

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/6.828/2012/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 (entryother.S, mp.h, mp.c, lapic.c)
FreeBSD (ioapic.c)
NetBSD (console.c)

The following people have made contributions:

Russ Cox (context switching, locking)
Cliff Frey (MP)
Xiao Yu (MP)
Nickolai Zeldovich
Austin Clements

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

The code in the files that constitute xv6 is
Copyright 2006-2014 Frans Kaashoek, Robert Morris, and Russ Cox.

ERROR REPORTS

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

BUILDING AND RUNNING XV6

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

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

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

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

# basic headers	# system calls	# string operations
01 types.h	31 traps.h	66 string.c
01 param.h	32 vectors.pl	
02 memlayout.h	32 trapasm.S	# low-level hardware
02 defs.h	33 trap.c	67 mp.h
04 x86.h	34 syscall.h	69 mp.c
06 asm.h	35 syscall.c	71 lapic.c
07 mmu.h	36 sysproc.c	74 ioapic.c
09 elf.h		75 picirq.c
	# file system	76 kbd.h
# entering xv6	37 buf.h	77 kbd.c
10 entry.S	38 fcntl.h	78 console.c
11 entryother.S	38 stat.h	81 timer.c
12 main.c	39 fs.h	82 uart.c
	40 file.h	
# locks	41 ide.c	# user-level
15 spinlock.h	43 bio.c	83 initcode.S
15 spinlock.c	45 log.c	83 usys.S
	47 fs.c	84 init.c
# processes	56 file.c	84 sh.c
17 vm.c	58 sysfile.c	
23 proc.h	63 exec.c	# bootloader
24 proc.c		90 bootasm.S
29 swtch.S	# pipes	91 bootmain.c
30 kalloc.c	64 pipe.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 2658
      0374 2428 2466 2657 2658
```

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

```

acquire 1574          3761 4187 4216 4221 4260
    0377 1574 1578 2460 2587
    2625 2658 2717 2774 2818
    2833 2866 2879 3076 3093
    3366 3722 3742 4207 4265
    4370 4430 4630 4657 4674
    4731 5008 5041 5061 5090
    5110 5120 5629 5654 5668
    6513 6534 6555 7860 8016
allocproc 2455
    2455 2507 2560
allocuvm 1953
    0422 1953 1967 2537 6346
    6358
alltraps 3254
    3209 3217 3230 3235 3253
    3254
ALT 7610
    7610 7638 7640
argfd 5819
    5819 5856 5871 5883 5894
    5906
argint 3545
    0395 3545 3558 3574 3683
    3706 3720 5824 5871 5883
    6108 6176 6177 6231
argptr 3554
    0396 3554 5871 5883 5906
    6257
argstr 3571
    0397 3571 5918 6008 6108
    6157 6175 6207 6231
__attribute__ 1310
    0271 0365 1209 1310
BACK 8461
    8461 8574 8720 8989
backcmd 8496 8714
    8496 8509 8575 8714 8716
    8842 8955 8990
BACKSPACE 7950
    7950 7967 7994 8026 8032
balloc 4804
    4804 4826 5167 5175 5179
BLOCK 3945
    3945 4813 4838
B_BUSY 3759
    3759 4258 4376 4377 4390
    4393 4416 4427 4439
B_DIRTY 3761
    3761 4187 4216 4221 4260
    4278 4390 4418 4739
begin_op 4628
    0335 2620 4628 5683 5774
    5921 6011 6111 6156 6174
    6206 6320
bfree 4831
    4831 5214 5224 5227
bget 4366
    4366 4398 4406
binit 4338
    0262 1231 4338
bmap 5160
    5160 5186 5269 5319
bootmain 9117
    9068 9117
BPB 3942
    3942 3945 4812 4814 4839
bread 4402
    0263 4402 4577 4578 4590
    4606 4688 4689 4781 4792
    4813 4838 4963 4984 5068
    5176 5220 5269 5319
brelse 4425
    0264 4425 4428 4581 4582
    4597 4614 4692 4693 4783
    4795 4819 4824 4845 4969
    4972 4993 5076 5182 5226
    5272 5323
BSIZE 3911
    3911 3922 3936 3942 4558
    4579 4690 4793 5269 5270
    5271 5315 5319 5320 5321
buf 3750
    0250 0263 0264 0265 0307
    0334 2120 2123 2132 2134
    3750 3754 3755 3756 4111
    4126 4129 4175 4204 4254
    4256 4259 4326 4330 4334
    4340 4353 4365 4368 4401
    4404 4414 4425 4505 4577
    4578 4590 4591 4597 4606
    4607 4613 4614 4688 4689
    4722 4768 4779 4790 4807
    4833 4956 4981 5055 5163
    5209 5255 5305 7829 7840
    7844 7847 8003 8024 8038
    8068 8101 8108 8584 8587
    8588 8589 8603 8615 8616
    8619 8620 8621 8625

```

```

B_VALID 3760
    3760 4220 4260 4278 4407
bwrite 4414
    0265 4414 4417 4580 4613
    4691
bzero 4788
    4788 4820
C 7631 8009
    7631 7679 7704 7705 7706
    7707 7708 7710 8009 8019
    8022 8029 8040 8069
CAPSLOCK 7612
    7612 7645 7786
cgaputc 7955
    7955 7998
clearpteu 2029
    0431 2029 2035 6360
cli 0557
    0557 0559 1126 1660 7910
    7989 9012
cmd 8465
    8465 8477 8486 8487 8492
    8493 8498 8502 8506 8515
    8518 8523 8531 8537 8541
    8551 8575 8577 8652 8655
    8657 8658 8659 8660 8663
    8664 8666 8668 8669 8670
    8671 8672 8673 8674 8675
    8676 8679 8680 8682 8684
    8685 8686 8687 8688 8689
    8700 8701 8703 8705 8706
    8707 8708 8709 8710 8713
    8714 8716 8718 8719 8720
    8721 8722 8812 8813 8814
    8815 8817 8821 8824 8830
    8831 8834 8837 8839 8842
    8846 8848 8850 8853 8855
    8858 8860 8863 8864 8875
    8878 8881 8885 8900 8903
    8908 8912 8913 8916 8921
    8922 8928 8937 8938 8944
    8945 8951 8952 8961 8964
    8966 8972 8973 8978 8984
    8990 8991 8994
CMOS_PORT 7285
    7285 7299 7300 7338
CMOS_RETURN 7286
    7286 7341
CMOS_STATATA 7325
    7325 7373
CMOS_STATB 7326
    7326 7366
CMOS_UIP 7327
    7327 7373
COM1 8213
    8213 8223 8226 8227 8228
    8229 8230 8231 8234 8240
    8241 8257 8259 8267 8269
commit 4701
    4553 4673 4701
CONSOLE 4037
    4037 8121 8122
consoleinit 8116
    0268 1227 8116
consoleintr 8012
    0270 7798 8012 8275
consoleread 8051
    8051 8122
consolewrite 8101
    8101 8121
consputc 7986
    7816 7847 7868 7886 7889
    7893 7894 7986 8026 8032
    8039 8108
context 2343
    0251 0374 2306 2343 2361
    2488 2489 2490 2491 2728
    2766 2928
CONV 7382
    7382 7383 7384 7385 7386
    7387 7388 7389
copyout 2118
    0430 2118 6368 6379
copyuvm 2053
    0427 2053 2064 2066 2564
cprintf 7852
    0269 1224 1264 1967 2926
    2930 2932 3390 3403 3408
    3633 7019 7039 7261 7462
    7852 7912 7913 7914 7917
cpu 2304
    0310 1224 1264 1266 1278
    1506 1566 1587 1608 1646
    1661 1662 1670 1672 1718
    1731 1737 1876 1877 1878
    1879 2304 2314 2318 2329
    2728 2759 2765 2766 2767
    3365 3390 3391 3403 3404
    3408 3410 6913 6914 7261
    7912

```

```

cpunum 7251
  0325 1288 1724 7251 7473
  7482
CR0_PE 0727
  0727 1135 1171 9043
CR0_PG 0737
  0737 1050 1171
CR0_WP 0733
  0733 1050 1171
CR4_PSE 0739
  0739 1043 1164
create 6057
  6057 6077 6090 6094 6114
  6157 6178
CRTPORT 7951
  7951 7960 7961 7962 7963
  7978 7979 7980 7981
CTL 7609
  7609 7635 7639 7785
DAY 7332
  7332 7355
deallocumv 1982
  0423 1968 1982 2016 2540
DEVSPACE 0204
  0204 1832 1845
devsw 4030
  4030 4035 5258 5260 5308
  5310 5611 8121 8122
dinode 3926
  3926 3936 4957 4964 4982
  4985 5056 5069
dirent 3950
  3950 5364 5405 5966 6004
dirlink 5402
  0287 5371 5402 5417 5425
  5941 6089 6093 6094
dirlookup 5361
  0288 5361 5367 5409 5525
  6023 6067
DIRSIZ 3948
  3948 3952 5355 5422 5478
  5479 5542 5915 6005 6061
DPL_USER 0779
  0779 1727 1728 2514 2515
  3323 3418 3427
EOESC 7616
  7616 7770 7774 7775 7777
  7780
elfhdr 0955
  0955 6315 9119 9124
  ELF_MAGIC 0952
    0952 6331 9130
  ELF_PROG_LOAD 0986
    0986 6342
  end_op 4653
    0336 2622 4653 5685 5779
    5923 5930 5948 5957 6013
    6047 6052 6116 6121 6127
    6136 6140 6158 6162 6179
    6183 6208 6214 6219 6322
    6352 6405
  entry 1040
    0961 1036 1039 1040 3202
    3203 6392 6771 9121 9145
    9146
  EOI 7115
    7115 7234 7275
  ERROR 7136
    7136 7227
  ESR 7118
    7118 7230 7231
  exec 6310
    0274 6247 6310 8368 8429
    8430 8526 8527
  EXEC 8457
    8457 8522 8659 8965
  execcmd 8469 8653
    8469 8510 8523 8653 8655
    8921 8927 8928 8956 8966
  exit 2604
    0359 2604 2642 3355 3359
    3419 3428 3668 8316 8319
    8361 8426 8431 8516 8525
    8535 8580 8628 8635
  EXTMEM 0202
    0202 0208 1829
  fdalloc 5838
    5838 5858 6132 6262
  fetchint 3517
    0398 3517 3547 6238
  fetchstr 3529
    0399 3529 3576 6244
  file 4000
    0252 0277 0278 0279 0281
    0282 0283 0351 2364 4000
    4770 5608 5614 5624 5627
    5630 5651 5652 5664 5666
    5702 5715 5752 5813 5819
    5822 5838 5853 5867 5879
    5892 5903 6105 6254 6456

```

```

  6471 7810 8208 8478 8533
  8534 8664 8672 8872
filealloc 5625
  0277 5625 6132 6477
fileclose 5664
  0278 2615 5664 5670 5897
  6134 6265 6266 6504 6506
filedup 5652
  0279 2579 5652 5656 5860
fileinit 5618
  0280 1232 5618
fileread 5715
  0281 5715 5730 5873
filestat 5702
  0282 5702 5908
filewrite 5752
  0283 5752 5784 5789 5885
FL_IF 0710
  0710 1662 1668 2518 2763
  7258
fork 2554
  0360 2554 3662 8360 8423
  8425 8643 8645
forkl 8639
  8500 8542 8554 8561 8576
  8624 8639
forkret 2783
  2417 2491 2783
freerange 3051
  3011 3034 3040 3051
freevm 2010
  0424 2010 2015 2078 2671
  6395 6402
gatedesc 0901
  0523 0526 0901 3311
getcallerpcs 1626
  0378 1588 1626 2928 7915
getcmd 8584
  8584 8615
gettoken 8756
  8756 8841 8845 8857 8870
  8871 8907 8911 8933
growproc 2531
  0361 2531 3709
havedisk1 4128
  4128 4164 4262
holding 1644
  0379 1577 1604 1644 2757
HOURS 7331
  7331 7354
  ialloc 4953
    0289 4953 4974 6076 6077
  IBLOCK 3939
    3939 4963 4984 5068
  I_BUSY 4025
    4025 5062 5064 5087 5091
    5113 5115
  ICRHI 7129
    7129 7237 7307 7319
  ICRLO 7119
    7119 7238 7239 7308 7310
    7320
  ID 7112
    7112 7148 7266
  IDE_BSY 4113
    4113 4137
  IDE_CMD_READ 4118
    4118 4191
  IDE_CMD_WRITE 4119
    4119 4188
  IDE_DF 4115
    4115 4139
  IDE_DRDY 4114
    4114 4137
  IDE_ERR 4116
    4116 4139
  ideinit 4151
    0305 1234 4151
  ideintr 4202
    0306 3374 4202
  idelock 4125
    4125 4155 4207 4209 4228
    4265 4279 4282
  iderw 4254
    0307 4254 4259 4261 4263
    4408 4419
  idestart 4175
    4129 4175 4178 4226 4275
  idewait 4133
    4133 4158 4180 4216
  idtinit 3329
    0406 1265 3329
  idup 5039
    0290 2580 5039 5512
  iget 5004
    4923 4970 5004 5024 5379
    5510
  iinit 4918
    0291 1233 4918
  ilock 5053

```

```

0292 5053 5059 5079 5515
5705 5724 5775 5927 5940
5953 6017 6025 6065 6069
6079 6124 6211 6325 8063
8083 8110
inb 0453
0453 4137 4163 7054 7341
7764 7767 7961 7963 8234
8240 8241 8257 8267 8269
9023 9031 9154
initlock 1562
0380 1562 2425 3032 3325
4155 4342 4562 4920 5620
6485 8118 8119
initlog 4556
0333 2794 4556 4559
inituvm 1903
0425 1903 1908 2511
inode 4012
0253 0287 0288 0289 0290
0292 0293 0294 0295 0296
0298 0299 0300 0301 0302
0426 1918 2365 4006 4012
4031 4032 4773 4914 4923
4952 4979 5003 5006 5012
5038 5039 5053 5085 5108
5130 5160 5206 5237 5252
5302 5360 5361 5402 5406
5504 5507 5539 5550 5916
5963 6003 6056 6060 6106
6154 6169 6204 6316 8051
8101
INPUT_BUF 8000
8000 8003 8024 8036 8038
8040 8068
insl 0462
0462 0464 4217 9173
install_trans 4572
4572 4621 4706
INT_DISABLED 7419
7419 7467
ioapic 7427
7007 7029 7030 7424 7427
7436 7437 7443 7444 7458
IOAPIC 7408
7408 7458
ioapicenable 7473
0310 4157 7473 8126 8243
ioapicid 6917
0311 6917 7030 7047 7461
7462
ioapicinit 7451
0312 1226 7451 7462
ioapicread 7434
7434 7459 7460
ioapicwrite 7441
7441 7467 7468 7481 7482
IO_PIC1 7507
7507 7520 7535 7544 7547
7552 7562 7576 7577
IO_PIC2 7508
7508 7521 7536 7565 7566
7567 7570 7579 7580
IO_TIMER1 8159
8159 8168 8178 8179
IPB 3936
3936 3939 3945 4964 4985
5069
iput 5108
0293 2621 5108 5114 5133
5410 5533 5684 5946 6218
IRQ_COM1 3183
3183 3384 8242 8243
IRQ_ERROR 3185
3185 7227
IRQ_IDE 3184
3184 3373 3377 4156 4157
IRQ_KBD 3182
3182 3380 8125 8126
IRQ_SLAVE 7510
7510 7514 7552 7567
IRQ_SPURIOUS 3186
3186 3389 7207
IRQ_TIMER 3181
3181 3364 3423 7214 8180
isdirempty 5963
5963 5970 6029
ismp 6915
0339 1235 6915 7012 7020
7040 7043 7455 7475
itrunc 5206
4773 5117 5206
iunlock 5085
0294 5085 5088 5132 5522
5707 5727 5778 5936 6139
6217 8056 8105
iunlockput 5130
0295 5130 5517 5526 5529
5929 5942 5945 5956 6030
6041 6045 6051 6068 6072

```

```

6096 6126 6135 6161 6182
6213 6351 6404
iupdate 4979
0296 4979 5119 5232 5328
5935 5955 6039 6044 6083
6087
I_INVALID 4026
4026 5067 5077 5111
kalloc 3088
0315 1294 1763 1842 1909
1965 2069 2473 3088 6479
KBDATAP 7604
7604 7767
kbdgetc 7756
7756 7798
kbdtintr 7796
0321 3381 7796
KBS_DIB 7603
7603 7765
KBSTAP 7602
7602 7764
KERNBASE 0207
0207 0208 0212 0213 0217
0218 0220 0221 1315 1633
1829 1958 2016
KERNLINK 0208
0208 1830
KEY_DEL 7628
7628 7669 7691 7715
KEY_DN 7622
7622 7665 7687 7711
KEY_END 7620
7620 7668 7690 7714
KEY_HOME 7619
7619 7668 7690 7714
KEY_INS 7627
7627 7669 7691 7715
KEY_LF 7623
7623 7667 7689 7713
KEY_PGDN 7626
7626 7666 7688 7712
KEY_PGUP 7625
7625 7666 7688 7712
KEY_RT 7624
7624 7667 7689 7713
KEY_UP 7621
7621 7665 7687 7711
kfree 3065
0316 1998 2000 2020 2023
2565 2669 3056 3065 3070
6502 6523
kill 2875
0362 2875 3409 3685 8367
kinit1 3030
0317 1219 3030
kinit2 3038
0318 1238 3038
KSTACKSIZE 0151
0151 1054 1063 1295 1879
2477
kvmalloc 1857
0418 1220 1857
lapiceoi 7272
0327 3371 3375 3382 3386
3392 7272
lapicinit 7201
0328 1222 1256 7201
lapicstartap 7291
0329 1299 7291
lapicw 7145
7145 7207 7213 7214 7215
7218 7219 7224 7227 7230
7231 7234 7237 7238 7243
7275 7307 7308 7310 7319
7320
lcr3 0590
0590 1868 1883
lgdt 0512
0512 0520 1133 1733 9041
lidt 0526
0526 0534 3331
LINT0 7134
7134 7218
LINT1 7135
7135 7219
LIST 8460
8460 8540 8707 8983
listcmd 8490 8701
8490 8511 8541 8701 8703
8846 8957 8984
loadgs 0551
0551 1734
loaduvm 1918
0426 1918 1924 1927 6348
log 4537 4550
4537 4550 4562 4564 4565
4566 4576 4577 4578 4590
4593 4594 4595 4606 4609
4610 4611 4622 4630 4632
4633 4634 4636 4638 4639

```

```

4657 4658 4659 4660 4661
4663 4666 4668 4674 4675
4676 4677 4687 4688 4689
4703 4707 4726 4728 4731
4732 4733 4736 4737 4738
4740
logheader 4532
4532 4544 4558 4559 4591
4607
LOGSIZE 0160
0160 4534 4634 4726 5767
log_write 4722
0334 4722 4729 4794 4818
4844 4968 4992 5180 5322
ltr 0538
0538 0540 1880
mappages 1779
1779 1848 1911 1972 2072
MAXARG 0158
0158 6227 6314 6365
MAXARGS 8463
8463 8471 8472 8940
MAXFILE 3923
3923 5315
MAXOPBLOCKS 0159
0159 0160 0161 4634
memcmp 6615
0386 6615 6945 6988 7376
memmove 6631
0387 1285 1912 2071 2132
4579 4690 4782 4991 5075
5271 5321 5479 5481 6631
6654 7973
memset 6604
0388 1766 1844 1910 1971
2490 2513 3073 4793 4966
6034 6234 6604 7975 8587
8658 8669 8685 8706 8719
microdelay 7281
0330 7281 7309 7311 7321
7339 8258
min 4772
4772 5270 5320
MINS 7330
7330 7353
MONTH 7333
7333 7356
mp 6752
6752 6908 6937 6944 6945
6946 6955 6960 6964 6965
6968 6969 6980 6983 6985
6987 6994 7004 7010 7050
mpbcpu 6920
0340 6920
MPBUS 6802
6802 7033
mpconf 6763
6763 6979 6982 6987 7005
mpconfig 6980
6980 7010
mpenter 1252
1252 1296
mpinit 7001
0341 1221 7001 7019 7039
mpioapic 6789
6789 7007 7029 7031
MPIOPIC 6803
6803 7028
MPIOINTR 6804
6804 7034
MPLINTR 6805
6805 7035
mpmain 1262
1209 1241 1257 1262
mpproc 6778
6778 7006 7017 7026
MPPROC 6801
6801 7016
mpsearch 6956
6956 6985
mpsearch1 6938
6938 6964 6968 6971
multiboot_header 1025
1024 1025
namecmp 5353
0297 5353 5374 6020
namei 5540
0298 2523 5540 5922 6120
6207 6321
nameiparent 5551
0299 5505 5520 5532 5551
5938 6012 6063
namex 5505
5505 5543 5553
NBUF 0161
0161 4330 4353
ncpu 6916
1224 1287 2319 4157 6916
7018 7019 7023 7024 7025
7045

```

```

NCPU 0152
0152 2318 6913
NDEV 0156
0156 5258 5308 5611
NDIRECT 3921
3921 3923 3932 4023 5165
5170 5174 5175 5212 5219
5220 5227 5228
NELEM 0434
0434 1847 2922 3630 6236
nextpid 2416
2416 2469
NFILE 0154
0154 5614 5630
NINDIRECT 3922
3922 3923 5172 5222
NINODE 0155
0155 4914 5012
NO 7606
7606 7652 7655 7657 7658
7659 7660 7662 7674 7677
7679 7680 7681 7682 7684
7702 7703 7705 7706 7707
7708
NOFILE 0153
0153 2364 2577 2613 5826
5842
NPENTRIES 0821
0821 1311 2017
NPROC 0150
0150 2411 2461 2631 2662
2718 2857 2880 2919
NPENTRIES 0822
0822 1994
NSEGS 2301
1711 2301 2308
nulterminate 8952
8815 8830 8952 8973 8979
8980 8985 8986 8991
NUMLOCK 7613
7613 7646
O_CREATE 3803
3803 6113 8878 8881
O_RDONLY 3800
3800 6125 8875
O_RDWR 3802
3802 6146 8414 8416 8607
outb 0471
0471 4161 4170 4181 4182
4183 4184 4185 4186 4188
4191 7053 7054 7299 7300
7338 7520 7521 7535 7536
7544 7547 7552 7562 7565
7566 7567 7570 7576 7577
7579 7580 7960 7962 7978
7979 7980 7981 8177 8178
8179 8223 8226 8227 8228
8229 8230 8231 8259 9028
9036 9164 9165 9166 9167
9168 9169
outsl 0483
0483 0485 4189
outw 0477
0477 1181 1183 9074 9076
O_WRONLY 3801
3801 6145 6146 8878 8881
P2V 0218
0218 1219 1238 6962 7301
7952
panic 7905 8632
0271 1578 1605 1669 1671
1790 1846 1882 1908 1924
1927 1998 2015 2035 2064
2066 2510 2610 2642 2758
2760 2762 2764 2806 2809
3070 3405 4178 4259 4261
4263 4398 4417 4428 4559
4660 4727 4729 4826 4842
4974 5024 5059 5079 5088
5114 5186 5367 5371 5417
5425 5656 5670 5730 5784
5789 5970 6028 6036 6077
6090 6094 7863 7905 7912
8501 8520 8553 8632 8645
8828 8872 8906 8910 8936
8941
panicked 7818
7818 7918 7988
parseblock 8901
8901 8906 8925
parsecmd 8818
8502 8625 8818
parseexec 8917
8814 8855 8917
parseline 8835
8812 8824 8835 8846 8908
parsepipe 8851
8813 8839 8851 8858
parseredirs 8864
8864 8912 8931 8942

```

```

PCINT 7133
  7133 7224
pde_t 0103
  0103 0420 0421 0422 0423
  0424 0425 0426 0427 0430
  0431 1210 1270 1311 1710
  1754 1756 1779 1836 1839
  1842 1903 1918 1953 1982
  2010 2029 2052 2053 2055
  2102 2118 2355 6318
PDX 0812
  0812 1759
PDXSHIFT 0827
  0812 0818 0827 1315
peek 8801
  8801 8825 8840 8844 8856
  8869 8905 8909 8924 8932
PGROUNDNDOWN 0830
  0830 1784 1785 2125
PGROUNDUP 0829
  0829 1963 1990 3054 6357
PGSIZE 0823
  0823 0829 0830 1310 1766
  1794 1795 1844 1907 1910
  1911 1923 1925 1929 1932
  1964 1971 1972 1991 1994
  2062 2071 2072 2129 2135
  2512 2519 3055 3069 3073
  6358 6360
PHYSTOP 0203
  0203 1238 1831 1845 1846
  3069
picenable 7525
  0345 4156 7525 8125 8180
  8242
picinit 7532
  0346 1225 7532
picsetmask 7517
  7517 7527 7583
pinit 2423
  0363 1229 2423
pipe 6461
  0254 0352 0353 0354 4005
  5681 5722 5759 6461 6473
  6479 6485 6489 6493 6511
  6530 6551 8363 8552 8553
PIPE 8459
  8459 8550 8686 8977
pipealloc 6471
  0351 6259 6471

```

```

pipeclose 6511
  0352 5681 6511
pipecmd 8484 8680
  8484 8512 8551 8680 8682
  8858 8958 8978
piperead 6551
  0353 5722 6551
PIPESIZE 6459
  6459 6463 6536 6544 6566
pipewrite 6530
  0354 5759 6530
popcli 1666
  0383 1621 1666 1669 1671
  1884
printint 7826
  7826 7876 7880
proc 2353
  0255 0358 0428 1205 1558
  1706 1738 1873 1879 2315
  2330 2353 2359 2406 2411
  2414 2454 2457 2461 2504
  2535 2537 2540 2543 2544
  2557 2564 2570 2571 2572
  2578 2579 2580 2582 2606
  2609 2614 2615 2616 2621
  2623 2628 2631 2632 2640
  2655 2662 2663 2683 2689
  2710 2718 2725 2728 2733
  2761 2766 2775 2805 2823
  2824 2828 2855 2857 2877
  2880 2915 2919 3305 3354
  3356 3358 3401 3409 3410
  3412 3418 3423 3427 3505
  3519 3533 3536 3547 3560
  3629 3631 3634 3635 3657
  3691 3708 3725 4107 4766
  5512 5811 5826 5843 5844
  5896 6218 6220 6264 6304
  6386 6389 6390 6391 6392
  6393 6394 6454 6537 6557
  6911 7006 7017 7018 7019
  7022 7813 8061 8210
procdump 2904
  0364 2904 8020
proghdr 0974
  0974 6317 9120 9134
PTE_ADDR 0844
  0844 1761 1928 1996 2019
  2067 2111
PTE_FLAGS 0845

```

```

  0845 2068
PTE_P 0833
  0833 1313 1315 1760 1770
  1789 1791 1995 2018 2065
  2107
PTE_PS 0840
  0840 1313 1315
pte_t 0848
  0848 1753 1757 1761 1763
  1782 1921 1984 2031 2056
  2104
PTE_U 0835
  0835 1770 1911 1972 2036
  2109
PTE_W 0834
  0834 1313 1315 1770 1829
  1831 1832 1911 1972
PTX 0815
  0815 1772
PTXSHIFT 0826
  0815 0818 0826
pushcli 1655
  0382 1576 1655 1875
rcr2 0582
  0582 3404 3411
readeflags 0544
  0544 1659 1668 2763 7258
read_head 4588
  4588 4620
readi 5252
  0300 1933 5252 5370 5416
  5725 5969 5970 6329 6340
readsb 4777
  0286 4563 4777 4811 4837
  4960
readsect 9160
  9160 9195
readseg 9179
  9114 9127 9138 9179
recover_from_log 4618
  4552 4567 4618
REDIR 8458
  8458 8530 8670 8971
redircmd 8475 8664
  8475 8513 8531 8664 8666
  8875 8878 8881 8959 8972
REG_ID 7410
  7410 7460
REG_TABLE 7412
  7412 7467 7468 7481 7482

```

