

NOTE: we have stopped maintaining the x86 version of xv6, and switched our efforts to the RISC-V version (<https://github.com/mit-pdos/xv6-riscv.git>)

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 <https://pdos.csail.mit.edu/6.828/>, which provides pointers to on-line resources for v6.

xv6 borrows code from the following sources:

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

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

We are also grateful for the bug reports and patches contributed by Silas Boyd-Wickizer, Anton Burtsev, Cody Cutler, Mike CAT, Tej Chajed, eyalz800, Nelson Elhage, Saar Ettinger, Alice Ferrazzi, Nathaniel Filardo, Peter Froehlich, Yakir Goaron, Shivam Handa, Bryan Henry, Jim Huang, Alexander Kapshuk, Anders Kaseorg, kehao95, Wolfgang Keller, Eddie Kohler, Austin Liew, Imbar Marinescu, Yandong Mao, Matan Shabtay, Hitoshi Mitake, Carmi Merimovich, Mark Morrissey, mtasm, Joel Nider, Greg Price, Ayan Shafqat, Eldar Sehayek, Yongming Shen, Cam Tenny, tyfkda, Rafael Ubal, Warren Toomey, Stephen Tu, Pablo Ventura, Xi Wang, Keiichi Watanabe, Nicolas Wolovick, wxdao, Grant Wu, Jindong Zhang, Icenowy Zheng, and Zou Chang Wei.

The code in the files that constitute xv6 is
Copyright 2006-2018 Frans Kaashoek, Robert Morris, and Russ Cox.

ERROR REPORTS

We don't process error reports (see note on top of this file).

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 <https://pdos.csail.mit.edu/6.828/>). Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC

simulator and run "make qemu".

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	
01 types.h	32 traps.h	# string operations
01 param.h	32 vectors.pl	69 string.c
02 memlayout.h	33 trapasm.S	
02 defs.h	33 trap.c	# low-level hardware
04 x86.h	35 syscall.h	70 mp.h
06 asm.h	35 syscall.c	72 mp.c
07 mmu.h	37 sysproc.c	73 lapic.c
09 elf.h		76 ioapic.c
09 date.h	# file system	77 kbd.h
	38 buf.h	78 kbd.c
# entering xv6	39 sleeplock.h	79 console.c
10 entry.S	39 fcntl.h	83 uart.c
11 entryother.S	40 stat.h	
12 main.c	40 fs.h	# user-level
	41 file.h	84 initcode.S
# locks	42 ide.c	84 usys.S
15 spinlock.h	44 bio.c	85 init.c
15 spinlock.c	46 sleeplock.c	85 sh.c
	47 log.c	
# processes	49 fs.c	# bootloader
17 vm.c	58 file.c	91 bootasm.S
23 proc.h	60 sysfile.c	92 bootmain.c
24 proc.c	66 exec.c	
30 swtch.S		# link
31 kalloc.c	# pipes	93 kernel.ld
	67 pipe.c	

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

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

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

acquire 1574	3862 4294 4318 4323 4360
0380 1574 1578 2478 2548	4378 4486 4519 4939
2614 2649 2677 2769 2830	begin_op 4828
2891 2906 2966 2979 3175	0336 2644 4828 5933 6024
3192 3416 3822 3842 4309	6210 6311 6411 6456 6473
4365 4470 4533 4624 4636	6507 6621
4655 4830 4857 4876 4931	bfree 5052
5258 5291 5362 5375 5880	5052 5464 5474 5477
5904 5918 6813 6834 6855	bget 4466
8010 8181 8228 8264	4466 4496 4506
acquiresleep 4622	binit 4438
0389 4477 4492 4622 5311	0263 1230 4438
5360	bmap 5410
allocproc 2473	5152 5410 5436 5520 5570
2473 2525 2587	bootmain 9217
allocvm 1927	9168 9217
0430 1927 1941 1947 2565	BPB 4107
6651 6665	4107 4110 5022 5024 5058
alltraps 3304	bread 4502
3259 3267 3280 3285 3303	0264 4502 4777 4778 4790
3304	4806 4890 4891 4985 5006
ALT 7710	5023 5057 5211 5235 5314
7710 7738 7740	5426 5470 5520 5570
argfd 6071	breelse 4526
6071 6123 6138 6157 6168	0265 4526 4529 4781 4782
6181	4797 4814 4894 4895 4987
argint 3602	5009 5029 5034 5064 5217
0404 3602 3616 3632 3783	5220 5244 5322 5432 5476
3806 3820 6076 6138 6157	5523 5574
6408 6475 6476 6532	BSIZE 4055
argptr 3611	3859 4055 4074 4101 4107
0405 3611 6138 6157 6181	4280 4296 4319 4758 4779
6557	4892 5007 5520 5521 5522
argstr 3629	5566 5570 5571 5572
0406 3629 6207 6308 6408	buf 3850
6457 6474 6508 6532	0250 0264 0265 0266 0308
__attribute__ 1305	0335 2120 2123 2132 2134
0272 0367 1209 1305	3850 3856 3857 3858 4213
BACK 8561	4231 4234 4274 4306 4354
8561 8674 8820 9089	4356 4359 4426 4430 4434
backcmd 8596 8814	4440 4453 4465 4468 4501
8596 8609 8675 8814 8816	4504 4515 4526 4706 4777
8942 9055 9090	4778 4790 4791 4797 4806
BACKSPACE 8100	4807 4813 4814 4890 4891
8100 8117 8159 8192 8198	4922 4970 4983 5004 5019
ballocc 5016	5054 5207 5232 5305 5413
5016 5036 5417 5425 5429	5459 5506 5556 7930 7941
BBLOCK 4110	7945 7948 8168 8190 8204
4110 5023 5057	8238 8259 8266 8684 8687
B_DIRTY 3862	8688 8689 8703 8715 8716

```

      8718 8719 8720 8724
B_VALID 3861
      3861 4322 4360 4378 4507
bwrite 4515
      0266 4515 4518 4780 4813
      4893
bzero 5002
      5002 5030
C 7731 8174
      7731 7779 7804 7805 7806
      7807 7808 7810 8174 8184
      8188 8195 8206 8239
CAPSLOCK 7712
      7712 7745 7886
cgaputc 8105
      8105 8163
clearpteu 2022
      0439 2022 2028 6667
cli 0557
      0557 0559 1124 1672 8060
      8154 9112
cmd 8565
      8565 8577 8586 8587 8592
      8593 8598 8602 8606 8615
      8618 8623 8631 8637 8641
      8651 8675 8677 8752 8755
      8757 8758 8759 8760 8763
      8764 8766 8768 8769 8770
      8771 8772 8773 8774 8775
      8776 8779 8780 8782 8784
      8785 8786 8787 8788 8789
      8800 8801 8803 8805 8806
      8807 8808 8809 8810 8813
      8814 8816 8818 8819 8820
      8821 8822 8912 8913 8914
      8915 8917 8921 8924 8930
      8931 8934 8937 8939 8942
      8946 8948 8950 8953 8955
      8958 8960 8963 8964 8975
      8978 8981 8985 9000 9003
      9008 9012 9013 9016 9021
      9022 9028 9037 9038 9044
      9045 9051 9052 9061 9064
      9066 9072 9073 9078 9084
      9090 9091 9094
CMOS_PORT 7477
      7477 7491 7492 7534
cmos_read 7532
      7532 7543 7544 7545 7546
      7547 7548 7557 7564
CMOS_RETURN 7478
      7478 7537
CMOS_STATA 7520
      7520 7564
CMOS_STATB 7521
      7521 7557
cmostime 7552
      0325 7552
CMOS_UIP 7522
      7522 7564
COM1 8314
      8314 8324 8327 8328 8329
      8330 8331 8332 8335 8341
      8342 8357 8359 8367 8369
commit 4901
      4753 4875 4901
CONSOLE 4186
      4186 8278 8279
consoleinit 8274
      0269 1226 8274
consoleintr 8177
      0271 7898 8177 8375
consoleread 8221
      8221 8279
consolewrite 8259
      8259 8278
consputc 8151
      7917 7948 8018 8036 8039
      8043 8044 8151 8192 8198
      8205 8266
context 2326
      0251 0377 2303 2326 2345
      2509 2510 2511 2512 2781
      2822 3028
CONV 7573
      7573 7574 7575 7576 7577
      7578 7579 7580
copyout 2118
      0438 2118 6675 6686
copyvum 2035
      0435 2035 2046 2048 2592
cprintf 8002
      0270 1254 1941 1947 3026
      3030 3032 3440 3453 3458
      3710 5151 6625 7659 8002
      8063 8064 8065 8068
cpu 2301
      0311 0363 1254 1268 1506
      1566 1590 1608 1656 1717
      2301 2312 2436 2458 2761

```

```

      3440 3453 3458 7213
cpuid 2430
      0358 1254 1723 2430 3415
      3441 3454 3461
CR0_PE 0707
      0707 1137 1170 9143
CR0_PG 0709
      0709 1054 1170
CR0_WP 0708
      0708 1054 1170
CR4_PSE 0711
      0711 1047 1163
create 6357
      6357 6376 6389 6393 6414
      6457 6477
CRTPORT 8101
      8101 8110 8111 8112 8113
      8131 8132 8133 8134
CTL 7709
      7709 7735 7739 7885
DAY 7527
      7527 7546
deallocvm 1961
      0431 1942 1948 1961 2009
      2568
DEVSPACE 0204
      0204 1813 1826
devsw 4179
      4179 4184 5509 5511 5559
      5561 5862 8278 8279
dinode 4078
      4078 4101 5208 5212 5233
      5236 5306 5315
dirent 4115
      4115 5614 5655 6255 6304
dirlink 5652
      0288 5652 5667 5675 6230
      6388 6392 6393
dirlookup 5611
      0289 5611 5617 5621 5659
      5775 6323 6366
DIRSIZ 4113
      4113 4117 5605 5672 5728
      5729 5792 6204 6305 6360
DPL_USER 0761
      0761 1726 1727 2533 2534
      3373 3468 3478
EOESC 7716
      7716 7870 7874 7875 7877
      7880
elfhdr 0905
      0905 6615 9219 9224
ELF_MAGIC 0902
      0902 6634 9230
ELF_PROG_LOAD 0936
      0936 6645
end_op 4853
      0337 2646 4853 5935 6029
      6212 6219 6237 6246 6313
      6347 6352 6416 6421 6427
      6436 6440 6458 6462 6478
      6482 6509 6515 6520 6624
      6659 6710
entry 1044
      0911 1040 1043 1044 3252
      3253 6699 7071 9221 9245
      9246
EOI 7366
      7366 7441 7467
ERROR 7387
      7387 7434
ESR 7369
      7369 7437 7438
exec 6610
      0275 6548 6610 6625 8468
      8529 8530 8626 8627
EXEC 8557
      8557 8622 8759 9065
execcmd 8569 8753
      8569 8610 8623 8753 8755
      9021 9027 9028 9056 9066
exit 2627
      0359 2627 2665 3405 3409
      3469 3479 3768 8417 8420
      8461 8526 8531 8616 8625
      8635 8680 8727 8734
EXTMEM 0202
      0202 0208 1810
fdalloc 6103
      6103 6125 6432 6562
fetchint 3567
      0407 3567 3604 6539
fetchstr 3581
      0408 3581 3634 6545
file 4150
      0252 0278 0279 0280 0282
      0283 0284 0351 2348 4150
      4971 5860 5865 5875 5878
      5881 5901 5902 5914 5916
      5952 5965 6002 6065 6071

```

```

6074 6103 6120 6134 6153
6166 6178 6405 6554 6758
6772 7911 8309 8578 8633
8634 8764 8772 8972
filealloc 5876
0278 5876 6432 6778
fileclose 5914
0279 2639 5914 5920 6171
6434 6565 6566 6804 6806
filedup 5902
0280 2607 5902 5906 6127
fileinit 5869
0281 1231 5869
fileread 5965
0282 5965 5980 6140
filestat 5952
0283 5952 6183
filewrite 6002
0284 6002 6034 6039 6159
fill_rtcdade 7541
7541 7563 7566
FL_IF 0704
0704 1674 1681 2441 2537
2819
fork 2580
0360 2580 3762 8460 8523
8525 8742 8744
fork1 8738
8600 8642 8654 8661 8676
8723 8738
forkret 2853
2417 2512 2853
freerange 3151
3111 3135 3141 3151
freevm 2003
0432 1831 2003 2008 2062
2690 6702 6707
FSSIZE 0162
0162 4278
gatedesc 0855
0523 0526 0855 3361
getcallerpcs 1627
0381 1591 1627 3028 8066
getcmd 8684
8684 8715
gettoken 8856
8856 8941 8945 8957 8970
8971 9007 9011 9033
growproc 2558
0361 2558 3809

```

```

havedisk1 4233
4233 4263 4362
holding 1652
0382 1577 1604 1652 2813
holdingsleep 4651
0391 4358 4517 4528 4651
5333
HOURS 7526
7526 7545
ialloc 5204
0290 5204 5222 6375 6376
IBLOCK 4104
4104 5211 5235 5314
ICRHI 7380
7380 7444 7502 7514
ICRLO 7370
7370 7445 7446 7503 7505
7515
ID 7363
7363 7404 7459
IDE_BSY 4216
4216 4242
IDE_CMD_RDMUL 4223
4223 4282
IDE_CMD_READ 4221
4221 4282
IDE_CMD_WRITE 4222
4222 4283
IDE_CMD_WRMUL 4224
4224 4283
IDE_DF 4218
4218 4244
IDE_DRDY 4217
4217 4242
IDE_ERR 4219
4219 4244
ideinit 4251
0306 1232 4251
ideintr 4304
0307 3424 4304
idelock 4230
4230 4255 4309 4312 4330
4365 4379 4383
iderw 4354
0308 4354 4359 4361 4363
4508 4520
idestart 4274
4234 4274 4277 4285 4328
4375
idewait 4238

```

```

4238 4257 4287 4318
idtinit 3379
0415 1255 3379
idup 5289
0291 2608 5289 5762
iget 5254
5157 5218 5254 5274 5629
5760
iinit 5142
0292 2864 5142
ilock 5303
0293 5303 5309 5325 5765
5955 5974 6025 6216 6229
6242 6317 6325 6364 6368
6378 6424 6512 6628 8233
8253 8268
inb 0453
0453 4242 4262 7346 7537
7864 7867 8111 8113 8335
8341 8342 8357 8367 8369
9123 9131 9254
initlock 1562
0383 1562 2425 3133 3375
4255 4442 4615 4762 5146
5871 6786 8276
initlog 4756
0334 2865 4756 4759
initsleeplock 4613
0392 4456 4613 5148
initvm 1886
0433 1886 1891 2530
inode 4162
0253 0288 0289 0290 0291
0293 0294 0295 0296 0297
0299 0300 0301 0302 0303
0434 1903 2349 4156 4162
4180 4181 4974 5138 5148
5157 5203 5230 5253 5256
5262 5288 5289 5303 5331
5358 5382 5410 5456 5488
5503 5553 5610 5611 5652
5656 5754 5757 5789 5800
6205 6252 6303 6356 6359
6406 6454 6469 6504 6616
8221 8259
INPUT_BUF 8166
8166 8168 8190 8202 8204
8206 8238
insl 0462
0462 0464 4319 9273

```

```

install_trans 4772
4772 4821 4906
INT_DISABLED 7619
7619 7664
ioapic 7627
7308 7325 7326 7624 7627
7636 7637 7643 7644 7655
IOAPIC 7608
7608 7655
ioapicenable 7670
0311 4256 7670 8282 8343
ioapicid 7215
0312 7215 7326 7658 7659
ioapicinit 7651
0313 1225 7651 7659
ioapicread 7634
7634 7656 7657
ioapicwrite 7641
7641 7664 7665 7675 7676
IPB 4101
4101 4104 5212 5236 5315
iput 5358
0294 2645 5358 5385 5660
5783 5934 6235 6519
IRQ_COM1 3233
3233 3434 8343
IRQ_ERROR 3235
3235 7434
IRQ_IDE 3234
3234 3423 3427 4256
IRQ_KBD 3232
3232 3430 8282
IRQ_SPURIOUS 3236
3236 3439 7414
IRQ_TIMER 3231
3231 3414 3474 7421
isdiempty 6252
6252 6259 6329
itrunc 5456
4974 5367 5456
iunlock 5331
0295 5331 5334 5384 5772
5957 5977 6028 6225 6439
6518 8226 8263
iunlockput 5382
0296 5382 5767 5776 5779
6218 6231 6234 6245 6330
6341 6345 6351 6367 6371
6395 6426 6435 6461 6481
6514 6658 6709

```

```

iupdate 5230
  0297 5230 5369 5482 5579
  6224 6244 6339 6344 6382
  6386
kalloc 3187
  0316 1284 1744 1823 1892
  1939 2051 2494 3187 6780
KBDATAP 7704
  7704 7867
kbdgetc 7856
  7856 7898
kbdintr 7896
  0322 3431 7896
KBS_DIB 7703
  7703 7865
KBSTATP 7702
  7702 7864
KERNBASE 0207
  0207 0208 0210 0211 0213
  0214 1310 1634 1810 1932
  2009
KERNLINK 0208
  0208 1811 9310
KEY_DEL 7728
  7728 7769 7791 7815
KEY_DN 7722
  7722 7765 7787 7811
KEY_END 7720
  7720 7768 7790 7814
KEY_HOME 7719
  7719 7768 7790 7814
KEY_INS 7727
  7727 7769 7791 7815
KEY_LF 7723
  7723 7767 7789 7813
KEY_PGDN 7726
  7726 7766 7788 7812
KEY_PGUP 7725
  7725 7766 7788 7812
KEY_RT 7724
  7724 7767 7789 7813
KEY_UP 7721
  7721 7765 7787 7811
kfree 3164
  0317 1949 1977 1979 2013
  2016 2055 2593 2688 3156
  3164 3169 6802 6823
kill 2975
  0362 2975 3459 3785 8467
kinit1 3131
  0318 1219 3131
  kinit2 3139
  0319 1234 3139
  KSTACKSIZE 0151
  0151 1058 1067 1285 1874
  2498
  kvmalloc 1840
  0427 1220 1840
  lapiceoi 7464
  0328 3421 3425 3432 3436
  3442 7464
  lapicid 7455
  0326 2444 7455 8063
  lapicinit 7408
  0329 1222 1245 7408
  lapicstartap 7483
  0330 1289 7483
  lapicw 7401
  7401 7414 7420 7421 7422
  7425 7426 7431 7434 7437
  7438 7441 7444 7445 7451
  7467 7502 7503 7505 7514
  7515
  lcr3 0590
  0590 1855 1879
  lgdt 0512
  0512 0520 1135 1728 9141
  lidt 0526
  0526 0534 3381
  LINT0 7385
  7385 7425
  LINT1 7386
  7386 7426
  LIST 8560
  8560 8640 8807 9083
  listcmd 8590 8801
  8590 8611 8641 8801 8803
  8946 9057 9084
  loaduvm 1903
  0434 1903 1909 1912 6655
  log 4738 4750
  4738 4750 4762 4764 4765
  4766 4776 4777 4778 4790
  4793 4794 4795 4806 4809
  4810 4811 4822 4830 4832
  4833 4834 4836 4838 4839
  4857 4858 4859 4860 4861
  4863 4868 4870 4876 4877
  4878 4879 4889 4890 4891
  4903 4907 4926 4928 4931

```

```

  4932 4933 4936 4937 4938
  4940
  logheader 4733
  4733 4745 4758 4759 4791
  4807
  LOGSIZE 0160
  0160 4735 4834 4926
  log_write 4922
  0335 4922 4929 5008 5028
  5063 5216 5243 5430 5573
  ltr 0538
  0538 0540 1878
  mappages 1760
  1760 1829 1894 1946 2054
  MAXARG 0158
  0158 6528 6614 6672
  MAXARGS 8563
  8563 8571 8572 9040
  MAXFILE 4075
  4075 5566
  MAXOPBLOCKS 0159
  0159 0160 0161 4834 6017
  memcmp 6915
  0395 6915 7237 7288 7567
  memmove 6931
  0396 1275 1895 2053 2132
  4779 4892 4986 5242 5321
  5522 5572 5729 5731 6931
  6954 8126
  memset 6904
  0397 1747 1825 1893 1945
  2511 2532 3172 5007 5214
  6334 6535 6904 8128 8687
  8758 8769 8785 8806 8819
  microdelay 7473
  0331 7473 7504 7506 7516
  7535 8358
  min 4973
  4973 5521 5571
  MINS 7525
  7525 7544
  MONTH 7528
  7528 7547
  mp 7052
  7052 7208 7229 7236 7237
  7238 7255 7260 7264 7265
  7268 7269 7280 7283 7285
  7287 7294 7305 7310 7342
  MPBUS 7102
  7102 7329
  mpconf 7063
  7063 7279 7282 7287 7306
  mpconfig 7280
  7280 7310
  mpenter 1241
  1241 1286
  mpinit 7301
  0341 1221 7301
  mpioapic 7089
  7089 7308 7325 7327
  MPIOAPIC 7103
  7103 7324
  MPIOINTR 7104
  7104 7330
  MPLINTR 7105
  7105 7331
  mpmain 1252
  1209 1236 1246 1252
  mpproc 7078
  7078 7307 7317 7322
  MPPROC 7101
  7101 7316
  mpsearch 7256
  7256 7285
  mpsearch1 7230
  7230 7264 7268 7271
  multiboot_header 1029
  1028 1029
  mycpu 2437
  0363 1256 1278 1590 1656
  1673 1674 1675 1683 1685
  1870 1871 1872 1873 1874
  1877 2431 2437 2442 2461
  2761 2815 2821 2822 2823
  myproc 2457
  0364 2457 2561 2584 2629
  2675 2811 2831 2876 3404
  3406 3408 3451 3460 3462
  3468 3473 3478 3569 3584
  3604 3614 3704 3791 3808
  3825 4629 4656 5762 6078
  6106 6170 6505 6564 6619
  6837 6857 8231
  namecmp 5603
  0298 5603 5624 6320
  namei 5790
  0299 2542 5790 6211 6420
  6508 6623
  nameiparent 5801
  0300 5755 5770 5782 5801

```

```

6227 6312 6362
namex 5755
5755 5793 5803
NBUF 0161
0161 4430 4453
ncpu 7214
1277 2313 2447 4256 7214
7318 7319 7320
NCPU 0152
0152 2312 7213 7318
NDEV 0156
0156 5509 5559 5862
NDIRECT 4073
4073 4075 4084 4174 5415
5420 5424 5425 5462 5469
5470 5477 5478
NELEM 0442
0442 1828 3022 3707 6537
nextpid 2416
2416 2489
NFILE 0154
0154 5865 5881
NINDIRECT 4074
4074 4075 5422 5472
NINODE 0155
0155 5138 5147 5262
NO 7706
7706 7752 7755 7757 7758
7759 7760 7762 7774 7777
7779 7780 7781 7782 7784
7802 7803 7805 7806 7807
7808
NOFILE 0153
0153 2348 2605 2637 6078
6108
NPENTRIES 0791
0791 1306 2010
NPROC 0150
0150 2411 2480 2654 2681
2770 2957 2980 3019
NSEGS 0721
0721 2305
nulterminate 9052
8915 8930 9052 9073 9079
9080 9085 9086 9091
NUMLOCK 7713
7713 7746
O_CREATE 3953
3953 6413 8978 8981
O_RDONLY 3950
3950 6425 8975
O_RDWR 3952
3952 6446 8514 8516 8707
outb 0471
0471 4260 4269 4288 4289
4290 4291 4292 4293 4295
4298 7345 7346 7491 7492
7534 8110 8112 8131 8132
8133 8134 8324 8327 8328
8329 8330 8331 8332 8359
9128 9136 9264 9265 9266
9267 9268 9269
outsl 0483
0483 0485 4296
outw 0477
0477 1180 1182 9174 9176
O_WRONLY 3951
3951 6445 6446 8978 8981
P2V 0211
0211 1219 1234 1274 1742
1826 1918 1978 2012 2053
2111 7234 7262 7287 7493
8102
panic 8055 8731
0272 1578 1605 1682 1684
1771 1827 1863 1865 1867
1891 1909 1912 1977 2008
2028 2046 2048 2442 2451
2529 2634 2665 2814 2816
2818 2820 2879 2882 3169
3455 4277 4279 4285 4359
4361 4363 4496 4518 4529
4759 4860 4927 4929 5036
5061 5222 5274 5309 5325
5334 5436 5617 5621 5667
5675 5906 5920 5980 6034
6039 6259 6328 6336 6376
6389 6393 7311 7340 8013
8055 8063 8123 8601 8620
8653 8731 8744 8928 8972
9006 9010 9036 9041
panicked 7919
7919 8069 8153
parseblock 9001
9001 9006 9025
parsecmd 8918
8602 8724 8918
parseexec 9017
8914 8955 9017
parseline 8935

```

```

8912 8924 8935 8946 9008
parsepipe 8951
8913 8939 8951 8958
parseredirs 8964
8964 9012 9031 9042
PCINT 7384
7384 7431
pde_t 0103
0103 0428 0429 0430 0431
0432 0433 0434 0435 0438
0439 1210 1260 1306 1710
1735 1737 1760 1817 1820
1823 1886 1903 1927 1961
2003 2022 2034 2035 2037
2102 2118 2339 6618
PDX 0782
0782 1740 1973
PDXSHIFT 0796
0782 0788 0796 1310
peek 8901
8901 8925 8940 8944 8956
8969 9005 9009 9024 9032
PGADDR 0788
0788 1973
PGROUNDOWN 0799
0799 1765 1766 2125
PGROUNDUP 0798
0798 1937 1969 3154 6664
PGSIZE 0793
0793 0798 0799 1305 1747
1775 1776 1825 1890 1893
1894 1908 1910 1914 1917
1938 1945 1946 1970 1973
2044 2053 2054 2129 2135
2531 2538 3155 3168 3172
6653 6665 6667
PHYSTOP 0203
0203 1234 1812 1826 1827
3168
pinit 2423
0365 1228 2423
pipe 6762
0254 0352 0353 0354 4155
5931 5972 6009 6762 6774
6780 6786 6790 6794 6811
6830 6851 8463 8652 8653
PIPE 8559
8559 8650 8786 9077
pipealloc 6772
0351 6559 6772
pipeclose 6811
0352 5931 6811
pipecmd 8584 8780
8584 8612 8651 8780 8782
8958 9058 9078
piperead 6851
0353 5972 6851
PIPESIZE 6760
6760 6764 6836 6844 6866
pipewrite 6830
0354 6009 6830
popcli 1679
0386 1622 1657 1679 1682
1684 1880 2463
printint 7927
7927 8026 8030
proc 2337
0255 0364 0369 0436 1205
1558 1706 1860 2309 2337
2343 2406 2411 2414 2456
2459 2462 2472 2475 2480
2522 2561 2583 2584 2629
2630 2654 2673 2675 2681
2760 2762 2770 2777 2786
2811 2876 2955 2957 2977
2980 3015 3019 3355 3459
3555 3569 3584 3614 3704
3757 4207 4608 4966 6061
6106 6505 6604 6619 6754
7211 7307 7317 7319 7914
8311
procdump 3004
0366 3004 8216
proghdr 0924
0924 6617 9220 9234
PTE_ADDR 0807
0807 1742 1913 1975 2012
2049 2111
PTE_FLAGS 0808
0808 2050
PTE_P 0801
0801 1308 1310 1741 1751
1770 1772 1974 2011 2047
2107
PTE_PS 0804
0804 1308 1310
pte_t 0811
0811 1734 1738 1742 1744
1763 1906 1963 2024 2038
2104

```

```

PTE_U 0803          5377 5884 5888 5908 5922
    0803 1751 1894 1946 2029      5928 6822 6825 6838 6847
    2109          6858 6869 8051 8214 8232
PTE_W 0802          8252 8267
    0802 1308 1310 1751 1810      releasesleep 4634
    1812 1813 1894 1946          0390 4531 4634 5336 5373
PTX 0785          ROOTDEV 0157
    0785 1753          0157 2864 2865 5760
PTXSHIFT 0795      ROOTINO 4054
    0785 0788 0795          4054 5760
pushcli 1667      rtcdate 0950
    0385 1576 1655 1667 1869      0256 0325 0950 7541 7552
    2460          7554
rcr2 0582          run 3115
    0582 3454 3461          3011 3115 3116 3122 3166
readeflags 0544      3176 3189 7311
    0544 1671 1681 2441 2819      runcmd 8606
read_head 4788      8606 8620 8637 8643 8645
    4788 4820          8659 8666 8677 8724
readi 5503          RUNNING 2334
    0301 1918 5503 5620 5666      2334 2779 2817 3011 3473
    5975 6258 6259 6632 6643      safestrcpy 6982
readsb 4981          0398 2541 2610 6693 6982
    0287 4763 4981 5150      sb 4977
readsect 9260      0287 4104 4110 4761 4763
    9260 9295          4764 4765 4977 4981 4986
readseg 9279      5022 5023 5024 5057 5150
    9214 9227 9238 9279          5151 5152 5153 5154 5210
recover_from_log 4818      5211 5235 5314 7555 7557
    4752 4767 4818          7559
REDIR 8558          sched 2808
    8558 8630 8770 9071          0368 2664 2808 2814 2816
redircmd 8575 8764      2818 2820 2832 2898
    8575 8613 8631 8764 8766      scheduler 2758
    8975 8978 8981 9059 9072      0367 1257 2303 2758 2781
REG_ID 7610          2822
    7610 7657          SCROLLLOCK 7714
REG_TABLE 7612      7714 7747
    7612 7664 7665 7675 7676      SECS 7524
REG_VER 7611          7524 7543
    7611 7656          SECTOR_SIZE 4215
release 1602      4215 4280
    0384 1602 1605 2484 2491      SECTSIZE 9212
    2552 2618 2696 2702 2788          9212 9273 9286 9289 9294
    2833 2857 2892 2905 2968      SEG 0751
    2986 2990 3180 3197 3419          0751 1724 1725 1726 1727
    3826 3831 3844 4312 4330      SEG16 0755
    4383 4476 4491 4545 4630          0755 1870
    4640 4657 4839 4870 4879      SEG_ASM 0660
    4940 5265 5281 5293 5364          0660 1189 1190 9184 9185

