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/6.828/2016/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, and Austin
Clements.

We are also grateful for the bug reports and patches contributed by Silas
Boyd-Wickizer, Cody Cutler, Mike CAT, Nelson Elhage, Nathaniel Filardo, Peter
Froehlich, Yakir Goaron, Shivam Handa, Bryan Henry, Jim Huang, Anders Kaseorg,
kehao95, Wolfgang Keller, Eddie Kohler, Imbar Marinescu, Yandong Mao, Hitoshi
Mitake, Carmi Merimovich, Joel Nider, Greg Price, Ayan Shafqat, Eldar Sehayek,
Yongming Shen, Cam Tenny, Rafael Ubal, Warren Toomey, Stephen Tu, Pablo Ventura,
Xi Wang, Keiichi Watanabe, Nicolas Wolovick, Jindong Zhang, and Zou Chang Wei.

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). If you have
suggestions for improvements, please keep in mind that the main purpose of xv6
is as a teaching operating system for MIT's 6.828. For example, we are in
particular interested in simplifications and clarifications, instead of
suggestions for new systems calls, more portability, etc.

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/6.828/2016/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.

```
    9290 9291 9292 9296
B_VALID 3961
    3961 4473 4510 4528 4657
bwrite 4665
    0266 4665 4668 4930 4963
    5041
bzero 5152
    5152 5180
C 8181 8624
    8181 8229 8254 8255 8256
    8257 8258 8260 8624 8634
    8638 8645 8656 8689
CAPSLOCK 8162
    8162 8195 8336
cgaputc 8555
    8555 8613
clearpteu 2034
    0438 2034 2040 6833
cli 0557
    0557 0559 1224 1660 8510
    8604 9712
cmd 9068
    9068 9080 9089 9090 9095
    9096 9102 9107 9182 9191
    9194 9201 9211 9217 9221
    9229 9253 9255 9352 9355
    9357 9358 9359 9360 9363
    9364 9366 9368 9369 9370
    9371 9372 9373 9374 9375
    9376 9379 9380 9382 9384
    9385 9386 9387 9388 9389
    9400 9401 9403 9405 9406
    9407 9408 9409 9410 9413
    9414 9416 9418 9419 9420
    9421 9422 9512 9513 9514
    9515 9517 9521 9524 9530
    9531 9534 9537 9539 9542
    9546 9548 9550 9553 9555
    9558 9560 9563 9564 9575
    9578 9581 9585 9600 9603
    9608 9612 9613 9616 9621
    9622 9628 9637 9638 9644
    9645 9651 9652 9661 9664
    9666 9672 9673 9678 9684
    9690 9691 9694
CMOS_PORT 7750
    7750 7764 7765 7813
CMOS_RETURN 7751
    7751 7816
CMOS_STATA 7800
    7800 7842
CMOS_STATB 7801
    7801 7835
CMOS_UIP 7802
    7802 7842
COM1 8814
    8814 8824 8827 8828 8829
    8830 8831 8832 8835 8841
    8842 8857 8859 8867 8869
commit 5051
    4903 5023 5051
CONSOLE 4287
    4287 8728 8729
consoleinit 8724
    0269 1327 8724
consoleintr 8627
    0271 8348 8627 8875
consoleread 8671
    8671 8729
consolewrite 8709
    8709 8728
consputc 8601
    8367 8398 8468 8486 8489
    8493 8494 8601 8642 8648
    8655 8716
context 2341
    0251 0376 2303 2341 2372
    2505 2506 2507 2508 2856
    2894 3078
CONV 7852
    7852 7853 7854 7855 7856
    7857 7858 7859
copyout 2118
    0437 2118 6835 6845 6856
copyuvm 2053
    0434 2053 2064 2066 2635
cprintf 8452
    0270 1324 1364 1967 1973
    3076 3080 3082 3540 3553
    3558 3788 5294 5672 5674
    5676 7713 7962 8452 8512
    8513 8514 8517
cpu 2301
    0311 1364 1366 1378 1506
    1566 1590 1608 1647 1661
    1662 1663 1671 1673 1717
    1730 1736 1883 1884 1885
    1886 1889 2301 2312 2316
    2327 2473 2513 2527 2529
    2856 2887 2893 2894 2895
```



```
    3540 3553 3558 7363 7713
    8512
cpunum 7701
    0326 1324 1364 1388 1723
    3515 3541 3554 3560 7701
    7973 7982
CR0_PE 0727
    0727 1237 1270 9743
CR0_PG 0737
    0737 1154 1270
CR0_WP 0733
    0733 1154 1270
CR4_PSE 0739
    0739 1147 1263
create 6507
    6507 6527 6540 6544 6564
    6607 6627
CRTPORT 8551
    8551 8560 8561 8562 8563
    8581 8582 8583 8584
CTL 8159
    8159 8185 8189 8335
DAY 7807
    7807 7824
deallocuvm 1987
    0430 1968 1974 1987 2021
    2612
DEVSPACE 0204
    0204 1832 1845
devsw 4280
    4280 4285 5658 5660 5708
    5710 6012 8728 8729
dinode 4178
    4178 4201 5357 5361 5379
    5382 5456 5465
dirent 4215
    4215 5764 5805 6405 6454
dirlink 5802
    0288 5771 5802 5817 5825
    6380 6539 6543 6544
dirlookup 5761
    0289 5761 5767 5809 5925
    6473 6517
DIRSIZ 4213
    4213 4217 5755 5822 5878
    5879 5942 6354 6455 6511
DPL_USER 0829
    0829 1726 1727 2566 2567
    3473 3568 3577
DYNAMIC 2412
    2412 2480
E0ESC 8166
    8166 8320 8324 8325 8327
    8330
elfhdr 1005
    1005 6780 9819 9824
ELF_MAGIC 1002
    1002 6797 9830
ELF_PROG_LOAD 1036
    1036 6808
end_op 5003
    0337 2691 5003 6085 6179
    6362 6369 6387 6396 6463
    6497 6502 6566 6571 6577
    6586 6590 6608 6612 6628
    6632 6658 6664 6669 6788
    6822 6880
entry 1144
    1011 1140 1143 1144 3352
    3353 6762 6764 6851 7221
    9821 9845 9846
EOI 7567
    7567 7684 7733
ERROR 7588
    7588 7677
ESR 7570
    7570 7680 7681
exec 6774
    0275 6697 6774 6835 8968
    9029 9030 9206 9207
EXEC 9060
    9060 9200 9359 9665
execcmd 9072 9353
    9072 9124 9186 9201 9353
    9355 9621 9627 9628 9656
    9666
exit 2673
    0358 2673 2712 3505 3509
    3569 3578 3822 8917 8920
    8961 9026 9031 9192 9203
    9215 9258 9299 9306
EXTMEM 0202
    0202 0208 1829
fdalloc 6253
    6253 6274 6582 6712
fetchint 3667
    0406 3667 3697 6688
fetchstr 3679
    0407 3679 3726 6694
file 4250
```

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

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



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

```
0250 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 sleeplock;
0259 struct stat;
0260 struct superblock;
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 void            mpinit(void);
0342
0343 // picirq.c
0344 void            picenable(int);
0345 void            picinit(void);
0346
0347
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 void            exit(int status);
0359 int             fork(void);
0360 int             growproc(int);
0361 int             kill(int);
0362 void            pinit(void);
0363 void            procdump(void);
0364 void            scheduler(void) __attribute__((noreturn));
0365 void            sched(void);
0366 void            sleep(void*, struct spinlock*);
0367 void            userinit(void);
0368 int             wait(int *status);
0369 void            wakeup(void*);
0370 void            yield(void);
0371 int                       priority(int);
0372 int                       policy(int);
0373
0374
0375 // swtch.S
0376 void            swtch(struct context**, struct context*);
0377
0378 // spinlock.c
0379 void            acquire(struct spinlock*);
0380 void            getcallerpcs(void*, uint*);
0381 int             holding(struct spinlock*);
0382 void            initlock(struct spinlock*, char*);
0383 void            release(struct spinlock*);
0384 void            pushcli(void);
0385 void            popcli(void);
0386
0387 // sleeplock.c
0388 void            acquiresleep(struct sleeplock*);
0389 void            releasesleep(struct sleeplock*);
0390 int             holdingsleep(struct sleeplock*);
0391 void            initsleeplock(struct sleeplock*, char*);
0392
0393 // string.c
0394 int             memcmp(const void*, const void*, uint);
0395 void*           memmove(void*, const void*, uint);
0396 void*           memset(void*, int, uint);
0397 char*           safestrcpy(char*, const char*, int);
0398 int             strlen(const char*);
0399 int             strncmp(const char*, const char*, uint);
```



```
0400 char*           strncpy(char*, const char*, int);
0401
0402 // syscall.c
0403 int             argint(int, int*);
0404 int             argptr(int, char**, int);
0405 int             argstr(int, char**);
0406 int             fetchint(uint, int*);
0407 int             fetchstr(uint, char**);
0408 void            syscall(void);
0409
0410 // timer.c
0411 void            timerinit(void);
0412
0413 // trap.c
0414 void            idtinit(void);
0415 extern uint     ticks;
0416 void            tvinit(void);
0417 extern struct spinlock tickslock;
0418
0419 // uart.c
0420 void            uartinit(void);
0421 void            uartintr(void);
0422 void            uartputc(int);
0423
0424 // vm.c
0425 void            seginit(void);
0426 void            kvmalloc(void);
0427 pde_t*          setupkvm(void);
0428 char*           uva2ka(pde_t*, char*);
0429 int             allocuvm(pde_t*, uint, uint);
0430 int             deallocuvm(pde_t*, uint, uint);
0431 void            freevm(pde_t*);
0432 void            inituvm(pde_t*, char*, uint);
0433 int             loaduvm(pde_t*, char*, struct inode*, uint, uint);
0434 pde_t*          copyuvm(pde_t*, uint);
0435 void            switchuvm(struct proc*);
0436 void            switchkvm(void);
0437 int             copyout(pde_t*, uint, void*, uint);
0438 void            clearpteu(pde_t *pgdir, char *uva);
0439
0440 // number of elements in fixed-size array
0441 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
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 // various segment selectors.
0742 #define SEG_KCODE 1  // kernel code
0743 #define SEG_KDATA 2  // kernel data+stack
0744 #define SEG_KCPU  3  // kernel per-cpu data
0745 #define SEG_UCODE 4  // user code
0746 #define SEG_UDATA 5  // user data+stack
0747 #define SEG_TSS   6  // this process's task state
0748
0749
```

```
0750 // cpu->gdt[NSEGS] holds the above segments.
0751 #define NSEGS     7
0752
0753
0754
0755
0756
0757
0758
0759
0760
0761
0762
0763
0764
0765
0766
0767
0768
0769
0770
0771
0772
0773
0774
0775
0776
0777
0778
0779
0780
0781
0782
0783
0784
0785
0786
0787
0788
0789
0790
0791
0792
0793
0794
0795
0796
0797
0798
0799
```



```
0800 #ifndef __ASSEMBLER__
0801 // Segment Descriptor
0802 struct segdesc {
0803   uint lim_15_0 : 16;  // Low bits of segment limit
0804   uint base_15_0 : 16; // Low bits of segment base address
0805   uint base_23_16 : 8; // Middle bits of segment base address
0806   uint type : 4;       // Segment type (see STS_ constants)
0807   uint s : 1;          // 0 = system, 1 = application
0808   uint dpl : 2;        // Descriptor Privilege Level
0809   uint p : 1;          // Present
0810   uint lim_19_16 : 4;  // High bits of segment limit
0811   uint avl : 1;        // Unused (available for software use)
0812   uint rsv1 : 1;       // Reserved
0813   uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
0814   uint g : 1;          // Granularity: limit scaled by 4K when set
0815   uint base_31_24 : 8; // High bits of segment base address
0816 };
0817
0818 // Normal segment
0819 #define SEG(type, base, lim, dpl) (struct segdesc)    \
0820 { ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff,      \
0821   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
0822   (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
0823 #define SEG16(type, base, lim, dpl) (struct segdesc)  \
0824 { (lim) & 0xffff, (uint)(base) & 0xffff,              \
0825   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
0826   (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
0827 #endif
0828
0829 #define DPL_USER    0x3     // User DPL
0830
0831 // Application segment type bits
0832 #define STA_X       0x8     // Executable segment
0833 #define STA_E       0x4     // Expand down (non-executable segments)
0834 #define STA_C       0x4     // Conforming code segment (executable only)
0835 #define STA_W       0x2     // Writeable (non-executable segments)
0836 #define STA_R       0x2     // Readable (executable segments)
0837 #define STA_A       0x1     // Accessed
0838
0839 // System segment type bits
0840 #define STS_T16A    0x1     // Available 16-bit TSS
0841 #define STS_LDT     0x2     // Local Descriptor Table
0842 #define STS_T16B    0x3     // Busy 16-bit TSS
0843 #define STS_CG16    0x4     // 16-bit Call Gate
0844 #define STS_TG      0x5     // Task Gate / Coum Transmitions
0845 #define STS_IG16    0x6     // 16-bit Interrupt Gate
0846 #define STS_TG16    0x7     // 16-bit Trap Gate
0847 #define STS_T32A    0x9     // Available 32-bit TSS
0848 #define STS_T32B    0xB     // Busy 32-bit TSS
0849 #define STS_CG32    0xC     // 32-bit Call Gate
```

```
0850 #define STS_IG32    0xE     // 32-bit Interrupt Gate
0851 #define STS_TG32    0xF     // 32-bit Trap Gate
0852
0853 // A virtual address 'la' has a three-part structure as follows:
0854 //
0855 // +--------10------+-------10-------+---------12----------+
0856 // | Page Directory |   Page Table   | Offset within Page  |
0857 // |      Index     |      Index     |                     |
0858 // +----------------+----------------+---------------------+
0859 //  \--- PDX(va) --/ \--- PTX(va) --/
0860
0861 // page directory index
0862 #define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)
0863
0864 // page table index
0865 #define PTX(va)         (((uint)(va) >> PTXSHIFT) & 0x3FF)
0866
0867 // construct virtual address from indexes and offset
0868 #define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
0869
0870 // Page directory and page table constants.
0871 #define NPDENTRIES      1024    // # directory entries per page directory
0872 #define NPTENTRIES      1024    // # PTEs per page table
0873 #define PGSIZE          4096    // bytes mapped by a page
0874
0875 #define PGSHIFT         12      // log2(PGSIZE)
0876 #define PTXSHIFT        12      // offset of PTX in a linear address
0877 #define PDXSHIFT        22      // offset of PDX in a linear address
0878
0879 #define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
0880 #define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
0881
0882 // Page table/directory entry flags.
0883 #define PTE_P           0x001   // Present
0884 #define PTE_W           0x002   // Writeable
0885 #define PTE_U           0x004   // User
0886 #define PTE_PWT         0x008   // Write-Through
0887 #define PTE_PCD         0x010   // Cache-Disable
0888 #define PTE_A           0x020   // Accessed
0889 #define PTE_D           0x040   // Dirty
0890 #define PTE_PS          0x080   // Page Size
0891 #define PTE_MBZ         0x180   // Bits must be zero
0892
0893 // Address in page table or page directory entry
0894 #define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
0895 #define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF)
0896
0897 #ifndef __ASSEMBLER__
0898 typedef uint pte_t;
0899
```



```
0900 // Task state segment format
0901 struct taskstate {
0902   uint link;         // Old ts selector
0903   uint esp0;         // Stack pointers and segment selectors
0904   ushort ss0;        //   after an increase in privilege level
0905   ushort padding1;
0906   uint *esp1;
0907   ushort ss1;
0908   ushort padding2;
0909   uint *esp2;
0910   ushort ss2;
0911   ushort padding3;
0912   void *cr3;         // Page directory base
0913   uint *eip;         // Saved state from last task switch
0914   uint eflags;
0915   uint eax;          // More saved state (registers)
0916   uint ecx;
0917   uint edx;
0918   uint ebx;
0919   uint *esp;
0920   uint *ebp;
0921   uint esi;
0922   uint edi;
0923   ushort es;         // Even more saved state (segment selectors)
0924   ushort padding4;
0925   ushort cs;
0926   ushort padding5;
0927   ushort ss;
0928   ushort padding6;
0929   ushort ds;
0930   ushort padding7;
0931   ushort fs;
0932   ushort padding8;
0933   ushort gs;
0934   ushort padding9;
0935   ushort ldt;
0936   ushort padding10;
0937   ushort t;          // Trap on task switch
0938   ushort iomb;       // I/O map base address
0939 };
0940
0941
0942
0943
0944
0945
0946
0947
0948
0949
```

```
0950 // Gate descriptors for interrupts and traps
0951 struct gatedesc {
0952   uint off_15_0 : 16;   // low 16 bits of offset in segment
0953   uint cs : 16;         // code segment selector
0954   uint args : 5;        // # args, 0 for interrupt/trap gates
0955   uint rsv1 : 3;        // reserved(should be zero I guess)
0956   uint type : 4;        // type(STS_{TG,IG32,TG32})
0957   uint s : 1;           // must be 0 (system)
0958   uint dpl : 2;         // descriptor(meaning new) privilege level
0959   uint p : 1;           // Present
0960   uint off_31_16 : 16;  // high bits of offset in segment
0961 };
0962
0963 // Set up a normal interrupt/trap gate descriptor.
0964 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
0965 //   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
0966 // - sel: Code segment selector for interrupt/trap handler
0967 // - off: Offset in code segment for interrupt/trap handler
0968 // - dpl: Descriptor Privilege Level -
0969 //        the privilege level required for software to invoke
0970 //        this interrupt/trap gate explicitly using an int instruction.
0971 #define SETGATE(gate, istrap, sel, off, d)                \
0972 {                                                         \
0973   (gate).off_15_0 = (uint)(off) & 0xffff;                \
0974   (gate).cs = (sel);                                      \
0975   (gate).args = 0;                                        \
0976   (gate).rsv1 = 0;                                        \
0977   (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
0978   (gate).s = 0;                                           \
0979   (gate).dpl = (d);                                       \
0980   (gate).p = 1;                                           \
0981   (gate).off_31_16 = (uint)(off) >> 16;                  \
0982 }
0983
0984 #endif
0985
0986
0987
0988
0989
0990
0991
0992
0993
0994
0995
0996
0997
0998
0999
```



```
1000 // Format of an ELF executable file
1001
1002 #define ELF_MAGIC 0x464C457FU  // "\x7FELF" in little endian
1003
1004 // File header
1005 struct elfhdr {
1006   uint magic;  // must equal ELF_MAGIC
1007   uchar elf[12];
1008   ushort type;
1009   ushort machine;
1010   uint version;
1011   uint entry;
1012   uint phoff;
1013   uint shoff;
1014   uint flags;
1015   ushort ehsize;
1016   ushort phentsize;
1017   ushort phnum;
1018   ushort shentsize;
1019   ushort shnum;
1020   ushort shstrndx;
1021 };
1022
1023 // Program section header
1024 struct proghdr {
1025   uint type;
1026   uint off;
1027   uint vaddr;
1028   uint paddr;
1029   uint filesz;
1030   uint memsz;
1031   uint flags;
1032   uint align;
1033 };
1034
1035 // Values for Proghdr type
1036 #define ELF_PROG_LOAD           1
1037
1038 // Flag bits for Proghdr flags
1039 #define ELF_PROG_FLAG_EXEC      1
1040 #define ELF_PROG_FLAG_WRITE     2
1041 #define ELF_PROG_FLAG_READ      4
1042
1043
1044
1045
1046
1047
1048
1049
```

```
1050 // Blank page.
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
```



