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

ACKNOWLEDGMENTS

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

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

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

The code in the files that constitute xv6 is
Copyright 2006-2007 Frans Kaashoek, Robert Morris, and Russ Cox.

ERROR REPORTS

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

BUILDING AND RUNNING XV6

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

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

To create a typeset version of the code, run "make xv6.pdf".
This requires the "mpage" text formatting 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.

basic headers
01 types.h
01 param.h
02 defs.h
04 x86.h
06 asm.h
06 mmu.h
08 elf.h

startup
09 bootasm.S
10 bootother.S
11 bootmain.c
12 main.c

locks
13 spinlock.h
13 spinlock.c

processes
15 proc.h
16 proc.c
22 swtch.S
22 kalloc.c

system calls
24 traps.h
24 vectors.pl
25 trapasm.S
25 trap.c
27 syscall.h
27 syscall.c
29 sysproc.c

file system
30 buf.h
30 fcntl.h
31 stat.h
31 fs.h
32 file.h
33 ide.c
35 bio.c
36 fs.c
44 file.c
45 sysfile.c
50 exec.c

pipes
51 pipe.c

string operations
53 string.c

low-level hardware
54 mp.h
55 mp.c
56 lapic.c
58 ioapic.c
59 picirq.c
61 kbd.h
62 kbd.c
63 console.c
66 timer.c

user-level
67 initcode.S
67 usys.S
68 init.c
68 sh.c

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 2256
        0311 1928 1962 2255
        2256
```

indicates that swtch is defined on line 2256 and is mentioned on five lines
on sheets 03, 19, and 22.

acquire 1373
0321 1373 1377 1759
1917 1975 2018 2033
2066 2079 2123 2158
2315 2362 2616 2971
3407 3465 3570 3629
3857 3890 3910 3939
3954 3964 4424 4440
4456 5213 5234 5255
6360 6516 6558 6606
allocproc 1754
1754 1807 1860
alltraps 2506
2459 2467 2480 2485
2505 2506
ALT 6110
6110 6138 6140
argfd 4563
4563 4606 4621 4633
4644 4656
argint 2794
0339 2794 2808 2824
2931 2956 2969 4568
4621 4633 4858 4921
4922 4957
argptr 2804
0340 2804 4621 4633
4656 4982
argstr 2821
0341 2821 4668 4758
4858 4906 4920 4935
4957
BACK 6861
6861 6974 7120 7389
backcmd 6896 7114
6896 6909 6975 7114
7116 7242 7355 7390
BACKSPACE 6450
6450 6467 6526 6532
balloc 3704
3704 3725 4017 4025
4029
BBLOCK 3191
3191 3713 3739
B_BUSY 3009
3009 3458 3576 3577
3588 3591 3616 3626
3638
B_DIRTY 3011
3011 3387 3416 3421
3460 3479 3618
bfree 3730
3730 4062 4072 4075
bget 3566
3566 3596 3606
binit 3539
0210 1229 3539
bmap 4010
4010 4036 4119 4169
4222
bootmain 1116
0976 1116
bootothers 1267
1210 1236 1267
BPB 3188
3188 3191 3712 3714
3740
bread 3602
0211 3602 3682 3693
3713 3739 3811 3832
3917 4026 4068 4119
4169 4222
brelse 3624
0212 3624 3627 3684
3696 3719 3723 3746
3817 3820 3841 3925
4032 4074 4122 4173
4233 4237
BSIZE 3158
3158 3168 3182 3188
3694 4119 4120 4121
4165 4166 4169 4170
4171 4221 4222 4224
buf 3000
0200 0211 0212 0213
0253 3000 3004 3005
3006 3310 3325 3328
3375 3404 3454 3456
3459 3527 3531 3535
3541 3553 3565 3568
3601 3604 3614 3624
3669 3680 3691 3707
3732 3805 3829 3904
4013 4057 4105 4155
4215 6327 6338 6341
6344 6503 6524 6537
6568 6601 6608 6984
6987 6988 6989 7003
7015 7016 7019 7020
7021 7025

B_VALID 3010
3010 3420 3460 3479
3607
bwrite 3614
0213 3614 3617 3695
3718 3745 3816 3840
4030 4172
bzero 3689
3689 3736
C 6131 6509
6131 6179 6204 6205
6206 6207 6208 6210
6509 6519 6522 6529
6539 6569
CAPSLOCK 6112
6112 6145 6286
cgaputc 6455
6455 6496
cli 0521
0521 0523 0915 1029
1460 6406 6490
cmd 6865
6865 6877 6886 6887
6892 6893 6898 6902
6906 6915 6918 6923
6931 6937 6941 6951
6975 6977 7052 7055
7057 7058 7059 7060
7063 7064 7066 7068
7069 7070 7071 7072
7073 7074 7075 7076
7079 7080 7082 7084
7085 7086 7087 7088
7089 7100 7101 7103
7105 7106 7107 7108
7109 7110 7113 7114
7116 7118 7119 7120
7121 7122 7212 7213
7214 7215 7217 7221
7224 7230 7231 7234
7237 7239 7242 7246
7248 7250 7253 7255
7258 7260 7263 7264
7275 7278 7281 7285
7300 7303 7308 7312
7313 7316 7321 7322
7328 7337 7338 7344
7345 7351 7352 7361
7364 7366 7372 7373
7378 7384 7390 7391
7394
CONSOLE 3290
3290 6621 6622
consoleinit 6616
0216 1222 6616
consoleintr 6512
0218 6298 6512
consoleread 6551
6551 6622
consolewrite 6601
6601 6621
consputc 6487
6314 6344 6366 6384
6387 6391 6392 6487
6526 6532 6538 6608
context 1518
0201 0318 1518 1537
1559 1678 1787 1788
1789 1790 1928 1967
cprintf 6352
0217 1224 1258 1262
1676 1680 1682 2286
2375 2637 2645 2650
2882 3410 5637 5761
5912 6352 6408 6409
6410 6413
cpu 1557 5751
0256 0269 1207 1224
1255 1256 1258 1262
1271 1279 1306 1365
1389 1408 1446 1557
1568 1577 1705 1707
2615 2637 2638 2645
2646 2650 2651 5512
5513 5751 5761 6408
CR0_PE 0910 1024
0956 1056
create 4801
4801 4821 4834 4838
4862 4906 4923
CRTPORT 6451
6451 6460 6461 6462
6463 6479 6480 6481
6482
CTL 6109
6109 6135 6139 6285
devsw 3283
3283 3288 4108 4110
4158 4160 4406 6621
6622

dinode 3172
3172 3182 3806 3812
3830 3833 3905 3918
dirent 3203
3203 4216 4223 4224
4255 4705 4754
dirlink 4252
0234 4252 4267 4275
4684 4833 4837 4838
dirlookup 4212
0235 4212 4219 4259
4374 4770 4811
DIRSIZ 3201
3201 3205 4205 4272
4328 4329 4391 4665
4755 4805
DPL_USER 0711
0711 1725 1726 1817
1818 2572 2658 2667
EOESC 6116
6116 6270 6274 6275
6277 6280
elfhdr 0855
0855 1118 1123 5014
ELF_MAGIC 0852
0852 1129 5028
ELF_PROG_LOAD 0886
0886 5036 5067
EOI 5663
5663 5734 5775
ERROR 5681
5681 5727
ESR 5666
5666 5730 5731
exec 5009
0222 4972 5009 6768
6829 6830 6926 6927
EXEC 6857
6857 6922 7059 7365
execcmd 6869 7053
6869 6910 6923 7053
7055 7321 7327 7328
7356 7366
exit 2104
0302 2104 2140 2605
2609 2659 2668 2916
6715 6718 6761 6826
6831 6916 6925 6935
6980 7028 7035
fdalloc 4582
4582 4608 4874 4987
fetchint 2766
0342 2766 2796 4963
fetchstr 2778
0343 2778 2826 4969
file 3250
0202 0225 0226 0227
0229 0230 0231 0287
1540 3250 3671 4403
4409 4419 4422 4425
4437 4438 4452 4454
4476 4502 4522 4557
4563 4566 4582 4603
4617 4629 4642 4653
4855 4979 5155 5170
6309 6878 6933 6934
7064 7072 7272
filealloc 4420
0225 4420 4874 5176
fileclose 4452
0226 2115 4452 4458
4647 4876 4990 4991
5204 5206
filedup 4438
0227 1880 4438 4442
4610
fileinit 4413
0228 1230 4413
fileread 4502
0229 4502 4517 4623
filestat 4476
0230 4476 4658
filewrite 4522
0231 4522 4537 4635
FL_IF 0660
0660 1462 1468 1821
1963 5758
fork 1854
0303 1854 2910 6760
6823 6825 7043 7045
fork1 7039
6900 6942 6954 6961
6976 7024 7039
forkret 1984
1616 1790 1984
gatedesc 0801
0464 0467 0801 2560
getcallerpcs 1426
0322 1390 1426 1678
6411

getcmd 6984
6984 7015
gettoken 7156
7156 7241 7245 7257
7270 7271 7307 7311
7333
growproc 1834
0304 1834 2959
havedisk1 3327
3327 3364 3462
holding 1444
0323 1376 1404 1444
1957
ialloc 3802
0236 3802 3822 4820
4821
IBLOCK 3185
3185 3811 3832 3917
I_BUSY 3277
3277 3911 3913 3936
3940 3957 3959
ICRHI 5674
5674 5737 5821 5833
ICRLO 5667
5667 5738 5739 5822
5824 5834
ID 5660
5660 5693 5766
IDE_BSY 3312
3312 3336
IDE_CMD_READ 3317
3317 3391
IDE_CMD_WRITE 3318
3318 3388
IDE_DF 3314
3314 3338
IDE_DRDY 3313
3313 3336
IDE_ERR 3315
3315 3338
ideinit 3351
0251 1232 3351
ideintr 3402
0252 2624 3402
idelock 3324
3324 3355 3407 3409
3428 3465 3480 3482
iderw 3454
0253 3454 3459 3461
3608 3619
idestart 3375
3328 3375 3378 3426
3475
idewait 3332
3332 3358 3380 3416
idtinit 2578
0351 1259 2578
idup 3888
0237 1881 3888 4361
iget 3853
3794 3818 3853 3873
4234 4359
iinit 3789
0238 1231 3789
ilock 3902
0239 3902 3908 3928
4364 4479 4511 4531
4672 4683 4693 4762
4774 4809 4813 4823
4867 4937 5023 6563
6583 6610
inb 0403
0403 0928 0936 1154
3336 3363 5646 6264
6267 6461 6463
initlock 1361
0324 1361 1622 2283
2574 3355 3543 3791
4415 5184 6618 6619
inode 3263
0203 0234 0235 0236
0237 0239 0240 0241
0242 0243 0245 0246
0247 0248 0249 1541
3256 3263 3284 3285
3674 3785 3794 3801
3827 3852 3855 3861
3887 3888 3902 3934
3952 3974 4010 4054
4085 4102 4152 4211
4212 4252 4256 4353
4356 4388 4395 4666
4702 4753 4800 4804
4856 4904 4915 4933
5015 6551 6601
INPUT_BUF 6500
6500 6503 6524 6536
6537 6539 6568
insl 0412
0412 0414 1173 3417

INT_DISABLED 5869
5869 5917
ioapic 5877
5607 5627 5628 5874
5877 5886 5887 5893
5894 5908
IOAPIC 5858
5858 5908
ioapicenable 5923
0256 3357 5923 6626
ioapicid 5516
0257 5516 5628 5911
5912
ioapicinit 5901
0258 1221 5901 5912
ioapicread 5884
5884 5909 5910
ioapicwrite 5891
5891 5917 5918 5931
5932
IO_PIC1 5957
5957 5970 5985 5994
5997 6002 6012 6026
6027
IO_PIC2 5958
5958 5971 5986 6015
6016 6017 6020 6029
6030
IO_RTC 5800
5800 5813 5814
IO_TIMER1 6659
6659 6668 6678 6679
IPB 3182
3182 3185 3191 3812
3833 3918
iput 3952
0240 2120 3952 3958
3977 4260 4382 4471
4689 4943
IRQ_COM1 2433
2433 2631
IRQ_ERROR 2435
2435 5727
IRQ_IDE 2434
2434 2623 3356 3357
IRQ_KBD 2432
2432 2627 6625 6626
IRQ_SLAVE 5960
5960 5964 6002 6017
IRQ_SPURIOUS 2436
2436 2636 5707
IRQ_TIMER 2431
2431 2614 2663 5714
6680
isdirempty 4702
4702 4709 4778
ismp 5514
0277 1233 5514 5613
5905 5925
itrunc 4054
3674 3961 4054
iunlock 3934
0241 3934 3937 3976
4371 4481 4514 4534
4679 4880 4942 6556
6605
iunlockput 3974
0242 3974 4366 4375
4378 4674 4685 4688
4696 4766 4771 4779
4780 4791 4795 4812
4816 4840 4869 4877
4908 4925 4939 5077
5118
iupdate 3827
0243 3827 3963 4080
4178 4678 4695 4789
4794 4827 4831
I_VALID 3278
3278 3916 3926 3955
kalloc 2354
0261 1283 1772 1812
1838 1865 2354 2360
2375 5058 5178
KBDATAP 6104
6104 6267
kbdgetc 6256
6256 6298
kbdintr 6296
0266 2628 6296
KBS_DIB 6103
6103 6265
KBSTATP 6102
6102 6264
KEY_DEL 6128
6128 6169 6191 6215
KEY_DN 6122
6122 6165 6187 6211
KEY_END 6120
6120 6168 6190 6214

KEY_HOME 6119
6119 6168 6190 6214
KEY_INS 6127
6127 6169 6191 6215
KEY_LF 6123
6123 6167 6189 6213
KEY_PGDN 6126
6126 6166 6188 6212
KEY_PGUP 6125
6125 6166 6188 6212
KEY_RT 6124
6124 6167 6189 6213
KEY_UP 6121
6121 6165 6187 6211
kfree 2305
0262 1843 1866 2169
2170 2287 2305 2310
5107 5117 5202 5223
kill 2075
0305 2075 2650 2933
6767
kinit 2277
0263 1226 2277
ksegment 1703
0309 1219 1257 1703
KSTACKSIZE 0152
0152 1283 1284 1730
1772 1776 1866 2170
lapiceoi 5772
0271 2621 2625 2629
2633 2639 5772
lapicinit 5701
0272 1218 1256 5701
lapicstartap 5805
0273 1286 5805
lapicw 5690
5690 5707 5713 5714
5715 5718 5719 5724
5727 5730 5731 5734
5737 5738 5743 5775
5821 5822 5824 5833
5834
lgdt 0453
0453 0461 0954 1054
1711
lidt 0467
0467 0475 2580
LINT0 5679
5679 5718
LINT1 5680
5680 5719
LIST 6860
6860 6940 7107 7383
listcmd 6890 7101
6890 6911 6941 7101
7103 7246 7357 7384
loadfsgs 0514
0514 1712
ltr 0479
0479 0481 1731
MAXARGS 6863
6863 6871 6872 7340
MAXFILE 3169
3169 4165 4166
memcmp 5311
0330 5311 5543 5588
memmove 5327
0331 1276 1814 1841
1871 3683 3839 3924
4121 4171 4329 4331
5088 5327 6474
memset 5304
0332 1789 1813 1816
1842 2313 3694 3814
4784 4959 5061 5075
5304 6476 6987 7058
7069 7085 7106 7119
microdelay 5781
0274 5781 5823 5825
5835
min 3673
3673 4120 4170
mp 5402
5402 5507 5536 5542
5543 5544 5555 5560
5564 5565 5568 5569
5580 5583 5585 5587
5594 5604 5610 5642
mpbcpu 5519
0278 1218 1255 5519
MPBUS 5452
5452 5631
mpconf 5413
5413 5579 5582 5587
5605
mpconfig 5580
5580 5610
mpinit 5601
0279 1217 5601 5637
5638

mpioapic 5439
5439 5607 5627 5629
MPIOAPIC 5453
5453 5626
MPIOINTR 5454
5454 5632
MPLINTR 5455
5455 5633
mpmain 1253
1211 1239 1253 1258
1285
mpproc 5428
5428 5606 5619 5624
MPPROC 5451
5451 5618
mpsearch 5556
5556 5585
mpsearch1 5537
5537 5564 5568 5571
namecmp 4203
0244 4203 4228 4765
namei 4389
0245 1826 4389 4670
4865 4935 5021
nameiparent 4396
0246 4354 4369 4381
4396 4681 4760 4807
namex 4354
4354 4392 4398
NBUF 0156
0156 3531 3553
ncpu 5515
1224 1278 1569 3357
5515 5609 5620 5622
5623
NCPU 0153
0153 1568 5512
NDEV 0158
0158 4108 4158 4406
NDIRECT 3167
3167 3169 3178 3274
4015 4020 4024 4025
4060 4067 4068 4075
4076
NELEM 0362
0362 1672 2879 4961
nextpid 1615
1615 1768
NFILE 0155
0155 4409 4425
NINDIRECT 3168
3168 3169 4022 4070
NINODE 0157
0157 3785 3861
NO 6106
6106 6152 6155 6157
6158 6159 6160 6162
6174 6177 6179 6180
6181 6182 6184 6202
6203 6205 6206 6207
6208
NOFILE 0154
0154 1540 1878 2113
4570 4586
NPROC 0150
0150 1610 1669 1760
1918 2057 2080 2129
2162
NSEGS 1508
1508 1561
nulterminate 7352
7215 7230 7352 7373
7379 7380 7385 7386
7391
NUMLOCK 6113
6113 6146
O_CREATE 3053
3053 4861 7278 7281
O_RDONLY 3050
3050 4868 7275
O_RDWR 3052
3052 4886 6814 6816
7007
outb 0421
0421 0933 0941 1164
1165 1166 1167 1168
1169 3361 3370 3381
3382 3383 3384 3385
3386 3388 3391 5645
5646 5813 5814 5970
5971 5985 5986 5994
5997 6002 6012 6015
6016 6017 6020 6026
6027 6029 6030 6460
6462 6479 6480 6481
6482 6677 6678 6679
outsl 0433
0433 0435 3389
outw 0427
0427 0982 0984 1082

1084
O_WRONLY 3051
3051 4885 4886 7278
7281
PAGE 0151
0151 0152 1811 2284
2285 2309 2359 5054
5057 5178 5202 5223
panic 6401 7032
0219 1377 1405 1469
1471 1958 1960 1962
1964 2006 2009 2110
2140 2310 2321 2360
2647 3378 3459 3461
3463 3596 3617 3627
3725 3743 3822 3873
3908 3928 3937 3958
4036 4219 4267 4275
4442 4458 4517 4537
4709 4777 4786 4821
4834 4838 5638 6401
6408 6901 6920 6953
7032 7045 7228 7272
7306 7310 7336 7341
panicked 6316
6316 6414 6489
parseblock 7301
7301 7306 7325
parsecmd 7218
6902 7025 7218
parseexec 7317
7214 7255 7317
parseline 7235
7212 7224 7235 7246
7308
parsepipe 7251
7213 7239 7251 7258
parseredirs 7264
7264 7312 7331 7342
PCINT 5678
5678 5724
peek 7201
7201 7225 7240 7244
7256 7269 7305 7309
7324 7332
picenable 5975
0283 3356 5975 6625
6680
picinit 5982
0284 1220 5982
picsetmask 5967
5967 5977 6033
pinit 1620
0306 1227 1620
pipe 5160
0204 0288 0289 0290
3255 4469 4509 4529
5160 5172 5178 5184
5188 5192 5211 5230
5251 6763 6952 6953
PIPE 6859
6859 6950 7086 7377
pipealloc 5170
0287 4984 5170
pipeclose 5211
0288 4469 5211
pipecmd 6884 7080
6884 6912 6951 7080
7082 7258 7358 7378
piperead 5251
0289 4509 5251
PIPESIZE 5158
5158 5162 5236 5244
5266
pipewrite 5230
0290 4529 5230
popcli 1466
0327 1421 1466 1469
1471 1732
printint 6324
6324 6374 6378
proc 1529
0205 0301 0342 0343
1204 1208 1357 1529
1535 1578 1605 1610
1613 1665 1669 1753
1756 1760 1804 1857
1910 1918 2055 2057
2077 2080 2106 2129
2155 2162 2554 2650
2754 2766 2778 2905
3306 3667 4555 5003
5154 5510 5606 5619
5620 5621 6311
procdump 1654
0307 1654 6520
proghdr 0874
0874 1119 1133 5016
pushcli 1455
0326 1375 1455 1724

readeflags 0485
0485 1459 1468 1963
5758
readi 4102
0247 4102 4266 4512
4708 4709 5026 5034
5065 5073
readsb 3678
3678 3711 3738 3809
readsect 1160
1160 1195
readseg 1179
1113 1126 1137 1179
REDIR 6858
6858 6930 7070 7371
redircmd 6875 7064
6875 6913 6931 7064
7066 7275 7278 7281
7359 7372
REG_ID 5860
5860 5910
REG_TABLE 5862
5862 5917 5918 5931
5932
REG_VER 5861
5861 5909
release 1402
0325 1402 1405 1763
1769 1934 1978 1987
2019 2032 2068 2086
2090 2176 2183 2343
2369 2373 2619 2975
2980 3409 3428 3482
3578 3592 3641 3864
3880 3892 3914 3942
3960 3969 4428 4432
4444 4460 4466 5222
5225 5238 5247 5258
5269 6398 6547 6562
6582 6609
ROOTDEV 0159
0159 4359
ROOTINO 3157
3157 4359
run 2262
1661 2262 2263 2269
2307 2316 2317 2319
2357
runcmd 6906
6906 6920 6937 6943
6945 6959 6966 6977
7025
RUNNING 1526
1526 1661 1927 1961
2663
safestrcpy 5375
0333 1825 5104 5375
sched 1953
1953 1958 1960 1962
1964 1977 2025 2139
scheduler 1908
0308 1263 1908
SCROLLLOCK 6114
6114 6147
SECTSIZE 1111
1111 1173 1186 1189
1194
SEG 0701
0701 1708 1709 1710
1725 1726
SEG16 0706
0706 1727
SEG_ASM 0608
0608 0992 0993 1092
1093
segdesc 0677
0450 0453 0677 0701
0706 1561
SEG_KCODE 0907 1021 1502 2500
0961 1061 1502 1708
2571 2572
SEG_KCPU 1504 2502
1504 1710 1712 2518
SEG_KDATA 0908 1022 1503 2501
0966 1066 1503 1709
1729 2515
SEG_NULLASM 0604
0604 0991 1091
SEG_TSS 1507
1507 1727 1728 1731
SEG_UCODE 1505
1505 1725 1817
SEG_UDATA 1506
1506 1726 1818
SETGATE 0821
0821 2571 2572
SHIFT 6108
6108 6136 6137 6285
skipelem 4315
4315 4363

sleep 2003
0311 1659 2003 2006
2009 2188 2978 3480
3581 3912 5242 5261
6566 6779
spinlock 1301
0206 0311 0321 0323
0324 0325 0354 1301
1358 1361 1373 1402
1444 1606 1609 2003
2260 2268 2557 2562
3309 3324 3526 3530
3668 3784 4404 4408
5156 5161 6308 6319
6502
STA_R 0617 0718
0617 0718 0992 1092
1708 1725
start 0914 1028 6707
0913 0914 0975 1027
1028 1075 1076 6706
6707
stat 3104
0207 0230 0248 3104
3665 4085 4476 4553
4654 6803
stati 4085
0248 4085 4480
STA_W 0616 0717
0616 0717 0993 1093
1709 1710 1726
STA_X 0613 0714
0613 0714 0992 1092
1708 1725
sti 0527
0527 0529 1473 1914
stosb 0442
0442 0444 1139 5306
strlen 5389
0334 5046 5086 5389
7019 7223
strncmp 5351
0335 4205 5351
strncpy 5361
0336 4272 5361
STS_IG32 0732
0732 0827
STS_T32A 0729
0729 1727
STS_TG32 0733
0733 0827
sum 5525
5525 5527 5529 5531
5532 5543 5592
superblock 3161
3161 3678 3708 3733
3807
SVR 5664
5664 5707
swtch 2208
0318 1928 1967 2207
2208
syscall 2874
0344 2607 2756 2874
SYSCALL 6753 6760 6761 6762 6763 67
6760 6761 6762 6763
6764 6765 6766 6767
6768 6769 6770 6771
6772 6773 6774 6775
6776 6777 6778 6779
sys_chdir 4930
2829 2851 4930
SYS_chdir 2716
2716 2851
sys_close 4639
2830 2852 4639
SYS_close 2707
2707 2852
sys_dup 4601
2831 2853 4601
SYS_dup 2717
2717 2853
sys_exec 4951
2832 2854 4951
SYS_exec 2709
2709 2854 6711
sys_exit 2914
2833 2855 2914
SYS_exit 2702
2702 2855 6716
sys_fork 2908
2834 2856 2908
SYS_fork 2701
2701 2856
sys_fstat 4651
2835 2857 4651
SYS_fstat 2713
2713 2857
sys_getpid 2937
2836 2858 2937

SYS_getpid 2718
2718 2858
sys_kill 2927
2837 2859 2927
SYS_kill 2708
2708 2859
sys_link 4663
2838 2860 4663
SYS_link 2714
2714 2860
sys_mkdir 4901
2839 2861 4901
SYS_mkdir 2715
2715 2861
sys_mknod 4913
2840 2862 4913
SYS_mknod 2711
2711 2862
sys_open 4851
2841 2863 4851
SYS_open 2710
2710 2863
sys_pipe 4976
2842 2864 4976
SYS_pipe 2704
2704 2864
sys_read 4615
2843 2865 4615
SYS_read 2706
2706 2865
sys_sbrk 2951
2844 2866 2951
SYS_sbrk 2719
2719 2866
sys_sleep 2965
2845 2867 2965
SYS_sleep 2720
2720 2867
sys_unlink 4751
2846 2868 4751
SYS_unlink 2712
2712 2868
sys_wait 2921
2847 2869 2921
SYS_wait 2703
2703 2869
sys_write 4627
2848 2870 4627
SYS_write 2705
2705 2870
taskstate 0751
0751 1560
TDCR 5685
5685 5713
T_DEV 3102
3102 4107 4157 4923
T_DIR 3100
3100 4218 4365 4673
4778 4787 4829 4868
4906 4938
T_FILE 3101
3101 4814 4862
ticks 2563
0352 2563 2617 2618
2972 2973 2978
tickslock 2562
0354 2562 2574 2616
2619 2971 2975 2978
2980
TICR 5683
5683 5715
TIMER 5675
5675 5714
TIMER_16BIT 6671
6671 6677
TIMER_DIV 6666
6666 6678 6679
TIMER_FREQ 6665
6665 6666
timerinit 6674
0347 1234 6674
TIMER_MODE 6668
6668 6677
TIMER_RATEGEN 6670
6670 6677
TIMER_SELO 6669
6669 6677
T_IRQ0 2429
2429 2614 2623 2627
2631 2635 2636 2663
5707 5714 5727 5917
5931 5997 6016
TPR 5662
5662 5743
trap 2601
2452 2454 2524 2601
2645 2647 2650
trapframe 0552
0552 1536 1780 2601
trapret 2529

1617 1785 2528 2529
T_SYSCALL 2426
2426 2572 2603 6712
6717 6757
tvinit 2566
0353 1228 2566
usegment 1722
0310 1722 1846 1926
5112
userinit 1802
0312 1235 1802
VER 5661
5661 5723
wait 2153
0313 2153 2923 6762
6833 6944 6970 6971
7026
waitdisk 1151
1151 1163 1172
wakeup 2064
0314 2064 2618 3422
3639 3941 3966 5216
5219 5241 5246 5268
6541
wakeup1 2053
2053 2067 2126 2133
writei 4152
0249 4152 4274 4532
4785 4786
xchg 0501
0501 1260 1382 1419
yield 1973
0315 1973 2664

