;
2542   }
2543   while(p->next){//get to end of list
2544      p = p->next;
2545   }
2546   toadd->next = 0; //just in case
2547   p->next = toadd;
2548   return 0;
2549 }
```



```
2550 int
2551 removefromready(struct proc *torm)//remove item from ready list
2552 {
2553   struct proc* tmp;
2554   struct proc* prev;
2555   int val = torm->priority;
2556   if(!torm)
2557     panic("tried to remove nothing");
2558   tmp = ptable.pReadyList[val];
2559   if(tmp->pid == torm->pid){//first item
2560    ptable.pReadyList[val] = ptable.pReadyList[val]->next;
2561    torm->next = 0;
2562    return 0;
2563   }
2564   else{
2565     while(tmp){
2566       prev = tmp;
2567       tmp = tmp->next; //traverse
2568       if(tmp->pid == torm->pid){
2569         prev->next = torm->next;
2570         torm->next = 0;
2571         return 0;
2572       }
2573     }
2574   }
2575   cprintf("didnt actually remove anything\n");
2576   return 0;
2577 }
2578 void
2579 addtofree(struct proc *toadd)//add to front of list
2580 {
2581   toadd->next = ptable.pFreeList;
2582   ptable.pFreeList = toadd;
2583 }
2584
2585 static struct proc*
2586 allocproc(void)
2587 {
2588   struct proc *p;
2589   char *sp;
2590
2591   acquire(&ptable.lock);
2592   p = ptable.pFreeList;
2593   if(!p){ /*list empty*/
2594    release(&ptable.lock);
2595    //panic("free list empty");
2596    return 0;
2597   }
2598   ptable.pFreeList = ptable.pFreeList->next; /*so it doesnt point at p*/
2599 /*//how it was originally implemented
```

```
2600   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2601   if(p->state == UNUSED)
2602      goto found;
2603   release(&ptable.lock);
2604   return 0;
2605
2606 found:
2607 */
2608   release(&ptable.lock);
2609   p->state = EMBRYO;
2610   p->pid = nextpid++;
2611   p->gid =0; /*place holder*/
2612   p->uid =0; /*place holder*/
2613   p->ppid=0;
2614   p->next=0;
2615   p->priority=1;
2616
2617   // Allocate kernel stack.
2618   if((p->kstack = kalloc()) == 0){
2619     p->state = UNUSED;
2620     acquire(&ptable.lock);
2621     addtofree(p);
2622     release(&ptable.lock);
2623     return 0;
2624   }
2625   sp = p->kstack + KSTACKSIZE;
2626
2627   // Leave room for trap frame.
2628   sp -= sizeof *p->tf;
2629   p->tf = (struct trapframe*)sp;
2630
2631   // Set up new context to start executing at forkret,
2632   // which returns to trapret.
2633   sp -= 4;
2634   *(uint*)sp = (uint)trapret;
2635
2636   sp -= sizeof *p->context;
2637   p->context = (struct context*)sp;
2638   memset(p->context, 0, sizeof *p->context);
2639   p->context->eip = (uint)forkret;
2640   p->next = 0; /*state = embryo, not ready for ready list*/
2641   return p;
2642 }
2643
2644
2645
2646
2647
2648
2649
```



```
2650 // Set up first user process.
2651 void
2652 userinit(void)
2653 {
2654   struct proc *p;
2655   //struct proc *tmp;
2656   int i;
2657   extern char _binary_initcode_start[], _binary_initcode_size[];
2658
2659    /*set up free list*/
2660    ptable.pFreeList=0; /*initialize*/
2661    for(i = 0; i < NPROC; i++){
2662      p = &ptable.proc[i];
2663      addtofree(p);
2664   }
2665   /*get the first process started*/
2666   p = allocproc();
2667   initproc = p;
2668   if((p->pgdir = setupkvm()) == 0)
2669     panic("userinit: out of memory?");
2670   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2671   p->sz = PGSIZE;
2672   memset(p->tf, 0, sizeof(*p->tf));
2673   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2674   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2675   p->tf->es = p->tf->ds;
2676   p->tf->ss = p->tf->ds;
2677   p->tf->eflags = FL_IF;
2678   p->tf->esp = PGSIZE;
2679   p->tf->eip = 0;  // beginning of initcode.S
2680   safestrcpy(p->name, "initcode", sizeof(p->name));
2681   p->cwd = namei("/");
2682
2683   //initialize ready lists
2684   ptable.pReadyList[0]=0;
2685   ptable.pReadyList[1]=0;
2686   ptable.pReadyList[2]=0;
2687   //away we go!
2688   p->state = RUNNABLE;
2689   p->priority = 1;
2690   addtoready(p);//should stick on Ready[1]
2691 }
2692
2693
2694
2695
2696
2697
2698
2699
```

```
2700 // Grow current process's memory by n bytes.
2701 // Return 0 on success, -1 on failure.
2702 int
2703 growproc(int n)
2704 {
2705   uint sz;
2706
2707   sz = proc->sz;
2708   if(n > 0){
2709     if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2710       return -1;
2711   } else if(n < 0){
2712     if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2713       return -1;
2714   }
2715   proc->sz = sz;
2716   switchuvm(proc);
2717   return 0;
2718 }
2719 // Create a new process copying p as the parent.
2720 // Sets up stack to return as if from system call.
2721 // Caller must set state of returned proc to RUNNABLE.
2722 int
2723 fork(void)
2724 {
2725   int i, pid;
2726   struct proc *np;
2727   // Allocate process.
2728   if((np = allocproc()) == 0)
2729     return -1;
2730
2731   // Copy process state from p.
2732   if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2733     kfree(np->kstack);
2734     np->kstack = 0;
2735     np->state = UNUSED;
2736     acquire(&ptable.lock);
2737     addtofree(np);
2738     release(&ptable.lock);
2739     return -1;
2740   }
2741   np->sz = proc->sz;
2742   np->parent = proc;
2743   *np->tf = *proc->tf;
2744   np->gid = proc->gid;/*inherit parent GID*/
2745   np->uid = proc->uid;/*inherit parent UID*/
2746   np->ppid = proc->pid;
2747   // Clear %eax so that fork returns 0 in the child.
2748   np->tf->eax = 0;
2749
```



```
2750   for(i = 0; i < NOFILE; i++)
2751     if(proc->ofile[i])
2752       np->ofile[i] = filedup(proc->ofile[i]);
2753   np->cwd = idup(proc->cwd);
2754
2755   safestrcpy(np->name, proc->name, sizeof(proc->name));
2756
2757   pid = np->pid;
2758
2759   // lock to force the compiler to emit the np->state write last.
2760   acquire(&ptable.lock);
2761   np->state = RUNNABLE;
2762   addtoready(np);
2763   release(&ptable.lock);
2764   return pid;
2765 }
2766
2767 // Exit the current process.  Does not return.
2768 // An exited process remains in the zombie state
2769 // until its parent calls wait() to find out it exited.
2770 void
2771 exit(void)
2772 {
2773   struct proc *p;
2774   int fd;
2775
2776   if(proc == initproc)
2777     panic("init exiting");
2778
2779   // Close all open files.
2780   for(fd = 0; fd < NOFILE; fd++){
2781     if(proc->ofile[fd]){
2782       fileclose(proc->ofile[fd]);
2783       proc->ofile[fd] = 0;
2784     }
2785   }
2786
2787   begin_op();
2788   iput(proc->cwd);
2789   end_op();
2790   proc->cwd = 0;
2791
2792   acquire(&ptable.lock);
2793
2794   // Parent might be sleeping in wait().
2795   wakeup1(proc->parent);
2796
2797
2798
2799
```

```
2800   // Pass abandoned children to init.
2801   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2802     if(p->parent == proc){
2803       p->parent = initproc;
2804       if(p->state == ZOMBIE)
2805         wakeup1(initproc);
2806     }
2807   }
2808   //release(&ptable.lock);
2809   // Jump into the scheduler, never to return.
2810   proc->state = ZOMBIE;
2811   p->next = 0;
2812   sched();
2813   panic("zombie exit");
2814 }
2815
2816 // Wait for a child process to exit and return its pid.
2817 // Return -1 if this process has no children.
2818 int
2819 wait(void)
2820 {
2821   struct proc *p;
2822   int havekids, pid;
2823   acquire(&ptable.lock);
2824   for(;;){
2825     // Scan through table looking for zombie children.
2826     havekids = 0;
2827     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2828       if(p->parent != proc)
2829         continue;
2830       havekids = 1;
2831       if(p->state == ZOMBIE){
2832         // Found one.
2833    /* reset the data*/
2834         pid = p->pid;
2835         kfree(p->kstack);
2836         p->kstack = 0;
2837         freevm(p->pgdir);
2838         p->state = UNUSED;
2839            p->pid = 0;
2840    p->uid = 0;
2841    p->gid = 0;
2842    p->ppid = 0;
2843         p->parent = 0;
2844         p->name[0] = 0;
2845         p->killed = 0;
2846    p->next = 0;
2847    p->priority=0;
2848         addtofree(p);
2849         release(&ptable.lock);
```



```
2850         return pid;
2851       }
2852     }
2853
2854     // No point waiting if we don't have any children.
2855     if(!havekids || proc->killed){
2856       release(&ptable.lock);
2857       return -1;
2858     }
2859
2860     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2861     sleep(proc, &ptable.lock);
2862   }
2863 }
2864 int//called by things that have locks
2865 dspri2(int pid, int priority)
2866 {
2867  struct proc * p;
2868  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2869    if(p->pid == pid){
2870      if(p->state == RUNNING){
2871       p->priority = priority;
2872       return 0;
2873      }
2874      removefromready(p);/*remove from the list it was in*/
2875      p->priority = priority;
2876      addtoready(p);/*stick in different list*/
2877      return 0;
2878    }
2879  }
2880  cprintf("no dice, baby\n");
2881  return 1;
2882 }
2883
2884 int//called by stuff that does not have locks
2885 dspri(int pid, int priority)
2886 {
2887  int ret;
2888  if(priority < 0 || priority > 2)//invalid priority number
2889    return 1;
2890  acquire(&ptable.lock);
2891  //printlists();
2892  ret = dspri2(pid, priority);
2893  //printlists();
2894  release(&ptable.lock);
2895  return ret; /*invalid PID*/
2896 }
2897
2898
2899
```

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.

void
resetpriority(void)
{
  int toset=1;
  struct proc *tmp;
  //toset=1;
  acquire(&ptable.lock);
  ptable.TimeToReset = MaxTimeToReset;
  for(tmp = ptable.proc; tmp < &ptable.proc[NPROC];tmp++){
   if (tmp->state == UNUSED)//dont change unused processes
    continue;
   if(tmp->priority == toset)//dont change processes of the right priority
    continue;
   removefromready(tmp);//remove from list if on list
   tmp->priority = toset;
   addtoready(tmp);

  }
 /* for(i = 0; i<3; i+=2){ //i should go from r[0] to r[2]
    tmp = ptable.pReadyList[i];//no need to change r[1]
    while(tmp){
      ptable.pReadyList=tmp->next; // dspri will chng wher tmp pt
      removefromready(tmp)
      tmp->priority = toset;
      addtoready(tmp)
      tmp=ptable.pReadyList[i];//move to next item in ready list
    }//tmp = 0 at end of loop
  }*/
 release(&ptable.lock);
}
```

Sheet 29


```
void
scheduler(void)
{
  struct proc *p=0;
  ptable.TimeToReset=MaxTimeToReset;//initialize
  proc = 0;
  //sti();
  for(;;){
    // Enable interrupts on this processor
    sti();
    //manage priority teset
    acquire(&ptable.lock);
    --ptable.TimeToReset;
    release(&ptable.lock);
    if(!ptable.TimeToReset)/*misleading, it is time to reset*/
      resetpriority();
    acquire(&ptable.lock);
    //find a process to run
    //lowest priority first
    if(ptable.pReadyList[2]){
      //cprintf("item on ready2\n");
      p=ptable.pReadyList[2];
    }
    //default priority
    if(ptable.pReadyList[1]){
      //cprintf("item on ready1\n");
      p=ptable.pReadyList[1];
    }
    //highest priority
    if(ptable.pReadyList[0]){
      //cprintf("item on ready0\n");
      p=ptable.pReadyList[0];
    }
    if(!p){//lists are all empty
      release(&ptable.lock);
      continue;
    }
    if(p->state != RUNNABLE){
      panic("oops");
        //see if process is any of the state we would expect
       /* //debug stuff
        if(p->state == SLEEPING)
     panic("sleeping ");
        if(p->state == ZOMBIE)
          panic("zombie ");
        if(p->state == UNUSED);
          panic("unused ");
        if(p->state == EMBRYO)
          panic("embryo ");
        if(p->state == RUNNING)
```

Sheet 29

```
3000           panic("already running ");
3001         */
3002       }
3003     // Switch to chosen process.  It is the process's job
3004     // to release ptable.lock and then reacquire it
3005     // before jumping back to us.
3006     //printlists();
3007     proc = p;
3008     //cprintf("priority of pid %d is %d\n", p->pid, p->priority);
3009     switchuvm(p);
3010     removefromready(p);
3011     p->state = RUNNING;
3012     p->next = 0;
3013     swtch(&cpu->scheduler, proc->context);
3014     switchkvm();
3015     //cprintf("priority of pid %d is %d\n", p->pid, p->priority);
3016     // Process is done running for now.
3017     // It should have changed its p->state before coming back.
3018     p=0;
3019     proc = 0;//the old scheduelr did this, kept getting panics without it
3020     release(&ptable.lock);
3021     //proc = 0;
3022   }
3023 }
3024
3025 // Enter scheduler.  Must hold only ptable.lock
3026 // and have changed proc->state.
3027 void
3028 sched(void)
3029 {
3030   int intena;
3031
3032   if(!holding(&ptable.lock))
3033     panic("sched ptable.lock");
3034   if(cpu->ncli != 1)
3035     panic("sched locks");
3036   if(proc->state == RUNNING)
3037     panic("sched running");
3038   if(readeflags()&FL_IF)
3039     panic("sched interruptible");
3040   intena = cpu->intena;
3041   swtch(&proc->context, cpu->scheduler);
3042   cpu->intena = intena;
3043 }
3044
3045
3046
3047
3048
3049
```



```
3050 // Give up the CPU for one scheduling round.
3051 void
3052 yield(void)
3053 {
3054   acquire(&ptable.lock);
3055   proc->state = RUNNABLE;
3056   addtoready(proc);
3057   sched();
3058   release(&ptable.lock);
3059 }
3060
3061 // A fork child's very first scheduling by scheduler()
3062 // will swtch here.  "Return" to user space.
3063 void
3064 forkret(void)
3065 {
3066   static int first = 1;
3067   // Still holding ptable.lock from scheduler.
3068   release(&ptable.lock);
3069
3070   if (first) {
3071     // Some initialization functions must be run in the context
3072     // of a regular process (e.g., they call sleep), and thus cannot
3073     // be run from main().
3074     first = 0;
3075     iinit(ROOTDEV);
3076     initlog(ROOTDEV);
3077   }
3078
3079   // Return to "caller", actually trapret (see allocproc).
3080 }
3081
3082 // Atomically release lock and sleep on chan.
3083 // Reacquires lock when awakened.
3084 void
3085 sleep(void *chan, struct spinlock *lk)
3086 {
3087   if(proc == 0)
3088     panic("sleep");
3089
3090   if(lk == 0)
3091     panic("sleep without lk");
3092
3093   // Must acquire ptable.lock in order to
3094   // change p->state and then call sched.
3095   // Once we hold ptable.lock, we can be
3096   // guaranteed that we won't miss any wakeup
3097   // (wakeup runs with ptable.lock locked),
3098   // so it's okay to release lk.
3099   if(lk != &ptable.lock){
```

```
3100     acquire(&ptable.lock);
3101     release(lk);
3102   }
3103
3104   // Go to sleep.
3105   proc->chan = chan;
3106   proc->state = SLEEPING;
3107   proc->next = 0;
3108   //remove from ready list;
3109   sched();
3110   // Tidy up.
3111   proc->chan = 0;
3112
3113   // Reacquire original lock.
3114   if(lk != &ptable.lock){
3115     release(&ptable.lock);
3116     acquire(lk);
3117   }
3118 }
3119
3120
3121
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 // Wake up all processes sleeping on chan.
3151 // The ptable lock must be held.
3152 static void
3153 wakeup1(void *chan)
3154 {
3155   struct proc *p;
3156
3157   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
3158     if(p->state == SLEEPING && p->chan == chan){
3159       p->state = RUNNABLE;
3160       addtoready(p);
3161     }
3162 }
3163
3164 // Wake up all processes sleeping on chan.
3165 void
3166 wakeup(void *chan)
3167 {
3168   acquire(&ptable.lock);
3169   wakeup1(chan);
3170   release(&ptable.lock);
3171 }
3172
3173 // Kill the process with the given pid.
3174 // Process won't exit until it returns
3175 // to user space (see trap in trap.c).
3176 int
3177 kill(int pid)
3178 {
3179   struct proc *p;
3180
3181   acquire(&ptable.lock);
3182   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3183     if(p->pid == pid){
3184       p->killed = 1;
3185       // Wake process from sleep if necessary.
3186       if(p->state == SLEEPING){
3187         p->state = RUNNABLE;
3188         addtoready(p);
3189       }
3190       release(&ptable.lock);
3191       return 0;
3192     }
3193   }
3194   release(&ptable.lock);
3195   return -1;
3196 }
3197
3198
3199
```

```
3200 // Print a process listing to console.  For debugging.
3201 // Runs when user types ^P on console.
3202 // No lock to avoid wedging a stuck machine further.
3203 void
3204 procdump(void)
3205 {
3206   static char *states[] = {
3207   [UNUSED]    "unused",
3208   [EMBRYO]    "embryo",
3209   [SLEEPING]  "sleep ",
3210   [RUNNABLE]  "runble",
3211   [RUNNING]   "run   ",
3212   [ZOMBIE]    "zombie"
3213   };
3214   int i;
3215   struct proc *p;
3216   char *state;
3217   uint pc[10];
3218
3219   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3220     if(p->state == UNUSED)
3221       continue;
3222     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
3223       state = states[p->state];
3224     else
3225       state = "???";
3226     cprintf("%d %d %d %s %s %d", p->pid, p->gid, p->uid, state, p->name,p->p
3227     if(p->state == SLEEPING){
3228       getcallerpcs((uint*)p->context->ebp+2, pc);
3229       for(i=0; i<10 && pc[i] != 0; i++)
3230         cprintf(" %p", pc[i]);
3231     }
3232     cprintf("\n");
3233   }
3234 }
3235
3236
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
```



```
3250 int
3251 dogetprocs (int max, struct uproc *table)
3252 {
3253   struct proc *p;
3254   int ret = 0;
3255   int i = 0;
3256   enum procstate tmp;
3257   acquire(&ptable.lock);
3258   if (max > NPROC)
3259     return -1;
3260   for(p=ptable.proc; p<&ptable.proc[NPROC]; p++){
3261     table[i].pid = p->pid;
3262     table[i].uid = p->uid;
3263     table[i].gid = p->gid;
3264     table[i].ppid = p->ppid;
3265     tmp = p->state;
3266    if(tmp == UNUSED){
3267      goto unused;
3268    }
3269    if(tmp == EMBRYO)
3270      table[i].state =0;// "EMBRYO";
3271    if(tmp == SLEEPING)
3272      table[i].state =1;// "SLEEPING";
3273    if(tmp == RUNNABLE)
3274      table[i].state =2;// "RUNNABLE";
3275    if(tmp == RUNNING)
3276      table[i].state =3;// "RUNNING";
3277    if(tmp == ZOMBIE)
3278      table[i].state =4;// "ZOMBIE";
3279     table[i].priority = p->priority;
3280     table[i].size = p->sz;
3281     safestrcpy(table[i].name , p->name,16);
3282     /**/
3283     ++i;
3284 unused:
3285     ret = i; /*return how many times run*/
3286  }
3287
3288   release(&ptable.lock);
3289  return ret;
3290 }
3291
3292
3293
3294
3295
3296
3297
3298
3299
```

```
3300 int
3301 dgpri(int pid)
3302 {
3303   struct proc * p;
3304   acquire(&ptable.lock);
3305   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3306     if(p->pid == pid){
3307      release(&ptable.lock);
3308      return p->priority;
3309     }
3310   }
3311  release(&ptable.lock);
3312  return -1; /*invalid PID*/
3313 }
3314
3315
3316
3317
3318
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
```



```
3350 # Context switch
3351 #
3352 #   void swtch(struct context **old, struct context *new);
3353 #
3354 # Save current register context in old
3355 # and then load register context from new.
3356
3357 .globl swtch
3358 swtch:
3359   movl 4(%esp), %eax
3360   movl 8(%esp), %edx
3361
3362   # Save old callee-save registers
3363   pushl %ebp
3364   pushl %ebx
3365   pushl %esi
3366   pushl %edi
3367
3368   # Switch stacks
3369   movl %esp, (%eax)
3370   movl %edx, %esp
3371
3372   # Load new callee-save registers
3373   popl %edi
3374   popl %esi
3375   popl %ebx
3376   popl %ebp
3377   ret
3378
3379
3380
3381
3382
3383
3384
3385
3386
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399
```

```
3400 // Physical memory allocator, intended to allocate
3401 // memory for user processes, kernel stacks, page table pages,
3402 // and pipe buffers. Allocates 4096-byte pages.
3403
3404 #include "types.h"
3405 #include "defs.h"
3406 #include "param.h"
3407 #include "memlayout.h"
3408 #include "mmu.h"
3409 #include "spinlock.h"
3410
3411 void freerange(void *vstart, void *vend);
3412 extern char end[]; // first address after kernel loaded from ELF file
3413
3414 struct run {
3415   struct run *next;
3416 };
3417
3418 struct {
3419   struct spinlock lock;
3420   int use_lock;
3421   struct run *freelist;
3422 } kmem;
3423
3424 // Initialization happens in two phases.
3425 // 1. main() calls kinit1() while still using entrypgdir to place just
3426 // the pages mapped by entrypgdir on free list.
3427 // 2. main() calls kinit2() with the rest of the physical pages
3428 // after installing a full page table that maps them on all cores.
3429 void
3430 kinit1(void *vstart, void *vend)
3431 {
3432   initlock(&kmem.lock, "kmem");
3433   kmem.use_lock = 0;
3434   freerange(vstart, vend);
3435 }
3436
3437 void
3438 kinit2(void *vstart, void *vend)
3439 {
3440   freerange(vstart, vend);
3441   kmem.use_lock = 1;
3442 }
3443
3444
3445
3446
3447
3448
3449
```



```
3450 void
3451 freerange(void *vstart, void *vend)
3452 {
3453   char *p;
3454   p = (char*)PGROUNDUP((uint)vstart);
3455   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3456     kfree(p);
3457 }
3458
3459
3460 // Free the page of physical memory pointed at by v,
3461 // which normally should have been returned by a
3462 // call to kalloc().  (The exception is when
3463 // initializing the allocator; see kinit above.)
3464 void
3465 kfree(char *v)
3466 {
3467   struct run *r;
3468
3469   if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
3470     panic("kfree");
3471
3472   // Fill with junk to catch dangling refs.
3473   memset(v, 1, PGSIZE);
3474
3475   if(kmem.use_lock)
3476     acquire(&kmem.lock);
3477   r = (struct run*)v;
3478   r->next = kmem.freelist;
3479   kmem.freelist = r;
3480   if(kmem.use_lock)
3481     release(&kmem.lock);
3482 }
3483
3484 // Allocate one 4096-byte page of physical memory.
3485 // Returns a pointer that the kernel can use.
3486 // Returns 0 if the memory cannot be allocated.
3487 char*
3488 kalloc(void)
3489 {
3490   struct run *r;
3491
3492   if(kmem.use_lock)
3493     acquire(&kmem.lock);
3494   r = kmem.freelist;
3495   if(r)
3496     kmem.freelist = r->next;
3497   if(kmem.use_lock)
3498     release(&kmem.lock);
3499   return (char*)r;
```

```
3500 }
3501
3502
3503
3504
3505
3506
3507
3508
3509
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 // x86 trap and interrupt constants.
3551
3552 // Processor-defined:
3553 #define T_DIVIDE         0      // divide error
3554 #define T_DEBUG          1      // debug exception
3555 #define T_NMI            2      // non-maskable interrupt
3556 #define T_BRKPT          3      // breakpoint
3557 #define T_OFLOW          4      // overflow
3558 #define T_BOUND          5      // bounds check
3559 #define T_ILLOP          6      // illegal opcode
3560 #define T_DEVICE         7      // device not available
3561 #define T_DBLFLT         8      // double fault
3562 // #define T_COPROC      9      // reserved (not used since 486)
3563 #define T_TSS           10      // invalid task switch segment
3564 #define T_SEGNP         11      // segment not present
3565 #define T_STACK         12      // stack exception
3566 #define T_GPFLT         13      // general protection fault
3567 #define T_PGFLT         14      // page fault
3568 // #define T_RES        15      // reserved
3569 #define T_FPERR         16      // floating point error
3570 #define T_ALIGN         17      // aligment check
3571 #define T_MCHK          18      // machine check
3572 #define T_SIMDERR       19      // SIMD floating point error
3573
3574 // These are arbitrarily chosen, but with care not to overlap
3575 // processor defined exceptions or interrupt vectors.
3576 #define T_SYSCALL       64      // system call
3577 #define T_DEFAULT      500      // catchall
3578
3579 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
3580
3581 #define IRQ_TIMER        0
3582 #define IRQ_KBD          1
3583 #define IRQ_COM1         4
3584 #define IRQ_IDE         14
3585 #define IRQ_ERROR       19
3586 #define IRQ_SPURIOUS    31
3587
3588
3589
3590
3591
3592
3593
3594
3595
3596
3597
3598
3599
```

