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

ACKNOWLEDGMENTS

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

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

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

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

ERROR REPORTS

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

BUILDING AND RUNNING XV6

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

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

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

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

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

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

locks
15 spinlock.h
15 spinlock.c

processes
17 proc.h
18 proc.c
23 swtch.S
24 kalloc.c
25 data.S
25 vm.c

system calls
29 traps.h
30 vectors.pl
30 trapasm.S
31 trap.c
32 syscall.h
33 syscall.c
34 sysproc.c

file system
35 buf.h
36 fcntl.h
36 stat.h
37 fs.h
37 file.h
38 ide.c
40 bio.c
41 fs.c
49 file.c
50 sysfile.c
55 exec.c

pipes
56 pipe.c

string operations
57 string.c

low-level hardware
59 mp.h
60 mp.c
62 lapic.c
64 ioapic.c
65 picirq.c
66 kbd.h
67 kbd.c
68 console.c
71 timer.c
72 uart.c
73 multiboot.S

user-level
74 initcode.S
74 usys.S
75 init.c
75 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 2358
        0317 2128 2166 2357 2358

indicates that swtch is defined on line 2358 and is mentioned on five lines
on sheets 03, 21, and 23.

acquire 1573
    0320 1573 1577 1860 2023
    2058 2117 2174 2218 2233
    2266 2279 2465 2480 3166
    3522 3542 3907 3965 4070
    4129 4357 4390 4410 4439
    4454 4464 4925 4941 4956
    5663 5684 5705 6860 7016
    7058 7106
allocproc 1855
    1855 1907 1960
allocuvm 2777
    0367 1937 2777 2791 5542
    5552
alltraps 3056
    3009 3017 3030 3035 3055
    3056
ALT 6610
    6610 6638 6640
argfd 5063
    5063 5106 5121 5133 5144
    5156
argint 3344
    0338 3344 3358 3374 3481
    3506 3520 5068 5121 5133
    5358 5409 5410 5457
argptr 3354
    0339 3354 5121 5133 5156
    5483
argstr 3371
    0340 3371 5168 5258 5358
    5395 5408 5423 5457
BACK 7561
    7561 7674 7820 8089
backcmd 7596 7814
    7596 7609 7675 7814 7816
    7942 8055 8090
BACKSPACE 6950
    6950 6967 6994 7026 7032
balloc 4204
    4204 4225 4517 4525 4529
BBLOCK 3741
    3741 4213 4239
B_BUSY 3559
    3559 3958 4076 4077 4088
    4091 4116 4126 4138
B_DIRTY 3561
    3561 3887 3916 3921 3960
    3979 4118
bfree 4230
    4230 4562 4572 4575
bget 4066
    4066 4096 4106
binit 4039
    0210 1363 4039
bmap 4510
    4510 4536 4619 4669 4722
bootmain 1216
    1068 1216
bootothers 1402
    1307 1370 1402
BPB 3738
    3738 3741 4212 4214 4240
bread 4102
    0211 4102 4182 4193 4213
    4239 4311 4332 4417 4526
    4568 4619 4669 4722
brelse 4124
    0212 4124 4127 4184 4196
    4219 4223 4246 4317 4320
    4341 4425 4532 4574 4622
    4673 4733 4737
BSIZE 3708
    3708 3718 3732 3738 4194
    4619 4620 4621 4665 4666
    4669 4670 4671 4721 4722
    4724
buf 3550
    0200 0211 0212 0213 0253
    2920 2923 2932 2934 3550
    3554 3555 3556 3810 3825
    3828 3875 3904 3954 3956
    3959 4027 4031 4035 4041
    4053 4065 4068 4101 4104
    4114 4124 4169 4180 4191
    4207 4232 4305 4329 4404
    4513 4557 4605 4655 4715
    6828 6840 6843 6846 7003
    7024 7038 7068 7101 7108
    7684 7687 7688 7689 7703
    7715 7716 7719 7720 7721
    7725
B_VALID 3560
    3560 3920 3960 3979 4107
bwrite 4114
    0213 4114 4117 4195 4218
    4245 4316 4340 4530 4672
bzero 4189
    4189 4236
C 6631 7009

    6631 6679 6704 6705 6706
    6707 6708 6710 7009 7019
    7022 7029 7040 7069
CAPSLOCK 6612
    6612 6645 6786
cgaputc 6955
    6955 6998
cli 0517
    0517 0519 1015 1129 1660
    6906 6989
cmd 7565
    7565 7577 7586 7587 7592
    7593 7598 7602 7606 7615
    7618 7623 7631 7637 7641
    7651 7675 7677 7752 7755
    7757 7758 7759 7760 7763
    7764 7766 7768 7769 7770
    7771 7772 7773 7774 7775
    7776 7779 7780 7782 7784
    7785 7786 7787 7788 7789
    7800 7801 7803 7805 7806
    7807 7808 7809 7810 7813
    7814 7816 7818 7819 7820
    7821 7822 7912 7913 7914
    7915 7917 7921 7924 7930
    7931 7934 7937 7939 7942
    7946 7948 7950 7953 7955
    7958 7960 7963 7964 7975
    7978 7981 7985 8000 8003
    8008 8012 8013 8016 8021
    8022 8028 8037 8038 8044
    8045 8051 8052 8061 8064
    8066 8072 8073 8078 8084
    8090 8091 8094
COM1 7213
    7213 7223 7226 7227 7228
    7229 7230 7231 7234 7240
    7241 7257 7259 7267 7269
CONSOLE 3790
    3790 7121 7122
consoleinit 7116
    0216 1358 7116
consoleintr 7012
    0218 6798 7012 7275
consoleread 7051
    7051 7122
consolewrite 7101
    7101 7121
consputc 6986
    6815 6846 6866 6884 6887
    6891 6892 6986 7026 7032
    7039 7108
context 1760
    0201 0317 1713 1760 1779
    1888 1889 1890 1891 2128
    2166 2328
copyout 2918
    0375 2918 5562 5573
copyuvm 2853
    0372 1964 2853 2864 2866
cprintf 6852
    0217 1355 1387 2326 2330
    2332 2791 3190 3203 3208
    3433 6119 6139 6253 6311
    6462 6852 6908 6909 6910
    6913
cpu 1711
    0256 1355 1387 1389 1406
    1506 1565 1586 1608 1646
    1661 1662 1670 1672 1711
    1721 1725 1736 2128 2159
    2165 2166 2167 2575 2588
    2594 2725 2726 2727 2728
    3165 3190 3191 3203 3204
    3208 3210 6012 6013 6311
    6908
cpunum 6301
    0269 1382 1384 1416 2581
    6301 6473 6482
CRO_PE 0727 1010 1124
    0727 1046 1138
CRO_PG 0737
    0737 2708
create 5301
    2604 2613 5301 5321 5334
    5338 5361 5395 5411
CRTPORT 6951
    6951 6960 6961 6962 6963
    6978 6979 6980 6981
CTL 6609
    6609 6635 6639 6785
data 2523
    0405 0407 0408 0421 0423
    0427 0429 0442 0446 2521
    2522 2523 2558 2678 2679
    3021 3039 3557 3889 3917
    4183 4194 4216 4217 4242
    4244 4312 4333 4418 4527
    4569 4621 4671 4723 4724
    4731 5613 5694 5716 6430

    6437 6441 6444 6762 6767
    6769 6772 6774 6775 6779
    6783 6784 6785
deallocuvm 2805
    0368 1940 2792 2805 2836
devsw 3783
    3783 3788 4608 4610 4658
    4660 4907 7121 7122
dinode 3722
    3722 3732 4306 4312 4330
    4333 4405 4418
dirent 3746
    3746 4716 4723 4724 4755
    5205 5254
dirlink 4752
    0234 4752 4767 4775 5184
    5333 5337 5338
dirlookup 4712
    0235 4712 4719 4759 4874
    5270 5311
DIRSIZ 3744
    3744 3748 4705 4772 4828
    4829 4891 5165 5255 5305
DPL_USER 0777
    0777 1914 1915 2584 2585
    3122 3218 3227
EOESC 6616
    6616 6770 6774 6775 6777
    6780
elfhdr 0955
    0955 1218 1223 5514
ELF_MAGIC 0952
    0952 1229 5527
ELF_PROG_LOAD 0986
    0986 5538
EOI 6213
    6213 6285 6325
ERROR 6234
    6234 6278
ESR 6216
    6216 6281 6282
exec 5509
    0222 5473 5509 7468 7529
    7530 7626 7627
EXEC 7557
    7557 7622 7759 8065
execcmd 7569 7753
    7569 7610 7623 7753 7755
    8021 8027 8028 8056 8066
exit 2004
    0302 2004 2040 3155 3159
    3219 3228 3466 7415 7418
    7461 7526 7531 7616 7625
    7635 7680 7728 7735
fdalloc 5082
    5082 5108 5373 5488
fetchint 3316
    0341 3316 3346 5464
fetchstr 3328
    0342 3328 3376 5470
file 3750
    0202 0225 0226 0227 0229
    0230 0231 0287 1782 3750
    4171 4904 4910 4920 4923
    4926 4938 4939 4952 4954
    4976 5002 5022 5057 5063
    5066 5082 5103 5117 5129
    5142 5153 5355 5480 5606
    5621 6810 7208 7578 7633
    7634 7764 7772 7972
filealloc 4921
    0225 4921 5373 5627
fileclose 4952
    0226 2015 4952 4958 5147
    5375 5491 5492 5654 5656
filedup 4939
    0227 1979 4939 4943 5110
fileinit 4914
    0228 1364 4914
fileread 5002
    0229 5002 5017 5123
filestat 4976
    0230 4976 5158
filewrite 5022
    0231 5022 5037 5135
FL_IF 0710
    0710 1662 1668 1918 2163
    6308
fork 1954
    0303 1954 3460 7460 7523
    7525 7743 7745
fork1 7739
    7600 7642 7654 7661 7676
    7724 7739
forkret 2183
    1816 1891 2183
freevm 2830
    0369 2071 2830 2835 2877
    5589 5595
gatedesc 0901

    0464 0467 0901 3110
getcallerpcs 1626
    0321 1587 1626 2328 6911
getcmd 7684
    7684 7715
gettoken 7856
    7856 7941 7945 7957 7970
    7971 8007 8011 8033
growproc 1931
    0304 1931 3509
havedisk1 3827
    3827 3864 3962
holding 1644
    0322 1576 1604 1644 2157
ialloc 4302
    0236 4302 4322 5320 5321
IBLOCK 3735
    3735 4311 4332 4417
I_BUSY 3777
    3777 4411 4413 4436 4440
    4457 4459
ICRHI 6227
    6227 6288 6356 6368
ICRLO 6217
    6217 6289 6290 6357 6359
    6369
ID 6210
    6210 6246 6316
IDE_BSY 3812
    3812 3836
IDE_CMD_READ 3817
    3817 3891
IDE_CMD_WRITE 3818
    3818 3888
IDE_DF 3814
    3814 3838
IDE_DRDY 3813
    3813 3836
IDE_ERR 3815
    3815 3838
ideinit 3851
    0251 1366 3851
ideintr 3902
    0252 3174 3902
idelock 3824
    3824 3855 3907 3909 3928
    3965 3980 3983
iderw 3954
    0253 3954 3959 3961 3963
    4108 4119
idestart 3875
    3828 3875 3878 3926 3975
idewait 3832
    3832 3858 3880 3916
idtinit 3128
    0351 1388 3128
idup 4388
    0237 1980 4388 4861
iget 4353
    4294 4318 4353 4373 4734
    4859
iinit 4289
    0238 1365 4289
ilock 4402
    0239 4402 4408 4428 4864
    4979 5011 5031 5172 5183
    5193 5262 5274 5309 5313
    5323 5366 5425 5521 7063
    7083 7110
inb 0403
    0403 1026 1034 1254 3836
    3863 6154 6764 6767 6961
    6963 7234 7240 7241 7257
    7267 7269
initlock 1561
    0323 1561 1824 2427 3124
    3855 4043 4291 4916 5635
    7118 7119
inituvm 2739
    0370 1911 2739 2744
inode 3763
    0203 0234 0235 0236 0237
    0239 0240 0241 0242 0243
    0245 0246 0247 0248 0249
    0371 1783 2753 3756 3763
    3784 3785 4174 4285 4294
    4301 4327 4352 4355 4361
    4387 4388 4402 4434 4452
    4474 4510 4554 4585 4602
    4652 4711 4712 4752 4756
    4853 4856 4888 4895 5166
    5202 5253 5300 5304 5356
    5393 5403 5421 5515 7051
    7101
INPUT_BUF 7000
    7000 7003 7024 7036 7038
    7040 7068
insl 0412
    0412 0414 1273 3917
INT_DISABLED 6419

    6419 6467
ioapic 6427
    6107 6129 6130 6424 6427
    6436 6437 6443 6444 6458
IOAPIC 6408
    6408 6458
ioapicenable 6473
    0256 3857 6473 7126 7243
ioapicid 6016
    0257 6016 6130 6147 6461
    6462
ioapicinit 6451
    0258 1357 6451 6462
ioapicread 6434
    6434 6459 6460
ioapicwrite 6441
    6441 6467 6468 6481 6482
IO_PIC1 6507
    6507 6520 6535 6544 6547
    6552 6562 6576 6577
IO_PIC2 6508
    6508 6521 6536 6565 6566
    6567 6570 6579 6580
IO_RTC 6335
    6335 6348 6349
IO_TIMER1 7159
    7159 7168 7178 7179
IPB 3732
    3732 3735 3741 4312 4333
    4418
iput 4452
    0240 2020 4452 4458 4477
    4760 4882 4971 5189 5431
IRQ_COM1 2983
    2983 3184 7242 7243
IRQ_ERROR 2985
    2985 6278
IRQ_IDE 2984
    2984 3173 3177 3856 3857
IRQ_KBD 2982
    2982 3180 7125 7126
IRQ_SLAVE 6510
    6510 6514 6552 6567
IRQ_SPURIOUS 2986
    2986 3189 6258
IRQ_TIMER 2981
    2981 3164 3223 6265 7180
isdirempty 5202
    5202 5209 5278
ismp 6014
    0277 1367 6014 6112 6120
    6140 6143 6455 6475
itrunc 4554
    4174 4461 4554
iunlock 4434
    0241 4434 4437 4476 4871
    4981 5014 5034 5179 5379
    5430 7056 7105
iunlockput 4474
    0242 4474 4866 4875 4878
    5174 5185 5188 5196 5266
    5271 5279 5280 5291 5295
    5312 5316 5340 5368 5376
    5397 5413 5427 5547 5597
iupdate 4327
    0243 4327 4463 4580 4678
    5178 5195 5289 5294 5327
    5331
I_VALID 3778
    3778 4416 4426 4455
jmpkstack 1326
    1309 1322 1326 1332 1335
kalloc 2476
    0261 1330 1332 1422 1873
    2476 2613 2690 2745 2789
    2868 5629
KBDATAP 6604
    6604 6767
kbdgetc 6756
    6756 6798
kbdintr 6796
    0266 3181 6796
KBS_DIB 6603
    6603 6765
KBSTATP 6602
    6602 6764
KEY_DEL 6628
    6628 6669 6691 6715
KEY_DN 6622
    6622 6665 6687 6711
KEY_END 6620
    6620 6668 6690 6714
KEY_HOME 6619
    6619 6668 6690 6714
KEY_INS 6627
    6627 6669 6691 6715
KEY_LF 6623
    6623 6667 6689 6713
KEY_PGDN 6626
    6626 6666 6688 6712

KEY_PGUP 6625
    6625 6666 6688 6712
KEY_RT 6624
    6624 6667 6689 6713
KEY_UP 6621
    6621 6665 6687 6711
kfree 2455
    0262 1965 2069 2430 2455
    2460 2819 2820 2839 2841
    5652 5673
kill 2275
    0305 2275 3209 3483 7467
kinit 2423
    0263 1321 2423
KSTACKSIZE 0151
    0151 1423 1877 2728
kvmalloc 2565
    0363 1360 2565
lapiceoi 6322
    0271 3171 3175 3182 3186
    3192 6322
lapicinit 6251
    0272 1319 1384 6251 6253
lapicstartap 6340
    0273 1426 6340
lapicw 6243
    6243 6258 6264 6265 6266
    6269 6270 6275 6278 6281
    6282 6285 6288 6289 6294
    6325 6356 6357 6359 6368
    6369
lcr0 0551
    0551 2709
lcr3 0573
    0573 2717 2732
lgdt 0453
    0453 0461 1044 1136 2590
    7344
lidt 0467
    0467 0475 3130
LINT0 6232
    6232 6269
LINT1 6233
    6233 6270
LIST 7560
    7560 7640 7807 8083
listcmd 7590 7801
    7590 7611 7641 7801 7803
    7946 8057 8084
loadgs 0493
    0493 2591
loaduvm 2753
    0371 2753 2759 2762 5544
ltr 0479
    0479 0481 2729
mainc 1353
    1310 1334 1353
mappages 2629
    2629 2695 2747 2796 2871
MAXARG 0161
    0161 5453 5513 5558
MAXARGS 7563
    7563 7571 7572 8040
MAXFILE 3719
    3719 4665 4666
memcmp 5761
    0329 5761 6043 6088
memmove 5777
    0330 1413 2748 2870 2932
    4183 4339 4424 4621 4671
    4829 4831 5777 5804 6973
memset 5754
    0331 1890 1913 2463 2616
    2692 2746 2795 4194 4314
    5284 5460 5754 6975 7687
    7758 7769 7785 7806 7819
microdelay 6331
    0274 6331 6358 6360 6370
    7258
min 4173
    4173 4620 4670
mp 5902
    5902 6007 6036 6042 6043
    6044 6055 6060 6064 6065
    6068 6069 6080 6083 6085
    6087 6094 6104 6110 6150
mpbcpu 6019
    0278 1319 1382 6019
MPBUS 5952
    5952 6133
mpconf 5913
    5913 6079 6082 6087 6105
mpconfig 6080
    6080 6110
mpinit 6101
    0279 1318 6101 6119 6139
mpioapic 5939
    5939 6107 6129 6131
MPIOAPIC 5953
    5953 6128

MPIOINTR 5954
    5954 6134
MPLINTR 5955
    5955 6135
mpmain 1380
    1308 1373 1380 1424
mpproc 5928
    5928 6106 6117 6126
MPPROC 5951
    5951 6116
mpsearch 6056
    6056 6085
mpsearch1 6037
    6037 6064 6068 6071
multiboot_entry 7343
    7337 7342 7343
multiboot_header 7327
    7326 7327 7333 7334
namecmp 4703
    0244 4703 4728 5265
namei 4889
    0245 1923 4889 5170 5364
    5423 5519
nameiparent 4896
    0246 4854 4869 4881 4896
    5181 5260 5307
namex 4854
    4854 4892 4898
NBUF 0155
    0155 4031 4053
ncpu 6015
    1355 1415 1726 3857 6015
    6118 6119 6123 6124 6125
    6145
NCPU 0152
    0152 1725 6012
NDEV 0157
    0157 4608 4658 4907
NDIRECT 3717
    3717 3719 3728 3774 4515
    4520 4524 4525 4560 4567
    4568 4575 4576
NELEM 0378
    0378 2322 2694 3430 5462
nextpid 1815
    1815 1869
NFILE 0154
    0154 4910 4926
NINDIRECT 3718
    3718 3719 4522 4570
NINODE 0156
    0156 4285 4361
NO 6606
    6606 6652 6655 6657 6658
    6659 6660 6662 6674 6677
    6679 6680 6681 6682 6684
    6702 6703 6705 6706 6707
    6708
NOFILE 0153
    0153 1782 1977 2013 5070
    5086
NPDENTRIES 0823
    0823 2837
NPROC 0150
    0150 1810 1861 2029 2062
    2118 2257 2280 2319
NSEGS 1708
    1708 1715
nulterminate 8052
    7915 7930 8052 8073 8079
    8080 8085 8086 8091
NUMLOCK 6613
    6613 6646
O_CREATE 3603
    3603 5360 7978 7981
O_RDONLY 3600
    3600 5367 7975
O_RDWR 3602
    3602 5385 7514 7516 7707
outb 0421
    0421 1031 1039 1264 1265
    1266 1267 1268 1269 3861
    3870 3881 3882 3883 3884
    3885 3886 3888 3891 6153
    6154 6348 6349 6520 6521
    6535 6536 6544 6547 6552
    6562 6565 6566 6567 6570
    6576 6577 6579 6580 6960
    6962 6978 6979 6980 6981
    7177 7178 7179 7223 7226
    7227 7228 7229 7230 7231
    7259
outsl 0433
    0433 0435 3889
outw 0427
    0427 1074 1076 1170 1172
O_WRONLY 3601
    3601 5384 5385 7978 7981
PADDR 0820
    0820 2620 2717 2732 2747

    2796 2871
panic 6901 7732
    0219 1332 1335 1577 1605
    1669 1671 1910 2010 2040
    2158 2160 2162 2164 2206
    2209 2460 2641 2731 2744
    2759 2762 2819 2835 2864
    2866 3205 3878 3959 3961
    3963 4096 4117 4127 4225
    4243 4322 4373 4408 4428
    4437 4458 4536 4719 4767
    4775 4943 4958 5017 5037
    5209 5277 5286 5321 5334
    5338 6901 6908 7601 7620
    7653 7732 7745 7928 7972
    8006 8010 8036 8041
panicked 6817
    6817 6914 6988
parseblock 8001
    8001 8006 8025
parsecmd 7918
    7602 7725 7918
parseexec 8017
    7914 7955 8017
parseline 7935
    7912 7924 7935 7946 8008
parsepipe 7951
    7913 7939 7951 7958
parseredirs 7964
    7964 8012 8031 8042
PCINT 6231
    6231 6275
pde_t 0103
    0103 0365 0366 0367 0368
    0369 0370 0371 0372 0375
    1773 2560 2604 2606 2629
    2684 2687 2690 2739 2753
    2777 2805 2830 2852 2853
    2855 2902 2918 5517
PDX 0809
    0809 2609
PDXSHIFT 0830
    0809 0815 0830
peek 7901
    7901 7925 7940 7944 7956
    7969 8005 8009 8024 8032
PGROUNDDOWN 0833
    0833 2634 2635 2925
PGROUNDUP 0832
    0832 2428 2787 2813 5551
PGSIZE 0826
    0826 0832 0833 1333 1912
    1919 2429 2459 2463 2616
    2645 2646 2692 2743 2746
    2747 2758 2760 2764 2767
    2788 2795 2796 2814 2862
    2870 2871 2929 2935 5552
PHYSTOP 0160
    0160 2429 2459 2679
picenable 6525
    0283 3856 6525 7125 7180
    7242
picinit 6532
    0284 1356 6532
picsetmask 6517
    6517 6527 6583
pinit 1822
    0306 1361 1822
pipe 5611
    0204 0288 0289 0290 3755
    4969 5009 5029 5611 5623
    5629 5635 5639 5643 5661
    5680 5701 7463 7652 7653
PIPE 7559
    7559 7650 7786 8077
pipealloc 5621
    0287 5485 5621
pipeclose 5661
    0288 4969 5661
pipecmd 7584 7780
    7584 7612 7651 7780 7782
    7958 8058 8078
piperead 5701
    0289 5009 5701
PIPESIZE 5609
    5609 5613 5686 5694 5716
pipewrite 5680
    0290 5029 5680
popcli 1666
    0326 1621 1666 1669 1671
    2733
printint 6825
    6825 6874 6878
proc 1771
    0205 0301 0341 0342 0373
    1304 1557 1722 1737 1771
    1777 1805 1810 1813 1854
    1857 1861 1904 1935 1937
    1940 1943 1944 1957 1964
    1970 1971 1972 1978 1979