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

```
1150   movl    $(V2P_WO(entrypgdir)), %eax
1151   movl    %eax, %cr3
1152   # Turn on paging.
1153   movl    %cr0, %eax
1154   orl     $(CR0_PG|CR0_WP), %eax
1155   movl    %eax, %cr0
1156
1157   # Set up the stack pointer.
1158   movl $(stack + KSTACKSIZE), %esp
1159
1160   # Jump to main(), and switch to executing at
1161   # high addresses. The indirect call is needed because
1162   # the assembler produces a PC-relative instruction
1163   # for a direct jump.
1164   mov $main, %eax
1165   jmp *%eax
1166
1167 .comm stack, KSTACKSIZE
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
```



```
1200 #include "asm.h"
1201 #include "memlayout.h"
1202 #include "mmu.h"
1203
1204 # Each non-boot CPU ("AP") is started up in response to a STARTUP
1205 # IPI from the boot CPU.  Section B.4.2 of the Multi-Processor
1206 # Specification says that the AP will start in real mode with CS:IP
1207 # set to XY00:0000, where XY is an 8-bit value sent with the
1208 # STARTUP. Thus this code must start at a 4096-byte boundary.
1209 #
1210 # Because this code sets DS to zero, it must sit
1211 # at an address in the low 2^16 bytes.
1212 #
1213 # Startothers (in main.c) sends the STARTUPs one at a time.
1214 # It copies this code (start) at 0x7000.  It puts the address of
1215 # a newly allocated per-core stack in start-4,the address of the
1216 # place to jump to (mpenter) in start-8, and the physical address
1217 # of entrypgdir in start-12.
1218 #
1219 # This code combines elements of bootasm.S and entry.S.
1220
1221 .code16
1222 .globl start
1223 start:
1224   cli
1225
1226   # Zero data segment registers DS, ES, and SS.
1227   xorw    %ax,%ax
1228   movw    %ax,%ds
1229   movw    %ax,%es
1230   movw    %ax,%ss
1231
1232   # Switch from real to protected mode.  Use a bootstrap GDT that makes
1233   # virtual addresses map directly to physical addresses so that the
1234   # effective memory map doesn't change during the transition.
1235   lgdt    gdtdesc
1236   movl    %cr0, %eax
1237   orl     $CR0_PE, %eax
1238   movl    %eax, %cr0
1239
1240   # Complete the transition to 32-bit protected mode by using a long jmp
1241   # to reload %cs and %eip.  The segment descriptors are set up with no
1242   # translation, so that the mapping is still the identity mapping.
1243   ljmpl    $(SEG_KCODE<<3), $(start32)
1244
1245
1246
1247
1248
1249
```

```
1250 .code32  # Tell assembler to generate 32-bit code now.
1251 start32:
1252   # Set up the protected-mode data segment registers
1253   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
1254   movw    %ax, %ds                # -> DS: Data Segment
1255   movw    %ax, %es                # -> ES: Extra Segment
1256   movw    %ax, %ss                # -> SS: Stack Segment
1257   movw    $0, %ax                 # Zero segments not ready for use
1258   movw    %ax, %fs                # -> FS
1259   movw    %ax, %gs                # -> GS
1260
1261   # Turn on page size extension for 4Mbyte pages
1262   movl    %cr4, %eax
1263   orl     $(CR4_PSE), %eax
1264   movl    %eax, %cr4
1265   # Use entrypgdir as our initial page table
1266   movl    (start-12), %eax
1267   movl    %eax, %cr3
1268   # Turn on paging.
1269   movl    %cr0, %eax
1270   orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
1271   movl    %eax, %cr0
1272
1273   # Switch to the stack allocated by startothers()
1274   movl    (start-4), %esp
1275   # Call mpenter()
1276   call      *(start-8)
1277
1278   movw    $0x8a00, %ax
1279   movw    %ax, %dx
1280   outw    %ax, %dx
1281   movw    $0x8ae0, %ax
1282   outw    %ax, %dx
1283 spin:
1284   jmp     spin
1285
1286 .p2align 2
1287 gdt:
1288   SEG_NULLASM
1289   SEG_ASM(STA_X|STA_R, 0, 0xffffffff)
1290   SEG_ASM(STA_W, 0, 0xffffffff)
1291
1292
1293 gdtdesc:
1294   .word   (gdtdesc - gdt - 1)
1295   .long   gdt
1296
1297
1298
1299
```



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

```
1350 // Other CPUs jump here from entryother.S.
1351 static void
1352 mpenter(void)
1353 {
1354   switchkvm();
1355   seginit();
1356   lapicinit();
1357   mpmain();
1358 }
1359
1360 // Common CPU setup code.
1361 static void
1362 mpmain(void)
1363 {
1364   cprintf("cpu%d: starting\n", cpunum());
1365   idtinit();       // load idt register
1366   xchg(&cpu->started, 1); // tell startothers() we're up
1367   scheduler();     // start running processes
1368 }
1369
1370 pde_t entrypgdir[];  // For entry.S
1371
1372 // Start the non-boot (AP) processors.
1373 static void
1374 startothers(void)
1375 {
1376   extern uchar _binary_entryother_start[], _binary_entryother_size[];
1377   uchar *code;
1378   struct cpu *c;
1379   char *stack;
1380
1381   // Write entry code to unused memory at 0x7000.
1382   // The linker has placed the image of entryother.S in
1383   // _binary_entryother_start.
1384   code = P2V(0x7000);
1385   memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
1386
1387   for(c = cpus; c < cpus+ncpu; c++){
1388     if(c == cpus+cpunum())  // We've started already.
1389       continue;
1390
1391     // Tell entryother.S what stack to use, where to enter, and what
1392     // pgdir to use. We cannot use kpgdir yet, because the AP processor
1393     // is running in low  memory, so we use entrypgdir for the APs too.
1394     stack = kalloc();
1395     *(void**)(code-4) = stack + KSTACKSIZE;
1396     *(void**)(code-8) = mpenter;
1397     *(int**)(code-12) = (void *) V2P(entrypgdir);
1398
1399     lapicstartap(c->apicid, V2P(code));
```



```
1400     // wait for cpu to finish mpmain()
1401     while(c->started == 0)
1402       ;
1403   }
1404 }
1405
1406 // The boot page table used in entry.S and entryother.S.
1407 // Page directories (and page tables) must start on page boundaries,
1408 // hence the __aligned__ attribute.
1409 // PTE_PS in a page directory entry enables 4Mbyte pages.
1410
1411 __attribute__((__aligned__(PGSIZE)))
1412 pde_t entrypgdir[NPDENTRIES] = {
1413   // Map VA's [0, 4MB) to PA's [0, 4MB)
1414   [0] = (0) | PTE_P | PTE_W | PTE_PS,
1415   // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1416   [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1417 };
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
```

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



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

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



```
1600 // Release the lock.
1601 void
1602 release(struct spinlock *lk)
1603 {
1604   if(!holding(lk))
1605     panic("release");
1606
1607   lk->pcs[0] = 0;
1608   lk->cpu = 0;
1609
1610   // Tell the C compiler and the processor to not move loads or stores
1611   // past this point, to ensure that all the stores in the critical
1612   // section are visible to other cores before the lock is released.
1613   // Both the C compiler and the hardware may re-order loads and
1614   // stores; __sync_synchronize() tells them both not to.
1615   __sync_synchronize();
1616
1617   // Release the lock, equivalent to lk->locked = 0.
1618   // This code can't use a C assignment, since it might
1619   // not be atomic. A real OS would use C atomics here.
1620   asm volatile("movl $0, %0" : "+m" (lk->locked) : );
1621
1622   popcli();
1623 }
1624
1625 // Record the current call stack in pcs[] by following the %ebp chain.
1626 void
1627 getcallerpcs(void *v, uint pcs[])
1628 {
1629   uint *ebp;
1630   int i;
1631
1632   ebp = (uint*)v - 2;
1633   for(i = 0; i < 10; i++){
1634     if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
1635       break;
1636     pcs[i] = ebp[1];     // saved %eip
1637     ebp = (uint*)ebp[0]; // saved %ebp
1638   }
1639   for(; i < 10; i++)
1640     pcs[i] = 0;
1641 }
1642
1643 // Check whether this cpu is holding the lock.
1644 int
1645 holding(struct spinlock *lock)
1646 {
1647   return lock->locked && lock->cpu == cpu;
1648 }
1649
```

```
1650 // Pushcli/popcli are like cli/sti except that they are matched:
1651 // it takes two popcli to undo two pushcli.  Also, if interrupts
1652 // are off, then pushcli, popcli leaves them off.
1653
1654 void
1655 pushcli(void)
1656 {
1657   int eflags;
1658
1659   eflags = readeflags();
1660   cli();
1661   if(cpu->ncli == 0)
1662     cpu->intena = eflags & FL_IF;
1663   cpu->ncli += 1;
1664 }
1665
1666 void
1667 popcli(void)
1668 {
1669   if(readeflags()&FL_IF)
1670     panic("popcli - interruptible");
1671   if(--cpu->ncli < 0)
1672     panic("popcli");
1673   if(cpu->ncli == 0 && cpu->intena)
1674     sti();
1675 }
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
```



```
1700 #include "param.h"
1701 #include "types.h"
1702 #include "defs.h"
1703 #include "x86.h"
1704 #include "memlayout.h"
1705 #include "mmu.h"
1706 #include "proc.h"
1707 #include "elf.h"
1708
1709 extern char data[];  // defined by kernel.ld
1710 pde_t *kpgdir;  // for use in scheduler()
1711
1712 // Set up CPU's kernel segment descriptors.
1713 // Run once on entry on each CPU.
1714 void
1715 seginit(void)
1716 {
1717   struct cpu *c;
1718
1719   // Map "logical" addresses to virtual addresses using identity map.
1720   // Cannot share a CODE descriptor for both kernel and user
1721   // because it would have to have DPL_USR, but the CPU forbids
1722   // an interrupt from CPL=0 to DPL=3.
1723   c = &cpus[cpunum()];
1724   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
1725   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1726   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
1727   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1728
1729   // Map cpu and proc -- these are private per cpu.
1730   c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
1731
1732   lgdt(c->gdt, sizeof(c->gdt));
1733   loadgs(SEG_KCPU << 3);
1734
1735   // Initialize cpu-local storage.
1736   cpu = c;
1737   proc = 0;
1738 }
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
```

```
1750 // Return the address of the PTE in page table pgdir
1751 // that corresponds to virtual address va.  If alloc!=0,
1752 // create any required page table pages.
1753 static pte_t *
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756   pde_t *pde;
1757   pte_t *pgtab;
1758
1759   pde = &pgdir[PDX(va)];
1760   if(*pde & PTE_P){
1761     pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1762   } else {
1763     if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1764       return 0;
1765     // Make sure all those PTE_P bits are zero.
1766     memset(pgtab, 0, PGSIZE);
1767     // The permissions here are overly generous, but they can
1768     // be further restricted by the permissions in the page table
1769     // entries, if necessary.
1770     *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1771   }
1772   return &pgtab[PTX(va)];
1773 }
1774
1775 // Create PTEs for virtual addresses starting at va that refer to
1776 // physical addresses starting at pa. va and size might not
1777 // be page-aligned.
1778 static int
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781   char *a, *last;
1782   pte_t *pte;
1783
1784   a = (char*)PGROUNDDOWN((uint)va);
1785   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786   for(;;){
1787     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788       return -1;
1789     if(*pte & PTE_P)
1790       panic("remap");
1791     *pte = pa | perm | PTE_P;
1792     if(a == last)
1793       break;
1794     a += PGSIZE;
1795     pa += PGSIZE;
1796   }
1797   return 0;
1798 }
1799
```



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

```
1850       return 0;
1851   return pgdir;
1852 }
1853
1854 // Allocate one page table for the machine for the kernel address
1855 // space for scheduler processes.
1856 void
1857 kvmalloc(void)
1858 {
1859   kpgdir = setupkvm();
1860   switchkvm();
1861 }
1862
1863 // Switch h/w page table register to the kernel-only page table,
1864 // for when no process is running.
1865 void
1866 switchkvm(void)
1867 {
1868   lcr3(V2P(kpgdir));   // switch to the kernel page table
1869 }
1870
1871 // Switch TSS and h/w page table to correspond to process p.
1872 void
1873 switchuvm(struct proc *p)
1874 {
1875   if(p == 0)
1876     panic("switchuvm: no process");
1877   if(p->kstack == 0)
1878     panic("switchuvm: no kstack");
1879   if(p->pgdir == 0)
1880     panic("switchuvm: no pgdir");
1881
1882   pushcli();
1883   cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
1884   cpu->gdt[SEG_TSS].s = 0;
1885   cpu->ts.ss0 = SEG_KDATA << 3;
1886   cpu->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
1887   // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
1888   // forbids I/O instructions (e.g., inb and outb) from user space
1889   cpu->ts.iomb = (ushort) 0xFFFF;
1890   ltr(SEG_TSS << 3);
1891   lcr3(V2P(p->pgdir));  // switch to process's address space
1892   popcli();
1893 }
1894
1895
1896
1897
1898
1899
```



```
1900 // Load the initcode into address 0 of pgdir.
1901 // sz must be less than a page.
1902 void
1903 inituvm(pde_t *pgdir, char *init, uint sz)
1904 {
1905   char *mem;
1906
1907   if(sz >= PGSIZE)
1908     panic("inituvm: more than a page");
1909   mem = kalloc();
1910   memset(mem, 0, PGSIZE);
1911   mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W|PTE_U);
1912   memmove(mem, init, sz);
1913 }
1914
1915 // Load a program segment into pgdir.  addr must be page-aligned
1916 // and the pages from addr to addr+sz must already be mapped.
1917 int
1918 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
1919 {
1920   uint i, pa, n;
1921   pte_t *pte;
1922
1923   if((uint) addr % PGSIZE != 0)
1924     panic("loaduvm: addr must be page aligned");
1925   for(i = 0; i < sz; i += PGSIZE){
1926     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1927       panic("loaduvm: address should exist");
1928     pa = PTE_ADDR(*pte);
1929     if(sz - i < PGSIZE)
1930       n = sz - i;
1931     else
1932       n = PGSIZE;
1933     if(readi(ip, P2V(pa), offset+i, n) != n)
1934       return -1;
1935   }
1936   return 0;
1937 }
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
```

```
1950 // Allocate page tables and physical memory to grow process from oldsz to
1951 // newsz, which need not be page aligned.  Returns new size or 0 on error.
1952 int
1953 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1954 {
1955   char *mem;
1956   uint a;
1957
1958   if(newsz >= KERNBASE)
1959     return 0;
1960   if(newsz < oldsz)
1961     return oldsz;
1962
1963   a = PGROUNDUP(oldsz);
1964   for(; a < newsz; a += PGSIZE){
1965     mem = kalloc();
1966     if(mem == 0){
1967       cprintf("allocuvm out of memory\n");
1968       deallocuvm(pgdir, newsz, oldsz);
1969       return 0;
1970     }
1971     memset(mem, 0, PGSIZE);
1972     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
1973       cprintf("allocuvm out of memory (2)\n");
1974       deallocuvm(pgdir, newsz, oldsz);
1975       kfree(mem);
1976       return 0;
1977     }
1978   }
1979   return newsz;
1980 }
1981
1982 // Deallocate user pages to bring the process size from oldsz to
1983 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
1984 // need to be less than oldsz.  oldsz can be larger than the actual
1985 // process size.  Returns the new process size.
1986 int
1987 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1988 {
1989   pte_t *pte;
1990   uint a, pa;
1991
1992   if(newsz >= oldsz)
1993     return oldsz;
1994
1995   a = PGROUNDUP(newsz);
1996   for(; a  < oldsz; a += PGSIZE){
1997     pte = walkpgdir(pgdir, (char*)a, 0);
1998     if(!pte)
1999       a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
```



```
2000     else if((*pte & PTE_P) != 0){
2001       pa = PTE_ADDR(*pte);
2002       if(pa == 0)
2003         panic("kfree");
2004       char *v = P2V(pa);
2005       kfree(v);
2006       *pte = 0;
2007     }
2008   }
2009   return newsz;
2010 }
2011
2012 // Free a page table and all the physical memory pages
2013 // in the user part.
2014 void
2015 freevm(pde_t *pgdir)
2016 {
2017   uint i;
2018
2019   if(pgdir == 0)
2020     panic("freevm: no pgdir");
2021   deallocuvm(pgdir, KERNBASE, 0);
2022   for(i = 0; i < NPDENTRIES; i++){
2023     if(pgdir[i] & PTE_P){
2024       char * v = P2V(PTE_ADDR(pgdir[i]));
2025       kfree(v);
2026     }
2027   }
2028   kfree((char*)pgdir);
2029 }
2030
2031 // Clear PTE_U on a page. Used to create an inaccessible
2032 // page beneath the user stack.
2033 void
2034 clearpteu(pde_t *pgdir, char *uva)
2035 {
2036   pte_t *pte;
2037
2038   pte = walkpgdir(pgdir, uva, 0);
2039   if(pte == 0)
2040     panic("clearpteu");
2041   *pte &= ~PTE_U;
2042 }
2043
2044
2045
2046
2047
2048
2049
```

```
2050 // Given a parent process's page table, create a copy
2051 // of it for a child.
2052 pde_t*
2053 copyuvm(pde_t *pgdir, uint sz)
2054 {
2055   pde_t *d;
2056   pte_t *pte;
2057   uint pa, i, flags;
2058   char *mem;
2059
2060   if((d = setupkvm()) == 0)
2061     return 0;
2062   for(i = 0; i < sz; i += PGSIZE){
2063     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
2064       panic("copyuvm: pte should exist");
2065     if(!(*pte & PTE_P))
2066       panic("copyuvm: page not present");
2067     pa = PTE_ADDR(*pte);
2068     flags = PTE_FLAGS(*pte);
2069     if((mem = kalloc()) == 0)
2070       goto bad;
2071     memmove(mem, (char*)P2V(pa), PGSIZE);
2072     if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
2073       goto bad;
2074   }
2075   return d;
2076
2077 bad:
2078   freevm(d);
2079   return 0;
2080 }
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
```



```
2100 // Map user virtual address to kernel address.
2101 char*
2102 uva2ka(pde_t *pgdir, char *uva)
2103 {
2104   pte_t *pte;
2105
2106   pte = walkpgdir(pgdir, uva, 0);
2107   if((*pte & PTE_P) == 0)
2108     return 0;
2109   if((*pte & PTE_U) == 0)
2110     return 0;
2111   return (char*)P2V(PTE_ADDR(*pte));
2112 }
2113
2114 // Copy len bytes from p to user address va in page table pgdir.
2115 // Most useful when pgdir is not the current page table.
2116 // uva2ka ensures this only works for PTE_U pages.
2117 int
2118 copyout(pde_t *pgdir, uint va, void *p, uint len)
2119 {
2120   char *buf, *pa0;
2121   uint n, va0;
2122
2123   buf = (char*)p;
2124   while(len > 0){
2125     va0 = (uint)PGROUNDDOWN(va);
2126     pa0 = uva2ka(pgdir, (char*)va0);
2127     if(pa0 == 0)
2128       return -1;
2129     n = PGSIZE - (va - va0);
2130     if(n > len)
2131       n = len;
2132     memmove(pa0 + (va - va0), buf, n);
2133     len -= n;
2134     buf += n;
2135     va = va0 + PGSIZE;
2136   }
2137   return 0;
2138 }
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
```

