

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

ACKNOWLEDGMENTS

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

xv6 borrows code from the following sources:

JOS (asm.h, elf.h, mmu.h, bootasm.S, ide.c, console.c, and others)
Plan 9 (entryother.S, mp.h, mp.c, lapic.c)
FreeBSD (ioapic.c)
NetBSD (console.c)

The following people have made contributions:

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

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

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.

# My additions	33 swtch.S	
01 BC_ps.c	33 kalloc.c	# string operations
01 BC_ps.h		71 string.c
02 BCdate.c	# system calls	
02 BC_testgiduid.c	35 traps.h	# low-level hardware
03 Bctime.c	35 vectors.pl	73 mp.h
	36 trapasm.S	74 mp.c
# basic headers	36 trap.c	76 lapic.c
03 types.h	38 syscall.h	79 ioapic.c
04 param.h	38 syscall.c	80 picirq.c
04 memlayout.h	40 sysproc.c	81 kbd.h
05 defs.h	42 halt.c	83 kbd.c
07 x86.h		83 console.c
09 asm.h	# file system	87 timer.c
09 mmu.h	43 buf.h	87 uart.c
12 elf.h	43 fcntl.h	
	44 stat.h	# user-level
# entering xv6	44 fs.h	88 initcode.S
12 entry.S	45 file.h	89 usys.S
13 entryother.S	46 ide.c	89 init.c
14 main.c	48 bio.c	90 sh.c
	50 log.c	
# locks	53 fs.c	# bootloader
17 spinlock.h	61 file.c	97 bootasm.S
18 spinlock.c	63 sysfile.c	98 bootmain.c
	68 exec.c	
# processes		
19 vm.c	# pipes	
25 proc.h	70 pipe.c	# add student files her
26 proc.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 1824
    0627 1824 1828 2697 2760
    2910 2942 2973 3067 3124
    3168 3183 3216 3229 3426
    3443 3716 4122 4142 4757
    4815 4920 4981 5180 5207
    5224 5281 5558 5591 5611
    5640 5660 5670 6179 6204
    6218 7063 7084 7105 8410
    8581 8627 8663
allocproc 2755
    2755 2816 2880
allocuvn 2203
    0672 2203 2217 2859 6896
    6908
alltraps 3604
    3559 3567 3580 3585 3603
    3604
ALT 8160
    8160 8188 8190
argfd 6369
    6369 6406 6421 6433 6444
    6456
argint 3895
    0645 3895 3908 3924 4083
    4106 4120 4193 4205 4220
    6374 6421 6433 6658 6726
    6727 6781
argptr 3904
    0646 3904 4162 4224 6421
    6433 6456 6807
argstr 3921
    0647 3921 6468 6558 6658
    6707 6725 6757 6781
__attribute__ 1560
    0521 0615 1459 1560
BACK 9012
    9012 9127 9420 9689
backcmd 9050 9414
    9050 9064 9128 9414 9416
    9542 9655 9690
BACKSPACE 8500
    8500 8517 8559 8591 8597
balloc 5354
    5354 5374 5717 5725 5729
BBLOCK 4510
    4510 5361 5385
B_BUSY 4309
    4309 4808 4926 4927 4940
    4943 4967 4978 4990
B_DIRTY 4311
    4311 4743 4766 4771 4810
    4828 4940 4969 5289
begin_op 5178
    0585 2937 5178 6233 6324
    6471 6561 6661 6706 6724
    6756 6870
bfree 5379
    5379 5764 5774 5777
bget 4916
    4916 4948 4956
binit 4889
    0512 1481 4889
bmap 5710
    5472 5710 5736 5819 5869
bootmain 9817
    9768 9817
BPB 4507
    4507 4510 5360 5362 5386
bread 4952
    0513 4952 5127 5128 5140
    5156 5238 5239 5332 5343
    5361 5385 5510 5531 5618
    5726 5770 5819 5869
brelse 4976
    0514 4976 4979 5131 5132
    5147 5164 5242 5243 5334
    5346 5367 5372 5392 5516
    5519 5540 5626 5732 5776
    5822 5873
BSIZE 4455
    4307 4455 4473 4501 4507
    4731 4745 4767 5108 5129
    5240 5344 5819 5820 5821
    5865 5869 5870 5871
buf 4300
    0500 0513 0514 0515 0557
    0584 2370 2373 2382 2384
    4300 4304 4305 4306 4662
    4678 4681 4725 4754 4804
    4806 4809 4877 4881 4885
    4891 4903 4915 4918 4951
    4954 4965 4976 5055 5127
    5128 5140 5141 5147 5156
    5157 5163 5164 5238 5239
    5272 5319 5330 5341 5357
    5381 5506 5528 5605 5713
    5759 5805 5855 8379 8390
    8394 8397 8568 8589 8603
    8637 8658 8665 9137 9140

```

```

    9141 9142 9255 9267 9269
    9272 9273 9274 9277 9278
    9282
B_VALID 4310
    4310 4770 4810 4828 4957
bwrite 4965
    0515 4965 4968 5130 5163
    5241
bzero 5339
    5339 5368
C 8181 8574
    8181 8229 8254 8255 8256
    8257 8258 8260 8574 8584
    8587 8594 8605 8638
CAPSLOCK 8162
    8162 8195 8336
cgaputc 8505
    8505 8563
clearpteu 2279
    0681 2279 2285 6910
cli 0807
    0807 0809 1376 1910 8460
    8554 9712
cmd 9016
    9016 9028 9037 9038 9043
    9044 9052 9057 9061 9070
    9073 9078 9086 9092 9096
    9104 9128 9130 9219 9231
    9235 9236 9352 9355 9357
    9358 9359 9360 9363 9364
    9366 9368 9369 9370 9371
    9372 9373 9374 9375 9376
    9379 9380 9382 9384 9385
    9386 9387 9388 9389 9400
    9401 9403 9405 9406 9407
    9408 9409 9410 9413 9414
    9416 9418 9419 9420 9421
    9422 9512 9513 9514 9515
    9517 9521 9524 9530 9531
    9534 9537 9539 9542 9546
    9548 9550 9553 9555 9558
    9560 9563 9564 9575 9578
    9581 9585 9600 9603 9608
    9612 9613 9616 9621 9622
    9628 9637 9638 9644 9645
    9651 9652 9661 9664 9666
    9672 9673 9678 9684 9690
    9691 9694
CMOS_PORT 7835
    7835 7849 7850 7888
CMOS_RETURN 7836
    7836 7891
CMOS_STATA 7875
    7875 7923
CMOS_STATB 7876
    7876 7916
CMOS_UIP 7877
    7877 7923
COM1 8763
    8763 8773 8776 8777 8778
    8779 8780 8781 8784 8790
    8791 8807 8809 8817 8819
commit 5251
    5103 5223 5251
CONSOLE 4587
    4587 8677 8678
consoleinit 8673
    0518 1477 8673
consoleintr 8577
    0520 8348 8577 8825
consoleread 8620
    8620 8678
consolewrite 8658
    8658 8677
consputc 8551
    8366 8397 8418 8436 8439
    8443 8444 8551 8591 8597
    8604 8665
context 2610
    0501 0624 2560 2610 2632
    2800 2801 2802 2803 3078
    3116 3278
CONV 7932
    7932 7933 7934 7935 7936
    7937 7938 7939
copyout 2368
    0680 2368 6918 6929
copyvm 2303
    0677 2303 2314 2316 2884
cprintf 8402
    0519 1474 1514 2217 3276
    3280 3282 3740 3753 3758
    4038 4152 5472 7569 7589
    7811 8012 8402 8462 8463
    8464 8467
cpu 2558
    0560 1474 1514 1516 1528
    1756 1816 1837 1858 1896
    1911 1912 1920 1922 1968
    1981 1987 2126 2127 2128

```

```

2129 2558 2568 2572 2583
3078 3109 3115 3116 3117
3715 3740 3741 3753 3754
3758 3760 7463 7464 7811
8462
cpunum 7801
0575 1538 1974 7801 8023
8032
CR0_PE 0977
0977 1385 1421 9743
CR0_PG 0987
0987 1300 1421
CR0_WP 0983
0983 1300 1421
CR4_PSE 0989
0989 1293 1414
create 6607
6607 6627 6640 6644 6664
6707 6728
CRTPORT 8501
8501 8510 8511 8512 8513
8531 8532 8533 8534
CTL 8159
8159 8185 8189 8335
DAY 7882
7882 7905
deallocvum 2232
0673 2218 2232 2266 2862
DEVSPACE 0454
0454 2082 2095
devsw 4580
4580 4585 5808 5810 5858
5860 6161 8677 8678
dinode 4477
4477 4501 5507 5511 5529
5532 5606 5619
dirent 4515
4515 5914 5955 6516 6554
dirlink 5952
0537 5921 5952 5967 5975
6491 6639 6643 6644
dirlookup 5911
0538 5911 5917 5959 6075
6573 6617
DIRSIZ 4513
4513 4517 5905 5972 6028
6029 6092 6465 6555 6611
dobuiltin 9231
9231 9278
DPL_USER 1029

```

```

1029 1977 1978 2823 2824
3673 3768 3777
EOESC 8166
8166 8320 8324 8325 8327
8330
elfhdr 1205
1205 6865 9819 9824
ELF_MAGIC 1202
1202 6881 9830
ELF_PROG_LOAD 1236
1236 6892
end_op 5203
0586 2939 5203 6235 6329
6473 6480 6498 6507 6563
6597 6602 6666 6671 6677
6686 6690 6708 6712 6729
6733 6758 6764 6769 6872
6902 6955
entry 1290
1211 1286 1289 1290 3552
3553 6942 7321 9821 9845
9846
EOI 7665
7665 7784 7825
ERROR 7686
7686 7777
ESR 7668
7668 7780 7781
exec 6860
0322 0323 0524 4007 6797
6860 8918 8979 8980 9081
9082
EXEC 9008
9008 9077 9359 9665
execcmd 9020 9353
9020 9065 9078 9353 9355
9621 9627 9628 9656 9666
exit 2921
0129 0212 0219 0276 0313
0324 0337 0609 2921 2962
3705 3709 3769 3778 4002
4068 8866 8869 8911 8976
8981 9071 9080 9090 9133
9285 9292
EXTMEM 0452
0452 0458 2079
fdalloc 6388
6388 6408 6682 6812
fetchint 3867
0648 3867 3897 6788

```

```

fetchstr 3879
0649 3879 3926 6794
file 4550
0502 0527 0528 0529 0531
0532 0533 0601 2635 4550
5320 6158 6164 6174 6177
6180 6201 6202 6214 6216
6252 6265 6302 6363 6369
6372 6388 6403 6417 6429
6442 6453 6655 6804 7006
7021 8360 8758 9029 9088
9089 9364 9372 9572
filealloc 6175
0527 6175 6682 7027
fileclose 6214
0528 2932 6214 6220 6447
6684 6815 6816 7054 7056
filedup 6202
0529 2902 6202 6206 6410
fileinit 6168
0530 1482 6168
fileread 6265
0531 6265 6280 6423
filestat 6252
0532 6252 6458
filewrite 6302
0533 6302 6334 6339 6435
FIRST_PROC_GID 2553
2553 2771
FIRST_PROC_UID 2554
2554 2772
FL_IF 0960
0960 1912 1918 2827 3113
7808
fork 2874
0321 0610 2874 4001 4062
8910 8973 8975 9305 9307
forkl 9301
9055 9097 9107 9114 9129
9281 9301
forkret 3133
2668 2803 3133
freerange 3401
3361 3384 3390 3401
freevm 2260
0674 2260 2265 2328 2986
6945 6952
FSSIZE 0412
0412 4729
gatedesc 1151

```

```

0773 0776 1151 3661
getbuiltin 9201
9201 9226
getcallerpcs 1876
0628 1838 1876 3278 8465
getcmd 9137
9137 9267
gettoken 9456
9456 9541 9545 9557 9570
9571 9607 9611 9633
growproc 2853
0611 2853 4109
havedisk1 4680
4680 4714 4812
holding 1894
0629 1827 1854 1894 3107
HOURS 7881
7881 7904
ialloc 5503
0539 5503 5521 6626 6627
IBLOCK 4504
4504 5510 5531 5618
I_BUSY 4575
4575 5612 5614 5637 5641
5663 5665
ICRHI 7679
7679 7787 7857 7869
ICRLO 7669
7669 7788 7789 7858 7860
7870
ID 7662
7662 7698 7816
IDE_BSY 4665
4665 4689
IDE_CMD_READ 4670
4670 4747
IDE_CMD_WRITE 4671
4671 4744
IDE_DF 4667
4667 4691
IDE_DRDY 4666
4666 4689
IDE_ERR 4668
4668 4691
ideinit 4701
0555 1483 4701
ideintr 4752
0556 3724 4752
idelock 4677
4677 4705 4757 4759 4778

```

```

      4815 4829 4832
iderw 4804
      0557 4804 4809 4811 4813
      4958 4970
idestart 4725
      4681 4725 4728 4734 4776
      4825
idewait 4685
      4685 4708 4736 4766
idtinit 3679
      0656 1515 3679
idup 5589
      0540 2903 5589 6062
iget 5554
      5476 5517 5554 5574 5929
      6060
iinit 5468
      0541 3144 5468
ilock 5603
      0542 5603 5609 5629 6065
      6255 6274 6325 6477 6490
      6503 6567 6575 6615 6619
      6629 6674 6761 6875 8632
      8652 8667
inb 0703
      0703 4689 4713 7604 7891
      8314 8317 8511 8513 8784
      8790 8791 8807 8817 8819
      9723 9731 9854
initlock 1812
      0630 1812 2677 3382 3675
      4705 4893 5112 5470 6170
      7035 8675
initlog 5106
      0583 3145 5106 5109
inituvm 2153
      0675 2153 2158 2820
inode 4562
      0503 0537 0538 0539 0540
      0542 0543 0544 0545 0546
      0548 0549 0550 0551 0552
      0676 2168 2636 4556 4562
      4581 4582 5323 5464 5476
      5502 5526 5553 5556 5562
      5588 5589 5603 5635 5658
      5680 5710 5756 5787 5802
      5852 5910 5911 5952 5956
      6054 6057 6089 6100 6466
      6513 6553 6606 6610 6656
      6704 6719 6754 6866 8620
      8658
INPUT_BUF 8566
      8566 8568 8589 8601 8603
      8605 8637
insl 0712
      0712 0714 4767 9873
install_trans 5122
      5122 5171 5256
INT_DISABLED 7969
      7969 8017
ioapic 7977
      7557 7579 7580 7974 7977
      7986 7987 7993 7994 8008
IOAPIC 7958
      7958 8008
ioapicenable 8023
      0560 4707 8023 8682 8793
ioapicid 7467
      0561 7467 7580 7597 8011
      8012
ioapicinit 8001
      0562 1476 8001 8012
ioapicread 7984
      7984 8009 8010
ioapicwrite 7991
      7991 8017 8018 8031 8032
IO_PIC1 8057
      8057 8070 8085 8094 8097
      8102 8112 8126 8127
IO_PIC2 8058
      8058 8071 8086 8115 8116
      8117 8120 8129 8130
IO_TIMER1 8709
      8709 8718 8728 8729
IPB 4501
      4501 4504 5511 5532 5619
iput 5658
      0543 2938 5658 5664 5683
      5960 6083 6234 6496 6768
IRQ_COM1 3533
      3533 3734 8792 8793
IRQ_ERROR 3535
      3535 7777
IRQ_IDE 3534
      3534 3723 3727 4706 4707
IRQ_KBD 3532
      3532 3730 8681 8682
IRQ_SLAVE 8060
      8060 8064 8102 8117
IRQ_SPURIOUS 3536

```

```

      3536 3739 7757
IRQ_TIMER 3531
      3531 3714 3773 7764 8730
isdirempty 6513
      6513 6520 6579
ismp 7465
      0589 1484 7465 7562 7570
      7590 7593 8005 8025
itrunc 5756
      5323 5667 5756
iunlock 5635
      0544 5635 5638 5682 6072
      6257 6277 6328 6486 6689
      6767 8625 8662
iunlockput 5680
      0545 5680 6067 6076 6079
      6479 6492 6495 6506 6580
      6591 6595 6601 6618 6622
      6646 6676 6685 6711 6732
      6763 6901 6954
iupdate 5526
      0546 5526 5669 5782 5878
      6485 6505 6589 6594 6633
      6637
I_INVALID 4576
      4576 5617 5627 5661
kalloc 3438
      0565 1544 2013 2092 2159
      2215 2319 2784 3438 7029
KBDATAP 8154
      8154 8317
kbdgetc 8306
      8306 8348
kbdintr 8346
      0571 3731 8346
KBS_DIB 8153
      8153 8315
KBSTATP 8152
      8152 8314
KERNBASE 0457
      0457 0458 0462 0463 0467
      0468 0470 0471 1565 1883
      2079 2208 2266
KERNLINK 0458
      0458 2080
KEY_DEL 8178
      8178 8219 8241 8265
KEY_DN 8172
      8172 8215 8237 8261
KEY_END 8170
      8170 8218 8240 8264
KEY_HOME 8169
      8169 8218 8240 8264
KEY_INS 8177
      8177 8219 8241 8265
KEY_LF 8173
      8173 8217 8239 8263
KEY_PGDN 8176
      8176 8216 8238 8262
KEY_PGUP 8175
      8175 8216 8238 8262
KEY_RT 8174
      8174 8217 8239 8263
KEY_UP 8171
      8171 8215 8237 8261
kfree 3415
      0566 2248 2250 2270 2273
      2885 2984 3406 3415 3420
      7052 7073
kill 3225
      0612 3225 3759 4006 4085
      8917
kinit1 3380
      0567 1469 3380
kinit2 3388
      0568 1487 3388
KSTACKSIZE 0401
      0401 1304 1313 1545 2129
      2788
kvmalloc 2107
      0668 1470 2107
lapiceoi 7822
      0577 3721 3725 3732 3736
      3742 7822
lapicinit 7751
      0578 1472 1506 7751
lapicstartap 7841
      0579 1549 7841
lapicw 7695
      7695 7757 7763 7764 7765
      7768 7769 7774 7777 7780
      7781 7784 7787 7788 7793
      7825 7857 7858 7860 7869
      7870
lcr3 0840
      0840 2118 2133
lgdt 0762
      0762 0770 1383 1983 9741
lidt 0776
      0776 0784 3681

```

LINT0 7684
 7684 7768
 LINT1 7685
 7685 7769
 LIST 9011
 9011 9095 9407 9683
 listcmd 9041 9401
 9041 9066 9096 9401 9403
 9546 9657 9684
 loadgs 0801
 0801 1984
 loadvm 2168
 0676 2168 2174 2177 6898
 log 5087 5100
 5087 5100 5112 5114 5115
 5116 5126 5127 5128 5140
 5143 5144 5145 5156 5159
 5160 5161 5172 5180 5182
 5183 5184 5186 5188 5189
 5207 5208 5209 5210 5211
 5213 5216 5218 5224 5225
 5226 5227 5237 5238 5239
 5253 5257 5276 5278 5281
 5282 5283 5286 5287 5288
 5290
 logheader 5082
 5082 5094 5108 5109 5141
 5157
 LOGSIZE 0410
 0410 5084 5184 5276 6317
 log_write 5272
 0584 5272 5279 5345 5366
 5391 5515 5539 5730 5872
 ltr 0788
 0788 0790 2130
 makeint 9163
 9163 9184 9190
 mappages 2029
 2029 2098 2161 2222 2322
 MAXARG 0408
 0408 6777 6864 6915
 MAXARGS 9014
 9014 9022 9023 9640
 MAXFILE 4474
 4474 5865
 MAXOPBLOCKS 0409
 0409 0410 0411 5184
 MAX_PROCESS_TABLE 0104
 0104 0109
 memcmp 7165

 0636 7165 7495 7538 7926
 memmove 7181
 0637 1535 2162 2321 2382
 5129 5240 5333 5538 5625
 5821 5871 6029 6031 7181
 7204 8526
 memset 7154
 0638 2016 2094 2160 2221
 2802 2822 3423 5344 5513
 6584 6784 7154 8528 9140
 9358 9369 9385 9406 9419
 microdelay 7831
 0580 7831 7859 7861 7871
 7889 8808
 min 5322
 0345 0346 5322 5820 5870
 MINS 7880
 7880 7903
 MONTH 7883
 7883 7906
 mp 7302
 7302 7458 7487 7494 7495
 7496 7505 7510 7514 7515
 7518 7519 7530 7533 7535
 7537 7544 7554 7560 7600
 mpbcpu 7470
 0590 7470
 MPBUS 7352
 7352 7583
 mpconf 7313
 7313 7529 7532 7537 7555
 mpconfig 7530
 7530 7560
 mpenter 1502
 1502 1546
 mpinit 7551
 0591 1471 7551 7569 7589
 mpioapic 7339
 7339 7557 7579 7581
 MPIOAPIC 7353
 7353 7578
 MPIOINTR 7354
 7354 7584
 MPLINTR 7355
 7355 7585
 mpmain 1512
 1459 1490 1507 1512
 mpproc 7328
 7328 7556 7567 7576
 MPPROC 7351

 7351 7566
 mpsearch 7506
 7506 7535
 mpsearch1 7488
 7488 7514 7518 7521
 multiboot_header 1275
 1274 1275
 namecmp 5903
 0547 5903 5924 6570
 namei 6090
 0548 2832 6090 6472 6670
 6757 6871
 nameiparent 6101
 0549 6055 6070 6082 6101
 6488 6562 6613
 namex 6055
 6055 6093 6103
 NBUF 0411
 0411 4881 4903
 ncpu 7466
 1474 1537 2573 4707 7466
 7568 7569 7573 7574 7575
 7595
 NCPU 0402
 0402 2572 7463
 NDEV 0406
 0406 5808 5858 6161
 NDIRECT 4472
 4472 4474 4483 4573 5715
 5720 5724 5725 5762 5769
 5770 5777 5778
 NELEM 0684
 0684 2097 2721 3272 4032
 6786
 nextpid 2667
 2667 2769 2770 2780
 NFILE 0404
 0404 6164 6180
 NINDIRECT 4473
 4473 4474 5722 5772
 NINODE 0405
 0405 5464 5562
 NO 8156
 8156 8202 8205 8207 8208
 8209 8210 8212 8224 8227
 8229 8230 8231 8232 8234
 8252 8253 8255 8256 8257
 8258
 NOFILE 0403
 0403 2635 2900 2930 6376

 6392
 NPENTRIES 1071
 1071 1561 2267
 NPROC 0400
 0400 2662 2702 2761 2951
 2977 3068 3207 3230 3269
 NPTENTRIES 1072
 1072 2244
 NSEGS 2552
 1961 2552 2562
 nulterminate 9652
 9515 9530 9652 9673 9679
 9680 9685 9686 9691
 NUMLOCK 8163
 8163 8196
 O_CREATE 4353
 4353 6663 9578 9581
 O_RDONLY 4350
 4350 6675 9575
 O_RDWR 4352
 4352 6696 8964 8966 9259
 outb 0721
 0721 4711 4720 4737 4738
 4739 4740 4741 4742 4744
 4747 7603 7604 7849 7850
 7888 8070 8071 8085 8086
 8094 8097 8102 8112 8115
 8116 8117 8120 8126 8127
 8129 8130 8510 8512 8531
 8532 8533 8534 8727 8728
 8729 8773 8776 8777 8778
 8779 8780 8781 8809 9728
 9736 9864 9865 9866 9867
 9868 9869
 outsl 0733
 0733 0735 4745
 outw 0727
 0727 1431 1433 4153 9774
 9776
 O_WRONLY 4351
 4351 6695 6696 9578 9581
 P2V 0468
 0468 1469 1487 7512 7851
 8502
 panic 8455 9289
 0521 1828 1855 1919 1921
 2040 2096 2132 2158 2174
 2177 2248 2265 2285 2314
 2316 2819 2927 2962 3108
 3110 3112 3114 3156 3159

```

3420 3755 4728 4730 4734
4809 4811 4813 4948 4968
4979 5109 5210 5277 5279
5374 5389 5521 5574 5609
5629 5638 5664 5736 5917
5921 5967 5975 6206 6220
6280 6334 6339 6520 6578
6586 6627 6640 6644 8413
8455 8462 8523 9056 9075
9106 9289 9307 9528 9572
9606 9610 9636 9641
panicked 8368
8368 8468 8553
parseblock 9601
9601 9606 9625
parsecmd 9518
9057 9282 9518
parseexec 9617
9514 9555 9617
parseline 9535
9512 9524 9535 9546 9608
parsepipe 9551
9513 9539 9551 9558
parseredirs 9564
9564 9612 9631 9642
PCINT 7683
7683 7774
pde_t 0353
0353 0670 0671 0672 0673
0674 0675 0676 0677 0680
0681 1460 1520 1561 1960
2004 2006 2029 2086 2089
2092 2153 2168 2203 2232
2260 2279 2302 2303 2305
2352 2368 2623 6868
PDX 1062
1062 2009
PDXSHIFT 1077
1062 1068 1077 1565
peek 9501
9501 9525 9540 9544 9556
9569 9605 9609 9624 9632
PGROUNDDOWN 1080
1080 2034 2035 2375
PGROUNDUP 1079
1079 2213 2240 3404 6907
PGSIZE 1073
1073 1079 1080 1560 2016
2044 2045 2094 2157 2160
2161 2173 2175 2179 2182
2214 2221 2222 2241 2244
2312 2321 2322 2379 2385
2821 2828 3405 3419 3423
6908 6910
PHYSTOP 0453
0453 1487 2081 2095 2096
3419
picenable 8075
0595 4706 8075 8681 8730
8792
picinit 8082
0596 1475 8082
picsetmask 8067
8067 8077 8133
pinit 2675
0613 1479 2675
pipe 7011
0504 0602 0603 0604 4004
4555 6231 6272 6309 7011
7023 7029 7035 7039 7043
7061 7080 7101 8913 9105
9106
PIPE 9010
9010 9103 9386 9677
pipealloc 7021
0601 6809 7021
pipeclose 7061
0602 6231 7061
pipecmd 9035 9380
9035 9067 9104 9380 9382
9558 9658 9678
piperead 7101
0603 6272 7101
PIPESIZE 7009
7009 7013 7086 7094 7116
pipewrite 7080
0604 6309 7080
popcli 1916
0633 1871 1916 1919 1921
2134
printint 8376
8376 8426 8430
proc 2621
0505 0608 0678 1455 1808
1956 1988 2123 2129 2569
2584 2621 2630 2656 2662
2665 2684 2702 2754 2757
2761 2775 2776 2777 2813
2857 2859 2862 2865 2866
2877 2884 2890 2891 2892

```

```

2901 2902 2903 2905 2923
2926 2931 2932 2933 2938
2940 2945 2951 2952 2960
2970 2977 2978 3001 3007
3060 3068 3075 3078 3083
3111 3116 3125 3155 3173
3174 3178 3205 3207 3227
3230 3265 3269 3655 3704
3706 3708 3751 3759 3760
3762 3768 3773 3777 3855
3869 3883 3886 3897 3910
4031 4033 4039 4040 4057
4091 4108 4125 4173 4179
4185 4196 4208 4657 5316
6062 6361 6376 6393 6394
6446 6768 6770 6814 6854
6936 6939 6940 6941 6942
6943 6944 7004 7087 7107
7461 7556 7567 7568 7569
7572 8363 8630 8760
procdump 3254
0614 3254 8615
proghdr 1224
1224 6867 9820 9834
PTE_ADDR 1094
1094 2011 2178 2246 2269
2317 2361
PTE_FLAGS 1095
1095 2318
PTE_P 1083
1083 1563 1565 2010 2020
2039 2041 2245 2268 2315
2357
PTE_PS 1090
1090 1563 1565
pte_t 1098
1098 2003 2007 2011 2013
2032 2171 2234 2281 2306
2354
PTE_U 1085
1085 2020 2161 2222 2286
2359
PTE_W 1084
1084 1563 1565 2020 2079
2081 2082 2161 2222
PTX 1065
1065 2022
PTXSHIFT 1076
1065 1068 1076
pushcli 1905
0632 1826 1905 2125
rcr2 0832
0832 3754 3761
readeflags 0794
0794 1909 1918 3113 7808
read_head 5138
5138 5170
readi 5802
0550 2183 5802 5920 5966
6275 6519 6520 6879 6890
readsb 5328
0536 5113 5328 5384 5471
readsect 9860
9860 9895
readseg 9879
9814 9827 9838 9879
recover_from_log 5168
5102 5117 5168
REDIR 9009
9009 9085 9370 9671
redircmd 9026 9364
9026 9068 9086 9364 9366
9575 9578 9581 9659 9672
REG_ID 7960
7960 8010
REG_TABLE 7962
7962 8017 8018 8031 8032
REG_VER 7961
7961 8009
release 1852
0631 1852 1855 2731 2764
2781 2912 2992 3002 3085
3127 3137 3169 3182 3218
3236 3240 3431 3448 3719
4126 4131 4144 4759 4778
4832 4928 4944 4993 5189
5218 5227 5290 5565 5581
5593 5615 5643 5666 5675
6183 6187 6208 6222 6228
7072 7075 7088 7097 7108
7119 8451 8613 8631 8651
8666
ROOTDEV 0407
0407 3144 3145 6060
ROOTINO 4454
4454 6060
run 3364
3261 3364 3365 3371 3417
3427 3440
runcmd 9061

```

```

9061 9075 9092 9098 9100
9112 9119 9130 9282
RUNNING 2618
2618 2693 3077 3111 3261
3773
safestrcpy 7232
0639 2831 2905 6936 7232
sb 5324
0536 4504 4510 5111 5113
5114 5115 5324 5328 5333
5360 5361 5362 5384 5385
5471 5472 5473 5509 5510
5531 5618 7914 7916 7918
sched 3103
0616 2961 3103 3108 3110
3112 3114 3126 3175
scheduler 3058
0615 1517 2560 3058 3078
3116
SCROLLLOCK 8164
8164 8197
SECS 7879
7879 7902
SECTOR_SIZE 4664
4664 4731
SECTSIZE 9812
9812 9873 9886 9889 9894
SEG 1019
1019 1975 1976 1977 1978
1981
SEG16 1023
1023 2126
SEG_ASM 0910
0910 1440 1441 9784 9785
segdesc 1002
0759 0762 1002 1019 1023
1961 2562
seginit 1966
0667 1473 1505 1966
SEG_KCODE 0991
0991 1400 1975 3672 3673
9753
SEG_KCPU 0993
0993 1981 1984 3616
SEG_KDATA 0992
0992 1404 1976 2128 3613
9758
SEG_NULLASM 0904
0904 1439 9783
SEG_TSS 0996
0996 2126 2127 2130
SEG_UCODE 0994
0994 1977 2823
SEG_UDATA 0995
0995 1978 2824
setbuiltin 9175
9175 9225
SETGATE 1171
1171 3672 3673
setupkvm 2087
0670 2087 2109 2310 2818
6884
SHIFT 8158
8158 8186 8187 8335
skipelem 6015
6015 6064
sleep 3153
0617 3007 3153 3156 3159
3259 4013 4129 4829 4931
5183 5186 5613 7092 7111
8635 8929
spinlock 1751
0507 0617 0627 0629 0630
0631 0659 1751 1809 1812
1824 1852 1894 2657 2661
3153 3359 3369 3658 3663
4660 4677 4875 4880 5053
5088 5317 5463 6159 6163
7007 7012 8358 8371 8756
STA_R 0919 1036
0919 1036 1440 1975 1977
9784
start 1375 8858 9711
0319 0331 1374 1375 1417
1425 1427 5089 5114 5127
5140 5156 5238 5472 8857
8858 9710 9711 9767
startothers 1524
1458 1486 1524
stat 4404
0508 0532 0551 4404 5314
5787 6252 6359 6454 8953
stati 5787
0551 5787 6256
STA_W 0918 1035
0918 1035 1441 1976 1978
1981 9785
STA_X 0915 1032
0915 1032 1440 1975 1977
9784

```