    1980 1984 2006 2009 2014
    2015 2016 2020 2021 2026
    2029 2030 2038 2055 2062
    2063 2083 2089 2110 2118
    2125 2128 2133 2161 2166
    2175 2205 2223 2224 2228
    2255 2257 2277 2280 2315
    2319 2555 2595 2722 2728
    3104 3154 3156 3158 3201
    3209 3210 3212 3218 3223
    3227 3304 3316 3328 3346
    3360 3376 3429 3431 3434
    3435 3455 3489 3508 3525
    3806 4167 4861 5055 5070
    5087 5088 5146 5431 5432
    5464 5470 5490 5503 5580
    5583 5584 5585 5586 5587
    5588 5604 5687 5707 6010
    6106 6117 6118 6119 6122
    6812 7061 7210
procdump 2304
    0307 2304 7020
proghdr 0974
    0974 1219 1233 5516
PTE_ADDR 0847
    0847 2611 2763 2817 2839
    2867 2911
PTE_P 0836
    0836 2610 2620 2640 2642
    2816 2838 2865 2907
pte_t 0849
    0849 2603 2607 2611 2613
    2632 2756 2807 2856 2904
PTE_U 0838
    0838 2620 2747 2796 2871
    2909
PTE_W 0837
    0837 2620 2677 2679 2680
    2747 2796 2871
PTX 0812
    0812 2622
PTXSHIFT 0829
    0812 0815 0829
pushcli 1655
    0325 1575 1655 2724
rcr0 0557
    0557 2707
rcr2 0565
    0565 3204 3211
readeflags 0485
    0485 1659 1668 2163 6308
readi 4602
    0247 2768 4602 4766 5012
    5208 5209 5525 5536
readsb 4178
    4178 4211 4238 4309
readsect 1260
    1260 1295
readseg 1279
    1213 1226 1237 1279
REDIR 7558
    7558 7630 7770 8071
redircmd 7575 7764
    7575 7613 7631 7764 7766
    7975 7978 7981 8059 8072
REG_ID 6410
    6410 6460
REG_TABLE 6412
    6412 6467 6468 6481 6482
REG_VER 6411
    6411 6459
release 1602
    0324 1602 1605 1864 1870
    2077 2084 2135 2177 2186
    2219 2232 2268 2286 2290
    2469 2484 3169 3526 3531
    3544 3909 3928 3983 4078
    4092 4141 4364 4380 4392
    4414 4442 4460 4469 4929
    4933 4945 4960 4966 5672
    5675 5688 5697 5708 5719
    6898 7048 7062 7082 7109
ROOTDEV 0158
    0158 4859
ROOTINO 3707
    3707 4859
run 2410
    2311 2410 2411 2416 2457
    2466 2478
runcmd 7606
    7606 7620 7637 7643 7645
    7659 7666 7677 7725
RUNNING 1768
    1768 2127 2161 2311 3223
safestrcpy 5832
    0332 1922 1984 5580 5832
sched 2153
    0309 2039 2153 2158 2160
    2162 2164 2176 2225
scheduler 2108

    0308 1390 1713 2108 2128
    2166
SCROLLLOCK 6614
    6614 6647
SECTSIZE 1211
    1211 1273 1286 1289 1294
SEG 0768
    0768 2582 2583 2584 2585
    2588
SEG16 0772
    0772 2725
SEG_ASM 0660
    0660 1084 1085 1179 1180
    7370 7371
segdesc 0751
    0450 0453 0751 0768 0772
    1715
seginit 2573
    0362 1320 1383 2573
SEG_KCODE 1007 1121 1702 3050 7320
    1053 1150 1702 2582 3121
    3122 7345
SEG_KCPU 1704 3052
    1704 2588 2591 3068
SEG_KDATA 1008 1122 1703 3051 7321
    1058 1154 1703 2583 2727
    3065 7352
SEG_NULLASM 0654
    0654 1083 1178 7369
SEG_TSS 1707
    1707 2725 2726 2729
SEG_UCODE 1705
    1705 1914 2584
SEG_UDATA 1706
    1706 1915 2585
SETGATE 0921
    0921 3121 3122
setupkvm 2685
    0365 1909 2567 2685 2860
    5530
SHIFT 6608
    6608 6636 6637 6785
skipelem 4815
    4815 4863
sleep 2203
    0310 2089 2203 2206 2209
    2309 3529 3980 4081 4412
    5692 5711 7066 7479
spinlock 1501
    0206 0310 0320 0322 0323
    0324 0354 1501 1558 1561
    1573 1602 1644 1806 1809
    2203 2408 2415 3107 3112
    3809 3824 4026 4030 4168
    4284 4905 4909 5607 5612
    6808 6820 7002 7206
STACK 7318
    7361 7377
STA_R 0669 0784
    0669 0784 1084 1179 2582
    2584 7370
start 1014 1128 7407
    1013 1014 1067 1127 1128
    1163 1166 7406 7407
stat 3654
    0207 0230 0248 3654 4165
    4585 4976 5053 5154 7503
stati 4585
    0248 4585 4980
STA_W 0668 0783
    0668 0783 1085 1180 2583
    2585 2588 7371
STA_X 0665 0780
    0665 0780 1084 1179 2582
    2584 7370
sti 0523
    0523 0525 1673 2114
stosb 0442
    0442 0444 1239 5756
strlen 5851
    0333 5560 5562 5851 7719
    7923
strncmp 5808
    0334 4705 5808
strncpy 5818
    0335 4772 5818
STS_IG32 0798
    0798 0927
STS_T32A 0795
    0795 2725
STS_TG32 0799
    0799 0927
sum 6025
    6025 6027 6029 6031 6032
    6043 6092
superblock 3711
    3711 4178 4208 4233 4307
SVR 6214
    6214 6258
switchkvm 2715

    0374 2129 2706 2715
switchuvm 2722
    0373 1944 2126 2722 2731
    5588
swtch 2358
    0317 2128 2166 2357 2358
syscall 3425
    0343 3157 3306 3425
SYSCALL 7453 7460 7461 7462 7463 74
    7460 7461 7462 7463 7464
    7465 7466 7467 7468 7469
    7470 7471 7472 7473 7474
    7475 7476 7477 7478 7479
    7480
sys_chdir 5418
    3379 3401 5418
SYS_chdir 3266
    3266 3401
sys_close 5139
    3380 3402 5139
SYS_close 3257
    3257 3402
sys_dup 5101
    3381 3403 5101
SYS_dup 3267
    3267 3403
sys_exec 5451
    3382 3404 5451
SYS_exec 3259
    3259 3404 7411
sys_exit 3464
    3383 3405 3464
SYS_exit 3252
    3252 3405 7416
sys_fork 3458
    3384 3406 3458
SYS_fork 3251
    3251 3406
sys_fstat 5151
    3385 3407 5151
SYS_fstat 3263
    3263 3407
sys_getpid 3487
    3386 3408 3487
SYS_getpid 3268
    3268 3408
sys_kill 3477
    3387 3409 3477
SYS_kill 3258
    3258 3409
sys_link 5163
    3388 3410 5163
SYS_link 3264
    3264 3410
sys_mkdir 5390
    3389 3411 5390
SYS_mkdir 3265
    3265 3411
sys_mknod 5401
    3390 3412 5401
SYS_mknod 3261
    3261 3412
sys_open 5351
    3391 3413 5351
SYS_open 3260
    3260 3413
sys_pipe 5477
    3392 3414 5477
SYS_pipe 3254
    3254 3414
sys_read 5115
    3393 3415 5115
SYS_read 3256
    3256 3415
sys_sbrk 3501
    3394 3416 3501
SYS_sbrk 3269
    3269 3416
sys_sleep 3515
    3395 3417 3515
SYS_sleep 3270
    3270 3417
sys_unlink 5251
    3396 3418 5251
SYS_unlink 3262
    3262 3418
sys_uptime 3538
    3399 3421 3538
SYS_uptime 3271
    3271 3421
sys_wait 3471
    3397 3419 3471
SYS_wait 3253
    3253 3419
sys_write 5127
    3398 3420 5127
SYS_write 3255
    3255 3420
taskstate 0851
    0851 1714

TDCR 6238
    6238 6264
T_DEV 3652
    3652 4607 4657 5411
T_DIR 3650
    3650 4718 4865 5173 5278
    5287 5329 5367 5395 5426
T_FILE 3651
    3651 5314 5361
ticks 3113
    0352 3113 3167 3168 3523
    3524 3529 3543
tickslock 3112
    0354 3112 3124 3166 3169
    3522 3526 3529 3531 3542
    3544
TICR 6236
    6236 6266
TIMER 6228
    6228 6265
TIMER_16BIT 7171
    7171 7177
TIMER_DIV 7166
    7166 7178 7179
TIMER_FREQ 7165
    7165 7166
timerinit 7174
    0346 1368 7174
TIMER_MODE 7168
    7168 7177
TIMER_RATEGEN 7170
    7170 7177
TIMER_SEL0 7169
    7169 7177
T_IRQ0 2979
    2979 3164 3173 3177 3180
    3184 3188 3189 3223 6258
    6265 6278 6467 6481 6547
    6566
TPR 6212
    6212 6294
trap 3151
    3002 3004 3074 3151 3203
    3205 3208
trapframe 0602
    0602 1778 1881 3151
trapret 3079
    1817 1886 3078 3079
T_SYSCALL 2976
    2976 3122 3153 7412 7417
    7457
tvinit 3116
    0353 1362 3116
uart 7215
    7215 7236 7255 7265
uartgetc 7263
    7263 7275
uartinit 7218
    0357 1359 7218
uartintr 7273
    0358 3185 7273
uartputc 7251
    0359 6995 6997 7247 7251
userinit 1902
    0311 1369 1902 1910
USERTOP 0159
    0159 2677 2782 2836
uva2ka 2902
    0366 2902 2926
VER 6211
    6211 6274
vmenable 2702
    0364 1386 2702
wait 2053
    0312 2053 3473 7462 7533
    7644 7670 7671 7726
waitdisk 1251
    1251 1263 1272
wakeup 2264
    0313 2264 3168 3922 4139
    4441 4466 5666 5669 5691
    5696 5718 7042
wakeup1 2253
    1819 2026 2033 2253 2267
walkpgdir 2604
    2604 2637 2761 2815 2863
    2906
writei 4652
    0249 4652 4774 5032 5285
    5286
xchg 0529
    0529 1389 1582 1619
yield 2172
    0314 2172 3224

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



```
0150 #define NPROC        64  // maximum number of processes
0151 #define KSTACKSIZE 4096  // size of per-process kernel stack
0152 #define NCPU          8  // maximum number of CPUs
0153 #define NOFILE       16  // open files per process
0154 #define NFILE       100  // open files per system
0155 #define NBUF         10  // size of disk block cache
0156 #define NINODE       50  // maximum number of active i-nodes
0157 #define NDEV         10  // maximum major device number
0158 #define ROOTDEV       1  // device number of file system root disk
0159 #define USERTOP  0xA0000 // end of user address space
0160 #define PHYSTOP  0x1000000 // use phys mem up to here as free pool
0161 #define MAXARG       32  // max exec arguments
0162
0163
0164
0165
0166
0167
0168
0169
0170
0171
0172
0173
0174
0175
0176
0177
0178
0179
0180
0181
0182
0183
0184
0185
0186
0187
0188
0189
0190
0191
0192
0193
0194
0195
0196
0197
0198
0199
```

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



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

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



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

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



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

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



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

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



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

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



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

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



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

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



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

```
1000 #include "asm.h"
1001
1002 # Start the first CPU: switch to 32-bit protected mode, jump into C.
1003 # The BIOS loads this code from the first sector of the hard disk into
1004 # memory at physical address 0x7c00 and starts executing in real mode
1005 # with %cs=0 %ip=7c00.
1006
1007 #define SEG_KCODE 1  // kernel code
1008 #define SEG_KDATA 2  // kernel data+stack
1009
1010 #define CR0_PE    1  // protected mode enable bit
1011
1012 .code16                       # Assemble for 16-bit mode
1013 .globl start
1014 start:
1015   cli                         # BIOS enabled interrupts ; disable
1016
1017   # Set up the important data segment registers (DS, ES, SS).
1018   xorw    %ax,%ax             # Segment number zero
1019   movw    %ax,%ds             # -> Data Segment
1020   movw    %ax,%es             # -> Extra Segment
1021   movw    %ax,%ss             # -> Stack Segment
1022
1023   # Physical address line A20 is tied to zero so that the first PCs
1024   # with 2 MB would run software that assumed 1 MB.  Undo that.
1025 seta20.1:
1026   inb     $0x64,%al               # Wait for not busy
1027   testb   $0x2,%al
1028   jnz     seta20.1
1029
1030   movb    $0xd1,%al               # 0xd1 -> port 0x64
1031   outb    %al,$0x64
1032
1033 seta20.2:
1034   inb     $0x64,%al               # Wait for not busy
1035   testb   $0x2,%al
1036   jnz     seta20.2
1037
1038   movb    $0xdf,%al               # 0xdf -> port 0x60
1039   outb    %al,$0x60
1040
1041   # Switch from real to protected mode.  Use a bootstrap GDT that makes
1042   # virtual addresses map dierctly to  physical addresses so that the
1043   # effective memory map doesn't change during the transition.
1044   lgdt    gdtdesc
1045   movl    %cr0, %eax
1046   orl     $CR0_PE, %eax
1047   movl    %eax, %cr0
1048
1049
```



```
1050   # Complete transition to 32-bit protected mode by using long jmp
1051   # to reload %cs and %eip.  The segment registers are set up with no
1052   # translation, so that the mapping is still the identity mapping.
1053   ljmp    $(SEG_KCODE<<3), $start32
1054
1055 .code32  # Tell assembler to generate 32-bit code now.
1056 start32:
1057   # Set up the protected-mode data segment registers
1058   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
1059   movw    %ax, %ds                # -> DS: Data Segment
1060   movw    %ax, %es                # -> ES: Extra Segment
1061   movw    %ax, %ss                # -> SS: Stack Segment
1062   movw    $0, %ax                 # Zero segments not ready for use
1063   movw    %ax, %fs                # -> FS
1064   movw    %ax, %gs                # -> GS
1065
1066   # Set up the stack pointer and call into C.
1067   movl    $start, %esp
1068   call    bootmain
1069
1070   # If bootmain returns (it shouldn't), trigger a Bochs
1071   # breakpoint if running under Bochs, then loop.
1072   movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
1073   movw    %ax, %dx
1074   outw    %ax, %dx
1075   movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
1076   outw    %ax, %dx
1077 spin:
1078   jmp     spin
1079
1080 # Bootstrap GDT
1081 .p2align 2                                # force 4 byte alignment
1082 gdt:
1083   SEG_NULLASM                             # null seg
1084   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
1085   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
1086
1087 gdtdesc:
1088   .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
1089   .long   gdt                             # address gdt
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
```

```
1100 #include "asm.h"
1101
1102 # Each non-boot CPU ("AP") is started up in response to a STARTUP
1103 # IPI from the boot CPU.  Section B.4.2 of the Multi-Processor
1104 # Specification says that the AP will start in real mode with CS:IP
1105 # set to XY00:0000, where XY is an 8-bit value sent with the
1106 # STARTUP. Thus this code must start at a 4096-byte boundary.
1107 #
1108 # Because this code sets DS to zero, it must sit
1109 # at an address in the low 2^16 bytes.
1110 #
1111 # Bootothers (in main.c) sends the STARTUPs one at a time.
1112 # It copies this code (start) at 0x7000.
1113 # It puts the address of a newly allocated per-core stack in start-4,
1114 # and the address of the place to jump to (mpmain) in start-8.
1115 #
1116 # This code is identical to bootasm.S except:
1117 #   - it does not need to enable A20
1118 #   - it uses the address at start-4 for the %esp
1119 #   - it jumps to the address at start-8 instead of calling bootmain
1120
1121 #define SEG_KCODE 1
1122 #define SEG_KDATA 2
1123
1124 #define CR0_PE    1
1125
1126 .code16
1127 .globl start
1128 start:
1129   cli
1130
1131   xorw    %ax,%ax
1132   movw    %ax,%ds
1133   movw    %ax,%es
1134   movw    %ax,%ss
1135
1136   lgdt    gdtdesc
1137   movl    %cr0, %eax
1138   orl     $CR0_PE, %eax
1139   movl    %eax, %cr0
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
```



```
1150   ljmp    $(SEG_KCODE<<3), $start32
1151
1152 .code32
1153 start32:
1154   movw    $(SEG_KDATA<<3), %ax
1155   movw    %ax, %ds
1156   movw    %ax, %es
1157   movw    %ax, %ss
1158   movw    $0, %ax
1159   movw    %ax, %fs
1160   movw    %ax, %gs
1161
1162   # switch to the stack allocated by bootothers()
1163   movl    start-4, %esp
1164
1165   # call mpmain()
1166   call     *(start-8)
1167
1168   movw    $0x8a00, %ax
1169   movw    %ax, %dx
1170   outw    %ax, %dx
1171   movw    $0x8ae0, %ax
1172   outw    %ax, %dx
1173 spin:
1174   jmp     spin
1175
1176 .p2align 2
1177 gdt:
1178   SEG_NULLASM
1179   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)
1180   SEG_ASM(STA_W, 0x0, 0xffffffff)
1181
1182 gdtdesc:
1183   .word   (gdtdesc - gdt - 1)
1184   .long   gdt
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
```

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



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

```
1300 #include "types.h"
1301 #include "defs.h"
1302 #include "param.h"
1303 #include "mmu.h"
1304 #include "proc.h"
1305 #include "x86.h"
1306
1307 static void bootothers(void);
1308 static void mpmain(void);
1309 void jmpkstack(void)  __attribute__((noreturn));
1310 void mainc(void);
1311
1312 // Bootstrap processor starts running C code here.
1313 // Allocate a real stack and switch to it, first
1314 // doing some setup required for memory allocator to work.
1315 int
1316 main(void)
1317 {
1318   mpinit();        // collect info about this machine
1319   lapicinit(mpbcpu());
1320   seginit();       // set up segments
1321   kinit();         // initialize memory allocator
1322   jmpkstack();       // call mainc() on a properly-allocated stack
1323 }
1324
1325 void
1326 jmpkstack(void)
1327 {
1328   char *kstack, *top;
1329
1330   kstack = kalloc();
1331   if(kstack == 0)
1332     panic("jmpkstack kalloc");
1333   top = kstack + PGSIZE;
1334   asm volatile("movl %0,%%esp; call mainc" : : "r" (top));
1335   panic("jmpkstack");
1336 }
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
```



```
1350 // Set up hardware and software.
1351 // Runs only on the boostrap processor.
1352 void
1353 mainc(void)
1354 {
1355   cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
1356   picinit();       // interrupt controller
1357   ioapicinit();    // another interrupt controller
1358   consoleinit();   // I/O devices & their interrupts
1359   uartinit();      // serial port
1360   kvmalloc();      // initialize the kernel page table
1361   pinit();         // process table
1362   tvinit();        // trap vectors
1363   binit();         // buffer cache
1364   fileinit();      // file table
1365   iinit();         // inode cache
1366   ideinit();       // disk
1367   if(!ismp)
1368     timerinit();   // uniprocessor timer
1369   userinit();      // first user process
1370   bootothers();    // start other processors
1371
1372   // Finish setting up this processor in mpmain.
1373   mpmain();
1374 }
1375
1376 // Common CPU setup code.
1377 // Bootstrap CPU comes here from mainc().
1378 // Other CPUs jump here from bootother.S.
1379 static void
1380 mpmain(void)
1381 {
1382   if(cpunum() != mpbcpu()){
1383     seginit();
1384     lapicinit(cpunum());
1385   }
1386   vmenable();        // turn on paging
1387   cprintf("cpu%d: starting\n", cpu->id);
1388   idtinit();       // load idt register
1389   xchg(&cpu->booted, 1); // tell bootothers() we're up
1390   scheduler();     // start running processes
1391 }
1392
1393
1394
1395
1396
1397
1398
1399
```

```
// Start the non-boot processors.
static void
bootothers(void)
{
  extern uchar _binary_bootother_start[], _binary_bootother_size[];
  uchar *code;
  struct cpu *c;
  char *stack;

  // Write bootstrap code to unused memory at 0x7000.
  // The linker has placed the image of bootother.S in
  // _binary_bootother_start.
  code = (uchar*)0x7000;
  memmove(code, _binary_bootother_start, (uint)_binary_bootother_size);

  for(c = cpus; c < cpus+ncpu; c++){
    if(c == cpus+cpunum())  // We've started already.
      continue;

    // Tell bootother.S what stack to use and the address of mpmain;
    // it expects to find these two addresses stored just before
    // its first instruction.
    stack = kalloc();
    *(void**)(code-4) = stack + KSTACKSIZE;
    *(void**)(code-8) = mpmain;

    lapicstartap(c->id, (uint)code);

    // Wait for cpu to finish mpmain()
    while(c->booted == 0)
      ;
  }
}
```



```
// Blank page.
```

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



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

```
1600 // Release the lock.
1601 void
1602 release(struct spinlock *lk)
1603 {
1604   if(!holding(lk))
1605     panic("release");
1606
1607   lk->pcs[0] = 0;
1608   lk->cpu = 0;
1609
1610   // The xchg serializes, so that reads before release are
1611   // not reordered after it.  The 1996 PentiumPro manual (Volume 3,
1612   // 7.2) says reads can be carried out speculatively and in
1613   // any order, which implies we need to serialize here.
1614   // But the 2007 Intel 64 Architecture Memory Ordering White
1615   // Paper says that Intel 64 and IA-32 will not move a load
1616   // after a store. So lock->locked = 0 would work here.
1617   // The xchg being asm volatile ensures gcc emits it after
1618   // the above assignments (and after the critical section).
1619   xchg(&lk->locked, 0);
1620
1621   popcli();
1622 }
1623
1624 // Record the current call stack in pcs[] by following the %ebp chain.
1625 void
1626 getcallerpcs(void *v, uint pcs[])
1627 {
1628   uint *ebp;
1629   int i;
1630
1631   ebp = (uint*)v - 2;
1632   for(i = 0; i < 10; i++){
1633     if(ebp == 0 || ebp < (uint*)0x100000 || ebp == (uint*)0xffffffff)
1634       break;
1635     pcs[i] = ebp[1];     // saved %eip
1636     ebp = (uint*)ebp[0]; // saved %ebp
1637   }
1638   for(; i < 10; i++)
1639     pcs[i] = 0;
1640 }
1641
1642 // Check whether this cpu is holding the lock.
1643 int
1644 holding(struct spinlock *lock)
1645 {
1646   return lock->locked && lock->cpu == cpu;
1647 }
1648
1649
```