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

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

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

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

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



```
#include "mmu.h"

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

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

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

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

```
3700 #include "types.h"
3701 #include "defs.h"
3702 #include "param.h"
3703 #include "memlayout.h"
3704 #include "mmu.h"
3705 #include "proc.h"
3706 #include "x86.h"
3707 #include "traps.h"
3708 #include "spinlock.h"
3709
3710 // Interrupt descriptor table (shared by all CPUs).
3711 struct gatedesc idt[256];
3712 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
3713 struct spinlock tickslock;
3714 uint ticks;
3715
3716 void
3717 tvinit(void)
3718 {
3719   int i;
3720
3721   for(i = 0; i < 256; i++)
3722     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3723   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3724
3725   initlock(&tickslock, "time");
3726 }
3727
3728 void
3729 idtinit(void)
3730 {
3731   lidt(idt, sizeof(idt));
3732 }
3733
3734
3735
3736
3737
3738
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3749
```



```
3750 void
3751 trap(struct trapframe *tf)
3752 {
3753   if(tf->trapno == T_SYSCALL){
3754     if(proc->killed)
3755       exit();
3756     proc->tf = tf;
3757     syscall();
3758     if(proc->killed)
3759       exit();
3760     return;
3761   }
3762
3763   switch(tf->trapno){
3764   case T_IRQ0 + IRQ_TIMER:
3765     if(cpu->id == 0){
3766       acquire(&tickslock);
3767       ticks++;
3768       wakeup(&ticks);
3769       release(&tickslock);
3770     }
3771     lapiceoi();
3772     break;
3773   case T_IRQ0 + IRQ_IDE:
3774     ideintr();
3775     lapiceoi();
3776     break;
3777   case T_IRQ0 + IRQ_IDE+1:
3778     // Bochs generates spurious IDE1 interrupts.
3779     break;
3780   case T_IRQ0 + IRQ_KBD:
3781     kbdintr();
3782     lapiceoi();
3783     break;
3784   case T_IRQ0 + IRQ_COM1:
3785     uartintr();
3786     lapiceoi();
3787     break;
3788   case T_IRQ0 + 7:
3789   case T_IRQ0 + IRQ_SPURIOUS:
3790     cprintf("cpu%d: spurious interrupt at %x:%x\n",
3791             cpu->id, tf->cs, tf->eip);
3792     lapiceoi();
3793     break;
3794
3795
3796
3797
3798
3799
```

```
3800   default:
3801     if(proc == 0 || (tf->cs&3) == 0){
3802       // In kernel, it must be our mistake.
3803       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3804               tf->trapno, cpu->id, tf->eip, rcr2());
3805       panic("trap");
3806     }
3807     // In user space, assume process misbehaved.
3808     cprintf("pid %d %s: trap %d err %d on cpu %d "
3809             "eip 0x%x addr 0x%x--kill proc\n",
3810             proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
3811             rcr2());
3812     proc->killed = 1;
3813   }
3814
3815   // Force process exit if it has been killed and is in user space.
3816   // (If it is still executing in the kernel, let it keep running
3817   // until it gets to the regular system call return.)
3818   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3819     exit();
3820
3821   // Force process to give up CPU on clock tick.
3822   // If interrupts were on while locks held, would need to check nlock.
3823   if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3824     yield();
3825
3826   // Check if the process has been killed since we yielded
3827   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3828     exit();
3829 }
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
```



```
3850 // System call numbers
3851 #define SYS_fork    1
3852 #define SYS_exit    2
3853 #define SYS_wait    3
3854 #define SYS_pipe    4
3855 #define SYS_read    5
3856 #define SYS_kill    6
3857 #define SYS_exec    7
3858 #define SYS_fstat   8
3859 #define SYS_chdir   9
3860 #define SYS_dup    10
3861 #define SYS_getpid 11
3862 #define SYS_sbrk   12
3863 #define SYS_sleep  13
3864 #define SYS_uptime 14
3865 #define SYS_open   15
3866 #define SYS_write  16
3867 #define SYS_mknod  17
3868 #define SYS_unlink 18
3869 #define SYS_link   19
3870 #define SYS_mkdir  20
3871 #define SYS_close  21
3872 #define SYS_halt   22
3873 #define SYS_date   23
3874 #define SYS_getuid 24
3875 #define SYS_getgid 25
3876 #define SYS_getppid 26
3877 #define SYS_setuid 27
3878 #define SYS_setgid 28
3879 #define SYS_getprocs 29
3880 #define SYS_getpriority 30
3881 #define SYS_setpriority 31
3882
3883
3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
```

```
3900 #include "types.h"
3901 #include "defs.h"
3902 #include "param.h"
3903 #include "memlayout.h"
3904 #include "mmu.h"
3905 #include "proc.h"
3906 #include "x86.h"
3907 #include "syscall.h"
3908
3909 // User code makes a system call with INT T_SYSCALL.
3910 // System call number in %eax.
3911 // Arguments on the stack, from the user call to the C
3912 // library system call function. The saved user %esp points
3913 // to a saved program counter, and then the first argument.
3914
3915 // Fetch the int at addr from the current process.
3916 int
3917 fetchint(uint addr, int *ip)
3918 {
3919   if(addr >= proc->sz || addr+4 > proc->sz)
3920     return -1;
3921   *ip = *(int*)(addr);
3922   return 0;
3923 }
3924
3925 // Fetch the nul-terminated string at addr from the current process.
3926 // Doesn't actually copy the string - just sets *pp to point at it.
3927 // Returns length of string, not including nul.
3928 int
3929 fetchstr(uint addr, char **pp)
3930 {
3931   char *s, *ep;
3932
3933   if(addr >= proc->sz)
3934     return -1;
3935   *pp = (char*)addr;
3936   ep = (char*)proc->sz;
3937   for(s = *pp; s < ep; s++)
3938     if(*s == 0)
3939       return s - *pp;
3940   return -1;
3941 }
3942
3943 // Fetch the nth 32-bit system call argument.
3944 int
3945 argint(int n, int *ip)
3946 {
3947   return fetchint(proc->tf->esp + 4 + 4*n, ip);
3948 }
3949
```



```
3950 // Fetch the nth word-sized system call argument as a pointer
3951 // to a block of memory of size n bytes.  Check that the pointer
3952 // lies within the process address space.
3953 int
3954 argptr(int n, char **pp, int size)
3955 {
3956   int i;
3957
3958   if(argint(n, &i) < 0)
3959     return -1;
3960   if((uint)i >= proc->sz || (uint)i+size > proc->sz)
3961     return -1;
3962   *pp = (char*)i;
3963   return 0;
3964 }
3965
3966 // Fetch the nth word-sized system call argument as a string pointer.
3967 // Check that the pointer is valid and the string is nul-terminated.
3968 // (There is no shared writable memory, so the string can't change
3969 // between this check and being used by the kernel.)
3970 int
3971 argstr(int n, char **pp)
3972 {
3973   int addr;
3974   if(argint(n, &addr) < 0)
3975     return -1;
3976   return fetchstr(addr, pp);
3977 }
3978
3979 extern int sys_chdir(void);
3980 extern int sys_close(void);
3981 extern int sys_dup(void);
3982 extern int sys_exec(void);
3983 extern int sys_exit(void);
3984 extern int sys_fork(void);
3985 extern int sys_fstat(void);
3986 extern int sys_getpid(void);
3987 extern int sys_kill(void);
3988 extern int sys_link(void);
3989 extern int sys_mkdir(void);
3990 extern int sys_mknod(void);
3991 extern int sys_open(void);
3992 extern int sys_pipe(void);
3993 extern int sys_read(void);
3994 extern int sys_sbrk(void);
3995 extern int sys_sleep(void);
3996 extern int sys_unlink(void);
3997 extern int sys_wait(void);
3998 extern int sys_write(void);
3999 extern int sys_uptime(void);
```

```
4000 extern int sys_halt(void);
4001 extern int sys_date(void);
4002 extern int sys_getuid(void);
4003 extern int sys_getgid(void);
4004 extern int sys_getppid(void);
4005 extern int sys_setuid(void);
4006 extern int sys_setgid(void);
4007 extern int sys_getprocs(void);
4008 extern int sys_getpriority(void);
4009 extern int sys_setpriority(void);
4010 static int (*syscalls[])(void) = {
4011 [SYS_fork]    sys_fork,
4012 [SYS_exit]    sys_exit,
4013 [SYS_wait]    sys_wait,
4014 [SYS_pipe]    sys_pipe,
4015 [SYS_read]    sys_read,
4016 [SYS_kill]    sys_kill,
4017 [SYS_exec]    sys_exec,
4018 [SYS_fstat]   sys_fstat,
4019 [SYS_chdir]   sys_chdir,
4020 [SYS_dup]     sys_dup,
4021 [SYS_getpid]  sys_getpid,
4022 [SYS_sbrk]    sys_sbrk,
4023 [SYS_sleep]   sys_sleep,
4024 [SYS_uptime]  sys_uptime,
4025 [SYS_open]    sys_open,
4026 [SYS_write]   sys_write,
4027 [SYS_mknod]   sys_mknod,
4028 [SYS_unlink]  sys_unlink,
4029 [SYS_link]    sys_link,
4030 [SYS_mkdir]   sys_mkdir,
4031 [SYS_close]   sys_close,
4032 [SYS_halt]    sys_halt,
4033 [SYS_date]    sys_date,
4034 [SYS_getuid]  sys_getuid,
4035 [SYS_getgid]  sys_getgid,
4036 [SYS_getppid] sys_getppid,
4037 [SYS_setuid]  sys_setuid,
4038 [SYS_setgid]  sys_setgid,
4039 [SYS_getprocs] sys_getprocs,
4040 [SYS_getpriority]  sys_getpriority,
4041 [SYS_setpriority]  sys_setpriority,
4042 };
4043
4044
4045
4046
4047
4048
4049
```



```
4050 void getname(int num)
4051 {
4052     switch(num)/*figure out which syscall*/
4053     {
4054             case 1:
4055                     cprintf("fork -> ");
4056                     return;
4057             case 2:
4058                     cprintf("exit -> ");
4059                     return;
4060             case 3:
4061                     cprintf("wait -> ");
4062                     return;
4063             case 4:
4064                     cprintf("pipe -> ");
4065                     return;
4066             case 5:
4067                     cprintf("read -> ");
4068                     return;
4069             case 6:
4070                     cprintf("kill -> ");
4071                     return;
4072             case 7:
4073                     cprintf("exec -> ");
4074                     return;
4075             case 8:
4076                     cprintf("fstat -> ");
4077                     return;
4078             case 9:
4079                     cprintf("chdir -> ");
4080                     return;
4081             case 10:
4082                     cprintf("dup -> ");
4083                     return;
4084             case 11:
4085                     cprintf("getpid -> ");
4086                     return;
4087             case 12:
4088                     cprintf("sbrk -> ");
4089                     return;
4090             case 13:
4091                     cprintf("sleep -> ");
4092                     return;
4093             case 14:
4094                     cprintf("uptime -> ");
4095                     return;
4096             case 15:
4097                     cprintf("open -> ");
4098                     return;
4099             case 16:
```

```
4100                    cprintf("write -> ");
4101                    return;
4102            case 17:
4103                    cprintf("mknod -> ");
4104                    return;
4105            case 18:
4106                    cprintf("unlink -> ");
4107                    return;
4108            case 19:
4109                    cprintf("link -> ");
4110                    return;
4111            case 20:
4112                    cprintf("mkdir -> ");
4113                    return;
4114            case 21:
4115                    cprintf("close -> ");
4116                    return;
4117            case 22:
4118                    cprintf("halt -> ");
4119                    return;
4120            default:
4121                    cprintf("ERROR \n");
4122    }
4123 }
4124
4125 void
4126 syscall(void)
4127 {
4128   int num;
4129   num = proc->tf->eax;
4130   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
4131     /*getname(num);*//*call lookup table*/
4132     proc->tf->eax = syscalls[num]();
4133     /*cprintf("%d\n",proc->tf->eax);*/
4134
4135
4136   } else {
4137     cprintf("%d %s: unknown sys call %d\n",
4138             proc->pid, proc->name, num);
4139     proc->tf->eax = -1;
4140   }
4141 }
4142
4143
4144
4145
4146
4147
4148
4149
```



```
4150 #include "types.h"
4151 #include "x86.h"
4152 #include "defs.h"
4153 #include "date.h"
4154 #include "param.h"
4155 #include "memlayout.h"
4156 #include "mmu.h"
4157 #include "proc.h"
4158 #include "ps.h"
4159 int
4160 sys_fork(void)
4161 {
4162   return fork();
4163 }
4164
4165 int
4166 sys_exit(void)
4167 {
4168   exit();
4169   return 0;  // not reached
4170 }
4171
4172 int
4173 sys_wait(void)
4174 {
4175   return wait();
4176 }
4177
4178 int
4179 sys_kill(void)
4180 {
4181   int pid;
4182
4183   if(argint(0, &pid) < 0)
4184     return -1;
4185   return kill(pid);
4186 }
4187
4188 int
4189 sys_getpid(void)
4190 {
4191   return proc->pid;
4192 }
4193
4194
4195
4196
4197
4198
4199
```

```
4200 int
4201 sys_sbrk(void)
4202 {
4203   int addr;
4204   int n;
4205
4206   if(argint(0, &n) < 0)
4207     return -1;
4208   addr = proc->sz;
4209   if(growproc(n) < 0)
4210     return -1;
4211   return addr;
4212 }
4213
4214 int
4215 sys_sleep(void)
4216 {
4217   int n;
4218   uint ticks0;
4219
4220   if(argint(0, &n) < 0)
4221     return -1;
4222   acquire(&tickslock);
4223   ticks0 = ticks;
4224   while(ticks - ticks0 < n){
4225     if(proc->killed){
4226       release(&tickslock);
4227       return -1;
4228     }
4229     sleep(&ticks, &tickslock);
4230   }
4231   release(&tickslock);
4232   return 0;
4233 }
4234
4235 // return how many clock tick interrupts have occurred
4236 // since start.
4237 int
4238 sys_uptime(void)
4239 {
4240   uint xticks;
4241
4242   acquire(&tickslock);
4243   xticks = ticks;
4244   release(&tickslock);
4245   return xticks;
4246 }
4247
4248
4249
```



```
4250 //Turn of the computer
4251 int sys_halt(void){
4252   cprintf("Shutting down ...\n");
4253   outw (0xB004, 0x0 | 0x2000);
4254   return 0;
4255 }
4256
4257 int
4258 sys_date(void){
4259
4260   struct rtcdate *r;
4261   if (argptr(0,(void*)&r, sizeof(&r)) < 0)
4262     return -1;
4263   cmostime(r);
4264   return 0;
4265 }
4266
4267 int
4268 sys_getuid(void){
4269  return proc->uid;
4270 }
4271
4272 int
4273 sys_getgid(void){
4274  return proc->gid;
4275 }
4276
4277 int
4278 sys_getppid(void){
4279  return proc->parent->pid;
4280 }
4281
4282 int
4283 sys_setuid(void){
4284  int val;
4285  if(argint(0, &val) < 0)
4286    return -1;
4287  proc->uid = val;
4288  return 0;
4289 }
4290
4291 int
4292 sys_setgid(void){
4293  int val;
4294  if(argint(0, &val) < 0)
4295    return -1;
4296  proc->gid = val;
4297  return 0;
4298 }
4299
```

```
4300 int
4301 sys_getprocs(void){
4302  int max;
4303  struct uproc *table;
4304
4305  if(argint(0, &max)<0)
4306    return -1;
4307
4308  if (argptr(1,(void*)&table, sizeof(&table)) < 0)
4309    return -1;
4310  max = dogetprocs (max, table);
4311  return max; /*so usr prog knows hw mny itrs*/
4312 }
4313
4314 int
4315 sys_getpriority(void){
4316  int pid;
4317  if(argint(0, &pid)<0)
4318    return -1;
4319  pid = dgpri(pid);
4320  return pid;
4321 }
4322
4323 int
4324 sys_setpriority(void){
4325
4326  int pid;
4327  int priority;
4328
4329  if(argint(0, &pid)<0)
4330    return -1;
4331  if(argint(1, &priority)<0)
4332    return -1;
4333  dspri(pid, priority);
4334 return 0;
4335 }
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
```



```
4350 // halt the system.
4351 #include "types.h"
4352 #include "user.h"
4353
4354 int
4355 main(void) {
4356   halt();
4357   return 0;
4358 }
4359
4360
4361
4362
4363
4364
4365
4366
4367
4368
4369
4370
4371
4372
4373
4374
4375
4376
4377
4378
4379
4380
4381
4382
4383
4384
4385
4386
4387
4388
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
```

```
4400 struct buf {
4401   int flags;
4402   uint dev;
4403   uint blockno;
4404   struct buf *prev; // LRU cache list
4405   struct buf *next;
4406   struct buf *qnext; // disk queue
4407   uchar data[BSIZE];
4408 };
4409 #define B_BUSY  0x1  // buffer is locked by some process
4410 #define B_VALID 0x2  // buffer has been read from disk
4411 #define B_DIRTY 0x4  // buffer needs to be written to disk
4412
4413
4414
4415
4416
4417
4418
4419
4420
4421
4422
4423
4424
4425
4426
4427
4428
4429
4430
4431
4432
4433
4434
4435
4436
4437
4438
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
```



```
4450 #define O_RDONLY  0x000
4451 #define O_WRONLY  0x001
4452 #define O_RDWR    0x002
4453 #define O_CREATE  0x200
4454
4455
4456
4457
4458
4459
4460
4461
4462
4463
4464
4465
4466
4467
4468
4469
4470
4471
4472
4473
4474
4475
4476
4477
4478
4479
4480
4481
4482
4483
4484
4485
4486
4487
4488
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499
```

```
#define T_DIR  1   // Directory
#define T_FILE 2   // File
#define T_DEV  3   // Device

struct stat {
  short type;  // Type of file
  int dev;     // File system's disk device
  uint ino;    // Inode number
  short nlink; // Number of links to file
  uint size;   // Size of file in bytes
};
```



```
// On-disk file system format.
// Both the kernel and user programs use this header file.