```

sti 0813
0813 0815 1923 3064
stosb 0742
0742 0744 7160 9840
stosl 0751
0751 0753 7158
strlen 7251
0640 6917 6918 7251 9179
9182 9188 9203 9235 9272
9523
strncmp 7208 9153
0641 5905 7208 9153 9180
9181 9183 9187 9189 9204
9205 9209 9235
strncpy 7218
0642 5972 7218
STS_IG32 1050
1050 1177
STS_T32A 1047
1047 2126
STS_TG32 1051
1051 1177
sum 7476
7476 7478 7480 7482 7483
7495 7542
superblock 4462
0509 0536 4462 5111 5324
5328
SVR 7666
7666 7757
switchkvm 2116
0679 1504 2110 2116 3079
switchvm 2123
0678 2123 2132 2866 3076
6944
swtch 3308
0624 3078 3116 3307 3308
syscall 4027
0650 3707 3857 4027
SYSCALL 8903 8910 8911 8912 8913 89
8910 8911 8912 8913 8914
8915 8916 8917 8918 8919
8920 8921 8922 8923 8924
8925 8926 8927 8928 8929
8930 8931 8932 8933 8934
8935 8936 8937 8938
sys_chdir 6751
3929 3968 6751
SYS_chdir 3809
3809 3968 4009
sys_close 6439
3930 3980 6439
SYS_close 3821
3821 3980 4021
sys_date 4158
3951 3982 4158
SYS_date 3823
3823 3982 4023
sys_dup 6401
3931 3969 6401
SYS_dup 3810
3810 3969 4010
sys_exec 6775
3932 3966 6775
SYS_exec 3807
3807 3966 4007 8862
sys_exit 4066
3933 3961 4066
SYS_exit 3802
3802 3961 4002 8867
sys_fork 4060
3934 3960 4060
SYS_fork 3801
3801 3960 4001
sys_fstat 6451
3935 3967 6451
SYS_fstat 3808
3808 3967 4008
sys_getgid 4177
3953 3984 4177
SYS_getgid 3825
3825 3984
sys_getpid 4089
3936 3970 4089
SYS_getpid 3811
3811 3970 4011
sys_getppid 4183
3954 3985 4183
SYS_getppid 3826
3826 3985
sys_getprocs 4213
3957 3988 4213
SYS_getprocs 3829
3829 3988
sys_getuid 4171
3952 3983 4171
SYS_getuid 3824
3824 3983
SYS_halt 3822
3822 3981 4022

```

```

sys_kill 4079
  3937 3965 4079
SYS_kill 3806
  3806 3965 4006
sys_link 6463
  3938 3978 6463
SYS_link 3819
  3819 3978 4019
sys_mkdir 6701
  3939 3979 6701
SYS_mkdir 3820
  3820 3979 4020
sys_mknod 6717
  3940 3976 6717
SYS_mknod 3817
  3817 3976 4017
sys_open 6651
  3941 3974 6651
SYS_open 3815
  3815 3974 4015
sys_pipe 6801
  3942 3963 6801
SYS_pipe 3804
  3804 3963 4004
sys_read 6415
  3943 3964 6415
SYS_read 3805
  3805 3964 4005
sys_sbrk 4101
  3944 3971 4101
SYS_sbrk 3812
  3812 3971 4012
sys_setgid 4201
  3956 3987 4201
SYS_setgid 3828
  3828 3987
sys_setuid 4189
  3955 3986 4189
SYS_setuid 3827
  3827 3986
sys_sleep 4115
  3945 3972 4115
SYS_sleep 3813
  3813 3972 4013
sys_unlink 6551
  3946 3977 6551
SYS_unlink 3818
  3818 3977 4018
sys_uptime 4138
  3949 3973 4138
SYS_uptime 3814
  3814 3973 4014
sys_wait 4073
  3947 3962 4073
SYS_wait 3803
  3803 3962 4003
sys_write 6427
  3948 3975 6427
SYS_write 3816
  3816 3975 4016
taskstate 1101
  1101 2561
TDCR 7690
  7690 7763
T_DEV 4402
  4402 5807 5857 6728
T_DIR 4400
  4400 5916 6066 6478 6579
  6587 6635 6675 6707 6762
T_FILE 4401
  4401 6620 6664
ticks 3664
  0657 3664 3717 3718 4123
  4124 4129 4143
tickslock 3663
  0659 3663 3675 3716 3719
  4122 4126 4129 4131 4142
  4144
TICR 7688
  7688 7765
TIMER 7680
  7680 7764
TIMER_16BIT 8721
  8721 8727
TIMER_DIV 8716
  8716 8728 8729
TIMER_FREQ 8715
  8715 8716
timerinit 8724
  0653 1485 8724
TIMER_MODE 8718
  8718 8727
TIMER_RATEGEN 8720
  8720 8727
TIMER_SEL0 8719
  8719 8727
T_IRQ0 3529
  3529 3714 3723 3727 3730
  3734 3738 3739 3773 7757
  7764 7777 8017 8031 8097

```

```

8116
TPR 7664
  7664 7793
trap 3701
  3552 3554 3622 3701 3753
  3755 3758
trapframe 0852
  0852 2631 2792 3701
trapret 3627
  2669 2797 3626 3627
T_SYSCALL 3526
  3526 3673 3703 8863 8868
  8907
tvinit 3667
  0658 1480 3667
uart 8765
  8765 8786 8805 8815
uartgetc 8813
  8813 8825
uartinit 8768
  0662 1478 8768
uartintr 8823
  0663 3735 8823
uartputc 8801
  0664 8560 8562 8797 8801
uproc 0150
  0110 0150 2556 2681 4216
userinit 2811
  0618 1488 2811 2819
uva2ka 2352
  0671 2352 2376
V2P 0467
  0467 2080 2081
V2P_WO 0470
  0470 1286 1296
VER 7663
  7663 7773
wait 2968
  0326 0619 2968 4003 4075
  8912 8983 9099 9123 9124
  9283
waitdisk 9851
  9851 9863 9872
wakeup 3214
  0620 3214 3718 4772 4991
  5216 5226 5642 5672 7066
  7069 7091 7096 7118 8607
wakeup1 3203
  2671 2945 2955 3203 3217
walkpgdir 2004
  2004 2037 2176 2242 2283
  2313 2356
write_head 5154
  5154 5173 5255 5258
writei 5852
  0552 5852 5974 6326 6585
  6586
write_log 5233
  5233 5254
xchg 0819
  0819 1516 1833 1869
YEAR 7884
  7884 7907
yield 3122
  0621 3122 3774

```



```

0100 #include "types.h"
0101 #include "user.h"
0102 #include "BC_ps.h"
0103
0104 #define MAX_PROCESS_TABLE 64 //64 because this is the max number of active
0105 ;
0106 int
0107 main (int argc, char * argv[])
0108 {
0109     int max = MAX_PROCESS_TABLE;
0110     struct uproc table[max];
0111
0112     int procs_recieved = getprocs(max, table);
0113
0114     if(procs_recieved == -1) printf(1, "The system call getprocs failed.\n");
0115
0116     if(procs_recieved == 0) printf(1, "System call getprocs() returned no pr
0117
0118     int i;
0119     for(i = 1; i < procs_recieved; i++) {
0120         printf(1, "Process %d: Name: %s, ", i, table[i].name);
0121         printf(1, "pid: %d, ", table[i].pid);
0122         printf(1, "uid: %d, ", table[i].uid);
0123         printf(1, "gid: %d, ", table[i].gid);
0124         printf(1, "ppid: %d, ", table[i].ppid);
0125         printf(1, "size: %d, ", table[i].size);
0126         printf(1, "state: %s, \n", table[i].state);
0127     }
0128
0129     exit();
0130 }
0131
0132
0133
0134
0135
0136
0137
0138
0139
0140
0141
0142
0143
0144
0145
0146
0147
0148
0149

```

```

0150 struct uproc {
0151     int pid;
0152     int uid;
0153     int gid;
0154     int ppid;
0155     char state[12];
0156     int size;
0157     char name[12];
0158 };
0159
0160
0161
0162
0163
0164
0165
0166
0167
0168
0169
0170
0171
0172
0173
0174
0175
0176
0177
0178
0179
0180
0181
0182
0183
0184
0185
0186
0187
0188
0189
0190
0191
0192
0193
0194
0195
0196
0197
0198
0199

```

```

0200 #include "types.h"
0201 #include "user.h"
0202 #include "date.h"
0203 // #include "lapic.c"
0204
0205 int
0206 main (int argc, char * argv[])
0207 {
0208     struct rtcdate r;
0209
0210     if (date(&r)) {
0211         printf (2, "date~failed\n");
0212     }
0213     exit();
0214
0215     printf(1, "\n %d Hours, %d Minutes, %d Seconds, %d Months, %d Day, %d
0216             r.hour, r.minute, r.second, r.month, r.day, r.year);
0217
0218     exit();
0219 }
0220
0221
0222
0223
0224
0225
0226
0227
0228
0229
0230
0231
0232
0233
0234
0235
0236
0237
0238
0239
0240
0241
0242
0243
0244
0245
0246
0247
0248
0249

```

```

0250 #include "types.h"
0251 #include "user.h"
0252
0253 int
0254 main (int argc, char * argv[])
0255 {
0256     int uid, gid, ppid;
0257
0258     uid = getuid();
0259     printf(2, "Current_UID_is : _%d\n", uid);
0260     printf(2, "Setting_UID_to_100\n" );
0261     setuid(100);
0262     uid = getuid();
0263     printf(2, "Current_UID_is: %d\n", uid);
0264
0265     gid = getgid();
0266     printf(2, "Current_GID_is : _%d\n", gid);
0267     printf(2, "Setting_GID_to_100\n" );
0268     setgid(100);
0269     gid = getgid();
0270     printf(2, "Current_GID_is: %d\n", gid);
0271
0272     ppid = getppid();
0273     printf(2, "My_parent_process_is: %d\n", ppid);
0274     printf(2, "Done!\n");
0275
0276     exit();
0277 }
0278
0279
0280
0281
0282
0283
0284
0285
0286
0287
0288
0289
0290
0291
0292
0293
0294
0295
0296
0297
0298
0299

```

```

0300 #include "types.h"
0301 #include "user.h"
0302 #include "date.h"
0303 // #include "lapic.c"
0304 int get_secs();
0305
0306 int
0307 main (int argc, char * argv[])
0308 {
0309     //error check
0310     if(argc < 2) {
0311         printf(2, "No arguments passed to time\n");
0312         exit();
0313     }
0314
0315     char ** args = argv+1;
0316
0317     //get start time
0318     int start = get_secs();
0319
0320     if(fork() == 0) {
0321         exec(argv[1], args);
0322         printf(2, "exec failed running %s\n", argv[1] );
0323         exit();
0324     }
0325     wait();
0326
0327     //get end time
0328     int finish = get_secs();
0329
0330     int total_time = finish - start;
0331
0332     printf(1, "%s ran in %d seconds \n", argv[1], total_time);
0333
0334     exit();
0335 }
0336
0337 int get_secs()
0338 {
0339     struct rtcdate time;
0340     date(&time);
0341     int hour = time.hour * 3600;
0342     int min = time.minute * 60;
0343     return time.second + min + hour;
0344 }
0345
0346
0347
0348
0349

```

```

0350 typedef unsigned int    uint;
0351 typedef unsigned short  ushort;
0352 typedef unsigned char   uchar;
0353 typedef uint pde_t;
0354
0355
0356
0357
0358
0359
0360
0361
0362
0363
0364
0365
0366
0367
0368
0369
0370
0371
0372
0373
0374
0375
0376
0377
0378
0379
0380
0381
0382
0383
0384
0385
0386
0387
0388
0389
0390
0391
0392
0393
0394
0395
0396
0397
0398
0399

```

```
0400 #define NPROC      64 // maximum number of processes
0401 #define KSTACKSIZE 4096 // size of per-process kernel stack
0402 #define NCPU        8 // maximum number of CPUs
0403 #define NOFILE      16 // open files per process
0404 #define NFILE       100 // open files per system
0405 #define NINODE       50 // maximum number of active i-nodes
0406 #define NDEV        10 // maximum major device number
0407 #define ROOTDEV      1 // device number of file system root disk
0408 #define MAXARG       32 // max exec arguments
0409 #define MAXOPBLOCKS  10 // max # of blocks any FS op writes
0410 #define LOGSIZE      (MAXOPBLOCKS*3) // max data blocks in on-disk log
0411 #define NBUF         (MAXOPBLOCKS*3) // size of disk block cache
0412 #define FSSIZE       1000 // size of file system in blocks
0413
0414
0415
0416
0417
0418
0419
0420
0421
0422
0423
0424
0425
0426
0427
0428
0429
0430
0431
0432
0433
0434
0435
0436
0437
0438
0439
0440
0441
0442
0443
0444
0445
0446
0447
0448
0449
```

```
0450 // Memory layout
0451
0452 #define EXTMEM  0x100000 // Start of extended memory
0453 #define PHYSTOP 0xE000000 // Top physical memory
0454 #define DEVSPACE 0xFE000000 // Other devices are at high addresses
0455
0456 // Key addresses for address space layout (see kmap in vm.c for layout)
0457 #define KERNBASE 0x80000000 // First kernel virtual address
0458 #define KERNLINK (KERNBASE+EXTMEM) // Address where kernel is linked
0459
0460 #ifndef __ASSEMBLER__
0461
0462 static inline uint v2p(void *a) { return ((uint) (a)) - KERNBASE; }
0463 static inline void *p2v(uint a) { return (void *) ((a) + KERNBASE); }
0464
0465 #endif
0466
0467 #define V2P(a) (((uint) (a)) - KERNBASE)
0468 #define P2V(a) (((void *) (a)) + KERNBASE)
0469
0470 #define V2P_WO(x) ((x) - KERNBASE) // same as V2P, but without casts
0471 #define P2V_WO(x) ((x) + KERNBASE) // same as P2V, but without casts
0472
0473
0474
0475
0476
0477
0478
0479
0480
0481
0482
0483
0484
0485
0486
0487
0488
0489
0490
0491
0492
0493
0494
0495
0496
0497
0498
0499
```

```

0500 struct buf;
0501 struct context;
0502 struct file;
0503 struct inode;
0504 struct pipe;
0505 struct proc;
0506 struct rtcdate;
0507 struct spinlock;
0508 struct stat;
0509 struct superblock;
0510
0511 // bio.c
0512 void          binit(void);
0513 struct buf*   bread(uint, uint);
0514 void          brelse(struct buf*);
0515 void          bwrite(struct buf*);
0516
0517 // console.c
0518 void          consoleinit(void);
0519 void          cprintf(char*, ...);
0520 void          consoleintr(int (*)(void));
0521 void          panic(char*) __attribute__((noreturn));
0522
0523 // exec.c
0524 int           exec(char*, char**);
0525
0526 // file.c
0527 struct file*  filealloc(void);
0528 void          fileclose(struct file*);
0529 struct file*  filedup(struct file*);
0530 void          fileinit(void);
0531 int           fileread(struct file*, char*, int n);
0532 int           filestat(struct file*, struct stat*);
0533 int           filewrite(struct file*, char*, int n);
0534
0535 // fs.c
0536 void          readsb(int dev, struct superblock *sb);
0537 int           dirlink(struct inode*, char*, uint);
0538 struct inode* dirlookup(struct inode*, char*, uint*);
0539 struct inode* ialloc(uint, short);
0540 struct inode* idup(struct inode*);
0541 void          iinit(int dev);
0542 void          ilock(struct inode*);
0543 void          iput(struct inode*);
0544 void          iunlock(struct inode*);
0545 void          iunlockput(struct inode*);
0546 void          iupdate(struct inode*);
0547 int           namecmp(const char*, const char*);
0548 struct inode* namei(char*);
0549 struct inode* nameiparent(char*, char*);

```

```

0550 int           readi(struct inode*, char*, uint, uint);
0551 void          stati(struct inode*, struct stat*);
0552 int           writei(struct inode*, char*, uint, uint);
0553
0554 // ide.c
0555 void          ideinit(void);
0556 void          ideintr(void);
0557 void          iderw(struct buf*);
0558
0559 // ioapic.c
0560 void          ioapicenable(int irq, int cpu);
0561 extern uchar  ioapicid;
0562 void          ioapicinit(void);
0563
0564 // kalloc.c
0565 char*         kalloc(void);
0566 void          kfree(char*);
0567 void          kinit1(void*, void*);
0568 void          kinit2(void*, void*);
0569
0570 // kbd.c
0571 void          kbdtintr(void);
0572
0573 // lapic.c
0574 void          cmostime(struct rtcdate *r);
0575 int           cpunum(void);
0576 extern volatile uint* lapic;
0577 void          lapiceoi(void);
0578 void          lapicinit(void);
0579 void          lapicstartap(uchar, uint);
0580 void          microdelay(int);
0581
0582 // log.c
0583 void          initlog(int dev);
0584 void          log_write(struct buf*);
0585 void          begin_op();
0586 void          end_op();
0587
0588 // mp.c
0589 extern int     ismp;
0590 int           mpbcpu(void);
0591 void          mpinit(void);
0592 void          mpstartthem(void);
0593
0594 // picirq.c
0595 void          picenable(int);
0596 void          picinit(void);
0597
0598
0599

```

```

0600 // pipe.c
0601 int      pipealloc(struct file**, struct file**);
0602 void      pipeclose(struct pipe*, int);
0603 int      piperead(struct pipe*, char*, int);
0604 int      pipewrite(struct pipe*, char*, int);
0605
0606
0607 // proc.c
0608 struct proc* copyproc(struct proc*);
0609 void      exit(void);
0610 int      fork(void);
0611 int      growproc(int);
0612 int      kill(int);
0613 void      pinit(void);
0614 void      procdump(void);
0615 void      scheduler(void) __attribute__((noreturn));
0616 void      sched(void);
0617 void      sleep(void*, struct spinlock*);
0618 void      userinit(void);
0619 int      wait(void);
0620 void      wakeup(void*);
0621 void      yield(void);
0622
0623 // swtch.S
0624 void      swtch(struct context**, struct context*);
0625
0626 // spinlock.c
0627 void      acquire(struct spinlock*);
0628 void      getcallerpcs(void*, uint*);
0629 int      holding(struct spinlock*);
0630 void      initlock(struct spinlock*, char*);
0631 void      release(struct spinlock*);
0632 void      pushcli(void);
0633 void      popcli(void);
0634
0635 // string.c
0636 int      memcmp(const void*, const void*, uint);
0637 void*     memmove(void*, const void*, uint);
0638 void*     memset(void*, int, uint);
0639 char*     safestrcpy(char*, const char*, int);
0640 int      strlen(const char*);
0641 int      strncmp(const char*, const char*, uint);
0642 char*     strncpy(char*, const char*, int);
0643
0644 // syscall.c
0645 int      argint(int, int*);
0646 int      argptr(int, char**, int);
0647 int      argstr(int, char**);
0648 int      fetchint(uint, int*);
0649 int      fetchstr(uint, char**);

```

```

0650 void      syscall(void);
0651
0652 // timer.c
0653 void      timerinit(void);
0654
0655 // trap.c
0656 void      idtinit(void);
0657 extern uint ticks;
0658 void      tvinit(void);
0659 extern struct spinlock tickslock;
0660
0661 // uart.c
0662 void      uartinit(void);
0663 void      uartintr(void);
0664 void      uartputc(int);
0665
0666 // vm.c
0667 void      seginit(void);
0668 void      kvmalloc(void);
0669 void      vmenable(void);
0670 pde_t*     setupkvm(void);
0671 char*     uva2ka(pde_t*, char*);
0672 int      allocvm(pde_t*, uint, uint);
0673 int      deallocvm(pde_t*, uint, uint);
0674 void      freevm(pde_t*);
0675 void      inituvmm(pde_t*, char*, uint);
0676 int      loaduvmm(pde_t*, char*, struct inode*, uint, uint);
0677 pde_t*     copyuvmm(pde_t*, uint);
0678 void      switchuvmm(struct proc*);
0679 void      switchkvm(void);
0680 int      copyout(pde_t*, uint, void*, uint);
0681 void      clearpteu(pde_t *pgdir, char *uva);
0682
0683 // number of elements in fixed-size array
0684 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
0685
0686
0687
0688
0689
0690
0691
0692
0693
0694
0695
0696
0697
0698
0699

```

```

0700 // Routines to let C code use special x86 instructions.
0701
0702 static inline uchar
0703 inb(ushort port)
0704 {
0705     uchar data;
0706
0707     asm volatile("in %1,%0" : "=a" (data) : "d" (port));
0708     return data;
0709 }
0710
0711 static inline void
0712 insl(int port, void *addr, int cnt)
0713 {
0714     asm volatile("cld; rep insl" :
0715                 "=D" (addr), "=c" (cnt) :
0716                 "d" (port), "0" (addr), "1" (cnt) :
0717                 "memory", "cc");
0718 }
0719
0720 static inline void
0721 outb(ushort port, uchar data)
0722 {
0723     asm volatile("out %0,%1" : : "a" (data), "d" (port));
0724 }
0725
0726 static inline void
0727 outw(ushort port, ushort data)
0728 {
0729     asm volatile("out %0,%1" : : "a" (data), "d" (port));
0730 }
0731
0732 static inline void
0733 outsl(int port, const void *addr, int cnt)
0734 {
0735     asm volatile("cld; rep outsl" :
0736                 "=S" (addr), "=c" (cnt) :
0737                 "d" (port), "0" (addr), "1" (cnt) :
0738                 "cc");
0739 }
0740
0741 static inline void
0742 stosb(void *addr, int data, int cnt)
0743 {
0744     asm volatile("cld; rep stosb" :
0745                 "=D" (addr), "=c" (cnt) :
0746                 "0" (addr), "1" (cnt), "a" (data) :
0747                 "memory", "cc");
0748 }
0749

```

```

0750 static inline void
0751 stosl(void *addr, int data, int cnt)
0752 {
0753     asm volatile("cld; rep stosl" :
0754                 "=D" (addr), "=c" (cnt) :
0755                 "0" (addr), "1" (cnt), "a" (data) :
0756                 "memory", "cc");
0757 }
0758
0759 struct segdesc;
0760
0761 static inline void
0762 lgdt(struct segdesc *p, int size)
0763 {
0764     volatile ushort pd[3];
0765
0766     pd[0] = size-1;
0767     pd[1] = (uint)p;
0768     pd[2] = (uint)p >> 16;
0769
0770     asm volatile("lgdt (%0)" : : "r" (pd));
0771 }
0772
0773 struct gatedesc;
0774
0775 static inline void
0776 lidt(struct gatedesc *p, int size)
0777 {
0778     volatile ushort pd[3];
0779
0780     pd[0] = size-1;
0781     pd[1] = (uint)p;
0782     pd[2] = (uint)p >> 16;
0783
0784     asm volatile("lidt (%0)" : : "r" (pd));
0785 }
0786
0787 static inline void
0788 ltr(ushort sel)
0789 {
0790     asm volatile("ltr %0" : : "r" (sel));
0791 }
0792
0793 static inline uint
0794 readeflags(void)
0795 {
0796     uint eflags;
0797     asm volatile("pushfl; popl %0" : "=r" (eflags));
0798     return eflags;
0799 }

```

```

0800 static inline void
0801 loadgs(ushort v)
0802 {
0803     asm volatile("movw %0, %%gs" : : "r" (v));
0804 }
0805
0806 static inline void
0807 cli(void)
0808 {
0809     asm volatile("cli");
0810 }
0811
0812 static inline void
0813 sti(void)
0814 {
0815     asm volatile("sti");
0816 }
0817
0818 static inline uint
0819 xchg(volatile uint *addr, uint newval)
0820 {
0821     uint result;
0822
0823     // The + in "+m" denotes a read-modify-write operand.
0824     asm volatile("lock; xchgl %0, %1" :
0825         "+m" (*addr), "=a" (result) :
0826         "l" (newval) :
0827         "cc");
0828     return result;
0829 }
0830
0831 static inline uint
0832 rcr2(void)
0833 {
0834     uint val;
0835     asm volatile("movl %%cr2,%0" : "=r" (val));
0836     return val;
0837 }
0838
0839 static inline void
0840 lcr3(uint val)
0841 {
0842     asm volatile("movl %0,%%cr3" : : "r" (val));
0843 }
0844
0845
0846
0847
0848
0849

```

```

0850 // Layout of the trap frame built on the stack by the
0851 // hardware and by trapasm.S, and passed to trap().
0852 struct trapframe {
0853     // registers as pushed by pusha
0854     uint edi;
0855     uint esi;
0856     uint ebp;
0857     uint oesp;      // useless & ignored
0858     uint ebx;
0859     uint edx;
0860     uint ecx;
0861     uint eax;
0862
0863     // rest of trap frame
0864     ushort gs;
0865     ushort padding1;
0866     ushort fs;
0867     ushort padding2;
0868     ushort es;
0869     ushort padding3;
0870     ushort ds;
0871     ushort padding4;
0872     uint trapno;
0873
0874     // below here defined by x86 hardware
0875     uint err;
0876     uint eip;
0877     ushort cs;
0878     ushort padding5;
0879     uint eflags;
0880
0881     // below here only when crossing rings, such as from user to kernel
0882     uint esp;
0883     ushort ss;
0884     ushort padding6;
0885 };
0886
0887
0888
0889
0890
0891
0892
0893
0894
0895
0896
0897
0898
0899

```



```

0900 //
0901 // assembler macros to create x86 segments
0902 //
0903
0904 #define SEG_NULLASM \
0905     .word 0, 0; \
0906     .byte 0, 0, 0, 0
0907
0908 // The 0xC0 means the limit is in 4096-byte units
0909 // and (for executable segments) 32-bit mode.
0910 #define SEG_ASM(type,base,lim) \
0911     .word (((lim) >> 12) & 0xffff), ((base) & 0xffff); \
0912     .byte (((base) >> 16) & 0xff), (0x90 | (type)), \
0913     (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
0914
0915 #define STA_X 0x8 // Executable segment
0916 #define STA_E 0x4 // Expand down (non-executable segments)
0917 #define STA_C 0x4 // Conforming code segment (executable only)
0918 #define STA_W 0x2 // Writeable (non-executable segments)
0919 #define STA_R 0x2 // Readable (executable segments)
0920 #define STA_A 0x1 // Accessed
0921
0922
0923
0924
0925
0926
0927
0928
0929
0930
0931
0932
0933
0934
0935
0936
0937
0938
0939
0940
0941
0942
0943
0944
0945
0946
0947
0948
0949

```

```

0950 // This file contains definitions for the
0951 // x86 memory management unit (MMU).
0952
0953 // Eflags register
0954 #define FL_CF 0x00000001 // Carry Flag
0955 #define FL_PF 0x00000004 // Parity Flag
0956 #define FL_AF 0x00000010 // Auxiliary carry Flag
0957 #define FL_ZF 0x00000040 // Zero Flag
0958 #define FL_SF 0x00000080 // Sign Flag
0959 #define FL_TF 0x00000100 // Trap Flag
0960 #define FL_IF 0x00000200 // Interrupt Enable
0961 #define FL_DF 0x00000400 // Direction Flag
0962 #define FL_OF 0x00000800 // Overflow Flag
0963 #define FL_IOPL_MASK 0x00003000 // I/O Privilege Level bitmask
0964 #define FL_IOPL_0 0x00000000 // IOPL == 0
0965 #define FL_IOPL_1 0x00001000 // IOPL == 1
0966 #define FL_IOPL_2 0x00002000 // IOPL == 2
0967 #define FL_IOPL_3 0x00003000 // IOPL == 3
0968 #define FL_NT 0x00004000 // Nested Task
0969 #define FL_RF 0x00010000 // Resume Flag
0970 #define FL_VM 0x00020000 // Virtual 8086 mode
0971 #define FL_AC 0x00040000 // Alignment Check
0972 #define FL_VIF 0x00080000 // Virtual Interrupt Flag
0973 #define FL_VIP 0x00100000 // Virtual Interrupt Pending
0974 #define FL_ID 0x00200000 // ID flag
0975
0976 // Control Register flags
0977 #define CR0_PE 0x00000001 // Protection Enable
0978 #define CR0_MP 0x00000002 // Monitor coProcessor
0979 #define CR0_EM 0x00000004 // Emulation
0980 #define CR0_TS 0x00000008 // Task Switched
0981 #define CR0_ET 0x00000010 // Extension Type
0982 #define CR0_NE 0x00000020 // Numeric Error
0983 #define CR0_WP 0x00010000 // Write Protect
0984 #define CR0_AM 0x00040000 // Alignment Mask
0985 #define CR0_NW 0x00080000 // Not Writethrough
0986 #define CR0_CD 0x00100000 // Cache Disable
0987 #define CR0_PG 0x00200000 // Paging
0988
0989 #define CR4_PSE 0x00000010 // Page size extension
0990
0991 #define SEG_KCODE 1 // kernel code
0992 #define SEG_KDATA 2 // kernel data+stack
0993 #define SEG_KCPU 3 // kernel per-cpu data
0994 #define SEG_UCODE 4 // user code
0995 #define SEG_UDATA 5 // user data+stack
0996 #define SEG_TSS 6 // this process's task state
0997
0998
0999

```

```

1000 #ifndef __ASSEMBLER__
1001 // Segment Descriptor
1002 struct segdesc {
1003     uint lim_15_0 : 16; // Low bits of segment limit
1004     uint base_15_0 : 16; // Low bits of segment base address
1005     uint base_23_16 : 8; // Middle bits of segment base address
1006     uint type : 4; // Segment type (see STS_constants)
1007     uint s : 1; // 0 = system, 1 = application
1008     uint dpl : 2; // Descriptor Privilege Level
1009     uint p : 1; // Present
1010     uint lim_19_16 : 4; // High bits of segment limit
1011     uint avl : 1; // Unused (available for software use)
1012     uint rsvl : 1; // Reserved
1013     uint db : 1; // 0 = 16-bit segment, 1 = 32-bit segment
1014     uint g : 1; // Granularity: limit scaled by 4K when set
1015     uint base_31_24 : 8; // High bits of segment base address
1016 };
1017
1018 // Normal segment
1019 #define SEG(type, base, lim, dpl) (struct segdesc) \
1020 { ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff, \
1021   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
1022   (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
1023 #define SEG16(type, base, lim, dpl) (struct segdesc) \
1024 { (lim) & 0xffff, (uint)(base) & 0xffff, \
1025   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
1026   (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
1027 #endif
1028
1029 #define DPL_USER 0x3 // User DPL
1030
1031 // Application segment type bits
1032 #define STA_X 0x8 // Executable segment
1033 #define STA_E 0x4 // Expand down (non-executable segments)
1034 #define STA_C 0x4 // Conforming code segment (executable only)
1035 #define STA_W 0x2 // Writeable (non-executable segments)
1036 #define STA_R 0x2 // Readable (executable segments)
1037 #define STA_A 0x1 // Accessed
1038
1039 // System segment type bits
1040 #define STS_T16A 0x1 // Available 16-bit TSS
1041 #define STS_LDT 0x2 // Local Descriptor Table
1042 #define STS_T16B 0x3 // Busy 16-bit TSS
1043 #define STS_CG16 0x4 // 16-bit Call Gate
1044 #define STS_TG 0x5 // Task Gate / Coum Transmissions
1045 #define STS_IG16 0x6 // 16-bit Interrupt Gate
1046 #define STS_TG16 0x7 // 16-bit Trap Gate
1047 #define STS_T32A 0x9 // Available 32-bit TSS
1048 #define STS_T32B 0xB // Busy 32-bit TSS
1049 #define STS_CG32 0xC // 32-bit Call Gate

```

