

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

ACKNOWLEDGMENTS

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

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

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

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

The code in the files that constitute xv6 is
 Copyright 2006-2011 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/6.828/2011/tools.html>. Then run "make TOOLPREFIX=i386-jos-elf-".

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

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

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

# basic headers	# system calls	# string operations
01 types.h	28 traps.h	59 string.c
01 param.h	28 vectors.pl	
02 memlayout.h	29 trapasm.S	# low-level hardware
02 defs.h	29 trap.c	61 mp.h
04 x86.h	31 syscall.h	62 mp.c
06 asm.h	31 syscall.c	64 lapic.c
07 mmu.h	33 sysproc.c	66 ioapic.c
09 elf.h		67 picirq.c
	# file system	68 kbd.h
# entering xv6	34 buf.h	69 kbd.c
10 entry.S	34 fcntl.h	70 console.c
11 entryother.S	35 stat.h	73 timer.c
12 main.c	35 fs.h	74 uart.c
	36 file.h	
# locks	36 ide.c	# user-level
14 spinlock.h	38 bio.c	75 initcode.S
14 spinlock.c	40 log.c	75 usys.S
	42 fs.c	76 init.c
# processes	50 file.c	76 sh.c
16 vm.c	52 sysfile.c	
20 proc.h	57 exec.c	# bootloader
21 proc.c		82 bootasm.S
26 swtch.S	# pipes	83 bootmain.c
27 kalloc.c	58 pipe.c	

The source listing is preceded by a cross-reference that lists every defined constant, struct, global variable, and function in xv6. Each entry gives, on the same line as the name, the line number (or, in a few cases, numbers) where the name is defined. Successive lines in an entry list the line numbers where the name is used. For example, this entry:

```
swtch 2658
      0374 2428 2466 2657 2658
```

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

```

acquire 1474
  0377 1474 1478 2160 2323
  2358 2417 2474 2518 2533
  2566 2579 2766 2781 3016
  3372 3392 3757 3815 3920
  3979 4127 4145 4457 4490
  4510 4539 4554 4564 5025
  5041 5056 5863 5884 5905
  7060 7216 7258 7306
allocproc 2155
  2155 2207 2260
allocuvm 1827
  0422 1827 1841 2237 5743
  5753
alltraps 2904
  2859 2867 2880 2885 2903
  2904
ALT 6810
  6810 6838 6840
argfd 5213
  5213 5256 5271 5283 5294
  5306
argint 3195
  0395 3195 3208 3224 3332
  3356 3370 5218 5271 5283
  5508 5576 5577 5626
argptr 3204
  0396 3204 5271 5283 5306
  5657
argstr 3221
  0397 3221 5318 5408 5508
  5557 5575 5606 5626
__attribute__ 1316
  0270 0365 1209 1316
BACK 7661
  7661 7774 7920 8189
backcmd 7696 7914
  7696 7709 7775 7914 7916
  8042 8155 8190
BACKSPACE 7150
  7150 7167 7194 7226 7232
ballocc 4304
  4304 4326 4617 4625 4629
BBLOCK 3591
  3591 4313 4338
B_BUSY 3409
  3409 3808 3926 3927 3938
  3941 3966 3976 3988
B_DIRTY 3411
  3411 3737 3766 3771 3810
  3829 3968
begin_trans 4125
  0334 4125 5071 5174 5323
  5413 5511 5556 5574
bfree 4331
  4331 4662 4672 4675
bget 3916
  3916 3946 3956
binit 3889
  0261 1229 3889
bmap 4610
  4610 4636 4719 4769
bootmain 8317
  8268 8317
BPB 3588
  3588 3591 4312 4314 4339
bread 3952
  0262 3952 4076 4077 4089
  4104 4173 4282 4293 4313
  4338 4411 4432 4517 4626
  4668 4719 4769
brelse 3974
  0263 3974 3977 4080 4081
  4096 4112 4176 4284 4296
  4319 4324 4345 4417 4420
  4441 4525 4632 4674 4722
  4773
BSIZE 3557
  3557 3568 3582 3588 4057
  4078 4174 4294 4719 4720
  4721 4765 4769 4770 4771
buf 3400
  0250 0262 0263 0264 0306
  0333 1970 1973 1982 1984
  3400 3404 3405 3406 3661
  3676 3679 3725 3754 3804
  3806 3809 3877 3881 3885
  3891 3903 3915 3918 3951
  3954 3964 3974 4005 4076
  4077 4089 4090 4096 4104
  4105 4111 4112 4159 4173
  4269 4280 4291 4307 4333
  4405 4429 4504 4613 4657
  4705 4755 7029 7040 7044
  7047 7203 7224 7238 7268
  7301 7308 7784 7787 7788
  7789 7803 7815 7816 7819
  7820 7821 7825
B_INVALID 3410
  3410 3770 3810 3829 3957

```

```

bwrite 3964
  0264 3964 3967 4079 4111
  4175
bzero 4289
  4289 4320
C 6831 7209
  6831 6879 6904 6905 6906
  6907 6908 6910 7209 7219
  7222 7229 7240 7269
CAPSLOCK 6812
  6812 6845 6986
cgaputc 7155
  7155 7198
clearpteu 1903
  0431 1903 1909 5755
cli 0557
  0557 0559 1126 1560 7110
  7189 8212
cmd 7665
  7665 7677 7686 7687 7692
  7693 7698 7702 7706 7715
  7718 7723 7731 7737 7741
  7751 7775 7777 7852 7855
  7857 7858 7859 7860 7863
  7864 7866 7868 7869 7870
  7871 7872 7873 7874 7875
  7876 7879 7880 7882 7884
  7885 7886 7887 7888 7889
  7900 7901 7903 7905 7906
  7907 7908 7909 7910 7913
  7914 7916 7918 7919 7920
  7921 7922 8012 8013 8014
  8015 8017 8021 8024 8030
  8031 8034 8037 8039 8042
  8046 8048 8050 8053 8055
  8058 8060 8063 8064 8075
  8078 8081 8085 8100 8103
  8108 8112 8113 8116 8121
  8122 8128 8137 8138 8144
  8145 8151 8152 8161 8164
  8166 8172 8173 8178 8184
  8190 8191 8194
COM1 7413
  7413 7423 7426 7427 7428
  7429 7430 7431 7434 7440
  7441 7457 7459 7467 7469
commit_trans 4136
  0335 4136 5073 5179 5328
  5346 5355 5445 5452 5513
  5558 5562 5579 5583
CONSOLE 3639
  3639 7321 7322
consoleinit 7316
  0267 1225 7316
consoleintr 7212
  0269 6998 7212 7475
consoleread 7251
  7251 7322
consolewrite 7301
  7301 7321
consputc 7186
  7016 7047 7069 7087 7090
  7094 7095 7186 7226 7232
  7239 7308
context 2043
  0251 0374 2006 2043 2061
  2188 2189 2190 2191 2428
  2466 2628
copyout 1968
  0430 1968 5763 5774
copyuvm 1916
  0427 1916 1927 1929 2264
cprintf 7052
  0268 1222 1264 1841 2626
  2630 2632 3040 3053 3058
  3285 6319 6339 6511 6662
  7052 7112 7113 7114 7117
cpu 2004
  0309 1222 1264 1266 1278
  1406 1466 1487 1508 1546
  1561 1562 1570 1572 1618
  1631 1637 1772 1773 1774
  1775 2004 2014 2018 2029
  2428 2459 2465 2466 2467
  3015 3040 3041 3053 3054
  3058 3060 6213 6214 6511
  7112
cpunum 6501
  0324 1256 1288 1624 6501
  6673 6682
CRO_PE 0727
  0727 1135 1171 8243
CRO_PG 0737
  0737 1050 1171
CRO_WP 0733
  0733 1050 1171
CR4_PSE 0739
  0739 1043 1164
create 5457
  5457 5477 5490 5494 5512

```

5557 5578
 CRTPORT 7151
 7151 7160 7161 7162 7163
 7178 7179 7180 7181
 CTL 6809
 6809 6835 6839 6985
 deallocvm 1855
 0423 1842 1855 1889 2240
 DEVSPACE 0204
 0204 1729 1742
 devsw 3632
 3632 3637 4708 4710 4758
 4760 5007 7321 7322
 dinode 3572
 3572 3582 4406 4412 4430
 4433 4505 4518
 dirent 3596
 3596 4815 4855 5364 5404
 dirlink 4852
 0286 4822 4852 4867 4875
 5339 5489 5493 5494
 dirlookup 4812
 0287 4812 4818 4859 4974
 5421 5467
 DIRSIZ 3594
 3594 3598 4805 4872 4928
 4929 4991 5315 5405 5461
 DPL_USER 0779
 0779 1627 1628 2214 2215
 2973 3068 3077
 EOESC 6816
 6816 6970 6974 6975 6977
 6980
 elfhdr 0955
 0955 5715 8319 8324
 ELF_MAGIC 0952
 0952 5728 8330
 ELF_PROG_LOAD 0986
 0986 5739
 enter_alloc 2725
 0314 1299 1755 2725
 entry 1040
 0961 1036 1039 1040 2731
 2852 2853 5787 6121 8321
 8345 8346
 EOI 6414
 6414 6484 6525
 ERROR 6435
 6435 6477
 ESR 6417

6417 6480 6481
 exec 5710
 0273 5642 5710 7568 7629
 7630 7726 7727
 EXEC 7657
 7657 7722 7859 8165
 execcmd 7669 7853
 7669 7710 7723 7853 7855
 8121 8127 8128 8156 8166
 exit 2304
 0359 2304 2340 3005 3009
 3069 3078 3317 7516 7519
 7561 7626 7631 7716 7725
 7735 7780 7828 7835
 EXTMEM 0202
 0202 0208
 fdalloc 5232
 5232 5258 5526 5662
 fetchint 3167
 0398 3167 3197 5633
 fetchstr 3179
 0399 3179 3226 5639
 file 3600
 0252 0276 0277 0278 0280
 0281 0282 0351 2064 3600
 4271 5004 5010 5020 5023
 5026 5038 5039 5052 5054
 5079 5102 5152 5207 5213
 5216 5232 5253 5267 5279
 5292 5303 5505 5654 5806
 5821 7010 7408 7678 7733
 7734 7864 7872 8072
 filealloc 5021
 0276 5021 5526 5827
 fileclose 5052
 0277 2315 5052 5058 5297
 5528 5665 5666 5854 5856
 filedup 5039
 0278 2279 5039 5043 5260
 fileinit 5014
 0279 1230 5014
 fileread 5102
 0280 5102 5117 5273
 filestat 5079
 0281 5079 5308
 filewrite 5152
 0282 5152 5184 5189 5285
 FL_IF 0710
 0710 1562 1568 2218 2463
 6508

fork 2254
 0360 2254 3311 7560 7623
 7625 7843 7845
 fork1 7839
 7700 7742 7754 7761 7776
 7824 7839
 forkret 2483
 2117 2191 2483
 freevm 1883
 0424 1883 1888 1940 2371
 5790 5795
 gatedesc 0901
 0523 0526 0901 2961
 getcallerpcs 1526
 0378 1488 1526 2628 7115
 getcmd 7784
 7784 7815
 gettoken 7956
 7956 8041 8045 8057 8070
 8071 8107 8111 8133
 growproc 2231
 0361 2231 3359
 havedisk1 3678
 3678 3714 3812
 holding 1544
 0379 1477 1504 1544 2457
 ialloc 4402
 0288 4402 4422 5476 5477
 IBLOCK 3585
 3585 4411 4432 4517
 I_BUSY 3627
 3627 4511 4513 4536 4540
 4557 4559
 ICRHI 6428
 6428 6487 6556 6568
 ICRL0 6418
 6418 6488 6489 6557 6559
 6569
 ID 6411
 6411 6447 6516
 IDE_BSY 3663
 3663 3687
 IDE_CMD_READ 3668
 3668 3741
 IDE_CMD_WRITE 3669
 3669 3738
 IDE_DF 3665
 3665 3689
 IDE_DRDY 3664
 3664 3687

IDE_ERR 3666
 3666 3689
 ideinit 3701
 0304 1232 3701
 ideintr 3752
 0305 3024 3752
 idelock 3675
 3675 3705 3757 3759 3778
 3815 3830 3833
 iderw 3804
 0306 3804 3809 3811 3813
 3958 3969
 idestart 3725
 3679 3725 3728 3776 3825
 idewait 3683
 3683 3708 3730 3766
 idtinit 2979
 0406 1265 2979
 idup 4488
 0289 2280 4488 4961
 igit 4453
 4394 4418 4453 4473 4830
 4959
 iinit 4389
 0290 1231 4389
 ilock 4502
 0291 4502 4508 4528 4964
 5082 5111 5175 5325 5338
 5351 5415 5423 5465 5469
 5479 5519 5608 5722 7263
 7283 7310
 inb 0453
 0453 3687 3713 6354 6964
 6967 7161 7163 7434 7440
 7441 7457 7467 7469 8223
 8231 8354
 initlock 1462
 0380 1462 2125 2744 2975
 3705 3893 4061 4391 5016
 5835 7318 7319
 initlog 4055
 0332 2494 4055 4058
 initvm 1786
 0425 1786 1791 2211
 inode 3613
 0253 0286 0287 0288 0289
 0291 0292 0293 0294 0295
 0297 0298 0299 0300 0301
 0426 1803 2065 3606 3613
 3633 3634 4274 4385 4394

4401 4427 4452 4455 4461
 4487 4488 4502 4534 4552
 4574 4610 4654 4685 4702
 4752 4811 4812 4852 4856
 4953 4956 4988 4995 5316
 5361 5403 5456 5460 5506
 5554 5569 5604 5716 7251
 7301
 INPUT_BUF 7200
 7200 7203 7224 7236 7238
 7240 7268
 insl 0462
 0462 0464 3767 8373
 install_trans 4071
 4071 4119 4140
 INT_DISABLED 6619
 6619 6667
 ioapic 6627
 6307 6329 6330 6624 6627
 6636 6637 6643 6644 6658
 IOAPIC 6608
 6608 6658
 ioapicenable 6673
 0309 3707 6673 7326 7443
 ioapicid 6217
 0310 6217 6330 6347 6661
 6662
 ioapicinit 6651
 0311 1224 6651 6662
 ioapicread 6634
 6634 6659 6660
 ioapicwrite 6641
 6641 6667 6668 6681 6682
 IO_PIC1 6707
 6707 6720 6735 6744 6747
 6752 6762 6776 6777
 IO_PIC2 6708
 6708 6721 6736 6765 6766
 6767 6770 6779 6780
 IO_RTC 6535
 6535 6548 6549
 IO_TIMER1 7359
 7359 7368 7378 7379
 IPB 3582
 3582 3585 3591 4412 4433
 4518
 iput 4552
 0292 2320 4552 4558 4577
 4860 4982 5072 5344 5614
 IRQ_COM1 2833

2833 3034 7442 7443
 IRQ_ERROR 2835
 2835 6477
 IRQ_IDE 2834
 2834 3023 3027 3706 3707
 IRQ_KBD 2832
 2832 3030 7325 7326
 IRQ_SLAVE 6710
 6710 6714 6752 6767
 IRQ_SPURIOUS 2836
 2836 3039 6457
 IRQ_TIMER 2831
 2831 3014 3073 6464 7380
 isdirempty 5361
 5361 5368 5427
 ismp 6215
 0338 1233 6215 6312 6320
 6340 6343 6655 6675
 itrunc 4654
 4274 4561 4654
 iunlock 4534
 0293 4534 4537 4576 4971
 5084 5114 5178 5334 5532
 5613 7256 7305
 iunlockput 4574
 0294 4574 4966 4975 4978
 5327 5340 5343 5354 5428
 5439 5443 5451 5468 5472
 5496 5521 5529 5561 5582
 5610 5748 5797
 iupdate 4427
 0295 4427 4563 4680 4778
 5333 5353 5437 5442 5483
 5487
 I_INVALID 3628
 3628 4516 4526 4555
 kalloc 2777
 0315 1792 1794 1839 1846
 1923 1931 1934 2173 2209
 2777 5731 5829
 KBDATAP 6804
 6804 6967
 kbdgetc 6956
 6956 6998
 kbdirtr 6996
 0321 3031 6996
 KBS_DIB 6803
 6803 6965
 KBSTATP 6802
 6802 6964

KERNBASE 0207
 0207 0208 0212 0213 0217
 0218 0220 0221 1321 1533
 1832 1889 2730
 KERNLINK 0208
 0208 1727
 KEY_DEL 6828
 6828 6869 6891 6915
 KEY_DN 6822
 6822 6865 6887 6911
 KEY_END 6820
 6820 6868 6890 6914
 KEY_HOME 6819
 6819 6868 6890 6914
 KEY_INS 6827
 6827 6869 6891 6915
 KEY_LF 6823
 6823 6867 6889 6913
 KEY_PGDN 6826
 6826 6866 6888 6912
 KEY_PGUP 6825
 6825 6866 6888 6912
 KEY_RT 6824
 6824 6867 6889 6913
 KEY_UP 6821
 6821 6865 6887 6911
 kfree 2756
 0316 1871 1873 1893 1896
 2265 2369 2747 2756 2761
 5852 5873
 kill 2575
 0362 2575 3059 3334 7567
 kinit 2740
 0317 1236 2740
 KSTACKSIZE 0151
 0151 1054 1063 1300 1775
 2177
 kvmalloc 1753
 0418 1218 1753
 lapiceoi 6522
 0326 3021 3025 3032 3036
 3042 6522
 lapicinit 6451
 0327 1220 1256 6451
 lapicstartap 6540
 0328 1304 6540
 lapicw 6444
 6444 6457 6463 6464 6465
 6468 6469 6474 6477 6480
 6481 6484 6487 6488 6493
 6525 6556 6557 6559 6568
 6569
 lcr3 0590
 0590 1764 1779
 lgdt 0512
 0512 0520 1133 1633 8241
 lidt 0526
 0526 0534 2981
 LINT0 6433
 6433 6468
 LINT1 6434
 6434 6469
 LIST 7660
 7660 7740 7907 8183
 listcmd 7690 7901
 7690 7711 7741 7901 7903
 8046 8157 8184
 loadgs 0551
 0551 1634
 loadvm 1803
 0426 1803 1809 1812 5745
 log 4040 4050
 4040 4050 4061 4063 4064
 4065 4075 4076 4077 4089
 4092 4093 4094 4104 4107
 4108 4109 4120 4127 4128
 4129 4131 4132 4138 4141
 4145 4146 4147 4148 4163
 4165 4168 4169 4172 4173
 4177 4178
 logheader 4035
 4035 4046 4057 4058 4090
 4105
 LOGSIZE 0160
 0160 4037 4163 5167
 log_write 4159
 0333 4159 4295 4318 4344
 4416 4440 4630 4772
 ltr 0538
 0538 0540 1776
 mappages 1679
 1679 1745 1794 1846 1934
 MAXARG 0159
 0159 5622 5714 5760
 MAXARGS 7663
 7663 7671 7672 8140
 MAXFILE 3569
 3569 4765
 memcmp 5965
 0386 5965 6245 6288

```

memmove 5981
  0387 1285 1795 1933 1982
  4078 4174 4283 4439 4524
  4721 4771 4929 4931 5981
  6004 7173
memset 5954
  0388 1666 1741 1793 1845
  2190 2213 2733 2764 4294
  4414 5432 5629 5954 7175
  7787 7858 7869 7885 7906
  7919
microdelay 6531
  0329 6531 6558 6560 6570
  7458
min 4273
  4273 4720 4770
mp 6102
  6102 6208 6237 6244 6245
  6246 6255 6260 6264 6265
  6268 6269 6280 6283 6285
  6287 6294 6304 6310 6350
mpbcpu 6220
  0339 1220 6220
MPBUS 6152
  6152 6333
mpconf 6113
  6113 6279 6282 6287 6305
mpconfig 6280
  6280 6310
mpenter 1252
  1252 1301
mpinit 6301
  0340 1219 6301 6319 6339
mpioapic 6139
  6139 6307 6329 6331
MPIOAPIC 6153
  6153 6328
MPIOINTR 6154
  6154 6334
MPLINTR 6155
  6155 6335
mpmain 1262
  1209 1239 1257 1262
mpproc 6128
  6128 6306 6317 6326
MPPROC 6151
  6151 6316
mpsearch 6256
  6256 6285
mpsearch1 6238
  6238 6264 6268 6271
  multiboot_header 1025
  1024 1025
  namecmp 4803
  0296 4803 4825 5418
  namei 4989
  0297 2223 4989 5320 5517
  5606 5720
  nameiparent 4996
  0298 4954 4969 4981 4996
  5336 5410 5463
  namex 4954
  4954 4992 4998
  NBUF 0155
  0155 3881 3903
  ncpu 6216
  1222 1287 2019 3707 6216
  6318 6319 6323 6324 6325
  6345
  NCPU 0152
  0152 2018 6213
  NDEV 0157
  0157 4708 4758 5007
  NDIRECT 3567
  3567 3569 3578 3624 4615
  4620 4624 4625 4660 4667
  4668 4675 4676
  NELEM 0434
  0434 1744 2622 3282 5631
  nextpid 2116
  2116 2169
  NFILE 0154
  0154 5010 5026
  NINDIRECT 3568
  3568 3569 4622 4670
  NINODE 0156
  0156 4385 4461
  NO 6806
  6806 6852 6855 6857 6858
  6859 6860 6862 6874 6877
  6879 6880 6881 6882 6884
  6902 6903 6905 6906 6907
  6908
  NOFILE 0153
  0153 2064 2277 2313 5220
  5236
  NPENTRIES 0821
  0821 1317 1890
  NPROC 0150
  0150 2111 2161 2329 2362

```

```

  2418 2557 2580 2619
  NPTENTRIES 0822
  0822 1867
  NSEGS 2001
  1611 2001 2008
  nulterminate 8152
  8015 8030 8152 8173 8179
  8180 8185 8186 8191
  NUMLOCK 6813
  6813 6846
  O_CREATE 3453
  3453 5510 8078 8081
  O_RDONLY 3450
  3450 5520 8075
  O_RDWR 3452
  3452 5538 7614 7616 7807
  outb 0471
  0471 3711 3720 3731 3732
  3733 3734 3735 3736 3738
  3741 6353 6354 6548 6549
  6720 6721 6735 6736 6744
  6747 6752 6762 6765 6766
  6767 6770 6776 6777 6779
  6780 7160 7162 7178 7179
  7180 7181 7377 7378 7379
  7423 7426 7427 7428 7429
  7430 7431 7459 8228 8236
  8364 8365 8366 8367 8368
  8369
  outsl 0483
  0483 0485 3739
  outw 0477
  0477 1181 1183 8274 8276
  O_WRONLY 3451
  3451 5537 5538 8078 8081
  P2V 0218
  0218 1726 6262 6550 7152
  panic 7105 7832
  0270 1478 1505 1569 1571
  1691 1743 1778 1791 1809
  1812 1871 1888 1909 1927
  1929 2210 2310 2340 2458
  2460 2462 2464 2506 2509
  2731 2761 3055 3728 3809
  3811 3813 3946 3967 3977
  4058 4164 4166 4326 4342
  4422 4473 4508 4528 4537
  4558 4636 4818 4822 4867
  4875 5043 5058 5117 5184
  5189 5368 5426 5434 5477
  5490 5494 7063 7105 7112
  7701 7720 7753 7832 7845
  8028 8072 8106 8110 8136
  8141
  panicked 7018
  7018 7118 7188
  parseblock 8101
  8101 8106 8125
  parsecmd 8018
  7702 7825 8018
  parseexec 8117
  8014 8055 8117
  parseline 8035
  8012 8024 8035 8046 8108
  parsepipe 8051
  8013 8039 8051 8058
  parseredirs 8064
  8064 8112 8131 8142
  PCINT 6432
  6432 6474
  pde_t 0103
  0103 0420 0421 0422 0423
  0424 0425 0426 0427 0430
  0431 1210 1270 1317 1610
  1654 1656 1679 1733 1736
  1739 1786 1803 1827 1855
  1883 1903 1915 1916 1918
  1952 1968 2055 5718
  PDX 0812
  0812 1659
  PDXSHIFT 0827
  0812 0818 0827 1321
  peek 8001
  8001 8025 8040 8044 8056
  8069 8105 8109 8124 8132
  PGROUNDDOWN 0830
  0830 1685 1686 1975
  PGROUNDUP 0829
  0829 1837 1863 2732 2745
  5752
  PGSIZE 0823
  0823 0829 0830 1316 1666
  1695 1696 1741 1790 1793
  1794 1808 1810 1814 1817
  1838 1845 1846 1864 1867
  1925 1933 1934 1979 1985
  2212 2219 2733 2734 2746
  2760 2764 5753 5755
  PHYSTOP 0203
  0203 1728 1742 1743 2746

```

```

2760
picenable 6725
 0344 3706 6725 7325 7380
 7442
picinit 6732
 0345 1223 6732
picsetmask 6717
 6717 6727 6783
pinit 2123
 0363 1227 2123
pipe 5811
 0254 0352 0353 0354 3605
 5069 5109 5159 5811 5823
 5829 5835 5839 5843 5861
 5880 5901 7563 7752 7753
PIPE 7659
 7659 7750 7886 8177
pipealloc 5821
 0351 5659 5821
pipeclose 5861
 0352 5069 5861
pipecmd 7684 7880
 7684 7712 7751 7880 7882
 8058 8158 8178
piperead 5901
 0353 5109 5901
PIPESIZE 5809
 5809 5813 5886 5894 5916
pipewrite 5880
 0354 5159 5880
popcli 1566
 0383 1521 1566 1569 1571
 1780
printint 7026
 7026 7077 7081
proc 2053
 0255 0358 0398 0399 0428
 1205 1458 1606 1638 1769
 1775 2015 2030 2053 2059
 2106 2111 2114 2154 2157
 2161 2204 2235 2237 2240
 2243 2244 2257 2264 2270
 2271 2272 2278 2279 2280
 2284 2306 2309 2314 2315
 2316 2320 2321 2326 2329
 2330 2338 2355 2362 2363
 2383 2389 2410 2418 2425
 2428 2433 2461 2466 2475
 2505 2523 2524 2528 2555
 2557 2577 2580 2615 2619
2955 3004 3006 3008 3051
3059 3060 3062 3068 3073
3077 3155 3167 3179 3197
3210 3226 3279 3281 3283
3286 3287 3306 3340 3358
3375 3657 4267 4961 5205
5220 5237 5238 5296 5614
5615 5633 5639 5664 5704
5781 5784 5785 5786 5787
5788 5789 5804 5887 5907
6211 6306 6317 6318 6319
6322 7013 7261 7410
procdump 2604
 0364 2604 7220
proghdr 0974
 0974 5717 8320 8334
PTE_ADDR 0844
 0844 1661 1813 1869 1892
 1930 1961
PTE_P 0833
 0833 1319 1321 1660 1670
 1690 1692 1868 1891 1928
 1957
PTE_PS 0840
 0840 1319 1321
pte_t 0847
 0847 1653 1657 1661 1663
 1683 1806 1857 1905 1919
 1954
PTE_U 0835
 0835 1670 1794 1846 1910
 1934 1959
PTE_W 0834
 0834 1319 1321 1670 1726
 1728 1729 1794 1846 1934
PTX 0815
 0815 1672
PTXSHIFT 0826
 0815 0818 0826
pushcli 1555
 0382 1476 1555 1771
rcr2 0582
 0582 3054 3061
readeflags 0544
 0544 1559 1568 2463 6508
read_head 4087
 4087 4118
readi 4702
 0299 1818 4702 4821 4866
 5112 5367 5368 5726 5737