```
0100 typedef unsigned int   uint;
0101 typedef unsigned short ushort;
0102 typedef unsigned char  uchar;
0103
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 PAGE       4096  // granularity of user-space memory allocation
0152 #define KSTACKSIZE PAGE  // size of per-process kernel stack
0153 #define NCPU          8  // maximum number of CPUs
0154 #define NOFILE       16  // open files per process
0155 #define NFILE       100  // open files per system
0156 #define NBUF         10  // size of disk block cache
0157 #define NINODE       50  // maximum number of active i-nodes
0158 #define NDEV         10  // maximum major device number
0159 #define ROOTDEV       1  // device number of file system root disk
0160
0161
0162
0163
0164
0165
0166
0167
0168
0169
0170
0171
0172
0173
0174
0175
0176
0177
0178
0179
0180
0181
0182
0183
0184
0185
0186
0187
0188
0189
0190
0191
0192
0193
0194
0195
0196
0197
0198
0199
```

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



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

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



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

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



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

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



```
0550 // Layout of the trap frame built on the stack by the
0551 // hardware and by trapasm.S, and passed to trap().
0552 struct trapframe {
0553   // registers as pushed by pusha
0554   uint edi;
0555   uint esi;
0556   uint ebp;
0557   uint oesp;      // useless & ignored
0558   uint ebx;
0559   uint edx;
0560   uint ecx;
0561   uint eax;
0562
0563   // rest of trap frame
0564   ushort gs;
0565   ushort padding1;
0566   ushort fs;
0567   ushort padding2;
0568   ushort es;
0569   ushort padding3;
0570   ushort ds;
0571   ushort padding4;
0572   uint trapno;
0573
0574   // below here defined by x86 hardware
0575   uint err;
0576   uint eip;
0577   ushort cs;
0578   ushort padding5;
0579   uint eflags;
0580
0581   // below here only when crossing rings, such as from user to kernel
0582   uint esp;
0583   ushort ss;
0584   ushort padding6;
0585 };
0586
0587
0588
0589
0590
0591
0592
0593
0594
0595
0596
0597
0598
0599
```

```
0600 //
0601 // assembler macros to create x86 segments
0602 //
0603
0604 #define SEG_NULLASM                                             \
0605         .word 0, 0;                                             \
0606         .byte 0, 0, 0, 0
0607
0608 #define SEG_ASM(type,base,lim)                                  \
0609         .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
0610         .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
0611                 (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
0612
0613 #define STA_X     0x8       // Executable segment
0614 #define STA_E     0x4       // Expand down (non-executable segments)
0615 #define STA_C     0x4       // Conforming code segment (executable only)
0616 #define STA_W     0x2       // Writeable (non-executable segments)
0617 #define STA_R     0x2       // Readable (executable segments)
0618 #define STA_A     0x1       // Accessed
0619
0620
0621
0622
0623
0624
0625
0626
0627
0628
0629
0630
0631
0632
0633
0634
0635
0636
0637
0638
0639
0640
0641
0642
0643
0644
0645
0646
0647
0648
0649
```



```
0650 // This file contains definitions for the
0651 // x86 memory management unit (MMU).
0652
0653 // Eflags register
0654 #define FL_CF           0x00000001      // Carry Flag
0655 #define FL_PF           0x00000004      // Parity Flag
0656 #define FL_AF           0x00000010      // Auxiliary carry Flag
0657 #define FL_ZF           0x00000040      // Zero Flag
0658 #define FL_SF           0x00000080      // Sign Flag
0659 #define FL_TF           0x00000100      // Trap Flag
0660 #define FL_IF           0x00000200      // Interrupt Enable
0661 #define FL_DF           0x00000400      // Direction Flag
0662 #define FL_OF           0x00000800      // Overflow Flag
0663 #define FL_IOPL_MASK    0x00003000      // I/O Privilege Level bitmask
0664 #define FL_IOPL_0       0x00000000      //   IOPL == 0
0665 #define FL_IOPL_1       0x00001000      //   IOPL == 1
0666 #define FL_IOPL_2       0x00002000      //   IOPL == 2
0667 #define FL_IOPL_3       0x00003000      //   IOPL == 3
0668 #define FL_NT           0x00004000      // Nested Task
0669 #define FL_RF           0x00010000      // Resume Flag
0670 #define FL_VM           0x00020000      // Virtual 8086 mode
0671 #define FL_AC           0x00040000      // Alignment Check
0672 #define FL_VIF          0x00080000      // Virtual Interrupt Flag
0673 #define FL_VIP          0x00100000      // Virtual Interrupt Pending
0674 #define FL_ID           0x00200000      // ID flag
0675
0676 // Segment Descriptor
0677 struct segdesc {
0678   uint lim_15_0 : 16;  // Low bits of segment limit
0679   uint base_15_0 : 16; // Low bits of segment base address
0680   uint base_23_16 : 8; // Middle bits of segment base address
0681   uint type : 4;       // Segment type (see STS_ constants)
0682   uint s : 1;          // 0 = system, 1 = application
0683   uint dpl : 2;        // Descriptor Privilege Level
0684   uint p : 1;          // Present
0685   uint lim_19_16 : 4;  // High bits of segment limit
0686   uint avl : 1;        // Unused (available for software use)
0687   uint rsv1 : 1;       // Reserved
0688   uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
0689   uint g : 1;          // Granularity: limit scaled by 4K when set
0690   uint base_31_24 : 8; // High bits of segment base address
0691 };
0692
0693
0694
0695
0696
0697
0698
0699
```

```
0700 // Normal segment
0701 #define SEG(type, base, lim, dpl) (struct segdesc)                    \
0702 { ((lim) >> 12) & 0xffff, (base) & 0xffff, ((base) >> 16) & 0xff,       \
0703     type, 1, dpl, 1, (uint) (lim) >> 28, 0, 0, 1, 1,                   \
0704     (uint) (base) >> 24 }
0705
0706 #define SEG16(type, base, lim, dpl) (struct segdesc)                   \
0707 { (lim) & 0xffff, (base) & 0xffff, ((base) >> 16) & 0xff,              \
0708     type, 1, dpl, 1, (uint) (lim) >> 16, 0, 0, 1, 0,                   \
0709     (uint) (base) >> 24 }
0710
0711 #define DPL_USER    0x3     // User DPL
0712
0713 // Application segment type bits
0714 #define STA_X       0x8     // Executable segment
0715 #define STA_E       0x4     // Expand down (non-executable segments)
0716 #define STA_C       0x4     // Conforming code segment (executable only)
0717 #define STA_W       0x2     // Writeable (non-executable segments)
0718 #define STA_R       0x2     // Readable (executable segments)
0719 #define STA_A       0x1     // Accessed
0720
0721 // System segment type bits
0722 #define STS_T16A    0x1     // Available 16-bit TSS
0723 #define STS_LDT     0x2     // Local Descriptor Table
0724 #define STS_T16B    0x3     // Busy 16-bit TSS
0725 #define STS_CG16    0x4     // 16-bit Call Gate
0726 #define STS_TG      0x5     // Task Gate / Coum Transmitions
0727 #define STS_IG16    0x6     // 16-bit Interrupt Gate
0728 #define STS_TG16    0x7     // 16-bit Trap Gate
0729 #define STS_T32A    0x9     // Available 32-bit TSS
0730 #define STS_T32B    0xB     // Busy 32-bit TSS
0731 #define STS_CG32    0xC     // 32-bit Call Gate
0732 #define STS_IG32    0xE     // 32-bit Interrupt Gate
0733 #define STS_TG32    0xF     // 32-bit Trap Gate
0734
0735
0736
0737
0738
0739
0740
0741
0742
0743
0744
0745
0746
0747
0748
0749
```



```
0750 // Task state segment format
0751 struct taskstate {
0752   uint link;         // Old ts selector
0753   uint esp0;         // Stack pointers and segment selectors
0754   ushort ss0;        //   after an increase in privilege level
0755   ushort padding1;
0756   uint *esp1;
0757   ushort ss1;
0758   ushort padding2;
0759   uint *esp2;
0760   ushort ss2;
0761   ushort padding3;
0762   void *cr3;         // Page directory base
0763   uint *eip;         // Saved state from last task switch
0764   uint eflags;
0765   uint eax;          // More saved state (registers)
0766   uint ecx;
0767   uint edx;
0768   uint ebx;
0769   uint *esp;
0770   uint *ebp;
0771   uint esi;
0772   uint edi;
0773   ushort es;         // Even more saved state (segment selectors)
0774   ushort padding4;
0775   ushort cs;
0776   ushort padding5;
0777   ushort ss;
0778   ushort padding6;
0779   ushort ds;
0780   ushort padding7;
0781   ushort fs;
0782   ushort padding8;
0783   ushort gs;
0784   ushort padding9;
0785   ushort ldt;
0786   ushort padding10;
0787   ushort t;          // Trap on task switch
0788   ushort iomb;       // I/O map base address
0789 };
0790
0791
0792
0793
0794
0795
0796
0797
0798
0799
```

```
0800 // Gate descriptors for interrupts and traps
0801 struct gatedesc {
0802   uint off_15_0 : 16;   // low 16 bits of offset in segment
0803   uint cs : 16;         // code segment selector
0804   uint args : 5;        // # args, 0 for interrupt/trap gates
0805   uint rsv1 : 3;        // reserved(should be zero I guess)
0806   uint type : 4;        // type(STS_{TG,IG32,TG32})
0807   uint s : 1;           // must be 0 (system)
0808   uint dpl : 2;         // descriptor(meaning new) privilege level
0809   uint p : 1;           // Present
0810   uint off_31_16 : 16;  // high bits of offset in segment
0811 };
0812
0813 // Set up a normal interrupt/trap gate descriptor.
0814 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
0815 //   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
0816 // - sel: Code segment selector for interrupt/trap handler
0817 // - off: Offset in code segment for interrupt/trap handler
0818 // - dpl: Descriptor Privilege Level -
0819 //        the privilege level required for software to invoke
0820 //        this interrupt/trap gate explicitly using an int instruction.
0821 #define SETGATE(gate, istrap, sel, off, d)                \
0822 {                                                         \
0823   (gate).off_15_0 = (uint) (off) & 0xffff;                \
0824   (gate).cs = (sel);                                      \
0825   (gate).args = 0;                                        \
0826   (gate).rsv1 = 0;                                        \
0827   (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
0828   (gate).s = 0;                                           \
0829   (gate).dpl = (d);                                       \
0830   (gate).p = 1;                                           \
0831   (gate).off_31_16 = (uint) (off) >> 16;                  \
0832 }
0833
0834
0835
0836
0837
0838
0839
0840
0841
0842
0843
0844
0845
0846
0847
0848
0849
```



```
0850 // Format of an ELF executable file
0851
0852 #define ELF_MAGIC 0x464C457FU  // "\x7FELF" in little endian
0853
0854 // File header
0855 struct elfhdr {
0856   uint magic;  // must equal ELF_MAGIC
0857   uchar elf[12];
0858   ushort type;
0859   ushort machine;
0860   uint version;
0861   uint entry;
0862   uint phoff;
0863   uint shoff;
0864   uint flags;
0865   ushort ehsize;
0866   ushort phentsize;
0867   ushort phnum;
0868   ushort shentsize;
0869   ushort shnum;
0870   ushort shstrndx;
0871 };
0872
0873 // Program section header
0874 struct proghdr {
0875   uint type;
0876   uint offset;
0877   uint va;
0878   uint pa;
0879   uint filesz;
0880   uint memsz;
0881   uint flags;
0882   uint align;
0883 };
0884
0885 // Values for Proghdr type
0886 #define ELF_PROG_LOAD           1
0887
0888 // Flag bits for Proghdr flags
0889 #define ELF_PROG_FLAG_EXEC      1
0890 #define ELF_PROG_FLAG_WRITE     2
0891 #define ELF_PROG_FLAG_READ      4
0892
0893
0894
0895
0896
0897
0898
0899
```

```
0900 #include "asm.h"
0901
0902 # Start the first CPU: switch to 32-bit protected mode, jump into C.
0903 # The BIOS loads this code from the first sector of the hard disk into
0904 # memory at physical address 0x7c00 and starts executing in real mode
0905 # with %cs=0 %ip=7c00.
0906
0907 #define SEG_KCODE 1  // kernel code
0908 #define SEG_KDATA 2  // kernel data+stack
0909
0910 #define CR0_PE    1  // protected mode enable bit
0911
0912 .code16                       # Assemble for 16-bit mode
0913 .globl start
0914 start:
0915   cli                         # Disable interrupts
0916
0917   # Set up the important data segment registers (DS, ES, SS).
0918   xorw    %ax,%ax             # Segment number zero
0919   movw    %ax,%ds             # -> Data Segment
0920   movw    %ax,%es             # -> Extra Segment
0921   movw    %ax,%ss             # -> Stack Segment
0922
0923   # Enable A20:
0924   #   For backwards compatibility with the earliest PCs, physical
0925   #   address line 20 is tied low, so that addresses higher than
0926   #   1MB wrap around to zero by default.  This code undoes this.
0927 seta20.1:
0928   inb     $0x64,%al               # Wait for not busy
0929   testb   $0x2,%al
0930   jnz     seta20.1
0931
0932   movb    $0xd1,%al               # 0xd1 -> port 0x64
0933   outb    %al,$0x64
0934
0935 seta20.2:
0936   inb     $0x64,%al               # Wait for not busy
0937   testb   $0x2,%al
0938   jnz     seta20.2
0939
0940   movb    $0xdf,%al               # 0xdf -> port 0x60
0941   outb    %al,$0x60
0942
0943
0944
0945
0946
0947
0948
0949
```



```
0950   # Switch from real to protected mode, using a bootstrap GDT
0951   # and segment translation that makes virtual addresses
0952   # identical to physical addresses, so that the
0953   # effective memory map does not change during the switch.
0954   lgdt    gdtdesc
0955   movl    %cr0, %eax
0956   orl     $CR0_PE, %eax
0957   movl    %eax, %cr0
0958
0959   # Jump to next instruction, but in 32-bit code segment.
0960   # Switches processor into 32-bit mode.
0961   ljmp    $(SEG_KCODE<<3), $start32
0962
0963 .code32                       # Assemble for 32-bit mode
0964 start32:
0965   # Set up the protected-mode data segment registers
0966   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
0967   movw    %ax, %ds                # -> DS: Data Segment
0968   movw    %ax, %es                # -> ES: Extra Segment
0969   movw    %ax, %ss                # -> SS: Stack Segment
0970   movw    $0, %ax                 # Zero segments not ready for use
0971   movw    %ax, %fs                # -> FS
0972   movw    %ax, %gs                # -> GS
0973
0974   # Set up the stack pointer and call into C.
0975   movl    $start, %esp
0976   call    bootmain
0977
0978   # If bootmain returns (it shouldn't), trigger a Bochs
0979   # breakpoint if running under Bochs, then loop.
0980   movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
0981   movw    %ax, %dx
0982   outw    %ax, %dx
0983   movw    $0x8e00, %ax            # 0x8e00 -> port 0x8a00
0984   outw    %ax, %dx
0985 spin:
0986   jmp     spin
0987
0988 # Bootstrap GDT
0989 .p2align 2                                # force 4 byte alignment
0990 gdt:
0991   SEG_NULLASM                             # null seg
0992   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
0993   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
0994
0995 gdtdesc:
0996   .word   (gdtdesc - gdt - 1)                           # sizeof(gdt) - 1
0997   .long   gdt                             # address gdt
0998
0999
```

```
1000 #include "asm.h"
1001
1002 # Each non-boot CPU ("AP") is started up in response to a STARTUP
1003 # IPI from the boot CPU.  Section B.4.2 of the Multi-Processor
1004 # Specification says that the AP will start in real mode with CS:IP
1005 # set to XY00:0000, where XY is an 8-bit value sent with the
1006 # STARTUP. Thus this code must start at a 4096-byte boundary.
1007 #
1008 # Because this code sets DS to zero, it must sit
1009 # at an address in the low 2^16 bytes.
1010 #
1011 # Bootothers (in main.c) sends the STARTUPs, one at a time.
1012 # It puts this code (start) at 0x7000.
1013 # It puts the correct %esp in start-4,
1014 # and the place to jump to in start-8.
1015 #
1016 # This code is identical to bootasm.S except:
1017 #   - it does not need to enable A20
1018 #   - it uses the address at start-4 for the %esp
1019 #   - it jumps to the address at start-8 instead of calling bootmain
1020
1021 #define SEG_KCODE 1  // kernel code
1022 #define SEG_KDATA 2  // kernel data+stack
1023
1024 #define CR0_PE    1  // protected mode enable bit
1025
1026 .code16                       # Assemble for 16-bit mode
1027 .globl start
1028 start:
1029   cli                         # Disable interrupts
1030
1031   # Set up the important data segment registers (DS, ES, SS).
1032   xorw    %ax,%ax             # Segment number zero
1033   movw    %ax,%ds             # -> Data Segment
1034   movw    %ax,%es             # -> Extra Segment
1035   movw    %ax,%ss             # -> Stack Segment
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
```



```
1050   # Switch from real to protected mode, using a bootstrap GDT
1051   # and segment translation that makes virtual addresses
1052   # identical to physical addresses, so that the
1053   # effective memory map does not change during the switch.
1054   lgdt    gdtdesc
1055   movl    %cr0, %eax
1056   orl     $CR0_PE, %eax
1057   movl    %eax, %cr0
1058
1059   # Jump to next instruction, but in 32-bit code segment.
1060   # Switches processor into 32-bit mode.
1061   ljmp    $(SEG_KCODE<<3), $start32
1062
1063 .code32                       # Assemble for 32-bit mode
1064 start32:
1065   # Set up the protected-mode data segment registers
1066   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
1067   movw    %ax, %ds                # -> DS: Data Segment
1068   movw    %ax, %es                # -> ES: Extra Segment
1069   movw    %ax, %ss                # -> SS: Stack Segment
1070   movw    $0, %ax                 # Zero segments not ready for use
1071   movw    %ax, %fs                # -> FS
1072   movw    %ax, %gs                # -> GS
1073
1074   # Set up the stack pointer and call into C.
1075   movl    start-4, %esp
1076   call     *(start-8)
1077
1078   # If the call returns (it shouldn't), trigger a Bochs
1079   # breakpoint if running under Bochs, then loop.
1080   movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
1081   movw    %ax, %dx
1082   outw    %ax, %dx
1083   movw    $0x8e00, %ax            # 0x8e00 -> port 0x8a00
1084   outw    %ax, %dx
1085 spin:
1086   jmp     spin
1087
1088 # Bootstrap GDT
1089 .p2align 2                                # force 4 byte alignment
1090 gdt:
1091   SEG_NULLASM                             # null seg
1092   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
1093   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
1094
1095 gdtdesc:
1096   .word   (gdtdesc - gdt - 1)                           # sizeof(gdt) - 1
1097   .long   gdt                             # address gdt
1098
1099
```

```
1100 // Boot loader.
1101 //
1102 // Part of the boot sector, along with bootasm.S, which calls bootmain().
1103 // bootasm.S has put the processor into protected 32-bit mode.
1104 // bootmain() loads an ELF kernel image from the disk starting at
1105 // sector 1 and then jumps to the kernel entry routine.
1106
1107 #include "types.h"
1108 #include "elf.h"
1109 #include "x86.h"
1110
1111 #define SECTSIZE  512
1112
1113 void readseg(uchar*, uint, uint);
1114
1115 void
1116 bootmain(void)
1117 {
1118   struct elfhdr *elf;
1119   struct proghdr *ph, *eph;
1120   void (*entry)(void);
1121   uchar* va;
1122
1123   elf = (struct elfhdr*)0x10000;  // scratch space
1124
1125   // Read 1st page off disk
1126   readseg((uchar*)elf, 4096, 0);
1127
1128   // Is this an ELF executable?
1129   if(elf->magic != ELF_MAGIC)
1130     return;  // let bootasm.S handle error
1131
1132   // Load each program segment (ignores ph flags).
1133   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
1134   eph = ph + elf->phnum;
1135   for(; ph < eph; ph++) {
1136     va = (uchar*)(ph->va & 0xFFFFFF);
1137     readseg(va, ph->filesz, ph->offset);
1138     if(ph->memsz > ph->filesz)
1139       stosb(va + ph->filesz, 0, ph->memsz - ph->filesz);
1140   }
1141
1142   // Call the entry point from the ELF header.
1143   // Does not return!
1144   entry = (void(*)(void))(elf->entry & 0xFFFFFF);
1145   entry();
1146 }
1147
1148
1149
```