#define ROOTINO 1  // root i-number
#define BSIZE 512  // block size

// Disk layout:
// [ boot block | super block | log | inode blocks | free bit map | data bloc
//
// mkfs computes the super block and builds an initial file system. The super
// the disk layout:
struct superblock {
  uint size;         // Size of file system image (blocks)
  uint nblocks;      // Number of data blocks
  uint ninodes;      // Number of inodes.
  uint nlog;         // Number of log blocks
  uint logstart;     // Block number of first log block
  uint inodestart;   // Block number of first inode block
  uint bmapstart;    // Block number of first free map block
};

#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT)

// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};
```

```
4600 // Inodes per block.
4601 #define IPB           (BSIZE / sizeof(struct dinode))
4602
4603 // Block containing inode i
4604 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
4605
4606 // Bitmap bits per block
4607 #define BPB           (BSIZE*8)
4608
4609 // Block of free map containing bit for block b
4610 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
4611
4612 // Directory is a file containing a sequence of dirent structures.
4613 #define DIRSIZ 14
4614
4615 struct dirent {
4616   ushort inum;
4617   char name[DIRSIZ];
4618 };
4619
4620
4621
4622
4623
4624
4625
4626
4627
4628
4629
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649
```



```
4650 struct file {
4651   enum { FD_NONE, FD_PIPE, FD_INODE } type;
4652   int ref; // reference count
4653   char readable;
4654   char writable;
4655   struct pipe *pipe;
4656   struct inode *ip;
4657   uint off;
4658 };
4659
4660
4661 // in-memory copy of an inode
4662 struct inode {
4663   uint dev;           // Device number
4664   uint inum;          // Inode number
4665   int ref;            // Reference count
4666   int flags;          // I_BUSY, I_VALID
4667
4668   short type;         // copy of disk inode
4669   short major;
4670   short minor;
4671   short nlink;
4672   uint size;
4673   uint addrs[NDIRECT+1];
4674 };
4675 #define I_BUSY 0x1
4676 #define I_VALID 0x2
4677
4678 // table mapping major device number to
4679 // device functions
4680 struct devsw {
4681   int (*read)(struct inode*, char*, int);
4682   int (*write)(struct inode*, char*, int);
4683 };
4684
4685 extern struct devsw devsw[];
4686
4687 #define CONSOLE 1
4688
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4699
```

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



```
4750 // Simple PIO-based (non-DMA) IDE driver code.
4751
4752 #include "types.h"
4753 #include "defs.h"
4754 #include "param.h"
4755 #include "memlayout.h"
4756 #include "mmu.h"
4757 #include "proc.h"
4758 #include "x86.h"
4759 #include "traps.h"
4760 #include "spinlock.h"
4761 #include "fs.h"
4762 #include "buf.h"
4763
4764 #define SECTOR_SIZE   512
4765 #define IDE_BSY       0x80
4766 #define IDE_DRDY      0x40
4767 #define IDE_DF        0x20
4768 #define IDE_ERR       0x01
4769
4770 #define IDE_CMD_READ  0x20
4771 #define IDE_CMD_WRITE 0x30
4772
4773 // idequeue points to the buf now being read/written to the disk.
4774 // idequeue->qnext points to the next buf to be processed.
4775 // You must hold idelock while manipulating queue.
4776
4777 static struct spinlock idelock;
4778 static struct buf *idequeue;
4779
4780 static int havedisk1;
4781 static void idestart(struct buf*);
4782
4783 // Wait for IDE disk to become ready.
4784 static int
4785 idewait(int checkerr)
4786 {
4787   int r;
4788
4789   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4790     ;
4791   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4792     return -1;
4793   return 0;
4794 }
4795
4796
4797
4798
4799
```

```
4800 void
4801 ideinit(void)
4802 {
4803   int i;
4804
4805   initlock(&idelock, "ide");
4806   picenable(IRQ_IDE);
4807   ioapicenable(IRQ_IDE, ncpu - 1);
4808   idewait(0);
4809
4810   // Check if disk 1 is present
4811   outb(0x1f6, 0xe0 | (1<<4));
4812   for(i=0; i<1000; i++){
4813     if(inb(0x1f7) != 0){
4814       havedisk1 = 1;
4815       break;
4816     }
4817   }
4818
4819   // Switch back to disk 0.
4820   outb(0x1f6, 0xe0 | (0<<4));
4821 }
4822
4823 // Start the request for b.  Caller must hold idelock.
4824 static void
4825 idestart(struct buf *b)
4826 {
4827   if(b == 0)
4828     panic("idestart");
4829   if(b->blockno >= FSSIZE)
4830     panic("incorrect blockno");
4831   int sector_per_block =  BSIZE/SECTOR_SIZE;
4832   int sector = b->blockno * sector_per_block;
4833
4834   if (sector_per_block > 7) panic("idestart");
4835
4836   idewait(0);
4837   outb(0x3f6, 0);  // generate interrupt
4838   outb(0x1f2, sector_per_block);  // number of sectors
4839   outb(0x1f3, sector & 0xff);
4840   outb(0x1f4, (sector >> 8) & 0xff);
4841   outb(0x1f5, (sector >> 16) & 0xff);
4842   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4843   if(b->flags & B_DIRTY){
4844     outb(0x1f7, IDE_CMD_WRITE);
4845     outsl(0x1f0, b->data, BSIZE/4);
4846   } else {
4847     outb(0x1f7, IDE_CMD_READ);
4848   }
4849 }
```



```
4850 // Interrupt handler.
4851 void
4852 ideintr(void)
4853 {
4854   struct buf *b;
4855
4856   // First queued buffer is the active request.
4857   acquire(&idelock);
4858   if((b = idequeue) == 0){
4859     release(&idelock);
4860     // cprintf("spurious IDE interrupt\n");
4861     return;
4862   }
4863   idequeue = b->qnext;
4864
4865   // Read data if needed.
4866   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4867     insl(0x1f0, b->data, BSIZE/4);
4868
4869   // Wake process waiting for this buf.
4870   b->flags |= B_VALID;
4871   b->flags &= ~B_DIRTY;
4872   wakeup(b);
4873
4874   // Start disk on next buf in queue.
4875   if(idequeue != 0)
4876     idestart(idequeue);
4877
4878   release(&idelock);
4879 }
4880
4881
4882
4883
4884
4885
4886
4887
4888
4889
4890
4891
4892
4893
4894
4895
4896
4897
4898
4899
```

```
4900 // Sync buf with disk.
4901 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
4902 // Else if B_VALID is not set, read buf from disk, set B_VALID.
4903 void
4904 iderw(struct buf *b)
4905 {
4906   struct buf **pp;
4907
4908   if(!(b->flags & B_BUSY))
4909     panic("iderw: buf not busy");
4910   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
4911     panic("iderw: nothing to do");
4912   if(b->dev != 0 && !havedisk1)
4913     panic("iderw: ide disk 1 not present");
4914
4915   acquire(&idelock);
4916
4917   // Append b to idequeue.
4918   b->qnext = 0;
4919   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
4920     ;
4921   *pp = b;
4922
4923   // Start disk if necessary.
4924   if(idequeue == b)
4925     idestart(b);
4926
4927   // Wait for request to finish.
4928   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
4929     sleep(b, &idelock);
4930   }
4931
4932   release(&idelock);
4933 }
4934
4935
4936
4937
4938
4939
4940
4941
4942
4943
4944
4945
4946
4947
4948
4949
```



```
4950 // Buffer cache.
4951 //
4952 // The buffer cache is a linked list of buf structures holding
4953 // cached copies of disk block contents.  Caching disk blocks
4954 // in memory reduces the number of disk reads and also provides
4955 // a synchronization point for disk blocks used by multiple processes.
4956 //
4957 // Interface:
4958 // * To get a buffer for a particular disk block, call bread.
4959 // * After changing buffer data, call bwrite to write it to disk.
4960 // * When done with the buffer, call brelse.
4961 // * Do not use the buffer after calling brelse.
4962 // * Only one process at a time can use a buffer,
4963 //     so do not keep them longer than necessary.
4964 //
4965 // The implementation uses three state flags internally:
4966 // * B_BUSY: the block has been returned from bread
4967 //     and has not been passed back to brelse.
4968 // * B_VALID: the buffer data has been read from the disk.
4969 // * B_DIRTY: the buffer data has been modified
4970 //     and needs to be written to disk.
4971
4972 #include "types.h"
4973 #include "defs.h"
4974 #include "param.h"
4975 #include "spinlock.h"
4976 #include "fs.h"
4977 #include "buf.h"
4978
4979 struct {
4980   struct spinlock lock;
4981   struct buf buf[NBUF];
4982
4983   // Linked list of all buffers, through prev/next.
4984   // head.next is most recently used.
4985   struct buf head;
4986 } bcache;
4987
4988 void
4989 binit(void)
4990 {
4991   struct buf *b;
4992
4993   initlock(&bcache.lock, "bcache");
4994
4995
4996
4997
4998
4999
```

```
5000   // Create linked list of buffers
5001   bcache.head.prev = &bcache.head;
5002   bcache.head.next = &bcache.head;
5003   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
5004     b->next = bcache.head.next;
5005     b->prev = &bcache.head;
5006     b->dev = -1;
5007     bcache.head.next->prev = b;
5008     bcache.head.next = b;
5009   }
5010 }
5011
5012 // Look through buffer cache for block on device dev.
5013 // If not found, allocate a buffer.
5014 // In either case, return B_BUSY buffer.
5015 static struct buf*
5016 bget(uint dev, uint blockno)
5017 {
5018   struct buf *b;
5019
5020   acquire(&bcache.lock);
5021
5022  loop:
5023   // Is the block already cached?
5024   for(b = bcache.head.next; b != &bcache.head; b = b->next){
5025     if(b->dev == dev && b->blockno == blockno){
5026       if(!(b->flags & B_BUSY)){
5027         b->flags |= B_BUSY;
5028         release(&bcache.lock);
5029         return b;
5030       }
5031       sleep(b, &bcache.lock);
5032       goto loop;
5033     }
5034   }
5035
5036   // Not cached; recycle some non-busy and clean buffer.
5037   // "clean" because B_DIRTY and !B_BUSY means log.c
5038   // hasn't yet committed the changes to the buffer.
5039   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
5040     if((b->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
5041       b->dev = dev;
5042       b->blockno = blockno;
5043       b->flags = B_BUSY;
5044       release(&bcache.lock);
5045       return b;
5046     }
5047   }
5048   panic("bget: no buffers");
5049 }
```



```
5050 // Return a B_BUSY buf with the contents of the indicated block.
5051 struct buf*
5052 bread(uint dev, uint blockno)
5053 {
5054   struct buf *b;
5055
5056   b = bget(dev, blockno);
5057   if(!(b->flags & B_VALID)) {
5058     iderw(b);
5059   }
5060   return b;
5061 }
5062
5063 // Write b's contents to disk.  Must be B_BUSY.
5064 void
5065 bwrite(struct buf *b)
5066 {
5067   if((b->flags & B_BUSY) == 0)
5068     panic("bwrite");
5069   b->flags |= B_DIRTY;
5070   iderw(b);
5071 }
5072
5073 // Release a B_BUSY buffer.
5074 // Move to the head of the MRU list.
5075 void
5076 brelse(struct buf *b)
5077 {
5078   if((b->flags & B_BUSY) == 0)
5079     panic("brelse");
5080
5081   acquire(&bcache.lock);
5082
5083   b->next->prev = b->prev;
5084   b->prev->next = b->next;
5085   b->next = bcache.head.next;
5086   b->prev = &bcache.head;
5087   bcache.head.next->prev = b;
5088   bcache.head.next = b;
5089
5090   b->flags &= ~B_BUSY;
5091   wakeup(b);
5092
5093   release(&bcache.lock);
5094 }
5095
5096
5097
5098
5099
```

```
5100 // Blank page.
5101
5102
5103
5104
5105
5106
5107
5108
5109
5110
5111
5112
5113
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 #include "types.h"
5151 #include "defs.h"
5152 #include "param.h"
5153 #include "spinlock.h"
5154 #include "fs.h"
5155 #include "buf.h"
5156
5157 // Simple logging that allows concurrent FS system calls.
5158 //
5159 // A log transaction contains the updates of multiple FS system
5160 // calls. The logging system only commits when there are
5161 // no FS system calls active. Thus there is never
5162 // any reasoning required about whether a commit might
5163 // write an uncommitted system call's updates to disk.
5164 //
5165 // A system call should call begin_op()/end_op() to mark
5166 // its start and end. Usually begin_op() just increments
5167 // the count of in-progress FS system calls and returns.
5168 // But if it thinks the log is close to running out, it
5169 // sleeps until the last outstanding end_op() commits.
5170 //
5171 // The log is a physical re-do log containing disk blocks.
5172 // The on-disk log format:
5173 //   header block, containing block #s for block A, B, C, ...
5174 //   block A
5175 //   block B
5176 //   block C
5177 //   ...
5178 // Log appends are synchronous.
5179
5180 // Contents of the header block, used for both the on-disk header block
5181 // and to keep track in memory of logged block# before commit.
5182 struct logheader {
5183   int n;
5184   int block[LOGSIZE];
5185 };
5186
5187 struct log {
5188   struct spinlock lock;
5189   int start;
5190   int size;
5191   int outstanding; // how many FS sys calls are executing.
5192   int committing;  // in commit(), please wait.
5193   int dev;
5194   struct logheader lh;
5195 };
5196
5197
5198
5199
```

```
5200 struct log log;
5201
5202 static void recover_from_log(void);
5203 static void commit();
5204
5205 void
5206 initlog(int dev)
5207 {
5208   if (sizeof(struct logheader) >= BSIZE)
5209     panic("initlog: too big logheader");
5210
5211   struct superblock sb;
5212   initlock(&log.lock, "log");
5213   readsb(dev, &sb);
5214   log.start = sb.logstart;
5215   log.size = sb.nlog;
5216   log.dev = dev;
5217   recover_from_log();
5218 }
5219
5220 // Copy committed blocks from log to their home location
5221 static void
5222 install_trans(void)
5223 {
5224   int tail;
5225
5226   for (tail = 0; tail < log.lh.n; tail++) {
5227     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
5228     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
5229     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
5230     bwrite(dbuf);  // write dst to disk
5231     brelse(lbuf);
5232     brelse(dbuf);
5233   }
5234 }
5235
5236 // Read the log header from disk into the in-memory log header
5237 static void
5238 read_head(void)
5239 {
5240   struct buf *buf = bread(log.dev, log.start);
5241   struct logheader *lh = (struct logheader *) (buf->data);
5242   int i;
5243   log.lh.n = lh->n;
5244   for (i = 0; i < log.lh.n; i++) {
5245     log.lh.block[i] = lh->block[i];
5246   }
5247   brelse(buf);
5248 }
5249
```



```
5250 // Write in-memory log header to disk.
5251 // This is the true point at which the
5252 // current transaction commits.
5253 static void
5254 write_head(void)
5255 {
5256   struct buf *buf = bread(log.dev, log.start);
5257   struct logheader *hb = (struct logheader *) (buf->data);
5258   int i;
5259   hb->n = log.lh.n;
5260   for (i = 0; i < log.lh.n; i++) {
5261     hb->block[i] = log.lh.block[i];
5262   }
5263   bwrite(buf);
5264   brelse(buf);
5265 }
5266
5267 static void
5268 recover_from_log(void)
5269 {
5270   read_head();
5271   install_trans(); // if committed, copy from log to disk
5272   log.lh.n = 0;
5273   write_head(); // clear the log
5274 }
5275
5276 // called at the start of each FS system call.
5277 void
5278 begin_op(void)
5279 {
5280   acquire(&log.lock);
5281   while(1){
5282     if(log.committing){
5283       sleep(&log, &log.lock);
5284     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
5285       // this op might exhaust log space; wait for commit.
5286       sleep(&log, &log.lock);
5287     } else {
5288       log.outstanding += 1;
5289       release(&log.lock);
5290       break;
5291     }
5292   }
5293 }
5294
5295
5296
5297
5298
5299
```

```
5300 // called at the end of each FS system call.
5301 // commits if this was the last outstanding operation.
5302 void
5303 end_op(void)
5304 {
5305   int do_commit = 0;
5306
5307   acquire(&log.lock);
5308   log.outstanding -= 1;
5309   if(log.committing)
5310     panic("log.committing");
5311   if(log.outstanding == 0){
5312     do_commit = 1;
5313     log.committing = 1;
5314   } else {
5315     // begin_op() may be waiting for log space.
5316     wakeup(&log);
5317   }
5318   release(&log.lock);
5319
5320   if(do_commit){
5321     // call commit w/o holding locks, since not allowed
5322     // to sleep with locks.
5323     commit();
5324     acquire(&log.lock);
5325     log.committing = 0;
5326     wakeup(&log);
5327     release(&log.lock);
5328   }
5329 }
5330
5331 // Copy modified blocks from cache to log.
5332 static void
5333 write_log(void)
5334 {
5335   int tail;
5336
5337   for (tail = 0; tail < log.lh.n; tail++) {
5338     struct buf *to = bread(log.dev, log.start+tail+1); // log block
5339     struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
5340     memmove(to->data, from->data, BSIZE);
5341     bwrite(to);  // write the log
5342     brelse(from);
5343     brelse(to);
5344   }
5345 }
5346
5347
5348
5349
```



```
5350 static void
5351 commit()
5352 {
5353   if (log.lh.n > 0) {
5354     write_log();     // Write modified blocks from cache to log
5355     write_head();    // Write header to disk -- the real commit
5356     install_trans(); // Now install writes to home locations
5357     log.lh.n = 0;
5358     write_head();    // Erase the transaction from the log
5359   }
5360 }
5361
5362 // Caller has modified b->data and is done with the buffer.
5363 // Record the block number and pin in the cache with B_DIRTY.
5364 // commit()/write_log() will do the disk write.
5365 //
5366 // log_write() replaces bwrite(); a typical use is:
5367 //   bp = bread(...)
5368 //   modify bp->data[]
5369 //   log_write(bp)
5370 //   brelse(bp)
5371 void
5372 log_write(struct buf *b)
5373 {
5374   int i;
5375
5376   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
5377     panic("too big a transaction");
5378   if (log.outstanding < 1)
5379     panic("log_write outside of trans");
5380
5381   acquire(&log.lock);
5382   for (i = 0; i < log.lh.n; i++) {
5383     if (log.lh.block[i] == b->blockno)   // log absorbtion
5384       break;
5385   }
5386   log.lh.block[i] = b->blockno;
5387   if (i == log.lh.n)
5388     log.lh.n++;
5389   b->flags |= B_DIRTY; // prevent eviction
5390   release(&log.lock);
5391 }
5392
5393
5394
5395
5396
5397
5398
5399
```

```
5400 // File system implementation.  Five layers:
5401 //   + Blocks: allocator for raw disk blocks.
5402 //   + Log: crash recovery for multi-step updates.
5403 //   + Files: inode allocator, reading, writing, metadata.
5404 //   + Directories: inode with special contents (list of other inodes!)
5405 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
5406 //
5407 // This file contains the low-level file system manipulation
5408 // routines.  The (higher-level) system call implementations
5409 // are in sysfile.c.
5410
5411 #include "types.h"
5412 #include "defs.h"
5413 #include "param.h"
5414 #include "stat.h"
5415 #include "mmu.h"
5416 #include "proc.h"
5417 #include "spinlock.h"
5418 #include "fs.h"
5419 #include "buf.h"
5420 #include "file.h"
5421
5422 #define min(a, b) ((a) < (b) ? (a) : (b))
5423 static void itrunc(struct inode*);
5424 struct superblock sb;   // there should be one per dev, but we run with one d
5425
5426 // Read the super block.
5427 void
5428 readsb(int dev, struct superblock *sb)
5429 {
5430   struct buf *bp;
5431
5432   bp = bread(dev, 1);
5433   memmove(sb, bp->data, sizeof(*sb));
5434   brelse(bp);
5435 }
5436
5437 // Zero a block.
5438 static void
5439 bzero(int dev, int bno)
5440 {
5441   struct buf *bp;
5442
5443   bp = bread(dev, bno);
5444   memset(bp->data, 0, BSIZE);
5445   log_write(bp);
5446   brelse(bp);
5447 }
5448
5449
```



```
5450 // Blocks.
5451
5452 // Allocate a zeroed disk block.
5453 static uint
5454 balloc(uint dev)
5455 {
5456   int b, bi, m;
5457   struct buf *bp;
5458
5459   bp = 0;
5460   for(b = 0; b < sb.size; b += BPB){
5461     bp = bread(dev, BBLOCK(b, sb));
5462     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
5463       m = 1 << (bi % 8);
5464       if((bp->data[bi/8] & m) == 0){  // Is block free?
5465         bp->data[bi/8] |= m;  // Mark block in use.
5466         log_write(bp);
5467         brelse(bp);
5468         bzero(dev, b + bi);
5469         return b + bi;
5470       }
5471     }
5472     brelse(bp);
5473   }
5474   panic("balloc: out of blocks");
5475 }
5476
5477 // Free a disk block.
5478 static void
5479 bfree(int dev, uint b)
5480 {
5481   struct buf *bp;
5482   int bi, m;
5483
5484   readsb(dev, &sb);
5485   bp = bread(dev, BBLOCK(b, sb));
5486   bi = b % BPB;
5487   m = 1 << (bi % 8);
5488   if((bp->data[bi/8] & m) == 0)
5489     panic("freeing free block");
5490   bp->data[bi/8] &= ~m;
5491   log_write(bp);
5492   brelse(bp);
5493 }
5494
5495
5496
5497
5498
5499
```