```

REG_VER 7411
  7411 7459
release 1602
  0381 1602 1605 2464 2470
  2589 2677 2684 2735 2777
  2787 2819 2832 2868 2886
  2890 3081 3098 3369 3726
  3731 3744 4209 4228 4282
  4378 4394 4442 4639 4668
  4677 4740 5015 5031 5043
  5065 5093 5116 5125 5633
  5637 5658 5672 5678 6522
  6525 6538 6547 6558 6569
  7901 8048 8062 8082 8109
ROOTDEV 0157
  0157 4563 4566 5510
ROOTINO 3910
  3910 5510
run 3014
  2911 3014 3015 3021 3067
  3077 3090
runcmd 8506
  8506 8520 8537 8543 8545
  8559 8566 8577 8625
RUNNING 2350
  2350 2727 2761 2911 3423
safestrcpy 6682
  0389 2522 2582 6386 6682
sched 2753
  0366 2641 2753 2758 2760
  2762 2764 2776 2825
scheduler 2708
  0365 1267 2306 2708 2728
  2766
SCROLLLOCK 7614
  7614 7647
SECS 7329
  7329 7352
SECTSIZE 9112
  9112 9173 9186 9189 9194
SEG 0769
  0769 1725 1726 1727 1728
  1731
SEG16 0773
  0773 1876
SEG_ASM 0660
  0660 1190 1191 9084 9085
segdesc 0752
  0509 0512 0752 0769 0773
  1711 2308

```

```

seginit 1716
0417 1223 1255 1716
SEG_KCODE 0741
0741 1150 1725 3322 3323
9053
SEG_KCPU 0743
0743 1731 1734 3266
SEG_KDATA 0742
0742 1154 1726 1878 3263
9058
SEG_NULLASM 0654
0654 1189 9083
SEG_TSS 0746
0746 1876 1877 1880
SEG_UCODE 0744
0744 1727 2514
SEG_UDATA 0745
0745 1728 2515
SETGATE 0921
0921 3322 3323
setupkvm 1837
0420 1837 1859 2060 2509
6334
SHIFT 7608
7608 7636 7637 7785
skipelem 5465
5465 5514
sleep 2803
0367 2689 2803 2806 2809
2909 3729 4279 4381 4633
4636 5063 6542 6561 8066
8379
spinlock 1501
0257 0367 0377 0379 0380
0381 0409 1501 1559 1562
1574 1602 1644 2407 2410
2803 3009 3019 3308 3313
4110 4125 4325 4329 4503
4538 4767 4913 5609 5613
6457 6462 7808 7821 8002
8206
STA_R 0669 0786
0669 0786 1190 1725 1727
9084
start 1125 8308 9011
1124 1125 1167 1175 1177
4539 4564 4577 4590 4606
4688 8307 8308 9010 9011
9067
startothers 1274
1208 1237 1274
stat 3854
0258 0282 0301 3854 4764
5237 5702 5809 5904 8403
stati 5237
0301 5237 5706
STA_W 0668 0785
0668 0785 1191 1726 1728
1731 9085
STA_X 0665 0782
0665 0782 1190 1725 1727
9084
sti 0563
0563 0565 1673 2714
stosb 0492
0492 0494 6610 9140
stosl 0501
0501 0503 6608
strlen 6701
0390 6367 6368 6701 8619
8823
strncmp 6658
0391 5355 6658
strncpy 6668
0392 5422 6668
STS_IG32 0800
0800 0927
STS_T32A 0797
0797 1876
STS_TG32 0801
0801 0927
sum 6926
6926 6928 6930 6932 6933
6945 6992
superblock 3914
0259 0286 3914 4561 4777
4808 4834 4958
SVR 7116
7116 7207
switchkvm 1866
0429 1254 1860 1866 2729
switchvm 1873
0428 1873 1882 2544 2726
6394
swtch 2958
0374 2728 2766 2957 2958
syscall 3625
0400 3357 3507 3625
SYSCALL 8353 8360 8361 8362 8363 83
8360 8361 8362 8363 8364

```

```

8365 8366 8367 8368 8369
8370 8371 8372 8373 8374
8375 8376 8377 8378 8379
8380
sys_chdir 6201
3579 3609 6201
SYS_chdir 3459
3459 3609
sys_close 5889
3580 3621 5889
SYS_close 3471
3471 3621
sys_dup 5851
3581 3610 5851
SYS_dup 3460
3460 3610
sys_exec 6225
3582 3607 6225
SYS_exec 3457
3457 3607 8312
sys_exit 3666
3583 3602 3666
SYS_exit 3452
3452 3602 8317
sys_fork 3660
3584 3601 3660
SYS_fork 3451
3451 3601
sys_fstat 5901
3585 3608 5901
SYS_fstat 3458
3458 3608
sys_getpid 3689
3586 3611 3689
SYS_getpid 3461
3461 3611
sys_kill 3679
3587 3606 3679
SYS_kill 3456
3456 3606
sys_link 5913
3588 3619 5913
SYS_link 3469
3469 3619
sys_mkdir 6151
3589 3620 6151
SYS_mkdir 3470
3470 3620
sys_mknod 6167
3590 3617 6167
SYS_mknod 3467
3467 3617
sys_open 6101
3591 3615 6101
SYS_open 3465
3465 3615
sys_pipe 6251
3592 3604 6251
SYS_pipe 3454
3454 3604
sys_read 5865
3593 3605 5865
SYS_read 3455
3455 3605
sys_sbrk 3701
3594 3612 3701
SYS_sbrk 3462
3462 3612
sys_sleep 3715
3595 3613 3715
SYS_sleep 3463
3463 3613
sys_unlink 6001
3596 3618 6001
SYS_unlink 3468
3468 3618
sys_uptime 3738
3599 3614 3738
SYS_uptime 3464
3464 3614
sys_wait 3673
3597 3603 3673
SYS_wait 3453
3453 3603
sys_write 5877
3598 3616 5877
SYS_write 3466
3466 3616
taskstate 0851
0851 2307
TDCR 7140
7140 7213
T_DEV 3852
3852 5257 5307 6178
T_DIR 3850
3850 5366 5516 5928 6029
6037 6085 6125 6157 6212
T_FILE 3851
3851 6070 6114
ticks 3314

```

0407 3314 3367 3368 3723	uartgetc 8263
3724 3729 3743	8263 8275
tickslock 3313	uartinit 8218
0409 3313 3325 3366 3369	0412 1228 8218
3722 3726 3729 3731 3742	uartintr 8273
3744	0413 3385 8273
TICR 7138	uartputc 8251
7138 7215	0414 7995 7997 8247 8251
TIMER 7130	userinit 2502
7130 7214	0368 1239 2502 2510
TIMER_16BIT 8171	uva2ka 2102
8171 8177	0421 2102 2126
TIMER_DIV 8166	V2P 0217
8166 8178 8179	0217 1830 1831
TIMER_FREQ 8165	V2P_WO 0220
8165 8166	0220 1036 1046
timerinit 8174	VER 7113
0403 1236 8174	7113 7223
TIMER_MODE 8168	wait 2653
8168 8177	0369 2653 3675 8362 8433
TIMER_RATEGEN 8170	8544 8570 8571 8626
8170 8177	waitdisk 9151
TIMER_SEL0 8169	9151 9163 9172
8169 8177	wakeup 2864
T_IRQ0 3179	0370 2864 3368 4222 4440
3179 3364 3373 3377 3380	4666 4676 5092 5122 6516
3384 3388 3389 3423 7207	6519 6541 6546 6568 8042
7214 7227 7467 7481 7547	wakeup1 2853
7566	2420 2628 2635 2853 2867
TPR 7114	walkpgdir 1754
7114 7243	1754 1787 1926 1992 2033
trap 3351	2063 2106
3202 3204 3272 3351 3403	write_head 4604
3405 3408	4604 4623 4705 4708
trapframe 0602	writei 5302
0602 2360 2481 3351	0302 5302 5424 5776 6035
trapret 3277	6036
2418 2486 3276 3277	write_log 4683
T_SYSCALL 3176	4683 4704
3176 3323 3353 8313 8318	xchg 0569
8357	0569 1266 1583 1619
tvinit 3317	YEAR 7334
0408 1230 3317	7334 7357
uart 8215	yield 2772
8215 8236 8255 8265	0371 2772 3424


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

```
0150 #define NPROC          64 // maximum number of processes
0151 #define KSTACKSIZE 4096 // size of per-process kernel stack
0152 #define NCPU            8 // maximum number of CPUs
0153 #define NOFILE          16 // open files per process
0154 #define NFILE           100 // open files per system
0155 #define NINODE           50 // maximum number of active i-nodes
0156 #define NDEV            10 // maximum major device number
0157 #define ROOTDEV         1 // device number of file system root disk
0158 #define MAXARG           32 // max exec arguments
0159 #define MAXOPBLOCKS      10 // max # of blocks any FS op writes
0160 #define LOGSIZE          (MAXOPBLOCKS*3) // max data sectors in on-disk log
0161 #define NBUF             (MAXOPBLOCKS*3) // size of disk block cache
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 // Memory layout
0201
0202 #define EXTMEM 0x100000          // Start of extended memory
0203 #define PHYSTOP 0xE000000       // Top physical memory
0204 #define DEVSPACE 0xFE000000     // Other devices are at high addresses
0205
0206 // Key addresses for address space layout (see kmap in vm.c for layout)
0207 #define KERNBASE 0x80000000      // First kernel virtual address
0208 #define KERNLINK (KERNBASE+EXTMEM) // Address where kernel is linked
0209
0210 #ifndef __ASSEMBLER__
0211
0212 static inline uint v2p(void *a) { return ((uint) (a)) - KERNBASE; }
0213 static inline void *p2v(uint a) { return (void *) ((a) + KERNBASE); }
0214
0215 #endif
0216
0217 #define V2P(a) (((uint) (a)) - KERNBASE)
0218 #define P2V(a) (((void *) (a)) + KERNBASE)
0219
0220 #define V2P_WO(x) ((x) - KERNBASE) // same as V2P, but without casts
0221 #define P2V_WO(x) ((x) + KERNBASE) // same as V2P, but without casts
0222
0223
0224
0225
0226
0227
0228
0229
0230
0231
0232
0233
0234
0235
0236
0237
0238
0239
0240
0241
0242
0243
0244
0245
0246
0247
0248
0249

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

1000 # Multiboot header, for multiboot boot loaders like GNU Grub.
1001 # http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
1002 #
1003 # Using GRUB 2, you can boot xv6 from a file stored in a
1004 # Linux file system by copying kernel or kernelmemfs to /boot
1005 # and then adding this menu entry:
1006 #
1007 # menuentry "xv6" {
1008 #   insmod ext2
1009 #   set root='(hd0,msdos1)'
1010 #   set kernel='/boot/kernel'
1011 #   echo "Loading ${kernel}..."
1012 #   multiboot ${kernel} ${kernel}
1013 #   boot
1014 # }
1015
1016 #include "asm.h"
1017 #include "memlayout.h"
1018 #include "mmu.h"
1019 #include "param.h"
1020
1021 # Multiboot header. Data to direct multiboot loader.
1022 .p2align 2
1023 .text
1024 .globl multiboot_header
1025 multiboot_header:
1026     #define magic 0x1badb002
1027     #define flags 0
1028     .long magic
1029     .long flags
1030     .long (-magic-flags)
1031
1032 # By convention, the _start symbol specifies the ELF entry point.
1033 # Since we haven't set up virtual memory yet, our entry point is
1034 # the physical address of 'entry'.
1035 .globl _start
1036 _start = V2P_W0(entry)
1037
1038 # Entering xv6 on boot processor, with paging off.
1039 .globl entry
1040 entry:
1041     # Turn on page size extension for 4Mbyte pages
1042     movl    %cr4, %eax
1043     orl     $(CR4_PSE), %eax
1044     movl    %eax, %cr4
1045     # Set page directory
1046     movl    $(V2P_W0(entrypgdir)), %eax
1047     movl    %eax, %cr3
1048     # Turn on paging.
1049     movl    %cr0, %eax

```

```

1050     orl     $(CR0_PG|CR0_WP), %eax
1051     movl    %eax, %cr0
1052
1053     # Set up the stack pointer.
1054     movl    $(stack + KSTACKSIZE), %esp
1055
1056     # Jump to main(), and switch to executing at
1057     # high addresses. The indirect call is needed because
1058     # the assembler produces a PC-relative instruction
1059     # for a direct jump.
1060     mov     $main, %eax
1061     jmp     *%eax
1062
1063 .comm stack, KSTACKSIZE
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099

```

```

1100 #include "asm.h"
1101 #include "memlayout.h"
1102 #include "mmu.h"
1103
1104 # Each non-boot CPU ("AP") is started up in response to a STARTUP
1105 # IPI from the boot CPU. Section B.4.2 of the Multi-Processor
1106 # Specification says that the AP will start in real mode with CS:IP
1107 # set to XY00:0000, where XY is an 8-bit value sent with the
1108 # STARTUP. Thus this code must start at a 4096-byte boundary.
1109 #
1110 # Because this code sets DS to zero, it must sit
1111 # at an address in the low 2^16 bytes.
1112 #
1113 # Startothers (in main.c) sends the STARTUPs one at a time.
1114 # It copies this code (start) at 0x7000. It puts the address of
1115 # a newly allocated per-core stack in start-4, the address of the
1116 # place to jump to (mpenter) in start-8, and the physical address
1117 # of entrypgdir in start-12.
1118 #
1119 # This code is identical to bootasm.S except:
1120 #   - it does not need to enable A20
1121 #   - it uses the address at start-4, start-8, and start-12
1122
1123 .code16
1124 .globl start
1125 start:
1126     cli
1127
1128     xorw    %ax,%ax
1129     movw    %ax,%ds
1130     movw    %ax,%es
1131     movw    %ax,%ss
1132
1133     lgdt    gdtdesc
1134     movl    %cr0, %eax
1135     orl     $CR0_PE, %eax
1136     movl    %eax, %cr0
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149

```

```

1150     ljmpl   $(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     # Turn on page size extension for 4Mbyte pages
1163     movl    %cr4, %eax
1164     orl     $(CR4_PSE), %eax
1165     movl    %eax, %cr4
1166     # Use enterpgdir as our initial page table
1167     movl    (start-12), %eax
1168     movl    %eax, %cr3
1169     # Turn on paging.
1170     movl    %cr0, %eax
1171     orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
1172     movl    %eax, %cr0
1173
1174     # Switch to the stack allocated by startothers()
1175     movl    (start-4), %esp
1176     # Call mpenter()
1177     call    *(start-8)
1178
1179     movw    $0x8a00, %ax
1180     movw    %ax, %dx
1181     outw    %ax, %dx
1182     movw    $0x8ae0, %ax
1183     outw    %ax, %dx
1184 spin:
1185     jmp     spin
1186
1187 .p2align 2
1188 gdt:
1189     SEG_NULLASM
1190     SEG_ASM(STA_X|STA_R, 0, 0xffffffff)
1191     SEG_ASM(STA_W, 0, 0xffffffff)
1192
1193
1194 gdtdesc:
1195     .word   (gdtdesc - gdt - 1)
1196     .long   gdt
1197
1198
1199

```

```

1200 #include "types.h"
1201 #include "defs.h"
1202 #include "param.h"
1203 #include "memlayout.h"
1204 #include "mmu.h"
1205 #include "proc.h"
1206 #include "x86.h"
1207
1208 static void startothers(void);
1209 static void mpmain(void) __attribute__((noreturn));
1210 extern pde_t *kpgdir;
1211 extern char end[]; // first address after kernel loaded from ELF file
1212
1213 // Bootstrap processor starts running C code here.
1214 // Allocate a real stack and switch to it, first
1215 // doing some setup required for memory allocator to work.
1216 int
1217 main(void)
1218 {
1219     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1220     kvmalloc(); // kernel page table
1221     mpinit(); // collect info about this machine
1222     lapicinit();
1223     seginit(); // set up segments
1224     cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
1225     picinit(); // interrupt controller
1226     ioapicinit(); // another interrupt controller
1227     consoleinit(); // I/O devices & their interrupts
1228     uartinit(); // serial port
1229     pinit(); // process table
1230     tvinit(); // trap vectors
1231     binit(); // buffer cache
1232     fileinit(); // file table
1233     iinit(); // inode cache
1234     ideinit(); // disk
1235     if(!ismp)
1236         timerinit(); // uniprocessor timer
1237     startothers(); // start other processors
1238     kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1239     userinit(); // first user process
1240     // Finish setting up this processor in mpmain.
1241     mpmain();
1242 }
1243
1244
1245
1246
1247
1248
1249

```

```

1250 // Other CPUs jump here from entryother.S.
1251 static void
1252 mpenter(void)
1253 {
1254     switchkvm();
1255     seginit();
1256     lapicinit();
1257     mpmain();
1258 }
1259
1260 // Common CPU setup code.
1261 static void
1262 mpmain(void)
1263 {
1264     cprintf("cpu%d: starting\n", cpu->id);
1265     idtinit(); // load idt register
1266     xchg(&cpu->started, 1); // tell startothers() we're up
1267     scheduler(); // start running processes
1268 }
1269
1270 pde_t entrypgdir[]; // For entry.S
1271
1272 // Start the non-boot (AP) processors.
1273 static void
1274 startothers(void)
1275 {
1276     extern uchar _binary_entryother_start[], _binary_entryother_size[];
1277     uchar *code;
1278     struct cpu *c;
1279     char *stack;
1280
1281     // Write entry code to unused memory at 0x7000.
1282     // The linker has placed the image of entryother.S in
1283     // _binary_entryother_start.
1284     code = p2v(0x7000);
1285     memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
1286
1287     for(c = cpus; c < cpus+ncpu; c++){
1288         if(c == cpus+cpunum()) // We've started already.
1289             continue;
1290
1291         // Tell entryother.S what stack to use, where to enter, and what
1292         // pgdir to use. We cannot use kpgdir yet, because the AP processor
1293         // is running in low memory, so we use entrypgdir for the APs too.
1294         stack = kalloc();
1295         *(void**)(code-4) = stack + KSTACKSIZE;
1296         *(void**)(code-8) = mpenter;
1297         *(int**)(code-12) = (void *) v2p(entrypgdir);
1298
1299         lapicstartap(c->id, v2p(code));

```

```
1300    // wait for cpu to finish mpmain()
1301    while(c->started == 0)
1302        ;
1303    }
1304 }
1305
1306 // Boot page table used in entry.S and entryother.S.
1307 // Page directories (and page tables), must start on a page boundary,
1308 // hence the "__aligned__" attribute.
1309 // Use PTE_PS in page directory entry to enable 4Mbyte pages.
1310 __attribute__((__aligned__(PGSIZE)))
1311 pde_t entrypgdir[NPDENTRIES] = {
1312     // Map VA's [0, 4MB) to PA's [0, 4MB)
1313     [0] = (0) | PTE_P | PTE_W | PTE_PS,
1314     // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1315     [KERNBASE >> PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1316 };
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334 // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1335 [KERNBASE >> PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
```

```
1350 // Blank page.
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
```

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

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

```

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

```

```

1550 // Mutual exclusion spin locks.
1551
1552 #include "types.h"
1553 #include "defs.h"
1554 #include "param.h"
1555 #include "x86.h"
1556 #include "memlayout.h"
1557 #include "mmu.h"
1558 #include "proc.h"
1559 #include "spinlock.h"
1560
1561 void
1562 initlock(struct spinlock *lk, char *name)
1563 {
1564     lk->name = name;
1565     lk->locked = 0;
1566     lk->cpu = 0;
1567 }
1568
1569 // Acquire the lock.
1570 // Loops (spins) until the lock is acquired.
1571 // Holding a lock for a long time may cause
1572 // other CPUs to waste time spinning to acquire it.
1573 void
1574 acquire(struct spinlock *lk)
1575 {
1576     pushcli(); // disable interrupts to avoid deadlock.
1577     if(holding(lk))
1578         panic("acquire");
1579
1580     // The xchg is atomic.
1581     // It also serializes, so that reads after acquire are not
1582     // reordered before it.
1583     while(xchg(&lk->locked, 1) != 0)
1584         ;
1585
1586     // Record info about lock acquisition for debugging.
1587     lk->cpu = cpu;
1588     getcallerpcs(&lk, lk->pcs);
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*)KERNBASE || 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 #include "param.h"
1701 #include "types.h"
1702 #include "defs.h"
1703 #include "x86.h"
1704 #include "memlayout.h"
1705 #include "mmu.h"
1706 #include "proc.h"
1707 #include "elf.h"
1708
1709 extern char data[]; // defined by kernel.ld
1710 pde_t *kpgdir; // for use in scheduler()
1711 struct segdesc gdt[NSEGS];
1712
1713 // Set up CPU's kernel segment descriptors.
1714 // Run once on entry on each CPU.
1715 void
1716 seginit(void)
1717 {
1718     struct cpu *c;
1719
1720     // Map "logical" addresses to virtual addresses using identity map.
1721     // Cannot share a CODE descriptor for both kernel and user
1722     // because it would have to have DPL_USR, but the CPU forbids
1723     // an interrupt from CPL=0 to DPL=3.
1724     c = &cpus[cpunum()];
1725     c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
1726     c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1727     c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
1728     c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1729
1730     // Map cpu, and curproc
1731     c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
1732
1733     lgdt(c->gdt, sizeof(c->gdt));
1734     loadgs(SEG_KCPU << 3);
1735
1736     // Initialize cpu-local storage.
1737     cpu = c;
1738     proc = 0;
1739 }
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749

```

```

1750 // Return the address of the PTE in page table pgdir
1751 // that corresponds to virtual address va. If alloc!=0,
1752 // create any required page table pages.
1753 static pte_t *
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756     pde_t *pde;
1757     pte_t *pgtab;
1758
1759     pde = &pgdir[PDX(va)];
1760     if(*pde & PTE_P){
1761         pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
1762     } else {
1763         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1764             return 0;
1765         // Make sure all those PTE_P bits are zero.
1766         memset(pgtab, 0, PGSIZE);
1767         // The permissions here are overly generous, but they can
1768         // be further restricted by the permissions in the page table
1769         // entries, if necessary.
1770         *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
1771     }
1772     return &pgtab[PTX(va)];
1773 }
1774
1775 // Create PTEs for virtual addresses starting at va that refer to
1776 // physical addresses starting at pa. va and size might not
1777 // be page-aligned.
1778 static int
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781     char *a, *last;
1782     pte_t *pte;
1783
1784     a = (char*)PGROUNDDOWN((uint)va);
1785     last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786     for(;;){
1787         if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788             return -1;
1789         if(*pte & PTE_P)
1790             panic("remap");
1791         *pte = pa | perm | PTE_P;
1792         if(a == last)
1793             break;
1794         a += PGSIZE;
1795         pa += PGSIZE;
1796     }
1797     return 0;
1798 }
1799

```

```

1800 // There is one page table per process, plus one that's used when
1801 // a CPU is not running any process (kpgdir). The kernel uses the
1802 // current process's page table during system calls and interrupts;
1803 // page protection bits prevent user code from using the kernel's
1804 // mappings.
1805 //
1806 // setupkvm() and exec() set up every page table like this:
1807 //
1808 // 0..KERNBASE: user memory (text+data+stack+heap), mapped to
1809 //      phys memory allocated by the kernel
1810 // KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
1811 // KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
1812 //      for the kernel's instructions and r/o data
1813 // data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
1814 //      rw data + free physical memory
1815 // 0xfe000000..0: mapped direct (devices such as ioapic)
1816 //
1817 // The kernel allocates physical memory for its heap and for user memory
1818 // between V2P(end) and the end of physical memory (PHYSTOP)
1819 // (directly addressable from end..P2V(PHYSTOP)).
1820 //
1821 // This table defines the kernel's mappings, which are present in
1822 // every process's page table.
1823 static struct kmap {
1824     void *virt;
1825     uint phys_start;
1826     uint phys_end;
1827     int perm;
1828 } kmap[] = {
1829     { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
1830     { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},      // kern text+rodata
1831     { (void*)data,     V2P(data),     PHYSTOP,    PTE_W}, // kern data+memory
1832     { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
1833 };
1834
1835 // Set up kernel part of a page table.
1836 pde_t*
1837 setupkvm(void)
1838 {
1839     pde_t *pgdir;
1840     struct kmap *k;
1841
1842     if((pgdir = (pde_t*)kalloc()) == 0)
1843         return 0;
1844     memset(pgdir, 0, PGSIZE);
1845     if (p2v(PHYSTOP) > (void*)DEVSPACE)
1846         panic("PHYSTOP too high");
1847     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848         if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849             (uint)k->phys_start, k->perm) < 0)

```

```

1850         return 0;
1851     return pgdir;
1852 }
1853
1854 // Allocate one page table for the machine for the kernel address
1855 // space for scheduler processes.
1856 void
1857 kvmalloc(void)
1858 {
1859     kpgdir = setupkvm();
1860     switchkvm();
1861 }
1862
1863 // Switch h/w page table register to the kernel-only page table,
1864 // for when no process is running.
1865 void
1866 switchkvm(void)
1867 {
1868     lcr3(v2p(kpgdir)); // switch to the kernel page table
1869 }
1870
1871 // Switch TSS and h/w page table to correspond to process p.
1872 void
1873 switchvm(struct proc *p)
1874 {
1875     pushcli();
1876     cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
1877     cpu->gdt[SEG_TSS].s = 0;
1878     cpu->ts.ss0 = SEG_KDATA << 3;
1879     cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
1880     ltr(SEG_TSS << 3);
1881     if(p->pgdir == 0)
1882         panic("switchvm: no pgdir");
1883     lcr3(v2p(p->pgdir)); // switch to new address space
1884     popcli();
1885 }
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899

```

```

1900 // Load the initcode into address 0 of pgdir.
1901 // sz must be less than a page.
1902 void
1903 inituvm(pde_t *pgdir, char *init, uint sz)
1904 {
1905     char *mem;
1906
1907     if(sz >= PGSIZE)
1908         panic("inituvm: more than a page");
1909     mem = kalloc();
1910     memset(mem, 0, PGSIZE);
1911     mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
1912     memmove(mem, init, sz);
1913 }
1914
1915 // Load a program segment into pgdir.  addr must be page-aligned
1916 // and the pages from addr to addr+sz must already be mapped.
1917 int
1918 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
1919 {
1920     uint i, pa, n;
1921     pte_t *pte;
1922
1923     if((uint) addr % PGSIZE != 0)
1924         panic("loaduvm: addr must be page aligned");
1925     for(i = 0; i < sz; i += PGSIZE){
1926         if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1927             panic("loaduvm: address should exist");
1928         pa = PTE_ADDR(*pte);
1929         if(sz - i < PGSIZE)
1930             n = sz - i;
1931         else
1932             n = PGSIZE;
1933         if(readi(ip, p2v(pa), offset+i, n) != n)
1934             return -1;
1935     }
1936     return 0;
1937 }
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949

```

```

1950 // Allocate page tables and physical memory to grow process from oldsz to
1951 // newsz, which need not be page aligned.  Returns new size or 0 on error.
1952 int
1953 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1954 {
1955     char *mem;
1956     uint a;
1957
1958     if(newsz >= KERNBASE)
1959         return 0;
1960     if(newsz < oldsz)
1961         return oldsz;
1962
1963     a = PGROUNDUP(oldsz);
1964     for(; a < newsz; a += PGSIZE){
1965         mem = kalloc();
1966         if(mem == 0){
1967             cprintf("allocuvm out of memory\n");
1968             deallocuvm(pgdir, newsz, oldsz);
1969             return 0;
1970         }
1971         memset(mem, 0, PGSIZE);
1972         mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U);
1973     }
1974     return newsz;
1975 }
1976
1977 // Deallocate user pages to bring the process size from oldsz to
1978 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
1979 // need to be less than oldsz.  oldsz can be larger than the actual
1980 // process size.  Returns the new process size.
1981 int
1982 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1983 {
1984     pte_t *pte;
1985     uint a, pa;
1986
1987     if(newsz >= oldsz)
1988         return oldsz;
1989
1990     a = PGROUNDUP(newsz);
1991     for(; a < oldsz; a += PGSIZE){
1992         pte = walkpgdir(pgdir, (char*)a, 0);
1993         if(!pte)
1994             a += (NPENTRIES - 1) * PGSIZE;
1995         else if((*pte & PTE_P) != 0){
1996             pa = PTE_ADDR(*pte);
1997             if(pa == 0)
1998                 panic("kfree");
1999             char *v = p2v(pa);

```