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

```
1700 // Segments in proc->gdt.
1701 // Also known to bootasm.S and trapasm.S
1702 #define SEG_KCODE 1  // kernel code
1703 #define SEG_KDATA 2  // kernel data+stack
1704 #define SEG_KCPU  3  // kernel per-cpu data
1705 #define SEG_UCODE 4  // user code
1706 #define SEG_UDATA 5  // user data+stack
1707 #define SEG_TSS   6  // this process's task state
1708 #define NSEGS     7
1709
1710 // Per-CPU state
1711 struct cpu {
1712   uchar id;                    // Local APIC ID; index into cpus[] below
1713   struct context *scheduler;   // swtch() here to enter scheduler
1714   struct taskstate ts;         // Used by x86 to find stack for interrupt
1715   struct segdesc gdt[NSEGS];   // x86 global descriptor table
1716   volatile uint booted;        // Has the CPU started?
1717   int ncli;                    // Depth of pushcli nesting.
1718   int intena;                  // Were interrupts enabled before pushcli?
1719
1720   // Cpu-local storage variables; see below
1721   struct cpu *cpu;
1722   struct proc *proc;           // The currently-running process.
1723 };
1724
1725 extern struct cpu cpus[NCPU];
1726 extern int ncpu;
1727
1728 // Per-CPU variables, holding pointers to the
1729 // current cpu and to the current process.
1730 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
1731 // and "%gs:4" to refer to proc.  seginit sets up the
1732 // %gs segment register so that %gs refers to the memory
1733 // holding those two variables in the local cpu's struct cpu.
1734 // This is similar to how thread-local variables are implemented
1735 // in thread libraries such as Linux pthreads.
1736 extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
1737 extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
```



```
1750 // Saved registers for kernel context switches.
1751 // Don't need to save all the segment registers (%cs, etc),
1752 // because they are constant across kernel contexts.
1753 // Don't need to save %eax, %ecx, %edx, because the
1754 // x86 convention is that the caller has saved them.
1755 // Contexts are stored at the bottom of the stack they
1756 // describe; the stack pointer is the address of the context.
1757 // The layout of the context matches the layout of the stack in swtch.S
1758 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
1759 // but it is on the stack and allocproc() manipulates it.
1760 struct context {
1761   uint edi;
1762   uint esi;
1763   uint ebx;
1764   uint ebp;
1765   uint eip;
1766 };
1767
1768 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
1769
1770 // Per-process state
1771 struct proc {
1772   uint sz;                     // Size of process memory (bytes)
1773   pde_t* pgdir;                // Page table
1774   char *kstack;                // Bottom of kernel stack for this process
1775   enum procstate state;        // Process state
1776   volatile int pid;            // Process ID
1777   struct proc *parent;         // Parent process
1778   struct trapframe *tf;        // Trap frame for current syscall
1779   struct context *context;     // swtch() here to run process
1780   void *chan;                  // If non-zero, sleeping on chan
1781   int killed;                  // If non-zero, have been killed
1782   struct file *ofile[NOFILE];  // Open files
1783   struct inode *cwd;           // Current directory
1784   char name[16];               // Process name (debugging)
1785 };
1786
1787 // Process memory is laid out contiguously, low addresses first:
1788 //   text
1789 //   original data and bss
1790 //   fixed-size stack
1791 //   expandable heap
1792
1793
1794
1795
1796
1797
1798
1799
```

```
1800 #include "types.h"
1801 #include "defs.h"
1802 #include "param.h"
1803 #include "mmu.h"
1804 #include "x86.h"
1805 #include "proc.h"
1806 #include "spinlock.h"
1807
1808 struct {
1809   struct spinlock lock;
1810   struct proc proc[NPROC];
1811 } ptable;
1812
1813 static struct proc *initproc;
1814
1815 int nextpid = 1;
1816 extern void forkret(void);
1817 extern void trapret(void);
1818
1819 static void wakeup1(void *chan);
1820
1821 void
1822 pinit(void)
1823 {
1824   initlock(&ptable.lock, "ptable");
1825 }
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
```