```
1150 void
1151 waitdisk(void)
1152 {
1153   // Wait for disk ready.
1154   while((inb(0x1F7) & 0xC0) != 0x40)
1155     ;
1156 }
1157
1158 // Read a single sector at offset into dst.
1159 void
1160 readsect(void *dst, uint offset)
1161 {
1162   // Issue command.
1163   waitdisk();
1164   outb(0x1F2, 1);   // count = 1
1165   outb(0x1F3, offset);
1166   outb(0x1F4, offset >> 8);
1167   outb(0x1F5, offset >> 16);
1168   outb(0x1F6, (offset >> 24) | 0xE0);
1169   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
1170
1171   // Read data.
1172   waitdisk();
1173   insl(0x1F0, dst, SECTSIZE/4);
1174 }
1175
1176 // Read 'count' bytes at 'offset' from kernel into virtual address 'va'.
1177 // Might copy more than asked.
1178 void
1179 readseg(uchar* va, uint count, uint offset)
1180 {
1181   uchar* eva;
1182
1183   eva = va + count;
1184
1185   // Round down to sector boundary.
1186   va -= offset % SECTSIZE;
1187
1188   // Translate from bytes to sectors; kernel starts at sector 1.
1189   offset = (offset / SECTSIZE) + 1;
1190
1191   // If this is too slow, we could read lots of sectors at a time.
1192   // We'd write more to memory than asked, but it doesn't matter --
1193   // we load in increasing order.
1194   for(; va < eva; va += SECTSIZE, offset++)
1195     readsect(va, offset);
1196 }
1197
1198
1199
```

```
1200 #include "types.h"
1201 #include "defs.h"
1202 #include "param.h"
1203 #include "mmu.h"
1204 #include "proc.h"
1205 #include "x86.h"
1206
1207 __thread struct cpu *c;
1208 __thread struct proc *cp;
1209
1210 static void bootothers(void);
1211 static void mpmain(void) __attribute__((noreturn));
1212
1213 // Bootstrap processor starts running C code here.
1214 int
1215 main(void)
1216 {
1217   mpinit(); // collect info about this machine
1218   lapicinit(mpbcpu());
1219   ksegment();
1220   picinit();       // interrupt controller
1221   ioapicinit();    // another interrupt controller
1222   consoleinit();   // I/O devices & their interrupts
1223   uartinit();      // serial port
1224   cprintf("\ncpu%d: starting xv6\n\n", cpu());
1225
1226   kinit();         // physical memory allocator
1227   pinit();         // process table
1228   tvinit();        // trap vectors
1229   binit();         // buffer cache
1230   fileinit();      // file table
1231   iinit();         // inode cache
1232   ideinit();       // disk
1233   if(!ismp)
1234     timerinit();   // uniprocessor timer
1235   userinit();      // first user process
1236   bootothers();    // start other processors
1237
1238   // Finish setting up this processor in mpmain.
1239   mpmain();
1240 }
1241
1242
1243
1244
1245
1246
1247
1248
1249
```



```
1250 // Bootstrap processor gets here after setting up the hardware.
1251 // Additional processors start here.
1252 static void
1253 mpmain(void)
1254 {
1255   if(cpu() != mpbcpu())
1256     lapicinit(cpu());
1257   ksegment();
1258   cprintf("cpu%d: mpmain\n", cpu());
1259   idtinit();
1260   xchg(&c->booted, 1);
1261
1262   cprintf("cpu%d: scheduling\n", cpu());
1263   scheduler();
1264 }
1265
1266 static void
1267 bootothers(void)
1268 {
1269   extern uchar _binary_bootother_start[], _binary_bootother_size[];
1270   uchar *code;
1271   struct cpu *c;
1272   char *stack;
1273
1274   // Write bootstrap code to unused memory at 0x7000.
1275   code = (uchar*)0x7000;
1276   memmove(code, _binary_bootother_start, (uint)_binary_bootother_size);
1277
1278   for(c = cpus; c < cpus+ncpu; c++){
1279     if(c == cpus+cpu())  // We've started already.
1280       continue;
1281
1282     // Fill in %esp, %eip and start code on cpu.
1283     stack = kalloc(KSTACKSIZE);
1284     *(void**)(code-4) = stack + KSTACKSIZE;
1285     *(void**)(code-8) = mpmain;
1286     lapicstartap(c->apicid, (uint)code);
1287
1288     // Wait for cpu to get through bootstrap.
1289     while(c->booted == 0)
1290       ;
1291   }
1292 }
1293
1294
1295
1296
1297
1298
1299
```

```
1300 // Mutual exclusion lock.
1301 struct spinlock {
1302   uint locked;   // Is the lock held?
1303
1304   // For debugging:
1305   char *name;    // Name of lock.
1306   int  cpu;      // The number of the cpu holding the lock.
1307   uint pcs[10];  // The call stack (an array of program counters)
1308                  // that locked the lock.
1309 };
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
```



```
1350 // Mutual exclusion spin locks.
1351
1352 #include "types.h"
1353 #include "defs.h"
1354 #include "param.h"
1355 #include "x86.h"
1356 #include "mmu.h"
1357 #include "proc.h"
1358 #include "spinlock.h"
1359
1360 void
1361 initlock(struct spinlock *lk, char *name)
1362 {
1363   lk->name = name;
1364   lk->locked = 0;
1365   lk->cpu = 0xffffffff;
1366 }
1367
1368 // Acquire the lock.
1369 // Loops (spins) until the lock is acquired.
1370 // Holding a lock for a long time may cause
1371 // other CPUs to waste time spinning to acquire it.
1372 void
1373 acquire(struct spinlock *lk)
1374 {
1375   pushcli();
1376   if(holding(lk))
1377     panic("acquire");
1378
1379   // The xchg is atomic.
1380   // It also serializes, so that reads after acquire are not
1381   // reordered before it.
1382   while(xchg(&lk->locked, 1) != 0)
1383     ;
1384
1385   // Record info about lock acquisition for debugging.
1386   // The +10 is only so that we can tell the difference
1387   // between forgetting to initialize lock->cpu
1388   // and holding a lock on cpu 0.
1389   lk->cpu = cpu() + 10;
1390   getcallerpcs(&lk, lk->pcs);
1391 }
1392
1393
1394
1395
1396
1397
1398
1399
```

```
1400 // Release the lock.
1401 void
1402 release(struct spinlock *lk)
1403 {
1404   if(!holding(lk))
1405     panic("release");
1406
1407   lk->pcs[0] = 0;
1408   lk->cpu = 0xffffffff;
1409
1410   // The xchg serializes, so that reads before release are
1411   // not reordered after it.  The 1996 PentiumPro manual (Volume 3,
1412   // 7.2) says reads can be carried out speculatively and in
1413   // any order, which implies we need to serialize here.
1414   // But the 2007 Intel 64 Architecture Memory Ordering White
1415   // Paper says that Intel 64 and IA-32 will not move a load
1416   // after a store. So lock->locked = 0 would work here.
1417   // The xchg being asm volatile ensures gcc emits it after
1418   // the above assignments (and after the critical section).
1419   xchg(&lk->locked, 0);
1420
1421   popcli();
1422 }
1423
1424 // Record the current call stack in pcs[] by following the %ebp chain.
1425 void
1426 getcallerpcs(void *v, uint pcs[])
1427 {
1428   uint *ebp;
1429   int i;
1430
1431   ebp = (uint*)v - 2;
1432   for(i = 0; i < 10; i++){
1433     if(ebp == 0 || ebp == (uint*)0xffffffff)
1434       break;
1435     pcs[i] = ebp[1];     // saved %eip
1436     ebp = (uint*)ebp[0]; // saved %ebp
1437   }
1438   for(; i < 10; i++)
1439     pcs[i] = 0;
1440 }
1441
1442 // Check whether this cpu is holding the lock.
1443 int
1444 holding(struct spinlock *lock)
1445 {
1446   return lock->locked && lock->cpu == cpu() + 10;
1447 }
1448
1449
```



```
1450 // Pushcli/popcli are like cli/sti except that they are matched:
1451 // it takes two popcli to undo two pushcli.  Also, if interrupts
1452 // are off, then pushcli, popcli leaves them off.
1453
1454 void
1455 pushcli(void)
1456 {
1457   int eflags;
1458
1459   eflags = readeflags();
1460   cli();
1461   if(c->ncli++ == 0)
1462     c->intena = eflags & FL_IF;
1463 }
1464
1465 void
1466 popcli(void)
1467 {
1468   if(readeflags()&FL_IF)
1469     panic("popcli - interruptible");
1470   if(--c->ncli < 0)
1471     panic("popcli");
1472   if(c->ncli == 0 && c->intena)
1473     sti();
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 // Segments in proc->gdt.
1501 // Also known to bootasm.S and trapasm.S
1502 #define SEG_KCODE 1  // kernel code
1503 #define SEG_KDATA 2  // kernel data+stack
1504 #define SEG_KCPU  3  // kernel per-cpu data
1505 #define SEG_UCODE 4
1506 #define SEG_UDATA 5
1507 #define SEG_TSS   6  // this process's task state
1508 #define NSEGS     7
1509
1510 // Saved registers for kernel context switches.
1511 // Don't need to save all the segment registers (%cs, etc),
1512 // because they are constant across kernel contexts.
1513 // Don't need to save %eax, %ecx, %edx, because the
1514 // x86 convention is that the caller has saved them.
1515 // Contexts are stored at the bottom of the stack they
1516 // describe; the stack pointer is the address of the context.
1517 // The layout of the context must match the code in swtch.S.
1518 struct context {
1519   uint edi;
1520   uint esi;
1521   uint ebx;
1522   uint ebp;
1523   uint eip;
1524 };
1525
1526 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
1527
1528 // Per-process state
1529 struct proc {
1530   char *mem;                   // Start of process memory (kernel address)
1531   uint sz;                     // Size of process memory (bytes)
1532   char *kstack;                // Bottom of kernel stack for this process
1533   enum procstate state;       // Process state
1534   volatile int pid;            // Process ID
1535   struct proc *parent;         // Parent process
1536   struct trapframe *tf;        // Trap frame for current syscall
1537   struct context *context;     // Switch here to run process
1538   void *chan;                  // If non-zero, sleeping on chan
1539   int killed;                  // If non-zero, have been killed
1540   struct file *ofile[NOFILE];  // Open files
1541   struct inode *cwd;           // Current directory
1542   char name[16];               // Process name (debugging)
1543 };
1544
1545
1546
1547
1548
1549
```



```
1550 // Process memory is laid out contiguously, low addresses first:
1551 //   text
1552 //   original data and bss
1553 //   fixed-size stack
1554 //   expandable heap
1555
1556 // Per-CPU state
1557 struct cpu {
1558   uchar apicid;                // Local APIC ID
1559   struct context *context;     // Switch here to enter scheduler
1560   struct taskstate ts;         // Used by x86 to find stack for interrupt
1561   struct segdesc gdt[NSEGS];   // x86 global descriptor table
1562   volatile uint booted;        // Has the CPU started?
1563   int ncli;                    // Depth of pushcli nesting.
1564   int intena;                  // Were interrupts enabled before pushcli?
1565   void *tls[2];
1566 };
1567
1568 extern struct cpu cpus[NCPU];
1569 extern int ncpu;
1570
1571 // Per-CPU variables, holding pointers to the
1572 // current cpu and to the current process.
1573 // The __thread prefix tells gcc to refer to them in the segment
1574 // pointed at by gs; the name __thread derives from the use
1575 // of the same mechanism to provide per-thread storage in
1576 // multithreaded user programs.
1577 extern __thread struct cpu *c;       // This cpu.
1578 extern __thread struct proc *cp;     // Current process on this cpu.
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
```

```
1600 #include "types.h"
1601 #include "defs.h"
1602 #include "param.h"
1603 #include "mmu.h"
1604 #include "x86.h"
1605 #include "proc.h"
1606 #include "spinlock.h"
1607
1608 struct {
1609   struct spinlock lock;
1610   struct proc proc[NPROC];
1611 } ptable;
1612
1613 static struct proc *initproc;
1614
1615 int nextpid = 1;
1616 extern void forkret(void);
1617 extern void trapret(void);
1618
1619 void
1620 pinit(void)
1621 {
1622   initlock(&ptable.lock, "ptable");
1623 }
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
```



```
1650 // Print a process listing to console.  For debugging.
1651 // Runs when user types ^P on console.
1652 // No lock to avoid wedging a stuck machine further.
1653 void
1654 procdump(void)
1655 {
1656   static char *states[] = {
1657   [UNUSED]    "unused",
1658   [EMBRYO]    "embryo",
1659   [SLEEPING]  "sleep ",
1660   [RUNNABLE]  "runble",
1661   [RUNNING]   "run   ",
1662   [ZOMBIE]    "zombie"
1663   };
1664   int i;
1665   struct proc *p;
1666   char *state;
1667   uint pc[10];
1668
1669   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
1670     if(p->state == UNUSED)
1671       continue;
1672     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
1673       state = states[p->state];
1674     else
1675       state = "???";
1676     cprintf("%d %s %s", p->pid, state, p->name);
1677     if(p->state == SLEEPING){
1678       getcallerpcs((uint*)p->context->ebp+2, pc);
1679       for(i=0; i<10 && pc[i] != 0; i++)
1680         cprintf(" %p", pc[i]);
1681     }
1682     cprintf("\n");
1683   }
1684 }
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
```

```
1700 // Set up CPU's kernel segment descriptors.
1701 // Run once at boot time on each CPU.
1702 void
1703 ksegment(void)
1704 {
1705   struct cpu *c1;
1706
1707   c1 = &cpus[cpu()];
1708   c1->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0x100000 + 64*1024-1, 0);
1709   c1->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1710   c1->gdt[SEG_KCPU] = SEG(STA_W, (uint)(&c1->tls+1), 0xffffffff, 0);
1711   lgdt(c1->gdt, sizeof(c1->gdt));
1712   loadfsgs(SEG_KCPU << 3);
1713
1714   // Initialize cpu-local variables.
1715   c = c1;
1716   cp = 0;
1717 }
1718
1719 // Set up CPU's segment descriptors and current process task state.
1720 // If cp==0, set up for "idle" state for when scheduler() is running.
1721 void
1722 usegment(void)
1723 {
1724   pushcli();
1725   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, (uint)cp->mem, cp->sz-1, DPL_USER);
1726   c->gdt[SEG_UDATA] = SEG(STA_W, (uint)cp->mem, cp->sz-1, DPL_USER);
1727   c->gdt[SEG_TSS] = SEG16(STS_T32A, (uint)&c->ts, sizeof(c->ts)-1, 0);
1728   c->gdt[SEG_TSS].s = 0;
1729   c->ts.ss0 = SEG_KDATA << 3;
1730   c->ts.esp0 = (uint)cp->kstack + KSTACKSIZE;
1731   ltr(SEG_TSS << 3);
1732   popcli();
1733 }
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
```



```
1750 // Look in the process table for an UNUSED proc.
1751 // If found, change state to EMBRYO and return it.
1752 // Otherwise return 0.
1753 static struct proc*
1754 allocproc(void)
1755 {
1756   struct proc *p;
1757   char *sp;
1758
1759   acquire(&ptable.lock);
1760   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
1761     if(p->state == UNUSED)
1762       goto found;
1763   release(&ptable.lock);
1764   return 0;
1765
1766 found:
1767   p->state = EMBRYO;
1768   p->pid = nextpid++;
1769   release(&ptable.lock);
1770
1771   // Allocate kernel stack if necessary.
1772   if((p->kstack = kalloc(KSTACKSIZE)) == 0){
1773     p->state = UNUSED;
1774     return 0;
1775   }
1776   sp = p->kstack + KSTACKSIZE;
1777
1778   // Leave room for trap frame.
1779   sp -= sizeof *p->tf;
1780   p->tf = (struct trapframe*)sp;
1781
1782   // Set up new context to start executing at forkret,
1783   // which returns to trapret (see below).
1784   sp -= 4;
1785   *(uint*)sp = (uint)trapret;
1786
1787   sp -= sizeof *p->context;
1788   p->context = (struct context*)sp;
1789   memset(p->context, 0, sizeof *p->context);
1790   p->context->eip = (uint)forkret;
1791   return p;
1792 }
1793
1794
1795
1796
1797
1798
1799
```

```
1800 // Set up first user process.
1801 void
1802 userinit(void)
1803 {
1804   struct proc *p;
1805   extern char _binary_initcode_start[], _binary_initcode_size[];
1806
1807   p = allocproc();
1808   initproc = p;
1809
1810   // Initialize memory from initcode.S
1811   p->sz = PAGE;
1812   p->mem = kalloc(p->sz);
1813   memset(p->mem, 0, p->sz);
1814   memmove(p->mem, _binary_initcode_start, (int)_binary_initcode_size);
1815
1816   memset(p->tf, 0, sizeof(*p->tf));
1817   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
1818   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
1819   p->tf->es = p->tf->ds;
1820   p->tf->ss = p->tf->ds;
1821   p->tf->eflags = FL_IF;
1822   p->tf->esp = p->sz;
1823   p->tf->eip = 0;  // beginning of initcode.S
1824
1825   safestrcpy(p->name, "initcode", sizeof(p->name));
1826   p->cwd = namei("/");
1827
1828   p->state = RUNNABLE;
1829 }
1830
1831 // Grow current process's memory by n bytes.
1832 // Return 0 on success, -1 on failure.
1833 int
1834 growproc(int n)
1835 {
1836   char *newmem;
1837
1838   newmem = kalloc(cp->sz + n);
1839   if(newmem == 0)
1840     return -1;
1841   memmove(newmem, cp->mem, cp->sz);
1842   memset(newmem + cp->sz, 0, n);
1843   kfree(cp->mem, cp->sz);
1844   cp->mem = newmem;
1845   cp->sz += n;
1846   usegment();
1847   return 0;
1848 }
1849
```



```
1850 // Create a new process copying p as the parent.
1851 // Sets up stack to return as if from system call.
1852 // Caller must set state of returned proc to RUNNABLE.
1853 int
1854 fork(void)
1855 {
1856   int i, pid;
1857   struct proc *np;
1858
1859   // Allocate process.
1860   if((np = allocproc()) == 0)
1861     return -1;
1862
1863   // Copy process state from p.
1864   np->sz = cp->sz;
1865   if((np->mem = kalloc(np->sz)) == 0){
1866     kfree(np->kstack, KSTACKSIZE);
1867     np->kstack = 0;
1868     np->state = UNUSED;
1869     return -1;
1870   }
1871   memmove(np->mem, cp->mem, np->sz);
1872   np->parent = cp;
1873   *np->tf = *cp->tf;
1874
1875   // Clear %eax so that fork returns 0 in the child.
1876   np->tf->eax = 0;
1877
1878   for(i = 0; i < NOFILE; i++)
1879     if(cp->ofile[i])
1880       np->ofile[i] = filedup(cp->ofile[i]);
1881   np->cwd = idup(cp->cwd);
1882
1883   pid = np->pid;
1884   np->state = RUNNABLE;
1885
1886   return pid;
1887 }
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
```

```
1900 // Per-CPU process scheduler.
1901 // Each CPU calls scheduler() after setting itself up.
1902 // Scheduler never returns.  It loops, doing:
1903 //  - choose a process to run
1904 //  - swtch to start running that process
1905 //  - eventually that process transfers control
1906 //      via swtch back to the scheduler.
1907 void
1908 scheduler(void)
1909 {
1910   struct proc *p;
1911
1912   for(;;){
1913     // Enable interrupts on this processor.
1914     sti();
1915
1916     // Loop over process table looking for process to run.
1917     acquire(&ptable.lock);
1918     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
1919       if(p->state != RUNNABLE)
1920         continue;
1921
1922       // Switch to chosen process.  It is the process's job
1923       // to release ptable.lock and then reacquire it
1924       // before jumping back to us.
1925       cp = p;
1926       usegment();
1927       p->state = RUNNING;
1928       swtch(&c->context, p->context);
1929
1930       // Process is done running for now.
1931       // It should have changed its p->state before coming back.
1932       cp = 0;
1933     }
1934     release(&ptable.lock);
1935
1936   }
1937 }
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
```



```
1950 // Enter scheduler.  Must hold only ptable.lock
1951 // and have changed cp->state.
1952 void
1953 sched(void)
1954 {
1955   int intena;
1956
1957   if(!holding(&ptable.lock))
1958     panic("sched ptable.lock");
1959   if(c->ncli != 1)
1960     panic("sched locks");
1961   if(cp->state == RUNNING)
1962     panic("sched running");
1963   if(readeflags()&FL_IF)
1964     panic("sched interruptible");
1965
1966   intena = c->intena;
1967   swtch(&cp->context, c->context);
1968   c->intena = intena;
1969 }
1970
1971 // Give up the CPU for one scheduling round.
1972 void
1973 yield(void)
1974 {
1975   acquire(&ptable.lock);
1976   cp->state = RUNNABLE;
1977   sched();
1978   release(&ptable.lock);
1979 }
1980
1981 // A fork child's very first scheduling by scheduler()
1982 // will swtch here.  "Return" to user space.
1983 void
1984 forkret(void)
1985 {
1986   // Still holding ptable.lock from scheduler.
1987   release(&ptable.lock);
1988
1989   // Return to "caller", actually trapret (see allocproc).
1990 }
1991
1992
1993
1994
1995
1996
1997
1998
1999
```

```
2000 // Atomically release lock and sleep on chan.
2001 // Reacquires lock when reawakened.
2002 void
2003 sleep(void *chan, struct spinlock *lk)
2004 {
2005   if(cp == 0)
2006     panic("sleep");
2007
2008   if(lk == 0)
2009     panic("sleep without lk");
2010
2011   // Must acquire ptable.lock in order to
2012   // change p->state and then call sched.
2013   // Once we hold ptable.lock, we can be
2014   // guaranteed that we won't miss any wakeup
2015   // (wakeup runs with ptable.lock locked),
2016   // so it's okay to release lk.
2017   if(lk != &ptable.lock){
2018     acquire(&ptable.lock);
2019     release(lk);
2020   }
2021
2022   // Go to sleep.
2023   cp->chan = chan;
2024   cp->state = SLEEPING;
2025   sched();
2026
2027   // Tidy up.
2028   cp->chan = 0;
2029
2030   // Reacquire original lock.
2031   if(lk != &ptable.lock){
2032     release(&ptable.lock);
2033     acquire(lk);
2034   }
2035 }
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
```



```
2050 // Wake up all processes sleeping on chan.
2051 // The ptable lock must be held.
2052 static void
2053 wakeup1(void *chan)
2054 {
2055   struct proc *p;
2056
2057   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2058     if(p->state == SLEEPING && p->chan == chan)
2059       p->state = RUNNABLE;
2060 }
2061
2062 // Wake up all processes sleeping on chan.
2063 void
2064 wakeup(void *chan)
2065 {
2066   acquire(&ptable.lock);
2067   wakeup1(chan);
2068   release(&ptable.lock);
2069 }
2070
2071 // Kill the process with the given pid.
2072 // Process won't actually exit until it returns
2073 // to user space (see trap in trap.c).
2074 int
2075 kill(int pid)
2076 {
2077   struct proc *p;
2078
2079   acquire(&ptable.lock);
2080   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2081     if(p->pid == pid){
2082       p->killed = 1;
2083       // Wake process from sleep if necessary.
2084       if(p->state == SLEEPING)
2085         p->state = RUNNABLE;
2086       release(&ptable.lock);
2087       return 0;
2088     }
2089   }
2090   release(&ptable.lock);
2091   return -1;
2092 }
2093
2094
2095
2096
2097
2098
2099
```

```
2100 // Exit the current process.  Does not return.
2101 // Exited processes remain in the zombie state
2102 // until their parent calls wait() to find out they exited.
2103 void
2104 exit(void)
2105 {
2106   struct proc *p;
2107   int fd;
2108
2109   if(cp == initproc)
2110     panic("init exiting");
2111
2112   // Close all open files.
2113   for(fd = 0; fd < NOFILE; fd++){
2114     if(cp->ofile[fd]){
2115       fileclose(cp->ofile[fd]);
2116       cp->ofile[fd] = 0;
2117     }
2118   }
2119
2120   iput(cp->cwd);
2121   cp->cwd = 0;
2122
2123   acquire(&ptable.lock);
2124
2125   // Parent might be sleeping in wait().
2126   wakeup1(cp->parent);
2127
2128   // Pass abandoned children to init.
2129   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2130     if(p->parent == cp){
2131       p->parent = initproc;
2132       if(p->state == ZOMBIE)
2133         wakeup1(initproc);
2134     }
2135   }
2136
2137   // Jump into the scheduler, never to return.
2138   cp->state = ZOMBIE;
2139   sched();
2140   panic("zombie exit");
2141 }
2142
2143
2144
2145
2146
2147
2148
2149
```



```
2150 // Wait for a child process to exit and return its pid.
2151 // Return -1 if this process has no children.
2152 int
2153 wait(void)
2154 {
2155   struct proc *p;
2156   int havekids, pid;
2157
2158   acquire(&ptable.lock);
2159   for(;;){
2160     // Scan through table looking for zombie children.
2161     havekids = 0;
2162     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2163       if(p->parent != cp)
2164         continue;
2165       havekids = 1;
2166       if(p->state == ZOMBIE){
2167         // Found one.
2168         pid = p->pid;
2169         kfree(p->mem, p->sz);
2170         kfree(p->kstack, KSTACKSIZE);
2171         p->state = UNUSED;
2172         p->pid = 0;
2173         p->parent = 0;
2174         p->name[0] = 0;
2175         p->killed = 0;
2176         release(&ptable.lock);
2177         return pid;
2178       }
2179     }
2180
2181     // No point waiting if we don't have any children.
2182     if(!havekids || cp->killed){
2183       release(&ptable.lock);
2184       return -1;
2185     }
2186
2187     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2188     sleep(cp, &ptable.lock);
2189   }
2190 }
2191
2192
2193
2194
2195
2196
2197
2198
2199
```

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```



```
// Physical memory allocator, intended to allocate
// memory for user processes. Allocates in 4096-byte "pages".
// Free list is kept sorted and combines adjacent pages into
// long runs, to make it easier to allocate big segments.
// One reason the page size is 4k is that the x86 segment size
// granularity is 4k.