```

1050 #define STS_IG32 0xE // 32-bit Interrupt Gate
1051 #define STS_TG32 0xF // 32-bit Trap Gate
1052
1053 // A virtual address 'la' has a three-part structure as follows:
1054 //
1055 // +-----10-----+-----10-----+-----12-----+
1056 // | Page Directory | Page Table | Offset within Page |
1057 // | Index         | Index       |                       |
1058 // +-----+-----+-----+
1059 // \--- PDX(va) --/ \--- PTX(va) --/
1060
1061 // page directory index
1062 #define PDX(va) (((uint)(va) >> PDXSHIFT) & 0x3FF)
1063
1064 // page table index
1065 #define PTX(va) (((uint)(va) >> PTXSHIFT) & 0x3FF)
1066
1067 // construct virtual address from indexes and offset
1068 #define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
1069
1070 // Page directory and page table constants.
1071 #define NPENTRIES 1024 // # directory entries per page directory
1072 #define NPTENTRIES 1024 // # PTEs per page table
1073 #define PGSIZE 4096 // bytes mapped by a page
1074
1075 #define PGSHIFT 12 // log2(PGSIZE)
1076 #define PTXSHIFT 12 // offset of PTX in a linear address
1077 #define PDXSHIFT 22 // offset of PDX in a linear address
1078
1079 #define PGROUNDUP(sz) (((sz)+PGSIZE-1) & ~(PGSIZE-1))
1080 #define PGROUNDDOWN(a) (((a) & ~(PGSIZE-1))
1081
1082 // Page table/directory entry flags.
1083 #define PTE_P 0x001 // Present
1084 #define PTE_W 0x002 // Writeable
1085 #define PTE_U 0x004 // User
1086 #define PTE_PWT 0x008 // Write-Through
1087 #define PTE_PCD 0x010 // Cache-Disable
1088 #define PTE_A 0x020 // Accessed
1089 #define PTE_D 0x040 // Dirty
1090 #define PTE_PS 0x080 // Page Size
1091 #define PTE_MBZ 0x180 // Bits must be zero
1092
1093 // Address in page table or page directory entry
1094 #define PTE_ADDR(pte) ((uint)(pte) & ~0xFFF)
1095 #define PTE_FLAGS(pte) ((uint)(pte) & 0xFFF)
1096
1097 #ifndef __ASSEMBLER__
1098 typedef uint pte_t;
1099

```

```

1100 // Task state segment format
1101 struct taskstate {
1102     uint link;           // Old ts selector
1103     uint esp0;           // Stack pointers and segment selectors
1104     ushort ss0;          // after an increase in privilege level
1105     ushort padding1;
1106     uint *esp1;
1107     ushort ss1;
1108     ushort padding2;
1109     uint *esp2;
1110     ushort ss2;
1111     ushort padding3;
1112     void *cr3;           // Page directory base
1113     uint *eip;           // Saved state from last task switch
1114     uint eflags;
1115     uint eax;            // More saved state (registers)
1116     uint ecx;
1117     uint edx;
1118     uint ebx;
1119     uint *esp;
1120     uint *ebp;
1121     uint esi;
1122     uint edi;
1123     ushort es;           // Even more saved state (segment selectors)
1124     ushort padding4;
1125     ushort cs;
1126     ushort padding5;
1127     ushort ss;
1128     ushort padding6;
1129     ushort ds;
1130     ushort padding7;
1131     ushort fs;
1132     ushort padding8;
1133     ushort gs;
1134     ushort padding9;
1135     ushort ldt;
1136     ushort padding10;
1137     ushort t;            // Trap on task switch
1138     ushort iomb;         // I/O map base address
1139 };
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149

```

```

1150 // Gate descriptors for interrupts and traps
1151 struct gatedesc {
1152     uint off_15_0 : 16;  // low 16 bits of offset in segment
1153     uint cs : 16;         // code segment selector
1154     uint args : 5;       // # args, 0 for interrupt/trap gates
1155     uint rsv1 : 3;       // reserved(should be zero I guess)
1156     uint type : 4;       // type(STS_{TG,IG32,TG32})
1157     uint s : 1;         // must be 0 (system)
1158     uint dpl : 2;       // descriptor(meaning new) privilege level
1159     uint p : 1;         // Present
1160     uint off_31_16 : 16; // high bits of offset in segment
1161 };
1162
1163 // Set up a normal interrupt/trap gate descriptor.
1164 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
1165 // - interrupt gate clears FL_IF, trap gate leaves FL_IF alone
1166 // - sel: Code segment selector for interrupt/trap handler
1167 // - off: Offset in code segment for interrupt/trap handler
1168 // - dpl: Descriptor Privilege Level -
1169 //       the privilege level required for software to invoke
1170 //       this interrupt/trap gate explicitly using an int instruction.
1171 #define SETGATE(gate, istrap, sel, off, d) \
1172 { \
1173     (gate).off_15_0 = (uint)(off) & 0xffff; \
1174     (gate).cs = (sel); \
1175     (gate).args = 0; \
1176     (gate).rsv1 = 0; \
1177     (gate).type = (istrap) ? STS_TG32 : STS_IG32; \
1178     (gate).s = 0; \
1179     (gate).dpl = (d); \
1180     (gate).p = 1; \
1181     (gate).off_31_16 = (uint)(off) >> 16; \
1182 } \
1183
1184 #endif
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199

```

```

1200 // Format of an ELF executable file
1201
1202 #define ELF_MAGIC 0x464C457FU // "\x7FELF" in little endian
1203
1204 // File header
1205 struct elfhdr {
1206     uint magic; // must equal ELF_MAGIC
1207     uchar elf[12];
1208     ushort type;
1209     ushort machine;
1210     uint version;
1211     uint entry;
1212     uint phoff;
1213     uint shoff;
1214     uint flags;
1215     ushort ehsize;
1216     ushort phentsize;
1217     ushort phnum;
1218     ushort shentsize;
1219     ushort shnum;
1220     ushort shstrndx;
1221 };
1222
1223 // Program section header
1224 struct proghdr {
1225     uint type;
1226     uint off;
1227     uint vaddr;
1228     uint paddr;
1229     uint filesz;
1230     uint memsz;
1231     uint flags;
1232     uint align;
1233 };
1234
1235 // Values for Proghdr type
1236 #define ELF_PROG_LOAD 1
1237
1238 // Flag bits for Proghdr flags
1239 #define ELF_PROG_FLAG_EXEC 1
1240 #define ELF_PROG_FLAG_WRITE 2
1241 #define ELF_PROG_FLAG_READ 4
1242
1243
1244
1245
1246
1247
1248
1249

```

```

1250 # Multiboot header, for multiboot boot loaders like GNU Grub.
1251 # http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
1252 #
1253 # Using GRUB 2, you can boot xv6 from a file stored in a
1254 # Linux file system by copying kernel or kernelmemfs to /boot
1255 # and then adding this menu entry:
1256 #
1257 # menuentry "xv6" {
1258 #     insmod ext2
1259 #     set root='(hd0,msdos1)'
1260 #     set kernel='/boot/kernel'
1261 #     echo "Loading ${kernel}..."
1262 #     multiboot ${kernel} ${kernel}
1263 #     boot
1264 # }
1265
1266 #include "asm.h"
1267 #include "memlayout.h"
1268 #include "mmu.h"
1269 #include "param.h"
1270
1271 # Multiboot header. Data to direct multiboot loader.
1272 .p2align 2
1273 .text
1274 .globl multiboot_header
1275 multiboot_header:
1276     #define magic 0x1badb002
1277     #define flags 0
1278     .long magic
1279     .long flags
1280     .long (-magic-flags)
1281
1282 # By convention, the _start symbol specifies the ELF entry point.
1283 # Since we haven't set up virtual memory yet, our entry point is
1284 # the physical address of 'entry'.
1285 .globl _start
1286 _start = V2P_WO(entry)
1287
1288 # Entering xv6 on boot processor, with paging off.
1289 .globl entry
1290 entry:
1291     # Turn on page size extension for 4Mbyte pages
1292     movl    %cr4, %eax
1293     orl     $(CR4_PSE), %eax
1294     movl    %eax, %cr4
1295     # Set page directory
1296     movl    $(V2P_WO(entrypgdir)), %eax
1297     movl    %eax, %cr3
1298     # Turn on paging.
1299     movl    %cr0, %eax

```

```

1300 orl    $(CR0_PG|CR0_WP), %eax
1301 movl   %eax, %cr0
1302
1303 # Set up the stack pointer.
1304 movl $(stack + KSTACKSIZE), %esp
1305
1306 # Jump to main(), and switch to executing at
1307 # high addresses. The indirect call is needed because
1308 # the assembler produces a PC-relative instruction
1309 # for a direct jump.
1310 mov $main, %eax
1311 jmp *%eax
1312
1313 .comm stack, KSTACKSIZE
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349

```

```

1350 #include "asm.h"
1351 #include "memlayout.h"
1352 #include "mmu.h"
1353
1354 # Each non-boot CPU ("AP") is started up in response to a STARTUP
1355 # IPI from the boot CPU. Section B.4.2 of the Multi-Processor
1356 # Specification says that the AP will start in real mode with CS:IP
1357 # set to XY00:0000, where XY is an 8-bit value sent with the
1358 # STARTUP. Thus this code must start at a 4096-byte boundary.
1359 #
1360 # Because this code sets DS to zero, it must sit
1361 # at an address in the low 2^16 bytes.
1362 #
1363 # Startothers (in main.c) sends the STARTUPs one at a time.
1364 # It copies this code (start) at 0x7000. It puts the address of
1365 # a newly allocated per-core stack in start-4, the address of the
1366 # place to jump to (mpenter) in start-8, and the physical address
1367 # of entrypgdir in start-12.
1368 #
1369 # This code is identical to bootasm.S except:
1370 #   - it does not need to enable A20
1371 #   - it uses the address at start-4, start-8, and start-12
1372
1373 .code16
1374 .globl start
1375 start:
1376 cli
1377
1378 xorw    %ax,%ax
1379 movw    %ax,%ds
1380 movw    %ax,%es
1381 movw    %ax,%ss
1382
1383 lgdt    gdtDESC
1384 movl    %cr0, %eax
1385 orl     $CR0_PE, %eax
1386 movl    %eax, %cr0
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399

```

```

1400  ljmp1    $(SEG_KCODE<<3), $(start32)
1401
1402  .code32
1403  start32:
1404  movw     $(SEG_KDATA<<3), %ax
1405  movw     %ax, %ds
1406  movw     %ax, %es
1407  movw     %ax, %ss
1408  movw     $0, %ax
1409  movw     %ax, %fs
1410  movw     %ax, %gs
1411
1412  # Turn on page size extension for 4Mbyte pages
1413  movl     %cr4, %eax
1414  orl      $(CR4_PSE), %eax
1415  movl     %eax, %cr4
1416  # Use enterpgdir as our initial page table
1417  movl     (start-12), %eax
1418  movl     %eax, %cr3
1419  # Turn on paging.
1420  movl     %cr0, %eax
1421  orl      $(CR0_PE|CR0_PG|CR0_WP), %eax
1422  movl     %eax, %cr0
1423
1424  # Switch to the stack allocated by startothers()
1425  movl     (start-4), %esp
1426  # Call mpenter()
1427  call     *(start-8)
1428
1429  movw     $0x8a00, %ax
1430  movw     %ax, %dx
1431  outw     %ax, %dx
1432  movw     $0x8ae0, %ax
1433  outw     %ax, %dx
1434  spin:
1435  jmp      spin
1436
1437  .p2align 2
1438  gdt:
1439  SEG_NULLASM
1440  SEG_ASM(STA_X|STA_R, 0, 0xffffffff)
1441  SEG_ASM(STA_W, 0, 0xffffffff)
1442
1443
1444  gdtdesc:
1445  .word    (gdtdesc - gdt - 1)
1446  .long    gdt
1447
1448
1449

```

```

1450  #include "types.h"
1451  #include "defs.h"
1452  #include "param.h"
1453  #include "memlayout.h"
1454  #include "mmu.h"
1455  #include "proc.h"
1456  #include "x86.h"
1457
1458  static void startothers(void);
1459  static void mpmain(void) __attribute__((noreturn));
1460  extern pde_t *kpgdir;
1461  extern char end[]; // first address after kernel loaded from ELF file
1462
1463  // Bootstrap processor starts running C code here.
1464  // Allocate a real stack and switch to it, first
1465  // doing some setup required for memory allocator to work.
1466  int
1467  main(void)
1468  {
1469    kinit1(end, P2V(4*1024*1024)); // phys page allocator
1470    kvmalloc(); // kernel page table
1471    mpinit(); // collect info about this machine
1472    lapicinit();
1473    seginit(); // set up segments
1474    cprintf("ncpu%d: starting xv6\n\n", cpu->id);
1475    picinit(); // interrupt controller
1476    ioapicinit(); // another interrupt controller
1477    consoleinit(); // I/O devices & their interrupts
1478    uartinit(); // serial port
1479    pinit(); // process table
1480    tvinit(); // trap vectors
1481    binit(); // buffer cache
1482    fileinit(); // file table
1483    ideinit(); // disk
1484    if(!ismp)
1485        timerinit(); // uniprocessor timer
1486    startothers(); // start other processors
1487    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1488    userinit(); // first user process
1489    // Finish setting up this processor in mpmain.
1490    mpmain();
1491  }
1492
1493
1494
1495
1496
1497
1498
1499

```

```

1500 // Other CPUs jump here from entryother.S.
1501 static void
1502 mpenter(void)
1503 {
1504     switchkvm();
1505     seginit();
1506     lapicinit();
1507     mpmain();
1508 }
1509
1510 // Common CPU setup code.
1511 static void
1512 mpmain(void)
1513 {
1514     cprintf("cpu%d: starting\n", cpu->id);
1515     idtinit(); // load idt register
1516     xchg(&cpu->started, 1); // tell startothers() we're up
1517     scheduler(); // start running processes
1518 }
1519
1520 pde_t entrypgdir[]; // For entry.S
1521
1522 // Start the non-boot (AP) processors.
1523 static void
1524 startothers(void)
1525 {
1526     extern uchar _binary_entryother_start[], _binary_entryother_size[];
1527     uchar *code;
1528     struct cpu *c;
1529     char *stack;
1530
1531     // Write entry code to unused memory at 0x7000.
1532     // The linker has placed the image of entryother.S in
1533     // _binary_entryother_start.
1534     code = p2v(0x7000);
1535     memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
1536
1537     for(c = cpus; c < cpus+ncpu; c++){
1538         if(c == cpus+cpunum()) // We've started already.
1539             continue;
1540
1541         // Tell entryother.S what stack to use, where to enter, and what
1542         // pgdir to use. We cannot use kpgdir yet, because the AP processor
1543         // is running in low memory, so we use entrypgdir for the APs too.
1544         stack = kalloc();
1545         *(void**)(code-4) = stack + KSTACKSIZE;
1546         *(void**)(code-8) = mpenter;
1547         *(int**)(code-12) = (void *) v2p(entrypgdir);
1548
1549         lapicstartap(c->id, v2p(code));

```

```

1550     // wait for cpu to finish mpmain()
1551     while(c->started == 0)
1552         ;
1553 }
1554 }
1555
1556 // Boot page table used in entry.S and entryother.S.
1557 // Page directories (and page tables), must start on a page boundary,
1558 // hence the "__aligned__" attribute.
1559 // Use PTE_PS in page directory entry to enable 4Mbyte pages.
1560 __attribute__((__aligned__(PGSIZE)))
1561 pde_t entrypgdir[NPDENTRIES] = {
1562     // Map VA's [0, 4MB) to PA's [0, 4MB)
1563     [0] = (0) | PTE_P | PTE_W | PTE_PS,
1564     // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1565     [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1566 };
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599

```

```
1600 // Blank page.  
1601  
1602  
1603  
1604  
1605  
1606  
1607  
1608  
1609  
1610  
1611  
1612  
1613  
1614  
1615  
1616  
1617  
1618  
1619  
1620  
1621  
1622  
1623  
1624  
1625  
1626  
1627  
1628  
1629  
1630  
1631  
1632  
1633  
1634  
1635  
1636  
1637  
1638  
1639  
1640  
1641  
1642  
1643  
1644  
1645  
1646  
1647  
1648  
1649
```

```
1650 // Blank page.  
1651  
1652  
1653  
1654  
1655  
1656  
1657  
1658  
1659  
1660  
1661  
1662  
1663  
1664  
1665  
1666  
1667  
1668  
1669  
1670  
1671  
1672  
1673  
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 // Blank page.
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
```

```
1750 // Mutual exclusion lock.
1751 struct spinlock {
1752     uint locked;      // Is the lock held?
1753
1754     // For debugging:
1755     char *name;        // Name of lock.
1756     struct cpu *cpu;   // The cpu holding the lock.
1757     uint pcs[10];      // The call stack (an array of program counters)
1758                       // that locked the lock.
1759 };
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
```

```

1800 // Mutual exclusion spin locks.
1801
1802 #include "types.h"
1803 #include "defs.h"
1804 #include "param.h"
1805 #include "x86.h"
1806 #include "memlayout.h"
1807 #include "mmu.h"
1808 #include "proc.h"
1809 #include "spinlock.h"
1810
1811 void
1812 initlock(struct spinlock *lk, char *name)
1813 {
1814     lk->name = name;
1815     lk->locked = 0;
1816     lk->cpu = 0;
1817 }
1818
1819 // Acquire the lock.
1820 // Loops (spins) until the lock is acquired.
1821 // Holding a lock for a long time may cause
1822 // other CPUs to waste time spinning to acquire it.
1823 void
1824 acquire(struct spinlock *lk)
1825 {
1826     pushcli(); // disable interrupts to avoid deadlock.
1827     if(holding(lk))
1828         panic("acquire");
1829
1830     // The xchg is atomic.
1831     // It also serializes, so that reads after acquire are not
1832     // reordered before it.
1833     while(xchg(&lk->locked, 1) != 0)
1834         ;
1835
1836     // Record info about lock acquisition for debugging.
1837     lk->cpu = cpu;
1838     getcallerpcs(&lk, lk->pcs);
1839 }
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849

```

```

1850 // Release the lock.
1851 void
1852 release(struct spinlock *lk)
1853 {
1854     if(!holding(lk))
1855         panic("release");
1856
1857     lk->pcs[0] = 0;
1858     lk->cpu = 0;
1859
1860     // The xchg serializes, so that reads before release are
1861     // not reordered after it. The 1996 PentiumPro manual (Volume 3,
1862     // 7.2) says reads can be carried out speculatively and in
1863     // any order, which implies we need to serialize here.
1864     // But the 2007 Intel 64 Architecture Memory Ordering White
1865     // Paper says that Intel 64 and IA-32 will not move a load
1866     // after a store. So lock->locked = 0 would work here.
1867     // The xchg being asm volatile ensures gcc emits it after
1868     // the above assignments (and after the critical section).
1869     xchg(&lk->locked, 0);
1870
1871     popcli();
1872 }
1873
1874 // Record the current call stack in pcs[] by following the %ebp chain.
1875 void
1876 getcallerpcs(void *v, uint pcs[])
1877 {
1878     uint *ebp;
1879     int i;
1880
1881     ebp = (uint*)v - 2;
1882     for(i = 0; i < 10; i++){
1883         if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
1884             break;
1885         pcs[i] = ebp[1]; // saved %eip
1886         ebp = (uint*)ebp[0]; // saved %ebp
1887     }
1888     for(; i < 10; i++)
1889         pcs[i] = 0;
1890 }
1891
1892 // Check whether this cpu is holding the lock.
1893 int
1894 holding(struct spinlock *lock)
1895 {
1896     return lock->locked && lock->cpu == cpu;
1897 }
1898
1899

```

```

1900 // Pushcli/popcli are like cli/sti except that they are matched:
1901 // it takes two popcli to undo two pushcli. Also, if interrupts
1902 // are off, then pushcli, popcli leaves them off.
1903
1904 void
1905 pushcli(void)
1906 {
1907     int eflags;
1908
1909     eflags = readeflags();
1910     cli();
1911     if(cpu->ncli++ == 0)
1912         cpu->intena = eflags & FL_IF;
1913 }
1914
1915 void
1916 popcli(void)
1917 {
1918     if(readeflags() & FL_IF)
1919         panic("popcli - interruptible");
1920     if(--cpu->ncli < 0)
1921         panic("popcli");
1922     if(cpu->ncli == 0 && cpu->intena)
1923         sti();
1924 }
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949

```

```

1950 #include "param.h"
1951 #include "types.h"
1952 #include "defs.h"
1953 #include "x86.h"
1954 #include "memlayout.h"
1955 #include "mmu.h"
1956 #include "proc.h"
1957 #include "elf.h"
1958
1959 extern char data[]; // defined by kernel.ld
1960 pde_t *kpgdir; // for use in scheduler()
1961 struct segdesc gdt[NSEGs];
1962
1963 // Set up CPU's kernel segment descriptors.
1964 // Run once on entry on each CPU.
1965 void
1966 seginit(void)
1967 {
1968     struct cpu *c;
1969
1970     // Map "logical" addresses to virtual addresses using identity map.
1971     // Cannot share a CODE descriptor for both kernel and user
1972     // because it would have to have DPL_USR, but the CPU forbids
1973     // an interrupt from CPL=0 to DPL=3.
1974     c = &cpus[cpunum()];
1975     c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
1976     c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1977     c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
1978     c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1979
1980     // Map cpu, and curproc
1981     c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
1982
1983     lgdt(c->gdt, sizeof(c->gdt));
1984     loadgs(SEG_KCPU << 3);
1985
1986     // Initialize cpu-local storage.
1987     cpu = c;
1988     proc = 0;
1989 }
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999

```

```

2000 // Return the address of the PTE in page table pgdir
2001 // that corresponds to virtual address va. If alloc!=0,
2002 // create any required page table pages.
2003 static pte_t *
2004 walkpgdir(pte_t *pgdir, const void *va, int alloc)
2005 {
2006     pde_t *pde;
2007     pte_t *pgtab;
2008
2009     pde = &pgdir[PDX(va)];
2010     if(*pde & PTE_P){
2011         pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
2012     } else {
2013         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
2014             return 0;
2015         // Make sure all those PTE_P bits are zero.
2016         memset(pgtab, 0, PGSIZE);
2017         // The permissions here are overly generous, but they can
2018         // be further restricted by the permissions in the page table
2019         // entries, if necessary.
2020         *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
2021     }
2022     return &pgtab[PTX(va)];
2023 }
2024
2025 // Create PTEs for virtual addresses starting at va that refer to
2026 // physical addresses starting at pa. va and size might not
2027 // be page-aligned.
2028 static int
2029 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
2030 {
2031     char *a, *last;
2032     pte_t *pte;
2033
2034     a = (char*)PGROUNDDOWN((uint)va);
2035     last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
2036     for(;;){
2037         if((pte = walkpgdir(pgdir, a, 1)) == 0)
2038             return -1;
2039         if(*pte & PTE_P)
2040             panic("remap");
2041         *pte = pa | perm | PTE_P;
2042         if(a == last)
2043             break;
2044         a += PGSIZE;
2045         pa += PGSIZE;
2046     }
2047     return 0;
2048 }
2049

```

```

2050 // There is one page table per process, plus one that's used when
2051 // a CPU is not running any process (kpgdir). The kernel uses the
2052 // current process's page table during system calls and interrupts;
2053 // page protection bits prevent user code from using the kernel's
2054 // mappings.
2055 //
2056 // setupkvm() and exec() set up every page table like this:
2057 //
2058 // 0..KERNBASE: user memory (text+data+stack+heap), mapped to
2059 // phys memory allocated by the kernel
2060 // KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
2061 // KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
2062 // for the kernel's instructions and r/o data
2063 // data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
2064 // rw data + free physical memory
2065 // 0xfe000000..0: mapped direct (devices such as ioapic)
2066 //
2067 // The kernel allocates physical memory for its heap and for user memory
2068 // between V2P(end) and the end of physical memory (PHYSTOP)
2069 // (directly addressable from end..P2V(PHYSTOP)).
2070
2071 // This table defines the kernel's mappings, which are present in
2072 // every process's page table.
2073 static struct kmap {
2074     void *virt;
2075     uint phys_start;
2076     uint phys_end;
2077     int perm;
2078 } kmap[] = {
2079     { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
2080     { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},    // kern text+rodata
2081     { (void*)data,     V2P(data),     PHYSTOP,    PTE_W}, // kern data+memory
2082     { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
2083 };
2084
2085 // Set up kernel part of a page table.
2086 pde_t *
2087 setupkvm(void)
2088 {
2089     pde_t *pgdir;
2090     struct kmap *k;
2091
2092     if((pgdir = (pde_t*)kalloc()) == 0)
2093         return 0;
2094     memset(pgdir, 0, PGSIZE);
2095     if (p2v(PHYSTOP) > (void*)DEVSPACE)
2096         panic("PHYSTOP too high");
2097     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
2098         if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
2099                     (uint)k->phys_start, k->perm) < 0)

```

```

2100     return 0;
2101     return pgdir;
2102 }
2103
2104 // Allocate one page table for the machine for the kernel address
2105 // space for scheduler processes.
2106 void
2107 kvmalloc(void)
2108 {
2109     kpgdir = setupkvm();
2110     switchkvm();
2111 }
2112
2113 // Switch h/w page table register to the kernel-only page table,
2114 // for when no process is running.
2115 void
2116 switchkvm(void)
2117 {
2118     lcr3(v2p(kpgdir)); // switch to the kernel page table
2119 }
2120
2121 // Switch TSS and h/w page table to correspond to process p.
2122 void
2123 switchvm(struct proc *p)
2124 {
2125     pushcli();
2126     cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
2127     cpu->gdt[SEG_TSS].s = 0;
2128     cpu->ts.ss0 = SEG_KDATA << 3;
2129     cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
2130     ltr(SEG_TSS << 3);
2131     if(p->pgdir == 0)
2132         panic("switchvm: no pgdir");
2133     lcr3(v2p(p->pgdir)); // switch to new address space
2134     popcli();
2135 }
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149

```

```

2150 // Load the initcode into address 0 of pgdir.
2151 // sz must be less than a page.
2152 void
2153 initvm(pde_t *pgdir, char *init, uint sz)
2154 {
2155     char *mem;
2156
2157     if(sz >= PGSIZE)
2158         panic("initvm: more than a page");
2159     mem = kalloc();
2160     memset(mem, 0, PGSIZE);
2161     mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
2162     memmove(mem, init, sz);
2163 }
2164
2165 // Load a program segment into pgdir. addr must be page-aligned
2166 // and the pages from addr to addr+sz must already be mapped.
2167 int
2168 loadvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
2169 {
2170     uint i, pa, n;
2171     pte_t *pte;
2172
2173     if((uint) addr % PGSIZE != 0)
2174         panic("loadvm: addr must be page aligned");
2175     for(i = 0; i < sz; i += PGSIZE){
2176         if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
2177             panic("loadvm: address should exist");
2178         pa = PTE_ADDR(*pte);
2179         if(sz - i < PGSIZE)
2180             n = sz - i;
2181         else
2182             n = PGSIZE;
2183         if(readi(ip, p2v(pa), offset+i, n) != n)
2184             return -1;
2185     }
2186     return 0;
2187 }
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199

```

```

2200 // Allocate page tables and physical memory to grow process from oldsz to
2201 // newsz, which need not be page aligned. Returns new size or 0 on error.
2202 int
2203 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
2204 {
2205     char *mem;
2206     uint a;
2207
2208     if(newsz >= KERNBASE)
2209         return 0;
2210     if(newsz < oldsz)
2211         return oldsz;
2212
2213     a = PGROUNDUP(oldsz);
2214     for(; a < newsz; a += PGSIZE){
2215         mem = kalloc();
2216         if(mem == 0){
2217             cprintf("allocuvm out of memory\n");
2218             deallocuvm(pgdir, newsz, oldsz);
2219             return 0;
2220         }
2221         memset(mem, 0, PGSIZE);
2222         mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U);
2223     }
2224     return newsz;
2225 }
2226
2227 // Deallocate user pages to bring the process size from oldsz to
2228 // newsz. oldsz and newsz need not be page-aligned, nor does newsz
2229 // need to be less than oldsz. oldsz can be larger than the actual
2230 // process size. Returns the new process size.
2231 int
2232 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
2233 {
2234     pte_t *pte;
2235     uint a, pa;
2236
2237     if(newsz >= oldsz)
2238         return oldsz;
2239
2240     a = PGROUNDUP(newsz);
2241     for(; a < oldsz; a += PGSIZE){
2242         pte = walkpgdir(pgdir, (char*)a, 0);
2243         if(!pte)
2244             a += (NPENTRIES - 1) * PGSIZE;
2245         else if((*pte & PTE_P) != 0){
2246             pa = PTE_ADDR(*pte);
2247             if(pa == 0)
2248                 panic("kfree");
2249             char *v = p2v(pa);

```

```

2250         kfree(v);
2251         *pte = 0;
2252     }
2253 }
2254 return newsz;
2255 }
2256
2257 // Free a page table and all the physical memory pages
2258 // in the user part.
2259 void
2260 freevm(pde_t *pgdir)
2261 {
2262     uint i;
2263
2264     if(pgdir == 0)
2265         panic("freevm: no pgdir");
2266     deallocuvm(pgdir, KERNBASE, 0);
2267     for(i = 0; i < NPENTRIES; i++){
2268         if(pgdir[i] & PTE_P){
2269             char *v = p2v(PTE_ADDR(pgdir[i]));
2270             kfree(v);
2271         }
2272     }
2273     kfree((char*)pgdir);
2274 }
2275
2276 // Clear PTE_U on a page. Used to create an inaccessible
2277 // page beneath the user stack.
2278 void
2279 clearpteu(pde_t *pgdir, char *uva)
2280 {
2281     pte_t *pte;
2282
2283     pte = walkpgdir(pgdir, uva, 0);
2284     if(pte == 0)
2285         panic("clearpteu");
2286     *pte &= ~PTE_U;
2287 }
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299

```

```

2300 // Given a parent process's page table, create a copy
2301 // of it for a child.
2302 pde_t*
2303 copyuvm(pde_t *pgdir, uint sz)
2304 {
2305     pde_t *d;
2306     pte_t *pte;
2307     uint pa, i, flags;
2308     char *mem;
2309
2310     if((d = setupkvm()) == 0)
2311         return 0;
2312     for(i = 0; i < sz; i += PGSIZE){
2313         if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
2314             panic("copyuvm: pte should exist");
2315         if(!(*pte & PTE_P))
2316             panic("copyuvm: page not present");
2317         pa = PTE_ADDR(*pte);
2318         flags = PTE_FLAGS(*pte);
2319         if((mem = kalloc()) == 0)
2320             goto bad;
2321         memmove(mem, (char*)p2v(pa), PGSIZE);
2322         if(mappages(d, (void*)i, PGSIZE, v2p(mem), flags) < 0)
2323             goto bad;
2324     }
2325     return d;
2326
2327 bad:
2328     freevm(d);
2329     return 0;
2330 }
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349

```

```

2350 // Map user virtual address to kernel address.
2351 char*
2352 uva2ka(pde_t *pgdir, char *uva)
2353 {
2354     pte_t *pte;
2355
2356     pte = walkpgdir(pgdir, uva, 0);
2357     if((*pte & PTE_P) == 0)
2358         return 0;
2359     if((*pte & PTE_U) == 0)
2360         return 0;
2361     return (char*)p2v(PTE_ADDR(*pte));
2362 }
2363
2364 // Copy len bytes from p to user address va in page table pgdir.
2365 // Most useful when pgdir is not the current page table.
2366 // uva2ka ensures this only works for PTE_U pages.
2367 int
2368 copyout(pde_t *pgdir, uint va, void *p, uint len)
2369 {
2370     char *buf, *pa0;
2371     uint n, va0;
2372
2373     buf = (char*)p;
2374     while(len > 0){
2375         va0 = (uint)PGROUNDDOWN(va);
2376         pa0 = uva2ka(pgdir, (char*)va0);
2377         if(pa0 == 0)
2378             return -1;
2379         n = PGSIZE - (va - va0);
2380         if(n > len)
2381             n = len;
2382         memmove(pa0 + (va - va0), buf, n);
2383         len -= n;
2384         buf += n;
2385         va = va0 + PGSIZE;
2386     }
2387     return 0;
2388 }
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399

```

2400 // Blank page.

2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
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 // Blank page.

2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499


```

2500 // Blank page.
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549

```