```

2000     kfree(v);
2001     *pte = 0;
2002 }
2003 }
2004 return newsz;
2005 }
2006
2007 // Free a page table and all the physical memory pages
2008 // in the user part.
2009 void
2010 freevm(pde_t *pgdir)
2011 {
2012     uint i;
2013
2014     if(pgdir == 0)
2015         panic("freevm: no pgdir");
2016     deallocvm(pgdir, KERNBASE, 0);
2017     for(i = 0; i < NPENTRIES; i++){
2018         if(pgdir[i] & PTE_P){
2019             char * v = p2v(PTE_ADDR(pgdir[i]));
2020             kfree(v);
2021         }
2022     }
2023     kfree((char*)pgdir);
2024 }
2025
2026 // Clear PTE_U on a page. Used to create an inaccessible
2027 // page beneath the user stack.
2028 void
2029 clearpteu(pde_t *pgdir, char *uva)
2030 {
2031     pte_t *pte;
2032
2033     pte = walkpgdir(pgdir, uva, 0);
2034     if(pte == 0)
2035         panic("clearpteu");
2036     *pte &= ~PTE_U;
2037 }
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049

```

```

2050 // Given a parent process's page table, create a copy
2051 // of it for a child.
2052 pde_t*
2053 copyvm(pde_t *pgdir, uint sz)
2054 {
2055     pde_t *d;
2056     pte_t *pte;
2057     uint pa, i, flags;
2058     char *mem;
2059
2060     if((d = setupkvm()) == 0)
2061         return 0;
2062     for(i = 0; i < sz; i += PGSIZE){
2063         if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
2064             panic("copyvm: pte should exist");
2065         if(!(*pte & PTE_P))
2066             panic("copyvm: page not present");
2067         pa = PTE_ADDR(*pte);
2068         flags = PTE_FLAGS(*pte);
2069         if((mem = kalloc()) == 0)
2070             goto bad;
2071         memmove(mem, (char*)p2v(pa), PGSIZE);
2072         if(mappages(d, (void*)i, PGSIZE, v2p(mem), flags) < 0)
2073             goto bad;
2074     }
2075     return d;
2076
2077 bad:
2078     freevm(d);
2079     return 0;
2080 }
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099

```

```

2100 // Map user virtual address to kernel address.
2101 char*
2102 uva2ka(pde_t *pgdir, char *uva)
2103 {
2104     pte_t *pte;
2105
2106     pte = walkpgdir(pgdir, uva, 0);
2107     if((*pte & PTE_P) == 0)
2108         return 0;
2109     if((*pte & PTE_U) == 0)
2110         return 0;
2111     return (char*)p2v(PTE_ADDR(*pte));
2112 }
2113
2114 // Copy len bytes from p to user address va in page table pgdir.
2115 // Most useful when pgdir is not the current page table.
2116 // uva2ka ensures this only works for PTE_U pages.
2117 int
2118 copyout(pde_t *pgdir, uint va, void *p, uint len)
2119 {
2120     char *buf, *pa0;
2121     uint n, va0;
2122
2123     buf = (char*)p;
2124     while(len > 0){
2125         va0 = (uint)PGROUNDDOWN(va);
2126         pa0 = uva2ka(pgdir, (char*)va0);
2127         if(pa0 == 0)
2128             return -1;
2129         n = PGSIZE - (va - va0);
2130         if(n > len)
2131             n = len;
2132         memmove(pa0 + (va - va0), buf, n);
2133         len -= n;
2134         buf += n;
2135         va = va0 + PGSIZE;
2136     }
2137     return 0;
2138 }
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149

```

```

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

```

2200 // Blank page.

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

2250 // Blank page.

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

```

2300 // Segments in proc->gdt.
2301 #define NSEGS      7
2302
2303 // Per-CPU state
2304 struct cpu {
2305     uchar id;                    // Local APIC ID; index into cpus[] below
2306     struct context *scheduler;   // swtch() here to enter scheduler
2307     struct taskstate ts;         // Used by x86 to find stack for interrupt
2308     struct segdesc gdt[NSEGS];   // x86 global descriptor table
2309     volatile uint started;       // Has the CPU started?
2310     int ncli;                    // Depth of pushcli nesting.
2311     int intena;                  // Were interrupts enabled before pushcli?
2312
2313     // Cpu-local storage variables; see below
2314     struct cpu *cpu;
2315     struct proc *proc;           // The currently-running process.
2316 };
2317
2318 extern struct cpu cpus[NCPU];
2319 extern int ncpu;
2320
2321 // Per-CPU variables, holding pointers to the
2322 // current cpu and to the current process.
2323 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2324 // and "%gs:4" to refer to proc.  seginit sets up the
2325 // %gs segment register so that %gs refers to the memory
2326 // holding those two variables in the local cpu's struct cpu.
2327 // This is similar to how thread-local variables are implemented
2328 // in thread libraries such as Linux pthreads.
2329 extern struct cpu *cpu asm("%gs:0"); // &cpus[cpunum()]
2330 extern struct proc *proc asm("%gs:4"); // cpus[cpunum()].proc
2331
2332
2333 // Saved registers for kernel context switches.
2334 // Don't need to save all the segment registers (%cs, etc),
2335 // because they are constant across kernel contexts.
2336 // Don't need to save %eax, %ecx, %edx, because the
2337 // x86 convention is that the caller has saved them.
2338 // Contexts are stored at the bottom of the stack they
2339 // describe; the stack pointer is the address of the context.
2340 // The layout of the context matches the layout of the stack in swtch.S
2341 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2342 // but it is on the stack and allocproc() manipulates it.
2343 struct context {
2344     uint edi;
2345     uint esi;
2346     uint ebx;
2347     uint ebp;
2348     uint eip;
2349 };

```

```

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

```

```

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

```

```

2450 // Look in the process table for an UNUSED proc.
2451 // If found, change state to EMBRYO and initialize
2452 // state required to run in the kernel.
2453 // Otherwise return 0.
2454 static struct proc*
2455 allocproc(void)
2456 {
2457   struct proc *p;
2458   char *sp;
2459
2460   acquire(&ptable.lock);
2461   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2462     if(p->state == UNUSED)
2463       goto found;
2464   release(&ptable.lock);
2465   return 0;
2466
2467 found:
2468   p->state = EMBRYO;
2469   p->pid = nextpid++;
2470   release(&ptable.lock);
2471
2472   // Allocate kernel stack.
2473   if((p->kstack = kalloc()) == 0){
2474     p->state = UNUSED;
2475     return 0;
2476   }
2477   sp = p->kstack + KSTACKSIZE;
2478
2479   // Leave room for trap frame.
2480   sp -= sizeof *p->tf;
2481   p->tf = (struct trapframe*)sp;
2482
2483   // Set up new context to start executing at forkret,
2484   // which returns to trapret.
2485   sp -= 4;
2486   *(uint*)sp = (uint)trapret;
2487
2488   sp -= sizeof *p->context;
2489   p->context = (struct context*)sp;
2490   memset(p->context, 0, sizeof *p->context);
2491   p->context->eip = (uint)forkret;
2492
2493   return p;
2494 }
2495
2496
2497
2498
2499

```



```

2500 // Set up first user process.
2501 void
2502 userinit(void)
2503 {
2504     struct proc *p;
2505     extern char _binary_initcode_start[], _binary_initcode_size[];
2506
2507     p = allocproc();
2508     initproc = p;
2509     if((p->pgdir = setupkvm()) == 0)
2510         panic("userinit: out of memory?");
2511     inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2512     p->sz = PGSIZE;
2513     memset(p->tf, 0, sizeof(*p->tf));
2514     p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2515     p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2516     p->tf->es = p->tf->ds;
2517     p->tf->ss = p->tf->ds;
2518     p->tf->eflags = FL_IF;
2519     p->tf->esp = PGSIZE;
2520     p->tf->eip = 0; // beginning of initcode.S
2521
2522     safestrcpy(p->name, "initcode", sizeof(p->name));
2523     p->cwd = namei("/");
2524
2525     p->state = RUNNABLE;
2526 }
2527
2528 // Grow current process's memory by n bytes.
2529 // Return 0 on success, -1 on failure.
2530 int
2531 growproc(int n)
2532 {
2533     uint sz;
2534
2535     sz = proc->sz;
2536     if(n > 0){
2537         if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2538             return -1;
2539     } else if(n < 0){
2540         if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2541             return -1;
2542     }
2543     proc->sz = sz;
2544     switchuvm(proc);
2545     return 0;
2546 }
2547
2548
2549

```

```

2550 // Create a new process copying p as the parent.
2551 // Sets up stack to return as if from system call.
2552 // Caller must set state of returned proc to RUNNABLE.
2553 int
2554 fork(void)
2555 {
2556     int i, pid;
2557     struct proc *np;
2558
2559     // Allocate process.
2560     if((np = allocproc()) == 0)
2561         return -1;
2562
2563     // Copy process state from p.
2564     if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2565         kfree(np->kstack);
2566         np->kstack = 0;
2567         np->state = UNUSED;
2568         return -1;
2569     }
2570     np->sz = proc->sz;
2571     np->parent = proc;
2572     *np->tf = *proc->tf;
2573
2574     // Clear %eax so that fork returns 0 in the child.
2575     np->tf->eax = 0;
2576
2577     for(i = 0; i < NOFILE; i++)
2578         if(proc->ofile[i])
2579             np->ofile[i] = filedup(proc->ofile[i]);
2580     np->cwd = idup(proc->cwd);
2581
2582     safestrcpy(np->name, proc->name, sizeof(proc->name));
2583
2584     pid = np->pid;
2585
2586     // lock to force the compiler to emit the np->state write last.
2587     acquire(&ptable.lock);
2588     np->state = RUNNABLE;
2589     release(&ptable.lock);
2590
2591     return pid;
2592 }
2593
2594
2595
2596
2597
2598
2599

```

```

2600 // Exit the current process. Does not return.
2601 // An exited process remains in the zombie state
2602 // until its parent calls wait() to find out it exited.
2603 void
2604 exit(void)
2605 {
2606     struct proc *p;
2607     int fd;
2608
2609     if(proc == initproc)
2610         panic("init exiting");
2611
2612     // Close all open files.
2613     for(fd = 0; fd < NOFILE; fd++){
2614         if(proc->ofile[fd]){
2615             fileclose(proc->ofile[fd]);
2616             proc->ofile[fd] = 0;
2617         }
2618     }
2619
2620     begin_op();
2621     iput(proc->cwd);
2622     end_op();
2623     proc->cwd = 0;
2624
2625     acquire(&ptable.lock);
2626
2627     // Parent might be sleeping in wait().
2628     wakeup1(proc->parent);
2629
2630     // Pass abandoned children to init.
2631     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2632         if(p->parent == proc){
2633             p->parent = initproc;
2634             if(p->state == ZOMBIE)
2635                 wakeup1(initproc);
2636         }
2637     }
2638
2639     // Jump into the scheduler, never to return.
2640     proc->state = ZOMBIE;
2641     sched();
2642     panic("zombie exit");
2643 }
2644
2645
2646
2647
2648
2649

```

```

2650 // Wait for a child process to exit and return its pid.
2651 // Return -1 if this process has no children.
2652 int
2653 wait(void)
2654 {
2655     struct proc *p;
2656     int havekids, pid;
2657
2658     acquire(&ptable.lock);
2659     for(;;){
2660         // Scan through table looking for zombie children.
2661         havekids = 0;
2662         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2663             if(p->parent != proc)
2664                 continue;
2665             havekids = 1;
2666             if(p->state == ZOMBIE){
2667                 // Found one.
2668                 pid = p->pid;
2669                 kfree(p->kstack);
2670                 p->kstack = 0;
2671                 freevm(p->pgdir);
2672                 p->state = UNUSED;
2673                 p->pid = 0;
2674                 p->parent = 0;
2675                 p->name[0] = 0;
2676                 p->killed = 0;
2677                 release(&ptable.lock);
2678                 return pid;
2679             }
2680         }
2681
2682         // No point waiting if we don't have any children.
2683         if(!havekids || proc->killed){
2684             release(&ptable.lock);
2685             return -1;
2686         }
2687
2688         // Wait for children to exit. (See wakeup1 call in proc_exit.)
2689         sleep(proc, &ptable.lock);
2690     }
2691 }
2692
2693
2694
2695
2696
2697
2698
2699

```

```

2700 // Per-CPU process scheduler.
2701 // Each CPU calls scheduler() after setting itself up.
2702 // Scheduler never returns. It loops, doing:
2703 // - choose a process to run
2704 // - swtch to start running that process
2705 // - eventually that process transfers control
2706 //   via swtch back to the scheduler.
2707 void
2708 scheduler(void)
2709 {
2710     struct proc *p;
2711
2712     for(;;){
2713         // Enable interrupts on this processor.
2714         sti();
2715
2716         // Loop over process table looking for process to run.
2717         acquire(&ptable.lock);
2718         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2719             if(p->state != RUNNABLE)
2720                 continue;
2721
2722             // Switch to chosen process. It is the process's job
2723             // to release ptable.lock and then reacquire it
2724             // before jumping back to us.
2725             proc = p;
2726             switchvm(p);
2727             p->state = RUNNING;
2728             swtch(&cpu->scheduler, proc->context);
2729             switchkvm();
2730
2731             // Process is done running for now.
2732             // It should have changed its p->state before coming back.
2733             proc = 0;
2734         }
2735         release(&ptable.lock);
2736     }
2737 }
2738 }
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749

```

```

2750 // Enter scheduler. Must hold only ptable.lock
2751 // and have changed proc->state.
2752 void
2753 sched(void)
2754 {
2755     int intena;
2756
2757     if(!holding(&ptable.lock))
2758         panic("sched ptable.lock");
2759     if(cpu->ncli != 1)
2760         panic("sched locks");
2761     if(proc->state == RUNNING)
2762         panic("sched running");
2763     if(readeflags() & FL_IF)
2764         panic("sched interruptible");
2765     intena = cpu->intena;
2766     swtch(&proc->context, cpu->scheduler);
2767     cpu->intena = intena;
2768 }
2769
2770 // Give up the CPU for one scheduling round.
2771 void
2772 yield(void)
2773 {
2774     acquire(&ptable.lock);
2775     proc->state = RUNNABLE;
2776     sched();
2777     release(&ptable.lock);
2778 }
2779
2780 // A fork child's very first scheduling by scheduler()
2781 // will swtch here. "Return" to user space.
2782 void
2783 forkret(void)
2784 {
2785     static int first = 1;
2786     // Still holding ptable.lock from scheduler.
2787     release(&ptable.lock);
2788
2789     if (first) {
2790         // Some initialization functions must be run in the context
2791         // of a regular process (e.g., they call sleep), and thus cannot
2792         // be run from main().
2793         first = 0;
2794         initlog();
2795     }
2796
2797     // Return to "caller", actually trapret (see allocproc).
2798 }
2799

```

```

2800 // Atomically release lock and sleep on chan.
2801 // Reacquires lock when awakened.
2802 void
2803 sleep(void *chan, struct spinlock *lk)
2804 {
2805     if(proc == 0)
2806         panic("sleep");
2807
2808     if(lk == 0)
2809         panic("sleep without lk");
2810
2811     // Must acquire ptable.lock in order to
2812     // change p->state and then call sched.
2813     // Once we hold ptable.lock, we can be
2814     // guaranteed that we won't miss any wakeup
2815     // (wakeup runs with ptable.lock locked),
2816     // so it's okay to release lk.
2817     if(lk != &ptable.lock){
2818         acquire(&ptable.lock);
2819         release(lk);
2820     }
2821
2822     // Go to sleep.
2823     proc->chan = chan;
2824     proc->state = SLEEPING;
2825     sched();
2826
2827     // Tidy up.
2828     proc->chan = 0;
2829
2830     // Reacquire original lock.
2831     if(lk != &ptable.lock){
2832         release(&ptable.lock);
2833         acquire(lk);
2834     }
2835 }
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849

```

```

2850 // Wake up all processes sleeping on chan.
2851 // The ptable lock must be held.
2852 static void
2853 wakeup1(void *chan)
2854 {
2855     struct proc *p;
2856
2857     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2858         if(p->state == SLEEPING && p->chan == chan)
2859             p->state = RUNNABLE;
2860     }
2861
2862 // Wake up all processes sleeping on chan.
2863 void
2864 wakeup(void *chan)
2865 {
2866     acquire(&ptable.lock);
2867     wakeup1(chan);
2868     release(&ptable.lock);
2869 }
2870
2871 // Kill the process with the given pid.
2872 // Process won't exit until it returns
2873 // to user space (see trap in trap.c).
2874 int
2875 kill(int pid)
2876 {
2877     struct proc *p;
2878
2879     acquire(&ptable.lock);
2880     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2881         if(p->pid == pid){
2882             p->killed = 1;
2883             // Wake process from sleep if necessary.
2884             if(p->state == SLEEPING)
2885                 p->state = RUNNABLE;
2886             release(&ptable.lock);
2887             return 0;
2888         }
2889     }
2890     release(&ptable.lock);
2891     return -1;
2892 }
2893
2894
2895
2896
2897
2898
2899

```

```

2900 // Print a process listing to console. For debugging.
2901 // Runs when user types ^P on console.
2902 // No lock to avoid wedging a stuck machine further.
2903 void
2904 procdump(void)
2905 {
2906     static char *states[] = {
2907         [UNUSED]    "unused",
2908         [EMBRYO]    "embryo",
2909         [SLEEPING]  "sleep ",
2910         [RUNNABLE]  "runble",
2911         [RUNNING]   "run   ",
2912         [ZOMBIE]    "zombie"
2913     };
2914     int i;
2915     struct proc *p;
2916     char *state;
2917     uint pc[10];
2918
2919     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2920         if(p->state == UNUSED)
2921             continue;
2922         if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
2923             state = states[p->state];
2924         else
2925             state = "???";
2926         cprintf("%d %s %s", p->pid, state, p->name);
2927         if(p->state == SLEEPING){
2928             getcallerpcs((uint*)p->context->ebp+2, pc);
2929             for(i=0; i<10 && pc[i] != 0; i++)
2930                 cprintf(" %p", pc[i]);
2931         }
2932         cprintf("\n");
2933     }
2934 }
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949

```

```

2950 # Context switch
2951 #
2952 # void swtch(struct context **old, struct context *new);
2953 #
2954 # Save current register context in old
2955 # and then load register context from new.
2956
2957 .globl swtch
2958 swtch:
2959     movl 4(%esp), %eax
2960     movl 8(%esp), %edx
2961
2962     # Save old callee-save registers
2963     pushl %ebp
2964     pushl %ebx
2965     pushl %esi
2966     pushl %edi
2967
2968     # Switch stacks
2969     movl %esp, (%eax)
2970     movl %edx, %esp
2971
2972     # Load new callee-save registers
2973     popl %edi
2974     popl %esi
2975     popl %ebx
2976     popl %ebp
2977     ret
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999

```

```

3000 // Physical memory allocator, intended to allocate
3001 // memory for user processes, kernel stacks, page table pages,
3002 // and pipe buffers. Allocates 4096-byte pages.
3003
3004 #include "types.h"
3005 #include "defs.h"
3006 #include "param.h"
3007 #include "memlayout.h"
3008 #include "mmu.h"
3009 #include "spinlock.h"
3010
3011 void freerange(void *vstart, void *vend);
3012 extern char end[]; // first address after kernel loaded from ELF file
3013
3014 struct run {
3015     struct run *next;
3016 };
3017
3018 struct {
3019     struct spinlock lock;
3020     int use_lock;
3021     struct run *freelist;
3022 } kmem;
3023
3024 // Initialization happens in two phases.
3025 // 1. main() calls kinit1() while still using entrypgdir to place just
3026 // the pages mapped by entrypgdir on free list.
3027 // 2. main() calls kinit2() with the rest of the physical pages
3028 // after installing a full page table that maps them on all cores.
3029 void
3030 kinit1(void *vstart, void *vend)
3031 {
3032     initlock(&kmem.lock, "kmem");
3033     kmem.use_lock = 0;
3034     freerange(vstart, vend);
3035 }
3036
3037 void
3038 kinit2(void *vstart, void *vend)
3039 {
3040     freerange(vstart, vend);
3041     kmem.use_lock = 1;
3042 }
3043
3044
3045
3046
3047
3048
3049

```

```

3050 void
3051 freerange(void *vstart, void *vend)
3052 {
3053     char *p;
3054     p = (char*)PGROUNDUP((uint)vstart);
3055     for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3056         kfree(p);
3057 }
3058
3059
3060 // Free the page of physical memory pointed at by v,
3061 // which normally should have been returned by a
3062 // call to kalloc(). (The exception is when
3063 // initializing the allocator; see kinit above.)
3064 void
3065 kfree(char *v)
3066 {
3067     struct run *r;
3068
3069     if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
3070         panic("kfree");
3071
3072     // Fill with junk to catch dangling refs.
3073     memset(v, 1, PGSIZE);
3074
3075     if(kmem.use_lock)
3076         acquire(&kmem.lock);
3077     r = (struct run*)v;
3078     r->next = kmem.freelist;
3079     kmem.freelist = r;
3080     if(kmem.use_lock)
3081         release(&kmem.lock);
3082 }
3083
3084 // Allocate one 4096-byte page of physical memory.
3085 // Returns a pointer that the kernel can use.
3086 // Returns 0 if the memory cannot be allocated.
3087 char*
3088 kalloc(void)
3089 {
3090     struct run *r;
3091
3092     if(kmem.use_lock)
3093         acquire(&kmem.lock);
3094     r = kmem.freelist;
3095     if(r)
3096         kmem.freelist = r->next;
3097     if(kmem.use_lock)
3098         release(&kmem.lock);
3099     return (char*)r;

```

```

3100 }
3101
3102
3103
3104
3105
3106
3107
3108
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149

```

```

3150 // x86 trap and interrupt constants.
3151
3152 // Processor-defined:
3153 #define T_DIVIDE      0      // divide error
3154 #define T_DEBUG      1      // debug exception
3155 #define T_NMI         2      // non-maskable interrupt
3156 #define T_BRKPT      3      // breakpoint
3157 #define T_OFLOW      4      // overflow
3158 #define T_BOUND      5      // bounds check
3159 #define T_ILLOP      6      // illegal opcode
3160 #define T_DEVICE      7      // device not available
3161 #define T_DBLFLT      8      // double fault
3162 // #define T_COPROC    9      // reserved (not used since 486)
3163 #define T_TSS        10     // invalid task switch segment
3164 #define T_SEGNP      11     // segment not present
3165 #define T_STACK      12     // stack exception
3166 #define T_GPFLT      13     // general protection fault
3167 #define T_PGFLT      14     // page fault
3168 // #define T_RES       15     // reserved
3169 #define T_FPERR      16     // floating point error
3170 #define T_ALIGN      17     // alignment check
3171 #define T_MCHK       18     // machine check
3172 #define T_SIMDERR    19     // SIMD floating point error
3173
3174 // These are arbitrarily chosen, but with care not to overlap
3175 // processor defined exceptions or interrupt vectors.
3176 #define T_SYSCALL     64     // system call
3177 #define T_DEFAULT     500    // catchall
3178
3179 #define T_IRQ0        32     // IRQ 0 corresponds to int T_IRQ
3180
3181 #define IRQ_TIMER      0
3182 #define IRQ_KBD        1
3183 #define IRQ_COM1       4
3184 #define IRQ_IDE        14
3185 #define IRQ_ERROR      19
3186 #define IRQ_SPURIOUS   31
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199

```

```

3200 #!/usr/bin/perl -w
3201
3202 # Generate vectors.S, the trap/interrupt entry points.
3203 # There has to be one entry point per interrupt number
3204 # since otherwise there's no way for trap() to discover
3205 # the interrupt number.
3206
3207 print "# generated by vectors.pl - do not edit\n";
3208 print "# handlers\n";
3209 print ".globl alltraps\n";
3210 for(my $i = 0; $i < 256; $i++){
3211     print ".globl vector$i\n";
3212     print "vector$i:\n";
3213     if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
3214         print "    pushl $0\n";
3215     }
3216     print "    pushl $$i\n";
3217     print "    jmp alltraps\n";
3218 }
3219
3220 print "\n# vector table\n";
3221 print ".data\n";
3222 print ".globl vectors\n";
3223 print "vectors:\n";
3224 for(my $i = 0; $i < 256; $i++){
3225     print "    .long vector$i\n";
3226 }
3227
3228 # sample output:
3229 # # handlers
3230 # .globl alltraps
3231 # .globl vector0
3232 # vector0:
3233 #     pushl $0
3234 #     pushl $0
3235 #     jmp alltraps
3236 # ...
3237 #
3238 # # vector table
3239 # .data
3240 # .globl vectors
3241 # vectors:
3242 #     .long vector0
3243 #     .long vector1
3244 #     .long vector2
3245 # ...
3246
3247
3248
3249

```

```

3250 #include "mmu.h"
3251
3252 # vectors.S sends all traps here.
3253 .globl alltraps
3254 alltraps:
3255     # Build trap frame.
3256     pushl %ds
3257     pushl %es
3258     pushl %fs
3259     pushl %gs
3260     pushal
3261
3262     # Set up data and per-cpu segments.
3263     movw $(SEG_KDATA<<3), %ax
3264     movw %ax, %ds
3265     movw %ax, %es
3266     movw $(SEG_KCPU<<3), %ax
3267     movw %ax, %fs
3268     movw %ax, %gs
3269
3270     # Call trap(tf), where tf=%esp
3271     pushl %esp
3272     call trap
3273     addl $4, %esp
3274
3275     # Return falls through to trapret...
3276 .globl trapret
3277 trapret:
3278     popal
3279     popl %gs
3280     popl %fs
3281     popl %es
3282     popl %ds
3283     addl $0x8, %esp # trapno and errcode
3284     iret
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 "memlayout.h"
3304 #include "mmu.h"
3305 #include "proc.h"
3306 #include "x86.h"
3307 #include "traps.h"
3308 #include "spinlock.h"
3309
3310 // Interrupt descriptor table (shared by all CPUs).
3311 struct gatedesc idt[256];
3312 extern uint vectors[]; // in vectors.S: array of 256 entry pointers
3313 struct spinlock tickslock;
3314 uint ticks;
3315
3316 void
3317 tvinit(void)
3318 {
3319     int i;
3320
3321     for(i = 0; i < 256; i++)
3322         SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3323     SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3324
3325     initlock(&tickslock, "time");
3326 }
3327
3328 void
3329 idtinit(void)
3330 {
3331     lidt(idt, sizeof(idt));
3332 }
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349

```

```

3350 void
3351 trap(struct trapframe *tf)
3352 {
3353     if(tf->trapno == T_SYSCALL){
3354         if(proc->killed)
3355             exit();
3356         proc->tf = tf;
3357         syscall();
3358         if(proc->killed)
3359             exit();
3360         return;
3361     }
3362
3363     switch(tf->trapno){
3364     case T_IRQ0 + IRQ_TIMER:
3365         if(cpu->id == 0){
3366             acquire(&tickslock);
3367             ticks++;
3368             wakeup(&ticks);
3369             release(&tickslock);
3370         }
3371         lapiceoi();
3372         break;
3373     case T_IRQ0 + IRQ_IDE:
3374         ideintr();
3375         lapiceoi();
3376         break;
3377     case T_IRQ0 + IRQ_IDE+1:
3378         // Bochs generates spurious IDE1 interrupts.
3379         break;
3380     case T_IRQ0 + IRQ_KBD:
3381         kbdintr();
3382         lapiceoi();
3383         break;
3384     case T_IRQ0 + IRQ_COM1:
3385         uartintr();
3386         lapiceoi();
3387         break;
3388     case T_IRQ0 + 7:
3389     case T_IRQ0 + IRQ_SPURIOUS:
3390         cprintf("cpu%d: spurious interrupt at %x:%x\n",
3391             cpu->id, tf->cs, tf->eip);
3392         lapiceoi();
3393         break;
3394
3395
3396
3397
3398
3399

```