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



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

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



```
2300 // Per-CPU state
2301 struct cpu {
2302   uchar apicid;                // Local APIC ID
2303   struct context *scheduler;   // swtch() here to enter scheduler
2304   struct taskstate ts;         // Used by x86 to find stack for interrupt
2305   struct segdesc gdt[NSEGS];   // x86 global descriptor table
2306   volatile uint started;       // Has the CPU started?
2307   int ncli;                    // Depth of pushcli nesting.
2308   int intena;                  // Were interrupts enabled before pushcli?
2309   int policy;
2310
2311   // Cpu-local storage variables; see below
2312   struct cpu *cpu;
2313   struct proc *proc;           // The currently-running process.
2314 };
2315
2316 extern struct cpu cpus[NCPU];
2317 extern int ncpu;
2318
2319 // Per-CPU variables, holding pointers to the
2320 // current cpu and to the current process.
2321 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2322 // and "%gs:4" to refer to proc.  seginit sets up the
2323 // %gs segment register so that %gs refers to the memory
2324 // holding those two variables in the local cpu's struct cpu.
2325 // This is similar to how thread-local variables are implemented
2326 // in thread libraries such as Linux pthreads.
2327 extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
2328 extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc
2329
2330
2331 // Saved registers for kernel context switches.
2332 // Don't need to save all the segment registers (%cs, etc),
2333 // because they are constant across kernel contexts.
2334 // Don't need to save %eax, %ecx, %edx, because the
2335 // x86 convention is that the caller has saved them.
2336 // Contexts are stored at the bottom of the stack they
2337 // describe; the stack pointer is the address of the context.
2338 // The layout of the context matches the layout of the stack in swtch.S
2339 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2340 // but it is on the stack and allocproc() manipulates it.
2341 struct context {
2342   uint edi;
2343   uint esi;
2344   uint ebx;
2345   uint ebp;
2346   uint eip;
2347 };
2348
2349
```

```
2350 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2351
2352
2353 struct perf {
2354   int ctime;
2355   int ttime;
2356   int stime;
2357   int retime;
2358   int rutime;
2359 };
2360
2361
2362
2363 // Per-process state
2364 struct proc {
2365   uint sz;                     // Size of process memory (bytes)
2366   pde_t* pgdir;                // Page table
2367   char *kstack;                // Bottom of kernel stack for this process
2368   enum procstate state;        // Process state
2369   int pid;                     // Process ID
2370   struct proc *parent;         // Parent process
2371   struct trapframe *tf;        // Trap frame for current syscall
2372   struct context *context;     // swtch() here to run process
2373   void *chan;                  // If non-zero, sleeping on chan
2374   int killed;                  // If non-zero, have been killed
2375   struct file *ofile[NOFILE];  // Open files
2376   struct inode *cwd;           // Current directory
2377   char name[16];               // Process name (debugging)
2378   int exitStat;                 //Process exit status
2379   int ntickets;                 //number of tickets for the process
2380   struct pref;
2381
2382
2383
2384       //      ctime      //process creation time
2385       //      ttime        //process termination time
2386       //      stime         //the time the process spent in the SLEE PING st
2387       //      retime       //the time the process spent in the READY state
2388       //      rutime         //the time the process spent in the RUNNING sta
2389
2390 };
2391
2392 // Process memory is laid out contiguously, low addresses first:
2393 //   text
2394 //   original data and bss
2395 //   fixed-size stack
2396 //   expandable heap
2397
2398
2399
```



```
2400 #include "types.h"
2401 #include "defs.h"
2402 #include "param.h"
2403 #include "memlayout.h"
2404 #include "mmu.h"
2405 #include "x86.h"
2406 #include "proc.h"
2407 #include "spinlock.h"
2408
2409
2410 #define  UNIFORM 0
2411 #define  P_SCHED 1
2412 #define  DYNAMIC 2
2413
2414
2415 struct {
2416   struct spinlock lock;
2417   struct proc proc[NPROC];
2418 } ptable;
2419
2420 static struct proc *initproc;
2421
2422 int nextpid = 1;
2423 extern void forkret(void);
2424 extern void trapret(void);
2425
2426 static void wakeup1(void *chan);
2427
2428 void
2429 pinit(void)
2430 {
2431   initlock(&ptable.lock, "ptable");
2432 }
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
```

```
2450 // Look in the process table for an UNUSED proc.
2451 // If found, change state to EMBRYO and initialize
2452 // state required to run in the kernel.
2453 // Otherwise return 0.
2454 static struct proc*
2455 allocproc(void)
2456 {
2457   struct proc *p;
2458   char *sp;
2459
2460   acquire(&ptable.lock);
2461
2462   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2463     if(p->state == UNUSED)
2464       goto found;
2465
2466   release(&ptable.lock);
2467   return 0;
2468
2469 found:
2470   p->state = EMBRYO;
2471
2472
2473   int poli = cpu->policy;
2474   if(poli == UNIFORM)
2475     p->ntickets = 1;
2476
2477   if(poli == P_SCHED)
2478     p->ntickets = 10;
2479
2480   if(poli == DYNAMIC)
2481     p->ntickets = 20;
2482
2483
2484   p->pid = nextpid++;
2485
2486   release(&ptable.lock);
2487
2488   // Allocate kernel stack.
2489   if((p->kstack = kalloc()) == 0){
2490     p->state = UNUSED;
2491     return 0;
2492   }
2493   sp = p->kstack + KSTACKSIZE;
2494
2495   // Leave room for trap frame.
2496   sp -= sizeof *p->tf;
2497   p->tf = (struct trapframe*)sp;
2498
2499
```



```
2500   // Set up new context to start executing at forkret,
2501   // which returns to trapret.
2502   sp -= 4;
2503   *(uint*)sp = (uint)trapret;
2504
2505   sp -= sizeof *p->context;
2506   p->context = (struct context*)sp;
2507   memset(p->context, 0, sizeof *p->context);
2508   p->context->eip = (uint)forkret;
2509
2510   return p;
2511 }
2512 int priority(int pri){
2513   if(pri<=0 || cpu->policy!=P_SCHED);{
2514     return -1;
2515   }
2516
2517   proc->ntickets=pri;
2518
2519   return 1;
2520 }
2521
2522
2523 int policy(int pol){
2524   if(pol< 0 || pol>2);{
2525     return -1;
2526   }
2527   if(pol == cpu->policy)
2528     return 1;
2529   cpu->policy = pol;
2530
2531   struct proc *p;
2532
2533   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2534     if(pol== UNIFORM){
2535      p->ntickets = 1;
2536     }
2537     else if(pol== P_SCHED) {
2538       p->ntickets = 10;
2539     }
2540
2541     else {
2542       p->ntickets = 20;
2543     }
2544   }
2545
2546   return 1;
2547
2548 }
2549
```

```
2550
2551 // Set up first user process.
2552 void
2553 userinit(void)
2554 {
2555   struct proc *p;
2556   extern char _binary_initcode_start[], _binary_initcode_size[];
2557
2558   p = allocproc();
2559
2560   initproc = p;
2561   if((p->pgdir = setupkvm()) == 0)
2562     panic("userinit: out of memory?");
2563   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2564   p->sz = PGSIZE;
2565   memset(p->tf, 0, sizeof(*p->tf));
2566   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2567   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2568   p->tf->es = p->tf->ds;
2569   p->tf->ss = p->tf->ds;
2570   p->tf->eflags = FL_IF;
2571   p->tf->esp = PGSIZE;
2572   p->tf->eip = 0;  // beginning of initcode.S
2573
2574   safestrcpy(p->name, "initcode", sizeof(p->name));
2575   p->cwd = namei("/");
2576
2577   // this assignment to p->state lets other cores
2578   // run this process. the acquire forces the above
2579   // writes to be visible, and the lock is also needed
2580   // because the assignment might not be atomic.
2581   acquire(&ptable.lock);
2582
2583   p->state = RUNNABLE;
2584
2585   release(&ptable.lock);
2586 }
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
```



```
2600 // Grow current process's memory by n bytes.
2601 // Return 0 on success, -1 on failure.
2602 int
2603 growproc(int n)
2604 {
2605   uint sz;
2606
2607   sz = proc->sz;
2608   if(n > 0){
2609     if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2610       return -1;
2611   } else if(n < 0){
2612     if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2613       return -1;
2614   }
2615   proc->sz = sz;
2616   switchuvm(proc);
2617   return 0;
2618 }
2619
2620 // Create a new process copying p as the parent.
2621 // Sets up stack to return as if from system call.
2622 // Caller must set state of returned proc to RUNNABLE.
2623 int
2624 fork(void)
2625 {
2626   int i, pid;
2627   struct proc *np;
2628
2629   // Allocate process.
2630   if((np = allocproc()) == 0){
2631     return -1;
2632   }
2633
2634   // Copy process state from p.
2635   if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2636     kfree(np->kstack);
2637     np->kstack = 0;
2638     np->state = UNUSED;
2639     return -1;
2640   }
2641   np->sz = proc->sz;
2642   np->parent = proc;
2643   *np->tf = *proc->tf;
2644
2645
2646   // Clear %eax so that fork returns 0 in the child.
2647   np->tf->eax = 0;
2648
2649
```

```
2650   for(i = 0; i < NOFILE; i++)
2651     if(proc->ofile[i])
2652       np->ofile[i] = filedup(proc->ofile[i]);
2653   np->cwd = idup(proc->cwd);
2654
2655   safestrcpy(np->name, proc->name, sizeof(proc->name));
2656
2657   pid = np->pid;
2658
2659   acquire(&ptable.lock);
2660
2661   np->state = RUNNABLE;
2662
2663
2664   release(&ptable.lock);
2665
2666   return pid;
2667 }
2668
2669 // Exit the current process.  Does not return.
2670 // An exited process remains in the zombie state
2671 // until its parent calls wait() to find out it exited.
2672 void
2673 exit(int status)
2674 {
2675   struct proc *p;
2676   int fd;
2677
2678   if(proc == initproc)
2679     panic("init exiting");
2680
2681   // Close all open files.
2682   for(fd = 0; fd < NOFILE; fd++){
2683     if(proc->ofile[fd]){
2684       fileclose(proc->ofile[fd]);
2685       proc->ofile[fd] = 0;
2686     }
2687   }
2688
2689   begin_op();
2690   iput(proc->cwd);
2691   end_op();
2692   proc->cwd = 0;
2693
2694
2695   acquire(&ptable.lock);
2696
2697   // Parent might be sleeping in wait().
2698   wakeup1(proc->parent);
2699
```



```
2700   // Pass abandoned children to init.
2701   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2702     if(p->parent == proc){
2703       p->parent = initproc;
2704       if(p->state == ZOMBIE)
2705         wakeup1(initproc);
2706     }
2707   }
2708    proc->exitStat=status;
2709   // Jump into the scheduler, never to return.
2710   proc->state = ZOMBIE;
2711   sched();
2712   panic("zombie exit");
2713 }
2714
2715 // Wait for a child process to exit and return its pid.
2716 // Return -1 if this process has no children.
2717 int
2718 wait(int *status)
2719 {
2720   struct proc *p;
2721   int havekids, pid;
2722
2723   acquire(&ptable.lock);
2724   for(;;){
2725     // Scan through table looking for exited children.
2726     havekids = 0;
2727     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2728       if(p->parent != proc)
2729         continue;
2730       havekids = 1;
2731       if(p->state == ZOMBIE){
2732         // Found one.
2733         pid = p->pid;
2734         kfree(p->kstack);
2735         p->kstack = 0;
2736         freevm(p->pgdir);
2737         p->pid = 0;
2738         p->parent = 0;
2739         p->name[0] = 0;
2740         p->killed = 0;
2741         p->state = UNUSED;
2742         if(status!=0)
2743           *status=(p->exitStat);
2744         release(&ptable.lock);
2745         return pid;
2746       }
2747     }
2748
2749
```

```
2750     // No point waiting if we don't have any children.
2751     if(!havekids || proc->killed){
2752       release(&ptable.lock);
2753       return -1;
2754     }
2755
2756     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2757     sleep(proc, &ptable.lock);
2758   }
2759 }
2760
2761
2762 int rando(void)
2763 {
2764   static int z1 = 12345, z2 = 12345, z3 = 12345, z4 = 12345;
2765   int b;
2766   b  = ((z1 << 6) ^ z1) >> 13;
2767   z1 = ((z1 & 4294967294U) << 18) ^ b;
2768   b  = ((z2 << 2) ^ z2) >> 27;
2769   z2 = ((z2 & 4294967288U) << 2) ^ b;
2770   b  = ((z3 << 13) ^ z3) >> 21;
2771   z3 = ((z3 & 4294967280U) << 7) ^ b;
2772   b  = ((z4 << 3) ^ z4) >> 12;
2773   z4 = ((z4 & 4294967168U) << 13) ^ b;
2774   int ret= (z1 ^ z2 ^ z3 ^ z4);
2775   if(ret<0)
2776     ret*=(-1);
2777   return ret;
2778 }
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
```



```
2800 // Per-CPU process scheduler.
2801 // Each CPU calls scheduler() after setting itself up.
2802 // Scheduler never returns.  It loops, doing:
2803 //  - choose a process to run
2804 //  - swtch to start running that process
2805 //  - eventually that process transfers control
2806 //      via swtch back to the scheduler.
2807 void
2808 scheduler(void)
2809 {
2810   struct proc *p;
2811
2812   for(;;){
2813     // Enable interrupts on this processor.
2814     sti();
2815
2816     // Loop over process table looking for process to run.
2817     acquire(&ptable.lock);
2818
2819     int totalNumTickets=0;
2820
2821
2822     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2823         if(p->state == RUNNABLE || p->state == RUNNING)
2824           totalNumTickets+= p->ntickets;
2825     }
2826
2827     if(totalNumTickets<=0){
2828       release(&ptable.lock);
2829       continue;
2830     }
2831
2832     //get random number
2833     int ran = rando();
2834     ran = ran % totalNumTickets;
2835
2836     int  preSumNtic=0;
2837
2838     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2839       if(p->state != RUNNABLE)
2840         continue;
2841
2842       if(!(ran>=preSumNtic && ran <= (preSumNtic+ p-> ntickets) -1 )){
2843         preSumNtic+=p-> ntickets;
2844         continue;
2845       }
2846
2847
2848
2849
```

```
2850       // Switch to chosen process.  It is the process's job
2851       // to release ptable.lock and then reacquire it
2852       // before jumping back to us.
2853       proc = p;
2854       switchuvm(p);
2855       p->state = RUNNING;
2856       swtch(&cpu->scheduler, p->context);
2857       switchkvm();
2858
2859       // Process is done running for now.
2860       // It should have changed its p->state before coming back.
2861       proc = 0;
2862
2863       if(p->ntickets>1)
2864         priority(p->ntickets -1);
2865
2866       break;
2867     }
2868     release(&ptable.lock);
2869
2870   }
2871 }
2872
2873 // Enter scheduler.  Must hold only ptable.lock
2874 // and have changed proc->state. Saves and restores
2875 // intena because intena is a property of this
2876 // kernel thread, not this CPU. It should
2877 // be proc->intena and proc->ncli, but that would
2878 // break in the few places where a lock is held but
2879 // there's no process.
2880 void
2881 sched(void)
2882 {
2883   int intena;
2884
2885   if(!holding(&ptable.lock))
2886     panic("sched ptable.lock");
2887   if(cpu->ncli != 1)
2888     panic("sched locks");
2889   if(proc->state == RUNNING)
2890     panic("sched running");
2891   if(readeflags()&FL_IF)
2892     panic("sched interruptible");
2893   intena = cpu->intena;
2894   swtch(&proc->context, cpu->scheduler);
2895   cpu->intena = intena;
2896 }
2897
2898
2899
```



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

```
2950     release(lk);
2951   }
2952
2953   // Go to sleep.
2954   proc->chan = chan;
2955   proc->state = SLEEPING;
2956
2957   if(proc->ntickets<91)
2958   priority(proc->ntickets + 10);
2959
2960   sched();
2961
2962   // Tidy up.
2963   proc->chan = 0;
2964
2965   // Reacquire original lock.
2966   if(lk != &ptable.lock){
2967     release(&ptable.lock);
2968     acquire(lk);
2969   }
2970 }
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
```



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

```
3050 // Print a process listing to console.  For debugging.
3051 // Runs when user types ^P on console.
3052 // No lock to avoid wedging a stuck machine further.
3053 void
3054 procdump(void)
3055 {
3056   static char *states[] = {
3057   [UNUSED]    "unused",
3058   [EMBRYO]    "embryo",
3059   [SLEEPING]  "sleep ",
3060   [RUNNABLE]  "runble",
3061   [RUNNING]   "run   ",
3062   [ZOMBIE]    "zombie"
3063   };
3064   int i;
3065   struct proc *p;
3066   char *state;
3067   uint pc[10];
3068
3069   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3070     if(p->state == UNUSED)
3071       continue;
3072     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
3073       state = states[p->state];
3074     else
3075       state = "???";
3076     cprintf("%d %s %s", p->pid, state, p->name);
3077     if(p->state == SLEEPING){
3078       getcallerpcs((uint*)p->context->ebp+2, pc);
3079       for(i=0; i<10 && pc[i] != 0; i++)
3080         cprintf(" %p", pc[i]);
3081     }
3082     cprintf("\n");
3083   }
3084 }
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
```



```
3100 # Context switch
3101 #
3102 #   void swtch(struct context **old, struct context *new);
3103 #
3104 # Save current register context in old
3105 # and then load register context from new.
3106
3107 .globl swtch
3108 swtch:
3109   movl 4(%esp), %eax
3110   movl 8(%esp), %edx
3111
3112   # Save old callee-save registers
3113   pushl %ebp
3114   pushl %ebx
3115   pushl %esi
3116   pushl %edi
3117
3118   # Switch stacks
3119   movl %esp, (%eax)
3120   movl %edx, %esp
3121
3122   # Load new callee-save registers
3123   popl %edi
3124   popl %esi
3125   popl %ebx
3126   popl %ebp
3127   ret
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
```

```
3150 // Physical memory allocator, intended to allocate
3151 // memory for user processes, kernel stacks, page table pages,
3152 // and pipe buffers. Allocates 4096-byte pages.
3153
3154 #include "types.h"
3155 #include "defs.h"
3156 #include "param.h"
3157 #include "memlayout.h"
3158 #include "mmu.h"
3159 #include "spinlock.h"
3160
3161 void freerange(void *vstart, void *vend);
3162 extern char end[]; // first address after kernel loaded from ELF file
3163
3164 struct run {
3165   struct run *next;
3166 };
3167
3168 struct {
3169   struct spinlock lock;
3170   int use_lock;
3171   struct run *freelist;
3172 } kmem;
3173
3174 // Initialization happens in two phases.
3175 // 1. main() calls kinit1() while still using entrypgdir to place just
3176 // the pages mapped by entrypgdir on free list.
3177 // 2. main() calls kinit2() with the rest of the physical pages
3178 // after installing a full page table that maps them on all cores.
3179 void
3180 kinit1(void *vstart, void *vend)
3181 {
3182   initlock(&kmem.lock, "kmem");
3183   kmem.use_lock = 0;
3184   freerange(vstart, vend);
3185 }
3186
3187 void
3188 kinit2(void *vstart, void *vend)
3189 {
3190   freerange(vstart, vend);
3191   kmem.use_lock = 1;
3192 }
3193
3194
3195
3196
3197
3198
3199
```



```
3200 void
3201 freerange(void *vstart, void *vend)
3202 {
3203   char *p;
3204   p = (char*)PGROUNDUP((uint)vstart);
3205   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3206     kfree(p);
3207 }
3208
3209
3210 // Free the page of physical memory pointed at by v,
3211 // which normally should have been returned by a
3212 // call to kalloc().  (The exception is when
3213 // initializing the allocator; see kinit above.)
3214 void
3215 kfree(char *v)
3216 {
3217   struct run *r;
3218
3219   if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
3220     panic("kfree");
3221
3222   // Fill with junk to catch dangling refs.
3223   memset(v, 1, PGSIZE);
3224
3225   if(kmem.use_lock)
3226     acquire(&kmem.lock);
3227   r = (struct run*)v;
3228   r->next = kmem.freelist;
3229   kmem.freelist = r;
3230   if(kmem.use_lock)
3231     release(&kmem.lock);
3232 }
3233
3234 // Allocate one 4096-byte page of physical memory.
3235 // Returns a pointer that the kernel can use.
3236 // Returns 0 if the memory cannot be allocated.
3237 char*
3238 kalloc(void)
3239 {
3240   struct run *r;
3241
3242   if(kmem.use_lock)
3243     acquire(&kmem.lock);
3244   r = kmem.freelist;
3245   if(r)
3246     kmem.freelist = r->next;
3247   if(kmem.use_lock)
3248     release(&kmem.lock);
3249   return (char*)r;
```

```
3250 }
3251
3252
3253
3254
3255
3256
3257
3258
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
```



```
3300 // x86 trap and interrupt constants.
3301
3302 // Processor-defined:
3303 #define T_DIVIDE         0      // divide error
3304 #define T_DEBUG          1      // debug exception
3305 #define T_NMI            2      // non-maskable interrupt
3306 #define T_BRKPT          3      // breakpoint
3307 #define T_OFLOW          4      // overflow
3308 #define T_BOUND          5      // bounds check
3309 #define T_ILLOP          6      // illegal opcode
3310 #define T_DEVICE         7      // device not available
3311 #define T_DBLFLT         8      // double fault
3312 // #define T_COPROC      9      // reserved (not used since 486)
3313 #define T_TSS           10      // invalid task switch segment
3314 #define T_SEGNP         11      // segment not present
3315 #define T_STACK         12      // stack exception
3316 #define T_GPFLT         13      // general protection fault
3317 #define T_PGFLT         14      // page fault
3318 // #define T_RES        15      // reserved
3319 #define T_FPERR         16      // floating point error
3320 #define T_ALIGN         17      // aligment check
3321 #define T_MCHK          18      // machine check
3322 #define T_SIMDERR       19      // SIMD floating point error
3323
3324 // These are arbitrarily chosen, but with care not to overlap
3325 // processor defined exceptions or interrupt vectors.
3326 #define T_SYSCALL       64      // system call
3327 #define T_DEFAULT      500      // catchall
3328
3329 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
3330
3331 #define IRQ_TIMER        0
3332 #define IRQ_KBD          1
3333 #define IRQ_COM1         4
3334 #define IRQ_IDE         14
3335 #define IRQ_ERROR       19
3336 #define IRQ_SPURIOUS    31
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
```

```
#!/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
```

```
3450 #include "types.h"
3451 #include "defs.h"
3452 #include "param.h"
3453 #include "memlayout.h"
3454 #include "mmu.h"
3455 #include "proc.h"
3456 #include "x86.h"
3457 #include "traps.h"
3458 #include "spinlock.h"
3459
3460 // Interrupt descriptor table (shared by all CPUs).
3461 struct gatedesc idt[256];
3462 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
3463 struct spinlock tickslock;
3464 uint ticks;
3465
3466 void
3467 tvinit(void)
3468 {
3469   int i;
3470
3471   for(i = 0; i < 256; i++)
3472     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3473   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3474
3475   initlock(&tickslock, "time");
3476 }
3477
3478 void
3479 idtinit(void)
3480 {
3481   lidt(idt, sizeof(idt));
3482 }
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
```



```
3500 void
3501 trap(struct trapframe *tf)
3502 {
3503   if(tf->trapno == T_SYSCALL){
3504     if(proc->killed)
3505       exit(0);
3506     proc->tf = tf;
3507     syscall();
3508     if(proc->killed)
3509       exit(0);
3510     return;
3511   }
3512
3513   switch(tf->trapno){
3514   case T_IRQ0 + IRQ_TIMER:
3515     if(cpunum() == 0){
3516       acquire(&tickslock);
3517       ticks++;
3518       wakeup(&ticks);
3519       release(&tickslock);
3520     }
3521     lapiceoi();
3522     break;
3523   case T_IRQ0 + IRQ_IDE:
3524     ideintr();
3525     lapiceoi();
3526     break;
3527   case T_IRQ0 + IRQ_IDE+1:
3528     // Bochs generates spurious IDE1 interrupts.
3529     break;
3530   case T_IRQ0 + IRQ_KBD:
3531     kbdintr();
3532     lapiceoi();
3533     break;
3534   case T_IRQ0 + IRQ_COM1:
3535     uartintr();
3536     lapiceoi();
3537     break;
3538   case T_IRQ0 + 7:
3539   case T_IRQ0 + IRQ_SPURIOUS:
3540     cprintf("cpu%d: spurious interrupt at %x:%x\n",
3541             cpunum(), tf->cs, tf->eip);
3542     lapiceoi();
3543     break;
3544
3545
3546
3547
3548
3549
```

```
3550   default:
3551     if(proc == 0 || (tf->cs&3) == 0){
3552       // In kernel, it must be our mistake.
3553       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3554               tf->trapno, cpunum(), tf->eip, rcr2());
3555       panic("trap");
3556     }
3557     // In user space, assume process misbehaved.
3558     cprintf("pid %d %s: trap %d err %d on cpu %d "
3559             "eip 0x%x addr 0x%x--kill proc\n",
3560             proc->pid, proc->name, tf->trapno, tf->err, cpunum(), tf->eip,
3561             rcr2());
3562     proc->killed = 1;
3563   }
3564
3565   // Force process exit if it has been killed and is in user space.
3566   // (If it is still executing in the kernel, let it keep running
3567   // until it gets to the regular system call return.)
3568   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3569     exit(0);
3570
3571   // Force process to give up CPU on clock tick.
3572   // If interrupts were on while locks held, would need to check nlock.
3573   if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3574     yield();
3575
3576   // Check if the process has been killed since we yielded
3577   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3578     exit(0);
3579 }
3580
3581
3582
3583
3584
3585
3586
3587
3588
3589
3590
3591
3592
3593
3594
3595
3596
3597
3598
3599
```



```
3600 // System call numbers
3601 #define SYS_fork    1
3602 #define SYS_exit    2
3603 #define SYS_wait    3
3604 #define SYS_pipe    4
3605 #define SYS_read    5
3606 #define SYS_kill    6
3607 #define SYS_exec    7
3608 #define SYS_fstat   8
3609 #define SYS_chdir   9
3610 #define SYS_dup    10
3611 #define SYS_getpid 11
3612 #define SYS_sbrk   12
3613 #define SYS_sleep  13
3614 #define SYS_uptime 14
3615 #define SYS_open   15
3616 #define SYS_write  16
3617 #define SYS_mknod  17
3618 #define SYS_unlink 18
3619 #define SYS_link   19
3620 #define SYS_mkdir  20
3621 #define SYS_close  21
3622 #define SYS_priority  22
3623 #define SYS_policy  23
3624
3625
3626
3627
3628
3629
3630
3631
3632
3633
3634
3635
3636
3637
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649
```

```
3650 #include "types.h"
3651 #include "defs.h"
3652 #include "param.h"
3653 #include "memlayout.h"
3654 #include "mmu.h"
3655 #include "proc.h"
3656 #include "x86.h"
3657 #include "syscall.h"
3658
3659 // User code makes a system call with INT T_SYSCALL.
3660 // System call number in %eax.
3661 // Arguments on the stack, from the user call to the C
3662 // library system call function. The saved user %esp points
3663 // to a saved program counter, and then the first argument.
3664
3665 // Fetch the int at addr from the current process.
3666 int
3667 fetchint(uint addr, int *ip)
3668 {
3669   if(addr >= proc->sz || addr+4 > proc->sz)
3670     return -1;
3671   *ip = *(int*)(addr);
3672   return 0;
3673 }
3674
3675 // Fetch the nul-terminated string at addr from the current process.
3676 // Doesn't actually copy the string - just sets *pp to point at it.
3677 // Returns length of string, not including nul.
3678 int
3679 fetchstr(uint addr, char **pp)
3680 {
3681   char *s, *ep;
3682
3683   if(addr >= proc->sz)
3684     return -1;
3685   *pp = (char*)addr;
3686   ep = (char*)proc->sz;
3687   for(s = *pp; s < ep; s++)
3688     if(*s == 0)
3689       return s - *pp;
3690   return -1;
3691 }
3692
3693 // Fetch the nth 32-bit system call argument.
3694 int
3695 argint(int n, int *ip)
3696 {
3697   return fetchint(proc->tf->esp + 4 + 4*n, ip);
3698 }
3699
```



```
3700 // Fetch the nth word-sized system call argument as a pointer
3701 // to a block of memory of size bytes.  Check that the pointer
3702 // lies within the process address space.
3703 int
3704 argptr(int n, char **pp, int size)
3705 {
3706   int i;
3707
3708   if(argint(n, &i) < 0)
3709     return -1;
3710   if(size < 0 || (uint)i >= proc->sz || (uint)i+size > proc->sz)
3711     return -1;
3712   *pp = (char*)i;
3713   return 0;
3714 }
3715
3716 // Fetch the nth word-sized system call argument as a string pointer.
3717 // Check that the pointer is valid and the string is nul-terminated.
3718 // (There is no shared writable memory, so the string can't change
3719 // between this check and being used by the kernel.)
3720 int
3721 argstr(int n, char **pp)
3722 {
3723   int addr;
3724   if(argint(n, &addr) < 0)
3725     return -1;
3726   return fetchstr(addr, pp);
3727 }
3728
3729 extern int sys_chdir(void);
3730 extern int sys_close(void);
3731 extern int sys_dup(void);
3732 extern int sys_exec(void);
3733 extern int sys_exit(void);
3734 extern int sys_fork(void);
3735 extern int sys_fstat(void);
3736 extern int sys_getpid(void);
3737 extern int sys_kill(void);
3738 extern int sys_link(void);
3739 extern int sys_mkdir(void);
3740 extern int sys_mknod(void);
3741 extern int sys_open(void);
3742 extern int sys_pipe(void);
3743 extern int sys_read(void);
3744 extern int sys_sbrk(void);
3745 extern int sys_sleep(void);
3746 extern int sys_unlink(void);
3747 extern int sys_wait(void);
3748 extern int sys_write(void);
3749 extern int sys_uptime(void);
```

```
3750 extern int sys_priority(void);
3751 extern int sys_policy(void);
3752
3753 static int (*syscalls[])(void) = {
3754 [SYS_fork]    sys_fork,
3755 [SYS_exit]    sys_exit,
3756 [SYS_wait]    sys_wait,
3757 [SYS_pipe]    sys_pipe,
3758 [SYS_read]    sys_read,
3759 [SYS_kill]    sys_kill,
3760 [SYS_exec]    sys_exec,
3761 [SYS_fstat]   sys_fstat,
3762 [SYS_chdir]   sys_chdir,
3763 [SYS_dup]     sys_dup,
3764 [SYS_getpid]  sys_getpid,
3765 [SYS_sbrk]    sys_sbrk,
3766 [SYS_sleep]   sys_sleep,
3767 [SYS_uptime]  sys_uptime,
3768 [SYS_open]    sys_open,
3769 [SYS_write]   sys_write,
3770 [SYS_mknod]   sys_mknod,
3771 [SYS_unlink]  sys_unlink,
3772 [SYS_link]    sys_link,
3773 [SYS_mkdir]   sys_mkdir,
3774 [SYS_close]   sys_close,
3775 [SYS_priority] sys_priority,
3776 [SYS_priority] sys_policy
3777 };
3778
3779 void
3780 syscall(void)
3781 {
3782   int num;
3783
3784   num = proc->tf->eax;
3785   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3786     proc->tf->eax = syscalls[num]();
3787   } else {
3788     cprintf("%d %s: unknown sys call %d\n",
3789             proc->pid, proc->name, num);
3790     proc->tf->eax = -1;
3791   }
3792 }
3793
3794
3795
3796
3797
3798
3799
```