```

2550 // Segments in proc->gdt.
2551 #include "BC_ps.h"
2552 #define NSEGS      7
2553 #define FIRST_PROC_GID  0
2554 #define FIRST_PROC_UID  0
2555
2556 int get_current_procs(int, struct uproc* table); //sys_getprocs helper proto
2557 // Per-CPU state
2558 struct cpu {
2559     uchar id;                    // Local APIC ID; index into cpus[] below
2560     struct context *scheduler;   // switch() here to enter scheduler
2561     struct taskstate ts;         // Used by x86 to find stack for interrupt
2562     struct segdesc gdt[NSEGS];   // x86 global descriptor table
2563     volatile uint started;       // Has the CPU started?
2564     int ncli;                    // Depth of pushcli nesting.
2565     int intena;                  // Were interrupts enabled before pushcli?
2566
2567     // Cpu-local storage variables; see below
2568     struct cpu *cpu;
2569     struct proc *proc;           // The currently-running process.
2570 };
2571
2572 extern struct cpu cpus[NCPU];
2573 extern int ncpu;
2574
2575 // Per-CPU variables, holding pointers to the
2576 // current cpu and to the current process.
2577 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2578 // and "%gs:4" to refer to proc. seginit sets up the
2579 // %gs segment register so that %gs refers to the memory
2580 // holding those two variables in the local cpu's struct cpu.
2581 // This is similar to how thread-local variables are implemented
2582 // in thread libraries such as Linux pthreads.
2583 extern struct cpu *cpu asm("%gs:0"); // &cpus[cpunum()]
2584 extern struct proc *proc asm("%gs:4"); // cpus[cpunum()].proc
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599

```

```

2600 // Saved registers for kernel context switches.
2601 // Don't need to save all the segment registers (%cs, etc),
2602 // because they are constant across kernel contexts.
2603 // Don't need to save %eax, %ecx, %edx, because the
2604 // x86 convention is that the caller has saved them.
2605 // Contexts are stored at the bottom of the stack they
2606 // describe; the stack pointer is the address of the context.
2607 // The layout of the context matches the layout of the stack in swtch.S
2608 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2609 // but it is on the stack and allocproc() manipulates it.
2610 struct context {
2611     uint edi;
2612     uint esi;
2613     uint ebx;
2614     uint ebp;
2615     uint eip;
2616 };
2617
2618 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2619
2620 // Per-process state
2621 struct proc {
2622     uint sz;                // Size of process memory (bytes)
2623     pde_t* pgdir;          // Page table
2624     char *kstack;          // Bottom of kernel stack for this process
2625     enum procstate state;   // Process state
2626     int pid;               // Process ID
2627     int ppid;              // Parent Process ID
2628     int uid;               // User ID
2629     int gid;               // Group ID *****
2630     struct proc *parent;    // Parent process
2631     struct trapframe *tf;   // Trap frame for current syscall
2632     struct context *context; // swtch() here to run process
2633     void *chan;             // If non-zero, sleeping on chan
2634     int killed;             // If non-zero, have been killed
2635     struct file *ofile[NOFILE]; // Open files
2636     struct inode *cwd;      // Current directory
2637     char name[16];          // Process name (debugging)
2638 };
2639
2640 // Process memory is laid out contiguously, low addresses first:
2641 //   text
2642 //   original data and bss
2643 //   fixed-size stack
2644 //   expandable heap
2645
2646
2647
2648
2649

```

```

2650 #include "types.h"
2651 #include "defs.h"
2652 #include "param.h"
2653 #include "memlayout.h"
2654 #include "mmu.h"
2655 #include "x86.h"
2656 #include "proc.h"
2657 #include "spinlock.h"
2658 ;
2659
2660 struct {
2661     struct spinlock lock;
2662     struct proc proc[NPROC];
2663 } ptable;
2664
2665 static struct proc *initproc;
2666
2667 int nextpid = 1;
2668 extern void forkret(void);
2669 extern void trapret(void);
2670
2671 static void wakeup1(void *chan);
2672
2673 void
2674 pinit(void)
2675 {
2676     initlock(&ptable.lock, "ptable");
2677 }
2678
2679 //helper function for sys_getprocs
2680 int get_current_procs(int max, struct uproc* table)
2681 {
2682     struct proc *p;          //temp to get procs from table
2683     int i = 0;               //index for max
2684
2685     //To get state string from the proc struct
2686     static char *states[] = {
2687         [UNUSED]    "UNUSED",
2688         [EMBRYO]    "EMBRYO",
2689         [SLEEPING]  "SLEEP",
2690         [RUNNABLE]  "RUNABLE",
2691         [RUNNING]   "RUN ",
2692         [ZOMBIE]    "ZOMBIE"
2693     };
2694
2695     acquire(&ptable.lock); //acquire lock when reading from the Ptable to pre
2696
2697
2698
2699

```

```

2700 //run through the ptable and find all procs that are not marked unused.
2701 //stop if end if reached, or the max requested is used
2702 for(p = ptable.proc; p < &ptable.proc[NPROC] && i < max; p++) {
2703     if(p->state != UNUSED) {
2704
2705         table[i].uid = p->uid;
2706         table[i].gid = p->gid;
2707         table[i].pid = p->pid;
2708         table[i].ppid = p->ppid;
2709         table[i].size = p->sz;
2710         ++i;
2711
2712
2713         //strcpy for name
2714         unsigned k;
2715         for(k = 0; p->name[k] != '\0'; ++k) {
2716             table[i].name[k] = p->name[k];
2717         }
2718         table[i].name[k] = '\0';
2719
2720         //strcpy for state
2721         if(p->state >= 0 && p->state < NELEM(states) && states[p->state]) {
2722             char * state = states[p->state];
2723             unsigned j;
2724             for (j=0; state[j] != '\0'; ++j) {
2725                 table[i].state[j] = state[j];
2726             }
2727             table[i].state[j] = '\0';
2728         }
2729     }
2730 }
2731 release(&ptable.lock);
2732 return i;
2733 }
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749

```

```

2750 // Look in the process table for an UNUSED proc.
2751 // If found, change state to EMBRYO and initialize
2752 // state required to run in the kernel.
2753 // Otherwise return 0.
2754 static struct proc*
2755 allocproc(void)
2756 {
2757     struct proc *p;
2758     char *sp;
2759
2760     acquire(&ptable.lock);
2761     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2762         if(p->state == UNUSED)
2763             goto found;
2764     release(&ptable.lock);
2765     return 0;
2766
2767 found:
2768     p->state = EMBRYO;
2769     if(nextpid == 1) {
2770         p->ppid = nextpid;
2771         p->gid = FIRST_PROC_GID;
2772         p->uid = FIRST_PROC_UID;
2773     }
2774     else {
2775         p->ppid = proc->pid;
2776         p->gid = proc->gid;
2777         p->uid = proc->uid;
2778     }
2779
2780     p->pid = nextpid++;
2781     release(&ptable.lock);
2782
2783     // Allocate kernel stack.
2784     if((p->kstack = kalloc()) == 0){
2785         p->state = UNUSED;
2786         return 0;
2787     }
2788     sp = p->kstack + KSTACKSIZE;
2789
2790     // Leave room for trap frame.
2791     sp -= sizeof *p->tf;
2792     p->tf = (struct trapframe*)sp;
2793
2794     // Set up new context to start executing at forkret,
2795     // which returns to trapret.
2796     sp -= 4;
2797     *(uint*)sp = (uint)trapret;
2798
2799

```

```

2800 sp -= sizeof *p->context;
2801 p->context = (struct context*)sp;
2802 memset(p->context, 0, sizeof *p->context);
2803 p->context->eip = (uint)forkret;
2804
2805 return p;
2806 }
2807
2808
2809 // Set up first user process.
2810 void
2811 userinit(void)
2812 {
2813     struct proc *p;
2814     extern char _binary_initcode_start[], _binary_initcode_size[];
2815
2816     p = allocproc();
2817     initproc = p;
2818     if((p->pgdir = setupkvm()) == 0)
2819         panic("userinit: out of memory?");
2820     inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2821     p->sz = PGSIZE;
2822     memset(p->tf, 0, sizeof(*p->tf));
2823     p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2824     p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2825     p->tf->es = p->tf->ds;
2826     p->tf->ss = p->tf->ds;
2827     p->tf->eflags = FL_IF;
2828     p->tf->esp = PGSIZE;
2829     p->tf->eip = 0; // beginning of initcode.S
2830
2831     safestrcpy(p->name, "initcode", sizeof(p->name));
2832     p->cwd = namei("/");
2833
2834     p->state = RUNNABLE;
2835 }
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849

```

```

2850 // Grow current process's memory by n bytes.
2851 // Return 0 on success, -1 on failure.
2852 int
2853 growproc(int n)
2854 {
2855     uint sz;
2856
2857     sz = proc->sz;
2858     if(n > 0){
2859         if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2860             return -1;
2861     } else if(n < 0){
2862         if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2863             return -1;
2864     }
2865     proc->sz = sz;
2866     switchuvm(proc);
2867     return 0;
2868 }
2869
2870 // Create a new process copying p as the parent.
2871 // Sets up stack to return as if from system call.
2872 // Caller must set state of returned proc to RUNNABLE.
2873 int
2874 fork(void)
2875 {
2876     int i, pid;
2877     struct proc *np;
2878
2879     // Allocate process.
2880     if((np = allocproc()) == 0)
2881         return -1;
2882
2883     // Copy process state from p.
2884     if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2885         kfree(np->kstack);
2886         np->kstack = 0;
2887         np->state = UNUSED;
2888         return -1;
2889     }
2890     np->sz = proc->sz;
2891     np->parent = proc;
2892     *np->tf = *proc->tf;
2893
2894     // Clear %eax so that fork returns 0 in the child.
2895     np->tf->eax = 0;
2896
2897
2898
2899

```

```

2900 for(i = 0; i < NOFILE; i++)
2901     if(proc->ofile[i])
2902         np->ofile[i] = filedup(proc->ofile[i]);
2903 np->cwd = idup(proc->cwd);
2904
2905 safestrcpy(np->name, proc->name, sizeof(proc->name));
2906
2907 pid = np->pid;
2908
2909 // lock to force the compiler to emit the np->state write last.
2910 acquire(&ptable.lock);
2911 np->state = RUNNABLE;
2912 release(&ptable.lock);
2913
2914 return pid;
2915 }
2916
2917 // Exit the current process. Does not return.
2918 // An exited process remains in the zombie state
2919 // until its parent calls wait() to find out it exited.
2920 void
2921 exit(void)
2922 {
2923     struct proc *p;
2924     int fd;
2925
2926     if(proc == initproc)
2927         panic("init exiting");
2928
2929     // Close all open files.
2930     for(fd = 0; fd < NOFILE; fd++){
2931         if(proc->ofile[fd]){
2932             fileclose(proc->ofile[fd]);
2933             proc->ofile[fd] = 0;
2934         }
2935     }
2936
2937     begin_op();
2938     iput(proc->cwd);
2939     end_op();
2940     proc->cwd = 0;
2941
2942     acquire(&ptable.lock);
2943
2944     // Parent might be sleeping in wait().
2945     wakeup1(proc->parent);
2946
2947
2948
2949

```

```

2950 // Pass abandoned children to init.
2951 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2952     if(p->parent == proc){
2953         p->parent = initproc;
2954         if(p->state == ZOMBIE)
2955             wakeup1(initproc);
2956     }
2957 }
2958
2959 // Jump into the scheduler, never to return.
2960 proc->state = ZOMBIE;
2961 sched();
2962 panic("zombie exit");
2963 }
2964
2965 // Wait for a child process to exit and return its pid.
2966 // Return -1 if this process has no children.
2967 int
2968 wait(void)
2969 {
2970     struct proc *p;
2971     int havekids, pid;
2972
2973     acquire(&ptable.lock);
2974     for(;;){
2975         // Scan through table looking for zombie children.
2976         havekids = 0;
2977         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2978             if(p->parent != proc)
2979                 continue;
2980             havekids = 1;
2981             if(p->state == ZOMBIE){
2982                 // Found one.
2983                 pid = p->pid;
2984                 kfree(p->kstack);
2985                 p->kstack = 0;
2986                 freevm(p->pgdir);
2987                 p->state = UNUSED;
2988                 p->pid = 0;
2989                 p->parent = 0;
2990                 p->name[0] = 0;
2991                 p->killed = 0;
2992                 release(&ptable.lock);
2993                 return pid;
2994             }
2995         }
2996
2997
2998
2999

```

```

3000 // No point waiting if we don't have any children.
3001 if(!havekids || proc->killed){
3002     release(&ptable.lock);
3003     return -1;
3004 }
3005
3006 // Wait for children to exit. (See wakeup1 call in proc_exit.)
3007 sleep(proc, &ptable.lock);
3008 }
3009 }
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049

```

```

3050 // Per-CPU process scheduler.
3051 // Each CPU calls scheduler() after setting itself up.
3052 // Scheduler never returns. It loops, doing:
3053 // - choose a process to run
3054 // - switch to start running that process
3055 // - eventually that process transfers control
3056 //   via switch back to the scheduler.
3057 void
3058 scheduler(void)
3059 {
3060     struct proc *p;
3061
3062     for(;;){
3063         // Enable interrupts on this processor.
3064         sti();
3065
3066         // Loop over process table looking for process to run.
3067         acquire(&ptable.lock);
3068         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3069             if(p->state != RUNNABLE)
3070                 continue;
3071
3072             // Switch to chosen process. It is the process's job
3073             // to release ptable.lock and then reacquire it
3074             // before jumping back to us.
3075             proc = p;
3076             switchvm(p);
3077             p->state = RUNNING;
3078             switch(&cpu->scheduler, proc->context);
3079             switchkvm();
3080
3081             // Process is done running for now.
3082             // It should have changed its p->state before coming back.
3083             proc = 0;
3084         }
3085         release(&ptable.lock);
3086     }
3087 }
3088 }
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099

```

```

3100 // Enter scheduler. Must hold only ptable.lock
3101 // and have changed proc->state.
3102 void
3103 sched(void)
3104 {
3105     int intena;
3106
3107     if(!holding(&ptable.lock))
3108         panic("sched ptable.lock");
3109     if(cpu->ncli != 1)
3110         panic("sched locks");
3111     if(proc->state == RUNNING)
3112         panic("sched running");
3113     if(readeflags() & FL_IF)
3114         panic("sched interruptible");
3115     intena = cpu->intena;
3116     swtch(&proc->context, cpu->scheduler);
3117     cpu->intena = intena;
3118 }
3119
3120 // Give up the CPU for one scheduling round.
3121 void
3122 yield(void)
3123 {
3124     acquire(&ptable.lock);
3125     proc->state = RUNNABLE;
3126     sched();
3127     release(&ptable.lock);
3128 }
3129
3130 // A fork child's very first scheduling by scheduler()
3131 // will swtch here. "Return" to user space.
3132 void
3133 forkret(void)
3134 {
3135     static int first = 1;
3136     // Still holding ptable.lock from scheduler.
3137     release(&ptable.lock);
3138
3139     if (first) {
3140         // Some initialization functions must be run in the context
3141         // of a regular process (e.g., they call sleep), and thus cannot
3142         // be run from main().
3143         first = 0;
3144         iinit(ROOTDEV);
3145         initlog(ROOTDEV);
3146     }
3147
3148     // Return to "caller", actually trapret (see allocproc).
3149 }

```

```

3150 // Atomically release lock and sleep on chan.
3151 // Reacquires lock when awakened.
3152 void
3153 sleep(void *chan, struct spinlock *lk)
3154 {
3155     if(proc == 0)
3156         panic("sleep");
3157
3158     if(lk == 0)
3159         panic("sleep without lk");
3160
3161     // Must acquire ptable.lock in order to
3162     // change p->state and then call sched.
3163     // Once we hold ptable.lock, we can be
3164     // guaranteed that we won't miss any wakeup
3165     // (wakeup runs with ptable.lock locked),
3166     // so it's okay to release lk.
3167     if(lk != &ptable.lock){
3168         acquire(&ptable.lock);
3169         release(lk);
3170     }
3171
3172     // Go to sleep.
3173     proc->chan = chan;
3174     proc->state = SLEEPING;
3175     sched();
3176
3177     // Tidy up.
3178     proc->chan = 0;
3179
3180     // Reacquire original lock.
3181     if(lk != &ptable.lock){
3182         release(&ptable.lock);
3183         acquire(lk);
3184     }
3185 }
3186
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199

```

```

3200 // Wake up all processes sleeping on chan.
3201 // The ptable lock must be held.
3202 static void
3203 wakeup1(void *chan)
3204 {
3205     struct proc *p;
3206
3207     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
3208         if(p->state == SLEEPING && p->chan == chan)
3209             p->state = RUNNABLE;
3210 }
3211
3212 // Wake up all processes sleeping on chan.
3213 void
3214 wakeup(void *chan)
3215 {
3216     acquire(&ptable.lock);
3217     wakeup1(chan);
3218     release(&ptable.lock);
3219 }
3220
3221 // Kill the process with the given pid.
3222 // Process won't exit until it returns
3223 // to user space (see trap in trap.c).
3224 int
3225 kill(int pid)
3226 {
3227     struct proc *p;
3228
3229     acquire(&ptable.lock);
3230     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3231         if(p->pid == pid){
3232             p->killed = 1;
3233             // Wake process from sleep if necessary.
3234             if(p->state == SLEEPING)
3235                 p->state = RUNNABLE;
3236             release(&ptable.lock);
3237             return 0;
3238         }
3239     }
3240     release(&ptable.lock);
3241     return -1;
3242 }
3243
3244
3245
3246
3247
3248
3249

```

```

3250 // Print a process listing to console. For debugging.
3251 // Runs when user types ^P on console.
3252 // No lock to avoid wedging a stuck machine further.
3253 void
3254 procdump(void)
3255 {
3256     static char *states[] = {
3257         [UNUSED]    "unused",
3258         [EMBRYO]    "embryo",
3259         [SLEEPING]  "sleep ",
3260         [RUNNABLE]  "runble",
3261         [RUNNING]   "run   ",
3262         [ZOMBIE]    "zombie"
3263     };
3264     int i;
3265     struct proc *p;
3266     char *state;
3267     uint pc[10];
3268
3269     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3270         if(p->state == UNUSED)
3271             continue;
3272         if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
3273             state = states[p->state];
3274         else
3275             state = "???";
3276         cprintf("Process name: %s, state: %s, pid: %d, gid: %d, uid: %d \n ", p->
3277             if(p->state == SLEEPING){
3278                 getcallerpcs((uint*)p->context->ebp+2, pc);
3279                 for(i=0; i<10 && pc[i] != 0; i++)
3280                     cprintf(" %p", pc[i]);
3281             }
3282         cprintf("\n");
3283     }
3284 }
3285
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299

```



```

3300 # Context switch
3301 #
3302 # void switch(struct context **old, struct context *new);
3303 #
3304 # Save current register context in old
3305 # and then load register context from new.
3306
3307 .globl switch
3308 switch:
3309     movl 4(%esp), %eax
3310     movl 8(%esp), %edx
3311
3312 # Save old callee-save registers
3313     pushl %ebp
3314     pushl %ebx
3315     pushl %esi
3316     pushl %edi
3317
3318 # Switch stacks
3319     movl %esp, (%eax)
3320     movl %edx, %esp
3321
3322 # Load new callee-save registers
3323     popl %edi
3324     popl %esi
3325     popl %ebx
3326     popl %ebp
3327     ret
3328
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349

```

```

3350 // Physical memory allocator, intended to allocate
3351 // memory for user processes, kernel stacks, page table pages,
3352 // and pipe buffers. Allocates 4096-byte pages.
3353
3354 #include "types.h"
3355 #include "defs.h"
3356 #include "param.h"
3357 #include "memlayout.h"
3358 #include "mmu.h"
3359 #include "spinlock.h"
3360
3361 void freerange(void *vstart, void *vend);
3362 extern char end[]; // first address after kernel loaded from ELF file
3363
3364 struct run {
3365     struct run *next;
3366 };
3367
3368 struct {
3369     struct spinlock lock;
3370     int use_lock;
3371     struct run *freelist;
3372 } kmem;
3373
3374 // Initialization happens in two phases.
3375 // 1. main() calls kinit1() while still using entrypgdir to place just
3376 // the pages mapped by entrypgdir on free list.
3377 // 2. main() calls kinit2() with the rest of the physical pages
3378 // after installing a full page table that maps them on all cores.
3379 void
3380 kinit1(void *vstart, void *vend)
3381 {
3382     initlock(&kmem.lock, "kmem");
3383     kmem.use_lock = 0;
3384     freerange(vstart, vend);
3385 }
3386
3387 void
3388 kinit2(void *vstart, void *vend)
3389 {
3390     freerange(vstart, vend);
3391     kmem.use_lock = 1;
3392 }
3393
3394
3395
3396
3397
3398
3399

```

```

3400 void
3401 freerange(void *vstart, void *vend)
3402 {
3403     char *p;
3404     p = (char*)PGROUNDUP((uint)vstart);
3405     for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3406         kfree(p);
3407 }
3408
3409
3410 // Free the page of physical memory pointed at by v,
3411 // which normally should have been returned by a
3412 // call to kalloc(). (The exception is when
3413 // initializing the allocator; see kinit above.)
3414 void
3415 kfree(char *v)
3416 {
3417     struct run *r;
3418
3419     if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
3420         panic("kfree");
3421
3422     // Fill with junk to catch dangling refs.
3423     memset(v, 1, PGSIZE);
3424
3425     if(kmem.use_lock)
3426         acquire(&kmem.lock);
3427     r = (struct run*)v;
3428     r->next = kmem.freelist;
3429     kmem.freelist = r;
3430     if(kmem.use_lock)
3431         release(&kmem.lock);
3432 }
3433
3434 // Allocate one 4096-byte page of physical memory.
3435 // Returns a pointer that the kernel can use.
3436 // Returns 0 if the memory cannot be allocated.
3437 char*
3438 kalloc(void)
3439 {
3440     struct run *r;
3441
3442     if(kmem.use_lock)
3443         acquire(&kmem.lock);
3444     r = kmem.freelist;
3445     if(r)
3446         kmem.freelist = r->next;
3447     if(kmem.use_lock)
3448         release(&kmem.lock);
3449     return (char*)r;

```

```

3450 }
3451
3452
3453
3454
3455
3456
3457
3458
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3469
3470
3471
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 // x86 trap and interrupt constants.
3501
3502 // Processor-defined:
3503 #define T_DIVIDE      0      // divide error
3504 #define T_DEBUG      1      // debug exception
3505 #define T_NMI        2      // non-maskable interrupt
3506 #define T_BRKPT      3      // breakpoint
3507 #define T_OFLOW      4      // overflow
3508 #define T_BOUND      5      // bounds check
3509 #define T_ILLOP      6      // illegal opcode
3510 #define T_DEVICE      7      // device not available
3511 #define T_DBLFLT      8      // double fault
3512 // #define T_COPROC    9      // reserved (not used since 486)
3513 #define T_TSS        10     // invalid task switch segment
3514 #define T_SEGNP      11     // segment not present
3515 #define T_STACK      12     // stack exception
3516 #define T_GPFLT      13     // general protection fault
3517 #define T_PGFLT      14     // page fault
3518 // #define T_RES       15     // reserved
3519 #define T_FPERR      16     // floating point error
3520 #define T_ALIGN      17     // alignment check
3521 #define T_MCHK       18     // machine check
3522 #define T_SIMDERR     19     // SIMD floating point error
3523
3524 // These are arbitrarily chosen, but with care not to overlap
3525 // processor defined exceptions or interrupt vectors.
3526 #define T_SYSCALL     64     // system call
3527 #define T_DEFAULT     500    // catchall
3528
3529 #define T_IRQ0        32     // IRQ 0 corresponds to int T_IRQ
3530
3531 #define IRQ_TIMER      0
3532 #define IRQ_KBD        1
3533 #define IRQ_COM1       4
3534 #define IRQ_IDE       14
3535 #define IRQ_ERROR      19
3536 #define IRQ_SPURIOUS   31
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549

```

```

3550 #!/usr/bin/perl -w
3551
3552 # Generate vectors.S, the trap/interrupt entry points.
3553 # There has to be one entry point per interrupt number
3554 # since otherwise there's no way for trap() to discover
3555 # the interrupt number.
3556
3557 print "# generated by vectors.pl - do not edit\n";
3558 print "# handlers\n";
3559 print ".globl alltraps\n";
3560 for(my $i = 0; $i < 256; $i++){
3561     print ".globl vector$i\n";
3562     print "vector$i:\n";
3563     if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
3564         print "    pushl \($0\n";
3565     }
3566     print "    pushl \($$i\n";
3567     print "    jmp alltraps\n";
3568 }
3569
3570 print "\n# vector table\n";
3571 print ".data\n";
3572 print ".globl vectors\n";
3573 print "vectors:\n";
3574 for(my $i = 0; $i < 256; $i++){
3575     print "    .long vector$i\n";
3576 }
3577
3578 # sample output:
3579 # # handlers
3580 # .globl alltraps
3581 # .globl vector0
3582 # vector0:
3583 #     pushl $0
3584 #     pushl $0
3585 #     jmp alltraps
3586 # ...
3587 #
3588 # # vector table
3589 # .data
3590 # .globl vectors
3591 # vectors:
3592 #     .long vector0
3593 #     .long vector1
3594 #     .long vector2
3595 # ...
3596
3597
3598
3599

```

```

3600 #include "mmu.h"
3601
3602 # vectors.S sends all traps here.
3603 .globl alltraps
3604 alltraps:
3605 # Build trap frame.
3606 pushl %ds
3607 pushl %es
3608 pushl %fs
3609 pushl %gs
3610 pushal
3611
3612 # Set up data and per-cpu segments.
3613 movw $(SEG_KDATA<<3), %ax
3614 movw %ax, %ds
3615 movw %ax, %es
3616 movw $(SEG_KCPU<<3), %ax
3617 movw %ax, %fs
3618 movw %ax, %gs
3619
3620 # Call trap(tf), where tf=%esp
3621 pushl %esp
3622 call trap
3623 addl $4, %esp
3624
3625 # Return falls through to trapret...
3626 .globl trapret
3627 trapret:
3628 popal
3629 popl %gs
3630 popl %fs
3631 popl %es
3632 popl %ds
3633 addl $0x8, %esp # trapno and errcode
3634 iret
3635
3636
3637
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649

```

```

3650 #include "types.h"
3651 #include "defs.h"
3652 #include "param.h"
3653 #include "memlayout.h"
3654 #include "mmu.h"
3655 #include "proc.h"
3656 #include "x86.h"
3657 #include "traps.h"
3658 #include "spinlock.h"
3659
3660 // Interrupt descriptor table (shared by all CPUs).
3661 struct gatedesc idt[256];
3662 extern uint vectors[]; // in vectors.S: array of 256 entry pointers
3663 struct spinlock tickslock;
3664 uint ticks;
3665
3666 void
3667 tvinit(void)
3668 {
3669     int i;
3670
3671     for(i = 0; i < 256; i++)
3672         SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3673     SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3674
3675     initlock(&tickslock, "time");
3676 }
3677
3678 void
3679 idtinit(void)
3680 {
3681     lidt(idt, sizeof(idt));
3682 }
3683
3684
3685
3686
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699

```

```

3700 void
3701 trap(struct trapframe *tf)
3702 {
3703     if(tf->trapno == T_SYSCALL){
3704         if(proc->killed)
3705             exit();
3706         proc->tf = tf;
3707         syscall();
3708         if(proc->killed)
3709             exit();
3710         return;
3711     }
3712     switch(tf->trapno){
3713     case T_IRQ0 + IRQ_TIMER:
3714         if(cpu->id == 0){
3715             acquire(&tickslock);
3716             ticks++;
3717             wakeup(&ticks);
3718             release(&tickslock);
3719         }
3720         lapiceoi();
3721         break;
3722     case T_IRQ0 + IRQ_IDE:
3723         ideintr();
3724         lapiceoi();
3725         break;
3726     case T_IRQ0 + IRQ_IDE+1:
3727         // Bochs generates spurious IDE1 interrupts.
3728         break;
3729     case T_IRQ0 + IRQ_KBD:
3730         kbdintr();
3731         lapiceoi();
3732         break;
3733     case T_IRQ0 + IRQ_COM1:
3734         uartintr();
3735         lapiceoi();
3736         break;
3737     case T_IRQ0 + 7:
3738     case T_IRQ0 + IRQ_SPURIOUS:
3739         cprintf("cpu%d: spurious interrupt at %x:%x\n",
3740             cpu->id, tf->cs, tf->eip);
3741         lapiceoi();
3742         break;
3743     }
3744 }
3745
3746
3747
3748
3749

```

```

3750 default:
3751     if(proc == 0 || (tf->cs&3) == 0){
3752         // In kernel, it must be our mistake.
3753         cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3754             tf->trapno, cpu->id, tf->eip, rcr2());
3755         panic("trap");
3756     }
3757     // In user space, assume process misbehaved.
3758     cprintf("pid %d %s: trap %d err %d on cpu %d "
3759         "eip 0x%x addr 0x%x--kill proc\n",
3760         proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
3761         rcr2());
3762     proc->killed = 1;
3763 }
3764
3765 // Force process exit if it has been killed and is in user space.
3766 // (If it is still executing in the kernel, let it keep running
3767 // until it gets to the regular system call return.)
3768 if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3769     exit();
3770
3771 // Force process to give up CPU on clock tick.
3772 // If interrupts were on while locks held, would need to check nlock.
3773 if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3774     yield();
3775
3776 // Check if the process has been killed since we yielded
3777 if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3778     exit();
3779 }
3780
3781
3782
3783
3784
3785
3786
3787
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3799

```

```

3800 // System call numbers
3801 #define SYS_fork    1
3802 #define SYS_exit    2
3803 #define SYS_wait    3
3804 #define SYS_pipe    4
3805 #define SYS_read    5
3806 #define SYS_kill    6
3807 #define SYS_exec    7
3808 #define SYS_fstat   8
3809 #define SYS_chdir   9
3810 #define SYS_dup    10
3811 #define SYS_getpid  11
3812 #define SYS_sbrk   12
3813 #define SYS_sleep  13
3814 #define SYS_uptime 14
3815 #define SYS_open   15
3816 #define SYS_write  16
3817 #define SYS_mknod  17
3818 #define SYS_unlink 18
3819 #define SYS_link   19
3820 #define SYS_mkdir  20
3821 #define SYS_close  21
3822 #define SYS_halt   22
3823 #define SYS_date   23
3824 #define SYS_getuid 24
3825 #define SYS_getgid 25
3826 #define SYS_getppid 26
3827 #define SYS_setuid  27
3828 #define SYS_setgid  28
3829 #define SYS_getprocs 29
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849

```

```

3850 #include "types.h"
3851 #include "defs.h"
3852 #include "param.h"
3853 #include "memlayout.h"
3854 #include "mmu.h"
3855 #include "proc.h"
3856 #include "x86.h"
3857 #include "syscall.h"
3858
3859 // User code makes a system call with INT T_SYSCALL.
3860 // System call number in %eax.
3861 // Arguments on the stack, from the user call to the C
3862 // library system call function. The saved user %esp points
3863 // to a saved program counter, and then the first argument.
3864
3865 // Fetch the int at addr from the current process.
3866 int
3867 fetchint(uint addr, int *ip)
3868 {
3869     if(addr >= proc->sz || addr+4 > proc->sz)
3870         return -1;
3871     *ip = *(int*)(addr);
3872     return 0;
3873 }
3874
3875 // Fetch the nul-terminated string at addr from the current process.
3876 // Doesn't actually copy the string - just sets *pp to point at it.
3877 // Returns length of string, not including nul.
3878 int
3879 fetchstr(uint addr, char **pp)
3880 {
3881     char *s, *ep;
3882
3883     if(addr >= proc->sz)
3884         return -1;
3885     *pp = (char*)addr;
3886     ep = (char*)proc->sz;
3887     for(s = *pp; s < ep; s++)
3888         if(*s == 0)
3889             return s - *pp;
3890     return -1;
3891 }
3892
3893 // Fetch the nth 32-bit system call argument.
3894 int
3895 argint(int n, int *ip)
3896 {
3897     return fetchint(proc->tf->esp + 4 + 4*n, ip);
3898 }
3899

```