```

3400 default:
3401     if(proc == 0 || (tf->cs&3) == 0){
3402         // In kernel, it must be our mistake.
3403         cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3404             tf->trapno, cpu->id, tf->eip, rcr2());
3405         panic("trap");
3406     }
3407     // In user space, assume process misbehaved.
3408     cprintf("pid %d %s: trap %d err %d on cpu %d "
3409         "eip 0x%x addr 0x%x--kill proc\n",
3410         proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
3411         rcr2());
3412     proc->killed = 1;
3413 }
3414
3415 // Force process exit if it has been killed and is in user space.
3416 // (If it is still executing in the kernel, let it keep running
3417 // until it gets to the regular system call return.)
3418 if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3419     exit();
3420
3421 // Force process to give up CPU on clock tick.
3422 // If interrupts were on while locks held, would need to check nlock.
3423 if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3424     yield();
3425
3426 // Check if the process has been killed since we yielded
3427 if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3428     exit();
3429 }
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449

```

```

3450 // System call numbers
3451 #define SYS_fork    1
3452 #define SYS_exit    2
3453 #define SYS_wait    3
3454 #define SYS_pipe    4
3455 #define SYS_read    5
3456 #define SYS_kill    6
3457 #define SYS_exec    7
3458 #define SYS_fstat   8
3459 #define SYS_chdir   9
3460 #define SYS_dup     10
3461 #define SYS_getpid  11
3462 #define SYS_sbrk    12
3463 #define SYS_sleep   13
3464 #define SYS_uptime  14
3465 #define SYS_open    15
3466 #define SYS_write   16
3467 #define SYS_mknod   17
3468 #define SYS_unlink  18
3469 #define SYS_link    19
3470 #define SYS_mkdir   20
3471 #define SYS_close   21
3472
3473
3474
3475
3476
3477
3478
3479
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499

```

```

3500 #include "types.h"
3501 #include "defs.h"
3502 #include "param.h"
3503 #include "memlayout.h"
3504 #include "mmu.h"
3505 #include "proc.h"
3506 #include "x86.h"
3507 #include "syscall.h"
3508
3509 // User code makes a system call with INT T_SYSCALL.
3510 // System call number in %eax.
3511 // Arguments on the stack, from the user call to the C
3512 // library system call function. The saved user %esp points
3513 // to a saved program counter, and then the first argument.
3514
3515 // Fetch the int at addr from the current process.
3516 int
3517 fetchint(uint addr, int *ip)
3518 {
3519     if(addr >= proc->sz || addr+4 > proc->sz)
3520         return -1;
3521     *ip = *(int*)(addr);
3522     return 0;
3523 }
3524
3525 // Fetch the nul-terminated string at addr from the current process.
3526 // Doesn't actually copy the string - just sets *pp to point at it.
3527 // Returns length of string, not including nul.
3528 int
3529 fetchstr(uint addr, char **pp)
3530 {
3531     char *s, *ep;
3532
3533     if(addr >= proc->sz)
3534         return -1;
3535     *pp = (char*)addr;
3536     ep = (char*)proc->sz;
3537     for(s = *pp; s < ep; s++)
3538         if(*s == 0)
3539             return s - *pp;
3540     return -1;
3541 }
3542
3543 // Fetch the nth 32-bit system call argument.
3544 int
3545 argint(int n, int *ip)
3546 {
3547     return fetchint(proc->tf->esp + 4 + 4*n, ip);
3548 }
3549

```

```

3550 // Fetch the nth word-sized system call argument as a pointer
3551 // to a block of memory of size n bytes. Check that the pointer
3552 // lies within the process address space.
3553 int
3554 argptr(int n, char **pp, int size)
3555 {
3556     int i;
3557
3558     if(argint(n, &i) < 0)
3559         return -1;
3560     if((uint)i >= proc->sz || (uint)i+size > proc->sz)
3561         return -1;
3562     *pp = (char*)i;
3563     return 0;
3564 }
3565
3566 // Fetch the nth word-sized system call argument as a string pointer.
3567 // Check that the pointer is valid and the string is nul-terminated.
3568 // (There is no shared writable memory, so the string can't change
3569 // between this check and being used by the kernel.)
3570 int
3571 argstr(int n, char **pp)
3572 {
3573     int addr;
3574     if(argint(n, &addr) < 0)
3575         return -1;
3576     return fetchstr(addr, pp);
3577 }
3578
3579 extern int sys_chdir(void);
3580 extern int sys_close(void);
3581 extern int sys_dup(void);
3582 extern int sys_exec(void);
3583 extern int sys_exit(void);
3584 extern int sys_fork(void);
3585 extern int sys_fstat(void);
3586 extern int sys_getpid(void);
3587 extern int sys_kill(void);
3588 extern int sys_link(void);
3589 extern int sys_mkdir(void);
3590 extern int sys_mknod(void);
3591 extern int sys_open(void);
3592 extern int sys_pipe(void);
3593 extern int sys_read(void);
3594 extern int sys_sbrk(void);
3595 extern int sys_sleep(void);
3596 extern int sys_unlink(void);
3597 extern int sys_wait(void);
3598 extern int sys_write(void);
3599 extern int sys_uptime(void);

```

```

3600 static int (*syscalls[])(void) = {
3601 [SYS_fork]    sys_fork,
3602 [SYS_exit]    sys_exit,
3603 [SYS_wait]    sys_wait,
3604 [SYS_pipe]    sys_pipe,
3605 [SYS_read]    sys_read,
3606 [SYS_kill]    sys_kill,
3607 [SYS_exec]    sys_exec,
3608 [SYS_fstat]   sys_fstat,
3609 [SYS_chdir]   sys_chdir,
3610 [SYS_dup]     sys_dup,
3611 [SYS_getpid]  sys_getpid,
3612 [SYS_sbrk]    sys_sbrk,
3613 [SYS_sleep]   sys_sleep,
3614 [SYS_uptime]  sys_uptime,
3615 [SYS_open]    sys_open,
3616 [SYS_write]   sys_write,
3617 [SYS_mknod]   sys_mknod,
3618 [SYS_unlink]  sys_unlink,
3619 [SYS_link]    sys_link,
3620 [SYS_mkdir]   sys_mkdir,
3621 [SYS_close]   sys_close,
3622 };
3623
3624 void
3625 syscall(void)
3626 {
3627     int num;
3628
3629     num = proc->tf->eax;
3630     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3631         proc->tf->eax = syscalls[num]();
3632     } else {
3633         cprintf("%d %s: unknown sys call %d\n",
3634             proc->pid, proc->name, num);
3635         proc->tf->eax = -1;
3636     }
3637 }
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649

```

```

3650 #include "types.h"
3651 #include "x86.h"
3652 #include "defs.h"
3653 #include "date.h"
3654 #include "param.h"
3655 #include "memlayout.h"
3656 #include "mmu.h"
3657 #include "proc.h"
3658
3659 int
3660 sys_fork(void)
3661 {
3662     return fork();
3663 }
3664
3665 int
3666 sys_exit(void)
3667 {
3668     exit();
3669     return 0; // not reached
3670 }
3671
3672 int
3673 sys_wait(void)
3674 {
3675     return wait();
3676 }
3677
3678 int
3679 sys_kill(void)
3680 {
3681     int pid;
3682
3683     if(argint(0, &pid) < 0)
3684         return -1;
3685     return kill(pid);
3686 }
3687
3688 int
3689 sys_getpid(void)
3690 {
3691     return proc->pid;
3692 }
3693
3694
3695
3696
3697
3698
3699

```

```

3700 int
3701 sys_sbrk(void)
3702 {
3703     int addr;
3704     int n;
3705
3706     if(argint(0, &n) < 0)
3707         return -1;
3708     addr = proc->sz;
3709     if(growproc(n) < 0)
3710         return -1;
3711     return addr;
3712 }
3713
3714 int
3715 sys_sleep(void)
3716 {
3717     int n;
3718     uint ticks0;
3719
3720     if(argint(0, &n) < 0)
3721         return -1;
3722     acquire(&tickslock);
3723     ticks0 = ticks;
3724     while(ticks - ticks0 < n){
3725         if(proc->killed){
3726             release(&tickslock);
3727             return -1;
3728         }
3729         sleep(&ticks, &tickslock);
3730     }
3731     release(&tickslock);
3732     return 0;
3733 }
3734
3735 // return how many clock tick interrupts have occurred
3736 // since start.
3737 int
3738 sys_uptime(void)
3739 {
3740     uint xticks;
3741
3742     acquire(&tickslock);
3743     xticks = ticks;
3744     release(&tickslock);
3745     return xticks;
3746 }
3747
3748
3749

```

```

3750 struct buf {
3751     int flags;
3752     uint dev;
3753     uint sector;
3754     struct buf *prev; // LRU cache list
3755     struct buf *next;
3756     struct buf *qnext; // disk queue
3757     uchar data[512];
3758 };
3759 #define B_BUSY 0x1 // buffer is locked by some process
3760 #define B_VALID 0x2 // buffer has been read from disk
3761 #define B_DIRTY 0x4 // buffer needs to be written to disk
3762
3763
3764
3765
3766
3767
3768
3769
3770
3771
3772
3773
3774
3775
3776
3777
3778
3779
3780
3781
3782
3783
3784
3785
3786
3787
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3799

```

```
3800 #define O_RDONLY 0x000
3801 #define O_WRONLY 0x001
3802 #define O_RDWR 0x002
3803 #define O_CREATE 0x200
3804
3805
3806
3807
3808
3809
3810
3811
3812
3813
3814
3815
3816
3817
3818
3819
3820
3821
3822
3823
3824
3825
3826
3827
3828
3829
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
```

```
3850 #define T_DIR 1 // Directory
3851 #define T_FILE 2 // File
3852 #define T_DEV 3 // Device
3853
3854 struct stat {
3855     short type; // Type of file
3856     int dev; // File system's disk device
3857     uint ino; // Inode number
3858     short nlink; // Number of links to file
3859     uint size; // Size of file in bytes
3860 };
3861
3862
3863
3864
3865
3866
3867
3868
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
```

```

3900 // On-disk file system format.
3901 // Both the kernel and user programs use this header file.
3902
3903 // Block 0 is unused.
3904 // Block 1 is super block.
3905 // Blocks 2 through sb.ninodes/IPB hold inodes.
3906 // Then free bitmap blocks holding sb.size bits.
3907 // Then sb.nblocks data blocks.
3908 // Then sb.nlog log blocks.
3909
3910 #define ROOTINO 1 // root i-number
3911 #define BSIZE 512 // block size
3912
3913 // File system super block
3914 struct superblock {
3915     uint size; // Size of file system image (blocks)
3916     uint nblocks; // Number of data blocks
3917     uint ninodes; // Number of inodes.
3918     uint nlog; // Number of log blocks
3919 };
3920
3921 #define NDIRECT 12
3922 #define NINDIRECT (BSIZE / sizeof(uint))
3923 #define MAXFILE (NDIRECT + NINDIRECT)
3924
3925 // On-disk inode structure
3926 struct dinode {
3927     short type; // File type
3928     short major; // Major device number (T_DEV only)
3929     short minor; // Minor device number (T_DEV only)
3930     short nlink; // Number of links to inode in file system
3931     uint size; // Size of file (bytes)
3932     uint addrs[NDIRECT+1]; // Data block addresses
3933 };
3934
3935 // Inodes per block.
3936 #define IPB (BSIZE / sizeof(struct dinode))
3937
3938 // Block containing inode i
3939 #define IBLOCK(i) ((i) / IPB + 2)
3940
3941 // Bitmap bits per block
3942 #define BPB (BSIZE*8)
3943
3944 // Block containing bit for block b
3945 #define BBLOCK(b, ninodes) (b/BPB + (ninodes)/IPB + 3)
3946
3947 // Directory is a file containing a sequence of dirent structures.
3948 #define DIRSIZ 14
3949

```

```

3950 struct dirent {
3951     ushort inum;
3952     char name[DIRSIZ];
3953 };
3954
3955
3956
3957
3958
3959
3960
3961
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999

```

```
4000 struct file {
4001     enum { FD_NONE, FD_PIPE, FD_INODE } type;
4002     int ref; // reference count
4003     char readable;
4004     char writable;
4005     struct pipe *pipe;
4006     struct inode *ip;
4007     uint off;
4008 };
4009
4010
4011 // in-memory copy of an inode
4012 struct inode {
4013     uint dev;           // Device number
4014     uint inum;          // Inode number
4015     int ref;            // Reference count
4016     int flags;          // I_BUSY, I_INVALID
4017
4018     short type;         // copy of disk inode
4019     short major;
4020     short minor;
4021     short nlink;
4022     uint size;
4023     uint addrs[NDIRECT+1];
4024 };
4025 #define I_BUSY 0x1
4026 #define I_INVALID 0x2
4027
4028 // table mapping major device number to
4029 // device functions
4030 struct devsw {
4031     int (*read)(struct inode*, char*, int);
4032     int (*write)(struct inode*, char*, int);
4033 };
4034
4035 extern struct devsw devsw[];
4036
4037 #define CONSOLE 1
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
```

```
4050 // Blank page.
4051
4052
4053
4054
4055
4056
4057
4058
4059
4060
4061
4062
4063
4064
4065
4066
4067
4068
4069
4070
4071
4072
4073
4074
4075
4076
4077
4078
4079
4080
4081
4082
4083
4084
4085
4086
4087
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
```



```

4100 // Simple PIO-based (non-DMA) IDE driver code.
4101
4102 #include "types.h"
4103 #include "defs.h"
4104 #include "param.h"
4105 #include "memlayout.h"
4106 #include "mmu.h"
4107 #include "proc.h"
4108 #include "x86.h"
4109 #include "traps.h"
4110 #include "spinlock.h"
4111 #include "buf.h"
4112
4113 #define IDE_BSY      0x80
4114 #define IDE_DRDY     0x40
4115 #define IDE_DF       0x20
4116 #define IDE_ERR      0x01
4117
4118 #define IDE_CMD_READ  0x20
4119 #define IDE_CMD_WRITE 0x30
4120
4121 // idequeue points to the buf now being read/written to the disk.
4122 // idequeue->qnext points to the next buf to be processed.
4123 // You must hold idelock while manipulating queue.
4124
4125 static struct spinlock idelock;
4126 static struct buf *idequeue;
4127
4128 static int havdisk1;
4129 static void idestart(struct buf*);
4130
4131 // Wait for IDE disk to become ready.
4132 static int
4133 idewait(int checkerr)
4134 {
4135     int r;
4136
4137     while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4138         ;
4139     if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4140         return -1;
4141     return 0;
4142 }
4143
4144
4145
4146
4147
4148
4149

```

```

4150 void
4151 ideinit(void)
4152 {
4153     int i;
4154
4155     initlock(&idelock, "ide");
4156     picenable(IRQ_IDE);
4157     ioapicenable(IRQ_IDE, ncpu - 1);
4158     idewait(0);
4159
4160     // Check if disk 1 is present
4161     outb(0x1f6, 0xe0 | (1<<4));
4162     for(i=0; i<1000; i++){
4163         if(inb(0x1f7) != 0){
4164             havdisk1 = 1;
4165             break;
4166         }
4167     }
4168
4169     // Switch back to disk 0.
4170     outb(0x1f6, 0xe0 | (0<<4));
4171 }
4172
4173 // Start the request for b. Caller must hold idelock.
4174 static void
4175 idestart(struct buf *b)
4176 {
4177     if(b == 0)
4178         panic("idestart");
4179
4180     idewait(0);
4181     outb(0x3f6, 0); // generate interrupt
4182     outb(0x1f2, 1); // number of sectors
4183     outb(0x1f3, b->sector & 0xff);
4184     outb(0x1f4, (b->sector >> 8) & 0xff);
4185     outb(0x1f5, (b->sector >> 16) & 0xff);
4186     outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
4187     if(b->flags & B_DIRTY){
4188         outb(0x1f7, IDE_CMD_WRITE);
4189         outsl(0x1f0, b->data, 512/4);
4190     } else {
4191         outb(0x1f7, IDE_CMD_READ);
4192     }
4193 }
4194
4195
4196
4197
4198
4199

```

```

4200 // Interrupt handler.
4201 void
4202 ideintr(void)
4203 {
4204     struct buf *b;
4205
4206     // First queued buffer is the active request.
4207     acquire(&idelock);
4208     if((b = idequeue) == 0){
4209         release(&idelock);
4210         // cprintf("spurious IDE interrupt\n");
4211         return;
4212     }
4213     idequeue = b->qnext;
4214
4215     // Read data if needed.
4216     if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4217         insl(0x1f0, b->data, 512/4);
4218
4219     // Wake process waiting for this buf.
4220     b->flags |= B_VALID;
4221     b->flags &= ~B_DIRTY;
4222     wakeup(b);
4223
4224     // Start disk on next buf in queue.
4225     if(idequeue != 0)
4226         idestart(idequeue);
4227
4228     release(&idelock);
4229 }
4230
4231
4232
4233
4234
4235
4236
4237
4238
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249

```

```

4250 // Sync buf with disk.
4251 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
4252 // Else if B_VALID is not set, read buf from disk, set B_VALID.
4253 void
4254 iderw(struct buf *b)
4255 {
4256     struct buf **pp;
4257
4258     if(!(b->flags & B_BUSY))
4259         panic("iderw: buf not busy");
4260     if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
4261         panic("iderw: nothing to do");
4262     if(b->dev != 0 && !havedisk1)
4263         panic("iderw: ide disk 1 not present");
4264
4265     acquire(&idelock);
4266
4267     // Append b to idequeue.
4268     b->qnext = 0;
4269     for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
4270         ;
4271     *pp = b;
4272
4273     // Start disk if necessary.
4274     if(idequeue == b)
4275         idestart(b);
4276
4277     // Wait for request to finish.
4278     while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
4279         sleep(b, &idelock);
4280     }
4281
4282     release(&idelock);
4283 }
4284
4285
4286
4287
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299

```

```

4300 // Buffer cache.
4301 //
4302 // The buffer cache is a linked list of buf structures holding
4303 // cached copies of disk block contents. Caching disk blocks
4304 // in memory reduces the number of disk reads and also provides
4305 // a synchronization point for disk blocks used by multiple processes.
4306 //
4307 // Interface:
4308 // * To get a buffer for a particular disk block, call bread.
4309 // * After changing buffer data, call bwrite to write it to disk.
4310 // * When done with the buffer, call brelse.
4311 // * Do not use the buffer after calling brelse.
4312 // * Only one process at a time can use a buffer,
4313 //   so do not keep them longer than necessary.
4314 //
4315 // The implementation uses three state flags internally:
4316 // * B_BUSY: the block has been returned from bread
4317 //   and has not been passed back to brelse.
4318 // * B_VALID: the buffer data has been read from the disk.
4319 // * B_DIRTY: the buffer data has been modified
4320 //   and needs to be written to disk.
4321
4322 #include "types.h"
4323 #include "defs.h"
4324 #include "param.h"
4325 #include "spinlock.h"
4326 #include "buf.h"
4327
4328 struct {
4329   struct spinlock lock;
4330   struct buf buf[NBUF];
4331
4332   // Linked list of all buffers, through prev/next.
4333   // head.next is most recently used.
4334   struct buf head;
4335 } bcache;
4336
4337 void
4338 binit(void)
4339 {
4340   struct buf *b;
4341
4342   initlock(&bcache.lock, "bcache");
4343
4344
4345
4346
4347
4348
4349

```

```

4350 // Create linked list of buffers
4351 bcache.head.prev = &bcache.head;
4352 bcache.head.next = &bcache.head;
4353 for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4354   b->next = bcache.head.next;
4355   b->prev = &bcache.head;
4356   b->dev = -1;
4357   bcache.head.next->prev = b;
4358   bcache.head.next = b;
4359 }
4360 }
4361
4362 // Look through buffer cache for sector on device dev.
4363 // If not found, allocate a buffer.
4364 // In either case, return B_BUSY buffer.
4365 static struct buf*
4366 bget(uint dev, uint sector)
4367 {
4368   struct buf *b;
4369
4370   acquire(&bcache.lock);
4371
4372   loop:
4373   // Is the sector already cached?
4374   for(b = bcache.head.next; b != &bcache.head; b = b->next){
4375     if(b->dev == dev && b->sector == sector){
4376       if(!(b->flags & B_BUSY)){
4377         b->flags |= B_BUSY;
4378         release(&bcache.lock);
4379         return b;
4380       }
4381       sleep(b, &bcache.lock);
4382       goto loop;
4383     }
4384   }
4385
4386   // Not cached; recycle some non-busy and clean buffer.
4387   // "clean" because B_DIRTY and !B_BUSY means log.c
4388   // hasn't yet committed the changes to the buffer.
4389   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4390     if((b->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
4391       b->dev = dev;
4392       b->sector = sector;
4393       b->flags = B_BUSY;
4394       release(&bcache.lock);
4395       return b;
4396     }
4397   }
4398   panic("bget: no buffers");
4399 }

```

```

4400 // Return a B_BUSY buf with the contents of the indicated disk sector.
4401 struct buf*
4402 bread(uint dev, uint sector)
4403 {
4404     struct buf *b;
4405
4406     b = bget(dev, sector);
4407     if(!(b->flags & B_VALID))
4408         iderw(b);
4409     return b;
4410 }
4411
4412 // Write b's contents to disk. Must be B_BUSY.
4413 void
4414 bwrite(struct buf *b)
4415 {
4416     if((b->flags & B_BUSY) == 0)
4417         panic("bwrite");
4418     b->flags |= B_DIRTY;
4419     iderw(b);
4420 }
4421
4422 // Release a B_BUSY buffer.
4423 // Move to the head of the MRU list.
4424 void
4425 brelse(struct buf *b)
4426 {
4427     if((b->flags & B_BUSY) == 0)
4428         panic("brelse");
4429
4430     acquire(&bcache.lock);
4431
4432     b->next->prev = b->prev;
4433     b->prev->next = b->next;
4434     b->next = bcache.head.next;
4435     b->prev = &bcache.head;
4436     bcache.head.next->prev = b;
4437     bcache.head.next = b;
4438
4439     b->flags &= ~B_BUSY;
4440     wakeup(b);
4441
4442     release(&bcache.lock);
4443 }
4444
4445
4446
4447
4448
4449

```

```

4450 // Blank page.
4451
4452
4453
4454
4455
4456
4457
4458
4459
4460
4461
4462
4463
4464
4465
4466
4467
4468
4469
4470
4471
4472
4473
4474
4475
4476
4477
4478
4479
4480
4481
4482
4483
4484
4485
4486
4487
4488
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499

```

```

4500 #include "types.h"
4501 #include "defs.h"
4502 #include "param.h"
4503 #include "spinlock.h"
4504 #include "fs.h"
4505 #include "buf.h"
4506
4507 // Simple logging that allows concurrent FS system calls.
4508 //
4509 // A log transaction contains the updates of multiple FS system
4510 // calls. The logging system only commits when there are
4511 // no FS system calls active. Thus there is never
4512 // any reasoning required about whether a commit might
4513 // write an uncommitted system call's updates to disk.
4514 //
4515 // A system call should call begin_op()/end_op() to mark
4516 // its start and end. Usually begin_op() just increments
4517 // the count of in-progress FS system calls and returns.
4518 // But if it thinks the log is close to running out, it
4519 // sleeps until the last outstanding end_op() commits.
4520 //
4521 // The log is a physical re-do log containing disk blocks.
4522 // The on-disk log format:
4523 //   header block, containing sector #s for block A, B, C, ...
4524 //   block A
4525 //   block B
4526 //   block C
4527 //   ...
4528 // Log appends are synchronous.
4529
4530 // Contents of the header block, used for both the on-disk header block
4531 // and to keep track in memory of logged sector #s before commit.
4532 struct logheader {
4533   int n;
4534   int sector[LOGSIZE];
4535 };
4536
4537 struct log {
4538   struct spinlock lock;
4539   int start;
4540   int size;
4541   int outstanding; // how many FS sys calls are executing.
4542   int committing; // in commit(), please wait.
4543   int dev;
4544   struct logheader lh;
4545 };
4546
4547
4548
4549

```

```

4550 struct log log;
4551
4552 static void recover_from_log(void);
4553 static void commit();
4554
4555 void
4556 initlog(void)
4557 {
4558   if (sizeof(struct logheader) >= BSIZE)
4559     panic("initlog: too big logheader");
4560
4561   struct superblock sb;
4562   initlock(&log.lock, "log");
4563   readsb(ROOTDEV, &sb);
4564   log.start = sb.size - sb.nlog;
4565   log.size = sb.nlog;
4566   log.dev = ROOTDEV;
4567   recover_from_log();
4568 }
4569
4570 // Copy committed blocks from log to their home location
4571 static void
4572 install_trans(void)
4573 {
4574   int tail;
4575
4576   for (tail = 0; tail < log.lh.n; tail++) {
4577     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
4578     struct buf *dbuf = bread(log.dev, log.lh.sector[tail]); // read dst
4579     memmove(dbuf->data, lbuf->data, BSIZE); // copy block to dst
4580     bwrite(dbuf); // write dst to disk
4581     brelse(lbuf);
4582     brelse(dbuf);
4583   }
4584 }
4585
4586 // Read the log header from disk into the in-memory log header
4587 static void
4588 read_head(void)
4589 {
4590   struct buf *buf = bread(log.dev, log.start);
4591   struct logheader *lh = (struct logheader *) (buf->data);
4592   int i;
4593   log.lh.n = lh->n;
4594   for (i = 0; i < log.lh.n; i++) {
4595     log.lh.sector[i] = lh->sector[i];
4596   }
4597   brelse(buf);
4598 }
4599

```

```

4600 // Write in-memory log header to disk.
4601 // This is the true point at which the
4602 // current transaction commits.
4603 static void
4604 write_head(void)
4605 {
4606     struct buf *buf = bread(log.dev, log.start);
4607     struct logheader *hb = (struct logheader *) (buf->data);
4608     int i;
4609     hb->n = log.lh.n;
4610     for (i = 0; i < log.lh.n; i++) {
4611         hb->sector[i] = log.lh.sector[i];
4612     }
4613     bwrite(buf);
4614     brelse(buf);
4615 }
4616
4617 static void
4618 recover_from_log(void)
4619 {
4620     read_head();
4621     install_trans(); // if committed, copy from log to disk
4622     log.lh.n = 0;
4623     write_head(); // clear the log
4624 }
4625
4626 // called at the start of each FS system call.
4627 void
4628 begin_op(void)
4629 {
4630     acquire(&log.lock);
4631     while(1){
4632         if(log.committing){
4633             sleep(&log, &log.lock);
4634         } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
4635             // this op might exhaust log space; wait for commit.
4636             sleep(&log, &log.lock);
4637         } else {
4638             log.outstanding += 1;
4639             release(&log.lock);
4640             break;
4641         }
4642     }
4643 }
4644
4645
4646
4647
4648
4649

```

```

4650 // called at the end of each FS system call.
4651 // commits if this was the last outstanding operation.
4652 void
4653 end_op(void)
4654 {
4655     int do_commit = 0;
4656
4657     acquire(&log.lock);
4658     log.outstanding -= 1;
4659     if(log.committing)
4660         panic("log.committing");
4661     if(log.outstanding == 0){
4662         do_commit = 1;
4663         log.committing = 1;
4664     } else {
4665         // begin_op() may be waiting for log space.
4666         wakeup(&log);
4667     }
4668     release(&log.lock);
4669
4670     if(do_commit){
4671         // call commit w/o holding locks, since not allowed
4672         // to sleep with locks.
4673         commit();
4674         acquire(&log.lock);
4675         log.committing = 0;
4676         wakeup(&log);
4677         release(&log.lock);
4678     }
4679 }
4680
4681 // Copy modified blocks from cache to log.
4682 static void
4683 write_log(void)
4684 {
4685     int tail;
4686
4687     for (tail = 0; tail < log.lh.n; tail++) {
4688         struct buf *to = bread(log.dev, log.start+tail+1); // log block
4689         struct buf *from = bread(log.dev, log.lh.sector[tail]); // cache block
4690         memmove(to->data, from->data, BSIZE);
4691         bwrite(to); // write the log
4692         brelse(from);
4693         brelse(to);
4694     }
4695 }
4696
4697
4698
4699

```