```

```

segdesc 0725          9184
    0509 0512 0725 0751 0755      start 1123 8409 9111
    2305          1122 1123 1166 1174 1176
seginit 1715          4740 4764 4777 4790 4806
    0426 1223 1244 1715          4890 5152 8408 8409 9110
SEG_KCODE 0714      9111 9167
    0714 1143 1724 3372 3373      startothers 1264
    9153          1208 1233 1264
SEG_KDATA 0715      stat 4004
    0715 1153 1725 1873 3313      0259 0283 0302 4004 4964
    9158          5488 5952 6059 6179 8503
SEG_NULLASM 0654      stati 5488
    0654 1188 9183          0302 5488 5956
SEG_TSS 0718          STA_W 0666 0765
    0718 1870 1872 1878          0666 0765 1190 1725 1727
SEG_UCODE 0716      9185
    0716 1726 2533          STA_X 0665 0764
SEG_UDATA 0717      0665 0764 1189 1724 1726
    0717 1727 2534          9184
SETGATE 0875          sti 0563
    0875 3372 3373          0563 0565 1686 2766
setupkvm 1818      stosb 0492
    0428 1818 1842 2042 2528      0492 0494 6910 9240
    6637          stosl 0501
SHIFT 7708          0501 0503 6908
    7708 7736 7737 7885          strlen 7001
skipelem 5715      0399 6674 6675 7001 8718
    5715 5764          8923
sleep 2874          strncmp 6958
    0370 2707 2874 2879 2882          0400 5605 6958
    3009 3829 4379 4615 4626          strncpy 6968
    4833 4836 6842 6861 8236          0401 5672 6968
    8479          STS_IG32 0770
sleeplock 3901      0770 0881
    0258 0389 0390 0391 0392          STS_T32A 0769
    3854 3901 4166 4211 4424          0769 1870
    4610 4613 4622 4634 4651          STS_TG32 0771
    4704 4968 5859 6064 6757          0771 0881
    7909 8307          sum 7218
spinlock 1501      7218 7220 7222 7224 7225
    0257 0370 0380 0382 0383          7237 7292
    0384 0418 1501 1559 1562          superblock 4063
    1574 1602 1652 2407 2410          0260 0287 4063 4761 4977
    2874 3109 3120 3358 3363          4981
    3903 4210 4230 4423 4429          SVR 7367
    4609 4703 4739 4967 5137          7367 7414
    5858 5864 6063 6756 6763          switchkvm 1853
    7908 7922 8306          0437 1243 1843 1853 2782
STA_R 0667 0766      switchvum 1860
    0667 0766 1189 1724 1726          0436 1860 1863 1865 1867

```

```

2572 2778 6701
swtch 3059
0377 2781 2822 3058 3059
syscall 3701
0409 3407 3557 3701
SYSCALL 8453 8460 8461 8462 8463 84
8460 8461 8462 8463 8464
8465 8466 8467 8468 8469
8470 8471 8472 8473 8474
8475 8476 8477 8478 8479
8480
sys_chdir 6501
3650 3681 6501
SYS_chdir 3509
3509 3681
sys_close 6163
3651 3693 6163
SYS_close 3521
3521 3693
sys_dup 6118
3652 3682 6118
SYS_dup 3510
3510 3682
sys_exec 6526
3653 3679 6526
SYS_exec 3507
3507 3679 8413
sys_exit 3766
3654 3674 3766
SYS_exit 3502
3502 3674 8418
sys_fork 3760
3655 3673 3760
SYS_fork 3501
3501 3673
sys_fstat 6176
3656 3680 6176
SYS_fstat 3508
3508 3680
sys_getpid 3789
3657 3683 3789
SYS_getpid 3511
3511 3683
sys_kill 3779
3658 3678 3779
SYS_kill 3506
3506 3678
sys_link 6202
3659 3691 6202
SYS_link 3519
3519 3691
sys_mkdir 6451
3660 3692 6451
SYS_mkdir 3520
3520 3692
sys_mknod 6467
3661 3689 6467
SYS_mknod 3517
3517 3689
sys_open 6401
3662 3687 6401
SYS_open 3515
3515 3687
sys_pipe 6551
3663 3676 6551
SYS_pipe 3504
3504 3676
sys_read 6132
3664 3677 6132
SYS_read 3505
3505 3677
sys_sbrk 3801
3665 3684 3801
SYS_sbrk 3512
3512 3684
sys_sleep 3815
3666 3685 3815
SYS_sleep 3513
3513 3685
sys_unlink 6301
3667 3690 6301
SYS_unlink 3518
3518 3690
sys_uptime 3838
3670 3686 3838
SYS_uptime 3514
3514 3686
sys_wait 3773
3668 3675 3773
SYS_wait 3503
3503 3675
sys_write 6151
3669 3688 6151
SYS_write 3516
3516 3688
taskstate 0814
0814 2304
TDCR 7391
7391 7420
T_DEV 4002

```

```

4002 5508 5558 6477
T_DIR 4000
4000 5616 5766 6217 6329
6337 6384 6425 6457 6513
T_FILE 4001
4001 6369 6414
ticks 3364
0416 3364 3417 3418 3823
3824 3829 3843
tickslock 3363
0418 3363 3375 3416 3419
3822 3826 3829 3831 3842
3844
TICR 7389
7389 7422
TIMER 7381
7381 7421
T_IRQ0 3229
3229 3414 3423 3427 3430
3434 3438 3439 3474 7414
7421 7434 7664 7675
TPR 7365
7365 7451
trap 3401
3252 3254 3319 3401 3453
3455 3458
trapframe 0602
0602 2344 2502 3401
trapret 3324
2418 2507 3323 3324
T_SYSCALL 3226
3226 3373 3403 8414 8419
8457
tvinit 3367
0417 1229 3367
uart 8316
8316 8337 8355 8365
uartgetc 8363
8363 8375
uartinit 8319
0421 1227 8319
uartintr 8373
0422 3435 8373
uartputc 8351
0423 8160 8162 8347 8351
userinit 2520
0371 1235 2520 2529
uva2ka 2102
0429 2102 2126
V2P 0210
0210 1287 1289 1751 1811
1812 1855 1879 1894 1946
2054 3168
V2P_WO 0213
0213 1040 1050
VER 7364
7364 7430
wait 2671
0372 2671 3775 8462 8533
8644 8670 8671 8725
waitdisk 9251
9251 9263 9272
wakeup 2964
0373 2964 3418 4324 4639
4868 4878 6816 6819 6841
6846 6868 8208
wakeupl 2953
2420 2651 2658 2953 2967
walkpgdir 1735
1735 1768 1911 1971 2026
2045 2106
write_head 4804
4804 4823 4905 4908
writei 5553
0303 5553 5674 6026 6335
6336
write_log 4885
4885 4904
xchg 0569
0569 1256 1581
YEAR 7529
7529 7548
yield 2828
0374 2828 3475

```



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

```

0150 #define NPROC      64 // maximum number of processes
0151 #define KSTACKSIZE 4096 // size of per-process kernel stack
0152 #define NCPU        8 // maximum number of CPUs
0153 #define NOFILE      16 // open files per process
0154 #define NFILE       100 // open files per system
0155 #define NINODE       50 // maximum number of active i-nodes
0156 #define NDEV        10 // maximum major device number
0157 #define ROOTDEV      1 // device number of file system root disk
0158 #define MAXARG       32 // max exec arguments
0159 #define MAXOPBLOCKS  10 // max # of blocks any FS op writes
0160 #define LOGSIZE      (MAXOPBLOCKS*3) // max data blocks in on-disk log
0161 #define NBUF         (MAXOPBLOCKS*3) // size of disk block cache
0162 #define FSSIZE       1000 // size of file system in blocks
0163
0164
0165
0166
0167
0168
0169
0170
0171
0172
0173
0174
0175
0176
0177
0178
0179
0180
0181
0182
0183
0184
0185
0186
0187
0188
0189
0190
0191
0192
0193
0194
0195
0196
0197
0198
0199

```

```

0200 // Memory layout
0201
0202 #define EXTMEM  0x100000 // Start of extended memory
0203 #define PHYSTOP 0xE000000 // Top physical memory
0204 #define DEVSPACE 0xFE000000 // Other devices are at high addresses
0205
0206 // Key addresses for address space layout (see kmap in vm.c for layout)
0207 #define KERNBASE 0x80000000 // First kernel virtual address
0208 #define KERNLINK (KERNBASE+EXTMEM) // Address where kernel is linked
0209
0210 #define V2P(a) (((uint) (a)) - KERNBASE)
0211 #define P2V(a) ((void *) (((char *) (a)) + KERNBASE))
0212
0213 #define V2P_WO(x) ((x) - KERNBASE) // same as V2P, but without casts
0214 #define P2V_WO(x) ((x) + KERNBASE) // same as P2V, but without casts
0215
0216
0217
0218
0219
0220
0221
0222
0223
0224
0225
0226
0227
0228
0229
0230
0231
0232
0233
0234
0235
0236
0237
0238
0239
0240
0241
0242
0243
0244
0245
0246
0247
0248
0249

```

```

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

```

```

0300 struct inode* nameiparent(char*, char*);
0301 int           readi(struct inode*, char*, uint, uint);
0302 void          stati(struct inode*, struct stat*);
0303 int           writei(struct inode*, char*, uint, uint);
0304
0305 // ide.c
0306 void          ideinit(void);
0307 void          ideintr(void);
0308 void          iderw(struct buf*);
0309
0310 // ioapic.c
0311 void          ioapicenable(int irq, int cpu);
0312 extern uchar  ioapicid;
0313 void          ioapicinit(void);
0314
0315 // kalloc.c
0316 char*         kalloc(void);
0317 void          kfree(char*);
0318 void          kinit1(void*, void*);
0319 void          kinit2(void*, void*);
0320
0321 // kbd.c
0322 void          kbdtintr(void);
0323
0324 // lapic.c
0325 void          cmostime(struct rtcdate *r);
0326 int           lapicid(void);
0327 extern volatile uint* lapic;
0328 void          lapiceoi(void);
0329 void          lapicinit(void);
0330 void          lapicstartap(uchar, uint);
0331 void          microdelay(int);
0332
0333 // log.c
0334 void          initlog(int dev);
0335 void          log_write(struct buf*);
0336 void          begin_op();
0337 void          end_op();
0338
0339 // mp.c
0340 extern int     ismp;
0341 void          mpinit(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 int      cpuid(void);
0359 void     exit(void);
0360 int      fork(void);
0361 int      growproc(int);
0362 int      kill(int);
0363 struct cpu* mycpu(void);
0364 struct proc* myproc();
0365 void     pinit(void);
0366 void     procdump(void);
0367 void     scheduler(void) __attribute__((noreturn));
0368 void     sched(void);
0369 void     setproc(struct proc*);
0370 void     sleep(void*, struct spinlock*);
0371 void     userinit(void);
0372 int      wait(void);
0373 void     wakeup(void*);
0374 void     yield(void);
0375
0376 // swtch.S
0377 void     swtch(struct context**, struct context*);
0378
0379 // spinlock.c
0380 void     acquire(struct spinlock*);
0381 void     getcallerpcs(void*, uint*);
0382 int      holding(struct spinlock*);
0383 void     initlock(struct spinlock*, char*);
0384 void     release(struct spinlock*);
0385 void     pushcli(void);
0386 void     popcli(void);
0387
0388 // sleeplock.c
0389 void     acquiresleep(struct sleeplock*);
0390 void     releasesleep(struct sleeplock*);
0391 int      holdingsleep(struct sleeplock*);
0392 void     initsleeplock(struct sleeplock*, char*);
0393
0394 // string.c
0395 int      memcmp(const void*, const void*, uint);
0396 void*    memmove(void*, const void*, uint);
0397 void*    memset(void*, int, uint);
0398 char*    safestrcpy(char*, const char*, int);
0399 int      strlen(const char*);

```

```

0400 int      strcmp(const char*, const char*, uint);
0401 char*    strncpy(char*, const char*, int);
0402
0403 // syscall.c
0404 int      argint(int, int*);
0405 int      argptr(int, char**, int);
0406 int      argstr(int, char**);
0407 int      fetchint(uint, int*);
0408 int      fetchstr(uint, char**);
0409 void     syscall(void);
0410
0411 // timer.c
0412 void     timerinit(void);
0413
0414 // trap.c
0415 void     idtinit(void);
0416 extern uint ticks;
0417 void     tvinit(void);
0418 extern struct spinlock tickslock;
0419
0420 // uart.c
0421 void     uartinit(void);
0422 void     uartintr(void);
0423 void     uartputc(int);
0424
0425 // vm.c
0426 void     seginit(void);
0427 void     kvmalloc(void);
0428 pde_t*   setupkvm(void);
0429 char*    uva2ka(pde_t*, char*);
0430 int      allocuvmm(pde_t*, uint, uint);
0431 int      deallocuvmm(pde_t*, uint, uint);
0432 void     freevm(pde_t*);
0433 void     inituvm(pde_t*, char*, uint);
0434 int      loaduvm(pde_t*, char*, struct inode*, uint, uint);
0435 pde_t*   copyuvm(pde_t*, uint);
0436 void     switchuvm(struct proc*);
0437 void     switchkvm(void);
0438 int      copyout(pde_t*, uint, void*, uint);
0439 void     clearpteu(pde_t *pgdir, char *uva);
0440
0441 // number of elements in fixed-size array
0442 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
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
0457     asm volatile("in %1,%0" : "=a" (data) : "d" (port));
0458     return data;
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_W 0x2 // Writeable (non-executable segments)
0667 #define STA_R 0x2 // Readable (executable segments)
0668
0669
0670
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_IF 0x00000200 // Interrupt Enable
0705
0706 // Control Register flags
0707 #define CR0_PE 0x00000001 // Protection Enable
0708 #define CR0_WP 0x00010000 // Write Protect
0709 #define CR0_PG 0x80000000 // Paging
0710
0711 #define CR4_PSE 0x00000010 // Page size extension
0712
0713 // various segment selectors.
0714 #define SEG_KCODE 1 // kernel code
0715 #define SEG_KDATA 2 // kernel data+stack
0716 #define SEG_UCODE 3 // user code
0717 #define SEG_UDATA 4 // user data+stack
0718 #define SEG_TSS 5 // this process's task state
0719
0720 // cpu->gdt[NSEGS] holds the above segments.
0721 #define NSEGS 6
0722
0723 #ifndef __ASSEMBLER__
0724 // Segment Descriptor
0725 struct segdesc {
0726     uint lim_15_0 : 16; // Low bits of segment limit
0727     uint base_15_0 : 16; // Low bits of segment base address
0728     uint base_23_16 : 8; // Middle bits of segment base address
0729     uint type : 4; // Segment type (see STS_ constants)
0730     uint s : 1; // 0 = system, 1 = application
0731     uint dpl : 2; // Descriptor Privilege Level
0732     uint p : 1; // Present
0733     uint lim_19_16 : 4; // High bits of segment limit
0734     uint avl : 1; // Unused (available for software use)
0735     uint rsv1 : 1; // Reserved
0736     uint db : 1; // 0 = 16-bit segment, 1 = 32-bit segment
0737     uint g : 1; // Granularity: limit scaled by 4K when set
0738     uint base_31_24 : 8; // High bits of segment base address
0739 };
0740
0741
0742
0743
0744
0745
0746
0747
0748
0749

```



```

0750 // Normal segment
0751 #define SEG(type, base, lim, dpl) (struct segdesc) \
0752 { ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff, \
0753   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
0754   (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
0755 #define SEG16(type, base, lim, dpl) (struct segdesc) \
0756 { (lim) & 0xffff, (uint)(base) & 0xffff, \
0757   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
0758   (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
0759 #endif
0760
0761 #define DPL_USER 0x3 // User DPL
0762
0763 // Application segment type bits
0764 #define STA_X 0x8 // Executable segment
0765 #define STA_W 0x2 // Writeable (non-executable segments)
0766 #define STA_R 0x2 // Readable (executable segments)
0767
0768 // System segment type bits
0769 #define STS_T32A 0x9 // Available 32-bit TSS
0770 #define STS_IG32 0xE // 32-bit Interrupt Gate
0771 #define STS_TG32 0xF // 32-bit Trap Gate
0772
0773 // A virtual address 'la' has a three-part structure as follows:
0774 //
0775 // +-----10-----+-----10-----+-----12-----+
0776 // | Page Directory | Page Table | Offset within Page |
0777 // | Index         | Index         |                     |
0778 // +-----+-----+-----+
0779 // \--- PDX(va) --/ \--- PTX(va) --/
0780
0781 // page directory index
0782 #define PDX(va) (((uint)(va) >> PDXSHIFT) & 0x3FF)
0783
0784 // page table index
0785 #define PTX(va) (((uint)(va) >> PTXSHIFT) & 0x3FF)
0786
0787 // construct virtual address from indexes and offset
0788 #define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
0789
0790 // Page directory and page table constants.
0791 #define NPENTRIES 1024 // # directory entries per page directory
0792 #define NPTENTRIES 1024 // # PTEs per page table
0793 #define PGSIZE 4096 // bytes mapped by a page
0794
0795 #define PTXSHIFT 12 // offset of PTX in a linear address
0796 #define PDXSHIFT 22 // offset of PDX in a linear address
0797
0798 #define PGROUNDUP(sz) (((sz)+PGSIZE-1) & ~(PGSIZE-1))
0799 #define PGROUNDDOWN(a) (((a) & ~(PGSIZE-1))

```

```

0800 // Page table/directory entry flags.
0801 #define PTE_P 0x001 // Present
0802 #define PTE_W 0x002 // Writeable
0803 #define PTE_U 0x004 // User
0804 #define PTE_PS 0x080 // Page Size
0805
0806 // Address in page table or page directory entry
0807 #define PTE_ADDR(pte) ((uint)(pte) & ~0xFFF)
0808 #define PTE_FLAGS(pte) ((uint)(pte) & 0xFFF)
0809
0810 #ifndef __ASSEMBLER__
0811 typedef uint pte_t;
0812
0813 // Task state segment format
0814 struct taskstate {
0815   uint link; // Old ts selector
0816   uint esp0; // Stack pointers and segment selectors
0817   ushort ss0; // after an increase in privilege level
0818   ushort padding1;
0819   uint *esp1;
0820   ushort ssl;
0821   ushort padding2;
0822   uint *esp2;
0823   ushort ss2;
0824   ushort padding3;
0825   void *cr3; // Page directory base
0826   uint *eip; // Saved state from last task switch
0827   uint eflags;
0828   uint eax; // More saved state (registers)
0829   uint ecx;
0830   uint edx;
0831   uint ebx;
0832   uint *esp;
0833   uint *ebp;
0834   uint esi;
0835   uint edi;
0836   ushort es; // Even more saved state (segment selectors)
0837   ushort padding4;
0838   ushort cs;
0839   ushort padding5;
0840   ushort ss;
0841   ushort padding6;
0842   ushort ds;
0843   ushort padding7;
0844   ushort fs;
0845   ushort padding8;
0846   ushort gs;
0847   ushort padding9;
0848   ushort ldt;
0849   ushort padding10;

```

```

0850  ushort t;           // Trap on task switch
0851  ushort iomb;        // I/O map base address
0852  };
0853
0854 // Gate descriptors for interrupts and traps
0855 struct gatedesc {
0856     uint off_15_0 : 16; // low 16 bits of offset in segment
0857     uint cs : 16;        // code segment selector
0858     uint args : 5;      // # args, 0 for interrupt/trap gates
0859     uint rsv1 : 3;      // reserved(should be zero I guess)
0860     uint type : 4;      // type(STS_{IG32,TG32})
0861     uint s : 1;        // must be 0 (system)
0862     uint dpl : 2;      // descriptor(meaning new) privilege level
0863     uint p : 1;        // Present
0864     uint off_31_16 : 16; // high bits of offset in segment
0865 };
0866
0867 // Set up a normal interrupt/trap gate descriptor.
0868 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
0869 // - interrupt gate clears FL_IF, trap gate leaves FL_IF alone
0870 // - sel: Code segment selector for interrupt/trap handler
0871 // - off: Offset in code segment for interrupt/trap handler
0872 // - dpl: Descriptor Privilege Level -
0873 //       the privilege level required for software to invoke
0874 //       this interrupt/trap gate explicitly using an int instruction.
0875 #define SETGATE(gate, istrap, sel, off, d) \
0876 { \
0877     (gate).off_15_0 = (uint)(off) & 0xffff; \
0878     (gate).cs = (sel); \
0879     (gate).args = 0; \
0880     (gate).rsv1 = 0; \
0881     (gate).type = (istrap) ? STS_TG32 : STS_IG32; \
0882     (gate).s = 0; \
0883     (gate).dpl = (d); \
0884     (gate).p = 1; \
0885     (gate).off_31_16 = (uint)(off) >> 16; \
0886 }
0887
0888 #endif
0889
0890
0891
0892
0893
0894
0895
0896
0897
0898
0899

```

```

0900 // Format of an ELF executable file
0901
0902 #define ELF_MAGIC 0x464C457FU // "\x7FELF" in little endian
0903
0904 // File header
0905 struct elfhdr {
0906     uint magic; // must equal ELF_MAGIC
0907     uchar elf[12];
0908     ushort type;
0909     ushort machine;
0910     uint version;
0911     uint entry;
0912     uint phoff;
0913     uint shoff;
0914     uint flags;
0915     ushort ehsize;
0916     ushort phentsize;
0917     ushort phnum;
0918     ushort shentsize;
0919     ushort shnum;
0920     ushort shstrndx;
0921 };
0922
0923 // Program section header
0924 struct proghdr {
0925     uint type;
0926     uint off;
0927     uint vaddr;
0928     uint paddr;
0929     uint filesz;
0930     uint memsz;
0931     uint flags;
0932     uint align;
0933 };
0934
0935 // Values for Proghdr type
0936 #define ELF_PROG_LOAD 1
0937
0938 // Flag bits for Proghdr flags
0939 #define ELF_PROG_FLAG_EXEC 1
0940 #define ELF_PROG_FLAG_WRITE 2
0941 #define ELF_PROG_FLAG_READ 4
0942
0943
0944
0945
0946
0947
0948
0949

```

```

0950 struct rtcdate {
0951     uint second;
0952     uint minute;
0953     uint hour;
0954     uint day;
0955     uint month;
0956     uint year;
0957 };
0958
0959
0960
0961
0962
0963
0964
0965
0966
0967
0968
0969
0970
0971
0972
0973
0974
0975
0976
0977
0978
0979
0980
0981
0982
0983
0984
0985
0986
0987
0988
0989
0990
0991
0992
0993
0994
0995
0996
0997
0998
0999

```

```

1000 # The xv6 kernel starts executing in this file. This file is linked with
1001 # the kernel C code, so it can refer to kernel symbols such as main().
1002 # The boot block (bootasm.S and bootmain.c) jumps to entry below.
1003
1004 # Multiboot header, for multiboot boot loaders like GNU Grub.
1005 # http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
1006 #
1007 # Using GRUB 2, you can boot xv6 from a file stored in a
1008 # Linux file system by copying kernel or kernelmemfs to /boot
1009 # and then adding this menu entry:
1010 #
1011 # menuentry "xv6" {
1012 #     insmod ext2
1013 #     set root='(hd0,msdos1)'
1014 #     set kernel='/boot/kernel'
1015 #     echo "Loading ${kernel}..."
1016 #     multiboot ${kernel} ${kernel}
1017 #     boot
1018 # }
1019
1020 #include "asm.h"
1021 #include "memlayout.h"
1022 #include "mmu.h"
1023 #include "param.h"
1024
1025 # Multiboot header. Data to direct multiboot loader.
1026 .p2align 2
1027 .text
1028 .globl multiboot_header
1029 multiboot_header:
1030     #define magic 0x1badb002
1031     #define flags 0
1032     .long magic
1033     .long flags
1034     .long (-magic-flags)
1035
1036 # By convention, the _start symbol specifies the ELF entry point.
1037 # Since we haven't set up virtual memory yet, our entry point is
1038 # the physical address of 'entry'.
1039 .globl _start
1040 _start = V2P_WO(entry)
1041
1042 # Entering xv6 on boot processor, with paging off.
1043 .globl entry
1044 entry:
1045     # Turn on page size extension for 4Mbyte pages
1046     movl    %cr4, %eax
1047     orl     $(CR4_PSE), %eax
1048     movl    %eax, %cr4
1049     # Set page directory

```

```

1050 movl    $(V2P_W0(entrypgdir)), %eax
1051 movl    %eax, %cr3
1052 # Turn on paging.
1053 movl    %cr0, %eax
1054 orl     $(CR0_PG|CR0_WP), %eax
1055 movl    %eax, %cr0
1056
1057 # Set up the stack pointer.
1058 movl    $(stack + KSTACKSIZE), %esp
1059
1060 # Jump to main(), and switch to executing at
1061 # high addresses. The indirect call is needed because
1062 # the assembler produces a PC-relative instruction
1063 # for a direct jump.
1064 mov $main, %eax
1065 jmp *%eax
1066
1067 .comm stack, KSTACKSIZE
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 combines elements of bootasm.S and entry.S.
1120
1121 .code16
1122 .globl start
1123 start:
1124     cli
1125
1126     # Zero data segment registers DS, ES, and SS.
1127     xorw    %ax, %ax
1128     movw    %ax, %ds
1129     movw    %ax, %es
1130     movw    %ax, %ss
1131
1132     # Switch from real to protected mode. Use a bootstrap GDT that makes
1133     # virtual addresses map directly to physical addresses so that the
1134     # effective memory map doesn't change during the transition.
1135     lgdt    gdtdesc
1136     movl    %cr0, %eax
1137     orl     $CR0_PE, %eax
1138     movl    %eax, %cr0
1139
1140     # Complete the transition to 32-bit protected mode by using a long jmp
1141     # to reload %cs and %eip. The segment descriptors are set up with no
1142     # translation, so that the mapping is still the identity mapping.
1143     ljmpl   $(SEG_KCODE<<3), $(start32)
1144
1145
1146
1147
1148
1149

```

```

1150 .code32 # Tell assembler to generate 32-bit code now.
1151 start32:
1152 # Set up the protected-mode data segment registers
1153 movw $(SEG_KDATA<<3), %ax # Our data segment selector
1154 movw %ax, %ds # -> DS: Data Segment
1155 movw %ax, %es # -> ES: Extra Segment
1156 movw %ax, %ss # -> SS: Stack Segment
1157 movw $0, %ax # Zero segments not ready for use
1158 movw %ax, %fs # -> FS
1159 movw %ax, %gs # -> GS
1160
1161 # Turn on page size extension for 4Mbyte pages
1162 movl %cr4, %eax
1163 orl $(CR4_PSE), %eax
1164 movl %eax, %cr4
1165 # Use entrypgdir as our initial page table
1166 movl (start-12), %eax
1167 movl %eax, %cr3
1168 # Turn on paging.
1169 movl %cr0, %eax
1170 orl $(CR0_PE|CR0_PG|CR0_WP), %eax
1171 movl %eax, %cr0
1172
1173 # Switch to the stack allocated by startothers()
1174 movl (start-4), %esp
1175 # Call mpenter()
1176 call *(start-8)
1177
1178 movw $0x8a00, %ax
1179 movw %ax, %dx
1180 outw %ax, %dx
1181 movw $0x8ae0, %ax
1182 outw %ax, %dx
1183 spin:
1184 jmp spin
1185
1186 .p2align 2
1187 gdt:
1188 SEG_NULLASM
1189 SEG_ASM(STA_X|STA_R, 0, 0xffffffff)
1190 SEG_ASM(STA_W, 0, 0xffffffff)
1191
1192
1193 gdtdesc:
1194 .word (gdtdesc - gdt - 1)
1195 .long gdt
1196
1197
1198
1199

```

```

1200 #include "types.h"
1201 #include "defs.h"
1202 #include "param.h"
1203 #include "memlayout.h"
1204 #include "mmu.h"
1205 #include "proc.h"
1206 #include "x86.h"
1207
1208 static void startothers(void);
1209 static void mpmain(void) __attribute__((noreturn));
1210 extern pde_t *kpgdir;
1211 extern char end[]; // first address after kernel loaded from ELF file
1212
1213 // Bootstrap processor starts running C code here.
1214 // Allocate a real stack and switch to it, first
1215 // doing some setup required for memory allocator to work.
1216 int
1217 main(void)
1218 {
1219     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1220     kvmalloc(); // kernel page table
1221     mpinit(); // detect other processors
1222     lapicinit(); // interrupt controller
1223     seginit(); // segment descriptors
1224     picinit(); // disable pic
1225     ioapicinit(); // another interrupt controller
1226     consoleinit(); // console hardware
1227     uartinit(); // serial port
1228     pinit(); // process table
1229     tvinit(); // trap vectors
1230     binit(); // buffer cache
1231     fileinit(); // file table
1232     ideinit(); // disk
1233     startothers(); // start other processors
1234     kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1235     userinit(); // first user process
1236     mpmain(); // finish this processor's setup
1237 }
1238
1239 // Other CPUs jump here from entryother.S.
1240 static void
1241 mpenter(void)
1242 {
1243     switchkvm();
1244     seginit();
1245     lapicinit();
1246     mpmain();
1247 }
1248
1249

```

```

1250 // Common CPU setup code.
1251 static void
1252 mpmain(void)
1253 {
1254     printf("cpu%d: starting %d\n", cpuid(), cpuid());
1255     idtinit(); // load idt register
1256     xchg(&(mycpu()->started), 1); // tell startothers() we're up
1257     scheduler(); // start running processes
1258 }
1259
1260 pde_t entrypgdir[]; // For entry.S
1261
1262 // Start the non-boot (AP) processors.
1263 static void
1264 startothers(void)
1265 {
1266     extern uchar _binary_entryother_start[], _binary_entryother_size[];
1267     uchar *code;
1268     struct cpu *c;
1269     char *stack;
1270
1271     // Write entry code to unused memory at 0x7000.
1272     // The linker has placed the image of entryother.S in
1273     // _binary_entryother_start.
1274     code = P2V(0x7000);
1275     memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
1276
1277     for(c = cpus; c < cpus+ncpu; c++){
1278         if(c == mycpu()) // We've started already.
1279             continue;
1280
1281         // Tell entryother.S what stack to use, where to enter, and what
1282         // pgdir to use. We cannot use kpgdir yet, because the AP processor
1283         // is running in low memory, so we use entrypgdir for the APs too.
1284         stack = kalloc();
1285         *(void**) (code-4) = stack + KSTACKSIZE;
1286         *(void**) (void) (code-8) = mpenter;
1287         *(int**) (code-12) = (void *) V2P(entrypgdir);
1288
1289         lapicstartap(c->apicid, V2P(code));
1290
1291         // wait for cpu to finish mpmain()
1292         while(c->started == 0)
1293             ;
1294     }
1295 }
1296
1297
1298
1299

```

```

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

```

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

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

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

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



```

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