```

3900 // Fetch the nth word-sized system call argument as a pointer
3901 // to a block of memory of size n bytes. Check that the pointer
3902 // lies within the process address space.
3903 int
3904 argptr(int n, char **pp, int size)
3905 {
3906     int i;
3907
3908     if(argint(n, &i) < 0)
3909         return -1;
3910     if((uint)i >= proc->sz || (uint)i+size > proc->sz)
3911         return -1;
3912     *pp = (char*)i;
3913     return 0;
3914 }
3915
3916 // Fetch the nth word-sized system call argument as a string pointer.
3917 // Check that the pointer is valid and the string is nul-terminated.
3918 // (There is no shared writable memory, so the string can't change
3919 // between this check and being used by the kernel.)
3920 int
3921 argstr(int n, char **pp)
3922 {
3923     int addr;
3924     if(argint(n, &addr) < 0)
3925         return -1;
3926     return fetchstr(addr, pp);
3927 }
3928
3929 extern int sys_chdir(void);
3930 extern int sys_close(void);
3931 extern int sys_dup(void);
3932 extern int sys_exec(void);
3933 extern int sys_exit(void);
3934 extern int sys_fork(void);
3935 extern int sys_fstat(void);
3936 extern int sys_getpid(void);
3937 extern int sys_kill(void);
3938 extern int sys_link(void);
3939 extern int sys_mkdir(void);
3940 extern int sys_mknod(void);
3941 extern int sys_open(void);
3942 extern int sys_pipe(void);
3943 extern int sys_read(void);
3944 extern int sys_sbrk(void);
3945 extern int sys_sleep(void);
3946 extern int sys_unlink(void);
3947 extern int sys_wait(void);
3948 extern int sys_write(void);
3949 extern int sys_uptime(void);

```

```

3950 extern int sys_halt(void);
3951 extern int sys_date(void);
3952 extern int sys_getuid(void);
3953 extern int sys_getgid(void);
3954 extern int sys_getppid(void);
3955 extern int sys_setuid(void);
3956 extern int sys_setgid(void);
3957 extern int sys_getprocs(void);
3958
3959 static int (*syscalls[])(void) = {
3960     [SYS_fork]    sys_fork,
3961     [SYS_exit]    sys_exit,
3962     [SYS_wait]    sys_wait,
3963     [SYS_pipe]    sys_pipe,
3964     [SYS_read]    sys_read,
3965     [SYS_kill]    sys_kill,
3966     [SYS_exec]    sys_exec,
3967     [SYS_fstat]   sys_fstat,
3968     [SYS_chdir]   sys_chdir,
3969     [SYS_dup]     sys_dup,
3970     [SYS_getpid]  sys_getpid,
3971     [SYS_sbrk]    sys_sbrk,
3972     [SYS_sleep]   sys_sleep,
3973     [SYS_uptime]  sys_uptime,
3974     [SYS_open]    sys_open,
3975     [SYS_write]   sys_write,
3976     [SYS_mknod]   sys_mknod,
3977     [SYS_unlink]  sys_unlink,
3978     [SYS_link]    sys_link,
3979     [SYS_mkdir]   sys_mkdir,
3980     [SYS_close]   sys_close,
3981     [SYS_halt]    sys_halt,
3982     [SYS_date]    sys_date,
3983     [SYS_getuid]  sys_getuid,
3984     [SYS_getgid]  sys_getgid,
3985     [SYS_getppid] sys_getppid,
3986     [SYS_setuid]  sys_setuid,
3987     [SYS_setgid]  sys_setgid,
3988     [SYS_getprocs] sys_getprocs,
3989 };
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999

```

```

4000 char * sysCallNames[] = {
4001 [SYS_fork]    "fork",
4002 [SYS_exit]    "exit",
4003 [SYS_wait]    "wait",
4004 [SYS_pipe]    "pipe",
4005 [SYS_read]    "read",
4006 [SYS_kill]    "kill",
4007 [SYS_exec]    "exec",
4008 [SYS_fstat]   "fstat",
4009 [SYS_chdir]   "chdir",
4010 [SYS_dup]     "dup",
4011 [SYS_getpid]  "getpid",
4012 [SYS_sbrk]    "sbrk",
4013 [SYS_sleep]   "sleep",
4014 [SYS_uptime]  "uptime",
4015 [SYS_open]    "open",
4016 [SYS_write]   "write",
4017 [SYS_mknod]   "mknod",
4018 [SYS_unlink]  "unlink",
4019 [SYS_link]    "link",
4020 [SYS_mkdir]   "mkdir",
4021 [SYS_close]   "close",
4022 [SYS_halt]    "halt",
4023 [SYS_date]    "date",
4024 };
4025
4026 void
4027 syscall(void)
4028 {
4029     int num;
4030
4031     num = proc->tf->eax;
4032     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
4033         proc->tf->eax = syscalls[num]();
4034         //int returnValue = proc->tf->eax;
4035         //Omits printing the 'write' system call (16) for readability
4036         //if(num != 16) { cprintf("%s -> %d\n", sysCallNames[num], returnValue);
4037     } else {
4038         cprintf("%d %s: unknown sys call %d\n",
4039             proc->pid, proc->name, num);
4040         proc->tf->eax = -1;
4041     }
4042 }
4043
4044
4045
4046
4047
4048
4049

```

```

4050 #include "types.h"
4051 #include "x86.h"
4052 #include "defs.h"
4053 #include "date.h"
4054 #include "param.h"
4055 #include "memlayout.h"
4056 #include "mmu.h"
4057 #include "proc.h"
4058 ;
4059 int
4060 sys_fork(void)
4061 {
4062     return fork();
4063 }
4064
4065 int
4066 sys_exit(void)
4067 {
4068     exit();
4069     return 0; // not reached
4070 }
4071
4072 int
4073 sys_wait(void)
4074 {
4075     return wait();
4076 }
4077
4078 int
4079 sys_kill(void)
4080 {
4081     int pid;
4082
4083     if(argint(0, &pid) < 0)
4084         return -1;
4085     return kill(pid);
4086 }
4087
4088 int
4089 sys_getpid(void)
4090 {
4091     return proc->pid;
4092 }
4093
4094
4095
4096
4097
4098
4099

```



```

4100 int
4101 sys_sbrk(void)
4102 {
4103     int addr;
4104     int n;
4105
4106     if(argint(0, &n) < 0)
4107         return -1;
4108     addr = proc->sz;
4109     if(growproc(n) < 0)
4110         return -1;
4111     return addr;
4112 }
4113
4114 int
4115 sys_sleep(void)
4116 {
4117     int n;
4118     uint ticks0;
4119
4120     if(argint(0, &n) < 0)
4121         return -1;
4122     acquire(&tickslock);
4123     ticks0 = ticks;
4124     while(ticks - ticks0 < n){
4125         if(proc->killed){
4126             release(&tickslock);
4127             return -1;
4128         }
4129         sleep(&ticks, &tickslock);
4130     }
4131     release(&tickslock);
4132     return 0;
4133 }
4134
4135 // return how many clock tick interrupts have occurred
4136 // since start.
4137 int
4138 sys_uptime(void)
4139 {
4140     uint xticks;
4141
4142     acquire(&tickslock);
4143     xticks = ticks;
4144     release(&tickslock);
4145     return xticks;
4146 }
4147
4148
4149

```

```

4150 //Turn of the computer
4151 int sys_halt(void){
4152     cprintf("Shutting down ...\n");
4153     outw(0xB004, 0x0 | 0x2000);
4154     return 0;
4155 }
4156
4157 int
4158 sys_date(void)
4159 {
4160     struct rtcdate * d;
4161
4162     if(argptr(0, (void*)&d, sizeof(*d)) < 0)
4163         return -1;
4164
4165     cmostime(d);
4166
4167     return 0;
4168 }
4169
4170 int
4171 sys_getuid(void)
4172 {
4173     return proc->uid;
4174 }
4175
4176 int
4177 sys_getgid(void)
4178 {
4179     return proc->gid;
4180 }
4181
4182 int
4183 sys_getppid(void)
4184 {
4185     return proc->ppid;
4186 }
4187
4188 int
4189 sys_setuid(void)
4190 {
4191     int new_uid;
4192
4193     if(argint(0, &new_uid) < 0) //test if argint failed
4194         return -1;
4195
4196     proc->uid = new_uid;
4197     return 0;
4198 }
4199

```

```
4200 int
4201 sys_setgid(void)
4202 {
4203     int new_gid;
4204
4205     if(argint(0, &new_gid) < 0) //test if argint failed
4206         return -1;
4207
4208     proc->gid = new_gid;
4209     return 0;
4210 }
4211
4212 int
4213 sys_getprocs(void)
4214 {
4215
4216     struct uproc * table;
4217     int max;
4218
4219     //get int max arg off the stack
4220     if(argint(0, &max) < 0)
4221         return -1;
4222
4223     //get uproc structs off the stack
4224     if(argptr(1, (void*)&table, sizeof(*table)) < 0)
4225         return -1;
4226
4227     int procs_retrieved =get_current_procs(max, table);
4228
4229     return procs_retrieved;
4230 }
4231
4232
4233
4234
4235
4236
4237
4238
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
```

```
4250 // halt the system.
4251 #include "types.h"
4252 #include "user.h"
4253
4254 int
4255 main(void) {
4256     halt();
4257     return 0;
4258 }
4259
4260
4261
4262
4263
4264
4265
4266
4267
4268
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
```

```
4300 struct buf {
4301     int flags;
4302     uint dev;
4303     uint blockno;
4304     struct buf *prev; // LRU cache list
4305     struct buf *next;
4306     struct buf *qnext; // disk queue
4307     uchar data[BSIZE];
4308 };
4309 #define B_BUSY 0x1 // buffer is locked by some process
4310 #define B_VALID 0x2 // buffer has been read from disk
4311 #define B_DIRTY 0x4 // buffer needs to be written to disk
4312
4313
4314
4315
4316
4317
4318
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4330
4331
4332
4333
4334
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
```

```
4350 #define O_RDONLY 0x000
4351 #define O_WRONLY 0x001
4352 #define O_RDWR 0x002
4353 #define O_CREATE 0x200
4354
4355
4356
4357
4358
4359
4360
4361
4362
4363
4364
4365
4366
4367
4368
4369
4370
4371
4372
4373
4374
4375
4376
4377
4378
4379
4380
4381
4382
4383
4384
4385
4386
4387
4388
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
```

```

4400 #define T_DIR 1 // Directory
4401 #define T_FILE 2 // File
4402 #define T_DEV 3 // Device
4403
4404 struct stat {
4405     short type; // Type of file
4406     int dev; // File system's disk device
4407     uint ino; // Inode number
4408     short nlink; // Number of links to file
4409     uint size; // Size of file in bytes
4410 };
4411
4412
4413
4414
4415
4416
4417
4418
4419
4420
4421
4422
4423
4424
4425
4426
4427
4428
4429
4430
4431
4432
4433
4434
4435
4436
4437
4438
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449

```

```

4450 // On-disk file system format.
4451 // Both the kernel and user programs use this header file.
4452
4453
4454 #define ROOTINO 1 // root i-number
4455 #define BSIZE 512 // block size
4456
4457 // Disk layout:
4458 // [ boot block | super block | log | inode blocks | free bit map | data blocks ]
4459 //
4460 // mkfs computes the super block and builds an initial file system. The super block
4461 // the disk layout:
4462 struct superblock {
4463     uint size; // Size of file system image (blocks)
4464     uint nblocks; // Number of data blocks
4465     uint ninodes; // Number of inodes.
4466     uint nlog; // Number of log blocks
4467     uint logstart; // Block number of first log block
4468     uint inodestart; // Block number of first inode block
4469     uint bmapstart; // Block number of first free map block
4470 };
4471
4472 #define NDIRECT 12
4473 #define NINDIRECT (BSIZE / sizeof(uint))
4474 #define MAXFILE (NDIRECT + NINDIRECT)
4475
4476 // On-disk inode structure
4477 struct dinode {
4478     short type; // File type
4479     short major; // Major device number (T_DEV only)
4480     short minor; // Minor device number (T_DEV only)
4481     short nlink; // Number of links to inode in file system
4482     uint size; // Size of file (bytes)
4483     uint addrs[NDIRECT+1]; // Data block addresses
4484 };
4485
4486
4487
4488
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499

```

```

4500 // Inodes per block.
4501 #define IPB          (BSIZE / sizeof(struct dinode))
4502
4503 // Block containing inode i
4504 #define IBLOCK(i, sb) ((i) / IPB + sb.inodestart)
4505
4506 // Bitmap bits per block
4507 #define BPB          (BSIZE*8)
4508
4509 // Block of free map containing bit for block b
4510 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
4511
4512 // Directory is a file containing a sequence of dirent structures.
4513 #define DIRSIZ 14
4514
4515 struct dirent {
4516     ushort inum;
4517     char name[DIRSIZ];
4518 };
4519
4520
4521
4522
4523
4524
4525
4526
4527
4528
4529
4530
4531
4532
4533
4534
4535
4536
4537
4538
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4549

```

```

4550 struct file {
4551     enum { FD_NONE, FD_PIPE, FD_INODE } type;
4552     int ref; // reference count
4553     char readable;
4554     char writable;
4555     struct pipe *pipe;
4556     struct inode *ip;
4557     uint off;
4558 };
4559
4560
4561 // in-memory copy of an inode
4562 struct inode {
4563     uint dev;           // Device number
4564     uint inum;          // Inode number
4565     int ref;            // Reference count
4566     int flags;          // I_BUSY, I_VALID
4567
4568     short type;         // copy of disk inode
4569     short major;
4570     short minor;
4571     short nlink;
4572     uint size;
4573     uint addrs[NDIRECT+1];
4574 };
4575 #define I_BUSY 0x1
4576 #define I_VALID 0x2
4577
4578 // table mapping major device number to
4579 // device functions
4580 struct devsw {
4581     int (*read)(struct inode*, char*, int);
4582     int (*write)(struct inode*, char*, int);
4583 };
4584
4585 extern struct devsw devsw[];
4586
4587 #define CONSOLE 1
4588
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4599

```

```

4600 // Blank page.
4601
4602
4603
4604
4605
4606
4607
4608
4609
4610
4611
4612
4613
4614
4615
4616
4617
4618
4619
4620
4621
4622
4623
4624
4625
4626
4627
4628
4629
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649

```

```

4650 // Simple PIO-based (non-DMA) IDE driver code.
4651
4652 #include "types.h"
4653 #include "defs.h"
4654 #include "param.h"
4655 #include "memlayout.h"
4656 #include "mmu.h"
4657 #include "proc.h"
4658 #include "x86.h"
4659 #include "traps.h"
4660 #include "spinlock.h"
4661 #include "fs.h"
4662 #include "buf.h"
4663
4664 #define SECTOR_SIZE 512
4665 #define IDE_BSY 0x80
4666 #define IDE_DRDY 0x40
4667 #define IDE_DF 0x20
4668 #define IDE_ERR 0x01
4669
4670 #define IDE_CMD_READ 0x20
4671 #define IDE_CMD_WRITE 0x30
4672
4673 // idequeue points to the buf now being read/written to the disk.
4674 // idequeue->qnext points to the next buf to be processed.
4675 // You must hold idelock while manipulating queue.
4676
4677 static struct spinlock idelock;
4678 static struct buf *idequeue;
4679
4680 static int havedisk1;
4681 static void idestart(struct buf*);
4682
4683 // Wait for IDE disk to become ready.
4684 static int
4685 idewait(int checkerr)
4686 {
4687     int r;
4688     while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4689         ;
4690     if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4691         return -1;
4692     return 0;
4693 }
4694
4695
4696
4697
4698
4699

```

```

4700 void
4701 ideinit(void)
4702 {
4703     int i;
4704
4705     initlock(&idelock, "ide");
4706     picenable(IRQ_IDE);
4707     ioapicenable(IRQ_IDE, ncpu - 1);
4708     idewait(0);
4709
4710     // Check if disk 1 is present
4711     outb(0x1f6, 0xe0 | (1<<4));
4712     for(i=0; i<1000; i++){
4713         if(inb(0x1f7) != 0){
4714             havedisk1 = 1;
4715             break;
4716         }
4717     }
4718
4719     // Switch back to disk 0.
4720     outb(0x1f6, 0xe0 | (0<<4));
4721 }
4722
4723 // Start the request for b. Caller must hold idelock.
4724 static void
4725 idestart(struct buf *b)
4726 {
4727     if(b == 0)
4728         panic("idestart");
4729     if(b->blockno >= FSSIZE)
4730         panic("incorrect blockno");
4731     int sector_per_block = BSIZE/SECTOR_SIZE;
4732     int sector = b->blockno * sector_per_block;
4733
4734     if (sector_per_block > 7) panic("idestart");
4735
4736     idewait(0);
4737     outb(0x3f6, 0); // generate interrupt
4738     outb(0x1f2, sector_per_block); // number of sectors
4739     outb(0x1f3, sector & 0xff);
4740     outb(0x1f4, (sector >> 8) & 0xff);
4741     outb(0x1f5, (sector >> 16) & 0xff);
4742     outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4743     if(b->flags & B_DIRTY){
4744         outb(0x1f7, IDE_CMD_WRITE);
4745         outsl(0x1f0, b->data, BSIZE/4);
4746     } else {
4747         outb(0x1f7, IDE_CMD_READ);
4748     }
4749 }

```

```

4750 // Interrupt handler.
4751 void
4752 ideintr(void)
4753 {
4754     struct buf *b;
4755
4756     // First queued buffer is the active request.
4757     acquire(&idelock);
4758     if((b = idequeue) == 0){
4759         release(&idelock);
4760         // cprintf("spurious IDE interrupt\n");
4761         return;
4762     }
4763     idequeue = b->qnext;
4764
4765     // Read data if needed.
4766     if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4767         insl(0x1f0, b->data, BSIZE/4);
4768
4769     // Wake process waiting for this buf.
4770     b->flags |= B_VALID;
4771     b->flags &= ~B_DIRTY;
4772     wakeup(b);
4773
4774     // Start disk on next buf in queue.
4775     if(idequeue != 0)
4776         idestart(idequeue);
4777
4778     release(&idelock);
4779 }
4780
4781
4782
4783
4784
4785
4786
4787
4788
4789
4790
4791
4792
4793
4794
4795
4796
4797
4798
4799

```

```

4800 // Sync buf with disk.
4801 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
4802 // Else if B_VALID is not set, read buf from disk, set B_VALID.
4803 void
4804 iderw(struct buf *b)
4805 {
4806     struct buf **pp;
4807
4808     if(!(b->flags & B_BUSY))
4809         panic("iderw: buf not busy");
4810     if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
4811         panic("iderw: nothing to do");
4812     if(b->dev != 0 && !havedisk1)
4813         panic("iderw: ide disk 1 not present");
4814
4815     acquire(&idelock);
4816
4817     // Append b to idequeue.
4818     b->qnext = 0;
4819     for(pp=&idequeue; *pp; pp=(*pp)->qnext)
4820         ;
4821     *pp = b;
4822
4823     // Start disk if necessary.
4824     if(idequeue == b)
4825         idestart(b);
4826
4827     // Wait for request to finish.
4828     while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
4829         sleep(b, &idelock);
4830     }
4831
4832     release(&idelock);
4833 }
4834
4835
4836
4837
4838
4839
4840
4841
4842
4843
4844
4845
4846
4847
4848
4849

```

```

4850 // Buffer cache.
4851 //
4852 // The buffer cache is a linked list of buf structures holding
4853 // cached copies of disk block contents. Caching disk blocks
4854 // in memory reduces the number of disk reads and also provides
4855 // a synchronization point for disk blocks used by multiple processes.
4856 //
4857 // Interface:
4858 // * To get a buffer for a particular disk block, call bread.
4859 // * After changing buffer data, call bwrite to write it to disk.
4860 // * When done with the buffer, call brelse.
4861 // * Do not use the buffer after calling brelse.
4862 // * Only one process at a time can use a buffer,
4863 //   so do not keep them longer than necessary.
4864 //
4865 // The implementation uses three state flags internally:
4866 // * B_BUSY: the block has been returned from bread
4867 //   and has not been passed back to brelse.
4868 // * B_VALID: the buffer data has been read from the disk.
4869 // * B_DIRTY: the buffer data has been modified
4870 //   and needs to be written to disk.
4871
4872 #include "types.h"
4873 #include "defs.h"
4874 #include "param.h"
4875 #include "spinlock.h"
4876 #include "fs.h"
4877 #include "buf.h"
4878
4879 struct {
4880     struct spinlock lock;
4881     struct buf buf[NBUF];
4882
4883     // Linked list of all buffers, through prev/next.
4884     // head.next is most recently used.
4885     struct buf head;
4886 } bcache;
4887
4888 void
4889 binit(void)
4890 {
4891     struct buf *b;
4892
4893     initlock(&bcache.lock, "bcache");
4894
4895
4896
4897
4898
4899

```



```

4900 // Create linked list of buffers
4901 bcache.head.prev = &bcache.head;
4902 bcache.head.next = &bcache.head;
4903 for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4904     b->next = bcache.head.next;
4905     b->prev = &bcache.head;
4906     b->dev = -1;
4907     bcache.head.next->prev = b;
4908     bcache.head.next = b;
4909 }
4910 }
4911
4912 // Look through buffer cache for block on device dev.
4913 // If not found, allocate a buffer.
4914 // In either case, return B_BUSY buffer.
4915 static struct buf*
4916 bget(uint dev, uint blockno)
4917 {
4918     struct buf *b;
4919
4920     acquire(&bcache.lock);
4921
4922     loop:
4923     // Is the block already cached?
4924     for(b = bcache.head.next; b != &bcache.head; b = b->next){
4925         if(b->dev == dev && b->blockno == blockno){
4926             if(!(b->flags & B_BUSY)){
4927                 b->flags |= B_BUSY;
4928                 release(&bcache.lock);
4929                 return b;
4930             }
4931             sleep(b, &bcache.lock);
4932             goto loop;
4933         }
4934     }
4935
4936     // Not cached; recycle some non-busy and clean buffer.
4937     // "clean" because B_DIRTY and !B_BUSY means log.c
4938     // hasn't yet committed the changes to the buffer.
4939     for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4940         if((b->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
4941             b->dev = dev;
4942             b->blockno = blockno;
4943             b->flags = B_BUSY;
4944             release(&bcache.lock);
4945             return b;
4946         }
4947     }
4948     panic("bget: no buffers");
4949 }

```

```

4950 // Return a B_BUSY buf with the contents of the indicated block.
4951 struct buf*
4952 bread(uint dev, uint blockno)
4953 {
4954     struct buf *b;
4955
4956     b = bget(dev, blockno);
4957     if(!(b->flags & B_VALID)) {
4958         iderw(b);
4959     }
4960     return b;
4961 }
4962
4963 // Write b's contents to disk. Must be B_BUSY.
4964 void
4965 bwrite(struct buf *b)
4966 {
4967     if((b->flags & B_BUSY) == 0)
4968         panic("bwrite");
4969     b->flags |= B_DIRTY;
4970     iderw(b);
4971 }
4972
4973 // Release a B_BUSY buffer.
4974 // Move to the head of the MRU list.
4975 void
4976 brelse(struct buf *b)
4977 {
4978     if((b->flags & B_BUSY) == 0)
4979         panic("brelse");
4980
4981     acquire(&bcache.lock);
4982
4983     b->next->prev = b->prev;
4984     b->prev->next = b->next;
4985     b->next = bcache.head.next;
4986     b->prev = &bcache.head;
4987     bcache.head.next->prev = b;
4988     bcache.head.next = b;
4989
4990     b->flags &= ~B_BUSY;
4991     wakeup(b);
4992
4993     release(&bcache.lock);
4994 }
4995
4996
4997
4998
4999

```

```

5000 // Blank page.
5001
5002
5003
5004
5005
5006
5007
5008
5009
5010
5011
5012
5013
5014
5015
5016
5017
5018
5019
5020
5021
5022
5023
5024
5025
5026
5027
5028
5029
5030
5031
5032
5033
5034
5035
5036
5037
5038
5039
5040
5041
5042
5043
5044
5045
5046
5047
5048
5049

```

```

5050 #include "types.h"
5051 #include "defs.h"
5052 #include "param.h"
5053 #include "spinlock.h"
5054 #include "fs.h"
5055 #include "buf.h"
5056
5057 // Simple logging that allows concurrent FS system calls.
5058 //
5059 // A log transaction contains the updates of multiple FS system
5060 // calls. The logging system only commits when there are
5061 // no FS system calls active. Thus there is never
5062 // any reasoning required about whether a commit might
5063 // write an uncommitted system call's updates to disk.
5064 //
5065 // A system call should call begin_op()/end_op() to mark
5066 // its start and end. Usually begin_op() just increments
5067 // the count of in-progress FS system calls and returns.
5068 // But if it thinks the log is close to running out, it
5069 // sleeps until the last outstanding end_op() commits.
5070 //
5071 // The log is a physical re-do log containing disk blocks.
5072 // The on-disk log format:
5073 //   header block, containing block #s for block A, B, C, ...
5074 //   block A
5075 //   block B
5076 //   block C
5077 //   ...
5078 // Log appends are synchronous.
5079
5080 // Contents of the header block, used for both the on-disk header block
5081 // and to keep track in memory of logged block# before commit.
5082 struct logheader {
5083   int n;
5084   int block[LOGSIZE];
5085 };
5086
5087 struct log {
5088   struct spinlock lock;
5089   int start;
5090   int size;
5091   int outstanding; // how many FS sys calls are executing.
5092   int committing; // in commit(), please wait.
5093   int dev;
5094   struct logheader lh;
5095 };
5096
5097
5098
5099

```

```

5100 struct log log;
5101
5102 static void recover_from_log(void);
5103 static void commit();
5104
5105 void
5106 initlog(int dev)
5107 {
5108     if (sizeof(struct logheader) >= BSIZE)
5109         panic("initlog: too big logheader");
5110
5111     struct superblock sb;
5112     initlock(&log.lock, "log");
5113     readsb(dev, &sb);
5114     log.start = sb.logstart;
5115     log.size = sb.nlog;
5116     log.dev = dev;
5117     recover_from_log();
5118 }
5119
5120 // Copy committed blocks from log to their home location
5121 static void
5122 install_trans(void)
5123 {
5124     int tail;
5125
5126     for (tail = 0; tail < log.lh.n; tail++) {
5127         struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
5128         struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
5129         memmove(dbuf->data, lbuf->data, BSIZE); // copy block to dst
5130         bwrite(dbuf); // write dst to disk
5131         brelse(lbuf);
5132         brelse(dbuf);
5133     }
5134 }
5135
5136 // Read the log header from disk into the in-memory log header
5137 static void
5138 read_head(void)
5139 {
5140     struct buf *buf = bread(log.dev, log.start);
5141     struct logheader *lh = (struct logheader *) (buf->data);
5142     int i;
5143     log.lh.n = lh->n;
5144     for (i = 0; i < log.lh.n; i++) {
5145         log.lh.block[i] = lh->block[i];
5146     }
5147     brelse(buf);
5148 }
5149

```

```

5150 // Write in-memory log header to disk.
5151 // This is the true point at which the
5152 // current transaction commits.
5153 static void
5154 write_head(void)
5155 {
5156     struct buf *buf = bread(log.dev, log.start);
5157     struct logheader *hb = (struct logheader *) (buf->data);
5158     int i;
5159     hb->n = log.lh.n;
5160     for (i = 0; i < log.lh.n; i++) {
5161         hb->block[i] = log.lh.block[i];
5162     }
5163     bwrite(buf);
5164     brelse(buf);
5165 }
5166
5167 static void
5168 recover_from_log(void)
5169 {
5170     read_head();
5171     install_trans(); // if committed, copy from log to disk
5172     log.lh.n = 0;
5173     write_head(); // clear the log
5174 }
5175
5176 // called at the start of each FS system call.
5177 void
5178 begin_op(void)
5179 {
5180     acquire(&log.lock);
5181     while(1){
5182         if(log.committing){
5183             sleep(&log, &log.lock);
5184         } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
5185             // this op might exhaust log space; wait for commit.
5186             sleep(&log, &log.lock);
5187         } else {
5188             log.outstanding += 1;
5189             release(&log.lock);
5190             break;
5191         }
5192     }
5193 }
5194
5195
5196
5197
5198
5199

```

```

5200 // called at the end of each FS system call.
5201 // commits if this was the last outstanding operation.
5202 void
5203 end_op(void)
5204 {
5205     int do_commit = 0;
5206
5207     acquire(&log.lock);
5208     log.outstanding -= 1;
5209     if(log.committing)
5210         panic("log.committing");
5211     if(log.outstanding == 0){
5212         do_commit = 1;
5213         log.committing = 1;
5214     } else {
5215         // begin_op() may be waiting for log space.
5216         wakeup(&log);
5217     }
5218     release(&log.lock);
5219
5220     if(do_commit){
5221         // call commit w/o holding locks, since not allowed
5222         // to sleep with locks.
5223         commit();
5224         acquire(&log.lock);
5225         log.committing = 0;
5226         wakeup(&log);
5227         release(&log.lock);
5228     }
5229 }
5230
5231 // Copy modified blocks from cache to log.
5232 static void
5233 write_log(void)
5234 {
5235     int tail;
5236
5237     for (tail = 0; tail < log.lh.n; tail++) {
5238         struct buf *to = bread(log.dev, log.start+tail+1); // log block
5239         struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
5240         memmove(to->data, from->data, BSIZE);
5241         bwrite(to); // write the log
5242         brelse(from);
5243         brelse(to);
5244     }
5245 }
5246
5247
5248
5249

```

```

5250 static void
5251 commit()
5252 {
5253     if (log.lh.n > 0) {
5254         write_log(); // Write modified blocks from cache to log
5255         write_head(); // Write header to disk -- the real commit
5256         install_trans(); // Now install writes to home locations
5257         log.lh.n = 0;
5258         write_head(); // Erase the transaction from the log
5259     }
5260 }
5261
5262 // Caller has modified b->data and is done with the buffer.
5263 // Record the block number and pin in the cache with B_DIRTY.
5264 // commit()/write_log() will do the disk write.
5265 //
5266 // log_write() replaces bwrite(); a typical use is:
5267 //   bp = bread(...)
5268 //   modify bp->data[]
5269 //   log_write(bp)
5270 //   brelse(bp)
5271 void
5272 log_write(struct buf *b)
5273 {
5274     int i;
5275
5276     if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
5277         panic("too big a transaction");
5278     if (log.outstanding < 1)
5279         panic("log_write outside of trans");
5280
5281     acquire(&log.lock);
5282     for (i = 0; i < log.lh.n; i++) {
5283         if (log.lh.block[i] == b->blockno) // log absorption
5284             break;
5285     }
5286     log.lh.block[i] = b->blockno;
5287     if (i == log.lh.n)
5288         log.lh.n++;
5289     b->flags |= B_DIRTY; // prevent eviction
5290     release(&log.lock);
5291 }
5292
5293
5294
5295
5296
5297
5298
5299

```

```

5300 // File system implementation. Five layers:
5301 //   + Blocks: allocator for raw disk blocks.
5302 //   + Log: crash recovery for multi-step updates.
5303 //   + Files: inode allocator, reading, writing, metadata.
5304 //   + Directories: inode with special contents (list of other inodes!)
5305 //   + Names: paths like /usr/rtrm/xv6/fs.c for convenient naming.
5306 //
5307 // This file contains the low-level file system manipulation
5308 // routines. The (higher-level) system call implementations
5309 // are in sysfile.c.
5310
5311 #include "types.h"
5312 #include "defs.h"
5313 #include "param.h"
5314 #include "stat.h"
5315 #include "mmu.h"
5316 #include "proc.h"
5317 #include "spinlock.h"
5318 #include "fs.h"
5319 #include "buf.h"
5320 #include "file.h"
5321
5322 #define min(a, b) ((a) < (b) ? (a) : (b))
5323 static void itrunc(struct inode*);
5324 struct superblock sb; // there should be one per dev, but we run with one
5325
5326 // Read the super block.
5327 void
5328 readsb(int dev, struct superblock *sb)
5329 {
5330     struct buf *bp;
5331
5332     bp = bread(dev, 1);
5333     memmove(sb, bp->data, sizeof(*sb));
5334     brelse(bp);
5335 }
5336
5337 // Zero a block.
5338 static void
5339 bzero(int dev, int bno)
5340 {
5341     struct buf *bp;
5342
5343     bp = bread(dev, bno);
5344     memset(bp->data, 0, BSIZE);
5345     log_write(bp);
5346     brelse(bp);
5347 }
5348
5349

```