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

```
3850 int
3851 sys_wait(void)
3852 {
3853   int* status;
3854   int retVal;
3855   if(argptr(0,(char**)&status,sizeof(status))>=0)
3856     retVal=wait(status);
3857   else
3858     retVal=-1;
3859
3860   return retVal;
3861 }
3862
3863 int
3864 sys_kill(void)
3865 {
3866   int pid;
3867
3868   if(argint(0, &pid) < 0)
3869     return -1;
3870   return kill(pid);
3871 }
3872
3873 int
3874 sys_getpid(void)
3875 {
3876   return proc->pid;
3877 }
3878
3879 int
3880 sys_sbrk(void)
3881 {
3882   int addr;
3883   int n;
3884
3885   if(argint(0, &n) < 0)
3886     return -1;
3887   addr = proc->sz;
3888   if(growproc(n) < 0)
3889     return -1;
3890   return addr;
3891 }
3892
3893
3894
3895
3896
3897
3898
3899
```



```
3900 int
3901 sys_sleep(void)
3902 {
3903   int n;
3904   uint ticks0;
3905
3906   if(argint(0, &n) < 0)
3907     return -1;
3908   acquire(&tickslock);
3909   ticks0 = ticks;
3910   while(ticks - ticks0 < n){
3911     if(proc->killed){
3912       release(&tickslock);
3913       return -1;
3914     }
3915     sleep(&ticks, &tickslock);
3916   }
3917   release(&tickslock);
3918   return 0;
3919 }
3920
3921 // return how many clock tick interrupts have occurred
3922 // since start.
3923 int
3924 sys_uptime(void)
3925 {
3926   uint xticks;
3927
3928   acquire(&tickslock);
3929   xticks = ticks;
3930   release(&tickslock);
3931   return xticks;
3932 }
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
```

```
3950 struct buf {
3951   int flags;
3952   uint dev;
3953   uint blockno;
3954   struct sleeplock lock;
3955   uint refcnt;
3956   struct buf *prev; // LRU cache list
3957   struct buf *next;
3958   struct buf *qnext; // disk queue
3959   uchar data[BSIZE];
3960 };
3961 #define B_VALID 0x2  // buffer has been read from disk
3962 #define B_DIRTY 0x4  // buffer needs to be written to disk
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
```



```
4000 // Long-term locks for processes
4001 struct sleeplock {
4002   uint locked;       // Is the lock held?
4003   struct spinlock lk; // spinlock protecting this sleep lock
4004
4005   // For debugging:
4006   char *name;        // Name of lock.
4007   int pid;           // Process holding lock
4008 };
4009
4010
4011
4012
4013
4014
4015
4016
4017
4018
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
```

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



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

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



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

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



```
4300 // Blank page.
4301
4302
4303
4304
4305
4306
4307
4308
4309
4310
4311
4312
4313
4314
4315
4316
4317
4318
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4330
4331
4332
4333
4334
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
```

```
4350 // Simple PIO-based (non-DMA) IDE driver code.
4351
4352 #include "types.h"
4353 #include "defs.h"
4354 #include "param.h"
4355 #include "memlayout.h"
4356 #include "mmu.h"
4357 #include "proc.h"
4358 #include "x86.h"
4359 #include "traps.h"
4360 #include "spinlock.h"
4361 #include "sleeplock.h"
4362 #include "fs.h"
4363 #include "buf.h"
4364
4365 #define SECTOR_SIZE   512
4366 #define IDE_BSY       0x80
4367 #define IDE_DRDY      0x40
4368 #define IDE_DF        0x20
4369 #define IDE_ERR       0x01
4370
4371 #define IDE_CMD_READ  0x20
4372 #define IDE_CMD_WRITE 0x30
4373 #define IDE_CMD_RDMUL 0xc4
4374 #define IDE_CMD_WRMUL 0xc5
4375
4376 // idequeue points to the buf now being read/written to the disk.
4377 // idequeue->qnext points to the next buf to be processed.
4378 // You must hold idelock while manipulating queue.
4379
4380 static struct spinlock idelock;
4381 static struct buf *idequeue;
4382
4383 static int havedisk1;
4384 static void idestart(struct buf*);
4385
4386 // Wait for IDE disk to become ready.
4387 static int
4388 idewait(int checkerr)
4389 {
4390   int r;
4391
4392   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4393     ;
4394   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4395     return -1;
4396   return 0;
4397 }
4398
4399
```