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

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



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

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



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

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



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

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



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

```
2300 // Print a process listing to console.  For debugging.
2301 // Runs when user types ^P on console.
2302 // No lock to avoid wedging a stuck machine further.
2303 void
2304 procdump(void)
2305 {
2306   static char *states[] = {
2307   [UNUSED]    "unused",
2308   [EMBRYO]    "embryo",
2309   [SLEEPING]  "sleep ",
2310   [RUNNABLE]  "runble",
2311   [RUNNING]   "run   ",
2312   [ZOMBIE]    "zombie"
2313   };
2314   int i;
2315   struct proc *p;
2316   char *state;
2317   uint pc[10];
2318
2319   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2320     if(p->state == UNUSED)
2321       continue;
2322     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
2323       state = states[p->state];
2324     else
2325       state = "???";
2326     cprintf("%d %s %s", p->pid, state, p->name);
2327     if(p->state == SLEEPING){
2328       getcallerpcs((uint*)p->context->ebp+2, pc);
2329       for(i=0; i<10 && pc[i] != 0; i++)
2330         cprintf(" %p", pc[i]);
2331     }
2332     cprintf("\n");
2333   }
2334 }
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
```



```
2350 # Context switch
2351 #
2352 #   void swtch(struct context **old, struct context *new);
2353 #
2354 # Save current register context in old
2355 # and then load register context from new.
2356
2357 .globl swtch
2358 swtch:
2359   movl 4(%esp), %eax
2360   movl 8(%esp), %edx
2361
2362   # Save old callee-save registers
2363   pushl %ebp
2364   pushl %ebx
2365   pushl %esi
2366   pushl %edi
2367
2368   # Switch stacks
2369   movl %esp, (%eax)
2370   movl %edx, %esp
2371
2372   # Load new callee-save registers
2373   popl %edi
2374   popl %esi
2375   popl %ebx
2376   popl %ebp
2377   ret
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
```

```
2400 // Physical memory allocator, intended to allocate
2401 // memory for user processes, kernel stacks, page table pages,
2402 // and pipe buffers. Allocates 4096-byte pages.
2403
2404 #include "types.h"
2405 #include "defs.h"
2406 #include "param.h"
2407 #include "mmu.h"
2408 #include "spinlock.h"
2409
2410 struct run {
2411   struct run *next;
2412 };
2413
2414 struct {
2415   struct spinlock lock;
2416   struct run *freelist;
2417 } kmem;
2418
2419 extern char end[]; // first address after kernel loaded from ELF file
2420
2421 // Initialize free list of physical pages.
2422 void
2423 kinit(void)
2424 {
2425   char *p;
2426
2427   initlock(&kmem.lock, "kmem");
2428   p = (char*)PGROUNDUP((uint)end);
2429   for(; p + PGSIZE <= (char*)PHYSTOP; p += PGSIZE)
2430     kfree(p);
2431 }
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
```



```
2450 // Free the page of physical memory pointed at by v,
2451 // which normally should have been returned by a
2452 // call to kalloc().  (The exception is when
2453 // initializing the allocator; see kinit above.)
2454 void
2455 kfree(char *v)
2456 {
2457   struct run *r;
2458
2459   if((uint)v % PGSIZE || v < end || (uint)v >= PHYSTOP)
2460     panic("kfree");
2461
2462   // Fill with junk to catch dangling refs.
2463   memset(v, 1, PGSIZE);
2464
2465   acquire(&kmem.lock);
2466   r = (struct run*)v;
2467   r->next = kmem.freelist;
2468   kmem.freelist = r;
2469   release(&kmem.lock);
2470 }
2471
2472 // Allocate one 4096-byte page of physical memory.
2473 // Returns a pointer that the kernel can use.
2474 // Returns 0 if the memory cannot be allocated.
2475 char*
2476 kalloc(void)
2477 {
2478   struct run *r;
2479
2480   acquire(&kmem.lock);
2481   r = kmem.freelist;
2482   if(r)
2483     kmem.freelist = r->next;
2484   release(&kmem.lock);
2485   return (char*)r;
2486 }
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
```

```
2500 // The kernel layout is:
2501 //
2502 //     text
2503 //     rodata
2504 //     data
2505 //     bss
2506 //
2507 // Conventionally, Unix linkers provide pseudo-symbols
2508 // etext, edata, and end, at the end of the text, data, and bss.
2509 // For the kernel mapping, we need the address at the beginning
2510 // of the data section, but that's not one of the conventional
2511 // symbols, because the convention started before there was a
2512 // read-only rodata section between text and data.
2513 //
2514 // To get the address of the data section, we define a symbol
2515 // named data and make sure this is the first object passed to
2516 // the linker, so that it will be the first symbol in the data section.
2517 //
2518 // Alternative approaches would be to parse our own ELF header
2519 // or to write a linker script, but this is simplest.
2520
2521 .data
2522 .globl data
2523 data:
2524   .word 1
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
```



```
2550 #include "param.h"
2551 #include "types.h"
2552 #include "defs.h"
2553 #include "x86.h"
2554 #include "mmu.h"
2555 #include "proc.h"
2556 #include "elf.h"
2557
2558 extern char data[];  // defined in data.S
2559
2560 static pde_t *kpgdir;  // for use in scheduler()
2561
2562 // Allocate one page table for the machine for the kernel address
2563 // space for scheduler processes.
2564 void
2565 kvmalloc(void)
2566 {
2567   kpgdir = setupkvm();
2568 }
2569
2570 // Set up CPU's kernel segment descriptors.
2571 // Run once at boot time on each CPU.
2572 void
2573 seginit(void)
2574 {
2575   struct cpu *c;
2576
2577   // Map virtual addresses to linear addresses using identity map.
2578   // Cannot share a CODE descriptor for both kernel and user
2579   // because it would have to have DPL_USR, but the CPU forbids
2580   // an interrupt from CPL=0 to DPL=3.
2581   c = &cpus[cpunum()];
2582   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
2583   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
2584   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
2585   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
2586
2587   // Map cpu, and curproc
2588   c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
2589
2590   lgdt(c->gdt, sizeof(c->gdt));
2591   loadgs(SEG_KCPU << 3);
2592
2593   // Initialize cpu-local storage.
2594   cpu = c;
2595   proc = 0;
2596 }
2597
2598
2599
```

```
2600 // Return the address of the PTE in page table pgdir
2601 // that corresponds to linear address va.  If create!=0,
2602 // create any required page table pages.
2603 static pte_t *
2604 walkpgdir(pde_t *pgdir, const void *va, int create)
2605 {
2606   pde_t *pde;
2607   pte_t *pgtab;
2608
2609   pde = &pgdir[PDX(va)];
2610   if(*pde & PTE_P){
2611     pgtab = (pte_t*)PTE_ADDR(*pde);
2612   } else {
2613     if(!create || (pgtab = (pte_t*)kalloc()) == 0)
2614       return 0;
2615     // Make sure all those PTE_P bits are zero.
2616     memset(pgtab, 0, PGSIZE);
2617     // The permissions here are overly generous, but they can
2618     // be further restricted by the permissions in the page table
2619     // entries, if necessary.
2620     *pde = PADDR(pgtab) | PTE_P | PTE_W | PTE_U;
2621   }
2622   return &pgtab[PTX(va)];
2623 }
2624
2625 // Create PTEs for linear addresses starting at la that refer to
2626 // physical addresses starting at pa. la and size might not
2627 // be page-aligned.
2628 static int
2629 mappages(pde_t *pgdir, void *la, uint size, uint pa, int perm)
2630 {
2631   char *a, *last;
2632   pte_t *pte;
2633
2634   a = PGROUNDDOWN(la);
2635   last = PGROUNDDOWN(la + size - 1);
2636   for(;;){
2637     pte = walkpgdir(pgdir, a, 1);
2638     if(pte == 0)
2639       return -1;
2640     if(*pte & PTE_P)
2641       panic("remap");
2642     *pte = pa | perm | PTE_P;
2643     if(a == last)
2644       break;
2645     a += PGSIZE;
2646     pa += PGSIZE;
2647   }
2648   return 0;
2649 }
```



```
2650 // The mappings from logical to linear are one to one (i.e.,
2651 // segmentation doesn't do anything).
2652 // There is one page table per process, plus one that's used
2653 // when a CPU is not running any process (kpgdir).
2654 // A user process uses the same page table as the kernel; the
2655 // page protection bits prevent it from using anything other
2656 // than its memory.
2657 //
2658 // setupkvm() and exec() set up every page table like this:
2659 //   0..640K          : user memory (text, data, stack, heap)
2660 //   640K..1M         : mapped direct (for IO space)
2661 //   1M..end          : mapped direct (for the kernel's text and data)
2662 //   end..PHYSTOP     : mapped direct (kernel heap and user pages)
2663 //   0xfe000000..0    : mapped direct (devices such as ioapic)
2664 //
2665 // The kernel allocates memory for its heap and for user memory
2666 // between kernend and the end of physical memory (PHYSTOP).
2667 // The virtual address space of each user program includes the kernel
2668 // (which is inaccessible in user mode).  The user program addresses
2669 // range from 0 till 640KB (USERTOP), which where the I/O hole starts
2670 // (both in physical memory and in the kernel's virtual address
2671 // space).
2672 static struct kmap {
2673   void *p;
2674   void *e;
2675   int perm;
2676 } kmap[] = {
2677   {(void*)USERTOP,    (void*)0x100000, PTE_W},  // I/O space
2678   {(void*)0x100000,   data,            0    },  // kernel text, rodata
2679   {data,              (void*)PHYSTOP,  PTE_W},  // kernel data, memory
2680   {(void*)0xFE000000, 0,               PTE_W},  // device mappings
2681 };
2682
2683 // Set up kernel part of a page table.
2684 pde_t*
2685 setupkvm(void)
2686 {
2687   pde_t *pgdir;
2688   struct kmap *k;
2689
2690   if((pgdir = (pde_t*)kalloc()) == 0)
2691     return 0;
2692   memset(pgdir, 0, PGSIZE);
2693   k = kmap;
2694   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
2695     if(mappages(pgdir, k->p, k->e - k->p, (uint)k->p, k->perm) < 0)
2696       return 0;
2697
2698   return pgdir;
2699 }
```

```
2700 // Turn on paging.
2701 void
2702 vmenable(void)
2703 {
2704   uint cr0;
2705
2706   switchkvm(); // load kpgdir into cr3
2707   cr0 = rcr0();
2708   cr0 |= CR0_PG;
2709   lcr0(cr0);
2710 }
2711
2712 // Switch h/w page table register to the kernel-only page table,
2713 // for when no process is running.
2714 void
2715 switchkvm(void)
2716 {
2717   lcr3(PADDR(kpgdir));   // switch to the kernel page table
2718 }
2719
2720 // Switch TSS and h/w page table to correspond to process p.
2721 void
2722 switchuvm(struct proc *p)
2723 {
2724   pushcli();
2725   cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
2726   cpu->gdt[SEG_TSS].s = 0;
2727   cpu->ts.ss0 = SEG_KDATA << 3;
2728   cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
2729   ltr(SEG_TSS << 3);
2730   if(p->pgdir == 0)
2731     panic("switchuvm: no pgdir");
2732   lcr3(PADDR(p->pgdir));  // switch to new address space
2733   popcli();
2734 }
2735
2736 // Load the initcode into address 0 of pgdir.
2737 // sz must be less than a page.
2738 void
2739 inituvm(pde_t *pgdir, char *init, uint sz)
2740 {
2741   char *mem;
2742
2743   if(sz >= PGSIZE)
2744     panic("inituvm: more than a page");
2745   mem = kalloc();
2746   memset(mem, 0, PGSIZE);
2747   mappages(pgdir, 0, PGSIZE, PADDR(mem), PTE_W|PTE_U);
2748   memmove(mem, init, sz);
2749 }
```



```
2750 // Load a program segment into pgdir.  addr must be page-aligned
2751 // and the pages from addr to addr+sz must already be mapped.
2752 int
2753 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
2754 {
2755   uint i, pa, n;
2756   pte_t *pte;
2757
2758   if((uint)addr % PGSIZE != 0)
2759     panic("loaduvm: addr must be page aligned");
2760   for(i = 0; i < sz; i += PGSIZE){
2761     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
2762       panic("loaduvm: address should exist");
2763     pa = PTE_ADDR(*pte);
2764     if(sz - i < PGSIZE)
2765       n = sz - i;
2766     else
2767       n = PGSIZE;
2768     if(readi(ip, (char*)pa, offset+i, n) != n)
2769       return -1;
2770   }
2771   return 0;
2772 }
2773
2774 // Allocate page tables and physical memory to grow process from oldsz to
2775 // newsz, which need not be page aligned.  Returns new size or 0 on error.
2776 int
2777 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
2778 {
2779   char *mem;
2780   uint a;
2781
2782   if(newsz > USERTOP)
2783     return 0;
2784   if(newsz < oldsz)
2785     return oldsz;
2786
2787   a = PGROUNDUP(oldsz);
2788   for(; a < newsz; a += PGSIZE){
2789     mem = kalloc();
2790     if(mem == 0){
2791       cprintf("allocuvm out of memory\n");
2792       deallocuvm(pgdir, newsz, oldsz);
2793       return 0;
2794     }
2795     memset(mem, 0, PGSIZE);
2796     mappages(pgdir, (char*)a, PGSIZE, PADDR(mem), PTE_W|PTE_U);
2797   }
2798   return newsz;
2799 }
```

```
2800 // Deallocate user pages to bring the process size from oldsz to
2801 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
2802 // need to be less than oldsz.  oldsz can be larger than the actual
2803 // process size.  Returns the new process size.
2804 int
2805 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
2806 {
2807   pte_t *pte;
2808   uint a, pa;
2809
2810   if(newsz >= oldsz)
2811     return oldsz;
2812
2813   a = PGROUNDUP(newsz);
2814   for(; a  < oldsz; a += PGSIZE){
2815     pte = walkpgdir(pgdir, (char*)a, 0);
2816     if(pte && (*pte & PTE_P) != 0){
2817       pa = PTE_ADDR(*pte);
2818       if(pa == 0)
2819         panic("kfree");
2820       kfree((char*)pa);
2821       *pte = 0;
2822     }
2823   }
2824   return newsz;
2825 }
2826
2827 // Free a page table and all the physical memory pages
2828 // in the user part.
2829 void
2830 freevm(pde_t *pgdir)
2831 {
2832   uint i;
2833
2834   if(pgdir == 0)
2835     panic("freevm: no pgdir");
2836   deallocuvm(pgdir, USERTOP, 0);
2837   for(i = 0; i < NPDENTRIES; i++){
2838     if(pgdir[i] & PTE_P)
2839       kfree((char*)PTE_ADDR(pgdir[i]));
2840   }
2841   kfree((char*)pgdir);
2842 }
2843
2844
2845
2846
2847
2848
2849
```



```
2850 // Given a parent process's page table, create a copy
2851 // of it for a child.
2852 pde_t*
2853 copyuvm(pde_t *pgdir, uint sz)
2854 {
2855   pde_t *d;
2856   pte_t *pte;
2857   uint pa, i;
2858   char *mem;
2859
2860   if((d = setupkvm()) == 0)
2861     return 0;
2862   for(i = 0; i < sz; i += PGSIZE){
2863     if((pte = walkpgdir(pgdir, (void*)i, 0)) == 0)
2864       panic("copyuvm: pte should exist");
2865     if(!(*pte & PTE_P))
2866       panic("copyuvm: page not present");
2867     pa = PTE_ADDR(*pte);
2868     if((mem = kalloc()) == 0)
2869       goto bad;
2870     memmove(mem, (char*)pa, PGSIZE);
2871     if(mappages(d, (void*)i, PGSIZE, PADDR(mem), PTE_W|PTE_U) < 0)
2872       goto bad;
2873   }
2874   return d;
2875
2876 bad:
2877   freevm(d);
2878   return 0;
2879 }
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
```

```
2900 // Map user virtual address to kernel physical address.
2901 char*
2902 uva2ka(pde_t *pgdir, char *uva)
2903 {
2904   pte_t *pte;
2905
2906   pte = walkpgdir(pgdir, uva, 0);
2907   if((*pte & PTE_P) == 0)
2908     return 0;
2909   if((*pte & PTE_U) == 0)
2910     return 0;
2911   return (char*)PTE_ADDR(*pte);
2912 }
2913
2914 // Copy len bytes from p to user address va in page table pgdir.
2915 // Most useful when pgdir is not the current page table.
2916 // uva2ka ensures this only works for PTE_U pages.
2917 int
2918 copyout(pde_t *pgdir, uint va, void *p, uint len)
2919 {
2920   char *buf, *pa0;
2921   uint n, va0;
2922
2923   buf = (char*)p;
2924   while(len > 0){
2925     va0 = (uint)PGROUNDDOWN(va);
2926     pa0 = uva2ka(pgdir, (char*)va0);
2927     if(pa0 == 0)
2928       return -1;
2929     n = PGSIZE - (va - va0);
2930     if(n > len)
2931       n = len;
2932     memmove(pa0 + (va - va0), buf, n);
2933     len -= n;
2934     buf += n;
2935     va = va0 + PGSIZE;
2936   }
2937   return 0;
2938 }
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
```



```
2950 // x86 trap and interrupt constants.
2951
2952 // Processor-defined:
2953 #define T_DIVIDE         0      // divide error
2954 #define T_DEBUG          1      // debug exception
2955 #define T_NMI            2      // non-maskable interrupt
2956 #define T_BRKPT          3      // breakpoint
2957 #define T_OFLOW          4      // overflow
2958 #define T_BOUND          5      // bounds check
2959 #define T_ILLOP          6      // illegal opcode
2960 #define T_DEVICE         7      // device not available
2961 #define T_DBLFLT         8      // double fault
2962 // #define T_COPROC      9      // reserved (not used since 486)
2963 #define T_TSS           10      // invalid task switch segment
2964 #define T_SEGNP         11      // segment not present
2965 #define T_STACK         12      // stack exception
2966 #define T_GPFLT         13      // general protection fault
2967 #define T_PGFLT         14      // page fault
2968 // #define T_RES        15      // reserved
2969 #define T_FPERR         16      // floating point error
2970 #define T_ALIGN         17      // aligment check
2971 #define T_MCHK          18      // machine check
2972 #define T_SIMDERR       19      // SIMD floating point error
2973
2974 // These are arbitrarily chosen, but with care not to overlap
2975 // processor defined exceptions or interrupt vectors.
2976 #define T_SYSCALL       64      // system call
2977 #define T_DEFAULT      500      // catchall
2978
2979 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
2980
2981 #define IRQ_TIMER        0
2982 #define IRQ_KBD          1
2983 #define IRQ_COM1         4
2984 #define IRQ_IDE         14
2985 #define IRQ_ERROR       19
2986 #define IRQ_SPURIOUS    31
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
```

```
3000 #!/usr/bin/perl -w
3001
3002 # Generate vectors.S, the trap/interrupt entry points.
3003 # There has to be one entry point per interrupt number
3004 # since otherwise there's no way for trap() to discover
3005 # the interrupt number.
3006
3007 print "# generated by vectors.pl - do not edit\n";
3008 print "# handlers\n";
3009 print ".globl alltraps\n";
3010 for(my $i = 0; $i < 256; $i++){
3011     print ".globl vector$i\n";
3012     print "vector$i:\n";
3013     if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
3014         print "  pushl \$0\n";
3015     }
3016     print "  pushl \$$i\n";
3017     print "  jmp alltraps\n";
3018 }
3019
3020 print "\n# vector table\n";
3021 print ".data\n";
3022 print ".globl vectors\n";
3023 print "vectors:\n";
3024 for(my $i = 0; $i < 256; $i++){
3025     print "  .long vector$i\n";
3026 }
3027
3028 # sample output:
3029 #   # handlers
3030 #   .globl alltraps
3031 #   .globl vector0
3032 #   vector0:
3033 #     pushl $0
3034 #     pushl $0
3035 #     jmp alltraps
3036 #   ...
3037 #
3038 #   # vector table
3039 #   .data
3040 #   .globl vectors
3041 #   vectors:
3042 #     .long vector0
3043 #     .long vector1
3044 #     .long vector2
3045 #   ...
3046
3047
3048
3049
```



```
3050 #define SEG_KCODE 1  // kernel code
3051 #define SEG_KDATA 2  // kernel data+stack
3052 #define SEG_KCPU  3  // kernel per-cpu data
3053
3054   # vectors.S sends all traps here.
3055 .globl alltraps
3056 alltraps:
3057   # Build trap frame.
3058   pushl %ds
3059   pushl %es
3060   pushl %fs
3061   pushl %gs
3062   pushal
3063
3064   # Set up data and per-cpu segments.
3065   movw $(SEG_KDATA<<3), %ax
3066   movw %ax, %ds
3067   movw %ax, %es
3068   movw $(SEG_KCPU<<3), %ax
3069   movw %ax, %fs
3070   movw %ax, %gs
3071
3072   # Call trap(tf), where tf=%esp
3073   pushl %esp
3074   call trap
3075   addl $4, %esp
3076
3077   # Return falls through to trapret...
3078 .globl trapret
3079 trapret:
3080   popal
3081   popl %gs
3082   popl %fs
3083   popl %es
3084   popl %ds
3085   addl $0x8, %esp  # trapno and errcode
3086   iret
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
```

```
3100 #include "types.h"
3101 #include "defs.h"
3102 #include "param.h"
3103 #include "mmu.h"
3104 #include "proc.h"
3105 #include "x86.h"
3106 #include "traps.h"
3107 #include "spinlock.h"
3108
3109 // Interrupt descriptor table (shared by all CPUs).
3110 struct gatedesc idt[256];
3111 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
3112 struct spinlock tickslock;
3113 uint ticks;
3114
3115 void
3116 tvinit(void)
3117 {
3118   int i;
3119
3120   for(i = 0; i < 256; i++)
3121     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3122   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3123
3124   initlock(&tickslock, "time");
3125 }
3126
3127 void
3128 idtinit(void)
3129 {
3130   lidt(idt, sizeof(idt));
3131 }
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
```



```
3150 void
3151 trap(struct trapframe *tf)
3152 {
3153   if(tf->trapno == T_SYSCALL){
3154     if(proc->killed)
3155       exit();
3156     proc->tf = tf;
3157     syscall();
3158     if(proc->killed)
3159       exit();
3160     return;
3161   }
3162
3163   switch(tf->trapno){
3164   case T_IRQ0 + IRQ_TIMER:
3165     if(cpu->id == 0){
3166       acquire(&tickslock);
3167       ticks++;
3168       wakeup(&ticks);
3169       release(&tickslock);
3170     }
3171     lapiceoi();
3172     break;
3173   case T_IRQ0 + IRQ_IDE:
3174     ideintr();
3175     lapiceoi();
3176     break;
3177   case T_IRQ0 + IRQ_IDE+1:
3178     // Bochs generates spurious IDE1 interrupts.
3179     break;
3180   case T_IRQ0 + IRQ_KBD:
3181     kbdintr();
3182     lapiceoi();
3183     break;
3184   case T_IRQ0 + IRQ_COM1:
3185     uartintr();
3186     lapiceoi();
3187     break;
3188   case T_IRQ0 + 7:
3189   case T_IRQ0 + IRQ_SPURIOUS:
3190     cprintf("cpu%d: spurious interrupt at %x:%x\n",
3191             cpu->id, tf->cs, tf->eip);
3192     lapiceoi();
3193     break;
3194
3195
3196
3197
3198
3199
```

```
3200   default:
3201     if(proc == 0 || (tf->cs&3) == 0){
3202       // In kernel, it must be our mistake.
3203       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3204               tf->trapno, cpu->id, tf->eip, rcr2());
3205       panic("trap");
3206     }
3207     // In user space, assume process misbehaved.
3208     cprintf("pid %d %s: trap %d err %d on cpu %d "
3209             "eip 0x%x addr 0x%x--kill proc\n",
3210             proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
3211             rcr2());
3212     proc->killed = 1;
3213   }
3214
3215   // Force process exit if it has been killed and is in user space.
3216   // (If it is still executing in the kernel, let it keep running
3217   // until it gets to the regular system call return.)
3218   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3219     exit();
3220
3221   // Force process to give up CPU on clock tick.
3222   // If interrupts were on while locks held, would need to check nlock.
3223   if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3224     yield();
3225
3226   // Check if the process has been killed since we yielded
3227   if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3228     exit();
3229 }
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
```



```
3250 // System call numbers
3251 #define SYS_fork    1
3252 #define SYS_exit    2
3253 #define SYS_wait    3
3254 #define SYS_pipe    4
3255 #define SYS_write   5
3256 #define SYS_read    6
3257 #define SYS_close   7
3258 #define SYS_kill    8
3259 #define SYS_exec    9
3260 #define SYS_open   10
3261 #define SYS_mknod  11
3262 #define SYS_unlink 12
3263 #define SYS_fstat  13
3264 #define SYS_link   14
3265 #define SYS_mkdir  15
3266 #define SYS_chdir  16
3267 #define SYS_dup    17
3268 #define SYS_getpid 18
3269 #define SYS_sbrk   19
3270 #define SYS_sleep  20
3271 #define SYS_uptime 21
3272
3273
3274
3275
3276
3277
3278
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
```

```
3300 #include "types.h"
3301 #include "defs.h"
3302 #include "param.h"
3303 #include "mmu.h"
3304 #include "proc.h"
3305 #include "x86.h"
3306 #include "syscall.h"
3307
3308 // User code makes a system call with INT T_SYSCALL.
3309 // System call number in %eax.
3310 // Arguments on the stack, from the user call to the C
3311 // library system call function. The saved user %esp points
3312 // to a saved program counter, and then the first argument.
3313
3314 // Fetch the int at addr from process p.
3315 int
3316 fetchint(struct proc *p, uint addr, int *ip)
3317 {
3318   if(addr >= p->sz || addr+4 > p->sz)
3319     return -1;
3320   *ip = *(int*)(addr);
3321   return 0;
3322 }
3323
3324 // Fetch the nul-terminated string at addr from process p.
3325 // Doesn't actually copy the string - just sets *pp to point at it.
3326 // Returns length of string, not including nul.
3327 int
3328 fetchstr(struct proc *p, uint addr, char **pp)
3329 {
3330   char *s, *ep;
3331
3332   if(addr >= p->sz)
3333     return -1;
3334   *pp = (char*)addr;
3335   ep = (char*)p->sz;
3336   for(s = *pp; s < ep; s++)
3337     if(*s == 0)
3338       return s - *pp;
3339   return -1;
3340 }
3341
3342 // Fetch the nth 32-bit system call argument.
3343 int
3344 argint(int n, int *ip)
3345 {
3346   return fetchint(proc, proc->tf->esp + 4 + 4*n, ip);
3347 }
3348
3349
```



```
3350 // Fetch the nth word-sized system call argument as a pointer
3351 // to a block of memory of size n bytes.  Check that the pointer
3352 // lies within the process address space.
3353 int
3354 argptr(int n, char **pp, int size)
3355 {
3356   int i;
3357
3358   if(argint(n, &i) < 0)
3359     return -1;
3360   if((uint)i >= proc->sz || (uint)i+size >= proc->sz)
3361     return -1;
3362   *pp = (char*)i;
3363   return 0;
3364 }
3365
3366 // Fetch the nth word-sized system call argument as a string pointer.
3367 // Check that the pointer is valid and the string is nul-terminated.
3368 // (There is no shared writable memory, so the string can't change
3369 // between this check and being used by the kernel.)
3370 int
3371 argstr(int n, char **pp)
3372 {
3373   int addr;
3374   if(argint(n, &addr) < 0)
3375     return -1;
3376   return fetchstr(proc, addr, pp);
3377 }
3378
3379 extern int sys_chdir(void);
3380 extern int sys_close(void);
3381 extern int sys_dup(void);
3382 extern int sys_exec(void);
3383 extern int sys_exit(void);
3384 extern int sys_fork(void);
3385 extern int sys_fstat(void);
3386 extern int sys_getpid(void);
3387 extern int sys_kill(void);
3388 extern int sys_link(void);
3389 extern int sys_mkdir(void);
3390 extern int sys_mknod(void);
3391 extern int sys_open(void);
3392 extern int sys_pipe(void);
3393 extern int sys_read(void);
3394 extern int sys_sbrk(void);
3395 extern int sys_sleep(void);
3396 extern int sys_unlink(void);
3397 extern int sys_wait(void);
3398 extern int sys_write(void);
3399 extern int sys_uptime(void);
```

```
3400 static int (*syscalls[])(void) = {
3401 [SYS_chdir]   sys_chdir,
3402 [SYS_close]   sys_close,
3403 [SYS_dup]     sys_dup,
3404 [SYS_exec]    sys_exec,
3405 [SYS_exit]    sys_exit,
3406 [SYS_fork]    sys_fork,
3407 [SYS_fstat]   sys_fstat,
3408 [SYS_getpid]  sys_getpid,
3409 [SYS_kill]    sys_kill,
3410 [SYS_link]    sys_link,
3411 [SYS_mkdir]   sys_mkdir,
3412 [SYS_mknod]   sys_mknod,
3413 [SYS_open]    sys_open,
3414 [SYS_pipe]    sys_pipe,
3415 [SYS_read]    sys_read,
3416 [SYS_sbrk]    sys_sbrk,
3417 [SYS_sleep]   sys_sleep,
3418 [SYS_unlink]  sys_unlink,
3419 [SYS_wait]    sys_wait,
3420 [SYS_write]   sys_write,
3421 [SYS_uptime]  sys_uptime,
3422 };
3423
3424 void
3425 syscall(void)
3426 {
3427   int num;
3428
3429   num = proc->tf->eax;
3430   if(num >= 0 && num < NELEM(syscalls) && syscalls[num])
3431     proc->tf->eax = syscalls[num]();
3432   else {
3433     cprintf("%d %s: unknown sys call %d\n",
3434             proc->pid, proc->name, num);
3435     proc->tf->eax = -1;
3436   }
3437 }
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
```



```
3450 #include "types.h"
3451 #include "x86.h"
3452 #include "defs.h"
3453 #include "param.h"
3454 #include "mmu.h"
3455 #include "proc.h"
3456
3457 int
3458 sys_fork(void)
3459 {
3460   return fork();
3461 }
3462
3463 int
3464 sys_exit(void)
3465 {
3466   exit();
3467   return 0;  // not reached
3468 }
3469
3470 int
3471 sys_wait(void)
3472 {
3473   return wait();
3474 }
3475
3476 int
3477 sys_kill(void)
3478 {
3479   int pid;
3480
3481   if(argint(0, &pid) < 0)
3482     return -1;
3483   return kill(pid);
3484 }
3485
3486 int
3487 sys_getpid(void)
3488 {
3489   return proc->pid;
3490 }
3491
3492
3493
3494
3495
3496
3497
3498
3499
```

```
3500 int
3501 sys_sbrk(void)
3502 {
3503   int addr;
3504   int n;
3505
3506   if(argint(0, &n) < 0)
3507     return -1;
3508   addr = proc->sz;
3509   if(growproc(n) < 0)
3510     return -1;
3511   return addr;
3512 }
3513
3514 int
3515 sys_sleep(void)
3516 {
3517   int n;
3518   uint ticks0;