```

5350 // Blocks.
5351
5352 // Allocate a zeroed disk block.
5353 static uint
5354 balloc(uint dev)
5355 {
5356     int b, bi, m;
5357     struct buf *bp;
5358
5359     bp = 0;
5360     for(b = 0; b < sb.size; b += BPB){
5361         bp = bread(dev, BBLOCK(b, sb));
5362         for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
5363             m = 1 << (bi % 8);
5364             if((bp->data[bi/8] & m) == 0){ // Is block free?
5365                 bp->data[bi/8] |= m; // Mark block in use.
5366                 log_write(bp);
5367                 brelse(bp);
5368                 bzero(dev, b + bi);
5369                 return b + bi;
5370             }
5371         }
5372         brelse(bp);
5373     }
5374     panic("balloc: out of blocks");
5375 }
5376
5377 // Free a disk block.
5378 static void
5379 bfree(int dev, uint b)
5380 {
5381     struct buf *bp;
5382     int bi, m;
5383
5384     readsb(dev, &sb);
5385     bp = bread(dev, BBLOCK(b, sb));
5386     bi = b % BPB;
5387     m = 1 << (bi % 8);
5388     if((bp->data[bi/8] & m) == 0)
5389         panic("freeing free block");
5390     bp->data[bi/8] &= ~m;
5391     log_write(bp);
5392     brelse(bp);
5393 }
5394
5395
5396
5397
5398
5399

```

```

5400 // Inodes.
5401 //
5402 // An inode describes a single unnamed file.
5403 // The inode disk structure holds metadata: the file's type,
5404 // its size, the number of links referring to it, and the
5405 // list of blocks holding the file's content.
5406 //
5407 // The inodes are laid out sequentially on disk at
5408 // sb.startinode. Each inode has a number, indicating its
5409 // position on the disk.
5410 //
5411 // The kernel keeps a cache of in-use inodes in memory
5412 // to provide a place for synchronizing access
5413 // to inodes used by multiple processes. The cached
5414 // inodes include book-keeping information that is
5415 // not stored on disk: ip->ref and ip->flags.
5416 //
5417 // An inode and its in-memory representative go through a
5418 // sequence of states before they can be used by the
5419 // rest of the file system code.
5420 //
5421 // * Allocation: an inode is allocated if its type (on disk)
5422 //   is non-zero. ialloc() allocates, iput() frees if
5423 //   the link count has fallen to zero.
5424 //
5425 // * Referencing in cache: an entry in the inode cache
5426 //   is free if ip->ref is zero. Otherwise ip->ref tracks
5427 //   the number of in-memory pointers to the entry (open
5428 //   files and current directories). iget() to find or
5429 //   create a cache entry and increment its ref, iput()
5430 //   to decrement ref.
5431 //
5432 // * Valid: the information (type, size, &c) in an inode
5433 //   cache entry is only correct when the I_VALID bit
5434 //   is set in ip->flags. ilock() reads the inode from
5435 //   the disk and sets I_VALID, while iput() clears
5436 //   I_VALID if ip->ref has fallen to zero.
5437 //
5438 // * Locked: file system code may only examine and modify
5439 //   the information in an inode and its content if it
5440 //   has first locked the inode. The I_BUSY flag indicates
5441 //   that the inode is locked. ilock() sets I_BUSY,
5442 //   while iunlock clears it.
5443 //
5444 // Thus a typical sequence is:
5445 //   ip = iget(dev, inum)
5446 //   ilock(ip)
5447 //   ... examine and modify ip->xxx ...
5448 //   iunlock(ip)
5449 //   iput(ip)

```

```

5450 //
5451 // ilock() is separate from iget() so that system calls can
5452 // get a long-term reference to an inode (as for an open file)
5453 // and only lock it for short periods (e.g., in read()).
5454 // The separation also helps avoid deadlock and races during
5455 // pathname lookup. iget() increments ip->ref so that the inode
5456 // stays cached and pointers to it remain valid.
5457 //
5458 // Many internal file system functions expect the caller to
5459 // have locked the inodes involved; this lets callers create
5460 // multi-step atomic operations.
5461 //
5462 struct {
5463   struct spinlock lock;
5464   struct inode inode[NINODE];
5465 } icache;
5466
5467 void
5468 iinit(int dev)
5469 {
5470   initlock(&icache.lock, "icache");
5471   readsb(dev, &sb);
5472   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d inodestart %d\n",
5473          sb.nblocks, sb.ninodes, sb.nlog, sb.logstart, sb.inodestart, sb.bmap);
5474 }
5475
5476 static struct inode* iget(uint dev, uint inum);
5477
5478
5479
5480
5481
5482
5483
5484
5485
5486
5487
5488
5489
5490
5491
5492
5493
5494
5495
5496
5497
5498
5499

```

```

5500 // Allocate a new inode with the given type on device dev.
5501 // A free inode has a type of zero.
5502 struct inode*
5503 ialloc(uint dev, short type)
5504 {
5505     int inum;
5506     struct buf *bp;
5507     struct dinode *dip;
5508
5509     for(inum = 1; inum < sb.ninodes; inum++){
5510         bp = bread(dev, IBLOCK(inum, sb));
5511         dip = (struct dinode*)bp->data + inum%IPB;
5512         if(dip->type == 0){ // a free inode
5513             memset(dip, 0, sizeof(*dip));
5514             dip->type = type;
5515             log_write(bp); // mark it allocated on the disk
5516             brelse(bp);
5517             return iget(dev, inum);
5518         }
5519         brelse(bp);
5520     }
5521     panic("ialloc: no inodes");
5522 }
5523
5524 // Copy a modified in-memory inode to disk.
5525 void
5526 iupdate(struct inode *ip)
5527 {
5528     struct buf *bp;
5529     struct dinode *dip;
5530
5531     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5532     dip = (struct dinode*)bp->data + ip->inum%IPB;
5533     dip->type = ip->type;
5534     dip->major = ip->major;
5535     dip->minor = ip->minor;
5536     dip->nlink = ip->nlink;
5537     dip->size = ip->size;
5538     memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5539     log_write(bp);
5540     brelse(bp);
5541 }
5542
5543
5544
5545
5546
5547
5548
5549

```

```

5550 // Find the inode with number inum on device dev
5551 // and return the in-memory copy. Does not lock
5552 // the inode and does not read it from disk.
5553 static struct inode*
5554 iget(uint dev, uint inum)
5555 {
5556     struct inode *ip, *empty;
5557
5558     acquire(&icache.lock);
5559
5560     // Is the inode already cached?
5561     empty = 0;
5562     for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5563         if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5564             ip->ref++;
5565             release(&icache.lock);
5566             return ip;
5567         }
5568         if(empty == 0 && ip->ref == 0) // Remember empty slot.
5569             empty = ip;
5570     }
5571
5572     // Recycle an inode cache entry.
5573     if(empty == 0)
5574         panic("iget: no inodes");
5575
5576     ip = empty;
5577     ip->dev = dev;
5578     ip->inum = inum;
5579     ip->ref = 1;
5580     ip->flags = 0;
5581     release(&icache.lock);
5582
5583     return ip;
5584 }
5585
5586 // Increment reference count for ip.
5587 // Returns ip to enable ip = idup(ip1) idiom.
5588 struct inode*
5589 idup(struct inode *ip)
5590 {
5591     acquire(&icache.lock);
5592     ip->ref++;
5593     release(&icache.lock);
5594     return ip;
5595 }
5596
5597
5598
5599

```

```

5600 // Lock the given inode.
5601 // Reads the inode from disk if necessary.
5602 void
5603 ilock(struct inode *ip)
5604 {
5605     struct buf *bp;
5606     struct dinode *dip;
5607
5608     if(ip == 0 || ip->ref < 1)
5609         panic("ilock");
5610
5611     acquire(&icache.lock);
5612     while(ip->flags & I_BUSY)
5613         sleep(ip, &icache.lock);
5614     ip->flags |= I_BUSY;
5615     release(&icache.lock);
5616
5617     if(!(ip->flags & I_VALID)){
5618         bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5619         dip = (struct dinode*)bp->data + ip->inum*IPB;
5620         ip->type = dip->type;
5621         ip->major = dip->major;
5622         ip->minor = dip->minor;
5623         ip->nlink = dip->nlink;
5624         ip->size = dip->size;
5625         memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5626         brelse(bp);
5627         ip->flags |= I_VALID;
5628         if(ip->type == 0)
5629             panic("ilock: no type");
5630     }
5631 }
5632
5633 // Unlock the given inode.
5634 void
5635 iunlock(struct inode *ip)
5636 {
5637     if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
5638         panic("iunlock");
5639
5640     acquire(&icache.lock);
5641     ip->flags &= ~I_BUSY;
5642     wakeup(ip);
5643     release(&icache.lock);
5644 }
5645
5646
5647
5648
5649

```

```

5650 // Drop a reference to an in-memory inode.
5651 // If that was the last reference, the inode cache entry can
5652 // be recycled.
5653 // If that was the last reference and the inode has no links
5654 // to it, free the inode (and its content) on disk.
5655 // All calls to iput() must be inside a transaction in
5656 // case it has to free the inode.
5657 void
5658 iput(struct inode *ip)
5659 {
5660     acquire(&icache.lock);
5661     if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
5662         // inode has no links and no other references: truncate and free.
5663         if(ip->flags & I_BUSY)
5664             panic("iput busy");
5665         ip->flags |= I_BUSY;
5666         release(&icache.lock);
5667         itrunc(ip);
5668         ip->type = 0;
5669         iupdate(ip);
5670         acquire(&icache.lock);
5671         ip->flags = 0;
5672         wakeup(ip);
5673     }
5674     ip->ref--;
5675     release(&icache.lock);
5676 }
5677
5678 // Common idiom: unlock, then put.
5679 void
5680 iunlockput(struct inode *ip)
5681 {
5682     iunlock(ip);
5683     iput(ip);
5684 }
5685
5686
5687
5688
5689
5690
5691
5692
5693
5694
5695
5696
5697
5698
5699

```



```

5700 // Inode content
5701 //
5702 // The content (data) associated with each inode is stored
5703 // in blocks on the disk. The first NDIRECT block numbers
5704 // are listed in ip->addrs[]. The next NINDIRECT blocks are
5705 // listed in block ip->addrs[NDIRECT].
5706
5707 // Return the disk block address of the nth block in inode ip.
5708 // If there is no such block, bmap allocates one.
5709 static uint
5710 bmap(struct inode *ip, uint bn)
5711 {
5712     uint addr, *a;
5713     struct buf *bp;
5714
5715     if(bn < NDIRECT){
5716         if((addr = ip->addrs[bn]) == 0)
5717             ip->addrs[bn] = addr = balloc(ip->dev);
5718         return addr;
5719     }
5720     bn -= NDIRECT;
5721
5722     if(bn < NINDIRECT){
5723         // Load indirect block, allocating if necessary.
5724         if((addr = ip->addrs[NDIRECT]) == 0)
5725             ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5726         bp = bread(ip->dev, addr);
5727         a = (uint*)bp->data;
5728         if((addr = a[bn]) == 0){
5729             a[bn] = addr = balloc(ip->dev);
5730             log_write(bp);
5731         }
5732         brelse(bp);
5733         return addr;
5734     }
5735
5736     panic("bmap: out of range");
5737 }
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749

```

```

5750 // Truncate inode (discard contents).
5751 // Only called when the inode has no links
5752 // to it (no directory entries referring to it)
5753 // and has no in-memory reference to it (is
5754 // not an open file or current directory).
5755 static void
5756 itrunc(struct inode *ip)
5757 {
5758     int i, j;
5759     struct buf *bp;
5760     uint *a;
5761
5762     for(i = 0; i < NDIRECT; i++){
5763         if(ip->addrs[i]){
5764             bfree(ip->dev, ip->addrs[i]);
5765             ip->addrs[i] = 0;
5766         }
5767     }
5768
5769     if(ip->addrs[NDIRECT]){
5770         bp = bread(ip->dev, ip->addrs[NDIRECT]);
5771         a = (uint*)bp->data;
5772         for(j = 0; j < NINDIRECT; j++){
5773             if(a[j])
5774                 bfree(ip->dev, a[j]);
5775         }
5776         brelse(bp);
5777         bfree(ip->dev, ip->addrs[NDIRECT]);
5778         ip->addrs[NDIRECT] = 0;
5779     }
5780
5781     ip->size = 0;
5782     iupdate(ip);
5783 }
5784
5785 // Copy stat information from inode.
5786 void
5787 stati(struct inode *ip, struct stat *st)
5788 {
5789     st->dev = ip->dev;
5790     st->ino = ip->inum;
5791     st->type = ip->type;
5792     st->nlink = ip->nlink;
5793     st->size = ip->size;
5794 }
5795
5796
5797
5798
5799

```

```

5800 // Read data from inode.
5801 int
5802 readi(struct inode *ip, char *dst, uint off, uint n)
5803 {
5804     uint tot, m;
5805     struct buf *bp;
5806
5807     if(ip->type == T_DEV){
5808         if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
5809             return -1;
5810         return devsw[ip->major].read(ip, dst, n);
5811     }
5812
5813     if(off > ip->size || off + n < off)
5814         return -1;
5815     if(off + n > ip->size)
5816         n = ip->size - off;
5817
5818     for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5819         bp = bread(ip->dev, bmap(ip, off/BSIZE));
5820         m = min(n - tot, BSIZE - off%BSIZE);
5821         memmove(dst, bp->data + off%BSIZE, m);
5822         brelse(bp);
5823     }
5824     return n;
5825 }
5826
5827
5828
5829
5830
5831
5832
5833
5834
5835
5836
5837
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849

```

```

5850 // Write data to inode.
5851 int
5852 writei(struct inode *ip, char *src, uint off, uint n)
5853 {
5854     uint tot, m;
5855     struct buf *bp;
5856
5857     if(ip->type == T_DEV){
5858         if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
5859             return -1;
5860         return devsw[ip->major].write(ip, src, n);
5861     }
5862
5863     if(off > ip->size || off + n < off)
5864         return -1;
5865     if(off + n > MAXFILE*BSIZE)
5866         return -1;
5867
5868     for(tot=0; tot<n; tot+=m, off+=m, src+=m){
5869         bp = bread(ip->dev, bmap(ip, off/BSIZE));
5870         m = min(n - tot, BSIZE - off%BSIZE);
5871         memmove(bp->data + off%BSIZE, src, m);
5872         log_write(bp);
5873         brelse(bp);
5874     }
5875
5876     if(n > 0 && off > ip->size){
5877         ip->size = off;
5878         iupdate(ip);
5879     }
5880     return n;
5881 }
5882
5883
5884
5885
5886
5887
5888
5889
5890
5891
5892
5893
5894
5895
5896
5897
5898
5899

```

```

5900 // Directories
5901
5902 int
5903 namecmp(const char *s, const char *t)
5904 {
5905     return strncmp(s, t, DIRSIZ);
5906 }
5907
5908 // Look for a directory entry in a directory.
5909 // If found, set *poff to byte offset of entry.
5910 struct inode*
5911 dirlookup(struct inode *dp, char *name, uint *poff)
5912 {
5913     uint off, inum;
5914     struct dirent de;
5915
5916     if(dp->type != T_DIR)
5917         panic("dirlookup not DIR");
5918
5919     for(off = 0; off < dp->size; off += sizeof(de)){
5920         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5921             panic("dirlink read");
5922         if(de.inum == 0)
5923             continue;
5924         if(namecmp(name, de.name) == 0){
5925             // entry matches path element
5926             if(poff)
5927                 *poff = off;
5928             inum = de.inum;
5929             return iget(dp->dev, inum);
5930         }
5931     }
5932
5933     return 0;
5934 }
5935
5936
5937
5938
5939
5940
5941
5942
5943
5944
5945
5946
5947
5948
5949

```

```

5950 // Write a new directory entry (name, inum) into the directory dp.
5951 int
5952 dirlink(struct inode *dp, char *name, uint inum)
5953 {
5954     int off;
5955     struct dirent de;
5956     struct inode *ip;
5957
5958     // Check that name is not present.
5959     if((ip = dirlookup(dp, name, 0)) != 0){
5960         iput(ip);
5961         return -1;
5962     }
5963
5964     // Look for an empty dirent.
5965     for(off = 0; off < dp->size; off += sizeof(de)){
5966         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5967             panic("dirlink read");
5968         if(de.inum == 0)
5969             break;
5970     }
5971
5972     strncpy(de.name, name, DIRSIZ);
5973     de.inum = inum;
5974     if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5975         panic("dirlink");
5976
5977     return 0;
5978 }
5979
5980
5981
5982
5983
5984
5985
5986
5987
5988
5989
5990
5991
5992
5993
5994
5995
5996
5997
5998
5999

```

```

6000 // Paths
6001
6002 // Copy the next path element from path into name.
6003 // Return a pointer to the element following the copied one.
6004 // The returned path has no leading slashes,
6005 // so the caller can check *path=='\0' to see if the name is the last one.
6006 // If no name to remove, return 0.
6007 //
6008 // Examples:
6009 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
6010 //   skipelem("///a//bb", name) = "bb", setting name = "a"
6011 //   skipelem("a", name) = "", setting name = "a"
6012 //   skipelem("", name) = skipelem("///", name) = 0
6013 //
6014 static char*
6015 skipelem(char *path, char *name)
6016 {
6017     char *s;
6018     int len;
6019
6020     while(*path == '/')
6021         path++;
6022     if(*path == 0)
6023         return 0;
6024     s = path;
6025     while(*path != '/' && *path != 0)
6026         path++;
6027     len = path - s;
6028     if(len >= DIRSIZ)
6029         memmove(name, s, DIRSIZ);
6030     else {
6031         memmove(name, s, len);
6032         name[len] = 0;
6033     }
6034     while(*path == '/')
6035         path++;
6036     return path;
6037 }
6038
6039
6040
6041
6042
6043
6044
6045
6046
6047
6048
6049

```

```

6050 // Look up and return the inode for a path name.
6051 // If parent != 0, return the inode for the parent and copy the final
6052 // path element into name, which must have room for DIRSIZ bytes.
6053 // Must be called inside a transaction since it calls iput().
6054 static struct inode*
6055 namex(char *path, int nameiparent, char *name)
6056 {
6057     struct inode *ip, *next;
6058
6059     if(*path == '/')
6060         ip = iget(ROOTDEV, ROOTINO);
6061     else
6062         ip = idup(proc->cwd);
6063
6064     while((path = skipelem(path, name)) != 0){
6065         ilock(ip);
6066         if(ip->type != T_DIR){
6067             iunlockput(ip);
6068             return 0;
6069         }
6070         if(nameiparent && *path == '\0'){
6071             // Stop one level early.
6072             iunlock(ip);
6073             return ip;
6074         }
6075         if((next = dirlookup(ip, name, 0)) == 0){
6076             iunlockput(ip);
6077             return 0;
6078         }
6079         iunlockput(ip);
6080         ip = next;
6081     }
6082     if(nameiparent){
6083         iput(ip);
6084         return 0;
6085     }
6086     return ip;
6087 }
6088
6089 struct inode*
6090 namei(char *path)
6091 {
6092     char name[DIRSIZ];
6093     return namex(path, 0, name);
6094 }
6095
6096
6097
6098
6099

```

```

6100 struct inode*
6101 nameiparent(char *path, char *name)
6102 {
6103     return namex(path, 1, name);
6104 }
6105
6106
6107
6108
6109
6110
6111
6112
6113
6114
6115
6116
6117
6118
6119
6120
6121
6122
6123
6124
6125
6126
6127
6128
6129
6130
6131
6132
6133
6134
6135
6136
6137
6138
6139
6140
6141
6142
6143
6144
6145
6146
6147
6148
6149

```

```

6150 //
6151 // File descriptors
6152 //
6153
6154 #include "types.h"
6155 #include "defs.h"
6156 #include "param.h"
6157 #include "fs.h"
6158 #include "file.h"
6159 #include "spinlock.h"
6160
6161 struct devsw devsw[NDEV];
6162 struct {
6163     struct spinlock lock;
6164     struct file file[NFILE];
6165 } ftable;
6166
6167 void
6168 fileinit(void)
6169 {
6170     initlock(&ftable.lock, "ftable");
6171 }
6172
6173 // Allocate a file structure.
6174 struct file*
6175 filealloc(void)
6176 {
6177     struct file *f;
6178
6179     acquire(&ftable.lock);
6180     for(f = ftable.file; f < ftable.file + NFILE; f++){
6181         if(f->ref == 0){
6182             f->ref = 1;
6183             release(&ftable.lock);
6184             return f;
6185         }
6186     }
6187     release(&ftable.lock);
6188     return 0;
6189 }
6190
6191
6192
6193
6194
6195
6196
6197
6198
6199

```

```

6200 // Increment ref count for file f.
6201 struct file*
6202 filedup(struct file *f)
6203 {
6204     acquire(&ftable.lock);
6205     if(f->ref < 1)
6206         panic("filedup");
6207     f->ref++;
6208     release(&ftable.lock);
6209     return f;
6210 }
6211
6212 // Close file f. (Decrement ref count, close when reaches 0.)
6213 void
6214 fileclose(struct file *f)
6215 {
6216     struct file ff;
6217
6218     acquire(&ftable.lock);
6219     if(f->ref < 1)
6220         panic("fileclose");
6221     if(--f->ref > 0){
6222         release(&ftable.lock);
6223         return;
6224     }
6225     ff = *f;
6226     f->ref = 0;
6227     f->type = FD_NONE;
6228     release(&ftable.lock);
6229
6230     if(ff.type == FD_PIPE)
6231         pipeclose(ff.pipe, ff.writable);
6232     else if(ff.type == FD_INODE){
6233         begin_op();
6234         iput(ff.ip);
6235         end_op();
6236     }
6237 }
6238
6239
6240
6241
6242
6243
6244
6245
6246
6247
6248
6249

```

```

6250 // Get metadata about file f.
6251 int
6252 filestat(struct file *f, struct stat *st)
6253 {
6254     if(f->type == FD_INODE){
6255         ilock(f->ip);
6256         stati(f->ip, st);
6257         iunlock(f->ip);
6258         return 0;
6259     }
6260     return -1;
6261 }
6262
6263 // Read from file f.
6264 int
6265 fileread(struct file *f, char *addr, int n)
6266 {
6267     int r;
6268
6269     if(f->readable == 0)
6270         return -1;
6271     if(f->type == FD_PIPE)
6272         return piperead(f->pipe, addr, n);
6273     if(f->type == FD_INODE){
6274         ilock(f->ip);
6275         if((r = readi(f->ip, addr, f->off, n)) > 0)
6276             f->off += r;
6277         iunlock(f->ip);
6278         return r;
6279     }
6280     panic("fileread");
6281 }
6282
6283
6284
6285
6286
6287
6288
6289
6290
6291
6292
6293
6294
6295
6296
6297
6298
6299

```

```

6300 // Write to file f.
6301 int
6302 filewrite(struct file *f, char *addr, int n)
6303 {
6304     int r;
6305
6306     if(f->writable == 0)
6307         return -1;
6308     if(f->type == FD_PIPE)
6309         return pipewrite(f->pipe, addr, n);
6310     if(f->type == FD_INODE){
6311         // write a few blocks at a time to avoid exceeding
6312         // the maximum log transaction size, including
6313         // i-node, indirect block, allocation blocks,
6314         // and 2 blocks of slop for non-aligned writes.
6315         // this really belongs lower down, since writei()
6316         // might be writing a device like the console.
6317         int max = ((LOGSIZE-1-1-2) / 2) * 512;
6318         int i = 0;
6319         while(i < n){
6320             int n1 = n - i;
6321             if(n1 > max)
6322                 n1 = max;
6323
6324             begin_op();
6325             ilock(f->ip);
6326             if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
6327                 f->off += r;
6328             iunlock(f->ip);
6329             end_op();
6330
6331             if(r < 0)
6332                 break;
6333             if(r != n1)
6334                 panic("short filewrite");
6335             i += r;
6336         }
6337         return i == n ? n : -1;
6338     }
6339     panic("filewrite");
6340 }
6341
6342
6343
6344
6345
6346
6347
6348
6349

```

```

6350 //
6351 // File-system system calls.
6352 // Mostly argument checking, since we don't trust
6353 // user code, and calls into file.c and fs.c.
6354 //
6355
6356 #include "types.h"
6357 #include "defs.h"
6358 #include "param.h"
6359 #include "stat.h"
6360 #include "mmu.h"
6361 #include "proc.h"
6362 #include "fs.h"
6363 #include "file.h"
6364 #include "fcntl.h"
6365
6366 // Fetch the nth word-sized system call argument as a file descriptor
6367 // and return both the descriptor and the corresponding struct file.
6368 static int
6369 argfd(int n, int *pfd, struct file **pf)
6370 {
6371     int fd;
6372     struct file *f;
6373
6374     if(argint(n, &fd) < 0)
6375         return -1;
6376     if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
6377         return -1;
6378     if(pfd)
6379         *pfd = fd;
6380     if(pf)
6381         *pf = f;
6382     return 0;
6383 }
6384
6385 // Allocate a file descriptor for the given file.
6386 // Takes over file reference from caller on success.
6387 static int
6388 fdalloc(struct file *f)
6389 {
6390     int fd;
6391
6392     for(fd = 0; fd < NOFILE; fd++){
6393         if(proc->ofile[fd] == 0){
6394             proc->ofile[fd] = f;
6395             return fd;
6396         }
6397     }
6398     return -1;
6399 }

```

```

6400 int
6401 sys_dup(void)
6402 {
6403     struct file *f;
6404     int fd;
6405
6406     if(argfd(0, 0, &f) < 0)
6407         return -1;
6408     if((fd=fdalloc(f)) < 0)
6409         return -1;
6410     filedup(f);
6411     return fd;
6412 }
6413
6414 int
6415 sys_read(void)
6416 {
6417     struct file *f;
6418     int n;
6419     char *p;
6420
6421     if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6422         return -1;
6423     return fileread(f, p, n);
6424 }
6425
6426 int
6427 sys_write(void)
6428 {
6429     struct file *f;
6430     int n;
6431     char *p;
6432
6433     if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6434         return -1;
6435     return filewrite(f, p, n);
6436 }
6437
6438 int
6439 sys_close(void)
6440 {
6441     int fd;
6442     struct file *f;
6443
6444     if(argfd(0, &fd, &f) < 0)
6445         return -1;
6446     proc->ofile[fd] = 0;
6447     fileclose(f);
6448     return 0;
6449 }

```

```

6450 int
6451 sys_fstat(void)
6452 {
6453     struct file *f;
6454     struct stat *st;
6455
6456     if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
6457         return -1;
6458     return filestat(f, st);
6459 }
6460
6461 // Create the path new as a link to the same inode as old.
6462 int
6463 sys_link(void)
6464 {
6465     char name[DIRSIZ], *new, *old;
6466     struct inode *dp, *ip;
6467
6468     if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
6469         return -1;
6470
6471     begin_op();
6472     if((ip = namei(old)) == 0){
6473         end_op();
6474         return -1;
6475     }
6476
6477     ilock(ip);
6478     if(ip->type == T_DIR){
6479         iunlockput(ip);
6480         end_op();
6481         return -1;
6482     }
6483
6484     ip->nlink++;
6485     iupdate(ip);
6486     iunlock(ip);
6487
6488     if((dp = nameiparent(new, name)) == 0)
6489         goto bad;
6490     ilock(dp);
6491     if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
6492         iunlockput(dp);
6493         goto bad;
6494     }
6495     iunlockput(dp);
6496     iput(ip);
6497
6498     end_op();
6499 }

```



```

6500     return 0;
6501
6502 bad:
6503     ilock(ip);
6504     ip->nlink--;
6505     iupdate(ip);
6506     iunlockput(ip);
6507     end_op();
6508     return -1;
6509 }
6510
6511 // Is the directory dp empty except for "." and ".." ?
6512 static int
6513 isdirempty(struct inode *dp)
6514 {
6515     int off;
6516     struct dirent de;
6517
6518     for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
6519         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6520             panic("isdirempty: readi");
6521         if(de.inum != 0)
6522             return 0;
6523     }
6524     return 1;
6525 }
6526
6527
6528
6529
6530
6531
6532
6533
6534
6535
6536
6537
6538
6539
6540
6541
6542
6543
6544
6545
6546
6547
6548
6549

```

```

6550 int
6551 sys_unlink(void)
6552 {
6553     struct inode *ip, *dp;
6554     struct dirent de;
6555     char name[DIRSIZ], *path;
6556     uint off;
6557
6558     if(argstr(0, &path) < 0)
6559         return -1;
6560
6561     begin_op();
6562     if((dp = nameiparent(path, name)) == 0){
6563         end_op();
6564         return -1;
6565     }
6566
6567     ilock(dp);
6568
6569     // Cannot unlink "." or "..".
6570     if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6571         goto bad;
6572
6573     if((ip = dirlookup(dp, name, &off)) == 0)
6574         goto bad;
6575     ilock(ip);
6576
6577     if(ip->nlink < 1)
6578         panic("unlink: nlink < 1");
6579     if(ip->type == T_DIR && !isdirempty(ip)){
6580         iunlockput(ip);
6581         goto bad;
6582     }
6583
6584     memset(&de, 0, sizeof(de));
6585     if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6586         panic("unlink: writei");
6587     if(ip->type == T_DIR){
6588         dp->nlink--;
6589         iupdate(dp);
6590     }
6591     iunlockput(dp);
6592
6593     ip->nlink--;
6594     iupdate(ip);
6595     iunlockput(ip);
6596
6597     end_op();
6598
6599     return 0;

```

```

6600 bad:
6601   iunlockput(dp);
6602   end_op();
6603   return -1;
6604 }
6605
6606 static struct inode*
6607 create(char *path, short type, short major, short minor)
6608 {
6609     uint off;
6610     struct inode *ip, *dp;
6611     char name[DIRSIZ];
6612
6613     if((dp = nameiparent(path, name)) == 0)
6614         return 0;
6615     ilock(dp);
6616
6617     if((ip = dirlookup(dp, name, &off)) != 0){
6618         iunlockput(dp);
6619         ilock(ip);
6620         if(type == T_FILE && ip->type == T_FILE)
6621             return ip;
6622         iunlockput(ip);
6623         return 0;
6624     }
6625
6626     if((ip = ialloc(dp->dev, type)) == 0)
6627         panic("create: ialloc");
6628
6629     ilock(ip);
6630     ip->major = major;
6631     ip->minor = minor;
6632     ip->nlink = 1;
6633     iupdate(ip);
6634
6635     if(type == T_DIR){ // Create . and .. entries.
6636         dp->nlink++; // for ".."
6637         iupdate(dp);
6638         // No ip->nlink++ for "."; avoid cyclic ref count.
6639         if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6640             panic("create dots");
6641     }
6642
6643     if(dirlink(dp, name, ip->inum) < 0)
6644         panic("create: dirlink");
6645
6646     iunlockput(dp);
6647
6648     return ip;
6649 }

```