```

4700 static void
4701 commit()
4702 {
4703     if (log.lh.n > 0) {
4704         write_log(); // Write modified blocks from cache to log
4705         write_head(); // Write header to disk -- the real commit
4706         install_trans(); // Now install writes to home locations
4707         log.lh.n = 0;
4708         write_head(); // Erase the transaction from the log
4709     }
4710 }
4711
4712 // Caller has modified b->data and is done with the buffer.
4713 // Record the block number and pin in the cache with B_DIRTY.
4714 // commit()/write_log() will do the disk write.
4715 //
4716 // log_write() replaces bwrite(); a typical use is:
4717 //   bp = bread(...)
4718 //   modify bp->data[]
4719 //   log_write(bp)
4720 //   brelse(bp)
4721 void
4722 log_write(struct buf *b)
4723 {
4724     int i;
4725
4726     if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
4727         panic("too big a transaction");
4728     if (log.outstanding < 1)
4729         panic("log_write outside of trans");
4730
4731     acquire(&log.lock);
4732     for (i = 0; i < log.lh.n; i++) {
4733         if (log.lh.sector[i] == b->sector) // log absorbtion
4734             break;
4735     }
4736     log.lh.sector[i] = b->sector;
4737     if (i == log.lh.n)
4738         log.lh.n++;
4739     b->flags |= B_DIRTY; // prevent eviction
4740     release(&log.lock);
4741 }
4742
4743
4744
4745
4746
4747
4748
4749

```

```

4750 // File system implementation. Five layers:
4751 //   + Blocks: allocator for raw disk blocks.
4752 //   + Log: crash recovery for multi-step updates.
4753 //   + Files: inode allocator, reading, writing, metadata.
4754 //   + Directories: inode with special contents (list of other inodes!)
4755 //   + Names: paths like /usr/rtn/xv6/fs.c for convenient naming.
4756 //
4757 // This file contains the low-level file system manipulation
4758 // routines. The (higher-level) system call implementations
4759 // are in sysfile.c.
4760
4761 #include "types.h"
4762 #include "defs.h"
4763 #include "param.h"
4764 #include "stat.h"
4765 #include "mmu.h"
4766 #include "proc.h"
4767 #include "spinlock.h"
4768 #include "buf.h"
4769 #include "fs.h"
4770 #include "file.h"
4771
4772 #define min(a, b) ((a) < (b) ? (a) : (b))
4773 static void itrunc(struct inode*);
4774
4775 // Read the super block.
4776 void
4777 readsb(int dev, struct superblock *sb)
4778 {
4779     struct buf *bp;
4780
4781     bp = bread(dev, 1);
4782     memmove(sb, bp->data, sizeof(*sb));
4783     brelse(bp);
4784 }
4785
4786 // Zero a block.
4787 static void
4788 bzero(int dev, int bno)
4789 {
4790     struct buf *bp;
4791
4792     bp = bread(dev, bno);
4793     memset(bp->data, 0, BSIZE);
4794     log_write(bp);
4795     brelse(bp);
4796 }
4797
4798
4799

```

```

4800 // Blocks.
4801
4802 // Allocate a zeroed disk block.
4803 static uint
4804 balloc(uint dev)
4805 {
4806     int b, bi, m;
4807     struct buf *bp;
4808     struct superblock sb;
4809
4810     bp = 0;
4811     readsb(dev, &sb);
4812     for(b = 0; b < sb.size; b += BPB){
4813         bp = bread(dev, BBLOCK(b, sb.ninodes));
4814         for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
4815             m = 1 << (bi % 8);
4816             if((bp->data[bi/8] & m) == 0){ // Is block free?
4817                 bp->data[bi/8] |= m; // Mark block in use.
4818                 log_write(bp);
4819                 brelse(bp);
4820                 bzero(dev, b + bi);
4821                 return b + bi;
4822             }
4823         }
4824         brelse(bp);
4825     }
4826     panic("balloc: out of blocks");
4827 }
4828
4829 // Free a disk block.
4830 static void
4831 bfree(int dev, uint b)
4832 {
4833     struct buf *bp;
4834     struct superblock sb;
4835     int bi, m;
4836
4837     readsb(dev, &sb);
4838     bp = bread(dev, BBLOCK(b, sb.ninodes));
4839     bi = b % BPB;
4840     m = 1 << (bi % 8);
4841     if((bp->data[bi/8] & m) == 0)
4842         panic("freeing free block");
4843     bp->data[bi/8] &= ~m;
4844     log_write(bp);
4845     brelse(bp);
4846 }
4847
4848
4849

```

```

4850 // Inodes.
4851 //
4852 // An inode describes a single unnamed file.
4853 // The inode disk structure holds metadata: the file's type,
4854 // its size, the number of links referring to it, and the
4855 // list of blocks holding the file's content.
4856 //
4857 // The inodes are laid out sequentially on disk immediately after
4858 // the superblock. Each inode has a number, indicating its
4859 // position on the disk.
4860 //
4861 // The kernel keeps a cache of in-use inodes in memory
4862 // to provide a place for synchronizing access
4863 // to inodes used by multiple processes. The cached
4864 // inodes include book-keeping information that is
4865 // not stored on disk: ip->ref and ip->flags.
4866 //
4867 // An inode and its in-memory representative go through a
4868 // sequence of states before they can be used by the
4869 // rest of the file system code.
4870 //
4871 // * Allocation: an inode is allocated if its type (on disk)
4872 //   is non-zero. ialloc() allocates, iput() frees if
4873 //   the link count has fallen to zero.
4874 //
4875 // * Referencing in cache: an entry in the inode cache
4876 //   is free if ip->ref is zero. Otherwise ip->ref tracks
4877 //   the number of in-memory pointers to the entry (open
4878 //   files and current directories). iget() to find or
4879 //   create a cache entry and increment its ref, iput()
4880 //   to decrement ref.
4881 //
4882 // * Valid: the information (type, size, &c) in an inode
4883 //   cache entry is only correct when the I_VALID bit
4884 //   is set in ip->flags. ilock() reads the inode from
4885 //   the disk and sets I_VALID, while iput() clears
4886 //   I_VALID if ip->ref has fallen to zero.
4887 //
4888 // * Locked: file system code may only examine and modify
4889 //   the information in an inode and its content if it
4890 //   has first locked the inode. The I_BUSY flag indicates
4891 //   that the inode is locked. ilock() sets I_BUSY,
4892 //   while iunlock clears it.
4893 //
4894 // Thus a typical sequence is:
4895 //   ip = iget(dev, inum)
4896 //   ilock(ip)
4897 //   ... examine and modify ip->xxx ...
4898 //   iunlock(ip)
4899 //   iput(ip)

```



```

4900 //
4901 // ilock() is separate from iget() so that system calls can
4902 // get a long-term reference to an inode (as for an open file)
4903 // and only lock it for short periods (e.g., in read()).
4904 // The separation also helps avoid deadlock and races during
4905 // pathname lookup. iget() increments ip->ref so that the inode
4906 // stays cached and pointers to it remain valid.
4907 //
4908 // Many internal file system functions expect the caller to
4909 // have locked the inodes involved; this lets callers create
4910 // multi-step atomic operations.
4911
4912 struct {
4913   struct spinlock lock;
4914   struct inode inode[NINODE];
4915 } icache;
4916
4917 void
4918 iinit(void)
4919 {
4920   initlock(&icache.lock, "icache");
4921 }
4922
4923 static struct inode* iget(uint dev, uint inum);
4924
4925
4926
4927
4928
4929
4930
4931
4932
4933
4934
4935
4936
4937
4938
4939
4940
4941
4942
4943
4944
4945
4946
4947
4948
4949

```

```

4950 // Allocate a new inode with the given type on device dev.
4951 // A free inode has a type of zero.
4952 struct inode*
4953 ialloc(uint dev, short type)
4954 {
4955   int inum;
4956   struct buf *bp;
4957   struct dinode *dip;
4958   struct superblock sb;
4959
4960   readsb(dev, &sb);
4961
4962   for(inum = 1; inum < sb.ninodes; inum++){
4963     bp = bread(dev, IBLOCK(inum));
4964     dip = (struct dinode*)bp->data + inum%IPB;
4965     if(dip->type == 0){ // a free inode
4966       memset(dip, 0, sizeof(*dip));
4967       dip->type = type;
4968       log_write(bp); // mark it allocated on the disk
4969       brelse(bp);
4970       return iget(dev, inum);
4971     }
4972     brelse(bp);
4973   }
4974   panic("ialloc: no inodes");
4975 }
4976
4977 // Copy a modified in-memory inode to disk.
4978 void
4979 iupdate(struct inode *ip)
4980 {
4981   struct buf *bp;
4982   struct dinode *dip;
4983
4984   bp = bread(ip->dev, IBLOCK(ip->inum));
4985   dip = (struct dinode*)bp->data + ip->inum%IPB;
4986   dip->type = ip->type;
4987   dip->major = ip->major;
4988   dip->minor = ip->minor;
4989   dip->nlink = ip->nlink;
4990   dip->size = ip->size;
4991   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
4992   log_write(bp);
4993   brelse(bp);
4994 }
4995
4996
4997
4998
4999

```

```

5000 // Find the inode with number inum on device dev
5001 // and return the in-memory copy. Does not lock
5002 // the inode and does not read it from disk.
5003 static struct inode*
5004 iget(uint dev, uint inum)
5005 {
5006     struct inode *ip, *empty;
5007
5008     acquire(&icache.lock);
5009
5010     // Is the inode already cached?
5011     empty = 0;
5012     for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5013         if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5014             ip->ref++;
5015             release(&icache.lock);
5016             return ip;
5017         }
5018         if(empty == 0 && ip->ref == 0)    // Remember empty slot.
5019             empty = ip;
5020     }
5021
5022     // Recycle an inode cache entry.
5023     if(empty == 0)
5024         panic("iget: no inodes");
5025
5026     ip = empty;
5027     ip->dev = dev;
5028     ip->inum = inum;
5029     ip->ref = 1;
5030     ip->flags = 0;
5031     release(&icache.lock);
5032
5033     return ip;
5034 }
5035
5036 // Increment reference count for ip.
5037 // Returns ip to enable ip = idup(ip1) idiom.
5038 struct inode*
5039 idup(struct inode *ip)
5040 {
5041     acquire(&icache.lock);
5042     ip->ref++;
5043     release(&icache.lock);
5044     return ip;
5045 }
5046
5047
5048
5049

```

```

5050 // Lock the given inode.
5051 // Reads the inode from disk if necessary.
5052 void
5053 ilock(struct inode *ip)
5054 {
5055     struct buf *bp;
5056     struct dinode *dip;
5057
5058     if(ip == 0 || ip->ref < 1)
5059         panic("ilock");
5060
5061     acquire(&icache.lock);
5062     while(ip->flags & I_BUSY)
5063         sleep(ip, &icache.lock);
5064     ip->flags |= I_BUSY;
5065     release(&icache.lock);
5066
5067     if(!(ip->flags & I_VALID)){
5068         bp = bread(ip->dev, IBLOCK(ip->inum));
5069         dip = (struct dinode*)bp->data + ip->inum%IPB;
5070         ip->type = dip->type;
5071         ip->major = dip->major;
5072         ip->minor = dip->minor;
5073         ip->nlink = dip->nlink;
5074         ip->size = dip->size;
5075         memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5076         brelse(bp);
5077         ip->flags |= I_VALID;
5078         if(ip->type == 0)
5079             panic("ilock: no type");
5080     }
5081 }
5082
5083 // Unlock the given inode.
5084 void
5085 iunlock(struct inode *ip)
5086 {
5087     if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
5088         panic("iunlock");
5089
5090     acquire(&icache.lock);
5091     ip->flags &= ~I_BUSY;
5092     wakeup(ip);
5093     release(&icache.lock);
5094 }
5095
5096
5097
5098
5099

```

```

5100 // Drop a reference to an in-memory inode.
5101 // If that was the last reference, the inode cache entry can
5102 // be recycled.
5103 // If that was the last reference and the inode has no links
5104 // to it, free the inode (and its content) on disk.
5105 // All calls to iput() must be inside a transaction in
5106 // case it has to free the inode.
5107 void
5108 iput(struct inode *ip)
5109 {
5110     acquire(&icache.lock);
5111     if(ip->ref == 1 && (ip->flags & I_INVALID) && ip->nlink == 0){
5112         // inode has no links and no other references: truncate and free.
5113         if(ip->flags & I_BUSY)
5114             panic("iput busy");
5115         ip->flags |= I_BUSY;
5116         release(&icache.lock);
5117         itrunc(ip);
5118         ip->type = 0;
5119         iupdate(ip);
5120         acquire(&icache.lock);
5121         ip->flags = 0;
5122         wakeup(ip);
5123     }
5124     ip->ref--;
5125     release(&icache.lock);
5126 }
5127
5128 // Common idiom: unlock, then put.
5129 void
5130 iunlockput(struct inode *ip)
5131 {
5132     iunlock(ip);
5133     iput(ip);
5134 }
5135
5136
5137
5138
5139
5140
5141
5142
5143
5144
5145
5146
5147
5148
5149

```

```

5150 // Inode content
5151 //
5152 // The content (data) associated with each inode is stored
5153 // in blocks on the disk. The first NDIRECT block numbers
5154 // are listed in ip->addrs[]. The next NINDIRECT blocks are
5155 // listed in block ip->addrs[NDIRECT].
5156
5157 // Return the disk block address of the nth block in inode ip.
5158 // If there is no such block, bmap allocates one.
5159 static uint
5160 bmap(struct inode *ip, uint bn)
5161 {
5162     uint addr, *a;
5163     struct buf *bp;
5164
5165     if(bn < NDIRECT){
5166         if((addr = ip->addrs[bn]) == 0)
5167             ip->addrs[bn] = addr = balloc(ip->dev);
5168         return addr;
5169     }
5170     bn -= NDIRECT;
5171
5172     if(bn < NINDIRECT){
5173         // Load indirect block, allocating if necessary.
5174         if((addr = ip->addrs[NDIRECT]) == 0)
5175             ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5176         bp = bread(ip->dev, addr);
5177         a = (uint*)bp->data;
5178         if((addr = a[bn]) == 0){
5179             a[bn] = addr = balloc(ip->dev);
5180             log_write(bp);
5181         }
5182         brelse(bp);
5183         return addr;
5184     }
5185
5186     panic("bmap: out of range");
5187 }
5188
5189
5190
5191
5192
5193
5194
5195
5196
5197
5198
5199

```

```

5200 // Truncate inode (discard contents).
5201 // Only called when the inode has no links
5202 // to it (no directory entries referring to it)
5203 // and has no in-memory reference to it (is
5204 // not an open file or current directory).
5205 static void
5206 itrunc(struct inode *ip)
5207 {
5208     int i, j;
5209     struct buf *bp;
5210     uint *a;
5211
5212     for(i = 0; i < NDIRECT; i++){
5213         if(ip->addrs[i]){
5214             bfree(ip->dev, ip->addrs[i]);
5215             ip->addrs[i] = 0;
5216         }
5217     }
5218
5219     if(ip->addrs[NDIRECT]){
5220         bp = bread(ip->dev, ip->addrs[NDIRECT]);
5221         a = (uint*)bp->data;
5222         for(j = 0; j < NINDIRECT; j++){
5223             if(a[j])
5224                 bfree(ip->dev, a[j]);
5225         }
5226         brelse(bp);
5227         bfree(ip->dev, ip->addrs[NDIRECT]);
5228         ip->addrs[NDIRECT] = 0;
5229     }
5230
5231     ip->size = 0;
5232     iupdate(ip);
5233 }
5234
5235 // Copy stat information from inode.
5236 void
5237 stati(struct inode *ip, struct stat *st)
5238 {
5239     st->dev = ip->dev;
5240     st->ino = ip->inum;
5241     st->type = ip->type;
5242     st->nlink = ip->nlink;
5243     st->size = ip->size;
5244 }
5245
5246
5247
5248
5249

```

```

5250 // Read data from inode.
5251 int
5252 readi(struct inode *ip, char *dst, uint off, uint n)
5253 {
5254     uint tot, m;
5255     struct buf *bp;
5256
5257     if(ip->type == T_DEV){
5258         if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
5259             return -1;
5260         return devsw[ip->major].read(ip, dst, n);
5261     }
5262
5263     if(off > ip->size || off + n < off)
5264         return -1;
5265     if(off + n > ip->size)
5266         n = ip->size - off;
5267
5268     for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5269         bp = bread(ip->dev, bmap(ip, off/BSIZE));
5270         m = min(n - tot, BSIZE - off%BSIZE);
5271         memmove(dst, bp->data + off%BSIZE, m);
5272         brelse(bp);
5273     }
5274     return n;
5275 }
5276
5277
5278
5279
5280
5281
5282
5283
5284
5285
5286
5287
5288
5289
5290
5291
5292
5293
5294
5295
5296
5297
5298
5299

```

```

5300 // Write data to inode.
5301 int
5302 writei(struct inode *ip, char *src, uint off, uint n)
5303 {
5304     uint tot, m;
5305     struct buf *bp;
5306
5307     if(ip->type == T_DEV){
5308         if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
5309             return -1;
5310         return devsw[ip->major].write(ip, src, n);
5311     }
5312
5313     if(off > ip->size || off + n < off)
5314         return -1;
5315     if(off + n > MAXFILE*BSIZE)
5316         return -1;
5317
5318     for(tot=0; tot<n; tot+=m, off+=m, src+=m){
5319         bp = bread(ip->dev, bmap(ip, off/BSIZE));
5320         m = min(n - tot, BSIZE - off%BSIZE);
5321         memmove(bp->data + off%BSIZE, src, m);
5322         log_write(bp);
5323         brelse(bp);
5324     }
5325
5326     if(n > 0 && off > ip->size){
5327         ip->size = off;
5328         iupdate(ip);
5329     }
5330     return n;
5331 }
5332
5333
5334
5335
5336
5337
5338
5339
5340
5341
5342
5343
5344
5345
5346
5347
5348
5349

```

```

5350 // Directories
5351
5352 int
5353 namecmp(const char *s, const char *t)
5354 {
5355     return strncmp(s, t, DIRSIZ);
5356 }
5357
5358 // Look for a directory entry in a directory.
5359 // If found, set *poff to byte offset of entry.
5360 struct inode*
5361 dirlookup(struct inode *dp, char *name, uint *poff)
5362 {
5363     uint off, inum;
5364     struct dirent de;
5365
5366     if(dp->type != T_DIR)
5367         panic("dirlookup not DIR");
5368
5369     for(off = 0; off < dp->size; off += sizeof(de)){
5370         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5371             panic("dirlink read");
5372         if(de.inum == 0)
5373             continue;
5374         if(namecmp(name, de.name) == 0){
5375             // entry matches path element
5376             if(poff)
5377                 *poff = off;
5378             inum = de.inum;
5379             return iget(dp->dev, inum);
5380         }
5381     }
5382
5383     return 0;
5384 }
5385
5386
5387
5388
5389
5390
5391
5392
5393
5394
5395
5396
5397
5398
5399

```

```

5400 // Write a new directory entry (name, inum) into the directory dp.
5401 int
5402 dirlink(struct inode *dp, char *name, uint inum)
5403 {
5404     int off;
5405     struct dirent de;
5406     struct inode *ip;
5407
5408     // Check that name is not present.
5409     if((ip = dirlookup(dp, name, 0)) != 0){
5410         iput(ip);
5411         return -1;
5412     }
5413
5414     // Look for an empty dirent.
5415     for(off = 0; off < dp->size; off += sizeof(de)){
5416         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5417             panic("dirlink read");
5418         if(de.inum == 0)
5419             break;
5420     }
5421
5422     strncpy(de.name, name, DIRSIZ);
5423     de.inum = inum;
5424     if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5425         panic("dirlink");
5426
5427     return 0;
5428 }
5429
5430
5431
5432
5433
5434
5435
5436
5437
5438
5439
5440
5441
5442
5443
5444
5445
5446
5447
5448
5449

```

```

5450 // Paths
5451
5452 // Copy the next path element from path into name.
5453 // Return a pointer to the element following the copied one.
5454 // The returned path has no leading slashes,
5455 // so the caller can check *path=='\0' to see if the name is the last one.
5456 // If no name to remove, return 0.
5457 //
5458 // Examples:
5459 //  skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5460 //  skipelem("///a//bb", name) = "bb", setting name = "a"
5461 //  skipelem("a", name) = "", setting name = "a"
5462 //  skipelem("", name) = skipelem("///", name) = 0
5463 //
5464 static char*
5465 skipelem(char *path, char *name)
5466 {
5467     char *s;
5468     int len;
5469
5470     while(*path == '/')
5471         path++;
5472     if(*path == 0)
5473         return 0;
5474     s = path;
5475     while(*path != '/' && *path != 0)
5476         path++;
5477     len = path - s;
5478     if(len >= DIRSIZ)
5479         memmove(name, s, DIRSIZ);
5480     else {
5481         memmove(name, s, len);
5482         name[len] = 0;
5483     }
5484     while(*path == '/')
5485         path++;
5486     return path;
5487 }
5488
5489
5490
5491
5492
5493
5494
5495
5496
5497
5498
5499

```

```

5500 // Look up and return the inode for a path name.
5501 // If parent != 0, return the inode for the parent and copy the final
5502 // path element into name, which must have room for DIRSIZ bytes.
5503 // Must be called inside a transaction since it calls iput().
5504 static struct inode*
5505 namex(char *path, int nameparent, char *name)
5506 {
5507     struct inode *ip, *next;
5508
5509     if(*path == '/')
5510         ip = iget(ROOTDEV, ROOTINO);
5511     else
5512         ip = idup(proc->cwd);
5513
5514     while((path = skipelem(path, name)) != 0){
5515         ilock(ip);
5516         if(ip->type != T_DIR){
5517             iunlockput(ip);
5518             return 0;
5519         }
5520         if(nameparent && *path == '\0'){
5521             // Stop one level early.
5522             iunlock(ip);
5523             return ip;
5524         }
5525         if((next = dirlookup(ip, name, 0)) == 0){
5526             iunlockput(ip);
5527             return 0;
5528         }
5529         iunlockput(ip);
5530         ip = next;
5531     }
5532     if(nameparent){
5533         iput(ip);
5534         return 0;
5535     }
5536     return ip;
5537 }
5538
5539 struct inode*
5540 namei(char *path)
5541 {
5542     char name[DIRSIZ];
5543     return namex(path, 0, name);
5544 }
5545
5546
5547
5548
5549

```

```

5550 struct inode*
5551 nameiparent(char *path, char *name)
5552 {
5553     return namex(path, 1, name);
5554 }
5555
5556
5557
5558
5559
5560
5561
5562
5563
5564
5565
5566
5567
5568
5569
5570
5571
5572
5573
5574
5575
5576
5577
5578
5579
5580
5581
5582
5583
5584
5585
5586
5587
5588
5589
5590
5591
5592
5593
5594
5595
5596
5597
5598
5599

```

```

5600 //
5601 // File descriptors
5602 //
5603
5604 #include "types.h"
5605 #include "defs.h"
5606 #include "param.h"
5607 #include "fs.h"
5608 #include "file.h"
5609 #include "spinlock.h"
5610
5611 struct devsw devsw[NDEV];
5612 struct {
5613     struct spinlock lock;
5614     struct file file[NFILE];
5615 } ftable;
5616
5617 void
5618 fileinit(void)
5619 {
5620     initlock(&ftable.lock, "ftable");
5621 }
5622
5623 // Allocate a file structure.
5624 struct file*
5625 filealloc(void)
5626 {
5627     struct file *f;
5628
5629     acquire(&ftable.lock);
5630     for(f = ftable.file; f < ftable.file + NFILE; f++){
5631         if(f->ref == 0){
5632             f->ref = 1;
5633             release(&ftable.lock);
5634             return f;
5635         }
5636     }
5637     release(&ftable.lock);
5638     return 0;
5639 }
5640
5641
5642
5643
5644
5645
5646
5647
5648
5649

```

```

5650 // Increment ref count for file f.
5651 struct file*
5652 filedup(struct file *f)
5653 {
5654     acquire(&ftable.lock);
5655     if(f->ref < 1)
5656         panic("filedup");
5657     f->ref++;
5658     release(&ftable.lock);
5659     return f;
5660 }
5661
5662 // Close file f. (Decrement ref count, close when reaches 0.)
5663 void
5664 fileclose(struct file *f)
5665 {
5666     struct file ff;
5667
5668     acquire(&ftable.lock);
5669     if(f->ref < 1)
5670         panic("fileclose");
5671     if(--f->ref > 0){
5672         release(&ftable.lock);
5673         return;
5674     }
5675     ff = *f;
5676     f->ref = 0;
5677     f->type = FD_NONE;
5678     release(&ftable.lock);
5679
5680     if(ff.type == FD_PIPE)
5681         pipeclose(ff.pipe, ff.writable);
5682     else if(ff.type == FD_INODE){
5683         begin_op();
5684         iput(ff.ip);
5685         end_op();
5686     }
5687 }
5688
5689
5690
5691
5692
5693
5694
5695
5696
5697
5698
5699

```



```

5700 // Get metadata about file f.
5701 int
5702 filestat(struct file *f, struct stat *st)
5703 {
5704     if(f->type == FD_INODE){
5705         ilock(f->ip);
5706         stati(f->ip, st);
5707         iunlock(f->ip);
5708         return 0;
5709     }
5710     return -1;
5711 }
5712
5713 // Read from file f.
5714 int
5715 fileread(struct file *f, char *addr, int n)
5716 {
5717     int r;
5718
5719     if(f->readable == 0)
5720         return -1;
5721     if(f->type == FD_PIPE)
5722         return piperead(f->pipe, addr, n);
5723     if(f->type == FD_INODE){
5724         ilock(f->ip);
5725         if((r = readi(f->ip, addr, f->off, n)) > 0)
5726             f->off += r;
5727         iunlock(f->ip);
5728         return r;
5729     }
5730     panic("fileread");
5731 }
5732
5733
5734
5735
5736
5737
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749

```

```

5750 // Write to file f.
5751 int
5752 filewrite(struct file *f, char *addr, int n)
5753 {
5754     int r;
5755
5756     if(f->writable == 0)
5757         return -1;
5758     if(f->type == FD_PIPE)
5759         return pipewrite(f->pipe, addr, n);
5760     if(f->type == FD_INODE){
5761         // write a few blocks at a time to avoid exceeding
5762         // the maximum log transaction size, including
5763         // i-node, indirect block, allocation blocks,
5764         // and 2 blocks of slop for non-aligned writes.
5765         // this really belongs lower down, since writei()
5766         // might be writing a device like the console.
5767         int max = ((LOGSIZE-1-1-2) / 2) * 512;
5768         int i = 0;
5769         while(i < n){
5770             int nl = n - i;
5771             if(nl > max)
5772                 nl = max;
5773
5774             begin_op();
5775             ilock(f->ip);
5776             if ((r = writei(f->ip, addr + i, f->off, nl)) > 0)
5777                 f->off += r;
5778             iunlock(f->ip);
5779             end_op();
5780
5781             if(r < 0)
5782                 break;
5783             if(r != nl)
5784                 panic("short filewrite");
5785             i += r;
5786         }
5787         return i == n ? n : -1;
5788     }
5789     panic("filewrite");
5790 }
5791
5792
5793
5794
5795
5796
5797
5798
5799

```

```

5800 //
5801 // File-system system calls.
5802 // Mostly argument checking, since we don't trust
5803 // user code, and calls into file.c and fs.c.
5804 //
5805
5806 #include "types.h"
5807 #include "defs.h"
5808 #include "param.h"
5809 #include "stat.h"
5810 #include "mmu.h"
5811 #include "proc.h"
5812 #include "fs.h"
5813 #include "file.h"
5814 #include "fcntl.h"
5815
5816 // Fetch the nth word-sized system call argument as a file descriptor
5817 // and return both the descriptor and the corresponding struct file.
5818 static int
5819 argfd(int n, int *pfd, struct file **pf)
5820 {
5821     int fd;
5822     struct file *f;
5823
5824     if(argint(n, &fd) < 0)
5825         return -1;
5826     if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
5827         return -1;
5828     if(pfd)
5829         *pfd = fd;
5830     if(pf)
5831         *pf = f;
5832     return 0;
5833 }
5834
5835 // Allocate a file descriptor for the given file.
5836 // Takes over file reference from caller on success.
5837 static int
5838 fdalloc(struct file *f)
5839 {
5840     int fd;
5841
5842     for(fd = 0; fd < NOFILE; fd++){
5843         if(proc->ofile[fd] == 0){
5844             proc->ofile[fd] = f;
5845             return fd;
5846         }
5847     }
5848     return -1;
5849 }

```