```

```

1600 // Release the lock.
1601 void
1602 release(struct spinlock *lk)
1603 {
1604     if(!holding(lk))
1605         panic("release");
1606
1607     lk->pcs[0] = 0;
1608     lk->cpu = 0;
1609
1610     // Tell the C compiler and the processor to not move loads or stores
1611     // past this point, to ensure that all the stores in the critical
1612     // section are visible to other cores before the lock is released.
1613     // Both the C compiler and the hardware may re-order loads and
1614     // stores; __sync_synchronize() tells them both not to.
1615     __sync_synchronize();
1616
1617     // Release the lock, equivalent to lk->locked = 0.
1618     // This code can't use a C assignment, since it might
1619     // not be atomic. A real OS would use C atomics here.
1620     asm volatile("movl $0, %0" : "+m" (lk->locked) : );
1621
1622     popcli();
1623 }
1624
1625 // Record the current call stack in pcs[] by following the %ebp chain.
1626 void
1627 getcallerpcs(void *v, uint pcs[])
1628 {
1629     uint *ebp;
1630     int i;
1631
1632     ebp = (uint*)v - 2;
1633     for(i = 0; i < 10; i++){
1634         if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
1635             break;
1636         pcs[i] = ebp[1]; // saved %eip
1637         ebp = (uint*)ebp[0]; // saved %ebp
1638     }
1639     for(; i < 10; i++)
1640         pcs[i] = 0;
1641 }
1642
1643
1644
1645
1646
1647
1648
1649

```

```

1650 // Check whether this cpu is holding the lock.
1651 int
1652 holding(struct spinlock *lock)
1653 {
1654     int r;
1655     pushcli();
1656     r = lock->locked && lock->cpu == mycpu();
1657     popcli();
1658     return r;
1659 }
1660
1661
1662 // Pushcli/popcli are like cli/sti except that they are matched:
1663 // it takes two popcli to undo two pushcli. Also, if interrupts
1664 // are off, then pushcli, popcli leaves them off.
1665
1666 void
1667 pushcli(void)
1668 {
1669     int eflags;
1670
1671     eflags = readeflags();
1672     cli();
1673     if(mycpu()->ncli == 0)
1674         mycpu()->intena = eflags & FL_IF;
1675     mycpu()->ncli += 1;
1676 }
1677
1678 void
1679 popcli(void)
1680 {
1681     if(readeflags() & FL_IF)
1682         panic("popcli - interruptible");
1683     if(--mycpu()->ncli < 0)
1684         panic("popcli");
1685     if(mycpu()->ncli == 0 && mycpu()->intena)
1686         sti();
1687 }
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699

```

```

1700 #include "param.h"
1701 #include "types.h"
1702 #include "defs.h"
1703 #include "x86.h"
1704 #include "memlayout.h"
1705 #include "mmu.h"
1706 #include "proc.h"
1707 #include "elf.h"
1708
1709 extern char data[]; // defined by kernel.ld
1710 pde_t *kpgdir; // for use in scheduler()
1711
1712 // Set up CPU's kernel segment descriptors.
1713 // Run once on entry on each CPU.
1714 void
1715 seginit(void)
1716 {
1717     struct cpu *c;
1718
1719     // Map "logical" addresses to virtual addresses using identity map.
1720     // Cannot share a CODE descriptor for both kernel and user
1721     // because it would have to have DPL_USR, but the CPU forbids
1722     // an interrupt from CPL=0 to DPL=3.
1723     c = &cpu[cuid()];
1724     c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
1725     c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1726     c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
1727     c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1728     lgdt(c->gdt, sizeof(c->gdt));
1729 }
1730
1731 // Return the address of the PTE in page table pgdir
1732 // that corresponds to virtual address va. If alloc!=0,
1733 // create any required page table pages.
1734 static pte_t *
1735 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1736 {
1737     pde_t *pde;
1738     pte_t *pgtab;
1739
1740     pde = &pgdir[PDX(va)];
1741     if(*pde & PTE_P){
1742         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1743     } else {
1744         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1745             return 0;
1746         // Make sure all those PTE_P bits are zero.
1747         memset(pgtab, 0, PGSIZE);
1748         // The permissions here are overly generous, but they can
1749         // be further restricted by the permissions in the page table

```

```

1750 // entries, if necessary.
1751 *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1752 }
1753 return &pgtab[PTX(va)];
1754 }
1755
1756 // Create PTEs for virtual addresses starting at va that refer to
1757 // physical addresses starting at pa. va and size might not
1758 // be page-aligned.
1759 static int
1760 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1761 {
1762     char *a, *last;
1763     pte_t *pte;
1764
1765     a = (char*)PGROUNDDOWN((uint)va);
1766     last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1767     for(;;){
1768         if((pte = walkpgdir(pgdir, a, 1)) == 0)
1769             return -1;
1770         if(*pte & PTE_P)
1771             panic("remap");
1772         *pte = pa | perm | PTE_P;
1773         if(a == last)
1774             break;
1775         a += PGSIZE;
1776         pa += PGSIZE;
1777     }
1778     return 0;
1779 }
1780
1781 // There is one page table per process, plus one that's used when
1782 // a CPU is not running any process (kpgdir). The kernel uses the
1783 // current process's page table during system calls and interrupts;
1784 // page protection bits prevent user code from using the kernel's
1785 // mappings.
1786 //
1787 // setupkvm() and exec() set up every page table like this:
1788 //
1789 // 0..KERNBASE: user memory (text+data+stack+heap), mapped to
1790 //               phys memory allocated by the kernel
1791 // KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
1792 // KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
1793 //                       for the kernel's instructions and r/o data
1794 // data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
1795 //                       rw data + free physical memory
1796 // 0xfe000000..0: mapped direct (devices such as ioapic)
1797 //
1798 // The kernel allocates physical memory for its heap and for user memory
1799 // between V2P(end) and the end of physical memory (PHYSTOP)

```

```

1800 // (directly addressable from end..P2V(PHYSTOP)).
1801
1802 // This table defines the kernel's mappings, which are present in
1803 // every process's page table.
1804 static struct kmap {
1805     void *virt;
1806     uint phys_start;
1807     uint phys_end;
1808     int perm;
1809 } kmap[] = {
1810     { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
1811     { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},    // kern text+rodata
1812     { (void*)data,     V2P(data),     PHYSTOP,    PTE_W}, // kern data+memory
1813     { (void*)DEVSPACE, DEVSPACE,      0,          PTE_W}, // more devices
1814 };
1815
1816 // Set up kernel part of a page table.
1817 pde_t*
1818 setupkvm(void)
1819 {
1820     pde_t *pgdir;
1821     struct kmap *k;
1822
1823     if((pgdir = (pde_t*)kalloc()) == 0)
1824         return 0;
1825     memset(pgdir, 0, PGSIZE);
1826     if (P2V(PHYSTOP) > (void*)DEVSPACE)
1827         panic("PHYSTOP too high");
1828     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1829         if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1830                     (uint)k->phys_start, k->perm) < 0) {
1831             freevm(pgdir);
1832             return 0;
1833         }
1834     return pgdir;
1835 }
1836
1837 // Allocate one page table for the machine for the kernel address
1838 // space for scheduler processes.
1839 void
1840 kvmalloc(void)
1841 {
1842     kpgdir = setupkvm();
1843     switchkvm();
1844 }
1845
1846
1847
1848
1849

```

```

1850 // Switch h/w page table register to the kernel-only page table,
1851 // for when no process is running.
1852 void
1853 switchkvm(void)
1854 {
1855     lcr3(V2P(kpgdir)); // switch to the kernel page table
1856 }
1857
1858 // Switch TSS and h/w page table to correspond to process p.
1859 void
1860 switchvm(struct proc *p)
1861 {
1862     if(p == 0)
1863         panic("switchvm: no process");
1864     if(p->kstack == 0)
1865         panic("switchvm: no kstack");
1866     if(p->pgdir == 0)
1867         panic("switchvm: no pgdir");
1868
1869     pushcli();
1870     mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
1871                                 sizeof(mycpu()->ts)-1, 0);
1872     mycpu()->gdt[SEG_TSS].s = 0;
1873     mycpu()->ts.ss0 = SEG_KDATA << 3;
1874     mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
1875     // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
1876     // forbids I/O instructions (e.g., inb and outb) from user space
1877     mycpu()->ts.iomb = (ushort) 0xFFFF;
1878     ltr(SEG_TSS << 3);
1879     lcr3(V2P(p->pgdir)); // switch to process's address space
1880     popcli();
1881 }
1882
1883 // Load the initcode into address 0 of pgdir.
1884 // sz must be less than a page.
1885 void
1886 initvm(pde_t *pgdir, char *init, uint sz)
1887 {
1888     char *mem;
1889
1890     if(sz >= PGSIZE)
1891         panic("initvm: more than a page");
1892     mem = kalloc();
1893     memset(mem, 0, PGSIZE);
1894     mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W|PTE_U);
1895     memmove(mem, init, sz);
1896 }
1897
1898
1899

```

```

1900 // Load a program segment into pgdir. addr must be page-aligned
1901 // and the pages from addr to addr+sz must already be mapped.
1902 int
1903 loadvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
1904 {
1905     uint i, pa, n;
1906     pte_t *pte;
1907
1908     if((uint) addr % PGSIZE != 0)
1909         panic("loadvm: addr must be page aligned");
1910     for(i = 0; i < sz; i += PGSIZE){
1911         if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1912             panic("loadvm: address should exist");
1913         pa = PTE_ADDR(*pte);
1914         if(sz - i < PGSIZE)
1915             n = sz - i;
1916         else
1917             n = PGSIZE;
1918         if(readi(ip, P2V(pa), offset+i, n) != n)
1919             return -1;
1920     }
1921     return 0;
1922 }
1923
1924 // Allocate page tables and physical memory to grow process from oldsz to
1925 // newsz, which need not be page aligned. Returns new size or 0 on error.
1926 int
1927 allocvm(pde_t *pgdir, uint oldsz, uint newsz)
1928 {
1929     char *mem;
1930     uint a;
1931
1932     if(newsz >= KERNBASE)
1933         return 0;
1934     if(newsz < oldsz)
1935         return oldsz;
1936
1937     a = PGROUNDUP(oldsz);
1938     for(; a < newsz; a += PGSIZE){
1939         mem = kalloc();
1940         if(mem == 0){
1941             cprintf("allocvm out of memory\n");
1942             deallocvm(pgdir, newsz, oldsz);
1943             return 0;
1944         }
1945         memset(mem, 0, PGSIZE);
1946         if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
1947             cprintf("allocvm out of memory (2)\n");
1948             deallocvm(pgdir, newsz, oldsz);
1949             kfree(mem);

```

```

1950     return 0;
1951 }
1952 }
1953 return newsz;
1954 }
1955
1956 // Deallocate user pages to bring the process size from oldsz to
1957 // newsz. oldsz and newsz need not be page-aligned, nor does newsz
1958 // need to be less than oldsz. oldsz can be larger than the actual
1959 // process size. Returns the new process size.
1960 int
1961 deallocvm(pde_t *pgdir, uint oldsz, uint newsz)
1962 {
1963     pte_t *pte;
1964     uint a, pa;
1965
1966     if(newsz >= oldsz)
1967         return oldsz;
1968
1969     a = PGROUNDUP(newsz);
1970     for(; a < oldsz; a += PGSIZE){
1971         pte = walkpgdir(pgdir, (char*)a, 0);
1972         if(!pte)
1973             a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
1974         else if((*pte & PTE_P) != 0){
1975             pa = PTE_ADDR(*pte);
1976             if(pa == 0)
1977                 panic("kfree");
1978             char *v = P2V(pa);
1979             kfree(v);
1980             *pte = 0;
1981         }
1982     }
1983     return newsz;
1984 }
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999

```

```

2000 // Free a page table and all the physical memory pages
2001 // in the user part.
2002 void
2003 freevm(pde_t *pgdir)
2004 {
2005     uint i;
2006
2007     if(pgdir == 0)
2008         panic("freevm: no pgdir");
2009     deallocvm(pgdir, KERNBASE, 0);
2010     for(i = 0; i < NPENTRIES; i++){
2011         if(pgdir[i] & PTE_P){
2012             char *v = P2V(PTE_ADDR(pgdir[i]));
2013             kfree(v);
2014         }
2015     }
2016     kfree((char*)pgdir);
2017 }
2018
2019 // Clear PTE_U on a page. Used to create an inaccessible
2020 // page beneath the user stack.
2021 void
2022 clearpteu(pde_t *pgdir, char *uva)
2023 {
2024     pte_t *pte;
2025
2026     pte = walkpgdir(pgdir, uva, 0);
2027     if(pte == 0)
2028         panic("clearpteu");
2029     *pte &= ~PTE_U;
2030 }
2031
2032 // Given a parent process's page table, create a copy
2033 // of it for a child.
2034 pde_t*
2035 copyvm(pde_t *pgdir, uint sz)
2036 {
2037     pde_t *d;
2038     pte_t *pte;
2039     uint pa, i, flags;
2040     char *mem;
2041
2042     if((d = setupkvmm()) == 0)
2043         return 0;
2044     for(i = 0; i < sz; i += PGSIZE){
2045         if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
2046             panic("copyvm: pte should exist");
2047         if(!(*pte & PTE_P))
2048             panic("copyvm: page not present");
2049         pa = PTE_ADDR(*pte);

```

```

2050     flags = PTE_FLAGS(*pte);
2051     if((mem = kalloc()) == 0)
2052         goto bad;
2053     memmove(mem, (char*)P2V(pa), PGSIZE);
2054     if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
2055         kfree(mem);
2056         goto bad;
2057     }
2058 }
2059 return d;
2060
2061 bad:
2062     freevm(d);
2063     return 0;
2064 }
2065
2066
2067
2068
2069
2070
2071
2072
2073
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 // Map user virtual address to kernel address.
2101 char*
2102 uva2ka(pde_t *pgdir, char *uva)
2103 {
2104     pte_t *pte;
2105
2106     pte = walkpgdir(pgdir, uva, 0);
2107     if((*pte & PTE_P) == 0)
2108         return 0;
2109     if((*pte & PTE_U) == 0)
2110         return 0;
2111     return (char*)P2V(PTE_ADDR(*pte));
2112 }
2113
2114 // Copy len bytes from p to user address va in page table pgdir.
2115 // Most useful when pgdir is not the current page table.
2116 // uva2ka ensures this only works for PTE_U pages.
2117 int
2118 copyout(pde_t *pgdir, uint va, void *p, uint len)
2119 {
2120     char *buf, *pa0;
2121     uint n, va0;
2122
2123     buf = (char*)p;
2124     while(len > 0){
2125         va0 = (uint)PGROUNDDOWN(va);
2126         pa0 = uva2ka(pgdir, (char*)va0);
2127         if(pa0 == 0)
2128             return -1;
2129         n = PGSIZE - (va - va0);
2130         if(n > len)
2131             n = len;
2132         memmove(pa0 + (va - va0), buf, n);
2133         len -= n;
2134         buf += n;
2135         va = va0 + PGSIZE;
2136     }
2137     return 0;
2138 }
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149

```

2150 // Blank page.

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

2200 // Blank page.

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

```

2250 // Blank page.
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299

```

```

2300 // Per-CPU state
2301 struct cpu {
2302     uchar apicid;           // Local APIC ID
2303     struct context *scheduler; // swtch() here to enter scheduler
2304     struct taskstate ts;     // Used by x86 to find stack for interrupt
2305     struct segdesc gdt[NSEGS]; // x86 global descriptor table
2306     volatile uint started;   // Has the CPU started?
2307     int ncli;                // Depth of pushcli nesting.
2308     int intena;              // Were interrupts enabled before pushcli?
2309     struct proc *proc;       // The process running on this cpu or null
2310 };
2311
2312 extern struct cpu cpus[NCPU];
2313 extern int ncpu;
2314
2315
2316 // Saved registers for kernel context switches.
2317 // Don't need to save all the segment registers (%cs, etc),
2318 // because they are constant across kernel contexts.
2319 // Don't need to save %eax, %ecx, %edx, because the
2320 // x86 convention is that the caller has saved them.
2321 // Contexts are stored at the bottom of the stack they
2322 // describe; the stack pointer is the address of the context.
2323 // The layout of the context matches the layout of the stack in swtch.S
2324 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2325 // but it is on the stack and allocproc() manipulates it.
2326 struct context {
2327     uint edi;
2328     uint esi;
2329     uint ebx;
2330     uint ebp;
2331     uint eip;
2332 };
2333
2334 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2335
2336 // Per-process state
2337 struct proc {
2338     uint sz;                // Size of process memory (bytes)
2339     pde_t * pgdir;          // Page table
2340     char *kstack;           // Bottom of kernel stack for this process
2341     enum procstate state;   // Process state
2342     int pid;                // Process ID
2343     struct proc *parent;    // Parent process
2344     struct trapframe *tf;   // Trap frame for current syscall
2345     struct context *context; // swtch() here to run process
2346     void *chan;             // If non-zero, sleeping on chan
2347     int killed;             // If non-zero, have been killed
2348     struct file *ofile[NOFILE]; // Open files
2349     struct inode *cwd;      // Current directory

```



```

2350 char name[16];           // Process name (debugging)
2351 };
2352
2353 // Process memory is laid out contiguously, low addresses first:
2354 //  text
2355 //  original data and bss
2356 //  fixed-size stack
2357 //  expandable heap
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399

```

```

2400 #include "types.h"
2401 #include "defs.h"
2402 #include "param.h"
2403 #include "memlayout.h"
2404 #include "mmu.h"
2405 #include "x86.h"
2406 #include "proc.h"
2407 #include "spinlock.h"
2408
2409 struct {
2410     struct spinlock lock;
2411     struct proc proc[NPROC];
2412 } ptable;
2413
2414 static struct proc *initproc;
2415
2416 int nextpid = 1;
2417 extern void forkret(void);
2418 extern void trapret(void);
2419
2420 static void wakeup1(void *chan);
2421
2422 void
2423 pinit(void)
2424 {
2425     initlock(&ptable.lock, "ptable");
2426 }
2427
2428 // Must be called with interrupts disabled
2429 int
2430 cpuid() {
2431     return mycpu() - cpus;
2432 }
2433
2434 // Must be called with interrupts disabled to avoid the caller being
2435 // rescheduled between reading lapicid and running through the loop.
2436 struct cpu*
2437 mycpu(void)
2438 {
2439     int apicid, i;
2440
2441     if(readeflags() & FL_IF)
2442         panic("mycpu called with interrupts enabled\n");
2443
2444     apicid = lapicid();
2445     // APIC IDs are not guaranteed to be contiguous. Maybe we should have
2446     // a reverse map, or reserve a register to store &cpus[i].
2447     for (i = 0; i < ncpu; ++i) {
2448         if (cpus[i].apicid == apicid)
2449             return &cpus[i];

```

```

2450 }
2451 panic("unknown apicid\n");
2452 }
2453
2454 // Disable interrupts so that we are not rescheduled
2455 // while reading proc from the cpu structure
2456 struct proc*
2457 myproc(void) {
2458     struct cpu *c;
2459     struct proc *p;
2460     pushcli();
2461     c = mycpu();
2462     p = c->proc;
2463     popcli();
2464     return p;
2465 }
2466
2467
2468 // Look in the process table for an UNUSED proc.
2469 // If found, change state to EMBRYO and initialize
2470 // state required to run in the kernel.
2471 // Otherwise return 0.
2472 static struct proc*
2473 allocproc(void)
2474 {
2475     struct proc *p;
2476     char *sp;
2477
2478     acquire(&ptable.lock);
2479
2480     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2481         if(p->state == UNUSED)
2482             goto found;
2483
2484     release(&ptable.lock);
2485     return 0;
2486
2487 found:
2488     p->state = EMBRYO;
2489     p->pid = nextpid++;
2490
2491     release(&ptable.lock);
2492
2493     // Allocate kernel stack.
2494     if((p->kstack = kalloc()) == 0){
2495         p->state = UNUSED;
2496         return 0;
2497     }
2498     sp = p->kstack + KSTACKSIZE;
2499

```

```

2500 // Leave room for trap frame.
2501 sp -= sizeof *p->tf;
2502 p->tf = (struct trapframe*)sp;
2503
2504 // Set up new context to start executing at forkret,
2505 // which returns to trapret.
2506 sp -= 4;
2507 *(uint*)sp = (uint)trapret;
2508
2509 sp -= sizeof *p->context;
2510 p->context = (struct context*)sp;
2511 memset(p->context, 0, sizeof *p->context);
2512 p->context->eip = (uint)forkret;
2513
2514 return p;
2515 }
2516
2517
2518 // Set up first user process.
2519 void
2520 userinit(void)
2521 {
2522     struct proc *p;
2523     extern char _binary_initcode_start[], _binary_initcode_size[];
2524
2525     p = allocproc();
2526
2527     initproc = p;
2528     if((p->pgdir = setupkvm()) == 0)
2529         panic("userinit: out of memory?");
2530     inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2531     p->sz = PGSIZE;
2532     memset(p->tf, 0, sizeof(*p->tf));
2533     p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2534     p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2535     p->tf->es = p->tf->ds;
2536     p->tf->ss = p->tf->ds;
2537     p->tf->eflags = FL_IF;
2538     p->tf->esp = PGSIZE;
2539     p->tf->eip = 0; // beginning of initcode.S
2540
2541     safestrcpy(p->name, "initcode", sizeof(p->name));
2542     p->cwd = namei("/");
2543
2544     // this assignment to p->state lets other cores
2545     // run this process. the acquire forces the above
2546     // writes to be visible, and the lock is also needed
2547     // because the assignment might not be atomic.
2548     acquire(&ptable.lock);
2549

```

```

2550 p->state = RUNNABLE;
2551
2552 release(&ptable.lock);
2553 }
2554
2555 // Grow current process's memory by n bytes.
2556 // Return 0 on success, -1 on failure.
2557 int
2558 growproc(int n)
2559 {
2560     uint sz;
2561     struct proc *curproc = myproc();
2562
2563     sz = curproc->sz;
2564     if(n > 0){
2565         if((sz = allocvm(curproc->pgdir, sz, sz + n)) == 0)
2566             return -1;
2567     } else if(n < 0){
2568         if((sz = deallocvm(curproc->pgdir, sz, sz + n)) == 0)
2569             return -1;
2570     }
2571     curproc->sz = sz;
2572     switchvm(curproc);
2573     return 0;
2574 }
2575
2576 // Create a new process copying p as the parent.
2577 // Sets up stack to return as if from system call.
2578 // Caller must set state of returned proc to RUNNABLE.
2579 int
2580 fork(void)
2581 {
2582     int i, pid;
2583     struct proc *np;
2584     struct proc *curproc = myproc();
2585
2586     // Allocate process.
2587     if((np = allocproc()) == 0){
2588         return -1;
2589     }
2590
2591     // Copy process state from proc.
2592     if((np->pgdir = copyvm(curproc->pgdir, curproc->sz)) == 0){
2593         kfree(np->kstack);
2594         np->kstack = 0;
2595         np->state = UNUSED;
2596         return -1;
2597     }
2598     np->sz = curproc->sz;
2599     np->parent = curproc;

```

```

2600 *np->tf = *curproc->tf;
2601
2602 // Clear %eax so that fork returns 0 in the child.
2603 np->tf->eax = 0;
2604
2605 for(i = 0; i < NOFILE; i++){
2606     if(curproc->ofile[i])
2607         np->ofile[i] = filedup(curproc->ofile[i]);
2608     np->cwd = idup(curproc->cwd);
2609
2610     safestrcpy(np->name, curproc->name, sizeof(curproc->name));
2611
2612     pid = np->pid;
2613
2614     acquire(&ptable.lock);
2615
2616     np->state = RUNNABLE;
2617
2618     release(&ptable.lock);
2619
2620     return pid;
2621 }
2622
2623 // Exit the current process. Does not return.
2624 // An exited process remains in the zombie state
2625 // until its parent calls wait() to find out it exited.
2626 void
2627 exit(void)
2628 {
2629     struct proc *curproc = myproc();
2630     struct proc *p;
2631     int fd;
2632
2633     if(curproc == initproc)
2634         panic("init exiting");
2635
2636     // Close all open files.
2637     for(fd = 0; fd < NOFILE; fd++){
2638         if(curproc->ofile[fd]){
2639             fileclose(curproc->ofile[fd]);
2640             curproc->ofile[fd] = 0;
2641         }
2642     }
2643
2644     begin_op();
2645     iput(curproc->cwd);
2646     end_op();
2647     curproc->cwd = 0;
2648
2649     acquire(&ptable.lock);

```

```

2650 // Parent might be sleeping in wait().
2651 wakeup1(curproc->parent);
2652
2653 // Pass abandoned children to init.
2654 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2655     if(p->parent == curproc){
2656         p->parent = initproc;
2657         if(p->state == ZOMBIE)
2658             wakeup1(initproc);
2659     }
2660 }
2661
2662 // Jump into the scheduler, never to return.
2663 curproc->state = ZOMBIE;
2664 sched();
2665 panic("zombie exit");
2666 }
2667
2668 // Wait for a child process to exit and return its pid.
2669 // Return -1 if this process has no children.
2670 int
2671 wait(void)
2672 {
2673     struct proc *p;
2674     int havekids, pid;
2675     struct proc *curproc = myproc();
2676
2677     acquire(&ptable.lock);
2678     for(;;){
2679         // Scan through table looking for exited children.
2680         havekids = 0;
2681         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2682             if(p->parent != curproc)
2683                 continue;
2684             havekids = 1;
2685             if(p->state == ZOMBIE){
2686                 // Found one.
2687                 pid = p->pid;
2688                 kfree(p->kstack);
2689                 p->kstack = 0;
2690                 freevm(p->pgdir);
2691                 p->pid = 0;
2692                 p->parent = 0;
2693                 p->name[0] = 0;
2694                 p->killed = 0;
2695                 p->state = UNUSED;
2696                 release(&ptable.lock);
2697                 return pid;
2698             }
2699         }

```

```

2700 // No point waiting if we don't have any children.
2701 if(!havekids || curproc->killed){
2702     release(&ptable.lock);
2703     return -1;
2704 }
2705
2706 // Wait for children to exit. (See wakeup1 call in proc_exit.)
2707 sleep(curproc, &ptable.lock);
2708 }
2709 }
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749

```

```

2750 // Per-CPU process scheduler.
2751 // Each CPU calls scheduler() after setting itself up.
2752 // Scheduler never returns. It loops, doing:
2753 // - choose a process to run
2754 // - switch to start running that process
2755 // - eventually that process transfers control
2756 //   via switch back to the scheduler.
2757 void
2758 scheduler(void)
2759 {
2760     struct proc *p;
2761     struct cpu *c = mycpu();
2762     c->proc = 0;
2763
2764     for(;;){
2765         // Enable interrupts on this processor.
2766         sti();
2767
2768         // Loop over process table looking for process to run.
2769         acquire(&ptable.lock);
2770         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2771             if(p->state != RUNNABLE)
2772                 continue;
2773
2774             // Switch to chosen process. It is the process's job
2775             // to release ptable.lock and then reacquire it
2776             // before jumping back to us.
2777             c->proc = p;
2778             switchvm(p);
2779             p->state = RUNNING;
2780
2781             swtch(&(c->scheduler), p->context);
2782             switchkvm();
2783
2784             // Process is done running for now.
2785             // It should have changed its p->state before coming back.
2786             c->proc = 0;
2787         }
2788         release(&ptable.lock);
2789     }
2790 }
2791 }
2792
2793
2794
2795
2796
2797
2798
2799

```

```

2800 // Enter scheduler. Must hold only ptable.lock
2801 // and have changed proc->state. Saves and restores
2802 // intena because intena is a property of this
2803 // kernel thread, not this CPU. It should
2804 // be proc->intena and proc->ncli, but that would
2805 // break in the few places where a lock is held but
2806 // there's no process.
2807 void
2808 sched(void)
2809 {
2810     int intena;
2811     struct proc *p = myproc();
2812
2813     if(!holding(&ptable.lock))
2814         panic("sched ptable.lock");
2815     if(mycpu()->ncli != 1)
2816         panic("sched locks");
2817     if(p->state == RUNNING)
2818         panic("sched running");
2819     if(readeflags() & FL_IF)
2820         panic("sched interruptible");
2821     intena = mycpu()->intena;
2822     swtch(&p->context, mycpu()->scheduler);
2823     mycpu()->intena = intena;
2824 }
2825
2826 // Give up the CPU for one scheduling round.
2827 void
2828 yield(void)
2829 {
2830     acquire(&ptable.lock);
2831     myproc()->state = RUNNABLE;
2832     sched();
2833     release(&ptable.lock);
2834 }
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849

```

```

2850 // A fork child's very first scheduling by scheduler()
2851 // will swtch here. "Return" to user space.
2852 void
2853 forkret(void)
2854 {
2855     static int first = 1;
2856     // Still holding ptable.lock from scheduler.
2857     release(&ptable.lock);
2858
2859     if (first) {
2860         // Some initialization functions must be run in the context
2861         // of a regular process (e.g., they call sleep), and thus cannot
2862         // be run from main().
2863         first = 0;
2864         iinit(ROOTDEV);
2865         initlog(ROOTDEV);
2866     }
2867
2868     // Return to "caller", actually trapret (see allocproc).
2869 }
2870
2871 // Atomically release lock and sleep on chan.
2872 // Reacquires lock when awakened.
2873 void
2874 sleep(void *chan, struct spinlock *lk)
2875 {
2876     struct proc *p = myproc();
2877
2878     if (p == 0)
2879         panic("sleep");
2880
2881     if (lk == 0)
2882         panic("sleep without lk");
2883
2884     // Must acquire ptable.lock in order to
2885     // change p->state and then call sched.
2886     // Once we hold ptable.lock, we can be
2887     // guaranteed that we won't miss any wakeup
2888     // (wakeup runs with ptable.lock locked),
2889     // so it's okay to release lk.
2890     if (lk != &ptable.lock) {
2891         acquire(&ptable.lock);
2892         release(lk);
2893     }
2894     // Go to sleep.
2895     p->chan = chan;
2896     p->state = SLEEPING;
2897
2898     sched();
2899

```

```

2900 // Tidy up.
2901 p->chan = 0;
2902
2903 // Reacquire original lock.
2904 if (lk != &ptable.lock) {
2905     release(&ptable.lock);
2906     acquire(lk);
2907 }
2908 }
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949

```

```

2950 // Wake up all processes sleeping on chan.
2951 // The ptable lock must be held.
2952 static void
2953 wakeup1(void *chan)
2954 {
2955     struct proc *p;
2956     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2957         if(p->state == SLEEPING && p->chan == chan)
2958             p->state = RUNNABLE;
2960 }
2961
2962 // Wake up all processes sleeping on chan.
2963 void
2964 wakeup(void *chan)
2965 {
2966     acquire(&ptable.lock);
2967     wakeup1(chan);
2968     release(&ptable.lock);
2969 }
2970
2971 // Kill the process with the given pid.
2972 // Process won't exit until it returns
2973 // to user space (see trap in trap.c).
2974 int
2975 kill(int pid)
2976 {
2977     struct proc *p;
2978     acquire(&ptable.lock);
2979     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2980         if(p->pid == pid){
2981             p->killed = 1;
2982             // Wake process from sleep if necessary.
2983             if(p->state == SLEEPING)
2984                 p->state = RUNNABLE;
2985             release(&ptable.lock);
2986             return 0;
2987         }
2988     }
2989     release(&ptable.lock);
2990     return -1;
2991 }
2992
2993
2994
2995
2996
2997
2998
2999