```

```

readsb 4278
 0285 4062 4278 4311 4337
 4409
readsect 8360
 8360 8395
readseg 8379
 8314 8327 8338 8379
recover_from_log 4116
 4052 4066 4116
REDIR 7658
 7658 7730 7870 8171
redircmd 7675 7864
 7675 7713 7731 7864 7866
 8075 8078 8081 8159 8172
REG_ID 6610
 6610 6660
REG_TABLE 6612
 6612 6667 6668 6681 6682
REG_VER 6611
 6611 6659
release 1502
 0381 1502 1505 2164 2170
 2377 2384 2435 2477 2487
 2519 2532 2568 2586 2590
 2770 2785 3019 3376 3381
 3394 3759 3778 3833 3928
 3942 3991 4132 4148 4464
 4480 4492 4514 4542 4560
 4569 5029 5033 5045 5060
 5066 5872 5875 5888 5897
 5908 5919 7101 7248 7262
 7282 7309
ROOTDEV 0158
 0158 4062 4065 4959
ROOTINO 3556
 3556 4959
run 2711
 2611 2711 2712 2717 2758
 2767 2779
runcmd 7706
 7706 7720 7737 7743 7745
 7759 7766 7777 7825
RUNNING 2050
 2050 2427 2461 2611 3073
safestrcpy 6032
 0389 2222 2284 5781 6032
sched 2453
 0366 2339 2453 2458 2460
 2462 2464 2476 2525
scheduler 2408
0365 1267 2006 2408 2428
2466
SCROLLLOCK 6814
 6814 6847
SECTSIZE 8312
 8312 8373 8386 8389 8394
SEG 0769
 0769 1625 1626 1627 1628
 1631
SEG16 0773
 0773 1772
SEG_ASM 0660
 0660 1190 1191 8284 8285
segdesc 0752
 0509 0512 0752 0769 0773
 1611 2008
seginit 1616
 0417 1221 1255 1616
SEG_KCODE 0741
 0741 1150 1625 2972 2973
 8253
SEG_KCPU 0743
 0743 1631 1634 2916
SEG_KDATA 0742
 0742 1154 1626 1774 2913
 8258
SEG_NULLASM 0654
 0654 1189 8283
SEG_TSS 0746
 0746 1772 1773 1776
SEG_UCODE 0744
 0744 1627 2214
SEG_UDATA 0745
 0745 1628 2215
SETGATE 0921
 0921 2972 2973
setupkvm 1734
 0420 1734 1755 1923 2209
 5731
SHIFT 6808
 6808 6836 6837 6985
skipelem 4915
 4915 4963
sleep 2503
 0367 2389 2503 2506 2509
 2609 3379 3830 3931 4129
 4512 5892 5911 7266 7579
spinlock 1401
 0256 0367 0377 0379 0380
 0381 0409 1401 1459 1462

```

```

1474 1502 1544 2107 2110
2503 2709 2716 2958 2963
3660 3675 3876 3880 4003
4041 4268 4384 5005 5009
5807 5812 7008 7021 7202
7406
STA_R 0669 0786
0669 0786 1190 1625 1627
8284
start 1125 7508 8211
1124 1125 1167 1175 1177
4042 4063 4076 4089 4104
4173 7507 7508 8210 8211
8267
startothers 1274
1208 1235 1274
stat 3504
0257 0281 0300 3504 4265
4685 5079 5203 5304 7603
stati 4685
0300 4685 5083
STA_W 0668 0785
0668 0785 1191 1626 1628
1631 8285
STA_X 0665 0782
0665 0782 1190 1625 1627
8284
sti 0563
0563 0565 1573 2414
stosb 0492
0492 0494 5960 8340
stosl 0501
0501 0503 5958
strlen 6051
0390 5762 5763 6051 7819
8023
strncmp 6008
0391 4805 6008
strncpy 6018
0392 4872 6018
STS_IG32 0800
0800 0927
STS_T32A 0797
0797 1772
STS_TG32 0801
0801 0927
sum 6226
6226 6228 6230 6232 6233
6245 6292
superblock 3560
0258 0285 3560 4060 4278
4308 4334 4407
SVR 6415
6415 6457
switchkvm 1762
0429 1254 1756 1762 2429
switchvm 1769
0428 1769 1778 2244 2426
5789
swtch 2658
0374 2428 2466 2657 2658
syscall 3275
0400 3007 3157 3275
SYSCALL 7553 7560 7561 7562 7563 75
7560 7561 7562 7563 7564
7565 7566 7567 7568 7569
7570 7571 7572 7573 7574
7575 7576 7577 7578 7579
7580
sys_chdir 5601
3229 3259 5601
SYS_chdir 3109
3109 3259
sys_close 5289
3230 3271 5289
SYS_close 3122
3122 3271
sys_dup 5251
3231 3260 5251
SYS_dup 3110
3110 3260
sys_exec 5620
3232 3257 5620
SYS_exec 3107
3107 3257 7512
sys_exit 3315
3233 3252 3315
SYS_exit 3102
3102 3252 7517
sys_fork 3309
3234 3251 3309
SYS_fork 3101
3101 3251
sys_fstat 5301
3235 3258 5301
SYS_fstat 3108
3108 3258
sys_getpid 3338
3236 3261 3338
SYS_getpid 3111

```

```

3111 3261
sys_kill 3328
3237 3256 3328
SYS_kill 3106
3106 3256
sys_link 5313
3238 3269 5313
SYS_link 3120
3120 3269
sys_mkdir 5551
3239 3270 5551
SYS_mkdir 3121
3121 3270
sys_mknod 5567
3240 3267 5567
SYS_mknod 3118
3118 3267
sys_open 5501
3241 3265 5501
SYS_open 3116
3116 3265 3280 3282
sys_pipe 5651
3242 3254 5651
SYS_pipe 3104
3104 3254
sys_read 5265
3243 3255 5265
SYS_read 3105
3105 3255
sys_sbrk 3351
3244 3262 3351
SYS_sbrk 3112
3112 3262
sys_sleep 3365
3245 3263 3365
SYS_sleep 3113
3113 3263
sys_unlink 5401
3246 3268 5401
SYS_unlink 3119
3119 3268
sys_uptime 3388
3249 3264 3388
SYS_uptime 3114
3114 3264
sys_wait 3322
3247 3253 3322
SYS_wait 3103
3103 3253
sys_write 5277
3248 3266 5277
SYS_write 3117
3117 3266
taskstate 0851
0851 2007
TDCR 6439
6439 6463
T_DEV 3502
3502 4707 4757 5578
T_DIR 3500
3500 4817 4965 5326 5427
5435 5485 5520 5557 5609
T_FILE 3501
3501 5470 5512
ticks 2964
0407 2964 3017 3018 3373
3374 3379 3393
tickslock 2963
0409 2963 2975 3016 3019
3372 3376 3379 3381 3392
3394
TICR 6437
6437 6465
TIMER 6429
6429 6464
TIMER_16BIT 7371
7371 7377
TIMER_DIV 7366
7366 7378 7379
TIMER_FREQ 7365
7365 7366
timerinit 7374
0403 1234 7374
TIMER_MODE 7368
7368 7377
TIMER_RATEGEN 7370
7370 7377
TIMER_SELO 7369
7369 7377
T_IRQO 2829
2829 3014 3023 3027 3030
3034 3038 3039 3073 6457
6464 6477 6667 6681 6747
6766
TPR 6413
6413 6493
trap 3001
2852 2854 2922 3001 3053
3055 3058
trapframe 0602

```

0602 2060 2181 3001	VER 6412
trapret 2927	6412 6473
2118 2186 2926 2927	wait 2353
T_SYSCALL 2826	0369 2353 3324 7562 7633
2826 2973 3003 7513 7518	7744 7770 7771 7826
7557	waitdisk 8351
tvinit 2967	8351 8363 8372
0408 1228 2967	wakeup 2564
uart 7415	0370 2564 3018 3772 3989
7415 7436 7455 7465	4147 4541 4566 5866 5869
uartgetc 7463	5891 5896 5918 7242
7463 7475	wakeup1 2553
uartinit 7418	2120 2326 2333 2553 2567
0412 1226 7418	walkpgdir 1654
uartintr 7473	1654 1688 1811 1865 1907
0413 3035 7473	1926 1956
uartputc 7451	write_head 4102
0414 7195 7197 7447 7451	4102 4121 4139 4142
userinit 2202	writei 4752
0368 1237 2202 2210	0301 4752 4874 5176 5433
uva2ka 1952	5434
0421 1952 1976	xchg 0569
V2P 0217	0569 1266 1483 1519
0217 1727 1728	yield 2472
V2P_WO 0220	0371 2472 3074
0220 1036 1046	


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

```
0150 #define NPROC          64 // maximum number of processes
0151 #define KSTACKSIZE 4096 // size of per-process kernel stack
0152 #define NCPU           8 // maximum number of CPUs
0153 #define NOFILE         16 // open files per process
0154 #define NFILE          100 // open files per system
0155 #define NBUF           10 // size of disk block cache
0156 #define NINODE         50 // maximum number of active i-nodes
0157 #define NDEV           10 // maximum major device number
0158 #define ROOTDEV        1 // device number of file system root disk
0159 #define MAXARG         32 // max exec arguments
0160 #define LOGSIZE        10 // max data sectors in on-disk log
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 // Memory layout
0201
0202 #define EXTMEM 0x100000 // Start of extended memory
0203 #define PHYSTOP 0xE000000 // Top physical memory
0204 #define DEVSPACE 0xFE000000 // Other devices are at high addresses
0205
0206 // Key addresses for address space layout (see kmap in vm.c for layout)
0207 #define KERNBASE 0x8000000 // First kernel virtual address
0208 #define KERNLINK (KERNBASE+EXTMEM) // Address where kernel is linked
0209
0210 #ifndef __ASSEMBLER__
0211
0212 static inline uint v2p(void *a) { return (uint) a - KERNBASE; }
0213 static inline void *p2v(uint a) { return (void *) a + KERNBASE; }
0214
0215 #endif
0216
0217 #define V2P(a) ((uint) a - KERNBASE)
0218 #define P2V(a) ((void *) a + KERNBASE)
0219
0220 #define V2P_WO(x) ((x) - KERNBASE) // same as V2P, but without casts
0221 #define P2V_WO(x) ((x) + KERNBASE) // same as P2V, but without casts
0222
0223
0224
0225
0226
0227
0228
0229
0230
0231
0232
0233
0234
0235
0236
0237
0238
0239
0240
0241
0242
0243
0244
0245
0246
0247
0248
0249

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

0700 // This file contains definitions for the
0701 // x86 memory management unit (MMU).
0702
0703 // Eflags register
0704 #define FL_CF          0x00000001 // Carry Flag
0705 #define FL_PF          0x00000004 // Parity Flag
0706 #define FL_AF          0x00000010 // Auxiliary carry Flag
0707 #define FL_ZF          0x00000040 // Zero Flag
0708 #define FL_SF          0x00000080 // Sign Flag
0709 #define FL_TF          0x00000100 // Trap Flag
0710 #define FL_IF          0x00000200 // Interrupt Enable
0711 #define FL_DF          0x00000400 // Direction Flag
0712 #define FL_OF          0x00000800 // Overflow Flag
0713 #define FL_IOPL_MASK  0x00003000 // I/O Privilege Level bitmask
0714 #define FL_IOPL_0     0x00000000 // IOPL == 0
0715 #define FL_IOPL_1     0x00001000 // IOPL == 1
0716 #define FL_IOPL_2     0x00002000 // IOPL == 2
0717 #define FL_IOPL_3     0x00003000 // IOPL == 3
0718 #define FL_NT          0x00004000 // Nested Task
0719 #define FL_RF          0x00010000 // Resume Flag
0720 #define FL_VM          0x00020000 // Virtual 8086 mode
0721 #define FL_AC          0x00040000 // Alignment Check
0722 #define FL_VIF         0x00080000 // Virtual Interrupt Flag
0723 #define FL_VIP         0x00100000 // Virtual Interrupt Pending
0724 #define FL_ID          0x00200000 // ID flag
0725
0726 // Control Register flags
0727 #define CRO_PE          0x00000001 // Protection Enable
0728 #define CRO_MP          0x00000002 // Monitor coProcessor
0729 #define CRO_EM          0x00000004 // Emulation
0730 #define CRO_TS          0x00000008 // Task Switched
0731 #define CRO_ET          0x00000010 // Extension Type
0732 #define CRO_NE          0x00000020 // Numeric Error
0733 #define CRO_WP          0x00010000 // Write Protect
0734 #define CRO_AM          0x00040000 // Alignment Mask
0735 #define CRO_NW          0x02000000 // Not Writethrough
0736 #define CRO_CD          0x40000000 // Cache Disable
0737 #define CRO_PG          0x80000000 // Paging
0738
0739 #define CR4_PSE        0x00000010 // Page size extension
0740
0741 #define SEG_KCODE 1 // kernel code
0742 #define SEG_KDATA 2 // kernel data+stack
0743 #define SEG_KCPU 3 // kernel per-cpu data
0744 #define SEG_UCODE 4 // user code
0745 #define SEG_UDATA 5 // user data+stack
0746 #define SEG_TSS 6 // this process's task state
0747
0748
0749

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

1150  jmpl  $(SEG_KCODE<<3), $(start32)
1151
1152 .code32
1153 start32:
1154 movw  $(SEG_KDATA<<3), %ax
1155 movw  %ax, %ds
1156 movw  %ax, %es
1157 movw  %ax, %ss
1158 movw  $0, %ax
1159 movw  %ax, %fs
1160 movw  %ax, %gs
1161
1162 # Turn on page size extension for 4Mbyte pages
1163 movl  %cr4, %eax
1164 orl   $(CR4_PSE), %eax
1165 movl  %eax, %cr4
1166 # Use enterpgdir as our initial page table
1167 movl  (start-12), %eax
1168 movl  %eax, %cr3
1169 # Turn on paging.
1170 movl  %cr0, %eax
1171 orl   $(CR0_PE|CR0_PG|CR0_WP), %eax
1172 movl  %eax, %cr0
1173
1174 # Switch to the stack allocated by startothers()
1175 movl  (start-4), %esp
1176 # Call mpenter()
1177 call  *(start-8)
1178
1179 movw  $0x8a00, %ax
1180 movw  %ax, %dx
1181 outw  %ax, %dx
1182 movw  $0x8ae0, %ax
1183 outw  %ax, %dx
1184 spin:
1185 jmp  spin
1186
1187 .p2align 2
1188 gdt:
1189 SEG_NULLASM
1190 SEG_ASM(STA_X|STA_R, 0, 0xffffffff)
1191 SEG_ASM(STA_W, 0, 0xffffffff)
1192
1193
1194 gdtdesc:
1195 .word  (gdtdesc - gdt - 1)
1196 .long  gdt
1197
1198
1199

```

```

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

```

```

1250 // Other CPUs jump here from entryother.S.
1251 static void
1252 mpenter(void)
1253 {
1254     switchkvm();
1255     seginit();
1256     lapicinit(cpunum());
1257     mpmain();
1258 }
1259
1260 // Common CPU setup code.
1261 static void
1262 mpmain(void)
1263 {
1264     printf("cpu%d: starting\n", cpu->id);
1265     idtinit(); // load idt register
1266     xchg(&cpu->started, 1); // tell startothers() we're up
1267     scheduler(); // start running processes
1268 }
1269
1270 pde_t entrypgdir[]; // For entry.S
1271
1272 // Start the non-boot (AP) processors.
1273 static void
1274 startothers(void)
1275 {
1276     extern uchar _binary_entryother_start[], _binary_entryother_size[];
1277     uchar *code;
1278     struct cpu *c;
1279     char *stack;
1280
1281     // Write entry code to unused memory at 0x7000.
1282     // The linker has placed the image of entryother.S in
1283     // _binary_entryother_start.
1284     code = p2v(0x7000);
1285     memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
1286
1287     for(c = cpus; c < cpus+ncpu; c++){
1288         if(c == cpus+cpunum()) // We've started already.
1289             continue;
1290
1291         // Tell entryother.S what stack to use, where to enter, and what
1292         // pgdir to use. We cannot use kpgdir yet, because the AP processor
1293         // is running in low memory, so we use entrypgdir for the APs too.
1294         // kalloc can return addresses above 4Mbyte (the machine may have
1295         // much more physical memory than 4Mbyte), which aren't mapped by
1296         // entrypgdir, so we must allocate a stack using enter_alloc();
1297         // this introduces the constraint that xv6 cannot use kalloc until
1298         // after these last enter_alloc invocations.
1299         stack = enter_alloc();

```

```

1300 *(void**)(code-4) = stack + KSTACKSIZE;
1301 *(void**)(code-8) = mpenter;
1302 *(int**)(code-12) = (void *) v2p(entrypgdir);
1303
1304   tapicstartap(c->id, v2p(code));
1305
1306   // wait for cpu to finish mpmain()
1307   while(c->started == 0)
1308     ;
1309 }
1310 }
1311
1312 // Boot page table used in entry.S and entryother.S.
1313 // Page directories (and page tables), must start on a page boundary,
1314 // hence the "__aligned__" attribute.
1315 // Use PTE_PS in page directory entry to enable 4Mbyte pages.
1316 __attribute__((__aligned__(PGSIZE)))
1317 pde_t entrypgdir[NPDENTRIES] = {
1318   // Map VA's [0, 4MB) to PA's [0, 4MB)
1319   [0] = (0) + PTE_P + PTE_W + PTE_PS,
1320   // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1321   [KERNBASE >> PDXSHIFT] = (0) + PTE_P + PTE_W + PTE_PS,
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 // Blank page.
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399

```

```

1400 // Mutual exclusion lock.
1401 struct spinlock {
1402     uint locked;        // Is the lock held?
1403
1404     // For debugging:
1405     char *name;        // Name of lock.
1406     struct cpu *cpu;   // The cpu holding the lock.
1407     uint pcs[10];     // The call stack (an array of program counters)
1408                     // that locked the lock.
1409 };
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449

```

```

1450 // Mutual exclusion spin locks.
1451
1452 #include "types.h"
1453 #include "defs.h"
1454 #include "param.h"
1455 #include "x86.h"
1456 #include "memlayout.h"
1457 #include "mmu.h"
1458 #include "proc.h"
1459 #include "spinlock.h"
1460
1461 void
1462 initlock(struct spinlock *lk, char *name)
1463 {
1464     lk->name = name;
1465     lk->locked = 0;
1466     lk->cpu = 0;
1467 }
1468
1469 // Acquire the lock.
1470 // Loops (spins) until the lock is acquired.
1471 // Holding a lock for a long time may cause
1472 // other CPUs to waste time spinning to acquire it.
1473 void
1474 acquire(struct spinlock *lk)
1475 {
1476     pushcli(); // disable interrupts to avoid deadlock.
1477     if(holding(lk))
1478         panic("acquire");
1479
1480     // The xchg is atomic.
1481     // It also serializes, so that reads after acquire are not
1482     // reordered before it.
1483     while(xchg(&lk->locked, 1) != 0)
1484         ;
1485
1486     // Record info about lock acquisition for debugging.
1487     lk->cpu = cpu;
1488     getcallerpcs(&lk, lk->pcs);
1489 }
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499

```

```

1500 // Release the lock.
1501 void
1502 release(struct spinlock *lk)
1503 {
1504     if(!holding(lk))
1505         panic("release");
1506
1507     lk->pcs[0] = 0;
1508     lk->cpu = 0;
1509
1510     // The xchg serializes, so that reads before release are
1511     // not reordered after it. The 1996 PentiumPro manual (Volume 3,
1512     // 7.2) says reads can be carried out speculatively and in
1513     // any order, which implies we need to serialize here.
1514     // But the 2007 Intel 64 Architecture Memory Ordering White
1515     // Paper says that Intel 64 and IA-32 will not move a load
1516     // after a store. So lock->locked = 0 would work here.
1517     // The xchg being asm volatile ensures gcc emits it after
1518     // the above assignments (and after the critical section).
1519     xchg(&lk->locked, 0);
1520
1521     popcli();
1522 }
1523
1524 // Record the current call stack in pcs[] by following the %ebp chain.
1525 void
1526 getcallerpcs(void *v, uint pcs[])
1527 {
1528     uint *ebp;
1529     int i;
1530
1531     ebp = (uint*)v - 2;
1532     for(i = 0; i < 10; i++){
1533         if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
1534             break;
1535         pcs[i] = ebp[1]; // saved %eip
1536         ebp = (uint*)ebp[0]; // saved %ebp
1537     }
1538     for(; i < 10; i++)
1539         pcs[i] = 0;
1540 }
1541
1542 // Check whether this cpu is holding the lock.
1543 int
1544 holding(struct spinlock *lock)
1545 {
1546     return lock->locked && lock->cpu == cpu;
1547 }
1548
1549

```

```

1550 // Pushcli/popcli are like cli/sti except that they are matched:
1551 // it takes two popcli to undo two pushcli. Also, if interrupts
1552 // are off, then pushcli, popcli leaves them off.
1553
1554 void
1555 pushcli(void)
1556 {
1557     int eflags;
1558
1559     eflags = readeflags();
1560     cli();
1561     if(cpu->ncli++ == 0)
1562         cpu->intena = eflags & FL_IF;
1563 }
1564
1565 void
1566 popcli(void)
1567 {
1568     if(readeflags() & FL_IF)
1569         panic("popcli - interruptible");
1570     if(--cpu->ncli < 0)
1571         panic("popcli");
1572     if(cpu->ncli == 0 && cpu->intena)
1573         sti();
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 #include "param.h"
1601 #include "types.h"
1602 #include "defs.h"
1603 #include "x86.h"
1604 #include "memlayout.h"
1605 #include "mmu.h"
1606 #include "proc.h"
1607 #include "elf.h"
1608
1609 extern char data[]; // defined by kernel.ld
1610 pde_t *kpgdir; // for use in scheduler()
1611 struct segdesc gdt[NSEGS];
1612
1613 // Set up CPU's kernel segment descriptors.
1614 // Run once on entry on each CPU.
1615 void
1616 seginit(void)
1617 {
1618     struct cpu *c;
1619
1620     // Map "logical" addresses to virtual addresses using identity map.
1621     // Cannot share a CODE descriptor for both kernel and user
1622     // because it would have to have DPL_USR, but the CPU forbids
1623     // an interrupt from CPL=0 to DPL=3.
1624     c = &cpus[cpunum()];
1625     c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
1626     c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1627     c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
1628     c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1629
1630     // Map cpu, and curproc
1631     c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
1632
1633     lgdt(c->gdt, sizeof(c->gdt));
1634     loadgs(SEG_KCPU << 3);
1635
1636     // Initialize cpu-local storage.
1637     cpu = c;
1638     proc = 0;
1639 }
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649

```

```

1650 // Return the address of the PTE in page table pgdir
1651 // that corresponds to virtual address va. If alloc!=0,
1652 // create any required page table pages.
1653 static pte_t *
1654 walkpgdir(pde_t *pgdir, const void *va, char* (*alloc)(void))
1655 {
1656     pde_t *pde;
1657     pte_t *pgtab;
1658
1659     pde = &pgdir[PDX(va)];
1660     if(*pde & PTE_P){
1661         pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
1662     } else {
1663         if(!alloc || (pgtab = (pte_t*)alloc()) == 0)
1664             return 0;
1665         // Make sure all those PTE_P bits are zero.
1666         memset(pgtab, 0, PGSIZE);
1667         // The permissions here are overly generous, but they can
1668         // be further restricted by the permissions in the page table
1669         // entries, if necessary.
1670         *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
1671     }
1672     return &pgtab[PTX(va)];
1673 }
1674
1675 // Create PTEs for virtual addresses starting at va that refer to
1676 // physical addresses starting at pa. va and size might not
1677 // be page-aligned.
1678 static int
1679 mappages(pde_t *pgdir, void *va, uint size, uint pa,
1680          int perm, char* (*alloc)(void))
1681 {
1682     char *a, *last;
1683     pte_t *pte;
1684
1685     a = (char*)PGROUNDDOWN((uint)va);
1686     last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1687     for(;;){
1688         if((pte = walkpgdir(pgdir, a, alloc)) == 0)
1689             return -1;
1690         if(*pte & PTE_P)
1691             panic("remap");
1692         *pte = pa | perm | PTE_P;
1693         if(a == last)
1694             break;
1695         a += PGSIZE;
1696         pa += PGSIZE;
1697     }
1698     return 0;
1699 }

```



```

1700 // The mappings from logical to virtual are one to one (i.e.,
1701 // segmentation doesn't do anything). There is one page table per
1702 // process, plus one that's used when a CPU is not running any process
1703 // (kpgdir). A user process uses the same page table as the kernel; the
1704 // page protection bits prevent it from accessing kernel memory.
1705 //
1706 // setupkvm() and exec() set up every page table like this:
1707 // 0..KERNBASE: user memory (text+data+stack+heap), mapped to some free
1708 //      phys memory
1709 // KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
1710 // KERNBASE+EXTMEM..KERNBASE+end: mapped to EXTMEM.end kernel,
1711 //      w. no write permission
1712 // KERNBASE+end..KERNBASE+PHYSTOP: mapped to end..PHYSTOP,
1713 //      rw data + free memory
1714 // 0xfe000000..0: mapped direct (devices such as ioapic)
1715 //
1716 // The kernel allocates memory for its heap and for user memory
1717 // between KERNBASE+end and the end of physical memory (PHYSTOP).
1718 // The user program sits in the bottom of the address space, and the
1719 // kernel at the top at KERNBASE.
1720 static struct kmap {
1721     void *virt;
1722     uint phys_start;
1723     uint phys_end;
1724     int perm;
1725 } kmap[] = {
1726     { P2V(0), 0, 1024*1024, PTE_W}, // I/O space
1727     { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kernel text+rodata
1728     { data, V2P(data), PHYSTOP, PTE_W}, // kernel data, memory
1729     { (void*)DEVSPACE, DEVSPACE, 0, PTE_W}, // more devices
1730 };
1731
1732 // Set up kernel part of a page table.
1733 pde_t*
1734 setupkvm(char* (*alloc)(void))
1735 {
1736     pde_t *pgdir;
1737     struct kmap *k;
1738
1739     if((pgdir = (pde_t*)alloc()) == 0)
1740         return 0;
1741     memset(pgdir, 0, PGSIZE);
1742     if (p2v(PHYSTOP) > (void*)DEVSPACE)
1743         panic("PHYSTOP too high");
1744     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1745         if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1746             (uint)k->phys_start, k->perm, alloc) < 0)
1747             return 0;
1748     return pgdir;
1749 }

```

```

1750 // Allocate one page table for the machine for the kernel address
1751 // space for scheduler processes.
1752 void
1753 kvmalloc(void)
1754 {
1755     kpgdir = setupkvm(enter_alloc);
1756     switchkvm();
1757 }
1758
1759 // Switch h/w page table register to the kernel-only page table,
1760 // for when no process is running.
1761 void
1762 switchkvm(void)
1763 {
1764     lcr3(v2p(kpgdir)); // switch to the kernel page table
1765 }
1766
1767 // Switch TSS and h/w page table to correspond to process p.
1768 void
1769 switchvm(struct proc *p)
1770 {
1771     pushcli();
1772     cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
1773     cpu->gdt[SEG_TSS].s = 0;
1774     cpu->ts.ss0 = SEG_KDATA << 3;
1775     cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
1776     ltr(SEG_TSS << 3);
1777     if(p->pgdir == 0)
1778         panic("switchvm: no pgdir");
1779     lcr3(v2p(p->pgdir)); // switch to new address space
1780     popcli();
1781 }
1782
1783 // Load the initcode into address 0 of pgdir.
1784 // sz must be less than a page.
1785 void
1786 initvm(pde_t *pgdir, char *init, uint sz)
1787 {
1788     char *mem;
1789
1790     if(sz >= PGSIZE)
1791         panic("initvm: more than a page");
1792     mem = kalloc();
1793     memset(mem, 0, PGSIZE);
1794     mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U, kalloc);
1795     memmove(mem, init, sz);
1796 }
1797
1798
1799