3519
3520   if(argint(0, &n) < 0)
3521     return -1;
3522   acquire(&tickslock);
3523   ticks0 = ticks;
3524   while(ticks - ticks0 < n){
3525     if(proc->killed){
3526       release(&tickslock);
3527       return -1;
3528     }
3529     sleep(&ticks, &tickslock);
3530   }
3531   release(&tickslock);
3532   return 0;
3533 }
3534
3535 // return how many clock tick interrupts have occurred
3536 // since boot.
3537 int
3538 sys_uptime(void)
3539 {
3540   uint xticks;
3541
3542   acquire(&tickslock);
3543   xticks = ticks;
3544   release(&tickslock);
3545   return xticks;
3546 }
3547
3548
3549
```



```
3550 struct buf {
3551   int flags;
3552   uint dev;
3553   uint sector;
3554   struct buf *prev; // LRU cache list
3555   struct buf *next;
3556   struct buf *qnext; // disk queue
3557   uchar data[512];
3558 };
3559 #define B_BUSY  0x1  // buffer is locked by some process
3560 #define B_VALID 0x2  // buffer has been read from disk
3561 #define B_DIRTY 0x4  // buffer needs to be written to disk
3562
3563
3564
3565
3566
3567
3568
3569
3570
3571
3572
3573
3574
3575
3576
3577
3578
3579
3580
3581
3582
3583
3584
3585
3586
3587
3588
3589
3590
3591
3592
3593
3594
3595
3596
3597
3598
3599
```

```
3600 #define O_RDONLY  0x000
3601 #define O_WRONLY  0x001
3602 #define O_RDWR    0x002
3603 #define O_CREATE  0x200
3604
3605
3606
3607
3608
3609
3610
3611
3612
3613
3614
3615
3616
3617
3618
3619
3620
3621
3622
3623
3624
3625
3626
3627
3628
3629
3630
3631
3632
3633
3634
3635
3636
3637
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649
```



```
3650 #define T_DIR  1   // Directory
3651 #define T_FILE 2   // File
3652 #define T_DEV  3   // Special device
3653
3654 struct stat {
3655   short type;  // Type of file
3656   int dev;     // Device number
3657   uint ino;    // Inode number on device
3658   short nlink; // Number of links to file
3659   uint size;   // Size of file in bytes
3660 };
3661
3662
3663
3664
3665
3666
3667
3668
3669
3670
3671
3672
3673
3674
3675
3676
3677
3678
3679
3680
3681
3682
3683
3684
3685
3686
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699
```

```
3700 // On-disk file system format.
3701 // Both the kernel and user programs use this header file.
3702
3703 // Block 0 is unused.
3704 // Block 1 is super block.
3705 // Inodes start at block 2.
3706
3707 #define ROOTINO 1  // root i-number
3708 #define BSIZE 512  // block size
3709
3710 // File system super block
3711 struct superblock {
3712   uint size;         // Size of file system image (blocks)
3713   uint nblocks;      // Number of data blocks
3714   uint ninodes;      // Number of inodes.
3715 };
3716
3717 #define NDIRECT 12
3718 #define NINDIRECT (BSIZE / sizeof(uint))
3719 #define MAXFILE (NDIRECT + NINDIRECT)
3720
3721 // On-disk inode structure
3722 struct dinode {
3723   short type;           // File type
3724   short major;          // Major device number (T_DEV only)
3725   short minor;          // Minor device number (T_DEV only)
3726   short nlink;          // Number of links to inode in file system
3727   uint size;            // Size of file (bytes)
3728   uint addrs[NDIRECT+1];   // Data block addresses
3729 };
3730
3731 // Inodes per block.
3732 #define IPB           (BSIZE / sizeof(struct dinode))
3733
3734 // Block containing inode i
3735 #define IBLOCK(i)     ((i) / IPB + 2)
3736
3737 // Bitmap bits per block
3738 #define BPB           (BSIZE*8)
3739
3740 // Block containing bit for block b
3741 #define BBLOCK(b, ninodes) (b/BPB + (ninodes)/IPB + 3)
3742
3743 // Directory is a file containing a sequence of dirent structures.
3744 #define DIRSIZ 14
3745
3746 struct dirent {
3747   ushort inum;
3748   char name[DIRSIZ];
3749 };
```



```
3750 struct file {
3751   enum { FD_NONE, FD_PIPE, FD_INODE } type;
3752   int ref; // reference count
3753   char readable;
3754   char writable;
3755   struct pipe *pipe;
3756   struct inode *ip;
3757   uint off;
3758 };
3759
3760
3761 // in-core file system types
3762
3763 struct inode {
3764   uint dev;           // Device number
3765   uint inum;          // Inode number
3766   int ref;            // Reference count
3767   int flags;          // I_BUSY, I_VALID
3768
3769   short type;         // copy of disk inode
3770   short major;
3771   short minor;
3772   short nlink;
3773   uint size;
3774   uint addrs[NDIRECT+1];
3775 };
3776
3777 #define I_BUSY 0x1
3778 #define I_VALID 0x2
3779
3780
3781 // device implementations
3782
3783 struct devsw {
3784   int (*read)(struct inode*, char*, int);
3785   int (*write)(struct inode*, char*, int);
3786 };
3787
3788 extern struct devsw devsw[];
3789
3790 #define CONSOLE 1
3791
3792
3793
3794
3795
3796
3797
3798
3799
```

```
3800 // Simple PIO-based (non-DMA) IDE driver code.
3801
3802 #include "types.h"
3803 #include "defs.h"
3804 #include "param.h"
3805 #include "mmu.h"
3806 #include "proc.h"
3807 #include "x86.h"
3808 #include "traps.h"
3809 #include "spinlock.h"
3810 #include "buf.h"
3811
3812 #define IDE_BSY       0x80
3813 #define IDE_DRDY      0x40
3814 #define IDE_DF        0x20
3815 #define IDE_ERR       0x01
3816
3817 #define IDE_CMD_READ  0x20
3818 #define IDE_CMD_WRITE 0x30
3819
3820 // idequeue points to the buf now being read/written to the disk.
3821 // idequeue->qnext points to the next buf to be processed.
3822 // You must hold idelock while manipulating queue.
3823
3824 static struct spinlock idelock;
3825 static struct buf *idequeue;
3826
3827 static int havedisk1;
3828 static void idestart(struct buf*);
3829
3830 // Wait for IDE disk to become ready.
3831 static int
3832 idewait(int checkerr)
3833 {
3834   int r;
3835
3836   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
3837     ;
3838   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
3839     return -1;
3840   return 0;
3841 }
3842
3843
3844
3845
3846
3847
3848
3849
```



```
3850 void
3851 ideinit(void)
3852 {
3853   int i;
3854
3855   initlock(&idelock, "ide");
3856   picenable(IRQ_IDE);
3857   ioapicenable(IRQ_IDE, ncpu - 1);
3858   idewait(0);
3859
3860   // Check if disk 1 is present
3861   outb(0x1f6, 0xe0 | (1<<4));
3862   for(i=0; i<1000; i++){
3863     if(inb(0x1f7) != 0){
3864       havedisk1 = 1;
3865       break;
3866     }
3867   }
3868
3869   // Switch back to disk 0.
3870   outb(0x1f6, 0xe0 | (0<<4));
3871 }
3872
3873 // Start the request for b.  Caller must hold idelock.
3874 static void
3875 idestart(struct buf *b)
3876 {
3877   if(b == 0)
3878     panic("idestart");
3879
3880   idewait(0);
3881   outb(0x3f6, 0);  // generate interrupt
3882   outb(0x1f2, 1);  // number of sectors
3883   outb(0x1f3, b->sector & 0xff);
3884   outb(0x1f4, (b->sector >> 8) & 0xff);
3885   outb(0x1f5, (b->sector >> 16) & 0xff);
3886   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
3887   if(b->flags & B_DIRTY){
3888     outb(0x1f7, IDE_CMD_WRITE);
3889     outsl(0x1f0, b->data, 512/4);
3890   } else {
3891     outb(0x1f7, IDE_CMD_READ);
3892   }
3893 }
3894
3895
3896
3897
3898
3899
```

```
3900 // Interrupt handler.
3901 void
3902 ideintr(void)
3903 {
3904   struct buf *b;
3905
3906   // Take first buffer off queue.
3907   acquire(&idelock);
3908   if((b = idequeue) == 0){
3909     release(&idelock);
3910     // cprintf("spurious IDE interrupt\n");
3911     return;
3912   }
3913   idequeue = b->qnext;
3914
3915   // Read data if needed.
3916   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
3917     insl(0x1f0, b->data, 512/4);
3918
3919   // Wake process waiting for this buf.
3920   b->flags |= B_VALID;
3921   b->flags &= ~B_DIRTY;
3922   wakeup(b);
3923
3924   // Start disk on next buf in queue.
3925   if(idequeue != 0)
3926     idestart(idequeue);
3927
3928   release(&idelock);
3929 }
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
```



```
3950 // Sync buf with disk.
3951 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
3952 // Else if B_VALID is not set, read buf from disk, set B_VALID.
3953 void
3954 iderw(struct buf *b)
3955 {
3956   struct buf **pp;
3957
3958   if(!(b->flags & B_BUSY))
3959     panic("iderw: buf not busy");
3960   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
3961     panic("iderw: nothing to do");
3962   if(b->dev != 0 && !havedisk1)
3963     panic("iderw: ide disk 1 not present");
3964
3965   acquire(&idelock);
3966
3967   // Append b to idequeue.
3968   b->qnext = 0;
3969   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
3970     ;
3971   *pp = b;
3972
3973   // Start disk if necessary.
3974   if(idequeue == b)
3975     idestart(b);
3976
3977   // Wait for request to finish.
3978   // Assuming will not sleep too long: ignore proc->killed.
3979   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
3980     sleep(b, &idelock);
3981   }
3982
3983   release(&idelock);
3984 }
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
```

```
4000 // Buffer cache.
4001 //
4002 // The buffer cache is a linked list of buf structures holding
4003 // cached copies of disk block contents.  Caching disk blocks
4004 // in memory reduces the number of disk reads and also provides
4005 // a synchronization point for disk blocks used by multiple processes.
4006 //
4007 // Interface:
4008 // * To get a buffer for a particular disk block, call bread.
4009 // * After changing buffer data, call bwrite to flush it to disk.
4010 // * When done with the buffer, call brelse.
4011 // * Do not use the buffer after calling brelse.
4012 // * Only one process at a time can use a buffer,
4013 //     so do not keep them longer than necessary.
4014 //
4015 // The implementation uses three state flags internally:
4016 // * B_BUSY: the block has been returned from bread
4017 //     and has not been passed back to brelse.
4018 // * B_VALID: the buffer data has been initialized
4019 //     with the associated disk block contents.
4020 // * B_DIRTY: the buffer data has been modified
4021 //     and needs to be written to disk.
4022
4023 #include "types.h"
4024 #include "defs.h"
4025 #include "param.h"
4026 #include "spinlock.h"
4027 #include "buf.h"
4028
4029 struct {
4030   struct spinlock lock;
4031   struct buf buf[NBUF];
4032
4033   // Linked list of all buffers, through prev/next.
4034   // head.next is most recently used.
4035   struct buf head;
4036 } bcache;
4037
4038 void
4039 binit(void)
4040 {
4041   struct buf *b;
4042
4043   initlock(&bcache.lock, "bcache");
4044
4045
4046
4047
4048
4049
```



```
4050   // Create linked list of buffers
4051   bcache.head.prev = &bcache.head;
4052   bcache.head.next = &bcache.head;
4053   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4054     b->next = bcache.head.next;
4055     b->prev = &bcache.head;
4056     b->dev = -1;
4057     bcache.head.next->prev = b;
4058     bcache.head.next = b;
4059   }
4060 }
4061
4062 // Look through buffer cache for sector on device dev.
4063 // If not found, allocate fresh block.
4064 // In either case, return locked buffer.
4065 static struct buf*
4066 bget(uint dev, uint sector)
4067 {
4068   struct buf *b;
4069
4070   acquire(&bcache.lock);
4071
4072  loop:
4073   // Try for cached block.
4074   for(b = bcache.head.next; b != &bcache.head; b = b->next){
4075     if(b->dev == dev && b->sector == sector){
4076       if(!(b->flags & B_BUSY)){
4077         b->flags |= B_BUSY;
4078         release(&bcache.lock);
4079         return b;
4080       }
4081       sleep(b, &bcache.lock);
4082       goto loop;
4083     }
4084   }
4085
4086   // Allocate fresh block.
4087   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4088     if((b->flags & B_BUSY) == 0){
4089       b->dev = dev;
4090       b->sector = sector;
4091       b->flags = B_BUSY;
4092       release(&bcache.lock);
4093       return b;
4094     }
4095   }
4096   panic("bget: no buffers");
4097 }
4098
4099
```

```
4100 // Return a B_BUSY buf with the contents of the indicated disk sector.
4101 struct buf*
4102 bread(uint dev, uint sector)
4103 {
4104   struct buf *b;
4105
4106   b = bget(dev, sector);
4107   if(!(b->flags & B_VALID))
4108     iderw(b);
4109   return b;
4110 }
4111
4112 // Write b's contents to disk.  Must be locked.
4113 void
4114 bwrite(struct buf *b)
4115 {
4116   if((b->flags & B_BUSY) == 0)
4117     panic("bwrite");
4118   b->flags |= B_DIRTY;
4119   iderw(b);
4120 }
4121
4122 // Release the buffer b.
4123 void
4124 brelse(struct buf *b)
4125 {
4126   if((b->flags & B_BUSY) == 0)
4127     panic("brelse");
4128
4129   acquire(&bcache.lock);
4130
4131   b->next->prev = b->prev;
4132   b->prev->next = b->next;
4133   b->next = bcache.head.next;
4134   b->prev = &bcache.head;
4135   bcache.head.next->prev = b;
4136   bcache.head.next = b;
4137
4138   b->flags &= ~B_BUSY;
4139   wakeup(b);
4140
4141   release(&bcache.lock);
4142 }
4143
4144
4145
4146
4147
4148
4149
```



```
4150 // File system implementation.  Four layers:
4151 //   + Blocks: allocator for raw disk blocks.
4152 //   + Files: inode allocator, reading, writing, metadata.
4153 //   + Directories: inode with special contents (list of other inodes!)
4154 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
4155 //
4156 // Disk layout is: superblock, inodes, block in-use bitmap, data blocks.
4157 //
4158 // This file contains the low-level file system manipulation
4159 // routines.  The (higher-level) system call implementations
4160 // are in sysfile.c.
4161
4162 #include "types.h"
4163 #include "defs.h"
4164 #include "param.h"
4165 #include "stat.h"
4166 #include "mmu.h"
4167 #include "proc.h"
4168 #include "spinlock.h"
4169 #include "buf.h"
4170 #include "fs.h"
4171 #include "file.h"
4172
4173 #define min(a, b) ((a) < (b) ? (a) : (b))
4174 static void itrunc(struct inode*);
4175
4176 // Read the super block.
4177 static void
4178 readsb(int dev, struct superblock *sb)
4179 {
4180   struct buf *bp;
4181
4182   bp = bread(dev, 1);
4183   memmove(sb, bp->data, sizeof(*sb));
4184   brelse(bp);
4185 }
4186
4187 // Zero a block.
4188 static void
4189 bzero(int dev, int bno)
4190 {
4191   struct buf *bp;
4192
4193   bp = bread(dev, bno);
4194   memset(bp->data, 0, BSIZE);
4195   bwrite(bp);
4196   brelse(bp);
4197 }
4198
4199
```

```
4200 // Blocks.
4201
4202 // Allocate a disk block.
4203 static uint
4204 balloc(uint dev)
4205 {
4206   int b, bi, m;
4207   struct buf *bp;
4208   struct superblock sb;
4209
4210   bp = 0;
4211   readsb(dev, &sb);
4212   for(b = 0; b < sb.size; b += BPB){
4213     bp = bread(dev, BBLOCK(b, sb.ninodes));
4214     for(bi = 0; bi < BPB; bi++){
4215       m = 1 << (bi % 8);
4216       if((bp->data[bi/8] & m) == 0){  // Is block free?
4217         bp->data[bi/8] |= m;  // Mark block in use on disk.
4218         bwrite(bp);
4219         brelse(bp);
4220         return b + bi;
4221       }
4222     }
4223     brelse(bp);
4224   }
4225   panic("balloc: out of blocks");
4226 }
4227
4228 // Free a disk block.
4229 static void
4230 bfree(int dev, uint b)
4231 {
4232   struct buf *bp;
4233   struct superblock sb;
4234   int bi, m;
4235
4236   bzero(dev, b);
4237
4238   readsb(dev, &sb);
4239   bp = bread(dev, BBLOCK(b, sb.ninodes));
4240   bi = b % BPB;
4241   m = 1 << (bi % 8);
4242   if((bp->data[bi/8] & m) == 0)
4243     panic("freeing free block");
4244   bp->data[bi/8] &= ~m;  // Mark block free on disk.
4245   bwrite(bp);
4246   brelse(bp);
4247 }
4248
4249
```



```
4250 // Inodes.
4251 //
4252 // An inode is a single, unnamed file in the file system.
4253 // The inode disk structure holds metadata (the type, device numbers,
4254 // and data size) along with a list of blocks where the associated
4255 // data can be found.
4256 //
4257 // The inodes are laid out sequentially on disk immediately after
4258 // the superblock.  The kernel keeps a cache of the in-use
4259 // on-disk structures to provide a place for synchronizing access
4260 // to inodes shared between multiple processes.
4261 //
4262 // ip->ref counts the number of pointer references to this cached
4263 // inode; references are typically kept in struct file and in proc->cwd.
4264 // When ip->ref falls to zero, the inode is no longer cached.
4265 // It is an error to use an inode without holding a reference to it.
4266 //
4267 // Processes are only allowed to read and write inode
4268 // metadata and contents when holding the inode's lock,
4269 // represented by the I_BUSY flag in the in-memory copy.
4270 // Because inode locks are held during disk accesses,
4271 // they are implemented using a flag rather than with
4272 // spin locks.  Callers are responsible for locking
4273 // inodes before passing them to routines in this file; leaving
4274 // this responsibility with the caller makes it possible for them
4275 // to create arbitrarily-sized atomic operations.
4276 //
4277 // To give maximum control over locking to the callers,
4278 // the routines in this file that return inode pointers
4279 // return pointers to *unlocked* inodes.  It is the callers'
4280 // responsibility to lock them before using them.  A non-zero
4281 // ip->ref keeps these unlocked inodes in the cache.
4282
4283 struct {
4284   struct spinlock lock;
4285   struct inode inode[NINODE];
4286 } icache;
4287
4288 void
4289 iinit(void)
4290 {
4291   initlock(&icache.lock, "icache");
4292 }
4293
4294 static struct inode* iget(uint dev, uint inum);
4295
4296
4297
4298
4299
```

```
4300 // Allocate a new inode with the given type on device dev.
4301 struct inode*
4302 ialloc(uint dev, short type)
4303 {
4304   int inum;
4305   struct buf *bp;
4306   struct dinode *dip;
4307   struct superblock sb;
4308
4309   readsb(dev, &sb);
4310   for(inum = 1; inum < sb.ninodes; inum++){  // loop over inode blocks
4311     bp = bread(dev, IBLOCK(inum));
4312     dip = (struct dinode*)bp->data + inum%IPB;
4313     if(dip->type == 0){  // a free inode
4314       memset(dip, 0, sizeof(*dip));
4315       dip->type = type;
4316       bwrite(bp);   // mark it allocated on the disk
4317       brelse(bp);
4318       return iget(dev, inum);
4319     }
4320     brelse(bp);
4321   }
4322   panic("ialloc: no inodes");
4323 }
4324
4325 // Copy inode, which has changed, from memory to disk.
4326 void
4327 iupdate(struct inode *ip)
4328 {
4329   struct buf *bp;
4330   struct dinode *dip;
4331
4332   bp = bread(ip->dev, IBLOCK(ip->inum));
4333   dip = (struct dinode*)bp->data + ip->inum%IPB;
4334   dip->type = ip->type;
4335   dip->major = ip->major;
4336   dip->minor = ip->minor;
4337   dip->nlink = ip->nlink;
4338   dip->size = ip->size;
4339   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
4340   bwrite(bp);
4341   brelse(bp);
4342 }
4343
4344
4345
4346
4347
4348
4349
```



```
4350 // Find the inode with number inum on device dev
4351 // and return the in-memory copy.
4352 static struct inode*
4353 iget(uint dev, uint inum)
4354 {
4355   struct inode *ip, *empty;
4356
4357   acquire(&icache.lock);
4358
4359   // Try for cached inode.
4360   empty = 0;
4361   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
4362     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
4363       ip->ref++;
4364       release(&icache.lock);
4365       return ip;
4366     }
4367     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
4368       empty = ip;
4369   }
4370
4371   // Allocate fresh inode.
4372   if(empty == 0)
4373     panic("iget: no inodes");
4374
4375   ip = empty;
4376   ip->dev = dev;
4377   ip->inum = inum;
4378   ip->ref = 1;
4379   ip->flags = 0;
4380   release(&icache.lock);
4381
4382   return ip;
4383 }
4384
4385 // Increment reference count for ip.
4386 // Returns ip to enable ip = idup(ip1) idiom.
4387 struct inode*
4388 idup(struct inode *ip)
4389 {
4390   acquire(&icache.lock);
4391   ip->ref++;
4392   release(&icache.lock);
4393   return ip;
4394 }
4395
4396
4397
4398
4399
```

```
4400 // Lock the given inode.
4401 void
4402 ilock(struct inode *ip)
4403 {
4404   struct buf *bp;
4405   struct dinode *dip;
4406
4407   if(ip == 0 || ip->ref < 1)
4408     panic("ilock");
4409
4410   acquire(&icache.lock);
4411   while(ip->flags & I_BUSY)
4412     sleep(ip, &icache.lock);
4413   ip->flags |= I_BUSY;
4414   release(&icache.lock);
4415
4416   if(!(ip->flags & I_VALID)){
4417     bp = bread(ip->dev, IBLOCK(ip->inum));
4418     dip = (struct dinode*)bp->data + ip->inum%IPB;
4419     ip->type = dip->type;
4420     ip->major = dip->major;
4421     ip->minor = dip->minor;
4422     ip->nlink = dip->nlink;
4423     ip->size = dip->size;
4424     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
4425     brelse(bp);
4426     ip->flags |= I_VALID;
4427     if(ip->type == 0)
4428       panic("ilock: no type");
4429   }
4430 }
4431
4432 // Unlock the given inode.
4433 void
4434 iunlock(struct inode *ip)
4435 {
4436   if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
4437     panic("iunlock");
4438
4439   acquire(&icache.lock);
4440   ip->flags &= ~I_BUSY;
4441   wakeup(ip);
4442   release(&icache.lock);
4443 }
4444
4445
4446
4447
4448
4449
```



```
4450 // Caller holds reference to unlocked ip.  Drop reference.
4451 void
4452 iput(struct inode *ip)
4453 {
4454   acquire(&icache.lock);
4455   if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
4456     // inode is no longer used: truncate and free inode.
4457     if(ip->flags & I_BUSY)
4458       panic("iput busy");
4459     ip->flags |= I_BUSY;
4460     release(&icache.lock);
4461     itrunc(ip);
4462     ip->type = 0;
4463     iupdate(ip);
4464     acquire(&icache.lock);
4465     ip->flags = 0;
4466     wakeup(ip);
4467   }
4468   ip->ref--;
4469   release(&icache.lock);
4470 }
4471
4472 // Common idiom: unlock, then put.
4473 void
4474 iunlockput(struct inode *ip)
4475 {
4476   iunlock(ip);
4477   iput(ip);
4478 }
4479
4480
4481
4482
4483
4484
4485
4486
4487
4488
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499
```

```
4500 // Inode contents
4501 //
4502 // The contents (data) associated with each inode is stored
4503 // in a sequence of blocks on the disk.  The first NDIRECT blocks
4504 // are listed in ip->addrs[].  The next NINDIRECT blocks are
4505 // listed in the block ip->addrs[NDIRECT].
4506
4507 // Return the disk block address of the nth block in inode ip.
4508 // If there is no such block, bmap allocates one.
4509 static uint
4510 bmap(struct inode *ip, uint bn)
4511 {
4512   uint addr, *a;
4513   struct buf *bp;
4514
4515   if(bn < NDIRECT){
4516     if((addr = ip->addrs[bn]) == 0)
4517       ip->addrs[bn] = addr = balloc(ip->dev);
4518     return addr;
4519   }
4520   bn -= NDIRECT;
4521
4522   if(bn < NINDIRECT){
4523     // Load indirect block, allocating if necessary.
4524     if((addr = ip->addrs[NDIRECT]) == 0)
4525       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
4526     bp = bread(ip->dev, addr);
4527     a = (uint*)bp->data;
4528     if((addr = a[bn]) == 0){
4529       a[bn] = addr = balloc(ip->dev);
4530       bwrite(bp);
4531     }
4532     brelse(bp);
4533     return addr;
4534   }
4535
4536   panic("bmap: out of range");
4537 }
4538
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4549
```



```
4550 // Truncate inode (discard contents).
4551 // Only called after the last dirent referring
4552 // to this inode has been erased on disk.
4553 static void
4554 itrunc(struct inode *ip)
4555 {
4556   int i, j;
4557   struct buf *bp;
4558   uint *a;
4559
4560   for(i = 0; i < NDIRECT; i++){
4561     if(ip->addrs[i]){
4562       bfree(ip->dev, ip->addrs[i]);
4563       ip->addrs[i] = 0;
4564     }
4565   }
4566
4567   if(ip->addrs[NDIRECT]){
4568     bp = bread(ip->dev, ip->addrs[NDIRECT]);
4569     a = (uint*)bp->data;
4570     for(j = 0; j < NINDIRECT; j++){
4571       if(a[j])
4572         bfree(ip->dev, a[j]);
4573     }
4574     brelse(bp);
4575     bfree(ip->dev, ip->addrs[NDIRECT]);
4576     ip->addrs[NDIRECT] = 0;
4577   }
4578
4579   ip->size = 0;
4580   iupdate(ip);
4581 }
4582
4583 // Copy stat information from inode.
4584 void
4585 stati(struct inode *ip, struct stat *st)
4586 {
4587   st->dev = ip->dev;
4588   st->ino = ip->inum;
4589   st->type = ip->type;
4590   st->nlink = ip->nlink;
4591   st->size = ip->size;
4592 }
4593
4594
4595
4596
4597
4598
4599
```

```
4600 // Read data from inode.
4601 int
4602 readi(struct inode *ip, char *dst, uint off, uint n)
4603 {
4604   uint tot, m;
4605   struct buf *bp;
4606
4607   if(ip->type == T_DEV){
4608     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
4609       return -1;
4610     return devsw[ip->major].read(ip, dst, n);
4611   }
4612
4613   if(off > ip->size || off + n < off)
4614     return -1;
4615   if(off + n > ip->size)
4616     n = ip->size - off;
4617
4618   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
4619     bp = bread(ip->dev, bmap(ip, off/BSIZE));
4620     m = min(n - tot, BSIZE - off%BSIZE);
4621     memmove(dst, bp->data + off%BSIZE, m);
4622     brelse(bp);
4623   }
4624   return n;
4625 }
4626
4627
4628
4629
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649
```



```
4650 // Write data to inode.
4651 int
4652 writei(struct inode *ip, char *src, uint off, uint n)
4653 {
4654   uint tot, m;
4655   struct buf *bp;
4656
4657   if(ip->type == T_DEV){
4658     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
4659       return -1;
4660     return devsw[ip->major].write(ip, src, n);
4661   }
4662
4663   if(off > ip->size || off + n < off)
4664     return -1;
4665   if(off + n > MAXFILE*BSIZE)
4666     n = MAXFILE*BSIZE - off;
4667
4668   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
4669     bp = bread(ip->dev, bmap(ip, off/BSIZE));
4670     m = min(n - tot, BSIZE - off%BSIZE);
4671     memmove(bp->data + off%BSIZE, src, m);
4672     bwrite(bp);
4673     brelse(bp);
4674   }
4675
4676   if(n > 0 && off > ip->size){
4677     ip->size = off;
4678     iupdate(ip);
4679   }
4680   return n;
4681 }
4682
4683
4684
4685
4686
4687
4688
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4699
```

```
4700 // Directories
4701
4702 int
4703 namecmp(const char *s, const char *t)
4704 {
4705   return strncmp(s, t, DIRSIZ);
4706 }
4707
4708 // Look for a directory entry in a directory.
4709 // If found, set *poff to byte offset of entry.
4710 // Caller must have already locked dp.
4711 struct inode*
4712 dirlookup(struct inode *dp, char *name, uint *poff)
4713 {
4714   uint off, inum;
4715   struct buf *bp;
4716   struct dirent *de;
4717
4718   if(dp->type != T_DIR)
4719     panic("dirlookup not DIR");
4720
4721   for(off = 0; off < dp->size; off += BSIZE){
4722     bp = bread(dp->dev, bmap(dp, off / BSIZE));
4723     for(de = (struct dirent*)bp->data;
4724         de < (struct dirent*)(bp->data + BSIZE);
4725         de++){
4726       if(de->inum == 0)
4727         continue;
4728       if(namecmp(name, de->name) == 0){
4729         // entry matches path element
4730         if(poff)
4731           *poff = off + (uchar*)de - bp->data;
4732         inum = de->inum;
4733         brelse(bp);
4734         return iget(dp->dev, inum);
4735       }
4736     }
4737     brelse(bp);
4738   }
4739   return 0;
4740 }
4741
4742
4743
4744
4745
4746
4747
4748
4749
```



```
4750 // Write a new directory entry (name, inum) into the directory dp.
4751 int
4752 dirlink(struct inode *dp, char *name, uint inum)
4753 {
4754   int off;
4755   struct dirent de;
4756   struct inode *ip;
4757
4758   // Check that name is not present.
4759   if((ip = dirlookup(dp, name, 0)) != 0){
4760     iput(ip);
4761     return -1;
4762   }
4763
4764   // Look for an empty dirent.
4765   for(off = 0; off < dp->size; off += sizeof(de)){
4766     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4767       panic("dirlink read");
4768     if(de.inum == 0)
4769       break;
4770   }
4771
4772   strncpy(de.name, name, DIRSIZ);
4773   de.inum = inum;
4774   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4775     panic("dirlink");
4776
4777   return 0;
4778 }
4779
4780
4781
4782
4783
4784
4785
4786
4787
4788
4789
4790
4791
4792
4793
4794
4795
4796
4797
4798
4799
```

```
4800 // Paths
4801
4802 // Copy the next path element from path into name.
4803 // Return a pointer to the element following the copied one.
4804 // The returned path has no leading slashes,
4805 // so the caller can check *path=='\0' to see if the name is the last one.
4806 // If no name to remove, return 0.
4807 //
4808 // Examples:
4809 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
4810 //   skipelem("///a//bb", name) = "bb", setting name = "a"
4811 //   skipelem("a", name) = "", setting name = "a"
4812 //   skipelem("", name) = skipelem("////", name) = 0
4813 //
4814 static char*
4815 skipelem(char *path, char *name)
4816 {
4817   char *s;
4818   int len;
4819
4820   while(*path == '/')
4821     path++;
4822   if(*path == 0)
4823     return 0;
4824   s = path;
4825   while(*path != '/' && *path != 0)
4826     path++;
4827   len = path - s;
4828   if(len >= DIRSIZ)
4829     memmove(name, s, DIRSIZ);
4830   else {
4831     memmove(name, s, len);
4832     name[len] = 0;
4833   }
4834   while(*path == '/')