```

```

3000 // Print a process listing to console. For debugging.
3001 // Runs when user types ^P on console.
3002 // No lock to avoid wedging a stuck machine further.
3003 void
3004 procdump(void)
3005 {
3006     static char *states[] = {
3007         [UNUSED]    "unused",
3008         [EMBRYO]    "embryo",
3009         [SLEEPING]  "sleep ",
3010         [RUNNABLE]  "runble",
3011         [RUNNING]   "run   ",
3012         [ZOMBIE]    "zombie"
3013     };
3014     int i;
3015     struct proc *p;
3016     char *state;
3017     uint pc[10];
3018
3019     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3020         if(p->state == UNUSED)
3021             continue;
3022         if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
3023             state = states[p->state];
3024         else
3025             state = "???";
3026         cprintf("%d %s %s", p->pid, state, p->name);
3027         if(p->state == SLEEPING){
3028             getcallerpcs((uint*)p->context->ebp+2, pc);
3029             for(i=0; i<10 && pc[i] != 0; i++)
3030                 cprintf(" %p", pc[i]);
3031         }
3032         cprintf("\n");
3033     }
3034 }
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049

```

```

3050 # Context switch
3051 #
3052 # void swtch(struct context **old, struct context *new);
3053 #
3054 # Save the current registers on the stack, creating
3055 # a struct context, and save its address in *old.
3056 # Switch stacks to new and pop previously-saved registers.
3057
3058 .globl swtch
3059 swtch:
3060     movl 4(%esp), %eax
3061     movl 8(%esp), %edx
3062
3063     # Save old callee-saved registers
3064     pushl %ebp
3065     pushl %ebx
3066     pushl %esi
3067     pushl %edi
3068
3069     # Switch stacks
3070     movl %esp, (%eax)
3071     movl %edx, %esp
3072
3073     # Load new callee-saved registers
3074     popl %edi
3075     popl %esi
3076     popl %ebx
3077     popl %ebp
3078     ret
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099

```

```

3100 // Physical memory allocator, intended to allocate
3101 // memory for user processes, kernel stacks, page table pages,
3102 // and pipe buffers. Allocates 4096-byte pages.
3103
3104 #include "types.h"
3105 #include "defs.h"
3106 #include "param.h"
3107 #include "memlayout.h"
3108 #include "mmu.h"
3109 #include "spinlock.h"
3110
3111 void freerange(void *vstart, void *vend);
3112 extern char end[]; // first address after kernel loaded from ELF file
3113                    // defined by the kernel linker script in kernel.ld
3114
3115 struct run {
3116     struct run *next;
3117 };
3118
3119 struct {
3120     struct spinlock lock;
3121     int use_lock;
3122     struct run *freelist;
3123 } kmem;
3124
3125 // Initialization happens in two phases.
3126 // 1. main() calls kinit1() while still using entrypgdir to place just
3127 // the pages mapped by entrypgdir on free list.
3128 // 2. main() calls kinit2() with the rest of the physical pages
3129 // after installing a full page table that maps them on all cores.
3130 void
3131 kinit1(void *vstart, void *vend)
3132 {
3133     initlock(&kmem.lock, "kmem");
3134     kmem.use_lock = 0;
3135     freerange(vstart, vend);
3136 }
3137
3138 void
3139 kinit2(void *vstart, void *vend)
3140 {
3141     freerange(vstart, vend);
3142     kmem.use_lock = 1;
3143 }
3144
3145
3146
3147
3148
3149

```



```

3150 void
3151 freerange(void *vstart, void *vend)
3152 {
3153     char *p;
3154     p = (char*)PGROUNDUP((uint)vstart);
3155     for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3156         kfree(p);
3157 }
3158
3159 // Free the page of physical memory pointed at by v,
3160 // which normally should have been returned by a
3161 // call to kalloc(). (The exception is when
3162 // initializing the allocator; see kinit above.)
3163 void
3164 kfree(char *v)
3165 {
3166     struct run *r;
3167
3168     if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
3169         panic("kfree");
3170
3171     // Fill with junk to catch dangling refs.
3172     memset(v, 1, PGSIZE);
3173
3174     if(kmem.use_lock)
3175         acquire(&kmem.lock);
3176     r = (struct run*)v;
3177     r->next = kmem.freelist;
3178     kmem.freelist = r;
3179     if(kmem.use_lock)
3180         release(&kmem.lock);
3181 }
3182
3183 // Allocate one 4096-byte page of physical memory.
3184 // Returns a pointer that the kernel can use.
3185 // Returns 0 if the memory cannot be allocated.
3186 char*
3187 kalloc(void)
3188 {
3189     struct run *r;
3190
3191     if(kmem.use_lock)
3192         acquire(&kmem.lock);
3193     r = kmem.freelist;
3194     if(r)
3195         kmem.freelist = r->next;
3196     if(kmem.use_lock)
3197         release(&kmem.lock);
3198     return (char*)r;
3199 }

```

```

3200 // x86 trap and interrupt constants.
3201
3202 // Processor-defined:
3203 #define T_DIVIDE 0 // divide error
3204 #define T_DEBUG 1 // debug exception
3205 #define T_NMI 2 // non-maskable interrupt
3206 #define T_BRKPT 3 // breakpoint
3207 #define T_OFLOW 4 // overflow
3208 #define T_BOUND 5 // bounds check
3209 #define T_ILLOP 6 // illegal opcode
3210 #define T_DEVICE 7 // device not available
3211 #define T_DBLFLT 8 // double fault
3212 // #define T_COPROC 9 // reserved (not used since 486)
3213 #define T_TSS 10 // invalid task switch segment
3214 #define T_SEGNP 11 // segment not present
3215 #define T_STACK 12 // stack exception
3216 #define T_GPFLT 13 // general protection fault
3217 #define T_PGFLT 14 // page fault
3218 // #define T_RES 15 // reserved
3219 #define T_FPERR 16 // floating point error
3220 #define T_ALIGN 17 // alignment check
3221 #define T_MCHK 18 // machine check
3222 #define T_SIMDERR 19 // SIMD floating point error
3223
3224 // These are arbitrarily chosen, but with care not to overlap
3225 // processor defined exceptions or interrupt vectors.
3226 #define T_SYSCALL 64 // system call
3227 #define T_DEFAULT 500 // catchall
3228
3229 #define T_IRQ0 32 // IRQ 0 corresponds to int T_IRQ
3230
3231 #define IRQ_TIMER 0
3232 #define IRQ_KBD 1
3233 #define IRQ_COM1 4
3234 #define IRQ_IDE 14
3235 #define IRQ_ERROR 19
3236 #define IRQ_SPURIOUS 31
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249

```

```

3250 #!/usr/bin/perl -w
3251
3252 # Generate vectors.S, the trap/interrupt entry points.
3253 # There has to be one entry point per interrupt number
3254 # since otherwise there's no way for trap() to discover
3255 # the interrupt number.
3256
3257 print "# generated by vectors.pl - do not edit\n";
3258 print "# handlers\n";
3259 print ".globl alltraps\n";
3260 for(my $i = 0; $i < 256; $i++){
3261     print ".globl vector$i\n";
3262     print "vector$i:\n";
3263     if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
3264         print "    pushl $0\n";
3265     }
3266     print "    pushl $$i\n";
3267     print "    jmp alltraps\n";
3268 }
3269
3270 print "\n# vector table\n";
3271 print ".data\n";
3272 print ".globl vectors\n";
3273 print "vectors:\n";
3274 for(my $i = 0; $i < 256; $i++){
3275     print "    .long vector$i\n";
3276 }
3277
3278 # sample output:
3279 # # handlers
3280 # .globl alltraps
3281 # .globl vector0
3282 # vector0:
3283 #     pushl $0
3284 #     pushl $0
3285 #     jmp alltraps
3286 # ...
3287 #
3288 # # vector table
3289 # .data
3290 # .globl vectors
3291 # vectors:
3292 #     .long vector0
3293 #     .long vector1
3294 #     .long vector2
3295 # ...
3296
3297
3298
3299

```

```

3300 #include "mmu.h"
3301
3302 # vectors.S sends all traps here.
3303 .globl alltraps
3304 alltraps:
3305     # Build trap frame.
3306     pushl %ds
3307     pushl %es
3308     pushl %fs
3309     pushl %gs
3310     pushal
3311
3312     # Set up data segments.
3313     movw $(SEG_KDATA<<3), %ax
3314     movw %ax, %ds
3315     movw %ax, %es
3316
3317     # Call trap(tf), where tf=%esp
3318     pushl %esp
3319     call trap
3320     addl $4, %esp
3321
3322     # Return falls through to trapret...
3323 .globl trapret
3324 trapret:
3325     popal
3326     popl %gs
3327     popl %fs
3328     popl %es
3329     popl %ds
3330     addl $0x8, %esp # trapno and errcode
3331     iret
3332
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349

```

```

3350 #include "types.h"
3351 #include "defs.h"
3352 #include "param.h"
3353 #include "memlayout.h"
3354 #include "mmu.h"
3355 #include "proc.h"
3356 #include "x86.h"
3357 #include "traps.h"
3358 #include "spinlock.h"
3359
3360 // Interrupt descriptor table (shared by all CPUs).
3361 struct gatedesc idt[256];
3362 extern uint vectors[]; // in vectors.S: array of 256 entry pointers
3363 struct spinlock tickslock;
3364 uint ticks;
3365
3366 void
3367 tvinit(void)
3368 {
3369     int i;
3370
3371     for(i = 0; i < 256; i++)
3372         SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3373     SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3374
3375     initlock(&tickslock, "time");
3376 }
3377
3378 void
3379 idtinit(void)
3380 {
3381     lidt(idt, sizeof(idt));
3382 }
3383
3384
3385
3386
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399

```

```

3400 void
3401 trap(struct trapframe *tf)
3402 {
3403     if(tf->trapno == T_SYSCALL){
3404         if(myproc()->killed)
3405             exit();
3406         myproc()->tf = tf;
3407         syscall();
3408         if(myproc()->killed)
3409             exit();
3410         return;
3411     }
3412
3413     switch(tf->trapno){
3414     case T_IRQ0 + IRQ_TIMER:
3415         if(cpuid() == 0){
3416             acquire(&tickslock);
3417             ticks++;
3418             wakeup(&ticks);
3419             release(&tickslock);
3420         }
3421         lapiceoi();
3422         break;
3423     case T_IRQ0 + IRQ_IDE:
3424         ideintr();
3425         lapiceoi();
3426         break;
3427     case T_IRQ0 + IRQ_IDE+1:
3428         // Bochs generates spurious IDE1 interrupts.
3429         break;
3430     case T_IRQ0 + IRQ_KBD:
3431         kbdintr();
3432         lapiceoi();
3433         break;
3434     case T_IRQ0 + IRQ_COM1:
3435         uartintr();
3436         lapiceoi();
3437         break;
3438     case T_IRQ0 + 7:
3439     case T_IRQ0 + IRQ_SPURIOUS:
3440         cprintf("cpu%d: spurious interrupt at %x:%x\n",
3441             cpuid(), tf->cs, tf->eip);
3442         lapiceoi();
3443         break;
3444
3445
3446
3447
3448
3449

```

```

3450 default:
3451     if(myproc() == 0 || (tf->cs&3) == 0){
3452         // In kernel, it must be our mistake.
3453         cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3454             tf->trapno, cpuid(), tf->eip, rcr2());
3455         panic("trap");
3456     }
3457     // In user space, assume process misbehaved.
3458     cprintf("pid %d %s: trap %d err %d on cpu %d "
3459         "eip 0x%x addr 0x%x--kill proc\n",
3460         myproc()->pid, myproc()->name, tf->trapno,
3461         tf->err, cpuid(), tf->eip, rcr2());
3462     myproc()->killed = 1;
3463 }
3464
3465 // Force process exit if it has been killed and is in user space.
3466 // (If it is still executing in the kernel, let it keep running
3467 // until it gets to the regular system call return.)
3468 if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
3469     exit();
3470
3471 // Force process to give up CPU on clock tick.
3472 // If interrupts were on while locks held, would need to check nlock.
3473 if(myproc() && myproc()->state == RUNNING &&
3474     tf->trapno == T_IRQ0+IRQ_TIMER)
3475     yield();
3476
3477 // Check if the process has been killed since we yielded
3478 if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
3479     exit();
3480 }
3481
3482
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499

```

```

3500 // System call numbers
3501 #define SYS_fork    1
3502 #define SYS_exit    2
3503 #define SYS_wait    3
3504 #define SYS_pipe    4
3505 #define SYS_read    5
3506 #define SYS_kill    6
3507 #define SYS_exec    7
3508 #define SYS_fstat   8
3509 #define SYS_chdir   9
3510 #define SYS_dup     10
3511 #define SYS_getpid  11
3512 #define SYS_sbrk    12
3513 #define SYS_sleep   13
3514 #define SYS_uptime  14
3515 #define SYS_open    15
3516 #define SYS_write   16
3517 #define SYS_mknod   17
3518 #define SYS_unlink  18
3519 #define SYS_link    19
3520 #define SYS_mkdir   20
3521 #define SYS_close   21
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 #include "types.h"
3551 #include "defs.h"
3552 #include "param.h"
3553 #include "memlayout.h"
3554 #include "mmu.h"
3555 #include "proc.h"
3556 #include "x86.h"
3557 #include "syscall.h"
3558
3559 // User code makes a system call with INT T_SYSCALL.
3560 // System call number in %eax.
3561 // Arguments on the stack, from the user call to the C
3562 // library system call function. The saved user %esp points
3563 // to a saved program counter, and then the first argument.
3564
3565 // Fetch the int at addr from the current process.
3566 int
3567 fetchint(uint addr, int *ip)
3568 {
3569     struct proc *curproc = myproc();
3570
3571     if(addr >= curproc->sz || addr+4 > curproc->sz)
3572         return -1;
3573     *ip = *(int*)(addr);
3574     return 0;
3575 }
3576
3577 // Fetch the nul-terminated string at addr from the current process.
3578 // Doesn't actually copy the string - just sets *pp to point at it.
3579 // Returns length of string, not including nul.
3580 int
3581 fetchstr(uint addr, char **pp)
3582 {
3583     char *s, *ep;
3584     struct proc *curproc = myproc();
3585
3586     if(addr >= curproc->sz)
3587         return -1;
3588     *pp = (char*)addr;
3589     ep = (char*)curproc->sz;
3590     for(s = *pp; s < ep; s++){
3591         if(*s == 0)
3592             return s - *pp;
3593     }
3594     return -1;
3595 }
3596
3597
3598
3599

```

```

3600 // Fetch the nth 32-bit system call argument.
3601 int
3602 argint(int n, int *ip)
3603 {
3604     return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
3605 }
3606
3607 // Fetch the nth word-sized system call argument as a pointer
3608 // to a block of memory of size bytes. Check that the pointer
3609 // lies within the process address space.
3610 int
3611 argptr(int n, char **pp, int size)
3612 {
3613     int i;
3614     struct proc *curproc = myproc();
3615
3616     if(argint(n, &i) < 0)
3617         return -1;
3618     if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
3619         return -1;
3620     *pp = (char*)i;
3621     return 0;
3622 }
3623
3624 // Fetch the nth word-sized system call argument as a string pointer.
3625 // Check that the pointer is valid and the string is nul-terminated.
3626 // (There is no shared writable memory, so the string can't change
3627 // between this check and being used by the kernel.)
3628 int
3629 argstr(int n, char **pp)
3630 {
3631     int addr;
3632     if(argint(n, &addr) < 0)
3633         return -1;
3634     return fetchstr(addr, pp);
3635 }
3636
3637
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649

```

```

3650 extern int sys_chdir(void);
3651 extern int sys_close(void);
3652 extern int sys_dup(void);
3653 extern int sys_exec(void);
3654 extern int sys_exit(void);
3655 extern int sys_fork(void);
3656 extern int sys_fstat(void);
3657 extern int sys_getpid(void);
3658 extern int sys_kill(void);
3659 extern int sys_link(void);
3660 extern int sys_mkdir(void);
3661 extern int sys_mknod(void);
3662 extern int sys_open(void);
3663 extern int sys_pipe(void);
3664 extern int sys_read(void);
3665 extern int sys_sbrk(void);
3666 extern int sys_sleep(void);
3667 extern int sys_unlink(void);
3668 extern int sys_wait(void);
3669 extern int sys_write(void);
3670 extern int sys_uptime(void);
3671
3672 static int (*syscalls[])(void) = {
3673 [SYS_fork]    sys_fork,
3674 [SYS_exit]    sys_exit,
3675 [SYS_wait]    sys_wait,
3676 [SYS_pipe]    sys_pipe,
3677 [SYS_read]    sys_read,
3678 [SYS_kill]    sys_kill,
3679 [SYS_exec]    sys_exec,
3680 [SYS_fstat]   sys_fstat,
3681 [SYS_chdir]   sys_chdir,
3682 [SYS_dup]     sys_dup,
3683 [SYS_getpid]  sys_getpid,
3684 [SYS_sbrk]    sys_sbrk,
3685 [SYS_sleep]   sys_sleep,
3686 [SYS_uptime]  sys_uptime,
3687 [SYS_open]    sys_open,
3688 [SYS_write]   sys_write,
3689 [SYS_mknod]   sys_mknod,
3690 [SYS_unlink]  sys_unlink,
3691 [SYS_link]    sys_link,
3692 [SYS_mkdir]   sys_mkdir,
3693 [SYS_close]   sys_close,
3694 };
3695
3696
3697
3698
3699

```

```

3700 void
3701 syscall(void)
3702 {
3703     int num;
3704     struct proc *curproc = myproc();
3705
3706     num = curproc->tf->eax;
3707     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3708         curproc->tf->eax = syscalls[num]();
3709     } else {
3710         cprintf("%d %s: unknown sys call %d\n",
3711             curproc->pid, curproc->name, num);
3712         curproc->tf->eax = -1;
3713     }
3714 }
3715
3716
3717
3718
3719
3720
3721
3722
3723
3724
3725
3726
3727
3728
3729
3730
3731
3732
3733
3734
3735
3736
3737
3738
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3749

```

```

3750 #include "types.h"
3751 #include "x86.h"
3752 #include "defs.h"
3753 #include "date.h"
3754 #include "param.h"
3755 #include "memlayout.h"
3756 #include "mmu.h"
3757 #include "proc.h"
3758
3759 int
3760 sys_fork(void)
3761 {
3762     return fork();
3763 }
3764
3765 int
3766 sys_exit(void)
3767 {
3768     exit();
3769     return 0; // not reached
3770 }
3771
3772 int
3773 sys_wait(void)
3774 {
3775     return wait();
3776 }
3777
3778 int
3779 sys_kill(void)
3780 {
3781     int pid;
3782
3783     if(argint(0, &pid) < 0)
3784         return -1;
3785     return kill(pid);
3786 }
3787
3788 int
3789 sys_getpid(void)
3790 {
3791     return myproc()->pid;
3792 }
3793
3794
3795
3796
3797
3798
3799

```

```

3800 int
3801 sys_sbrk(void)
3802 {
3803     int addr;
3804     int n;
3805
3806     if(argint(0, &n) < 0)
3807         return -1;
3808     addr = myproc()->sz;
3809     if(growproc(n) < 0)
3810         return -1;
3811     return addr;
3812 }
3813
3814 int
3815 sys_sleep(void)
3816 {
3817     int n;
3818     uint ticks0;
3819
3820     if(argint(0, &n) < 0)
3821         return -1;
3822     acquire(&tickslock);
3823     ticks0 = ticks;
3824     while(ticks - ticks0 < n){
3825         if(myproc()->killed){
3826             release(&tickslock);
3827             return -1;
3828         }
3829         sleep(&ticks, &tickslock);
3830     }
3831     release(&tickslock);
3832     return 0;
3833 }
3834
3835 // return how many clock tick interrupts have occurred
3836 // since start.
3837 int
3838 sys_uptime(void)
3839 {
3840     uint xticks;
3841
3842     acquire(&tickslock);
3843     xticks = ticks;
3844     release(&tickslock);
3845     return xticks;
3846 }
3847
3848
3849

```

```
3850 struct buf {
3851     int flags;
3852     uint dev;
3853     uint blockno;
3854     struct sleeplock lock;
3855     uint refcnt;
3856     struct buf *prev; // LRU cache list
3857     struct buf *next;
3858     struct buf *qnext; // disk queue
3859     uchar data[BSIZE];
3860 };
3861 #define B_VALID 0x2 // buffer has been read from disk
3862 #define B_DIRTY 0x4 // buffer needs to be written to disk
3863
3864
3865
3866
3867
3868
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
```

```
3900 // Long-term locks for processes
3901 struct sleeplock {
3902     uint locked; // Is the lock held?
3903     struct spinlock lk; // spinlock protecting this sleep lock
3904
3905     // For debugging:
3906     char *name; // Name of lock.
3907     int pid; // Process holding lock
3908 };
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
```



```
3950 #define O_RDONLY 0x000
3951 #define O_WRONLY 0x001
3952 #define O_RDWR 0x002
3953 #define O_CREATE 0x200
3954
3955
3956
3957
3958
3959
3960
3961
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
```

```
4000 #define T_DIR 1 // Directory
4001 #define T_FILE 2 // File
4002 #define T_DEV 3 // Device
4003
4004 struct stat {
4005     short type; // Type of file
4006     int dev; // File system's disk device
4007     uint ino; // Inode number
4008     short nlink; // Number of links to file
4009     uint size; // Size of file in bytes
4010 };
4011
4012
4013
4014
4015
4016
4017
4018
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
```

```

4050 // On-disk file system format.
4051 // Both the kernel and user programs use this header file.
4052
4053
4054 #define ROOTINO 1 // root i-number
4055 #define BSIZE 512 // block size
4056
4057 // Disk layout:
4058 // [ boot block | super block | log | inode blocks |
4059 //                               free bit map | data blocks]
4060 //
4061 // mkfs computes the super block and builds an initial file system. The
4062 // super block describes the disk layout:
4063 struct superblock {
4064     uint size;           // Size of file system image (blocks)
4065     uint nblocks;        // Number of data blocks
4066     uint ninodes;        // Number of inodes.
4067     uint nlog;           // Number of log blocks
4068     uint logstart;       // Block number of first log block
4069     uint inodestart;     // Block number of first inode block
4070     uint bmapstart;      // Block number of first free map block
4071 };
4072
4073 #define NDIRECT 12
4074 #define NINDIRECT (BSIZE / sizeof(uint))
4075 #define MAXFILE (NDIRECT + NINDIRECT)
4076
4077 // On-disk inode structure
4078 struct dinode {
4079     short type;          // File type
4080     short major;         // Major device number (T_DEV only)
4081     short minor;         // Minor device number (T_DEV only)
4082     short nlink;         // Number of links to inode in file system
4083     uint size;           // Size of file (bytes)
4084     uint addrs[NDIRECT+1]; // Data block addresses
4085 };
4086
4087
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099

```

```

4100 // Inodes per block.
4101 #define IPB (BSIZE / sizeof(struct dinode))
4102
4103 // Block containing inode i
4104 #define IBLOCK(i, sb) ((i) / IPB + sb.inodestart)
4105
4106 // Bitmap bits per block
4107 #define BPB (BSIZE*8)
4108
4109 // Block of free map containing bit for block b
4110 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
4111
4112 // Directory is a file containing a sequence of dirent structures.
4113 #define DIRSIZ 14
4114
4115 struct dirent {
4116     ushort inum;
4117     char name[DIRSIZ];
4118 };
4119
4120
4121
4122
4123
4124
4125
4126
4127
4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149

```

```

4150 struct file {
4151     enum { FD_NONE, FD_PIPE, FD_INODE } type;
4152     int ref; // reference count
4153     char readable;
4154     char writable;
4155     struct pipe *pipe;
4156     struct inode *ip;
4157     uint off;
4158 };
4159
4160
4161 // in-memory copy of an inode
4162 struct inode {
4163     uint dev;           // Device number
4164     uint inum;          // Inode number
4165     int ref;            // Reference count
4166     struct sleeplock lock; // protects everything below here
4167     int valid;          // inode has been read from disk?
4168
4169     short type;         // copy of disk inode
4170     short major;
4171     short minor;
4172     short nlink;
4173     uint size;
4174     uint addrs[NDIRECT+1];
4175 };
4176
4177 // table mapping major device number to
4178 // device functions
4179 struct devsw {
4180     int (*read)(struct inode*, char*, int);
4181     int (*write)(struct inode*, char*, int);
4182 };
4183
4184 extern struct devsw devsw[];
4185
4186 #define CONSOLE 1
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199

```

```

4200 // Simple PIO-based (non-DMA) IDE driver code.
4201
4202 #include "types.h"
4203 #include "defs.h"
4204 #include "param.h"
4205 #include "memlayout.h"
4206 #include "mmu.h"
4207 #include "proc.h"
4208 #include "x86.h"
4209 #include "traps.h"
4210 #include "spinlock.h"
4211 #include "sleeplock.h"
4212 #include "fs.h"
4213 #include "buf.h"
4214
4215 #define SECTOR_SIZE 512
4216 #define IDE_BSY 0x80
4217 #define IDE_DRDY 0x40
4218 #define IDE_DF 0x20
4219 #define IDE_ERR 0x01
4220
4221 #define IDE_CMD_READ 0x20
4222 #define IDE_CMD_WRITE 0x30
4223 #define IDE_CMD_RDMDL 0xc4
4224 #define IDE_CMD_WRMDL 0xc5
4225
4226 // idequeue points to the buf now being read/written to the disk.
4227 // idequeue->qnext points to the next buf to be processed.
4228 // You must hold idelock while manipulating queue.
4229
4230 static struct spinlock idelock;
4231 static struct buf *idequeue;
4232
4233 static int havdisk1;
4234 static void idestart(struct buf*);
4235
4236 // Wait for IDE disk to become ready.
4237 static int
4238 idewait(int checkerr)
4239 {
4240     int r;
4241
4242     while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4243         ;
4244     if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4245         return -1;
4246     return 0;
4247 }
4248
4249

```

```

4250 void
4251 ideinit(void)
4252 {
4253     int i;
4254
4255     initlock(&idelock, "ide");
4256     ioapicenable(IRQ_IDE, ncpu - 1);
4257     idewait(0);
4258
4259     // Check if disk 1 is present
4260     outb(0x1f6, 0xe0 | (1<<4));
4261     for(i=0; i<1000; i++){
4262         if(inb(0x1f7) != 0){
4263             havedisk1 = 1;
4264             break;
4265         }
4266     }
4267
4268     // Switch back to disk 0.
4269     outb(0x1f6, 0xe0 | (0<<4));
4270 }
4271
4272 // Start the request for b. Caller must hold idelock.
4273 static void
4274 idestart(struct buf *b)
4275 {
4276     if(b == 0)
4277         panic("idestart");
4278     if(b->blockno >= FSSIZE)
4279         panic("incorrect blockno");
4280     int sector_per_block = BSIZE/SECTOR_SIZE;
4281     int sector = b->blockno * sector_per_block;
4282     int read_cmd = (sector_per_block == 1) ? IDE_CMD_READ : IDE_CMD_RDMDL;
4283     int write_cmd = (sector_per_block == 1) ? IDE_CMD_WRITE : IDE_CMD_WRMDL;
4284
4285     if (sector_per_block > 7) panic("idestart");
4286
4287     idewait(0);
4288     outb(0x3f6, 0); // generate interrupt
4289     outb(0x1f2, sector_per_block); // number of sectors
4290     outb(0x1f3, sector & 0xff);
4291     outb(0x1f4, (sector >> 8) & 0xff);
4292     outb(0x1f5, (sector >> 16) & 0xff);
4293     outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4294     if(b->flags & B_DIRTY){
4295         outb(0x1f7, write_cmd);
4296         outsl(0x1f0, b->data, BSIZE/4);
4297     } else {
4298         outb(0x1f7, read_cmd);
4299     }

```

```

4300 }
4301
4302 // Interrupt handler.
4303 void
4304 ideintr(void)
4305 {
4306     struct buf *b;
4307
4308     // First queued buffer is the active request.
4309     acquire(&idelock);
4310
4311     if((b = idequeue) == 0){
4312         release(&idelock);
4313         return;
4314     }
4315     idequeue = b->qnext;
4316
4317     // Read data if needed.
4318     if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
4319         insl(0x1f0, b->data, BSIZE/4);
4320
4321     // Wake process waiting for this buf.
4322     b->flags |= B_VALID;
4323     b->flags &= ~B_DIRTY;
4324     wakeup(b);
4325
4326     // Start disk on next buf in queue.
4327     if(idequeue != 0)
4328         idestart(idequeue);
4329
4330     release(&idelock);
4331 }
4332
4333
4334
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349

```

```

4350 // Sync buf with disk.
4351 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
4352 // Else if B_VALID is not set, read buf from disk, set B_VALID.
4353 void
4354 iderw(struct buf *b)
4355 {
4356     struct buf **pp;
4357
4358     if(!holdingsleep(&b->lock))
4359         panic("iderw: buf not locked");
4360     if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
4361         panic("iderw: nothing to do");
4362     if(b->dev != 0 && !havedisk1)
4363         panic("iderw: ide disk 1 not present");
4364
4365     acquire(&idelock);
4366
4367     // Append b to idequeue.
4368     b->qnext = 0;
4369     for(pp=&idequeue; *pp; pp=(*pp)->qnext)
4370         ;
4371     *pp = b;
4372
4373     // Start disk if necessary.
4374     if(idequeue == b)
4375         idestart(b);
4376
4377     // Wait for request to finish.
4378     while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
4379         sleep(b, &idelock);
4380     }
4381
4382
4383     release(&idelock);
4384 }
4385
4386
4387
4388
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399

```

```

4400 // Buffer cache.
4401 //
4402 // The buffer cache is a linked list of buf structures holding
4403 // cached copies of disk block contents. Caching disk blocks
4404 // in memory reduces the number of disk reads and also provides
4405 // a synchronization point for disk blocks used by multiple processes.
4406 //
4407 // Interface:
4408 // * To get a buffer for a particular disk block, call bread.
4409 // * After changing buffer data, call bwrite to write it to disk.
4410 // * When done with the buffer, call brelse.
4411 // * Do not use the buffer after calling brelse.
4412 // * Only one process at a time can use a buffer,
4413 //   so do not keep them longer than necessary.
4414 //
4415 // The implementation uses two state flags internally:
4416 // * B_VALID: the buffer data has been read from the disk.
4417 // * B_DIRTY: the buffer data has been modified
4418 //   and needs to be written to disk.
4419
4420 #include "types.h"
4421 #include "defs.h"
4422 #include "param.h"
4423 #include "spinlock.h"
4424 #include "sleeplock.h"
4425 #include "fs.h"
4426 #include "buf.h"
4427
4428 struct {
4429     struct spinlock lock;
4430     struct buf buf[NBUF];
4431
4432     // Linked list of all buffers, through prev/next.
4433     // head.next is most recently used.
4434     struct buf head;
4435 } bcache;
4436
4437 void
4438 binit(void)
4439 {
4440     struct buf *b;
4441
4442     initlock(&bcache.lock, "bcache");
4443
4444
4445
4446
4447
4448
4449