```

```

1800 // Load a program segment into pgdir.  addr must be page-aligned
1801 // and the pages from addr to addr+sz must already be mapped.
1802 int
1803 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
1804 {
1805     uint i, pa, n;
1806     pte_t *pte;
1807
1808     if((uint) addr % PGSIZE != 0)
1809         panic("loaduvm: addr must be page aligned");
1810     for(i = 0; i < sz; i += PGSIZE){
1811         if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1812             panic("loaduvm: address should exist");
1813         pa = PTE_ADDR(*pte);
1814         if(sz - i < PGSIZE)
1815             n = sz - i;
1816         else
1817             n = PGSIZE;
1818         if(readi(ip, p2v(pa), offset+i, n) != n)
1819             return -1;
1820     }
1821     return 0;
1822 }
1823
1824 // Allocate page tables and physical memory to grow process from oldsz to
1825 // newsz, which need not be page aligned.  Returns new size or 0 on error.
1826 int
1827 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1828 {
1829     char *mem;
1830     uint a;
1831
1832     if(newsz >= KERNBASE)
1833         return 0;
1834     if(newsz < oldsz)
1835         return oldsz;
1836
1837     a = PGROUNDUP(oldsz);
1838     for(; a < newsz; a += PGSIZE){
1839         mem = kalloc();
1840         if(mem == 0){
1841             cprintf("allocuvm out of memory\n");
1842             deallocuvm(pgdir, newsz, oldsz);
1843             return 0;
1844         }
1845         memset(mem, 0, PGSIZE);
1846         mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U, kalloc);
1847     }
1848     return newsz;
1849 }

```

```

1850 // Deallocate user pages to bring the process size from oldsz to
1851 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
1852 // need to be less than oldsz.  oldsz can be larger than the actual
1853 // process size.  Returns the new process size.
1854 int
1855 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1856 {
1857     pte_t *pte;
1858     uint a, pa;
1859
1860     if(newsz >= oldsz)
1861         return oldsz;
1862
1863     a = PGROUNDUP(newsz);
1864     for(; a < oldsz; a += PGSIZE){
1865         pte = walkpgdir(pgdir, (char*)a, 0);
1866         if(!pte)
1867             a += (NPENTRIES - 1) * PGSIZE;
1868         else if((*pte & PTE_P) != 0){
1869             pa = PTE_ADDR(*pte);
1870             if(pa == 0)
1871                 panic("kfree");
1872             char *v = p2v(pa);
1873             kfree(v);
1874             *pte = 0;
1875         }
1876     }
1877     return newsz;
1878 }
1879
1880 // Free a page table and all the physical memory pages
1881 // in the user part.
1882 void
1883 freevm(pde_t *pgdir)
1884 {
1885     uint i;
1886
1887     if(pgdir == 0)
1888         panic("freevm: no pgdir");
1889     deallocuvm(pgdir, KERNBASE, 0);
1890     for(i = 0; i < NPENTRIES; i++){
1891         if(pgdir[i] & PTE_P){
1892             char *v = p2v(PTE_ADDR(pgdir[i]));
1893             kfree(v);
1894         }
1895     }
1896     kfree((char*)pgdir);
1897 }
1898
1899

```

```

1900 // Clear PTE_U on a page. Used to create an inaccessible
1901 // page beneath the user stack.
1902 void
1903 clearpteu(pde_t *pgdir, char *uva)
1904 {
1905     pte_t *pte;
1906
1907     pte = walkpgdir(pgdir, uva, 0);
1908     if(pte == 0)
1909         panic("clearpteu");
1910     *pte &= ~PTE_U;
1911 }
1912
1913 // Given a parent process's page table, create a copy
1914 // of it for a child.
1915 pde_t*
1916 copyuvm(pde_t *pgdir, uint sz)
1917 {
1918     pde_t *d;
1919     pte_t *pte;
1920     uint pa, i;
1921     char *mem;
1922
1923     if((d = setupkvm(kalloc)) == 0)
1924         return 0;
1925     for(i = 0; i < sz; i += PGSIZE){
1926         if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
1927             panic("copyuvm: pte should exist");
1928         if(!(*pte & PTE_P))
1929             panic("copyuvm: page not present");
1930         pa = PTE_ADDR(*pte);
1931         if((mem = kalloc()) == 0)
1932             goto bad;
1933         memmove(mem, (char*)p2v(pa), PGSIZE);
1934         if(mappages(d, (void*)i, PGSIZE, v2p(mem), PTE_W|PTE_U, kalloc) < 0)
1935             goto bad;
1936     }
1937     return d;
1938
1939 bad:
1940     freevm(d);
1941     return 0;
1942 }
1943
1944
1945
1946
1947
1948
1949

```

```

1950 // Map user virtual address to kernel address.
1951 char*
1952 uva2ka(pde_t *pgdir, char *uva)
1953 {
1954     pte_t *pte;
1955
1956     pte = walkpgdir(pgdir, uva, 0);
1957     if((*pte & PTE_P) == 0)
1958         return 0;
1959     if((*pte & PTE_U) == 0)
1960         return 0;
1961     return (char*)p2v(PTE_ADDR(*pte));
1962 }
1963
1964 // Copy len bytes from p to user address va in page table pgdir.
1965 // Most useful when pgdir is not the current page table.
1966 // uva2ka ensures this only works for PTE_U pages.
1967 int
1968 copyout(pde_t *pgdir, uint va, void *p, uint len)
1969 {
1970     char *buf, *pa0;
1971     uint n, va0;
1972
1973     buf = (char*)p;
1974     while(len > 0){
1975         va0 = (uint)PGROUNDDOWN(va);
1976         pa0 = uva2ka(pgdir, (char*)va0);
1977         if(pa0 == 0)
1978             return -1;
1979         n = PGSIZE - (va - va0);
1980         if(n > len)
1981             n = len;
1982         memmove(pa0 + (va - va0), buf, n);
1983         len -= n;
1984         buf += n;
1985         va = va0 + PGSIZE;
1986     }
1987     return 0;
1988 }
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999

```

```

2000 // Segments in proc->gdt.
2001 #define NSEGS    7
2002
2003 // Per-CPU state
2004 struct cpu {
2005     uchar id;                    // Local APIC ID; index into cpus[] below
2006     struct context *scheduler;   // swtch() here to enter scheduler
2007     struct taskstate ts;        // Used by x86 to find stack for interrupt
2008     struct segdesc gdt[NSEGS];  // x86 global descriptor table
2009     volatile uint started;      // Has the CPU started?
2010     int ncli;                   // Depth of pushcli nesting.
2011     int intena;                 // Were interrupts enabled before pushcli?
2012
2013     // Cpu-local storage variables; see below
2014     struct cpu *cpu;
2015     struct proc *proc;         // The currently-running process.
2016 };
2017
2018 extern struct cpu cpus[NCPU];
2019 extern int ncpu;
2020
2021 // Per-CPU variables, holding pointers to the
2022 // current cpu and to the current process.
2023 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2024 // and "%gs:4" to refer to proc. seginit sets up the
2025 // %gs segment register so that %gs refers to the memory
2026 // holding those two variables in the local cpu's struct cpu.
2027 // This is similar to how thread-local variables are implemented
2028 // in thread libraries such as Linux pthreads.
2029 extern struct cpu *cpu asm("%gs:0"); // &cpus[cpunum()]
2030 extern struct proc *proc asm("%gs:4"); // cpus[cpunum()].proc
2031
2032
2033 // Saved registers for kernel context switches.
2034 // Don't need to save all the segment registers (%cs, etc),
2035 // because they are constant across kernel contexts.
2036 // Don't need to save %eax, %ecx, %edx, because the
2037 // x86 convention is that the caller has saved them.
2038 // Contexts are stored at the bottom of the stack they
2039 // describe; the stack pointer is the address of the context.
2040 // The layout of the context matches the layout of the stack in swtch.S
2041 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2042 // but it is on the stack and allocproc() manipulates it.
2043 struct context {
2044     uint edi;
2045     uint esi;
2046     uint ebx;
2047     uint ebp;
2048     uint eip;
2049 };

```

```

2050 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2051
2052 // Per-process state
2053 struct proc {
2054     uint sz;                    // Size of process memory (bytes)
2055     pde_t* pgdir;               // Page table
2056     char *kstack;               // Bottom of kernel stack for this process
2057     enum procstate state;       // Process state
2058     volatile int pid;           // Process ID
2059     struct proc *parent;        // Parent process
2060     struct trapframe *tf;       // Trap frame for current syscall
2061     struct context *context;    // swtch() here to run process
2062     void *chan;                 // If non-zero, sleeping on chan
2063     int killed;                 // If non-zero, have been killed
2064     struct file *ofile[NOFILE]; // Open files
2065     struct inode *cwd;          // Current directory
2066     char name[16];              // Process name (debugging)
2067 };
2068
2069 // Process memory is laid out contiguously, low addresses first:
2070 // text
2071 // original data and bss
2072 // fixed-size stack
2073 // expandable heap
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099

```

```

2100 #include "types.h"
2101 #include "defs.h"
2102 #include "param.h"
2103 #include "memlayout.h"
2104 #include "mmu.h"
2105 #include "x86.h"
2106 #include "proc.h"
2107 #include "spinlock.h"
2108
2109 struct {
2110   struct spinlock lock;
2111   struct proc proc[NPROC];
2112 } ptable;
2113
2114 static struct proc *initproc;
2115
2116 int nextpid = 1;
2117 extern void forkret(void);
2118 extern void trapret(void);
2119
2120 static void wakeup1(void *chan);
2121
2122 void
2123 pinit(void)
2124 {
2125   initlock(&ptable.lock, "ptable");
2126 }
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149

```

```

2150 // Look in the process table for an UNUSED proc.
2151 // If found, change state to EMBRYO and initialize
2152 // state required to run in the kernel.
2153 // Otherwise return 0.
2154 static struct proc*
2155 allocproc(void)
2156 {
2157   struct proc *p;
2158   char *sp;
2159
2160   acquire(&ptable.lock);
2161   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2162     if(p->state == UNUSED)
2163       goto found;
2164   release(&ptable.lock);
2165   return 0;
2166
2167 found:
2168   p->state = EMBRYO;
2169   p->pid = nextpid++;
2170   release(&ptable.lock);
2171
2172   // Allocate kernel stack.
2173   if((p->kstack = kalloc()) == 0){
2174     p->state = UNUSED;
2175     return 0;
2176   }
2177   sp = p->kstack + KSTACKSIZE;
2178
2179   // Leave room for trap frame.
2180   sp -= sizeof *p->tf;
2181   p->tf = (struct trapframe*)sp;
2182
2183   // Set up new context to start executing at forkret,
2184   // which returns to trapret.
2185   sp -= 4;
2186   *(uint*)sp = (uint)trapret;
2187
2188   sp -= sizeof *p->context;
2189   p->context = (struct context*)sp;
2190   memset(p->context, 0, sizeof *p->context);
2191   p->context->eip = (uint)forkret;
2192
2193   return p;
2194 }
2195
2196
2197
2198
2199

```

```

2200 // Set up first user process.
2201 void
2202 userinit(void)
2203 {
2204     struct proc *p;
2205     extern char _binary_initcode_start[], _binary_initcode_size[];
2206
2207     p = allocproc();
2208     initproc = p;
2209     if((p->pgdir = setupkvm(kalloc)) == 0)
2210         panic("userinit: out of memory?");
2211     inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2212     p->sz = PGSIZE;
2213     memset(p->tf, 0, sizeof(*p->tf));
2214     p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2215     p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2216     p->tf->es = p->tf->ds;
2217     p->tf->ss = p->tf->ds;
2218     p->tf->eflags = FL_IF;
2219     p->tf->esp = PGSIZE;
2220     p->tf->eip = 0; // beginning of initcode.S
2221
2222     safestrcpy(p->name, "initcode", sizeof(p->name));
2223     p->cwd = namei("/");
2224
2225     p->state = RUNNABLE;
2226 }
2227
2228 // Grow current process's memory by n bytes.
2229 // Return 0 on success, -1 on failure.
2230 int
2231 growproc(int n)
2232 {
2233     uint sz;
2234
2235     sz = proc->sz;
2236     if(n > 0){
2237         if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2238             return -1;
2239     } else if(n < 0){
2240         if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2241             return -1;
2242     }
2243     proc->sz = sz;
2244     switchuvm(proc);
2245     return 0;
2246 }
2247
2248
2249

```

```

2250 // Create a new process copying p as the parent.
2251 // Sets up stack to return as if from system call.
2252 // Caller must set state of returned proc to RUNNABLE.
2253 int
2254 fork(void)
2255 {
2256     int i, pid;
2257     struct proc *np;
2258
2259     // Allocate process.
2260     if((np = allocproc()) == 0)
2261         return -1;
2262
2263     // Copy process state from p.
2264     if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
2265         kfree(np->kstack);
2266         np->kstack = 0;
2267         np->state = UNUSED;
2268         return -1;
2269     }
2270     np->sz = proc->sz;
2271     np->parent = proc;
2272     *np->tf = *proc->tf;
2273
2274     // Clear %eax so that fork returns 0 in the child.
2275     np->tf->eax = 0;
2276
2277     for(i = 0; i < NOFILE; i++){
2278         if(proc->ofile[i])
2279             np->ofile[i] = filedup(proc->ofile[i]);
2280     }
2281     np->cwd = idup(proc->cwd);
2282
2283     pid = np->pid;
2284     np->state = RUNNABLE;
2285     safestrcpy(np->name, proc->name, sizeof(proc->name));
2286     return pid;
2287 }
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299

```

```

2300 // Exit the current process. Does not return.
2301 // An exited process remains in the zombie state
2302 // until its parent calls wait() to find out it exited.
2303 void
2304 exit(void)
2305 {
2306     struct proc *p;
2307     int fd;
2308
2309     if(proc == initproc)
2310         panic("init exiting");
2311
2312     // Close all open files.
2313     for(fd = 0; fd < NOFILE; fd++){
2314         if(proc->ofile[fd]){
2315             fclose(proc->ofile[fd]);
2316             proc->ofile[fd] = 0;
2317         }
2318     }
2319
2320     iput(proc->cwd);
2321     proc->cwd = 0;
2322
2323     acquire(&ptable.lock);
2324
2325     // Parent might be sleeping in wait().
2326     wakeup1(proc->parent);
2327
2328     // Pass abandoned children to init.
2329     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2330         if(p->parent == proc){
2331             p->parent = initproc;
2332             if(p->state == ZOMBIE)
2333                 wakeup1(initproc);
2334         }
2335     }
2336
2337     // Jump into the scheduler, never to return.
2338     proc->state = ZOMBIE;
2339     sched();
2340     panic("zombie exit");
2341 }
2342
2343
2344
2345
2346
2347
2348
2349

```

```

2350 // Wait for a child process to exit and return its pid.
2351 // Return -1 if this process has no children.
2352 int
2353 wait(void)
2354 {
2355     struct proc *p;
2356     int havekids, pid;
2357
2358     acquire(&ptable.lock);
2359     for(;;){
2360         // Scan through table looking for zombie children.
2361         havekids = 0;
2362         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2363             if(p->parent != proc)
2364                 continue;
2365             havekids = 1;
2366             if(p->state == ZOMBIE){
2367                 // Found one.
2368                 pid = p->pid;
2369                 kfree(p->kstack);
2370                 p->kstack = 0;
2371                 freevm(p->pgdir);
2372                 p->state = UNUSED;
2373                 p->pid = 0;
2374                 p->parent = 0;
2375                 p->name[0] = 0;
2376                 p->killed = 0;
2377                 release(&ptable.lock);
2378                 return pid;
2379             }
2380         }
2381
2382         // No point waiting if we don't have any children.
2383         if(!havekids || proc->killed){
2384             release(&ptable.lock);
2385             return -1;
2386         }
2387
2388         // Wait for children to exit. (See wakeup1 call in proc_exit.)
2389         sleep(proc, &ptable.lock);
2390     }
2391 }
2392
2393
2394
2395
2396
2397
2398
2399

```

```

2400 // Per-CPU process scheduler.
2401 // Each CPU calls scheduler() after setting itself up.
2402 // Scheduler never returns. It loops, doing:
2403 // - choose a process to run
2404 // - swtch to start running that process
2405 // - eventually that process transfers control
2406 //   via swtch back to the scheduler.
2407 void
2408 scheduler(void)
2409 {
2410     struct proc *p;
2411
2412     for(;;){
2413         // Enable interrupts on this processor.
2414         sti();
2415
2416         // Loop over process table looking for process to run.
2417         acquire(&ptable.lock);
2418         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2419             if(p->state != RUNNABLE)
2420                 continue;
2421
2422             // Switch to chosen process. It is the process's job
2423             // to release ptable.lock and then reacquire it
2424             // before jumping back to us.
2425             proc = p;
2426             switchvm(p);
2427             p->state = RUNNING;
2428             swtch(&cpu->scheduler, proc->context);
2429             switchkvm();
2430
2431             // Process is done running for now.
2432             // It should have changed its p->state before coming back.
2433             proc = 0;
2434         }
2435         release(&ptable.lock);
2436     }
2437 }
2438 }
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449

```

```

2450 // Enter scheduler. Must hold only ptable.lock
2451 // and have changed proc->state.
2452 void
2453 sched(void)
2454 {
2455     int intena;
2456
2457     if(!holding(&ptable.lock))
2458         panic("sched ptable.lock");
2459     if(cpu->ncli != 1)
2460         panic("sched locks");
2461     if(proc->state == RUNNING)
2462         panic("sched running");
2463     if(readeflags() & FL_IF)
2464         panic("sched interruptible");
2465     intena = cpu->intena;
2466     swtch(&proc->context, cpu->scheduler);
2467     cpu->intena = intena;
2468 }
2469
2470 // Give up the CPU for one scheduling round.
2471 void
2472 yield(void)
2473 {
2474     acquire(&ptable.lock);
2475     proc->state = RUNNABLE;
2476     sched();
2477     release(&ptable.lock);
2478 }
2479
2480 // A fork child's very first scheduling by scheduler()
2481 // will swtch here. "Return" to user space.
2482 void
2483 forkret(void)
2484 {
2485     static int first = 1;
2486     // Still holding ptable.lock from scheduler.
2487     release(&ptable.lock);
2488
2489     if (first) {
2490         // Some initialization functions must be run in the context
2491         // of a regular process (e.g., they call sleep), and thus cannot
2492         // be run from main().
2493         first = 0;
2494         initlog();
2495     }
2496
2497     // Return to "caller", actually trapret (see allocproc).
2498 }
2499

```



```

2500 // Atomically release lock and sleep on chan.
2501 // Reacquires lock when awakened.
2502 void
2503 sleep(void *chan, struct spinlock *lk)
2504 {
2505     if(proc == 0)
2506         panic("sleep");
2507
2508     if(lk == 0)
2509         panic("sleep without lk");
2510
2511     // Must acquire ptable.lock in order to
2512     // change p->state and then call sched.
2513     // Once we hold ptable.lock, we can be
2514     // guaranteed that we won't miss any wakeup
2515     // (wakeup runs with ptable.lock locked),
2516     // so it's okay to release lk.
2517     if(lk != &ptable.lock){
2518         acquire(&ptable.lock);
2519         release(lk);
2520     }
2521
2522     // Go to sleep.
2523     proc->chan = chan;
2524     proc->state = SLEEPING;
2525     sched();
2526
2527     // Tidy up.
2528     proc->chan = 0;
2529
2530     // Reacquire original lock.
2531     if(lk != &ptable.lock){
2532         release(&ptable.lock);
2533         acquire(lk);
2534     }
2535 }
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549

```

```

2550 // Wake up all processes sleeping on chan.
2551 // The ptable lock must be held.
2552 static void
2553 wakeup1(void *chan)
2554 {
2555     struct proc *p;
2556
2557     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2558         if(p->state == SLEEPING && p->chan == chan)
2559             p->state = RUNNABLE;
2560     }
2561
2562     // Wake up all processes sleeping on chan.
2563     void
2564     wakeup(void *chan)
2565     {
2566         acquire(&ptable.lock);
2567         wakeup1(chan);
2568         release(&ptable.lock);
2569     }
2570
2571     // Kill the process with the given pid.
2572     // Process won't exit until it returns
2573     // to user space (see trap in trap.c).
2574     int
2575     kill(int pid)
2576     {
2577         struct proc *p;
2578
2579         acquire(&ptable.lock);
2580         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2581             if(p->pid == pid){
2582                 p->killed = 1;
2583                 // Wake process from sleep if necessary.
2584                 if(p->state == SLEEPING)
2585                     p->state = RUNNABLE;
2586                 release(&ptable.lock);
2587                 return 0;
2588             }
2589         }
2590         release(&ptable.lock);
2591         return -1;
2592     }
2593
2594
2595
2596
2597
2598
2599

```

```

2600 // Print a process listing to console. For debugging.
2601 // Runs when user types ^P on console.
2602 // No lock to avoid wedging a stuck machine further.
2603 void
2604 procdump(void)
2605 {
2606     static char *states[] = {
2607         [UNUSED]    "unused",
2608         [EMBRYO]    "embryo",
2609         [SLEEPING]  "sleep ",
2610         [RUNNABLE]  "runble",
2611         [RUNNING]   "run   ",
2612         [ZOMBIE]    "zombie"
2613     };
2614     int i;
2615     struct proc *p;
2616     char *state;
2617     uint pc[10];
2618
2619     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2620         if(p->state == UNUSED)
2621             continue;
2622         if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
2623             state = states[p->state];
2624         else
2625             state = "???";
2626         cprintf("%d %s %s", p->pid, state, p->name);
2627         if(p->state == SLEEPING){
2628             getcallerpcs((uint*)p->context->ebp+2, pc);
2629             for(i=0; i<10 && pc[i] != 0; i++)
2630                 cprintf(" %p", pc[i]);
2631         }
2632         cprintf("\n");
2633     }
2634 }
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649

```

```

2650 # Context switch
2651 #
2652 # void swtch(struct context **old, struct context *new);
2653 #
2654 # Save current register context in old
2655 # and then load register context from new.
2656
2657 .globl swtch
2658 swtch:
2659     movl 4(%esp), %eax
2660     movl 8(%esp), %edx
2661
2662     # Save old callee-save registers
2663     pushl %ebp
2664     pushl %ebx
2665     pushl %esi
2666     pushl %edi
2667
2668     # Switch stacks
2669     movl %esp, (%eax)
2670     movl %edx, %esp
2671
2672     # Load new callee-save registers
2673     popl %edi
2674     popl %esi
2675     popl %ebx
2676     popl %ebp
2677     ret
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699

```

```

2700 // Physical memory allocator, intended to allocate
2701 // memory for user processes, kernel stacks, page table pages,
2702 // and pipe buffers. Allocates 4096-byte pages.
2703
2704 #include "types.h"
2705 #include "defs.h"
2706 #include "param.h"
2707 #include "memlayout.h"
2708 #include "mmu.h"
2709 #include "spinlock.h"
2710
2711 struct run {
2712   struct run *next;
2713 };
2714
2715 struct {
2716   struct spinlock lock;
2717   struct run *freelist;
2718 } kmem;
2719
2720 extern char end[]; // first address after kernel loaded from ELF file
2721 static char *newend;
2722
2723 // A simple page allocator to get off the ground during entry
2724 char *
2725 enter_alloc(void)
2726 {
2727   if (newend == 0)
2728     newend = end;
2729
2730   if ((uint) newend >= KERNBASE + 0x400000)
2731     panic("only first 4Mbyte are mapped during entry");
2732   void *p = (void*)PGROUNDUP((uint)newend);
2733   memset(p, 0, PGSIZE);
2734   newend = newend + PGSIZE;
2735   return p;
2736 }
2737
2738 // Initialize free list of physical pages.
2739 void
2740 kinit(void)
2741 {
2742   char *p;
2743
2744   initlock(&kmem.lock, "kmem");
2745   p = (char*)PGROUNDUP((uint)newend);
2746   for(; p + PGSIZE <= (char*)p2v(PHYSTOP); p += PGSIZE)
2747     kfree(p);
2748 }
2749

```

```

2750
2751 // Free the page of physical memory pointed at by v,
2752 // which normally should have been returned by a
2753 // call to kalloc(). (The exception is when
2754 // initializing the allocator; see kinit above.)
2755 void
2756 kfree(char *v)
2757 {
2758   struct run *r;
2759
2760   if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
2761     panic("kfree");
2762
2763   // Fill with junk to catch dangling refs.
2764   memset(v, 1, PGSIZE);
2765
2766   acquire(&kmem.lock);
2767   r = (struct run*)v;
2768   r->next = kmem.freelist;
2769   kmem.freelist = r;
2770   release(&kmem.lock);
2771 }
2772
2773 // Allocate one 4096-byte page of physical memory.
2774 // Returns a pointer that the kernel can use.
2775 // Returns 0 if the memory cannot be allocated.
2776 char*
2777 kalloc(void)
2778 {
2779   struct run *r;
2780
2781   acquire(&kmem.lock);
2782   r = kmem.freelist;
2783   if(r)
2784     kmem.freelist = r->next;
2785   release(&kmem.lock);
2786   return (char*)r;
2787 }
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799

```

```

2800 // x86 trap and interrupt constants.
2801
2802 // Processor-defined:
2803 #define T_DIVIDE      0    // divide error
2804 #define T_DEBUG      1    // debug exception
2805 #define T_NMI        2    // non-maskable interrupt
2806 #define T_BRKPT      3    // breakpoint
2807 #define T_OFLOW      4    // overflow
2808 #define T_BOUND      5    // bounds check
2809 #define T_ILLOP      6    // illegal opcode
2810 #define T_DEVICE     7    // device not available
2811 #define T_DBLFLT     8    // double fault
2812 // #define T_COPROC   9    // reserved (not used since 486)
2813 #define T_TSS       10    // invalid task switch segment
2814 #define T_SEGNP     11    // segment not present
2815 #define T_STACK     12    // stack exception
2816 #define T_GPFLT     13    // general protection fault
2817 #define T_PGFLT     14    // page fault
2818 // #define T_RES      15    // reserved
2819 #define T_FPERR     16    // floating point error
2820 #define T_ALIGN     17    // alignment check
2821 #define T_MCHK      18    // machine check
2822 #define T_SIMDERR   19    // SIMD floating point error
2823
2824 // These are arbitrarily chosen, but with care not to overlap
2825 // processor defined exceptions or interrupt vectors.
2826 #define T_SYSCALL    64    // system call
2827 #define T_DEFAULT    500   // catchall
2828
2829 #define T_IRQ0       32    // IRQ 0 corresponds to int T_IRQ
2830
2831 #define IRQ_TIMER    0
2832 #define IRQ_KBD     1
2833 #define IRQ_COM1    4
2834 #define IRQ_IDE     14
2835 #define IRQ_ERROR   19
2836 #define IRQ_SPURIOUS 31
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849

```

```

2850 #!/usr/bin/perl -w
2851
2852 # Generate vectors.S, the trap/interrupt entry points.
2853 # There has to be one entry point per interrupt number
2854 # since otherwise there's no way for trap() to discover
2855 # the interrupt number.
2856
2857 print "# generated by vectors.pl - do not edit\n";
2858 print "# handlers\n";
2859 print ".globl alltraps\n";
2860 for(my $i = 0; $i < 256; $i++){
2861     print ".globl vector$i\n";
2862     print "vector$i:\n";
2863     if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
2864         print "    pushl \\\$0\n";
2865     }
2866     print "    pushl \\\$i\n";
2867     print "    jmp alltraps\n";
2868 }
2869
2870 print "\n# vector table\n";
2871 print ".data\n";
2872 print ".globl vectors\n";
2873 print "vectors:\n";
2874 for(my $i = 0; $i < 256; $i++){
2875     print "    .long vector$i\n";
2876 }
2877
2878 # sample output:
2879 # # handlers
2880 # .globl alltraps
2881 # .globl vector0
2882 # vector0:
2883 #     pushl $0
2884 #     pushl $0
2885 #     jmp alltraps
2886 # ...
2887 #
2888 # # vector table
2889 # .data
2890 # .globl vectors
2891 # vectors:
2892 #     .long vector0
2893 #     .long vector1
2894 #     .long vector2
2895 # ...
2896
2897
2898
2899