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

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



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

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



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

```
4650 // Return a locked buf with the contents of the indicated block.
4651 struct buf*
4652 bread(uint dev, uint blockno)
4653 {
4654   struct buf *b;
4655
4656   b = bget(dev, blockno);
4657   if(!(b->flags & B_VALID)) {
4658     iderw(b);
4659   }
4660   return b;
4661 }
4662
4663 // Write b's contents to disk.  Must be locked.
4664 void
4665 bwrite(struct buf *b)
4666 {
4667   if(!holdingsleep(&b->lock))
4668     panic("bwrite");
4669   b->flags |= B_DIRTY;
4670   iderw(b);
4671 }
4672
4673 // Release a locked buffer.
4674 // Move to the head of the MRU list.
4675 void
4676 brelse(struct buf *b)
4677 {
4678   if(!holdingsleep(&b->lock))
4679     panic("brelse");
4680
4681   releasesleep(&b->lock);
4682
4683   acquire(&bcache.lock);
4684   b->refcnt--;
4685   if (b->refcnt == 0) {
4686     // no one is waiting for it.
4687     b->next->prev = b->prev;
4688     b->prev->next = b->next;
4689     b->next = bcache.head.next;
4690     b->prev = &bcache.head;
4691     bcache.head.next->prev = b;
4692     bcache.head.next = b;
4693   }
4694
4695   release(&bcache.lock);
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 // Sleeping locks
4751
4752 #include "types.h"
4753 #include "defs.h"
4754 #include "param.h"
4755 #include "x86.h"
4756 #include "memlayout.h"
4757 #include "mmu.h"
4758 #include "proc.h"
4759 #include "spinlock.h"
4760 #include "sleeplock.h"
4761
4762 void
4763 initsleeplock(struct sleeplock *lk, char *name)
4764 {
4765   initlock(&lk->lk, "sleep lock");
4766   lk->name = name;
4767   lk->locked = 0;
4768   lk->pid = 0;
4769 }
4770
4771 void
4772 acquiresleep(struct sleeplock *lk)
4773 {
4774   acquire(&lk->lk);
4775   while (lk->locked) {
4776     sleep(lk, &lk->lk);
4777   }
4778   lk->locked = 1;
4779   lk->pid = proc->pid;
4780   release(&lk->lk);
4781 }
4782
4783 void
4784 releasesleep(struct sleeplock *lk)
4785 {
4786   acquire(&lk->lk);
4787   lk->locked = 0;
4788   lk->pid = 0;
4789   wakeup(lk);
4790   release(&lk->lk);
4791 }
4792
4793
4794
4795
4796
4797
4798
4799
```



```
4800 int
4801 holdingsleep(struct sleeplock *lk)
4802 {
4803   int r;
4804
4805   acquire(&lk->lk);
4806   r = lk->locked;
4807   release(&lk->lk);
4808   return r;
4809 }
4810
4811
4812
4813
4814
4815
4816
4817
4818
4819
4820
4821
4822
4823
4824
4825
4826
4827
4828
4829
4830
4831
4832
4833
4834
4835
4836
4837
4838
4839
4840
4841
4842
4843
4844
4845
4846
4847
4848
4849
```

```
4850 #include "types.h"
4851 #include "defs.h"
4852 #include "param.h"
4853 #include "spinlock.h"
4854 #include "sleeplock.h"
4855 #include "fs.h"
4856 #include "buf.h"
4857
4858 // Simple logging that allows concurrent FS system calls.
4859 //
4860 // A log transaction contains the updates of multiple FS system
4861 // calls. The logging system only commits when there are
4862 // no FS system calls active. Thus there is never
4863 // any reasoning required about whether a commit might
4864 // write an uncommitted system call's updates to disk.
4865 //
4866 // A system call should call begin_op()/end_op() to mark
4867 // its start and end. Usually begin_op() just increments
4868 // the count of in-progress FS system calls and returns.
4869 // But if it thinks the log is close to running out, it
4870 // sleeps until the last outstanding end_op() commits.
4871 //
4872 // The log is a physical re-do log containing disk blocks.
4873 // The on-disk log format:
4874 //   header block, containing block #s for block A, B, C, ...
4875 //   block A
4876 //   block B
4877 //   block C
4878 //   ...
4879 // Log appends are synchronous.
4880
4881 // Contents of the header block, used for both the on-disk header block
4882 // and to keep track in memory of logged block# before commit.
4883 struct logheader {
4884   int n;
4885   int block[LOGSIZE];
4886 };
4887
4888 struct log {
4889   struct spinlock lock;
4890   int start;
4891   int size;
4892   int outstanding; // how many FS sys calls are executing.
4893   int committing;  // in commit(), please wait.
4894   int dev;
4895   struct logheader lh;
4896 };
4897
4898
4899
```



```
4900 struct log log;
4901
4902 static void recover_from_log(void);
4903 static void commit();
4904
4905 void
4906 initlog(int dev)
4907 {
4908   if (sizeof(struct logheader) >= BSIZE)
4909     panic("initlog: too big logheader");
4910
4911   struct superblock sb;
4912   initlock(&log.lock, "log");
4913   readsb(dev, &sb);
4914   log.start = sb.logstart;
4915   log.size = sb.nlog;
4916   log.dev = dev;
4917   recover_from_log();
4918 }
4919
4920 // Copy committed blocks from log to their home location
4921 static void
4922 install_trans(void)
4923 {
4924   int tail;
4925
4926   for (tail = 0; tail < log.lh.n; tail++) {
4927     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
4928     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
4929     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
4930     bwrite(dbuf);  // write dst to disk
4931     brelse(lbuf);
4932     brelse(dbuf);
4933   }
4934 }
4935
4936 // Read the log header from disk into the in-memory log header
4937 static void
4938 read_head(void)
4939 {
4940   struct buf *buf = bread(log.dev, log.start);
4941   struct logheader *lh = (struct logheader *) (buf->data);
4942   int i;
4943   log.lh.n = lh->n;
4944   for (i = 0; i < log.lh.n; i++) {
4945     log.lh.block[i] = lh->block[i];
4946   }
4947   brelse(buf);
4948 }
4949
```

```
4950 // Write in-memory log header to disk.
4951 // This is the true point at which the
4952 // current transaction commits.
4953 static void
4954 write_head(void)
4955 {
4956   struct buf *buf = bread(log.dev, log.start);
4957   struct logheader *hb = (struct logheader *) (buf->data);
4958   int i;
4959   hb->n = log.lh.n;
4960   for (i = 0; i < log.lh.n; i++) {
4961     hb->block[i] = log.lh.block[i];
4962   }
4963   bwrite(buf);
4964   brelse(buf);
4965 }
4966
4967 static void
4968 recover_from_log(void)
4969 {
4970   read_head();
4971   install_trans(); // if committed, copy from log to disk
4972   log.lh.n = 0;
4973   write_head(); // clear the log
4974 }
4975
4976 // called at the start of each FS system call.
4977 void
4978 begin_op(void)
4979 {
4980   acquire(&log.lock);
4981   while(1){
4982     if(log.committing){
4983       sleep(&log, &log.lock);
4984     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
4985       // this op might exhaust log space; wait for commit.
4986       sleep(&log, &log.lock);
4987     } else {
4988       log.outstanding += 1;
4989       release(&log.lock);
4990       break;
4991     }
4992   }
4993 }
4994
4995
4996
4997
4998
4999
```



```
5000 // called at the end of each FS system call.
5001 // commits if this was the last outstanding operation.
5002 void
5003 end_op(void)
5004 {
5005   int do_commit = 0;
5006
5007   acquire(&log.lock);
5008   log.outstanding -= 1;
5009   if(log.committing)
5010     panic("log.committing");
5011   if(log.outstanding == 0){
5012     do_commit = 1;
5013     log.committing = 1;
5014   } else {
5015     // begin_op() may be waiting for log space.
5016     wakeup(&log);
5017   }
5018   release(&log.lock);
5019
5020   if(do_commit){
5021     // call commit w/o holding locks, since not allowed
5022     // to sleep with locks.
5023     commit();
5024     acquire(&log.lock);
5025     log.committing = 0;
5026     wakeup(&log);
5027     release(&log.lock);
5028   }
5029 }
5030
5031 // Copy modified blocks from cache to log.
5032 static void
5033 write_log(void)
5034 {
5035   int tail;
5036
5037   for (tail = 0; tail < log.lh.n; tail++) {
5038     struct buf *to = bread(log.dev, log.start+tail+1); // log block
5039     struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
5040     memmove(to->data, from->data, BSIZE);
5041     bwrite(to);  // write the log
5042     brelse(from);
5043     brelse(to);
5044   }
5045 }
5046
5047
5048
5049
```

```
5050 static void
5051 commit()
5052 {
5053   if (log.lh.n > 0) {
5054     write_log();     // Write modified blocks from cache to log
5055     write_head();    // Write header to disk -- the real commit
5056     install_trans(); // Now install writes to home locations
5057     log.lh.n = 0;
5058     write_head();    // Erase the transaction from the log
5059   }
5060 }
5061
5062 // Caller has modified b->data and is done with the buffer.
5063 // Record the block number and pin in the cache with B_DIRTY.
5064 // commit()/write_log() will do the disk write.
5065 //
5066 // log_write() replaces bwrite(); a typical use is:
5067 //   bp = bread(...)
5068 //   modify bp->data[]
5069 //   log_write(bp)
5070 //   brelse(bp)
5071 void
5072 log_write(struct buf *b)
5073 {
5074   int i;
5075
5076   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
5077     panic("too big a transaction");
5078   if (log.outstanding < 1)
5079     panic("log_write outside of trans");
5080
5081   acquire(&log.lock);
5082   for (i = 0; i < log.lh.n; i++) {
5083     if (log.lh.block[i] == b->blockno)   // log absorbtion
5084       break;
5085   }
5086   log.lh.block[i] = b->blockno;
5087   if (i == log.lh.n)
5088     log.lh.n++;
5089   b->flags |= B_DIRTY; // prevent eviction
5090   release(&log.lock);
5091 }
5092
5093
5094
5095
5096
5097
5098
5099
```



```
5100 // File system implementation.  Five layers:
5101 //   + Blocks: allocator for raw disk blocks.
5102 //   + Log: crash recovery for multi-step updates.
5103 //   + Files: inode allocator, reading, writing, metadata.
5104 //   + Directories: inode with special contents (list of other inodes!)
5105 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
5106 //
5107 // This file contains the low-level file system manipulation
5108 // routines.  The (higher-level) system call implementations
5109 // are in sysfile.c.
5110
5111 #include "types.h"
5112 #include "defs.h"
5113 #include "param.h"
5114 #include "stat.h"
5115 #include "mmu.h"
5116 #include "proc.h"
5117 #include "spinlock.h"
5118 #include "sleeplock.h"
5119 #include "fs.h"
5120 #include "buf.h"
5121 #include "file.h"
5122
5123 #define min(a, b) ((a) < (b) ? (a) : (b))
5124 static void itrunc(struct inode*);
5125 // there should be one superblock per disk device, but we run with
5126 // only one device
5127 struct superblock sb;
5128
5129 // Read the super block.
5130 void
5131 readsb(int dev, struct superblock *sb)
5132 {
5133   struct buf *bp;
5134
5135   bp = bread(dev, 1);
5136   memmove(sb, bp->data, sizeof(*sb));
5137   brelse(bp);
5138 }
5139
5140
5141
5142
5143
5144
5145
5146
5147
5148
5149
```

```
5150 // Zero a block.
5151 static void
5152 bzero(int dev, int bno)
5153 {
5154   struct buf *bp;
5155
5156   bp = bread(dev, bno);
5157   memset(bp->data, 0, BSIZE);
5158   log_write(bp);
5159   brelse(bp);
5160 }
5161
5162 // Blocks.
5163
5164 // Allocate a zeroed disk block.
5165 static uint
5166 balloc(uint dev)
5167 {
5168   int b, bi, m;
5169   struct buf *bp;
5170
5171   bp = 0;
5172   for(b = 0; b < sb.size; b += BPB){
5173     bp = bread(dev, BBLOCK(b, sb));
5174     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
5175       m = 1 << (bi % 8);
5176       if((bp->data[bi/8] & m) == 0){  // Is block free?
5177         bp->data[bi/8] |= m;  // Mark block in use.
5178         log_write(bp);
5179         brelse(bp);
5180         bzero(dev, b + bi);
5181         return b + bi;
5182       }
5183     }
5184     brelse(bp);
5185   }
5186   panic("balloc: out of blocks");
5187 }
5188
5189
5190
5191
5192
5193
5194
5195
5196
5197
5198
5199
```



```
5200 // Free a disk block.
5201 static void
5202 bfree(int dev, uint b)
5203 {
5204   struct buf *bp;
5205   int bi, m;
5206
5207   readsb(dev, &sb);
5208   bp = bread(dev, BBLOCK(b, sb));
5209   bi = b % BPB;
5210   m = 1 << (bi % 8);
5211   if((bp->data[bi/8] & m) == 0)
5212     panic("freeing free block");
5213   bp->data[bi/8] &= ~m;
5214   log_write(bp);
5215   brelse(bp);
5216 }
5217
5218 // Inodes.
5219 //
5220 // An inode describes a single unnamed file.
5221 // The inode disk structure holds metadata: the file's type,
5222 // its size, the number of links referring to it, and the
5223 // list of blocks holding the file's content.
5224 //
5225 // The inodes are laid out sequentially on disk at
5226 // sb.startinode. Each inode has a number, indicating its
5227 // position on the disk.
5228 //
5229 // The kernel keeps a cache of in-use inodes in memory
5230 // to provide a place for synchronizing access
5231 // to inodes used by multiple processes. The cached
5232 // inodes include book-keeping information that is
5233 // not stored on disk: ip->ref and ip->flags.
5234 //
5235 // An inode and its in-memory represtative go through a
5236 // sequence of states before they can be used by the
5237 // rest of the file system code.
5238 //
5239 // * Allocation: an inode is allocated if its type (on disk)
5240 //   is non-zero. ialloc() allocates, iput() frees if
5241 //   the link count has fallen to zero.
5242 //
5243 // * Referencing in cache: an entry in the inode cache
5244 //   is free if ip->ref is zero. Otherwise ip->ref tracks
5245 //   the number of in-memory pointers to the entry (open
5246 //   files and current directories). iget() to find or
5247 //   create a cache entry and increment its ref, iput()
5248 //   to decrement ref.
5249 //
```

```
5250 // * Valid: the information (type, size, &c) in an inode
5251 //   cache entry is only correct when the I_VALID bit
5252 //   is set in ip->flags. ilock() reads the inode from
5253 //   the disk and sets I_VALID, while iput() clears
5254 //   I_VALID if ip->ref has fallen to zero.
5255 //
5256 // * Locked: file system code may only examine and modify
5257 //   the information in an inode and its content if it
5258 //   has first locked the inode.
5259 //
5260 // Thus a typical sequence is:
5261 //   ip = iget(dev, inum)
5262 //   ilock(ip)
5263 //   ... examine and modify ip->xxx ...
5264 //   iunlock(ip)
5265 //   iput(ip)
5266 //
5267 // ilock() is separate from iget() so that system calls can
5268 // get a long-term reference to an inode (as for an open file)
5269 // and only lock it for short periods (e.g., in read()).
5270 // The separation also helps avoid deadlock and races during
5271 // pathname lookup. iget() increments ip->ref so that the inode
5272 // stays cached and pointers to it remain valid.
5273 //
5274 // Many internal file system functions expect the caller to
5275 // have locked the inodes involved; this lets callers create
5276 // multi-step atomic operations.
5277
5278 struct {
5279   struct spinlock lock;
5280   struct inode inode[NINODE];
5281 } icache;
5282
5283 void
5284 iinit(int dev)
5285 {
5286   int i = 0;
5287
5288   initlock(&icache.lock, "icache");
5289   for(i = 0; i < NINODE; i++) {
5290     initsleeplock(&icache.inode[i].lock, "inode");
5291   }
5292
5293   readsb(dev, &sb);
5294   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d\
5295  inodestart %d bmap start %d\n", sb.size, sb.nblocks,
5296           sb.ninodes, sb.nlog, sb.logstart, sb.inodestart,
5297           sb.bmapstart);
5298 }
5299
```



```
5300 static struct inode* iget(uint dev, uint inum);
5301
5302
5303
5304
5305
5306
5307
5308
5309
5310
5311
5312
5313
5314
5315
5316
5317
5318
5319
5320
5321
5322
5323
5324
5325
5326
5327
5328
5329
5330
5331
5332
5333
5334
5335
5336
5337
5338
5339
5340
5341
5342
5343
5344
5345
5346
5347
5348
5349
```

```
5350 // Allocate a new inode with the given type on device dev.
5351 // A free inode has a type of zero.
5352 struct inode*
5353 ialloc(uint dev, short type)
5354 {
5355   int inum;
5356   struct buf *bp;
5357   struct dinode *dip;
5358
5359   for(inum = 1; inum < sb.ninodes; inum++){
5360     bp = bread(dev, IBLOCK(inum, sb));
5361     dip = (struct dinode*)bp->data + inum%IPB;
5362     if(dip->type == 0){  // a free inode
5363       memset(dip, 0, sizeof(*dip));
5364       dip->type = type;
5365       log_write(bp);   // mark it allocated on the disk
5366       brelse(bp);
5367       return iget(dev, inum);
5368     }
5369     brelse(bp);
5370   }
5371   panic("ialloc: no inodes");
5372 }
5373
5374 // Copy a modified in-memory inode to disk.
5375 void
5376 iupdate(struct inode *ip)
5377 {
5378   struct buf *bp;
5379   struct dinode *dip;
5380
5381   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5382   dip = (struct dinode*)bp->data + ip->inum%IPB;
5383   dip->type = ip->type;
5384   dip->major = ip->major;
5385   dip->minor = ip->minor;
5386   dip->nlink = ip->nlink;
5387   dip->size = ip->size;
5388   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5389   log_write(bp);
5390   brelse(bp);
5391 }
5392
5393
5394
5395
5396
5397
5398
5399
```



```
5400 // Find the inode with number inum on device dev
5401 // and return the in-memory copy. Does not lock
5402 // the inode and does not read it from disk.
5403 static struct inode*
5404 iget(uint dev, uint inum)
5405 {
5406   struct inode *ip, *empty;
5407
5408   acquire(&icache.lock);
5409
5410   // Is the inode already cached?
5411   empty = 0;
5412   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5413     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5414       ip->ref++;
5415       release(&icache.lock);
5416       return ip;
5417     }
5418     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
5419       empty = ip;
5420   }
5421
5422   // Recycle an inode cache entry.
5423   if(empty == 0)
5424     panic("iget: no inodes");
5425
5426   ip = empty;
5427   ip->dev = dev;
5428   ip->inum = inum;
5429   ip->ref = 1;
5430   ip->flags = 0;
5431   release(&icache.lock);
5432
5433   return ip;
5434 }
5435
5436 // Increment reference count for ip.
5437 // Returns ip to enable ip = idup(ip1) idiom.
5438 struct inode*
5439 idup(struct inode *ip)
5440 {
5441   acquire(&icache.lock);
5442   ip->ref++;
5443   release(&icache.lock);
5444   return ip;
5445 }
5446
5447
5448
5449
```

```
5450 // Lock the given inode.
5451 // Reads the inode from disk if necessary.
5452 void
5453 ilock(struct inode *ip)
5454 {
5455   struct buf *bp;
5456   struct dinode *dip;
5457
5458   if(ip == 0 || ip->ref < 1)
5459     panic("ilock");
5460
5461   acquiresleep(&ip->lock);
5462
5463   if(!(ip->flags & I_VALID)){
5464     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5465     dip = (struct dinode*)bp->data + ip->inum%IPB;
5466     ip->type = dip->type;
5467     ip->major = dip->major;
5468     ip->minor = dip->minor;
5469     ip->nlink = dip->nlink;
5470     ip->size = dip->size;
5471     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5472     brelse(bp);
5473     ip->flags |= I_VALID;
5474     if(ip->type == 0)
5475       panic("ilock: no type");
5476   }
5477 }
5478
5479 // Unlock the given inode.
5480 void
5481 iunlock(struct inode *ip)
5482 {
5483   if(ip == 0 || !holdingsleep(&ip->lock) || ip->ref < 1)
5484     panic("iunlock");
5485
5486   releasesleep(&ip->lock);
5487 }
5488
5489
5490
5491
5492
5493
5494
5495
5496
5497
5498
5499
```



```
5500 // Drop a reference to an in-memory inode.
5501 // If that was the last reference, the inode cache entry can
5502 // be recycled.
5503 // If that was the last reference and the inode has no links
5504 // to it, free the inode (and its content) on disk.
5505 // All calls to iput() must be inside a transaction in
5506 // case it has to free the inode.
5507 void
5508 iput(struct inode *ip)
5509 {
5510   acquire(&icache.lock);
5511   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
5512     // inode has no links and no other references: truncate and free.
5513     release(&icache.lock);
5514     itrunc(ip);
5515     ip->type = 0;
5516     iupdate(ip);
5517     acquire(&icache.lock);
5518     ip->flags = 0;
5519   }
5520   ip->ref--;
5521   release(&icache.lock);
5522 }
5523
5524 // Common idiom: unlock, then put.
5525 void
5526 iunlockput(struct inode *ip)
5527 {
5528   iunlock(ip);
5529   iput(ip);
5530 }
5531
5532
5533
5534
5535
5536
5537
5538
5539
5540
5541
5542
5543
5544
5545
5546
5547
5548
5549
```

```
5550 // Inode content
5551 //
5552 // The content (data) associated with each inode is stored
5553 // in blocks on the disk. The first NDIRECT block numbers
5554 // are listed in ip->addrs[].  The next NINDIRECT blocks are
5555 // listed in block ip->addrs[NDIRECT].
5556
5557 // Return the disk block address of the nth block in inode ip.
5558 // If there is no such block, bmap allocates one.
5559 static uint
5560 bmap(struct inode *ip, uint bn)
5561 {
5562   uint addr, *a;
5563   struct buf *bp;
5564
5565   if(bn < NDIRECT){
5566     if((addr = ip->addrs[bn]) == 0)
5567       ip->addrs[bn] = addr = balloc(ip->dev);
5568     return addr;
5569   }
5570   bn -= NDIRECT;
5571
5572   if(bn < NINDIRECT){
5573     // Load indirect block, allocating if necessary.
5574     if((addr = ip->addrs[NDIRECT]) == 0)
5575       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5576     bp = bread(ip->dev, addr);
5577     a = (uint*)bp->data;
5578     if((addr = a[bn]) == 0){
5579       a[bn] = addr = balloc(ip->dev);
5580       log_write(bp);
5581     }
5582     brelse(bp);
5583     return addr;
5584   }
5585
5586   panic("bmap: out of range");
5587 }
5588
5589
5590
5591
5592
5593
5594
5595
5596
5597
5598
5599
```



```
5600 // Truncate inode (discard contents).
5601 // Only called when the inode has no links
5602 // to it (no directory entries referring to it)
5603 // and has no in-memory reference to it (is
5604 // not an open file or current directory).
5605 static void
5606 itrunc(struct inode *ip)
5607 {
5608   int i, j;
5609   struct buf *bp;
5610   uint *a;
5611
5612   for(i = 0; i < NDIRECT; i++){
5613     if(ip->addrs[i]){
5614       bfree(ip->dev, ip->addrs[i]);
5615       ip->addrs[i] = 0;
5616     }
5617   }
5618
5619   if(ip->addrs[NDIRECT]){
5620     bp = bread(ip->dev, ip->addrs[NDIRECT]);
5621     a = (uint*)bp->data;
5622     for(j = 0; j < NINDIRECT; j++){
5623       if(a[j])
5624         bfree(ip->dev, a[j]);
5625     }
5626     brelse(bp);
5627     bfree(ip->dev, ip->addrs[NDIRECT]);
5628     ip->addrs[NDIRECT] = 0;
5629   }
5630
5631   ip->size = 0;
5632   iupdate(ip);
5633 }
5634
5635 // Copy stat information from inode.
5636 void
5637 stati(struct inode *ip, struct stat *st)
5638 {
5639   st->dev = ip->dev;
5640   st->ino = ip->inum;
5641   st->type = ip->type;
5642   st->nlink = ip->nlink;
5643   st->size = ip->size;
5644 }
5645
5646
5647
5648
5649
```

```
5650 // Read data from inode.
5651 int
5652 readi(struct inode *ip, char *dst, uint off, uint n)
5653 {
5654   uint tot, m;
5655   struct buf *bp;
5656
5657   if(ip->type == T_DEV){
5658     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
5659       return -1;
5660     return devsw[ip->major].read(ip, dst, n);
5661   }
5662
5663   if(off > ip->size || off + n < off)
5664     return -1;
5665   if(off + n > ip->size)
5666     n = ip->size - off;
5667
5668   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5669     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5670     m = min(n - tot, BSIZE - off%BSIZE);
5671     /*
5672     cprintf("data off %d:\n", off);
5673     for (int j = 0; j < min(m, 10); j++) {
5674       cprintf("%x ", bp->data[off%BSIZE+j]);
5675     }
5676     cprintf("\n");
5677     */
5678     memmove(dst, bp->data + off%BSIZE, m);
5679     brelse(bp);
5680   }
5681   return n;
5682 }
5683
5684
5685
5686
5687
5688
5689
5690
5691
5692
5693
5694
5695
5696
5697
5698
5699
```



```
5700 // Write data to inode.
5701 int
5702 writei(struct inode *ip, char *src, uint off, uint n)
5703 {
5704   uint tot, m;
5705   struct buf *bp;
5706
5707   if(ip->type == T_DEV){
5708     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
5709       return -1;
5710     return devsw[ip->major].write(ip, src, n);
5711   }
5712
5713   if(off > ip->size || off + n < off)
5714     return -1;
5715   if(off + n > MAXFILE*BSIZE)
5716     return -1;
5717
5718   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
5719     bp = bread(ip->dev, bmap(ip, off/BSIZE));
5720     m = min(n - tot, BSIZE - off%BSIZE);
5721     memmove(bp->data + off%BSIZE, src, m);
5722     log_write(bp);
5723     brelse(bp);
5724   }
5725
5726   if(n > 0 && off > ip->size){
5727     ip->size = off;
5728     iupdate(ip);
5729   }
5730   return n;
5731 }
5732
5733
5734
5735
5736
5737
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749
```

```
5750 // Directories
5751
5752 int
5753 namecmp(const char *s, const char *t)
5754 {
5755   return strncmp(s, t, DIRSIZ);
5756 }
5757
5758 // Look for a directory entry in a directory.
5759 // If found, set *poff to byte offset of entry.
5760 struct inode*
5761 dirlookup(struct inode *dp, char *name, uint *poff)
5762 {
5763   uint off, inum;
5764   struct dirent de;
5765
5766   if(dp->type != T_DIR)
5767     panic("dirlookup not DIR");
5768
5769   for(off = 0; off < dp->size; off += sizeof(de)){
5770     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5771       panic("dirlink read");
5772     if(de.inum == 0)
5773       continue;
5774     if(namecmp(name, de.name) == 0){
5775       // entry matches path element
5776       if(poff)
5777         *poff = off;
5778       inum = de.inum;
5779       return iget(dp->dev, inum);
5780     }
5781   }
5782
5783   return 0;
5784 }
5785
5786
5787
5788
5789
5790
5791
5792
5793
5794
5795
5796
5797
5798
5799
```



```
5800 // Write a new directory entry (name, inum) into the directory dp.
5801 int
5802 dirlink(struct inode *dp, char *name, uint inum)
5803 {
5804   int off;
5805   struct dirent de;
5806   struct inode *ip;
5807
5808   // Check that name is not present.
5809   if((ip = dirlookup(dp, name, 0)) != 0){
5810     iput(ip);
5811     return -1;
5812   }
5813
5814   // Look for an empty dirent.
5815   for(off = 0; off < dp->size; off += sizeof(de)){
5816     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5817       panic("dirlink read");
5818     if(de.inum == 0)
5819       break;
5820   }
5821
5822   strncpy(de.name, name, DIRSIZ);
5823   de.inum = inum;
5824   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5825     panic("dirlink");
5826
5827   return 0;
5828 }
5829
5830
5831
5832
5833
5834
5835
5836
5837
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```

```
5850 // Paths
5851
5852 // Copy the next path element from path into name.
5853 // Return a pointer to the element following the copied one.
5854 // The returned path has no leading slashes,
5855 // so the caller can check *path=='\0' to see if the name is the last one.
5856 // If no name to remove, return 0.
5857 //
5858 // Examples:
5859 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5860 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5861 //   skipelem("a", name) = "", setting name = "a"
5862 //   skipelem("", name) = skipelem("////", name) = 0
5863 //
5864 static char*
5865 skipelem(char *path, char *name)
5866 {
5867   char *s;
5868   int len;
5869
5870   while(*path == '/')
5871     path++;
5872   if(*path == 0)
5873     return 0;
5874   s = path;
5875   while(*path != '/' && *path != 0)
5876     path++;
5877   len = path - s;
5878   if(len >= DIRSIZ)
5879     memmove(name, s, DIRSIZ);
5880   else {
5881     memmove(name, s, len);
5882     name[len] = 0;
5883   }
5884   while(*path == '/')
5885     path++;
5886   return path;
5887 }
5888
5889
5890
5891
5892
5893
5894
5895
5896
5897
5898
5899
```



```
5900 // Look up and return the inode for a path name.
5901 // If parent != 0, return the inode for the parent and copy the final
5902 // path element into name, which must have room for DIRSIZ bytes.
5903 // Must be called inside a transaction since it calls iput().
5904 static struct inode*
5905 namex(char *path, int nameiparent, char *name)
5906 {
5907   struct inode *ip, *next;
5908
5909   if(*path == '/')
5910     ip = iget(ROOTDEV, ROOTINO);
5911   else
5912     ip = idup(proc->cwd);
5913
5914   while((path = skipelem(path, name)) != 0){
5915     ilock(ip);
5916     if(ip->type != T_DIR){
5917       iunlockput(ip);
5918       return 0;
5919     }
5920     if(nameiparent && *path == '\0'){
5921       // Stop one level early.
5922       iunlock(ip);
5923       return ip;
5924     }
5925     if((next = dirlookup(ip, name, 0)) == 0){
5926       iunlockput(ip);
5927       return 0;
5928     }
5929     iunlockput(ip);
5930     ip = next;
5931   }
5932   if(nameiparent){
5933     iput(ip);
5934     return 0;
5935   }
5936   return ip;
5937 }
5938
5939 struct inode*
5940 namei(char *path)
5941 {
5942   char name[DIRSIZ];
5943   return namex(path, 0, name);
5944 }
5945
5946
5947
5948
5949
```

```
5950 struct inode*
5951 nameiparent(char *path, char *name)
5952 {
5953   return namex(path, 1, name);
5954 }
5955
5956
5957
5958
5959
5960
5961
5962
5963
5964
5965
5966
5967
5968
5969
5970
5971
5972
5973
5974
5975
5976
5977
5978
5979
5980
5981
5982
5983
5984
5985
5986
5987
5988
5989
5990
5991
5992
5993
5994
5995
5996
5997
5998
5999
```



```
6000 //
6001 // File descriptors
6002 //
6003
6004 #include "types.h"
6005 #include "defs.h"
6006 #include "param.h"
6007 #include "fs.h"
6008 #include "spinlock.h"
6009 #include "sleeplock.h"
6010 #include "file.h"
6011
6012 struct devsw devsw[NDEV];
6013 struct {
6014   struct spinlock lock;
6015   struct file file[NFILE];
6016 } ftable;
6017
6018 void
6019 fileinit(void)
6020 {
6021   initlock(&ftable.lock, "ftable");
6022 }
6023
6024 // Allocate a file structure.
6025 struct file*
6026 filealloc(void)
6027 {
6028   struct file *f;
6029
6030   acquire(&ftable.lock);
6031   for(f = ftable.file; f < ftable.file + NFILE; f++){
6032     if(f->ref == 0){
6033       f->ref = 1;
6034       release(&ftable.lock);
6035       return f;
6036     }
6037   }
6038   release(&ftable.lock);
6039   return 0;
6040 }
6041
6042
6043
6044
6045
6046
6047
6048
6049
```