```

```

4450 // Create linked list of buffers
4451 bcache.head.prev = &bcache.head;
4452 bcache.head.next = &bcache.head;
4453 for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4454     b->next = bcache.head.next;
4455     b->prev = &bcache.head;
4456     initsleeplock(&b->lock, "buffer");
4457     bcache.head.next->prev = b;
4458     bcache.head.next = b;
4459 }
4460 }
4461
4462 // Look through buffer cache for block on device dev.
4463 // If not found, allocate a buffer.
4464 // In either case, return locked buffer.
4465 static struct buf*
4466 bget(uint dev, uint blockno)
4467 {
4468     struct buf *b;
4469
4470     acquire(&bcache.lock);
4471
4472     // Is the block already cached?
4473     for(b = bcache.head.next; b != &bcache.head; b = b->next){
4474         if(b->dev == dev && b->blockno == blockno){
4475             b->refcnt++;
4476             release(&bcache.lock);
4477             acquiresleep(&b->lock);
4478             return b;
4479         }
4480     }
4481
4482     // Not cached; recycle an unused buffer.
4483     // Even if refcnt==0, B_DIRTY indicates a buffer is in use
4484     // because log.c has modified it but not yet committed it.
4485     for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4486         if(b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
4487             b->dev = dev;
4488             b->blockno = blockno;
4489             b->flags = 0;
4490             b->refcnt = 1;
4491             release(&bcache.lock);
4492             acquiresleep(&b->lock);
4493             return b;
4494         }
4495     }
4496     panic("bget: no buffers");
4497 }
4498
4499

```

```

4500 // Return a locked buf with the contents of the indicated block.
4501 struct buf*
4502 bread(uint dev, uint blockno)
4503 {
4504     struct buf *b;
4505
4506     b = bget(dev, blockno);
4507     if((b->flags & B_VALID) == 0) {
4508         iderw(b);
4509     }
4510     return b;
4511 }
4512
4513 // Write b's contents to disk. Must be locked.
4514 void
4515 bwrite(struct buf *b)
4516 {
4517     if(!holdingsleep(&b->lock))
4518         panic("bwrite");
4519     b->flags |= B_DIRTY;
4520     iderw(b);
4521 }
4522
4523 // Release a locked buffer.
4524 // Move to the head of the MRU list.
4525 void
4526 brelse(struct buf *b)
4527 {
4528     if(!holdingsleep(&b->lock))
4529         panic("brelse");
4530
4531     releasesleep(&b->lock);
4532
4533     acquire(&bcache.lock);
4534     b->refcnt--;
4535     if (b->refcnt == 0) {
4536         // no one is waiting for it.
4537         b->next->prev = b->prev;
4538         b->prev->next = b->next;
4539         b->next = bcache.head.next;
4540         b->prev = &bcache.head;
4541         bcache.head.next->prev = b;
4542         bcache.head.next = b;
4543     }
4544
4545     release(&bcache.lock);
4546 }
4547
4548
4549

```

```
4550 // Blank page.
4551
4552
4553
4554
4555
4556
4557
4558
4559
4560
4561
4562
4563
4564
4565
4566
4567
4568
4569
4570
4571
4572
4573
4574
4575
4576
4577
4578
4579
4580
4581
4582
4583
4584
4585
4586
4587
4588
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4599
```

```
4600 // Sleeping locks
4601
4602 #include "types.h"
4603 #include "defs.h"
4604 #include "param.h"
4605 #include "x86.h"
4606 #include "memlayout.h"
4607 #include "mmu.h"
4608 #include "proc.h"
4609 #include "spinlock.h"
4610 #include "sleeplock.h"
4611
4612 void
4613 initsleeplock(struct sleeplock *lk, char *name)
4614 {
4615     initlock(&lk->lk, "sleep lock");
4616     lk->name = name;
4617     lk->locked = 0;
4618     lk->pid = 0;
4619 }
4620
4621 void
4622 acquiresleep(struct sleeplock *lk)
4623 {
4624     acquire(&lk->lk);
4625     while (lk->locked) {
4626         sleep(lk, &lk->lk);
4627     }
4628     lk->locked = 1;
4629     lk->pid = myproc()->pid;
4630     release(&lk->lk);
4631 }
4632
4633 void
4634 releasesleep(struct sleeplock *lk)
4635 {
4636     acquire(&lk->lk);
4637     lk->locked = 0;
4638     lk->pid = 0;
4639     wakeup(lk);
4640     release(&lk->lk);
4641 }
4642
4643
4644
4645
4646
4647
4648
4649
```

```

4650 int
4651 holdingsleep(struct sleeplock *lk)
4652 {
4653     int r;
4654
4655     acquire(&lk->lk);
4656     r = lk->locked && (lk->pid == myproc()->pid);
4657     release(&lk->lk);
4658     return r;
4659 }
4660
4661
4662
4663
4664
4665
4666
4667
4668
4669
4670
4671
4672
4673
4674
4675
4676
4677
4678
4679
4680
4681
4682
4683
4684
4685
4686
4687
4688
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4699

```

```

4700 #include "types.h"
4701 #include "defs.h"
4702 #include "param.h"
4703 #include "spinlock.h"
4704 #include "sleeplock.h"
4705 #include "fs.h"
4706 #include "buf.h"
4707
4708 // Simple logging that allows concurrent FS system calls.
4709 //
4710 // A log transaction contains the updates of multiple FS system
4711 // calls. The logging system only commits when there are
4712 // no FS system calls active. Thus there is never
4713 // any reasoning required about whether a commit might
4714 // write an uncommitted system call's updates to disk.
4715 //
4716 // A system call should call begin_op()/end_op() to mark
4717 // its start and end. Usually begin_op() just increments
4718 // the count of in-progress FS system calls and returns.
4719 // But if it thinks the log is close to running out, it
4720 // sleeps until the last outstanding end_op() commits.
4721 //
4722 // The log is a physical re-do log containing disk blocks.
4723 // The on-disk log format:
4724 //   header block, containing block #s for block A, B, C, ...
4725 //   block A
4726 //   block B
4727 //   block C
4728 //   ...
4729 // Log appends are synchronous.
4730
4731 // Contents of the header block, used for both the on-disk header block
4732 // and to keep track in memory of logged block# before commit.
4733 struct logheader {
4734     int n;
4735     int block[LOGSIZE];
4736 };
4737
4738 struct log {
4739     struct spinlock lock;
4740     int start;
4741     int size;
4742     int outstanding; // how many FS sys calls are executing.
4743     int committing;  // in commit(), please wait.
4744     int dev;
4745     struct logheader lh;
4746 };
4747
4748
4749

```



```

4750 struct log log;
4751
4752 static void recover_from_log(void);
4753 static void commit();
4754
4755 void
4756 initlog(int dev)
4757 {
4758     if (sizeof(struct logheader) >= BSIZE)
4759         panic("initlog: too big logheader");
4760
4761     struct superblock sb;
4762     initlock(&log.lock, "log");
4763     readsb(dev, &sb);
4764     log.start = sb.logstart;
4765     log.size = sb.nlog;
4766     log.dev = dev;
4767     recover_from_log();
4768 }
4769
4770 // Copy committed blocks from log to their home location
4771 static void
4772 install_trans(void)
4773 {
4774     int tail;
4775
4776     for (tail = 0; tail < log.lh.n; tail++) {
4777         struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
4778         struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
4779         memmove(dbuf->data, lbuf->data, BSIZE); // copy block to dst
4780         bwrite(dbuf); // write dst to disk
4781         brelse(lbuf);
4782         brelse(dbuf);
4783     }
4784 }
4785
4786 // Read the log header from disk into the in-memory log header
4787 static void
4788 read_head(void)
4789 {
4790     struct buf *buf = bread(log.dev, log.start);
4791     struct logheader *lh = (struct logheader *) (buf->data);
4792     int i;
4793     log.lh.n = lh->n;
4794     for (i = 0; i < log.lh.n; i++) {
4795         log.lh.block[i] = lh->block[i];
4796     }
4797     brelse(buf);
4798 }
4799

```

```

4800 // Write in-memory log header to disk.
4801 // This is the true point at which the
4802 // current transaction commits.
4803 static void
4804 write_head(void)
4805 {
4806     struct buf *buf = bread(log.dev, log.start);
4807     struct logheader *hb = (struct logheader *) (buf->data);
4808     int i;
4809     hb->n = log.lh.n;
4810     for (i = 0; i < log.lh.n; i++) {
4811         hb->block[i] = log.lh.block[i];
4812     }
4813     bwrite(buf);
4814     brelse(buf);
4815 }
4816
4817 static void
4818 recover_from_log(void)
4819 {
4820     read_head();
4821     install_trans(); // if committed, copy from log to disk
4822     log.lh.n = 0;
4823     write_head(); // clear the log
4824 }
4825
4826 // called at the start of each FS system call.
4827 void
4828 begin_op(void)
4829 {
4830     acquire(&log.lock);
4831     while(1){
4832         if(log.committing){
4833             sleep(&log, &log.lock);
4834         } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
4835             // this op might exhaust log space; wait for commit.
4836             sleep(&log, &log.lock);
4837         } else {
4838             log.outstanding += 1;
4839             release(&log.lock);
4840             break;
4841         }
4842     }
4843 }
4844
4845
4846
4847
4848
4849

```

```

4850 // called at the end of each FS system call.
4851 // commits if this was the last outstanding operation.
4852 void
4853 end_op(void)
4854 {
4855     int do_commit = 0;
4856
4857     acquire(&log.lock);
4858     log.outstanding -= 1;
4859     if(log.committing)
4860         panic("log.committing");
4861     if(log.outstanding == 0){
4862         do_commit = 1;
4863         log.committing = 1;
4864     } else {
4865         // begin_op() may be waiting for log space,
4866         // and decrementing log.outstanding has decreased
4867         // the amount of reserved space.
4868         wakeup(&log);
4869     }
4870     release(&log.lock);
4871
4872     if(do_commit){
4873         // call commit w/o holding locks, since not allowed
4874         // to sleep with locks.
4875         commit();
4876         acquire(&log.lock);
4877         log.committing = 0;
4878         wakeup(&log);
4879         release(&log.lock);
4880     }
4881 }
4882
4883 // Copy modified blocks from cache to log.
4884 static void
4885 write_log(void)
4886 {
4887     int tail;
4888
4889     for (tail = 0; tail < log.lh.n; tail++) {
4890         struct buf *to = bread(log.dev, log.start+tail+1); // log block
4891         struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
4892         memmove(to->data, from->data, BSIZE);
4893         bwrite(to); // write the log
4894         brelse(from);
4895         brelse(to);
4896     }
4897 }
4898
4899

```

```

4900 static void
4901 commit()
4902 {
4903     if (log.lh.n > 0) {
4904         write_log(); // Write modified blocks from cache to log
4905         write_head(); // Write header to disk -- the real commit
4906         install_trans(); // Now install writes to home locations
4907         log.lh.n = 0;
4908         write_head(); // Erase the transaction from the log
4909     }
4910 }
4911
4912 // Caller has modified b->data and is done with the buffer.
4913 // Record the block number and pin in the cache with B_DIRTY.
4914 // commit()/write_log() will do the disk write.
4915 //
4916 // log_write() replaces bwrite(); a typical use is:
4917 //   bp = bread(...)
4918 //   modify bp->data[]
4919 //   log_write(bp)
4920 //   brelse(bp)
4921 void
4922 log_write(struct buf *b)
4923 {
4924     int i;
4925
4926     if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
4927         panic("too big a transaction");
4928     if (log.outstanding < 1)
4929         panic("log_write outside of trans");
4930
4931     acquire(&log.lock);
4932     for (i = 0; i < log.lh.n; i++) {
4933         if (log.lh.block[i] == b->blockno) // log absorption
4934             break;
4935     }
4936     log.lh.block[i] = b->blockno;
4937     if (i == log.lh.n)
4938         log.lh.n++;
4939     b->flags |= B_DIRTY; // prevent eviction
4940     release(&log.lock);
4941 }
4942
4943
4944
4945
4946
4947
4948
4949

```

```

4950 // File system implementation. Five layers:
4951 //   + Blocks: allocator for raw disk blocks.
4952 //   + Log: crash recovery for multi-step updates.
4953 //   + Files: inode allocator, reading, writing, metadata.
4954 //   + Directories: inode with special contents (list of other inodes!)
4955 //   + Names: paths like /usr/rtn/xv6/fs.c for convenient naming.
4956 //
4957 // This file contains the low-level file system manipulation
4958 // routines. The (higher-level) system call implementations
4959 // are in sysfile.c.
4960
4961 #include "types.h"
4962 #include "defs.h"
4963 #include "param.h"
4964 #include "stat.h"
4965 #include "mmu.h"
4966 #include "proc.h"
4967 #include "spinlock.h"
4968 #include "sleeplock.h"
4969 #include "fs.h"
4970 #include "buf.h"
4971 #include "file.h"
4972
4973 #define min(a, b) ((a) < (b) ? (a) : (b))
4974 static void itrunc(struct inode*);
4975 // there should be one superblock per disk device, but we run with
4976 // only one device
4977 struct superblock sb;
4978
4979 // Read the super block.
4980 void
4981 readsb(int dev, struct superblock *sb)
4982 {
4983     struct buf *bp;
4984
4985     bp = bread(dev, 1);
4986     memmove(sb, bp->data, sizeof(*sb));
4987     brelse(bp);
4988 }
4989
4990
4991
4992
4993
4994
4995
4996
4997
4998
4999

```

```

5000 // Zero a block.
5001 static void
5002 bzero(int dev, int bno)
5003 {
5004     struct buf *bp;
5005
5006     bp = bread(dev, bno);
5007     memset(bp->data, 0, BSIZE);
5008     log_write(bp);
5009     brelse(bp);
5010 }
5011
5012 // Blocks.
5013
5014 // Allocate a zeroed disk block.
5015 static uint
5016 balloc(uint dev)
5017 {
5018     int b, bi, m;
5019     struct buf *bp;
5020
5021     bp = 0;
5022     for(b = 0; b < sb.size; b += BPB){
5023         bp = bread(dev, BBLOCK(b, sb));
5024         for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
5025             m = 1 << (bi % 8);
5026             if((bp->data[bi/8] & m) == 0){ // Is block free?
5027                 bp->data[bi/8] |= m; // Mark block in use.
5028                 log_write(bp);
5029                 brelse(bp);
5030                 bzero(dev, b + bi);
5031                 return b + bi;
5032             }
5033         }
5034         brelse(bp);
5035     }
5036     panic("balloc: out of blocks");
5037 }
5038
5039
5040
5041
5042
5043
5044
5045
5046
5047
5048
5049

```

```

5050 // Free a disk block.
5051 static void
5052 bfree(int dev, uint b)
5053 {
5054     struct buf *bp;
5055     int bi, m;
5056
5057     bp = bread(dev, BBLOCK(b, sb));
5058     bi = b % BPB;
5059     m = 1 << (bi % 8);
5060     if((bp->data[bi/8] & m) == 0)
5061         panic("freeing free block");
5062     bp->data[bi/8] &= ~m;
5063     log_write(bp);
5064     brelse(bp);
5065 }
5066
5067 // Inodes.
5068 //
5069 // An inode describes a single unnamed file.
5070 // The inode disk structure holds metadata: the file's type,
5071 // its size, the number of links referring to it, and the
5072 // list of blocks holding the file's content.
5073 //
5074 // The inodes are laid out sequentially on disk at
5075 // sb.startinode. Each inode has a number, indicating its
5076 // position on the disk.
5077 //
5078 // The kernel keeps a cache of in-use inodes in memory
5079 // to provide a place for synchronizing access
5080 // to inodes used by multiple processes. The cached
5081 // inodes include book-keeping information that is
5082 // not stored on disk: ip->ref and ip->valid.
5083 //
5084 // An inode and its in-memory representation go through a
5085 // sequence of states before they can be used by the
5086 // rest of the file system code.
5087 //
5088 // * Allocation: an inode is allocated if its type (on disk)
5089 //   is non-zero. ialloc() allocates, and iput() frees if
5090 //   the reference and link counts have fallen to zero.
5091 //
5092 // * Referencing in cache: an entry in the inode cache
5093 //   is free if ip->ref is zero. Otherwise ip->ref tracks
5094 //   the number of in-memory pointers to the entry (open
5095 //   files and current directories). iget() finds or
5096 //   creates a cache entry and increments its ref; iput()
5097 //   decrements ref.
5098 //
5099 // * Valid: the information (type, size, &c) in an inode

```

```

5100 //   cache entry is only correct when ip->valid is 1.
5101 //   ilock() reads the inode from
5102 //   the disk and sets ip->valid, while iput() clears
5103 //   ip->valid if ip->ref has fallen to zero.
5104 //
5105 // * Locked: file system code may only examine and modify
5106 //   the information in an inode and its content if it
5107 //   has first locked the inode.
5108 //
5109 // Thus a typical sequence is:
5110 //   ip = iget(dev, inum)
5111 //   ilock(ip)
5112 //   ... examine and modify ip->xxx ...
5113 //   iunlock(ip)
5114 //   iput(ip)
5115 //
5116 // ilock() is separate from iget() so that system calls can
5117 // get a long-term reference to an inode (as for an open file)
5118 // and only lock it for short periods (e.g., in read()).
5119 // The separation also helps avoid deadlock and races during
5120 // pathname lookup. iget() increments ip->ref so that the inode
5121 // stays cached and pointers to it remain valid.
5122 //
5123 // Many internal file system functions expect the caller to
5124 // have locked the inodes involved; this lets callers create
5125 // multi-step atomic operations.
5126 //
5127 // The icache.lock spin-lock protects the allocation of icache
5128 // entries. Since ip->ref indicates whether an entry is free,
5129 // and ip->dev and ip->inum indicate which i-node an entry
5130 // holds, one must hold icache.lock while using any of those fields.
5131 //
5132 // An ip->lock sleep-lock protects all ip-> fields other than ref,
5133 // dev, and inum. One must hold ip->lock in order to
5134 // read or write that inode's ip->valid, ip->size, ip->type, &c.
5135
5136 struct {
5137     struct spinlock lock;
5138     struct inode inode[NINODE];
5139 } icache;
5140
5141 void
5142 iinit(int dev)
5143 {
5144     int i = 0;
5145
5146     initlock(&icache.lock, "icache");
5147     for(i = 0; i < NINODE; i++) {
5148         initsleeplock(&icache.inode[i].lock, "inode");
5149     }

```

```

5150  readsb(dev, &sb);
5151  cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d\
5152  inodestart %d bmap start %d\n", sb.size, sb.nblocks,
5153          sb.ninodes, sb.nlog, sb.logstart, sb.inodestart,
5154          sb.bmapstart);
5155 }
5156
5157 static struct inode* iget(uint dev, uint inum);
5158
5159
5160
5161
5162
5163
5164
5165
5166
5167
5168
5169
5170
5171
5172
5173
5174
5175
5176
5177
5178
5179
5180
5181
5182
5183
5184
5185
5186
5187
5188
5189
5190
5191
5192
5193
5194
5195
5196
5197
5198
5199

```

```

5200 // Allocate an inode on device dev.
5201 // Mark it as allocated by giving it type type.
5202 // Returns an unlocked but allocated and referenced inode.
5203 struct inode*
5204 ialloc(uint dev, short type)
5205 {
5206     int inum;
5207     struct buf *bp;
5208     struct dinode *dip;
5209
5210     for(inum = 1; inum < sb.ninodes; inum++){
5211         bp = bread(dev, IBLOCK(inum, sb));
5212         dip = (struct dinode*)bp->data + inum%IPB;
5213         if(dip->type == 0){ // a free inode
5214             memset(dip, 0, sizeof(*dip));
5215             dip->type = type;
5216             log_write(bp); // mark it allocated on the disk
5217             brelse(bp);
5218             return iget(dev, inum);
5219         }
5220         brelse(bp);
5221     }
5222     panic("ialloc: no inodes");
5223 }
5224
5225 // Copy a modified in-memory inode to disk.
5226 // Must be called after every change to an ip->xxx field
5227 // that lives on disk, since i-node cache is write-through.
5228 // Caller must hold ip->lock.
5229 void
5230 iupdate(struct inode *ip)
5231 {
5232     struct buf *bp;
5233     struct dinode *dip;
5234
5235     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5236     dip = (struct dinode*)bp->data + ip->inum%IPB;
5237     dip->type = ip->type;
5238     dip->major = ip->major;
5239     dip->minor = ip->minor;
5240     dip->nlink = ip->nlink;
5241     dip->size = ip->size;
5242     memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5243     log_write(bp);
5244     brelse(bp);
5245 }
5246
5247
5248
5249

```

```

5250 // Find the inode with number inum on device dev
5251 // and return the in-memory copy. Does not lock
5252 // the inode and does not read it from disk.
5253 static struct inode*
5254 iget(uint dev, uint inum)
5255 {
5256     struct inode *ip, *empty;
5257
5258     acquire(&icache.lock);
5259
5260     // Is the inode already cached?
5261     empty = 0;
5262     for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
5263         if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5264             ip->ref++;
5265             release(&icache.lock);
5266             return ip;
5267         }
5268         if(empty == 0 && ip->ref == 0)    // Remember empty slot.
5269             empty = ip;
5270     }
5271
5272     // Recycle an inode cache entry.
5273     if(empty == 0)
5274         panic("iget: no inodes");
5275
5276     ip = empty;
5277     ip->dev = dev;
5278     ip->inum = inum;
5279     ip->ref = 1;
5280     ip->valid = 0;
5281     release(&icache.lock);
5282
5283     return ip;
5284 }
5285
5286 // Increment reference count for ip.
5287 // Returns ip to enable ip = idup(ip1) idiom.
5288 struct inode*
5289 idup(struct inode *ip)
5290 {
5291     acquire(&icache.lock);
5292     ip->ref++;
5293     release(&icache.lock);
5294     return ip;
5295 }
5296
5297
5298
5299

```

```

5300 // Lock the given inode.
5301 // Reads the inode from disk if necessary.
5302 void
5303 ilock(struct inode *ip)
5304 {
5305     struct buf *bp;
5306     struct dinode *dip;
5307
5308     if(ip == 0 || ip->ref < 1)
5309         panic("ilock");
5310
5311     acquiresleep(&ip->lock);
5312
5313     if(ip->valid == 0){
5314         bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5315         dip = (struct dinode*)bp->data + ip->inum%IPB;
5316         ip->type = dip->type;
5317         ip->major = dip->major;
5318         ip->minor = dip->minor;
5319         ip->nlink = dip->nlink;
5320         ip->size = dip->size;
5321         memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5322         brelse(bp);
5323         ip->valid = 1;
5324         if(ip->type == 0)
5325             panic("ilock: no type");
5326     }
5327 }
5328
5329 // Unlock the given inode.
5330 void
5331 iunlock(struct inode *ip)
5332 {
5333     if(ip == 0 || !holdingsleep(&ip->lock) || ip->ref < 1)
5334         panic("iunlock");
5335
5336     releasesleep(&ip->lock);
5337 }
5338
5339
5340
5341
5342
5343
5344
5345
5346
5347
5348
5349

```

```

5350 // Drop a reference to an in-memory inode.
5351 // If that was the last reference, the inode cache entry can
5352 // be recycled.
5353 // If that was the last reference and the inode has no links
5354 // to it, free the inode (and its content) on disk.
5355 // All calls to iput() must be inside a transaction in
5356 // case it has to free the inode.
5357 void
5358 iput(struct inode *ip)
5359 {
5360     acquiresleep(&ip->lock);
5361     if(ip->valid && ip->nlink == 0){
5362         acquire(&icache.lock);
5363         int r = ip->ref;
5364         release(&icache.lock);
5365         if(r == 1){
5366             // inode has no links and no other references: truncate and free.
5367             itrunc(ip);
5368             ip->type = 0;
5369             iupdate(ip);
5370             ip->valid = 0;
5371         }
5372     }
5373     releasesleep(&ip->lock);
5374
5375     acquire(&icache.lock);
5376     ip->ref--;
5377     release(&icache.lock);
5378 }
5379
5380 // Common idiom: unlock, then put.
5381 void
5382 iunlockput(struct inode *ip)
5383 {
5384     iunlock(ip);
5385     iput(ip);
5386 }
5387
5388
5389
5390
5391
5392
5393
5394
5395
5396
5397
5398
5399

```

```

5400 // Inode content
5401 //
5402 // The content (data) associated with each inode is stored
5403 // in blocks on the disk. The first NDIRECT block numbers
5404 // are listed in ip->addrs[]. The next NINDIRECT blocks are
5405 // listed in block ip->addrs[NDIRECT].
5406
5407 // Return the disk block address of the nth block in inode ip.
5408 // If there is no such block, bmap allocates one.
5409 static uint
5410 bmap(struct inode *ip, uint bn)
5411 {
5412     uint addr, *a;
5413     struct buf *bp;
5414
5415     if(bn < NDIRECT){
5416         if((addr = ip->addrs[bn]) == 0)
5417             ip->addrs[bn] = addr = balloc(ip->dev);
5418         return addr;
5419     }
5420     bn -= NDIRECT;
5421
5422     if(bn < NINDIRECT){
5423         // Load indirect block, allocating if necessary.
5424         if((addr = ip->addrs[NDIRECT]) == 0)
5425             ip->addrs[NDIRECT] = addr = balloc(ip->dev);
5426         bp = bread(ip->dev, addr);
5427         a = (uint*)bp->data;
5428         if((addr = a[bn]) == 0){
5429             a[bn] = addr = balloc(ip->dev);
5430             log_write(bp);
5431         }
5432         brelse(bp);
5433         return addr;
5434     }
5435
5436     panic("bmap: out of range");
5437 }
5438
5439
5440
5441
5442
5443
5444
5445
5446
5447
5448
5449

```

```

5450 // Truncate inode (discard contents).
5451 // Only called when the inode has no links
5452 // to it (no directory entries referring to it)
5453 // and has no in-memory reference to it (is
5454 // not an open file or current directory).
5455 static void
5456 itrunc(struct inode *ip)
5457 {
5458     int i, j;
5459     struct buf *bp;
5460     uint *a;
5461
5462     for(i = 0; i < NDIRECT; i++){
5463         if(ip->addrs[i]){
5464             bfree(ip->dev, ip->addrs[i]);
5465             ip->addrs[i] = 0;
5466         }
5467     }
5468
5469     if(ip->addrs[NDIRECT]){
5470         bp = bread(ip->dev, ip->addrs[NDIRECT]);
5471         a = (uint*)bp->data;
5472         for(j = 0; j < NINDIRECT; j++){
5473             if(a[j])
5474                 bfree(ip->dev, a[j]);
5475         }
5476         brelse(bp);
5477         bfree(ip->dev, ip->addrs[NDIRECT]);
5478         ip->addrs[NDIRECT] = 0;
5479     }
5480
5481     ip->size = 0;
5482     iupdate(ip);
5483 }
5484
5485 // Copy stat information from inode.
5486 // Caller must hold ip->lock.
5487 void
5488 stati(struct inode *ip, struct stat *st)
5489 {
5490     st->dev = ip->dev;
5491     st->ino = ip->inum;
5492     st->type = ip->type;
5493     st->nlink = ip->nlink;
5494     st->size = ip->size;
5495 }
5496
5497
5498
5499

```

```

5500 // Read data from inode.
5501 // Caller must hold ip->lock.
5502 int
5503 readi(struct inode *ip, char *dst, uint off, uint n)
5504 {
5505     uint tot, m;
5506     struct buf *bp;
5507
5508     if(ip->type == T_DEV){
5509         if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
5510             return -1;
5511         return devsw[ip->major].read(ip, dst, n);
5512     }
5513
5514     if(off > ip->size || off + n < off)
5515         return -1;
5516     if(off + n > ip->size)
5517         n = ip->size - off;
5518
5519     for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5520         bp = bread(ip->dev, bmap(ip, off/BSIZE));
5521         m = min(n - tot, BSIZE - off%BSIZE);
5522         memmove(dst, bp->data + off%BSIZE, m);
5523         brelse(bp);
5524     }
5525     return n;
5526 }
5527
5528
5529
5530
5531
5532
5533
5534
5535
5536
5537
5538
5539
5540
5541
5542
5543
5544
5545
5546
5547
5548
5549

```



```

5550 // Write data to inode.
5551 // Caller must hold ip->lock.
5552 int
5553 writei(struct inode *ip, char *src, uint off, uint n)
5554 {
5555     uint tot, m;
5556     struct buf *bp;
5557
5558     if(ip->type == T_DEV){
5559         if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write){
5560             return -1;
5561         }
5562         return devsw[ip->major].write(ip, src, n);
5563     }
5564     if(off > ip->size || off + n < off)
5565         return -1;
5566     if(off + n > MAXFILE*BSIZE)
5567         return -1;
5568
5569     for(tot=0; tot<n; tot+=m, off+=m, src+=m){
5570         bp = bread(ip->dev, bmap(ip, off/BSIZE));
5571         m = min(n - tot, BSIZE - off%BSIZE);
5572         memmove(bp->data + off%BSIZE, src, m);
5573         log_write(bp);
5574         brelse(bp);
5575     }
5576
5577     if(n > 0 && off > ip->size){
5578         ip->size = off;
5579         iupdate(ip);
5580     }
5581     return n;
5582 }
5583
5584
5585
5586
5587
5588
5589
5590
5591
5592
5593
5594
5595
5596
5597
5598
5599

```

```

5600 // Directories
5601
5602 int
5603 namecmp(const char *s, const char *t)
5604 {
5605     return strncmp(s, t, DIRSIZ);
5606 }
5607
5608 // Look for a directory entry in a directory.
5609 // If found, set *poff to byte offset of entry.
5610 struct inode*
5611 dirlookup(struct inode *dp, char *name, uint *poff)
5612 {
5613     uint off, inum;
5614     struct dirent de;
5615
5616     if(dp->type != T_DIR)
5617         panic("dirlookup not DIR");
5618
5619     for(off = 0; off < dp->size; off += sizeof(de)){
5620         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5621             panic("dirlookup read");
5622         if(de.inum == 0)
5623             continue;
5624         if(namecmp(name, de.name) == 0){
5625             // entry matches path element
5626             if(poff)
5627                 *poff = off;
5628             inum = de.inum;
5629             return iget(dp->dev, inum);
5630         }
5631     }
5632
5633     return 0;
5634 }
5635
5636
5637
5638
5639
5640
5641
5642
5643
5644
5645
5646
5647
5648
5649