```

```

2900 #include "mmu.h"
2901
2902 # vectors.S sends all traps here.
2903 .globl alltraps
2904 alltraps:
2905 # Build trap frame.
2906 pushl %ds
2907 pushl %es
2908 pushl %fs
2909 pushl %gs
2910 pushal
2911
2912 # Set up data and per-cpu segments.
2913 movw $(SEG_KDATA<<3), %ax
2914 movw %ax, %ds
2915 movw %ax, %es
2916 movw $(SEG_KCPU<<3), %ax
2917 movw %ax, %fs
2918 movw %ax, %gs
2919
2920 # Call trap(tf), where tf=%esp
2921 pushl %esp
2922 call trap
2923 addl $4, %esp
2924
2925 # Return falls through to trapret...
2926 .globl trapret
2927 trapret:
2928 popal
2929 popl %gs
2930 popl %fs
2931 popl %es
2932 popl %ds
2933 addl $0x8, %esp # trapno and errcode
2934 iret
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949

```

```

2950 #include "types.h"
2951 #include "defs.h"
2952 #include "param.h"
2953 #include "memlayout.h"
2954 #include "mmu.h"
2955 #include "proc.h"
2956 #include "x86.h"
2957 #include "traps.h"
2958 #include "spinlock.h"
2959
2960 // Interrupt descriptor table (shared by all CPUs).
2961 struct gatedesc idt[256];
2962 extern uint vectors[]; // in vectors.S: array of 256 entry pointers
2963 struct spinlock tickslock;
2964 uint ticks;
2965
2966 void
2967 tvinit(void)
2968 {
2969     int i;
2970
2971     for(i = 0; i < 256; i++)
2972         SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
2973     SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
2974
2975     initlock(&tickslock, "time");
2976 }
2977
2978 void
2979 idtinit(void)
2980 {
2981     lidt(idt, sizeof(idt));
2982 }
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999

```

```

3000 void
3001 trap(struct trapframe *tf)
3002 {
3003     if(tf->trapno == T_SYSCALL){
3004         if(proc->killed)
3005             exit();
3006         proc->tf = tf;
3007         syscall();
3008         if(proc->killed)
3009             exit();
3010         return;
3011     }
3012     switch(tf->trapno){
3013     case T_IRQ0 + IRQ_TIMER:
3014         if(cpu->id == 0){
3015             acquire(&tickslock);
3016             ticks++;
3017             wakeup(&ticks);
3018             release(&tickslock);
3019         }
3020         lapiceoi();
3021         break;
3022     case T_IRQ0 + IRQ_IDE:
3023         ideintr();
3024         lapiceoi();
3025         break;
3026     case T_IRQ0 + IRQ_IDE+1:
3027         // Bochs generates spurious IDE1 interrupts.
3028         break;
3029     case T_IRQ0 + IRQ_KBD:
3030         kbdintr();
3031         lapiceoi();
3032         break;
3033     case T_IRQ0 + IRQ_COM1:
3034         uartintr();
3035         lapiceoi();
3036         break;
3037     case T_IRQ0 + 7:
3038     case T_IRQ0 + IRQ_SPURIOUS:
3039         cprintf("cpu%d: spurious interrupt at %x:%x\n",
3040             cpu->id, tf->cs, tf->eip);
3041         lapiceoi();
3042         break;
3043     }
3044 }
3045
3046
3047
3048
3049

```

```

3050     default:
3051         if(proc == 0 || (tf->cs&3) == 0){
3052             // In kernel, it must be our mistake.
3053             cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3054                 tf->trapno, cpu->id, tf->eip, rcr2());
3055             panic("trap");
3056         }
3057         // In user space, assume process misbehaved.
3058         cprintf("pid %d %s: trap %d err %d on cpu %d "
3059             "eip 0x%x addr 0x%x--kill proc\n",
3060             proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
3061             rcr2());
3062         proc->killed = 1;
3063     }
3064
3065     // Force process exit if it has been killed and is in user space.
3066     // (If it is still executing in the kernel, let it keep running
3067     // until it gets to the regular system call return.)
3068     if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3069         exit();
3070
3071     // Force process to give up CPU on clock tick.
3072     // If interrupts were on while locks held, would need to check nlock.
3073     if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3074         yield();
3075
3076     // Check if the process has been killed since we yielded
3077     if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3078         exit();
3079 }
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099

```

```

3100 // System call numbers
3101 #define SYS_fork 1
3102 #define SYS_exit 2
3103 #define SYS_wait 3
3104 #define SYS_pipe 4
3105 #define SYS_read 5
3106 #define SYS_kill 6
3107 #define SYS_exec 7
3108 #define SYS_fstat 8
3109 #define SYS_chdir 9
3110 #define SYS_dup 10
3111 #define SYS_getpid 11
3112 #define SYS_sbrk 12
3113 #define SYS_sleep 13
3114 #define SYS_uptime 14
3115
3116 #define SYS_open 15
3117 #define SYS_write 16
3118 #define SYS_mknod 17
3119 #define SYS_unlink 18
3120 #define SYS_link 19
3121 #define SYS_mkdir 20
3122 #define SYS_close 21
3123
3124
3125
3126
3127
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149

```

```

3150 #include "types.h"
3151 #include "defs.h"
3152 #include "param.h"
3153 #include "memlayout.h"
3154 #include "mmu.h"
3155 #include "proc.h"
3156 #include "x86.h"
3157 #include "syscall.h"
3158
3159 // User code makes a system call with INT T_SYSCALL.
3160 // System call number in %eax.
3161 // Arguments on the stack, from the user call to the C
3162 // library system call function. The saved user %esp points
3163 // to a saved program counter, and then the first argument.
3164
3165 // Fetch the int at addr from process p.
3166 int
3167 fetchint(struct proc *p, uint addr, int *ip)
3168 {
3169     if(addr >= p->sz || addr+4 > p->sz)
3170         return -1;
3171     *ip = *(int*)(addr);
3172     return 0;
3173 }
3174
3175 // Fetch the nul-terminated string at addr from process p.
3176 // Doesn't actually copy the string - just sets *pp to point at it.
3177 // Returns length of string, not including nul.
3178 int
3179 fetchstr(struct proc *p, uint addr, char **pp)
3180 {
3181     char *s, *ep;
3182
3183     if(addr >= p->sz)
3184         return -1;
3185     *pp = (char*)addr;
3186     ep = (char*)p->sz;
3187     for(s = *pp; s < ep; s++)
3188         if(*s == 0)
3189             return s - *pp;
3190     return -1;
3191 }
3192
3193 // Fetch the nth 32-bit system call argument.
3194 int
3195 argint(int n, int *ip)
3196 {
3197     return fetchint(proc, proc->tf->esp + 4 + 4*n, ip);
3198 }
3199

```

```

3200 // Fetch the nth word-sized system call argument as a pointer
3201 // to a block of memory of size n bytes. Check that the pointer
3202 // lies within the process address space.
3203 int
3204 argptr(int n, char **pp, int size)
3205 {
3206     int i;
3207
3208     if(argint(n, &i) < 0)
3209         return -1;
3210     if((uint)i >= proc->sz || (uint)i+size > proc->sz)
3211         return -1;
3212     *pp = (char*)i;
3213     return 0;
3214 }
3215
3216 // Fetch the nth word-sized system call argument as a string pointer.
3217 // Check that the pointer is valid and the string is nul-terminated.
3218 // (There is no shared writable memory, so the string can't change
3219 // between this check and being used by the kernel.)
3220 int
3221 argstr(int n, char **pp)
3222 {
3223     int addr;
3224     if(argint(n, &addr) < 0)
3225         return -1;
3226     return fetchstr(proc, addr, pp);
3227 }
3228
3229 extern int sys_chdir(void);
3230 extern int sys_close(void);
3231 extern int sys_dup(void);
3232 extern int sys_exec(void);
3233 extern int sys_exit(void);
3234 extern int sys_fork(void);
3235 extern int sys_fstat(void);
3236 extern int sys_getpid(void);
3237 extern int sys_kill(void);
3238 extern int sys_link(void);
3239 extern int sys_mkdir(void);
3240 extern int sys_mknod(void);
3241 extern int sys_open(void);
3242 extern int sys_pipe(void);
3243 extern int sys_read(void);
3244 extern int sys_sbrk(void);
3245 extern int sys_sleep(void);
3246 extern int sys_unlink(void);
3247 extern int sys_wait(void);
3248 extern int sys_write(void);
3249 extern int sys_uptime(void);

```

```

3250 static int (*syscalls[])(void) = {
3251     [SYS_fork]     sys_fork,
3252     [SYS_exit]    sys_exit,
3253     [SYS_wait]    sys_wait,
3254     [SYS_pipe]    sys_pipe,
3255     [SYS_read]    sys_read,
3256     [SYS_kill]    sys_kill,
3257     [SYS_exec]    sys_exec,
3258     [SYS_fstat]   sys_fstat,
3259     [SYS_chdir]   sys_chdir,
3260     [SYS_dup]     sys_dup,
3261     [SYS_getpid]  sys_getpid,
3262     [SYS_sbrk]    sys_sbrk,
3263     [SYS_sleep]   sys_sleep,
3264     [SYS_uptime]  sys_uptime,
3265     [SYS_open]    sys_open,
3266     [SYS_write]   sys_write,
3267     [SYS_mknod]   sys_mknod,
3268     [SYS_unlink]  sys_unlink,
3269     [SYS_link]    sys_link,
3270     [SYS_mkdir]   sys_mkdir,
3271     [SYS_close]   sys_close,
3272 };
3273
3274 void
3275 syscall(void)
3276 {
3277     int num;
3278
3279     num = proc->tf->eax;
3280     if(num >= 0 && num < SYS_open && syscalls[num]) {
3281         proc->tf->eax = syscalls[num]();
3282     } else if (num >= SYS_open && num < NELEM(syscalls) && syscalls[num]) {
3283         proc->tf->eax = syscalls[num]();
3284     } else {
3285         cprintf("%d %s: unknown sys call %d\n",
3286             proc->pid, proc->name, num);
3287         proc->tf->eax = -1;
3288     }
3289 }
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299

```



```

3300 #include "types.h"
3301 #include "x86.h"
3302 #include "defs.h"
3303 #include "param.h"
3304 #include "memlayout.h"
3305 #include "mmu.h"
3306 #include "proc.h"
3307
3308 int
3309 sys_fork(void)
3310 {
3311     return fork();
3312 }
3313
3314 int
3315 sys_exit(void)
3316 {
3317     exit();
3318     return 0; // not reached
3319 }
3320
3321 int
3322 sys_wait(void)
3323 {
3324     return wait();
3325 }
3326
3327 int
3328 sys_kill(void)
3329 {
3330     int pid;
3331
3332     if(argint(0, &pid) < 0)
3333         return -1;
3334     return kill(pid);
3335 }
3336
3337 int
3338 sys_getpid(void)
3339 {
3340     return proc->pid;
3341 }
3342
3343
3344
3345
3346
3347
3348
3349

```

```

3350 int
3351 sys_sbrk(void)
3352 {
3353     int addr;
3354     int n;
3355
3356     if(argint(0, &n) < 0)
3357         return -1;
3358     addr = proc->sz;
3359     if(growproc(n) < 0)
3360         return -1;
3361     return addr;
3362 }
3363
3364 int
3365 sys_sleep(void)
3366 {
3367     int n;
3368     uint ticks0;
3369
3370     if(argint(0, &n) < 0)
3371         return -1;
3372     acquire(&tickslock);
3373     ticks0 = ticks;
3374     while(ticks - ticks0 < n){
3375         if(proc->killed){
3376             release(&tickslock);
3377             return -1;
3378         }
3379         sleep(&ticks, &tickslock);
3380     }
3381     release(&tickslock);
3382     return 0;
3383 }
3384
3385 // return how many clock tick interrupts have occurred
3386 // since start.
3387 int
3388 sys_uptime(void)
3389 {
3390     uint xticks;
3391
3392     acquire(&tickslock);
3393     xticks = ticks;
3394     release(&tickslock);
3395     return xticks;
3396 }
3397
3398
3399

```

```
3400 struct buf {
3401     int flags;
3402     uint dev;
3403     uint sector;
3404     struct buf *prev; // LRU cache list
3405     struct buf *next;
3406     struct buf *qnext; // disk queue
3407     uchar data[512];
3408 };
3409 #define B_BUSY 0x1 // buffer is locked by some process
3410 #define B_VALID 0x2 // buffer has been read from disk
3411 #define B_DIRTY 0x4 // buffer needs to be written to disk
3412
3413
3414
3415
3416
3417
3418
3419
3420
3421
3422
3423
3424
3425
3426
3427
3428
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
```

```
3450 #define O_RDONLY 0x000
3451 #define O_WRONLY 0x001
3452 #define O_RDWR 0x002
3453 #define O_CREATE 0x200
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 #define T_DIR 1 // Directory
3501 #define T_FILE 2 // File
3502 #define T_DEV 3 // Special device
3503
3504 struct stat {
3505     short type; // Type of file
3506     int dev; // Device number
3507     uint ino; // Inode number on device
3508     short nlink; // Number of links to file
3509     uint size; // Size of file in bytes
3510 };
3511
3512
3513
3514
3515
3516
3517
3518
3519
3520
3521
3522
3523
3524
3525
3526
3527
3528
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549

```

```

3550 // On-disk file system format.
3551 // Both the kernel and user programs use this header file.
3552
3553 // Block 0 is unused. Block 1 is super block.
3554 // Inodes start at block 2.
3555
3556 #define ROOTINO 1 // root i-number
3557 #define BSIZE 512 // block size
3558
3559 // File system super block
3560 struct superblock {
3561     uint size; // Size of file system image (blocks)
3562     uint nblocks; // Number of data blocks
3563     uint ninodes; // Number of inodes.
3564     uint nlog; // Number of log blocks
3565 };
3566
3567 #define NDIRECT 12
3568 #define NINDIRECT (BSIZE / sizeof(uint))
3569 #define MAXFILE (NDIRECT + NINDIRECT)
3570
3571 // On-disk inode structure
3572 struct dinode {
3573     short type; // File type
3574     short major; // Major device number (T_DEV only)
3575     short minor; // Minor device number (T_DEV only)
3576     short nlink; // Number of links to inode in file system
3577     uint size; // Size of file (bytes)
3578     uint addrs[NDIRECT+1]; // Data block addresses
3579 };
3580
3581 // Inodes per block.
3582 #define IPB (BSIZE / sizeof(struct dinode))
3583
3584 // Block containing inode i
3585 #define IBLOCK(i) ((i) / IPB + 2)
3586
3587 // Bitmap bits per block
3588 #define BPB (BSIZE*8)
3589
3590 // Block containing bit for block b
3591 #define BBLOCK(b, ninodes) (b/BPB + (ninodes)/IPB + 3)
3592
3593 // Directory is a file containing a sequence of dirent structures.
3594 #define DIRSIZ 14
3595
3596 struct dirent {
3597     ushort inum;
3598     char name[DIRSIZ];
3599 };

```

```

3600 struct file {
3601     enum { FD_NONE, FD_PIPE, FD_INODE } type;
3602     int ref; // reference count
3603     char readable;
3604     char writable;
3605     struct pipe *pipe;
3606     struct inode *ip;
3607     uint off;
3608 };
3609
3610
3611 // in-core file system types
3612
3613 struct inode {
3614     uint dev;           // Device number
3615     uint inum;         // Inode number
3616     int ref;           // Reference count
3617     int flags;         // I_BUSY, I_INVALID
3618
3619     short type;        // copy of disk inode
3620     short major;
3621     short minor;
3622     short nlink;
3623     uint size;
3624     uint addrs[NDIRECT+1];
3625 };
3626
3627 #define I_BUSY 0x1
3628 #define I_INVALID 0x2
3629
3630 // device implementations
3631
3632 struct devsw {
3633     int (*read)(struct inode*, char*, int);
3634     int (*write)(struct inode*, char*, int);
3635 };
3636
3637 extern struct devsw devsw[];
3638
3639 #define CONSOLE 1
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649

```

```

3650 // Simple PIO-based (non-DMA) IDE driver code.
3651
3652 #include "types.h"
3653 #include "defs.h"
3654 #include "param.h"
3655 #include "memlayout.h"
3656 #include "mmu.h"
3657 #include "proc.h"
3658 #include "x86.h"
3659 #include "traps.h"
3660 #include "spinlock.h"
3661 #include "buf.h"
3662
3663 #define IDE_BSY      0x80
3664 #define IDE_DRDY    0x40
3665 #define IDE_DF      0x20
3666 #define IDE_ERR     0x01
3667
3668 #define IDE_CMD_READ 0x20
3669 #define IDE_CMD_WRITE 0x30
3670
3671 // idequeue points to the buf now being read/written to the disk.
3672 // idequeue->qnext points to the next buf to be processed.
3673 // You must hold idelock while manipulating queue.
3674
3675 static struct spinlock idelock;
3676 static struct buf *idequeue;
3677
3678 static int havedisk1;
3679 static void idestart(struct buf*);
3680
3681 // Wait for IDE disk to become ready.
3682 static int
3683 idewait(int checkerr)
3684 {
3685     int r;
3686
3687     while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
3688         ;
3689     if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
3690         return -1;
3691     return 0;
3692 }
3693
3694
3695
3696
3697
3698
3699

```

```

3700 void
3701 ideinit(void)
3702 {
3703     int i;
3704
3705     initlock(&idelock, "ide");
3706     picenable(IRQ_IDE);
3707     ioapicenable(IRQ_IDE, ncpu - 1);
3708     idewait(0);
3709
3710     // Check if disk 1 is present
3711     outb(0x1f6, 0xe0 | (1<<4));
3712     for(i=0; i<1000; i++){
3713         if(inb(0x1f7) != 0){
3714             havedisk1 = 1;
3715             break;
3716         }
3717     }
3718
3719     // Switch back to disk 0.
3720     outb(0x1f6, 0xe0 | (0<<4));
3721 }
3722
3723 // Start the request for b. Caller must hold idelock.
3724 static void
3725 idestart(struct buf *b)
3726 {
3727     if(b == 0)
3728         panic("idestart");
3729
3730     idewait(0);
3731     outb(0x3f6, 0); // generate interrupt
3732     outb(0x1f2, 1); // number of sectors
3733     outb(0x1f3, b->sector & 0xff);
3734     outb(0x1f4, (b->sector >> 8) & 0xff);
3735     outb(0x1f5, (b->sector >> 16) & 0xff);
3736     outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
3737     if(b->flags & B_DIRTY){
3738         outb(0x1f7, IDE_CMD_WRITE);
3739         outsl(0x1f0, b->data, 512/4);
3740     } else {
3741         outb(0x1f7, IDE_CMD_READ);
3742     }
3743 }
3744
3745
3746
3747
3748
3749

```

```

3750 // Interrupt handler.
3751 void
3752 ideintr(void)
3753 {
3754     struct buf *b;
3755
3756     // Take first buffer off queue.
3757     acquire(&idelock);
3758     if((b = idequeue) == 0){
3759         release(&idelock);
3760         // cprintf("spurious IDE interrupt\n");
3761         return;
3762     }
3763     idequeue = b->qnext;
3764
3765     // Read data if needed.
3766     if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
3767         insl(0x1f0, b->data, 512/4);
3768
3769     // Wake process waiting for this buf.
3770     b->flags |= B_VALID;
3771     b->flags &= ~B_DIRTY;
3772     wakeup(b);
3773
3774     // Start disk on next buf in queue.
3775     if(idequeue != 0)
3776         idestart(idequeue);
3777
3778     release(&idelock);
3779 }
3780
3781
3782
3783
3784
3785
3786
3787
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3799

```

```

3800 // Sync buf with disk.
3801 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
3802 // Else if B_VALID is not set, read buf from disk, set B_VALID.
3803 void
3804 iderw(struct buf *b)
3805 {
3806     struct buf **pp;
3807
3808     if(!(b->flags & B_BUSY))
3809         panic("iderw: buf not busy");
3810     if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
3811         panic("iderw: nothing to do");
3812     if(b->dev != 0 && !havedisk1)
3813         panic("iderw: ide disk 1 not present");
3814
3815     acquire(&idelock); // DOC:acquire-lock
3816
3817     // Append b to idequeue.
3818     b->qnext = 0;
3819     for(pp=&idequeue; *pp; pp=&(*pp)->qnext) // DOC:insert-queue
3820         ;
3821     *pp = b;
3822
3823     // Start disk if necessary.
3824     if(idequeue == b)
3825         idestart(b);
3826
3827     // Wait for request to finish.
3828     // Assuming will not sleep too long: ignore proc->killed.
3829     while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
3830         sleep(b, &idelock);
3831     }
3832
3833     release(&idelock);
3834 }
3835
3836
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849

```

```

3850 // Buffer cache.
3851 //
3852 // The buffer cache is a linked list of buf structures holding
3853 // cached copies of disk block contents. Caching disk blocks
3854 // in memory reduces the number of disk reads and also provides
3855 // a synchronization point for disk blocks used by multiple processes.
3856 //
3857 // Interface:
3858 // * To get a buffer for a particular disk block, call bread.
3859 // * After changing buffer data, call bwrite to flush it to disk.
3860 // * When done with the buffer, call brelse.
3861 // * Do not use the buffer after calling brelse.
3862 // * Only one process at a time can use a buffer,
3863 //   so do not keep them longer than necessary.
3864 //
3865 // The implementation uses three state flags internally:
3866 // * B_BUSY: the block has been returned from bread
3867 //   and has not been passed back to brelse.
3868 // * B_VALID: the buffer data has been initialized
3869 //   with the associated disk block contents.
3870 // * B_DIRTY: the buffer data has been modified
3871 //   and needs to be written to disk.
3872
3873 #include "types.h"
3874 #include "defs.h"
3875 #include "param.h"
3876 #include "spinlock.h"
3877 #include "buf.h"
3878
3879 struct {
3880     struct spinlock lock;
3881     struct buf buf[NBUF];
3882
3883     // Linked list of all buffers, through prev/next.
3884     // head.next is most recently used.
3885     struct buf head;
3886 } bcache;
3887
3888 void
3889 binit(void)
3890 {
3891     struct buf *b;
3892
3893     initlock(&bcache.lock, "bcache");
3894
3895
3896
3897
3898
3899

```

```

3900 // Create linked list of buffers
3901 bcache.head.prev = &bcache.head;
3902 bcache.head.next = &bcache.head;
3903 for(b = bcache.buf; b < bcache.buf+NBUF; b++){
3904     b->next = bcache.head.next;
3905     b->prev = &bcache.head;
3906     b->dev = -1;
3907     bcache.head.next->prev = b;
3908     bcache.head.next = b;
3909 }
3910 }
3911
3912 // Look through buffer cache for sector on device dev.
3913 // If not found, allocate fresh block.
3914 // In either case, return locked buffer.
3915 static struct buf*
3916 bget(uint dev, uint sector)
3917 {
3918     struct buf *b;
3919
3920     acquire(&bcache.lock);
3921
3922     loop:
3923     // Try for cached block.
3924     for(b = bcache.head.next; b != &bcache.head; b = b->next){
3925         if(b->dev == dev && b->sector == sector){
3926             if(!(b->flags & B_BUSY)){
3927                 b->flags |= B_BUSY;
3928                 release(&bcache.lock);
3929                 return b;
3930             }
3931             sleep(b, &bcache.lock);
3932             goto loop;
3933         }
3934     }
3935
3936     // Allocate fresh block.
3937     for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
3938         if((b->flags & B_BUSY) == 0){
3939             b->dev = dev;
3940             b->sector = sector;
3941             b->flags = B_BUSY;
3942             release(&bcache.lock);
3943             return b;
3944         }
3945     }
3946     panic("bget: no buffers");
3947 }
3948
3949

```

```

3950 // Return a B_BUSY buf with the contents of the indicated disk sector.
3951 struct buf*
3952 bread(uint dev, uint sector)
3953 {
3954     struct buf *b;
3955
3956     b = bget(dev, sector);
3957     if(!(b->flags & B_VALID))
3958         iderw(b);
3959     return b;
3960 }
3961
3962 // Write b's contents to disk. Must be locked.
3963 void
3964 bwrite(struct buf *b)
3965 {
3966     if((b->flags & B_BUSY) == 0)
3967         panic("bwrite");
3968     b->flags |= B_DIRTY;
3969     iderw(b);
3970 }
3971
3972 // Release the buffer b.
3973 void
3974 brelse(struct buf *b)
3975 {
3976     if((b->flags & B_BUSY) == 0)
3977         panic("brelse");
3978
3979     acquire(&bcache.lock);
3980
3981     b->next->prev = b->prev;
3982     b->prev->next = b->next;
3983     b->next = bcache.head.next;
3984     b->prev = &bcache.head;
3985     bcache.head.next->prev = b;
3986     bcache.head.next = b;
3987
3988     b->flags &= ~B_BUSY;
3989     wakeup(b);
3990
3991     release(&bcache.lock);
3992 }
3993
3994
3995
3996
3997
3998
3999

```

```

4000 #include "types.h"
4001 #include "defs.h"
4002 #include "param.h"
4003 #include "spinlock.h"
4004 #include "fs.h"
4005 #include "buf.h"
4006
4007 // Simple logging. Each system call that might write the file system
4008 // should be surrounded with begin_trans() and commit_trans() calls.
4009 //
4010 // The log holds at most one transaction at a time. Commit forces
4011 // the log (with commit record) to disk, then installs the affected
4012 // blocks to disk, then erases the log. begin_trans() ensures that
4013 // only one system call can be in a transaction; others must wait.
4014 //
4015 // Allowing only one transaction at a time means that the file
4016 // system code doesn't have to worry about the possibility of
4017 // one transaction reading a block that another one has modified,
4018 // for example an i-node block.
4019 //
4020 // Read-only system calls don't need to use transactions, though
4021 // this means that they may observe uncommitted data. I-node and
4022 // buffer locks prevent read-only calls from seeing inconsistent data.
4023 //
4024 // The log is a physical re-do log containing disk blocks.
4025 // The on-disk log format:
4026 //   header block, containing sector #s for block A, B, C, ...
4027 //   block A
4028 //   block B
4029 //   block C
4030 //   ...
4031 // Log appends are synchronous.
4032
4033 // Contents of the header block, used for both the on-disk header block
4034 // and to keep track in memory of logged sector #s before commit.
4035 struct logheader {
4036   int n;
4037   int sector[LOGSIZE];
4038 };
4039
4040 struct log {
4041   struct spinlock lock;
4042   int start;
4043   int size;
4044   int intrans;
4045   int dev;
4046   struct logheader lh;
4047 };
4048
4049

```

```

4050 struct log log;
4051
4052 static void recover_from_log(void);
4053
4054 void
4055 initlog(void)
4056 {
4057   if (sizeof(struct logheader) >= BSIZE)
4058     panic("initlog: too big logheader");
4059
4060   struct superblock sb;
4061   initlock(&log.lock, "log");
4062   readsb(ROOTDEV, &sb);
4063   log.start = sb.size - sb.nlog;
4064   log.size = sb.nlog;
4065   log.dev = ROOTDEV;
4066   recover_from_log();
4067 }
4068
4069 // Copy committed blocks from log to their home location
4070 static void
4071 install_trans(void)
4072 {
4073   int tail;
4074
4075   for (tail = 0; tail < log.lh.n; tail++) {
4076     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
4077     struct buf *dbuf = bread(log.dev, log.lh.sector[tail]); // read dst
4078     memmove(dbuf->data, lbuf->data, BSIZE); // copy block to dst
4079     bwrite(dbuf); // flush dst to disk
4080     brelse(lbuf);
4081     brelse(dbuf);
4082   }
4083 }
4084
4085 // Read the log header from disk into the in-memory log header
4086 static void
4087 read_head(void)
4088 {
4089   struct buf *buf = bread(log.dev, log.start);
4090   struct logheader *lh = (struct logheader *) (buf->data);
4091   int i;
4092   log.lh.n = lh->n;
4093   for (i = 0; i < log.lh.n; i++) {
4094     log.lh.sector[i] = lh->sector[i];
4095   }
4096   brelse(buf);
4097 }
4098
4099