#include "types.h"
#include "defs.h"
#include "param.h"
#include "spinlock.h"

struct run {
  struct run *next;
  int len; // bytes
};

struct {
  struct spinlock lock;
  struct run *freelist;
} kmem;

// Initialize free list of physical pages.
// This code cheats by just considering one megabyte of
// pages after end.  Real systems would determine the
// amount of memory available in the system and use it all.
void
kinit(void)
{
  extern char end[];
  uint len;
  char *p;

  initlock(&kmem.lock, "kmem");
  p = (char*)(((uint)end + PAGE) & ~(PAGE-1));
  len = 256*PAGE; // assume computer has 256 pages of RAM, 1 MB
  cprintf("mem = %d\n", len);
  kfree(p, len);
}
```

```
2300 // Free the len bytes of memory pointed at by v,
2301 // which normally should have been returned by a
2302 // call to kalloc(len).  (The exception is when
2303 // initializing the allocator; see kinit above.)
2304 void
2305 kfree(char *v, int len)
2306 {
2307   struct run *r, *rend, **rp, *p, *pend;
2308
2309   if(len <= 0 || len % PAGE)
2310     panic("kfree");
2311
2312   // Fill with junk to catch dangling refs.
2313   memset(v, 1, len);
2314
2315   acquire(&kmem.lock);
2316   p = (struct run*)v;
2317   pend = (struct run*)(v + len);
2318   for(rp=&kmem.freelist; (r=*rp) != 0 && r <= pend; rp=&r->next){
2319     rend = (struct run*)((char*)r + r->len);
2320     if(r <= p && p < rend)
2321       panic("freeing free page");
2322     if(rend == p){  // r before p: expand r to include p
2323       r->len += len;
2324       if(r->next && r->next == pend){  // r now next to r->next?
2325         r->len += r->next->len;
2326         r->next = r->next->next;
2327       }
2328       goto out;
2329     }
2330     if(pend == r){  // p before r: expand p to include, replace r
2331       p->len = len + r->len;
2332       p->next = r->next;
2333       *rp = p;
2334       goto out;
2335     }
2336   }
2337   // Insert p before r in list.
2338   p->len = len;
2339   p->next = r;
2340   *rp = p;
2341
2342  out:
2343   release(&kmem.lock);
2344 }
2345
2346
2347
2348
2349
```



```
2350 // Allocate n bytes of physical memory.
2351 // Returns a kernel-segment pointer.
2352 // Returns 0 if the memory cannot be allocated.
2353 char*
2354 kalloc(int n)
2355 {
2356   char *p;
2357   struct run *r, **rp;
2358
2359   if(n % PAGE || n <= 0)
2360     panic("kalloc");
2361
2362   acquire(&kmem.lock);
2363   for(rp=&kmem.freelist; (r=*rp) != 0; rp=&r->next){
2364     if(r->len >= n){
2365       r->len -= n;
2366       p = (char*)r + r->len;
2367       if(r->len == 0)
2368         *rp = r->next;
2369       release(&kmem.lock);
2370       return p;
2371     }
2372   }
2373   release(&kmem.lock);
2374
2375   cprintf("kalloc: out of memory\n");
2376   return 0;
2377 }
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
```

```
2400 // x86 trap and interrupt constants.
2401
2402 // Processor-defined:
2403 #define T_DIVIDE         0      // divide error
2404 #define T_DEBUG          1      // debug exception
2405 #define T_NMI            2      // non-maskable interrupt
2406 #define T_BRKPT          3      // breakpoint
2407 #define T_OFLOW          4      // overflow
2408 #define T_BOUND          5      // bounds check
2409 #define T_ILLOP          6      // illegal opcode
2410 #define T_DEVICE         7      // device not available
2411 #define T_DBLFLT         8      // double fault
2412 // #define T_COPROC      9      // reserved (not used since 486)
2413 #define T_TSS           10      // invalid task switch segment
2414 #define T_SEGNP         11      // segment not present
2415 #define T_STACK         12      // stack exception
2416 #define T_GPFLT         13      // general protection fault
2417 #define T_PGFLT         14      // page fault
2418 // #define T_RES        15      // reserved
2419 #define T_FPERR         16      // floating point error
2420 #define T_ALIGN         17      // aligment check
2421 #define T_MCHK          18      // machine check
2422 #define T_SIMDERR       19      // SIMD floating point error
2423
2424 // These are arbitrarily chosen, but with care not to overlap
2425 // processor defined exceptions or interrupt vectors.
2426 #define T_SYSCALL       64     // system call
2427 #define T_DEFAULT      500      // catchall
2428
2429 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
2430
2431 #define IRQ_TIMER        0
2432 #define IRQ_KBD          1
2433 #define IRQ_COM1         4
2434 #define IRQ_IDE         14
2435 #define IRQ_ERROR       19
2436 #define IRQ_SPURIOUS    31
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
```



```
2450 #!/usr/bin/perl -w
2451
2452 # Generate vectors.S, the trap/interrupt entry points.
2453 # There has to be one entry point per interrupt number
2454 # since otherwise there's no way for trap() to discover
2455 # the interrupt number.
2456
2457 print "# generated by vectors.pl - do not edit\n";
2458 print "# handlers\n";
2459 print ".globl alltraps\n";
2460 for(my $i = 0; $i < 256; $i++){
2461     print ".globl vector$i\n";
2462     print "vector$i:\n";
2463     if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
2464         print "  pushl \$0\n";
2465     }
2466     print "  pushl \$$i\n";
2467     print "  jmp alltraps\n";
2468 }
2469
2470 print "\n# vector table\n";
2471 print ".data\n";
2472 print ".globl vectors\n";
2473 print "vectors:\n";
2474 for(my $i = 0; $i < 256; $i++){
2475     print "  .long vector$i\n";
2476 }
2477
2478 # sample output:
2479 #   # handlers
2480 #   .globl alltraps
2481 #   .globl vector0
2482 #   vector0:
2483 #     pushl $0
2484 #     pushl $0
2485 #     jmp alltraps
2486 #   ...
2487 #
2488 #   # vector table
2489 #   .data
2490 #   .globl vectors
2491 #   vectors:
2492 #     .long vector0
2493 #     .long vector1
2494 #     .long vector2
2495 #   ...
2496
2497
2498
2499
```

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

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

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

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

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



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

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

void
tvinit(void)
{
  int i;

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

  initlock(&tickslock, "time");
}

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

```
2600 void
2601 trap(struct trapframe *tf)
2602 {
2603   if(tf->trapno == T_SYSCALL){
2604     if(cp->killed)
2605       exit();
2606     cp->tf = tf;
2607     syscall();
2608     if(cp->killed)
2609       exit();
2610     return;
2611   }
2612
2613   switch(tf->trapno){
2614   case T_IRQ0 + IRQ_TIMER:
2615     if(cpu() == 0){
2616       acquire(&tickslock);
2617       ticks++;
2618       wakeup(&ticks);
2619       release(&tickslock);
2620     }
2621     lapiceoi();
2622     break;
2623   case T_IRQ0 + IRQ_IDE:
2624     ideintr();
2625     lapiceoi();
2626     break;
2627   case T_IRQ0 + IRQ_KBD:
2628     kbdintr();
2629     lapiceoi();
2630     break;
2631   case T_IRQ0 + IRQ_COM1:
2632     uartintr();
2633     lapiceoi();
2634     break;
2635   case T_IRQ0 + 7:
2636   case T_IRQ0 + IRQ_SPURIOUS:
2637     cprintf("cpu%d: spurious interrupt at %x:%x\n",
2638             cpu(), tf->cs, tf->eip);
2639     lapiceoi();
2640     break;
2641
2642   default:
2643     if(cp == 0 || (tf->cs&3) == 0){
2644       // In kernel, it must be our mistake.
2645       cprintf("unexpected trap %d from cpu %d eip %x\n",
2646               tf->trapno, cpu(), tf->eip);
2647       panic("trap");
2648     }
2649     // In user space, assume process misbehaved.
```



```
2650     cprintf("pid %d %s: trap %d err %d on cpu %d eip %x -- kill proc\n",
2651             cp->pid, cp->name, tf->trapno, tf->err, cpu(), tf->eip);
2652     cp->killed = 1;
2653   }
2654
2655   // Force process exit if it has been killed and is in user space.
2656   // (If it is still executing in the kernel, let it keep running
2657   // until it gets to the regular system call return.)
2658   if(cp && cp->killed && (tf->cs&3) == DPL_USER)
2659     exit();
2660
2661   // Force process to give up CPU on clock tick.
2662   // If interrupts were on while locks held, would need to check nlock.
2663   if(cp && cp->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
2664     yield();
2665
2666   // Check if the process has been killed since we yielded
2667   if(cp && cp->killed && (tf->cs&3) == DPL_USER)
2668     exit();
2669 }
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
```

```
2700 // System call numbers
2701 #define SYS_fork    1
2702 #define SYS_exit    2
2703 #define SYS_wait    3
2704 #define SYS_pipe    4
2705 #define SYS_write   5
2706 #define SYS_read    6
2707 #define SYS_close   7
2708 #define SYS_kill    8
2709 #define SYS_exec    9
2710 #define SYS_open   10
2711 #define SYS_mknod  11
2712 #define SYS_unlink 12
2713 #define SYS_fstat  13
2714 #define SYS_link   14
2715 #define SYS_mkdir  15
2716 #define SYS_chdir  16
2717 #define SYS_dup    17
2718 #define SYS_getpid 18
2719 #define SYS_sbrk   19
2720 #define SYS_sleep  20
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
```



```
2750 #include "types.h"
2751 #include "defs.h"
2752 #include "param.h"
2753 #include "mmu.h"
2754 #include "proc.h"
2755 #include "x86.h"
2756 #include "syscall.h"
2757
2758 // User code makes a system call with INT T_SYSCALL.
2759 // System call number in %eax.
2760 // Arguments on the stack, from the user call to the C
2761 // library system call function. The saved user %esp points
2762 // to a saved program counter, and then the first argument.
2763
2764 // Fetch the int at addr from process p.
2765 int
2766 fetchint(struct proc *p, uint addr, int *ip)
2767 {
2768   if(addr >= p->sz || addr+4 > p->sz)
2769     return -1;
2770   *ip = *(int*)(p->mem + addr);
2771   return 0;
2772 }
2773
2774 // Fetch the nul-terminated string at addr from process p.
2775 // Doesn't actually copy the string - just sets *pp to point at it.
2776 // Returns length of string, not including nul.
2777 int
2778 fetchstr(struct proc *p, uint addr, char **pp)
2779 {
2780   char *s, *ep;
2781
2782   if(addr >= p->sz)
2783     return -1;
2784   *pp = p->mem + addr;
2785   ep = p->mem + p->sz;
2786   for(s = *pp; s < ep; s++)
2787     if(*s == 0)
2788       return s - *pp;
2789   return -1;
2790 }
2791
2792 // Fetch the nth 32-bit system call argument.
2793 int
2794 argint(int n, int *ip)
2795 {
2796   return fetchint(cp, cp->tf->esp + 4 + 4*n, ip);
2797 }
2798
2799
```

```
2800 // Fetch the nth word-sized system call argument as a pointer
2801 // to a block of memory of size n bytes.  Check that the pointer
2802 // lies within the process address space.
2803 int
2804 argptr(int n, char **pp, int size)
2805 {
2806   int i;
2807
2808   if(argint(n, &i) < 0)
2809     return -1;
2810   if((uint)i >= cp->sz || (uint)i+size >= cp->sz)
2811     return -1;
2812   *pp = cp->mem + i;
2813   return 0;
2814 }
2815
2816 // Fetch the nth word-sized system call argument as a string pointer.
2817 // Check that the pointer is valid and the string is nul-terminated.
2818 // (There is no shared writable memory, so the string can't change
2819 // between this check and being used by the kernel.)
2820 int
2821 argstr(int n, char **pp)
2822 {
2823   int addr;
2824   if(argint(n, &addr) < 0)
2825     return -1;
2826   return fetchstr(cp, addr, pp);
2827 }
2828
2829 extern int sys_chdir(void);
2830 extern int sys_close(void);
2831 extern int sys_dup(void);
2832 extern int sys_exec(void);
2833 extern int sys_exit(void);
2834 extern int sys_fork(void);
2835 extern int sys_fstat(void);
2836 extern int sys_getpid(void);
2837 extern int sys_kill(void);
2838 extern int sys_link(void);
2839 extern int sys_mkdir(void);
2840 extern int sys_mknod(void);
2841 extern int sys_open(void);
2842 extern int sys_pipe(void);
2843 extern int sys_read(void);
2844 extern int sys_sbrk(void);
2845 extern int sys_sleep(void);
2846 extern int sys_unlink(void);
2847 extern int sys_wait(void);
2848 extern int sys_write(void);
2849
```



```
2850 static int (*syscalls[])(void) = {
2851 [SYS_chdir]   sys_chdir,
2852 [SYS_close]   sys_close,
2853 [SYS_dup]     sys_dup,
2854 [SYS_exec]    sys_exec,
2855 [SYS_exit]    sys_exit,
2856 [SYS_fork]    sys_fork,
2857 [SYS_fstat]   sys_fstat,
2858 [SYS_getpid]  sys_getpid,
2859 [SYS_kill]    sys_kill,
2860 [SYS_link]    sys_link,
2861 [SYS_mkdir]   sys_mkdir,
2862 [SYS_mknod]   sys_mknod,
2863 [SYS_open]    sys_open,
2864 [SYS_pipe]    sys_pipe,
2865 [SYS_read]    sys_read,
2866 [SYS_sbrk]    sys_sbrk,
2867 [SYS_sleep]   sys_sleep,
2868 [SYS_unlink]  sys_unlink,
2869 [SYS_wait]    sys_wait,
2870 [SYS_write]   sys_write,
2871 };
2872
2873 void
2874 syscall(void)
2875 {
2876   int num;
2877
2878   num = cp->tf->eax;
2879   if(num >= 0 && num < NELEM(syscalls) && syscalls[num])
2880     cp->tf->eax = syscalls[num]();
2881   else {
2882     cprintf("%d %s: unknown sys call %d\n",
2883             cp->pid, cp->name, num);
2884     cp->tf->eax = -1;
2885   }
2886 }
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
```

```
2900 #include "types.h"
2901 #include "x86.h"
2902 #include "defs.h"
2903 #include "param.h"
2904 #include "mmu.h"
2905 #include "proc.h"
2906
2907 int
2908 sys_fork(void)
2909 {
2910   return fork();
2911 }
2912
2913 int
2914 sys_exit(void)
2915 {
2916   exit();
2917   return 0;  // not reached
2918 }
2919
2920 int
2921 sys_wait(void)
2922 {
2923   return wait();
2924 }
2925
2926 int
2927 sys_kill(void)
2928 {
2929   int pid;
2930
2931   if(argint(0, &pid) < 0)
2932     return -1;
2933   return kill(pid);
2934 }
2935
2936 int
2937 sys_getpid(void)
2938 {
2939   return cp->pid;
2940 }
2941
2942
2943
2944
2945
2946
2947
2948
2949
```



```
2950 int
2951 sys_sbrk(void)
2952 {
2953   int addr;
2954   int n;
2955
2956   if(argint(0, &n) < 0)
2957     return -1;
2958   addr = cp->sz;
2959   if(growproc(n) < 0)
2960     return -1;
2961   return addr;
2962 }
2963
2964 int
2965 sys_sleep(void)
2966 {
2967   int n, ticks0;
2968
2969   if(argint(0, &n) < 0)
2970     return -1;
2971   acquire(&tickslock);
2972   ticks0 = ticks;
2973   while(ticks - ticks0 < n){
2974     if(cp->killed){
2975       release(&tickslock);
2976       return -1;
2977     }
2978     sleep(&ticks, &tickslock);
2979   }
2980   release(&tickslock);
2981   return 0;
2982 }
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
```

```
3000 struct buf {
3001   int flags;
3002   uint dev;
3003   uint sector;
3004   struct buf *prev; // LRU cache list
3005   struct buf *next;
3006   struct buf *qnext; // disk queue
3007   uchar data[512];
3008 };
3009 #define B_BUSY  0x1  // buffer is locked by some process
3010 #define B_VALID 0x2  // buffer has been read from disk
3011 #define B_DIRTY 0x4  // buffer needs to be written to disk
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
```



```
3050 #define O_RDONLY  0x000
3051 #define O_WRONLY  0x001
3052 #define O_RDWR    0x002
3053 #define O_CREATE  0x200
3054
3055
3056
3057
3058
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
```

```
3100 #define T_DIR  1   // Directory
3101 #define T_FILE 2   // File
3102 #define T_DEV  3   // Special device
3103
3104 struct stat {
3105   short type;  // Type of file
3106   int dev;     // Device number
3107   uint ino;    // Inode number on device
3108   short nlink; // Number of links to file
3109   uint size;   // Size of file in bytes
3110 };
3111
3112
3113
3114
3115
3116
3117
3118
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
```



```
3150 // On-disk file system format.
3151 // Both the kernel and user programs use this header file.
3152
3153 // Block 0 is unused.
3154 // Block 1 is super block.
3155 // Inodes start at block 2.
3156
3157 #define ROOTINO 1  // root i-number
3158 #define BSIZE 512  // block size
3159
3160 // File system super block
3161 struct superblock {
3162   uint size;         // Size of file system image (blocks)
3163   uint nblocks;      // Number of data blocks
3164   uint ninodes;      // Number of inodes.
3165 };
3166
3167 #define NDIRECT 12
3168 #define NINDIRECT (BSIZE / sizeof(uint))
3169 #define MAXFILE (NDIRECT + NINDIRECT)
3170
3171 // On-disk inode structure
3172 struct dinode {
3173   short type;           // File type
3174   short major;          // Major device number (T_DEV only)
3175   short minor;          // Minor device number (T_DEV only)
3176   short nlink;          // Number of links to inode in file system
3177   uint size;            // Size of file (bytes)
3178   uint addrs[NDIRECT+1];   // Data block addresses
3179 };
3180
3181 // Inodes per block.
3182 #define IPB           (BSIZE / sizeof(struct dinode))
3183
3184 // Block containing inode i
3185 #define IBLOCK(i)     ((i) / IPB + 2)
3186
3187 // Bitmap bits per block
3188 #define BPB           (BSIZE*8)
3189
3190 // Block containing bit for block b
3191 #define BBLOCK(b, ninodes) (b/BPB + (ninodes)/IPB + 3)
3192
3193
3194
3195
3196
3197
3198
3199
```

```
3200 // Directory is a file containing a sequence of dirent structures.
3201 #define DIRSIZ 14
3202
3203 struct dirent {
3204   ushort inum;
3205   char name[DIRSIZ];
3206 };
3207
3208
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
```



```
3250 struct file {
3251   enum { FD_NONE, FD_PIPE, FD_INODE } type;
3252   int ref; // reference count
3253   char readable;
3254   char writable;
3255   struct pipe *pipe;
3256   struct inode *ip;
3257   uint off;
3258 };
3259
3260
3261 // in-core file system types
3262
3263 struct inode {
3264   uint dev;           // Device number
3265   uint inum;          // Inode number
3266   int ref;            // Reference count
3267   int flags;          // I_BUSY, I_VALID
3268
3269   short type;         // copy of disk inode
3270   short major;
3271   short minor;
3272   short nlink;
3273   uint size;
3274   uint addrs[NDIRECT+1];
3275 };
3276
3277 #define I_BUSY 0x1
3278 #define I_VALID 0x2
3279
3280
3281 // device implementations
3282
3283 struct devsw {
3284   int (*read)(struct inode*, char*, int);
3285   int (*write)(struct inode*, char*, int);
3286 };
3287
3288 extern struct devsw devsw[];
3289
3290 #define CONSOLE 1
3291
3292
3293
3294
3295
3296
3297
3298
3299
```

```
3300 // Simple PIO-based (non-DMA) IDE driver code.
3301
3302 #include "types.h"
3303 #include "defs.h"
3304 #include "param.h"
3305 #include "mmu.h"
3306 #include "proc.h"
3307 #include "x86.h"
3308 #include "traps.h"
3309 #include "spinlock.h"
3310 #include "buf.h"
3311
3312 #define IDE_BSY       0x80
3313 #define IDE_DRDY      0x40
3314 #define IDE_DF        0x20
3315 #define IDE_ERR       0x01
3316
3317 #define IDE_CMD_READ  0x20
3318 #define IDE_CMD_WRITE 0x30
3319
3320 // idequeue points to the buf now being read/written to the disk.
3321 // idequeue->qnext points to the next buf to be processed.
3322 // You must hold idelock while manipulating queue.
3323
3324 static struct spinlock idelock;
3325 static struct buf *idequeue;
3326
3327 static int havedisk1;
3328 static void idestart(struct buf*);
3329
3330 // Wait for IDE disk to become ready.
3331 static int
3332 idewait(int checkerr)
3333 {
3334   int r;
3335
3336   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
3337     ;
3338   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
3339     return -1;
3340   return 0;
3341 }
3342
3343
3344
3345
3346
3347
3348
3349
```



```
3350 void
3351 ideinit(void)
3352 {
3353   int i;
3354
3355   initlock(&idelock, "ide");
3356   picenable(IRQ_IDE);
3357   ioapicenable(IRQ_IDE, ncpu - 1);
3358   idewait(0);
3359
3360   // Check if disk 1 is present
3361   outb(0x1f6, 0xe0 | (1<<4));
3362   for(i=0; i<1000; i++){
3363     if(inb(0x1f7) != 0){
3364       havedisk1 = 1;
3365       break;
3366     }
3367   }
3368
3369   // Switch back to disk 0.
3370   outb(0x1f6, 0xe0 | (0<<4));
3371 }
3372
3373 // Start the request for b.  Caller must hold idelock.
3374 static void
3375 idestart(struct buf *b)
3376 {
3377   if(b == 0)
3378     panic("idestart");
3379
3380   idewait(0);
3381   outb(0x3f6, 0);  // generate interrupt
3382   outb(0x1f2, 1);  // number of sectors
3383   outb(0x1f3, b->sector & 0xff);
3384   outb(0x1f4, (b->sector >> 8) & 0xff);
3385   outb(0x1f5, (b->sector >> 16) & 0xff);
3386   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
3387   if(b->flags & B_DIRTY){
3388     outb(0x1f7, IDE_CMD_WRITE);
3389     outsl(0x1f0, b->data, 512/4);
3390   } else {
3391     outb(0x1f7, IDE_CMD_READ);
3392   }
3393 }
3394
3395
3396
3397
3398
3399
```

```
3400 // Interrupt handler.
3401 void
3402 ideintr(void)
3403 {
3404   struct buf *b;
3405
3406   // Take first buffer off queue.
3407   acquire(&idelock);
3408   if((b = idequeue) == 0){
3409     release(&idelock);
3410     cprintf("Spurious IDE interrupt.\n");
3411     return;
3412   }
3413   idequeue = b->qnext;
3414
3415   // Read data if needed.
3416   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
3417     insl(0x1f0, b->data, 512/4);
3418
3419   // Wake process waiting for this buf.
3420   b->flags |= B_VALID;
3421   b->flags &= ~B_DIRTY;
3422   wakeup(b);
3423
3424   // Start disk on next buf in queue.
3425   if(idequeue != 0)
3426     idestart(idequeue);
3427
3428   release(&idelock);
3429 }
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
```