4835     path++;
4836   return path;
4837 }
4838
4839
4840
4841
4842
4843
4844
4845
4846
4847
4848
4849
```



```
4850 // Look up and return the inode for a path name.
4851 // If parent != 0, return the inode for the parent and copy the final
4852 // path element into name, which must have room for DIRSIZ bytes.
4853 static struct inode*
4854 namex(char *path, int nameiparent, char *name)
4855 {
4856   struct inode *ip, *next;
4857
4858   if(*path == '/')
4859     ip = iget(ROOTDEV, ROOTINO);
4860   else
4861     ip = idup(proc->cwd);
4862
4863   while((path = skipelem(path, name)) != 0){
4864     ilock(ip);
4865     if(ip->type != T_DIR){
4866       iunlockput(ip);
4867       return 0;
4868     }
4869     if(nameiparent && *path == '\0'){
4870       // Stop one level early.
4871       iunlock(ip);
4872       return ip;
4873     }
4874     if((next = dirlookup(ip, name, 0)) == 0){
4875       iunlockput(ip);
4876       return 0;
4877     }
4878     iunlockput(ip);
4879     ip = next;
4880   }
4881   if(nameiparent){
4882     iput(ip);
4883     return 0;
4884   }
4885   return ip;
4886 }
4887
4888 struct inode*
4889 namei(char *path)
4890 {
4891   char name[DIRSIZ];
4892   return namex(path, 0, name);
4893 }
4894
4895 struct inode*
4896 nameiparent(char *path, char *name)
4897 {
4898   return namex(path, 1, name);
4899 }
```

```
4900 #include "types.h"
4901 #include "defs.h"
4902 #include "param.h"
4903 #include "fs.h"
4904 #include "file.h"
4905 #include "spinlock.h"
4906
4907 struct devsw devsw[NDEV];
4908 struct {
4909   struct spinlock lock;
4910   struct file file[NFILE];
4911 } ftable;
4912
4913 void
4914 fileinit(void)
4915 {
4916   initlock(&ftable.lock, "ftable");
4917 }
4918
4919 // Allocate a file structure.
4920 struct file*
4921 filealloc(void)
4922 {
4923   struct file *f;
4924
4925   acquire(&ftable.lock);
4926   for(f = ftable.file; f < ftable.file + NFILE; f++){
4927     if(f->ref == 0){
4928       f->ref = 1;
4929       release(&ftable.lock);
4930       return f;
4931     }
4932   }
4933   release(&ftable.lock);
4934   return 0;
4935 }
4936
4937 // Increment ref count for file f.
4938 struct file*
4939 filedup(struct file *f)
4940 {
4941   acquire(&ftable.lock);
4942   if(f->ref < 1)
4943     panic("filedup");
4944   f->ref++;
4945   release(&ftable.lock);
4946   return f;
4947 }
4948
4949
```



```
4950 // Close file f.  (Decrement ref count, close when reaches 0.)
4951 void
4952 fileclose(struct file *f)
4953 {
4954   struct file ff;
4955
4956   acquire(&ftable.lock);
4957   if(f->ref < 1)
4958     panic("fileclose");
4959   if(--f->ref > 0){
4960     release(&ftable.lock);
4961     return;
4962   }
4963   ff = *f;
4964   f->ref = 0;
4965   f->type = FD_NONE;
4966   release(&ftable.lock);
4967
4968   if(ff.type == FD_PIPE)
4969     pipeclose(ff.pipe, ff.writable);
4970   else if(ff.type == FD_INODE)
4971     iput(ff.ip);
4972 }
4973
4974 // Get metadata about file f.
4975 int
4976 filestat(struct file *f, struct stat *st)
4977 {
4978   if(f->type == FD_INODE){
4979     ilock(f->ip);
4980     stati(f->ip, st);
4981     iunlock(f->ip);
4982     return 0;
4983   }
4984   return -1;
4985 }
4986
4987
4988
4989
4990
4991
4992
4993
4994
4995
4996
4997
4998
4999
```

```
5000 // Read from file f.  Addr is kernel address.
5001 int
5002 fileread(struct file *f, char *addr, int n)
5003 {
5004   int r;
5005
5006   if(f->readable == 0)
5007     return -1;
5008   if(f->type == FD_PIPE)
5009     return piperead(f->pipe, addr, n);
5010   if(f->type == FD_INODE){
5011     ilock(f->ip);
5012     if((r = readi(f->ip, addr, f->off, n)) > 0)
5013       f->off += r;
5014     iunlock(f->ip);
5015     return r;
5016   }
5017   panic("fileread");
5018 }
5019
5020 // Write to file f.  Addr is kernel address.
5021 int
5022 filewrite(struct file *f, char *addr, int n)
5023 {
5024   int r;
5025
5026   if(f->writable == 0)
5027     return -1;
5028   if(f->type == FD_PIPE)
5029     return pipewrite(f->pipe, addr, n);
5030   if(f->type == FD_INODE){
5031     ilock(f->ip);
5032     if((r = writei(f->ip, addr, f->off, n)) > 0)
5033       f->off += r;
5034     iunlock(f->ip);
5035     return r;
5036   }
5037   panic("filewrite");
5038 }
5039
5040
5041
5042
5043
5044
5045
5046
5047
5048
5049
```



```
5050 #include "types.h"
5051 #include "defs.h"
5052 #include "param.h"
5053 #include "stat.h"
5054 #include "mmu.h"
5055 #include "proc.h"
5056 #include "fs.h"
5057 #include "file.h"
5058 #include "fcntl.h"
5059
5060 // Fetch the nth word-sized system call argument as a file descriptor
5061 // and return both the descriptor and the corresponding struct file.
5062 static int
5063 argfd(int n, int *pfd, struct file **pf)
5064 {
5065   int fd;
5066   struct file *f;
5067
5068   if(argint(n, &fd) < 0)
5069     return -1;
5070   if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
5071     return -1;
5072   if(pfd)
5073     *pfd = fd;
5074   if(pf)
5075     *pf = f;
5076   return 0;
5077 }
5078
5079 // Allocate a file descriptor for the given file.
5080 // Takes over file reference from caller on success.
5081 static int
5082 fdalloc(struct file *f)
5083 {
5084   int fd;
5085
5086   for(fd = 0; fd < NOFILE; fd++){
5087     if(proc->ofile[fd] == 0){
5088       proc->ofile[fd] = f;
5089       return fd;
5090     }
5091   }
5092   return -1;
5093 }
5094
5095
5096
5097
5098
5099
```

```
5100 int
5101 sys_dup(void)
5102 {
5103   struct file *f;
5104   int fd;
5105
5106   if(argfd(0, 0, &f) < 0)
5107     return -1;
5108   if((fd=fdalloc(f)) < 0)
5109     return -1;
5110   filedup(f);
5111   return fd;
5112 }
5113
5114 int
5115 sys_read(void)
5116 {
5117   struct file *f;
5118   int n;
5119   char *p;
5120
5121   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5122     return -1;
5123   return fileread(f, p, n);
5124 }
5125
5126 int
5127 sys_write(void)
5128 {
5129   struct file *f;
5130   int n;
5131   char *p;
5132
5133   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5134     return -1;
5135   return filewrite(f, p, n);
5136 }
5137
5138 int
5139 sys_close(void)
5140 {
5141   int fd;
5142   struct file *f;
5143
5144   if(argfd(0, &fd, &f) < 0)
5145     return -1;
5146   proc->ofile[fd] = 0;
5147   fileclose(f);
5148   return 0;
5149 }
```



```
5150 int
5151 sys_fstat(void)
5152 {
5153   struct file *f;
5154   struct stat *st;
5155
5156   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
5157     return -1;
5158   return filestat(f, st);
5159 }
5160
5161 // Create the path new as a link to the same inode as old.
5162 int
5163 sys_link(void)
5164 {
5165   char name[DIRSIZ], *new, *old;
5166   struct inode *dp, *ip;
5167
5168   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
5169     return -1;
5170   if((ip = namei(old)) == 0)
5171     return -1;
5172   ilock(ip);
5173   if(ip->type == T_DIR){
5174     iunlockput(ip);
5175     return -1;
5176   }
5177   ip->nlink++;
5178   iupdate(ip);
5179   iunlock(ip);
5180
5181   if((dp = nameiparent(new, name)) == 0)
5182     goto bad;
5183   ilock(dp);
5184   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
5185     iunlockput(dp);
5186     goto bad;
5187   }
5188   iunlockput(dp);
5189   iput(ip);
5190   return 0;
5191
5192 bad:
5193   ilock(ip);
5194   ip->nlink--;
5195   iupdate(ip);
5196   iunlockput(ip);
5197   return -1;
5198 }
5199
```

```
5200 // Is the directory dp empty except for "." and ".." ?
5201 static int
5202 isdirempty(struct inode *dp)
5203 {
5204   int off;
5205   struct dirent de;
5206
5207   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
5208     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5209       panic("isdirempty: readi");
5210     if(de.inum != 0)
5211       return 0;
5212   }
5213   return 1;
5214 }
5215
5216
5217
5218
5219
5220
5221
5222
5223
5224
5225
5226
5227
5228
5229
5230
5231
5232
5233
5234
5235
5236
5237
5238
5239
5240
5241
5242
5243
5244
5245
5246
5247
5248
5249
```



```
5250 int
5251 sys_unlink(void)
5252 {
5253   struct inode *ip, *dp;
5254   struct dirent de;
5255   char name[DIRSIZ], *path;
5256   uint off;
5257
5258   if(argstr(0, &path) < 0)
5259     return -1;
5260   if((dp = nameiparent(path, name)) == 0)
5261     return -1;
5262   ilock(dp);
5263
5264   // Cannot unlink "." or "..".
5265   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0){
5266     iunlockput(dp);
5267     return -1;
5268   }
5269
5270   if((ip = dirlookup(dp, name, &off)) == 0){
5271     iunlockput(dp);
5272     return -1;
5273   }
5274   ilock(ip);
5275
5276   if(ip->nlink < 1)
5277     panic("unlink: nlink < 1");
5278   if(ip->type == T_DIR && !isdirempty(ip)){
5279     iunlockput(ip);
5280     iunlockput(dp);
5281     return -1;
5282   }
5283
5284   memset(&de, 0, sizeof(de));
5285   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5286     panic("unlink: writei");
5287   if(ip->type == T_DIR){
5288     dp->nlink--;
5289     iupdate(dp);
5290   }
5291   iunlockput(dp);
5292
5293   ip->nlink--;
5294   iupdate(ip);
5295   iunlockput(ip);
5296   return 0;
5297 }
5298
5299
```

```
5300 static struct inode*
5301 create(char *path, short type, short major, short minor)
5302 {
5303   uint off;
5304   struct inode *ip, *dp;
5305   char name[DIRSIZ];
5306
5307   if((dp = nameiparent(path, name)) == 0)
5308     return 0;
5309   ilock(dp);
5310
5311   if((ip = dirlookup(dp, name, &off)) != 0){
5312     iunlockput(dp);
5313     ilock(ip);
5314     if(type == T_FILE && ip->type == T_FILE)
5315       return ip;
5316     iunlockput(ip);
5317     return 0;
5318   }
5319
5320   if((ip = ialloc(dp->dev, type)) == 0)
5321     panic("create: ialloc");
5322
5323   ilock(ip);
5324   ip->major = major;
5325   ip->minor = minor;
5326   ip->nlink = 1;
5327   iupdate(ip);
5328
5329   if(type == T_DIR){  // Create . and .. entries.
5330     dp->nlink++;  // for ".."
5331     iupdate(dp);
5332     // No ip->nlink++ for ".": avoid cyclic ref count.
5333     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
5334       panic("create dots");
5335   }
5336
5337   if(dirlink(dp, name, ip->inum) < 0)
5338     panic("create: dirlink");
5339
5340   iunlockput(dp);
5341   return ip;
5342 }
5343
5344
5345
5346
5347
5348
5349
```



```
5350 int
5351 sys_open(void)
5352 {
5353   char *path;
5354   int fd, omode;
5355   struct file *f;
5356   struct inode *ip;
5357
5358   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
5359     return -1;
5360   if(omode & O_CREATE){
5361     if((ip = create(path, T_FILE, 0, 0)) == 0)
5362       return -1;
5363   } else {
5364     if((ip = namei(path)) == 0)
5365       return -1;
5366     ilock(ip);
5367     if(ip->type == T_DIR && omode != O_RDONLY){
5368       iunlockput(ip);
5369       return -1;
5370     }
5371   }
5372
5373   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
5374     if(f)
5375       fileclose(f);
5376     iunlockput(ip);
5377     return -1;
5378   }
5379   iunlock(ip);
5380
5381   f->type = FD_INODE;
5382   f->ip = ip;
5383   f->off = 0;
5384   f->readable = !(omode & O_WRONLY);
5385   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
5386   return fd;
5387 }
5388
5389 int
5390 sys_mkdir(void)
5391 {
5392   char *path;
5393   struct inode *ip;
5394
5395   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0)
5396     return -1;
5397   iunlockput(ip);
5398   return 0;
5399 }
```

```
5400 int
5401 sys_mknod(void)
5402 {
5403   struct inode *ip;
5404   char *path;
5405   int len;
5406   int major, minor;
5407
5408   if((len=argstr(0, &path)) < 0 ||
5409      argint(1, &major) < 0 ||
5410      argint(2, &minor) < 0 ||
5411      (ip = create(path, T_DEV, major, minor)) == 0)
5412     return -1;
5413   iunlockput(ip);
5414   return 0;
5415 }
5416
5417 int
5418 sys_chdir(void)
5419 {
5420   char *path;
5421   struct inode *ip;
5422
5423   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0)
5424     return -1;
5425   ilock(ip);
5426   if(ip->type != T_DIR){
5427     iunlockput(ip);
5428     return -1;
5429   }
5430   iunlock(ip);
5431   iput(proc->cwd);
5432   proc->cwd = ip;
5433   return 0;
5434 }
5435
5436
5437
5438
5439
5440
5441
5442
5443
5444
5445
5446
5447
5448
5449
```



```
5450 int
5451 sys_exec(void)
5452 {
5453   char *path, *argv[MAXARG];
5454   int i;
5455   uint uargv, uarg;
5456
5457   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
5458     return -1;
5459   }
5460   memset(argv, 0, sizeof(argv));
5461   for(i=0;; i++){
5462     if(i >= NELEM(argv))
5463       return -1;
5464     if(fetchint(proc, uargv+4*i, (int*)&uarg) < 0)
5465       return -1;
5466     if(uarg == 0){
5467       argv[i] = 0;
5468       break;
5469     }
5470     if(fetchstr(proc, uarg, &argv[i]) < 0)
5471       return -1;
5472   }
5473   return exec(path, argv);
5474 }
5475
5476 int
5477 sys_pipe(void)
5478 {
5479   int *fd;
5480   struct file *rf, *wf;
5481   int fd0, fd1;
5482
5483   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
5484     return -1;
5485   if(pipealloc(&rf, &wf) < 0)
5486     return -1;
5487   fd0 = -1;
5488   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
5489     if(fd0 >= 0)
5490       proc->ofile[fd0] = 0;
5491     fileclose(rf);
5492     fileclose(wf);
5493     return -1;
5494   }
5495   fd[0] = fd0;
5496   fd[1] = fd1;
5497   return 0;
5498 }
5499
```

```
5500 #include "types.h"
5501 #include "param.h"
5502 #include "mmu.h"
5503 #include "proc.h"
5504 #include "defs.h"
5505 #include "x86.h"
5506 #include "elf.h"
5507
5508 int
5509 exec(char *path, char **argv)
5510 {
5511   char *s, *last;
5512   int i, off;
5513   uint argc, sz, sp, ustack[3+MAXARG+1];
5514   struct elfhdr elf;
5515   struct inode *ip;
5516   struct proghdr ph;
5517   pde_t *pgdir, *oldpgdir;
5518
5519   if((ip = namei(path)) == 0)
5520     return -1;
5521   ilock(ip);
5522   pgdir = 0;
5523
5524   // Check ELF header
5525   if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
5526     goto bad;
5527   if(elf.magic != ELF_MAGIC)
5528     goto bad;
5529
5530   if((pgdir = setupkvm()) == 0)
5531     goto bad;
5532
5533   // Load program into memory.
5534   sz = 0;
5535   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
5536     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
5537       goto bad;
5538     if(ph.type != ELF_PROG_LOAD)
5539       continue;
5540     if(ph.memsz < ph.filesz)
5541       goto bad;
5542     if((sz = allocuvm(pgdir, sz, ph.va + ph.memsz)) == 0)
5543       goto bad;
5544     if(loaduvm(pgdir, (char*)ph.va, ip, ph.offset, ph.filesz) < 0)
5545       goto bad;
5546   }
5547   iunlockput(ip);
5548   ip = 0;
5549
```



```
5550   // Allocate a one-page stack at the next page boundary
5551   sz = PGROUNDUP(sz);
5552   if((sz = allocuvm(pgdir, sz, sz + PGSIZE)) == 0)
5553     goto bad;
5554
5555   // Push argument strings, prepare rest of stack in ustack.
5556   sp = sz;
5557   for(argc = 0; argv[argc]; argc++) {
5558     if(argc >= MAXARG)
5559       goto bad;
5560     sp -= strlen(argv[argc]) + 1;
5561     sp &= ~3;
5562     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
5563       goto bad;
5564     ustack[3+argc] = sp;
5565   }
5566   ustack[3+argc] = 0;
5567
5568   ustack[0] = 0xffffffff;  // fake return PC
5569   ustack[1] = argc;
5570   ustack[2] = sp - (argc+1)*4;  // argv pointer
5571
5572   sp -= (3+argc+1) * 4;
5573   if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
5574     goto bad;
5575
5576   // Save program name for debugging.
5577   for(last=s=path; *s; s++)
5578     if(*s == '/')
5579       last = s+1;
5580   safestrcpy(proc->name, last, sizeof(proc->name));
5581
5582   // Commit to the user image.
5583   oldpgdir = proc->pgdir;
5584   proc->pgdir = pgdir;
5585   proc->sz = sz;
5586   proc->tf->eip = elf.entry;  // main
5587   proc->tf->esp = sp;
5588   switchuvm(proc);
5589   freevm(oldpgdir);
5590
5591   return 0;
5592
5593  bad:
5594   if(pgdir)
5595     freevm(pgdir);
5596   if(ip)
5597     iunlockput(ip);
5598   return -1;
5599 }
```

```
5600 #include "types.h"
5601 #include "defs.h"
5602 #include "param.h"
5603 #include "mmu.h"
5604 #include "proc.h"
5605 #include "fs.h"
5606 #include "file.h"
5607 #include "spinlock.h"
5608
5609 #define PIPESIZE 512
5610
5611 struct pipe {
5612   struct spinlock lock;
5613   char data[PIPESIZE];
5614   uint nread;     // number of bytes read
5615   uint nwrite;    // number of bytes written
5616   int readopen;   // read fd is still open
5617   int writeopen;  // write fd is still open
5618 };
5619
5620 int
5621 pipealloc(struct file **f0, struct file **f1)
5622 {
5623   struct pipe *p;
5624
5625   p = 0;
5626   *f0 = *f1 = 0;
5627   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
5628     goto bad;
5629   if((p = (struct pipe*)kalloc()) == 0)
5630     goto bad;
5631   p->readopen = 1;
5632   p->writeopen = 1;
5633   p->nwrite = 0;
5634   p->nread = 0;
5635   initlock(&p->lock, "pipe");
5636   (*f0)->type = FD_PIPE;
5637   (*f0)->readable = 1;
5638   (*f0)->writable = 0;
5639   (*f0)->pipe = p;
5640   (*f1)->type = FD_PIPE;
5641   (*f1)->readable = 0;
5642   (*f1)->writable = 1;
5643   (*f1)->pipe = p;
5644   return 0;
5645
5646
5647
5648
5649
```



```
5650  bad:
5651   if(p)
5652     kfree((char*)p);
5653   if(*f0)
5654     fileclose(*f0);
5655   if(*f1)
5656     fileclose(*f1);
5657   return -1;
5658 }
5659
5660 void
5661 pipeclose(struct pipe *p, int writable)
5662 {
5663   acquire(&p->lock);
5664   if(writable){
5665     p->writeopen = 0;
5666     wakeup(&p->nread);
5667   } else {
5668     p->readopen = 0;
5669     wakeup(&p->nwrite);
5670   }
5671   if(p->readopen == 0 && p->writeopen == 0){
5672     release(&p->lock);
5673     kfree((char*)p);
5674   } else
5675     release(&p->lock);
5676 }
5677
5678
5679 int
5680 pipewrite(struct pipe *p, char *addr, int n)
5681 {
5682   int i;
5683
5684   acquire(&p->lock);
5685   for(i = 0; i < n; i++){
5686     while(p->nwrite == p->nread + PIPESIZE){
5687       if(p->readopen == 0 || proc->killed){
5688         release(&p->lock);
5689         return -1;
5690       }
5691       wakeup(&p->nread);
5692       sleep(&p->nwrite, &p->lock);
5693     }
5694     p->data[p->nwrite++ % PIPESIZE] = addr[i];
5695   }
5696   wakeup(&p->nread);
5697   release(&p->lock);
5698   return n;
5699 }
```

```
5700 int
5701 piperead(struct pipe *p, char *addr, int n)
5702 {
5703   int i;
5704
5705   acquire(&p->lock);
5706   while(p->nread == p->nwrite && p->writeopen){
5707     if(proc->killed){
5708       release(&p->lock);
5709       return -1;
5710     }
5711     sleep(&p->nread, &p->lock);
5712   }
5713   for(i = 0; i < n; i++){
5714     if(p->nread == p->nwrite)
5715       break;
5716     addr[i] = p->data[p->nread++ % PIPESIZE];
5717   }
5718   wakeup(&p->nwrite);
5719   release(&p->lock);
5720   return i;
5721 }
5722
5723
5724
5725
5726
5727
5728
5729
5730
5731
5732
5733
5734
5735
5736
5737
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749
```



```
5750 #include "types.h"
5751 #include "x86.h"
5752
5753 void*
5754 memset(void *dst, int c, uint n)
5755 {
5756   stosb(dst, c, n);
5757   return dst;
5758 }
5759
5760 int
5761 memcmp(const void *v1, const void *v2, uint n)
5762 {
5763   const uchar *s1, *s2;
5764
5765   s1 = v1;
5766   s2 = v2;
5767   while(n-- > 0){
5768     if(*s1 != *s2)
5769       return *s1 - *s2;
5770     s1++, s2++;
5771   }
5772
5773   return 0;
5774 }
5775
5776 void*
5777 memmove(void *dst, const void *src, uint n)
5778 {
5779   const char *s;
5780   char *d;
5781
5782   s = src;
5783   d = dst;
5784   if(s < d && s + n > d){
5785     s += n;
5786     d += n;
5787     while(n-- > 0)
5788       *--d = *--s;
5789   } else
5790     while(n-- > 0)
5791       *d++ = *s++;
5792
5793   return dst;
5794 }
5795
5796
5797
5798
5799
```

```
5800 // memcpy exists to placate GCC.  Use memmove.
5801 void*
5802 memcpy(void *dst, const void *src, uint n)
5803 {
5804   return memmove(dst, src, n);
5805 }
5806
5807 int
5808 strncmp(const char *p, const char *q, uint n)
5809 {
5810   while(n > 0 && *p && *p == *q)
5811     n--, p++, q++;
5812   if(n == 0)
5813     return 0;
5814   return (uchar)*p - (uchar)*q;
5815 }
5816
5817 char*
5818 strncpy(char *s, const char *t, int n)
5819 {
5820   char *os;
5821
5822   os = s;
5823   while(n-- > 0 && (*s++ = *t++) != 0)
5824     ;
5825   while(n-- > 0)
5826     *s++ = 0;
5827   return os;
5828 }
5829
5830 // Like strncpy but guaranteed to NUL-terminate.
5831 char*
5832 safestrcpy(char *s, const char *t, int n)
5833 {
5834   char *os;
5835
5836   os = s;
5837   if(n <= 0)
5838     return os;
5839   while(--n > 0 && (*s++ = *t++) != 0)
5840     ;
5841   *s = 0;
5842   return os;
5843 }
5844
5845
5846
5847
5848
5849
```



```
5850 int
5851 strlen(const char *s)
5852 {
5853   int n;
5854
5855   for(n = 0; s[n]; n++)
5856     ;
5857   return n;
5858 }
5859
5860
5861
5862
5863
5864
5865
5866
5867
5868
5869
5870
5871
5872
5873
5874
5875
5876
5877
5878
5879
5880
5881
5882
5883
5884
5885
5886
5887
5888
5889
5890
5891
5892
5893
5894
5895
5896
5897
5898
5899
```

```
5900 // See MultiProcessor Specification Version 1.[14]
5901
5902 struct mp {             // floating pointer
5903   uchar signature[4];           // "_MP_"
5904   void *physaddr;               // phys addr of MP config table
5905   uchar length;                 // 1
5906   uchar specrev;                // [14]
5907   uchar checksum;               // all bytes must add up to 0
5908   uchar type;                   // MP system config type
5909   uchar imcrp;
5910   uchar reserved[3];
5911 };
5912
5913 struct mpconf {         // configuration table header
5914   uchar signature[4];           // "PCMP"
5915   ushort length;                // total table length
5916   uchar version;                // [14]
5917   uchar checksum;               // all bytes must add up to 0
5918   uchar product[20];            // product id
5919   uint *oemtable;               // OEM table pointer
5920   ushort oemlength;             // OEM table length
5921   ushort entry;                 // entry count
5922   uint *lapicaddr;              // address of local APIC
5923   ushort xlength;               // extended table length
5924   uchar xchecksum;              // extended table checksum
5925   uchar reserved;
5926 };
5927
5928 struct mpproc {         // processor table entry
5929   uchar type;                   // entry type (0)
5930   uchar apicid;                 // local APIC id
5931   uchar version;                // local APIC verison
5932   uchar flags;                  // CPU flags
5933     #define MPBOOT 0x02           // This proc is the bootstrap processor.
5934   uchar signature[4];           // CPU signature
5935   uint feature;                 // feature flags from CPUID instruction
5936   uchar reserved[8];
5937 };
5938
5939 struct mpioapic {       // I/O APIC table entry
5940   uchar type;                   // entry type (2)
5941   uchar apicno;                 // I/O APIC id
5942   uchar version;                // I/O APIC version
5943   uchar flags;                  // I/O APIC flags
5944   uint *addr;                  // I/O APIC address
5945 };
5946
5947
5948
5949
```



```
5950 // Table entry types
5951 #define MPPROC    0x00  // One per processor
5952 #define MPBUS     0x01  // One per bus
5953 #define MPIOAPIC  0x02  // One per I/O APIC
5954 #define MPIOINTR  0x03  // One per bus interrupt source
5955 #define MPLINTR   0x04  // One per system interrupt source
5956
5957
5958
5959
5960
5961
5962
5963
5964
5965
5966
5967
5968
5969
5970
5971
5972
5973
5974
5975
5976
5977
5978
5979
5980
5981
5982
5983
5984
5985
5986
5987
5988
5989
5990
5991
5992
5993
5994
5995
5996
5997
5998
5999
```

```
6000 // Multiprocessor bootstrap.
6001 // Search memory for MP description structures.
6002 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
6003
6004 #include "types.h"
6005 #include "defs.h"
6006 #include "param.h"
6007 #include "mp.h"
6008 #include "x86.h"
6009 #include "mmu.h"
6010 #include "proc.h"
6011
6012 struct cpu cpus[NCPU];
6013 static struct cpu *bcpu;
6014 int ismp;
6015 int ncpu;
6016 uchar ioapicid;
6017
6018 int
6019 mpbcpu(void)
6020 {
6021   return bcpu-cpus;
6022 }
6023
6024 static uchar
6025 sum(uchar *addr, int len)
6026 {
6027   int i, sum;
6028
6029   sum = 0;
6030   for(i=0; i<len; i++)
6031     sum += addr[i];
6032   return sum;
6033 }
6034
6035 // Look for an MP structure in the len bytes at addr.
6036 static struct mp*
6037 mpsearch1(uchar *addr, int len)
6038 {
6039   uchar *e, *p;
6040
6041   e = addr+len;
6042   for(p = addr; p < e; p += sizeof(struct mp))
6043     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
6044       return (struct mp*)p;
6045   return 0;
6046 }
6047
6048
6049
```



```
6050 // Search for the MP Floating Pointer Structure, which according to the
6051 // spec is in one of the following three locations:
6052 // 1) in the first KB of the EBDA;
6053 // 2) in the last KB of system base memory;
6054 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
6055 static struct mp*
6056 mpsearch(void)
6057 {
6058   uchar *bda;
6059   uint p;
6060   struct mp *mp;
6061
6062   bda = (uchar*)0x400;
6063   if((p = ((bda[0x0F]<<8)|bda[0x0E]) << 4)){
6064     if((mp = mpsearch1((uchar*)p, 1024)))
6065       return mp;
6066   } else {
6067     p = ((bda[0x14]<<8)|bda[0x13])*1024;
6068     if((mp = mpsearch1((uchar*)p-1024, 1024)))
6069       return mp;
6070   }
6071   return mpsearch1((uchar*)0xF0000, 0x10000);
6072 }
6073
6074 // Search for an MP configuration table.  For now,
6075 // don't accept the default configurations (physaddr == 0).
6076 // Check for correct signature, calculate the checksum and,
6077 // if correct, check the version.
6078 // To do: check extended table checksum.
6079 static struct mpconf*
6080 mpconfig(struct mp **pmp)
6081 {
6082   struct mpconf *conf;
6083   struct mp *mp;
6084
6085   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
6086     return 0;
6087   conf = (struct mpconf*)mp->physaddr;
6088   if(memcmp(conf, "PCMP", 4) != 0)
6089     return 0;
6090   if(conf->version != 1 && conf->version != 4)
6091     return 0;
6092   if(sum((uchar*)conf, conf->length) != 0)
6093     return 0;
6094   *pmp = mp;
6095   return conf;
6096 }
6097
6098
6099
```

```
6100 void
6101 mpinit(void)
6102 {
6103   uchar *p, *e;
6104   struct mp *mp;
6105   struct mpconf *conf;
6106   struct mpproc *proc;
6107   struct mpioapic *ioapic;
6108
6109   bcpu = &cpus[0];
6110   if((conf = mpconfig(&mp)) == 0)
6111     return;
6112   ismp = 1;
6113   lapic = (uint*)conf->lapicaddr;
6114   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
6115     switch(*p){
6116     case MPPROC:
6117       proc = (struct mpproc*)p;
6118       if(ncpu != proc->apicid){
6119         cprintf("mpinit: ncpu=%d apicid=%d\n", ncpu, proc->apicid);
6120         ismp = 0;
6121       }
6122       if(proc->flags & MPBOOT)
6123         bcpu = &cpus[ncpu];
6124       cpus[ncpu].id = ncpu;
6125       ncpu++;
6126       p += sizeof(struct mpproc);
6127       continue;
6128     case MPIOAPIC:
6129       ioapic = (struct mpioapic*)p;
6130       ioapicid = ioapic->apicno;
6131       p += sizeof(struct mpioapic);
6132       continue;
6133     case MPBUS:
6134     case MPIOINTR:
6135     case MPLINTR:
6136       p += 8;
6137       continue;
6138     default:
6139       cprintf("mpinit: unknown config type %x\n", *p);
6140       ismp = 0;
6141     }
6142   }
6143   if(!ismp){
6144     // Didn't like what we found; fall back to no MP.
6145     ncpu = 1;
6146     lapic = 0;
6147     ioapicid = 0;
6148     return;
6149   }
```



```
6150   if(mp->imcrp){
6151     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
6152     // But it would on real hardware.
6153     outb(0x22, 0x70);   // Select IMCR
6154     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
6155   }
6156 }
6157
6158
6159
6160
6161
6162
6163
6164
6165
6166
6167
6168
6169
6170
6171
6172
6173
6174
6175
6176
6177
6178
6179
6180
6181
6182
6183
6184
6185
6186
6187
6188
6189
6190
6191
6192
6193
6194
6195
6196
6197
6198
6199
```

```
6200 // The local APIC manages internal (non-I/O) interrupts.
6201 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
6202
6203 #include "types.h"
6204 #include "defs.h"
6205 #include "traps.h"
6206 #include "mmu.h"
6207 #include "x86.h"
6208
6209 // Local APIC registers, divided by 4 for use as uint[] indices.
6210 #define ID      (0x0020/4)   // ID
6211 #define VER     (0x0030/4)   // Version
6212 #define TPR     (0x0080/4)   // Task Priority
6213 #define EOI     (0x00B0/4)   // EOI
6214 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
6215   #define ENABLE     0x00000100   // Unit Enable
6216 #define ESR     (0x0280/4)   // Error Status
6217 #define ICRLO   (0x0300/4)   // Interrupt Command
6218   #define INIT       0x00000500   // INIT/RESET
6219   #define STARTUP    0x00000600   // Startup IPI
6220   #define DELIVS     0x00001000   // Delivery status
6221   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