```
5500 // Inodes.
5501 //
5502 // An inode describes a single unnamed file.
5503 // The inode disk structure holds metadata: the file's type,
5504 // its size, the number of links referring to it, and the
5505 // list of blocks holding the file's content.
5506 //
5507 // The inodes are laid out sequentially on disk at
5508 // sb.startinode. Each inode has a number, indicating its
5509 // position on the disk.
5510 //
5511 // The kernel keeps a cache of in-use inodes in memory
5512 // to provide a place for synchronizing access
5513 // to inodes used by multiple processes. The cached
5514 // inodes include book-keeping information that is
5515 // not stored on disk: ip->ref and ip->flags.
5516 //
5517 // An inode and its in-memory represtative go through a
5518 // sequence of states before they can be used by the
5519 // rest of the file system code.
5520 //
5521 // * Allocation: an inode is allocated if its type (on disk)
5522 //   is non-zero. ialloc() allocates, iput() frees if
5523 //   the link count has fallen to zero.
5524 //
5525 // * Referencing in cache: an entry in the inode cache
5526 //   is free if ip->ref is zero. Otherwise ip->ref tracks
5527 //   the number of in-memory pointers to the entry (open
5528 //   files and current directories). iget() to find or
5529 //   create a cache entry and increment its ref, iput()
5530 //   to decrement ref.
5531 //
5532 // * Valid: the information (type, size, &c) in an inode
5533 //   cache entry is only correct when the I_VALID bit
5534 //   is set in ip->flags. ilock() reads the inode from
5535 //   the disk and sets I_VALID, while iput() clears
5536 //   I_VALID if ip->ref has fallen to zero.
5537 //
5538 // * Locked: file system code may only examine and modify
5539 //   the information in an inode and its content if it
5540 //   has first locked the inode. The I_BUSY flag indicates
5541 //   that the inode is locked. ilock() sets I_BUSY,
5542 //   while iunlock clears it.
5543 //
5544 // Thus a typical sequence is:
5545 //   ip = iget(dev, inum)
5546 //   ilock(ip)
5547 //   ... examine and modify ip->xxx ...
5548 //   iunlock(ip)
5549 //   iput(ip)
```



```
5550 //
5551 // ilock() is separate from iget() so that system calls can
5552 // get a long-term reference to an inode (as for an open file)
5553 // and only lock it for short periods (e.g., in read()).
5554 // The separation also helps avoid deadlock and races during
5555 // pathname lookup. iget() increments ip->ref so that the inode
5556 // stays cached and pointers to it remain valid.
5557 //
5558 // Many internal file system functions expect the caller to
5559 // have locked the inodes involved; this lets callers create
5560 // multi-step atomic operations.
5561
5562 struct {
5563   struct spinlock lock;
5564   struct inode inode[NINODE];
5565 } icache;
5566
5567 void
5568 iinit(int dev)
5569 {
5570   initlock(&icache.lock, "icache");
5571   readsb(dev, &sb);
5572   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d inodestart %
5573           sb.nblocks, sb.ninodes, sb.nlog, sb.logstart, sb.inodestart, sb.bma
5574 }
5575
5576 static struct inode* iget(uint dev, uint inum);
5577
5578
5579
5580
5581
5582
5583
5584
5585
5586
5587
5588
5589
5590
5591
5592
5593
5594
5595
5596
5597
5598
5599
```

```
5600 // Allocate a new inode with the given type on device dev.
5601 // A free inode has a type of zero.
5602 struct inode*
5603 ialloc(uint dev, short type)
5604 {
5605   int inum;
5606   struct buf *bp;
5607   struct dinode *dip;
5608
5609   for(inum = 1; inum < sb.ninodes; inum++){
5610     bp = bread(dev, IBLOCK(inum, sb));
5611     dip = (struct dinode*)bp->data + inum%IPB;
5612     if(dip->type == 0){  // a free inode
5613       memset(dip, 0, sizeof(*dip));
5614       dip->type = type;
5615       log_write(bp);   // mark it allocated on the disk
5616       brelse(bp);
5617       return iget(dev, inum);
5618     }
5619     brelse(bp);
5620   }
5621   panic("ialloc: no inodes");
5622 }
5623
5624 // Copy a modified in-memory inode to disk.
5625 void
5626 iupdate(struct inode *ip)
5627 {
5628   struct buf *bp;
5629   struct dinode *dip;
5630
5631   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5632   dip = (struct dinode*)bp->data + ip->inum%IPB;
5633   dip->type = ip->type;
5634   dip->major = ip->major;
5635   dip->minor = ip->minor;
5636   dip->nlink = ip->nlink;
5637   dip->size = ip->size;
5638   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5639   log_write(bp);
5640   brelse(bp);
5641 }
5642
5643
5644
5645
5646
5647
5648
5649
```



```
5650 // Find the inode with number inum on device dev
5651 // and return the in-memory copy. Does not lock
5652 // the inode and does not read it from disk.
5653 static struct inode*
5654 iget(uint dev, uint inum)
5655 {
5656   struct inode *ip, *empty;
5657
5658   acquire(&icache.lock);
5659
5660   // Is the inode already cached?
5661   empty = 0;
5662   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5663     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5664       ip->ref++;
5665       release(&icache.lock);
5666       return ip;
5667     }
5668     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
5669       empty = ip;
5670   }
5671
5672   // Recycle an inode cache entry.
5673   if(empty == 0)
5674     panic("iget: no inodes");
5675
5676   ip = empty;
5677   ip->dev = dev;
5678   ip->inum = inum;
5679   ip->ref = 1;
5680   ip->flags = 0;
5681   release(&icache.lock);
5682
5683   return ip;
5684 }
5685
5686 // Increment reference count for ip.
5687 // Returns ip to enable ip = idup(ip1) idiom.
5688 struct inode*
5689 idup(struct inode *ip)
5690 {
5691   acquire(&icache.lock);
5692   ip->ref++;
5693   release(&icache.lock);
5694   return ip;
5695 }
5696
5697
5698
5699
```

```
5700 // Lock the given inode.
5701 // Reads the inode from disk if necessary.
5702 void
5703 ilock(struct inode *ip)
5704 {
5705   struct buf *bp;
5706   struct dinode *dip;
5707
5708   if(ip == 0 || ip->ref < 1)
5709     panic("ilock");
5710
5711   acquire(&icache.lock);
5712   while(ip->flags & I_BUSY)
5713     sleep(ip, &icache.lock);
5714   ip->flags |= I_BUSY;
5715   release(&icache.lock);
5716
5717   if(!(ip->flags & I_VALID)){
5718     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5719     dip = (struct dinode*)bp->data + ip->inum%IPB;
5720     ip->type = dip->type;
5721     ip->major = dip->major;
5722     ip->minor = dip->minor;
5723     ip->nlink = dip->nlink;
5724     ip->size = dip->size;
5725     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5726     brelse(bp);
5727     ip->flags |= I_VALID;
5728     if(ip->type == 0)
5729       panic("ilock: no type");
5730   }
5731 }
5732
5733 // Unlock the given inode.
5734 void
5735 iunlock(struct inode *ip)
5736 {
5737   if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
5738     panic("iunlock");
5739
5740   acquire(&icache.lock);
5741   ip->flags &= ~I_BUSY;
5742   wakeup(ip);
5743   release(&icache.lock);
5744 }
5745
5746
5747
5748
5749
```



```
5750 // Drop a reference to an in-memory inode.
5751 // If that was the last reference, the inode cache entry can
5752 // be recycled.
5753 // If that was the last reference and the inode has no links
5754 // to it, free the inode (and its content) on disk.
5755 // All calls to iput() must be inside a transaction in
5756 // case it has to free the inode.
5757 void
5758 iput(struct inode *ip)
5759 {
5760   acquire(&icache.lock);
5761   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
5762     // inode has no links and no other references: truncate and free.
5763     if(ip->flags & I_BUSY)
5764       panic("iput busy");
5765     ip->flags |= I_BUSY;
5766     release(&icache.lock);
5767     itrunc(ip);
5768     ip->type = 0;
5769     iupdate(ip);
5770     acquire(&icache.lock);
5771     ip->flags = 0;
5772     wakeup(ip);
5773   }
5774   ip->ref--;
5775   release(&icache.lock);
5776 }
5777
5778 // Common idiom: unlock, then put.
5779 void
5780 iunlockput(struct inode *ip)
5781 {
5782   iunlock(ip);
5783   iput(ip);
5784 }
5785
5786
5787
5788
5789
5790
5791
5792
5793
5794
5795
5796
5797
5798
5799
```

```
5800 // Inode content
5801 //
5802 // The content (data) associated with each inode is stored
5803 // in blocks on the disk. The first NDIRECT block numbers
5804 // are listed in ip->addrs[].  The next NINDIRECT blocks are
5805 // listed in block ip->addrs[NDIRECT].
5806
5807 // Return the disk block address of the nth block in inode ip.
5808 // If there is no such block, bmap allocates one.
5809 static uint
5810 bmap(struct inode *ip, uint bn)
5811 {
5812   uint addr, *a;
5813   struct buf *bp;
5814
5815   if(bn < NDIRECT){
5816     if((addr = ip->addrs[bn]) == 0)
5817       ip->addrs[bn] = addr = balloc(ip->dev);
5818     return addr;
5819   }
5820   bn -= NDIRECT;
5821
5822   if(bn < NINDIRECT){
5823     // Load indirect block, allocating if necessary.
5824     if((addr = ip->addrs[NDIRECT]) == 0)
5825       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5826     bp = bread(ip->dev, addr);
5827     a = (uint*)bp->data;
5828     if((addr = a[bn]) == 0){
5829       a[bn] = addr = balloc(ip->dev);
5830       log_write(bp);
5831     }
5832     brelse(bp);
5833     return addr;
5834   }
5835
5836   panic("bmap: out of range");
5837 }
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```



```
5850 // Truncate inode (discard contents).
5851 // Only called when the inode has no links
5852 // to it (no directory entries referring to it)
5853 // and has no in-memory reference to it (is
5854 // not an open file or current directory).
5855 static void
5856 itrunc(struct inode *ip)
5857 {
5858   int i, j;
5859   struct buf *bp;
5860   uint *a;
5861
5862   for(i = 0; i < NDIRECT; i++){
5863     if(ip->addrs[i]){
5864       bfree(ip->dev, ip->addrs[i]);
5865       ip->addrs[i] = 0;
5866     }
5867   }
5868
5869   if(ip->addrs[NDIRECT]){
5870     bp = bread(ip->dev, ip->addrs[NDIRECT]);
5871     a = (uint*)bp->data;
5872     for(j = 0; j < NINDIRECT; j++){
5873       if(a[j])
5874         bfree(ip->dev, a[j]);
5875     }
5876     brelse(bp);
5877     bfree(ip->dev, ip->addrs[NDIRECT]);
5878     ip->addrs[NDIRECT] = 0;
5879   }
5880
5881   ip->size = 0;
5882   iupdate(ip);
5883 }
5884
5885 // Copy stat information from inode.
5886 void
5887 stati(struct inode *ip, struct stat *st)
5888 {
5889   st->dev = ip->dev;
5890   st->ino = ip->inum;
5891   st->type = ip->type;
5892   st->nlink = ip->nlink;
5893   st->size = ip->size;
5894 }
5895
5896
5897
5898
5899
```

```
5900 // Read data from inode.
5901 int
5902 readi(struct inode *ip, char *dst, uint off, uint n)
5903 {
5904   uint tot, m;
5905   struct buf *bp;
5906
5907   if(ip->type == T_DEV){
5908     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
5909       return -1;
5910     return devsw[ip->major].read(ip, dst, n);
5911   }
5912
5913   if(off > ip->size || off + n < off)
5914     return -1;
5915   if(off + n > ip->size)
5916     n = ip->size - off;
5917
5918   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5919     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5920     m = min(n - tot, BSIZE - off%BSIZE);
5921     memmove(dst, bp->data + off%BSIZE, m);
5922     brelse(bp);
5923   }
5924   return n;
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 // Write data to inode.
5951 int
5952 writei(struct inode *ip, char *src, uint off, uint n)
5953 {
5954   uint tot, m;
5955   struct buf *bp;
5956
5957   if(ip->type == T_DEV){
5958     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
5959       return -1;
5960     return devsw[ip->major].write(ip, src, n);
5961   }
5962
5963   if(off > ip->size || off + n < off)
5964     return -1;
5965   if(off + n > MAXFILE*BSIZE)
5966     return -1;
5967
5968   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
5969     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5970     m = min(n - tot, BSIZE - off%BSIZE);
5971     memmove(bp->data + off%BSIZE, src, m);
5972     log_write(bp);
5973     brelse(bp);
5974   }
5975
5976   if(n > 0 && off > ip->size){
5977     ip->size = off;
5978     iupdate(ip);
5979   }
5980   return n;
5981 }
5982
5983
5984
5985
5986
5987
5988
5989
5990
5991
5992
5993
5994
5995
5996
5997
5998
5999
```

```
6000 // Directories
6001
6002 int
6003 namecmp(const char *s, const char *t)
6004 {
6005   return strncmp(s, t, DIRSIZ);
6006 }
6007
6008 // Look for a directory entry in a directory.
6009 // If found, set *poff to byte offset of entry.
6010 struct inode*
6011 dirlookup(struct inode *dp, char *name, uint *poff)
6012 {
6013   uint off, inum;
6014   struct dirent de;
6015
6016   if(dp->type != T_DIR)
6017     panic("dirlookup not DIR");
6018
6019   for(off = 0; off < dp->size; off += sizeof(de)){
6020     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6021       panic("dirlink read");
6022     if(de.inum == 0)
6023       continue;
6024     if(namecmp(name, de.name) == 0){
6025       // entry matches path element
6026       if(poff)
6027         *poff = off;
6028       inum = de.inum;
6029       return iget(dp->dev, inum);
6030     }
6031   }
6032
6033   return 0;
6034 }
6035
6036
6037
6038
6039
6040
6041
6042
6043
6044
6045
6046
6047
6048
6049
```



```
6050 // Write a new directory entry (name, inum) into the directory dp.
6051 int
6052 dirlink(struct inode *dp, char *name, uint inum)
6053 {
6054   int off;
6055   struct dirent de;
6056   struct inode *ip;
6057
6058   // Check that name is not present.
6059   if((ip = dirlookup(dp, name, 0)) != 0){
6060     iput(ip);
6061     return -1;
6062   }
6063
6064   // Look for an empty dirent.
6065   for(off = 0; off < dp->size; off += sizeof(de)){
6066     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6067       panic("dirlink read");
6068     if(de.inum == 0)
6069       break;
6070   }
6071
6072   strncpy(de.name, name, DIRSIZ);
6073   de.inum = inum;
6074   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6075     panic("dirlink");
6076
6077   return 0;
6078 }
6079
6080
6081
6082
6083
6084
6085
6086
6087
6088
6089
6090
6091
6092
6093
6094
6095
6096
6097
6098
6099
```

```
6100 // Paths
6101
6102 // Copy the next path element from path into name.
6103 // Return a pointer to the element following the copied one.
6104 // The returned path has no leading slashes,
6105 // so the caller can check *path=='\0' to see if the name is the last one.
6106 // If no name to remove, return 0.
6107 //
6108 // Examples:
6109 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
6110 //   skipelem("///a//bb", name) = "bb", setting name = "a"
6111 //   skipelem("a", name) = "", setting name = "a"
6112 //   skipelem("", name) = skipelem("////", name) = 0
6113 //
6114 static char*
6115 skipelem(char *path, char *name)
6116 {
6117   char *s;
6118   int len;
6119
6120   while(*path == '/')
6121     path++;
6122   if(*path == 0)
6123     return 0;
6124   s = path;
6125   while(*path != '/' && *path != 0)
6126     path++;
6127   len = path - s;
6128   if(len >= DIRSIZ)
6129     memmove(name, s, DIRSIZ);
6130   else {
6131     memmove(name, s, len);
6132     name[len] = 0;
6133   }
6134   while(*path == '/')
6135     path++;
6136   return path;
6137 }
6138
6139
6140
6141
6142
6143
6144
6145
6146
6147
6148
6149
```



```
6150 // Look up and return the inode for a path name.
6151 // If parent != 0, return the inode for the parent and copy the final
6152 // path element into name, which must have room for DIRSIZ bytes.
6153 // Must be called inside a transaction since it calls iput().
6154 static struct inode*
6155 namex(char *path, int nameiparent, char *name)
6156 {
6157   struct inode *ip, *next;
6158
6159   if(*path == '/')
6160     ip = iget(ROOTDEV, ROOTINO);
6161   else
6162     ip = idup(proc->cwd);
6163
6164   while((path = skipelem(path, name)) != 0){
6165     ilock(ip);
6166     if(ip->type != T_DIR){
6167       iunlockput(ip);
6168       return 0;
6169     }
6170     if(nameiparent && *path == '\0'){
6171       // Stop one level early.
6172       iunlock(ip);
6173       return ip;
6174     }
6175     if((next = dirlookup(ip, name, 0)) == 0){
6176       iunlockput(ip);
6177       return 0;
6178     }
6179     iunlockput(ip);
6180     ip = next;
6181   }
6182   if(nameiparent){
6183     iput(ip);
6184     return 0;
6185   }
6186   return ip;
6187 }
6188
6189 struct inode*
6190 namei(char *path)
6191 {
6192   char name[DIRSIZ];
6193   return namex(path, 0, name);
6194 }
6195
6196
6197
6198
6199
```

```
6200 struct inode*
6201 nameiparent(char *path, char *name)
6202 {
6203   return namex(path, 1, name);
6204 }
6205
6206
6207
6208
6209
6210
6211
6212
6213
6214
6215
6216
6217
6218
6219
6220
6221
6222
6223
6224
6225
6226
6227
6228
6229
6230
6231
6232
6233
6234
6235
6236
6237
6238
6239
6240
6241
6242
6243
6244
6245
6246
6247
6248
6249
```



```
6250 //
6251 // File descriptors
6252 //
6253
6254 #include "types.h"
6255 #include "defs.h"
6256 #include "param.h"
6257 #include "fs.h"
6258 #include "file.h"
6259 #include "spinlock.h"
6260
6261 struct devsw devsw[NDEV];
6262 struct {
6263   struct spinlock lock;
6264   struct file file[NFILE];
6265 } ftable;
6266
6267 void
6268 fileinit(void)
6269 {
6270   initlock(&ftable.lock, "ftable");
6271 }
6272
6273 // Allocate a file structure.
6274 struct file*
6275 filealloc(void)
6276 {
6277   struct file *f;
6278
6279   acquire(&ftable.lock);
6280   for(f = ftable.file; f < ftable.file + NFILE; f++){
6281     if(f->ref == 0){
6282       f->ref = 1;
6283       release(&ftable.lock);
6284       return f;
6285     }
6286   }
6287   release(&ftable.lock);
6288   return 0;
6289 }
6290
6291
6292
6293
6294
6295
6296
6297
6298
6299
```

```
6300 // Increment ref count for file f.
6301 struct file*
6302 filedup(struct file *f)
6303 {
6304   acquire(&ftable.lock);
6305   if(f->ref < 1)
6306     panic("filedup");
6307   f->ref++;
6308   release(&ftable.lock);
6309   return f;
6310 }
6311
6312 // Close file f.  (Decrement ref count, close when reaches 0.)
6313 void
6314 fileclose(struct file *f)
6315 {
6316   struct file ff;
6317
6318   acquire(&ftable.lock);
6319   if(f->ref < 1)
6320     panic("fileclose");
6321   if(--f->ref > 0){
6322     release(&ftable.lock);
6323     return;
6324   }
6325   ff = *f;
6326   f->ref = 0;
6327   f->type = FD_NONE;
6328   release(&ftable.lock);
6329
6330   if(ff.type == FD_PIPE)
6331     pipeclose(ff.pipe, ff.writable);
6332   else if(ff.type == FD_INODE){
6333     begin_op();
6334     iput(ff.ip);
6335     end_op();
6336   }
6337 }
6338
6339
6340
6341
6342
6343
6344
6345
6346
6347
6348
6349
```



```
6350 // Get metadata about file f.
6351 int
6352 filestat(struct file *f, struct stat *st)
6353 {
6354   if(f->type == FD_INODE){
6355     ilock(f->ip);
6356     stati(f->ip, st);
6357     iunlock(f->ip);
6358     return 0;
6359   }
6360   return -1;
6361 }
6362
6363 // Read from file f.
6364 int
6365 fileread(struct file *f, char *addr, int n)
6366 {
6367   int r;
6368
6369   if(f->readable == 0)
6370     return -1;
6371   if(f->type == FD_PIPE)
6372     return piperead(f->pipe, addr, n);
6373   if(f->type == FD_INODE){
6374     ilock(f->ip);
6375     if((r = readi(f->ip, addr, f->off, n)) > 0)
6376       f->off += r;
6377     iunlock(f->ip);
6378     return r;
6379   }
6380   panic("fileread");
6381 }
6382
6383
6384
6385
6386
6387
6388
6389
6390
6391
6392
6393
6394
6395
6396
6397
6398
6399
```

```
6400 // Write to file f.
6401 int
6402 filewrite(struct file *f, char *addr, int n)
6403 {
6404   int r;
6405
6406   if(f->writable == 0)
6407     return -1;
6408   if(f->type == FD_PIPE)
6409     return pipewrite(f->pipe, addr, n);
6410   if(f->type == FD_INODE){
6411     // write a few blocks at a time to avoid exceeding
6412     // the maximum log transaction size, including
6413     // i-node, indirect block, allocation blocks,
6414     // and 2 blocks of slop for non-aligned writes.
6415     // this really belongs lower down, since writei()
6416     // might be writing a device like the console.
6417     int max = ((LOGSIZE-1-1-2) / 2) * 512;
6418     int i = 0;
6419     while(i < n){
6420       int n1 = n - i;
6421       if(n1 > max)
6422         n1 = max;
6423
6424       begin_op();
6425       ilock(f->ip);
6426       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
6427         f->off += r;
6428       iunlock(f->ip);
6429       end_op();
6430
6431       if(r < 0)
6432         break;
6433       if(r != n1)
6434         panic("short filewrite");
6435       i += r;
6436     }
6437     return i == n ? n : -1;
6438   }
6439   panic("filewrite");
6440 }
6441
6442
6443
6444
6445
6446
6447
6448
6449
```



```
6450 //
6451 // File-system system calls.
6452 // Mostly argument checking, since we don't trust
6453 // user code, and calls into file.c and fs.c.
6454 //
6455
6456 #include "types.h"
6457 #include "defs.h"
6458 #include "param.h"
6459 #include "stat.h"
6460 #include "mmu.h"
6461 #include "proc.h"
6462 #include "fs.h"
6463 #include "file.h"
6464 #include "fcntl.h"
6465
6466 // Fetch the nth word-sized system call argument as a file descriptor
6467 // and return both the descriptor and the corresponding struct file.
6468 static int
6469 argfd(int n, int *pfd, struct file **pf)
6470 {
6471   int fd;
6472   struct file *f;
6473
6474   if(argint(n, &fd) < 0)
6475     return -1;
6476   if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
6477     return -1;
6478   if(pfd)
6479     *pfd = fd;
6480   if(pf)
6481     *pf = f;
6482   return 0;
6483 }
6484
6485 // Allocate a file descriptor for the given file.
6486 // Takes over file reference from caller on success.
6487 static int
6488 fdalloc(struct file *f)
6489 {
6490   int fd;
6491
6492   for(fd = 0; fd < NOFILE; fd++){
6493     if(proc->ofile[fd] == 0){
6494       proc->ofile[fd] = f;
6495       return fd;
6496     }
6497   }
6498   return -1;
6499 }
```