```

```

5650 // Write a new directory entry (name, inum) into the directory dp.
5651 int
5652 dirlink(struct inode *dp, char *name, uint inum)
5653 {
5654     int off;
5655     struct dirent de;
5656     struct inode *ip;
5657
5658     // Check that name is not present.
5659     if((ip = dirlookup(dp, name, 0)) != 0){
5660         iput(ip);
5661         return -1;
5662     }
5663
5664     // Look for an empty dirent.
5665     for(off = 0; off < dp->size; off += sizeof(de)){
5666         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5667             panic("dirlink read");
5668         if(de.inum == 0)
5669             break;
5670     }
5671
5672     strncpy(de.name, name, DIRSIZ);
5673     de.inum = inum;
5674     if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5675         panic("dirlink");
5676
5677     return 0;
5678 }
5679
5680
5681
5682
5683
5684
5685
5686
5687
5688
5689
5690
5691
5692
5693
5694
5695
5696
5697
5698
5699

```

```

5700 // Paths
5701
5702 // Copy the next path element from path into name.
5703 // Return a pointer to the element following the copied one.
5704 // The returned path has no leading slashes,
5705 // so the caller can check *path=='\0' to see if the name is the last one.
5706 // If no name to remove, return 0.
5707 //
5708 // Examples:
5709 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5710 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5711 //   skipelem("a", name) = "", setting name = "a"
5712 //   skipelem("", name) = skipelem("///", name) = 0
5713 //
5714 static char*
5715 skipelem(char *path, char *name)
5716 {
5717     char *s;
5718     int len;
5719
5720     while(*path == '/')
5721         path++;
5722     if(*path == 0)
5723         return 0;
5724     s = path;
5725     while(*path != '/' && *path != 0)
5726         path++;
5727     len = path - s;
5728     if(len >= DIRSIZ)
5729         memmove(name, s, DIRSIZ);
5730     else {
5731         memmove(name, s, len);
5732         name[len] = 0;
5733     }
5734     while(*path == '/')
5735         path++;
5736     return path;
5737 }
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749

```

```

5750 // Look up and return the inode for a path name.
5751 // If parent != 0, return the inode for the parent and copy the final
5752 // path element into name, which must have room for DIRSIZ bytes.
5753 // Must be called inside a transaction since it calls iput().
5754 static struct inode*
5755 namex(char *path, int nameparent, char *name)
5756 {
5757     struct inode *ip, *next;
5758
5759     if(*path == '/')
5760         ip = iget(ROOTDEV, ROOTINO);
5761     else
5762         ip = idup(myproc()->cwd);
5763
5764     while((path = skipelem(path, name)) != 0){
5765         ilock(ip);
5766         if(ip->type != T_DIR){
5767             iunlockput(ip);
5768             return 0;
5769         }
5770         if(nameparent && *path == '\0'){
5771             // Stop one level early.
5772             iunlock(ip);
5773             return ip;
5774         }
5775         if((next = dirlookup(ip, name, 0)) == 0){
5776             iunlockput(ip);
5777             return 0;
5778         }
5779         iunlockput(ip);
5780         ip = next;
5781     }
5782     if(nameparent){
5783         iput(ip);
5784         return 0;
5785     }
5786     return ip;
5787 }
5788
5789 struct inode*
5790 namei(char *path)
5791 {
5792     char name[DIRSIZ];
5793     return namex(path, 0, name);
5794 }
5795
5796
5797
5798
5799

```

```

5800 struct inode*
5801 nameiparent(char *path, char *name)
5802 {
5803     return namex(path, 1, name);
5804 }
5805
5806
5807
5808
5809
5810
5811
5812
5813
5814
5815
5816
5817
5818
5819
5820
5821
5822
5823
5824
5825
5826
5827
5828
5829
5830
5831
5832
5833
5834
5835
5836
5837
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849

```

```

5850 //
5851 // File descriptors
5852 //
5853
5854 #include "types.h"
5855 #include "defs.h"
5856 #include "param.h"
5857 #include "fs.h"
5858 #include "spinlock.h"
5859 #include "sleeplock.h"
5860 #include "file.h"
5861
5862 struct devsw devsw[NDEV];
5863 struct {
5864     struct spinlock lock;
5865     struct file file[NFILE];
5866 } ftable;
5867
5868 void
5869 fileinit(void)
5870 {
5871     initlock(&ftable.lock, "ftable");
5872 }
5873
5874 // Allocate a file structure.
5875 struct file*
5876 filealloc(void)
5877 {
5878     struct file *f;
5879
5880     acquire(&ftable.lock);
5881     for(f = ftable.file; f < ftable.file + NFILE; f++){
5882         if(f->ref == 0){
5883             f->ref = 1;
5884             release(&ftable.lock);
5885             return f;
5886         }
5887     }
5888     release(&ftable.lock);
5889     return 0;
5890 }
5891
5892
5893
5894
5895
5896
5897
5898
5899

```

```

5900 // Increment ref count for file f.
5901 struct file*
5902 filedup(struct file *f)
5903 {
5904     acquire(&ftable.lock);
5905     if(f->ref < 1)
5906         panic("filedup");
5907     f->ref++;
5908     release(&ftable.lock);
5909     return f;
5910 }
5911
5912 // Close file f. (Decrement ref count, close when reaches 0.)
5913 void
5914 fileclose(struct file *f)
5915 {
5916     struct file ff;
5917
5918     acquire(&ftable.lock);
5919     if(f->ref < 1)
5920         panic("fileclose");
5921     if(--f->ref > 0){
5922         release(&ftable.lock);
5923         return;
5924     }
5925     ff = *f;
5926     f->ref = 0;
5927     f->type = FD_NONE;
5928     release(&ftable.lock);
5929
5930     if(ff.type == FD_PIPE)
5931         pipeclose(ff.pipe, ff.writable);
5932     else if(ff.type == FD_INODE){
5933         begin_op();
5934         iput(ff.ip);
5935         end_op();
5936     }
5937 }
5938
5939
5940
5941
5942
5943
5944
5945
5946
5947
5948
5949

```

```

5950 // Get metadata about file f.
5951 int
5952 filestat(struct file *f, struct stat *st)
5953 {
5954     if(f->type == FD_INODE){
5955         ilock(f->ip);
5956         stati(f->ip, st);
5957         iunlock(f->ip);
5958         return 0;
5959     }
5960     return -1;
5961 }
5962
5963 // Read from file f.
5964 int
5965 fileread(struct file *f, char *addr, int n)
5966 {
5967     int r;
5968
5969     if(f->readable == 0)
5970         return -1;
5971     if(f->type == FD_PIPE)
5972         return piperead(f->pipe, addr, n);
5973     if(f->type == FD_INODE){
5974         ilock(f->ip);
5975         if((r = readi(f->ip, addr, f->off, n)) > 0)
5976             f->off += r;
5977         iunlock(f->ip);
5978         return r;
5979     }
5980     panic("fileread");
5981 }
5982
5983
5984
5985
5986
5987
5988
5989
5990
5991
5992
5993
5994
5995
5996
5997
5998
5999

```

```

6000 // Write to file f.
6001 int
6002 filewrite(struct file *f, char *addr, int n)
6003 {
6004     int r;
6005
6006     if(f->writable == 0)
6007         return -1;
6008     if(f->type == FD_PIPE)
6009         return pipewrite(f->pipe, addr, n);
6010     if(f->type == FD_INODE){
6011         // write a few blocks at a time to avoid exceeding
6012         // the maximum log transaction size, including
6013         // i-node, indirect block, allocation blocks,
6014         // and 2 blocks of slop for non-aligned writes.
6015         // this really belongs lower down, since writei()
6016         // might be writing a device like the console.
6017         int max = ((MAXOPBLOCKS-1-1-2) / 2) * 512;
6018         int i = 0;
6019         while(i < n){
6020             int n1 = n - i;
6021             if(n1 > max)
6022                 n1 = max;
6023
6024             begin_op();
6025             ilock(f->ip);
6026             if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
6027                 f->off += r;
6028             iunlock(f->ip);
6029             end_op();
6030
6031             if(r < 0)
6032                 break;
6033             if(r != n1)
6034                 panic("short filewrite");
6035             i += r;
6036         }
6037         return i == n ? n : -1;
6038     }
6039     panic("filewrite");
6040 }
6041
6042
6043
6044
6045
6046
6047
6048
6049

```

```

6050 //
6051 // File-system system calls.
6052 // Mostly argument checking, since we don't trust
6053 // user code, and calls into file.c and fs.c.
6054 //
6055
6056 #include "types.h"
6057 #include "defs.h"
6058 #include "param.h"
6059 #include "stat.h"
6060 #include "mmu.h"
6061 #include "proc.h"
6062 #include "fs.h"
6063 #include "spinlock.h"
6064 #include "sleeplock.h"
6065 #include "file.h"
6066 #include "fcntl.h"
6067
6068 // Fetch the nth word-sized system call argument as a file descriptor
6069 // and return both the descriptor and the corresponding struct file.
6070 static int
6071 argfd(int n, int *pfd, struct file **pf)
6072 {
6073     int fd;
6074     struct file *f;
6075
6076     if(argint(n, &fd) < 0)
6077         return -1;
6078     if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
6079         return -1;
6080     if(pfd)
6081         *pfd = fd;
6082     if(pf)
6083         *pf = f;
6084     return 0;
6085 }
6086
6087
6088
6089
6090
6091
6092
6093
6094
6095
6096
6097
6098
6099

```

```

6100 // Allocate a file descriptor for the given file.
6101 // Takes over file reference from caller on success.
6102 static int
6103 fdalloc(struct file *f)
6104 {
6105     int fd;
6106     struct proc *curproc = myproc();
6107
6108     for(fd = 0; fd < NOFILE; fd++){
6109         if(curproc->ofile[fd] == 0){
6110             curproc->ofile[fd] = f;
6111             return fd;
6112         }
6113     }
6114     return -1;
6115 }
6116
6117 int
6118 sys_dup(void)
6119 {
6120     struct file *f;
6121     int fd;
6122
6123     if(argfd(0, 0, &f) < 0)
6124         return -1;
6125     if((fd=fdalloc(f)) < 0)
6126         return -1;
6127     filedup(f);
6128     return fd;
6129 }
6130
6131 int
6132 sys_read(void)
6133 {
6134     struct file *f;
6135     int n;
6136     char *p;
6137
6138     if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6139         return -1;
6140     return fileread(f, p, n);
6141 }
6142
6143
6144
6145
6146
6147
6148
6149

```

```

6150 int
6151 sys_write(void)
6152 {
6153     struct file *f;
6154     int n;
6155     char *p;
6156
6157     if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6158         return -1;
6159     return filewrite(f, p, n);
6160 }
6161
6162 int
6163 sys_close(void)
6164 {
6165     int fd;
6166     struct file *f;
6167
6168     if(argfd(0, &fd, &f) < 0)
6169         return -1;
6170     myproc()->ofile[fd] = 0;
6171     fileclose(f);
6172     return 0;
6173 }
6174
6175 int
6176 sys_fstat(void)
6177 {
6178     struct file *f;
6179     struct stat *st;
6180
6181     if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
6182         return -1;
6183     return filestat(f, st);
6184 }
6185
6186
6187
6188
6189
6190
6191
6192
6193
6194
6195
6196
6197
6198
6199

```

```

6200 // Create the path new as a link to the same inode as old.
6201 int
6202 sys_link(void)
6203 {
6204     char name[DIRSIZ], *new, *old;
6205     struct inode *dp, *ip;
6206
6207     if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
6208         return -1;
6209
6210     begin_op();
6211     if((ip = namei(old)) == 0){
6212         end_op();
6213         return -1;
6214     }
6215
6216     ilock(ip);
6217     if(ip->type == T_DIR){
6218         iunlockput(ip);
6219         end_op();
6220         return -1;
6221     }
6222
6223     ip->nlink++;
6224     iupdate(ip);
6225     iunlock(ip);
6226
6227     if((dp = nameiparent(new, name)) == 0)
6228         goto bad;
6229     ilock(dp);
6230     if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
6231         iunlockput(dp);
6232         goto bad;
6233     }
6234     iunlockput(dp);
6235     iput(ip);
6236
6237     end_op();
6238
6239     return 0;
6240
6241 bad:
6242     ilock(ip);
6243     ip->nlink--;
6244     iupdate(ip);
6245     iunlockput(ip);
6246     end_op();
6247     return -1;
6248 }
6249

```

```

6250 // Is the directory dp empty except for "." and ".." ?
6251 static int
6252 isdirempty(struct inode *dp)
6253 {
6254     int off;
6255     struct dirent de;
6256     for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
6257         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6258             panic("isdirempty: readi");
6259         if(de.inum != 0)
6260             return 0;
6261     }
6262     return 1;
6263 }
6264 }
6265
6266
6267
6268
6269
6270
6271
6272
6273
6274
6275
6276
6277
6278
6279
6280
6281
6282
6283
6284
6285
6286
6287
6288
6289
6290
6291
6292
6293
6294
6295
6296
6297
6298
6299

```

```

6300 int
6301 sys_unlink(void)
6302 {
6303     struct inode *ip, *dp;
6304     struct dirent de;
6305     char name[DIRSIZ], *path;
6306     uint off;
6307
6308     if(argstr(0, &path) < 0)
6309         return -1;
6310
6311     begin_op();
6312     if((dp = nameiparent(path, name)) == 0){
6313         end_op();
6314         return -1;
6315     }
6316
6317     ilock(dp);
6318
6319     // Cannot unlink "." or "..".
6320     if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6321         goto bad;
6322
6323     if((ip = dirlookup(dp, name, &off)) == 0)
6324         goto bad;
6325     ilock(ip);
6326
6327     if(ip->nlink < 1)
6328         panic("unlink: nlink < 1");
6329     if(ip->type == T_DIR && !isdirempty(ip)){
6330         iunlockput(ip);
6331         goto bad;
6332     }
6333
6334     memset(&de, 0, sizeof(de));
6335     if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6336         panic("unlink: writei");
6337     if(ip->type == T_DIR){
6338         dp->nlink--;
6339         iupdate(dp);
6340     }
6341     iunlockput(dp);
6342
6343     ip->nlink--;
6344     iupdate(ip);
6345     iunlockput(ip);
6346
6347     end_op();
6348
6349     return 0;

```



```

6350 bad:
6351     iunlockput(dp);
6352     end_op();
6353     return -1;
6354 }
6355
6356 static struct inode*
6357 create(char *path, short type, short major, short minor)
6358 {
6359     struct inode *ip, *dp;
6360     char name[DIRSIZ];
6361
6362     if((dp = nameiparent(path, name)) == 0)
6363         return 0;
6364     ilock(dp);
6365
6366     if((ip = dirlookup(dp, name, 0)) != 0){
6367         iunlockput(dp);
6368         ilock(ip);
6369         if(type == T_FILE && ip->type == T_FILE)
6370             return ip;
6371         iunlockput(ip);
6372         return 0;
6373     }
6374
6375     if((ip = ialloc(dp->dev, type)) == 0)
6376         panic("create: ialloc");
6377
6378     ilock(ip);
6379     ip->major = major;
6380     ip->minor = minor;
6381     ip->nlink = 1;
6382     iupdate(ip);
6383
6384     if(type == T_DIR){ // Create . and .. entries.
6385         dp->nlink++; // for ".."
6386         iupdate(dp);
6387         // No ip->nlink++ for ".": avoid cyclic ref count.
6388         if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6389             panic("create dots");
6390     }
6391
6392     if(dirlink(dp, name, ip->inum) < 0)
6393         panic("create: dirlink");
6394
6395     iunlockput(dp);
6396
6397     return ip;
6398 }
6399

```

```

6400 int
6401 sys_open(void)
6402 {
6403     char *path;
6404     int fd, omode;
6405     struct file *f;
6406     struct inode *ip;
6407
6408     if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6409         return -1;
6410
6411     begin_op();
6412
6413     if(omode & O_CREATE){
6414         ip = create(path, T_FILE, 0, 0);
6415         if(ip == 0){
6416             end_op();
6417             return -1;
6418         }
6419     } else {
6420         if((ip = namei(path)) == 0){
6421             end_op();
6422             return -1;
6423         }
6424         ilock(ip);
6425         if(ip->type == T_DIR && omode != O_RDONLY){
6426             iunlockput(ip);
6427             end_op();
6428             return -1;
6429         }
6430     }
6431
6432     if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6433         if(f)
6434             fileclose(f);
6435         iunlockput(ip);
6436         end_op();
6437         return -1;
6438     }
6439     iunlock(ip);
6440     end_op();
6441
6442     f->type = FD_INODE;
6443     f->ip = ip;
6444     f->off = 0;
6445     f->readable = !(omode & O_WRONLY);
6446     f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6447     return fd;
6448 }
6449

```

```

6450 int
6451 sys_mkdir(void)
6452 {
6453     char *path;
6454     struct inode *ip;
6455
6456     begin_op();
6457     if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
6458         end_op();
6459         return -1;
6460     }
6461     iunlockput(ip);
6462     end_op();
6463     return 0;
6464 }
6465
6466 int
6467 sys_mknod(void)
6468 {
6469     struct inode *ip;
6470     char *path;
6471     int major, minor;
6472
6473     begin_op();
6474     if((argstr(0, &path)) < 0 ||
6475        argint(1, &major) < 0 ||
6476        argint(2, &minor) < 0 ||
6477        (ip = create(path, T_DEV, major, minor)) == 0){
6478         end_op();
6479         return -1;
6480     }
6481     iunlockput(ip);
6482     end_op();
6483     return 0;
6484 }
6485
6486
6487
6488
6489
6490
6491
6492
6493
6494
6495
6496
6497
6498
6499

```

```

6500 int
6501 sys_chdir(void)
6502 {
6503     char *path;
6504     struct inode *ip;
6505     struct proc *curproc = myproc();
6506
6507     begin_op();
6508     if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
6509         end_op();
6510         return -1;
6511     }
6512     ilock(ip);
6513     if(ip->type != T_DIR){
6514         iunlockput(ip);
6515         end_op();
6516         return -1;
6517     }
6518     iunlock(ip);
6519     iput(curproc->cwd);
6520     end_op();
6521     curproc->cwd = ip;
6522     return 0;
6523 }
6524
6525 int
6526 sys_exec(void)
6527 {
6528     char *path, *argv[MAXARG];
6529     int i;
6530     uint uargv, uarg;
6531
6532     if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6533         return -1;
6534     }
6535     memset(argv, 0, sizeof(argv));
6536     for(i=0; i++){
6537         if(i >= NELEM(argv))
6538             return -1;
6539         if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6540             return -1;
6541         if(uarg == 0){
6542             argv[i] = 0;
6543             break;
6544         }
6545         if(fetchstr(uarg, &argv[i]) < 0)
6546             return -1;
6547     }
6548     return exec(path, argv);
6549 }

```

```

6550 int
6551 sys_pipe(void)
6552 {
6553     int *fd;
6554     struct file *rf, *wf;
6555     int fd0, fd1;
6556
6557     if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6558         return -1;
6559     if(pipealloc(&rf, &wf) < 0)
6560         return -1;
6561     fd0 = -1;
6562     if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6563         if(fd0 >= 0)
6564             myproc()->ofile[fd0] = 0;
6565         fileclose(rf);
6566         fileclose(wf);
6567         return -1;
6568     }
6569     fd[0] = fd0;
6570     fd[1] = fd1;
6571     return 0;
6572 }
6573
6574
6575
6576
6577
6578
6579
6580
6581
6582
6583
6584
6585
6586
6587
6588
6589
6590
6591
6592
6593
6594
6595
6596
6597
6598
6599

```

```

6600 #include "types.h"
6601 #include "param.h"
6602 #include "memlayout.h"
6603 #include "mmu.h"
6604 #include "proc.h"
6605 #include "defs.h"
6606 #include "x86.h"
6607 #include "elf.h"
6608
6609 int
6610 exec(char *path, char **argv)
6611 {
6612     char *s, *last;
6613     int i, off;
6614     uint argc, sz, sp, ustack[3+MAXARG+1];
6615     struct elfhdr elf;
6616     struct inode *ip;
6617     struct proghdr ph;
6618     pde_t *pgdir, *oldpgdir;
6619     struct proc *curproc = myproc();
6620
6621     begin_op();
6622
6623     if((ip = namei(path)) == 0){
6624         end_op();
6625         cprintf("exec: fail\n");
6626         return -1;
6627     }
6628     ilock(ip);
6629     pgdir = 0;
6630
6631     // Check ELF header
6632     if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
6633         goto bad;
6634     if(elf.magic != ELF_MAGIC)
6635         goto bad;
6636
6637     if((pgdir = setupkvm()) == 0)
6638         goto bad;
6639
6640     // Load program into memory.
6641     sz = 0;
6642     for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6643         if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6644             goto bad;
6645         if(ph.type != ELF_PROG_LOAD)
6646             continue;
6647         if(ph.memsz < ph.filesz)
6648             goto bad;
6649         if(ph.vaddr + ph.memsz < ph.vaddr)

```

```

6650     goto bad;
6651     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6652         goto bad;
6653     if(ph.vaddr % PGSIZE != 0)
6654         goto bad;
6655     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6656         goto bad;
6657 }
6658 iunlockput(ip);
6659 end_op();
6660 ip = 0;
6661
6662 // Allocate two pages at the next page boundary.
6663 // Make the first inaccessible. Use the second as the user stack.
6664 sz = PGROUNDUP(sz);
6665 if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
6666     goto bad;
6667 clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
6668 sp = sz;
6669
6670 // Push argument strings, prepare rest of stack in ustack.
6671 for(argc = 0; argv[argc]; argc++) {
6672     if(argc >= MAXARG)
6673         goto bad;
6674     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
6675     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6676         goto bad;
6677     ustack[3+argc] = sp;
6678 }
6679 ustack[3+argc] = 0;
6680
6681 ustack[0] = 0xffffffff; // fake return PC
6682 ustack[1] = argc;
6683 ustack[2] = sp - (argc+1)*4; // argv pointer
6684
6685 sp -= (3+argc+1) * 4;
6686 if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
6687     goto bad;
6688
6689 // Save program name for debugging.
6690 for(last=s=path; *s; s++)
6691     if(*s == '/')
6692         last = s+1;
6693 safestrcpy(curproc->name, last, sizeof(curproc->name));
6694
6695 // Commit to the user image.
6696 oldpgdir = curproc->pgdir;
6697 curproc->pgdir = pgdir;
6698 curproc->sz = sz;
6699 curproc->tf->eip = elf.entry; // main

```

```

6700     curproc->tf->esp = sp;
6701     switchuvm(curproc);
6702     freevm(oldpgdir);
6703     return 0;
6704
6705 bad:
6706     if(pgdir)
6707         freevm(pgdir);
6708     if(ip) {
6709         iunlockput(ip);
6710         end_op();
6711     }
6712     return -1;
6713 }
6714
6715
6716
6717
6718
6719
6720
6721
6722
6723
6724
6725
6726
6727
6728
6729
6730
6731
6732
6733
6734
6735
6736
6737
6738
6739
6740
6741
6742
6743
6744
6745
6746
6747
6748
6749

```

```

6750 #include "types.h"
6751 #include "defs.h"
6752 #include "param.h"
6753 #include "mmu.h"
6754 #include "proc.h"
6755 #include "fs.h"
6756 #include "spinlock.h"
6757 #include "sleeplock.h"
6758 #include "file.h"
6759
6760 #define PIPESIZE 512
6761
6762 struct pipe {
6763     struct spinlock lock;
6764     char data[PIPESIZE];
6765     uint nread;    // number of bytes read
6766     uint nwrite;   // number of bytes written
6767     int readopen;  // read fd is still open
6768     int writeopen; // write fd is still open
6769 };
6770
6771 int
6772 pipealloc(struct file **f0, struct file **f1)
6773 {
6774     struct pipe *p;
6775
6776     p = 0;
6777     *f0 = *f1 = 0;
6778     if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
6779         goto bad;
6780     if((p = (struct pipe*)kalloc()) == 0)
6781         goto bad;
6782     p->readopen = 1;
6783     p->writeopen = 1;
6784     p->nwrite = 0;
6785     p->nread = 0;
6786     initlock(&p->lock, "pipe");
6787     (*f0)->type = FD_PIPE;
6788     (*f0)->readable = 1;
6789     (*f0)->writable = 0;
6790     (*f0)->pipe = p;
6791     (*f1)->type = FD_PIPE;
6792     (*f1)->readable = 0;
6793     (*f1)->writable = 1;
6794     (*f1)->pipe = p;
6795     return 0;
6796
6797
6798
6799

```

```

6800 bad:
6801     if(p)
6802         kfree((char*)p);
6803     if(*f0)
6804         fileclose(*f0);
6805     if(*f1)
6806         fileclose(*f1);
6807     return -1;
6808 }
6809
6810 void
6811 pipeclose(struct pipe *p, int writable)
6812 {
6813     acquire(&p->lock);
6814     if(writable){
6815         p->writeopen = 0;
6816         wakeup(&p->nread);
6817     } else {
6818         p->readopen = 0;
6819         wakeup(&p->nwrite);
6820     }
6821     if(p->readopen == 0 && p->writeopen == 0){
6822         release(&p->lock);
6823         kfree((char*)p);
6824     } else
6825         release(&p->lock);
6826 }
6827
6828 int
6829 pipewrite(struct pipe *p, char *addr, int n)
6830 {
6831     int i;
6832
6833     acquire(&p->lock);
6834     for(i = 0; i < n; i++){
6835         while(p->nwrite == p->nread + PIPESIZE){
6836             if(p->readopen == 0 || myproc()->killed){
6837                 release(&p->lock);
6838                 return -1;
6839             }
6840             wakeup(&p->nread);
6841             sleep(&p->nwrite, &p->lock);
6842         }
6843         p->data[p->nwrite++ % PIPESIZE] = addr[i];
6844     }
6845     wakeup(&p->nread);
6846     release(&p->lock);
6847     return n;
6848
6849 }

```

```

6850 int
6851 piperead(struct pipe *p, char *addr, int n)
6852 {
6853     int i;
6854
6855     acquire(&p->lock);
6856     while(p->nread == p->nwrite && p->writeopen){
6857         if(myproc()->killed){
6858             release(&p->lock);
6859             return -1;
6860         }
6861         sleep(&p->nread, &p->lock);
6862     }
6863     for(i = 0; i < n; i++){
6864         if(p->nread == p->nwrite)
6865             break;
6866         addr[i] = p->data[p->nread++ % PIPESIZE];
6867     }
6868     wakeup(&p->nwrite);
6869     release(&p->lock);
6870     return i;
6871 }
6872
6873
6874
6875
6876
6877
6878
6879
6880
6881
6882
6883
6884
6885
6886
6887
6888
6889
6890
6891
6892
6893
6894
6895
6896
6897
6898
6899

```

```

6900 #include "types.h"
6901 #include "x86.h"
6902
6903 void*
6904 memset(void *dst, int c, uint n)
6905 {
6906     if ((int)dst%4 == 0 && n%4 == 0){
6907         c &= 0xFF;
6908         stosl(dst, (c<<24) | (c<<16) | (c<<8) | c, n/4);
6909     } else
6910         stosb(dst, c, n);
6911     return dst;
6912 }
6913
6914 int
6915 memcmp(const void *v1, const void *v2, uint n)
6916 {
6917     const uchar *s1, *s2;
6918
6919     s1 = v1;
6920     s2 = v2;
6921     while(n-- > 0){
6922         if(*s1 != *s2)
6923             return *s1 - *s2;
6924         s1++, s2++;
6925     }
6926
6927     return 0;
6928 }
6929
6930 void*
6931 memmove(void *dst, const void *src, uint n)
6932 {
6933     const char *s;
6934     char *d;
6935
6936     s = src;
6937     d = dst;
6938     if(s < d && s + n > d){
6939         s += n;
6940         d += n;
6941         while(n-- > 0)
6942             *--d = *--s;
6943     } else
6944         while(n-- > 0)
6945             *d++ = *s++;
6946
6947     return dst;
6948 }
6949

```

```

6950 // memcpy exists to placate GCC. Use memmove.
6951 void*
6952 memcpy(void *dst, const void *src, uint n)
6953 {
6954     return memmove(dst, src, n);
6955 }
6956
6957 int
6958 strncmp(const char *p, const char *q, uint n)
6959 {
6960     while(n > 0 && *p && *p == *q)
6961         n--, p++, q++;
6962     if(n == 0)
6963         return 0;
6964     return (uchar)*p - (uchar)*q;
6965 }
6966
6967 char*
6968 strncpy(char *s, const char *t, int n)
6969 {
6970     char *os;
6971
6972     os = s;
6973     while(n-- > 0 && (*s++ = *t++) != 0)
6974         ;
6975     while(n-- > 0)
6976         *s++ = 0;
6977     return os;
6978 }
6979
6980 // Like strncpy but guaranteed to NUL-terminate.
6981 char*
6982 safestrcpy(char *s, const char *t, int n)
6983 {
6984     char *os;
6985
6986     os = s;
6987     if(n <= 0)
6988         return os;
6989     while(--n > 0 && (*s++ = *t++) != 0)
6990         ;
6991     *s = 0;
6992     return os;
6993 }
6994
6995
6996
6997
6998
6999

```

```

7000 int
7001 strlen(const char *s)
7002 {
7003     int n;
7004
7005     for(n = 0; s[n]; n++)
7006         ;
7007     return n;
7008 }
7009
7010
7011
7012
7013
7014
7015
7016
7017
7018
7019
7020
7021
7022
7023
7024
7025
7026
7027
7028
7029
7030
7031
7032
7033
7034
7035
7036
7037
7038
7039
7040
7041
7042
7043
7044
7045
7046
7047
7048
7049

```

```

7050 // See MultiProcessor Specification Version 1.[14]
7051
7052 struct mp {          // floating pointer
7053     uchar signature[4]; // "MP_"
7054     void *physaddr;     // phys addr of MP config table
7055     uchar length;       // 1
7056     uchar specrev;      // [14]
7057     uchar checksum;     // all bytes must add up to 0
7058     uchar type;         // MP system config type
7059     uchar imcrp;
7060     uchar reserved[3];
7061 };
7062
7063 struct mpconf {      // configuration table header
7064     uchar signature[4]; // "PCMP"
7065     ushort length;      // total table length
7066     uchar version;      // [14]
7067     uchar checksum;     // all bytes must add up to 0
7068     uchar product[20];  // product id
7069     uint *oemtable;     // OEM table pointer
7070     ushort oemlength;   // OEM table length
7071     ushort entry;       // entry count
7072     uint *lapicaddr;    // address of local APIC
7073     ushort xlength;     // extended table length
7074     uchar xchecksum;    // extended table checksum
7075     uchar reserved;
7076 };
7077
7078 struct mpproc {      // processor table entry
7079     uchar type;        // entry type (0)
7080     uchar apicid;      // local APIC id
7081     uchar version;     // local APIC version
7082     uchar flags;       // CPU flags
7083     #define MPBOOT 0x02 // This proc is the bootstrap processor.
7084     uchar signature[4]; // CPU signature
7085     uint feature;       // feature flags from CPUID instruction
7086     uchar reserved[8];
7087 };
7088
7089 struct mpioapic {    // I/O APIC table entry
7090     uchar type;       // entry type (2)
7091     uchar apicno;     // I/O APIC id
7092     uchar version;    // I/O APIC version
7093     uchar flags;      // I/O APIC flags
7094     uint *addr;       // I/O APIC address
7095 };
7096
7097
7098
7099