6222   #define DEASSERT   0x00000000
6223   #define LEVEL      0x00008000   // Level triggered
6224   #define BCAST      0x00080000   // Send to all APICs, including self.
6225   #define BUSY       0x00001000
6226   #define FIXED      0x00000000
6227 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
6228 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
6229   #define X1         0x0000000B   // divide counts by 1
6230   #define PERIODIC   0x00020000   // Periodic
6231 #define PCINT   (0x0340/4)   // Performance Counter LVT
6232 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
6233 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
6234 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
6235   #define MASKED     0x00010000   // Interrupt masked
6236 #define TICR    (0x0380/4)   // Timer Initial Count
6237 #define TCCR    (0x0390/4)   // Timer Current Count
6238 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
6239
6240 volatile uint *lapic;  // Initialized in mp.c
6241
6242 static void
6243 lapicw(int index, int value)
6244 {
6245   lapic[index] = value;
6246   lapic[ID];  // wait for write to finish, by reading
6247 }
6248
6249
```



```
6250 void
6251 lapicinit(int c)
6252 {
6253   cprintf("lapicinit: %d 0x%x\n", c, lapic);
6254   if(!lapic)
6255     return;
6256
6257   // Enable local APIC; set spurious interrupt vector.
6258   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
6259
6260   // The timer repeatedly counts down at bus frequency
6261   // from lapic[TICR] and then issues an interrupt.
6262   // If xv6 cared more about precise timekeeping,
6263   // TICR would be calibrated using an external time source.
6264   lapicw(TDCR, X1);
6265   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
6266   lapicw(TICR, 10000000);
6267
6268   // Disable logical interrupt lines.
6269   lapicw(LINT0, MASKED);
6270   lapicw(LINT1, MASKED);
6271
6272   // Disable performance counter overflow interrupts
6273   // on machines that provide that interrupt entry.
6274   if(((lapic[VER]>>16) & 0xFF) >= 4)
6275     lapicw(PCINT, MASKED);
6276
6277   // Map error interrupt to IRQ_ERROR.
6278   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
6279
6280   // Clear error status register (requires back-to-back writes).
6281   lapicw(ESR, 0);
6282   lapicw(ESR, 0);
6283
6284   // Ack any outstanding interrupts.
6285   lapicw(EOI, 0);
6286
6287   // Send an Init Level De-Assert to synchronise arbitration ID's.
6288   lapicw(ICRHI, 0);
6289   lapicw(ICRLO, BCAST | INIT | LEVEL);
6290   while(lapic[ICRLO] & DELIVS)
6291     ;
6292
6293   // Enable interrupts on the APIC (but not on the processor).
6294   lapicw(TPR, 0);
6295 }
6296
6297
6298
6299
```

```
6300 int
6301 cpunum(void)
6302 {
6303   // Cannot call cpu when interrupts are enabled:
6304   // result not guaranteed to last long enough to be used!
6305   // Would prefer to panic but even printing is chancy here:
6306   // almost everything, including cprintf and panic, calls cpu,
6307   // often indirectly through acquire and release.
6308   if(readeflags()&FL_IF){
6309     static int n;
6310     if(n++ == 0)
6311       cprintf("cpu called from %x with interrupts enabled\n",
6312         __builtin_return_address(0));
6313   }
6314
6315   if(lapic)
6316     return lapic[ID]>>24;
6317   return 0;
6318 }
6319
6320 // Acknowledge interrupt.
6321 void
6322 lapiceoi(void)
6323 {
6324   if(lapic)
6325     lapicw(EOI, 0);
6326 }
6327
6328 // Spin for a given number of microseconds.
6329 // On real hardware would want to tune this dynamically.
6330 void
6331 microdelay(int us)
6332 {
6333 }
6334
6335 #define IO_RTC  0x70
6336
6337 // Start additional processor running bootstrap code at addr.
6338 // See Appendix B of MultiProcessor Specification.
6339 void
6340 lapicstartap(uchar apicid, uint addr)
6341 {
6342   int i;
6343   ushort *wrv;
6344
6345   // "The BSP must initialize CMOS shutdown code to 0AH
6346   // and the warm reset vector (DWORD based at 40:67) to point at
6347   // the AP startup code prior to the [universal startup algorithm]."
6348   outb(IO_RTC, 0xF);  // offset 0xF is shutdown code
6349   outb(IO_RTC+1, 0x0A);
```



```
6350   wrv = (ushort*)(0x40<<4 | 0x67);  // Warm reset vector
6351   wrv[0] = 0;
6352   wrv[1] = addr >> 4;
6353
6354   // "Universal startup algorithm."
6355   // Send INIT (level-triggered) interrupt to reset other CPU.
6356   lapicw(ICRHI, apicid<<24);
6357   lapicw(ICRLO, INIT | LEVEL | ASSERT);
6358   microdelay(200);
6359   lapicw(ICRLO, INIT | LEVEL);
6360   microdelay(100);    // should be 10ms, but too slow in Bochs!
6361
6362   // Send startup IPI (twice!) to enter bootstrap code.
6363   // Regular hardware is supposed to only accept a STARTUP
6364   // when it is in the halted state due to an INIT.  So the second
6365   // should be ignored, but it is part of the official Intel algorithm.
6366   // Bochs complains about the second one.  Too bad for Bochs.
6367   for(i = 0; i < 2; i++){
6368     lapicw(ICRHI, apicid<<24);
6369     lapicw(ICRLO, STARTUP | (addr>>12));
6370     microdelay(200);
6371   }
6372 }
6373
6374
6375
6376
6377
6378
6379
6380
6381
6382
6383
6384
6385
6386
6387
6388
6389
6390
6391
6392
6393
6394
6395
6396
6397
6398
6399
```

```
6400 // The I/O APIC manages hardware interrupts for an SMP system.
6401 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
6402 // See also picirq.c.
6403
6404 #include "types.h"
6405 #include "defs.h"
6406 #include "traps.h"
6407
6408 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
6409
6410 #define REG_ID     0x00  // Register index: ID
6411 #define REG_VER    0x01  // Register index: version
6412 #define REG_TABLE  0x10  // Redirection table base
6413
6414 // The redirection table starts at REG_TABLE and uses
6415 // two registers to configure each interrupt.
6416 // The first (low) register in a pair contains configuration bits.
6417 // The second (high) register contains a bitmask telling which
6418 // CPUs can serve that interrupt.
6419 #define INT_DISABLED   0x00010000  // Interrupt disabled
6420 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
6421 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
6422 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
6423
6424 volatile struct ioapic *ioapic;
6425
6426 // IO APIC MMIO structure: write reg, then read or write data.
6427 struct ioapic {
6428   uint reg;
6429   uint pad[3];
6430   uint data;
6431 };
6432
6433 static uint
6434 ioapicread(int reg)
6435 {
6436   ioapic->reg = reg;
6437   return ioapic->data;
6438 }
6439
6440 static void
6441 ioapicwrite(int reg, uint data)
6442 {
6443   ioapic->reg = reg;
6444   ioapic->data = data;
6445 }
6446
6447
6448
6449
```



```
6450 void
6451 ioapicinit(void)
6452 {
6453   int i, id, maxintr;
6454
6455   if(!ismp)
6456     return;
6457
6458   ioapic = (volatile struct ioapic*)IOAPIC;
6459   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
6460   id = ioapicread(REG_ID) >> 24;
6461   if(id != ioapicid)
6462     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
6463
6464   // Mark all interrupts edge-triggered, active high, disabled,
6465   // and not routed to any CPUs.
6466   for(i = 0; i <= maxintr; i++){
6467     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
6468     ioapicwrite(REG_TABLE+2*i+1, 0);
6469   }
6470 }
6471
6472 void
6473 ioapicenable(int irq, int cpunum)
6474 {
6475   if(!ismp)
6476     return;
6477
6478   // Mark interrupt edge-triggered, active high,
6479   // enabled, and routed to the given cpunum,
6480   // which happens to be that cpu's APIC ID.
6481   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
6482   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
6483 }
6484
6485
6486
6487
6488
6489
6490
6491
6492
6493
6494
6495
6496
6497
6498
6499
```

```
6500 // Intel 8259A programmable interrupt controllers.
6501
6502 #include "types.h"
6503 #include "x86.h"
6504 #include "traps.h"
6505
6506 // I/O Addresses of the two programmable interrupt controllers
6507 #define IO_PIC1         0x20    // Master (IRQs 0-7)
6508 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
6509
6510 #define IRQ_SLAVE       2       // IRQ at which slave connects to master
6511
6512 // Current IRQ mask.
6513 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
6514 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
6515
6516 static void
6517 picsetmask(ushort mask)
6518 {
6519   irqmask = mask;
6520   outb(IO_PIC1+1, mask);
6521   outb(IO_PIC2+1, mask >> 8);
6522 }
6523
6524 void
6525 picenable(int irq)
6526 {
6527   picsetmask(irqmask & ~(1<<irq));
6528 }
6529
6530 // Initialize the 8259A interrupt controllers.
6531 void
6532 picinit(void)
6533 {
6534   // mask all interrupts
6535   outb(IO_PIC1+1, 0xFF);
6536   outb(IO_PIC2+1, 0xFF);
6537
6538   // Set up master (8259A-1)
6539
6540   // ICW1:  0001g0hi
6541   //    g:  0 = edge triggering, 1 = level triggering
6542   //    h:  0 = cascaded PICs, 1 = master only
6543   //    i:  0 = no ICW4, 1 = ICW4 required
6544   outb(IO_PIC1, 0x11);
6545
6546   // ICW2:  Vector offset
6547   outb(IO_PIC1+1, T_IRQ0);
6548
6549
```



```
6550   // ICW3:  (master PIC) bit mask of IR lines connected to slaves
6551   //        (slave PIC) 3-bit # of slave's connection to master
6552   outb(IO_PIC1+1, 1<<IRQ_SLAVE);
6553
6554   // ICW4:  000nbmap
6555   //    n:  1 = special fully nested mode
6556   //    b:  1 = buffered mode
6557   //    m:  0 = slave PIC, 1 = master PIC
6558   //      (ignored when b is 0, as the master/slave role
6559   //      can be hardwired).
6560   //    a:  1 = Automatic EOI mode
6561   //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
6562   outb(IO_PIC1+1, 0x3);
6563
6564   // Set up slave (8259A-2)
6565   outb(IO_PIC2, 0x11);                  // ICW1
6566   outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
6567   outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
6568   // NB Automatic EOI mode doesn't tend to work on the slave.
6569   // Linux source code says it's "to be investigated".
6570   outb(IO_PIC2+1, 0x3);                 // ICW4
6571
6572   // OCW3:  0ef01prs
6573   //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
6574   //    p:  0 = no polling, 1 = polling mode
6575   //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
6576   outb(IO_PIC1, 0x68);             // clear specific mask
6577   outb(IO_PIC1, 0x0a);             // read IRR by default
6578
6579   outb(IO_PIC2, 0x68);             // OCW3
6580   outb(IO_PIC2, 0x0a);             // OCW3
6581
6582   if(irqmask != 0xFFFF)
6583     picsetmask(irqmask);
6584 }
6585
6586
6587
6588
6589
6590
6591
6592
6593
6594
6595
6596
6597
6598
6599
```

```
6600 // PC keyboard interface constants
6601
6602 #define KBSTATP         0x64    // kbd controller status port(I)
6603 #define KBS_DIB         0x01    // kbd data in buffer
6604 #define KBDATAP         0x60    // kbd data port(I)
6605
6606 #define NO              0
6607
6608 #define SHIFT           (1<<0)
6609 #define CTL             (1<<1)
6610 #define ALT             (1<<2)
6611
6612 #define CAPSLOCK        (1<<3)
6613 #define NUMLOCK         (1<<4)
6614 #define SCROLLLOCK      (1<<5)
6615
6616 #define E0ESC           (1<<6)
6617
6618 // Special keycodes
6619 #define KEY_HOME        0xE0
6620 #define KEY_END         0xE1
6621 #define KEY_UP          0xE2
6622 #define KEY_DN          0xE3
6623 #define KEY_LF          0xE4
6624 #define KEY_RT          0xE5
6625 #define KEY_PGUP        0xE6
6626 #define KEY_PGDN        0xE7
6627 #define KEY_INS         0xE8
6628 #define KEY_DEL         0xE9
6629
6630 // C('A') == Control-A
6631 #define C(x) (x - '@')
6632
6633 static uchar shiftcode[256] =
6634 {
6635   [0x1D] CTL,
6636   [0x2A] SHIFT,
6637   [0x36] SHIFT,
6638   [0x38] ALT,
6639   [0x9D] CTL,
6640   [0xB8] ALT
6641 };
6642
6643 static uchar togglecode[256] =
6644 {
6645   [0x3A] CAPSLOCK,
6646   [0x45] NUMLOCK,
6647   [0x46] SCROLLLOCK
6648 };
6649
```



```
6650 static uchar normalmap[256] =
6651 {
6652   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
6653   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
6654   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
6655   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
6656   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
6657   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
6658   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
6659   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
6660   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
6661   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
6662   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
6663   [0x9C] '\n',      // KP_Enter
6664   [0xB5] '/',       // KP_Div
6665   [0xC8] KEY_UP,    [0xD0] KEY_DN,
6666   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
6667   [0xCB] KEY_LF,    [0xCD] KEY_RT,
6668   [0x97] KEY_HOME,  [0xCF] KEY_END,
6669   [0xD2] KEY_INS,   [0xD3] KEY_DEL
6670 };
6671
6672 static uchar shiftmap[256] =
6673 {
6674   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
6675   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
6676   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
6677   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
6678   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
6679   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
6680   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
6681   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
6682   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
6683   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
6684   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
6685   [0x9C] '\n',      // KP_Enter
6686   [0xB5] '/',       // KP_Div
6687   [0xC8] KEY_UP,    [0xD0] KEY_DN,
6688   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
6689   [0xCB] KEY_LF,    [0xCD] KEY_RT,
6690   [0x97] KEY_HOME,  [0xCF] KEY_END,
6691   [0xD2] KEY_INS,   [0xD3] KEY_DEL
6692 };
6693
6694
6695
6696
6697
6698
6699
```

```
6700 static uchar ctlmap[256] =
6701 {
6702   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
6703   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
6704   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
6705   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
6706   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
6707   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
6708   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
6709   [0x9C] '\r',      // KP_Enter
6710   [0xB5] C('/'),    // KP_Div
6711   [0xC8] KEY_UP,    [0xD0] KEY_DN,
6712   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
6713   [0xCB] KEY_LF,    [0xCD] KEY_RT,
6714   [0x97] KEY_HOME,  [0xCF] KEY_END,
6715   [0xD2] KEY_INS,   [0xD3] KEY_DEL
6716 };
6717
6718
6719
6720
6721
6722
6723
6724
6725
6726
6727
6728
6729
6730
6731
6732
6733
6734
6735
6736
6737
6738
6739
6740
6741
6742
6743
6744
6745
6746
6747
6748
6749
```



```
6750 #include "types.h"
6751 #include "x86.h"
6752 #include "defs.h"
6753 #include "kbd.h"
6754
6755 int
6756 kbdgetc(void)
6757 {
6758   static uint shift;
6759   static uchar *charcode[4] = {
6760     normalmap, shiftmap, ctlmap, ctlmap
6761   };
6762   uint st, data, c;
6763
6764   st = inb(KBSTATP);
6765   if((st & KBS_DIB) == 0)
6766     return -1;
6767   data = inb(KBDATAP);
6768
6769   if(data == 0xE0){
6770     shift |= E0ESC;
6771     return 0;
6772   } else if(data & 0x80){
6773     // Key released
6774     data = (shift & E0ESC ? data : data & 0x7F);
6775     shift &= ~(shiftcode[data] | E0ESC);
6776     return 0;
6777   } else if(shift & E0ESC){
6778     // Last character was an E0 escape; or with 0x80
6779     data |= 0x80;
6780     shift &= ~E0ESC;
6781   }
6782
6783   shift |= shiftcode[data];
6784   shift ^= togglecode[data];
6785   c = charcode[shift & (CTL | SHIFT)][data];
6786   if(shift & CAPSLOCK){
6787     if('a' <= c && c <= 'z')
6788       c += 'A' - 'a';
6789     else if('A' <= c && c <= 'Z')
6790       c += 'a' - 'A';
6791   }
6792   return c;
6793 }
6794
6795 void
6796 kbdintr(void)
6797 {
6798   consoleintr(kbdgetc);
6799 }
```

```
6800 // Console input and output.
6801 // Input is from the keyboard or serial port.
6802 // Output is written to the screen and serial port.
6803
6804 #include "types.h"
6805 #include "defs.h"
6806 #include "param.h"
6807 #include "traps.h"
6808 #include "spinlock.h"
6809 #include "fs.h"
6810 #include "file.h"
6811 #include "mmu.h"
6812 #include "proc.h"
6813 #include "x86.h"
6814
6815 static void consputc(int);
6816
6817 static int panicked = 0;
6818
6819 static struct {
6820   struct spinlock lock;
6821   int locking;
6822 } cons;
6823
6824 static void
6825 printint(int xx, int base, int sgn)
6826 {
6827   static char digits[] = "0123456789abcdef";
6828   char buf[16];
6829   int i, neg;
6830   uint x;
6831
6832   if(sgn && (neg = xx < 0)){
6833     neg = 1;
6834     x = -xx;
6835   } else
6836     x = xx;
6837
6838   i = 0;
6839   do{
6840     buf[i++] = digits[x % base];
6841   }while((x /= base) != 0);
6842   if(neg)
6843     buf[i++] = '-';
6844
6845   while(--i >= 0)
6846     consputc(buf[i]);
6847 }
6848
6849
```



```
6850 // Print to the console. only understands %d, %x, %p, %s.
6851 void
6852 cprintf(char *fmt, ...)
6853 {
6854   int i, c, state, locking;
6855   uint *argp;
6856   char *s;
6857
6858   locking = cons.locking;
6859   if(locking)
6860     acquire(&cons.lock);
6861
6862   argp = (uint*)(void*)(&fmt + 1);
6863   state = 0;
6864   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
6865     if(c != '%'){
6866       consputc(c);
6867       continue;
6868     }
6869     c = fmt[++i] & 0xff;
6870     if(c == 0)
6871       break;
6872     switch(c){
6873     case 'd':
6874       printint(*argp++, 10, 1);
6875       break;
6876     case 'x':
6877     case 'p':
6878       printint(*argp++, 16, 0);
6879       break;
6880     case 's':
6881       if((s = (char*)*argp++) == 0)
6882         s = "(null)";
6883       for(; *s; s++)
6884         consputc(*s);
6885       break;
6886     case '%':
6887       consputc('%');
6888       break;
6889     default:
6890       // Print unknown % sequence to draw attention.
6891       consputc('%');
6892       consputc(c);
6893       break;
6894     }
6895   }
6896
6897   if(locking)
6898     release(&cons.lock);
6899 }
```

```
6900 void
6901 panic(char *s)
6902 {
6903   int i;
6904   uint pcs[10];
6905
6906   cli();
6907   cons.locking = 0;
6908   cprintf("cpu%d: panic: ", cpu->id);
6909   cprintf(s);
6910   cprintf("\n");
6911   getcallerpcs(&s, pcs);
6912   for(i=0; i<10; i++)
6913     cprintf(" %p", pcs[i]);
6914   panicked = 1; // freeze other CPU
6915   for(;;)
6916     ;
6917 }
6918
6919
6920
6921
6922
6923
6924
6925
6926
6927
6928
6929
6930
6931
6932
6933
6934
6935
6936
6937
6938
6939
6940
6941
6942
6943
6944
6945
6946
6947
6948
6949
```



```
6950 #define BACKSPACE 0x100
6951 #define CRTPORT 0x3d4
6952 static ushort *crt = (ushort*)0xb8000;  // CGA memory
6953
6954 static void
6955 cgaputc(int c)
6956 {
6957   int pos;
6958
6959   // Cursor position: col + 80*row.
6960   outb(CRTPORT, 14);
6961   pos = inb(CRTPORT+1) << 8;
6962   outb(CRTPORT, 15);
6963   pos |= inb(CRTPORT+1);
6964
6965   if(c == '\n')
6966     pos += 80 - pos%80;
6967   else if(c == BACKSPACE){
6968     if(pos > 0) --pos;
6969   } else
6970     crt[pos++] = (c&0xff) | 0x0700;  // black on white
6971
6972   if((pos/80) >= 24){  // Scroll up.
6973     memmove(crt, crt+80, sizeof(crt[0])*23*80);
6974     pos -= 80;
6975     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
6976   }
6977
6978   outb(CRTPORT, 14);
6979   outb(CRTPORT+1, pos>>8);
6980   outb(CRTPORT, 15);
6981   outb(CRTPORT+1, pos);
6982   crt[pos] = ' ' | 0x0700;
6983 }
6984
6985 void
6986 consputc(int c)
6987 {
6988   if(panicked){
6989     cli();
6990     for(;;)
6991       ;
6992   }
6993
6994   if(c == BACKSPACE){
6995     uartputc('\b'); uartputc(' '); uartputc('\b');
6996   } else
6997     uartputc(c);
6998   cgaputc(c);
6999 }
```

```
7000 #define INPUT_BUF 128
7001 struct {
7002   struct spinlock lock;
7003   char buf[INPUT_BUF];
7004   uint r;  // Read index
7005   uint w;  // Write index
7006   uint e;  // Edit index
7007 } input;
7008
7009 #define C(x)  ((x)-'@')  // Control-x
7010
7011 void
7012 consoleintr(int (*getc)(void))
7013 {
7014   int c;
7015
7016   acquire(&input.lock);
7017   while((c = getc()) >= 0){
7018     switch(c){
7019     case C('P'):  // Process listing.
7020       procdump();
7021       break;
7022     case C('U'):  // Kill line.
7023       while(input.e != input.w &&
7024             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
7025         input.e--;
7026         consputc(BACKSPACE);
7027       }
7028       break;
7029     case C('H'): case '\x7f':  // Backspace
7030       if(input.e != input.w){
7031         input.e--;
7032         consputc(BACKSPACE);
7033       }
7034       break;
7035     default:
7036       if(c != 0 && input.e-input.r < INPUT_BUF){
7037         c = (c == '\r') ? '\n' : c;
7038         input.buf[input.e++ % INPUT_BUF] = c;
7039         consputc(c);
7040         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
7041           input.w = input.e;
7042           wakeup(&input.r);
7043         }
7044       }
7045       break;
7046     }
7047   }
7048   release(&input.lock);
7049 }
```



```
7050 int
7051 consoleread(struct inode *ip, char *dst, int n)
7052 {
7053   uint target;
7054   int c;
7055
7056   iunlock(ip);
7057   target = n;
7058   acquire(&input.lock);
7059   while(n > 0){
7060     while(input.r == input.w){
7061       if(proc->killed){
7062         release(&input.lock);
7063         ilock(ip);
7064         return -1;
7065       }
7066       sleep(&input.r, &input.lock);
7067     }
7068     c = input.buf[input.r++ % INPUT_BUF];
7069     if(c == C('D')){  // EOF
7070       if(n < target){
7071         // Save ^D for next time, to make sure
7072         // caller gets a 0-byte result.
7073         input.r--;
7074       }
7075       break;
7076     }
7077     *dst++ = c;
7078     --n;
7079     if(c == '\n')
7080       break;
7081   }
7082   release(&input.lock);
7083   ilock(ip);
7084
7085   return target - n;
7086 }
7087
7088
7089
7090
7091
7092
7093
7094
7095
7096
7097
7098
7099
```

```
7100 int
7101 consolewrite(struct inode *ip, char *buf, int n)
7102 {
7103   int i;
7104
7105   iunlock(ip);
7106   acquire(&cons.lock);
7107   for(i = 0; i < n; i++)
7108     consputc(buf[i] & 0xff);
7109   release(&cons.lock);
7110   ilock(ip);
7111
7112   return n;
7113 }
7114
7115 void
7116 consoleinit(void)
7117 {
7118   initlock(&cons.lock, "console");
7119   initlock(&input.lock, "input");
7120
7121   devsw[CONSOLE].write = consolewrite;
7122   devsw[CONSOLE].read = consoleread;
7123   cons.locking = 1;
7124
7125   picenable(IRQ_KBD);
7126   ioapicenable(IRQ_KBD, 0);
7127 }
7128
7129
7130
7131
7132
7133
7134
7135
7136
7137
7138
7139
7140
7141
7142
7143
7144
7145
7146
7147
7148
7149
```



```
7150 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
7151 // Only used on uniprocessors;
7152 // SMP machines use the local APIC timer.
7153
7154 #include "types.h"
7155 #include "defs.h"
7156 #include "traps.h"
7157 #include "x86.h"
7158
7159 #define IO_TIMER1       0x040           // 8253 Timer #1
7160
7161 // Frequency of all three count-down timers;
7162 // (TIMER_FREQ/freq) is the appropriate count
7163 // to generate a frequency of freq Hz.
7164
7165 #define TIMER_FREQ      1193182
7166 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
7167
7168 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
7169 #define TIMER_SEL0      0x00    // select counter 0
7170 #define TIMER_RATEGEN   0x04    // mode 2, rate generator
7171 #define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first
7172
7173 void
7174 timerinit(void)
7175 {
7176   // Interrupt 100 times/sec.
7177   outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
7178   outb(IO_TIMER1, TIMER_DIV(100) % 256);
7179   outb(IO_TIMER1, TIMER_DIV(100) / 256);
7180   picenable(IRQ_TIMER);
7181 }
7182
7183
7184
7185
7186
7187
7188
7189
7190
7191
7192
7193
7194
7195
7196
7197
7198
7199
```

```
7200 // Intel 8250 serial port (UART).
7201
7202 #include "types.h"
7203 #include "defs.h"
7204 #include "param.h"
7205 #include "traps.h"
7206 #include "spinlock.h"
7207 #include "fs.h"
7208 #include "file.h"
7209 #include "mmu.h"
7210 #include "proc.h"
7211 #include "x86.h"
7212
7213 #define COM1    0x3f8
7214
7215 static int uart;    // is there a uart?
7216
7217 void
7218 uartinit(void)
7219 {
7220   char *p;
7221
7222   // Turn off the FIFO
7223   outb(COM1+2, 0);
7224
7225   // 9600 baud, 8 data bits, 1 stop bit, parity off.
7226   outb(COM1+3, 0x80);    // Unlock divisor
7227   outb(COM1+0, 115200/9600);
7228   outb(COM1+1, 0);
7229   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
7230   outb(COM1+4, 0);
7231   outb(COM1+1, 0x01);    // Enable receive interrupts.
7232
7233   // If status is 0xFF, no serial port.
7234   if(inb(COM1+5) == 0xFF)
7235     return;
7236   uart = 1;
7237
7238   // Acknowledge pre-existing interrupt conditions;
7239   // enable interrupts.
7240   inb(COM1+2);
7241   inb(COM1+0);
7242   picenable(IRQ_COM1);
7243   ioapicenable(IRQ_COM1, 0);
7244
7245   // Announce that we're here.
7246   for(p="xv6...\n"; *p; p++)
7247     uartputc(*p);
7248 }
7249
```



```
7250 void
7251 uartputc(int c)
7252 {
7253   int i;
7254
7255   if(!uart)
7256     return;
7257   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
7258     microdelay(10);
7259   outb(COM1+0, c);
7260 }
7261
7262 static int
7263 uartgetc(void)
7264 {
7265   if(!uart)
7266     return -1;
7267   if(!(inb(COM1+5) & 0x01))
7268     return -1;
7269   return inb(COM1+0);
7270 }
7271
7272 void
7273 uartintr(void)
7274 {
7275   consoleintr(uartgetc);
7276 }
7277
7278
7279
7280
7281
7282
7283
7284
7285
7286
7287
7288
7289
7290
7291
7292
7293
7294
7295
7296
7297
7298
7299
```

```
7300 # Multiboot header, for multiboot boot loaders like GNU Grub.
7301 # http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
7302 #
7303 # Using GRUB 2, you can boot xv6 from a file stored in a
7304 # Linux file system by copying kernel or kernelmemfs to /boot
7305 # and then adding this menu entry:
7306 #
7307 # menuentry "xv6" {
7308 #  insmod ext2
7309 #  set root='(hd0,msdos1)'
7310 #  set kernel='/boot/kernel'
7311 #  echo "Loading ${kernel}..."
7312 #  multiboot ${kernel} ${kernel}
7313 #  boot
7314 # }
7315
7316 #include "asm.h"
7317
7318 #define STACK 4096
7319
7320 #define SEG_KCODE 1  // kernel code
7321 #define SEG_KDATA 2  // kernel data+stack
7322
7323 # Multiboot header.  Data to direct multiboot loader.
7324 .p2align 2
7325 .text
7326 .globl multiboot_header
7327 multiboot_header:
7328   #define magic 0x1badb002
7329   #define flags (1<<16 | 1<<0)
7330   .long magic
7331   .long flags
7332   .long (-magic-flags)
7333   .long multiboot_header  # beginning of image
7334   .long multiboot_header
7335   .long edata
7336   .long end
7337   .long multiboot_entry
7338
7339 # Multiboot entry point.  Machine is mostly set up.
7340 # Configure the GDT to match the environment that our usual
7341 # boot loader - bootasm.S - sets up.
7342 .globl multiboot_entry
7343 multiboot_entry:
7344   lgdt gdtdesc
7345   ljmp $(SEG_KCODE<<3), $mbstart32
7346
7347
7348
7349
```



```
7350 mbstart32:
7351   # Set up the protected-mode data segment registers
7352   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
7353   movw    %ax, %ds                # -> DS: Data Segment
7354   movw    %ax, %es                # -> ES: Extra Segment
7355   movw    %ax, %ss                # -> SS: Stack Segment
7356   movw    $0, %ax                 # Zero segments not ready for use
7357   movw    %ax, %fs                # -> FS
7358   movw    %ax, %gs                # -> GS
7359
7360   # Set up the stack pointer and call into C.
7361   movl $(stack + STACK), %esp
7362   call main
7363 spin:
7364   jmp spin
7365
7366 # Bootstrap GDT
7367 .p2align 2                                # force 4 byte alignment
7368 gdt:
7369   SEG_NULLASM                             # null seg
7370   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
7371   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
7372
7373 gdtdesc:
7374   .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
7375   .long   gdt                             # address gdt
7376
7377 .comm stack, STACK
7378
7379
7380
7381
7382
7383
7384
7385
7386
7387
7388
7389
7390
7391
7392
7393
7394
7395
7396
7397
7398
7399
```

```
# Initial process execs /init.

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

# exec(init, argv)
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL

# for(;;) exit();
exit:
  movl $SYS_exit, %eax
  int $T_SYSCALL
  jmp exit

# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .long init
  .long 0
```