```
6050 // Increment ref count for file f.
6051 struct file*
6052 filedup(struct file *f)
6053 {
6054   acquire(&ftable.lock);
6055   if(f->ref < 1)
6056     panic("filedup");
6057   f->ref++;
6058   release(&ftable.lock);
6059   return f;
6060 }
6061
6062 // Close file f.  (Decrement ref count, close when reaches 0.)
6063 void
6064 fileclose(struct file *f)
6065 {
6066   struct file ff;
6067
6068   acquire(&ftable.lock);
6069   if(f->ref < 1)
6070     panic("fileclose");
6071   if(--f->ref > 0){
6072     release(&ftable.lock);
6073     return;
6074   }
6075   ff = *f;
6076   f->ref = 0;
6077   f->type = FD_NONE;
6078   release(&ftable.lock);
6079
6080   if(ff.type == FD_PIPE)
6081     pipeclose(ff.pipe, ff.writable);
6082   else if(ff.type == FD_INODE){
6083     begin_op();
6084     iput(ff.ip);
6085     end_op();
6086   }
6087 }
6088
6089
6090
6091
6092
6093
6094
6095
6096
6097
6098
6099
```



```
6100 // Get metadata about file f.
6101 int
6102 filestat(struct file *f, struct stat *st)
6103 {
6104   if(f->type == FD_INODE){
6105     ilock(f->ip);
6106     stati(f->ip, st);
6107     iunlock(f->ip);
6108     return 0;
6109   }
6110   return -1;
6111 }
6112
6113 // Read from file f.
6114 int
6115 fileread(struct file *f, char *addr, int n)
6116 {
6117   int r;
6118
6119   if(f->readable == 0)
6120     return -1;
6121   if(f->type == FD_PIPE)
6122     return piperead(f->pipe, addr, n);
6123   if(f->type == FD_INODE){
6124     ilock(f->ip);
6125     if((r = readi(f->ip, addr, f->off, n)) > 0)
6126       f->off += r;
6127     iunlock(f->ip);
6128     return r;
6129   }
6130   panic("fileread");
6131 }
6132
6133
6134
6135
6136
6137
6138
6139
6140
6141
6142
6143
6144
6145
6146
6147
6148
6149
```

```
6150 // Write to file f.
6151 int
6152 filewrite(struct file *f, char *addr, int n)
6153 {
6154   int r;
6155
6156   if(f->writable == 0)
6157     return -1;
6158   if(f->type == FD_PIPE)
6159     return pipewrite(f->pipe, addr, n);
6160   if(f->type == FD_INODE){
6161     // write a few blocks at a time to avoid exceeding
6162     // the maximum log transaction size, including
6163     // i-node, indirect block, allocation blocks,
6164     // and 2 blocks of slop for non-aligned writes.
6165     // this really belongs lower down, since writei()
6166     // might be writing a device like the console.
6167     int max = ((LOGSIZE-1-1-2) / 2) * 512;
6168     int i = 0;
6169     while(i < n){
6170       int n1 = n - i;
6171       if(n1 > max)
6172         n1 = max;
6173
6174       begin_op();
6175       ilock(f->ip);
6176       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
6177         f->off += r;
6178       iunlock(f->ip);
6179       end_op();
6180
6181       if(r < 0)
6182         break;
6183       if(r != n1)
6184         panic("short filewrite");
6185       i += r;
6186     }
6187     return i == n ? n : -1;
6188   }
6189   panic("filewrite");
6190 }
6191
6192
6193
6194
6195
6196
6197
6198
6199
```



```
6200 //
6201 // File-system system calls.
6202 // Mostly argument checking, since we don't trust
6203 // user code, and calls into file.c and fs.c.
6204 //
6205
6206 #include "types.h"
6207 #include "defs.h"
6208 #include "param.h"
6209 #include "stat.h"
6210 #include "mmu.h"
6211 #include "proc.h"
6212 #include "fs.h"
6213 #include "spinlock.h"
6214 #include "sleeplock.h"
6215 #include "file.h"
6216 #include "fcntl.h"
6217
6218 // Fetch the nth word-sized system call argument as a file descriptor
6219 // and return both the descriptor and the corresponding struct file.
6220 static int
6221 argfd(int n, int *pfd, struct file **pf)
6222 {
6223   int fd;
6224   struct file *f;
6225
6226   if(argint(n, &fd) < 0)
6227     return -1;
6228   if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
6229     return -1;
6230   if(pfd)
6231     *pfd = fd;
6232   if(pf)
6233     *pf = f;
6234   return 0;
6235 }
6236
6237
6238
6239
6240
6241
6242
6243
6244
6245
6246
6247
6248
6249
```

```
6250 // Allocate a file descriptor for the given file.
6251 // Takes over file reference from caller on success.
6252 static int
6253 fdalloc(struct file *f)
6254 {
6255   int fd;
6256
6257   for(fd = 0; fd < NOFILE; fd++){
6258     if(proc->ofile[fd] == 0){
6259       proc->ofile[fd] = f;
6260       return fd;
6261     }
6262   }
6263   return -1;
6264 }
6265
6266 int
6267 sys_dup(void)
6268 {
6269   struct file *f;
6270   int fd;
6271
6272   if(argfd(0, 0, &f) < 0)
6273     return -1;
6274   if((fd=fdalloc(f)) < 0)
6275     return -1;
6276   filedup(f);
6277   return fd;
6278 }
6279
6280 int
6281 sys_read(void)
6282 {
6283   struct file *f;
6284   int n;
6285   char *p;
6286
6287   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6288     return -1;
6289   return fileread(f, p, n);
6290 }
6291
6292
6293
6294
6295
6296
6297
6298
6299
```



```
6300 int
6301 sys_write(void)
6302 {
6303   struct file *f;
6304   int n;
6305   char *p;
6306
6307   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6308     return -1;
6309   return filewrite(f, p, n);
6310 }
6311
6312 int
6313 sys_close(void)
6314 {
6315   int fd;
6316   struct file *f;
6317
6318   if(argfd(0, &fd, &f) < 0)
6319     return -1;
6320   proc->ofile[fd] = 0;
6321   fileclose(f);
6322   return 0;
6323 }
6324
6325 int
6326 sys_fstat(void)
6327 {
6328   struct file *f;
6329   struct stat *st;
6330
6331   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
6332     return -1;
6333   return filestat(f, st);
6334 }
6335
6336
6337
6338
6339
6340
6341
6342
6343
6344
6345
6346
6347
6348
6349
```

```
// Create the path new as a link to the same inode as old.
int
sys_link(void)
{
  char name[DIRSIZ], *new, *old;
  struct inode *dp, *ip;

  if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
    return -1;

  begin_op();
  if((ip = namei(old)) == 0){
    end_op();
    return -1;
  }

  ilock(ip);
  if(ip->type == T_DIR){
    iunlockput(ip);
    end_op();
    return -1;
  }

  ip->nlink++;
  iupdate(ip);
  iunlock(ip);

  if((dp = nameiparent(new, name)) == 0)
    goto bad;
  ilock(dp);
  if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
    iunlockput(dp);
    goto bad;
  }
  iunlockput(dp);
  iput(ip);

  end_op();

  return 0;

bad:
  ilock(ip);
  ip->nlink--;
  iupdate(ip);
  iunlockput(ip);
  end_op();
  return -1;
}

```



```
// Is the directory dp empty except for "." and ".." ?
static int
isdirempty(struct inode *dp)
{
  int off;
  struct dirent de;

  for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("isdirempty: readi");
    if(de.inum != 0)
      return 0;
  }
  return 1;
}
```

```
6450 int
6451 sys_unlink(void)
6452 {
6453   struct inode *ip, *dp;
6454   struct dirent de;
6455   char name[DIRSIZ], *path;
6456   uint off;
6457
6458   if(argstr(0, &path) < 0)
6459     return -1;
6460
6461   begin_op();
6462   if((dp = nameiparent(path, name)) == 0){
6463     end_op();
6464     return -1;
6465   }
6466
6467   ilock(dp);
6468
6469   // Cannot unlink "." or "..".
6470   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6471     goto bad;
6472
6473   if((ip = dirlookup(dp, name, &off)) == 0)
6474     goto bad;
6475   ilock(ip);
6476
6477   if(ip->nlink < 1)
6478     panic("unlink: nlink < 1");
6479   if(ip->type == T_DIR && !isdirempty(ip)){
6480     iunlockput(ip);
6481     goto bad;
6482   }
6483
6484   memset(&de, 0, sizeof(de));
6485   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6486     panic("unlink: writei");
6487   if(ip->type == T_DIR){
6488     dp->nlink--;
6489     iupdate(dp);
6490   }
6491   iunlockput(dp);
6492
6493   ip->nlink--;
6494   iupdate(ip);
6495   iunlockput(ip);
6496
6497   end_op();
6498
6499   return 0;
```



```
6500 bad:
6501   iunlockput(dp);
6502   end_op();
6503   return -1;
6504 }
6505
6506 static struct inode*
6507 create(char *path, short type, short major, short minor)
6508 {
6509   uint off;
6510   struct inode *ip, *dp;
6511   char name[DIRSIZ];
6512
6513   if((dp = nameiparent(path, name)) == 0)
6514     return 0;
6515   ilock(dp);
6516
6517   if((ip = dirlookup(dp, name, &off)) != 0){
6518     iunlockput(dp);
6519     ilock(ip);
6520     if(type == T_FILE && ip->type == T_FILE)
6521       return ip;
6522     iunlockput(ip);
6523     return 0;
6524   }
6525
6526   if((ip = ialloc(dp->dev, type)) == 0)
6527     panic("create: ialloc");
6528
6529   ilock(ip);
6530   ip->major = major;
6531   ip->minor = minor;
6532   ip->nlink = 1;
6533   iupdate(ip);
6534
6535   if(type == T_DIR){  // Create . and .. entries.
6536     dp->nlink++;  // for ".."
6537     iupdate(dp);
6538     // No ip->nlink++ for ".": avoid cyclic ref count.
6539     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6540       panic("create dots");
6541   }
6542
6543   if(dirlink(dp, name, ip->inum) < 0)
6544     panic("create: dirlink");
6545
6546   iunlockput(dp);
6547
6548   return ip;
6549 }
```

```
6550 int
6551 sys_open(void)
6552 {
6553   char *path;
6554   int fd, omode;
6555   struct file *f;
6556   struct inode *ip;
6557
6558   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6559     return -1;
6560
6561   begin_op();
6562
6563   if(omode & O_CREATE){
6564     ip = create(path, T_FILE, 0, 0);
6565     if(ip == 0){
6566       end_op();
6567       return -1;
6568     }
6569   } else {
6570     if((ip = namei(path)) == 0){
6571       end_op();
6572       return -1;
6573     }
6574     ilock(ip);
6575     if(ip->type == T_DIR && omode != O_RDONLY){
6576       iunlockput(ip);
6577       end_op();
6578       return -1;
6579     }
6580   }
6581
6582   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6583     if(f)
6584       fileclose(f);
6585     iunlockput(ip);
6586     end_op();
6587     return -1;
6588   }
6589   iunlock(ip);
6590   end_op();
6591
6592   f->type = FD_INODE;
6593   f->ip = ip;
6594   f->off = 0;
6595   f->readable = !(omode & O_WRONLY);
6596   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6597   return fd;
6598 }
6599
```



```
6600 int
6601 sys_mkdir(void)
6602 {
6603   char *path;
6604   struct inode *ip;
6605
6606   begin_op();
6607   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
6608     end_op();
6609     return -1;
6610   }
6611   iunlockput(ip);
6612   end_op();
6613   return 0;
6614 }
6615
6616 int
6617 sys_mknod(void)
6618 {
6619   struct inode *ip;
6620   char *path;
6621   int major, minor;
6622
6623   begin_op();
6624   if((argstr(0, &path)) < 0 ||
6625      argint(1, &major) < 0 ||
6626      argint(2, &minor) < 0 ||
6627      (ip = create(path, T_DEV, major, minor)) == 0){
6628     end_op();
6629     return -1;
6630   }
6631   iunlockput(ip);
6632   end_op();
6633   return 0;
6634 }
6635
6636
6637
6638
6639
6640
6641
6642
6643
6644
6645
6646
6647
6648
6649
```

```
6650 int
6651 sys_chdir(void)
6652 {
6653   char *path;
6654   struct inode *ip;
6655
6656   begin_op();
6657   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
6658     end_op();
6659     return -1;
6660   }
6661   ilock(ip);
6662   if(ip->type != T_DIR){
6663     iunlockput(ip);
6664     end_op();
6665     return -1;
6666   }
6667   iunlock(ip);
6668   iput(proc->cwd);
6669   end_op();
6670   proc->cwd = ip;
6671   return 0;
6672 }
6673
6674 int
6675 sys_exec(void)
6676 {
6677   char *path, *argv[MAXARG];
6678   int i;
6679   uint uargv, uarg;
6680
6681   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6682     return -1;
6683   }
6684   memset(argv, 0, sizeof(argv));
6685   for(i=0;; i++){
6686     if(i >= NELEM(argv))
6687       return -1;
6688     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6689       return -1;
6690     if(uarg == 0){
6691       argv[i] = 0;
6692       break;
6693     }
6694     if(fetchstr(uarg, &argv[i]) < 0)
6695       return -1;
6696   }
6697   return exec(path, argv);
6698 }
6699
```



```
6700 int
6701 sys_pipe(void)
6702 {
6703   int *fd;
6704   struct file *rf, *wf;
6705   int fd0, fd1;
6706
6707   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6708     return -1;
6709   if(pipealloc(&rf, &wf) < 0)
6710     return -1;
6711   fd0 = -1;
6712   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6713     if(fd0 >= 0)
6714       proc->ofile[fd0] = 0;
6715     fileclose(rf);
6716     fileclose(wf);
6717     return -1;
6718   }
6719   fd[0] = fd0;
6720   fd[1] = fd1;
6721   return 0;
6722 }
6723
6724
6725
6726
6727
6728
6729
6730
6731
6732
6733
6734
6735
6736
6737
6738
6739
6740
6741
6742
6743
6744
6745
6746
6747
6748
6749
```

```
6750 #include "types.h"
6751 #include "traps.h"
6752 #include "param.h"
6753 #include "memlayout.h"
6754 #include "mmu.h"
6755 #include "proc.h"
6756 #include "defs.h"
6757 #include "syscall.h"
6758 #include "x86.h"
6759 #include "elf.h"
6760
6761 void
6762 pseudo_main(int (*entry)(int, char**), int argc, char **argv)
6763 {
6764     int stat =entry(argc, argv);
6765
6766      asm("pushl %%eax\n"
6767     "pushl %%eax\n"
6768     "movl $2, %%eax\n"
6769     "int %1" :: "a"(stat), "i" (T_SYSCALL));
6770
6771 }
6772
6773 int
6774 exec(char *path, char **argv)
6775 {
6776   char *s, *last;
6777   int i, off;
6778   uint argc, sz, sp, ustack[3+MAXARG+1];
6779   uint pointer_pseudo_main;
6780   struct elfhdr elf;
6781   struct inode *ip;
6782   struct proghdr ph;
6783   pde_t *pgdir, *oldpgdir;
6784
6785   begin_op();
6786
6787   if((ip = namei(path)) == 0){
6788     end_op();
6789     return -1;
6790   }
6791   ilock(ip);
6792   pgdir = 0;
6793
6794   // Check ELF header
6795   if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
6796     goto bad;
6797   if(elf.magic != ELF_MAGIC)
6798     goto bad;
6799
```



```
6800   if((pgdir = setupkvm()) == 0)
6801     goto bad;
6802
6803   // Load program into memory.
6804   sz = 0;
6805   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6806     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6807       goto bad;
6808     if(ph.type != ELF_PROG_LOAD)
6809       continue;
6810     if(ph.memsz < ph.filesz)
6811       goto bad;
6812     if(ph.vaddr + ph.memsz < ph.vaddr)
6813       goto bad;
6814     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6815       goto bad;
6816     if(ph.vaddr % PGSIZE != 0)
6817       goto bad;
6818     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6819       goto bad;
6820   }
6821   iunlockput(ip);
6822   end_op();
6823   ip = 0;
6824
6825   pointer_pseudo_main = sz;
6826
6827
6828   // Allocate two pages at the next page boundary.
6829   // Make the first inaccessible.  Use the second as the user stack.
6830   sz = PGROUNDUP(sz);
6831   if((sz = allocuvm(pgdir, sz, sz + 3*PGSIZE)) == 0)
6832     goto bad;
6833   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6834
6835   if (copyout(pgdir, pointer_pseudo_main, pseudo_main, (uint)exec - (uint)ps
6836     goto bad;
6837
6838   sp = sz;
6839
6840   // Push argument strings, prepare rest of stack in ustack.
6841   for(argc = 0; argv[argc]; argc++) {
6842     if(argc >= MAXARG)
6843       goto bad;
6844     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6845     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6846       goto bad;
6847     ustack[3+argc] = sp;
6848   }
6849   ustack[3+argc] = 0;
```

```
6850   ustack[0] = 0xffffffff;  // fake return PC
6851   ustack[1]=elf.entry;
6852   ustack[2] = argc;
6853   ustack[3] = sp - (argc+1)*4;  // argv pointer
6854
6855   sp -= (3+argc+1) * 4;
6856   if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6857     goto bad;
6858
6859   // Save program name for debugging.
6860   for(last=s=path; *s; s++)
6861     if(*s == '/')
6862       last = s+1;
6863   safestrcpy(proc->name, last, sizeof(proc->name));
6864
6865   // Commit to the user image.
6866   oldpgdir = proc->pgdir;
6867   proc->pgdir = pgdir;
6868   proc->sz = sz;
6869   proc->tf->eip = pointer_pseudo_main;  // main
6870   proc->tf->esp = sp;
6871   switchuvm(proc);
6872   freevm(oldpgdir);
6873   return 0;
6874
6875  bad:
6876   if(pgdir)
6877     freevm(pgdir);
6878   if(ip){
6879     iunlockput(ip);
6880     end_op();
6881   }
6882   return -1;
6883 }
6884
6885
6886
6887
6888
6889
6890
6891
6892
6893
6894
6895
6896
6897
6898
6899
```



```
6900 #include "types.h"
6901 #include "defs.h"
6902 #include "param.h"
6903 #include "mmu.h"
6904 #include "proc.h"
6905 #include "fs.h"
6906 #include "spinlock.h"
6907 #include "sleeplock.h"
6908 #include "file.h"
6909
6910 #define PIPESIZE 512
6911
6912 struct pipe {
6913   struct spinlock lock;
6914   char data[PIPESIZE];
6915   uint nread;     // number of bytes read
6916   uint nwrite;    // number of bytes written
6917   int readopen;   // read fd is still open
6918   int writeopen;  // write fd is still open
6919 };
6920
6921 int
6922 pipealloc(struct file **f0, struct file **f1)
6923 {
6924   struct pipe *p;
6925
6926   p = 0;
6927   *f0 = *f1 = 0;
6928   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
6929     goto bad;
6930   if((p = (struct pipe*)kalloc()) == 0)
6931     goto bad;
6932   p->readopen = 1;
6933   p->writeopen = 1;
6934   p->nwrite = 0;
6935   p->nread = 0;
6936   initlock(&p->lock, "pipe");
6937   (*f0)->type = FD_PIPE;
6938   (*f0)->readable = 1;
6939   (*f0)->writable = 0;
6940   (*f0)->pipe = p;
6941   (*f1)->type = FD_PIPE;
6942   (*f1)->readable = 0;
6943   (*f1)->writable = 1;
6944   (*f1)->pipe = p;
6945   return 0;
6946
6947
6948
6949
```

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

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


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

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



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

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

```
7050 #include "types.h"
7051 #include "x86.h"
7052
7053 void*
7054 memset(void *dst, int c, uint n)
7055 {
7056   if ((int)dst%4 == 0 && n%4 == 0){
7057     c &= 0xFF;
7058     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
7059   } else
7060     stosb(dst, c, n);
7061   return dst;
7062 }
7063
7064 int
7065 memcmp(const void *v1, const void *v2, uint n)
7066 {
7067   const uchar *s1, *s2;
7068
7069   s1 = v1;
7070   s2 = v2;
7071   while(n-- > 0){
7072     if(*s1 != *s2)
7073       return *s1 - *s2;
7074     s1++, s2++;
7075   }
7076
7077   return 0;
7078 }
7079
7080 void*
7081 memmove(void *dst, const void *src, uint n)
7082 {
7083   const char *s;
7084   char *d;
7085
7086   s = src;
7087   d = dst;
7088   if(s < d && s + n > d){
7089     s += n;
7090     d += n;
7091     while(n-- > 0)
7092       *--d = *--s;
7093   } else
7094     while(n-- > 0)
7095       *d++ = *s++;
7096
7097   return dst;
7098 }
7099
```



```
7100 // memcpy exists to placate GCC.  Use memmove.
7101 void*
7102 memcpy(void *dst, const void *src, uint n)
7103 {
7104   return memmove(dst, src, n);
7105 }
7106
7107 int
7108 strncmp(const char *p, const char *q, uint n)
7109 {
7110   while(n > 0 && *p && *p == *q)
7111     n--, p++, q++;
7112   if(n == 0)
7113     return 0;
7114   return (uchar)*p - (uchar)*q;
7115 }
7116
7117 char*
7118 strncpy(char *s, const char *t, int n)
7119 {
7120   char *os;
7121
7122   os = s;
7123   while(n-- > 0 && (*s++ = *t++) != 0)
7124     ;
7125   while(n-- > 0)
7126     *s++ = 0;
7127   return os;
7128 }
7129
7130 // Like strncpy but guaranteed to NUL-terminate.
7131 char*
7132 safestrcpy(char *s, const char *t, int n)
7133 {
7134   char *os;
7135
7136   os = s;
7137   if(n <= 0)
7138     return os;
7139   while(--n > 0 && (*s++ = *t++) != 0)
7140     ;
7141   *s = 0;
7142   return os;
7143 }
7144
7145
7146
7147
7148
7149
```