```

5850 int
5851 sys_dup(void)
5852 {
5853     struct file *f;
5854     int fd;
5855
5856     if(argfd(0, 0, &f) < 0)
5857         return -1;
5858     if((fd=fdalloc(f)) < 0)
5859         return -1;
5860     filedup(f);
5861     return fd;
5862 }
5863
5864 int
5865 sys_read(void)
5866 {
5867     struct file *f;
5868     int n;
5869     char *p;
5870
5871     if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5872         return -1;
5873     return fileread(f, p, n);
5874 }
5875
5876 int
5877 sys_write(void)
5878 {
5879     struct file *f;
5880     int n;
5881     char *p;
5882
5883     if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5884         return -1;
5885     return filewrite(f, p, n);
5886 }
5887
5888 int
5889 sys_close(void)
5890 {
5891     int fd;
5892     struct file *f;
5893
5894     if(argfd(0, &fd, &f) < 0)
5895         return -1;
5896     proc->ofile[fd] = 0;
5897     fileclose(f);
5898     return 0;
5899 }

```

```

5900 int
5901 sys_fstat(void)
5902 {
5903     struct file *f;
5904     struct stat *st;
5905
5906     if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
5907         return -1;
5908     return filestat(f, st);
5909 }
5910
5911 // Create the path new as a link to the same inode as old.
5912 int
5913 sys_link(void)
5914 {
5915     char name[DIRSIZ], *new, *old;
5916     struct inode *dp, *ip;
5917
5918     if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
5919         return -1;
5920
5921     begin_op();
5922     if((ip = namei(old)) == 0){
5923         end_op();
5924         return -1;
5925     }
5926     ilock(ip);
5927     if(ip->type == T_DIR){
5928         iunlockput(ip);
5929         end_op();
5930         return -1;
5931     }
5932
5933     ip->nlink++;
5934     iupdate(ip);
5935     iunlock(ip);
5936
5937     if((dp = nameiparent(new, name)) == 0)
5938         goto bad;
5939     ilock(dp);
5940     if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
5941         iunlockput(dp);
5942         goto bad;
5943     }
5944     iunlockput(dp);
5945     iput(ip);
5946
5947     end_op();
5948
5949

```

```

5950     return 0;
5951
5952 bad:
5953     ilock(ip);
5954     ip->nlink--;
5955     iupdate(ip);
5956     iunlockput(ip);
5957     end_op();
5958     return -1;
5959 }
5960
5961 // Is the directory dp empty except for "." and ".." ?
5962 static int
5963 isdirempty(struct inode *dp)
5964 {
5965     int off;
5966     struct dirent de;
5967
5968     for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
5969         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5970             panic("isdirempty: readi");
5971         if(de.inum != 0)
5972             return 0;
5973     }
5974     return 1;
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 int
6001 sys_unlink(void)
6002 {
6003     struct inode *ip, *dp;
6004     struct dirent de;
6005     char name[DIRSIZ], *path;
6006     uint off;
6007
6008     if(argstr(0, &path) < 0)
6009         return -1;
6010
6011     begin_op();
6012     if((dp = nameiparent(path, name)) == 0){
6013         end_op();
6014         return -1;
6015     }
6016     ilock(dp);
6017
6018     // Cannot unlink "." or "..".
6020     if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6021         goto bad;
6022
6023     if((ip = dirlookup(dp, name, &off)) == 0)
6024         goto bad;
6025     ilock(ip);
6026
6027     if(ip->nlink < 1)
6028         panic("unlink: nlink < 1");
6029     if(ip->type == T_DIR && !isdirempty(ip)){
6030         iunlockput(ip);
6031         goto bad;
6032     }
6033
6034     memset(&de, 0, sizeof(de));
6035     if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6036         panic("unlink: writei");
6037     if(ip->type == T_DIR){
6038         dp->nlink--;
6039         iupdate(dp);
6040     }
6041     iunlockput(dp);
6042
6043     ip->nlink--;
6044     iupdate(ip);
6045     iunlockput(ip);
6046
6047     end_op();
6048
6049     return 0;

```

```

6050 bad:
6051     iunlockput(dp);
6052     end_op();
6053     return -1;
6054 }
6055
6056 static struct inode*
6057 create(char *path, short type, short major, short minor)
6058 {
6059     uint off;
6060     struct inode *ip, *dp;
6061     char name[DIRSIZ];
6062
6063     if((dp = nameiparent(path, name)) == 0)
6064         return 0;
6065     ilock(dp);
6066
6067     if((ip = dirlookup(dp, name, &off)) != 0){
6068         iunlockput(dp);
6069         ilock(ip);
6070         if(type == T_FILE && ip->type == T_FILE)
6071             return ip;
6072         iunlockput(ip);
6073         return 0;
6074     }
6075
6076     if((ip = ialloc(dp->dev, type)) == 0)
6077         panic("create: ialloc");
6078
6079     ilock(ip);
6080     ip->major = major;
6081     ip->minor = minor;
6082     ip->nlink = 1;
6083     iupdate(ip);
6084
6085     if(type == T_DIR){ // Create . and .. entries.
6086         dp->nlink++; // for ".."
6087         iupdate(dp);
6088         // No ip->nlink++ for ".": avoid cyclic ref count.
6089         if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6090             panic("create dots");
6091     }
6092
6093     if(dirlink(dp, name, ip->inum) < 0)
6094         panic("create: dirlink");
6095
6096     iunlockput(dp);
6097
6098     return ip;
6099 }

```

```

6100 int
6101 sys_open(void)
6102 {
6103     char *path;
6104     int fd, omode;
6105     struct file *f;
6106     struct inode *ip;
6107
6108     if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6109         return -1;
6110
6111     begin_op();
6112
6113     if(omode & O_CREATE){
6114         ip = create(path, T_FILE, 0, 0);
6115         if(ip == 0){
6116             end_op();
6117             return -1;
6118         }
6119     } else {
6120         if((ip = namei(path)) == 0){
6121             end_op();
6122             return -1;
6123         }
6124         ilock(ip);
6125         if(ip->type == T_DIR && omode != O_RDONLY){
6126             iunlockput(ip);
6127             end_op();
6128             return -1;
6129         }
6130     }
6131
6132     if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6133         if(f)
6134             fileclose(f);
6135         iunlockput(ip);
6136         end_op();
6137         return -1;
6138     }
6139     iunlock(ip);
6140     end_op();
6141
6142     f->type = FD_INODE;
6143     f->ip = ip;
6144     f->off = 0;
6145     f->readable = !(omode & O_WRONLY);
6146     f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6147     return fd;
6148 }
6149

```

```

6150 int
6151 sys_mkdir(void)
6152 {
6153     char *path;
6154     struct inode *ip;
6155
6156     begin_op();
6157     if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
6158         end_op();
6159         return -1;
6160     }
6161     iunlockput(ip);
6162     end_op();
6163     return 0;
6164 }
6165
6166 int
6167 sys_mknod(void)
6168 {
6169     struct inode *ip;
6170     char *path;
6171     int len;
6172     int major, minor;
6173
6174     begin_op();
6175     if((len=argstr(0, &path)) < 0 ||
6176        argint(1, &major) < 0 ||
6177        argint(2, &minor) < 0 ||
6178        (ip = create(path, T_DEV, major, minor)) == 0){
6179         end_op();
6180         return -1;
6181     }
6182     iunlockput(ip);
6183     end_op();
6184     return 0;
6185 }
6186
6187
6188
6189
6190
6191
6192
6193
6194
6195
6196
6197
6198
6199

```

```

6200 int
6201 sys_chdir(void)
6202 {
6203     char *path;
6204     struct inode *ip;
6205
6206     begin_op();
6207     if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
6208         end_op();
6209         return -1;
6210     }
6211     ilock(ip);
6212     if(ip->type != T_DIR){
6213         iunlockput(ip);
6214         end_op();
6215         return -1;
6216     }
6217     iunlock(ip);
6218     iput(proc->cwd);
6219     end_op();
6220     proc->cwd = ip;
6221     return 0;
6222 }
6223
6224 int
6225 sys_exec(void)
6226 {
6227     char *path, *argv[MAXARG];
6228     int i;
6229     uint uargv, uarg;
6230
6231     if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6232         return -1;
6233     }
6234     memset(argv, 0, sizeof(argv));
6235     for(i=0;; i++){
6236         if(i >= NELEM(argv))
6237             return -1;
6238         if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6239             return -1;
6240         if(uarg == 0){
6241             argv[i] = 0;
6242             break;
6243         }
6244         if(fetchstr(uarg, &argv[i]) < 0)
6245             return -1;
6246     }
6247     return exec(path, argv);
6248 }
6249

```

```

6250 int
6251 sys_pipe(void)
6252 {
6253     int *fd;
6254     struct file *rf, *wf;
6255     int fd0, fd1;
6256
6257     if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6258         return -1;
6259     if(pipealloc(&rf, &wf) < 0)
6260         return -1;
6261     fd0 = -1;
6262     if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6263         if(fd0 >= 0)
6264             proc->ofile[fd0] = 0;
6265         fileclose(rf);
6266         fileclose(wf);
6267         return -1;
6268     }
6269     fd[0] = fd0;
6270     fd[1] = fd1;
6271     return 0;
6272 }
6273
6274
6275
6276
6277
6278
6279
6280
6281
6282
6283
6284
6285
6286
6287
6288
6289
6290
6291
6292
6293
6294
6295
6296
6297
6298
6299

```

```

6300 #include "types.h"
6301 #include "param.h"
6302 #include "memlayout.h"
6303 #include "mmu.h"
6304 #include "proc.h"
6305 #include "defs.h"
6306 #include "x86.h"
6307 #include "elf.h"
6308
6309 int
6310 exec(char *path, char **argv)
6311 {
6312     char *s, *last;
6313     int i, off;
6314     uint argc, sz, sp, ustack[3+MAXARG+1];
6315     struct elfhdr elf;
6316     struct inode *ip;
6317     struct proghdr ph;
6318     pde_t *pgdir, *oldpgdir;
6319
6320     begin_op();
6321     if((ip = namei(path)) == 0){
6322         end_op();
6323         return -1;
6324     }
6325     ilock(ip);
6326     pgdir = 0;
6327
6328     // Check ELF header
6329     if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
6330         goto bad;
6331     if(elf.magic != ELF_MAGIC)
6332         goto bad;
6333
6334     if((pgdir = setupkvm()) == 0)
6335         goto bad;
6336
6337     // Load program into memory.
6338     sz = 0;
6339     for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6340         if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6341             goto bad;
6342         if(ph.type != ELF_PROG_LOAD)
6343             continue;
6344         if(ph.memsz < ph.filesz)
6345             goto bad;
6346         if((sz = allocvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6347             goto bad;
6348         if(loadvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6349             goto bad;

```

```

6350     }
6351     iunlockput(ip);
6352     end_op();
6353     ip = 0;
6354
6355     // Allocate two pages at the next page boundary.
6356     // Make the first inaccessible. Use the second as the user stack.
6357     sz = PGROUNDUP(sz);
6358     if((sz = allocvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
6359         goto bad;
6360     clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6361     sp = sz;
6362
6363     // Push argument strings, prepare rest of stack in ustack.
6364     for(argc = 0; argv[argc]; argc++) {
6365         if(argc >= MAXARG)
6366             goto bad;
6367         sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6368         if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6369             goto bad;
6370         ustack[3+argc] = sp;
6371     }
6372     ustack[3+argc] = 0;
6373
6374     ustack[0] = 0xffffffff; // fake return PC
6375     ustack[1] = argc;
6376     ustack[2] = sp - (argc+1)*4; // argv pointer
6377
6378     sp -= (3+argc+1) * 4;
6379     if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6380         goto bad;
6381
6382     // Save program name for debugging.
6383     for(last=s=path; *s; s++)
6384         if(*s == '/')
6385             last = s+1;
6386     safestrcpy(proc->name, last, sizeof(proc->name));
6387
6388     // Commit to the user image.
6389     oldpgdir = proc->pgdir;
6390     proc->pgdir = pgdir;
6391     proc->sz = sz;
6392     proc->tf->eip = elf.entry; // main
6393     proc->tf->esp = sp;
6394     switchvm(proc);
6395     freevm(oldpgdir);
6396     return 0;
6397
6398
6399

```

```
6400 bad:
6401     if(pgdir)
6402         freevm(pgdir);
6403     if(ip){
6404         iunlockput(ip);
6405         end_op();
6406     }
6407     return -1;
6408 }
6409
6410
6411
6412
6413
6414
6415
6416
6417
6418
6419
6420
6421
6422
6423
6424
6425
6426
6427
6428
6429
6430
6431
6432
6433
6434
6435
6436
6437
6438
6439
6440
6441
6442
6443
6444
6445
6446
6447
6448
6449
```

```
6450 #include "types.h"
6451 #include "defs.h"
6452 #include "param.h"
6453 #include "mmu.h"
6454 #include "proc.h"
6455 #include "fs.h"
6456 #include "file.h"
6457 #include "spinlock.h"
6458
6459 #define PIPESIZE 512
6460
6461 struct pipe {
6462     struct spinlock lock;
6463     char data[PIPESIZE];
6464     uint nread;    // number of bytes read
6465     uint nwrite;   // number of bytes written
6466     int readopen;  // read fd is still open
6467     int writeopen; // write fd is still open
6468 };
6469
6470 int
6471 pipealloc(struct file **f0, struct file **f1)
6472 {
6473     struct pipe *p;
6474
6475     p = 0;
6476     *f0 = *f1 = 0;
6477     if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
6478         goto bad;
6479     if((p = (struct pipe*)kalloc()) == 0)
6480         goto bad;
6481     p->readopen = 1;
6482     p->writeopen = 1;
6483     p->nwrite = 0;
6484     p->nread = 0;
6485     initlock(&p->lock, "pipe");
6486     (*f0)->type = FD_PIPE;
6487     (*f0)->readable = 1;
6488     (*f0)->writable = 0;
6489     (*f0)->pipe = p;
6490     (*f1)->type = FD_PIPE;
6491     (*f1)->readable = 0;
6492     (*f1)->writable = 1;
6493     (*f1)->pipe = p;
6494     return 0;
6495
6496
6497
6498
6499
```



```

6500 bad:
6501   if(p)
6502       kfree((char*)p);
6503   if(*f0)
6504       fileclose(*f0);
6505   if(*f1)
6506       fileclose(*f1);
6507   return -1;
6508 }
6509
6510 void
6511 pipeclose(struct pipe *p, int writable)
6512 {
6513     acquire(&p->lock);
6514     if(writable){
6515         p->writeopen = 0;
6516         wakeup(&p->nread);
6517     } else {
6518         p->readopen = 0;
6519         wakeup(&p->nwrite);
6520     }
6521     if(p->readopen == 0 && p->writeopen == 0){
6522         release(&p->lock);
6523         kfree((char*)p);
6524     } else
6525         release(&p->lock);
6526 }
6527
6528
6529 int
6530 pipewrite(struct pipe *p, char *addr, int n)
6531 {
6532     int i;
6533
6534     acquire(&p->lock);
6535     for(i = 0; i < n; i++){
6536         while(p->nwrite == p->nread + PIPESIZE){
6537             if(p->readopen == 0 || proc->killed){
6538                 release(&p->lock);
6539                 return -1;
6540             }
6541             wakeup(&p->nread);
6542             sleep(&p->nwrite, &p->lock);
6543         }
6544         p->data[p->nwrite++ % PIPESIZE] = addr[i];
6545     }
6546     wakeup(&p->nread);
6547     release(&p->lock);
6548     return n;
6549 }

```

```

6550 int
6551 piperead(struct pipe *p, char *addr, int n)
6552 {
6553     int i;
6554
6555     acquire(&p->lock);
6556     while(p->nread == p->nwrite && p->writeopen){
6557         if(proc->killed){
6558             release(&p->lock);
6559             return -1;
6560         }
6561         sleep(&p->nread, &p->lock);
6562     }
6563     for(i = 0; i < n; i++){
6564         if(p->nread == p->nwrite)
6565             break;
6566         addr[i] = p->data[p->nread++ % PIPESIZE];
6567     }
6568     wakeup(&p->nwrite);
6569     release(&p->lock);
6570     return i;
6571 }
6572
6573
6574
6575
6576
6577
6578
6579
6580
6581
6582
6583
6584
6585
6586
6587
6588
6589
6590
6591
6592
6593
6594
6595
6596
6597
6598
6599

```

```

6600 #include "types.h"
6601 #include "x86.h"
6602
6603 void*
6604 memset(void *dst, int c, uint n)
6605 {
6606     if ((int)dst%4 == 0 && n%4 == 0){
6607         c &= 0xFF;
6608         stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
6609     } else
6610         stosb(dst, c, n);
6611     return dst;
6612 }
6613
6614 int
6615 memcmp(const void *v1, const void *v2, uint n)
6616 {
6617     const uchar *s1, *s2;
6618
6619     s1 = v1;
6620     s2 = v2;
6621     while(n-- > 0){
6622         if(*s1 != *s2)
6623             return *s1 - *s2;
6624         s1++, s2++;
6625     }
6626
6627     return 0;
6628 }
6629
6630 void*
6631 memmove(void *dst, const void *src, uint n)
6632 {
6633     const char *s;
6634     char *d;
6635
6636     s = src;
6637     d = dst;
6638     if(s < d && s + n > d){
6639         s += n;
6640         d += n;
6641         while(n-- > 0)
6642             *--d = *--s;
6643     } else
6644         while(n-- > 0)
6645             *d++ = *s++;
6646
6647     return dst;
6648 }
6649

```

```

6650 // memcpy exists to placate GCC. Use memmove.
6651 void*
6652 memcpy(void *dst, const void *src, uint n)
6653 {
6654     return memmove(dst, src, n);
6655 }
6656
6657 int
6658 strncmp(const char *p, const char *q, uint n)
6659 {
6660     while(n > 0 && *p && *p == *q)
6661         n--, p++, q++;
6662     if(n == 0)
6663         return 0;
6664     return (uchar)*p - (uchar)*q;
6665 }
6666
6667 char*
6668 strncpy(char *s, const char *t, int n)
6669 {
6670     char *os;
6671
6672     os = s;
6673     while(n-- > 0 && (*s++ = *t++) != 0)
6674         ;
6675     while(n-- > 0)
6676         *s++ = 0;
6677     return os;
6678 }
6679
6680 // Like strncpy but guaranteed to NUL-terminate.
6681 char*
6682 safestrcpy(char *s, const char *t, int n)
6683 {
6684     char *os;
6685
6686     os = s;
6687     if(n <= 0)
6688         return os;
6689     while(--n > 0 && (*s++ = *t++) != 0)
6690         ;
6691     *s = 0;
6692     return os;
6693 }
6694
6695
6696
6697
6698
6699

```

```

6700 int
6701 strlen(const char *s)
6702 {
6703     int n;
6704
6705     for(n = 0; s[n]; n++)
6706         ;
6707     return n;
6708 }
6709
6710
6711
6712
6713
6714
6715
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 // See MultiProcessor Specification Version 1.[14]
6751
6752 struct mp {                // floating pointer
6753     uchar signature[4];    // "_MP_"
6754     void *physaddr;        // phys addr of MP config table
6755     uchar length;          // 1
6756     uchar specrev;         // [14]
6757     uchar checksum;        // all bytes must add up to 0
6758     uchar type;            // MP system config type
6759     uchar imcrp;
6760     uchar reserved[3];
6761 };
6762
6763 struct mpconf {            // configuration table header
6764     uchar signature[4];    // "PCMP"
6765     ushort length;         // total table length
6766     uchar version;         // [14]
6767     uchar checksum;        // all bytes must add up to 0
6768     uchar product[20];     // product id
6769     uint *oemtable;        // OEM table pointer
6770     ushort oemlength;      // OEM table length
6771     ushort entry;          // entry count
6772     uint *lapicaddr;       // address of local APIC
6773     ushort xlength;        // extended table length
6774     uchar xchecksum;       // extended table checksum
6775     uchar reserved;
6776 };
6777
6778 struct mpproc {            // processor table entry
6779     uchar type;            // entry type (0)
6780     uchar apicid;          // local APIC id
6781     uchar version;         // local APIC version
6782     uchar flags;           // CPU flags
6783     #define MPBOOT 0x02    // This proc is the bootstrap processor.
6784     uchar signature[4];    // CPU signature
6785     uint feature;          // feature flags from CPUID instruction
6786     uchar reserved[8];
6787 };
6788
6789 struct mpioapic {          // I/O APIC table entry
6790     uchar type;            // entry type (2)
6791     uchar apicno;          // I/O APIC id
6792     uchar version;         // I/O APIC version
6793     uchar flags;           // I/O APIC flags
6794     uint *addr;            // I/O APIC address
6795 };
6796
6797
6798
6799

```

```
6800 // Table entry types
6801 #define MPPROC    0x00 // One per processor
6802 #define MPBUS      0x01 // One per bus
6803 #define MPIOAPIC   0x02 // One per I/O APIC
6804 #define MPIOINTR   0x03 // One per bus interrupt source
6805 #define MPLINTR    0x04 // One per system interrupt source
6806
6807
6808
6809
6810
6811
6812
6813
6814
6815
6816
6817
6818
6819
6820
6821
6822
6823
6824
6825
6826
6827
6828
6829
6830
6831
6832
6833
6834
6835
6836
6837
6838
6839
6840
6841
6842
6843
6844
6845
6846
6847
6848
6849
```

```
6850 // Blank page.
6851
6852
6853
6854
6855
6856
6857
6858
6859
6860
6861
6862
6863
6864
6865
6866
6867
6868
6869
6870
6871
6872
6873
6874
6875
6876
6877
6878
6879
6880
6881
6882
6883
6884
6885
6886
6887
6888
6889
6890
6891
6892
6893
6894
6895
6896
6897
6898
6899
```

```

6900 // Multiprocessor support
6901 // Search memory for MP description structures.
6902 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
6903
6904 #include "types.h"
6905 #include "defs.h"
6906 #include "param.h"
6907 #include "memlayout.h"
6908 #include "mp.h"
6909 #include "x86.h"
6910 #include "mmu.h"
6911 #include "proc.h"
6912
6913 struct cpu cpus[NCPU];
6914 static struct cpu *bcpu;
6915 int ismp;
6916 int ncpu;
6917 uchar ioapicid;
6918
6919 int
6920 mpbcpu(void)
6921 {
6922     return bcpu-cpus;
6923 }
6924
6925 static uchar
6926 sum(uchar *addr, int len)
6927 {
6928     int i, sum;
6929
6930     sum = 0;
6931     for(i=0; i<len; i++)
6932         sum += addr[i];
6933     return sum;
6934 }
6935
6936 // Look for an MP structure in the len bytes at addr.
6937 static struct mp*
6938 mpsearch1(uint a, int len)
6939 {
6940     uchar *e, *p, *addr;
6941
6942     addr = p2v(a);
6943     e = addr+len;
6944     for(p = addr; p < e; p += sizeof(struct mp))
6945         if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
6946             return (struct mp*)p;
6947     return 0;
6948 }
6949
```

```

6950 // Search for the MP Floating Pointer Structure, which according to the
6951 // spec is in one of the following three locations:
6952 // 1) in the first KB of the EBDA;
6953 // 2) in the last KB of system base memory;
6954 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
6955 static struct mp*
6956 mpsearch(void)
6957 {
6958     uchar *bda;
6959     uint p;
6960     struct mp *mp;
6961
6962     bda = (uchar *) P2V(0x400);
6963     if((p = ((bda[0x0F]<<8) | bda[0x0E]) << 4)){
6964         if((mp = mpsearch1(p, 1024)))
6965             return mp;
6966     } else {
6967         p = ((bda[0x14]<<8) | bda[0x13])*1024;
6968         if((mp = mpsearch1(p-1024, 1024)))
6969             return mp;
6970     }
6971     return mpsearch1(0xF0000, 0x10000);
6972 }
6973
6974 // Search for an MP configuration table. For now,
6975 // don't accept the default configurations (physaddr == 0).
6976 // Check for correct signature, calculate the checksum and,
6977 // if correct, check the version.
6978 // To do: check extended table checksum.
6979 static struct mpconf*
6980 mpconfig(struct mp **pmp)
6981 {
6982     struct mpconf *conf;
6983     struct mp *mp;
6984
6985     if((mp = mpsearch()) == 0 || mp->physaddr == 0)
6986         return 0;
6987     conf = (struct mpconf*) p2v((uint) mp->physaddr);
6988     if(memcmp(conf, "PCMP", 4) != 0)
6989         return 0;
6990     if(conf->version != 1 && conf->version != 4)
6991         return 0;
6992     if(sum((uchar*)conf, conf->length) != 0)
6993         return 0;
6994     *pmp = mp;
6995     return conf;
6996 }
6997
6998
6999

```

```

7000 void
7001 mpinit(void)
7002 {
7003     uchar *p, *e;
7004     struct mp *mp;
7005     struct mpconf *conf;
7006     struct mpproc *proc;
7007     struct mpioapic *ioapic;
7008
7009     bcpu = &cpus[0];
7010     if((conf = mpconfig(&mp)) == 0)
7011         return;
7012     ismp = 1;
7013     lapic = (uint*)conf->lapicaddr;
7014     for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
7015         switch(*p){
7016             case MPPROC:
7017                 proc = (struct mpproc*)p;
7018                 if(ncpu != proc->apicid){
7019                     cprintf("mpinit: ncpu=%d apicid=%d\n", ncpu, proc->apicid);
7020                     ismp = 0;
7021                 }
7022                 if(proc->flags & MPBOOT)
7023                     bcpu = &cpus[ncpu];
7024                 cpus[ncpu].id = ncpu;
7025                 ncpu++;
7026                 p += sizeof(struct mpproc);
7027                 continue;
7028             case MPIOAPIC:
7029                 ioapic = (struct mpioapic*)p;
7030                 ioapicid = ioapic->apicno;
7031                 p += sizeof(struct mpioapic);
7032                 continue;
7033             case MPBUS:
7034             case MPIOINTR:
7035             case MPLINTR:
7036                 p += 8;
7037                 continue;
7038             default:
7039                 cprintf("mpinit: unknown config type %x\n", *p);
7040                 ismp = 0;
7041             }
7042         }
7043         if(!ismp){
7044             // Didn't like what we found; fall back to no MP.
7045             ncpu = 1;
7046             lapic = 0;
7047             ioapicid = 0;
7048             return;
7049         }

```

```

7050     if(mp->imcrp){
7051         // Bochs doesn't support IMCR, so this doesn't run on Bochs.
7052         // But it would on real hardware.
7053         outb(0x22, 0x70); // Select IMCR
7054         outb(0x23, inb(0x23) | 1); // Mask external interrupts.
7055     }
7056 }
7057
7058
7059
7060
7061
7062
7063
7064
7065
7066
7067
7068
7069
7070
7071
7072
7073
7074
7075
7076
7077
7078
7079
7080
7081
7082
7083
7084
7085
7086
7087
7088
7089
7090
7091
7092
7093
7094
7095
7096
7097
7098
7099

```

```

7100 // The local APIC manages internal (non-I/O) interrupts.
7101 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
7102
7103 #include "types.h"
7104 #include "defs.h"
7105 #include "date.h"
7106 #include "memlayout.h"
7107 #include "traps.h"
7108 #include "mmu.h"
7109 #include "x86.h"
7110
7111 // Local APIC registers, divided by 4 for use as uint[] indices.
7112 #define ID      (0x0020/4) // ID
7113 #define VER     (0x0030/4) // Version
7114 #define TPR    (0x0080/4) // Task Priority
7115 #define EOI     (0x00B0/4) // EOI
7116 #define SVR     (0x00F0/4) // Spurious Interrupt Vector
7117 #define ENABLE  (0x00000100 // Unit Enable
7118 #define ESR     (0x0280/4) // Error Status
7119 #define ICRLO   (0x0300/4) // Interrupt Command
7120 #define INIT    (0x00000500 // INIT/RESET
7121 #define STARTUP (0x00000600 // Startup IPI
7122 #define DELIVS  (0x00001000 // Delivery status
7123 #define ASSERT  (0x00004000 // Assert interrupt (vs deassert)
7124 #define DEASSERT (0x00000000
7125 #define LEVEL   (0x00008000 // Level triggered
7126 #define BCAST   (0x00080000 // Send to all APICs, including self.
7127 #define BUSY    (0x00001000
7128 #define FIXED   (0x00000000
7129 #define ICRHI   (0x0310/4) // Interrupt Command [63:32]
7130 #define TIMER   (0x0320/4) // Local Vector Table 0 (TIMER)
7131 #define X1      (0x0000000B // divide counts by 1
7132 #define PERIODIC (0x00020000 // Periodic
7133 #define PCINT   (0x0340/4) // Performance Counter LVT
7134 #define LINT0   (0x0350/4) // Local Vector Table 1 (LINT0)
7135 #define LINT1   (0x0360/4) // Local Vector Table 2 (LINT1)
7136 #define ERROR   (0x0370/4) // Local Vector Table 3 (ERROR)
7137 #define MASKED  (0x00010000 // Interrupt masked
7138 #define TICC    (0x0380/4) // Timer Initial Count
7139 #define TCCR    (0x0390/4) // Timer Current Count
7140 #define TDCR    (0x03E0/4) // Timer Divide Configuration
7141
7142 volatile uint *lapic; // Initialized in mp.c
7143
7144 static void
7145 lapicw(int index, int value)
7146 {
7147     lapic[index] = value;
7148     lapic[ID]; // wait for write to finish, by reading
7149 }

```