```

6650 int
6651 sys_open(void)
6652 {
6653     char *path;
6654     int fd, omode;
6655     struct file *f;
6656     struct inode *ip;
6657
6658     if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6659         return -1;
6660
6661     begin_op();
6662
6663     if(omode & O_CREATE){
6664         ip = create(path, T_FILE, 0, 0);
6665         if(ip == 0){
6666             end_op();
6667             return -1;
6668         }
6669     } else {
6670         if((ip = namei(path)) == 0){
6671             end_op();
6672             return -1;
6673         }
6674         ilock(ip);
6675         if(ip->type == T_DIR && omode != O_RDONLY){
6676             iunlockput(ip);
6677             end_op();
6678             return -1;
6679         }
6680     }
6681
6682     if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6683         if(f)
6684             fileclose(f);
6685         iunlockput(ip);
6686         end_op();
6687         return -1;
6688     }
6689     iunlock(ip);
6690     end_op();
6691
6692     f->type = FD_INODE;
6693     f->ip = ip;
6694     f->off = 0;
6695     f->readable = !(omode & O_WRONLY);
6696     f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6697     return fd;
6698 }
6699

```

```

6700 int
6701 sys_mkdir(void)
6702 {
6703     char *path;
6704     struct inode *ip;
6705
6706     begin_op();
6707     if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
6708         end_op();
6709         return -1;
6710     }
6711     iunlockput(ip);
6712     end_op();
6713     return 0;
6714 }
6715
6716 int
6717 sys_mknod(void)
6718 {
6719     struct inode *ip;
6720     char *path;
6721     int len;
6722     int major, minor;
6723
6724     begin_op();
6725     if((len=argstr(0, &path)) < 0 ||
6726         argint(1, &major) < 0 ||
6727         argint(2, &minor) < 0 ||
6728         (ip = create(path, T_DEV, major, minor)) == 0){
6729         end_op();
6730         return -1;
6731     }
6732     iunlockput(ip);
6733     end_op();
6734     return 0;
6735 }
6736
6737
6738
6739
6740
6741
6742
6743
6744
6745
6746
6747
6748
6749

```

```

6750 int
6751 sys_chdir(void)
6752 {
6753     char *path;
6754     struct inode *ip;
6755
6756     begin_op();
6757     if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
6758         end_op();
6759         return -1;
6760     }
6761     ilock(ip);
6762     if(ip->type != T_DIR){
6763         iunlockput(ip);
6764         end_op();
6765         return -1;
6766     }
6767     iunlock(ip);
6768     iput(proc->cwd);
6769     end_op();
6770     proc->cwd = ip;
6771     return 0;
6772 }
6773
6774 int
6775 sys_exec(void)
6776 {
6777     char *path, *argv[MAXARG];
6778     int i;
6779     uint uargv, uarg;
6780
6781     if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6782         return -1;
6783     }
6784     memset(argv, 0, sizeof(argv));
6785     for(i=0; i++){
6786         if(i >= NELEM(argv))
6787             return -1;
6788         if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6789             return -1;
6790         if(uarg == 0){
6791             argv[i] = 0;
6792             break;
6793         }
6794         if(fetchstr(uarg, &argv[i]) < 0)
6795             return -1;
6796     }
6797     return exec(path, argv);
6798 }
6799

```

```

6800 int
6801 sys_pipe(void)
6802 {
6803     int *fd;
6804     struct file *rf, *wf;
6805     int fd0, fd1;
6806
6807     if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6808         return -1;
6809     if(pipealloc(&rf, &wf) < 0)
6810         return -1;
6811     fd0 = -1;
6812     if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6813         if(fd0 >= 0)
6814             proc->ofile[fd0] = 0;
6815         fileclose(rf);
6816         fileclose(wf);
6817         return -1;
6818     }
6819     fd[0] = fd0;
6820     fd[1] = fd1;
6821     return 0;
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 #include "types.h"
6851 #include "param.h"
6852 #include "memlayout.h"
6853 #include "mmu.h"
6854 #include "proc.h"
6855 #include "defs.h"
6856 #include "x86.h"
6857 #include "elf.h"
6858
6859 int
6860 exec(char *path, char **argv)
6861 {
6862     char *s, *last;
6863     int i, off;
6864     uint argc, sz, sp, ustack[3+MAXARG+1];
6865     struct elfhdr elf;
6866     struct inode *ip;
6867     struct proghdr ph;
6868     pde_t *pgdir, *oldpgdir;
6869
6870     begin_op();
6871     if((ip = namei(path)) == 0){
6872         end_op();
6873         return -1;
6874     }
6875     ilock(ip);
6876     pgdir = 0;
6877
6878     // Check ELF header
6879     if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
6880         goto bad;
6881     if(elf.magic != ELF_MAGIC)
6882         goto bad;
6883
6884     if((pgdir = setupkvm()) == 0)
6885         goto bad;
6886
6887     // Load program into memory.
6888     sz = 0;
6889     for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6890         if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6891             goto bad;
6892         if(ph.type != ELF_PROG_LOAD)
6893             continue;
6894         if(ph.memsz < ph.filesz)
6895             goto bad;
6896         if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6897             goto bad;
6898         if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6899             goto bad;

```

```

6900 }
6901 iunlockput(ip);
6902 end_op();
6903 ip = 0;
6904
6905 // Allocate two pages at the next page boundary.
6906 // Make the first inaccessible. Use the second as the user stack.
6907 sz = PGROUNDUP(sz);
6908 if((sz = allocuvvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
6909     goto bad;
6910 clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6911 sp = sz;
6912
6913 // Push argument strings, prepare rest of stack in ustack.
6914 for(argc = 0; argv[argc]; argc++) {
6915     if(argc >= MAXARG)
6916         goto bad;
6917     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6918     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6919         goto bad;
6920     ustack[3+argc] = sp;
6921 }
6922 ustack[3+argc] = 0;
6923
6924 ustack[0] = 0xffffffff; // fake return PC
6925 ustack[1] = argc;
6926 ustack[2] = sp - (argc+1)*4; // argv pointer
6927
6928 sp -= (3+argc+1) * 4;
6929 if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6930     goto bad;
6931
6932 // Save program name for debugging.
6933 for(last=s=path; *s; s++)
6934     if(*s == '/')
6935         last = s+1;
6936 safestrcpy(proc->name, last, sizeof(proc->name));
6937
6938 // Commit to the user image.
6939 oldpgdir = proc->pgdir;
6940 proc->pgdir = pgdir;
6941 proc->sz = sz;
6942 proc->tf->eip = elf.entry; // main
6943 proc->tf->esp = sp;
6944 switchvm(proc);
6945 freevm(oldpgdir);
6946 return 0;
6947
6948
6949

```

```

6950 bad:
6951     if(pgdir)
6952         freevm(pgdir);
6953     if(ip){
6954         iunlockput(ip);
6955         end_op();
6956     }
6957     return -1;
6958 }
6959
6960
6961
6962
6963
6964
6965
6966
6967
6968
6969
6970
6971
6972
6973
6974
6975
6976
6977
6978
6979
6980
6981
6982
6983
6984
6985
6986
6987
6988
6989
6990
6991
6992
6993
6994
6995
6996
6997
6998
6999

```

```

7000 #include "types.h"
7001 #include "defs.h"
7002 #include "param.h"
7003 #include "mmu.h"
7004 #include "proc.h"
7005 #include "fs.h"
7006 #include "file.h"
7007 #include "spinlock.h"
7008
7009 #define PIPESIZE 512
7010
7011 struct pipe {
7012     struct spinlock lock;
7013     char data[PIPESIZE];
7014     uint nread;    // number of bytes read
7015     uint nwrite;   // number of bytes written
7016     int readopen;  // read fd is still open
7017     int writeopen; // write fd is still open
7018 };
7019
7020 int
7021 pipealloc(struct file **f0, struct file **f1)
7022 {
7023     struct pipe *p;
7024
7025     p = 0;
7026     *f0 = *f1 = 0;
7027     if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
7028         goto bad;
7029     if((p = (struct pipe*)kalloc()) == 0)
7030         goto bad;
7031     p->readopen = 1;
7032     p->writeopen = 1;
7033     p->nwrite = 0;
7034     p->nread = 0;
7035     initlock(&p->lock, "pipe");
7036     (*f0)->type = FD_PIPE;
7037     (*f0)->readable = 1;
7038     (*f0)->writable = 0;
7039     (*f0)->pipe = p;
7040     (*f1)->type = FD_PIPE;
7041     (*f1)->readable = 0;
7042     (*f1)->writable = 1;
7043     (*f1)->pipe = p;
7044     return 0;
7045
7046
7047
7048
7049

```

```

7050 bad:
7051     if(p)
7052         kfree((char*)p);
7053     if(*f0)
7054         fileclose(*f0);
7055     if(*f1)
7056         fileclose(*f1);
7057     return -1;
7058 }
7059
7060 void
7061 pipeclose(struct pipe *p, int writable)
7062 {
7063     acquire(&p->lock);
7064     if(writable){
7065         p->writeopen = 0;
7066         wakeup(&p->nread);
7067     } else {
7068         p->readopen = 0;
7069         wakeup(&p->nwrite);
7070     }
7071     if(p->readopen == 0 && p->writeopen == 0){
7072         release(&p->lock);
7073         kfree((char*)p);
7074     } else
7075         release(&p->lock);
7076 }
7077
7078 int
7079 pipewrite(struct pipe *p, char *addr, int n)
7080 {
7081     int i;
7082
7083     acquire(&p->lock);
7084     for(i = 0; i < n; i++){
7085         while(p->nwrite == p->nread + PIPESIZE){
7086             if(p->readopen == 0 || proc->killed){
7087                 release(&p->lock);
7088                 return -1;
7089             }
7090             wakeup(&p->nread);
7091             sleep(&p->nwrite, &p->lock);
7092         }
7093         p->data[p->nwrite++ % PIPESIZE] = addr[i];
7094     }
7095     wakeup(&p->nread);
7096     release(&p->lock);
7097     return n;
7098 }
7099

```

```

7100 int
7101 piperead(struct pipe *p, char *addr, int n)
7102 {
7103     int i;
7104
7105     acquire(&p->lock);
7106     while(p->nread == p->nwrite && p->writeopen){
7107         if(proc->killed){
7108             release(&p->lock);
7109             return -1;
7110         }
7111         sleep(&p->nread, &p->lock);
7112     }
7113     for(i = 0; i < n; i++){
7114         if(p->nread == p->nwrite)
7115             break;
7116         addr[i] = p->data[p->nread++ % PIPESIZE];
7117     }
7118     wakeup(&p->nwrite);
7119     release(&p->lock);
7120     return i;
7121 }
7122
7123
7124
7125
7126
7127
7128
7129
7130
7131
7132
7133
7134
7135
7136
7137
7138
7139
7140
7141
7142
7143
7144
7145
7146
7147
7148
7149

```

```

7150 #include "types.h"
7151 #include "x86.h"
7152
7153 void*
7154 memset(void *dst, int c, uint n)
7155 {
7156     if ((int)dst%4 == 0 && n%4 == 0){
7157         c &= 0xFF;
7158         stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
7159     } else
7160         stosb(dst, c, n);
7161     return dst;
7162 }
7163
7164 int
7165 memcmp(const void *v1, const void *v2, uint n)
7166 {
7167     const uchar *s1, *s2;
7168
7169     s1 = v1;
7170     s2 = v2;
7171     while(n-- > 0){
7172         if(*s1 != *s2)
7173             return *s1 - *s2;
7174         s1++, s2++;
7175     }
7176
7177     return 0;
7178 }
7179
7180 void*
7181 memmove(void *dst, const void *src, uint n)
7182 {
7183     const char *s;
7184     char *d;
7185
7186     s = src;
7187     d = dst;
7188     if(s < d && s + n > d){
7189         s += n;
7190         d += n;
7191         while(n-- > 0)
7192             *--d = *--s;
7193     } else
7194         while(n-- > 0)
7195             *d++ = *s++;
7196
7197     return dst;
7198 }
7199

```

```

7200 // memcpy exists to placate GCC. Use memmove.
7201 void*
7202 memcpy(void *dst, const void *src, uint n)
7203 {
7204     return memmove(dst, src, n);
7205 }
7206
7207 int
7208 strncmp(const char *p, const char *q, uint n)
7209 {
7210     while(n > 0 && *p && *p == *q)
7211         n--, p++, q++;
7212     if(n == 0)
7213         return 0;
7214     return (uchar)*p - (uchar)*q;
7215 }
7216
7217 char*
7218 strncpy(char *s, const char *t, int n)
7219 {
7220     char *os;
7221
7222     os = s;
7223     while(n-- > 0 && (*s++ = *t++) != 0)
7224         ;
7225     while(n-- > 0)
7226         *s++ = 0;
7227     return os;
7228 }
7229
7230 // Like strncpy but guaranteed to NUL-terminate.
7231 char*
7232 safestrcpy(char *s, const char *t, int n)
7233 {
7234     char *os;
7235
7236     os = s;
7237     if(n <= 0)
7238         return os;
7239     while(--n > 0 && (*s++ = *t++) != 0)
7240         ;
7241     *s = 0;
7242     return os;
7243 }
7244
7245
7246
7247
7248
7249

```

```

7250 int
7251 strlen(const char *s)
7252 {
7253     int n;
7254
7255     for(n = 0; s[n]; n++)
7256         ;
7257     return n;
7258 }
7259
7260
7261
7262
7263
7264
7265
7266
7267
7268
7269
7270
7271
7272
7273
7274
7275
7276
7277
7278
7279
7280
7281
7282
7283
7284
7285
7286
7287
7288
7289
7290
7291
7292
7293
7294
7295
7296
7297
7298
7299

```



```

7300 // See MultiProcessor Specification Version 1.[14]
7301
7302 struct mp {           // floating pointer
7303     uchar signature[4]; // "_MP_"
7304     void *physaddr;     // phys addr of MP config table
7305     uchar length;       // 1
7306     uchar specrev;      // [14]
7307     uchar checksum;     // all bytes must add up to 0
7308     uchar type;         // MP system config type
7309     uchar imcrp;
7310     uchar reserved[3];
7311 };
7312
7313 struct mpconf {       // configuration table header
7314     uchar signature[4]; // "PCMP"
7315     ushort length;      // total table length
7316     uchar version;      // [14]
7317     uchar checksum;     // all bytes must add up to 0
7318     uchar product[20];  // product id
7319     uint *oemtable;     // OEM table pointer
7320     ushort oemlength;   // OEM table length
7321     ushort entry;       // entry count
7322     uint *lapicaddr;    // address of local APIC
7323     ushort xlength;     // extended table length
7324     uchar xchecksum;    // extended table checksum
7325     uchar reserved;
7326 };
7327
7328 struct mpproc {       // processor table entry
7329     uchar type;         // entry type (0)
7330     uchar apicid;       // local APIC id
7331     uchar version;      // local APIC verison
7332     uchar flags;        // CPU flags
7333     #define MPBOOT 0x02 // This proc is the bootstrap processor.
7334     uchar signature[4]; // CPU signature
7335     uint feature;       // feature flags from CPUID instruction
7336     uchar reserved[8];
7337 };
7338
7339 struct mpioapic {     // I/O APIC table entry
7340     uchar type;         // entry type (2)
7341     uchar apicno;       // I/O APIC id
7342     uchar version;      // I/O APIC version
7343     uchar flags;        // I/O APIC flags
7344     uint *addr;         // I/O APIC address
7345 };
7346
7347
7348
7349

```

```

7350 // Table entry types
7351 #define MPPROC 0x00 // One per processor
7352 #define MPBUS 0x01 // One per bus
7353 #define MPIOAPIC 0x02 // One per I/O APIC
7354 #define MPIOINTR 0x03 // One per bus interrupt source
7355 #define MPLINTR 0x04 // One per system interrupt source
7356
7357
7358
7359
7360
7361
7362
7363
7364
7365
7366
7367
7368
7369
7370
7371
7372
7373
7374
7375
7376
7377
7378
7379
7380
7381
7382
7383
7384
7385
7386
7387
7388
7389
7390
7391
7392
7393
7394
7395
7396
7397
7398
7399

```

```

7400 // Blank page.
7401
7402
7403
7404
7405
7406
7407
7408
7409
7410
7411
7412
7413
7414
7415
7416
7417
7418
7419
7420
7421
7422
7423
7424
7425
7426
7427
7428
7429
7430
7431
7432
7433
7434
7435
7436
7437
7438
7439
7440
7441
7442
7443
7444
7445
7446
7447
7448
7449

```

```

7450 // Multiprocessor support
7451 // Search memory for MP description structures.
7452 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
7453
7454 #include "types.h"
7455 #include "defs.h"
7456 #include "param.h"
7457 #include "memlayout.h"
7458 #include "mp.h"
7459 #include "x86.h"
7460 #include "mmu.h"
7461 #include "proc.h"
7462
7463 struct cpu cpus[NCPU];
7464 static struct cpu *bcpu;
7465 int ismp;
7466 int ncpu;
7467 uchar ioapicid;
7468
7469 int
7470 mpbcpu(void)
7471 {
7472     return bcpu-cpus;
7473 }
7474
7475 static uchar
7476 sum(uchar *addr, int len)
7477 {
7478     int i, sum;
7479
7480     sum = 0;
7481     for(i=0; i<len; i++)
7482         sum += addr[i];
7483     return sum;
7484 }
7485
7486 // Look for an MP structure in the len bytes at addr.
7487 static struct mp*
7488 mpsearch1(uint a, int len)
7489 {
7490     uchar *e, *p, *addr;
7491
7492     addr = p2v(a);
7493     e = addr+len;
7494     for(p = addr; p < e; p += sizeof(struct mp))
7495         if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
7496             return (struct mp*)p;
7497     return 0;
7498 }
7499

```

```

7500 // Search for the MP Floating Pointer Structure, which according to the
7501 // spec is in one of the following three locations:
7502 // 1) in the first KB of the EBDA;
7503 // 2) in the last KB of system base memory;
7504 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
7505 static struct mp*
7506 mpsearch(void)
7507 {
7508     uchar *bda;
7509     uint p;
7510     struct mp *mp;
7511
7512     bda = (uchar *) P2V(0x400);
7513     if((p = ((bda[0x0F]<<8) | bda[0x0E]) << 4)){
7514         if((mp = mpsearch1(p, 1024)))
7515             return mp;
7516     } else {
7517         p = ((bda[0x14]<<8) | bda[0x13]) * 1024;
7518         if((mp = mpsearch1(p-1024, 1024)))
7519             return mp;
7520     }
7521     return mpsearch1(0xF0000, 0x10000);
7522 }
7523
7524 // Search for an MP configuration table. For now,
7525 // don't accept the default configurations (physaddr == 0).
7526 // Check for correct signature, calculate the checksum and,
7527 // if correct, check the version.
7528 // To do: check extended table checksum.
7529 static struct mpconf*
7530 mpconfig(struct mp **pmp)
7531 {
7532     struct mpconf *conf;
7533     struct mp *mp;
7534
7535     if((mp = mpsearch()) == 0 || mp->physaddr == 0)
7536         return 0;
7537     conf = (struct mpconf*) p2v((uint) mp->physaddr);
7538     if(memcmp(conf, "PCMP", 4) != 0)
7539         return 0;
7540     if(conf->version != 1 && conf->version != 4)
7541         return 0;
7542     if(sum((uchar*)conf, conf->length) != 0)
7543         return 0;
7544     *pmp = mp;
7545     return conf;
7546 }
7547
7548
7549

```

```

7550 void
7551 mpinit(void)
7552 {
7553     uchar *p, *e;
7554     struct mp *mp;
7555     struct mpconf *conf;
7556     struct mpproc *proc;
7557     struct mpioapic *ioapic;
7558
7559     bcpu = &cpus[0];
7560     if((conf = mpconfig(&mp)) == 0)
7561         return;
7562     ismp = 1;
7563     lapic = (uint*)conf->lapicaddr;
7564     for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
7565         switch(*p){
7566             case MPPROC:
7567                 proc = (struct mpproc*)p;
7568                 if(ncpu != proc->apicid){
7569                     cprintf("mpinit: ncpu=%d apicid=%d\n", ncpu, proc->apicid);
7570                     ismp = 0;
7571                 }
7572                 if(proc->flags & MPBOOT)
7573                     bcpu = &cpus[ncpu];
7574                 cpus[ncpu].id = ncpu;
7575                 ncpu++;
7576                 p += sizeof(struct mpproc);
7577                 continue;
7578             case MPIOAPIC:
7579                 ioapic = (struct mpioapic*)p;
7580                 ioapicid = ioapic->apicno;
7581                 p += sizeof(struct mpioapic);
7582                 continue;
7583             case MPBUS:
7584             case MPIOINTR:
7585             case MPLINTR:
7586                 p += 8;
7587                 continue;
7588             default:
7589                 cprintf("mpinit: unknown config type %x\n", *p);
7590                 ismp = 0;
7591         }
7592     }
7593     if(!ismp){
7594         // Didn't like what we found; fall back to no MP.
7595         ncpu = 1;
7596         lapic = 0;
7597         ioapicid = 0;
7598         return;
7599     }

```

```

7600 if(mp->imcrp){
7601     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
7602     // But it would on real hardware.
7603     outb(0x22, 0x70); // Select IMCR
7604     outb(0x23, inb(0x23) | 1); // Mask external interrupts.
7605 }
7606 }
7607
7608
7609
7610
7611
7612
7613
7614
7615
7616
7617
7618
7619
7620
7621
7622
7623
7624
7625
7626
7627
7628
7629
7630
7631
7632
7633
7634
7635
7636
7637
7638
7639
7640
7641
7642
7643
7644
7645
7646
7647
7648
7649

```

```

7650 // The local APIC manages internal (non-I/O) interrupts.
7651 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
7652
7653 #include "types.h"
7654 #include "defs.h"
7655 #include "date.h"
7656 #include "memlayout.h"
7657 #include "traps.h"
7658 #include "mmu.h"
7659 #include "x86.h"
7660
7661 // Local APIC registers, divided by 4 for use as uint[] indices.
7662 #define ID      (0x0020/4) // ID
7663 #define VER     (0x0030/4) // Version
7664 #define TPR     (0x0080/4) // Task Priority
7665 #define EOI     (0x00B0/4) // EOI
7666 #define SVR     (0x00F0/4) // Spurious Interrupt Vector
7667 #define ENABLE  0x00000100 // Unit Enable
7668 #define ESR     (0x0280/4) // Error Status
7669 #define ICRLO   (0x0300/4) // Interrupt Command
7670 #define INIT    0x00000500 // INIT/RESET
7671 #define STARTUP 0x00000600 // Startup IPI
7672 #define DELIVS  0x00001000 // Delivery status
7673 #define ASSERT  0x00004000 // Assert interrupt (vs deassert)
7674 #define DEASSERT 0x00000000
7675 #define LEVEL   0x00008000 // Level triggered
7676 #define BCAST   0x00080000 // Send to all APICs, including self.
7677 #define BUSY    0x00001000
7678 #define FIXED    0x00000000
7679 #define ICRHI   (0x0310/4) // Interrupt Command [63:32]
7680 #define TIMER   (0x0320/4) // Local Vector Table 0 (TIMER)
7681 #define X1      0x0000000B // divide counts by 1
7682 #define PERIODIC 0x00020000 // Periodic
7683 #define PCINT    (0x0340/4) // Performance Counter LVT
7684 #define LINT0    (0x0350/4) // Local Vector Table 1 (LINT0)
7685 #define LINT1    (0x0360/4) // Local Vector Table 2 (LINT1)
7686 #define ERROR    (0x0370/4) // Local Vector Table 3 (ERROR)
7687 #define MASKED   0x00010000 // Interrupt masked
7688 #define TICR     (0x0380/4) // Timer Initial Count
7689 #define TCCR     (0x0390/4) // Timer Current Count
7690 #define TDCR     (0x03E0/4) // Timer Divide Configuration
7691
7692 volatile uint *lapic; // Initialized in mp.c
7693
7694 static void
7695 lapicw(int index, int value)
7696 {
7697     lapic[index] = value;
7698     lapic[ID]; // wait for write to finish, by reading
7699 }

```

7700
7701
7702
7703
7704
7705
7706
7707
7708
7709
7710
7711
7712
7713
7714
7715
7716
7717
7718
7719
7720
7721
7722
7723
7724
7725
7726
7727
7728
7729
7730
7731
7732
7733
7734
7735
7736
7737
7738
7739
7740
7741
7742
7743
7744
7745
7746
7747
7748
7749

```

7750 void
7751 lapicinit(void)
7752 {
7753     if(!lapic)
7754         return;
7755
7756     // Enable local APIC; set spurious interrupt vector.
7757     lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
7758
7759     // The timer repeatedly counts down at bus frequency
7760     // from lapic[TICR] and then issues an interrupt.
7761     // If xv6 cared more about precise timekeeping,
7762     // TICR would be calibrated using an external time source.
7763     lapicw(TDCR, X1);
7764     lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
7765     lapicw(TICR, 10000000);
7766
7767     // Disable logical interrupt lines.
7768     lapicw(LINT0, MASKED);
7769     lapicw(LINT1, MASKED);
7770
7771     // Disable performance counter overflow interrupts
7772     // on machines that provide that interrupt entry.
7773     if(((lapic[VER]>>16) & 0xFF) >= 4)
7774         lapicw(PCINT, MASKED);
7775
7776     // Map error interrupt to IRQ_ERROR.
7777     lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
7778
7779     // Clear error status register (requires back-to-back writes).
7780     lapicw(ESR, 0);
7781     lapicw(ESR, 0);
7782
7783     // Ack any outstanding interrupts.
7784     lapicw(EOI, 0);
7785
7786     // Send an Init Level De-Assert to synchronise arbitration ID's.
7787     lapicw(ICRHI, 0);
7788     lapicw(ICRLO, BCAST | INIT | LEVEL);
7789     while(lapic[ICRLO] & DELIVS)
7790         ;
7791
7792     // Enable interrupts on the APIC (but not on the processor).
7793     lapicw(TPR, 0);
7794 }
7795
7796
7797
7798
7799

```

```

7800 int
7801 cpunum(void)
7802 {
7803     // Cannot call cpu when interrupts are enabled:
7804     // result not guaranteed to last long enough to be used!
7805     // Would prefer to panic but even printing is chancy here:
7806     // almost everything, including cprintf and panic, calls cpu,
7807     // often indirectly through acquire and release.
7808     if(readeflags() & FL_IF){
7809         static int n;
7810         if(n++ == 0)
7811             cprintf("cpu called from %x with interrupts enabled\n",
7812                 __builtin_return_address(0));
7813     }
7814
7815     if(lapic)
7816         return lapic[ID]>>24;
7817     return 0;
7818 }
7819
7820 // Acknowledge interrupt.
7821 void
7822 lapiceoi(void)
7823 {
7824     if(lapic)
7825         lapicw(EOI, 0);
7826 }
7827
7828 // Spin for a given number of microseconds.
7829 // On real hardware would want to tune this dynamically.
7830 void
7831 microdelay(int us)
7832 {
7833 }
7834
7835 #define CMOS_PORT    0x70
7836 #define CMOS_RETURN  0x71
7837
7838 // Start additional processor running entry code at addr.
7839 // See Appendix B of MultiProcessor Specification.
7840 void
7841 lapicstartap(uchar apicid, uint addr)
7842 {
7843     int i;
7844     ushort *wrv;
7845
7846     // "The BSP must initialize CMOS shutdown code to 0AH
7847     // and the warm reset vector (DWORD based at 40:67) to point at
7848     // the AP startup code prior to the [universal startup algorithm]."
7849     outb(CMOS_PORT, 0xF); // offset 0xF is shutdown code

```

```

7850     outb(CMOS_PORT+1, 0x0A);
7851     wrv = (ushort*)P2V((0x40<<4 | 0x67)); // Warm reset vector
7852     wrv[0] = 0;
7853     wrv[1] = addr >> 4;
7854
7855     // "Universal startup algorithm."
7856     // Send INIT (level-triggered) interrupt to reset other CPU.
7857     lapicw(ICRHI, apicid<<24);
7858     lapicw(ICRLO, INIT | LEVEL | ASSERT);
7859     microdelay(200);
7860     lapicw(ICRLO, INIT | LEVEL);
7861     microdelay(100); // should be 10ms, but too slow in Bochs!
7862
7863     // Send startup IPI (twice!) to enter code.
7864     // Regular hardware is supposed to only accept a STARTUP
7865     // when it is in the halted state due to an INIT. So the second
7866     // should be ignored, but it is part of the official Intel algorithm.
7867     // Bochs complains about the second one. Too bad for Bochs.
7868     for(i = 0; i < 2; i++){
7869         lapicw(ICRHI, apicid<<24);
7870         lapicw(ICRLO, STARTUP | (addr>>12));
7871         microdelay(200);
7872     }
7873 }
7874
7875 #define CMOS_STATA    0x0a
7876 #define CMOS_STATB    0x0b
7877 #define CMOS_UIP      (1 << 7) // RTC update in progress
7878
7879 #define SECS          0x00
7880 #define MINS          0x02
7881 #define HOURS         0x04
7882 #define DAY           0x07
7883 #define MONTH         0x08
7884 #define YEAR          0x09
7885
7886 static uint cmos_read(uint reg)
7887 {
7888     outb(CMOS_PORT, reg);
7889     microdelay(200);
7890
7891     return inb(CMOS_RETURN);
7892 }
7893
7894
7895
7896
7897
7898
7899

```

```

7900 static void fill_rtcddate(struct rtcdate *r)
7901 {
7902     r->second = cmos_read(SECS);
7903     r->minute = cmos_read(MINS);
7904     r->hour   = cmos_read(HOURS);
7905     r->day    = cmos_read(DAY);
7906     r->month  = cmos_read(MONTH);
7907     r->year   = cmos_read(YEAR);
7908 }
7909
7910 // qemu seems to use 24-hour GWT and the values are BCD encoded
7911 void cmostime(struct rtcdate *r)
7912 {
7913     struct rtcdate t1, t2;
7914     int sb, bcd;
7915
7916     sb = cmos_read(CMOS_STATB);
7917
7918     bcd = (sb & (1 << 2)) == 0;
7919
7920     // make sure CMOS doesn't modify time while we read it
7921     for (;;) {
7922         fill_rtcddate(&t1);
7923         if (cmos_read(CMOS_STATB) & CMOS_UIP)
7924             continue;
7925         fill_rtcddate(&t2);
7926         if (memcmp(&t1, &t2, sizeof(t1)) == 0)
7927             break;
7928     }
7929
7930     // convert
7931     if (bcd) {
7932 #define CONV(x)      (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
7933         CONV(second);
7934         CONV(minute);
7935         CONV(hour);
7936         CONV(day);
7937         CONV(month);
7938         CONV(year);
7939 #undef CONV
7940     }
7941
7942     *r = t1;
7943     r->year += 2000;
7944 }
7945
7946
7947
7948
7949

```

```

7950 // The I/O APIC manages hardware interrupts for an SMP system.
7951 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
7952 // See also picirq.c.
7953
7954 #include "types.h"
7955 #include "defs.h"
7956 #include "traps.h"
7957
7958 #define IOAPIC    0xFEC00000    // Default physical address of IO APIC
7959
7960 #define REG_ID     0x00    // Register index: ID
7961 #define REG_VER    0x01    // Register index: version
7962 #define REG_TABLE  0x10    // Redirection table base
7963
7964 // The redirection table starts at REG_TABLE and uses
7965 // two registers to configure each interrupt.
7966 // The first (low) register in a pair contains configuration bits.
7967 // The second (high) register contains a bitmask telling which
7968 // CPUs can serve that interrupt.
7969 #define INT_DISABLED 0x00010000 // Interrupt disabled
7970 #define INT_LEVEL    0x00008000 // Level-triggered (vs edge-)
7971 #define INT_ACTIVELOW 0x00002000 // Active low (vs high)
7972 #define INT_LOGICAL  0x00000800 // Destination is CPU id (vs APIC ID)
7973
7974 volatile struct ioapic *ioapic;
7975
7976 // IO APIC MMIO structure: write reg, then read or write data.
7977 struct ioapic {
7978     uint reg;
7979     uint pad[3];
7980     uint data;
7981 };
7982
7983 static uint
7984 ioapicread(int reg)
7985 {
7986     ioapic->reg = reg;
7987     return ioapic->data;
7988 }
7989
7990 static void
7991 ioapicwrite(int reg, uint data)
7992 {
7993     ioapic->reg = reg;
7994     ioapic->data = data;
7995 }
7996
7997
7998
7999

```