```
3450 // Sync buf with disk.
3451 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
3452 // Else if B_VALID is not set, read buf from disk, set B_VALID.
3453 void
3454 iderw(struct buf *b)
3455 {
3456   struct buf **pp;
3457
3458   if(!(b->flags & B_BUSY))
3459     panic("iderw: buf not busy");
3460   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
3461     panic("iderw: nothing to do");
3462   if(b->dev != 0 && !havedisk1)
3463     panic("idrw: ide disk 1 not present");
3464
3465   acquire(&idelock);
3466
3467   // Append b to idequeue.
3468   b->qnext = 0;
3469   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
3470     ;
3471   *pp = b;
3472
3473   // Start disk if necessary.
3474   if(idequeue == b)
3475     idestart(b);
3476
3477   // Wait for request to finish.
3478   // Assuming will not sleep too long: ignore cp->killed.
3479   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
3480     sleep(b, &idelock);
3481
3482   release(&idelock);
3483 }
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
```

```
3500 // Buffer cache.
3501 //
3502 // The buffer cache is a linked list of buf structures holding
3503 // cached copies of disk block contents.  Caching disk blocks
3504 // in memory reduces the number of disk reads and also provides
3505 // a synchronization point for disk blocks used by multiple processes.
3506 //
3507 // Interface:
3508 // * To get a buffer for a particular disk block, call bread.
3509 // * After changing buffer data, call bwrite to flush it to disk.
3510 // * When done with the buffer, call brelse.
3511 // * Do not use the buffer after calling brelse.
3512 // * Only one process at a time can use a buffer,
3513 //     so do not keep them longer than necessary.
3514 //
3515 // The implementation uses three state flags internally:
3516 // * B_BUSY: the block has been returned from bread
3517 //     and has not been passed back to brelse.
3518 // * B_VALID: the buffer data has been initialized
3519 //     with the associated disk block contents.
3520 // * B_DIRTY: the buffer data has been modified
3521 //     and needs to be written to disk.
3522
3523 #include "types.h"
3524 #include "defs.h"
3525 #include "param.h"
3526 #include "spinlock.h"
3527 #include "buf.h"
3528
3529 struct {
3530   struct spinlock lock;
3531   struct buf buf[NBUF];
3532
3533   // Linked list of all buffers, through prev/next.
3534   // head.next is most recently used.
3535   struct buf head;
3536 } bcache;
3537
3538 void
3539 binit(void)
3540 {
3541   struct buf *b;
3542
3543   initlock(&bcache.lock, "bcache");
3544
3545
3546
3547
3548
3549
```



```
3550   // Create linked list of buffers
3551   bcache.head.prev = &bcache.head;
3552   bcache.head.next = &bcache.head;
3553   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
3554     b->next = bcache.head.next;
3555     b->prev = &bcache.head;
3556     b->dev = -1;
3557     bcache.head.next->prev = b;
3558     bcache.head.next = b;
3559   }
3560 }
3561
3562 // Look through buffer cache for sector on device dev.
3563 // If not found, allocate fresh block.
3564 // In either case, return locked buffer.
3565 static struct buf*
3566 bget(uint dev, uint sector)
3567 {
3568   struct buf *b;
3569
3570   acquire(&bcache.lock);
3571
3572  loop:
3573   // Try for cached block.
3574   for(b = bcache.head.next; b != &bcache.head; b = b->next){
3575     if(b->dev == dev && b->sector == sector){
3576       if(!(b->flags & B_BUSY)){
3577         b->flags |= B_BUSY;
3578         release(&bcache.lock);
3579         return b;
3580       }
3581       sleep(b, &bcache.lock);
3582       goto loop;
3583     }
3584   }
3585
3586   // Allocate fresh block.
3587   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
3588     if((b->flags & B_BUSY) == 0){
3589       b->dev = dev;
3590       b->sector = sector;
3591       b->flags = B_BUSY;
3592       release(&bcache.lock);
3593       return b;
3594     }
3595   }
3596   panic("bget: no buffers");
3597 }
3598
3599
```

```
3600 // Return a B_BUSY buf with the contents of the indicated disk sector.
3601 struct buf*
3602 bread(uint dev, uint sector)
3603 {
3604   struct buf *b;
3605
3606   b = bget(dev, sector);
3607   if(!(b->flags & B_VALID))
3608     iderw(b);
3609   return b;
3610 }
3611
3612 // Write b's contents to disk.  Must be locked.
3613 void
3614 bwrite(struct buf *b)
3615 {
3616   if((b->flags & B_BUSY) == 0)
3617     panic("bwrite");
3618   b->flags |= B_DIRTY;
3619   iderw(b);
3620 }
3621
3622 // Release the buffer b.
3623 void
3624 brelse(struct buf *b)
3625 {
3626   if((b->flags & B_BUSY) == 0)
3627     panic("brelse");
3628
3629   acquire(&bcache.lock);
3630
3631   b->next->prev = b->prev;
3632   b->prev->next = b->next;
3633   b->next = bcache.head.next;
3634   b->prev = &bcache.head;
3635   bcache.head.next->prev = b;
3636   bcache.head.next = b;
3637
3638   b->flags &= ~B_BUSY;
3639   wakeup(b);
3640
3641   release(&bcache.lock);
3642 }
3643
3644
3645
3646
3647
3648
3649
```



```
3650 // File system implementation.  Four layers:
3651 //   + Blocks: allocator for raw disk blocks.
3652 //   + Files: inode allocator, reading, writing, metadata.
3653 //   + Directories: inode with special contents (list of other inodes!)
3654 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
3655 //
3656 // Disk layout is: superblock, inodes, block in-use bitmap, data blocks.
3657 //
3658 // This file contains the low-level file system manipulation
3659 // routines.  The (higher-level) system call implementations
3660 // are in sysfile.c.
3661
3662 #include "types.h"
3663 #include "defs.h"
3664 #include "param.h"
3665 #include "stat.h"
3666 #include "mmu.h"
3667 #include "proc.h"
3668 #include "spinlock.h"
3669 #include "buf.h"
3670 #include "fs.h"
3671 #include "file.h"
3672
3673 #define min(a, b) ((a) < (b) ? (a) : (b))
3674 static void itrunc(struct inode*);
3675
3676 // Read the super block.
3677 static void
3678 readsb(int dev, struct superblock *sb)
3679 {
3680   struct buf *bp;
3681
3682   bp = bread(dev, 1);
3683   memmove(sb, bp->data, sizeof(*sb));
3684   brelse(bp);
3685 }
3686
3687 // Zero a block.
3688 static void
3689 bzero(int dev, int bno)
3690 {
3691   struct buf *bp;
3692
3693   bp = bread(dev, bno);
3694   memset(bp->data, 0, BSIZE);
3695   bwrite(bp);
3696   brelse(bp);
3697 }
3698
3699
```

```
3700 // Blocks.
3701
3702 // Allocate a disk block.
3703 static uint
3704 balloc(uint dev)
3705 {
3706   int b, bi, m;
3707   struct buf *bp;
3708   struct superblock sb;
3709
3710   bp = 0;
3711   readsb(dev, &sb);
3712   for(b = 0; b < sb.size; b += BPB){
3713     bp = bread(dev, BBLOCK(b, sb.ninodes));
3714     for(bi = 0; bi < BPB; bi++){
3715       m = 1 << (bi % 8);
3716       if((bp->data[bi/8] & m) == 0){  // Is block free?
3717         bp->data[bi/8] |= m;  // Mark block in use on disk.
3718         bwrite(bp);
3719         brelse(bp);
3720         return b + bi;
3721       }
3722     }
3723     brelse(bp);
3724   }
3725   panic("balloc: out of blocks");
3726 }
3727
3728 // Free a disk block.
3729 static void
3730 bfree(int dev, uint b)
3731 {
3732   struct buf *bp;
3733   struct superblock sb;
3734   int bi, m;
3735
3736   bzero(dev, b);
3737
3738   readsb(dev, &sb);
3739   bp = bread(dev, BBLOCK(b, sb.ninodes));
3740   bi = b % BPB;
3741   m = 1 << (bi % 8);
3742   if((bp->data[bi/8] & m) == 0)
3743     panic("freeing free block");
3744   bp->data[bi/8] &= ~m;  // Mark block free on disk.
3745   bwrite(bp);
3746   brelse(bp);
3747 }
3748
3749
```



```
3750 // Inodes.
3751 //
3752 // An inode is a single, unnamed file in the file system.
3753 // The inode disk structure holds metadata (the type, device numbers,
3754 // and data size) along with a list of blocks where the associated
3755 // data can be found.
3756 //
3757 // The inodes are laid out sequentially on disk immediately after
3758 // the superblock.  The kernel keeps a cache of the in-use
3759 // on-disk structures to provide a place for synchronizing access
3760 // to inodes shared between multiple processes.
3761 //
3762 // ip->ref counts the number of pointer references to this cached
3763 // inode; references are typically kept in struct file and in cp->cwd.
3764 // When ip->ref falls to zero, the inode is no longer cached.
3765 // It is an error to use an inode without holding a reference to it.
3766 //
3767 // Processes are only allowed to read and write inode
3768 // metadata and contents when holding the inode's lock,
3769 // represented by the I_BUSY flag in the in-memory copy.
3770 // Because inode locks are held during disk accesses,
3771 // they are implemented using a flag rather than with
3772 // spin locks.  Callers are responsible for locking
3773 // inodes before passing them to routines in this file; leaving
3774 // this responsibility with the caller makes it possible for them
3775 // to create arbitrarily-sized atomic operations.
3776 //
3777 // To give maximum control over locking to the callers,
3778 // the routines in this file that return inode pointers
3779 // return pointers to *unlocked* inodes.  It is the callers'
3780 // responsibility to lock them before using them.  A non-zero
3781 // ip->ref keeps these unlocked inodes in the cache.
3782
3783 struct {
3784   struct spinlock lock;
3785   struct inode inode[NINODE];
3786 } icache;
3787
3788 void
3789 iinit(void)
3790 {
3791   initlock(&icache.lock, "icache");
3792 }
3793
3794 static struct inode* iget(uint dev, uint inum);
3795
3796
3797
3798
3799
```

```
3800 // Allocate a new inode with the given type on device dev.
3801 struct inode*
3802 ialloc(uint dev, short type)
3803 {
3804   int inum;
3805   struct buf *bp;
3806   struct dinode *dip;
3807   struct superblock sb;
3808
3809   readsb(dev, &sb);
3810   for(inum = 1; inum < sb.ninodes; inum++){  // loop over inode blocks
3811     bp = bread(dev, IBLOCK(inum));
3812     dip = (struct dinode*)bp->data + inum%IPB;
3813     if(dip->type == 0){  // a free inode
3814       memset(dip, 0, sizeof(*dip));
3815       dip->type = type;
3816       bwrite(bp);   // mark it allocated on the disk
3817       brelse(bp);
3818       return iget(dev, inum);
3819     }
3820     brelse(bp);
3821   }
3822   panic("ialloc: no inodes");
3823 }
3824
3825 // Copy inode, which has changed, from memory to disk.
3826 void
3827 iupdate(struct inode *ip)
3828 {
3829   struct buf *bp;
3830   struct dinode *dip;
3831
3832   bp = bread(ip->dev, IBLOCK(ip->inum));
3833   dip = (struct dinode*)bp->data + ip->inum%IPB;
3834   dip->type = ip->type;
3835   dip->major = ip->major;
3836   dip->minor = ip->minor;
3837   dip->nlink = ip->nlink;
3838   dip->size = ip->size;
3839   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
3840   bwrite(bp);
3841   brelse(bp);
3842 }
3843
3844
3845
3846
3847
3848
3849
```



```
3850 // Find the inode with number inum on device dev
3851 // and return the in-memory copy.
3852 static struct inode*
3853 iget(uint dev, uint inum)
3854 {
3855   struct inode *ip, *empty;
3856
3857   acquire(&icache.lock);
3858
3859   // Try for cached inode.
3860   empty = 0;
3861   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
3862     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
3863       ip->ref++;
3864       release(&icache.lock);
3865       return ip;
3866     }
3867     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
3868       empty = ip;
3869   }
3870
3871   // Allocate fresh inode.
3872   if(empty == 0)
3873     panic("iget: no inodes");
3874
3875   ip = empty;
3876   ip->dev = dev;
3877   ip->inum = inum;
3878   ip->ref = 1;
3879   ip->flags = 0;
3880   release(&icache.lock);
3881
3882   return ip;
3883 }
3884
3885 // Increment reference count for ip.
3886 // Returns ip to enable ip = idup(ip1) idiom.
3887 struct inode*
3888 idup(struct inode *ip)
3889 {
3890   acquire(&icache.lock);
3891   ip->ref++;
3892   release(&icache.lock);
3893   return ip;
3894 }
3895
3896
3897
3898
3899
```

```
3900 // Lock the given inode.
3901 void
3902 ilock(struct inode *ip)
3903 {
3904   struct buf *bp;
3905   struct dinode *dip;
3906
3907   if(ip == 0 || ip->ref < 1)
3908     panic("ilock");
3909
3910   acquire(&icache.lock);
3911   while(ip->flags & I_BUSY)
3912     sleep(ip, &icache.lock);
3913   ip->flags |= I_BUSY;
3914   release(&icache.lock);
3915
3916   if(!(ip->flags & I_VALID)){
3917     bp = bread(ip->dev, IBLOCK(ip->inum));
3918     dip = (struct dinode*)bp->data + ip->inum%IPB;
3919     ip->type = dip->type;
3920     ip->major = dip->major;
3921     ip->minor = dip->minor;
3922     ip->nlink = dip->nlink;
3923     ip->size = dip->size;
3924     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
3925     brelse(bp);
3926     ip->flags |= I_VALID;
3927     if(ip->type == 0)
3928       panic("ilock: no type");
3929   }
3930 }
3931
3932 // Unlock the given inode.
3933 void
3934 iunlock(struct inode *ip)
3935 {
3936   if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
3937     panic("iunlock");
3938
3939   acquire(&icache.lock);
3940   ip->flags &= ~I_BUSY;
3941   wakeup(ip);
3942   release(&icache.lock);
3943 }
3944
3945
3946
3947
3948
3949
```



```
3950 // Caller holds reference to unlocked ip.  Drop reference.
3951 void
3952 iput(struct inode *ip)
3953 {
3954   acquire(&icache.lock);
3955   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
3956     // inode is no longer used: truncate and free inode.
3957     if(ip->flags & I_BUSY)
3958       panic("iput busy");
3959     ip->flags |= I_BUSY;
3960     release(&icache.lock);
3961     itrunc(ip);
3962     ip->type = 0;
3963     iupdate(ip);
3964     acquire(&icache.lock);
3965     ip->flags = 0;
3966     wakeup(ip);
3967   }
3968   ip->ref--;
3969   release(&icache.lock);
3970 }
3971
3972 // Common idiom: unlock, then put.
3973 void
3974 iunlockput(struct inode *ip)
3975 {
3976   iunlock(ip);
3977   iput(ip);
3978 }
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
```

```
4000 // Inode contents
4001 //
4002 // The contents (data) associated with each inode is stored
4003 // in a sequence of blocks on the disk.  The first NDIRECT blocks
4004 // are listed in ip->addrs[].  The next NINDIRECT blocks are
4005 // listed in the block ip->addrs[INDIRECT].
4006
4007 // Return the disk block address of the nth block in inode ip.
4008 // If there is no such block, bmap allocates one.
4009 static uint
4010 bmap(struct inode *ip, uint bn)
4011 {
4012   uint addr, *a;
4013   struct buf *bp;
4014
4015   if(bn < NDIRECT){
4016     if((addr = ip->addrs[bn]) == 0)
4017       ip->addrs[bn] = addr = balloc(ip->dev);
4018     return addr;
4019   }
4020   bn -= NDIRECT;
4021
4022   if(bn < NINDIRECT){
4023     // Load indirect block, allocating if necessary.
4024     if((addr = ip->addrs[NDIRECT]) == 0)
4025       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
4026     bp = bread(ip->dev, addr);
4027     a = (uint*)bp->data;
4028     if((addr = a[bn]) == 0){
4029       a[bn] = addr = balloc(ip->dev);
4030       bwrite(bp);
4031     }
4032     brelse(bp);
4033     return addr;
4034   }
4035
4036   panic("bmap: out of range");
4037 }
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
```



```
4050 // Truncate inode (discard contents).
4051 // Only called after the last dirent referring
4052 // to this inode has been erased on disk.
4053 static void
4054 itrunc(struct inode *ip)
4055 {
4056   int i, j;
4057   struct buf *bp;
4058   uint *a;
4059
4060   for(i = 0; i < NDIRECT; i++){
4061     if(ip->addrs[i]){
4062       bfree(ip->dev, ip->addrs[i]);
4063       ip->addrs[i] = 0;
4064     }
4065   }
4066
4067   if(ip->addrs[NDIRECT]){
4068     bp = bread(ip->dev, ip->addrs[NDIRECT]);
4069     a = (uint*)bp->data;
4070     for(j = 0; j < NINDIRECT; j++){
4071       if(a[j])
4072         bfree(ip->dev, a[j]);
4073     }
4074     brelse(bp);
4075     bfree(ip->dev, ip->addrs[NDIRECT]);
4076     ip->addrs[NDIRECT] = 0;
4077   }
4078
4079   ip->size = 0;
4080   iupdate(ip);
4081 }
4082
4083 // Copy stat information from inode.
4084 void
4085 stati(struct inode *ip, struct stat *st)
4086 {
4087   st->dev = ip->dev;
4088   st->ino = ip->inum;
4089   st->type = ip->type;
4090   st->nlink = ip->nlink;
4091   st->size = ip->size;
4092 }
4093
4094
4095
4096
4097
4098
4099
```

```
4100 // Read data from inode.
4101 int
4102 readi(struct inode *ip, char *dst, uint off, uint n)
4103 {
4104   uint tot, m;
4105   struct buf *bp;
4106
4107   if(ip->type == T_DEV){
4108     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
4109       return -1;
4110     return devsw[ip->major].read(ip, dst, n);
4111   }
4112
4113   if(off > ip->size || off + n < off)
4114     return -1;
4115   if(off + n > ip->size)
4116     n = ip->size - off;
4117
4118   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
4119     bp = bread(ip->dev, bmap(ip, off/BSIZE));
4120     m = min(n - tot, BSIZE - off%BSIZE);
4121     memmove(dst, bp->data + off%BSIZE, m);
4122     brelse(bp);
4123   }
4124   return n;
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 // Write data to inode.
4151 int
4152 writei(struct inode *ip, char *src, uint off, uint n)
4153 {
4154   uint tot, m;
4155   struct buf *bp;
4156
4157   if(ip->type == T_DEV){
4158     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
4159       return -1;
4160     return devsw[ip->major].write(ip, src, n);
4161   }
4162
4163   if(off > ip->size || off + n < off)
4164     return -1;
4165   if(off + n > MAXFILE*BSIZE)
4166     n = MAXFILE*BSIZE - off;
4167
4168   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
4169     bp = bread(ip->dev, bmap(ip, off/BSIZE));
4170     m = min(n - tot, BSIZE - off%BSIZE);
4171     memmove(bp->data + off%BSIZE, src, m);
4172     bwrite(bp);
4173     brelse(bp);
4174   }
4175
4176   if(n > 0 && off > ip->size){
4177     ip->size = off;
4178     iupdate(ip);
4179   }
4180   return n;
4181 }
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
```

```
4200 // Directories
4201
4202 int
4203 namecmp(const char *s, const char *t)
4204 {
4205   return strncmp(s, t, DIRSIZ);
4206 }
4207
4208 // Look for a directory entry in a directory.
4209 // If found, set *poff to byte offset of entry.
4210 // Caller must have already locked dp.
4211 struct inode*
4212 dirlookup(struct inode *dp, char *name, uint *poff)
4213 {
4214   uint off, inum;
4215   struct buf *bp;
4216   struct dirent *de;
4217
4218   if(dp->type != T_DIR)
4219     panic("dirlookup not DIR");
4220
4221   for(off = 0; off < dp->size; off += BSIZE){
4222     bp = bread(dp->dev, bmap(dp, off / BSIZE));
4223     for(de = (struct dirent*)bp->data;
4224         de < (struct dirent*)(bp->data + BSIZE);
4225         de++){
4226       if(de->inum == 0)
4227         continue;
4228       if(namecmp(name, de->name) == 0){
4229         // entry matches path element
4230         if(poff)
4231           *poff = off + (uchar*)de - bp->data;
4232         inum = de->inum;
4233         brelse(bp);
4234         return iget(dp->dev, inum);
4235       }
4236     }
4237     brelse(bp);
4238   }
4239   return 0;
4240 }
4241
4242
4243
4244
4245
4246
4247
4248
4249
```



```
4250 // Write a new directory entry (name, inum) into the directory dp.
4251 int
4252 dirlink(struct inode *dp, char *name, uint inum)
4253 {
4254   int off;
4255   struct dirent de;
4256   struct inode *ip;
4257
4258   // Check that name is not present.
4259   if((ip = dirlookup(dp, name, 0)) != 0){
4260     iput(ip);
4261     return -1;
4262   }
4263
4264   // Look for an empty dirent.
4265   for(off = 0; off < dp->size; off += sizeof(de)){
4266     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4267       panic("dirlink read");
4268     if(de.inum == 0)
4269       break;
4270   }
4271
4272   strncpy(de.name, name, DIRSIZ);
4273   de.inum = inum;
4274   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4275     panic("dirlink");
4276
4277   return 0;
4278 }
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
```

```
4300 // Paths
4301
4302 // Copy the next path element from path into name.
4303 // Return a pointer to the element following the copied one.
4304 // The returned path has no leading slashes,
4305 // so the caller can check *path=='\0' to see if the name is the last one.
4306 // If no name to remove, return 0.
4307 //
4308 // Examples:
4309 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
4310 //   skipelem("///a//bb", name) = "bb", setting name = "a"
4311 //   skipelem("a", name) = "", setting name = "a"
4312 //   skipelem("", name) = skipelem("////", name) = 0
4313 //
4314 static char*
4315 skipelem(char *path, char *name)
4316 {
4317   char *s;
4318   int len;
4319
4320   while(*path == '/')
4321     path++;
4322   if(*path == 0)
4323     return 0;
4324   s = path;
4325   while(*path != '/' && *path != 0)
4326     path++;
4327   len = path - s;
4328   if(len >= DIRSIZ)
4329     memmove(name, s, DIRSIZ);
4330   else {
4331     memmove(name, s, len);
4332     name[len] = 0;
4333   }
4334   while(*path == '/')
4335     path++;
4336   return path;
4337 }
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
```



```
4350 // Look up and return the inode for a path name.
4351 // If parent != 0, return the inode for the parent and copy the final
4352 // path element into name, which must have room for DIRSIZ bytes.
4353 static struct inode*
4354 namex(char *path, int nameiparent, char *name)
4355 {
4356   struct inode *ip, *next;
4357
4358   if(*path == '/')
4359     ip = iget(ROOTDEV, ROOTINO);
4360   else
4361     ip = idup(cp->cwd);
4362
4363   while((path = skipelem(path, name)) != 0){
4364     ilock(ip);
4365     if(ip->type != T_DIR){
4366       iunlockput(ip);
4367       return 0;
4368     }
4369     if(nameiparent && *path == '\0'){
4370       // Stop one level early.
4371       iunlock(ip);
4372       return ip;
4373     }
4374     if((next = dirlookup(ip, name, 0)) == 0){
4375       iunlockput(ip);
4376       return 0;
4377     }
4378     iunlockput(ip);
4379     ip = next;
4380   }
4381   if(nameiparent){
4382     iput(ip);
4383     return 0;
4384   }
4385   return ip;
4386 }
4387
4388 struct inode*
4389 namei(char *path)
4390 {
4391   char name[DIRSIZ];
4392   return namex(path, 0, name);
4393 }
4394
4395 struct inode*
4396 nameiparent(char *path, char *name)
4397 {
4398   return namex(path, 1, name);
4399 }
```