```

7150
7151
7152
7153
7154
7155
7156
7157
7158
7159
7160
7161
7162
7163
7164
7165
7166
7167
7168
7169
7170
7171
7172
7173
7174
7175
7176
7177
7178
7179
7180
7181
7182
7183
7184
7185
7186
7187
7188
7189
7190
7191
7192
7193
7194
7195
7196
7197
7198
7199

```

```

7200 void
7201 lapicinit(void)
7202 {
7203     if(!lapic)
7204         return;
7205
7206     // Enable local APIC; set spurious interrupt vector.
7207     lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
7208
7209     // The timer repeatedly counts down at bus frequency
7210     // from lapic[TICR] and then issues an interrupt.
7211     // If xv6 cared more about precise timekeeping,
7212     // TICR would be calibrated using an external time source.
7213     lapicw(TDCR, X1);
7214     lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
7215     lapicw(TICR, 10000000);
7216
7217     // Disable logical interrupt lines.
7218     lapicw(LINT0, MASKED);
7219     lapicw(LINT1, MASKED);
7220
7221     // Disable performance counter overflow interrupts
7222     // on machines that provide that interrupt entry.
7223     if(((lapic[VER]>>16) & 0xFF) >= 4)
7224         lapicw(PCINT, MASKED);
7225
7226     // Map error interrupt to IRQ_ERROR.
7227     lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
7228
7229     // Clear error status register (requires back-to-back writes).
7230     lapicw(ESR, 0);
7231     lapicw(ESR, 0);
7232
7233     // Ack any outstanding interrupts.
7234     lapicw(EOI, 0);
7235
7236     // Send an Init Level De-Assert to synchronise arbitration ID's.
7237     lapicw(ICRHI, 0);
7238     lapicw(ICRLO, BCAST | INIT | LEVEL);
7239     while(lapic[ICRLO] & DELIVS)
7240         ;
7241
7242     // Enable interrupts on the APIC (but not on the processor).
7243     lapicw(TPR, 0);
7244 }
7245
7246
7247
7248
7249

```

```

7250 int
7251 cpunum(void)
7252 {
7253     // Cannot call cpu when interrupts are enabled:
7254     // result not guaranteed to last long enough to be used!
7255     // Would prefer to panic but even printing is chancy here:
7256     // almost everything, including cprintf and panic, calls cpu,
7257     // often indirectly through acquire and release.
7258     if(readeflags() & FL_IF){
7259         static int n;
7260         if(n++ == 0)
7261             cprintf("cpu called from %x with interrupts enabled\n",
7262                     __builtin_return_address(0));
7263     }
7264
7265     if(lapic)
7266         return lapic[ID]>>24;
7267     return 0;
7268 }
7269
7270 // Acknowledge interrupt.
7271 void
7272 lapiceoi(void)
7273 {
7274     if(lapic)
7275         lapicw(EOI, 0);
7276 }
7277
7278 // Spin for a given number of microseconds.
7279 // On real hardware would want to tune this dynamically.
7280 void
7281 microdelay(int us)
7282 {
7283 }
7284
7285 #define CMOS_PORT    0x70
7286 #define CMOS_RETURN  0x71
7287
7288 // Start additional processor running entry code at addr.
7289 // See Appendix B of MultiProcessor Specification.
7290 void
7291 lapicstartap(uchar apicid, uint addr)
7292 {
7293     int i;
7294     ushort *wrv;
7295
7296     // "The BSP must initialize CMOS shutdown code to 0AH
7297     // and the warm reset vector (DWORD based at 40:67) to point at
7298     // the AP startup code prior to the [universal startup algorithm]."
7299     outb(CMOS_PORT, 0xF); // offset 0xF is shutdown code

```



```

7300 outb(CMOS_PORT+1, 0x0A);
7301 wrv = (ushort*)P2V((0x40<<4 | 0x67)); // Warm reset vector
7302 wrv[0] = 0;
7303 wrv[1] = addr >> 4;
7304
7305 // "Universal startup algorithm."
7306 // Send INIT (level-triggered) interrupt to reset other CPU.
7307 lapicw(ICRHI, apicid<<24);
7308 lapicw(ICRLO, INIT | LEVEL | ASSERT);
7309 microdelay(200);
7310 lapicw(ICRLO, INIT | LEVEL);
7311 microdelay(100); // should be 10ms, but too slow in Bochs!
7312
7313 // Send startup IPI (twice!) to enter code.
7314 // Regular hardware is supposed to only accept a STARTUP
7315 // when it is in the halted state due to an INIT. So the second
7316 // should be ignored, but it is part of the official Intel algorithm.
7317 // Bochs complains about the second one. Too bad for Bochs.
7318 for(i = 0; i < 2; i++){
7319     lapicw(ICRHI, apicid<<24);
7320     lapicw(ICRLO, STARTUP | (addr>>12));
7321     microdelay(200);
7322 }
7323 }
7324
7325 #define CMOS_STATA 0x0a
7326 #define CMOS_STATB 0x0b
7327 #define CMOS_UIP (1 << 7) // RTC update in progress
7328
7329 #define SECS 0x00
7330 #define MINS 0x02
7331 #define HOURS 0x04
7332 #define DAY 0x07
7333 #define MONTH 0x08
7334 #define YEAR 0x09
7335
7336 static uint cmos_read(uint reg)
7337 {
7338     outb(CMOS_PORT, reg);
7339     microdelay(200);
7340
7341     return inb(CMOS_RETURN);
7342 }
7343
7344
7345
7346
7347
7348
7349

```

```

7350 static void fill_rtctime(struct rtcdate *r)
7351 {
7352     r->second = cmos_read(SECS);
7353     r->minute = cmos_read(MINS);
7354     r->hour = cmos_read(HOURS);
7355     r->day = cmos_read(DAY);
7356     r->month = cmos_read(MONTH);
7357     r->year = cmos_read(YEAR);
7358 }
7359
7360 // gemu seems to use 24-hour GWT and the values are BCD encoded
7361 void cmostime(struct rtcdate *r)
7362 {
7363     struct rtcdate t1, t2;
7364     int sb, bcd;
7365
7366     sb = cmos_read(CMOS_STATB);
7367
7368     bcd = (sb & (1 << 2)) == 0;
7369
7370     // make sure CMOS doesn't modify time while we read it
7371     for (;;) {
7372         fill_rtctime(&t1);
7373         if (cmos_read(CMOS_STATA) & CMOS_UIP)
7374             continue;
7375         fill_rtctime(&t2);
7376         if (memcmp(&t1, &t2, sizeof(t1)) == 0)
7377             break;
7378     }
7379
7380     // convert
7381     if (bcd) {
7382 #define CONV(x) ((t1.x >> 4) * 10) + (t1.x & 0xf)
7383         CONV(second);
7384         CONV(minute);
7385         CONV(hour);
7386         CONV(day);
7387         CONV(month);
7388         CONV(year);
7389 #undef CONV
7390     }
7391
7392     *r = t1;
7393     r->year += 2000;
7394 }
7395
7396
7397
7398
7399

```

```

7400 // The I/O APIC manages hardware interrupts for an SMP system.
7401 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
7402 // See also picirq.c.
7403
7404 #include "types.h"
7405 #include "defs.h"
7406 #include "traps.h"
7407
7408 #define IOAPIC 0xFEC00000 // Default physical address of IO APIC
7409
7410 #define REG_ID 0x00 // Register index: ID
7411 #define REG_VER 0x01 // Register index: version
7412 #define REG_TABLE 0x10 // Redirection table base
7413
7414 // The redirection table starts at REG_TABLE and uses
7415 // two registers to configure each interrupt.
7416 // The first (low) register in a pair contains configuration bits.
7417 // The second (high) register contains a bitmask telling which
7418 // CPUs can serve that interrupt.
7419 #define INT_DISABLED 0x00010000 // Interrupt disabled
7420 #define INT_LEVEL 0x00008000 // Level-triggered (vs edge-)
7421 #define INT_ACTIVELOW 0x00002000 // Active low (vs high)
7422 #define INT_LOGICAL 0x00000800 // Destination is CPU id (vs APIC ID)
7423
7424 volatile struct ioapic *ioapic;
7425
7426 // IO APIC MMIO structure: write reg, then read or write data.
7427 struct ioapic {
7428     uint reg;
7429     uint pad[3];
7430     uint data;
7431 };
7432
7433 static uint
7434 ioapicread(int reg)
7435 {
7436     ioapic->reg = reg;
7437     return ioapic->data;
7438 }
7439
7440 static void
7441 ioapicwrite(int reg, uint data)
7442 {
7443     ioapic->reg = reg;
7444     ioapic->data = data;
7445 }
7446
7447
7448
7449

```

```

7450 void
7451 ioapicinit(void)
7452 {
7453     int i, id, maxintr;
7454
7455     if(!ismp)
7456         return;
7457
7458     ioapic = (volatile struct ioapic*)IOAPIC;
7459     maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
7460     id = ioapicread(REG_ID) >> 24;
7461     if(id != ioapicid)
7462         cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
7463
7464     // Mark all interrupts edge-triggered, active high, disabled,
7465     // and not routed to any CPUs.
7466     for(i = 0; i <= maxintr; i++){
7467         ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
7468         ioapicwrite(REG_TABLE+2*i+1, 0);
7469     }
7470 }
7471
7472 void
7473 ioapicenable(int irq, int cpunum)
7474 {
7475     if(!ismp)
7476         return;
7477
7478     // Mark interrupt edge-triggered, active high,
7479     // enabled, and routed to the given cpunum,
7480     // which happens to be that cpu's APIC ID.
7481     ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
7482     ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
7483 }
7484
7485
7486
7487
7488
7489
7490
7491
7492
7493
7494
7495
7496
7497
7498
7499

```

```

7500 // Intel 8259A programmable interrupt controllers.
7501
7502 #include "types.h"
7503 #include "x86.h"
7504 #include "traps.h"
7505
7506 // I/O Addresses of the two programmable interrupt controllers
7507 #define IO_PIC1      0x20    // Master (IRQs 0-7)
7508 #define IO_PIC2      0xA0    // Slave (IRQs 8-15)
7509
7510 #define IRQ_SLAVE     2      // IRQ at which slave connects to master
7511
7512 // Current IRQ mask.
7513 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
7514 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
7515
7516 static void
7517 picsetmask(ushort mask)
7518 {
7519     irqmask = mask;
7520     outb(IO_PIC1+1, mask);
7521     outb(IO_PIC2+1, mask >> 8);
7522 }
7523
7524 void
7525 picenable(int irq)
7526 {
7527     picsetmask(irqmask & ~(1<<irq));
7528 }
7529
7530 // Initialize the 8259A interrupt controllers.
7531 void
7532 picinit(void)
7533 {
7534     // mask all interrupts
7535     outb(IO_PIC1+1, 0xFF);
7536     outb(IO_PIC2+1, 0xFF);
7537
7538     // Set up master (8259A-1)
7539
7540     // ICW1: 0001g0hi
7541     //   g: 0 = edge triggering, 1 = level triggering
7542     //   h: 0 = cascaded PICs, 1 = master only
7543     //   i: 0 = no ICW4, 1 = ICW4 required
7544     outb(IO_PIC1, 0x11);
7545
7546     // ICW2: Vector offset
7547     outb(IO_PIC1+1, T_IRQ0);
7548
7549

```

```

7550 // ICW3: (master PIC) bit mask of IR lines connected to slaves
7551 //         (slave PIC) 3-bit # of slave's connection to master
7552 outb(IO_PIC1+1, 1<<IRQ_SLAVE);
7553
7554 // ICW4: 000nbmap
7555 //   n: 1 = special fully nested mode
7556 //   b: 1 = buffered mode
7557 //   m: 0 = slave PIC, 1 = master PIC
7558 //         (ignored when b is 0, as the master/slave role
7559 //         can be hardwired).
7560 //   a: 1 = Automatic EOI mode
7561 //   p: 0 = MCS-80/85 mode, 1 = intel x86 mode
7562 outb(IO_PIC1+1, 0x3);
7563
7564 // Set up slave (8259A-2)
7565 outb(IO_PIC2, 0x11);           // ICW1
7566 outb(IO_PIC2+1, T_IRQ0 + 8);   // ICW2
7567 outb(IO_PIC2+1, IRQ_SLAVE);    // ICW3
7568 // NB Automatic EOI mode doesn't tend to work on the slave.
7569 // Linux source code says it's "to be investigated".
7570 outb(IO_PIC2+1, 0x3);          // ICW4
7571
7572 // OCW3: 0ef0lprs
7573 //   ef: 0x = NOP, 10 = clear specific mask, 11 = set specific mask
7574 //   p: 0 = no polling, 1 = polling mode
7575 //   rs: 0x = NOP, 10 = read IRR, 11 = read ISR
7576 outb(IO_PIC1, 0x68);           // clear specific mask
7577 outb(IO_PIC1, 0x0a);           // read IRR by default
7578
7579 outb(IO_PIC2, 0x68);           // OCW3
7580 outb(IO_PIC2, 0x0a);           // OCW3
7581
7582 if(irqmask != 0xFFFF)
7583     picsetmask(irqmask);
7584 }
7585
7586
7587
7588
7589
7590
7591
7592
7593
7594
7595
7596
7597
7598
7599

```

```

7600 // PC keyboard interface constants
7601
7602 #define KBSTATP      0x64    // kbd controller status port(I)
7603 #define KBS_DIB      0x01    // kbd data in buffer
7604 #define KBDATAP      0x60    // kbd data port(I)
7605
7606 #define NO            0
7607
7608 #define SHIFT         (1<<0)
7609 #define CTL           (1<<1)
7610 #define ALT           (1<<2)
7611
7612 #define CAPSLOCK      (1<<3)
7613 #define NUMLOCK       (1<<4)
7614 #define SCROLLLOCK    (1<<5)
7615
7616 #define E0ESC         (1<<6)
7617
7618 // Special keycodes
7619 #define KEY_HOME      0xE0
7620 #define KEY_END       0xE1
7621 #define KEY_UP        0xE2
7622 #define KEY_DN        0xE3
7623 #define KEY_LF        0xE4
7624 #define KEY_RT        0xE5
7625 #define KEY_PGUP      0xE6
7626 #define KEY_PGDN      0xE7
7627 #define KEY_INS       0xE8
7628 #define KEY_DEL       0xE9
7629
7630 // C('A') == Control-A
7631 #define C(x) (x - '@')
7632
7633 static uchar shiftcode[256] =
7634 {
7635     [0x1D] CTL,
7636     [0x2A] SHIFT,
7637     [0x36] SHIFT,
7638     [0x38] ALT,
7639     [0x9D] CTL,
7640     [0xB8] ALT
7641 };
7642
7643 static uchar togglecode[256] =
7644 {
7645     [0x3A] CAPSLOCK,
7646     [0x45] NUMLOCK,
7647     [0x46] SCROLLLOCK
7648 };
7649

```

```

7650 static uchar normalmap[256] =
7651 {
7652     NO,    0x1B, '1', '2', '3', '4', '5', '6', // 0x00
7653     '7', '8', '9', '0', '-', '=', '\b', '\t',
7654     'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', // 0x10
7655     'o', 'p', '[', ']', '\n', NO, 'a', 's',
7656     'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', // 0x20
7657     '\'', '`', NO, '\\', 'z', 'x', 'c', 'v',
7658     'b', 'n', 'm', ',', '.', '/', NO, '*', // 0x30
7659     NO, ' ', NO, NO, NO, NO, NO, NO,
7660     NO, NO, NO, NO, NO, NO, NO, '7', // 0x40
7661     '8', '9', '-', '4', '5', '6', '+', '1',
7662     '2', '3', '0', '.', NO, NO, NO, NO, // 0x50
7663     [0x9C] '\n', // KP_Enter
7664     [0xB5] '/', // KP_Div
7665     [0xC8] KEY_UP, [0xD0] KEY_DN,
7666     [0xC9] KEY_PGUP, [0xD1] KEY_PGDN,
7667     [0xCB] KEY_LF, [0xCD] KEY_RT,
7668     [0x97] KEY_HOME, [0xCF] KEY_END,
7669     [0xD2] KEY_INS, [0xD3] KEY_DEL
7670 };
7671
7672 static uchar shiftmap[256] =
7673 {
7674     NO,    033, '!', '@', '#', '$', '%', '^', // 0x00
7675     '&', '*', '(', ')', '_', '+', '\b', '\t',
7676     'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', // 0x10
7677     'O', 'P', '[', ']', '\n', NO, 'A', 'S',
7678     'D', 'F', 'G', 'H', 'J', 'K', 'L', ':', // 0x20
7679     '"', '~', NO, '|', 'Z', 'X', 'C', 'V',
7680     'B', 'N', 'M', '<', '>', '?', NO, '*', // 0x30
7681     NO, ' ', NO, NO, NO, NO, NO, NO,
7682     NO, NO, NO, NO, NO, NO, NO, '7', // 0x40
7683     '8', '9', '-', '4', '5', '6', '+', '1',
7684     '2', '3', '0', '.', NO, NO, NO, NO, // 0x50
7685     [0x9C] '\n', // KP_Enter
7686     [0xB5] '/', // KP_Div
7687     [0xC8] KEY_UP, [0xD0] KEY_DN,
7688     [0xC9] KEY_PGUP, [0xD1] KEY_PGDN,
7689     [0xCB] KEY_LF, [0xCD] KEY_RT,
7690     [0x97] KEY_HOME, [0xCF] KEY_END,
7691     [0xD2] KEY_INS, [0xD3] KEY_DEL
7692 };
7693
7694
7695
7696
7697
7698
7699

```

```

7700 static uchar ctlmap[256] =
7701 {
7702     NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7703     NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7704     C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
7705     C('O'),  C('P'),  NO,      NO,      '\r',  NO,      C('A'),  C('S'),
7706     C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
7707     NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
7708     C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'), NO,      NO,
7709     [0x9C] '\r',    // KP_Enter
7710     [0xB5] C('/'),  // KP_Div
7711     [0xC8] KEY_UP,  [0xD0] KEY_DN,
7712     [0xC9] KEY_PGUP, [0xD1] KEY_PGDN,
7713     [0xCB] KEY_LF,  [0xCD] KEY_RT,
7714     [0x97] KEY_HOME, [0xCF] KEY_END,
7715     [0xD2] KEY_INS,  [0xD3] KEY_DEL
7716 };
7717
7718
7719
7720
7721
7722
7723
7724
7725
7726
7727
7728
7729
7730
7731
7732
7733
7734
7735
7736
7737
7738
7739
7740
7741
7742
7743
7744
7745
7746
7747
7748
7749

```

```

7750 #include "types.h"
7751 #include "x86.h"
7752 #include "defs.h"
7753 #include "kbd.h"
7754
7755 int
7756 kbdgetc(void)
7757 {
7758     static uint shift;
7759     static uchar *charcode[4] = {
7760         normalmap, shiftmap, ctlmap, ctlmap
7761     };
7762     uint st, data, c;
7763
7764     st = inb(KBSTATP);
7765     if((st & KBS_DIB) == 0)
7766         return -1;
7767     data = inb(KBDATAP);
7768
7769     if(data == 0xE0){
7770         shift |= E0ESC;
7771         return 0;
7772     } else if(data & 0x80){
7773         // Key released
7774         data = (shift & E0ESC ? data : data & 0x7F);
7775         shift &= ~(shiftcode[data] | E0ESC);
7776         return 0;
7777     } else if(shift & E0ESC){
7778         // Last character was an E0 escape; or with 0x80
7779         data |= 0x80;
7780         shift &= ~E0ESC;
7781     }
7782
7783     shift |= shiftcode[data];
7784     shift ^= togglecode[data];
7785     c = charcode[shift & (CTL | SHIFT)][data];
7786     if(shift & CAPSLOCK){
7787         if('a' <= c && c <= 'z')
7788             c += 'A' - 'a';
7789         else if('A' <= c && c <= 'Z')
7790             c += 'a' - 'A';
7791     }
7792     return c;
7793 }
7794
7795 void
7796 kbdtintr(void)
7797 {
7798     consoleintr(kbdgetc);
7799 }

```

```

7800 // Console input and output.
7801 // Input is from the keyboard or serial port.
7802 // Output is written to the screen and serial port.
7803
7804 #include "types.h"
7805 #include "defs.h"
7806 #include "param.h"
7807 #include "traps.h"
7808 #include "spinlock.h"
7809 #include "fs.h"
7810 #include "file.h"
7811 #include "memlayout.h"
7812 #include "mmu.h"
7813 #include "proc.h"
7814 #include "x86.h"
7815
7816 static void consputc(int);
7817
7818 static int panicked = 0;
7819
7820 static struct {
7821   struct spinlock lock;
7822   int locking;
7823 } cons;
7824
7825 static void
7826 printint(int xx, int base, int sign)
7827 {
7828   static char digits[] = "0123456789abcdef";
7829   char buf[16];
7830   int i;
7831   uint x;
7832
7833   if(sign && (sign = xx < 0))
7834     x = -xx;
7835   else
7836     x = xx;
7837
7838   i = 0;
7839   do{
7840     buf[i++] = digits[x % base];
7841   }while((x /= base) != 0);
7842
7843   if(sign)
7844     buf[i++] = '-';
7845
7846   while(--i >= 0)
7847     consputc(buf[i]);
7848 }
7849
```

```

7850 // Print to the console. only understands %d, %x, %p, %s.
7851 void
7852 cprintf(char *fmt, ...)
7853 {
7854   int i, c, locking;
7855   uint *argp;
7856   char *s;
7857
7858   locking = cons.locking;
7859   if(locking)
7860     acquire(&cons.lock);
7861
7862   if (fmt == 0)
7863     panic("null fmt");
7864
7865   argp = (uint*)(void*)&fmt + 1;
7866   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
7867     if(c != '%'){
7868       consputc(c);
7869       continue;
7870     }
7871     c = fmt[++i] & 0xff;
7872     if(c == 0)
7873       break;
7874     switch(c){
7875     case 'd':
7876       printint(*argp++, 10, 1);
7877       break;
7878     case 'x':
7879     case 'p':
7880       printint(*argp++, 16, 0);
7881       break;
7882     case 's':
7883       if((s = (char*)*argp++) == 0)
7884         s = "(null)";
7885       for(; *s; s++)
7886         consputc(*s);
7887       break;
7888     case '%':
7889       consputc('%');
7890       break;
7891     default:
7892       // Print unknown % sequence to draw attention.
7893       consputc('%');
7894       consputc(c);
7895       break;
7896     }
7897   }
7898
7899
```

```

7900  if(locking)
7901      release(&cons.lock);
7902  }
7903
7904  void
7905  panic(char *s)
7906  {
7907      int i;
7908      uint pcs[10];
7909
7910      cli();
7911      cons.locking = 0;
7912      cprintf("cpu%d: panic: ", cpu->id);
7913      cprintf(s);
7914      cprintf("\n");
7915      getcallerpcs(&s, pcs);
7916      for(i=0; i<10; i++)
7917          cprintf(" %p", pcs[i]);
7918      panicked = 1; // freeze other CPU
7919      for(;;)
7920          ;
7921  }
7922
7923
7924
7925
7926
7927
7928
7929
7930
7931
7932
7933
7934
7935
7936
7937
7938
7939
7940
7941
7942
7943
7944
7945
7946
7947
7948
7949

```

```

7950  #define BACKSPACE 0x100
7951  #define CRTPORT 0x3d4
7952  static ushort *crt = (ushort*)P2V(0xb8000); // CGA memory
7953
7954  static void
7955  cgaputc(int c)
7956  {
7957      int pos;
7958
7959      // Cursor position: col + 80*row.
7960      outb(CRTPORT, 14);
7961      pos = inb(CRTPORT+1) << 8;
7962      outb(CRTPORT, 15);
7963      pos |= inb(CRTPORT+1);
7964
7965      if(c == '\n')
7966          pos += 80 - pos%80;
7967      else if(c == BACKSPACE){
7968          if(pos > 0) --pos;
7969      } else
7970          crt[pos++] = (c&0xff) | 0x0700; // black on white
7971
7972      if((pos/80) >= 24){ // Scroll up.
7973          memmove(crt, crt+80, sizeof(crt[0])*23*80);
7974          pos -= 80;
7975          memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
7976      }
7977
7978      outb(CRTPORT, 14);
7979      outb(CRTPORT+1, pos>>8);
7980      outb(CRTPORT, 15);
7981      outb(CRTPORT+1, pos);
7982      crt[pos] = ' ' | 0x0700;
7983  }
7984
7985  void
7986  consputc(int c)
7987  {
7988      if(panicked){
7989          cli();
7990          for(;;)
7991              ;
7992      }
7993
7994      if(c == BACKSPACE){
7995          uartputc('\b'); uartputc(' '); uartputc('\b');
7996      } else
7997          uartputc(c);
7998      cgaputc(c);
7999  }

```

```

8000 #define INPUT_BUF 128
8001 struct {
8002     struct spinlock lock;
8003     char buf[INPUT_BUF];
8004     uint r; // Read index
8005     uint w; // Write index
8006     uint e; // Edit index
8007 } input;
8008
8009 #define C(x) ((x) - '@') // Control-x
8010
8011 void
8012 consoleintr(int (*getc)(void))
8013 {
8014     int c;
8015
8016     acquire(&input.lock);
8017     while((c = getc()) >= 0){
8018         switch(c){
8019             case C('P'): // Process listing.
8020                 procdump();
8021                 break;
8022             case C('U'): // Kill line.
8023                 while(input.e != input.w &&
8024                     input.buf[(input.e-1) % INPUT_BUF] != '\n'){
8025                     input.e--;
8026                     consputc(BACKSPACE);
8027                 }
8028                 break;
8029             case C('H'): case '\x7f': // Backspace
8030                 if(input.e != input.w){
8031                     input.e--;
8032                     consputc(BACKSPACE);
8033                 }
8034                 break;
8035             default:
8036                 if(c != 0 && input.e-input.r < INPUT_BUF){
8037                     c = (c == '\r') ? '\n' : c;
8038                     input.buf[input.e++ % INPUT_BUF] = c;
8039                     consputc(c);
8040                     if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
8041                         input.w = input.e;
8042                         wakeup(&input.r);
8043                     }
8044                 }
8045                 break;
8046         }
8047     }
8048     release(&input.lock);
8049 }

```

```

8050 int
8051 consoleread(struct inode *ip, char *dst, int n)
8052 {
8053     uint target;
8054     int c;
8055
8056     iunlock(ip);
8057     target = n;
8058     acquire(&input.lock);
8059     while(n > 0){
8060         while(input.r == input.w){
8061             if(proc->killed){
8062                 release(&input.lock);
8063                 ilock(ip);
8064                 return -1;
8065             }
8066             sleep(&input.r, &input.lock);
8067         }
8068         c = input.buf[input.r++ % INPUT_BUF];
8069         if(c == C('D')){ // EOF
8070             if(n < target){
8071                 // Save ^D for next time, to make sure
8072                 // caller gets a 0-byte result.
8073                 input.r--;
8074             }
8075             break;
8076         }
8077         *dst++ = c;
8078         --n;
8079         if(c == '\n')
8080             break;
8081     }
8082     release(&input.lock);
8083     ilock(ip);
8084
8085     return target - n;
8086 }
8087
8088
8089
8090
8091
8092
8093
8094
8095
8096
8097
8098
8099

```