```
7150 int
7151 strlen(const char *s)
7152 {
7153   int n;
7154
7155   for(n = 0; s[n]; n++)
7156     ;
7157   return n;
7158 }
7159
7160
7161
7162
7163
7164
7165
7166
7167
7168
7169
7170
7171
7172
7173
7174
7175
7176
7177
7178
7179
7180
7181
7182
7183
7184
7185
7186
7187
7188
7189
7190
7191
7192
7193
7194
7195
7196
7197
7198
7199
```



```
7200 // See MultiProcessor Specification Version 1.[14]
7201
7202 struct mp {             // floating pointer
7203   uchar signature[4];           // "_MP_"
7204   void *physaddr;               // phys addr of MP config table
7205   uchar length;                 // 1
7206   uchar specrev;                // [14]
7207   uchar checksum;               // all bytes must add up to 0
7208   uchar type;                   // MP system config type
7209   uchar imcrp;
7210   uchar reserved[3];
7211 };
7212
7213 struct mpconf {         // configuration table header
7214   uchar signature[4];           // "PCMP"
7215   ushort length;                // total table length
7216   uchar version;                // [14]
7217   uchar checksum;               // all bytes must add up to 0
7218   uchar product[20];            // product id
7219   uint *oemtable;               // OEM table pointer
7220   ushort oemlength;             // OEM table length
7221   ushort entry;                 // entry count
7222   uint *lapicaddr;              // address of local APIC
7223   ushort xlength;               // extended table length
7224   uchar xchecksum;              // extended table checksum
7225   uchar reserved;
7226 };
7227
7228 struct mpproc {         // processor table entry
7229   uchar type;                   // entry type (0)
7230   uchar apicid;                 // local APIC id
7231   uchar version;                // local APIC verison
7232   uchar flags;                  // CPU flags
7233     #define MPBOOT 0x02           // This proc is the bootstrap processor.
7234   uchar signature[4];           // CPU signature
7235   uint feature;                 // feature flags from CPUID instruction
7236   uchar reserved[8];
7237 };
7238
7239 struct mpioapic {       // I/O APIC table entry
7240   uchar type;                   // entry type (2)
7241   uchar apicno;                 // I/O APIC id
7242   uchar version;                // I/O APIC version
7243   uchar flags;                  // I/O APIC flags
7244   uint *addr;                  // I/O APIC address
7245 };
7246
7247
7248
7249
```

```
7250 // Table entry types
7251 #define MPPROC    0x00  // One per processor
7252 #define MPBUS     0x01  // One per bus
7253 #define MPIOAPIC  0x02  // One per I/O APIC
7254 #define MPIOINTR  0x03  // One per bus interrupt source
7255 #define MPLINTR   0x04  // One per system interrupt source
7256
7257
7258
7259
7260
7261
7262
7263
7264
7265
7266
7267
7268
7269
7270
7271
7272
7273
7274
7275
7276
7277
7278
7279
7280
7281
7282
7283
7284
7285
7286
7287
7288
7289
7290
7291
7292
7293
7294
7295
7296
7297
7298
7299
```



```
7300 // Blank page.
7301
7302
7303
7304
7305
7306
7307
7308
7309
7310
7311
7312
7313
7314
7315
7316
7317
7318
7319
7320
7321
7322
7323
7324
7325
7326
7327
7328
7329
7330
7331
7332
7333
7334
7335
7336
7337
7338
7339
7340
7341
7342
7343
7344
7345
7346
7347
7348
7349
```

```
7350 // Multiprocessor support
7351 // Search memory for MP description structures.
7352 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
7353
7354 #include "types.h"
7355 #include "defs.h"
7356 #include "param.h"
7357 #include "memlayout.h"
7358 #include "mp.h"
7359 #include "x86.h"
7360 #include "mmu.h"
7361 #include "proc.h"
7362
7363 struct cpu cpus[NCPU];
7364 int ismp;
7365 int ncpu;
7366 uchar ioapicid;
7367
7368 static uchar
7369 sum(uchar *addr, int len)
7370 {
7371   int i, sum;
7372
7373   sum = 0;
7374   for(i=0; i<len; i++)
7375     sum += addr[i];
7376   return sum;
7377 }
7378
7379 // Look for an MP structure in the len bytes at addr.
7380 static struct mp*
7381 mpsearch1(uint a, int len)
7382 {
7383   uchar *e, *p, *addr;
7384
7385   addr = P2V(a);
7386   e = addr+len;
7387   for(p = addr; p < e; p += sizeof(struct mp))
7388     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
7389       return (struct mp*)p;
7390   return 0;
7391 }
7392
7393
7394
7395
7396
7397
7398
7399
```



```
7400 // Search for the MP Floating Pointer Structure, which according to the
7401 // spec is in one of the following three locations:
7402 // 1) in the first KB of the EBDA;
7403 // 2) in the last KB of system base memory;
7404 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
7405 static struct mp*
7406 mpsearch(void)
7407 {
7408   uchar *bda;
7409   uint p;
7410   struct mp *mp;
7411
7412   bda = (uchar *) P2V(0x400);
7413   if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
7414     if((mp = mpsearch1(p, 1024)))
7415       return mp;
7416   } else {
7417     p = ((bda[0x14]<<8)|bda[0x13])*1024;
7418     if((mp = mpsearch1(p-1024, 1024)))
7419       return mp;
7420   }
7421   return mpsearch1(0xF0000, 0x10000);
7422 }
7423
7424 // Search for an MP configuration table.  For now,
7425 // don't accept the default configurations (physaddr == 0).
7426 // Check for correct signature, calculate the checksum and,
7427 // if correct, check the version.
7428 // To do: check extended table checksum.
7429 static struct mpconf*
7430 mpconfig(struct mp **pmp)
7431 {
7432   struct mpconf *conf;
7433   struct mp *mp;
7434
7435   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
7436     return 0;
7437   conf = (struct mpconf*) P2V((uint) mp->physaddr);
7438   if(memcmp(conf, "PCMP", 4) != 0)
7439     return 0;
7440   if(conf->version != 1 && conf->version != 4)
7441     return 0;
7442   if(sum((uchar*)conf, conf->length) != 0)
7443     return 0;
7444   *pmp = mp;
7445   return conf;
7446 }
7447
7448
7449
```

```
7450 void
7451 mpinit(void)
7452 {
7453   uchar *p, *e;
7454   struct mp *mp;
7455   struct mpconf *conf;
7456   struct mpproc *proc;
7457   struct mpioapic *ioapic;
7458
7459   if((conf = mpconfig(&mp)) == 0)
7460     return;
7461   ismp = 1;
7462   lapic = (uint*)conf->lapicaddr;
7463   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
7464     switch(*p){
7465     case MPPROC:
7466       proc = (struct mpproc*)p;
7467       if(ncpu < NCPU) {
7468         cpus[ncpu].apicid = proc->apicid;  // apicid may differ from ncpu
7469         ncpu++;
7470       }
7471       p += sizeof(struct mpproc);
7472       continue;
7473     case MPIOAPIC:
7474       ioapic = (struct mpioapic*)p;
7475       ioapicid = ioapic->apicno;
7476       p += sizeof(struct mpioapic);
7477       continue;
7478     case MPBUS:
7479     case MPIOINTR:
7480     case MPLINTR:
7481       p += 8;
7482       continue;
7483     default:
7484       ismp = 0;
7485       break;
7486     }
7487   }
7488   if(!ismp){
7489     // Didn't like what we found; fall back to no MP.
7490     ncpu = 1;
7491     lapic = 0;
7492     ioapicid = 0;
7493     return;
7494   }
7495
7496
7497
7498
7499
```



```
7500   if(mp->imcrp){
7501     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
7502     // But it would on real hardware.
7503     outb(0x22, 0x70);   // Select IMCR
7504     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
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 // The local APIC manages internal (non-I/O) interrupts.
7551 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
7552
7553 #include "param.h"
7554 #include "types.h"
7555 #include "defs.h"
7556 #include "date.h"
7557 #include "memlayout.h"
7558 #include "traps.h"
7559 #include "mmu.h"
7560 #include "x86.h"
7561 #include "proc.h"  // ncpu
7562
7563 // Local APIC registers, divided by 4 for use as uint[] indices.
7564 #define ID      (0x0020/4)   // ID
7565 #define VER     (0x0030/4)   // Version
7566 #define TPR     (0x0080/4)   // Task Priority
7567 #define EOI     (0x00B0/4)   // EOI
7568 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
7569   #define ENABLE     0x00000100   // Unit Enable
7570 #define ESR     (0x0280/4)   // Error Status
7571 #define ICRLO   (0x0300/4)   // Interrupt Command
7572   #define INIT       0x00000500   // INIT/RESET
7573   #define STARTUP    0x00000600   // Startup IPI
7574   #define DELIVS     0x00001000   // Delivery status
7575   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
7576   #define DEASSERT   0x00000000
7577   #define LEVEL      0x00008000   // Level triggered
7578   #define BCAST      0x00080000   // Send to all APICs, including self.
7579   #define BUSY       0x00001000
7580   #define FIXED      0x00000000
7581 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
7582 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
7583   #define X1         0x0000000B   // divide counts by 1
7584   #define PERIODIC   0x00020000   // Periodic
7585 #define PCINT   (0x0340/4)   // Performance Counter LVT
7586 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
7587 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
7588 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
7589   #define MASKED     0x00010000   // Interrupt masked
7590 #define TICR    (0x0380/4)   // Timer Initial Count
7591 #define TCCR    (0x0390/4)   // Timer Current Count
7592 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
7593
7594 volatile uint *lapic;  // Initialized in mp.c
7595
7596
7597
7598
7599
```



```
7600 static void
7601 lapicw(int index, int value)
7602 {
7603   lapic[index] = value;
7604   lapic[ID];  // wait for write to finish, by reading
7605 }
7606
7607
7608
7609
7610
7611
7612
7613
7614
7615
7616
7617
7618
7619
7620
7621
7622
7623
7624
7625
7626
7627
7628
7629
7630
7631
7632
7633
7634
7635
7636
7637
7638
7639
7640
7641
7642
7643
7644
7645
7646
7647
7648
7649
```

```
7650 void
7651 lapicinit(void)
7652 {
7653   if(!lapic)
7654     return;
7655
7656   // Enable local APIC; set spurious interrupt vector.
7657   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
7658
7659   // The timer repeatedly counts down at bus frequency
7660   // from lapic[TICR] and then issues an interrupt.
7661   // If xv6 cared more about precise timekeeping,
7662   // TICR would be calibrated using an external time source.
7663   lapicw(TDCR, X1);
7664   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
7665   lapicw(TICR, 10000000);
7666
7667   // Disable logical interrupt lines.
7668   lapicw(LINT0, MASKED);
7669   lapicw(LINT1, MASKED);
7670
7671   // Disable performance counter overflow interrupts
7672   // on machines that provide that interrupt entry.
7673   if(((lapic[VER]>>16) & 0xFF) >= 4)
7674     lapicw(PCINT, MASKED);
7675
7676   // Map error interrupt to IRQ_ERROR.
7677   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
7678
7679   // Clear error status register (requires back-to-back writes).
7680   lapicw(ESR, 0);
7681   lapicw(ESR, 0);
7682
7683   // Ack any outstanding interrupts.
7684   lapicw(EOI, 0);
7685
7686   // Send an Init Level De-Assert to synchronise arbitration ID's.
7687   lapicw(ICRHI, 0);
7688   lapicw(ICRLO, BCAST | INIT | LEVEL);
7689   while(lapic[ICRLO] & DELIVS)
7690     ;
7691
7692   // Enable interrupts on the APIC (but not on the processor).
7693   lapicw(TPR, 0);
7694 }
7695
7696
7697
7698
7699
```



```
7700 int
7701 cpunum(void)
7702 {
7703   int apicid, i;
7704
7705   // Cannot call cpu when interrupts are enabled:
7706   // result not guaranteed to last long enough to be used!
7707   // Would prefer to panic but even printing is chancy here:
7708   // almost everything, including cprintf and panic, calls cpu,
7709   // often indirectly through acquire and release.
7710   if(readeflags()&FL_IF){
7711     static int n;
7712     if(n++ == 0)
7713       cprintf("cpu called from %x with interrupts enabled\n",
7714         __builtin_return_address(0));
7715   }
7716
7717   if (!lapic)
7718     return 0;
7719
7720   apicid = lapic[ID] >> 24;
7721   for (i = 0; i < ncpu; ++i) {
7722     if (cpus[i].apicid == apicid)
7723       return i;
7724   }
7725   panic("unknown apicid\n");
7726 }
7727
7728 // Acknowledge interrupt.
7729 void
7730 lapiceoi(void)
7731 {
7732   if(lapic)
7733     lapicw(EOI, 0);
7734 }
7735
7736 // Spin for a given number of microseconds.
7737 // On real hardware would want to tune this dynamically.
7738 void
7739 microdelay(int us)
7740 {
7741 }
7742
7743
7744
7745
7746
7747
7748
7749
```

```
7750 #define CMOS_PORT    0x70
7751 #define CMOS_RETURN  0x71
7752
7753 // Start additional processor running entry code at addr.
7754 // See Appendix B of MultiProcessor Specification.
7755 void
7756 lapicstartap(uchar apicid, uint addr)
7757 {
7758   int i;
7759   ushort *wrv;
7760
7761   // "The BSP must initialize CMOS shutdown code to 0AH
7762   // and the warm reset vector (DWORD based at 40:67) to point at
7763   // the AP startup code prior to the [universal startup algorithm]."
7764   outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
7765   outb(CMOS_PORT+1, 0x0A);
7766   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
7767   wrv[0] = 0;
7768   wrv[1] = addr >> 4;
7769
7770   // "Universal startup algorithm."
7771   // Send INIT (level-triggered) interrupt to reset other CPU.
7772   lapicw(ICRHI, apicid<<24);
7773   lapicw(ICRLO, INIT | LEVEL | ASSERT);
7774   microdelay(200);
7775   lapicw(ICRLO, INIT | LEVEL);
7776   microdelay(100);    // should be 10ms, but too slow in Bochs!
7777
7778   // Send startup IPI (twice!) to enter code.
7779   // Regular hardware is supposed to only accept a STARTUP
7780   // when it is in the halted state due to an INIT.  So the second
7781   // should be ignored, but it is part of the official Intel algorithm.
7782   // Bochs complains about the second one.  Too bad for Bochs.
7783   for(i = 0; i < 2; i++){
7784     lapicw(ICRHI, apicid<<24);
7785     lapicw(ICRLO, STARTUP | (addr>>12));
7786     microdelay(200);
7787   }
7788 }
7789
7790
7791
7792
7793
7794
7795
7796
7797
7798
7799
```



```
7800 #define CMOS_STATA   0x0a
7801 #define CMOS_STATB   0x0b
7802 #define CMOS_UIP    (1 << 7)        // RTC update in progress
7803
7804 #define SECS    0x00
7805 #define MINS    0x02
7806 #define HOURS   0x04
7807 #define DAY     0x07
7808 #define MONTH   0x08
7809 #define YEAR    0x09
7810
7811 static uint cmos_read(uint reg)
7812 {
7813   outb(CMOS_PORT,  reg);
7814   microdelay(200);
7815
7816   return inb(CMOS_RETURN);
7817 }
7818
7819 static void fill_rtcdate(struct rtcdate *r)
7820 {
7821   r->second = cmos_read(SECS);
7822   r->minute = cmos_read(MINS);
7823   r->hour   = cmos_read(HOURS);
7824   r->day    = cmos_read(DAY);
7825   r->month  = cmos_read(MONTH);
7826   r->year   = cmos_read(YEAR);
7827 }
7828
7829 // qemu seems to use 24-hour GWT and the values are BCD encoded
7830 void cmostime(struct rtcdate *r)
7831 {
7832   struct rtcdate t1, t2;
7833   int sb, bcd;
7834
7835   sb = cmos_read(CMOS_STATB);
7836
7837   bcd = (sb & (1 << 2)) == 0;
7838
7839   // make sure CMOS doesn't modify time while we read it
7840   for(;;) {
7841     fill_rtcdate(&t1);
7842     if(cmos_read(CMOS_STATA) & CMOS_UIP)
7843         continue;
7844     fill_rtcdate(&t2);
7845     if(memcmp(&t1, &t2, sizeof(t1)) == 0)
7846       break;
7847   }
7848
7849
```

```
  // convert
  if(bcd) {
#define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
    CONV(second);
    CONV(minute);
    CONV(hour  );
    CONV(day   );
    CONV(month );
    CONV(year  );
#undef     CONV
  }

  *r = t1;
  r->year += 2000;
}
```



```
// The I/O APIC manages hardware interrupts for an SMP system.
// http://www.intel.com/design/chipsets/datashts/29056601.pdf
// See also picirq.c.

#include "types.h"
#include "defs.h"
#include "traps.h"

#define IOAPIC  0xFEC00000   // Default physical address of IO APIC

#define REG_ID     0x00  // Register index: ID
#define REG_VER    0x01  // Register index: version
#define REG_TABLE  0x10  // Redirection table base

// The redirection table starts at REG_TABLE and uses
// two registers to configure each interrupt.
// The first (low) register in a pair contains configuration bits.
// The second (high) register contains a bitmask telling which
// CPUs can serve that interrupt.
#define INT_DISABLED   0x00010000  // Interrupt disabled
#define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
#define INT_ACTIVELOW  0x00002000  // Active low (vs high)
#define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)

volatile struct ioapic *ioapic;

// IO APIC MMIO structure: write reg, then read or write data.
struct ioapic {
  uint reg;
  uint pad[3];
  uint data;
};

static uint
ioapicread(int reg)
{
  ioapic->reg = reg;
  return ioapic->data;
}

static void
ioapicwrite(int reg, uint data)
{
  ioapic->reg = reg;
  ioapic->data = data;
}
```

```
7950 void
7951 ioapicinit(void)
7952 {
7953   int i, id, maxintr;
7954
7955   if(!ismp)
7956     return;
7957
7958   ioapic = (volatile struct ioapic*)IOAPIC;
7959   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
7960   id = ioapicread(REG_ID) >> 24;
7961   if(id != ioapicid)
7962     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
7963
7964   // Mark all interrupts edge-triggered, active high, disabled,
7965   // and not routed to any CPUs.
7966   for(i = 0; i <= maxintr; i++){
7967     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
7968     ioapicwrite(REG_TABLE+2*i+1, 0);
7969   }
7970 }
7971
7972 void
7973 ioapicenable(int irq, int cpunum)
7974 {
7975   if(!ismp)
7976     return;
7977
7978   // Mark interrupt edge-triggered, active high,
7979   // enabled, and routed to the given cpunum,
7980   // which happens to be that cpu's APIC ID.
7981   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
7982   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
7983 }
7984
7985
7986
7987
7988
7989
7990
7991
7992
7993
7994
7995
7996
7997
7998
7999
```



```
8000 // Intel 8259A programmable interrupt controllers.
8001
8002 #include "types.h"
8003 #include "x86.h"
8004 #include "traps.h"
8005
8006 // I/O Addresses of the two programmable interrupt controllers
8007 #define IO_PIC1         0x20    // Master (IRQs 0-7)
8008 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
8009
8010 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
8011
8012 // Current IRQ mask.
8013 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
8014 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
8015
8016 static void
8017 picsetmask(ushort mask)
8018 {
8019   irqmask = mask;
8020   outb(IO_PIC1+1, mask);
8021   outb(IO_PIC2+1, mask >> 8);
8022 }
8023
8024 void
8025 picenable(int irq)
8026 {
8027   picsetmask(irqmask & ~(1<<irq));
8028 }
8029
8030 // Initialize the 8259A interrupt controllers.
8031 void
8032 picinit(void)
8033 {
8034   // mask all interrupts
8035   outb(IO_PIC1+1, 0xFF);
8036   outb(IO_PIC2+1, 0xFF);
8037
8038   // Set up master (8259A-1)
8039
8040   // ICW1:  0001g0hi
8041   //    g:  0 = edge triggering, 1 = level triggering
8042   //    h:  0 = cascaded PICs, 1 = master only
8043   //    i:  0 = no ICW4, 1 = ICW4 required
8044   outb(IO_PIC1, 0x11);
8045
8046   // ICW2:  Vector offset
8047   outb(IO_PIC1+1, T_IRQ0);
8048
8049
```