```
4400 #include "types.h"
4401 #include "defs.h"
4402 #include "param.h"
4403 #include "file.h"
4404 #include "spinlock.h"
4405
4406 struct devsw devsw[NDEV];
4407 struct {
4408   struct spinlock lock;
4409   struct file file[NFILE];
4410 } ftable;
4411
4412 void
4413 fileinit(void)
4414 {
4415   initlock(&ftable.lock, "ftable");
4416 }
4417
4418 // Allocate a file structure.
4419 struct file*
4420 filealloc(void)
4421 {
4422   struct file *f;
4423
4424   acquire(&ftable.lock);
4425   for(f = ftable.file; f < ftable.file + NFILE; f++){
4426     if(f->ref == 0){
4427       f->ref = 1;
4428       release(&ftable.lock);
4429       return f;
4430     }
4431   }
4432   release(&ftable.lock);
4433   return 0;
4434 }
4435
4436 // Increment ref count for file f.
4437 struct file*
4438 filedup(struct file *f)
4439 {
4440   acquire(&ftable.lock);
4441   if(f->ref < 1)
4442     panic("filedup");
4443   f->ref++;
4444   release(&ftable.lock);
4445   return f;
4446 }
4447
4448
4449
```



```
4450 // Close file f.  (Decrement ref count, close when reaches 0.)
4451 void
4452 fileclose(struct file *f)
4453 {
4454   struct file ff;
4455
4456   acquire(&ftable.lock);
4457   if(f->ref < 1)
4458     panic("fileclose");
4459   if(--f->ref > 0){
4460     release(&ftable.lock);
4461     return;
4462   }
4463   ff = *f;
4464   f->ref = 0;
4465   f->type = FD_NONE;
4466   release(&ftable.lock);
4467
4468   if(ff.type == FD_PIPE)
4469     pipeclose(ff.pipe, ff.writable);
4470   else if(ff.type == FD_INODE)
4471     iput(ff.ip);
4472 }
4473
4474 // Get metadata about file f.
4475 int
4476 filestat(struct file *f, struct stat *st)
4477 {
4478   if(f->type == FD_INODE){
4479     ilock(f->ip);
4480     stati(f->ip, st);
4481     iunlock(f->ip);
4482     return 0;
4483   }
4484   return -1;
4485 }
4486
4487
4488
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499
```

```
4500 // Read from file f.  Addr is kernel address.
4501 int
4502 fileread(struct file *f, char *addr, int n)
4503 {
4504   int r;
4505
4506   if(f->readable == 0)
4507     return -1;
4508   if(f->type == FD_PIPE)
4509     return piperead(f->pipe, addr, n);
4510   if(f->type == FD_INODE){
4511     ilock(f->ip);
4512     if((r = readi(f->ip, addr, f->off, n)) > 0)
4513       f->off += r;
4514     iunlock(f->ip);
4515     return r;
4516   }
4517   panic("fileread");
4518 }
4519
4520 // Write to file f.  Addr is kernel address.
4521 int
4522 filewrite(struct file *f, char *addr, int n)
4523 {
4524   int r;
4525
4526   if(f->writable == 0)
4527     return -1;
4528   if(f->type == FD_PIPE)
4529     return pipewrite(f->pipe, addr, n);
4530   if(f->type == FD_INODE){
4531     ilock(f->ip);
4532     if((r = writei(f->ip, addr, f->off, n)) > 0)
4533       f->off += r;
4534     iunlock(f->ip);
4535     return r;
4536   }
4537   panic("filewrite");
4538 }
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4549
```



```
4550 #include "types.h"
4551 #include "defs.h"
4552 #include "param.h"
4553 #include "stat.h"
4554 #include "mmu.h"
4555 #include "proc.h"
4556 #include "fs.h"
4557 #include "file.h"
4558 #include "fcntl.h"
4559
4560 // Fetch the nth word-sized system call argument as a file descriptor
4561 // and return both the descriptor and the corresponding struct file.
4562 static int
4563 argfd(int n, int *pfd, struct file **pf)
4564 {
4565   int fd;
4566   struct file *f;
4567
4568   if(argint(n, &fd) < 0)
4569     return -1;
4570   if(fd < 0 || fd >= NOFILE || (f=cp->ofile[fd]) == 0)
4571     return -1;
4572   if(pfd)
4573     *pfd = fd;
4574   if(pf)
4575     *pf = f;
4576   return 0;
4577 }
4578
4579 // Allocate a file descriptor for the given file.
4580 // Takes over file reference from caller on success.
4581 static int
4582 fdalloc(struct file *f)
4583 {
4584   int fd;
4585
4586   for(fd = 0; fd < NOFILE; fd++){
4587     if(cp->ofile[fd] == 0){
4588       cp->ofile[fd] = f;
4589       return fd;
4590     }
4591   }
4592   return -1;
4593 }
4594
4595
4596
4597
4598
4599
```

```
4600 int
4601 sys_dup(void)
4602 {
4603   struct file *f;
4604   int fd;
4605
4606   if(argfd(0, 0, &f) < 0)
4607     return -1;
4608   if((fd=fdalloc(f)) < 0)
4609     return -1;
4610   filedup(f);
4611   return fd;
4612 }
4613
4614 int
4615 sys_read(void)
4616 {
4617   struct file *f;
4618   int n;
4619   char *p;
4620
4621   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
4622     return -1;
4623   return fileread(f, p, n);
4624 }
4625
4626 int
4627 sys_write(void)
4628 {
4629   struct file *f;
4630   int n;
4631   char *p;
4632
4633   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
4634     return -1;
4635   return filewrite(f, p, n);
4636 }
4637
4638 int
4639 sys_close(void)
4640 {
4641   int fd;
4642   struct file *f;
4643
4644   if(argfd(0, &fd, &f) < 0)
4645     return -1;
4646   cp->ofile[fd] = 0;
4647   fileclose(f);
4648   return 0;
4649 }
```



```
4650 int
4651 sys_fstat(void)
4652 {
4653   struct file *f;
4654   struct stat *st;
4655
4656   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
4657     return -1;
4658   return filestat(f, st);
4659 }
4660
4661 // Create the path new as a link to the same inode as old.
4662 int
4663 sys_link(void)
4664 {
4665   char name[DIRSIZ], *new, *old;
4666   struct inode *dp, *ip;
4667
4668   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
4669     return -1;
4670   if((ip = namei(old)) == 0)
4671     return -1;
4672   ilock(ip);
4673   if(ip->type == T_DIR){
4674     iunlockput(ip);
4675     return -1;
4676   }
4677   ip->nlink++;
4678   iupdate(ip);
4679   iunlock(ip);
4680
4681   if((dp = nameiparent(new, name)) == 0)
4682     goto bad;
4683   ilock(dp);
4684   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
4685     iunlockput(dp);
4686     goto bad;
4687   }
4688   iunlockput(dp);
4689   iput(ip);
4690   return 0;
4691
4692 bad:
4693   ilock(ip);
4694   ip->nlink--;
4695   iupdate(ip);
4696   iunlockput(ip);
4697   return -1;
4698 }
4699
```

```
4700 // Is the directory dp empty except for "." and ".." ?
4701 static int
4702 isdirempty(struct inode *dp)
4703 {
4704   int off;
4705   struct dirent de;
4706
4707   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
4708     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4709       panic("isdirempty: readi");
4710     if(de.inum != 0)
4711       return 0;
4712   }
4713   return 1;
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 int
4751 sys_unlink(void)
4752 {
4753   struct inode *ip, *dp;
4754   struct dirent de;
4755   char name[DIRSIZ], *path;
4756   uint off;
4757
4758   if(argstr(0, &path) < 0)
4759     return -1;
4760   if((dp = nameiparent(path, name)) == 0)
4761     return -1;
4762   ilock(dp);
4763
4764   // Cannot unlink "." or "..".
4765   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0){
4766     iunlockput(dp);
4767     return -1;
4768   }
4769
4770   if((ip = dirlookup(dp, name, &off)) == 0){
4771     iunlockput(dp);
4772     return -1;
4773   }
4774   ilock(ip);
4775
4776   if(ip->nlink < 1)
4777     panic("unlink: nlink < 1");
4778   if(ip->type == T_DIR && !isdirempty(ip)){
4779     iunlockput(ip);
4780     iunlockput(dp);
4781     return -1;
4782   }
4783
4784   memset(&de, 0, sizeof(de));
4785   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4786     panic("unlink: writei");
4787   if(ip->type == T_DIR){
4788     dp->nlink--;
4789     iupdate(dp);
4790   }
4791   iunlockput(dp);
4792
4793   ip->nlink--;
4794   iupdate(ip);
4795   iunlockput(ip);
4796   return 0;
4797 }
4798
4799
```

```
4800 static struct inode*
4801 create(char *path, short type, short major, short minor)
4802 {
4803   uint off;
4804   struct inode *ip, *dp;
4805   char name[DIRSIZ];
4806
4807   if((dp = nameiparent(path, name)) == 0)
4808     return 0;
4809   ilock(dp);
4810
4811   if((ip = dirlookup(dp, name, &off)) != 0){
4812     iunlockput(dp);
4813     ilock(ip);
4814     if(type == T_FILE && ip->type == T_FILE)
4815       return ip;
4816     iunlockput(ip);
4817     return 0;
4818   }
4819
4820   if((ip = ialloc(dp->dev, type)) == 0)
4821     panic("create: ialloc");
4822
4823   ilock(ip);
4824   ip->major = major;
4825   ip->minor = minor;
4826   ip->nlink = 1;
4827   iupdate(ip);
4828
4829   if(type == T_DIR){  // Create . and .. entries.
4830     dp->nlink++;  // for ".."
4831     iupdate(dp);
4832     // No ip->nlink++ for ".": avoid cyclic ref count.
4833     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
4834       panic("create dots");
4835   }
4836
4837   if(dirlink(dp, name, ip->inum) < 0)
4838     panic("create: dirlink");
4839
4840   iunlockput(dp);
4841   return ip;
4842 }
4843
4844
4845
4846
4847
4848
4849
```



```
4850 int
4851 sys_open(void)
4852 {
4853   char *path;
4854   int fd, omode;
4855   struct file *f;
4856   struct inode *ip;
4857
4858   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
4859     return -1;
4860
4861   if(omode & O_CREATE){
4862     if((ip = create(path, T_FILE, 0, 0)) == 0)
4863       return -1;
4864   } else {
4865     if((ip = namei(path)) == 0)
4866       return -1;
4867     ilock(ip);
4868     if(ip->type == T_DIR && omode != O_RDONLY){
4869       iunlockput(ip);
4870       return -1;
4871     }
4872   }
4873
4874   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
4875     if(f)
4876       fileclose(f);
4877     iunlockput(ip);
4878     return -1;
4879   }
4880   iunlock(ip);
4881
4882   f->type = FD_INODE;
4883   f->ip = ip;
4884   f->off = 0;
4885   f->readable = !(omode & O_WRONLY);
4886   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
4887
4888   return fd;
4889 }
4890
4891
4892
4893
4894
4895
4896
4897
4898
4899
```

```
4900 int
4901 sys_mkdir(void)
4902 {
4903   char *path;
4904   struct inode *ip;
4905
4906   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0)
4907     return -1;
4908   iunlockput(ip);
4909   return 0;
4910 }
4911
4912 int
4913 sys_mknod(void)
4914 {
4915   struct inode *ip;
4916   char *path;
4917   int len;
4918   int major, minor;
4919
4920   if((len=argstr(0, &path)) < 0 ||
4921      argint(1, &major) < 0 ||
4922      argint(2, &minor) < 0 ||
4923      (ip = create(path, T_DEV, major, minor)) == 0)
4924     return -1;
4925   iunlockput(ip);
4926   return 0;
4927 }
4928
4929 int
4930 sys_chdir(void)
4931 {
4932   char *path;
4933   struct inode *ip;
4934
4935   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0)
4936     return -1;
4937   ilock(ip);
4938   if(ip->type != T_DIR){
4939     iunlockput(ip);
4940     return -1;
4941   }
4942   iunlock(ip);
4943   iput(cp->cwd);
4944   cp->cwd = ip;
4945   return 0;
4946 }
4947
4948
4949
```



```
4950 int
4951 sys_exec(void)
4952 {
4953   char *path, *argv[20];
4954   int i;
4955   uint uargv, uarg;
4956
4957   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0)
4958     return -1;
4959   memset(argv, 0, sizeof(argv));
4960   for(i=0;; i++){
4961     if(i >= NELEM(argv))
4962       return -1;
4963     if(fetchint(cp, uargv+4*i, (int*)&uarg) < 0)
4964       return -1;
4965     if(uarg == 0){
4966       argv[i] = 0;
4967       break;
4968     }
4969     if(fetchstr(cp, uarg, &argv[i]) < 0)
4970       return -1;
4971   }
4972   return exec(path, argv);
4973 }
4974
4975 int
4976 sys_pipe(void)
4977 {
4978   int *fd;
4979   struct file *rf, *wf;
4980   int fd0, fd1;
4981
4982   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
4983     return -1;
4984   if(pipealloc(&rf, &wf) < 0)
4985     return -1;
4986   fd0 = -1;
4987   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
4988     if(fd0 >= 0)
4989       cp->ofile[fd0] = 0;
4990     fileclose(rf);
4991     fileclose(wf);
4992     return -1;
4993   }
4994   fd[0] = fd0;
4995   fd[1] = fd1;
4996   return 0;
4997 }
4998
4999
```

```
5000 #include "types.h"
5001 #include "param.h"
5002 #include "mmu.h"
5003 #include "proc.h"
5004 #include "defs.h"
5005 #include "x86.h"
5006 #include "elf.h"
5007
5008 int
5009 exec(char *path, char **argv)
5010 {
5011   char *mem, *s, *last;
5012   int i, argc, arglen, len, off;
5013   uint sz, sp, argp, x;
5014   struct elfhdr elf;
5015   struct inode *ip;
5016   struct proghdr ph;
5017
5018   mem = 0;
5019   sz = 0;
5020
5021   if((ip = namei(path)) == 0)
5022     return -1;
5023   ilock(ip);
5024
5025   // Check ELF header
5026   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
5027     goto bad;
5028   if(elf.magic != ELF_MAGIC)
5029     goto bad;
5030
5031   // Compute memory size of new process.
5032   // Program segments.
5033   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
5034     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
5035       goto bad;
5036     if(ph.type != ELF_PROG_LOAD)
5037       continue;
5038     if(ph.memsz < ph.filesz)
5039       goto bad;
5040     sz += ph.memsz;
5041   }
5042
5043   // Arguments.
5044   arglen = 0;
5045   for(argc=0; argv[argc]; argc++)
5046     arglen += strlen(argv[argc]) + 1;
5047   arglen = (arglen+3) & ~3;
5048   sz += arglen;
5049   sz += 4*(argc+1);  // argv data
```



```
5050   sz += 4;  // argv
5051   sz += 4;  // argc
5052
5053   // Stack.
5054   sz += PAGE;
5055
5056   // Allocate program memory.
5057   sz = (sz+PAGE-1) & ~(PAGE-1);
5058   mem = kalloc(sz);
5059   if(mem == 0)
5060     goto bad;
5061   memset(mem, 0, sz);
5062
5063   // Load program into memory.
5064   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
5065     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
5066       goto bad;
5067     if(ph.type != ELF_PROG_LOAD)
5068       continue;
5069     if(ph.va + ph.memsz < ph.va || ph.va + ph.memsz > sz)
5070       goto bad;
5071     if(ph.memsz < ph.filesz)
5072       goto bad;
5073     if(readi(ip, mem + ph.va, ph.offset, ph.filesz) != ph.filesz)
5074       goto bad;
5075     memset(mem + ph.va + ph.filesz, 0, ph.memsz - ph.filesz);
5076   }
5077   iunlockput(ip);
5078
5079   // Initialize stack.
5080   sp = sz;
5081   argp = sz - arglen - 4*(argc+1);
5082
5083   // Copy argv strings and pointers to stack.
5084   *(uint*)(mem+argp + 4*argc) = 0;  // argv[argc]
5085   for(i=argc-1; i>=0; i--){
5086     len = strlen(argv[i]) + 1;
5087     sp -= len;
5088     memmove(mem+sp, argv[i], len);
5089     *(uint*)(mem+argp + 4*i) = sp;  // argv[i]
5090   }
5091
5092   // Stack frame for main(argc, argv), below arguments.
5093   sp = argp;
5094   sp -= 4;
5095   *(uint*)(mem+sp) = argp;
5096   sp -= 4;
5097   *(uint*)(mem+sp) = argc;
5098   sp -= 4;
5099   *(uint*)(mem+sp) = 0xffffffff;   // fake return pc
```

```
5100   // Save program name for debugging.
5101   for(last=s=path; *s; s++)
5102     if(*s == '/')
5103       last = s+1;
5104   safestrcpy(cp->name, last, sizeof(cp->name));
5105
5106   // Commit to the new image.
5107   kfree(cp->mem, cp->sz);
5108   cp->mem = mem;
5109   cp->sz = sz;
5110   cp->tf->eip = elf.entry;  // main
5111   cp->tf->esp = sp;
5112   usegment();
5113   return 0;
5114
5115  bad:
5116   if(mem)
5117     kfree(mem, sz);
5118   iunlockput(ip);
5119   return -1;
5120 }
5121
5122
5123
5124
5125
5126
5127
5128
5129
5130
5131
5132
5133
5134
5135
5136
5137
5138
5139
5140
5141
5142
5143
5144
5145
5146
5147
5148
5149
```



```
5150 #include "types.h"
5151 #include "defs.h"
5152 #include "param.h"
5153 #include "mmu.h"
5154 #include "proc.h"
5155 #include "file.h"
5156 #include "spinlock.h"
5157
5158 #define PIPESIZE 512
5159
5160 struct pipe {
5161   struct spinlock lock;
5162   char data[PIPESIZE];
5163   uint nread;     // number of bytes read
5164   uint nwrite;    // number of bytes written
5165   int readopen;   // read fd is still open
5166   int writeopen;  // write fd is still open
5167 };
5168
5169 int
5170 pipealloc(struct file **f0, struct file **f1)
5171 {
5172   struct pipe *p;
5173
5174   p = 0;
5175   *f0 = *f1 = 0;
5176   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
5177     goto bad;
5178   if((p = (struct pipe*)kalloc(PAGE)) == 0)
5179     goto bad;
5180   p->readopen = 1;
5181   p->writeopen = 1;
5182   p->nwrite = 0;
5183   p->nread = 0;
5184   initlock(&p->lock, "pipe");
5185   (*f0)->type = FD_PIPE;
5186   (*f0)->readable = 1;
5187   (*f0)->writable = 0;
5188   (*f0)->pipe = p;
5189   (*f1)->type = FD_PIPE;
5190   (*f1)->readable = 0;
5191   (*f1)->writable = 1;
5192   (*f1)->pipe = p;
5193   return 0;
5194
5195
5196
5197
5198
5199
```

```
5200  bad:
5201    if(p)
5202      kfree((char*)p, PAGE);
5203    if(*f0)
5204      fileclose(*f0);
5205    if(*f1)
5206      fileclose(*f1);
5207    return -1;
5208  }
5209
5210  void
5211  pipeclose(struct pipe *p, int writable)
5212  {
5213    acquire(&p->lock);
5214    if(writable){
5215      p->writeopen = 0;
5216      wakeup(&p->nread);
5217    } else {
5218      p->readopen = 0;
5219      wakeup(&p->nwrite);
5220    }
5221    if(p->readopen == 0 && p->writeopen == 0) {
5222      release(&p->lock);
5223      kfree((char*)p, PAGE);
5224    } else
5225      release(&p->lock);
5226  }
5227
5228
5229  int
5230  pipewrite(struct pipe *p, char *addr, int n)
5231  {
5232    int i;
5233
5234    acquire(&p->lock);
5235    for(i = 0; i < n; i++){
5236      while(p->nwrite == p->nread + PIPESIZE) {
5237        if(p->readopen == 0 || cp->killed){
5238          release(&p->lock);
5239          return -1;
5240        }
5241        wakeup(&p->nread);
5242        sleep(&p->nwrite, &p->lock);
5243      }
5244      p->data[p->nwrite++ % PIPESIZE] = addr[i];
5245    }
5246    wakeup(&p->nread);
5247    release(&p->lock);
5248    return n;
5249  }
```



```
5250  int
5251  piperead(struct pipe *p, char *addr, int n)
5252  {
5253    int i;
5254
5255    acquire(&p->lock);
5256    while(p->nread == p->nwrite && p->writeopen){
5257      if(cp->killed){
5258        release(&p->lock);
5259        return -1;
5260      }
5261      sleep(&p->nread, &p->lock);
5262    }
5263    for(i = 0; i < n; i++){
5264      if(p->nread == p->nwrite)
5265        break;
5266      addr[i] = p->data[p->nread++ % PIPESIZE];
5267    }
5268    wakeup(&p->nwrite);
5269    release(&p->lock);
5270    return i;
5271  }
5272
5273
5274
5275
5276
5277
5278
5279
5280
5281
5282
5283
5284
5285
5286
5287
5288
5289
5290
5291
5292
5293
5294
5295
5296
5297
5298
5299
```

```
5300 #include "types.h"
5301 #include "x86.h"
5302
5303 void*
5304 memset(void *dst, int c, uint n)
5305 {
5306   stosb(dst, c, n);
5307   return dst;
5308 }
5309
5310 int
5311 memcmp(const void *v1, const void *v2, uint n)
5312 {
5313   const uchar *s1, *s2;
5314
5315   s1 = v1;
5316   s2 = v2;
5317   while(n-- > 0){
5318     if(*s1 != *s2)
5319       return *s1 - *s2;
5320     s1++, s2++;
5321   }
5322
5323   return 0;
5324 }
5325
5326 void*
5327 memmove(void *dst, const void *src, uint n)
5328 {
5329   const char *s;
5330   char *d;
5331
5332   s = src;
5333   d = dst;
5334   if(s < d && s + n > d){
5335     s += n;
5336     d += n;
5337     while(n-- > 0)
5338       *--d = *--s;
5339   } else
5340     while(n-- > 0)
5341       *d++ = *s++;
5342
5343   return dst;
5344 }
5345
5346
5347
5348
5349
```



```
5350 int
5351 strncmp(const char *p, const char *q, uint n)
5352 {
5353   while(n > 0 && *p && *p == *q)
5354     n--, p++, q++;
5355   if(n == 0)
5356     return 0;
5357   return (uchar)*p - (uchar)*q;
5358 }
5359
5360 char*
5361 strncpy(char *s, const char *t, int n)
5362 {
5363   char *os;
5364
5365   os = s;
5366   while(n-- > 0 && (*s++ = *t++) != 0)
5367     ;
5368   while(n-- > 0)
5369     *s++ = 0;
5370   return os;
5371 }
5372
5373 // Like strncpy but guaranteed to NUL-terminate.
5374 char*
5375 safestrcpy(char *s, const char *t, int n)
5376 {
5377   char *os;
5378
5379   os = s;
5380   if(n <= 0)
5381     return os;
5382   while(--n > 0 && (*s++ = *t++) != 0)
5383     ;
5384   *s = 0;
5385   return os;
5386 }
5387
5388 int
5389 strlen(const char *s)
5390 {
5391   int n;
5392
5393   for(n = 0; s[n]; n++)
5394     ;
5395   return n;
5396 }
5397
5398
5399
```

```
5400 // See MultiProcessor Specification Version 1.[14]
5401
5402 struct mp {             // floating pointer
5403   uchar signature[4];           // "_MP_"
5404   void *physaddr;               // phys addr of MP config table
5405   uchar length;                 // 1
5406   uchar specrev;                // [14]
5407   uchar checksum;               // all bytes must add up to 0
5408   uchar type;                   // MP system config type
5409   uchar imcrp;
5410   uchar reserved[3];
5411 };
5412
5413 struct mpconf {         // configuration table header
5414   uchar signature[4];           // "PCMP"
5415   ushort length;                // total table length
5416   uchar version;                // [14]
5417   uchar checksum;               // all bytes must add up to 0
5418   uchar product[20];            // product id
5419   uint *oemtable;               // OEM table pointer
5420   ushort oemlength;             // OEM table length
5421   ushort entry;                 // entry count
5422   uint *lapicaddr;              // address of local APIC
5423   ushort xlength;               // extended table length
5424   uchar xchecksum;              // extended table checksum
5425   uchar reserved;
5426 };
5427
5428 struct mpproc {         // processor table entry
5429   uchar type;                   // entry type (0)
5430   uchar apicid;                 // local APIC id
5431   uchar version;                // local APIC verison
5432   uchar flags;                  // CPU flags
5433     #define MPBOOT 0x02           // This proc is the bootstrap processor.
5434   uchar signature[4];           // CPU signature
5435   uint feature;                 // feature flags from CPUID instruction
5436   uchar reserved[8];
5437 };
5438
5439 struct mpioapic {       // I/O APIC table entry
5440   uchar type;                   // entry type (2)
5441   uchar apicno;                 // I/O APIC id
5442   uchar version;                // I/O APIC version
5443   uchar flags;                  // I/O APIC flags
5444   uint *addr;                  // I/O APIC address
5445 };
5446
5447
5448
5449
```