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



```
6550 int
6551 sys_fstat(void)
6552 {
6553   struct file *f;
6554   struct stat *st;
6555
6556   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
6557     return -1;
6558   return filestat(f, st);
6559 }
6560
6561 // Create the path new as a link to the same inode as old.
6562 int
6563 sys_link(void)
6564 {
6565   char name[DIRSIZ], *new, *old;
6566   struct inode *dp, *ip;
6567
6568   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
6569     return -1;
6570
6571   begin_op();
6572   if((ip = namei(old)) == 0){
6573     end_op();
6574     return -1;
6575   }
6576
6577   ilock(ip);
6578   if(ip->type == T_DIR){
6579     iunlockput(ip);
6580     end_op();
6581     return -1;
6582   }
6583
6584   ip->nlink++;
6585   iupdate(ip);
6586   iunlock(ip);
6587
6588   if((dp = nameiparent(new, name)) == 0)
6589     goto bad;
6590   ilock(dp);
6591   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
6592     iunlockput(dp);
6593     goto bad;
6594   }
6595   iunlockput(dp);
6596   iput(ip);
6597
6598   end_op();
6599
```

```
6600   return 0;
6601
6602 bad:
6603   ilock(ip);
6604   ip->nlink--;
6605   iupdate(ip);
6606   iunlockput(ip);
6607   end_op();
6608   return -1;
6609 }
6610
6611 // Is the directory dp empty except for "." and ".." ?
6612 static int
6613 isdirempty(struct inode *dp)
6614 {
6615   int off;
6616   struct dirent de;
6617
6618   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
6619     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6620       panic("isdirempty: readi");
6621     if(de.inum != 0)
6622       return 0;
6623   }
6624   return 1;
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 int
6651 sys_unlink(void)
6652 {
6653   struct inode *ip, *dp;
6654   struct dirent de;
6655   char name[DIRSIZ], *path;
6656   uint off;
6657
6658   if(argstr(0, &path) < 0)
6659     return -1;
6660
6661   begin_op();
6662   if((dp = nameiparent(path, name)) == 0){
6663     end_op();
6664     return -1;
6665   }
6666
6667   ilock(dp);
6668
6669   // Cannot unlink "." or "..".
6670   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6671     goto bad;
6672
6673   if((ip = dirlookup(dp, name, &off)) == 0)
6674     goto bad;
6675   ilock(ip);
6676
6677   if(ip->nlink < 1)
6678     panic("unlink: nlink < 1");
6679   if(ip->type == T_DIR && !isdirempty(ip)){
6680     iunlockput(ip);
6681     goto bad;
6682   }
6683
6684   memset(&de, 0, sizeof(de));
6685   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6686     panic("unlink: writei");
6687   if(ip->type == T_DIR){
6688     dp->nlink--;
6689     iupdate(dp);
6690   }
6691   iunlockput(dp);
6692
6693   ip->nlink--;
6694   iupdate(ip);
6695   iunlockput(ip);
6696
6697   end_op();
6698
6699   return 0;
```

```
6700 bad:
6701   iunlockput(dp);
6702   end_op();
6703   return -1;
6704 }
6705
6706 static struct inode*
6707 create(char *path, short type, short major, short minor)
6708 {
6709   uint off;
6710   struct inode *ip, *dp;
6711   char name[DIRSIZ];
6712
6713   if((dp = nameiparent(path, name)) == 0)
6714     return 0;
6715   ilock(dp);
6716
6717   if((ip = dirlookup(dp, name, &off)) != 0){
6718     iunlockput(dp);
6719     ilock(ip);
6720     if(type == T_FILE && ip->type == T_FILE)
6721       return ip;
6722     iunlockput(ip);
6723     return 0;
6724   }
6725
6726   if((ip = ialloc(dp->dev, type)) == 0)
6727     panic("create: ialloc");
6728
6729   ilock(ip);
6730   ip->major = major;
6731   ip->minor = minor;
6732   ip->nlink = 1;
6733   iupdate(ip);
6734
6735   if(type == T_DIR){  // Create . and .. entries.
6736     dp->nlink++;  // for ".."
6737     iupdate(dp);
6738     // No ip->nlink++ for ".": avoid cyclic ref count.
6739     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6740       panic("create dots");
6741   }
6742
6743   if(dirlink(dp, name, ip->inum) < 0)
6744     panic("create: dirlink");
6745
6746   iunlockput(dp);
6747
6748   return ip;
6749 }
```



```
6750 int
6751 sys_open(void)
6752 {
6753   char *path;
6754   int fd, omode;
6755   struct file *f;
6756   struct inode *ip;
6757
6758   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6759     return -1;
6760
6761   begin_op();
6762
6763   if(omode & O_CREATE){
6764     ip = create(path, T_FILE, 0, 0);
6765     if(ip == 0){
6766       end_op();
6767       return -1;
6768     }
6769   } else {
6770     if((ip = namei(path)) == 0){
6771       end_op();
6772       return -1;
6773     }
6774     ilock(ip);
6775     if(ip->type == T_DIR && omode != O_RDONLY){
6776       iunlockput(ip);
6777       end_op();
6778       return -1;
6779     }
6780   }
6781
6782   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6783     if(f)
6784       fileclose(f);
6785     iunlockput(ip);
6786     end_op();
6787     return -1;
6788   }
6789   iunlock(ip);
6790   end_op();
6791
6792   f->type = FD_INODE;
6793   f->ip = ip;
6794   f->off = 0;
6795   f->readable = !(omode & O_WRONLY);
6796   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6797   return fd;
6798 }
6799
```

```
6800 int
6801 sys_mkdir(void)
6802 {
6803   char *path;
6804   struct inode *ip;
6805
6806   begin_op();
6807   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
6808     end_op();
6809     return -1;
6810   }
6811   iunlockput(ip);
6812   end_op();
6813   return 0;
6814 }
6815
6816 int
6817 sys_mknod(void)
6818 {
6819   struct inode *ip;
6820   char *path;
6821   int len;
6822   int major, minor;
6823
6824   begin_op();
6825   if((len=argstr(0, &path)) < 0 ||
6826      argint(1, &major) < 0 ||
6827      argint(2, &minor) < 0 ||
6828      (ip = create(path, T_DEV, major, minor)) == 0){
6829     end_op();
6830     return -1;
6831   }
6832   iunlockput(ip);
6833   end_op();
6834   return 0;
6835 }
6836
6837
6838
6839
6840
6841
6842
6843
6844
6845
6846
6847
6848
6849
```



```
6850 int
6851 sys_chdir(void)
6852 {
6853   char *path;
6854   struct inode *ip;
6855
6856   begin_op();
6857   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
6858     end_op();
6859     return -1;
6860   }
6861   ilock(ip);
6862   if(ip->type != T_DIR){
6863     iunlockput(ip);
6864     end_op();
6865     return -1;
6866   }
6867   iunlock(ip);
6868   iput(proc->cwd);
6869   end_op();
6870   proc->cwd = ip;
6871   return 0;
6872 }
6873
6874 int
6875 sys_exec(void)
6876 {
6877   char *path, *argv[MAXARG];
6878   int i;
6879   uint uargv, uarg;
6880
6881   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6882     return -1;
6883   }
6884   memset(argv, 0, sizeof(argv));
6885   for(i=0;; i++){
6886     if(i >= NELEM(argv))
6887       return -1;
6888     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6889       return -1;
6890     if(uarg == 0){
6891       argv[i] = 0;
6892       break;
6893     }
6894     if(fetchstr(uarg, &argv[i]) < 0)
6895       return -1;
6896   }
6897   return exec(path, argv);
6898 }
6899
```

```
6900 int
6901 sys_pipe(void)
6902 {
6903   int *fd;
6904   struct file *rf, *wf;
6905   int fd0, fd1;
6906
6907   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6908     return -1;
6909   if(pipealloc(&rf, &wf) < 0)
6910     return -1;
6911   fd0 = -1;
6912   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6913     if(fd0 >= 0)
6914       proc->ofile[fd0] = 0;
6915     fileclose(rf);
6916     fileclose(wf);
6917     return -1;
6918   }
6919   fd[0] = fd0;
6920   fd[1] = fd1;
6921   return 0;
6922 }
6923
6924
6925
6926
6927
6928
6929
6930
6931
6932
6933
6934
6935
6936
6937
6938
6939
6940
6941
6942
6943
6944
6945
6946
6947
6948
6949
```



```
6950 #include "types.h"
6951 #include "param.h"
6952 #include "memlayout.h"
6953 #include "mmu.h"
6954 #include "proc.h"
6955 #include "defs.h"
6956 #include "x86.h"
6957 #include "elf.h"
6958
6959 int
6960 exec(char *path, char **argv)
6961 {
6962   char *s, *last;
6963   int i, off;
6964   uint argc, sz, sp, ustack[3+MAXARG+1];
6965   struct elfhdr elf;
6966   struct inode *ip;
6967   struct proghdr ph;
6968   pde_t *pgdir, *oldpgdir;
6969
6970   begin_op();
6971   if((ip = namei(path)) == 0){
6972     end_op();
6973     return -1;
6974   }
6975   ilock(ip);
6976   pgdir = 0;
6977
6978   // Check ELF header
6979   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
6980     goto bad;
6981   if(elf.magic != ELF_MAGIC)
6982     goto bad;
6983
6984   if((pgdir = setupkvm()) == 0)
6985     goto bad;
6986
6987   // Load program into memory.
6988   sz = 0;
6989   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6990     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6991       goto bad;
6992     if(ph.type != ELF_PROG_LOAD)
6993       continue;
6994     if(ph.memsz < ph.filesz)
6995       goto bad;
6996     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6997       goto bad;
6998     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6999       goto bad;
```

```
7000   }
7001   iunlockput(ip);
7002   end_op();
7003   ip = 0;
7004
7005   // Allocate two pages at the next page boundary.
7006   // Make the first inaccessible.  Use the second as the user stack.
7007   sz = PGROUNDUP(sz);
7008   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
7009     goto bad;
7010   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
7011   sp = sz;
7012
7013   // Push argument strings, prepare rest of stack in ustack.
7014   for(argc = 0; argv[argc]; argc++) {
7015     if(argc >= MAXARG)
7016       goto bad;
7017     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
7018     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
7019       goto bad;
7020     ustack[3+argc] = sp;
7021   }
7022   ustack[3+argc] = 0;
7023
7024   ustack[0] = 0xffffffff;  // fake return PC
7025   ustack[1] = argc;
7026   ustack[2] = sp - (argc+1)*4;  // argv pointer
7027
7028   sp -= (3+argc+1) * 4;
7029   if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
7030     goto bad;
7031
7032   // Save program name for debugging.
7033   for(last=s=path; *s; s++)
7034     if(*s == '/')
7035       last = s+1;
7036   safestrcpy(proc->name, last, sizeof(proc->name));
7037
7038   // Commit to the user image.
7039   oldpgdir = proc->pgdir;
7040   proc->pgdir = pgdir;
7041   proc->sz = sz;
7042   proc->tf->eip = elf.entry;  // main
7043   proc->tf->esp = sp;
7044   switchuvm(proc);
7045   freevm(oldpgdir);
7046   return 0;
7047
7048
7049
```



```
7050  bad:
7051    if(pgdir)
7052      freevm(pgdir);
7053    if(ip){
7054      iunlockput(ip);
7055      end_op();
7056    }
7057    return -1;
7058  }
7059
7060
7061
7062
7063
7064
7065
7066
7067
7068
7069
7070
7071
7072
7073
7074
7075
7076
7077
7078
7079
7080
7081
7082
7083
7084
7085
7086
7087
7088
7089
7090
7091
7092
7093
7094
7095
7096
7097
7098
7099
```

```
7100 #include "types.h"
7101 #include "defs.h"
7102 #include "param.h"
7103 #include "mmu.h"
7104 #include "proc.h"
7105 #include "fs.h"
7106 #include "file.h"
7107 #include "spinlock.h"
7108
7109 #define PIPESIZE 512
7110
7111 struct pipe {
7112   struct spinlock lock;
7113   char data[PIPESIZE];
7114   uint nread;     // number of bytes read
7115   uint nwrite;    // number of bytes written
7116   int readopen;   // read fd is still open
7117   int writeopen;  // write fd is still open
7118 };
7119
7120 int
7121 pipealloc(struct file **f0, struct file **f1)
7122 {
7123   struct pipe *p;
7124
7125   p = 0;
7126   *f0 = *f1 = 0;
7127   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
7128     goto bad;
7129   if((p = (struct pipe*)kalloc()) == 0)
7130     goto bad;
7131   p->readopen = 1;
7132   p->writeopen = 1;
7133   p->nwrite = 0;
7134   p->nread = 0;
7135   initlock(&p->lock, "pipe");
7136   (*f0)->type = FD_PIPE;
7137   (*f0)->readable = 1;
7138   (*f0)->writable = 0;
7139   (*f0)->pipe = p;
7140   (*f1)->type = FD_PIPE;
7141   (*f1)->readable = 0;
7142   (*f1)->writable = 1;
7143   (*f1)->pipe = p;
7144   return 0;
7145
7146
7147
7148
7149
```



```
7150  bad:
7151   if(p)
7152     kfree((char*)p);
7153   if(*f0)
7154     fileclose(*f0);
7155   if(*f1)
7156     fileclose(*f1);
7157   return -1;
7158 }
7159
7160 void
7161 pipeclose(struct pipe *p, int writable)
7162 {
7163   acquire(&p->lock);
7164   if(writable){
7165     p->writeopen = 0;
7166     wakeup(&p->nread);
7167   } else {
7168     p->readopen = 0;
7169     wakeup(&p->nwrite);
7170   }
7171   if(p->readopen == 0 && p->writeopen == 0){
7172     release(&p->lock);
7173     kfree((char*)p);
7174   } else
7175     release(&p->lock);
7176 }
7177
7178
7179 int
7180 pipewrite(struct pipe *p, char *addr, int n)
7181 {
7182   int i;
7183
7184   acquire(&p->lock);
7185   for(i = 0; i < n; i++){
7186     while(p->nwrite == p->nread + PIPESIZE){
7187       if(p->readopen == 0 || proc->killed){
7188         release(&p->lock);
7189         return -1;
7190       }
7191       wakeup(&p->nread);
7192       sleep(&p->nwrite, &p->lock);
7193     }
7194     p->data[p->nwrite++ % PIPESIZE] = addr[i];
7195   }
7196   wakeup(&p->nread);
7197   release(&p->lock);
7198   return n;
7199 }
```

```
7200 int
7201 piperead(struct pipe *p, char *addr, int n)
7202 {
7203   int i;
7204
7205   acquire(&p->lock);
7206   while(p->nread == p->nwrite && p->writeopen){
7207     if(proc->killed){
7208       release(&p->lock);
7209       return -1;
7210     }
7211     sleep(&p->nread, &p->lock);
7212   }
7213   for(i = 0; i < n; i++){
7214     if(p->nread == p->nwrite)
7215       break;
7216     addr[i] = p->data[p->nread++ % PIPESIZE];
7217   }
7218   wakeup(&p->nwrite);
7219   release(&p->lock);
7220   return i;
7221 }
7222
7223
7224
7225
7226
7227
7228
7229
7230
7231
7232
7233
7234
7235
7236
7237
7238
7239
7240
7241
7242
7243
7244
7245
7246
7247
7248
7249
```



```
7250 #include "types.h"
7251 #include "x86.h"
7252
7253 void*
7254 memset(void *dst, int c, uint n)
7255 {
7256   if ((int)dst%4 == 0 && n%4 == 0){
7257     c &= 0xFF;
7258     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
7259   } else
7260     stosb(dst, c, n);
7261   return dst;
7262 }
7263
7264 int
7265 memcmp(const void *v1, const void *v2, uint n)
7266 {
7267   const uchar *s1, *s2;
7268
7269   s1 = v1;
7270   s2 = v2;
7271   while(n-- > 0){
7272     if(*s1 != *s2)
7273       return *s1 - *s2;
7274     s1++, s2++;
7275   }
7276
7277   return 0;
7278 }
7279
7280 void*
7281 memmove(void *dst, const void *src, uint n)
7282 {
7283   const char *s;
7284   char *d;
7285
7286   s = src;
7287   d = dst;
7288   if(s < d && s + n > d){
7289     s += n;
7290     d += n;
7291     while(n-- > 0)
7292       *--d = *--s;
7293   } else
7294     while(n-- > 0)
7295       *d++ = *s++;
7296
7297   return dst;
7298 }
7299
```

```
7300 // memcpy exists to placate GCC.  Use memmove.
7301 void*
7302 memcpy(void *dst, const void *src, uint n)
7303 {
7304   return memmove(dst, src, n);
7305 }
7306
7307 int
7308 strncmp(const char *p, const char *q, uint n)
7309 {
7310   while(n > 0 && *p && *p == *q)
7311     n--, p++, q++;
7312   if(n == 0)
7313     return 0;
7314   return (uchar)*p - (uchar)*q;
7315 }
7316
7317 char*
7318 strncpy(char *s, const char *t, int n)
7319 {
7320   char *os;
7321
7322   os = s;
7323   while(n-- > 0 && (*s++ = *t++) != 0)
7324     ;
7325   while(n-- > 0)
7326     *s++ = 0;
7327   return os;
7328 }
7329
7330 // Like strncpy but guaranteed to NUL-terminate.
7331 char*
7332 safestrcpy(char *s, const char *t, int n)
7333 {
7334   char *os;
7335
7336   os = s;
7337   if(n <= 0)
7338     return os;
7339   while(--n > 0 && (*s++ = *t++) != 0)
7340     ;
7341   *s = 0;
7342   return os;
7343 }
7344
7345
7346
7347
7348
7349
```



```
7350 int
7351 strlen(const char *s)
7352 {
7353   int n;
7354
7355   for(n = 0; s[n]; n++)
7356     ;
7357   return n;
7358 }
7359
7360
7361
7362
7363
7364
7365
7366
7367
7368
7369
7370
7371
7372
7373
7374
7375
7376
7377
7378
7379
7380
7381
7382
7383
7384
7385
7386
7387
7388
7389
7390
7391
7392
7393
7394
7395
7396
7397
7398
7399
```