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

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

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

```
7500 // init: The initial user-level program
7501
7502 #include "types.h"
7503 #include "stat.h"
7504 #include "user.h"
7505 #include "fcntl.h"
7506
7507 char *argv[] = { "sh", 0 };
7508
7509 int
7510 main(void)
7511 {
7512   int pid, wpid;
7513
7514   if(open("console", O_RDWR) < 0){
7515     mknod("console", 1, 1);
7516     open("console", O_RDWR);
7517   }
7518   dup(0);  // stdout
7519   dup(0);  // stderr
7520
7521   for(;;){
7522     printf(1, "init: starting sh\n");
7523     pid = fork();
7524     if(pid < 0){
7525       printf(1, "init: fork failed\n");
7526       exit();
7527     }
7528     if(pid == 0){
7529       exec("sh", argv);
7530       printf(1, "init: exec sh failed\n");
7531       exit();
7532     }
7533     while((wpid=wait()) >= 0 && wpid != pid)
7534       printf(1, "zombie!\n");
7535   }
7536 }
7537
7538
7539
7540
7541
7542
7543
7544
7545
7546
7547
7548
7549
```



```
7550 // Shell.
7551
7552 #include "types.h"
7553 #include "user.h"
7554 #include "fcntl.h"
7555
7556 // Parsed command representation
7557 #define EXEC  1
7558 #define REDIR 2
7559 #define PIPE  3
7560 #define LIST  4
7561 #define BACK  5
7562
7563 #define MAXARGS 10
7564
7565 struct cmd {
7566   int type;
7567 };
7568
7569 struct execcmd {
7570   int type;
7571   char *argv[MAXARGS];
7572   char *eargv[MAXARGS];
7573 };
7574
7575 struct redircmd {
7576   int type;
7577   struct cmd *cmd;
7578   char *file;
7579   char *efile;
7580   int mode;
7581   int fd;
7582 };
7583
7584 struct pipecmd {
7585   int type;
7586   struct cmd *left;
7587   struct cmd *right;
7588 };
7589
7590 struct listcmd {
7591   int type;
7592   struct cmd *left;
7593   struct cmd *right;
7594 };
7595
7596 struct backcmd {
7597   int type;
7598   struct cmd *cmd;
7599 };
```

```
7600 int fork1(void);  // Fork but panics on failure.
7601 void panic(char*);
7602 struct cmd *parsecmd(char*);
7603
7604 // Execute cmd.  Never returns.
7605 void
7606 runcmd(struct cmd *cmd)
7607 {
7608   int p[2];
7609   struct backcmd *bcmd;
7610   struct execcmd *ecmd;
7611   struct listcmd *lcmd;
7612   struct pipecmd *pcmd;
7613   struct redircmd *rcmd;
7614
7615   if(cmd == 0)
7616     exit();
7617
7618   switch(cmd->type){
7619   default:
7620     panic("runcmd");
7621
7622   case EXEC:
7623     ecmd = (struct execcmd*)cmd;
7624     if(ecmd->argv[0] == 0)
7625       exit();
7626     exec(ecmd->argv[0], ecmd->argv);
7627     printf(2, "exec %s failed\n", ecmd->argv[0]);
7628     break;
7629
7630   case REDIR:
7631     rcmd = (struct redircmd*)cmd;
7632     close(rcmd->fd);
7633     if(open(rcmd->file, rcmd->mode) < 0){
7634       printf(2, "open %s failed\n", rcmd->file);
7635       exit();
7636     }
7637     runcmd(rcmd->cmd);
7638     break;
7639
7640   case LIST:
7641     lcmd = (struct listcmd*)cmd;
7642     if(fork1() == 0)
7643       runcmd(lcmd->left);
7644     wait();
7645     runcmd(lcmd->right);
7646     break;
7647
7648
7649
```



```
7650   case PIPE:
7651     pcmd = (struct pipecmd*)cmd;
7652     if(pipe(p) < 0)
7653       panic("pipe");
7654     if(fork1() == 0){
7655       close(1);
7656       dup(p[1]);
7657       close(p[0]);
7658       close(p[1]);
7659       runcmd(pcmd->left);
7660     }
7661     if(fork1() == 0){
7662       close(0);
7663       dup(p[0]);
7664       close(p[0]);
7665       close(p[1]);
7666       runcmd(pcmd->right);
7667     }
7668     close(p[0]);
7669     close(p[1]);
7670     wait();
7671     wait();
7672     break;
7673
7674   case BACK:
7675     bcmd = (struct backcmd*)cmd;
7676     if(fork1() == 0)
7677       runcmd(bcmd->cmd);
7678     break;
7679   }
7680   exit();
7681 }
7682
7683 int
7684 getcmd(char *buf, int nbuf)
7685 {
7686   printf(2, "$ ");
7687   memset(buf, 0, nbuf);
7688   gets(buf, nbuf);
7689   if(buf[0] == 0) // EOF
7690     return -1;
7691   return 0;
7692 }
7693
7694
7695
7696
7697
7698
7699
```

```
7700 int
7701 main(void)
7702 {
7703   static char buf[100];
7704   int fd;
7705
7706   // Assumes three file descriptors open.
7707   while((fd = open("console", O_RDWR)) >= 0){
7708     if(fd >= 3){
7709       close(fd);
7710       break;
7711     }
7712   }
7713
7714   // Read and run input commands.
7715   while(getcmd(buf, sizeof(buf)) >= 0){
7716     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
7717       // Clumsy but will have to do for now.
7718       // Chdir has no effect on the parent if run in the child.
7719       buf[strlen(buf)-1] = 0;  // chop \n
7720       if(chdir(buf+3) < 0)
7721         printf(2, "cannot cd %s\n", buf+3);
7722       continue;
7723     }
7724     if(fork1() == 0)
7725       runcmd(parsecmd(buf));
7726     wait();
7727   }
7728   exit();
7729 }
7730
7731 void
7732 panic(char *s)
7733 {
7734   printf(2, "%s\n", s);
7735   exit();
7736 }
7737
7738 int
7739 fork1(void)
7740 {
7741   int pid;
7742
7743   pid = fork();
7744   if(pid == -1)
7745     panic("fork");
7746   return pid;
7747 }
7748
7749
```



```
7750 // Constructors
7751
7752 struct cmd*
7753 execcmd(void)
7754 {
7755   struct execcmd *cmd;
7756
7757   cmd = malloc(sizeof(*cmd));
7758   memset(cmd, 0, sizeof(*cmd));
7759   cmd->type = EXEC;
7760   return (struct cmd*)cmd;
7761 }
7762
7763 struct cmd*
7764 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
7765 {
7766   struct redircmd *cmd;
7767
7768   cmd = malloc(sizeof(*cmd));
7769   memset(cmd, 0, sizeof(*cmd));
7770   cmd->type = REDIR;
7771   cmd->cmd = subcmd;
7772   cmd->file = file;
7773   cmd->efile = efile;
7774   cmd->mode = mode;
7775   cmd->fd = fd;
7776   return (struct cmd*)cmd;
7777 }
7778
7779 struct cmd*
7780 pipecmd(struct cmd *left, struct cmd *right)
7781 {
7782   struct pipecmd *cmd;
7783
7784   cmd = malloc(sizeof(*cmd));
7785   memset(cmd, 0, sizeof(*cmd));
7786   cmd->type = PIPE;
7787   cmd->left = left;
7788   cmd->right = right;
7789   return (struct cmd*)cmd;
7790 }
7791
7792
7793
7794
7795
7796
7797
7798
7799
```

```
7800 struct cmd*
7801 listcmd(struct cmd *left, struct cmd *right)
7802 {
7803   struct listcmd *cmd;
7804
7805   cmd = malloc(sizeof(*cmd));
7806   memset(cmd, 0, sizeof(*cmd));
7807   cmd->type = LIST;
7808   cmd->left = left;
7809   cmd->right = right;
7810   return (struct cmd*)cmd;
7811 }
7812
7813 struct cmd*
7814 backcmd(struct cmd *subcmd)
7815 {
7816   struct backcmd *cmd;
7817
7818   cmd = malloc(sizeof(*cmd));
7819   memset(cmd, 0, sizeof(*cmd));
7820   cmd->type = BACK;
7821   cmd->cmd = subcmd;
7822   return (struct cmd*)cmd;
7823 }
7824
7825
7826
7827
7828
7829
7830
7831
7832
7833
7834
7835
7836
7837
7838
7839
7840
7841
7842
7843
7844
7845
7846
7847
7848
7849
```



```
7850 // Parsing
7851
7852 char whitespace[] = " \t\r\n\v";
7853 char symbols[] = "<|>&;()";
7854
7855 int
7856 gettoken(char **ps, char *es, char **q, char **eq)
7857 {
7858   char *s;
7859   int ret;
7860
7861   s = *ps;
7862   while(s < es && strchr(whitespace, *s))
7863     s++;
7864   if(q)
7865     *q = s;
7866   ret = *s;
7867   switch(*s){
7868   case 0:
7869     break;
7870   case '|':
7871   case '(':
7872   case ')':
7873   case ';':
7874   case '&':
7875   case '<':
7876     s++;
7877     break;
7878   case '>':
7879     s++;
7880     if(*s == '>'){
7881       ret = '+';
7882       s++;
7883     }
7884     break;
7885   default:
7886     ret = 'a';
7887     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
7888       s++;
7889     break;
7890   }
7891   if(eq)
7892     *eq = s;
7893
7894   while(s < es && strchr(whitespace, *s))
7895     s++;
7896   *ps = s;
7897   return ret;
7898 }
7899
```

```
7900 int
7901 peek(char **ps, char *es, char *toks)
7902 {
7903   char *s;
7904
7905   s = *ps;
7906   while(s < es && strchr(whitespace, *s))
7907     s++;
7908   *ps = s;
7909   return *s && strchr(toks, *s);
7910 }
7911
7912 struct cmd *parseline(char**, char*);
7913 struct cmd *parsepipe(char**, char*);
7914 struct cmd *parseexec(char**, char*);
7915 struct cmd *nulterminate(struct cmd*);
7916
7917 struct cmd*
7918 parsecmd(char *s)
7919 {
7920   char *es;
7921   struct cmd *cmd;
7922
7923   es = s + strlen(s);
7924   cmd = parseline(&s, es);
7925   peek(&s, es, "");
7926   if(s != es){
7927     printf(2, "leftovers: %s\n", s);
7928     panic("syntax");
7929   }
7930   nulterminate(cmd);
7931   return cmd;
7932 }
7933
7934 struct cmd*
7935 parseline(char **ps, char *es)
7936 {
7937   struct cmd *cmd;
7938
7939   cmd = parsepipe(ps, es);
7940   while(peek(ps, es, "&")){
7941     gettoken(ps, es, 0, 0);
7942     cmd = backcmd(cmd);
7943   }
7944   if(peek(ps, es, ";")){
7945     gettoken(ps, es, 0, 0);
7946     cmd = listcmd(cmd, parseline(ps, es));
7947   }
7948   return cmd;
7949 }
```



```
7950 struct cmd*
7951 parsepipe(char **ps, char *es)
7952 {
7953   struct cmd *cmd;
7954
7955   cmd = parseexec(ps, es);
7956   if(peek(ps, es, "|")){
7957     gettoken(ps, es, 0, 0);
7958     cmd = pipecmd(cmd, parsepipe(ps, es));
7959   }
7960   return cmd;
7961 }
7962
7963 struct cmd*
7964 parseredirs(struct cmd *cmd, char **ps, char *es)
7965 {
7966   int tok;
7967   char *q, *eq;
7968
7969   while(peek(ps, es, "<>")){
7970     tok = gettoken(ps, es, 0, 0);
7971     if(gettoken(ps, es, &q, &eq) != 'a')
7972       panic("missing file for redirection");
7973     switch(tok){
7974     case '<':
7975       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
7976       break;
7977     case '>':
7978       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
7979       break;
7980     case '+':  // >>
7981       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
7982       break;
7983     }
7984   }
7985   return cmd;
7986 }
7987
7988
7989
7990
7991
7992
7993
7994
7995
7996
7997
7998
7999
```

```
8000 struct cmd*
8001 parseblock(char **ps, char *es)
8002 {
8003   struct cmd *cmd;
8004
8005   if(!peek(ps, es, "("))
8006     panic("parseblock");
8007   gettoken(ps, es, 0, 0);
8008   cmd = parseline(ps, es);
8009   if(!peek(ps, es, ")"))
8010     panic("syntax - missing )");
8011   gettoken(ps, es, 0, 0);
8012   cmd = parseredirs(cmd, ps, es);
8013   return cmd;
8014 }
8015
8016 struct cmd*
8017 parseexec(char **ps, char *es)
8018 {
8019   char *q, *eq;
8020   int tok, argc;
8021   struct execcmd *cmd;
8022   struct cmd *ret;
8023
8024   if(peek(ps, es, "("))
8025     return parseblock(ps, es);
8026
8027   ret = execcmd();
8028   cmd = (struct execcmd*)ret;
8029
8030   argc = 0;
8031   ret = parseredirs(ret, ps, es);
8032   while(!peek(ps, es, "|)&;")){
8033     if((tok=gettoken(ps, es, &q, &eq)) == 0)
8034       break;
8035     if(tok != 'a')
8036       panic("syntax");
8037     cmd->argv[argc] = q;
8038     cmd->eargv[argc] = eq;
8039     argc++;
8040     if(argc >= MAXARGS)
8041       panic("too many args");
8042     ret = parseredirs(ret, ps, es);
8043   }
8044   cmd->argv[argc] = 0;
8045   cmd->eargv[argc] = 0;
8046   return ret;
8047 }
8048
8049
```



```
8050 // NUL-terminate all the counted strings.
8051 struct cmd*
8052 nulterminate(struct cmd *cmd)
8053 {
8054   int i;
8055   struct backcmd *bcmd;
8056   struct execcmd *ecmd;
8057   struct listcmd *lcmd;
8058   struct pipecmd *pcmd;
8059   struct redircmd *rcmd;
8060
8061   if(cmd == 0)
8062     return 0;
8063
8064   switch(cmd->type){
8065   case EXEC:
8066     ecmd = (struct execcmd*)cmd;
8067     for(i=0; ecmd->argv[i]; i++)
8068       *ecmd->eargv[i] = 0;
8069     break;
8070
8071   case REDIR:
8072     rcmd = (struct redircmd*)cmd;
8073     nulterminate(rcmd->cmd);
8074     *rcmd->efile = 0;
8075     break;
8076
8077   case PIPE:
8078     pcmd = (struct pipecmd*)cmd;
8079     nulterminate(pcmd->left);
8080     nulterminate(pcmd->right);
8081     break;
8082
8083   case LIST:
8084     lcmd = (struct listcmd*)cmd;
8085     nulterminate(lcmd->left);
8086     nulterminate(lcmd->right);
8087     break;
8088
8089   case BACK:
8090     bcmd = (struct backcmd*)cmd;
8091     nulterminate(bcmd->cmd);
8092     break;
8093   }
8094   return cmd;
8095 }
8096
8097
8098
8099
```