```
5450 // Table entry types
5451 #define MPPROC    0x00  // One per processor
5452 #define MPBUS     0x01  // One per bus
5453 #define MPIOAPIC  0x02  // One per I/O APIC
5454 #define MPIOINTR  0x03  // One per bus interrupt source
5455 #define MPLINTR   0x04  // One per system interrupt source
5456
5457
5458
5459
5460
5461
5462
5463
5464
5465
5466
5467
5468
5469
5470
5471
5472
5473
5474
5475
5476
5477
5478
5479
5480
5481
5482
5483
5484
5485
5486
5487
5488
5489
5490
5491
5492
5493
5494
5495
5496
5497
5498
5499
```

```
5500 // Multiprocessor bootstrap.
5501 // Search memory for MP description structures.
5502 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
5503
5504 #include "types.h"
5505 #include "defs.h"
5506 #include "param.h"
5507 #include "mp.h"
5508 #include "x86.h"
5509 #include "mmu.h"
5510 #include "proc.h"
5511
5512 struct cpu cpus[NCPU];
5513 static struct cpu *bcpu;
5514 int ismp;
5515 int ncpu;
5516 uchar ioapicid;
5517
5518 int
5519 mpbcpu(void)
5520 {
5521   return bcpu-cpus;
5522 }
5523
5524 static uchar
5525 sum(uchar *addr, int len)
5526 {
5527   int i, sum;
5528
5529   sum = 0;
5530   for(i=0; i<len; i++)
5531     sum += addr[i];
5532   return sum;
5533 }
5534
5535 // Look for an MP structure in the len bytes at addr.
5536 static struct mp*
5537 mpsearch1(uchar *addr, int len)
5538 {
5539   uchar *e, *p;
5540
5541   e = addr+len;
5542   for(p = addr; p < e; p += sizeof(struct mp))
5543     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
5544       return (struct mp*)p;
5545   return 0;
5546 }
5547
5548
5549
```



```
5550 // Search for the MP Floating Pointer Structure, which according to the
5551 // spec is in one of the following three locations:
5552 // 1) in the first KB of the EBDA;
5553 // 2) in the last KB of system base memory;
5554 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
5555 static struct mp*
5556 mpsearch(void)
5557 {
5558   uchar *bda;
5559   uint p;
5560   struct mp *mp;
5561
5562   bda = (uchar*)0x400;
5563   if((p = ((bda[0x0F]<<8)|bda[0x0E]) << 4)){
5564     if((mp = mpsearch1((uchar*)p, 1024)))
5565       return mp;
5566   } else {
5567     p = ((bda[0x14]<<8)|bda[0x13])*1024;
5568     if((mp = mpsearch1((uchar*)p-1024, 1024)))
5569       return mp;
5570   }
5571   return mpsearch1((uchar*)0xF0000, 0x10000);
5572 }
5573
5574 // Search for an MP configuration table.  For now,
5575 // don't accept the default configurations (physaddr == 0).
5576 // Check for correct signature, calculate the checksum and,
5577 // if correct, check the version.
5578 // To do: check extended table checksum.
5579 static struct mpconf*
5580 mpconfig(struct mp **pmp)
5581 {
5582   struct mpconf *conf;
5583   struct mp *mp;
5584
5585   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
5586     return 0;
5587   conf = (struct mpconf*)mp->physaddr;
5588   if(memcmp(conf, "PCMP", 4) != 0)
5589     return 0;
5590   if(conf->version != 1 && conf->version != 4)
5591     return 0;
5592   if(sum((uchar*)conf, conf->length) != 0)
5593     return 0;
5594   *pmp = mp;
5595   return conf;
5596 }
5597
5598
5599
```

```
5600 void
5601 mpinit(void)
5602 {
5603   uchar *p, *e;
5604   struct mp *mp;
5605   struct mpconf *conf;
5606   struct mpproc *proc;
5607   struct mpioapic *ioapic;
5608
5609   bcpu = &cpus[ncpu];
5610   if((conf = mpconfig(&mp)) == 0)
5611     return;
5612
5613   ismp = 1;
5614   lapic = (uint*)conf->lapicaddr;
5615
5616   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
5617     switch(*p){
5618     case MPPROC:
5619       proc = (struct mpproc*)p;
5620       cpus[ncpu].apicid = proc->apicid;
5621       if(proc->flags & MPBOOT)
5622         bcpu = &cpus[ncpu];
5623       ncpu++;
5624       p += sizeof(struct mpproc);
5625       continue;
5626     case MPIOAPIC:
5627       ioapic = (struct mpioapic*)p;
5628       ioapicid = ioapic->apicno;
5629       p += sizeof(struct mpioapic);
5630       continue;
5631     case MPBUS:
5632     case MPIOINTR:
5633     case MPLINTR:
5634       p += 8;
5635       continue;
5636     default:
5637       cprintf("mpinit: unknown config type %x\n", *p);
5638       panic("mpinit");
5639     }
5640   }
5641
5642   if(mp->imcrp){
5643     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
5644     // But it would on real hardware.
5645     outb(0x22, 0x70);   // Select IMCR
5646     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
5647   }
5648 }
5649
```



```
5650 // The local APIC manages internal (non-I/O) interrupts.
5651 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
5652
5653 #include "types.h"
5654 #include "defs.h"
5655 #include "traps.h"
5656 #include "mmu.h"
5657 #include "x86.h"
5658
5659 // Local APIC registers, divided by 4 for use as uint[] indices.
5660 #define ID      (0x0020/4)   // ID
5661 #define VER     (0x0030/4)   // Version
5662 #define TPR     (0x0080/4)   // Task Priority
5663 #define EOI     (0x00B0/4)   // EOI
5664 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
5665   #define ENABLE     0x00000100   // Unit Enable
5666 #define ESR     (0x0280/4)   // Error Status
5667 #define ICRLO   (0x0300/4)   // Interrupt Command
5668   #define INIT       0x00000500   // INIT/RESET
5669   #define STARTUP    0x00000600   // Startup IPI
5670   #define DELIVS     0x00001000   // Delivery status
5671   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
5672   #define LEVEL      0x00008000   // Level triggered
5673   #define BCAST      0x00080000   // Send to all APICs, including self.
5674 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
5675 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
5676   #define X1         0x0000000B   // divide counts by 1
5677   #define PERIODIC   0x00020000   // Periodic
5678 #define PCINT   (0x0340/4)   // Performance Counter LVT
5679 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
5680 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
5681 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
5682   #define MASKED     0x00010000   // Interrupt masked
5683 #define TICR    (0x0380/4)   // Timer Initial Count
5684 #define TCCR    (0x0390/4)   // Timer Current Count
5685 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
5686
5687 volatile uint *lapic;  // Initialized in mp.c
5688
5689 static void
5690 lapicw(int index, int value)
5691 {
5692   lapic[index] = value;
5693   lapic[ID];  // wait for write to finish, by reading
5694 }
5695
5696
5697
5698
5699
```

```
5700 void
5701 lapicinit(int c)
5702 {
5703   if(!lapic)
5704     return;
5705
5706   // Enable local APIC; set spurious interrupt vector.
5707   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
5708
5709   // The timer repeatedly counts down at bus frequency
5710   // from lapic[TICR] and then issues an interrupt.
5711   // If xv6 cared more about precise timekeeping,
5712   // TICR would be calibrated using an external time source.
5713   lapicw(TDCR, X1);
5714   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
5715   lapicw(TICR, 10000000);
5716
5717   // Disable logical interrupt lines.
5718   lapicw(LINT0, MASKED);
5719   lapicw(LINT1, MASKED);
5720
5721   // Disable performance counter overflow interrupts
5722   // on machines that provide that interrupt entry.
5723   if(((lapic[VER]>>16) & 0xFF) >= 4)
5724     lapicw(PCINT, MASKED);
5725
5726   // Map error interrupt to IRQ_ERROR.
5727   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
5728
5729   // Clear error status register (requires back-to-back writes).
5730   lapicw(ESR, 0);
5731   lapicw(ESR, 0);
5732
5733   // Ack any outstanding interrupts.
5734   lapicw(EOI, 0);
5735
5736   // Send an Init Level De-Assert to synchronise arbitration ID's.
5737   lapicw(ICRHI, 0);
5738   lapicw(ICRLO, BCAST | INIT | LEVEL);
5739   while(lapic[ICRLO] & DELIVS)
5740     ;
5741
5742   // Enable interrupts on the APIC (but not on the processor).
5743   lapicw(TPR, 0);
5744 }
5745
5746
5747
5748
5749
```



```
5750 int
5751 cpu(void)
5752 {
5753   // Cannot call cpu when interrupts are enabled:
5754   // result not guaranteed to last long enough to be used!
5755   // Would prefer to panic but even printing is chancy here:
5756   // almost everything, including cprintf and panic, calls cpu,
5757   // often indirectly through acquire and release.
5758   if(readeflags()&FL_IF){
5759     static int n;
5760     if(n++ == 0)
5761       cprintf("cpu called from %x with interrupts enabled\n",
5762         __builtin_return_address(0));
5763   }
5764
5765   if(lapic)
5766     return lapic[ID]>>24;
5767   return 0;
5768 }
5769
5770 // Acknowledge interrupt.
5771 void
5772 lapiceoi(void)
5773 {
5774   if(lapic)
5775     lapicw(EOI, 0);
5776 }
5777
5778 // Spin for a given number of microseconds.
5779 // On real hardware would want to tune this dynamically.
5780 void
5781 microdelay(int us)
5782 {
5783   volatile int j = 0;
5784
5785   while(us-- > 0)
5786     for(j=0; j<10000; j++);
5787 }
5788
5789
5790
5791
5792
5793
5794
5795
5796
5797
5798
5799
```

```
5800 #define IO_RTC  0x70
5801
5802 // Start additional processor running bootstrap code at addr.
5803 // See Appendix B of MultiProcessor Specification.
5804 void
5805 lapicstartap(uchar apicid, uint addr)
5806 {
5807   int i;
5808   ushort *wrv;
5809
5810   // "The BSP must initialize CMOS shutdown code to 0AH
5811   // and the warm reset vector (DWORD based at 40:67) to point at
5812   // the AP startup code prior to the [universal startup algorithm]."
5813   outb(IO_RTC, 0xF);  // offset 0xF is shutdown code
5814   outb(IO_RTC+1, 0x0A);
5815   wrv = (ushort*)(0x40<<4 | 0x67);  // Warm reset vector
5816   wrv[0] = 0;
5817   wrv[1] = addr >> 4;
5818
5819   // "Universal startup algorithm."
5820   // Send INIT (level-triggered) interrupt to reset other CPU.
5821   lapicw(ICRHI, apicid<<24);
5822   lapicw(ICRLO, INIT | LEVEL | ASSERT);
5823   microdelay(200);
5824   lapicw(ICRLO, INIT | LEVEL);
5825   microdelay(100);    // should be 10ms, but too slow in Bochs!
5826
5827   // Send startup IPI (twice!) to enter bootstrap code.
5828   // Regular hardware is supposed to only accept a STARTUP
5829   // when it is in the halted state due to an INIT.  So the second
5830   // should be ignored, but it is part of the official Intel algorithm.
5831   // Bochs complains about the second one.  Too bad for Bochs.
5832   for(i = 0; i < 2; i++){
5833     lapicw(ICRHI, apicid<<24);
5834     lapicw(ICRLO, STARTUP | (addr>>12));
5835     microdelay(200);
5836   }
5837 }
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
```



```
5850 // The I/O APIC manages hardware interrupts for an SMP system.
5851 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
5852 // See also picirq.c.
5853
5854 #include "types.h"
5855 #include "defs.h"
5856 #include "traps.h"
5857
5858 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
5859
5860 #define REG_ID     0x00  // Register index: ID
5861 #define REG_VER    0x01  // Register index: version
5862 #define REG_TABLE  0x10  // Redirection table base
5863
5864 // The redirection table starts at REG_TABLE and uses
5865 // two registers to configure each interrupt.
5866 // The first (low) register in a pair contains configuration bits.
5867 // The second (high) register contains a bitmask telling which
5868 // CPUs can serve that interrupt.
5869 #define INT_DISABLED   0x00010000  // Interrupt disabled
5870 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
5871 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
5872 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
5873
5874 volatile struct ioapic *ioapic;
5875
5876 // IO APIC MMIO structure: write reg, then read or write data.
5877 struct ioapic {
5878   uint reg;
5879   uint pad[3];
5880   uint data;
5881 };
5882
5883 static uint
5884 ioapicread(int reg)
5885 {
5886   ioapic->reg = reg;
5887   return ioapic->data;
5888 }
5889
5890 static void
5891 ioapicwrite(int reg, uint data)
5892 {
5893   ioapic->reg = reg;
5894   ioapic->data = data;
5895 }
5896
5897
5898
5899
```

```
5900 void
5901 ioapicinit(void)
5902 {
5903   int i, id, maxintr;
5904
5905   if(!ismp)
5906     return;
5907
5908   ioapic = (volatile struct ioapic*)IOAPIC;
5909   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
5910   id = ioapicread(REG_ID) >> 24;
5911   if(id != ioapicid)
5912     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
5913
5914   // Mark all interrupts edge-triggered, active high, disabled,
5915   // and not routed to any CPUs.
5916   for(i = 0; i <= maxintr; i++){
5917     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
5918     ioapicwrite(REG_TABLE+2*i+1, 0);
5919   }
5920 }
5921
5922 void
5923 ioapicenable(int irq, int cpunum)
5924 {
5925   if(!ismp)
5926     return;
5927
5928   // Mark interrupt edge-triggered, active high,
5929   // enabled, and routed to the given cpunum,
5930   // which happens to be that cpu's APIC ID.
5931   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
5932   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
5933 }
5934
5935
5936
5937
5938
5939
5940
5941
5942
5943
5944
5945
5946
5947
5948
5949
```



```
5950 // Intel 8259A programmable interrupt controllers.
5951
5952 #include "types.h"
5953 #include "x86.h"
5954 #include "traps.h"
5955
5956 // I/O Addresses of the two programmable interrupt controllers
5957 #define IO_PIC1         0x20    // Master (IRQs 0-7)
5958 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
5959
5960 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
5961
5962 // Current IRQ mask.
5963 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
5964 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
5965
5966 static void
5967 picsetmask(ushort mask)
5968 {
5969   irqmask = mask;
5970   outb(IO_PIC1+1, mask);
5971   outb(IO_PIC2+1, mask >> 8);
5972 }
5973
5974 void
5975 picenable(int irq)
5976 {
5977   picsetmask(irqmask & ~(1<<irq));
5978 }
5979
5980 // Initialize the 8259A interrupt controllers.
5981 void
5982 picinit(void)
5983 {
5984   // mask all interrupts
5985   outb(IO_PIC1+1, 0xFF);
5986   outb(IO_PIC2+1, 0xFF);
5987
5988   // Set up master (8259A-1)
5989
5990   // ICW1:  0001g0hi
5991   //    g:  0 = edge triggering, 1 = level triggering
5992   //    h:  0 = cascaded PICs, 1 = master only
5993   //    i:  0 = no ICW4, 1 = ICW4 required
5994   outb(IO_PIC1, 0x11);
5995
5996   // ICW2:  Vector offset
5997   outb(IO_PIC1+1, T_IRQ0);
5998
5999
```

```
6000   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
6001   //        (slave PIC) 3-bit # of slave's connection to master
6002   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
6003
6004   // ICW4:  000nbmap
6005   //    n:  1 = special fully nested mode
6006   //    b:  1 = buffered mode
6007   //    m:  0 = slave PIC, 1 = master PIC
6008   //      (ignored when b is 0, as the master/slave role
6009   //      can be hardwired).
6010   //    a:  1 = Automatic EOI mode
6011   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
6012   outb(IO_PIC1+1, 0x3);
6013
6014   // Set up slave (8259A-2)
6015   outb(IO_PIC2, 0x11);                  // ICW1
6016   outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
6017   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
6018   // NB Automatic EOI mode doesn't tend to work on the slave.
6019   // Linux source code says it's "to be investigated".
6020   outb(IO_PIC2+1, 0x3);                 // ICW4
6021
6022   // OCW3:  0ef01prs
6023   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
6024   //    p:  0 = no polling, 1 = polling mode
6025   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
6026   outb(IO_PIC1, 0x68);             // clear specific mask
6027   outb(IO_PIC1, 0x0a);             // read IRR by default
6028
6029   outb(IO_PIC2, 0x68);             // OCW3
6030   outb(IO_PIC2, 0x0a);             // OCW3
6031
6032   if(irqmask != 0xFFFF)
6033     picsetmask(irqmask);
6034 }
6035
6036
6037
6038
6039
6040
6041
6042
6043
6044
6045
6046
6047
6048
6049
```



```
6050 // Blank page.
6051
6052
6053
6054
6055
6056
6057
6058
6059
6060
6061
6062
6063
6064
6065
6066
6067
6068
6069
6070
6071
6072
6073
6074
6075
6076
6077
6078
6079
6080
6081
6082
6083
6084
6085
6086
6087
6088
6089
6090
6091
6092
6093
6094
6095
6096
6097
6098
6099
```

```
6100 // PC keyboard interface constants
6101
6102 #define KBSTATP         0x64    // kbd controller status port(I)
6103 #define KBS_DIB         0x01    // kbd data in buffer
6104 #define KBDATAP         0x60    // kbd data port(I)
6105
6106 #define NO              0
6107
6108 #define SHIFT           (1<<0)
6109 #define CTL             (1<<1)
6110 #define ALT             (1<<2)
6111
6112 #define CAPSLOCK        (1<<3)
6113 #define NUMLOCK         (1<<4)
6114 #define SCROLLLOCK      (1<<5)
6115
6116 #define E0ESC           (1<<6)
6117
6118 // Special keycodes
6119 #define KEY_HOME        0xE0
6120 #define KEY_END         0xE1
6121 #define KEY_UP          0xE2
6122 #define KEY_DN          0xE3
6123 #define KEY_LF          0xE4
6124 #define KEY_RT          0xE5
6125 #define KEY_PGUP        0xE6
6126 #define KEY_PGDN        0xE7
6127 #define KEY_INS         0xE8
6128 #define KEY_DEL         0xE9
6129
6130 // C('A') == Control-A
6131 #define C(x) (x - '@')
6132
6133 static uchar shiftcode[256] =
6134 {
6135   [0x1D] CTL,
6136   [0x2A] SHIFT,
6137   [0x36] SHIFT,
6138   [0x38] ALT,
6139   [0x9D] CTL,
6140   [0xB8] ALT
6141 };
6142
6143 static uchar togglecode[256] =
6144 {
6145   [0x3A] CAPSLOCK,
6146   [0x45] NUMLOCK,
6147   [0x46] SCROLLLOCK
6148 };
6149
```



```
6150 static uchar normalmap[256] =
6151 {
6152   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
6153   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
6154   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
6155   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
6156   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
6157   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
6158   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
6159   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
6160   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
6161   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
6162   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
6163   [0x9C] '\n',      // KP_Enter
6164   [0xB5] '/',       // KP_Div
6165   [0xC8] KEY_UP,    [0xD0] KEY_DN,
6166   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
6167   [0xCB] KEY_LF,    [0xCD] KEY_RT,
6168   [0x97] KEY_HOME,  [0xCF] KEY_END,
6169   [0xD2] KEY_INS,   [0xD3] KEY_DEL
6170 };
6171
6172 static uchar shiftmap[256] =
6173 {
6174   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
6175   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
6176   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
6177   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
6178   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
6179   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
6180   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
6181   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
6182   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
6183   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
6184   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
6185   [0x9C] '\n',      // KP_Enter
6186   [0xB5] '/',       // KP_Div
6187   [0xC8] KEY_UP,    [0xD0] KEY_DN,
6188   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
6189   [0xCB] KEY_LF,    [0xCD] KEY_RT,
6190   [0x97] KEY_HOME,  [0xCF] KEY_END,
6191   [0xD2] KEY_INS,   [0xD3] KEY_DEL
6192 };
6193
6194
6195
6196
6197
6198
6199
```

```
6200 static uchar ctlmap[256] =
6201 {
6202   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
6203   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
6204   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
6205   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
6206   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
6207   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
6208   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
6209   [0x9C] '\r',      // KP_Enter
6210   [0xB5] C('/'),    // KP_Div
6211   [0xC8] KEY_UP,    [0xD0] KEY_DN,
6212   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
6213   [0xCB] KEY_LF,    [0xCD] KEY_RT,
6214   [0x97] KEY_HOME,  [0xCF] KEY_END,
6215   [0xD2] KEY_INS,   [0xD3] KEY_DEL
6216 };
6217
6218
6219
6220
6221
6222
6223
6224
6225
6226
6227
6228
6229
6230
6231
6232
6233
6234
6235
6236
6237
6238
6239
6240
6241
6242
6243
6244
6245
6246
6247
6248
6249
```



```
6250 #include "types.h"
6251 #include "x86.h"
6252 #include "defs.h"
6253 #include "kbd.h"
6254
6255 int
6256 kbdgetc(void)
6257 {
6258   static uint shift;
6259   static uchar *charcode[4] = {
6260     normalmap, shiftmap, ctlmap, ctlmap
6261   };
6262   uint st, data, c;
6263
6264   st = inb(KBSTATP);
6265   if((st & KBS_DIB) == 0)
6266     return -1;
6267   data = inb(KBDATAP);
6268
6269   if(data == 0xE0){
6270     shift |= E0ESC;
6271     return 0;
6272   } else if(data & 0x80){
6273     // Key released
6274     data = (shift & E0ESC ? data : data & 0x7F);
6275     shift &= ~(shiftcode[data] | E0ESC);
6276     return 0;
6277   } else if(shift & E0ESC){
6278     // Last character was an E0 escape; or with 0x80
6279     data |= 0x80;
6280     shift &= ~E0ESC;
6281   }
6282
6283   shift |= shiftcode[data];
6284   shift ^= togglecode[data];
6285   c = charcode[shift & (CTL | SHIFT)][data];
6286   if(shift & CAPSLOCK){
6287     if('a' <= c && c <= 'z')
6288       c += 'A' - 'a';
6289     else if('A' <= c && c <= 'Z')
6290       c += 'a' - 'A';
6291   }
6292   return c;
6293 }
6294
6295 void
6296 kbdintr(void)
6297 {
6298   consoleintr(kbdgetc);
6299 }
```

```
6300 // Console input and output.
6301 // Input is from the keyboard or serial port.
6302 // Output is written to the screen and serial port.
6303
6304 #include "types.h"
6305 #include "defs.h"
6306 #include "param.h"
6307 #include "traps.h"
6308 #include "spinlock.h"
6309 #include "file.h"
6310 #include "mmu.h"
6311 #include "proc.h"
6312 #include "x86.h"
6313
6314 static void consputc(int);
6315
6316 static int panicked = 0;
6317
6318 static struct {
6319    struct spinlock lock;
6320    int locking;
6321 } cons;
6322
6323 static void
6324 printint(int xx, int base, int sgn)
6325 {
6326   static char digits[] = "0123456789abcdef";
6327   char buf[16];
6328   int i = 0, neg = 0;
6329   uint x;
6330
6331   if(sgn && xx < 0){
6332     neg = 1;
6333     x = -xx;
6334   } else
6335     x = xx;
6336
6337   do{
6338     buf[i++] = digits[x % base];
6339   }while((x /= base) != 0);
6340   if(neg)
6341     buf[i++] = '-';
6342
6343   while(--i >= 0)
6344     consputc(buf[i]);
6345 }
6346
6347
6348
6349
```