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

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

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

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

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



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

```
7500 // Blank page.
7501
7502
7503
7504
7505
7506
7507
7508
7509
7510
7511
7512
7513
7514
7515
7516
7517
7518
7519
7520
7521
7522
7523
7524
7525
7526
7527
7528
7529
7530
7531
7532
7533
7534
7535
7536
7537
7538
7539
7540
7541
7542
7543
7544
7545
7546
7547
7548
7549
```



```
7550 // Multiprocessor support
7551 // Search memory for MP description structures.
7552 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
7553
7554 #include "types.h"
7555 #include "defs.h"
7556 #include "param.h"
7557 #include "memlayout.h"
7558 #include "mp.h"
7559 #include "x86.h"
7560 #include "mmu.h"
7561 #include "proc.h"
7562
7563 struct cpu cpus[NCPU];
7564 static struct cpu *bcpu;
7565 int ismp;
7566 int ncpu;
7567 uchar ioapicid;
7568
7569 int
7570 mpbcpu(void)
7571 {
7572   return bcpu-cpus;
7573 }
7574
7575 static uchar
7576 sum(uchar *addr, int len)
7577 {
7578   int i, sum;
7579
7580   sum = 0;
7581   for(i=0; i<len; i++)
7582     sum += addr[i];
7583   return sum;
7584 }
7585
7586 // Look for an MP structure in the len bytes at addr.
7587 static struct mp*
7588 mpsearch1(uint a, int len)
7589 {
7590   uchar *e, *p, *addr;
7591
7592   addr = p2v(a);
7593   e = addr+len;
7594   for(p = addr; p < e; p += sizeof(struct mp))
7595     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
7596       return (struct mp*)p;
7597   return 0;
7598 }
7599
```

```
7600 // Search for the MP Floating Pointer Structure, which according to the
7601 // spec is in one of the following three locations:
7602 // 1) in the first KB of the EBDA;
7603 // 2) in the last KB of system base memory;
7604 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
7605 static struct mp*
7606 mpsearch(void)
7607 {
7608   uchar *bda;
7609   uint p;
7610   struct mp *mp;
7611
7612   bda = (uchar *) P2V(0x400);
7613   if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
7614     if((mp = mpsearch1(p, 1024)))
7615       return mp;
7616   } else {
7617     p = ((bda[0x14]<<8)|bda[0x13])*1024;
7618     if((mp = mpsearch1(p-1024, 1024)))
7619       return mp;
7620   }
7621   return mpsearch1(0xF0000, 0x10000);
7622 }
7623
7624 // Search for an MP configuration table.  For now,
7625 // don't accept the default configurations (physaddr == 0).
7626 // Check for correct signature, calculate the checksum and,
7627 // if correct, check the version.
7628 // To do: check extended table checksum.
7629 static struct mpconf*
7630 mpconfig(struct mp **pmp)
7631 {
7632   struct mpconf *conf;
7633   struct mp *mp;
7634
7635   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
7636     return 0;
7637   conf = (struct mpconf*) p2v((uint) mp->physaddr);
7638   if(memcmp(conf, "PCMP", 4) != 0)
7639     return 0;
7640   if(conf->version != 1 && conf->version != 4)
7641     return 0;
7642   if(sum((uchar*)conf, conf->length) != 0)
7643     return 0;
7644   *pmp = mp;
7645   return conf;
7646 }
7647
7648
7649
```



```
7650 void
7651 mpinit(void)
7652 {
7653   uchar *p, *e;
7654   struct mp *mp;
7655   struct mpconf *conf;
7656   struct mpproc *proc;
7657   struct mpioapic *ioapic;
7658
7659   bcpu = &cpus[0];
7660   if((conf = mpconfig(&mp)) == 0)
7661     return;
7662   ismp = 1;
7663   lapic = (uint*)conf->lapicaddr;
7664   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
7665     switch(*p){
7666     case MPPROC:
7667       proc = (struct mpproc*)p;
7668       if(ncpu != proc->apicid){
7669         cprintf("mpinit: ncpu=%d apicid=%d\n", ncpu, proc->apicid);
7670         ismp = 0;
7671       }
7672       if(proc->flags & MPBOOT)
7673         bcpu = &cpus[ncpu];
7674       cpus[ncpu].id = ncpu;
7675       ncpu++;
7676       p += sizeof(struct mpproc);
7677       continue;
7678     case MPIOAPIC:
7679       ioapic = (struct mpioapic*)p;
7680       ioapicid = ioapic->apicno;
7681       p += sizeof(struct mpioapic);
7682       continue;
7683     case MPBUS:
7684     case MPIOINTR:
7685     case MPLINTR:
7686       p += 8;
7687       continue;
7688     default:
7689       cprintf("mpinit: unknown config type %x\n", *p);
7690       ismp = 0;
7691     }
7692   }
7693   if(!ismp){
7694     // Didn't like what we found; fall back to no MP.
7695     ncpu = 1;
7696     lapic = 0;
7697     ioapicid = 0;
7698     return;
7699   }
```

```
7700   if(mp->imcrp){
7701     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
7702     // But it would on real hardware.
7703     outb(0x22, 0x70);   // Select IMCR
7704     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
7705   }
7706 }
7707
7708
7709
7710
7711
7712
7713
7714
7715
7716
7717
7718
7719
7720
7721
7722
7723
7724
7725
7726
7727
7728
7729
7730
7731
7732
7733
7734
7735
7736
7737
7738
7739
7740
7741
7742
7743
7744
7745
7746
7747
7748
7749
```



```
7750 // The local APIC manages internal (non-I/O) interrupts.
7751 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
7752
7753 #include "types.h"
7754 #include "defs.h"
7755 #include "date.h"
7756 #include "memlayout.h"
7757 #include "traps.h"
7758 #include "mmu.h"
7759 #include "x86.h"
7760
7761 // Local APIC registers, divided by 4 for use as uint[] indices.
7762 #define ID      (0x0020/4)   // ID
7763 #define VER     (0x0030/4)   // Version
7764 #define TPR     (0x0080/4)   // Task Priority
7765 #define EOI     (0x00B0/4)   // EOI
7766 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
7767   #define ENABLE     0x00000100   // Unit Enable
7768 #define ESR     (0x0280/4)   // Error Status
7769 #define ICRLO   (0x0300/4)   // Interrupt Command
7770   #define INIT       0x00000500   // INIT/RESET
7771   #define STARTUP    0x00000600   // Startup IPI
7772   #define DELIVS     0x00001000   // Delivery status
7773   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
7774   #define DEASSERT   0x00000000
7775   #define LEVEL      0x00008000   // Level triggered
7776   #define BCAST      0x00080000   // Send to all APICs, including self.
7777   #define BUSY       0x00001000
7778   #define FIXED      0x00000000
7779 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
7780 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
7781   #define X1         0x0000000B   // divide counts by 1
7782   #define PERIODIC   0x00020000   // Periodic
7783 #define PCINT   (0x0340/4)   // Performance Counter LVT
7784 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
7785 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
7786 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
7787   #define MASKED     0x00010000   // Interrupt masked
7788 #define TICR    (0x0380/4)   // Timer Initial Count
7789 #define TCCR    (0x0390/4)   // Timer Current Count
7790 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
7791
7792 volatile uint *lapic;  // Initialized in mp.c
7793
7794 static void
7795 lapicw(int index, int value)
7796 {
7797   lapic[index] = value;
7798   lapic[ID];  // wait for write to finish, by reading
7799 }
```

```
7850 void
7851 lapicinit(void)
7852 {
7853   if(!lapic)
7854     return;
7855
7856   // Enable local APIC; set spurious interrupt vector.
7857   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
7858
7859   // The timer repeatedly counts down at bus frequency
7860   // from lapic[TICR] and then issues an interrupt.
7861   // If xv6 cared more about precise timekeeping,
7862   // TICR would be calibrated using an external time source.
7863   lapicw(TDCR, X1);
7864   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
7865   lapicw(TICR, 10000000);
7866
7867   // Disable logical interrupt lines.
7868   lapicw(LINT0, MASKED);
7869   lapicw(LINT1, MASKED);
7870
7871   // Disable performance counter overflow interrupts
7872   // on machines that provide that interrupt entry.
7873   if(((lapic[VER]>>16) & 0xFF) >= 4)
7874     lapicw(PCINT, MASKED);
7875
7876   // Map error interrupt to IRQ_ERROR.
7877   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
7878
7879   // Clear error status register (requires back-to-back writes).
7880   lapicw(ESR, 0);
7881   lapicw(ESR, 0);
7882
7883   // Ack any outstanding interrupts.
7884   lapicw(EOI, 0);
7885
7886   // Send an Init Level De-Assert to synchronise arbitration ID's.
7887   lapicw(ICRHI, 0);
7888   lapicw(ICRLO, BCAST | INIT | LEVEL);
7889   while(lapic[ICRLO] & DELIVS)
7890     ;
7891
7892   // Enable interrupts on the APIC (but not on the processor).
7893   lapicw(TPR, 0);
7894 }
7895
7896
7897
7898
7899
```

```
7900 int
7901 cpunum(void)
7902 {
7903   // Cannot call cpu when interrupts are enabled:
7904   // result not guaranteed to last long enough to be used!
7905   // Would prefer to panic but even printing is chancy here:
7906   // almost everything, including cprintf and panic, calls cpu,
7907   // often indirectly through acquire and release.
7908   if(readeflags()&FL_IF){
7909     static int n;
7910     if(n++ == 0)
7911       cprintf("cpu called from %x with interrupts enabled\n",
7912         __builtin_return_address(0));
7913   }
7914
7915   if(lapic)
7916     return lapic[ID]>>24;
7917   return 0;
7918 }
7919
7920 // Acknowledge interrupt.
7921 void
7922 lapiceoi(void)
7923 {
7924   if(lapic)
7925     lapicw(EOI, 0);
7926 }
7927
7928 // Spin for a given number of microseconds.
7929 // On real hardware would want to tune this dynamically.
7930 void
7931 microdelay(int us)
7932 {
7933 }
7934
7935 #define CMOS_PORT    0x70
7936 #define CMOS_RETURN  0x71
7937
7938 // Start additional processor running entry code at addr.
7939 // See Appendix B of MultiProcessor Specification.
7940 void
7941 lapicstartap(uchar apicid, uint addr)
7942 {
7943   int i;
7944   ushort *wrv;
7945
7946   // "The BSP must initialize CMOS shutdown code to 0AH
7947   // and the warm reset vector (DWORD based at 40:67) to point at
7948   // the AP startup code prior to the [universal startup algorithm]."
7949   outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
```



```
7950   outb(CMOS_PORT+1, 0x0A);
7951   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
7952   wrv[0] = 0;
7953   wrv[1] = addr >> 4;
7954
7955   // "Universal startup algorithm."
7956   // Send INIT (level-triggered) interrupt to reset other CPU.
7957   lapicw(ICRHI, apicid<<24);
7958   lapicw(ICRLO, INIT | LEVEL | ASSERT);
7959   microdelay(200);
7960   lapicw(ICRLO, INIT | LEVEL);
7961   microdelay(100);    // should be 10ms, but too slow in Bochs!
7962
7963   // Send startup IPI (twice!) to enter code.
7964   // Regular hardware is supposed to only accept a STARTUP
7965   // when it is in the halted state due to an INIT.  So the second
7966   // should be ignored, but it is part of the official Intel algorithm.
7967   // Bochs complains about the second one.  Too bad for Bochs.
7968   for(i = 0; i < 2; i++){
7969     lapicw(ICRHI, apicid<<24);
7970     lapicw(ICRLO, STARTUP | (addr>>12));
7971     microdelay(200);
7972   }
7973 }
7974
7975 #define CMOS_STATA   0x0a
7976 #define CMOS_STATB   0x0b
7977 #define CMOS_UIP    (1 << 7)        // RTC update in progress
7978
7979 #define SECS    0x00
7980 #define MINS    0x02
7981 #define HOURS   0x04
7982 #define DAY     0x07
7983 #define MONTH   0x08
7984 #define YEAR    0x09
7985
7986 static uint cmos_read(uint reg)
7987 {
7988   outb(CMOS_PORT,  reg);
7989   microdelay(200);
7990
7991   return inb(CMOS_RETURN);
7992 }
7993
7994
7995
7996
7997
7998
7999
```

```
8000 static void fill_rtcdate(struct rtcdate *r)
8001 {
8002   r->second = cmos_read(SECS);
8003   r->minute = cmos_read(MINS);
8004   r->hour   = cmos_read(HOURS);
8005   r->day    = cmos_read(DAY);
8006   r->month  = cmos_read(MONTH);
8007   r->year   = cmos_read(YEAR);
8008 }
8009
8010 // qemu seems to use 24-hour GWT and the values are BCD encoded
8011 void cmostime(struct rtcdate *r)
8012 {
8013   struct rtcdate t1, t2;
8014   int sb, bcd;
8015
8016   sb = cmos_read(CMOS_STATB);
8017
8018   bcd = (sb & (1 << 2)) == 0;
8019
8020   // make sure CMOS doesn't modify time while we read it
8021   for (;;) {
8022     fill_rtcdate(&t1);
8023     if (cmos_read(CMOS_STATA) & CMOS_UIP)
8024         continue;
8025     fill_rtcdate(&t2);
8026     if (memcmp(&t1, &t2, sizeof(t1)) == 0)
8027       break;
8028   }
8029
8030   // convert
8031   if (bcd) {
8032 #define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
8033     CONV(second);
8034     CONV(minute);
8035     CONV(hour  );
8036     CONV(day   );
8037     CONV(month );
8038     CONV(year  );
8039 #undef     CONV
8040   }
8041
8042   *r = t1;
8043   r->year += 2000;
8044 }
8045
8046
8047
8048
8049
```



```
8050 // The I/O APIC manages hardware interrupts for an SMP system.
8051 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
8052 // See also picirq.c.
8053
8054 #include "types.h"
8055 #include "defs.h"
8056 #include "traps.h"
8057
8058 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
8059
8060 #define REG_ID     0x00  // Register index: ID
8061 #define REG_VER    0x01  // Register index: version
8062 #define REG_TABLE  0x10  // Redirection table base
8063
8064 // The redirection table starts at REG_TABLE and uses
8065 // two registers to configure each interrupt.
8066 // The first (low) register in a pair contains configuration bits.
8067 // The second (high) register contains a bitmask telling which
8068 // CPUs can serve that interrupt.
8069 #define INT_DISABLED   0x00010000  // Interrupt disabled
8070 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
8071 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
8072 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
8073
8074 volatile struct ioapic *ioapic;
8075
8076 // IO APIC MMIO structure: write reg, then read or write data.
8077 struct ioapic {
8078   uint reg;
8079   uint pad[3];
8080   uint data;
8081 };
8082
8083 static uint
8084 ioapicread(int reg)
8085 {
8086   ioapic->reg = reg;
8087   return ioapic->data;
8088 }
8089
8090 static void
8091 ioapicwrite(int reg, uint data)
8092 {
8093   ioapic->reg = reg;
8094   ioapic->data = data;
8095 }
8096
8097
8098
8099
```

```
8100 void
8101 ioapicinit(void)
8102 {
8103   int i, id, maxintr;
8104
8105   if(!ismp)
8106     return;
8107
8108   ioapic = (volatile struct ioapic*)IOAPIC;
8109   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
8110   id = ioapicread(REG_ID) >> 24;
8111   if(id != ioapicid)
8112     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
8113
8114   // Mark all interrupts edge-triggered, active high, disabled,
8115   // and not routed to any CPUs.
8116   for(i = 0; i <= maxintr; i++){
8117     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
8118     ioapicwrite(REG_TABLE+2*i+1, 0);
8119   }
8120 }
8121
8122 void
8123 ioapicenable(int irq, int cpunum)
8124 {
8125   if(!ismp)
8126     return;
8127
8128   // Mark interrupt edge-triggered, active high,
8129   // enabled, and routed to the given cpunum,
8130   // which happens to be that cpu's APIC ID.
8131   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
8132   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
8133 }
8134
8135
8136
8137
8138
8139
8140
8141
8142
8143
8144
8145
8146
8147
8148
8149
```



```
8150 // Intel 8259A programmable interrupt controllers.
8151
8152 #include "types.h"
8153 #include "x86.h"
8154 #include "traps.h"
8155
8156 // I/O Addresses of the two programmable interrupt controllers
8157 #define IO_PIC1         0x20    // Master (IRQs 0-7)
8158 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
8159
8160 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
8161
8162 // Current IRQ mask.
8163 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
8164 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
8165
8166 static void
8167 picsetmask(ushort mask)
8168 {
8169   irqmask = mask;
8170   outb(IO_PIC1+1, mask);
8171   outb(IO_PIC2+1, mask >> 8);
8172 }
8173
8174 void
8175 picenable(int irq)
8176 {
8177   picsetmask(irqmask & ~(1<<irq));
8178 }
8179
8180 // Initialize the 8259A interrupt controllers.
8181 void
8182 picinit(void)
8183 {
8184   // mask all interrupts
8185   outb(IO_PIC1+1, 0xFF);
8186   outb(IO_PIC2+1, 0xFF);
8187
8188   // Set up master (8259A-1)
8189
8190   // ICW1:  0001g0hi
8191   //    g:  0 = edge triggering, 1 = level triggering
8192   //    h:  0 = cascaded PICs, 1 = master only
8193   //    i:  0 = no ICW4, 1 = ICW4 required
8194   outb(IO_PIC1, 0x11);
8195
8196   // ICW2:  Vector offset
8197   outb(IO_PIC1+1, T_IRQ0);
8198
8199
```

```
8200   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
8201   //        (slave PIC) 3-bit # of slave's connection to master
8202   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
8203
8204   // ICW4:  000nbmap
8205   //    n:  1 = special fully nested mode
8206   //    b:  1 = buffered mode
8207   //    m:  0 = slave PIC, 1 = master PIC
8208   //      (ignored when b is 0, as the master/slave role
8209   //      can be hardwired).
8210   //    a:  1 = Automatic EOI mode
8211   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
8212   outb(IO_PIC1+1, 0x3);
8213
8214   // Set up slave (8259A-2)
8215   outb(IO_PIC2, 0x11);                  // ICW1
8216   outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
8217   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
8218   // NB Automatic EOI mode doesn't tend to work on the slave.
8219   // Linux source code says it's "to be investigated".
8220   outb(IO_PIC2+1, 0x3);                 // ICW4
8221
8222   // OCW3:  0ef01prs
8223   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
8224   //    p:  0 = no polling, 1 = polling mode
8225   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
8226   outb(IO_PIC1, 0x68);             // clear specific mask
8227   outb(IO_PIC1, 0x0a);             // read IRR by default
8228
8229   outb(IO_PIC2, 0x68);             // OCW3
8230   outb(IO_PIC2, 0x0a);             // OCW3
8231
8232   if(irqmask != 0xFFFF)
8233     picsetmask(irqmask);
8234 }
8235
8236
8237
8238
8239
8240
8241
8242
8243
8244
8245
8246
8247
8248
8249
```



```
8250 // PC keyboard interface constants
8251
8252 #define KBSTATP         0x64    // kbd controller status port(I)
8253 #define KBS_DIB         0x01    // kbd data in buffer
8254 #define KBDATAP         0x60    // kbd data port(I)
8255
8256 #define NO              0
8257
8258 #define SHIFT           (1<<0)
8259 #define CTL             (1<<1)
8260 #define ALT             (1<<2)
8261
8262 #define CAPSLOCK        (1<<3)
8263 #define NUMLOCK         (1<<4)
8264 #define SCROLLLOCK      (1<<5)
8265
8266 #define E0ESC           (1<<6)
8267
8268 // Special keycodes
8269 #define KEY_HOME        0xE0
8270 #define KEY_END         0xE1
8271 #define KEY_UP          0xE2
8272 #define KEY_DN          0xE3
8273 #define KEY_LF          0xE4
8274 #define KEY_RT          0xE5
8275 #define KEY_PGUP        0xE6
8276 #define KEY_PGDN        0xE7
8277 #define KEY_INS         0xE8
8278 #define KEY_DEL         0xE9
8279
8280 // C('A') == Control-A
8281 #define C(x) (x - '@')
8282
8283 static uchar shiftcode[256] =
8284 {
8285   [0x1D] CTL,
8286   [0x2A] SHIFT,
8287   [0x36] SHIFT,
8288   [0x38] ALT,
8289   [0x9D] CTL,
8290   [0xB8] ALT
8291 };
8292
8293 static uchar togglecode[256] =
8294 {
8295   [0x3A] CAPSLOCK,
8296   [0x45] NUMLOCK,
8297   [0x46] SCROLLLOCK
8298 };
8299
```

```
8300 static uchar normalmap[256] =
8301 {
8302   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
8303   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
8304   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
8305   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
8306   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
8307   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
8308   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
8309   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
8310   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
8311   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