```



```

4100 // Write in-memory log header to disk, committing log entries till head
4101 static void
4102 write_head(void)
4103 {
4104     struct buf *buf = bread(log.dev, log.start);
4105     struct logheader *hb = (struct logheader *) (buf->data);
4106     int i;
4107     hb->n = log.lh.n;
4108     for (i = 0; i < log.lh.n; i++) {
4109         hb->sector[i] = log.lh.sector[i];
4110     }
4111     bwrite(buf);
4112     brelse(buf);
4113 }
4114
4115 static void
4116 recover_from_log(void)
4117 {
4118     read_head();
4119     install_trans(); // if committed, copy from log to disk
4120     log.lh.n = 0;
4121     write_head(); // clear the log
4122 }
4123
4124 void
4125 begin_trans(void)
4126 {
4127     acquire(&log.lock);
4128     while (log.intrans) {
4129         sleep(&log, &log.lock);
4130     }
4131     log.intrans = 1;
4132     release(&log.lock);
4133 }
4134
4135 void
4136 commit_trans(void)
4137 {
4138     if (log.lh.n > 0) {
4139         write_head(); // Causes all blocks till log.head to be committed
4140         install_trans(); // Install all the transactions till head
4141         log.lh.n = 0;
4142         write_head(); // Reclaim log
4143     }
4144
4145     acquire(&log.lock);
4146     log.intrans = 0;
4147     wakeup(&log);
4148     release(&log.lock);
4149 }

```

```

4150 // Caller has modified b->data and is done with the buffer.
4151 // Append the block to the log and record the block number,
4152 // but don't write the log header (which would commit the write).
4153 // log_write() replaces bwrite(); a typical use is:
4154 //   bp = bread(...)
4155 //   modify bp->data[]
4156 //   log_write(bp)
4157 //   brelse(bp)
4158 void
4159 log_write(struct buf *b)
4160 {
4161     int i;
4162
4163     if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
4164         panic("too big a transaction");
4165     if (!log.intrans)
4166         panic("write outside of trans");
4167
4168     for (i = 0; i < log.lh.n; i++) {
4169         if (log.lh.sector[i] == b->sector) // log absorbtion?
4170             break;
4171     }
4172     log.lh.sector[i] = b->sector;
4173     struct buf *lbuf = bread(b->dev, log.start+i+1);
4174     memmove(lbuf->data, b->data, BSIZE);
4175     bwrite(lbuf);
4176     brelse(lbuf);
4177     if (i == log.lh.n)
4178         log.lh.n++;
4179 }
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199

```

```

4200 // Blank page.
4201
4202
4203
4204
4205
4206
4207
4208
4209
4210
4211
4212
4213
4214
4215
4216
4217
4218
4219
4220
4221
4222
4223
4224
4225
4226
4227
4228
4229
4230
4231
4232
4233
4234
4235
4236
4237
4238
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249

```

```

4250 // File system implementation. Four layers:
4251 // + Blocks: allocator for raw disk blocks.
4252 // + Files: inode allocator, reading, writing, metadata.
4253 // + Directories: inode with special contents (list of other inodes!)
4254 // + Names: paths like /usr/rtn/xv6/fs.c for convenient naming.
4255 //
4256 // Disk layout is: superblock, inodes, block in-use bitmap, data blocks.
4257 //
4258 // This file contains the low-level file system manipulation
4259 // routines. The (higher-level) system call implementations
4260 // are in sysfile.c.
4261
4262 #include "types.h"
4263 #include "defs.h"
4264 #include "param.h"
4265 #include "stat.h"
4266 #include "mmu.h"
4267 #include "proc.h"
4268 #include "spinlock.h"
4269 #include "buf.h"
4270 #include "fs.h"
4271 #include "file.h"
4272
4273 #define min(a, b) ((a) < (b) ? (a) : (b))
4274 static void itrunc(struct inode*);
4275
4276 // Read the super block.
4277 void
4278 readsb(int dev, struct superblock *sb)
4279 {
4280     struct buf *bp;
4281
4282     bp = bread(dev, 1);
4283     memmove(sb, bp->data, sizeof(*sb));
4284     brelse(bp);
4285 }
4286
4287 // Zero a block.
4288 static void
4289 bzero(int dev, int bno)
4290 {
4291     struct buf *bp;
4292
4293     bp = bread(dev, bno);
4294     memset(bp->data, 0, BSIZE);
4295     log_write(bp);
4296     brelse(bp);
4297 }
4298
4299

```

```

4300 // Blocks.
4301
4302 // Allocate a zeroed disk block.
4303 static uint
4304 balloc(uint dev)
4305 {
4306     int b, bi, m;
4307     struct buf *bp;
4308     struct superblock sb;
4309
4310     bp = 0;
4311     readsb(dev, &sb);
4312     for(b = 0; b < sb.size; b += BPB){
4313         bp = bread(dev, BBLOCK(b, sb.ninodes));
4314         for(bi = 0; bi < BPB && bi < (sb.size - b); bi++){
4315             m = 1 << (bi % 8);
4316             if((bp->data[bi/8] & m) == 0){ // Is block free?
4317                 bp->data[bi/8] |= m; // Mark block in use on disk.
4318                 log_write(bp);
4319                 brelse(bp);
4320                 bzero(dev, b + bi);
4321                 return b + bi;
4322             }
4323         }
4324         brelse(bp);
4325     }
4326     panic("balloc: out of blocks");
4327 }
4328
4329 // Free a disk block.
4330 static void
4331 bfree(int dev, uint b)
4332 {
4333     struct buf *bp;
4334     struct superblock sb;
4335     int bi, m;
4336
4337     readsb(dev, &sb);
4338     bp = bread(dev, BBLOCK(b, sb.ninodes));
4339     bi = b % BPB;
4340     m = 1 << (bi % 8);
4341     if((bp->data[bi/8] & m) == 0)
4342         panic("freeing free block");
4343     bp->data[bi/8] &= ~m; // Mark block free on disk.
4344     log_write(bp);
4345     brelse(bp);
4346 }
4347
4348
4349

```

```

4350 // Inodes.
4351 //
4352 // An inode is a single, unnamed file in the file system.
4353 // The inode disk structure holds metadata (the type, device numbers,
4354 // and data size) along with a list of blocks where the associated
4355 // data can be found.
4356 //
4357 // The inodes are laid out sequentially on disk immediately after
4358 // the superblock. The kernel keeps a cache of the in-use
4359 // on-disk structures to provide a place for synchronizing access
4360 // to inodes shared between multiple processes.
4361 //
4362 // ip->ref counts the number of pointer references to this cached
4363 // inode; references are typically kept in struct file and in proc->cwd.
4364 // When ip->ref falls to zero, the inode is no longer cached.
4365 // It is an error to use an inode without holding a reference to it.
4366 //
4367 // Processes are only allowed to read and write inode
4368 // metadata and contents when holding the inode's lock,
4369 // represented by the I_BUSY flag in the in-memory copy.
4370 // Because inode locks are held during disk accesses,
4371 // they are implemented using a flag rather than with
4372 // spin locks. Callers are responsible for locking
4373 // inodes before passing them to routines in this file; leaving
4374 // this responsibility with the caller makes it possible for them
4375 // to create arbitrarily-sized atomic operations.
4376 //
4377 // To give maximum control over locking to the callers,
4378 // the routines in this file that return inode pointers
4379 // return pointers to *unlocked* inodes. It is the callers'
4380 // responsibility to lock them before using them. A non-zero
4381 // ip->ref keeps these unlocked inodes in the cache.
4382
4383 struct {
4384     struct spinlock lock;
4385     struct inode inode[NINODE];
4386 } icache;
4387
4388 void
4389 iinit(void)
4390 {
4391     initlock(&icache.lock, "icache");
4392 }
4393
4394 static struct inode* iget(uint dev, uint inum);
4395
4396
4397
4398
4399

```

```

4400 // Allocate a new inode with the given type on device dev.
4401 struct inode*
4402 ialloc(uint dev, short type)
4403 {
4404     int inum;
4405     struct buf *bp;
4406     struct dinode *dip;
4407     struct superblock sb;
4408
4409     readsb(dev, &sb);
4410     for(inum = 1; inum < sb.ninodes; inum++){ // loop over inode blocks
4411         bp = bread(dev, IBLOCK(inum));
4412         dip = (struct dinode*)bp->data + inum%IPB;
4413         if(dip->type == 0){ // a free inode
4414             memset(dip, 0, sizeof(*dip));
4415             dip->type = type;
4416             log_write(bp); // mark it allocated on the disk
4417             brelse(bp);
4418             return iget(dev, inum);
4419         }
4420         brelse(bp);
4421     }
4422     panic("ialloc: no inodes");
4423 }
4424
4425 // Copy inode, which has changed, from memory to disk.
4426 void
4427 iupdate(struct inode *ip)
4428 {
4429     struct buf *bp;
4430     struct dinode *dip;
4431
4432     bp = bread(ip->dev, IBLOCK(ip->inum));
4433     dip = (struct dinode*)bp->data + ip->inum%IPB;
4434     dip->type = ip->type;
4435     dip->major = ip->major;
4436     dip->minor = ip->minor;
4437     dip->nlink = ip->nlink;
4438     dip->size = ip->size;
4439     memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
4440     log_write(bp);
4441     brelse(bp);
4442 }
4443
4444
4445
4446
4447
4448
4449

```

```

4450 // Find the inode with number inum on device dev
4451 // and return the in-memory copy.
4452 static struct inode*
4453 iget(uint dev, uint inum)
4454 {
4455     struct inode *ip, *empty;
4456
4457     acquire(&icache.lock);
4458
4459     // Try for cached inode.
4460     empty = 0;
4461     for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
4462         if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
4463             ip->ref++;
4464             release(&icache.lock);
4465             return ip;
4466         }
4467         if(empty == 0 && ip->ref == 0) // Remember empty slot.
4468             empty = ip;
4469     }
4470
4471     // Allocate fresh inode.
4472     if(empty == 0)
4473         panic("iget: no inodes");
4474
4475     ip = empty;
4476     ip->dev = dev;
4477     ip->inum = inum;
4478     ip->ref = 1;
4479     ip->flags = 0;
4480     release(&icache.lock);
4481
4482     return ip;
4483 }
4484
4485 // Increment reference count for ip.
4486 // Returns ip to enable ip = idup(ip1) idiom.
4487 struct inode*
4488 idup(struct inode *ip)
4489 {
4490     acquire(&icache.lock);
4491     ip->ref++;
4492     release(&icache.lock);
4493     return ip;
4494 }
4495
4496
4497
4498
4499

```

```

4500 // Lock the given inode.
4501 void
4502 ilock(struct inode *ip)
4503 {
4504     struct buf *bp;
4505     struct dinode *dip;
4506
4507     if(ip == 0 || ip->ref < 1)
4508         panic("ilock");
4509
4510     acquire(&icache.lock);
4511     while(ip->flags & I_BUSY)
4512         sleep(ip, &icache.lock);
4513     ip->flags |= I_BUSY;
4514     release(&icache.lock);
4515
4516     if(!(ip->flags & I_VALID)){
4517         bp = bread(ip->dev, IBLOCK(ip->inum));
4518         dip = (struct dinode*)bp->data + ip->inum%IPB;
4519         ip->type = dip->type;
4520         ip->major = dip->major;
4521         ip->minor = dip->minor;
4522         ip->nlink = dip->nlink;
4523         ip->size = dip->size;
4524         memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
4525         brelse(bp);
4526         ip->flags |= I_VALID;
4527         if(ip->type == 0)
4528             panic("ilock: no type");
4529     }
4530 }
4531
4532 // Unlock the given inode.
4533 void
4534 iunlock(struct inode *ip)
4535 {
4536     if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
4537         panic("iunlock");
4538
4539     acquire(&icache.lock);
4540     ip->flags &= ~I_BUSY;
4541     wakeup(ip);
4542     release(&icache.lock);
4543 }
4544
4545
4546
4547
4548
4549

```

```

4550 // Caller holds reference to unlocked ip. Drop reference.
4551 void
4552 iput(struct inode *ip)
4553 {
4554     acquire(&icache.lock);
4555     if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
4556         // inode is no longer used: truncate and free inode.
4557         if(ip->flags & I_BUSY)
4558             panic("iput busy");
4559         ip->flags |= I_BUSY;
4560         release(&icache.lock);
4561         itrunc(ip);
4562         ip->type = 0;
4563         iupdate(ip);
4564         acquire(&icache.lock);
4565         ip->flags = 0;
4566         wakeup(ip);
4567     }
4568     ip->ref--;
4569     release(&icache.lock);
4570 }
4571
4572 // Common idiom: unlock, then put.
4573 void
4574 iunlockput(struct inode *ip)
4575 {
4576     iunlock(ip);
4577     iput(ip);
4578 }
4579
4580
4581
4582
4583
4584
4585
4586
4587
4588
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4599

```

```

4600 // Inode contents
4601 //
4602 // The contents (data) associated with each inode is stored
4603 // in a sequence of blocks on the disk. The first NDIRECT blocks
4604 // are listed in ip->addrs[]. The next NINDIRECT blocks are
4605 // listed in the block ip->addrs[NDIRECT].
4606
4607 // Return the disk block address of the nth block in inode ip.
4608 // If there is no such block, bmap allocates one.
4609 static uint
4610 bmap(struct inode *ip, uint bn)
4611 {
4612     uint addr, *a;
4613     struct buf *bp;
4614
4615     if(bn < NDIRECT){
4616         if((addr = ip->addrs[bn]) == 0)
4617             ip->addrs[bn] = addr = balloc(ip->dev);
4618         return addr;
4619     }
4620     bn -= NDIRECT;
4621
4622     if(bn < NINDIRECT){
4623         // Load indirect block, allocating if necessary.
4624         if((addr = ip->addrs[NDIRECT]) == 0)
4625             ip->addrs[NDIRECT] = addr = balloc(ip->dev);
4626         bp = bread(ip->dev, addr);
4627         a = (uint*)bp->data;
4628         if((addr = a[bn]) == 0){
4629             a[bn] = addr = balloc(ip->dev);
4630             log_write(bp);
4631         }
4632         brelse(bp);
4633         return addr;
4634     }
4635
4636     panic("bmap: out of range");
4637 }
4638
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649

```

```

4650 // Truncate inode (discard contents).
4651 // Only called after the last dirent referring
4652 // to this inode has been erased on disk.
4653 static void
4654 itrunc(struct inode *ip)
4655 {
4656     int i, j;
4657     struct buf *bp;
4658     uint *a;
4659
4660     for(i = 0; i < NDIRECT; i++){
4661         if(ip->addrs[i]){
4662             bfree(ip->dev, ip->addrs[i]);
4663             ip->addrs[i] = 0;
4664         }
4665     }
4666
4667     if(ip->addrs[NDIRECT]){
4668         bp = bread(ip->dev, ip->addrs[NDIRECT]);
4669         a = (uint*)bp->data;
4670         for(j = 0; j < NINDIRECT; j++){
4671             if(a[j])
4672                 bfree(ip->dev, a[j]);
4673         }
4674         brelse(bp);
4675         bfree(ip->dev, ip->addrs[NDIRECT]);
4676         ip->addrs[NDIRECT] = 0;
4677     }
4678
4679     ip->size = 0;
4680     iupdate(ip);
4681 }
4682
4683 // Copy stat information from inode.
4684 void
4685 stati(struct inode *ip, struct stat *st)
4686 {
4687     st->dev = ip->dev;
4688     st->ino = ip->inum;
4689     st->type = ip->type;
4690     st->nlink = ip->nlink;
4691     st->size = ip->size;
4692 }
4693
4694
4695
4696
4697
4698
4699

```

```

4700 // Read data from inode.
4701 int
4702 readi(struct inode *ip, char *dst, uint off, uint n)
4703 {
4704     uint tot, m;
4705     struct buf *bp;
4706
4707     if(ip->type == T_DEV){
4708         if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
4709             return -1;
4710         return devsw[ip->major].read(ip, dst, n);
4711     }
4712
4713     if(off > ip->size || off + n < off)
4714         return -1;
4715     if(off + n > ip->size)
4716         n = ip->size - off;
4717
4718     for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
4719         bp = bread(ip->dev, bmap(ip, off/BSIZE));
4720         m = min(n - tot, BSIZE - off%BSIZE);
4721         memmove(dst, bp->data + off%BSIZE, m);
4722         brelse(bp);
4723     }
4724     return n;
4725 }
4726
4727
4728
4729
4730
4731
4732
4733
4734
4735
4736
4737
4738
4739
4740
4741
4742
4743
4744
4745
4746
4747
4748
4749

```

```

4750 // Write data to inode.
4751 int
4752 writei(struct inode *ip, char *src, uint off, uint n)
4753 {
4754     uint tot, m;
4755     struct buf *bp;
4756
4757     if(ip->type == T_DEV){
4758         if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
4759             return -1;
4760         return devsw[ip->major].write(ip, src, n);
4761     }
4762
4763     if(off > ip->size || off + n < off)
4764         return -1;
4765     if(off + n > MAXFILE*BSIZE)
4766         return -1;
4767
4768     for(tot=0; tot<n; tot+=m, off+=m, src+=m){
4769         bp = bread(ip->dev, bmap(ip, off/BSIZE));
4770         m = min(n - tot, BSIZE - off%BSIZE);
4771         memmove(bp->data + off%BSIZE, src, m);
4772         log_write(bp);
4773         brelse(bp);
4774     }
4775
4776     if(n > 0 && off > ip->size){
4777         ip->size = off;
4778         iupdate(ip);
4779     }
4780     return n;
4781 }
4782
4783
4784
4785
4786
4787
4788
4789
4790
4791
4792
4793
4794
4795
4796
4797
4798
4799

```

```

4800 // Directories
4801
4802 int
4803 namecmp(const char *s, const char *t)
4804 {
4805     return strncmp(s, t, DIRSIZ);
4806 }
4807
4808 // Look for a directory entry in a directory.
4809 // If found, set *poff to byte offset of entry.
4810 // Caller must have already locked dp.
4811 struct inode*
4812 dirlookup(struct inode *dp, char *name, uint *poff)
4813 {
4814     uint off, inum;
4815     struct dirent de;
4816
4817     if(dp->type != T_DIR)
4818         panic("dirlookup not DIR");
4819
4820     for(off = 0; off < dp->size; off += sizeof(de)){
4821         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4822             panic("dirlink read");
4823         if(de.inum == 0)
4824             continue;
4825         if(namecmp(name, de.name) == 0){
4826             // entry matches path element
4827             if(poff)
4828                 *poff = off;
4829             inum = de.inum;
4830             return iget(dp->dev, inum);
4831         }
4832     }
4833
4834     return 0;
4835 }
4836
4837
4838
4839
4840
4841
4842
4843
4844
4845
4846
4847
4848
4849

```

```

4850 // Write a new directory entry (name, inum) into the directory dp.
4851 int
4852 dirlink(struct inode *dp, char *name, uint inum)
4853 {
4854     int off;
4855     struct dirent de;
4856     struct inode *ip;
4857
4858     // Check that name is not present.
4859     if((ip = dirlookup(dp, name, 0)) != 0){
4860         iput(ip);
4861         return -1;
4862     }
4863
4864     // Look for an empty dirent.
4865     for(off = 0; off < dp->size; off += sizeof(de)){
4866         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4867             panic("dirlink read");
4868         if(de.inum == 0)
4869             break;
4870     }
4871
4872     strncpy(de.name, name, DIRSIZ);
4873     de.inum = inum;
4874     if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
4875         panic("dirlink");
4876
4877     return 0;
4878 }
4879
4880
4881
4882
4883
4884
4885
4886
4887
4888
4889
4890
4891
4892
4893
4894
4895
4896
4897
4898
4899

```



```

4900 // Paths
4901
4902 // Copy the next path element from path into name.
4903 // Return a pointer to the element following the copied one.
4904 // The returned path has no leading slashes,
4905 // so the caller can check *path=='\0' to see if the name is the last one.
4906 // If no name to remove, return 0.
4907 //
4908 // Examples:
4909 //  skipelem("a/bb/c", name) = "bb/c", setting name = "a"
4910 //  skipelem("///a//bb", name) = "bb", setting name = "a"
4911 //  skipelem("a", name) = "", setting name = "a"
4912 //  skipelem("", name) = skipelem("///", name) = 0
4913 //
4914 static char*
4915 skipelem(char *path, char *name)
4916 {
4917     char *s;
4918     int len;
4919
4920     while(*path == '/')
4921         path++;
4922     if(*path == 0)
4923         return 0;
4924     s = path;
4925     while(*path != '/' && *path != 0)
4926         path++;
4927     len = path - s;
4928     if(len >= DIRSIZ)
4929         memmove(name, s, DIRSIZ);
4930     else {
4931         memmove(name, s, len);
4932         name[len] = 0;
4933     }
4934     while(*path == '/')
4935         path++;
4936     return path;
4937 }
4938
4939
4940
4941
4942
4943
4944
4945
4946
4947
4948
4949

```

```

4950 // Look up and return the inode for a path name.
4951 // If parent != 0, return the inode for the parent and copy the final
4952 // path element into name, which must have room for DIRSIZ bytes.
4953 static struct inode*
4954 namex(char *path, int nameparent, char *name)
4955 {
4956     struct inode *ip, *next;
4957
4958     if(*path == '/')
4959         ip = iget(ROOTDEV, ROOTINO);
4960     else
4961         ip = idup(proc->cwd);
4962
4963     while((path = skipelem(path, name)) != 0){
4964         ilock(ip);
4965         if(ip->type != T_DIR){
4966             iunlockput(ip);
4967             return 0;
4968         }
4969         if(nameparent && *path == '\0'){
4970             // Stop one level early.
4971             iunlock(ip);
4972             return ip;
4973         }
4974         if((next = dirlookup(ip, name, 0)) == 0){
4975             iunlockput(ip);
4976             return 0;
4977         }
4978         iunlockput(ip);
4979         ip = next;
4980     }
4981     if(nameparent){
4982         iput(ip);
4983         return 0;
4984     }
4985     return ip;
4986 }
4987
4988 struct inode*
4989 namei(char *path)
4990 {
4991     char name[DIRSIZ];
4992     return namex(path, 0, name);
4993 }
4994
4995 struct inode*
4996 nameparent(char *path, char *name)
4997 {
4998     return namex(path, 1, name);
4999 }

```

```

5000 #include "types.h"
5001 #include "defs.h"
5002 #include "param.h"
5003 #include "fs.h"
5004 #include "file.h"
5005 #include "spinlock.h"
5006
5007 struct devsw devsw[NDEV];
5008 struct {
5009     struct spinlock lock;
5010     struct file file[NFILE];
5011 } ftable;
5012
5013 void
5014 fileinit(void)
5015 {
5016     initlock(&ftable.lock, "ftable");
5017 }
5018
5019 // Allocate a file structure.
5020 struct file*
5021 filealloc(void)
5022 {
5023     struct file *f;
5024
5025     acquire(&ftable.lock);
5026     for(f = ftable.file; f < ftable.file + NFILE; f++){
5027         if(f->ref == 0){
5028             f->ref = 1;
5029             release(&ftable.lock);
5030             return f;
5031         }
5032     }
5033     release(&ftable.lock);
5034     return 0;
5035 }
5036
5037 // Increment ref count for file f.
5038 struct file*
5039 filedup(struct file *f)
5040 {
5041     acquire(&ftable.lock);
5042     if(f->ref < 1)
5043         panic("filedup");
5044     f->ref++;
5045     release(&ftable.lock);
5046     return f;
5047 }
5048
5049

```

```

5050 // Close file f. (Decrement ref count, close when reaches 0.)
5051 void
5052 fileclose(struct file *f)
5053 {
5054     struct file ff;
5055
5056     acquire(&ftable.lock);
5057     if(f->ref < 1)
5058         panic("fileclose");
5059     if(--f->ref > 0){
5060         release(&ftable.lock);
5061         return;
5062     }
5063     ff = *f;
5064     f->ref = 0;
5065     f->type = FD_NONE;
5066     release(&ftable.lock);
5067
5068     if(ff.type == FD_PIPE)
5069         pipeclose(ff.pipe, ff.writable);
5070     else if(ff.type == FD_INODE){
5071         begin_trans();
5072         iput(ff.ip);
5073         commit_trans();
5074     }
5075 }
5076
5077 // Get metadata about file f.
5078 int
5079 filestat(struct file *f, struct stat *st)
5080 {
5081     if(f->type == FD_INODE){
5082         ilock(f->ip);
5083         stati(f->ip, st);
5084         iunlock(f->ip);
5085         return 0;
5086     }
5087     return -1;
5088 }
5089
5090
5091
5092
5093
5094
5095
5096
5097
5098
5099

```

```

5100 // Read from file f. Addr is kernel address.
5101 int
5102 fileread(struct file *f, char *addr, int n)
5103 {
5104     int r;
5105
5106     if(f->readable == 0)
5107         return -1;
5108     if(f->type == FD_PIPE)
5109         return piperead(f->pipe, addr, n);
5110     if(f->type == FD_INODE){
5111         ilock(f->ip);
5112         if((r = readi(f->ip, addr, f->off, n)) > 0)
5113             f->off += r;
5114         iunlock(f->ip);
5115         return r;
5116     }
5117     panic("fileread");
5118 }
5119
5120
5121
5122
5123
5124
5125
5126
5127
5128
5129
5130
5131
5132
5133
5134
5135
5136
5137
5138
5139
5140
5141
5142
5143
5144
5145
5146
5147
5148
5149

```

```

5150 // Write to file f. Addr is kernel address.
5151 int
5152 filewrite(struct file *f, char *addr, int n)
5153 {
5154     int r;
5155
5156     if(f->writable == 0)
5157         return -1;
5158     if(f->type == FD_PIPE)
5159         return pipewrite(f->pipe, addr, n);
5160     if(f->type == FD_INODE){
5161         // write a few blocks at a time to avoid exceeding
5162         // the maximum log transaction size, including
5163         // i-node, indirect block, allocation blocks,
5164         // and 2 blocks of slop for non-aligned writes.
5165         // this really belongs lower down, since writei()
5166         // might be writing a device like the console.
5167         int max = ((LOGSIZE-1-1-2) / 2) * 512;
5168         int i = 0;
5169         while(i < n){
5170             int n1 = n - i;
5171             if(n1 > max)
5172                 n1 = max;
5173
5174             begin_trans();
5175             ilock(f->ip);
5176             if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
5177                 f->off += r;
5178             iunlock(f->ip);
5179             commit_trans();
5180
5181             if(r < 0)
5182                 break;
5183             if(r != n1)
5184                 panic("short filewrite");
5185             i += r;
5186         }
5187         return i == n ? n : -1;
5188     }
5189     panic("filewrite");
5190 }
5191
5192
5193
5194
5195
5196
5197
5198
5199