```

```

7100 // Table entry types
7101 #define MPPROC 0x00 // One per processor
7102 #define MPBUS 0x01 // One per bus
7103 #define MPIOAPIC 0x02 // One per I/O APIC
7104 #define MPIOINTR 0x03 // One per bus interrupt source
7105 #define MPLINTR 0x04 // One per system interrupt source
7106
7107
7108
7109
7110
7111
7112
7113
7114
7115
7116
7117
7118
7119
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 // Blank page.
7151
7152
7153
7154
7155
7156
7157
7158
7159
7160
7161
7162
7163
7164
7165
7166
7167
7168
7169
7170
7171
7172
7173
7174
7175
7176
7177
7178
7179
7180
7181
7182
7183
7184
7185
7186
7187
7188
7189
7190
7191
7192
7193
7194
7195
7196
7197
7198
7199

```

```

7200 // Multiprocessor support
7201 // Search memory for MP description structures.
7202 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
7203
7204 #include "types.h"
7205 #include "defs.h"
7206 #include "param.h"
7207 #include "memlayout.h"
7208 #include "mp.h"
7209 #include "x86.h"
7210 #include "mmu.h"
7211 #include "proc.h"
7212
7213 struct cpu cpus[NCPU];
7214 int ncpu;
7215 uchar ioapicid;
7216
7217 static uchar
7218 sum(uchar *addr, int len)
7219 {
7220     int i, sum;
7221
7222     sum = 0;
7223     for(i=0; i<len; i++)
7224         sum += addr[i];
7225     return sum;
7226 }
7227
7228 // Look for an MP structure in the len bytes at addr.
7229 static struct mp*
7230 mpsearch1(uint a, int len)
7231 {
7232     uchar *e, *p, *addr;
7233
7234     addr = P2V(a);
7235     e = addr+len;
7236     for(p = addr; p < e; p += sizeof(struct mp))
7237         if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
7238             return (struct mp*)p;
7239     return 0;
7240 }
7241
7242
7243
7244
7245
7246
7247
7248
7249

```

```

7250 // Search for the MP Floating Pointer Structure, which according to the
7251 // spec is in one of the following three locations:
7252 // 1) in the first KB of the EBDA;
7253 // 2) in the last KB of system base memory;
7254 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
7255 static struct mp*
7256 mpsearch(void)
7257 {
7258     uchar *bda;
7259     uint p;
7260     struct mp *mp;
7261
7262     bda = (uchar *) P2V(0x400);
7263     if((p = ((bda[0x0F]<<8) | bda[0x0E]) << 4)){
7264         if((mp = mpsearch1(p, 1024)))
7265             return mp;
7266     } else {
7267         p = ((bda[0x14]<<8) | bda[0x13])*1024;
7268         if((mp = mpsearch1(p-1024, 1024)))
7269             return mp;
7270     }
7271     return mpsearch1(0xF0000, 0x10000);
7272 }
7273
7274 // Search for an MP configuration table. For now,
7275 // don't accept the default configurations (physaddr == 0).
7276 // Check for correct signature, calculate the checksum and,
7277 // if correct, check the version.
7278 // To do: check extended table checksum.
7279 static struct mpconf*
7280 mpconfig(struct mp **pmp)
7281 {
7282     struct mpconf *conf;
7283     struct mp *mp;
7284
7285     if((mp = mpsearch()) == 0 || mp->physaddr == 0)
7286         return 0;
7287     conf = (struct mpconf*) P2V((uint) mp->physaddr);
7288     if(memcmp(conf, "PCMP", 4) != 0)
7289         return 0;
7290     if(conf->version != 1 && conf->version != 4)
7291         return 0;
7292     if(sum((uchar*)conf, conf->length) != 0)
7293         return 0;
7294     *pmp = mp;
7295     return conf;
7296 }
7297
7298
7299

```

```

7300 void
7301 mpinit(void)
7302 {
7303     uchar *p, *e;
7304     int ismp;
7305     struct mp *mp;
7306     struct mpconf *conf;
7307     struct mpproc *proc;
7308     struct mpioapic *ioapic;
7309
7310     if((conf = mpconfig(&mp)) == 0)
7311         panic("Expect to run on an SMP");
7312     ismp = 1;
7313     lapic = (uint*)conf->lapicaddr;
7314     for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
7315         switch(*p){
7316             case MPPROC:
7317                 proc = (struct mpproc*)p;
7318                 if(ncpu < NCPU) {
7319                     cpus[ncpu].apicid = proc->apicid; // apicid may differ from ncpu
7320                     ncpu++;
7321                 }
7322                 p += sizeof(struct mpproc);
7323                 continue;
7324             case MPIOAPIC:
7325                 ioapic = (struct mpioapic*)p;
7326                 ioapicid = ioapic->apicno;
7327                 p += sizeof(struct mpioapic);
7328                 continue;
7329             case MPBUS:
7330             case MPIOINTR:
7331             case MPLINTR:
7332                 p += 8;
7333                 continue;
7334             default:
7335                 ismp = 0;
7336                 break;
7337         }
7338     }
7339     if(!ismp)
7340         panic("Didn't find a suitable machine");
7341
7342     if(mp->imcrp){
7343         // Bochs doesn't support IMCR, so this doesn't run on Bochs.
7344         // But it would on real hardware.
7345         outb(0x22, 0x70); // Select IMCR
7346         outb(0x23, inb(0x23) | 1); // Mask external interrupts.
7347     }
7348 }
7349

```

```

7350 // The local APIC manages internal (non-I/O) interrupts.
7351 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
7352
7353 #include "param.h"
7354 #include "types.h"
7355 #include "defs.h"
7356 #include "date.h"
7357 #include "memlayout.h"
7358 #include "traps.h"
7359 #include "mmu.h"
7360 #include "x86.h"
7361
7362 // Local APIC registers, divided by 4 for use as uint[] indices.
7363 #define ID      (0x0020/4) // ID
7364 #define VER     (0x0030/4) // Version
7365 #define TPR     (0x0080/4) // Task Priority
7366 #define EOI     (0x00B0/4) // EOI
7367 #define SVR     (0x00F0/4) // Spurious Interrupt Vector
7368 #define ENABLE  0x00000100 // Unit Enable
7369 #define ESR     (0x0280/4) // Error Status
7370 #define ICRLO   (0x0300/4) // Interrupt Command
7371 #define INIT    0x00000500 // INIT/RESET
7372 #define STARTUP 0x00000600 // Startup IPI
7373 #define DELIVS  0x00001000 // Delivery status
7374 #define ASSERT  0x00004000 // Assert interrupt (vs deassert)
7375 #define DEASSERT 0x00000000
7376 #define LEVEL   0x00008000 // Level triggered
7377 #define BCAST   0x00008000 // Send to all APICs, including self.
7378 #define BUSY    0x00001000
7379 #define FIXED   0x00000000
7380 #define ICRHI   (0x0310/4) // Interrupt Command [63:32]
7381 #define TIMER   (0x0320/4) // Local Vector Table 0 (TIMER)
7382 #define X1      0x0000000B // divide counts by 1
7383 #define PERIODIC 0x00020000 // Periodic
7384 #define PCINT   (0x0340/4) // Performance Counter LVT
7385 #define LINT0   (0x0350/4) // Local Vector Table 1 (LINT0)
7386 #define LINT1   (0x0360/4) // Local Vector Table 2 (LINT1)
7387 #define ERROR   (0x0370/4) // Local Vector Table 3 (ERROR)
7388 #define MASKED  0x00010000 // Interrupt masked
7389 #define TICC    (0x0380/4) // Timer Initial Count
7390 #define TCCR    (0x0390/4) // Timer Current Count
7391 #define TDCR    (0x03E0/4) // Timer Divide Configuration
7392
7393 volatile uint *lapic; // Initialized in mp.c
7394
7395
7396
7397
7398
7399

```

```

7400 static void
7401 lapicw(int index, int value)
7402 {
7403     lapic[index] = value;
7404     lapic[ID]; // wait for write to finish, by reading
7405 }
7406
7407 void
7408 lapicinit(void)
7409 {
7410     if(!lapic)
7411         return;
7412
7413     // Enable local APIC; set spurious interrupt vector.
7414     lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
7415
7416     // The timer repeatedly counts down at bus frequency
7417     // from lapic[TICC] and then issues an interrupt.
7418     // If xv6 cared more about precise timekeeping,
7419     // TICC would be calibrated using an external time source.
7420     lapicw(TDCR, X1);
7421     lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
7422     lapicw(TICC, 10000000);
7423
7424     // Disable logical interrupt lines.
7425     lapicw(LINT0, MASKED);
7426     lapicw(LINT1, MASKED);
7427
7428     // Disable performance counter overflow interrupts
7429     // on machines that provide that interrupt entry.
7430     if(((lapic[VER]>>16) & 0xFF) >= 4)
7431         lapicw(PCINT, MASKED);
7432
7433     // Map error interrupt to IRQ_ERROR.
7434     lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
7435
7436     // Clear error status register (requires back-to-back writes).
7437     lapicw(ESR, 0);
7438     lapicw(ESR, 0);
7439
7440     // Ack any outstanding interrupts.
7441     lapicw(EOI, 0);
7442
7443     // Send an Init Level De-Assert to synchronise arbitration ID's.
7444     lapicw(ICRHI, 0);
7445     lapicw(ICRLO, BCAST | INIT | LEVEL);
7446     while(lapic[ICRLO] & DELIVS)
7447         ;
7448
7449

```

```

7450 // Enable interrupts on the APIC (but not on the processor).
7451 lapicw(TPR, 0);
7452 }
7453
7454 int
7455 lapicid(void)
7456 {
7457     if (!lapic)
7458         return 0;
7459     return lapic[ID] >> 24;
7460 }
7461
7462 // Acknowledge interrupt.
7463 void
7464 lapiceoi(void)
7465 {
7466     if (lapic)
7467         lapicw(EOI, 0);
7468 }
7469
7470 // Spin for a given number of microseconds.
7471 // On real hardware would want to tune this dynamically.
7472 void
7473 microdelay(int us)
7474 {
7475 }
7476
7477 #define CMOS_PORT    0x70
7478 #define CMOS_RETURN  0x71
7479
7480 // Start additional processor running entry code at addr.
7481 // See Appendix B of MultiProcessor Specification.
7482 void
7483 lapicstartap(uchar apicid, uint addr)
7484 {
7485     int i;
7486     ushort *wrv;
7487
7488     // "The BSP must initialize CMOS shutdown code to 0AH
7489     // and the warm reset vector (DWORD based at 40:67) to point at
7490     // the AP startup code prior to the [universal startup algorithm]".
7491     outb(CMOS_PORT, 0xF); // offset 0xF is shutdown code
7492     outb(CMOS_PORT+1, 0x0A);
7493     wrv = (ushort*)P2V((0x40<<4 | 0x67)); // Warm reset vector
7494     wrv[0] = 0;
7495     wrv[1] = addr >> 4;
7496
7497
7498
7499

```

```

7500 // "Universal startup algorithm."
7501 // Send INIT (level-triggered) interrupt to reset other CPU.
7502 lapicw(ICRHI, apicid<<24);
7503 lapicw(ICRLO, INIT | LEVEL | ASSERT);
7504 microdelay(200);
7505 lapicw(ICRLO, INIT | LEVEL);
7506 microdelay(100); // should be 10ms, but too slow in Bochs!
7507
7508 // Send startup IPI (twice!) to enter code.
7509 // Regular hardware is supposed to only accept a STARTUP
7510 // when it is in the halted state due to an INIT. So the second
7511 // should be ignored, but it is part of the official Intel algorithm.
7512 // Bochs complains about the second one. Too bad for Bochs.
7513 for(i = 0; i < 2; i++){
7514     lapicw(ICRHI, apicid<<24);
7515     lapicw(ICRLO, STARTUP | (addr>>12));
7516     microdelay(200);
7517 }
7518 }
7519
7520 #define CMOS_STATA    0x0a
7521 #define CMOS_STATB    0x0b
7522 #define CMOS_UIP      (1 << 7) // RTC update in progress
7523
7524 #define SECS    0x00
7525 #define MINS    0x02
7526 #define HOURS   0x04
7527 #define DAY      0x07
7528 #define MONTH    0x08
7529 #define YEAR     0x09
7530
7531 static uint
7532 cmos_read(uint reg)
7533 {
7534     outb(CMOS_PORT, reg);
7535     microdelay(200);
7536
7537     return inb(CMOS_RETURN);
7538 }
7539
7540 static void
7541 fill_rtcdate(struct rtcdate *r)
7542 {
7543     r->second = cmos_read(SECS);
7544     r->minute = cmos_read(MINS);
7545     r->hour   = cmos_read(HOURS);
7546     r->day    = cmos_read(DAY);
7547     r->month  = cmos_read(MONTH);
7548     r->year   = cmos_read(YEAR);
7549 }

```

```

7550 // qemu seems to use 24-hour GWT and the values are BCD encoded
7551 void
7552 cmostime(struct rtcdate *r)
7553 {
7554     struct rtcdate t1, t2;
7555     int sb, bcd;
7556
7557     sb = cmos_read(CMOS_STATB);
7558
7559     bcd = (sb & (1 << 2)) == 0;
7560
7561     // make sure CMOS doesn't modify time while we read it
7562     for(;;) {
7563         fill_rtcdate(&t1);
7564         if(cmos_read(CMOS_STATB) & CMOS_UIP)
7565             continue;
7566         fill_rtcdate(&t2);
7567         if(memcmp(&t1, &t2, sizeof(t1)) == 0)
7568             break;
7569     }
7570
7571     // convert
7572     if(bcd) {
7573 #define CONV(x)      (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
7574         CONV(second);
7575         CONV(minute);
7576         CONV(hour);
7577         CONV(day);
7578         CONV(month);
7579         CONV(year);
7580 #undef CONV
7581     }
7582
7583     *r = t1;
7584     r->year += 2000;
7585 }
7586
7587
7588
7589
7590
7591
7592
7593
7594
7595
7596
7597
7598
7599

```

```

7600 // The I/O APIC manages hardware interrupts for an SMP system.
7601 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
7602 // See also picirq.c.
7603
7604 #include "types.h"
7605 #include "defs.h"
7606 #include "traps.h"
7607
7608 #define IOAPIC    0xFEC00000    // Default physical address of IO APIC
7609
7610 #define REG_ID     0x00    // Register index: ID
7611 #define REG_VER    0x01    // Register index: version
7612 #define REG_TABLE  0x10    // Redirection table base
7613
7614 // The redirection table starts at REG_TABLE and uses
7615 // two registers to configure each interrupt.
7616 // The first (low) register in a pair contains configuration bits.
7617 // The second (high) register contains a bitmask telling which
7618 // CPUs can serve that interrupt.
7619 #define INT_DISABLED 0x00010000 // Interrupt disabled
7620 #define INT_LEVEL    0x00008000 // Level-triggered (vs edge-)
7621 #define INT_ACTIVELOW 0x00002000 // Active low (vs high)
7622 #define INT_LOGICAL  0x00000800 // Destination is CPU id (vs APIC ID)
7623
7624 volatile struct ioapic *ioapic;
7625
7626 // IO APIC MMIO structure: write reg, then read or write data.
7627 struct ioapic {
7628     uint reg;
7629     uint pad[3];
7630     uint data;
7631 };
7632
7633 static uint
7634 ioapicread(int reg)
7635 {
7636     ioapic->reg = reg;
7637     return ioapic->data;
7638 }
7639
7640 static void
7641 ioapicwrite(int reg, uint data)
7642 {
7643     ioapic->reg = reg;
7644     ioapic->data = data;
7645 }
7646
7647
7648
7649

```

```

7650 void
7651 ioapicinit(void)
7652 {
7653     int i, id, maxintr;
7654
7655     ioapic = (volatile struct ioapic*)IOAPIC;
7656     maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
7657     id = ioapicread(REG_ID) >> 24;
7658     if(id != ioapicid)
7659         cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
7660
7661     // Mark all interrupts edge-triggered, active high, disabled,
7662     // and not routed to any CPUs.
7663     for(i = 0; i <= maxintr; i++){
7664         ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
7665         ioapicwrite(REG_TABLE+2*i+1, 0);
7666     }
7667 }
7668
7669 void
7670 ioapicenable(int irq, int cpunum)
7671 {
7672     // Mark interrupt edge-triggered, active high,
7673     // enabled, and routed to the given cpunum,
7674     // which happens to be that cpu's APIC ID.
7675     ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
7676     ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
7677 }
7678
7679
7680
7681
7682
7683
7684
7685
7686
7687
7688
7689
7690
7691
7692
7693
7694
7695
7696
7697
7698
7699

```

```

7700 // PC keyboard interface constants
7701
7702 #define KBSTATP      0x64    // kbd controller status port(I)
7703 #define KBS_DIB      0x01    // kbd data in buffer
7704 #define KBDATAP      0x60    // kbd data port(I)
7705
7706 #define NO            0
7707
7708 #define SHIFT         (1<<0)
7709 #define CTL           (1<<1)
7710 #define ALT           (1<<2)
7711
7712 #define CAPSLOCK      (1<<3)
7713 #define NUMLOCK       (1<<4)
7714 #define SCROLLLOCK    (1<<5)
7715
7716 #define E0ESC         (1<<6)
7717
7718 // Special keycodes
7719 #define KEY_HOME      0xE0
7720 #define KEY_END       0xE1
7721 #define KEY_UP        0xE2
7722 #define KEY_DN        0xE3
7723 #define KEY_LF        0xE4
7724 #define KEY_RT        0xE5
7725 #define KEY_PGUP      0xE6
7726 #define KEY_PGDN      0xE7
7727 #define KEY_INS       0xE8
7728 #define KEY_DEL       0xE9
7729
7730 // C('A') == Control-A
7731 #define C(x) (x - '@')
7732
7733 static uchar shiftcode[256] =
7734 {
7735     [0x1D] CTL,
7736     [0x2A] SHIFT,
7737     [0x36] SHIFT,
7738     [0x38] ALT,
7739     [0x9D] CTL,
7740     [0xB8] ALT
7741 };
7742
7743 static uchar togglecode[256] =
7744 {
7745     [0x3A] CAPSLOCK,
7746     [0x45] NUMLOCK,
7747     [0x46] SCROLLLOCK
7748 };
7749

```

```

7750 static uchar normalmap[256] =
7751 {
7752     NO,    0x1B, '1', '2', '3', '4', '5', '6', // 0x00
7753     '7', '8', '9', '0', '-', '=', '\b', '\t',
7754     'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', // 0x10
7755     'o', 'p', '[', ']', '\n', NO, 'a', 's',
7756     'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', // 0x20
7757     '\'', ' ', NO, '\\', 'z', 'x', 'c', 'v',
7758     'b', 'n', 'm', ',', '.', '/', NO, '*', // 0x30
7759     NO, ' ', NO, NO, NO, NO, NO, NO,
7760     NO, NO, NO, NO, NO, NO, NO, '7', // 0x40
7761     '8', '9', '-', '4', '5', '6', '+', '1',
7762     '2', '3', '0', '.', NO, NO, NO, NO, // 0x50
7763     [0x9C] '\n', // KP_Enter
7764     [0xB5] '/', // KP_Div
7765     [0xC8] KEY_UP, [0xD0] KEY_DN,
7766     [0xC9] KEY_PGUP, [0xD1] KEY_PGDN,
7767     [0xCB] KEY_LF, [0xCD] KEY_RT,
7768     [0x97] KEY_HOME, [0xCF] KEY_END,
7769     [0xD2] KEY_INS, [0xD3] KEY_DEL
7770 };
7771
7772 static uchar shiftmap[256] =
7773 {
7774     NO,    033, '!', '@', '#', '$', '%', '^', // 0x00
7775     '&', '*', '(', ')', '_', '+', '\b', '\t',
7776     'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', // 0x10
7777     'O', 'P', '[', ']', '\n', NO, 'A', 'S',
7778     'D', 'F', 'G', 'H', 'J', 'K', 'L', ':', // 0x20
7779     '"', '~', NO, '|', 'Z', 'X', 'C', 'V',
7780     'B', 'N', 'M', '<', '>', '?', NO, '*', // 0x30
7781     NO, ' ', NO, NO, NO, NO, NO, NO,
7782     NO, NO, NO, NO, NO, NO, NO, '7', // 0x40
7783     '8', '9', '-', '4', '5', '6', '+', '1',
7784     '2', '3', '0', '.', NO, NO, NO, NO, // 0x50
7785     [0x9C] '\n', // KP_Enter
7786     [0xB5] '/', // KP_Div
7787     [0xC8] KEY_UP, [0xD0] KEY_DN,
7788     [0xC9] KEY_PGUP, [0xD1] KEY_PGDN,
7789     [0xCB] KEY_LF, [0xCD] KEY_RT,
7790     [0x97] KEY_HOME, [0xCF] KEY_END,
7791     [0xD2] KEY_INS, [0xD3] KEY_DEL
7792 };
7793
7794
7795
7796
7797
7798
7799

```

```

7800 static uchar ctlmap[256] =
7801 {
7802     NO,    NO,    NO,    NO,    NO,    NO,    NO,    NO,
7803     NO,    NO,    NO,    NO,    NO,    NO,    NO,    NO,
7804     C('Q'), C('W'), C('E'), C('R'), C('T'), C('Y'), C('U'), C('I'),
7805     C('O'), C('P'), NO,    NO,    '\r', NO,    C('A'), C('S'),
7806     C('D'), C('F'), C('G'), C('H'), C('J'), C('K'), C('L'), NO,
7807     NO,    NO,    NO,    C('\'), C('Z'), C('X'), C('C'), C('V'),
7808     C('B'), C('N'), C('M'), NO,    NO,    C('/'), NO,    NO,
7809     [0x9C] '\r', // KP_Enter
7810     [0xB5] C('/'), // KP_Div
7811     [0xC8] KEY_UP, [0xD0] KEY_DN,
7812     [0xC9] KEY_PGUP, [0xD1] KEY_PGDN,
7813     [0xCB] KEY_LF, [0xCD] KEY_RT,
7814     [0x97] KEY_HOME, [0xCF] KEY_END,
7815     [0xD2] KEY_INS, [0xD3] KEY_DEL
7816 };
7817
7818
7819
7820
7821
7822
7823
7824
7825
7826
7827
7828
7829
7830
7831
7832
7833
7834
7835
7836
7837
7838
7839
7840
7841
7842
7843
7844
7845
7846
7847
7848
7849

```

```

7850 #include "types.h"
7851 #include "x86.h"
7852 #include "defs.h"
7853 #include "kbd.h"
7854
7855 int
7856 kbdgetc(void)
7857 {
7858     static uint shift;
7859     static uchar *charcode[4] = {
7860         normalmap, shiftmap, ctlmap, ctlmap
7861     };
7862     uint st, data, c;
7863
7864     st = inb(KBSTATP);
7865     if((st & KBS_DIB) == 0)
7866         return -1;
7867     data = inb(KBDATAP);
7868
7869     if(data == 0xE0){
7870         shift |= EOESC;
7871         return 0;
7872     } else if(data & 0x80){
7873         // Key released
7874         data = (shift & EOESC ? data : data & 0x7F);
7875         shift &= ~(shiftcode[data] | EOESC);
7876         return 0;
7877     } else if(shift & EOESC){
7878         // Last character was an E0 escape; or with 0x80
7879         data |= 0x80;
7880         shift &= ~EOESC;
7881     }
7882
7883     shift |= shiftcode[data];
7884     shift ^= togglecode[data];
7885     c = charcode[shift & (CTL | SHIFT)][data];
7886     if(shift & CAPSLOCK){
7887         if('a' <= c && c <= 'z')
7888             c += 'A' - 'a';
7889         else if('A' <= c && c <= 'Z')
7890             c += 'a' - 'A';
7891     }
7892     return c;
7893 }
7894
7895 void
7896 kbdintr(void)
7897 {
7898     consoleintr(kbdgetc);
7899 }

```

```

7900 // Console input and output.
7901 // Input is from the keyboard or serial port.
7902 // Output is written to the screen and serial port.
7903
7904 #include "types.h"
7905 #include "defs.h"
7906 #include "param.h"
7907 #include "traps.h"
7908 #include "spinlock.h"
7909 #include "sleeplock.h"
7910 #include "fs.h"
7911 #include "file.h"
7912 #include "memlayout.h"
7913 #include "mmu.h"
7914 #include "proc.h"
7915 #include "x86.h"
7916
7917 static void consputc(int);
7918
7919 static int panicked = 0;
7920
7921 static struct {
7922     struct spinlock lock;
7923     int locking;
7924 } cons;
7925
7926 static void
7927 printint(int xx, int base, int sign)
7928 {
7929     static char digits[] = "0123456789abcdef";
7930     char buf[16];
7931     int i;
7932     uint x;
7933
7934     if(sign && (sign = xx < 0))
7935         x = -xx;
7936     else
7937         x = xx;
7938
7939     i = 0;
7940     do{
7941         buf[i++] = digits[x % base];
7942     }while((x /= base) != 0);
7943
7944     if(sign)
7945         buf[i++] = '-';
7946
7947     while(--i >= 0)
7948         consputc(buf[i]);
7949 }

```


7950
7951
7952
7953
7954
7955
7956
7957
7958
7959
7960
7961
7962
7963
7964
7965
7966
7967
7968
7969
7970
7971
7972
7973
7974
7975
7976
7977
7978
7979
7980
7981
7982
7983
7984
7985
7986
7987
7988
7989
7990
7991
7992
7993
7994
7995
7996
7997
7998
7999

```
8000 // Print to the console. only understands %d, %x, %p, %s.
8001 void
8002 cprintf(char *fmt, ...)
8003 {
8004     int i, c, locking;
8005     uint *argp;
8006     char *s;
8007
8008     locking = cons.locking;
8009     if(locking)
8010         acquire(&cons.lock);
8011
8012     if (fmt == 0)
8013         panic("null fmt");
8014
8015     argp = (uint*)(void*)&fmt + 1;
8016     for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
8017         if(c != '%'){
8018             consputc(c);
8019             continue;
8020         }
8021         c = fmt[++i] & 0xff;
8022         if(c == 0)
8023             break;
8024         switch(c){
8025             case 'd':
8026                 printint(*argp++, 10, 1);
8027                 break;
8028             case 'x':
8029             case 'p':
8030                 printint(*argp++, 16, 0);
8031                 break;
8032             case 's':
8033                 if((s = (char*)*argp++) == 0)
8034                     s = "(null)";
8035                 for(; *s; s++)
8036                     consputc(*s);
8037                 break;
8038             case '%':
8039                 consputc('%');
8040                 break;
8041             default:
8042                 // Print unknown % sequence to draw attention.
8043                 consputc('%');
8044                 consputc(c);
8045                 break;
8046         }
8047     }
8048
8049
```

```

8050  if(locking)
8051      release(&cons.lock);
8052  }
8053
8054  void
8055  panic(char *s)
8056  {
8057      int i;
8058      uint pcs[10];
8059
8060      cli();
8061      cons.locking = 0;
8062      // use lapiccpunum so that we can call panic from mycpu()
8063      cprintf("lapicid %d: panic: ", lapicid());
8064      cprintf(s);
8065      cprintf("\n");
8066      getcallerpcs(&s, pcs);
8067      for(i=0; i<10; i++)
8068          cprintf(" %p", pcs[i]);
8069      panicked = 1; // freeze other CPU
8070      for(;;)
8071          ;
8072  }
8073
8074
8075
8076
8077
8078
8079
8080
8081
8082
8083
8084
8085
8086
8087
8088
8089
8090
8091
8092
8093
8094
8095
8096
8097
8098
8099

```

```

8100  #define BACKSPACE 0x100
8101  #define CRTPORT 0x3d4
8102  static ushort *crt = (ushort*)P2V(0xb8000); // CGA memory
8103
8104  static void
8105  cgaputc(int c)
8106  {
8107      int pos;
8108
8109      // Cursor position: col + 80*row.
8110      outb(CRTPORT, 14);
8111      pos = inb(CRTPORT+1) << 8;
8112      outb(CRTPORT, 15);
8113      pos |= inb(CRTPORT+1);
8114
8115      if(c == '\n')
8116          pos += 80 - pos%80;
8117      else if(c == BACKSPACE){
8118          if(pos > 0) --pos;
8119      } else
8120          crt[pos++] = (c&0xff) | 0x0700; // black on white
8121
8122      if(pos < 0 || pos > 25*80)
8123          panic("pos under/overflow");
8124
8125      if((pos/80) >= 24){ // Scroll up.
8126          memmove(crt, crt+80, sizeof(crt[0])*23*80);
8127          pos -= 80;
8128          memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
8129      }
8130
8131      outb(CRTPORT, 14);
8132      outb(CRTPORT+1, pos>>8);
8133      outb(CRTPORT, 15);
8134      outb(CRTPORT+1, pos);
8135      crt[pos] = ' ' | 0x0700;
8136  }
8137
8138
8139
8140
8141
8142
8143
8144
8145
8146
8147
8148
8149

```

```

8150 void
8151 consputc(int c)
8152 {
8153     if(panicked){
8154         cli();
8155         for(;;)
8156             ;
8157     }
8158
8159     if(c == BACKSPACE){
8160         uartputc('\b'); uartputc(' '); uartputc('\b');
8161     } else
8162         uartputc(c);
8163     cgaputc(c);
8164 }
8165
8166 #define INPUT_BUF 128
8167 struct {
8168     char buf[INPUT_BUF];
8169     uint r; // Read index
8170     uint w; // Write index
8171     uint e; // Edit index
8172 } input;
8173
8174 #define C(x) ((x)-'@') // Control-x
8175
8176 void
8177 consoleintr(int (*getc)(void))
8178 {
8179     int c, doprocdump = 0;
8180
8181     acquire(&cons.lock);
8182     while((c = getc()) >= 0){
8183         switch(c){
8184             case C('P'): // Process listing.
8185                 // procdump() locks cons.lock indirectly; invoke later
8186                 doprocdump = 1;
8187                 break;
8188             case C('U'): // Kill line.
8189                 while(input.e != input.w &&
8190                     input.buf[(input.e-1) % INPUT_BUF] != '\n'){
8191                     input.e--;
8192                     consputc(BACKSPACE);
8193                 }
8194                 break;
8195             case C('H'): case '\x7f': // Backspace
8196                 if(input.e != input.w){
8197                     input.e--;
8198                     consputc(BACKSPACE);
8199                 }