8312   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
8313   [0x9C] '\n',      // KP_Enter
8314   [0xB5] '/',       // KP_Div
8315   [0xC8] KEY_UP,    [0xD0] KEY_DN,
8316   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
8317   [0xCB] KEY_LF,    [0xCD] KEY_RT,
8318   [0x97] KEY_HOME,  [0xCF] KEY_END,
8319   [0xD2] KEY_INS,   [0xD3] KEY_DEL
8320 };
8321
8322 static uchar shiftmap[256] =
8323 {
8324   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
8325   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
8326   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
8327   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
8328   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
8329   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
8330   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
8331   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
8332   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
8333   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
8334   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
8335   [0x9C] '\n',      // KP_Enter
8336   [0xB5] '/',       // KP_Div
8337   [0xC8] KEY_UP,    [0xD0] KEY_DN,
8338   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
8339   [0xCB] KEY_LF,    [0xCD] KEY_RT,
8340   [0x97] KEY_HOME,  [0xCF] KEY_END,
8341   [0xD2] KEY_INS,   [0xD3] KEY_DEL
8342 };
8343
8344
8345
8346
8347
8348
8349
```



```
8350 static uchar ctlmap[256] =
8351 {
8352   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
8353   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
8354   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
8355   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
8356   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
8357   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
8358   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
8359   [0x9C] '\r',      // KP_Enter
8360   [0xB5] C('/'),    // KP_Div
8361   [0xC8] KEY_UP,    [0xD0] KEY_DN,
8362   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
8363   [0xCB] KEY_LF,    [0xCD] KEY_RT,
8364   [0x97] KEY_HOME,  [0xCF] KEY_END,
8365   [0xD2] KEY_INS,   [0xD3] KEY_DEL
8366 };
8367
8368
8369
8370
8371
8372
8373
8374
8375
8376
8377
8378
8379
8380
8381
8382
8383
8384
8385
8386
8387
8388
8389
8390
8391
8392
8393
8394
8395
8396
8397
8398
8399
```

```
8400 #include "types.h"
8401 #include "x86.h"
8402 #include "defs.h"
8403 #include "kbd.h"
8404
8405 int
8406 kbdgetc(void)
8407 {
8408   static uint shift;
8409   static uchar *charcode[4] = {
8410     normalmap, shiftmap, ctlmap, ctlmap
8411   };
8412   uint st, data, c;
8413
8414   st = inb(KBSTATP);
8415   if((st & KBS_DIB) == 0)
8416     return -1;
8417   data = inb(KBDATAP);
8418
8419   if(data == 0xE0){
8420     shift |= E0ESC;
8421     return 0;
8422   } else if(data & 0x80){
8423     // Key released
8424     data = (shift & E0ESC ? data : data & 0x7F);
8425     shift &= ~(shiftcode[data] | E0ESC);
8426     return 0;
8427   } else if(shift & E0ESC){
8428     // Last character was an E0 escape; or with 0x80
8429     data |= 0x80;
8430     shift &= ~E0ESC;
8431   }
8432
8433   shift |= shiftcode[data];
8434   shift ^= togglecode[data];
8435   c = charcode[shift & (CTL | SHIFT)][data];
8436   if(shift & CAPSLOCK){
8437     if('a' <= c && c <= 'z')
8438       c += 'A' - 'a';
8439     else if('A' <= c && c <= 'Z')
8440       c += 'a' - 'A';
8441   }
8442   return c;
8443 }
8444
8445 void
8446 kbdintr(void)
8447 {
8448   consoleintr(kbdgetc);
8449 }
```



```
8450 // Console input and output.
8451 // Input is from the keyboard or serial port.
8452 // Output is written to the screen and serial port.
8453
8454 #include "types.h"
8455 #include "defs.h"
8456 #include "param.h"
8457 #include "traps.h"
8458 #include "spinlock.h"
8459 #include "fs.h"
8460 #include "file.h"
8461 #include "memlayout.h"
8462 #include "mmu.h"
8463 #include "proc.h"
8464 #include "x86.h"
8465
8466 static void consputc(int);
8467
8468 static int panicked = 0;
8469
8470 static struct {
8471   struct spinlock lock;
8472   int locking;
8473 } cons;
8474
8475 static void
8476 printint(int xx, int base, int sign)
8477 {
8478   static char digits[] = "0123456789abcdef";
8479   char buf[16];
8480   int i;
8481   uint x;
8482
8483   if(sign && (sign = xx < 0))
8484     x = -xx;
8485   else
8486     x = xx;
8487
8488   i = 0;
8489   do{
8490     buf[i++] = digits[x % base];
8491   }while((x /= base) != 0);
8492
8493   if(sign)
8494     buf[i++] = '-';
8495
8496   while(--i >= 0)
8497     consputc(buf[i]);
8498 }
8499
```

```
8500 // Print to the console. only understands %d, %x, %p, %s.
8501 void
8502 cprintf(char *fmt, ...)
8503 {
8504   int i, c, locking;
8505   uint *argp;
8506   char *s;
8507
8508   locking = cons.locking;
8509   if(locking)
8510     acquire(&cons.lock);
8511
8512   if (fmt == 0)
8513     panic("null fmt");
8514
8515   argp = (uint*)(void*)(&fmt + 1);
8516   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
8517     if(c != '%'){
8518       consputc(c);
8519       continue;
8520     }
8521     c = fmt[++i] & 0xff;
8522     if(c == 0)
8523       break;
8524     switch(c){
8525     case 'd':
8526       printint(*argp++, 10, 1);
8527       break;
8528     case 'x':
8529     case 'p':
8530       printint(*argp++, 16, 0);
8531       break;
8532     case 's':
8533       if((s = (char*)*argp++) == 0)
8534         s = "(null)";
8535       for(; *s; s++)
8536         consputc(*s);
8537       break;
8538     case '%':
8539       consputc('%');
8540       break;
8541     default:
8542       // Print unknown % sequence to draw attention.
8543       consputc('%');
8544       consputc(c);
8545       break;
8546     }
8547   }
8548
8549
```



```
8550   if(locking)
8551     release(&cons.lock);
8552 }
8553
8554 void
8555 panic(char *s)
8556 {
8557   int i;
8558   uint pcs[10];
8559
8560   cli();
8561   cons.locking = 0;
8562   cprintf("cpu%d: panic: ", cpu->id);
8563   cprintf(s);
8564   cprintf("\n");
8565   getcallerpcs(&s, pcs);
8566   for(i=0; i<10; i++)
8567     cprintf(" %p", pcs[i]);
8568   panicked = 1; // freeze other CPU
8569   for(;;)
8570     ;
8571 }
8572
8573
8574
8575
8576
8577
8578
8579
8580
8581
8582
8583
8584
8585
8586
8587
8588
8589
8590
8591
8592
8593
8594
8595
8596
8597
8598
8599
```

```
8600 #define BACKSPACE 0x100
8601 #define CRTPORT 0x3d4
8602 static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory
8603
8604 static void
8605 cgaputc(int c)
8606 {
8607   int pos;
8608
8609   // Cursor position: col + 80*row.
8610   outb(CRTPORT, 14);
8611   pos = inb(CRTPORT+1) << 8;
8612   outb(CRTPORT, 15);
8613   pos |= inb(CRTPORT+1);
8614
8615   if(c == '\n')
8616     pos += 80 - pos%80;
8617   else if(c == BACKSPACE){
8618     if(pos > 0) --pos;
8619   } else
8620     crt[pos++] = (c&0xff) | 0x0700;  // black on white
8621
8622   if(pos < 0 || pos > 25*80)
8623     panic("pos under/overflow");
8624
8625   if((pos/80) >= 24){  // Scroll up.
8626     memmove(crt, crt+80, sizeof(crt[0])*23*80);
8627     pos -= 80;
8628     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
8629   }
8630
8631   outb(CRTPORT, 14);
8632   outb(CRTPORT+1, pos>>8);
8633   outb(CRTPORT, 15);
8634   outb(CRTPORT+1, pos);
8635   crt[pos] = ' ' | 0x0700;
8636 }
8637
8638
8639
8640
8641
8642
8643
8644
8645
8646
8647
8648
8649
```



```
8650 void
8651 consputc(int c)
8652 {
8653   if(panicked){
8654     cli();
8655     for(;;)
8656       ;
8657   }
8658
8659   if(c == BACKSPACE){
8660     uartputc('\b'); uartputc(' '); uartputc('\b');
8661   } else
8662     uartputc(c);
8663   cgaputc(c);
8664 }
8665
8666 #define INPUT_BUF 128
8667 struct {
8668   char buf[INPUT_BUF];
8669   uint r;  // Read index
8670   uint w;  // Write index
8671   uint e;  // Edit index
8672 } input;
8673
8674 #define C(x)  ((x)-'@')  // Control-x
8675
8676 void
8677 consoleintr(int (*getc)(void))
8678 {
8679   int c, doprocdump = 0;
8680
8681   acquire(&cons.lock);
8682   while((c = getc()) >= 0){
8683     switch(c){
8684     case C('P'):  // Process listing.
8685       doprocdump = 1;   // procdump() locks cons.lock indirectly; invoke lat
8686       break;
8687     case C('U'):  // Kill line.
8688       while(input.e != input.w &&
8689             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
8690         input.e--;
8691         consputc(BACKSPACE);
8692       }
8693       break;
8694     case C('H'): case '\x7f':  // Backspace
8695       if(input.e != input.w){
8696         input.e--;
8697         consputc(BACKSPACE);
8698       }
8699       break;
```

```
8700     default:
8701       if(c != 0 && input.e-input.r < INPUT_BUF){
8702         c = (c == '\r') ? '\n' : c;
8703         input.buf[input.e++ % INPUT_BUF] = c;
8704         consputc(c);
8705         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
8706           input.w = input.e;
8707           wakeup(&input.r);
8708         }
8709       }
8710       break;
8711     }
8712   }
8713   release(&cons.lock);
8714   if(doprocdump) {
8715     procdump();  // now call procdump() wo. cons.lock held
8716   }
8717 }
8718
8719 int
8720 consoleread(struct inode *ip, char *dst, int n)
8721 {
8722   uint target;
8723   int c;
8724
8725   iunlock(ip);
8726   target = n;
8727   acquire(&cons.lock);
8728   while(n > 0){
8729     while(input.r == input.w){
8730       if(proc->killed){
8731         release(&cons.lock);
8732         ilock(ip);
8733         return -1;
8734       }
8735       sleep(&input.r, &cons.lock);
8736     }
8737     c = input.buf[input.r++ % INPUT_BUF];
8738     if(c == C('D')){  // EOF
8739       if(n < target){
8740         // Save ^D for next time, to make sure
8741         // caller gets a 0-byte result.
8742         input.r--;
8743       }
8744       break;
8745     }
8746     *dst++ = c;
8747     --n;
8748     if(c == '\n')
8749       break;
```



```
8750   }
8751   release(&cons.lock);
8752   ilock(ip);
8753
8754   return target - n;
8755 }
8756
8757 int
8758 consolewrite(struct inode *ip, char *buf, int n)
8759 {
8760   int i;
8761
8762   iunlock(ip);
8763   acquire(&cons.lock);
8764   for(i = 0; i < n; i++)
8765     consputc(buf[i] & 0xff);
8766   release(&cons.lock);
8767   ilock(ip);
8768
8769   return n;
8770 }
8771
8772 void
8773 consoleinit(void)
8774 {
8775   initlock(&cons.lock, "console");
8776
8777   devsw[CONSOLE].write = consolewrite;
8778   devsw[CONSOLE].read = consoleread;
8779   cons.locking = 1;
8780
8781   picenable(IRQ_KBD);
8782   ioapicenable(IRQ_KBD, 0);
8783 }
8784
8785
8786
8787
8788
8789
8790
8791
8792
8793
8794
8795
8796
8797
8798
8799
```

```
8800 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
8801 // Only used on uniprocessors;
8802 // SMP machines use the local APIC timer.
8803
8804 #include "types.h"
8805 #include "defs.h"
8806 #include "traps.h"
8807 #include "x86.h"
8808
8809 #define IO_TIMER1       0x040           // 8253 Timer #1
8810
8811 // Frequency of all three count-down timers;
8812 // (TIMER_FREQ/freq) is the appropriate count
8813 // to generate a frequency of freq Hz.
8814
8815 #define TIMER_FREQ      1193182
8816 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
8817
8818 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
8819 #define TIMER_SEL0      0x00    // select counter 0
8820 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
8821 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
8822
8823 void
8824 timerinit(void)
8825 {
8826   // Interrupt 100 times/sec.
8827   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
8828   outb(IO_TIMER1, TIMER_DIV(100) % 256);
8829   outb(IO_TIMER1, TIMER_DIV(100) / 256);
8830   picenable(IRQ_TIMER);
8831 }
8832
8833
8834
8835
8836
8837
8838
8839
8840
8841
8842
8843
8844
8845
8846
8847
8848
8849
```



```
8850 // Intel 8250 serial port (UART).
8851
8852 #include "types.h"
8853 #include "defs.h"
8854 #include "param.h"
8855 #include "traps.h"
8856 #include "spinlock.h"
8857 #include "fs.h"
8858 #include "file.h"
8859 #include "mmu.h"
8860 #include "proc.h"
8861 #include "x86.h"
8862
8863 #define COM1    0x3f8
8864
8865 static int uart;    // is there a uart?
8866
8867 void
8868 uartinit(void)
8869 {
8870   char *p;
8871
8872   // Turn off the FIFO
8873   outb(COM1+2, 0);
8874
8875   // 9600 baud, 8 data bits, 1 stop bit, parity off.
8876   outb(COM1+3, 0x80);    // Unlock divisor
8877   outb(COM1+0, 115200/9600);
8878   outb(COM1+1, 0);
8879   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
8880   outb(COM1+4, 0);
8881   outb(COM1+1, 0x01);    // Enable receive interrupts.
8882
8883   // If status is 0xFF, no serial port.
8884   if(inb(COM1+5) == 0xFF)
8885     return;
8886   uart = 1;
8887
8888   // Acknowledge pre-existing interrupt conditions;
8889   // enable interrupts.
8890   inb(COM1+2);
8891   inb(COM1+0);
8892   picenable(IRQ_COM1);
8893   ioapicenable(IRQ_COM1, 0);
8894
8895   // Announce that we're here.
8896   for(p="xv6...\n"; *p; p++)
8897     uartputc(*p);
8898 }
8899
```

```
8900 void
8901 uartputc(int c)
8902 {
8903   int i;
8904
8905   if(!uart)
8906     return;
8907   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
8908     microdelay(10);
8909   outb(COM1+0, c);
8910 }
8911
8912 static int
8913 uartgetc(void)
8914 {
8915   if(!uart)
8916     return -1;
8917   if(!(inb(COM1+5) & 0x01))
8918     return -1;
8919   return inb(COM1+0);
8920 }
8921
8922 void
8923 uartintr(void)
8924 {
8925   consoleintr(uartgetc);
8926 }
8927
8928
8929
8930
8931
8932
8933
8934
8935
8936
8937
8938
8939
8940
8941
8942
8943
8944
8945
8946
8947
8948
8949
```



```
8950 # Initial process execs /init.
8951
8952 #include "syscall.h"
8953 #include "traps.h"
8954
8955
8956 # exec(init, argv)
8957 .globl start
8958 start:
8959   pushl $argv
8960   pushl $init
8961   pushl $0  // where caller pc would be
8962   movl $SYS_exec, %eax
8963   int $T_SYSCALL
8964
8965 # for(;;) exit();
8966 exit:
8967   movl $SYS_exit, %eax
8968   int $T_SYSCALL
8969   jmp exit
8970
8971 # char init[] = "/init\0";
8972 init:
8973   .string "/init\0"
8974
8975 # char *argv[] = { init, 0 };
8976 .p2align 2
8977 argv:
8978   .long init
8979   .long 0
8980
8981
8982
8983
8984
8985
8986
8987
8988
8989
8990
8991
8992
8993
8994
8995
8996
8997
8998
8999
```

```
9000 #include "syscall.h"
9001 #include "traps.h"
9002
9003 #define SYSCALL(name) \
9004   .globl name; \
9005   name: \
9006     movl $SYS_ ## name, %eax; \
9007     int $T_SYSCALL; \
9008     ret
9009
9010 SYSCALL(fork)
9011 SYSCALL(exit)
9012 SYSCALL(wait)
9013 SYSCALL(pipe)
9014 SYSCALL(read)
9015 SYSCALL(write)
9016 SYSCALL(close)
9017 SYSCALL(kill)
9018 SYSCALL(exec)
9019 SYSCALL(open)
9020 SYSCALL(mknod)
9021 SYSCALL(unlink)
9022 SYSCALL(fstat)
9023 SYSCALL(link)
9024 SYSCALL(mkdir)
9025 SYSCALL(chdir)
9026 SYSCALL(dup)
9027 SYSCALL(getpid)
9028 SYSCALL(sbrk)
9029 SYSCALL(sleep)
9030 SYSCALL(uptime)
9031 SYSCALL(halt)
9032 SYSCALL(date)
9033 SYSCALL(getuid)
9034 SYSCALL(getgid)
9035 SYSCALL(getppid)
9036 SYSCALL(setuid)
9037 SYSCALL(setgid)
9038 SYSCALL(getprocs)
9039 SYSCALL(getpriority)
9040 SYSCALL(setpriority);
9041
9042
9043
9044
9045
9046
9047
9048
9049
```



```
9050 // init: The initial user-level program
9051
9052 #include "types.h"
9053 #include "stat.h"
9054 #include "user.h"
9055 #include "fcntl.h"
9056
9057 char *argv[] = { "sh", 0 };
9058
9059 int
9060 main(void)
9061 {
9062   int pid, wpid;
9063   if(open("console", O_RDWR) < 0){
9064     mknod("console", 1, 1);
9065     open("console", O_RDWR);
9066   }
9067   dup(0);  // stdout
9068   dup(0);  // stderr
9069
9070   for(;;){
9071     printf(1, "init: starting sh\n");
9072     pid = fork();
9073     if(pid < 0){
9074       printf(1, "init: fork failed\n");
9075       exit();
9076     }
9077     if(pid == 0){
9078       exec("sh", argv);
9079       printf(1, "init: exec sh failed\n");
9080       exit();
9081     }
9082     while((wpid=wait()) >= 0 && wpid != pid)
9083       printf(1, "zombie!\n");
9084   }
9085 }
9086
9087
9088
9089
9090
9091
9092
9093
9094
9095
9096
9097
9098
9099
```

```
9100 // Shell.
9101 // 2015-12-21. Added very simple processing for builtin commands
9102
9103 #include "types.h"
9104 #include "user.h"
9105 #include "fcntl.h"
9106
9107 // Parsed command representation
9108 #define EXEC  1
9109 #define REDIR 2
9110 #define PIPE  3
9111 #define LIST  4
9112 #define BACK  5
9113
9114 #define MAXARGS 10
9115
9116 struct cmd {
9117   int type;
9118 };
9119
9120 struct execcmd {
9121   int type;
9122   char *argv[MAXARGS];
9123   char *eargv[MAXARGS];
9124 };
9125
9126 struct redircmd {
9127   int type;
9128   struct cmd *cmd;
9129   char *file;
9130   char *efile;
9131   int mode;
9132   int fd;
9133 };
9134
9135 struct pipecmd {
9136   int type;
9137   struct cmd *left;
9138   struct cmd *right;
9139 };
9140
9141 struct listcmd {
9142   int type;
9143   struct cmd *left;
9144   struct cmd *right;
9145 };
9146
9147
9148
9149
```



```
9150 struct backcmd {
9151   int type;
9152   struct cmd *cmd;
9153 };
9154
9155 int fork1(void);  // Fork but panics on failure.
9156 void panic(char*);
9157 struct cmd *parsecmd(char*);
9158
9159 // Execute cmd.  Never returns.
9160 void
9161 runcmd(struct cmd *cmd)
9162 {
9163   int p[2];
9164   struct backcmd *bcmd;
9165   struct execcmd *ecmd;
9166   struct listcmd *lcmd;
9167   struct pipecmd *pcmd;
9168   struct redircmd *rcmd;
9169
9170   if(cmd == 0)
9171     exit();
9172
9173   switch(cmd->type){
9174   default:
9175     panic("runcmd");
9176
9177   case EXEC:
9178     ecmd = (struct execcmd*)cmd;
9179     if(ecmd->argv[0] == 0)
9180       exit();
9181     exec(ecmd->argv[0], ecmd->argv);
9182     printf(2, "exec %s failed\n", ecmd->argv[0]);
9183     break;
9184
9185   case REDIR:
9186     rcmd = (struct redircmd*)cmd;
9187     close(rcmd->fd);
9188     if(open(rcmd->file, rcmd->mode) < 0){
9189       printf(2, "open %s failed\n", rcmd->file);
9190       exit();
9191     }
9192     runcmd(rcmd->cmd);
9193     break;
9194
9195   case LIST:
9196     lcmd = (struct listcmd*)cmd;
9197     if(fork1() == 0)
9198       runcmd(lcmd->left);
9199     wait();
```

```
9200     runcmd(lcmd->right);
9201     break;
9202
9203   case PIPE:
9204     pcmd = (struct pipecmd*)cmd;
9205     if(pipe(p) < 0)
9206       panic("pipe");
9207     if(fork1() == 0){
9208       close(1);
9209       dup(p[1]);
9210       close(p[0]);
9211       close(p[1]);
9212       runcmd(pcmd->left);
9213     }
9214     if(fork1() == 0){
9215       close(0);
9216       dup(p[0]);
9217       close(p[0]);
9218       close(p[1]);
9219       runcmd(pcmd->right);
9220     }
9221     close(p[0]);
9222     close(p[1]);
9223     wait();
9224     wait();
9225     break;
9226
9227   case BACK:
9228     bcmd = (struct backcmd*)cmd;
9229     if(fork1() == 0)
9230       runcmd(bcmd->cmd);
9231     break;
9232   }
9233   exit();
9234 }
9235
9236 int
9237 getcmd(char *buf, int nbuf)
9238 {
9239   printf(2, "$ ");
9240   memset(buf, 0, nbuf);
9241   gets(buf, nbuf);
9242   if(buf[0] == 0) // EOF
9243     return -1;
9244   return 0;
9245 }
9246
9247
9248
9249
```



```
9250 // ***** processing for shell builtins begins here *****
9251
9252 int
9253 strncmp(const char *p, const char *q, uint n)
9254 {
9255     while(n > 0 && *p && *p == *q)
9256       n--, p++, q++;
9257     if(n == 0)
9258       return 0;
9259     return (uchar)*p - (uchar)*q;
9260 }
9261
9262 int
9263 makeint(char *p)
9264 {
9265   int val = 0;
9266
9267   while ((*p >= '0') && (*p <= '9')) {
9268     val = 10*val + (*p-'0');
9269     ++p;
9270   }
9271   return val;
9272 }
9273
9274 int
9275 setbuiltin(char *p)
9276 {
9277   int i;
9278
9279   p += strlen("_set");
9280   while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
9281   if (strncmp("uid", p, 3) == 0) {
9282     p += strlen("uid");
9283     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
9284     i = makeint(p); // ugly
9285     return (setuid(i));
9286   } else
9287   if (strncmp("gid", p, 3) == 0) {
9288     p += strlen("gid");
9289     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
9290     i = makeint(p); // ugly