```

```

5200 #include "types.h"
5201 #include "defs.h"
5202 #include "param.h"
5203 #include "stat.h"
5204 #include "mmu.h"
5205 #include "proc.h"
5206 #include "fs.h"
5207 #include "file.h"
5208 #include "fcntl.h"
5209
5210 // Fetch the nth word-sized system call argument as a file descriptor
5211 // and return both the descriptor and the corresponding struct file.
5212 static int
5213 argfd(int n, int *pfd, struct file **pf)
5214 {
5215     int fd;
5216     struct file *f;
5217
5218     if(argint(n, &fd) < 0)
5219         return -1;
5220     if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
5221         return -1;
5222     if(pfd)
5223         *pfd = fd;
5224     if(pf)
5225         *pf = f;
5226     return 0;
5227 }
5228
5229 // Allocate a file descriptor for the given file.
5230 // Takes over file reference from caller on success.
5231 static int
5232 fdalloc(struct file *f)
5233 {
5234     int fd;
5235
5236     for(fd = 0; fd < NOFILE; fd++){
5237         if(proc->ofile[fd] == 0){
5238             proc->ofile[fd] = f;
5239             return fd;
5240         }
5241     }
5242     return -1;
5243 }
5244
5245
5246
5247
5248
5249

```

```

5250 int
5251 sys_dup(void)
5252 {
5253     struct file *f;
5254     int fd;
5255
5256     if(argfd(0, 0, &f) < 0)
5257         return -1;
5258     if((fd=fdalloc(f)) < 0)
5259         return -1;
5260     filedup(f);
5261     return fd;
5262 }
5263
5264 int
5265 sys_read(void)
5266 {
5267     struct file *f;
5268     int n;
5269     char *p;
5270
5271     if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5272         return -1;
5273     return fileread(f, p, n);
5274 }
5275
5276 int
5277 sys_write(void)
5278 {
5279     struct file *f;
5280     int n;
5281     char *p;
5282
5283     if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5284         return -1;
5285     return filewrite(f, p, n);
5286 }
5287
5288 int
5289 sys_close(void)
5290 {
5291     int fd;
5292     struct file *f;
5293
5294     if(argfd(0, &fd, &f) < 0)
5295         return -1;
5296     proc->ofile[fd] = 0;
5297     fileclose(f);
5298     return 0;
5299 }

```

```

5300 int
5301 sys_fstat(void)
5302 {
5303     struct file *f;
5304     struct stat *st;
5305
5306     if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
5307         return -1;
5308     return filestat(f, st);
5309 }
5310
5311 // Create the path new as a link to the same inode as old.
5312 int
5313 sys_link(void)
5314 {
5315     char name[DIRSIZ], *new, *old;
5316     struct inode *dp, *ip;
5317
5318     if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
5319         return -1;
5320     if((ip = namei(old)) == 0)
5321         return -1;
5322
5323     begin_trans();
5324
5325     ilock(ip);
5326     if(ip->type == T_DIR){
5327         iunlockput(ip);
5328         commit_trans();
5329         return -1;
5330     }
5331
5332     ip->nlink++;
5333     iupdate(ip);
5334     iunlock(ip);
5335
5336     if((dp = nameiparent(new, name)) == 0)
5337         goto bad;
5338     ilock(dp);
5339     if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
5340         iunlockput(dp);
5341         goto bad;
5342     }
5343     iunlockput(dp);
5344     iput(ip);
5345
5346     commit_trans();
5347
5348     return 0;
5349

```

```

5350 bad:
5351     ilock(ip);
5352     ip->nlink--;
5353     iupdate(ip);
5354     iunlockput(ip);
5355     commit_trans();
5356     return -1;
5357 }
5358
5359 // Is the directory dp empty except for "." and ".." ?
5360 static int
5361 isdirempty(struct inode *dp)
5362 {
5363     int off;
5364     struct dirent de;
5365
5366     for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
5367         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5368             panic("isdirempty: readi");
5369         if(de.inum != 0)
5370             return 0;
5371     }
5372     return 1;
5373 }
5374
5375
5376
5377
5378
5379
5380
5381
5382
5383
5384
5385
5386
5387
5388
5389
5390
5391
5392
5393
5394
5395
5396
5397
5398
5399

```

```

5400 int
5401 sys_unlink(void)
5402 {
5403     struct inode *ip, *dp;
5404     struct dirent de;
5405     char name[DIRSIZ], *path;
5406     uint off;
5407
5408     if(argstr(0, &path) < 0)
5409         return -1;
5410     if((dp = nameiparent(path, name)) == 0)
5411         return -1;
5412
5413     begin_trans();
5414
5415     ilock(dp);
5416
5417     // Cannot unlink "." or "..".
5418     if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
5419         goto bad;
5420
5421     if((ip = dirlookup(dp, name, &off)) == 0)
5422         goto bad;
5423     ilock(ip);
5424
5425     if(ip->nlink < 1)
5426         panic("unlink: nlink < 1");
5427     if(ip->type == T_DIR && !isdirempty(ip)){
5428         iunlockput(ip);
5429         goto bad;
5430     }
5431
5432     memset(&de, 0, sizeof(de));
5433     if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5434         panic("unlink: writei");
5435     if(ip->type == T_DIR){
5436         dp->nlink--;
5437         iupdate(dp);
5438     }
5439     iunlockput(dp);
5440
5441     ip->nlink--;
5442     iupdate(ip);
5443     iunlockput(ip);
5444
5445     commit_trans();
5446
5447     return 0;
5448
5449

```

```

5450 bad:
5451     iunlockput(dp);
5452     commit_trans();
5453     return -1;
5454 }
5455
5456 static struct inode*
5457 create(char *path, short type, short major, short minor)
5458 {
5459     uint off;
5460     struct inode *ip, *dp;
5461     char name[DIRSIZ];
5462
5463     if((dp = nameiparent(path, name)) == 0)
5464         return 0;
5465     ilock(dp);
5466
5467     if((ip = dirlookup(dp, name, &off)) != 0){
5468         iunlockput(dp);
5469         ilock(ip);
5470         if(type == T_FILE && ip->type == T_FILE)
5471             return ip;
5472         iunlockput(ip);
5473         return 0;
5474     }
5475
5476     if((ip = ialloc(dp->dev, type)) == 0)
5477         panic("create: ialloc");
5478
5479     ilock(ip);
5480     ip->major = major;
5481     ip->minor = minor;
5482     ip->nlink = 1;
5483     iupdate(ip);
5484
5485     if(type == T_DIR){ // Create . and .. entries.
5486         dp->nlink++; // for ".."
5487         iupdate(dp);
5488         // No ip->nlink++ for ".": avoid cyclic ref count.
5489         if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
5490             panic("create dots");
5491     }
5492
5493     if(dirlink(dp, name, ip->inum) < 0)
5494         panic("create: dirlink");
5495
5496     iunlockput(dp);
5497
5498     return ip;
5499 }

```

```

5500 int
5501 sys_open(void)
5502 {
5503     char *path;
5504     int fd, omode;
5505     struct file *f;
5506     struct inode *ip;
5507
5508     if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
5509         return -1;
5510     if(omode & O_CREATE){
5511         begin_trans();
5512         ip = create(path, T_FILE, 0, 0);
5513         commit_trans();
5514         if(ip == 0)
5515             return -1;
5516     } else {
5517         if((ip = namei(path)) == 0)
5518             return -1;
5519         ilock(ip);
5520         if(ip->type == T_DIR && omode != O_RDONLY){
5521             iunlockput(ip);
5522             return -1;
5523         }
5524     }
5525
5526     if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
5527         if(f)
5528             fileclose(f);
5529         iunlockput(ip);
5530         return -1;
5531     }
5532     iunlock(ip);
5533
5534     f->type = FD_INODE;
5535     f->ip = ip;
5536     f->off = 0;
5537     f->readable = !(omode & O_WRONLY);
5538     f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
5539     return fd;
5540 }
5541
5542
5543
5544
5545
5546
5547
5548
5549

```

```

5550 int
5551 sys_mkdir(void)
5552 {
5553     char *path;
5554     struct inode *ip;
5555
5556     begin_trans();
5557     if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
5558         commit_trans();
5559         return -1;
5560     }
5561     iunlockput(ip);
5562     commit_trans();
5563     return 0;
5564 }
5565
5566 int
5567 sys_mknod(void)
5568 {
5569     struct inode *ip;
5570     char *path;
5571     int len;
5572     int major, minor;
5573
5574     begin_trans();
5575     if((len=argstr(0, &path)) < 0 ||
5576        argint(1, &major) < 0 ||
5577        argint(2, &minor) < 0 ||
5578        (ip = create(path, T_DEV, major, minor)) == 0){
5579         commit_trans();
5580         return -1;
5581     }
5582     iunlockput(ip);
5583     commit_trans();
5584     return 0;
5585 }
5586
5587
5588
5589
5590
5591
5592
5593
5594
5595
5596
5597
5598
5599

```

```

5600 int
5601 sys_chdir(void)
5602 {
5603     char *path;
5604     struct inode *ip;
5605
5606     if(argstr(0, &path) < 0 || (ip = namei(path)) == 0)
5607         return -1;
5608     ilock(ip);
5609     if(ip->type != T_DIR){
5610         iunlockput(ip);
5611         return -1;
5612     }
5613     iunlock(ip);
5614     iput(proc->cwd);
5615     proc->cwd = ip;
5616     return 0;
5617 }
5618
5619 int
5620 sys_exec(void)
5621 {
5622     char *path, *argv[MAXARG];
5623     int i;
5624     uint uargv, uarg;
5625
5626     if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
5627         return -1;
5628     }
5629     memset(argv, 0, sizeof(argv));
5630     for(i=0;; i++){
5631         if(i >= NELEM(argv))
5632             return -1;
5633         if(fetchint(proc, uargv+4*i, (int*)&uarg) < 0)
5634             return -1;
5635         if(uarg == 0){
5636             argv[i] = 0;
5637             break;
5638         }
5639         if(fetchstr(proc, uarg, &argv[i]) < 0)
5640             return -1;
5641     }
5642     return exec(path, argv);
5643 }
5644
5645
5646
5647
5648
5649

```

```

5650 int
5651 sys_pipe(void)
5652 {
5653     int *fd;
5654     struct file *rf, *wf;
5655     int fd0, fd1;
5656
5657     if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
5658         return -1;
5659     if(pipealloc(&rf, &wf) < 0)
5660         return -1;
5661     fd0 = -1;
5662     if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
5663         if(fd0 >= 0)
5664             proc->ofile[fd0] = 0;
5665         fileclose(rf);
5666         fileclose(wf);
5667         return -1;
5668     }
5669     fd[0] = fd0;
5670     fd[1] = fd1;
5671     return 0;
5672 }
5673
5674
5675
5676
5677
5678
5679
5680
5681
5682
5683
5684
5685
5686
5687
5688
5689
5690
5691
5692
5693
5694
5695
5696
5697
5698
5699

```



```

5700 #include "types.h"
5701 #include "param.h"
5702 #include "memlayout.h"
5703 #include "mmu.h"
5704 #include "proc.h"
5705 #include "defs.h"
5706 #include "x86.h"
5707 #include "elf.h"
5708
5709 int
5710 exec(char *path, char **argv)
5711 {
5712     char *s, *last;
5713     int i, off;
5714     uint argc, sz, sp, ustack[3+MAXARG+1];
5715     struct elfhdr elf;
5716     struct inode *ip;
5717     struct proghdr ph;
5718     pde_t *pgdir, *oldpgdir;
5719
5720     if((ip = namei(path)) == 0)
5721         return -1;
5722     ilock(ip);
5723     pgdir = 0;
5724
5725     // Check ELF header
5726     if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
5727         goto bad;
5728     if(elf.magic != ELF_MAGIC)
5729         goto bad;
5730
5731     if((pgdir = setupvm(kalloc)) == 0)
5732         goto bad;
5733
5734     // Load program into memory.
5735     sz = 0;
5736     for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
5737         if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
5738             goto bad;
5739         if(ph.type != ELF_PROG_LOAD)
5740             continue;
5741         if(ph.memsz < ph.filesz)
5742             goto bad;
5743         if((sz = allocvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
5744             goto bad;
5745         if(loadvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
5746             goto bad;
5747     }
5748     iunlockput(ip);
5749     ip = 0;

```

```

5750 // Allocate two pages at the next page boundary.
5751 // Make the first inaccessible. Use the second as the user stack.
5752 sz = PGROUNDUP(sz);
5753 if((sz = allocvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
5754     goto bad;
5755 clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
5756 sp = sz;
5757
5758 // Push argument strings, prepare rest of stack in ustack.
5759 for(argc = 0; argv[argc]; argc++) {
5760     if(argc >= MAXARG)
5761         goto bad;
5762     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
5763     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
5764         goto bad;
5765     ustack[3+argc] = sp;
5766 }
5767 ustack[3+argc] = 0;
5768
5769 ustack[0] = 0xffffffff; // fake return PC
5770 ustack[1] = argc;
5771 ustack[2] = sp - (argc+1)*4; // argv pointer
5772
5773 sp -= (3+argc+1) * 4;
5774 if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
5775     goto bad;
5776
5777 // Save program name for debugging.
5778 for(last=s=path; *s; s++)
5779     if(*s == '/')
5780         last = s+1;
5781 safestrcpy(proc->name, last, sizeof(proc->name));
5782
5783 // Commit to the user image.
5784 oldpgdir = proc->pgdir;
5785 proc->pgdir = pgdir;
5786 proc->sz = sz;
5787 proc->tf->eip = elf.entry; // main
5788 proc->tf->esp = sp;
5789 switchvm(proc);
5790 freevm(oldpgdir);
5791 return 0;
5792
5793 bad:
5794 if(pgdir)
5795     freevm(pgdir);
5796 if(ip)
5797     iunlockput(ip);
5798 return -1;
5799 }

```

```

5800 #include "types.h"
5801 #include "defs.h"
5802 #include "param.h"
5803 #include "mmu.h"
5804 #include "proc.h"
5805 #include "fs.h"
5806 #include "file.h"
5807 #include "spinlock.h"
5808
5809 #define PIPESIZE 512
5810
5811 struct pipe {
5812     struct spinlock lock;
5813     char data[PIPESIZE];
5814     uint nread;    // number of bytes read
5815     uint nwrite;   // number of bytes written
5816     int readopen; // read fd is still open
5817     int writeopen; // write fd is still open
5818 };
5819
5820 int
5821 pipealloc(struct file **f0, struct file **f1)
5822 {
5823     struct pipe *p;
5824
5825     p = 0;
5826     *f0 = *f1 = 0;
5827     if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
5828         goto bad;
5829     if((p = (struct pipe*)kalloc()) == 0)
5830         goto bad;
5831     p->readopen = 1;
5832     p->writeopen = 1;
5833     p->nwrite = 0;
5834     p->nread = 0;
5835     initlock(&p->lock, "pipe");
5836     (*f0)->type = FD_PIPE;
5837     (*f0)->readable = 1;
5838     (*f0)->writable = 0;
5839     (*f0)->pipe = p;
5840     (*f1)->type = FD_PIPE;
5841     (*f1)->readable = 0;
5842     (*f1)->writable = 1;
5843     (*f1)->pipe = p;
5844     return 0;
5845
5846
5847
5848
5849

```

```

5850 bad:
5851     if(p)
5852         kfree((char*)p);
5853     if(*f0)
5854         fileclose(*f0);
5855     if(*f1)
5856         fileclose(*f1);
5857     return -1;
5858 }
5859
5860 void
5861 pipeclose(struct pipe *p, int writable)
5862 {
5863     acquire(&p->lock);
5864     if(writable){
5865         p->writeopen = 0;
5866         wakeup(&p->nread);
5867     } else {
5868         p->readopen = 0;
5869         wakeup(&p->nwrite);
5870     }
5871     if(p->readopen == 0 && p->writeopen == 0){
5872         release(&p->lock);
5873         kfree((char*)p);
5874     } else
5875         release(&p->lock);
5876 }
5877
5878 int
5879 pipewrite(struct pipe *p, char *addr, int n)
5880 {
5881     int i;
5882
5883     acquire(&p->lock);
5884     for(i = 0; i < n; i++){
5885         while(p->nwrite == p->nread + PIPESIZE){
5886             if(p->readopen == 0 || proc->killed){
5887                 release(&p->lock);
5888                 return -1;
5889             }
5890             wakeup(&p->nread);
5891             sleep(&p->nwrite, &p->lock);
5892         }
5893         p->data[p->nwrite++ % PIPESIZE] = addr[i];
5894     }
5895     wakeup(&p->nread);
5896     release(&p->lock);
5897     return n;
5898 }
5899

```

```

5900 int
5901 piperead(struct pipe *p, char *addr, int n)
5902 {
5903     int i;
5904
5905     acquire(&p->lock);
5906     while(p->nread == p->nwrite && p->writeopen){
5907         if(proc->killed){
5908             release(&p->lock);
5909             return -1;
5910         }
5911         sleep(&p->nread, &p->lock);
5912     }
5913     for(i = 0; i < n; i++){
5914         if(p->nread == p->nwrite)
5915             break;
5916         addr[i] = p->data[p->nread++ % PIPESIZE];
5917     }
5918     wakeup(&p->nwrite);
5919     release(&p->lock);
5920     return i;
5921 }
5922
5923
5924
5925
5926
5927
5928
5929
5930
5931
5932
5933
5934
5935
5936
5937
5938
5939
5940
5941
5942
5943
5944
5945
5946
5947
5948
5949

```

```

5950 #include "types.h"
5951 #include "x86.h"
5952
5953 void*
5954 memset(void *dst, int c, uint n)
5955 {
5956     if ((int)dst%4 == 0 && n%4 == 0){
5957         c &= 0xFF;
5958         stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
5959     } else
5960         stosb(dst, c, n);
5961     return dst;
5962 }
5963
5964 int
5965 memcmp(const void *v1, const void *v2, uint n)
5966 {
5967     const uchar *s1, *s2;
5968
5969     s1 = v1;
5970     s2 = v2;
5971     while(n-- > 0){
5972         if(*s1 != *s2)
5973             return *s1 - *s2;
5974         s1++, s2++;
5975     }
5976
5977     return 0;
5978 }
5979
5980 void*
5981 memmove(void *dst, const void *src, uint n)
5982 {
5983     const char *s;
5984     char *d;
5985
5986     s = src;
5987     d = dst;
5988     if(s < d && s + n > d){
5989         s += n;
5990         d += n;
5991         while(n-- > 0)
5992             *--d = *--s;
5993     } else
5994         while(n-- > 0)
5995             *d++ = *s++;
5996
5997     return dst;
5998 }
5999

```

```

6000 // memcpy exists to placate GCC. Use memmove.
6001 void*
6002 memcpy(void *dst, const void *src, uint n)
6003 {
6004     return memmove(dst, src, n);
6005 }
6006
6007 int
6008 strncmp(const char *p, const char *q, uint n)
6009 {
6010     while(n > 0 && *p && *p == *q)
6011         n--, p++, q++;
6012     if(n == 0)
6013         return 0;
6014     return (uchar)*p - (uchar)*q;
6015 }
6016
6017 char*
6018 strncpy(char *s, const char *t, int n)
6019 {
6020     char *os;
6021
6022     os = s;
6023     while(n-- > 0 && (*s++ = *t++) != 0)
6024         ;
6025     while(n-- > 0)
6026         *s++ = 0;
6027     return os;
6028 }
6029
6030 // Like strncpy but guaranteed to NUL-terminate.
6031 char*
6032 safestrcpy(char *s, const char *t, int n)
6033 {
6034     char *os;
6035
6036     os = s;
6037     if(n <= 0)
6038         return os;
6039     while(--n > 0 && (*s++ = *t++) != 0)
6040         ;
6041     *s = 0;
6042     return os;
6043 }
6044
6045
6046
6047
6048
6049

```

```

6050 int
6051 strlen(const char *s)
6052 {
6053     int n;
6054
6055     for(n = 0; s[n]; n++)
6056         ;
6057     return n;
6058 }
6059
6060
6061
6062
6063
6064
6065
6066
6067
6068
6069
6070
6071
6072
6073
6074
6075
6076
6077
6078
6079
6080
6081
6082
6083
6084
6085
6086
6087
6088
6089
6090
6091
6092
6093
6094
6095
6096
6097
6098
6099

```

```

6100 // See MultiProcessor Specification Version 1.[14]
6101
6102 struct mp {           // floating pointer
6103     uchar signature[4]; // "_MP_"
6104     void *physaddr;     // phys addr of MP config table
6105     uchar length;       // 1
6106     uchar specrev;     // [14]
6107     uchar checksum;    // all bytes must add up to 0
6108     uchar type;        // MP system config type
6109     uchar imcrp;
6110     uchar reserved[3];
6111 };
6112
6113 struct mpconf {      // configuration table header
6114     uchar signature[4]; // "PCMP"
6115     ushort length;     // total table length
6116     uchar version;     // [14]
6117     uchar checksum;    // all bytes must add up to 0
6118     uchar product[20]; // product id
6119     uint *oemtable;    // OEM table pointer
6120     ushort oemlength; // OEM table length
6121     ushort entry;     // entry count
6122     uint *lapicaddr;  // address of local APIC
6123     ushort xlength;  // extended table length
6124     uchar xchecksum; // extended table checksum
6125     uchar reserved;
6126 };
6127
6128 struct mpproc {      // processor table entry
6129     uchar type;       // entry type (0)
6130     uchar apicid;    // local APIC id
6131     uchar version;   // local APIC verison
6132     uchar flags;     // CPU flags
6133     #define MPBOOT 0x02 // This proc is the bootstrap processor.
6134     uchar signature[4]; // CPU signature
6135     uint feature;     // feature flags from CPUID instruction
6136     uchar reserved[8];
6137 };
6138
6139 struct mpioapic {   // I/O APIC table entry
6140     uchar type;     // entry type (2)
6141     uchar apicno;  // I/O APIC id
6142     uchar version; // I/O APIC version
6143     uchar flags;   // I/O APIC flags
6144     uint *addr;    // I/O APIC address
6145 };
6146
6147
6148
6149

```

```

6150 // Table entry types
6151 #define MPPROC 0x00 // One per processor
6152 #define MPBUS 0x01 // One per bus
6153 #define MPPIOAPIC 0x02 // One per I/O APIC
6154 #define MPIOINTR 0x03 // One per bus interrupt source
6155 #define MPLINTR 0x04 // One per system interrupt source
6156
6157
6158
6159
6160
6161
6162
6163
6164
6165
6166
6167
6168
6169
6170
6171
6172
6173
6174
6175
6176
6177
6178
6179
6180
6181
6182
6183
6184
6185
6186
6187
6188
6189
6190
6191
6192
6193
6194
6195
6196
6197
6198
6199

```

```

6200 // Multiprocessor support
6201 // Search memory for MP description structures.
6202 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
6203
6204 #include "types.h"
6205 #include "defs.h"
6206 #include "param.h"
6207 #include "memlayout.h"
6208 #include "mp.h"
6209 #include "x86.h"
6210 #include "mmu.h"
6211 #include "proc.h"
6212
6213 struct cpu cpus[NCPU];
6214 static struct cpu *bcpu;
6215 int ismp;
6216 int ncpu;
6217 uchar ioapicid;
6218
6219 int
6220 mpbcpu(void)
6221 {
6222     return bcpu-cpus;
6223 }
6224
6225 static uchar
6226 sum(uchar *addr, int len)
6227 {
6228     int i, sum;
6229
6230     sum = 0;
6231     for(i=0; i<len; i++)
6232         sum += addr[i];
6233     return sum;
6234 }
6235
6236 // Look for an MP structure in the len bytes at addr.
6237 static struct mp*
6238 mpsearch1(uint a, int len)
6239 {
6240     uchar *e, *p, *addr;
6241
6242     addr = p2v(a);
6243     e = addr+len;
6244     for(p = addr; p < e; p += sizeof(struct mp))
6245         if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
6246             return (struct mp*)p;
6247     return 0;
6248 }
6249

```

```

6250 // Search for the MP Floating Pointer Structure, which according to the
6251 // spec is in one of the following three locations:
6252 // 1) in the first KB of the EBDA;
6253 // 2) in the last KB of system base memory;
6254 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
6255 static struct mp*
6256 mpsearch(void)
6257 {
6258     uchar *bda;
6259     uint p;
6260     struct mp *mp;
6261
6262     bda = (uchar *) P2V(0x400);
6263     if((p = ((bda[0x0F]<<8) | bda[0x0E]) << 4)){
6264         if((mp = mpsearch1(p, 1024)))
6265             return mp;
6266     } else {
6267         p = ((bda[0x14]<<8) | bda[0x13])*1024;
6268         if((mp = mpsearch1(p-1024, 1024)))
6269             return mp;
6270     }
6271     return mpsearch1(0xF0000, 0x10000);
6272 }
6273
6274 // Search for an MP configuration table. For now,
6275 // don't accept the default configurations (physaddr == 0).
6276 // Check for correct signature, calculate the checksum and,
6277 // if correct, check the version.
6278 // To do: check extended table checksum.
6279 static struct mpconf*
6280 mpconfig(struct mp **pmp)
6281 {
6282     struct mpconf *conf;
6283     struct mp *mp;
6284
6285     if((mp = mpsearch()) == 0 || mp->physaddr == 0)
6286         return 0;
6287     conf = (struct mpconf*) p2v((uint) mp->physaddr);
6288     if(memcmp(conf, "PCMP", 4) != 0)
6289         return 0;
6290     if(conf->version != 1 && conf->version != 4)
6291         return 0;
6292     if(sum((uchar*)conf, conf->length) != 0)
6293         return 0;
6294     *pmp = mp;
6295     return conf;
6296 }
6297
6298
6299

```

```

6300 void
6301 mpinit(void)
6302 {
6303     uchar *p, *e;
6304     struct mp *mp;
6305     struct mpconf *conf;
6306     struct mpproc *proc;
6307     struct mpioapic *ioapic;
6308
6309     bcpu = &cpus[0];
6310     if((conf = mpconfig(&mp)) == 0)
6311         return;
6312     ismp = 1;
6313     lapic = (uint*)conf->lapicaddr;
6314     for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
6315         switch(*p){
6316             case MPPROC:
6317                 proc = (struct mpproc*)p;
6318                 if(ncpu != proc->apicid){
6319                     cprintf("mpinit: ncpu=%d apicid=%d\n", ncpu, proc->apicid);
6320                     ismp = 0;
6321                 }
6322                 if(proc->flags & MPBOOT)
6323                     bcpu = &cpus[ncpu];
6324                 cpus[ncpu].id = ncpu;
6325                 ncpu++;
6326                 p += sizeof(struct mpproc);
6327                 continue;
6328             case MPIOAPIC:
6329                 ioapic = (struct mpioapic*)p;
6330                 ioapicid = ioapic->apicno;
6331                 p += sizeof(struct mpioapic);
6332                 continue;
6333             case MPBUS:
6334             case MPIOINTR:
6335             case MPLINTR:
6336                 p += 8;
6337                 continue;
6338             default:
6339                 cprintf("mpinit: unknown config type %x\n", *p);
6340                 ismp = 0;
6341         }
6342     }
6343     if(!ismp){
6344         // Didn't like what we found; fall back to no MP.
6345         ncpu = 1;
6346         lapic = 0;
6347         ioapicid = 0;
6348         return;
6349     }

```

```

6350     if(mp->imcrp){
6351         // Bochs doesn't support IMCR, so this doesn't run on Bochs.
6352         // But it would on real hardware.
6353         outb(0x22, 0x70); // Select IMCR
6354         outb(0x23, inb(0x23) | 1); // Mask external interrupts.
6355     }
6356 }
6357
6358
6359
6360
6361
6362
6363
6364
6365
6366
6367
6368
6369
6370
6371
6372
6373
6374
6375
6376
6377
6378
6379
6380
6381
6382
6383
6384
6385
6386
6387
6388
6389
6390
6391
6392
6393
6394
6395
6396
6397
6398
6399