```

8000 void
8001 ioapicinit(void)
8002 {
8003     int i, id, maxintr;
8004
8005     if(!ismp)
8006         return;
8007
8008     ioapic = (volatile struct ioapic*)IOAPIC;
8009     maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
8010     id = ioapicread(REG_ID) >> 24;
8011     if(id != ioapicid)
8012         cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
8013
8014     // Mark all interrupts edge-triggered, active high, disabled,
8015     // and not routed to any CPUs.
8016     for(i = 0; i <= maxintr; i++){
8017         ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
8018         ioapicwrite(REG_TABLE+2*i+1, 0);
8019     }
8020 }
8021
8022 void
8023 ioapicenable(int irq, int cpunum)
8024 {
8025     if(!ismp)
8026         return;
8027
8028     // Mark interrupt edge-triggered, active high,
8029     // enabled, and routed to the given cpunum,
8030     // which happens to be that cpu's APIC ID.
8031     ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
8032     ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
8033 }
8034
8035
8036
8037
8038
8039
8040
8041
8042
8043
8044
8045
8046
8047
8048
8049

```

```

8050 // Intel 8259A programmable interrupt controllers.
8051
8052 #include "types.h"
8053 #include "x86.h"
8054 #include "traps.h"
8055
8056 // I/O Addresses of the two programmable interrupt controllers
8057 #define IO_PIC1      0x20    // Master (IRQs 0-7)
8058 #define IO_PIC2      0xA0    // Slave (IRQs 8-15)
8059
8060 #define IRQ_SLAVE     2      // IRQ at which slave connects to master
8061
8062 // Current IRQ mask.
8063 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
8064 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
8065
8066 static void
8067 picsetmask(ushort mask)
8068 {
8069     irqmask = mask;
8070     outb(IO_PIC1+1, mask);
8071     outb(IO_PIC2+1, mask >> 8);
8072 }
8073
8074 void
8075 picenable(int irq)
8076 {
8077     picsetmask(irqmask & ~(1<<irq));
8078 }
8079
8080 // Initialize the 8259A interrupt controllers.
8081 void
8082 picinit(void)
8083 {
8084     // mask all interrupts
8085     outb(IO_PIC1+1, 0xFF);
8086     outb(IO_PIC2+1, 0xFF);
8087
8088     // Set up master (8259A-1)
8089
8090     // ICW1: 0001g0hi
8091     //   g: 0 = edge triggering, 1 = level triggering
8092     //   h: 0 = cascaded PICs, 1 = master only
8093     //   i: 0 = no ICW4, 1 = ICW4 required
8094     outb(IO_PIC1, 0x11);
8095
8096     // ICW2: Vector offset
8097     outb(IO_PIC1+1, T_IRQ0);
8098
8099

```



```

8100 // ICW3: (master PIC) bit mask of IR lines connected to slaves
8101 //      (slave PIC) 3-bit # of slave's connection to master
8102 outb(IO_PIC1+1, 1<<IRQ_SLAVE);
8103
8104 // ICW4: 000nbmap
8105 //      n: 1 = special fully nested mode
8106 //      b: 1 = buffered mode
8107 //      m: 0 = slave PIC, 1 = master PIC
8108 //      (ignored when b is 0, as the master/slave role
8109 //      can be hardwired).
8110 //      a: 1 = Automatic EOI mode
8111 //      p: 0 = MCS-80/85 mode, 1 = intel x86 mode
8112 outb(IO_PIC1+1, 0x3);
8113
8114 // Set up slave (8259A-2)
8115 outb(IO_PIC2, 0x11); // ICW1
8116 outb(IO_PIC2+1, T_IRQ0 + 8); // ICW2
8117 outb(IO_PIC2+1, IRQ_SLAVE); // ICW3
8118 // NB Automatic EOI mode doesn't tend to work on the slave.
8119 // Linux source code says it's "to be investigated".
8120 outb(IO_PIC2+1, 0x3); // ICW4
8121
8122 // OCW3: 0ef0lprs
8123 //      ef: 0x = NOP, 10 = clear specific mask, 11 = set specific mask
8124 //      p: 0 = no polling, 1 = polling mode
8125 //      rs: 0x = NOP, 10 = read IRR, 11 = read ISR
8126 outb(IO_PIC1, 0x68); // clear specific mask
8127 outb(IO_PIC1, 0x0a); // read IRR by default
8128
8129 outb(IO_PIC2, 0x68); // OCW3
8130 outb(IO_PIC2, 0x0a); // OCW3
8131
8132 if(irqmask != 0xFFFF)
8133     picsetmask(irqmask);
8134 }
8135
8136
8137
8138
8139
8140
8141
8142
8143
8144
8145
8146
8147
8148
8149

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

8400 // Print to the console. only understands %d, %x, %p, %s.
8401 void
8402 cprintf(char *fmt, ...)
8403 {
8404     int i, c, locking;
8405     uint *argp;
8406     char *s;
8407
8408     locking = cons.locking;
8409     if(locking)
8410         acquire(&cons.lock);
8411
8412     if (fmt == 0)
8413         panic("null fmt");
8414
8415     argp = (uint*)(void*)&fmt + 1;
8416     for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
8417         if(c != '%'){
8418             consputc(c);
8419             continue;
8420         }
8421         c = fmt[++i] & 0xff;
8422         if(c == 0)
8423             break;
8424         switch(c){
8425             case 'd':
8426                 printint(*argp++, 10, 1);
8427                 break;
8428             case 'x':
8429             case 'p':
8430                 printint(*argp++, 16, 0);
8431                 break;
8432             case 's':
8433                 if((s = (char*)*argp++) == 0)
8434                     s = "(null)";
8435                 for(; *s; s++)
8436                     consputc(*s);
8437                 break;
8438             case '%':
8439                 consputc('%');
8440                 break;
8441             default:
8442                 // Print unknown % sequence to draw attention.
8443                 consputc('%');
8444                 consputc(c);
8445                 break;
8446         }
8447     }
8448
8449

```

```

8450     if(locking)
8451         release(&cons.lock);
8452 }
8453
8454 void
8455 panic(char *s)
8456 {
8457     int i;
8458     uint pcs[10];
8459
8460     cli();
8461     cons.locking = 0;
8462     cprintf("cpu%d: panic: ", cpu->id);
8463     cprintf(s);
8464     cprintf("\n");
8465     getcallerpcs(&s, pcs);
8466     for(i=0; i<10; i++)
8467         cprintf(" %p", pcs[i]);
8468     panicked = 1; // freeze other CPU
8469     for(;;)
8470         ;
8471 }
8472
8473
8474
8475
8476
8477
8478
8479
8480
8481
8482
8483
8484
8485
8486
8487
8488
8489
8490
8491
8492
8493
8494
8495
8496
8497
8498
8499

```

```

8500 #define BACKSPACE 0x100
8501 #define CRTPORT 0x3d4
8502 static ushort *crt = (ushort*)P2V(0xb8000); // CGA memory
8503
8504 static void
8505 cgaputc(int c)
8506 {
8507     int pos;
8508
8509     // Cursor position: col + 80*row.
8510     outb(CRTPORT, 14);
8511     pos = inb(CRTPORT+1) << 8;
8512     outb(CRTPORT, 15);
8513     pos |= inb(CRTPORT+1);
8514
8515     if(c == '\n')
8516         pos += 80 - pos%80;
8517     else if(c == BACKSPACE){
8518         if(pos > 0) --pos;
8519     } else
8520         crt[pos++] = (c&0xff) | 0x0700; // black on white
8521
8522     if(pos < 0 || pos > 25*80)
8523         panic("pos under/overflow");
8524
8525     if((pos/80) >= 24){ // Scroll up.
8526         memmove(crt, crt+80, sizeof(crt[0])*23*80);
8527         pos -= 80;
8528         memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
8529     }
8530
8531     outb(CRTPORT, 14);
8532     outb(CRTPORT+1, pos>>8);
8533     outb(CRTPORT, 15);
8534     outb(CRTPORT+1, pos);
8535     crt[pos] = ' ' | 0x0700;
8536 }
8537
8538
8539
8540
8541
8542
8543
8544
8545
8546
8547
8548
8549

```

```

8550 void
8551 consputc(int c)
8552 {
8553     if(panicked){
8554         cli();
8555         for(;;)
8556             ;
8557     }
8558
8559     if(c == BACKSPACE){
8560         uartputc('\b'); uartputc(' '); uartputc('\b');
8561     } else
8562         uartputc(c);
8563     cgaputc(c);
8564 }
8565
8566 #define INPUT_BUF 128
8567 struct {
8568     char buf[INPUT_BUF];
8569     uint r; // Read index
8570     uint w; // Write index
8571     uint e; // Edit index
8572 } input;
8573
8574 #define C(x) ((x)-'@') // Control-x
8575
8576 void
8577 consoleintr(int (*getc)(void))
8578 {
8579     int c, doprocump = 0;
8580
8581     acquire(&cons.lock);
8582     while((c = getc()) >= 0){
8583         switch(c){
8584             case C('P'): // Process listing.
8585                 doprocump = 1; // procdump() locks cons.lock indirectly; invoke later
8586                 break;
8587             case C('U'): // Kill line.
8588                 while(input.e != input.w &&
8589                     input.buf[(input.e-1) % INPUT_BUF] != '\n'){
8590                     input.e--;
8591                     consputc(BACKSPACE);
8592                 }
8593                 break;
8594             case C('H'): case '\x7f': // Backspace
8595                 if(input.e != input.w){
8596                     input.e--;
8597                     consputc(BACKSPACE);
8598                 }
8599                 break;

```

```

8600     default:
8601         if(c != 0 && input.e-input.r < INPUT_BUF){
8602             c = (c == '\r') ? '\n' : c;
8603             input.buf[input.e++ % INPUT_BUF] = c;
8604             consputc(c);
8605             if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
8606                 input.w = input.e;
8607                 wakeup(&input.r);
8608             }
8609         }
8610         break;
8611     }
8612 }
8613 release(&cons.lock);
8614 if(doprocDump) {
8615     procDump(); // now call procDump() wo. cons.lock held
8616 }
8617 }
8618
8619 int
8620 consoleread(struct inode *ip, char *dst, int n)
8621 {
8622     uint target;
8623     int c;
8624
8625     iunlock(ip);
8626     target = n;
8627     acquire(&cons.lock);
8628     while(n > 0){
8629         while(input.r == input.w){
8630             if(proc->killed){
8631                 release(&cons.lock);
8632                 ilock(ip);
8633                 return -1;
8634             }
8635             sleep(&input.r, &cons.lock);
8636         }
8637         c = input.buf[input.r++ % INPUT_BUF];
8638         if(c == C('D')){ // EOF
8639             if(n < target){
8640                 // Save ^D for next time, to make sure
8641                 // caller gets a 0-byte result.
8642                 input.r--;
8643             }
8644             break;
8645         }
8646         *dst++ = c;
8647         --n;
8648         if(c == '\n')
8649             break;

```

```

8650     }
8651     release(&cons.lock);
8652     ilock(ip);
8653
8654     return target - n;
8655 }
8656
8657 int
8658 consolewrite(struct inode *ip, char *buf, int n)
8659 {
8660     int i;
8661
8662     iunlock(ip);
8663     acquire(&cons.lock);
8664     for(i = 0; i < n; i++){
8665         consputc(buf[i] & 0xff);
8666         release(&cons.lock);
8667         ilock(ip);
8668     }
8669     return n;
8670 }
8671
8672 void
8673 consoleinit(void)
8674 {
8675     initlock(&cons.lock, "console");
8676
8677     devsw[CONSOLE].write = consolewrite;
8678     devsw[CONSOLE].read = consoleread;
8679     cons.locking = 1;
8680
8681     picenable(IRQ_KBD);
8682     ioapicenable(IRQ_KBD, 0);
8683 }
8684
8685
8686
8687
8688
8689
8690
8691
8692
8693
8694
8695
8696
8697
8698
8699

```

```

8700 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
8701 // Only used on uniprocessors;
8702 // SMP machines use the local APIC timer.
8703
8704 #include "types.h"
8705 #include "defs.h"
8706 #include "traps.h"
8707 #include "x86.h"
8708
8709 #define IO_TIMER1      0x040      // 8253 Timer #1
8710
8711 // Frequency of all three count-down timers;
8712 // (TIMER_FREQ/freq) is the appropriate count
8713 // to generate a frequency of freq Hz.
8714
8715 #define TIMER_FREQ      1193182
8716 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
8717
8718 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
8719 #define TIMER_SEL0      0x00      // select counter 0
8720 #define TIMER_RATEGEN    0x04      // mode 2, rate generator
8721 #define TIMER_16BIT      0x30      // r/w counter 16 bits, LSB first
8722
8723 void
8724 timerinit(void)
8725 {
8726     // Interrupt 100 times/sec.
8727     outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
8728     outb(IO_TIMER1, TIMER_DIV(100) % 256);
8729     outb(IO_TIMER1, TIMER_DIV(100) / 256);
8730     picenable(IRQ_TIMER);
8731 }
8732
8733
8734
8735
8736
8737
8738
8739
8740
8741
8742
8743
8744
8745
8746
8747
8748
8749

```

```

8750 // Intel 8250 serial port (UART).
8751
8752 #include "types.h"
8753 #include "defs.h"
8754 #include "param.h"
8755 #include "traps.h"
8756 #include "spinlock.h"
8757 #include "fs.h"
8758 #include "file.h"
8759 #include "mmu.h"
8760 #include "proc.h"
8761 #include "x86.h"
8762
8763 #define COM1      0x3f8
8764
8765 static int uart;      // is there a uart?
8766
8767 void
8768 uartinit(void)
8769 {
8770     char *p;
8771
8772     // Turn off the FIFO
8773     outb(COM1+2, 0);
8774
8775     // 9600 baud, 8 data bits, 1 stop bit, parity off.
8776     outb(COM1+3, 0x80);      // Unlock divisor
8777     outb(COM1+0, 115200/9600);
8778     outb(COM1+1, 0);
8779     outb(COM1+3, 0x03);      // Lock divisor, 8 data bits.
8780     outb(COM1+4, 0);
8781     outb(COM1+1, 0x01);      // Enable receive interrupts.
8782
8783     // If status is 0xFF, no serial port.
8784     if(inb(COM1+5) == 0xFF)
8785         return;
8786     uart = 1;
8787
8788     // Acknowledge pre-existing interrupt conditions;
8789     // enable interrupts.
8790     inb(COM1+2);
8791     inb(COM1+0);
8792     picenable(IRQ_COM1);
8793     ioapicenable(IRQ_COM1, 0);
8794
8795     // Announce that we're here.
8796     for(p="xv6...\n"; *p; p++)
8797         uartputc(*p);
8798 }
8799

```

```

8800 void
8801 uartputc(int c)
8802 {
8803     int i;
8804
8805     if(!uart)
8806         return;
8807     for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
8808         microdelay(10);
8809     outb(COM1+0, c);
8810 }
8811
8812 static int
8813 uartgetc(void)
8814 {
8815     if(!uart)
8816         return -1;
8817     if(!(inb(COM1+5) & 0x01))
8818         return -1;
8819     return inb(COM1+0);
8820 }
8821
8822 void
8823 uartintr(void)
8824 {
8825     consoleintr(uartgetc);
8826 }
8827
8828
8829
8830
8831
8832
8833
8834
8835
8836
8837
8838
8839
8840
8841
8842
8843
8844
8845
8846
8847
8848
8849

```

```

8850 # Initial process execs /init.
8851
8852 #include "syscall.h"
8853 #include "traps.h"
8854
8855
8856 # exec(init, argv)
8857 .globl start
8858 start:
8859     pushl $argv
8860     pushl $init
8861     pushl $0 // where caller pc would be
8862     movl $SYS_exec, %eax
8863     int $T_SYSCALL
8864
8865 # for(;;) exit();
8866 exit:
8867     movl $SYS_exit, %eax
8868     int $T_SYSCALL
8869     jmp exit
8870
8871 # char init[] = "/init\0";
8872 init:
8873     .string "/init\0"
8874
8875 # char *argv[] = { init, 0 };
8876 .p2align 2
8877 argv:
8878     .long init
8879     .long 0
8880
8881
8882
8883
8884
8885
8886
8887
8888
8889
8890
8891
8892
8893
8894
8895
8896
8897
8898
8899

```



```

8900 #include "syscall.h"
8901 #include "traps.h"
8902
8903 #define SYSCALL(name) \
8904     .globl name; \
8905     name: \
8906     movl $SYS_ ## name, %eax; \
8907     int $T_SYSCALL; \
8908     ret
8909
8910 SYSCALL(fork)
8911 SYSCALL(exit)
8912 SYSCALL(wait)
8913 SYSCALL(pipe)
8914 SYSCALL(read)
8915 SYSCALL(write)
8916 SYSCALL(close)
8917 SYSCALL(kill)
8918 SYSCALL(exec)
8919 SYSCALL(open)
8920 SYSCALL(mknod)
8921 SYSCALL(unlink)
8922 SYSCALL(fstat)
8923 SYSCALL(link)
8924 SYSCALL(mkdir)
8925 SYSCALL(chdir)
8926 SYSCALL(dup)
8927 SYSCALL(getpid)
8928 SYSCALL(sbrk)
8929 SYSCALL(sleep)
8930 SYSCALL(uptime)
8931 SYSCALL(halt)
8932 SYSCALL(date)
8933 SYSCALL(getuid)
8934 SYSCALL(getgid)
8935 SYSCALL(getppid)
8936 SYSCALL(setuid)
8937 SYSCALL(setgid)
8938 SYSCALL(getprocs)
8939
8940
8941
8942
8943
8944
8945
8946
8947
8948
8949

```

```

8950 // init: The initial user-level program
8951
8952 #include "types.h"
8953 #include "stat.h"
8954 #include "user.h"
8955 #include "fcntl.h"
8956
8957 char *argv[] = { "sh", 0 };
8958
8959 int
8960 main(void)
8961 {
8962     int pid, wpid;
8963
8964     if(open("console", O_RDWR) < 0){
8965         mknod("console", 1, 1);
8966         open("console", O_RDWR);
8967     }
8968     dup(0); // stdout
8969     dup(0); // stderr
8970
8971     for(;;){
8972         printf(1, "init: starting sh\n");
8973         pid = fork();
8974         if(pid < 0){
8975             printf(1, "init: fork failed\n");
8976             exit();
8977         }
8978         if(pid == 0){
8979             exec("sh", argv);
8980             printf(1, "init: exec sh failed\n");
8981             exit();
8982         }
8983         while((wpid=wait()) >= 0 && wpid != pid)
8984             printf(1, "zombie!\n");
8985     }
8986 }
8987
8988
8989
8990
8991
8992
8993
8994
8995
8996
8997
8998
8999

```

```

9000 // Shell.
9001 // 2015-12-21. Added very simple processing for builtin commands
9002
9003 #include "types.h"
9004 #include "user.h"
9005 #include "fcntl.h"
9006
9007 // Parsed command representation
9008 #define EXEC 1
9009 #define REDIR 2
9010 #define PIPE 3
9011 #define LIST 4
9012 #define BACK 5
9013
9014 #define MAXARGS 10
9015
9016 struct cmd {
9017     int type;
9018 };
9019
9020 struct execcmd {
9021     int type;
9022     char *argv[MAXARGS];
9023     char *eargv[MAXARGS];
9024 };
9025
9026 struct redircmd {
9027     int type;
9028     struct cmd *cmd;
9029     char *file;
9030     char *efile;
9031     int mode;
9032     int fd;
9033 };
9034
9035 struct pipecmd {
9036     int type;
9037     struct cmd *left;
9038     struct cmd *right;
9039 };
9040
9041 struct listcmd {
9042     int type;
9043     struct cmd *left;
9044     struct cmd *right;
9045 };
9046
9047
9048
9049

```

```

9050 struct backcmd {
9051     int type;
9052     struct cmd *cmd;
9053 };
9054
9055 int fork1(void); // Fork but panics on failure.
9056 void panic(char*);
9057 struct cmd *parsecmd(char*);
9058
9059 // Execute cmd. Never returns.
9060 void
9061 runcmd(struct cmd *cmd)
9062 {
9063     int p[2];
9064     struct backcmd *bcmd;
9065     struct execcmd *ecmd;
9066     struct listcmd *lcmd;
9067     struct pipecmd *pcmd;
9068     struct redircmd *rcmd;
9069
9070     if(cmd == 0)
9071         exit();
9072
9073     switch(cmd->type){
9074     default:
9075         panic("runcmd");
9076
9077     case EXEC:
9078         ecmd = (struct execcmd*)cmd;
9079         if(ecmd->argv[0] == 0)
9080             exit();
9081         exec(ecmd->argv[0], ecmd->argv);
9082         printf(2, "exec %s failed\n", ecmd->argv[0]);
9083         break;
9084
9085     case REDIR:
9086         rcmd = (struct redircmd*)cmd;
9087         close(rcmd->fd);
9088         if(open(rcmd->file, rcmd->mode) < 0){
9089             printf(2, "open %s failed\n", rcmd->file);
9090             exit();
9091         }
9092         runcmd(rcmd->cmd);
9093         break;
9094
9095     case LIST:
9096         lcmd = (struct listcmd*)cmd;
9097         if(fork1() == 0)
9098             runcmd(lcmd->left);
9099         wait();

```

```

9100     runcmd(lcmd->right);
9101     break;
9102
9103     case PIPE:
9104         pcmd = (struct pipecmd*)cmd;
9105         if(pipe(p) < 0)
9106             panic("pipe");
9107         if(fork1() == 0){
9108             close(1);
9109             dup(p[1]);
9110             close(p[0]);
9111             close(p[1]);
9112             runcmd(pcmd->left);
9113         }
9114         if(fork1() == 0){
9115             close(0);
9116             dup(p[0]);
9117             close(p[0]);
9118             close(p[1]);
9119             runcmd(pcmd->right);
9120         }
9121         close(p[0]);
9122         close(p[1]);
9123         wait();
9124         wait();
9125         break;
9126
9127     case BACK:
9128         bcmd = (struct backcmd*)cmd;
9129         if(fork1() == 0)
9130             runcmd(bcmd->cmd);
9131         break;
9132     }
9133     exit();
9134 }
9135
9136 int
9137 getcmd(char *buf, int nbuf)
9138 {
9139     printf(2, "$ ");
9140     memset(buf, 0, nbuf);
9141     gets(buf, nbuf);
9142     if(buf[0] == 0) // EOF
9143         return -1;
9144     return 0;
9145 }
9146
9147
9148
9149

```

```

9150 // ***** processing for shell builtins begins here*****
9151
9152 int
9153 strncmp(const char *p, const char *q, uint n)
9154 {
9155     while(n > 0 && *p && *p == *q)
9156         n--, p++, q++;
9157     if(n == 0)
9158         return 0;
9159     return (uchar)*p - (uchar)*q;
9160 }
9161
9162 int
9163 makeint(char *p)
9164 {
9165     int val = 0;
9166
9167     while ((*p >= '0') && (*p <= '9')) {
9168         val = 10*val + (*p-'0');
9169         ++p;
9170     }
9171     return val;
9172 }
9173
9174 int
9175 setbuiltin(char *p)
9176 {
9177     int i;
9178
9179     p += strlen("_set");
9180     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
9181     if (strncmp("uid", p, 3) == 0) {
9182         p += strlen("uid");
9183         while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
9184         i = makeint(p); // ugly
9185         return (setuid(i));
9186     } else
9187     if (strncmp("gid", p, 3) == 0) {
9188         p += strlen("gid");
9189         while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
9190         i = makeint(p); // ugly
9191         return (setgid(i));
9192     }
9193     printf(2, "Invalid _set parameter\n");
9194     return -1;
9195 }
9196
9197
9198
9199

```

```

9200 int
9201 getbuiltin(char *p)
9202 {
9203     p += strlen("_get");
9204     while (strncmp(p, " ", 1) == 0) p++; // chomp spaces
9205     if (strncmp("uid", p, 3) == 0) {
9206         printf(2, "%d\n", getuid());
9207         return 0;
9208     }
9209     if (strncmp("gid", p, 3) == 0) {
9210         printf(2, "%d\n", getgid());
9211         return 0;
9212     }
9213     printf(2, "Invalid _get parameter\n");
9214     return -1;
9215 }
9216
9217 typedef int funcPtr_t(char *);
9218 typedef struct {
9219     char      *cmd;
9220     funcPtr_t *name;
9221 } dispatchTableEntry_t;
9222
9223 // Use a simple function dispatch table (FDT) to process builtin commands
9224 dispatchTableEntry_t fdt[] = {
9225     {"_set", setbuiltin},
9226     {"_get", getbuiltin}
9227 };
9228 int FDTcount = sizeof(fdt) / sizeof(fdt[0]); // # entris in FDT
9229
9230 void
9231 dobuiltin(char *cmd) {
9232     int i;
9233
9234     for (i=0; i<FDTcount; i++)
9235         if (strncmp(cmd, fdt[i].cmd, strlen(fdt[i].cmd)) == 0)
9236             (*fdt[i].name)(cmd);
9237 }
9238
9239
9240
9241
9242
9243
9244
9245
9246
9247
9248
9249

```

```

9250 // ***** processing for shell builtins ends here *****
9251
9252 int
9253 main(void)
9254 {
9255     static char buf[100];
9256     int fd;
9257
9258     // Assumes three file descriptors open.
9259     while((fd = open("console", O_RDWR)) >= 0){
9260         if(fd >= 3){
9261             close(fd);
9262             break;
9263         }
9264     }
9265
9266     // Read and run input commands.
9267     while(getcmd(buf, sizeof(buf)) >= 0){
9268         // add support for built-ins here. cd is a built-in
9269         if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
9270             // Clumsy but will have to do for now.
9271             // Chdir has no effect on the parent if run in the child.
9272             buf[strlen(buf)-1] = 0; // chop \n
9273             if(chdir(buf+3) < 0)
9274                 printf(2, "cannot cd %s\n", buf+3);
9275             continue;
9276         }
9277         if (buf[0]=='_') { // assume it is a builtin command
9278             dobuiltin(buf);
9279             continue;
9280         }
9281         if(fork1() == 0)
9282             runcmd(parsecmd(buf));
9283         wait();
9284     }
9285     exit();
9286 }
9287
9288 void
9289 panic(char *s)
9290 {
9291     printf(2, "%s\n", s);
9292     exit();
9293 }
9294
9295
9296
9297
9298
9299

```

```

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

```

```

9350 // Constructors
9351
9352 struct cmd*
9353 execcmd(void)
9354 {
9355     struct execcmd *cmd;
9356
9357     cmd = malloc(sizeof(*cmd));
9358     memset(cmd, 0, sizeof(*cmd));
9359     cmd->type = EXEC;
9360     return (struct cmd*)cmd;
9361 }
9362
9363 struct cmd*
9364 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
9365 {
9366     struct redircmd *cmd;
9367
9368     cmd = malloc(sizeof(*cmd));
9369     memset(cmd, 0, sizeof(*cmd));
9370     cmd->type = REDIR;
9371     cmd->cmd = subcmd;
9372     cmd->file = file;
9373     cmd->efile = efile;
9374     cmd->mode = mode;
9375     cmd->fd = fd;
9376     return (struct cmd*)cmd;
9377 }
9378
9379 struct cmd*
9380 pipecmd(struct cmd *left, struct cmd *right)
9381 {
9382     struct pipecmd *cmd;
9383
9384     cmd = malloc(sizeof(*cmd));
9385     memset(cmd, 0, sizeof(*cmd));
9386     cmd->type = PIPE;
9387     cmd->left = left;
9388     cmd->right = right;
9389     return (struct cmd*)cmd;
9390 }
9391
9392
9393
9394
9395
9396
9397
9398
9399

```

```

9400 struct cmd*
9401 listcmd(struct cmd *left, struct cmd *right)
9402 {
9403     struct listcmd *cmd;
9404
9405     cmd = malloc(sizeof(*cmd));
9406     memset(cmd, 0, sizeof(*cmd));
9407     cmd->type = LIST;
9408     cmd->left = left;
9409     cmd->right = right;
9410     return (struct cmd*)cmd;
9411 }
9412
9413 struct cmd*
9414 backcmd(struct cmd *subcmd)
9415 {
9416     struct backcmd *cmd;
9417
9418     cmd = malloc(sizeof(*cmd));
9419     memset(cmd, 0, sizeof(*cmd));
9420     cmd->type = BACK;
9421     cmd->cmd = subcmd;
9422     return (struct cmd*)cmd;
9423 }
9424
9425
9426
9427
9428
9429
9430
9431
9432
9433
9434
9435
9436
9437
9438
9439
9440
9441
9442
9443
9444
9445
9446
9447
9448
9449

```

```

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

```

```

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

```

```

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

```

```

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

```

```

9650 // NUL-terminate all the counted strings.
9651 struct cmd*
9652 nulterminate(struct cmd *cmd)
9653 {
9654     int i;
9655     struct backcmd *bcmd;
9656     struct execcmd *ecmd;
9657     struct listcmd *lcmd;
9658     struct pipecmd *pcmd;
9659     struct redircmd *rcmd;
9660
9661     if(cmd == 0)
9662         return 0;
9663
9664     switch(cmd->type){
9665     case EXEC:
9666         ecmd = (struct execcmd*)cmd;
9667         for(i=0; ecmd->argv[i]; i++)
9668             *ecmd->eargv[i] = 0;
9669         break;
9670
9671     case REDIR:
9672         rcmd = (struct redircmd*)cmd;
9673         nulterminate(rcmd->cmd);
9674         *rcmd->efile = 0;
9675         break;
9676
9677     case PIPE:
9678         pcmd = (struct pipecmd*)cmd;
9679         nulterminate(pcmd->left);
9680         nulterminate(pcmd->right);
9681         break;
9682
9683     case LIST:
9684         lcmd = (struct listcmd*)cmd;
9685         nulterminate(lcmd->left);
9686         nulterminate(lcmd->right);
9687         break;
9688
9689     case BACK:
9690         bcmd = (struct backcmd*)cmd;
9691         nulterminate(bcmd->cmd);
9692         break;
9693     }
9694     return cmd;
9695 }
9696
9697
9698
9699

```



```

9700 #include "asm.h"
9701 #include "memlayout.h"
9702 #include "mmu.h"
9703
9704 # Start the first CPU: switch to 32-bit protected mode, jump into C.
9705 # The BIOS loads this code from the first sector of the hard disk into
9706 # memory at physical address 0x7c00 and starts executing in real mode
9707 # with %cs=0 %ip=7c00.
9708
9709 .code16                                # Assemble for 16-bit mode
9710 .globl start
9711 start:
9712     cli                                # BIOS enabled interrupts; disable
9713
9714     # Zero data segment registers DS, ES, and SS.
9715     xorw    %ax,%ax                    # Set %ax to zero
9716     movw    %ax,%ds                    # -> Data Segment
9717     movw    %ax,%es                    # -> Extra Segment
9718     movw    %ax,%ss                    # -> Stack Segment
9719
9720     # Physical address line A20 is tied to zero so that the first PCs
9721     # with 2 MB would run software that assumed 1 MB. Undo that.
9722 seta20.1:
9723     inb     $0x64,%al                  # Wait for not busy
9724     testb   $0x2,%al
9725     jnz     seta20.1
9726
9727     movb    $0xd1,%al                  # 0xd1 -> port 0x64
9728     outb    %al,$0x64
9729
9730 seta20.2:
9731     inb     $0x64,%al                  # Wait for not busy
9732     testb   $0x2,%al
9733     jnz     seta20.2
9734
9735     movb    $0xdf,%al                  # 0xdf -> port 0x60
9736     outb    %al,$0x60
9737
9738     # Switch from real to protected mode. Use a bootstrap GDT that makes
9739     # virtual addresses map directly to physical addresses so that the
9740     # effective memory map doesn't change during the transition.
9741     lgdt    gdtdesc
9742     movl    %cr0,%eax
9743     orl     $CR0_PE,%eax
9744     movl    %eax,%cr0
9745
9746
9747
9748
9749

```

```

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

```

```

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

```

```

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

```