```

```

8200         break;
8201     default:
8202         if(c != 0 && input.e-input.r < INPUT_BUF){
8203             c = (c == '\r') ? '\n' : c;
8204             input.buf[input.e++ % INPUT_BUF] = c;
8205             consputc(c);
8206             if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
8207                 input.w = input.e;
8208                 wakeup(&input.r);
8209             }
8210         }
8211         break;
8212     }
8213 }
8214 release(&cons.lock);
8215 if(doprocdump) {
8216     procdump(); // now call procdump() wo. cons.lock held
8217 }
8218 }
8219
8220 int
8221 consleread(struct inode *ip, char *dst, int n)
8222 {
8223     uint target;
8224     int c;
8225
8226     iunlock(ip);
8227     target = n;
8228     acquire(&cons.lock);
8229     while(n > 0){
8230         while(input.r == input.w){
8231             if(myproc()->killed){
8232                 release(&cons.lock);
8233                 ilock(ip);
8234                 return -1;
8235             }
8236             sleep(&input.r, &cons.lock);
8237         }
8238         c = input.buf[input.r++ % INPUT_BUF];
8239         if(c == C('D')){ // EOF
8240             if(n < target){
8241                 // Save ^D for next time, to make sure
8242                 // caller gets a 0-byte result.
8243                 input.r--;
8244             }
8245             break;
8246         }
8247         *dst++ = c;
8248         --n;
8249         if(c == '\n')

```

```

8250     break;
8251 }
8252 release(&cons.lock);
8253 ilock(ip);
8254
8255 return target - n;
8256 }
8257
8258 int
8259 consolewrite(struct inode *ip, char *buf, int n)
8260 {
8261     int i;
8262
8263     iunlock(ip);
8264     acquire(&cons.lock);
8265     for(i = 0; i < n; i++)
8266         consputc(buf[i] & 0xff);
8267     release(&cons.lock);
8268     ilock(ip);
8269
8270     return n;
8271 }
8272
8273 void
8274 consoleinit(void)
8275 {
8276     initlock(&cons.lock, "console");
8277
8278     devsw[CONSOLE].write = consolewrite;
8279     devsw[CONSOLE].read = consolaread;
8280     cons.locking = 1;
8281
8282     ioapicenable(IRQ_KBD, 0);
8283 }
8284
8285
8286
8287
8288
8289
8290
8291
8292
8293
8294
8295
8296
8297
8298
8299

```

```

8300 // Intel 8250 serial port (UART).
8301
8302 #include "types.h"
8303 #include "defs.h"
8304 #include "param.h"
8305 #include "traps.h"
8306 #include "spinlock.h"
8307 #include "sleeplock.h"
8308 #include "fs.h"
8309 #include "file.h"
8310 #include "mmu.h"
8311 #include "proc.h"
8312 #include "x86.h"
8313
8314 #define COM1    0x3f8
8315
8316 static int uart;    // is there a uart?
8317
8318 void
8319 uartinit(void)
8320 {
8321     char *p;
8322
8323     // Turn off the FIFO
8324     outb(COM1+2, 0);
8325
8326     // 9600 baud, 8 data bits, 1 stop bit, parity off.
8327     outb(COM1+3, 0x80);    // Unlock divisor
8328     outb(COM1+0, 115200/9600);
8329     outb(COM1+1, 0);
8330     outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
8331     outb(COM1+4, 0);
8332     outb(COM1+1, 0x01);    // Enable receive interrupts.
8333
8334     // If status is 0xFF, no serial port.
8335     if(inb(COM1+5) == 0xFF)
8336         return;
8337     uart = 1;
8338
8339     // Acknowledge pre-existing interrupt conditions;
8340     // enable interrupts.
8341     inb(COM1+2);
8342     inb(COM1+0);
8343     ioapicenable(IRQ_COM1, 0);
8344
8345     // Announce that we're here.
8346     for(p="xv6...\n"; *p; p++)
8347         uartputc(*p);
8348 }
8349

```

```

8350 void
8351 uartputc(int c)
8352 {
8353     int i;
8354
8355     if(!uart)
8356         return;
8357     for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
8358         microdelay(10);
8359     outb(COM1+0, c);
8360 }
8361
8362 static int
8363 uartgetc(void)
8364 {
8365     if(!uart)
8366         return -1;
8367     if(!(inb(COM1+5) & 0x01))
8368         return -1;
8369     return inb(COM1+0);
8370 }
8371
8372 void
8373 uartintr(void)
8374 {
8375     consoleintr(uartgetc);
8376 }
8377
8378
8379
8380
8381
8382
8383
8384
8385
8386
8387
8388
8389
8390
8391
8392
8393
8394
8395
8396
8397
8398
8399

```

```

8400 # Initial process execs /init.
8401 # This code runs in user space.
8402
8403 #include "syscall.h"
8404 #include "traps.h"
8405
8406
8407 # exec(init, argv)
8408 .globl start
8409 start:
8410     pushl $argv
8411     pushl $init
8412     pushl $0 // where caller pc would be
8413     movl $SYS_exec, %eax
8414     int $T_SYSCALL
8415
8416 # for(;;) exit();
8417 exit:
8418     movl $SYS_exit, %eax
8419     int $T_SYSCALL
8420     jmp exit
8421
8422 # char init[] = "/init\0";
8423 init:
8424     .string "/init\0"
8425
8426 # char *argv[] = { init, 0 };
8427 .p2align 2
8428 argv:
8429     .long init
8430     .long 0
8431
8432
8433
8434
8435
8436
8437
8438
8439
8440
8441
8442
8443
8444
8445
8446
8447
8448
8449

```

```

8450 #include "syscall.h"
8451 #include "traps.h"
8452
8453 #define SYSCALL(name) \
8454     .globl name; \
8455     name: \
8456         movl $SYS_ ## name, %eax; \
8457         int $T_SYSCALL; \
8458         ret
8459
8460 SYSCALL(fork)
8461 SYSCALL(exit)
8462 SYSCALL(wait)
8463 SYSCALL(pipe)
8464 SYSCALL(read)
8465 SYSCALL(write)
8466 SYSCALL(close)
8467 SYSCALL(kill)
8468 SYSCALL(exec)
8469 SYSCALL(open)
8470 SYSCALL(mknod)
8471 SYSCALL(unlink)
8472 SYSCALL(fstat)
8473 SYSCALL(link)
8474 SYSCALL(mkdir)
8475 SYSCALL(chdir)
8476 SYSCALL(dup)
8477 SYSCALL(getpid)
8478 SYSCALL(sbrk)
8479 SYSCALL(sleep)
8480 SYSCALL(uptime)
8481
8482
8483
8484
8485
8486
8487
8488
8489
8490
8491
8492
8493
8494
8495
8496
8497
8498
8499

```

```

8500 // init: The initial user-level program
8501
8502 #include "types.h"
8503 #include "stat.h"
8504 #include "user.h"
8505 #include "fcntl.h"
8506
8507 char *argv[] = { "sh", 0 };
8508
8509 int
8510 main(void)
8511 {
8512     int pid, wpid;
8513
8514     if(open("console", O_RDWR) < 0){
8515         mknod("console", 1, 1);
8516         open("console", O_RDWR);
8517     }
8518     dup(0); // stdout
8519     dup(0); // stderr
8520
8521     for(;;){
8522         printf(1, "init: starting sh\n");
8523         pid = fork();
8524         if(pid < 0){
8525             printf(1, "init: fork failed\n");
8526             exit();
8527         }
8528         if(pid == 0){
8529             exec("sh", argv);
8530             printf(1, "init: exec sh failed\n");
8531             exit();
8532         }
8533         while((wpid=wait()) >= 0 && wpid != pid)
8534             printf(1, "zombie!\n");
8535     }
8536 }
8537
8538
8539
8540
8541
8542
8543
8544
8545
8546
8547
8548
8549

```

```

8550 // Shell.
8551
8552 #include "types.h"
8553 #include "user.h"
8554 #include "fcntl.h"
8555
8556 // Parsed command representation
8557 #define EXEC 1
8558 #define REDIR 2
8559 #define PIPE 3
8560 #define LIST 4
8561 #define BACK 5
8562
8563 #define MAXARGS 10
8564
8565 struct cmd {
8566     int type;
8567 };
8568
8569 struct execcmd {
8570     int type;
8571     char *argv[MAXARGS];
8572     char *eargv[MAXARGS];
8573 };
8574
8575 struct redircmd {
8576     int type;
8577     struct cmd *cmd;
8578     char *file;
8579     char *efile;
8580     int mode;
8581     int fd;
8582 };
8583
8584 struct pipecmd {
8585     int type;
8586     struct cmd *left;
8587     struct cmd *right;
8588 };
8589
8590 struct listcmd {
8591     int type;
8592     struct cmd *left;
8593     struct cmd *right;
8594 };
8595
8596 struct backcmd {
8597     int type;
8598     struct cmd *cmd;
8599 };

```

```

8600 int fork1(void); // Fork but panics on failure.
8601 void panic(char*);
8602 struct cmd *parsecmd(char*);
8603
8604 // Execute cmd. Never returns.
8605 void
8606 runcmd(struct cmd *cmd)
8607 {
8608     int p[2];
8609     struct backcmd *bcmd;
8610     struct execcmd *ecmd;
8611     struct listcmd *lcmd;
8612     struct pipecmd *pcmd;
8613     struct redircmd *rcmd;
8614
8615     if(cmd == 0)
8616         exit();
8617
8618     switch(cmd->type){
8619     default:
8620         panic("runcmd");
8621
8622     case EXEC:
8623         ecmd = (struct execcmd*)cmd;
8624         if(ecmd->argv[0] == 0)
8625             exit();
8626         exec(ecmd->argv[0], ecmd->argv);
8627         printf(2, "exec %s failed\n", ecmd->argv[0]);
8628         break;
8629
8630     case REDIR:
8631         rcmd = (struct redircmd*)cmd;
8632         close(rcmd->fd);
8633         if(open(rcmd->file, rcmd->mode) < 0){
8634             printf(2, "open %s failed\n", rcmd->file);
8635             exit();
8636         }
8637         runcmd(rcmd->cmd);
8638         break;
8639
8640     case LIST:
8641         lcmd = (struct listcmd*)cmd;
8642         if(fork1() == 0)
8643             runcmd(lcmd->left);
8644         wait();
8645         runcmd(lcmd->right);
8646         break;
8647
8648
8649

```

```

8650 case PIPE:
8651     pcmd = (struct pipecmd*)cmd;
8652     if(pipe(p) < 0)
8653         panic("pipe");
8654     if(fork1() == 0){
8655         close(1);
8656         dup(p[1]);
8657         close(p[0]);
8658         close(p[1]);
8659         runcmd(pcmd->left);
8660     }
8661     if(fork1() == 0){
8662         close(0);
8663         dup(p[0]);
8664         close(p[0]);
8665         close(p[1]);
8666         runcmd(pcmd->right);
8667     }
8668     close(p[0]);
8669     close(p[1]);
8670     wait();
8671     wait();
8672     break;
8673
8674 case BACK:
8675     bcmd = (struct backcmd*)cmd;
8676     if(fork1() == 0)
8677         runcmd(bcmd->cmd);
8678     break;
8679 }
8680 exit();
8681 }
8682
8683 int
8684 getcmd(char *buf, int nbuf)
8685 {
8686     printf(2, "$ ");
8687     memset(buf, 0, nbuf);
8688     gets(buf, nbuf);
8689     if(buf[0] == 0) // EOF
8690         return -1;
8691     return 0;
8692 }
8693
8694
8695
8696
8697
8698
8699

```

```

8700 int
8701 main(void)
8702 {
8703     static char buf[100];
8704     int fd;
8705
8706     // Ensure that three file descriptors are open.
8707     while((fd = open("console", O_RDWR)) >= 0){
8708         if(fd >= 3){
8709             close(fd);
8710             break;
8711         }
8712     }
8713
8714     // Read and run input commands.
8715     while(getcmd(buf, sizeof(buf)) >= 0){
8716         if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
8717             // Chdir must be called by the parent, not the child.
8718             buf[strlen(buf)-1] = 0; // chop \n
8719             if(chdir(buf+3) < 0)
8720                 printf(2, "cannot cd %s\n", buf+3);
8721             continue;
8722         }
8723         if(fork1() == 0)
8724             runcmd(parsecmd(buf));
8725         wait();
8726     }
8727     exit();
8728 }
8729
8730 void
8731 panic(char *s)
8732 {
8733     printf(2, "%s\n", s);
8734     exit();
8735 }
8736
8737 int
8738 fork1(void)
8739 {
8740     int pid;
8741
8742     pid = fork();
8743     if(pid == -1)
8744         panic("fork");
8745     return pid;
8746 }
8747
8748
8749

```



```

8750 // Constructors
8751
8752 struct cmd*
8753 execcmd(void)
8754 {
8755     struct execcmd *cmd;
8756
8757     cmd = malloc(sizeof(*cmd));
8758     memset(cmd, 0, sizeof(*cmd));
8759     cmd->type = EXEC;
8760     return (struct cmd*)cmd;
8761 }
8762
8763 struct cmd*
8764 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
8765 {
8766     struct redircmd *cmd;
8767
8768     cmd = malloc(sizeof(*cmd));
8769     memset(cmd, 0, sizeof(*cmd));
8770     cmd->type = REDIR;
8771     cmd->cmd = subcmd;
8772     cmd->file = file;
8773     cmd->efile = efile;
8774     cmd->mode = mode;
8775     cmd->fd = fd;
8776     return (struct cmd*)cmd;
8777 }
8778
8779 struct cmd*
8780 pipecmd(struct cmd *left, struct cmd *right)
8781 {
8782     struct pipecmd *cmd;
8783
8784     cmd = malloc(sizeof(*cmd));
8785     memset(cmd, 0, sizeof(*cmd));
8786     cmd->type = PIPE;
8787     cmd->left = left;
8788     cmd->right = right;
8789     return (struct cmd*)cmd;
8790 }
8791
8792
8793
8794
8795
8796
8797
8798
8799

```

```

8800 struct cmd*
8801 listcmd(struct cmd *left, struct cmd *right)
8802 {
8803     struct listcmd *cmd;
8804
8805     cmd = malloc(sizeof(*cmd));
8806     memset(cmd, 0, sizeof(*cmd));
8807     cmd->type = LIST;
8808     cmd->left = left;
8809     cmd->right = right;
8810     return (struct cmd*)cmd;
8811 }
8812
8813 struct cmd*
8814 backcmd(struct cmd *subcmd)
8815 {
8816     struct backcmd *cmd;
8817
8818     cmd = malloc(sizeof(*cmd));
8819     memset(cmd, 0, sizeof(*cmd));
8820     cmd->type = BACK;
8821     cmd->cmd = subcmd;
8822     return (struct cmd*)cmd;
8823 }
8824
8825
8826
8827
8828
8829
8830
8831
8832
8833
8834
8835
8836
8837
8838
8839
8840
8841
8842
8843
8844
8845
8846
8847
8848
8849

```

```

8850 // Parsing
8851
8852 char whitespace[] = " \t\r\n\v";
8853 char symbols[] = "<|>&()";
8854
8855 int
8856 gettoken(char **ps, char *es, char **q, char **eq)
8857 {
8858     char *s;
8859     int ret;
8860
8861     s = *ps;
8862     while(s < es && strchr(whitespace, *s))
8863         s++;
8864     if(*q)
8865         *q = s;
8866     ret = *s;
8867     switch(*s){
8868     case 0:
8869         break;
8870     case '|':
8871     case '(':
8872     case ')':
8873     case ';':
8874     case '&':
8875     case '<':
8876         s++;
8877         break;
8878     case '>':
8879         s++;
8880         if(*s == '>'){
8881             ret = '+';
8882             s++;
8883         }
8884         break;
8885     default:
8886         ret = 'a';
8887         while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
8888             s++;
8889         break;
8890     }
8891     if(eq)
8892         *eq = s;
8893
8894     while(s < es && strchr(whitespace, *s))
8895         s++;
8896     *ps = s;
8897     return ret;
8898 }
8899

```

```

8900 int
8901 peek(char **ps, char *es, char *toks)
8902 {
8903     char *s;
8904
8905     s = *ps;
8906     while(s < es && strchr(whitespace, *s))
8907         s++;
8908     *ps = s;
8909     return *s && strchr(toks, *s);
8910 }
8911
8912 struct cmd *parseline(char**, char*);
8913 struct cmd *parsepipe(char**, char*);
8914 struct cmd *parseexec(char**, char*);
8915 struct cmd *nulterminate(struct cmd*);
8916
8917 struct cmd*
8918 parsecmd(char *s)
8919 {
8920     char *es;
8921     struct cmd *cmd;
8922
8923     es = s + strlen(s);
8924     cmd = parseline(&s, es);
8925     peek(&s, es, "");
8926     if(s != es){
8927         printf(2, "leftovers: %s\n", s);
8928         panic("syntax");
8929     }
8930     nulterminate(cmd);
8931     return cmd;
8932 }
8933
8934 struct cmd*
8935 parseline(char **ps, char *es)
8936 {
8937     struct cmd *cmd;
8938
8939     cmd = parsepipe(ps, es);
8940     while(peek(ps, es, "&")){
8941         gettoken(ps, es, 0, 0);
8942         cmd = backcmd(cmd);
8943     }
8944     if(peek(ps, es, ";")){
8945         gettoken(ps, es, 0, 0);
8946         cmd = listcmd(cmd, parseline(ps, es));
8947     }
8948     return cmd;
8949 }

```

```

8950 struct cmd*
8951 parsepipe(char **ps, char *es)
8952 {
8953     struct cmd *cmd;
8954
8955     cmd = parseexec(ps, es);
8956     if(peek(ps, es, "|")){
8957         gettoken(ps, es, 0, 0);
8958         cmd = pipecmd(cmd, parsepipe(ps, es));
8959     }
8960     return cmd;
8961 }
8962
8963 struct cmd*
8964 parseredirs(struct cmd *cmd, char **ps, char *es)
8965 {
8966     int tok;
8967     char *q, *eq;
8968
8969     while(peek(ps, es, "<>")){
8970         tok = gettoken(ps, es, 0, 0);
8971         if(gettoken(ps, es, &q, &eq) != 'a')
8972             panic("missing file for redirection");
8973         switch(tok){
8974             case '<':
8975                 cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
8976                 break;
8977             case '>':
8978                 cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8979                 break;
8980             case '+': // >>
8981                 cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8982                 break;
8983         }
8984     }
8985     return cmd;
8986 }
8987
8988
8989
8990
8991
8992
8993
8994
8995
8996
8997
8998
8999

```

```

9000 struct cmd*
9001 parseblock(char **ps, char *es)
9002 {
9003     struct cmd *cmd;
9004
9005     if(!peek(ps, es, "("))
9006         panic("parseblock");
9007     gettoken(ps, es, 0, 0);
9008     cmd = parseline(ps, es);
9009     if(!peek(ps, es, ")"))
9010         panic("syntax - missing )");
9011     gettoken(ps, es, 0, 0);
9012     cmd = parseredirs(cmd, ps, es);
9013     return cmd;
9014 }
9015
9016 struct cmd*
9017 parseexec(char **ps, char *es)
9018 {
9019     char *q, *eq;
9020     int tok, argc;
9021     struct execcmd *cmd;
9022     struct cmd *ret;
9023
9024     if(peek(ps, es, "("))
9025         return parseblock(ps, es);
9026
9027     ret = execcmd();
9028     cmd = (struct execcmd*)ret;
9029
9030     argc = 0;
9031     ret = parseredirs(ret, ps, es);
9032     while(!peek(ps, es, "|)&")){
9033         if((tok=gettoken(ps, es, &q, &eq)) == 0)
9034             break;
9035         if(tok != 'a')
9036             panic("syntax");
9037         cmd->argv[argc] = q;
9038         cmd->eargv[argc] = eq;
9039         argc++;
9040         if(argc >= MAXARGS)
9041             panic("too many args");
9042         ret = parseredirs(ret, ps, es);
9043     }
9044     cmd->argv[argc] = 0;
9045     cmd->eargv[argc] = 0;
9046     return ret;
9047 }
9048
9049

```

```

9050 // NUL-terminate all the counted strings.
9051 struct cmd*
9052 nulterminate(struct cmd *cmd)
9053 {
9054     int i;
9055     struct backcmd *bcmd;
9056     struct execcmd *ecmd;
9057     struct listcmd *lcmd;
9058     struct pipecmd *pcmd;
9059     struct redircmd *rcmd;
9060
9061     if(cmd == 0)
9062         return 0;
9063
9064     switch(cmd->type){
9065     case EXEC:
9066         ecmd = (struct execcmd*)cmd;
9067         for(i=0; ecmd->argv[i]; i++)
9068             *ecmd->eargv[i] = 0;
9069         break;
9070
9071     case REDIR:
9072         rcmd = (struct redircmd*)cmd;
9073         nulterminate(rcmd->cmd);
9074         *rcmd->efile = 0;
9075         break;
9076
9077     case PIPE:
9078         pcmd = (struct pipecmd*)cmd;
9079         nulterminate(pcmd->left);
9080         nulterminate(pcmd->right);
9081         break;
9082
9083     case LIST:
9084         lcmd = (struct listcmd*)cmd;
9085         nulterminate(lcmd->left);
9086         nulterminate(lcmd->right);
9087         break;
9088
9089     case BACK:
9090         bcmd = (struct backcmd*)cmd;
9091         nulterminate(bcmd->cmd);
9092         break;
9093     }
9094     return cmd;
9095 }
9096
9097
9098
9099

```

```

9100 #include "asm.h"
9101 #include "memlayout.h"
9102 #include "mmu.h"
9103
9104 # Start the first CPU: switch to 32-bit protected mode, jump into C.
9105 # The BIOS loads this code from the first sector of the hard disk into
9106 # memory at physical address 0x7c00 and starts executing in real mode
9107 # with %cs=0 %ip=7c00.
9108
9109 .code16                                # Assemble for 16-bit mode
9110 .globl start
9111 start:
9112     cli                                # BIOS enabled interrupts; disable
9113
9114     # Zero data segment registers DS, ES, and SS.
9115     xorw    %ax,%ax                    # Set %ax to zero
9116     movw    %ax,%ds                    # -> Data Segment
9117     movw    %ax,%es                    # -> Extra Segment
9118     movw    %ax,%ss                    # -> Stack Segment
9119
9120     # Physical address line A20 is tied to zero so that the first PCs
9121     # with 2 MB would run software that assumed 1 MB. Undo that.
9122 seta20.1:
9123     inb     $0x64,%al                  # Wait for not busy
9124     testb   $0x2,%al
9125     jnz     seta20.1
9126
9127     movb    $0xd1,%al                  # 0xd1 -> port 0x64
9128     outb    %al,$0x64
9129
9130 seta20.2:
9131     inb     $0x64,%al                  # Wait for not busy
9132     testb   $0x2,%al
9133     jnz     seta20.2
9134
9135     movb    $0xdf,%al                  # 0xdf -> port 0x60
9136     outb    %al,$0x60
9137
9138     # Switch from real to protected mode. Use a bootstrap GDT that makes
9139     # virtual addresses map directly to physical addresses so that the
9140     # effective memory map doesn't change during the transition.
9141     lgdt    gdt_desc
9142     movl    %cr0,%eax
9143     orl     $CR0_PE,%eax
9144     movl    %eax,%cr0
9145
9146
9147
9148
9149

```

```

9150 # Complete the transition to 32-bit protected mode by using a long jmp
9151 # to reload %cs and %eip. The segment descriptors are set up with no
9152 # translation, so that the mapping is still the identity mapping.
9153 ljmp    $(SEG_KCODE<<3), $start32
9154
9155 .code32 # Tell assembler to generate 32-bit code now.
9156 start32:
9157 # Set up the protected-mode data segment registers
9158 movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
9159 movw    %ax, %ds                # -> DS: Data Segment
9160 movw    %ax, %es                # -> ES: Extra Segment
9161 movw    %ax, %ss                # -> SS: Stack Segment
9162 movw    $0, %ax                # Zero segments not ready for use
9163 movw    %ax, %fs                # -> FS
9164 movw    %ax, %gs                # -> GS
9165
9166 # Set up the stack pointer and call into C.
9167 movl    $start, %esp
9168 call    bootmain
9169
9170 # If bootmain returns (it shouldn't), trigger a Bochs
9171 # breakpoint if running under Bochs, then loop.
9172 movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
9173 movw    %ax, %dx
9174 outw    %ax, %dx
9175 movw    $0x8ae0, %ax            # 0x8ae0 -> port 0x8a00
9176 outw    %ax, %dx
9177 spin:
9178 jmp     spin
9179
9180 # Bootstrap GDT
9181 .p2align 2                        # force 4 byte alignment
9182 gdt:
9183     SEG_NULLASM                    # null seg
9184     SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg
9185     SEG_ASM(STA_W, 0x0, 0xffffffff) # data seg
9186
9187 gdtdesc:
9188     .word    (gdtdesc - gdt - 1)    # sizeof(gdt) - 1
9189     .long    gdt                    # address gdt
9190
9191
9192
9193
9194
9195
9196
9197
9198
9199

```

```

9200 // Boot loader.
9201 //
9202 // Part of the boot block, along with bootasm.S, which calls bootmain().
9203 // bootasm.S has put the processor into protected 32-bit mode.
9204 // bootmain() loads an ELF kernel image from the disk starting at
9205 // sector 1 and then jumps to the kernel entry routine.
9206
9207 #include "types.h"
9208 #include "elf.h"
9209 #include "x86.h"
9210 #include "memlayout.h"
9211
9212 #define SECTSIZE 512
9213
9214 void readseg(uchar*, uint, uint);
9215
9216 void
9217 bootmain(void)
9218 {
9219     struct elfhdr *elf;
9220     struct proghdr *ph, *eph;
9221     void (*entry)(void);
9222     uchar* pa;
9223
9224     elf = (struct elfhdr*)0x10000; // scratch space
9225
9226     // Read 1st page off disk
9227     readseg((uchar*)elf, 4096, 0);
9228
9229     // Is this an ELF executable?
9230     if(elf->magic != ELF_MAGIC)
9231         return; // let bootasm.S handle error
9232
9233     // Load each program segment (ignores ph flags).
9234     ph = (struct proghdr*)((uchar*)elf + elf->phoff);
9235     eph = ph + elf->phnum;
9236     for(; ph < eph; ph++){
9237         pa = (uchar*)ph->paddr;
9238         readseg(pa, ph->filesz, ph->off);
9239         if(ph->memsz > ph->filesz)
9240             stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
9241     }
9242
9243     // Call the entry point from the ELF header.
9244     // Does not return!
9245     entry = (void(*) (void))(elf->entry);
9246     entry();
9247 }
9248
9249

```

```

9250 void
9251 waitdisk(void)
9252 {
9253     // Wait for disk ready.
9254     while((inb(0x1F7) & 0xC0) != 0x40)
9255         ;
9256 }
9257
9258 // Read a single sector at offset into dst.
9259 void
9260 readsect(void *dst, uint offset)
9261 {
9262     // Issue command.
9263     waitdisk();
9264     outb(0x1F2, 1);    // count = 1
9265     outb(0x1F3, offset);
9266     outb(0x1F4, offset >> 8);
9267     outb(0x1F5, offset >> 16);
9268     outb(0x1F6, (offset >> 24) | 0xE0);
9269     outb(0x1F7, 0x20); // cmd 0x20 - read sectors
9270
9271     // Read data.
9272     waitdisk();
9273     insl(0x1F0, dst, SECTSIZE/4);
9274 }
9275
9276 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
9277 // Might copy more than asked.
9278 void
9279 readseg(uchar* pa, uint count, uint offset)
9280 {
9281     uchar* epa;
9282
9283     epa = pa + count;
9284
9285     // Round down to sector boundary.
9286     pa -= offset % SECTSIZE;
9287
9288     // Translate from bytes to sectors; kernel starts at sector 1.
9289     offset = (offset / SECTSIZE) + 1;
9290
9291     // If this is too slow, we could read lots of sectors at a time.
9292     // We'd write more to memory than asked, but it doesn't matter --
9293     // we load in increasing order.
9294     for(; pa < epa; pa += SECTSIZE, offset++)
9295         readsect(pa, offset);
9296 }
9297
9298
9299

```

```

9300 /* Simple linker script for the JOS kernel.
9301     See the GNU ld 'info' manual ("info ld") to learn the syntax. */
9302
9303 OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
9304 OUTPUT_ARCH(i386)
9305 ENTRY(_start)
9306
9307 SECTIONS
9308 {
9309     /* Link the kernel at this address: "." means the current address */
9310     /* Must be equal to KERNLINK */
9311     . = 0x80100000;
9312
9313     .text : AT(0x100000) {
9314         *(.text .stub .text.* .gnu.linkonce.t.*)
9315     }
9316
9317     PROVIDE(etext = .);    /* Define the 'etext' symbol to this value */
9318
9319     .rodata : {
9320         *(.rodata .rodata.* .gnu.linkonce.r.*)
9321     }
9322
9323     /* Include debugging information in kernel memory */
9324     .stab : {
9325         PROVIDE(__STAB_BEGIN__ = .);
9326         *(.stab);
9327         PROVIDE(__STAB_END__ = .);
9328     }
9329
9330     .stabstr : {
9331         PROVIDE(__STABSTR_BEGIN__ = .);
9332         *(.stabstr);
9333         PROVIDE(__STABSTR_END__ = .);
9334     }
9335
9336     /* Adjust the address for the data segment to the next page */
9337     . = ALIGN(0x1000);
9338
9339     /* Conventionally, Unix linkers provide pseudo-symbols
9340     * etext, edata, and end, at the end of the text, data, and bss.
9341     * For the kernel mapping, we need the address at the beginning
9342     * of the data section, but that's not one of the conventional
9343     * symbols, because the convention started before there was a
9344     * read-only rodata section between text and data. */
9345     PROVIDE(data = .);
9346
9347
9348
9349

```

```
9350  /* The data segment */
9351  .data : {
9352      *(.data)
9353  }
9354
9355  PROVIDE(edata = .);
9356
9357  .bss : {
9358      *(.bss)
9359  }
9360
9361  PROVIDE(end = .);
9362
9363  /DISCARD/ : {
9364      *(.eh_frame .note.GNU-stack)
9365  }
9366 }
9367
9368
9369
9370
9371
9372
9373
9374
9375
9376
9377
9378
9379
9380
9381
9382
9383
9384
9385
9386
9387
9388
9389
9390
9391
9392
9393
9394
9395
9396
9397
9398
9399
```