```

```

6400 // The local APIC manages internal (non-I/O) interrupts.
6401 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
6402
6403 #include "types.h"
6404 #include "defs.h"
6405 #include "memlayout.h"
6406 #include "traps.h"
6407 #include "mmu.h"
6408 #include "x86.h"
6409
6410 // Local APIC registers, divided by 4 for use as uint[] indices.
6411 #define ID      (0x0020/4) // ID
6412 #define VER     (0x0030/4) // Version
6413 #define TPR    (0x0080/4) // Task Priority
6414 #define EOI    (0x00B0/4) // EOI
6415 #define SVR    (0x00F0/4) // Spurious Interrupt Vector
6416 #define ENABLE 0x00000100 // Unit Enable
6417 #define ESR    (0x0280/4) // Error Status
6418 #define ICRLO (0x0300/4) // Interrupt Command
6419 #define INIT   0x00000500 // INIT/RESET
6420 #define STARTUP 0x00000600 // Startup IPI
6421 #define DELIVS 0x00001000 // Delivery status
6422 #define ASSERT 0x00004000 // Assert interrupt (vs deassert)
6423 #define DEASSERT 0x00000000
6424 #define LEVEL  0x00008000 // Level triggered
6425 #define BCAST  0x00080000 // Send to all APICs, including self.
6426 #define BUSY   0x00001000
6427 #define FIXED  0x00000000
6428 #define ICRHI  (0x0310/4) // Interrupt Command [63:32]
6429 #define TIMER  (0x0320/4) // Local Vector Table 0 (TIMER)
6430 #define X1     0x0000000B // divide counts by 1
6431 #define PERIODIC 0x00020000 // Periodic
6432 #define PCINT  (0x0340/4) // Performance Counter LVT
6433 #define LINT0  (0x0350/4) // Local Vector Table 1 (LINT0)
6434 #define LINT1  (0x0360/4) // Local Vector Table 2 (LINT1)
6435 #define ERROR  (0x0370/4) // Local Vector Table 3 (ERROR)
6436 #define MASKED 0x00010000 // Interrupt masked
6437 #define TICC  (0x0380/4) // Timer Initial Count
6438 #define TCCR  (0x0390/4) // Timer Current Count
6439 #define TDCR  (0x03E0/4) // Timer Divide Configuration
6440
6441 volatile uint *lapic; // Initialized in mp.c
6442
6443 static void
6444 lapicw(int index, int value)
6445 {
6446     lapic[index] = value;
6447     lapic[ID]; // wait for write to finish, by reading
6448 }
6449

```

```

6450 void
6451 lapicinit(int c)
6452 {
6453     if(!lapic)
6454         return;
6455
6456     // Enable local APIC; set spurious interrupt vector.
6457     lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
6458
6459     // The timer repeatedly counts down at bus frequency
6460     // from lapic[TICC] and then issues an interrupt.
6461     // If xv6 cared more about precise timekeeping,
6462     // TICC would be calibrated using an external time source.
6463     lapicw(TDCR, X1);
6464     lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
6465     lapicw(TICC, 1000000);
6466
6467     // Disable logical interrupt lines.
6468     lapicw(LINT0, MASKED);
6469     lapicw(LINT1, MASKED);
6470
6471     // Disable performance counter overflow interrupts
6472     // on machines that provide that interrupt entry.
6473     if(((lapic[VER]>>16) & 0xFF) >= 4)
6474         lapicw(PCINT, MASKED);
6475
6476     // Map error interrupt to IRQ_ERROR.
6477     lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
6478
6479     // Clear error status register (requires back-to-back writes).
6480     lapicw(ESR, 0);
6481     lapicw(ESR, 0);
6482
6483     // Ack any outstanding interrupts.
6484     lapicw(EOI, 0);
6485
6486     // Send an Init Level De-Assert to synchronise arbitration ID's.
6487     lapicw(ICRHI, 0);
6488     lapicw(ICRLO, BCAST | INIT | LEVEL);
6489     while(lapic[ICRLO] & DELIVS)
6490         ;
6491
6492     // Enable interrupts on the APIC (but not on the processor).
6493     lapicw(TPR, 0);
6494 }
6495
6496
6497
6498
6499

```



```

6500 int
6501 cpunum(void)
6502 {
6503     // Cannot call cpu when interrupts are enabled:
6504     // result not guaranteed to last long enough to be used!
6505     // Would prefer to panic but even printing is chancy here:
6506     // almost everything, including cprintf and panic, calls cpu,
6507     // often indirectly through acquire and release.
6508     if(readeflags() & FL_IF){
6509         static int n;
6510         if(n++ == 0)
6511             cprintf("cpu called from %x with interrupts enabled\n",
6512                 __builtin_return_address(0));
6513     }
6514
6515     if(lapic)
6516         return lapic[ID]>>24;
6517     return 0;
6518 }
6519
6520 // Acknowledge interrupt.
6521 void
6522 lapiceoi(void)
6523 {
6524     if(lapic)
6525         lapicw(EOI, 0);
6526 }
6527
6528 // Spin for a given number of microseconds.
6529 // On real hardware would want to tune this dynamically.
6530 void
6531 microdelay(int us)
6532 {
6533 }
6534
6535 #define IO_RTC 0x70
6536
6537 // Start additional processor running entry code at addr.
6538 // See Appendix B of MultiProcessor Specification.
6539 void
6540 lapicstartap(uchar apicid, uint addr)
6541 {
6542     int i;
6543     ushort *wrv;
6544
6545     // "The BSP must initialize CMOS shutdown code to 0AH
6546     // and the warm reset vector (DWORD based at 40:67) to point at
6547     // the AP startup code prior to the [universal startup algorithm]."
6548     outb(IO_RTC, 0xF); // offset 0xF is shutdown code
6549     outb(IO_RTC+1, 0x0A);

```

```

6550     wrv = (ushort*)P2V((0x40<<4 | 0x67)); // Warm reset vector
6551     wrv[0] = 0;
6552     wrv[1] = addr >> 4;
6553
6554     // "Universal startup algorithm."
6555     // Send INIT (level-triggered) interrupt to reset other CPU.
6556     lapicw(ICRHI, apicid<<24);
6557     lapicw(ICRLO, INIT | LEVEL | ASSERT);
6558     microdelay(200);
6559     lapicw(ICRLO, INIT | LEVEL);
6560     microdelay(100); // should be 10ms, but too slow in Bochs!
6561
6562     // Send startup IPI (twice!) to enter code.
6563     // Regular hardware is supposed to only accept a STARTUP
6564     // when it is in the halted state due to an INIT. So the second
6565     // should be ignored, but it is part of the official Intel algorithm.
6566     // Bochs complains about the second one. Too bad for Bochs.
6567     for(i = 0; i < 2; i++){
6568         lapicw(ICRHI, apicid<<24);
6569         lapicw(ICRLO, STARTUP | (addr>>12));
6570         microdelay(200);
6571     }
6572 }
6573
6574
6575
6576
6577
6578
6579
6580
6581
6582
6583
6584
6585
6586
6587
6588
6589
6590
6591
6592
6593
6594
6595
6596
6597
6598
6599

```

```

6600 // The I/O APIC manages hardware interrupts for an SMP system.
6601 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
6602 // See also picirq.c.
6603
6604 #include "types.h"
6605 #include "defs.h"
6606 #include "traps.h"
6607
6608 #define IOAPIC 0xFEC00000 // Default physical address of IO APIC
6609
6610 #define REG_ID 0x00 // Register index: ID
6611 #define REG_VER 0x01 // Register index: version
6612 #define REG_TABLE 0x10 // Redirection table base
6613
6614 // The redirection table starts at REG_TABLE and uses
6615 // two registers to configure each interrupt.
6616 // The first (low) register in a pair contains configuration bits.
6617 // The second (high) register contains a bitmask telling which
6618 // CPUs can serve that interrupt.
6619 #define INT_DISABLED 0x00010000 // Interrupt disabled
6620 #define INT_LEVEL 0x00008000 // Level-triggered (vs edge-)
6621 #define INT_ACTIVELOW 0x00002000 // Active low (vs high)
6622 #define INT_LOGICAL 0x00000800 // Destination is CPU id (vs APIC ID)
6623
6624 volatile struct ioapic *ioapic;
6625
6626 // IO APIC MMIO structure: write reg, then read or write data.
6627 struct ioapic {
6628     uint reg;
6629     uint pad[3];
6630     uint data;
6631 };
6632
6633 static uint
6634 ioapicread(int reg)
6635 {
6636     ioapic->reg = reg;
6637     return ioapic->data;
6638 }
6639
6640 static void
6641 ioapicwrite(int reg, uint data)
6642 {
6643     ioapic->reg = reg;
6644     ioapic->data = data;
6645 }
6646
6647
6648
6649

```

```

6650 void
6651 ioapicinit(void)
6652 {
6653     int i, id, maxintr;
6654
6655     if(!ismp)
6656         return;
6657
6658     ioapic = (volatile struct ioapic*)IOAPIC;
6659     maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
6660     id = ioapicread(REG_ID) >> 24;
6661     if(id != ioapicid)
6662         cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
6663
6664     // Mark all interrupts edge-triggered, active high, disabled,
6665     // and not routed to any CPUs.
6666     for(i = 0; i <= maxintr; i++){
6667         ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
6668         ioapicwrite(REG_TABLE+2*i+1, 0);
6669     }
6670 }
6671
6672 void
6673 ioapicenable(int irq, int cpunum)
6674 {
6675     if(!ismp)
6676         return;
6677
6678     // Mark interrupt edge-triggered, active high,
6679     // enabled, and routed to the given cpunum,
6680     // which happens to be that cpu's APIC ID.
6681     ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
6682     ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
6683 }
6684
6685
6686
6687
6688
6689
6690
6691
6692
6693
6694
6695
6696
6697
6698
6699

```

```

6700 // Intel 8259A programmable interrupt controllers.
6701
6702 #include "types.h"
6703 #include "x86.h"
6704 #include "traps.h"
6705
6706 // I/O Addresses of the two programmable interrupt controllers
6707 #define IO_PIC1      0x20 // Master (IRQs 0-7)
6708 #define IO_PIC2      0xA0 // Slave (IRQs 8-15)
6709
6710 #define IRQ_SLAVE    2 // IRQ at which slave connects to master
6711
6712 // Current IRQ mask.
6713 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
6714 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
6715
6716 static void
6717 picsetmask(ushort mask)
6718 {
6719     irqmask = mask;
6720     outb(IO_PIC1+1, mask);
6721     outb(IO_PIC2+1, mask >> 8);
6722 }
6723
6724 void
6725 picenable(int irq)
6726 {
6727     picsetmask(irqmask & ~(1<<irq));
6728 }
6729
6730 // Initialize the 8259A interrupt controllers.
6731 void
6732 picinit(void)
6733 {
6734     // mask all interrupts
6735     outb(IO_PIC1+1, 0xFF);
6736     outb(IO_PIC2+1, 0xFF);
6737
6738     // Set up master (8259A-1)
6739
6740     // ICW1: 0001g0hi
6741     //   g: 0 = edge triggering, 1 = level triggering
6742     //   h: 0 = cascaded PICs, 1 = master only
6743     //   i: 0 = no ICW4, 1 = ICW4 required
6744     outb(IO_PIC1, 0x11);
6745
6746     // ICW2: Vector offset
6747     outb(IO_PIC1+1, T_IRQ0);
6748
6749

```

```

6750 // ICW3: (master PIC) bit mask of IR lines connected to slaves
6751 //       (slave PIC) 3-bit # of slave's connection to master
6752 outb(IO_PIC1+1, 1<<IRQ_SLAVE);
6753
6754 // ICW4: 000nbmap
6755 //   n: 1 = special fully nested mode
6756 //   b: 1 = buffered mode
6757 //   m: 0 = slave PIC, 1 = master PIC
6758 //       (ignored when b is 0, as the master/slave role
6759 //       can be hardwired).
6760 //   a: 1 = Automatic EOI mode
6761 //   p: 0 = MCS-80/85 mode, 1 = intel x86 mode
6762 outb(IO_PIC1+1, 0x3);
6763
6764 // Set up slave (8259A-2)
6765 outb(IO_PIC2, 0x11); // ICW1
6766 outb(IO_PIC2+1, T_IRQ0 + 8); // ICW2
6767 outb(IO_PIC2+1, IRQ_SLAVE); // ICW3
6768 // NB Automatic EOI mode doesn't tend to work on the slave.
6769 // Linux source code says it's "to be investigated".
6770 outb(IO_PIC2+1, 0x3); // ICW4
6771
6772 // OCW3: 0ef01prs
6773 //   ef: 0x = NOP, 10 = clear specific mask, 11 = set specific mask
6774 //   p: 0 = no polling, 1 = polling mode
6775 //   rs: 0x = NOP, 10 = read IRR, 11 = read ISR
6776 outb(IO_PIC1, 0x68); // clear specific mask
6777 outb(IO_PIC1, 0x0a); // read IRR by default
6778
6779 outb(IO_PIC2, 0x68); // OCW3
6780 outb(IO_PIC2, 0x0a); // OCW3
6781
6782 if(irqmask != 0xFFFF)
6783     picsetmask(irqmask);
6784 }
6785
6786
6787
6788
6789
6790
6791
6792
6793
6794
6795
6796
6797
6798
6799

```

```

6800 // PC keyboard interface constants
6801
6802 #define KBSTATP      0x64    // kbd controller status port(I)
6803 #define KBS_DIB      0x01    // kbd data in buffer
6804 #define KBDATAP      0x60    // kbd data port(I)
6805
6806 #define NO            0
6807
6808 #define SHIFT        (1<<0)
6809 #define CTL          (1<<1)
6810 #define ALT          (1<<2)
6811
6812 #define CAPSLOCK     (1<<3)
6813 #define NUMLOCK     (1<<4)
6814 #define SCROLLLOCK  (1<<5)
6815
6816 #define EOESC        (1<<6)
6817
6818 // Special keycodes
6819 #define KEY_HOME     0xE0
6820 #define KEY_END      0xE1
6821 #define KEY_UP       0xE2
6822 #define KEY_DN       0xE3
6823 #define KEY_LF       0xE4
6824 #define KEY_RT       0xE5
6825 #define KEY_PGUP     0xE6
6826 #define KEY_PGDN     0xE7
6827 #define KEY_INS      0xE8
6828 #define KEY_DEL      0xE9
6829
6830 // C('A') == Control-A
6831 #define C(x) (x - '@')
6832
6833 static uchar shiftcode[256] =
6834 {
6835     [0x1D] CTL,
6836     [0x2A] SHIFT,
6837     [0x36] SHIFT,
6838     [0x38] ALT,
6839     [0x9D] CTL,
6840     [0xB8] ALT
6841 };
6842
6843 static uchar togglecode[256] =
6844 {
6845     [0x3A] CAPSLOCK,
6846     [0x45] NUMLOCK,
6847     [0x46] SCROLLLOCK
6848 };
6849

```

```

6850 static uchar normalmap[256] =
6851 {
6852     NO,    0x1B, '1', '2', '3', '4', '5', '6', // 0x00
6853     '7', '8', '9', '0', '-', '=', '\b', '\t',
6854     'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', // 0x10
6855     'o', 'p', '[', ']', '\n', NO, 'a', 's',
6856     'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', // 0x20
6857     '\'', ',', NO, '\\', 'z', 'x', 'c', 'v',
6858     'b', 'n', 'm', ',', '.', '/', NO, '*', // 0x30
6859     NO, ' ', NO, NO, NO, NO, NO, NO,
6860     NO, NO, NO, NO, NO, NO, NO, '7', // 0x40
6861     '8', '9', '-', '4', '5', '6', '+', '1',
6862     '2', '3', '0', '.', NO, NO, NO, NO, // 0x50
6863     [0x9C] '\n', // KP_Enter
6864     [0xB5] '/', // KP_Div
6865     [0xC8] KEY_UP, [0xD0] KEY_DN,
6866     [0xC9] KEY_PGUP, [0xD1] KEY_PGDN,
6867     [0xCB] KEY_LF, [0xCD] KEY_RT,
6868     [0x97] KEY_HOME, [0xCF] KEY_END,
6869     [0xD2] KEY_INS, [0xD3] KEY_DEL
6870 };
6871
6872 static uchar shiftmap[256] =
6873 {
6874     NO,    033, '! ', '@', '#', '$', '%', '^', // 0x00
6875     '&', '*', '(', ')', '-', '+', '\b', '\t',
6876     'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', // 0x10
6877     'O', 'P', '{', '}', '\n', NO, 'A', 'S',
6878     'D', 'F', 'G', 'H', 'J', 'K', 'L', ';', // 0x20
6879     '""', '~', NO, '|', 'Z', 'X', 'C', 'V',
6880     'B', 'N', 'M', '<', '>', '?', NO, '*', // 0x30
6881     NO, ' ', NO, NO, NO, NO, NO, NO,
6882     NO, NO, NO, NO, NO, NO, NO, '7', // 0x40
6883     '8', '9', '-', '4', '5', '6', '+', '1',
6884     '2', '3', '0', '.', NO, NO, NO, NO, // 0x50
6885     [0x9C] '\n', // KP_Enter
6886     [0xB5] '/', // KP_Div
6887     [0xC8] KEY_UP, [0xD0] KEY_DN,
6888     [0xC9] KEY_PGUP, [0xD1] KEY_PGDN,
6889     [0xCB] KEY_LF, [0xCD] KEY_RT,
6890     [0x97] KEY_HOME, [0xCF] KEY_END,
6891     [0xD2] KEY_INS, [0xD3] KEY_DEL
6892 };
6893
6894
6895
6896
6897
6898
6899

```

```

6900 static uchar ctlmap[256] =
6901 {
6902  NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
6903  NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
6904  C('Q'), C('W'), C('E'), C('R'), C('T'), C('Y'), C('U'), C('I'),
6905  C('O'), C('P'), NO,     NO,     '\r',  NO,     C('A'), C('S'),
6906  C('D'), C('F'), C('G'), C('H'), C('J'), C('K'), C('L'), NO,
6907  NO,     NO,     NO,     C('\'), C('Z'), C('X'), C('C'), C('V'),
6908  C('B'), C('N'), C('M'), NO,     NO,     C('/'), NO,     NO,
6909  [0x9C] '\r', // KP_Enter
6910  [0xB5] C('/'), // KP_Div
6911  [0xC8] KEY_UP, [0xD0] KEY_DN,
6912  [0xC9] KEY_PGUP, [0xD1] KEY_PGDN,
6913  [0xCB] KEY_LF, [0xCD] KEY_RT,
6914  [0x97] KEY_HOME, [0xCF] KEY_END,
6915  [0xD2] KEY_INS, [0xD3] KEY_DEL
6916 };
6917
6918
6919
6920
6921
6922
6923
6924
6925
6926
6927
6928
6929
6930
6931
6932
6933
6934
6935
6936
6937
6938
6939
6940
6941
6942
6943
6944
6945
6946
6947
6948
6949

```

```

6950 #include "types.h"
6951 #include "x86.h"
6952 #include "defs.h"
6953 #include "kbd.h"
6954
6955 int
6956 kbdgetc(void)
6957 {
6958  static uint shift;
6959  static uchar *charcode[4] = {
6960   normalmap, shiftmap, ctlmap, ctlmap
6961  };
6962  uint st, data, c;
6963
6964  st = inb(KBSTATP);
6965  if((st & KBS_DIB) == 0)
6966   return -1;
6967  data = inb(KBDATAP);
6968
6969  if(data == 0xE0){
6970   shift |= EOESC;
6971   return 0;
6972  } else if(data & 0x80){
6973   // Key released
6974   data = (shift & EOESC ? data : data & 0x7F);
6975   shift &= ~(shiftcode[data] | EOESC);
6976   return 0;
6977  } else if(shift & EOESC){
6978   // Last character was an E0 escape; or with 0x80
6979   data |= 0x80;
6980   shift &= ~EOESC;
6981  }
6982
6983  shift |= shiftcode[data];
6984  shift ^= togglecode[data];
6985  c = charcode[shift & (CTL | SHIFT)][data];
6986  if(shift & CAPSLOCK){
6987   if('a' <= c && c <= 'z')
6988    c += 'A' - 'a';
6989   else if('A' <= c && c <= 'Z')
6990    c += 'a' - 'A';
6991  }
6992  return c;
6993 }
6994
6995 void
6996 kbdtintr(void)
6997 {
6998  consoleintr(kbdgetc);
6999 }

```

```

7000 // Console input and output.
7001 // Input is from the keyboard or serial port.
7002 // Output is written to the screen and serial port.
7003
7004 #include "types.h"
7005 #include "defs.h"
7006 #include "param.h"
7007 #include "traps.h"
7008 #include "spinlock.h"
7009 #include "fs.h"
7010 #include "file.h"
7011 #include "memlayout.h"
7012 #include "mmu.h"
7013 #include "proc.h"
7014 #include "x86.h"
7015
7016 static void consputc(int);
7017
7018 static int panicked = 0;
7019
7020 static struct {
7021   struct spinlock lock;
7022   int locking;
7023 } cons;
7024
7025 static void
7026 printint(int xx, int base, int sign)
7027 {
7028   static char digits[] = "0123456789abcdef";
7029   char buf[16];
7030   int i;
7031   uint x;
7032
7033   if(sign && (sign = xx < 0))
7034     x = -xx;
7035   else
7036     x = xx;
7037
7038   i = 0;
7039   do{
7040     buf[i++] = digits[x % base];
7041   }while((x /= base) != 0);
7042
7043   if(sign)
7044     buf[i++] = '-';
7045
7046   while(--i >= 0)
7047     consputc(buf[i]);
7048 }
7049

```

```

7050 // Print to the console. only understands %d, %x, %p, %s.
7051 void
7052 cprintf(char *fmt, ...)
7053 {
7054   int i, c, state, locking;
7055   uint *argp;
7056   char *s;
7057
7058   locking = cons.locking;
7059   if(locking)
7060     acquire(&cons.lock);
7061
7062   if (fmt == 0)
7063     panic("null fmt");
7064
7065   argp = (uint*)(void*)&fmt + 1;
7066   state = 0;
7067   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
7068     if(c != '%'){
7069       consputc(c);
7070       continue;
7071     }
7072     c = fmt[++i] & 0xff;
7073     if(c == 0)
7074       break;
7075     switch(c){
7076     case 'd':
7077       printint(*argp++, 10, 1);
7078       break;
7079     case 'x':
7080     case 'p':
7081       printint(*argp++, 16, 0);
7082       break;
7083     case 's':
7084       if((s = (char*)*argp++) == 0)
7085         s = "(null)";
7086       for(; *s; s++)
7087         consputc(*s);
7088       break;
7089     case '%':
7090       consputc('%');
7091       break;
7092     default:
7093       // Print unknown % sequence to draw attention.
7094       consputc('%');
7095       consputc(c);
7096       break;
7097     }
7098   }
7099

```

```

7100  if(locking)
7101      release(&cons.lock);
7102  }
7103
7104  void
7105  panic(char *s)
7106  {
7107      int i;
7108      uint pcs[10];
7109
7110      cli();
7111      cons.locking = 0;
7112      cprintf("cpu%d: panic: ", cpu->id);
7113      cprintf(s);
7114      cprintf("\n");
7115      getcallerpcs(&s, pcs);
7116      for(i=0; i<10; i++)
7117          cprintf(" %p", pcs[i]);
7118      panicked = 1; // freeze other CPU
7119      for(;;)
7120          ;
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  #define BACKSPACE 0x100
7151  #define CRTPORT 0x3d4
7152  static ushort *crt = (ushort*)P2V(0xb8000); // CGA memory
7153
7154  static void
7155  cgaputc(int c)
7156  {
7157      int pos;
7158
7159      // Cursor position: col + 80*row.
7160      outb(CRTPORT, 14);
7161      pos = inb(CRTPORT+1) << 8;
7162      outb(CRTPORT, 15);
7163      pos |= inb(CRTPORT+1);
7164
7165      if(c == '\n')
7166          pos += 80 - pos%80;
7167      else if(c == BACKSPACE){
7168          if(pos > 0) --pos;
7169      } else
7170          crt[pos++] = (c&0xff) | 0x0700; // black on white
7171
7172      if((pos/80) >= 24){ // Scroll up.
7173          memmove(crt, crt+80, sizeof(crt[0])*23*80);
7174          pos -= 80;
7175          memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
7176      }
7177
7178      outb(CRTPORT, 14);
7179      outb(CRTPORT+1, pos>>8);
7180      outb(CRTPORT, 15);
7181      outb(CRTPORT+1, pos);
7182      crt[pos] = ' ' | 0x0700;
7183  }
7184
7185  void
7186  consputc(int c)
7187  {
7188      if(panicked){
7189          cli();
7190          for(;;)
7191              ;
7192      }
7193
7194      if(c == BACKSPACE){
7195          uartputc('\b'); uartputc(' '); uartputc('\b');
7196      } else
7197          uartputc(c);
7198      cgaputc(c);
7199  }

```

```

7200 #define INPUT_BUF 128
7201 struct {
7202     struct spinlock lock;
7203     char buf[INPUT_BUF];
7204     uint r; // Read index
7205     uint w; // Write index
7206     uint e; // Edit index
7207 } input;
7208
7209 #define C(x) ((x)-'@') // Control-x
7210
7211 void
7212 consoleintr(int (*getc)(void))
7213 {
7214     int c;
7215
7216     acquire(&input.lock);
7217     while((c = getc()) >= 0){
7218         switch(c){
7219             case C('P'): // Process listing.
7220                 procdump();
7221                 break;
7222             case C('U'): // Kill line.
7223                 while(input.e != input.w &&
7224                     input.buf[(input.e-1) % INPUT_BUF] != '\n'){
7225                     input.e--;
7226                     consputc(BACKSPACE);
7227                 }
7228                 break;
7229             case C('H'): case '\x7f': // Backspace
7230                 if(input.e != input.w){
7231                     input.e--;
7232                     consputc(BACKSPACE);
7233                 }
7234                 break;
7235             default:
7236                 if(c != 0 && input.e-input.r < INPUT_BUF){
7237                     c = (c == '\r') ? '\n' : c;
7238                     input.buf[input.e++ % INPUT_BUF] = c;
7239                     consputc(c);
7240                     if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
7241                         input.w = input.e;
7242                         wakeup(&input.r);
7243                     }
7244                 }
7245                 break;
7246         }
7247     }
7248     release(&input.lock);
7249 }

```

```

7250 int
7251 consoleread(struct inode *ip, char *dst, int n)
7252 {
7253     uint target;
7254     int c;
7255
7256     iunlock(ip);
7257     target = n;
7258     acquire(&input.lock);
7259     while(n > 0){
7260         while(input.r == input.w){
7261             if(proc->killed){
7262                 release(&input.lock);
7263                 ilock(ip);
7264                 return -1;
7265             }
7266             sleep(&input.r, &input.lock);
7267         }
7268         c = input.buf[input.r++ % INPUT_BUF];
7269         if(c == C('D')){ // EOF
7270             if(n < target){
7271                 // Save ^D for next time, to make sure
7272                 // caller gets a 0-byte result.
7273                 input.r--;
7274             }
7275             break;
7276         }
7277         *dst++ = c;
7278         --n;
7279         if(c == '\n')
7280             break;
7281     }
7282     release(&input.lock);
7283     ilock(ip);
7284
7285     return target - n;
7286 }
7287
7288
7289
7290
7291
7292
7293
7294
7295
7296
7297
7298
7299

```



```

7300 int
7301 consolewrite(struct inode *ip, char *buf, int n)
7302 {
7303     int i;
7304
7305     iunlock(ip);
7306     acquire(&cons.lock);
7307     for(i = 0; i < n; i++)
7308         consputc(buf[i] & 0xff);
7309     release(&cons.lock);
7310     ilock(ip);
7311
7312     return n;
7313 }
7314
7315 void
7316 consoleinit(void)
7317 {
7318     initlock(&cons.lock, "console");
7319     initlock(&input.lock, "input");
7320
7321     devsw[CONSOLE].write = consolewrite;
7322     devsw[CONSOLE].read = consoleread;
7323     cons.locking = 1;
7324
7325     picenable(IRQ_KBD);
7326     ioapicenable(IRQ_KBD, 0);
7327 }
7328
7329
7330
7331
7332
7333
7334
7335
7336
7337
7338
7339
7340
7341
7342
7343
7344
7345
7346
7347
7348
7349