9291     return (setgid(i));
9292   }
9293   printf(2, "Invalid _set parameter\n");
9294   return -1;
9295 }
9296
9297
9298
9299
```

```
9300 int
9301 getbuiltin(char *p)
9302 {
9303   p += strlen("_get");
9304   while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
9305   if (strncmp("uid", p, 3) == 0) {
9306     printf(2, "%d\n", getuid());
9307     return 0;
9308   }
9309   if (strncmp("gid", p, 3) == 0) {
9310     printf(2, "%d\n", getgid());
9311     return 0;
9312   }
9313   printf(2, "Invalid _get parameter\n");
9314   return -1;
9315 }
9316
9317 typedef int funcPtr_t(char *);
9318 typedef struct {
9319   char       *cmd;
9320   funcPtr_t  *name;
9321 } dispatchTableEntry_t;
9322
9323 // Use a simple function dispatch table (FDT) to process builtin commands
9324 dispatchTableEntry_t fdt[] = {
9325   {"_set", setbuiltin},
9326   {"_get", getbuiltin}
9327 };
9328 int FDTcount = sizeof(fdt) / sizeof(fdt[0]); // # entris in FDT
9329
9330 void
9331 dobuiltin(char *cmd) {
9332   int i;
9333
9334   for (i=0; i<FDTcount; i++)
9335     if (strncmp(cmd, fdt[i].cmd, strlen(fdt[i].cmd)) == 0)
9336      (*fdt[i].name)(cmd);
9337 }
9338
9339
9340
9341
9342
9343
9344
9345
9346
9347
9348
9349
```



```
9350 // ***** processing for shell builtins ends here *****
9351
9352 int
9353 main(void)
9354 {
9355   static char buf[100];
9356   int fd;
9357
9358   // Assumes three file descriptors open.
9359   while((fd = open("console", O_RDWR)) >= 0){
9360     if(fd >= 3){
9361       close(fd);
9362       break;
9363     }
9364   }
9365
9366   // Read and run input commands.
9367   while(getcmd(buf, sizeof(buf)) >= 0){
9368 // add support for built-ins here. cd is a built-in
9369     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
9370       // Clumsy but will have to do for now.
9371       // Chdir has no effect on the parent if run in the child.
9372       buf[strlen(buf)-1] = 0;  // chop \n
9373       if(chdir(buf+3) < 0)
9374         printf(2, "cannot cd %s\n", buf+3);
9375       continue;
9376     }
9377     if (buf[0]=='_') {     // assume it is a builtin command
9378       dobuiltin(buf);
9379       continue;
9380     }
9381     if(fork1() == 0)
9382       runcmd(parsecmd(buf));
9383     wait();
9384   }
9385   exit();
9386 }
9387
9388 void
9389 panic(char *s)
9390 {
9391   printf(2, "%s\n", s);
9392   exit();
9393 }
9394
9395
9396
9397
9398
9399
```

```
9400 int
9401 fork1(void)
9402 {
9403   int pid;
9404
9405   pid = fork();
9406   if(pid == -1)
9407     panic("fork");
9408   return pid;
9409 }
9410
9411
9412
9413
9414
9415
9416
9417
9418
9419
9420
9421
9422
9423
9424
9425
9426
9427
9428
9429
9430
9431
9432
9433
9434
9435
9436
9437
9438
9439
9440
9441
9442
9443
9444
9445
9446
9447
9448
9449
```



```
9450 // Constructors
9451
9452 struct cmd*
9453 execcmd(void)
9454 {
9455   struct execcmd *cmd;
9456
9457   cmd = malloc(sizeof(*cmd));
9458   memset(cmd, 0, sizeof(*cmd));
9459   cmd->type = EXEC;
9460   return (struct cmd*)cmd;
9461 }
9462
9463 struct cmd*
9464 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
9465 {
9466   struct redircmd *cmd;
9467
9468   cmd = malloc(sizeof(*cmd));
9469   memset(cmd, 0, sizeof(*cmd));
9470   cmd->type = REDIR;
9471   cmd->cmd = subcmd;
9472   cmd->file = file;
9473   cmd->efile = efile;
9474   cmd->mode = mode;
9475   cmd->fd = fd;
9476   return (struct cmd*)cmd;
9477 }
9478
9479 struct cmd*
9480 pipecmd(struct cmd *left, struct cmd *right)
9481 {
9482   struct pipecmd *cmd;
9483
9484   cmd = malloc(sizeof(*cmd));
9485   memset(cmd, 0, sizeof(*cmd));
9486   cmd->type = PIPE;
9487   cmd->left = left;
9488   cmd->right = right;
9489   return (struct cmd*)cmd;
9490 }
9491
9492
9493
9494
9495
9496
9497
9498
9499
```

```
9500 struct cmd*
9501 listcmd(struct cmd *left, struct cmd *right)
9502 {
9503   struct listcmd *cmd;
9504
9505   cmd = malloc(sizeof(*cmd));
9506   memset(cmd, 0, sizeof(*cmd));
9507   cmd->type = LIST;
9508   cmd->left = left;
9509   cmd->right = right;
9510   return (struct cmd*)cmd;
9511 }
9512
9513 struct cmd*
9514 backcmd(struct cmd *subcmd)
9515 {
9516   struct backcmd *cmd;
9517
9518   cmd = malloc(sizeof(*cmd));
9519   memset(cmd, 0, sizeof(*cmd));
9520   cmd->type = BACK;
9521   cmd->cmd = subcmd;
9522   return (struct cmd*)cmd;
9523 }
9524
9525
9526
9527
9528
9529
9530
9531
9532
9533
9534
9535
9536
9537
9538
9539
9540
9541
9542
9543
9544
9545
9546
9547
9548
9549
```



```
9550 // Parsing
9551
9552 char whitespace[] = " \t\r\n\v";
9553 char symbols[] = "<|>&;()";
9554
9555 int
9556 gettoken(char **ps, char *es, char **q, char **eq)
9557 {
9558   char *s;
9559   int ret;
9560
9561   s = *ps;
9562   while(s < es && strchr(whitespace, *s))
9563     s++;
9564   if(q)
9565     *q = s;
9566   ret = *s;
9567   switch(*s){
9568   case 0:
9569     break;
9570   case '|':
9571   case '(':
9572   case ')':
9573   case ';':
9574   case '&':
9575   case '<':
9576     s++;
9577     break;
9578   case '>':
9579     s++;
9580     if(*s == '>'){
9581       ret = '+';
9582       s++;
9583     }
9584     break;
9585   default:
9586     ret = 'a';
9587     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
9588       s++;
9589     break;
9590   }
9591   if(eq)
9592     *eq = s;
9593
9594   while(s < es && strchr(whitespace, *s))
9595     s++;
9596   *ps = s;
9597   return ret;
9598 }
9599
```

```
9600 int
9601 peek(char **ps, char *es, char *toks)
9602 {
9603   char *s;
9604
9605   s = *ps;
9606   while(s < es && strchr(whitespace, *s))
9607     s++;
9608   *ps = s;
9609   return *s && strchr(toks, *s);
9610 }
9611
9612 struct cmd *parseline(char**, char*);
9613 struct cmd *parsepipe(char**, char*);
9614 struct cmd *parseexec(char**, char*);
9615 struct cmd *nulterminate(struct cmd*);
9616
9617 struct cmd*
9618 parsecmd(char *s)
9619 {
9620   char *es;
9621   struct cmd *cmd;
9622
9623   es = s + strlen(s);
9624   cmd = parseline(&s, es);
9625   peek(&s, es, "");
9626   if(s != es){
9627     printf(2, "leftovers: %s\n", s);
9628     panic("syntax");
9629   }
9630   nulterminate(cmd);
9631   return cmd;
9632 }
9633
9634 struct cmd*
9635 parseline(char **ps, char *es)
9636 {
9637   struct cmd *cmd;
9638
9639   cmd = parsepipe(ps, es);
9640   while(peek(ps, es, "&")){
9641     gettoken(ps, es, 0, 0);
9642     cmd = backcmd(cmd);
9643   }
9644   if(peek(ps, es, ";")){
9645     gettoken(ps, es, 0, 0);
9646     cmd = listcmd(cmd, parseline(ps, es));
9647   }
9648   return cmd;
9649 }
```



```
9650 struct cmd*
9651 parsepipe(char **ps, char *es)
9652 {
9653   struct cmd *cmd;
9654
9655   cmd = parseexec(ps, es);
9656   if(peek(ps, es, "|")){
9657     gettoken(ps, es, 0, 0);
9658     cmd = pipecmd(cmd, parsepipe(ps, es));
9659   }
9660   return cmd;
9661 }
9662
9663 struct cmd*
9664 parseredirs(struct cmd *cmd, char **ps, char *es)
9665 {
9666   int tok;
9667   char *q, *eq;
9668
9669   while(peek(ps, es, "<>")){
9670     tok = gettoken(ps, es, 0, 0);
9671     if(gettoken(ps, es, &q, &eq) != 'a')
9672       panic("missing file for redirection");
9673     switch(tok){
9674     case '<':
9675       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
9676       break;
9677     case '>':
9678       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9679       break;
9680     case '+':  // >>
9681       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
9682       break;
9683     }
9684   }
9685   return cmd;
9686 }
9687
9688
9689
9690
9691
9692
9693
9694
9695
9696
9697
9698
9699
```

```
9700 struct cmd*
9701 parseblock(char **ps, char *es)
9702 {
9703   struct cmd *cmd;
9704
9705   if(!peek(ps, es, "("))
9706     panic("parseblock");
9707   gettoken(ps, es, 0, 0);
9708   cmd = parseline(ps, es);
9709   if(!peek(ps, es, ")"))
9710     panic("syntax - missing )");
9711   gettoken(ps, es, 0, 0);
9712   cmd = parseredirs(cmd, ps, es);
9713   return cmd;
9714 }
9715
9716 struct cmd*
9717 parseexec(char **ps, char *es)
9718 {
9719   char *q, *eq;
9720   int tok, argc;
9721   struct execcmd *cmd;
9722   struct cmd *ret;
9723
9724   if(peek(ps, es, "("))
9725     return parseblock(ps, es);
9726
9727   ret = execcmd();
9728   cmd = (struct execcmd*)ret;
9729
9730   argc = 0;
9731   ret = parseredirs(ret, ps, es);
9732   while(!peek(ps, es, "|)&;")){
9733     if((tok=gettoken(ps, es, &q, &eq)) == 0)
9734       break;
9735     if(tok != 'a')
9736       panic("syntax");
9737     cmd->argv[argc] = q;
9738     cmd->eargv[argc] = eq;
9739     argc++;
9740     if(argc >= MAXARGS)
9741       panic("too many args");
9742     ret = parseredirs(ret, ps, es);
9743   }
9744   cmd->argv[argc] = 0;
9745   cmd->eargv[argc] = 0;
9746   return ret;
9747 }
9748
9749
```



```
9750 // NUL-terminate all the counted strings.
9751 struct cmd*
9752 nulterminate(struct cmd *cmd)
9753 {
9754   int i;
9755   struct backcmd *bcmd;
9756   struct execcmd *ecmd;
9757   struct listcmd *lcmd;
9758   struct pipecmd *pcmd;
9759   struct redircmd *rcmd;
9760
9761   if(cmd == 0)
9762     return 0;
9763
9764   switch(cmd->type){
9765   case EXEC:
9766     ecmd = (struct execcmd*)cmd;
9767     for(i=0; ecmd->argv[i]; i++)
9768       *ecmd->eargv[i] = 0;
9769     break;
9770
9771   case REDIR:
9772     rcmd = (struct redircmd*)cmd;
9773     nulterminate(rcmd->cmd);
9774     *rcmd->efile = 0;
9775     break;
9776
9777   case PIPE:
9778     pcmd = (struct pipecmd*)cmd;
9779     nulterminate(pcmd->left);
9780     nulterminate(pcmd->right);
9781     break;
9782
9783   case LIST:
9784     lcmd = (struct listcmd*)cmd;
9785     nulterminate(lcmd->left);
9786     nulterminate(lcmd->right);
9787     break;
9788
9789   case BACK:
9790     bcmd = (struct backcmd*)cmd;
9791     nulterminate(bcmd->cmd);
9792     break;
9793   }
9794   return cmd;
9795 }
9796
9797
9798
9799
```

```
9800 #include "asm.h"
9801 #include "memlayout.h"
9802 #include "mmu.h"
9803
9804 # Start the first CPU: switch to 32-bit protected mode, jump into C.
9805 # The BIOS loads this code from the first sector of the hard disk into
9806 # memory at physical address 0x7c00 and starts executing in real mode
9807 # with %cs=0 %ip=7c00.
9808
9809 .code16                       # Assemble for 16-bit mode
9810 .globl start
9811 start:
9812   cli                         # BIOS enabled interrupts; disable
9813
9814   # Zero data segment registers DS, ES, and SS.
9815   xorw    %ax,%ax             # Set %ax to zero
9816   movw    %ax,%ds             # -> Data Segment
9817   movw    %ax,%es             # -> Extra Segment
9818   movw    %ax,%ss             # -> Stack Segment
9819
9820   # Physical address line A20 is tied to zero so that the first PCs
9821   # with 2 MB would run software that assumed 1 MB.  Undo that.
9822 seta20.1:
9823   inb     $0x64,%al               # Wait for not busy
9824   testb   $0x2,%al
9825   jnz     seta20.1
9826
9827   movb    $0xd1,%al               # 0xd1 -> port 0x64
9828   outb    %al,$0x64
9829
9830 seta20.2:
9831   inb     $0x64,%al               # Wait for not busy
9832   testb   $0x2,%al
9833   jnz     seta20.2
9834
9835   movb    $0xdf,%al               # 0xdf -> port 0x60
9836   outb    %al,$0x60
9837
9838   # Switch from real to protected mode.  Use a bootstrap GDT that makes
9839   # virtual addresses map directly to physical addresses so that the
9840   # effective memory map doesn't change during the transition.
9841   lgdt    gdtdesc
9842   movl    %cr0, %eax
9843   orl     $CR0_PE, %eax
9844   movl    %eax, %cr0
9845
9846
9847
9848
9849
```



```
9850   # Complete transition to 32-bit protected mode by using long jmp
9851   # to reload %cs and %eip.  The segment descriptors are set up with no
9852   # translation, so that the mapping is still the identity mapping.
9853   ljmp    $(SEG_KCODE<<3), $start32
9854
9855 .code32  # Tell assembler to generate 32-bit code now.
9856 start32:
9857   # Set up the protected-mode data segment registers
9858   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
9859   movw    %ax, %ds                # -> DS: Data Segment
9860   movw    %ax, %es                # -> ES: Extra Segment
9861   movw    %ax, %ss                # -> SS: Stack Segment
9862   movw    $0, %ax                 # Zero segments not ready for use
9863   movw    %ax, %fs                # -> FS
9864   movw    %ax, %gs                # -> GS
9865
9866   # Set up the stack pointer and call into C.
9867   movl    $start, %esp
9868   call    bootmain
9869
9870   # If bootmain returns (it shouldn't), trigger a Bochs
9871   # breakpoint if running under Bochs, then loop.
9872   movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
9873   movw    %ax, %dx
9874   outw    %ax, %dx
9875   movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
9876   outw    %ax, %dx
9877 spin:
9878   jmp     spin
9879
9880 # Bootstrap GDT
9881 .p2align 2                                # force 4 byte alignment
9882 gdt:
9883   SEG_NULLASM                             # null seg
9884   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
9885   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
9886
9887 gdtdesc:
9888   .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
9889   .long   gdt                             # address gdt
9890
9891
9892
9893
9894
9895
9896
9897
9898
9899
```

```
9900 // Boot loader.
9901 //
9902 // Part of the boot block, along with bootasm.S, which calls bootmain().
9903 // bootasm.S has put the processor into protected 32-bit mode.
9904 // bootmain() loads an ELF kernel image from the disk starting at
9905 // sector 1 and then jumps to the kernel entry routine.
9906
9907 #include "types.h"
9908 #include "elf.h"
9909 #include "x86.h"
9910 #include "memlayout.h"
9911
9912 #define SECTSIZE  512
9913
9914 void readseg(uchar*, uint, uint);
9915
9916 void
9917 bootmain(void)
9918 {
9919   struct elfhdr *elf;
9920   struct proghdr *ph, *eph;
9921   void (*entry)(void);
9922   uchar* pa;
9923
9924   elf = (struct elfhdr*)0x10000;  // scratch space
9925
9926   // Read 1st page off disk
9927   readseg((uchar*)elf, 4096, 0);
9928
9929   // Is this an ELF executable?
9930   if(elf->magic != ELF_MAGIC)
9931     return;  // let bootasm.S handle error
9932
9933   // Load each program segment (ignores ph flags).
9934   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
9935   eph = ph + elf->phnum;
9936   for(; ph < eph; ph++){
9937     pa = (uchar*)ph->paddr;
9938     readseg(pa, ph->filesz, ph->off);
9939     if(ph->memsz > ph->filesz)
9940       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
9941   }
9942
9943   // Call the entry point from the ELF header.
9944   // Does not return!
9945   entry = (void(*)(void))(elf->entry);
9946   entry();
9947 }
9948
9949
```



```
9950 void
9951 waitdisk(void)
9952 {
9953   // Wait for disk ready.
9954   while((inb(0x1F7) & 0xC0) != 0x40)
9955     ;
9956 }
9957
9958 // Read a single sector at offset into dst.
9959 void
9960 readsect(void *dst, uint offset)
9961 {
9962   // Issue command.
9963   waitdisk();
9964   outb(0x1F2, 1);   // count = 1
9965   outb(0x1F3, offset);
9966   outb(0x1F4, offset >> 8);
9967   outb(0x1F5, offset >> 16);
9968   outb(0x1F6, (offset >> 24) | 0xE0);
9969   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
9970
9971   // Read data.
9972   waitdisk();
9973   insl(0x1F0, dst, SECTSIZE/4);
9974 }
9975
9976 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
9977 // Might copy more than asked.
9978 void
9979 readseg(uchar* pa, uint count, uint offset)
9980 {
9981   uchar* epa;
9982
9983   epa = pa + count;
9984
9985   // Round down to sector boundary.
9986   pa -= offset % SECTSIZE;
9987
9988   // Translate from bytes to sectors; kernel starts at sector 1.
9989   offset = (offset / SECTSIZE) + 1;
9990
9991   // If this is too slow, we could read lots of sectors at a time.
9992   // We'd write more to memory than asked, but it doesn't matter --
9993   // we load in increasing order.
9994   for(; pa < epa; pa += SECTSIZE, offset++)
9995     readsect(pa, offset);
9996 }
9997
9998
9999
```

```
10000 #include "types.h"
10001 #include "user.h"
10002 #include "date.h"
10003
10004 int
10005 printdate(struct rtcdate *r)
10006 {
10007      if (!r) /*avoid kernel panic*/
10008   return -1;
10009
10010   printf(1,"%d :%d :%d :%d :%d\n",
10011         r->year, r->month, r->day,
10012         r->hour, r->minute, r->second);
10013      return 0;
10014 }
10015
10016 int
10017 main(int argc, char*argv[])
10018 {
10019
10020  struct rtcdate r;
10021  if (date(&r)){
10022   printf(1, "date failed\n");
10023   exit();
10024  }
10025  printdate(&r);
10026  exit();
10027 }
10028
10029
10030
10031
10032
10033
10034
10035
10036
10037
10038
10039
10040
10041
10042
10043
10044
10045
10046
10047
10048
10049
```



```
10050 #include "types.h"
10051 #include "user.h"
10052 #include "date.h"
10053
10054 int
10055 main(int argc ,char*argv[])
10056 {
10057  struct rtcdate t1;
10058  struct rtcdate t2;
10059  struct rtcdate tf;
10060  int rc;
10061
10062  tf.minute=0;
10063  date(&t1);
10064
10065  /*run program argv*/
10066  rc = fork();
10067
10068
10069  if(rc > 0) /*if parent function*/
10070   wait();
10071  else{
10072   exec(argv[1],argv);
10073   exit();
10074  }
10075  date (&t2);
10076  /*do the time math*/
10077
10078  if (t2.second < t1.second){
10079    t1.second = 60-t1.second;
10080    tf.second = t1.second + t2.second;
10081    tf.minute = -1;
10082  }
10083  else
10084   tf.second = t2.second-t1.second;
10085
10086  if(t2.minute < t1.minute){
10087      t1.minute = 60-t1.minute;
10088      tf.minute += t1.minute + t2.minute;
10089   }
10090  else
10091    tf.minute += (t2.minute)-(t1.minute);
10092
10093  /*print results*/
10094  printf(1,"the program ran in %d minutes and %d seconds\n",tf.minute, tf.se
10095  exit();
10096
10097 }
10098
10099
```

```
#include "types.h"
#include "user.h"
#include "ps.h"
int
main(int argc, char*argv[])
{
 int max = 64;
 struct uproc table[max];
 int i, j;
  j = getprocs(max, table);
 if (max > j) /*fewer processes than requested*/
   max = j;
 for(i=0; i<max; i++){
  printf(1 ,"%d  ",table[i].pid);
  printf(1 ,"%d  ",table[i].uid);
  printf(1 ,"%d  ",table[i].gid);
  printf(1 ,"%d  ",table[i].ppid);
  switch(table[i].state){
    case 0:
          printf(1, "EMBRYO");
          break;
    case 1:
  printf(1, "SLEEPING");
  break;
    case 2:
  printf(1, "RUNNABLE");
         break;
    case 3:
  printf(1, "RUNNING");
       break;
    case 4:
  printf(1, "ZOMBIE");
  }
  printf(1 ,"  %d  ",table[i].priority);
  printf(1 ,"  %d  ",table[i].size);
  printf(1 ,"%s\n",table[i].name);
 }
 exit();
}
```



```
#include "types.h"
#include "user.h"
int
main(int argc, char*argv[])
{
  int uid,gid ,ppid;
  uid = getuid();
  printf(1,"Current UID is : %d\n", uid );
  printf(1,"Setting UID to 100\n");
  setuid(100);
  uid = getuid();
  printf(1 , "Current UID is : %d\n",uid);
  gid = getgid();
  printf( 1,"Current GID is : %d\n",gid);
  printf(1,"Setting GID to 100\n");
  setgid(100);
  gid = getgid();
  printf(1,"Current GID is:%d\n" ,uid);
  ppid = getppid();
  printf(1 ,"My parent process is : %d\n" , ppid);
  printf(1 ,"Done! \n" );
  exit();
}
```

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

int
main(int argc, char*argv[])
{
 int tmp;
 int pid;
 pid = getpid();
 tmp = getpriority(pid);
 printf(1,"process %d priority is %d\n",pid,tmp);
 tmp = 2;
 setpriority(pid,tmp);
 tmp = getpriority(pid);
 printf(1,"process %d priority is now %d\n",pid,tmp);
 exit();
}
```



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

// We currently have 3 priority levels
#define PrioCount 3
#define numChildren 10

void
countForever(int p)
{
  int j;
  unsigned long count = 0;

  j = getpid();
  p = p%PrioCount;
  setpriority(j, p);
  printf(1, "%d: start prio %d\n", j, p);

  while (1) {
    count++;
    if ((count & 0xFFFFFFF) == 0) {
      p = (p+1) % PrioCount;
      setpriority(j, p);
      printf(1, "%d: new prio %d\n", j, p);
    }
  }
}

int
main(void)
{
  int i, rc;

  for (i=0; i<numChildren; i++) {
    rc = fork();
    if (!rc) { // child
      countForever(i);
    }
  }
  // what the heck, let's have the parent waste time as well!
  countForever(1);
  exit();
}
```