```
6350 // Print to the console. only understands %d, %x, %p, %s.
6351 void
6352 cprintf(char *fmt, ...)
6353 {
6354   int i, c, state, locking;
6355   uint *argp;
6356   char *s;
6357
6358   locking = cons.locking;
6359   if(locking)
6360     acquire(&cons.lock);
6361
6362   argp = (uint*)(void*)&fmt + 1;
6363   state = 0;
6364   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
6365     if(c != '%'){
6366       consputc(c);
6367       continue;
6368     }
6369     c = fmt[++i] & 0xff;
6370     if(c == 0)
6371       break;
6372     switch(c){
6373     case 'd':
6374       printint(*argp++, 10, 1);
6375       break;
6376     case 'x':
6377     case 'p':
6378       printint(*argp++, 16, 0);
6379       break;
6380     case 's':
6381       if((s = (char*)*argp++) == 0)
6382         s = "(null)";
6383       for(; *s; s++)
6384         consputc(*s);
6385       break;
6386     case '%':
6387       consputc('%');
6388       break;
6389     default:
6390       // Print unknown % sequence to draw attention.
6391       consputc('%');
6392       consputc(c);
6393       break;
6394     }
6395   }
6396
6397   if(locking)
6398     release(&cons.lock);
6399 }
```

```
6400 void
6401 panic(char *s)
6402 {
6403   int i;
6404   uint pcs[10];
6405
6406   cli();
6407   cons.locking = 0;
6408   cprintf("cpu%d: panic: ", cpu());
6409   cprintf(s);
6410   cprintf("\n");
6411   getcallerpcs(&s, pcs);
6412   for(i=0; i<10; i++)
6413     cprintf(" %p", pcs[i]);
6414   panicked = 1; // freeze other CPU
6415   for(;;)
6416     ;
6417 }
6418
6419
6420
6421
6422
6423
6424
6425
6426
6427
6428
6429
6430
6431
6432
6433
6434
6435
6436
6437
6438
6439
6440
6441
6442
6443
6444
6445
6446
6447
6448
6449
```



```
6450 #define BACKSPACE 0x100
6451 #define CRTPORT 0x3d4
6452 static ushort *crt = (ushort*)0xb8000;  // CGA memory
6453
6454 static void
6455 cgaputc(int c)
6456 {
6457   int pos;
6458
6459   // Cursor position: col + 80*row.
6460   outb(CRTPORT, 14);
6461   pos = inb(CRTPORT+1) << 8;
6462   outb(CRTPORT, 15);
6463   pos |= inb(CRTPORT+1);
6464
6465   if(c == '\n')
6466     pos += 80 - pos%80;
6467   else if(c == BACKSPACE){
6468     if(pos > 0)
6469       crt[--pos] = ' ' | 0x0700;
6470   } else
6471     crt[pos++] = (c&0xff) | 0x0700;  // black on white
6472
6473   if((pos/80) >= 24){  // Scroll up.
6474     memmove(crt, crt+80, sizeof(crt[0])*23*80);
6475     pos -= 80;
6476     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
6477   }
6478
6479   outb(CRTPORT, 14);
6480   outb(CRTPORT+1, pos>>8);
6481   outb(CRTPORT, 15);
6482   outb(CRTPORT+1, pos);
6483   crt[pos] = ' ' | 0x0700;
6484 }
6485
6486 void
6487 consputc(int c)
6488 {
6489   if(panicked){
6490     cli();
6491     for(;;)
6492       ;
6493   }
6494
6495   uartputc(c);
6496   cgaputc(c);
6497 }
6498
6499
```

```
6500 #define INPUT_BUF 128
6501 struct {
6502   struct spinlock lock;
6503   char buf[INPUT_BUF];
6504   uint r;  // Read index
6505   uint w;  // Write index
6506   uint e;  // Edit index
6507 } input;
6508
6509 #define C(x)  ((x)-'@')  // Control-x
6510
6511 void
6512 consoleintr(int (*getc)(void))
6513 {
6514   int c;
6515
6516   acquire(&input.lock);
6517   while((c = getc()) >= 0){
6518     switch(c){
6519     case C('P'):  // Process listing.
6520       procdump();
6521       break;
6522     case C('U'):  // Kill line.
6523       while(input.e != input.w &&
6524             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
6525         input.e--;
6526         consputc(BACKSPACE);
6527       }
6528       break;
6529     case C('H'):  // Backspace
6530       if(input.e != input.w){
6531         input.e--;
6532         consputc(BACKSPACE);
6533       }
6534       break;
6535     default:
6536       if(c != 0 && input.e-input.r < INPUT_BUF){
6537         input.buf[input.e++ % INPUT_BUF] = c;
6538         consputc(c);
6539         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
6540           input.w = input.e;
6541           wakeup(&input.r);
6542         }
6543       }
6544       break;
6545     }
6546   }
6547   release(&input.lock);
6548 }
6549
```



```
6550 int
6551 consoleread(struct inode *ip, char *dst, int n)
6552 {
6553   uint target;
6554   int c;
6555
6556   iunlock(ip);
6557   target = n;
6558   acquire(&input.lock);
6559   while(n > 0){
6560     while(input.r == input.w){
6561       if(cp->killed){
6562         release(&input.lock);
6563         ilock(ip);
6564         return -1;
6565       }
6566       sleep(&input.r, &input.lock);
6567     }
6568     c = input.buf[input.r++ % INPUT_BUF];
6569     if(c == C('D')){  // EOF
6570       if(n < target){
6571         // Save ^D for next time, to make sure
6572         // caller gets a 0-byte result.
6573         input.r--;
6574       }
6575       break;
6576     }
6577     *dst++ = c;
6578     --n;
6579     if(c == '\n')
6580       break;
6581   }
6582   release(&input.lock);
6583   ilock(ip);
6584
6585   return target - n;
6586 }
6587
6588
6589
6590
6591
6592
6593
6594
6595
6596
6597
6598
6599
```

```
6600 int
6601 consolewrite(struct inode *ip, char *buf, int n)
6602 {
6603   int i;
6604
6605   iunlock(ip);
6606   acquire(&cons.lock);
6607   for(i = 0; i < n; i++)
6608     consputc(buf[i] & 0xff);
6609   release(&cons.lock);
6610   ilock(ip);
6611
6612   return n;
6613 }
6614
6615 void
6616 consoleinit(void)
6617 {
6618   initlock(&cons.lock, "console");
6619   initlock(&input.lock, "input");
6620
6621   devsw[CONSOLE].write = consolewrite;
6622   devsw[CONSOLE].read = consoleread;
6623   cons.locking = 1;
6624
6625   picenable(IRQ_KBD);
6626   ioapicenable(IRQ_KBD, 0);
6627 }
6628
6629
6630
6631
6632
6633
6634
6635
6636
6637
6638
6639
6640
6641
6642
6643
6644
6645
6646
6647
6648
6649
```



```
6650 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
6651 // Only used on uniprocessors;
6652 // SMP machines use the local APIC timer.
6653
6654 #include "types.h"
6655 #include "defs.h"
6656 #include "traps.h"
6657 #include "x86.h"
6658
6659 #define IO_TIMER1       0x040           // 8253 Timer #1
6660
6661 // Frequency of all three count-down timers;
6662 // (TIMER_FREQ/freq) is the appropriate count
6663 // to generate a frequency of freq Hz.
6664
6665 #define TIMER_FREQ      1193182
6666 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
6667
6668 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
6669 #define TIMER_SEL0      0x00    // select counter 0
6670 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
6671 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
6672
6673 void
6674 timerinit(void)
6675 {
6676   // Interrupt 100 times/sec.
6677   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
6678   outb(IO_TIMER1, TIMER_DIV(100) % 256);
6679   outb(IO_TIMER1, TIMER_DIV(100) / 256);
6680   picenable(IRQ_TIMER);
6681 }
6682
6683
6684
6685
6686
6687
6688
6689
6690
6691
6692
6693
6694
6695
6696
6697
6698
6699
```

```
6700 # Initial process execs /init.
6701
6702 #include "syscall.h"
6703 #include "traps.h"
6704
6705 # exec(init, argv)
6706 .globl start
6707 start:
6708   pushl $argv
6709   pushl $init
6710   pushl $0  // where caller pc would be
6711   movl $SYS_exec, %eax
6712   int $T_SYSCALL
6713
6714 # for(;;) exit();
6715 exit:
6716   movl $SYS_exit, %eax
6717   int $T_SYSCALL
6718   jmp exit
6719
6720 # char init[] = "/init\0";
6721 init:
6722   .string "/init\0"
6723
6724 # char *argv[] = { init, 0 };
6725 .p2align 2
6726 argv:
6727   .long init
6728   .long 0
6729
6730
6731
6732
6733
6734
6735
6736
6737
6738
6739
6740
6741
6742
6743
6744
6745
6746
6747
6748
6749
```



```
6750 #include "syscall.h"
6751 #include "traps.h"
6752
6753 #define SYSCALL(name) \
6754   .globl name; \
6755   name: \
6756     movl $SYS_ ## name, %eax; \
6757     int $T_SYSCALL; \
6758     ret
6759
6760 SYSCALL(fork)
6761 SYSCALL(exit)
6762 SYSCALL(wait)
6763 SYSCALL(pipe)
6764 SYSCALL(read)
6765 SYSCALL(write)
6766 SYSCALL(close)
6767 SYSCALL(kill)
6768 SYSCALL(exec)
6769 SYSCALL(open)
6770 SYSCALL(mknod)
6771 SYSCALL(unlink)
6772 SYSCALL(fstat)
6773 SYSCALL(link)
6774 SYSCALL(mkdir)
6775 SYSCALL(chdir)
6776 SYSCALL(dup)
6777 SYSCALL(getpid)
6778 SYSCALL(sbrk)
6779 SYSCALL(sleep)
6780
6781
6782
6783
6784
6785
6786
6787
6788
6789
6790
6791
6792
6793
6794
6795
6796
6797
6798
6799
```

```
6800 // init: The initial user-level program
6801
6802 #include "types.h"
6803 #include "stat.h"
6804 #include "user.h"
6805 #include "fcntl.h"
6806
6807 char *argv[] = { "sh", 0 };
6808
6809 int
6810 main(void)
6811 {
6812   int pid, wpid;
6813
6814   if(open("console", O_RDWR) < 0){
6815     mknod("console", 1, 1);
6816     open("console", O_RDWR);
6817   }
6818   dup(0);  // stdout
6819   dup(0);  // stderr
6820
6821   for(;;){
6822     printf(1, "init: starting sh\n");
6823     pid = fork();
6824     if(pid < 0){
6825       printf(1, "init: fork failed\n");
6826       exit();
6827     }
6828     if(pid == 0){
6829       exec("sh", argv);
6830       printf(1, "init: exec sh failed\n");
6831       exit();
6832     }
6833     while((wpid=wait()) >= 0 && wpid != pid)
6834       printf(1, "zombie!\n");
6835   }
6836 }
6837
6838
6839
6840
6841
6842
6843
6844
6845
6846
6847
6848
6849
```



```
6850 // Shell.
6851
6852 #include "types.h"
6853 #include "user.h"
6854 #include "fcntl.h"
6855
6856 // Parsed command representation
6857 #define EXEC  1
6858 #define REDIR 2
6859 #define PIPE  3
6860 #define LIST  4
6861 #define BACK  5
6862
6863 #define MAXARGS 10
6864
6865 struct cmd {
6866   int type;
6867 };
6868
6869 struct execcmd {
6870   int type;
6871   char *argv[MAXARGS];
6872   char *eargv[MAXARGS];
6873 };
6874
6875 struct redircmd {
6876   int type;
6877   struct cmd *cmd;
6878   char *file;
6879   char *efile;
6880   int mode;
6881   int fd;
6882 };
6883
6884 struct pipecmd {
6885   int type;
6886   struct cmd *left;
6887   struct cmd *right;
6888 };
6889
6890 struct listcmd {
6891   int type;
6892   struct cmd *left;
6893   struct cmd *right;
6894 };
6895
6896 struct backcmd {
6897   int type;
6898   struct cmd *cmd;
6899 };
```

```
6900 int fork1(void);  // Fork but panics on failure.
6901 void panic(char*);
6902 struct cmd *parsecmd(char*);
6903
6904 // Execute cmd.  Never returns.
6905 void
6906 runcmd(struct cmd *cmd)
6907 {
6908   int p[2];
6909   struct backcmd *bcmd;
6910   struct execcmd *ecmd;
6911   struct listcmd *lcmd;
6912   struct pipecmd *pcmd;
6913   struct redircmd *rcmd;
6914
6915   if(cmd == 0)
6916     exit();
6917
6918   switch(cmd->type){
6919   default:
6920     panic("runcmd");
6921
6922   case EXEC:
6923     ecmd = (struct execcmd*)cmd;
6924     if(ecmd->argv[0] == 0)
6925       exit();
6926     exec(ecmd->argv[0], ecmd->argv);
6927     printf(2, "exec %s failed\n", ecmd->argv[0]);
6928     break;
6929
6930   case REDIR:
6931     rcmd = (struct redircmd*)cmd;
6932     close(rcmd->fd);
6933     if(open(rcmd->file, rcmd->mode) < 0){
6934       printf(2, "open %s failed\n", rcmd->file);
6935       exit();
6936     }
6937     runcmd(rcmd->cmd);
6938     break;
6939
6940   case LIST:
6941     lcmd = (struct listcmd*)cmd;
6942     if(fork1() == 0)
6943       runcmd(lcmd->left);
6944     wait();
6945     runcmd(lcmd->right);
6946     break;
6947
6948
6949
```



```
6950   case PIPE:
6951     pcmd = (struct pipecmd*)cmd;
6952     if(pipe(p) < 0)
6953       panic("pipe");
6954     if(fork1() == 0){
6955       close(1);
6956       dup(p[1]);
6957       close(p[0]);
6958       close(p[1]);
6959       runcmd(pcmd->left);
6960     }
6961     if(fork1() == 0){
6962       close(0);
6963       dup(p[0]);
6964       close(p[0]);
6965       close(p[1]);
6966       runcmd(pcmd->right);
6967     }
6968     close(p[0]);
6969     close(p[1]);
6970     wait();
6971     wait();
6972     break;
6973
6974   case BACK:
6975     bcmd = (struct backcmd*)cmd;
6976     if(fork1() == 0)
6977       runcmd(bcmd->cmd);
6978     break;
6979   }
6980   exit();
6981 }
6982
6983 int
6984 getcmd(char *buf, int nbuf)
6985 {
6986   printf(2, "$ ");
6987   memset(buf, 0, nbuf);
6988   gets(buf, nbuf);
6989   if(buf[0] == 0) // EOF
6990     return -1;
6991   return 0;
6992 }
6993
6994
6995
6996
6997
6998
6999
```

```
7000 int
7001 main(void)
7002 {
7003   static char buf[100];
7004   int fd;
7005
7006   // Assumes three file descriptors open.
7007   while((fd = open("console", O_RDWR)) >= 0){
7008     if(fd >= 3){
7009       close(fd);
7010       break;
7011     }
7012   }
7013
7014   // Read and run input commands.
7015   while(getcmd(buf, sizeof(buf)) >= 0){
7016     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
7017       // Clumsy but will have to do for now.
7018       // Chdir has no effect on the parent if run in the child.
7019       buf[strlen(buf)-1] = 0;  // chop \n
7020       if(chdir(buf+3) < 0)
7021         printf(2, "cannot cd %s\n", buf+3);
7022       continue;
7023     }
7024     if(fork1() == 0)
7025       runcmd(parsecmd(buf));
7026     wait();
7027   }
7028   exit();
7029 }
7030
7031 void
7032 panic(char *s)
7033 {
7034   printf(2, "%s\n", s);
7035   exit();
7036 }
7037
7038 int
7039 fork1(void)
7040 {
7041   int pid;
7042
7043   pid = fork();
7044   if(pid == -1)
7045     panic("fork");
7046   return pid;
7047 }
7048
7049
```



```
7050 // Constructors
7051
7052 struct cmd*
7053 execcmd(void)
7054 {
7055   struct execcmd *cmd;
7056
7057   cmd = malloc(sizeof(*cmd));
7058   memset(cmd, 0, sizeof(*cmd));
7059   cmd->type = EXEC;
7060   return (struct cmd*)cmd;
7061 }
7062
7063 struct cmd*
7064 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
7065 {
7066   struct redircmd *cmd;
7067
7068   cmd = malloc(sizeof(*cmd));
7069   memset(cmd, 0, sizeof(*cmd));
7070   cmd->type = REDIR;
7071   cmd->cmd = subcmd;
7072   cmd->file = file;
7073   cmd->efile = efile;
7074   cmd->mode = mode;
7075   cmd->fd = fd;
7076   return (struct cmd*)cmd;
7077 }
7078
7079 struct cmd*
7080 pipecmd(struct cmd *left, struct cmd *right)
7081 {
7082   struct pipecmd *cmd;
7083
7084   cmd = malloc(sizeof(*cmd));
7085   memset(cmd, 0, sizeof(*cmd));
7086   cmd->type = PIPE;
7087   cmd->left = left;
7088   cmd->right = right;
7089   return (struct cmd*)cmd;
7090 }
7091
7092
7093
7094
7095
7096
7097
7098
7099
```

```
7100 struct cmd*
7101 listcmd(struct cmd *left, struct cmd *right)
7102 {
7103   struct listcmd *cmd;
7104
7105   cmd = malloc(sizeof(*cmd));
7106   memset(cmd, 0, sizeof(*cmd));
7107   cmd->type = LIST;
7108   cmd->left = left;
7109   cmd->right = right;
7110   return (struct cmd*)cmd;
7111 }
7112
7113 struct cmd*
7114 backcmd(struct cmd *subcmd)
7115 {
7116   struct backcmd *cmd;
7117
7118   cmd = malloc(sizeof(*cmd));
7119   memset(cmd, 0, sizeof(*cmd));
7120   cmd->type = BACK;
7121   cmd->cmd = subcmd;
7122   return (struct cmd*)cmd;
7123 }
7124
7125
7126
7127
7128
7129
7130
7131
7132
7133
7134
7135
7136
7137
7138
7139
7140
7141
7142
7143
7144
7145
7146
7147
7148
7149
```



```
7150 // Parsing
7151
7152 char whitespace[] = " \t\r\n\v";
7153 char symbols[] = "<|>&;()";
7154
7155 int
7156 gettoken(char **ps, char *es, char **q, char **eq)
7157 {
7158   char *s;
7159   int ret;
7160
7161   s = *ps;
7162   while(s < es && strchr(whitespace, *s))
7163     s++;
7164   if(q)
7165     *q = s;
7166   ret = *s;
7167   switch(*s){
7168   case 0:
7169     break;
7170   case '|':
7171   case '(':
7172   case ')':
7173   case ';':
7174   case '&':
7175   case '<':
7176     s++;
7177     break;
7178   case '>':
7179     s++;
7180     if(*s == '>'){
7181       ret = '+';
7182       s++;
7183     }
7184     break;
7185   default:
7186     ret = 'a';
7187     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
7188       s++;
7189     break;
7190   }
7191   if(eq)
7192     *eq = s;
7193
7194   while(s < es && strchr(whitespace, *s))
7195     s++;
7196   *ps = s;
7197   return ret;
7198 }
7199
```

```
7200 int
7201 peek(char **ps, char *es, char *toks)
7202 {
7203   char *s;
7204
7205   s = *ps;
7206   while(s < es && strchr(whitespace, *s))
7207     s++;
7208   *ps = s;
7209   return *s && strchr(toks, *s);
7210 }
7211
7212 struct cmd *parseline(char**, char*);
7213 struct cmd *parsepipe(char**, char*);
7214 struct cmd *parseexec(char**, char*);
7215 struct cmd *nulterminate(struct cmd*);
7216
7217 struct cmd*
7218 parsecmd(char *s)
7219 {
7220   char *es;
7221   struct cmd *cmd;
7222
7223   es = s + strlen(s);
7224   cmd = parseline(&s, es);
7225   peek(&s, es, "");
7226   if(s != es){
7227     printf(2, "leftovers: %s\n", s);
7228     panic("syntax");
7229   }
7230   nulterminate(cmd);
7231   return cmd;
7232 }
7233
7234 struct cmd*
7235 parseline(char **ps, char *es)
7236 {
7237   struct cmd *cmd;
7238
7239   cmd = parsepipe(ps, es);
7240   while(peek(ps, es, "&")){
7241     gettoken(ps, es, 0, 0);
7242     cmd = backcmd(cmd);
7243   }
7244   if(peek(ps, es, ";")){
7245     gettoken(ps, es, 0, 0);
7246     cmd = listcmd(cmd, parseline(ps, es));
7247   }
7248   return cmd;
7249 }
```



```
7250 struct cmd*
7251 parsepipe(char **ps, char *es)
7252 {
7253   struct cmd *cmd;
7254
7255   cmd = parseexec(ps, es);
7256   if(peek(ps, es, "|")){
7257     gettoken(ps, es, 0, 0);
7258     cmd = pipecmd(cmd, parsepipe(ps, es));
7259   }
7260   return cmd;
7261 }
7262
7263 struct cmd*
7264 parseredirs(struct cmd *cmd, char **ps, char *es)
7265 {
7266   int tok;
7267   char *q, *eq;
7268
7269   while(peek(ps, es, "<>")){
7270     tok = gettoken(ps, es, 0, 0);
7271     if(gettoken(ps, es, &q, &eq) != 'a')
7272       panic("missing file for redirection");
7273     switch(tok){
7274     case '<':
7275       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
7276       break;
7277     case '>':
7278       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
7279       break;
7280     case '+':  // >>
7281       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
7282       break;
7283     }
7284   }
7285   return cmd;
7286 }
7287
7288
7289
7290
7291
7292
7293
7294
7295
7296
7297
7298
7299
```

```
7300 struct cmd*
7301 parseblock(char **ps, char *es)
7302 {
7303   struct cmd *cmd;
7304
7305   if(!peek(ps, es, "("))
7306     panic("parseblock");
7307   gettoken(ps, es, 0, 0);
7308   cmd = parseline(ps, es);
7309   if(!peek(ps, es, ")"))
7310     panic("syntax - missing )");
7311   gettoken(ps, es, 0, 0);
7312   cmd = parseredirs(cmd, ps, es);
7313   return cmd;
7314 }
7315
7316 struct cmd*
7317 parseexec(char **ps, char *es)
7318 {
7319   char *q, *eq;
7320   int tok, argc;
7321   struct execcmd *cmd;
7322   struct cmd *ret;
7323
7324   if(peek(ps, es, "("))
7325     return parseblock(ps, es);
7326
7327   ret = execcmd();
7328   cmd = (struct execcmd*)ret;
7329
7330   argc = 0;
7331   ret = parseredirs(ret, ps, es);
7332   while(!peek(ps, es, "|)&;")){
7333     if((tok=gettoken(ps, es, &q, &eq)) == 0)
7334       break;
7335     if(tok != 'a')
7336       panic("syntax");
7337     cmd->argv[argc] = q;
7338     cmd->eargv[argc] = eq;
7339     argc++;
7340     if(argc >= MAXARGS)
7341       panic("too many args");
7342     ret = parseredirs(ret, ps, es);
7343   }
7344   cmd->argv[argc] = 0;
7345   cmd->eargv[argc] = 0;
7346   return ret;
7347 }
7348
7349
```



```
7350 // NUL-terminate all the counted strings.
7351 struct cmd*
7352 nulterminate(struct cmd *cmd)
7353 {
7354   int i;
7355   struct backcmd *bcmd;
7356   struct execcmd *ecmd;
7357   struct listcmd *lcmd;
7358   struct pipecmd *pcmd;
7359   struct redircmd *rcmd;
7360
7361   if(cmd == 0)
7362     return 0;
7363
7364   switch(cmd->type){
7365   case EXEC:
7366     ecmd = (struct execcmd*)cmd;
7367     for(i=0; ecmd->argv[i]; i++)
7368       *ecmd->eargv[i] = 0;
7369     break;
7370
7371   case REDIR:
7372     rcmd = (struct redircmd*)cmd;
7373     nulterminate(rcmd->cmd);
7374     *rcmd->efile = 0;
7375     break;
7376
7377   case PIPE:
7378     pcmd = (struct pipecmd*)cmd;
7379     nulterminate(pcmd->left);
7380     nulterminate(pcmd->right);
7381     break;
7382
7383   case LIST:
7384     lcmd = (struct listcmd*)cmd;
7385     nulterminate(lcmd->left);
7386     nulterminate(lcmd->right);
7387     break;
7388
7389   case BACK:
7390     bcmd = (struct backcmd*)cmd;
7391     nulterminate(bcmd->cmd);
7392     break;
7393   }
7394   return cmd;
7395 }
7396
7397
7398
7399
```