```

```

7350 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
7351 // Only used on uniprocessors;
7352 // SMP machines use the local APIC timer.
7353
7354 #include "types.h"
7355 #include "defs.h"
7356 #include "traps.h"
7357 #include "x86.h"
7358
7359 #define IO_TIMER1      0x040          // 8253 Timer #1
7360
7361 // Frequency of all three count-down timers;
7362 // (TIMER_FREQ/freq) is the appropriate count
7363 // to generate a frequency of freq Hz.
7364
7365 #define TIMER_FREQ      1193182
7366 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
7367
7368 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
7369 #define TIMER_SELO      0x00          // select counter 0
7370 #define TIMER_RATEGEN    0x04          // mode 2, rate generator
7371 #define TIMER_16BIT      0x30          // r/w counter 16 bits, LSB first
7372
7373 void
7374 timerinit(void)
7375 {
7376     // Interrupt 100 times/sec.
7377     outb(TIMER_MODE, TIMER_SELO | TIMER_RATEGEN | TIMER_16BIT);
7378     outb(IO_TIMER1, TIMER_DIV(100) % 256);
7379     outb(IO_TIMER1, TIMER_DIV(100) / 256);
7380     picenable(IRQ_TIMER);
7381 }
7382
7383
7384
7385
7386
7387
7388
7389
7390
7391
7392
7393
7394
7395
7396
7397
7398
7399

```

```

7400 // Intel 8250 serial port (UART).
7401
7402 #include "types.h"
7403 #include "defs.h"
7404 #include "param.h"
7405 #include "traps.h"
7406 #include "spinlock.h"
7407 #include "fs.h"
7408 #include "file.h"
7409 #include "mmu.h"
7410 #include "proc.h"
7411 #include "x86.h"
7412
7413 #define COM1    0x3f8
7414
7415 static int uart;    // is there a uart?
7416
7417 void
7418 uartinit(void)
7419 {
7420     char *p;
7421
7422     // Turn off the FIFO
7423     outb(COM1+2, 0);
7424
7425     // 9600 baud, 8 data bits, 1 stop bit, parity off.
7426     outb(COM1+3, 0x80);    // Unlock divisor
7427     outb(COM1+0, 115200/9600);
7428     outb(COM1+1, 0);
7429     outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
7430     outb(COM1+4, 0);
7431     outb(COM1+1, 0x01);    // Enable receive interrupts.
7432
7433     // If status is 0xFF, no serial port.
7434     if(inb(COM1+5) == 0xFF)
7435         return;
7436     uart = 1;
7437
7438     // Acknowledge pre-existing interrupt conditions;
7439     // enable interrupts.
7440     inb(COM1+2);
7441     inb(COM1+0);
7442     picenable(IRQ_COM1);
7443     ioapicenable(IRQ_COM1, 0);
7444
7445     // Announce that we're here.
7446     for(p="xv6...\n"; *p; p++)
7447         uartputc(*p);
7448 }
7449

```

```

7450 void
7451 uartputc(int c)
7452 {
7453     int i;
7454
7455     if(!uart)
7456         return;
7457     for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
7458         microdelay(10);
7459     outb(COM1+0, c);
7460 }
7461
7462 static int
7463 uartgetc(void)
7464 {
7465     if(!uart)
7466         return -1;
7467     if(!(inb(COM1+5) & 0x01))
7468         return -1;
7469     return inb(COM1+0);
7470 }
7471
7472 void
7473 uartintr(void)
7474 {
7475     consoleintr(uartgetc);
7476 }
7477
7478
7479
7480
7481
7482
7483
7484
7485
7486
7487
7488
7489
7490
7491
7492
7493
7494
7495
7496
7497
7498
7499

```

```

7500 # Initial process execs /init.
7501
7502 #include "syscall.h"
7503 #include "traps.h"
7504
7505
7506 # exec(init, argv)
7507 .globl start
7508 start:
7509     pushl $argv
7510     pushl $init
7511     pushl $0 // where caller pc would be
7512     movl $SYS_exec, %eax
7513     int $_SYSCALL
7514
7515 # for(;;) exit();
7516 exit:
7517     movl $SYS_exit, %eax
7518     int $_SYSCALL
7519     jmp exit
7520
7521 # char init[] = "/init\0";
7522 init:
7523     .string "/init\0"
7524
7525 # char *argv[] = { init, 0 };
7526 .p2align 2
7527 argv:
7528     .long init
7529     .long 0
7530
7531
7532
7533
7534
7535
7536
7537
7538
7539
7540
7541
7542
7543
7544
7545
7546
7547
7548
7549

```

```

7550 #include "syscall.h"
7551 #include "traps.h"
7552
7553 #define SYSCALL(name) \
7554     .globl name; \
7555     name: \
7556     movl $SYS_ ## name, %eax; \
7557     int $_SYSCALL; \
7558     ret
7559
7560 SYSCALL(fork)
7561 SYSCALL(exit)
7562 SYSCALL(wait)
7563 SYSCALL(pipe)
7564 SYSCALL(read)
7565 SYSCALL(write)
7566 SYSCALL(close)
7567 SYSCALL(kill)
7568 SYSCALL(exec)
7569 SYSCALL(open)
7570 SYSCALL(mknod)
7571 SYSCALL(unlink)
7572 SYSCALL(fstat)
7573 SYSCALL(link)
7574 SYSCALL(mkdir)
7575 SYSCALL(chdir)
7576 SYSCALL(dup)
7577 SYSCALL(getpid)
7578 SYSCALL(sbrk)
7579 SYSCALL(sleep)
7580 SYSCALL(uptime)
7581
7582
7583
7584
7585
7586
7587
7588
7589
7590
7591
7592
7593
7594
7595
7596
7597
7598
7599

```

```

7600 // init: The initial user-level program
7601
7602 #include "types.h"
7603 #include "stat.h"
7604 #include "user.h"
7605 #include "fcntl.h"
7606
7607 char *argv[] = { "sh", 0 };
7608
7609 int
7610 main(void)
7611 {
7612     int pid, wpid;
7613
7614     if(open("console", O_RDWR) < 0){
7615         mknod("console", 1, 1);
7616         open("console", O_RDWR);
7617     }
7618     dup(0); // stdout
7619     dup(0); // stderr
7620
7621     for(;;){
7622         printf(1, "init: starting sh\n");
7623         pid = fork();
7624         if(pid < 0){
7625             printf(1, "init: fork failed\n");
7626             exit();
7627         }
7628         if(pid == 0){
7629             exec("sh", argv);
7630             printf(1, "init: exec sh failed\n");
7631             exit();
7632         }
7633         while((wpid=wait()) >= 0 && wpid != pid)
7634             printf(1, "zombie!\n");
7635     }
7636 }
7637
7638
7639
7640
7641
7642
7643
7644
7645
7646
7647
7648
7649

```

```

7650 // Shell.
7651
7652 #include "types.h"
7653 #include "user.h"
7654 #include "fcntl.h"
7655
7656 // Parsed command representation
7657 #define EXEC 1
7658 #define REDIR 2
7659 #define PIPE 3
7660 #define LIST 4
7661 #define BACK 5
7662
7663 #define MAXARGS 10
7664
7665 struct cmd {
7666     int type;
7667 };
7668
7669 struct execcmd {
7670     int type;
7671     char *argv[MAXARGS];
7672     char *eargv[MAXARGS];
7673 };
7674
7675 struct redircmd {
7676     int type;
7677     struct cmd *cmd;
7678     char *file;
7679     char *efile;
7680     int mode;
7681     int fd;
7682 };
7683
7684 struct pipecmd {
7685     int type;
7686     struct cmd *left;
7687     struct cmd *right;
7688 };
7689
7690 struct listcmd {
7691     int type;
7692     struct cmd *left;
7693     struct cmd *right;
7694 };
7695
7696 struct backcmd {
7697     int type;
7698     struct cmd *cmd;
7699 };

```

```

7700 int fork1(void); // Fork but panics on failure.
7701 void panic(char*);
7702 struct cmd *parsecmd(char*);
7703
7704 // Execute cmd. Never returns.
7705 void
7706 runcmd(struct cmd *cmd)
7707 {
7708     int p[2];
7709     struct backcmd *bcmd;
7710     struct execcmd *ecmd;
7711     struct listcmd *lcmd;
7712     struct pipecmd *pcmd;
7713     struct redircmd *rcmd;
7714
7715     if(cmd == 0)
7716         exit();
7717
7718     switch(cmd->type){
7719     default:
7720         panic("runcmd");
7721
7722     case EXEC:
7723         ecmd = (struct execcmd*)cmd;
7724         if(ecmd->argv[0] == 0)
7725             exit();
7726         exec(ecmd->argv[0], ecmd->argv);
7727         printf(2, "exec %s failed\n", ecmd->argv[0]);
7728         break;
7729
7730     case REDIR:
7731         rcmd = (struct redircmd*)cmd;
7732         close(rcmd->fd);
7733         if(open(rcmd->file, rcmd->mode) < 0){
7734             printf(2, "open %s failed\n", rcmd->file);
7735             exit();
7736         }
7737         runcmd(rcmd->cmd);
7738         break;
7739
7740     case LIST:
7741         lcmd = (struct listcmd*)cmd;
7742         if(fork1() == 0)
7743             runcmd(lcmd->left);
7744         wait();
7745         runcmd(lcmd->right);
7746         break;
7747
7748
7749

```

```

7750     case PIPE:
7751         pcmd = (struct pipecmd*)cmd;
7752         if(pipe(p) < 0)
7753             panic("pipe");
7754         if(fork1() == 0){
7755             close(1);
7756             dup(p[1]);
7757             close(p[0]);
7758             close(p[1]);
7759             runcmd(pcmd->left);
7760         }
7761         if(fork1() == 0){
7762             close(0);
7763             dup(p[0]);
7764             close(p[0]);
7765             close(p[1]);
7766             runcmd(pcmd->right);
7767         }
7768         close(p[0]);
7769         close(p[1]);
7770         wait();
7771         wait();
7772         break;
7773
7774     case BACK:
7775         bcmd = (struct backcmd*)cmd;
7776         if(fork1() == 0)
7777             runcmd(bcmd->cmd);
7778         break;
7779     }
7780     exit();
7781 }
7782
7783 int
7784 getcmd(char *buf, int nbuf)
7785 {
7786     printf(2, "$ ");
7787     memset(buf, 0, nbuf);
7788     gets(buf, nbuf);
7789     if(buf[0] == 0) // EOF
7790         return -1;
7791     return 0;
7792 }
7793
7794
7795
7796
7797
7798
7799

```

```

7800 int
7801 main(void)
7802 {
7803     static char buf[100];
7804     int fd;
7805
7806     // Assumes three file descriptors open.
7807     while((fd = open("console", O_RDWR)) >= 0){
7808         if(fd >= 3){
7809             close(fd);
7810             break;
7811         }
7812     }
7813
7814     // Read and run input commands.
7815     while(getcmd(buf, sizeof(buf)) >= 0){
7816         if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
7817             // Clumsy but will have to do for now.
7818             // Chdir has no effect on the parent if run in the child.
7819             buf[strlen(buf)-1] = 0; // chop \n
7820             if(chdir(buf+3) < 0)
7821                 printf(2, "cannot cd %s\n", buf+3);
7822             continue;
7823         }
7824         if(fork1() == 0)
7825             runcmd(parsecmd(buf));
7826         wait();
7827     }
7828     exit();
7829 }
7830
7831 void
7832 panic(char *s)
7833 {
7834     printf(2, "%s\n", s);
7835     exit();
7836 }
7837
7838 int
7839 fork1(void)
7840 {
7841     int pid;
7842
7843     pid = fork();
7844     if(pid == -1)
7845         panic("fork");
7846     return pid;
7847 }
7848
7849

```

```

7850 // Constructors
7851
7852 struct cmd*
7853 execcmd(void)
7854 {
7855     struct execcmd *cmd;
7856
7857     cmd = malloc(sizeof(*cmd));
7858     memset(cmd, 0, sizeof(*cmd));
7859     cmd->type = EXEC;
7860     return (struct cmd*)cmd;
7861 }
7862
7863 struct cmd*
7864 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
7865 {
7866     struct redircmd *cmd;
7867
7868     cmd = malloc(sizeof(*cmd));
7869     memset(cmd, 0, sizeof(*cmd));
7870     cmd->type = REDIR;
7871     cmd->cmd = subcmd;
7872     cmd->file = file;
7873     cmd->efile = efile;
7874     cmd->mode = mode;
7875     cmd->fd = fd;
7876     return (struct cmd*)cmd;
7877 }
7878
7879 struct cmd*
7880 pipecmd(struct cmd *left, struct cmd *right)
7881 {
7882     struct pipecmd *cmd;
7883
7884     cmd = malloc(sizeof(*cmd));
7885     memset(cmd, 0, sizeof(*cmd));
7886     cmd->type = PIPE;
7887     cmd->left = left;
7888     cmd->right = right;
7889     return (struct cmd*)cmd;
7890 }
7891
7892
7893
7894
7895
7896
7897
7898
7899

```

```

7900 struct cmd*
7901 listcmd(struct cmd *left, struct cmd *right)
7902 {
7903     struct listcmd *cmd;
7904
7905     cmd = malloc(sizeof(*cmd));
7906     memset(cmd, 0, sizeof(*cmd));
7907     cmd->type = LIST;
7908     cmd->left = left;
7909     cmd->right = right;
7910     return (struct cmd*)cmd;
7911 }
7912
7913 struct cmd*
7914 backcmd(struct cmd *subcmd)
7915 {
7916     struct backcmd *cmd;
7917
7918     cmd = malloc(sizeof(*cmd));
7919     memset(cmd, 0, sizeof(*cmd));
7920     cmd->type = BACK;
7921     cmd->cmd = subcmd;
7922     return (struct cmd*)cmd;
7923 }
7924
7925
7926
7927
7928
7929
7930
7931
7932
7933
7934
7935
7936
7937
7938
7939
7940
7941
7942
7943
7944
7945
7946
7947
7948
7949

```

```

7950 // Parsing
7951
7952 char whitespace[] = " \t\r\n\v";
7953 char symbols[] = "<|>&()";
7954
7955 int
7956 gettoken(char **ps, char *es, char **q, char **eq)
7957 {
7958     char *s;
7959     int ret;
7960
7961     s = *ps;
7962     while(s < es && strchr(whitespace, *s))
7963         s++;
7964     if(q)
7965         *q = s;
7966     ret = *s;
7967     switch(*s){
7968     case 0:
7969         break;
7970     case '|':
7971     case '(':
7972     case ')':
7973     case ';':
7974     case '&':
7975     case '<':
7976         s++;
7977         break;
7978     case '>':
7979         s++;
7980         if(*s == '>'){
7981             ret = '+';
7982             s++;
7983         }
7984         break;
7985     default:
7986         ret = 'a';
7987         while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
7988             s++;
7989         break;
7990     }
7991     if(eq)
7992         *eq = s;
7993
7994     while(s < es && strchr(whitespace, *s))
7995         s++;
7996     *ps = s;
7997     return ret;
7998 }
7999

```

```

8000 int
8001 peek(char **ps, char *es, char *toks)
8002 {
8003     char *s;
8004
8005     s = *ps;
8006     while(s < es && strchr(whitespace, *s))
8007         s++;
8008     *ps = s;
8009     return *s && strchr(toks, *s);
8010 }
8011
8012 struct cmd *parseline(char**, char*);
8013 struct cmd *parsepipe(char**, char*);
8014 struct cmd *parseexec(char**, char*);
8015 struct cmd *nulterminate(struct cmd*);
8016
8017 struct cmd*
8018 parsecmd(char *s)
8019 {
8020     char *es;
8021     struct cmd *cmd;
8022
8023     es = s + strlen(s);
8024     cmd = parseline(&s, es);
8025     peek(&s, es, "");
8026     if(s != es){
8027         printf(2, "leftovers: %s\n", s);
8028         panic("syntax");
8029     }
8030     nulterminate(cmd);
8031     return cmd;
8032 }
8033
8034 struct cmd*
8035 parseline(char **ps, char *es)
8036 {
8037     struct cmd *cmd;
8038
8039     cmd = parsepipe(ps, es);
8040     while(peek(ps, es, "&")){
8041         gettoken(ps, es, 0, 0);
8042         cmd = backcmd(cmd);
8043     }
8044     if(peek(ps, es, ";")){
8045         gettoken(ps, es, 0, 0);
8046         cmd = listcmd(cmd, parseline(ps, es));
8047     }
8048     return cmd;
8049 }

```

```

8050 struct cmd*
8051 parsepipe(char **ps, char *es)
8052 {
8053     struct cmd *cmd;
8054
8055     cmd = parseexec(ps, es);
8056     if(peek(ps, es, "|")){
8057         gettoken(ps, es, 0, 0);
8058         cmd = pipecmd(cmd, parsepipe(ps, es));
8059     }
8060     return cmd;
8061 }
8062
8063 struct cmd*
8064 parseredirs(struct cmd *cmd, char **ps, char *es)
8065 {
8066     int tok;
8067     char *q, *eq;
8068
8069     while(peek(ps, es, "<>")){
8070         tok = gettoken(ps, es, 0, 0);
8071         if(gettoken(ps, es, &q, &eq) != 'a')
8072             panic("missing file for redirection");
8073         switch(tok){
8074             case '<':
8075                 cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
8076                 break;
8077             case '>':
8078                 cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8079                 break;
8080             case '+': // >>
8081                 cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8082                 break;
8083         }
8084     }
8085     return cmd;
8086 }
8087
8088
8089
8090
8091
8092
8093
8094
8095
8096
8097
8098
8099

```



```

8100 struct cmd*
8101 parseblock(char **ps, char *es)
8102 {
8103     struct cmd *cmd;
8104
8105     if(!peek(ps, es, "("))
8106         panic("parseblock");
8107     gettoken(ps, es, 0, 0);
8108     cmd = parseline(ps, es);
8109     if(!peek(ps, es, ")"))
8110         panic("syntax - missing )");
8111     gettoken(ps, es, 0, 0);
8112     cmd = parseredirs(cmd, ps, es);
8113     return cmd;
8114 }
8115
8116 struct cmd*
8117 parseexec(char **ps, char *es)
8118 {
8119     char *q, *eq;
8120     int tok, argc;
8121     struct execcmd *cmd;
8122     struct cmd *ret;
8123
8124     if(peek(ps, es, "("))
8125         return parseblock(ps, es);
8126
8127     ret = execcmd();
8128     cmd = (struct execcmd*)ret;
8129
8130     argc = 0;
8131     ret = parseredirs(ret, ps, es);
8132     while(!peek(ps, es, "|&");){
8133         if((tok=gettoken(ps, es, &q, &eq)) == 0)
8134             break;
8135         if(tok != 'a')
8136             panic("syntax");
8137         cmd->argv[argc] = q;
8138         cmd->eargv[argc] = eq;
8139         argc++;
8140         if(argc >= MAXARGS)
8141             panic("too many args");
8142         ret = parseredirs(ret, ps, es);
8143     }
8144     cmd->argv[argc] = 0;
8145     cmd->eargv[argc] = 0;
8146     return ret;
8147 }
8148
8149

```

```

8150 // NUL-terminate all the counted strings.
8151 struct cmd*
8152 nulterminate(struct cmd *cmd)
8153 {
8154     int i;
8155     struct backcmd *bcmd;
8156     struct execcmd *ecmd;
8157     struct listcmd *lcmd;
8158     struct pipecmd *pcmd;
8159     struct redircmd *rcmd;
8160
8161     if(cmd == 0)
8162         return 0;
8163
8164     switch(cmd->type){
8165     case EXEC:
8166         ecmd = (struct execcmd*)cmd;
8167         for(i=0; ecmd->argv[i]; i++)
8168             *ecmd->eargv[i] = 0;
8169         break;
8170
8171     case REDIR:
8172         rcmd = (struct redircmd*)cmd;
8173         nulterminate(rcmd->cmd);
8174         *rcmd->efile = 0;
8175         break;
8176
8177     case PIPE:
8178         pcmd = (struct pipecmd*)cmd;
8179         nulterminate(pcmd->left);
8180         nulterminate(pcmd->right);
8181         break;
8182
8183     case LIST:
8184         lcmd = (struct listcmd*)cmd;
8185         nulterminate(lcmd->left);
8186         nulterminate(lcmd->right);
8187         break;
8188
8189     case BACK:
8190         bcmd = (struct backcmd*)cmd;
8191         nulterminate(bcmd->cmd);
8192         break;
8193     }
8194     return cmd;
8195 }
8196
8197
8198
8199

```

```

8200 #include "asm.h"
8201 #include "memlayout.h"
8202 #include "mmu.h"
8203
8204 # Start the first CPU: switch to 32-bit protected mode, jump into C.
8205 # The BIOS loads this code from the first sector of the hard disk into
8206 # memory at physical address 0x7c00 and starts executing in real mode
8207 # with %cs=0 %ip=7c00.
8208
8209 .code16                # Assemble for 16-bit mode
8210 .globl start
8211 start:
8212     cli                # BIOS enabled interrupts; disable
8213
8214 # Set up the important data segment registers (DS, ES, SS).
8215     xorw    %ax,%ax    # Segment number zero
8216     movw   %ax,%ds    # -> Data Segment
8217     movw   %ax,%es    # -> Extra Segment
8218     movw   %ax,%ss    # -> Stack Segment
8219
8220 # Physical address line A20 is tied to zero so that the first PCs
8221 # with 2 MB would run software that assumed 1 MB. Undo that.
8222 seta20.1:
8223     inb    $0x64,%al    # Wait for not busy
8224     testb  $0x2,%al
8225     jnz   seta20.1
8226
8227     movb  $0xd1,%al    # 0xd1 -> port 0x64
8228     outb  %al,$0x64
8229
8230 seta20.2:
8231     inb    $0x64,%al    # Wait for not busy
8232     testb  $0x2,%al
8233     jnz   seta20.2
8234
8235     movb  $0xdf,%al    # 0xdf -> port 0x60
8236     outb  %al,$0x60
8237
8238 # Switch from real to protected mode. Use a bootstrap GDT that makes
8239 # virtual addresses map directly to physical addresses so that the
8240 # effective memory map doesn't change during the transition.
8241     lgdt  gdtdesc
8242     movl  %cr0,%eax
8243     orl  $CR0_PE,%eax
8244     movl  %eax,%cr0
8245
8246
8247
8248
8249

```

```

8250 # Complete transition to 32-bit protected mode by using long jmp
8251 # to reload %cs and %eip. The segment descriptors are set up with no
8252 # translation, so that the mapping is still the identity mapping.
8253     ljmp  $(SEG_KCODE<<3), $start32
8254
8255 .code32 # Tell assembler to generate 32-bit code now.
8256 start32:
8257 # Set up the protected-mode data segment registers
8258     movw  $(SEG_KDATA<<3), %ax    # Our data segment selector
8259     movw  %ax,%ds                # -> DS: Data Segment
8260     movw  %ax,%es                # -> ES: Extra Segment
8261     movw  %ax,%ss                # -> SS: Stack Segment
8262     movw  $0,%ax                 # Zero segments not ready for use
8263     movw  %ax,%fs                # -> FS
8264     movw  %ax,%gs                # -> GS
8265
8266 # Set up the stack pointer and call into C.
8267     movl  $start,%esp
8268     call  bootmain
8269
8270 # If bootmain returns (it shouldn't), trigger a Bochs
8271 # breakpoint if running under Bochs, then loop.
8272     movw  $0x8a00,%ax            # 0x8a00 -> port 0x8a00
8273     movw  %ax,%dx
8274     outw  %ax,%dx
8275     movw  $0x8ae0,%ax           # 0x8ae0 -> port 0x8a00
8276     outw  %ax,%dx
8277 spin:
8278     jmp   spin
8279
8280 # Bootstrap GDT
8281     .p2align 2                # force 4 byte alignment
8282 gdt:
8283     SEG_NULLASM                # null seg
8284     SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg
8285     SEG_ASM(STA_W, 0x0, 0xffffffff)      # data seg
8286
8287 gdtdesc:
8288     .word  (gdtdesc - gdt - 1)    # sizeof(gdt) - 1
8289     .long  gdt                    # address gdt
8290
8291
8292
8293
8294
8295
8296
8297
8298
8299

```

```

8300 // Boot loader.
8301 //
8302 // Part of the boot sector, along with bootasm.S, which calls bootmain().
8303 // bootasm.S has put the processor into protected 32-bit mode.
8304 // bootmain() loads an ELF kernel image from the disk starting at
8305 // sector 1 and then jumps to the kernel entry routine.
8306
8307 #include "types.h"
8308 #include "elf.h"
8309 #include "x86.h"
8310 #include "memlayout.h"
8311
8312 #define SECTSIZE 512
8313
8314 void readseg(uchar*, uint, uint);
8315
8316 void
8317 bootmain(void)
8318 {
8319     struct elfhdr *elf;
8320     struct proghdr *ph, *eph;
8321     void (*entry)(void);
8322     uchar* pa;
8323
8324     elf = (struct elfhdr*)0x10000; // scratch space
8325
8326     // Read 1st page off disk
8327     readseg((uchar*)elf, 4096, 0);
8328
8329     // Is this an ELF executable?
8330     if(elf->magic != ELF_MAGIC)
8331         return; // let bootasm.S handle error
8332
8333     // Load each program segment (ignores ph flags).
8334     ph = (struct proghdr*)((uchar*)elf + elf->phoff);
8335     eph = ph + elf->phnum;
8336     for(; ph < eph; ph++){
8337         pa = (uchar*)ph->paddr;
8338         readseg(pa, ph->filesz, ph->off);
8339         if(ph->memsz > ph->filesz)
8340             stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
8341     }
8342
8343     // Call the entry point from the ELF header.
8344     // Does not return!
8345     entry = (void(*) (void))(elf->entry);
8346     entry();
8347 }
8348
8349

```

```

8350 void
8351 waitdisk(void)
8352 {
8353     // Wait for disk ready.
8354     while((inb(0x1F7) & 0xC0) != 0x40)
8355         ;
8356 }
8357
8358 // Read a single sector at offset into dst.
8359 void
8360 readsect(void *dst, uint offset)
8361 {
8362     // Issue command.
8363     waitdisk();
8364     outb(0x1F2, 1); // count = 1
8365     outb(0x1F3, offset);
8366     outb(0x1F4, offset >> 8);
8367     outb(0x1F5, offset >> 16);
8368     outb(0x1F6, (offset >> 24) | 0xE0);
8369     outb(0x1F7, 0x20); // cmd 0x20 - read sectors
8370
8371     // Read data.
8372     waitdisk();
8373     insl(0x1F0, dst, SECTSIZE/4);
8374 }
8375
8376 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
8377 // Might copy more than asked.
8378 void
8379 readseg(uchar* pa, uint count, uint offset)
8380 {
8381     uchar* epa;
8382
8383     epa = pa + count;
8384
8385     // Round down to sector boundary.
8386     pa -= offset % SECTSIZE;
8387
8388     // Translate from bytes to sectors; kernel starts at sector 1.
8389     offset = (offset / SECTSIZE) + 1;
8390
8391     // If this is too slow, we could read lots of sectors at a time.
8392     // We'd write more to memory than asked, but it doesn't matter --
8393     // we load in increasing order.
8394     for(; pa < epa; pa += SECTSIZE, offset++)
8395         readsect(pa, offset);
8396 }
8397
8398
8399

```