```

8100 int
8101 consolewrite(struct inode *ip, char *buf, int n)
8102 {
8103     int i;
8104
8105     iunlock(ip);
8106     acquire(&cons.lock);
8107     for(i = 0; i < n; i++)
8108         consputc(buf[i] & 0xff);
8109     release(&cons.lock);
8110     ilock(ip);
8111
8112     return n;
8113 }
8114
8115 void
8116 consoleinit(void)
8117 {
8118     initlock(&cons.lock, "console");
8119     initlock(&input.lock, "input");
8120
8121     devsw[CONSOLE].write = consolewrite;
8122     devsw[CONSOLE].read = consoleread;
8123     cons.locking = 1;
8124
8125     picenable(IRQ_KBD);
8126     ioapicenable(IRQ_KBD, 0);
8127 }
8128
8129
8130
8131
8132
8133
8134
8135
8136
8137
8138
8139
8140
8141
8142
8143
8144
8145
8146
8147
8148
8149

```

```

8150 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
8151 // Only used on uniprocessors;
8152 // SMP machines use the local APIC timer.
8153
8154 #include "types.h"
8155 #include "defs.h"
8156 #include "traps.h"
8157 #include "x86.h"
8158
8159 #define IO_TIMER1      0x040          // 8253 Timer #1
8160
8161 // Frequency of all three count-down timers;
8162 // (TIMER_FREQ/freq) is the appropriate count
8163 // to generate a frequency of freq Hz.
8164
8165 #define TIMER_FREQ      1193182
8166 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
8167
8168 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
8169 #define TIMER_SEL0      0x00          // select counter 0
8170 #define TIMER_RATEGEN    0x04          // mode 2, rate generator
8171 #define TIMER_16BIT      0x30          // r/w counter 16 bits, LSB first
8172
8173 void
8174 timerinit(void)
8175 {
8176     // Interrupt 100 times/sec.
8177     outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
8178     outb(IO_TIMER1, TIMER_DIV(100) % 256);
8179     outb(IO_TIMER1, TIMER_DIV(100) / 256);
8180     picenable(IRQ_TIMER);
8181 }
8182
8183
8184
8185
8186
8187
8188
8189
8190
8191
8192
8193
8194
8195
8196
8197
8198
8199

```

```

8200 // Intel 8250 serial port (UART).
8201
8202 #include "types.h"
8203 #include "defs.h"
8204 #include "param.h"
8205 #include "traps.h"
8206 #include "spinlock.h"
8207 #include "fs.h"
8208 #include "file.h"
8209 #include "mmu.h"
8210 #include "proc.h"
8211 #include "x86.h"
8212
8213 #define COM1      0x3f8
8214
8215 static int uart;    // is there a uart?
8216
8217 void
8218 uartinit(void)
8219 {
8220     char *p;
8221
8222     // Turn off the FIFO
8223     outb(COM1+2, 0);
8224
8225     // 9600 baud, 8 data bits, 1 stop bit, parity off.
8226     outb(COM1+3, 0x80);    // Unlock divisor
8227     outb(COM1+0, 115200/9600);
8228     outb(COM1+1, 0);
8229     outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
8230     outb(COM1+4, 0);
8231     outb(COM1+1, 0x01);    // Enable receive interrupts.
8232
8233     // If status is 0xFF, no serial port.
8234     if(inb(COM1+5) == 0xFF)
8235         return;
8236     uart = 1;
8237
8238     // Acknowledge pre-existing interrupt conditions;
8239     // enable interrupts.
8240     inb(COM1+2);
8241     inb(COM1+0);
8242     picenable(IRQ_COM1);
8243     ioapicenable(IRQ_COM1, 0);
8244
8245     // Announce that we're here.
8246     for(p="xv6...\n"; *p; p++)
8247         uartputc(*p);
8248 }
8249

```

```

8250 void
8251 uartputc(int c)
8252 {
8253     int i;
8254
8255     if(!uart)
8256         return;
8257     for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
8258         microdelay(10);
8259     outb(COM1+0, c);
8260 }
8261
8262 static int
8263 uartgetc(void)
8264 {
8265     if(!uart)
8266         return -1;
8267     if(!(inb(COM1+5) & 0x01))
8268         return -1;
8269     return inb(COM1+0);
8270 }
8271
8272 void
8273 uartintr(void)
8274 {
8275     consoleintr(uartgetc);
8276 }
8277
8278
8279
8280
8281
8282
8283
8284
8285
8286
8287
8288
8289
8290
8291
8292
8293
8294
8295
8296
8297
8298
8299

```

```
8300 # Initial process execs /init.
8301
8302 #include "syscall.h"
8303 #include "traps.h"
8304
8305
8306 # exec(init, argv)
8307 .globl start
8308 start:
8309     pushl $argv
8310     pushl $init
8311     pushl $0 // where caller pc would be
8312     movl $SYS_exec, %eax
8313     int $T_SYSCALL
8314
8315 # for(;;) exit();
8316 exit:
8317     movl $SYS_exit, %eax
8318     int $T_SYSCALL
8319     jmp exit
8320
8321 # char init[] = "/init\0";
8322 init:
8323     .string "/init\0"
8324
8325 # char *argv[] = { init, 0 };
8326 .p2align 2
8327 argv:
8328     .long init
8329     .long 0
8330
8331
8332
8333
8334
8335
8336
8337
8338
8339
8340
8341
8342
8343
8344
8345
8346
8347
8348
8349
```

```
8350 #include "syscall.h"
8351 #include "traps.h"
8352
8353 #define SYSCALL(name) \
8354     .globl name; \
8355     name: \
8356         movl $SYS_ ## name, %eax; \
8357         int $T_SYSCALL; \
8358         ret
8359
8360 SYSCALL(fork)
8361 SYSCALL(exit)
8362 SYSCALL(wait)
8363 SYSCALL(pipe)
8364 SYSCALL(read)
8365 SYSCALL(write)
8366 SYSCALL(close)
8367 SYSCALL(kill)
8368 SYSCALL(exec)
8369 SYSCALL(open)
8370 SYSCALL(mknod)
8371 SYSCALL(unlink)
8372 SYSCALL(fstat)
8373 SYSCALL(link)
8374 SYSCALL(mkdir)
8375 SYSCALL(chdir)
8376 SYSCALL(dup)
8377 SYSCALL(getpid)
8378 SYSCALL(sbrk)
8379 SYSCALL(sleep)
8380 SYSCALL(uptime)
8381
8382
8383
8384
8385
8386
8387
8388
8389
8390
8391
8392
8393
8394
8395
8396
8397
8398
8399
```

```

8400 // init: The initial user-level program
8401
8402 #include "types.h"
8403 #include "stat.h"
8404 #include "user.h"
8405 #include "fcntl.h"
8406
8407 char *argv[] = { "sh", 0 };
8408
8409 int
8410 main(void)
8411 {
8412     int pid, wpid;
8413
8414     if(open("console", O_RDWR) < 0){
8415         mknod("console", 1, 1);
8416         open("console", O_RDWR);
8417     }
8418     dup(0); // stdout
8419     dup(0); // stderr
8420
8421     for(;;){
8422         printf(1, "init: starting sh\n");
8423         pid = fork();
8424         if(pid < 0){
8425             printf(1, "init: fork failed\n");
8426             exit();
8427         }
8428         if(pid == 0){
8429             exec("sh", argv);
8430             printf(1, "init: exec sh failed\n");
8431             exit();
8432         }
8433         while((wpid=wait()) >= 0 && wpid != pid)
8434             printf(1, "zombie!\n");
8435     }
8436 }
8437
8438
8439
8440
8441
8442
8443
8444
8445
8446
8447
8448
8449

```

```

8450 // Shell.
8451
8452 #include "types.h"
8453 #include "user.h"
8454 #include "fcntl.h"
8455
8456 // Parsed command representation
8457 #define EXEC 1
8458 #define REDIR 2
8459 #define PIPE 3
8460 #define LIST 4
8461 #define BACK 5
8462
8463 #define MAXARGS 10
8464
8465 struct cmd {
8466     int type;
8467 };
8468
8469 struct execcmd {
8470     int type;
8471     char *argv[MAXARGS];
8472     char *eargv[MAXARGS];
8473 };
8474
8475 struct redircmd {
8476     int type;
8477     struct cmd *cmd;
8478     char *file;
8479     char *efile;
8480     int mode;
8481     int fd;
8482 };
8483
8484 struct pipecmd {
8485     int type;
8486     struct cmd *left;
8487     struct cmd *right;
8488 };
8489
8490 struct listcmd {
8491     int type;
8492     struct cmd *left;
8493     struct cmd *right;
8494 };
8495
8496 struct backcmd {
8497     int type;
8498     struct cmd *cmd;
8499 };

```

```

8500 int fork1(void); // Fork but panics on failure.
8501 void panic(char*);
8502 struct cmd *parsecmd(char*);
8503
8504 // Execute cmd. Never returns.
8505 void
8506 runcmd(struct cmd *cmd)
8507 {
8508     int p[2];
8509     struct backcmd *bcmd;
8510     struct execcmd *ecmd;
8511     struct listcmd *lcmd;
8512     struct pipecmd *pcmd;
8513     struct redircmd *rcmd;
8514
8515     if(cmd == 0)
8516         exit();
8517
8518     switch(cmd->type){
8519     default:
8520         panic("runcmd");
8521
8522     case EXEC:
8523         ecmd = (struct execcmd*)cmd;
8524         if(ecmd->argv[0] == 0)
8525             exit();
8526         exec(ecmd->argv[0], ecmd->argv);
8527         printf(2, "exec %s failed\n", ecmd->argv[0]);
8528         break;
8529
8530     case REDIR:
8531         rcmd = (struct redircmd*)cmd;
8532         close(rcmd->fd);
8533         if(open(rcmd->file, rcmd->mode) < 0){
8534             printf(2, "open %s failed\n", rcmd->file);
8535             exit();
8536         }
8537         runcmd(rcmd->cmd);
8538         break;
8539
8540     case LIST:
8541         lcmd = (struct listcmd*)cmd;
8542         if(fork1() == 0)
8543             runcmd(lcmd->left);
8544         wait();
8545         runcmd(lcmd->right);
8546         break;
8547
8548
8549

```

```

8550     case PIPE:
8551         pcmd = (struct pipecmd*)cmd;
8552         if(pipe(p) < 0)
8553             panic("pipe");
8554         if(fork1() == 0){
8555             close(1);
8556             dup(p[1]);
8557             close(p[0]);
8558             close(p[1]);
8559             runcmd(pcmd->left);
8560         }
8561         if(fork1() == 0){
8562             close(0);
8563             dup(p[0]);
8564             close(p[0]);
8565             close(p[1]);
8566             runcmd(pcmd->right);
8567         }
8568         close(p[0]);
8569         close(p[1]);
8570         wait();
8571         wait();
8572         break;
8573
8574     case BACK:
8575         bcmd = (struct backcmd*)cmd;
8576         if(fork1() == 0)
8577             runcmd(bcmd->cmd);
8578         break;
8579     }
8580     exit();
8581 }
8582
8583 int
8584 getcmd(char *buf, int nbuf)
8585 {
8586     printf(2, "$ ");
8587     memset(buf, 0, nbuf);
8588     gets(buf, nbuf);
8589     if(buf[0] == 0) // EOF
8590         return -1;
8591     return 0;
8592 }
8593
8594
8595
8596
8597
8598
8599

```

```

8600 int
8601 main(void)
8602 {
8603     static char buf[100];
8604     int fd;
8605
8606     // Assumes three file descriptors open.
8607     while((fd = open("console", O_RDWR)) >= 0){
8608         if(fd >= 3){
8609             close(fd);
8610             break;
8611         }
8612     }
8613
8614     // Read and run input commands.
8615     while(getcmd(buf, sizeof(buf)) >= 0){
8616         if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
8617             // Clumsy but will have to do for now.
8618             // Chdir has no effect on the parent if run in the child.
8619             buf[strlen(buf)-1] = 0; // chop \n
8620             if(chdir(buf+3) < 0)
8621                 printf(2, "cannot cd %s\n", buf+3);
8622             continue;
8623         }
8624         if(fork1() == 0)
8625             runcmd(parsecmd(buf));
8626         wait();
8627     }
8628     exit();
8629 }
8630
8631 void
8632 panic(char *s)
8633 {
8634     printf(2, "%s\n", s);
8635     exit();
8636 }
8637
8638 int
8639 fork1(void)
8640 {
8641     int pid;
8642
8643     pid = fork();
8644     if(pid == -1)
8645         panic("fork");
8646     return pid;
8647 }
8648
8649

```

```

8650 // Constructors
8651
8652 struct cmd*
8653 execcmd(void)
8654 {
8655     struct execcmd *cmd;
8656
8657     cmd = malloc(sizeof(*cmd));
8658     memset(cmd, 0, sizeof(*cmd));
8659     cmd->type = EXEC;
8660     return (struct cmd*)cmd;
8661 }
8662
8663 struct cmd*
8664 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
8665 {
8666     struct redircmd *cmd;
8667
8668     cmd = malloc(sizeof(*cmd));
8669     memset(cmd, 0, sizeof(*cmd));
8670     cmd->type = REDIR;
8671     cmd->cmd = subcmd;
8672     cmd->file = file;
8673     cmd->efile = efile;
8674     cmd->mode = mode;
8675     cmd->fd = fd;
8676     return (struct cmd*)cmd;
8677 }
8678
8679 struct cmd*
8680 pipecmd(struct cmd *left, struct cmd *right)
8681 {
8682     struct pipecmd *cmd;
8683
8684     cmd = malloc(sizeof(*cmd));
8685     memset(cmd, 0, sizeof(*cmd));
8686     cmd->type = PIPE;
8687     cmd->left = left;
8688     cmd->right = right;
8689     return (struct cmd*)cmd;
8690 }
8691
8692
8693
8694
8695
8696
8697
8698
8699

```

```

8700 struct cmd*
8701 listcmd(struct cmd *left, struct cmd *right)
8702 {
8703     struct listcmd *cmd;
8704
8705     cmd = malloc(sizeof(*cmd));
8706     memset(cmd, 0, sizeof(*cmd));
8707     cmd->type = LIST;
8708     cmd->left = left;
8709     cmd->right = right;
8710     return (struct cmd*)cmd;
8711 }
8712
8713 struct cmd*
8714 backcmd(struct cmd *subcmd)
8715 {
8716     struct backcmd *cmd;
8717
8718     cmd = malloc(sizeof(*cmd));
8719     memset(cmd, 0, sizeof(*cmd));
8720     cmd->type = BACK;
8721     cmd->cmd = subcmd;
8722     return (struct cmd*)cmd;
8723 }
8724
8725
8726
8727
8728
8729
8730
8731
8732
8733
8734
8735
8736
8737
8738
8739
8740
8741
8742
8743
8744
8745
8746
8747
8748
8749

```

```

8750 // Parsing
8751
8752 char whitespace[] = " \t\r\n\v";
8753 char symbols[] = "<|>&()";
8754
8755 int
8756 gettoken(char **ps, char *es, char **q, char **eq)
8757 {
8758     char *s;
8759     int ret;
8760
8761     s = *ps;
8762     while(s < es && strchr(whitespace, *s))
8763         s++;
8764     if(q)
8765         *q = s;
8766     ret = *s;
8767     switch(*s){
8768     case 0:
8769         break;
8770     case '|':
8771     case '(':
8772     case ')':
8773     case ';':
8774     case '&':
8775     case '<':
8776         s++;
8777         break;
8778     case '>':
8779         s++;
8780         if(*s == '>'){
8781             ret = '+';
8782             s++;
8783         }
8784         break;
8785     default:
8786         ret = 'a';
8787         while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
8788             s++;
8789         break;
8790     }
8791     if(eq)
8792         *eq = s;
8793
8794     while(s < es && strchr(whitespace, *s))
8795         s++;
8796     *ps = s;
8797     return ret;
8798 }
8799

```

```

8800 int
8801 peek(char **ps, char *es, char *toks)
8802 {
8803     char *s;
8804
8805     s = *ps;
8806     while(s < es && strchr(whitespace, *s))
8807         s++;
8808     *ps = s;
8809     return *s && strchr(toks, *s);
8810 }
8811
8812 struct cmd *parseline(char**, char*);
8813 struct cmd *parsepipe(char**, char*);
8814 struct cmd *parseexec(char**, char*);
8815 struct cmd *nulterminate(struct cmd*);
8816
8817 struct cmd*
8818 parsecmd(char *s)
8819 {
8820     char *es;
8821     struct cmd *cmd;
8822
8823     es = s + strlen(s);
8824     cmd = parseline(&s, es);
8825     peek(&s, es, "");
8826     if(s != es){
8827         printf(2, "leftovers: %s\n", s);
8828         panic("syntax");
8829     }
8830     nulterminate(cmd);
8831     return cmd;
8832 }
8833
8834 struct cmd*
8835 parseline(char **ps, char *es)
8836 {
8837     struct cmd *cmd;
8838
8839     cmd = parsepipe(ps, es);
8840     while(peek(ps, es, "&")){
8841         gettoken(ps, es, 0, 0);
8842         cmd = backcmd(cmd);
8843     }
8844     if(peek(ps, es, ";")){
8845         gettoken(ps, es, 0, 0);
8846         cmd = listcmd(cmd, parseline(ps, es));
8847     }
8848     return cmd;
8849 }

```

```

8850 struct cmd*
8851 parsepipe(char **ps, char *es)
8852 {
8853     struct cmd *cmd;
8854
8855     cmd = parseexec(ps, es);
8856     if(peek(ps, es, "|")){
8857         gettoken(ps, es, 0, 0);
8858         cmd = pipecmd(cmd, parsepipe(ps, es));
8859     }
8860     return cmd;
8861 }
8862
8863 struct cmd*
8864 parseredirs(struct cmd *cmd, char **ps, char *es)
8865 {
8866     int tok;
8867     char *q, *eq;
8868
8869     while(peek(ps, es, "<>")){
8870         tok = gettoken(ps, es, 0, 0);
8871         if(gettoken(ps, es, &q, &eq) != 'a')
8872             panic("missing file for redirection");
8873         switch(tok){
8874             case '<':
8875                 cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
8876                 break;
8877             case '>':
8878                 cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8879                 break;
8880             case '+': // >>
8881                 cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8882                 break;
8883         }
8884     }
8885     return cmd;
8886 }
8887
8888
8889
8890
8891
8892
8893
8894
8895
8896
8897
8898
8899

```



```

8900 struct cmd*
8901 parseblock(char **ps, char *es)
8902 {
8903     struct cmd *cmd;
8904
8905     if(!peek(ps, es, "("))
8906         panic("parseblock");
8907     gettoken(ps, es, 0, 0);
8908     cmd = parseline(ps, es);
8909     if(!peek(ps, es, ")"))
8910         panic("syntax - missing )");
8911     gettoken(ps, es, 0, 0);
8912     cmd = parseredirs(cmd, ps, es);
8913     return cmd;
8914 }
8915
8916 struct cmd*
8917 parseexec(char **ps, char *es)
8918 {
8919     char *q, *eq;
8920     int tok, argc;
8921     struct execcmd *cmd;
8922     struct cmd *ret;
8923
8924     if(peek(ps, es, "("))
8925         return parseblock(ps, es);
8926
8927     ret = execcmd();
8928     cmd = (struct execcmd*)ret;
8929
8930     argc = 0;
8931     ret = parseredirs(ret, ps, es);
8932     while(!peek(ps, es, "|&")){
8933         if((tok=gettoken(ps, es, &q, &eq)) == 0)
8934             break;
8935         if(tok != 'a')
8936             panic("syntax");
8937         cmd->argv[argc] = q;
8938         cmd->eargv[argc] = eq;
8939         argc++;
8940         if(argc >= MAXARGS)
8941             panic("too many args");
8942         ret = parseredirs(ret, ps, es);
8943     }
8944     cmd->argv[argc] = 0;
8945     cmd->eargv[argc] = 0;
8946     return ret;
8947 }
8948
8949

```

```

8950 // NUL-terminate all the counted strings.
8951 struct cmd*
8952 nulterminate(struct cmd *cmd)
8953 {
8954     int i;
8955     struct backcmd *bcmd;
8956     struct execcmd *ecmd;
8957     struct listcmd *lcmd;
8958     struct pipecmd *pcmd;
8959     struct redircmd *rcmd;
8960
8961     if(cmd == 0)
8962         return 0;
8963
8964     switch(cmd->type){
8965     case EXEC:
8966         ecmd = (struct execcmd*)cmd;
8967         for(i=0; ecmd->argv[i]; i++)
8968             *ecmd->eargv[i] = 0;
8969         break;
8970
8971     case REDIR:
8972         rcmd = (struct redircmd*)cmd;
8973         nulterminate(rcmd->cmd);
8974         *rcmd->efile = 0;
8975         break;
8976
8977     case PIPE:
8978         pcmd = (struct pipecmd*)cmd;
8979         nulterminate(pcmd->left);
8980         nulterminate(pcmd->right);
8981         break;
8982
8983     case LIST:
8984         lcmd = (struct listcmd*)cmd;
8985         nulterminate(lcmd->left);
8986         nulterminate(lcmd->right);
8987         break;
8988
8989     case BACK:
8990         bcmd = (struct backcmd*)cmd;
8991         nulterminate(bcmd->cmd);
8992         break;
8993     }
8994     return cmd;
8995 }
8996
8997
8998
8999

```

```

9000 #include "asm.h"
9001 #include "memlayout.h"
9002 #include "mmu.h"
9003
9004 # Start the first CPU: switch to 32-bit protected mode, jump into C.
9005 # The BIOS loads this code from the first sector of the hard disk into
9006 # memory at physical address 0x7c00 and starts executing in real mode
9007 # with %cs=0 %ip=7c00.
9008
9009 .code16                                # Assemble for 16-bit mode
9010 .globl start
9011 start:
9012     cli                                # BIOS enabled interrupts; disable
9013
9014     # Zero data segment registers DS, ES, and SS.
9015     xorw    %ax,%ax                    # Set %ax to zero
9016     movw    %ax,%ds                    # -> Data Segment
9017     movw    %ax,%es                    # -> Extra Segment
9018     movw    %ax,%ss                    # -> Stack Segment
9019
9020     # Physical address line A20 is tied to zero so that the first PCs
9021     # with 2 MB would run software that assumed 1 MB. Undo that.
9022 seta20.1:
9023     inb     $0x64,%al                    # Wait for not busy
9024     testb   $0x2,%al
9025     jnz     seta20.1
9026
9027     movb    $0xd1,%al                    # 0xd1 -> port 0x64
9028     outb    %al,$0x64
9029
9030 seta20.2:
9031     inb     $0x64,%al                    # Wait for not busy
9032     testb   $0x2,%al
9033     jnz     seta20.2
9034
9035     movb    $0xdf,%al                    # 0xdf -> port 0x60
9036     outb    %al,$0x60
9037
9038     # Switch from real to protected mode. Use a bootstrap GDT that makes
9039     # virtual addresses map directly to physical addresses so that the
9040     # effective memory map doesn't change during the transition.
9041     lgdt    gdtdesc
9042     movl    %cr0,%eax
9043     orl     $CR0_PE,%eax
9044     movl    %eax,%cr0
9045
9046
9047
9048
9049

```

```

9050     # Complete transition to 32-bit protected mode by using long jmp
9051     # to reload %cs and %eip. The segment descriptors are set up with no
9052     # translation, so that the mapping is still the identity mapping.
9053     ljmp    $(SEG_KCODE<<3), $start32
9054
9055 .code32 # Tell assembler to generate 32-bit code now.
9056 start32:
9057     # Set up the protected-mode data segment registers
9058     movw    $(SEG_KDATA<<3), %ax        # Our data segment selector
9059     movw    %ax,%ds                    # -> DS: Data Segment
9060     movw    %ax,%es                    # -> ES: Extra Segment
9061     movw    %ax,%ss                    # -> SS: Stack Segment
9062     movw    $0,%ax                    # Zero segments not ready for use
9063     movw    %ax,%fs                    # -> FS
9064     movw    %ax,%gs                    # -> GS
9065
9066     # Set up the stack pointer and call into C.
9067     movl    $start,%esp
9068     call    bootmain
9069
9070     # If bootmain returns (it shouldn't), trigger a Bochs
9071     # breakpoint if running under Bochs, then loop.
9072     movw    $0x8a00,%ax                # 0x8a00 -> port 0x8a00
9073     movw    %ax,%dx
9074     outw    %ax,%dx
9075     movw    $0x8ae0,%ax                # 0x8ae0 -> port 0x8a00
9076     outw    %ax,%dx
9077 spin:
9078     jmp     spin
9079
9080 # Bootstrap GDT
9081 .p2align 2                                # force 4 byte alignment
9082 gdt:
9083     SEG_NULLASM                            # null seg
9084     SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg
9085     SEG_ASM(STA_W, 0x0, 0xffffffff)        # data seg
9086
9087 gdtdesc:
9088     .word    (gdtdesc - gdt - 1)          # sizeof(gdt) - 1
9089     .long    gdt                          # address gdt
9090
9091
9092
9093
9094
9095
9096
9097
9098
9099

```

```

9100 // Boot loader.
9101 //
9102 // Part of the boot sector, along with bootasm.S, which calls bootmain().
9103 // bootasm.S has put the processor into protected 32-bit mode.
9104 // bootmain() loads an ELF kernel image from the disk starting at
9105 // sector 1 and then jumps to the kernel entry routine.
9106
9107 #include "types.h"
9108 #include "elf.h"
9109 #include "x86.h"
9110 #include "memlayout.h"
9111
9112 #define SECTSIZE 512
9113
9114 void readseg(uchar*, uint, uint);
9115
9116 void
9117 bootmain(void)
9118 {
9119     struct elfhdr *elf;
9120     struct proghdr *ph, *eph;
9121     void (*entry)(void);
9122     uchar* pa;
9123
9124     elf = (struct elfhdr*)0x10000; // scratch space
9125
9126     // Read 1st page off disk
9127     readseg((uchar*)elf, 4096, 0);
9128
9129     // Is this an ELF executable?
9130     if(elf->magic != ELF_MAGIC)
9131         return; // let bootasm.S handle error
9132
9133     // Load each program segment (ignores ph flags).
9134     ph = (struct proghdr*)((uchar*)elf + elf->phoff);
9135     eph = ph + elf->phnum;
9136     for(; ph < eph; ph++){
9137         pa = (uchar*)ph->paddr;
9138         readseg(pa, ph->filesz, ph->off);
9139         if(ph->memsz > ph->filesz)
9140             stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
9141     }
9142
9143     // Call the entry point from the ELF header.
9144     // Does not return!
9145     entry = (void(*) (void))(elf->entry);
9146     entry();
9147 }
9148
9149

```

```

9150 void
9151 waitdisk(void)
9152 {
9153     // Wait for disk ready.
9154     while((inb(0x1F7) & 0xC0) != 0x40)
9155         ;
9156 }
9157
9158 // Read a single sector at offset into dst.
9159 void
9160 readsect(void *dst, uint offset)
9161 {
9162     // Issue command.
9163     waitdisk();
9164     outb(0x1F2, 1); // count = 1
9165     outb(0x1F3, offset);
9166     outb(0x1F4, offset >> 8);
9167     outb(0x1F5, offset >> 16);
9168     outb(0x1F6, (offset >> 24) | 0xE0);
9169     outb(0x1F7, 0x20); // cmd 0x20 - read sectors
9170
9171     // Read data.
9172     waitdisk();
9173     insl(0x1F0, dst, SECTSIZE/4);
9174 }
9175
9176 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
9177 // Might copy more than asked.
9178 void
9179 readseg(uchar* pa, uint count, uint offset)
9180 {
9181     uchar* epa;
9182
9183     epa = pa + count;
9184
9185     // Round down to sector boundary.
9186     pa -= offset % SECTSIZE;
9187
9188     // Translate from bytes to sectors; kernel starts at sector 1.
9189     offset = (offset / SECTSIZE) + 1;
9190
9191     // If this is too slow, we could read lots of sectors at a time.
9192     // We'd write more to memory than asked, but it doesn't matter --
9193     // we load in increasing order.
9194     for(; pa < epa; pa += SECTSIZE, offset++)
9195         readsect(pa, offset);
9196 }
9197
9198
9199

```