```
8050   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
8051   //        (slave PIC) 3-bit # of slave's connection to master
8052   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
8053
8054   // ICW4:  000nbmap
8055   //    n:  1 = special fully nested mode
8056   //    b:  1 = buffered mode
8057   //    m:  0 = slave PIC, 1 = master PIC
8058   //      (ignored when b is 0, as the master/slave role
8059   //      can be hardwired).
8060   //    a:  1 = Automatic EOI mode
8061   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
8062   outb(IO_PIC1+1, 0x3);
8063
8064   // Set up slave (8259A-2)
8065   outb(IO_PIC2, 0x11);                  // ICW1
8066   outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
8067   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
8068   // NB Automatic EOI mode doesn't tend to work on the slave.
8069   // Linux source code says it's "to be investigated".
8070   outb(IO_PIC2+1, 0x3);                 // ICW4
8071
8072   // OCW3:  0ef01prs
8073   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
8074   //    p:  0 = no polling, 1 = polling mode
8075   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
8076   outb(IO_PIC1, 0x68);             // clear specific mask
8077   outb(IO_PIC1, 0x0a);             // read IRR by default
8078
8079   outb(IO_PIC2, 0x68);             // OCW3
8080   outb(IO_PIC2, 0x0a);             // OCW3
8081
8082   if(irqmask != 0xFFFF)
8083     picsetmask(irqmask);
8084 }
8085
8086
8087
8088
8089
8090
8091
8092
8093
8094
8095
8096
8097
8098
8099
```



```
8100 // Blank page.
8101
8102
8103
8104
8105
8106
8107
8108
8109
8110
8111
8112
8113
8114
8115
8116
8117
8118
8119
8120
8121
8122
8123
8124
8125
8126
8127
8128
8129
8130
8131
8132
8133
8134
8135
8136
8137
8138
8139
8140
8141
8142
8143
8144
8145
8146
8147
8148
8149
```

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



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

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



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

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



```
8400
8401
8402
8403
8404
8405
8406
8407
8408
8409
8410
8411
8412
8413
8414
8415
8416
8417
8418
8419
8420
8421
8422
8423
8424
8425
8426
8427
8428
8429
8430
8431
8432
8433
8434
8435
8436
8437
8438
8439
8440
8441
8442
8443
8444
8445
8446
8447
8448
8449
```

```
8450 // Print to the console. only understands %d, %x, %p, %s.
8451 void
8452 cprintf(char *fmt, ...)
8453 {
8454   int i, c, locking;
8455   uint *argp;
8456   char *s;
8457
8458   locking = cons.locking;
8459   if(locking)
8460     acquire(&cons.lock);
8461
8462   if (fmt == 0)
8463     panic("null fmt");
8464
8465   argp = (uint*)(void*)(&fmt + 1);
8466   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
8467     if(c != '%'){
8468       consputc(c);
8469       continue;
8470     }
8471     c = fmt[++i] & 0xff;
8472     if(c == 0)
8473       break;
8474     switch(c){
8475     case 'd':
8476       printint(*argp++, 10, 1);
8477       break;
8478     case 'x':
8479     case 'p':
8480       printint(*argp++, 16, 0);
8481       break;
8482     case 's':
8483       if((s = (char*)*argp++) == 0)
8484         s = "(null)";
8485       for(; *s; s++)
8486         consputc(*s);
8487       break;
8488     case '%':
8489       consputc('%');
8490       break;
8491     default:
8492       // Print unknown % sequence to draw attention.
8493       consputc('%');
8494       consputc(c);
8495       break;
8496     }
8497   }
8498
8499
```



```
8500   if(locking)
8501     release(&cons.lock);
8502 }
8503
8504 void
8505 panic(char *s)
8506 {
8507   int i;
8508   uint pcs[10];
8509
8510   cli();
8511   cons.locking = 0;
8512   cprintf("cpu with apicid %d: panic: ", cpu->apicid);
8513   cprintf(s);
8514   cprintf("\n");
8515   getcallerpcs(&s, pcs);
8516   for(i=0; i<10; i++)
8517     cprintf(" %p", pcs[i]);
8518   panicked = 1; // freeze other CPU
8519   for(;;)
8520     ;
8521 }
8522
8523
8524
8525
8526
8527
8528
8529
8530
8531
8532
8533
8534
8535
8536
8537
8538
8539
8540
8541
8542
8543
8544
8545
8546
8547
8548
8549
```

```
#define BACKSPACE 0x100
#define CRTPORT 0x3d4
static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory

static void
cgaputc(int c)
{
  int pos;

  // Cursor position: col + 80*row.
  outb(CRTPORT, 14);
  pos = inb(CRTPORT+1) << 8;
  outb(CRTPORT, 15);
  pos |= inb(CRTPORT+1);

  if(c == '\n')
    pos += 80 - pos%80;
  else if(c == BACKSPACE){
    if(pos > 0) --pos;
  } else
    crt[pos++] = (c&0xff) | 0x0700;  // black on white

  if(pos < 0 || pos > 25*80)
    panic("pos under/overflow");

  if((pos/80) >= 24){  // Scroll up.
    memmove(crt, crt+80, sizeof(crt[0])*23*80);
    pos -= 80;
    memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
  }

  outb(CRTPORT, 14);
  outb(CRTPORT+1, pos>>8);
  outb(CRTPORT, 15);
  outb(CRTPORT+1, pos);
  crt[pos] = ' ' | 0x0700;
}
```



```
void
consputc(int c)
{
  if(panicked){
    cli();
    for(;;)
      ;
  }

  if(c == BACKSPACE){
    uartputc('\b'); uartputc(' '); uartputc('\b');
  } else
    uartputc(c);
  cgaputc(c);
}

#define INPUT_BUF 128
struct {
  char buf[INPUT_BUF];
  uint r;  // Read index
  uint w;  // Write index
  uint e;  // Edit index
} input;

#define C(x)  ((x)-'@')  // Control-x

void
consoleintr(int (*getc)(void))
{
  int c, doprocdump = 0;

  acquire(&cons.lock);
  while((c = getc()) >= 0){
    switch(c){
    case C('P'):  // Process listing.
      // procdump() locks cons.lock indirectly; invoke later
      doprocdump = 1;
      break;
    case C('U'):  // Kill line.
      while(input.e != input.w &&
            input.buf[(input.e-1) % INPUT_BUF] != '\n'){
        input.e--;
        consputc(BACKSPACE);
      }
      break;
    case C('H'): case '\x7f':  // Backspace
      if(input.e != input.w){
        input.e--;
        consputc(BACKSPACE);
      }
```

```
8650       break;
8651     default:
8652       if(c != 0 && input.e-input.r < INPUT_BUF){
8653         c = (c == '\r') ? '\n' : c;
8654         input.buf[input.e++ % INPUT_BUF] = c;
8655         consputc(c);
8656         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
8657           input.w = input.e;
8658           wakeup(&input.r);
8659         }
8660       }
8661       break;
8662     }
8663   }
8664   release(&cons.lock);
8665   if(doprocdump) {
8666     procdump();  // now call procdump() wo. cons.lock held
8667   }
8668 }
8669
8670 int
8671 consoleread(struct inode *ip, char *dst, int n)
8672 {
8673   uint target;
8674   int c;
8675
8676   iunlock(ip);
8677   target = n;
8678   acquire(&cons.lock);
8679   while(n > 0){
8680     while(input.r == input.w){
8681       if(proc->killed){
8682         release(&cons.lock);
8683         ilock(ip);
8684         return -1;
8685       }
8686       sleep(&input.r, &cons.lock);
8687     }
8688     c = input.buf[input.r++ % INPUT_BUF];
8689     if(c == C('D')){  // EOF
8690       if(n < target){
8691         // Save ^D for next time, to make sure
8692         // caller gets a 0-byte result.
8693         input.r--;
8694       }
8695       break;
8696     }
8697     *dst++ = c;
8698     --n;
8699     if(c == '\n')
```



```
8700       break;
8701   }
8702   release(&cons.lock);
8703   ilock(ip);
8704
8705   return target - n;
8706 }
8707
8708 int
8709 consolewrite(struct inode *ip, char *buf, int n)
8710 {
8711   int i;
8712
8713   iunlock(ip);
8714   acquire(&cons.lock);
8715   for(i = 0; i < n; i++)
8716     consputc(buf[i] & 0xff);
8717   release(&cons.lock);
8718   ilock(ip);
8719
8720   return n;
8721 }
8722
8723 void
8724 consoleinit(void)
8725 {
8726   initlock(&cons.lock, "console");
8727
8728   devsw[CONSOLE].write = consolewrite;
8729   devsw[CONSOLE].read = consoleread;
8730   cons.locking = 1;
8731
8732   picenable(IRQ_KBD);
8733   ioapicenable(IRQ_KBD, 0);
8734 }
8735
8736
8737
8738
8739
8740
8741
8742
8743
8744
8745
8746
8747
8748
8749
```

```
8750 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
8751 // Only used on uniprocessors;
8752 // SMP machines use the local APIC timer.
8753
8754 #include "types.h"
8755 #include "defs.h"
8756 #include "traps.h"
8757 #include "x86.h"
8758
8759 #define IO_TIMER1       0x040           // 8253 Timer #1
8760
8761 // Frequency of all three count-down timers;
8762 // (TIMER_FREQ/freq) is the appropriate count
8763 // to generate a frequency of freq Hz.
8764
8765 #define TIMER_FREQ      1193182
8766 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
8767
8768 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
8769 #define TIMER_SEL0      0x00    // select counter 0
8770 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
8771 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
8772
8773 void
8774 timerinit(void)
8775 {
8776   // Interrupt 100 times/sec.
8777   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
8778   outb(IO_TIMER1, TIMER_DIV(100) % 256);
8779   outb(IO_TIMER1, TIMER_DIV(100) / 256);
8780   picenable(IRQ_TIMER);
8781 }
8782
8783
8784
8785
8786
8787
8788
8789
8790
8791
8792
8793
8794
8795
8796
8797
8798
8799
```



```
8800 // Intel 8250 serial port (UART).
8801
8802 #include "types.h"
8803 #include "defs.h"
8804 #include "param.h"
8805 #include "traps.h"
8806 #include "spinlock.h"
8807 #include "sleeplock.h"
8808 #include "fs.h"
8809 #include "file.h"
8810 #include "mmu.h"
8811 #include "proc.h"
8812 #include "x86.h"
8813
8814 #define COM1    0x3f8
8815
8816 static int uart;    // is there a uart?
8817
8818 void
8819 uartinit(void)
8820 {
8821   char *p;
8822
8823   // Turn off the FIFO
8824   outb(COM1+2, 0);
8825
8826   // 9600 baud, 8 data bits, 1 stop bit, parity off.
8827   outb(COM1+3, 0x80);    // Unlock divisor
8828   outb(COM1+0, 115200/9600);
8829   outb(COM1+1, 0);
8830   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
8831   outb(COM1+4, 0);
8832   outb(COM1+1, 0x01);    // Enable receive interrupts.
8833
8834   // If status is 0xFF, no serial port.
8835   if(inb(COM1+5) == 0xFF)
8836     return;
8837   uart = 1;
8838
8839   // Acknowledge pre-existing interrupt conditions;
8840   // enable interrupts.
8841   inb(COM1+2);
8842   inb(COM1+0);
8843   picenable(IRQ_COM1);
8844   ioapicenable(IRQ_COM1, 0);
8845
8846   // Announce that we're here.
8847   for(p="xv6...\n"; *p; p++)
8848     uartputc(*p);
8849 }
```

```
8850 void
8851 uartputc(int c)
8852 {
8853   int i;
8854
8855   if(!uart)
8856     return;
8857   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
8858     microdelay(10);
8859   outb(COM1+0, c);
8860 }
8861
8862 static int
8863 uartgetc(void)
8864 {
8865   if(!uart)
8866     return -1;
8867   if(!(inb(COM1+5) & 0x01))
8868     return -1;
8869   return inb(COM1+0);
8870 }
8871
8872 void
8873 uartintr(void)
8874 {
8875   consoleintr(uartgetc);
8876 }
8877
8878
8879
8880
8881
8882
8883
8884
8885
8886
8887
8888
8889
8890
8891
8892
8893
8894
8895
8896
8897
8898
8899
```



```
8900 # Initial process execs /init.
8901 # This code runs in user space.
8902
8903 #include "syscall.h"
8904 #include "traps.h"
8905
8906
8907 # exec(init, argv)
8908 .globl start
8909 start:
8910   pushl $argv
8911   pushl $init
8912   pushl $0  // where caller pc would be
8913   movl $SYS_exec, %eax
8914   int $T_SYSCALL
8915
8916 # for(;;) exit();
8917 exit:
8918   movl $SYS_exit, %eax
8919   int $T_SYSCALL
8920   jmp exit
8921
8922 # char init[] = "/bin/init\0";
8923 init:
8924   .string "/bin/init\0"
8925
8926 # char *argv[] = { init, 0 };
8927 .p2align 2
8928 argv:
8929   .long init
8930   .long 0
8931
8932
8933
8934
8935
8936
8937
8938
8939
8940
8941
8942
8943
8944
8945
8946
8947
8948
8949
```

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(priority)
SYSCALL(policy)
```



```
// init: The initial user-level program

#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

char *argv[] = { "sh", 0 };

int
main(void)
{
  int pid, wpid;

  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr

  for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit(0);
    }
    if(pid == 0){
      exec("/bin/sh", argv);
      printf(1, "init: exec sh failed\n");
      exit(0);
    }
    while((wpid=wait(0)) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

```
9050 // Shell.
9051
9052 #include "types.h"
9053 #include "user.h"
9054 #include "fcntl.h"
9055
9056
9057
9058
9059 // Parsed command representation
9060 #define EXEC  1
9061 #define REDIR 2
9062 #define PIPE  3
9063 #define LIST  4
9064 #define BACK  5
9065
9066 #define MAXARGS 10
9067
9068 struct cmd {
9069   int type;
9070 };
9071
9072 struct execcmd {
9073   int type;
9074   char *argv[MAXARGS];
9075   char *eargv[MAXARGS];
9076 };
9077
9078 struct redircmd {
9079   int type;
9080   struct cmd *cmd;
9081   char *file;
9082   char *efile;
9083   int mode;
9084   int fd;
9085 };
9086
9087 struct pipecmd {
9088   int type;
9089   struct cmd *left;
9090   struct cmd *right;
9091 };
9092
9093 struct listcmd {
9094   int type;
9095   struct cmd *left;
9096   struct cmd *right;
9097 };
9098
9099
```



```
9100 struct backcmd {
9101   int type;
9102   struct cmd *cmd;
9103 };
9104
9105 int fork1(void);  // Fork but panics on failure.
9106 void panic(char*);
9107 struct cmd *parsecmd(char*);
9108
9109
9110 //initilaize a string with 0;
9111 void strClear(char s[],int len){
9112   int i=0;
9113   if(len){
9114     while(i<len){
9115       s[i]=0;
9116       i++;
9117     }
9118   }
9119 }
9120
9121
9122
9123 //check and update cmd path with the global environment path
9124 void checkPath(struct execcmd *execCmd){
9125   int fd=open(execCmd->argv[0],O_RDWR);
9126   if(fd>0){
9127     return;
9128   }
9129
9130   fd=open("/path",O_RDWR);
9131   char tempPath[50];
9132   while(1){
9133     int firstLetter=1;
9134     int ind=0;
9135     strClear(tempPath,50);
9136     int stat=read(fd,tempPath,1);
9137     //printf(2,"stat is : %d\n",stat);
9138     //printf(2,"the first LEttet is %d\n",tempPath[0]);
9139     if(stat<=0||tempPath[0]=='\n'){
9140      // printf(2,"%s\n","end of file");
9141       break;
9142     }
9143     //printf(2,"the path is %s\n",tempPath);
9144
9145     while(1){
9146       if(firstLetter==1) { //dosen't need to read again'
9147         firstLetter=0;
9148
9149
```

```
9150         }
9151       else{
9152         read(fd,tempPath+ind,1);
9153       }
9154
9155       if(tempPath[ind]==':'){
9156         //printf(2,"read : %s\n",tempPath);
9157         break;
9158       }
9159
9160       else
9161         ind++;
9162     }//end of while
9163
9164     strcpy(tempPath+ind,execCmd->argv[0]);
9165     //printf(2,"the path is %s\n",tempPath);
9166     int tempfd=open(tempPath,O_RDONLY);
9167     if(tempfd>0){
9168       strcpy(execCmd->argv[0],tempPath);
9169       close(fd);
9170       close(tempfd);
9171       return;
9172     }
9173   }//end of while
9174   close(fd);
9175
9176 }
9177
9178
9179
9180 // Execute cmd.  Never returns.
9181 void
9182 runcmd(struct cmd *cmd)
9183 {
9184   int p[2];
9185   struct backcmd *bcmd;
9186   struct execcmd *ecmd;
9187   struct listcmd *lcmd;
9188   struct pipecmd *pcmd;
9189   struct redircmd *rcmd;
9190
9191   if(cmd == 0)
9192     exit(0);
9193
9194   switch(cmd->type){
9195   default:
9196     panic("runcmd");
9197
9198
9199
```



```
9200   case EXEC:
9201     ecmd = (struct execcmd*)cmd;
9202     if(ecmd->argv[0] == 0)
9203       exit(0);
9204     checkPath(ecmd);
9205     //printf(2,"%s",ecmd->argv[0]);
9206     exec(ecmd->argv[0], ecmd->argv);
9207     printf(2, "exec %s failed\n", ecmd->argv[0]);
9208     break;
9209
9210   case REDIR:
9211     rcmd = (struct redircmd*)cmd;
9212     close(rcmd->fd);
9213     if(open(rcmd->file, rcmd->mode) < 0){
9214       printf(2, "open %s failed\n", rcmd->file);
9215       exit(0);
9216     }
9217     runcmd(rcmd->cmd);
9218     break;
9219
9220   case LIST:
9221     lcmd = (struct listcmd*)cmd;
9222     if(fork1() == 0)
9223       runcmd(lcmd->left);
9224     wait(0);
9225     runcmd(lcmd->right);
9226     break;
9227
9228   case PIPE:
9229     pcmd = (struct pipecmd*)cmd;
9230     if(pipe(p) < 0)
9231       panic("pipe");
9232     if(fork1() == 0){
9233       close(1);
9234       dup(p[1]);
9235       close(p[0]);
9236       close(p[1]);
9237       runcmd(pcmd->left);
9238     }
9239     if(fork1() == 0){
9240       close(0);
9241       dup(p[0]);
9242       close(p[0]);
9243       close(p[1]);
9244       runcmd(pcmd->right);
9245     }
9246     close(p[0]);
9247     close(p[1]);
9248     wait(0);
9249     wait(0);
```

```
9250     break;
9251
9252   case BACK:
9253     bcmd = (struct backcmd*)cmd;
9254     if(fork1() == 0)
9255       runcmd(bcmd->cmd);
9256     break;
9257   }
9258   exit(0);
9259 }
9260
9261 int
9262 getcmd(char *buf, int nbuf)
9263 {
9264   printf(2, "$ ");
9265   memset(buf, 0, nbuf);
9266   gets(buf, nbuf);
9267   if(buf[0] == 0) // EOF
9268     return -1;
9269   return 0;
9270 }
9271
9272 int
9273 main(void)
9274 {
9275   static char buf[100];
9276   int fd;
9277
9278   // Ensure that three file descriptors are open.
9279   while((fd = open("console", O_RDWR)) >= 0){
9280     if(fd >= 3){
9281       close(fd);
9282       break;
9283     }
9284   }
9285
9286   // Read and run input commands.
9287   while(getcmd(buf, sizeof(buf)) >= 0){
9288     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
9289       // Chdir must be called by the parent, not the child.
9290       buf[strlen(buf)-1] = 0;  // chop \n
9291       if(chdir(buf+3) < 0)
9292         printf(2, "cannot cd %s\n", buf+3);
9293       continue;
9294     }
9295     if(fork1() == 0)
9296       runcmd(parsecmd(buf));
9297     wait(0);
9298   }
9299   exit(0);
```



```
9300 }
9301
9302 void
9303 panic(char *s)
9304 {
9305   printf(2, "%s\n", s);
9306   exit(0);
9307 }
9308
9309 int
9310 fork1(void)
9311 {
9312   int pid;
9313
9314   pid = fork();
9315   if(pid == -1)
9316     panic("fork");
9317   return pid;
9318 }
9319
9320
9321
9322
9323
9324
9325
9326
9327
9328
9329
9330
9331
9332
9333
9334
9335
9336
9337
9338
9339
9340
9341
9342
9343
9344
9345
9346
9347
9348
9349
```

```
// Constructors

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

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

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

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

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

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



```
struct cmd*
listcmd(struct cmd *left, struct cmd *right)
{
  struct listcmd *cmd;

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

struct cmd*
backcmd(struct cmd *subcmd)
{
  struct backcmd *cmd;

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

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



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

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



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

```
// NUL-terminate all the counted strings.
struct cmd*
nulterminate(struct cmd *cmd)
{
  int i;
  struct backcmd *bcmd;
  struct execcmd *ecmd;
  struct listcmd *lcmd;
  struct pipecmd *pcmd;
  struct redircmd *rcmd;

  if(cmd == 0)
    return 0;

  switch(cmd->type){
  case EXEC:
    ecmd = (struct execcmd*)cmd;
    for(i=0; ecmd->argv[i]; i++)
      *ecmd->eargv[i] = 0;
    break;

  case REDIR:
    rcmd = (struct redircmd*)cmd;
    nulterminate(rcmd->cmd);
    *rcmd->efile = 0;
    break;

  case PIPE:
    pcmd = (struct pipecmd*)cmd;
    nulterminate(pcmd->left);
    nulterminate(pcmd->right);
    break;

  case LIST:
    lcmd = (struct listcmd*)cmd;
    nulterminate(lcmd->left);
    nulterminate(lcmd->right);
    break;

  case BACK:
    bcmd = (struct backcmd*)cmd;
    nulterminate(bcmd->cmd);
    break;
  }
  return cmd;
}
```



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

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

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

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

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

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

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

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

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

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



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

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