

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

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14, 2000)). See also <http://pdos.csail.mit.edu/6.828/2016/xv6.html>, which provides pointers to on-line resources for v6.

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

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

We are also grateful for the bug reports and patches contributed by Silas Boyd-Wickizer, Mike CAT, Nelson Elhage, Nathaniel Filardo, Peter Froehlich, Yakir Goaran, Shivam Handa, Bryan Henry, Jim Huang, Anders Kaseorg, kehao95, Eddie Kohler, Imbar Marinescu, Yandong Mao, Hitoshi Mitake, Carmi Merimovich, Joel Nider, Greg Price, Ayan Shafqat, Eldar Sehayek, Yongming Shen, Cam Tenny, Rafael Ubal, Warren Toomey, Stephen Tu, Pablo Ventura, Xi Wang, Keiichi Watanabe, Nicolas Wolovick, Jindong Zhang, and Zou Chang Wei.

The code in the files that constitute xv6 is
 Copyright 2006-2016 Frans Kaashoek, Robert Morris, and Russ Cox.

ERROR REPORTS

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

BUILDING AND RUNNING XV6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run "make". On non-x86 or non-ELF machines (like OS X, even on x86), you will need to install a cross-compiler gcc suite capable of producing x86 ELF binaries. See <http://pdos.csail.mit.edu/6.828/2016/tools.html>. Then run "make TOOLPREFIX=i386-jos-elf-".

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

To create a typeset version of the code, run "make xv6.pdf". This requires the "mpage" utility. See <http://www.mesa.n1/pub/mpage/>.

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

# basic headers	# system calls	# string operations
01 types.h	31 traps.h	66 string.c
01 param.h	32 vectors.pl	
02 memlayout.h	32 trapasm.S	# low-level hardware
02 defs.h	33 trap.c	67 mp.h
04 x86.h	34 syscall.h	69 mp.c
06 asm.h	35 syscall.c	71 lapic.c
07 mmu.h	36 sysproc.c	74 ioapic.c
10 elf.h		75 picirq.c
	# file system	77 kbd.h
# entering xv6	37 buf.h	78 kbd.c
11 entry.S	38 fcntl.h	79 console.c
12 entryother.S	38 stat.h	82 timer.c
13 main.c	39 fs.h	83 uart.c
	40 file.h	
# locks	41 ide.c	# user-level
15 spinlock.h	43 bio.c	84 initcode.S
15 spinlock.c	45 log.c	84 usys.S
	47 fs.c	85 init.c
# processes	56 file.c	85 sh.c
17 vm.c	58 sysfile.c	
23 proc.h	63 exec.c	# bootloader
24 proc.c		91 bootasm.S
29 swtch.S	# pipes	92 bootmain.c
30 kalloc.c	64 pipe.c	

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

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

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

```

acquire 1574
  0376 1574 1578 2507 2561
  2625 2658 2717 2779 2824
  2839 2866 2879 3076 3093
  3366 3722 3742 4210 4265
  4370 4431 4630 4657 4674
  4731 5008 5041 5061 5090
  5110 5120 5629 5654 5668
  6513 6534 6555 7960 8131
  8178 8214
allocproc 2456
  2456 2509 2564
allocuvn 1953
  0419 1953 1967 1973 2541
  6348 6362
alltraps 3254
  3209 3217 3230 3235 3253
  3254
ALT 7710
  7710 7738 7740
argfd 5819
  5819 5856 5871 5883 5894
  5906
argint 3545
  0394 3545 3558 3574 3683
  3706 3720 5824 5871 5883
  6108 6175 6176 6231
argptr 3554
  0395 3554 5871 5883 5906
  6257
argstr 3571
  0396 3571 5918 6008 6108
  6157 6174 6207 6231
__attribute__ 1411
  0271 0364 1309 1411
BACK 8561
  8561 8674 8820 9089
backcmd 8596 8814
  8596 8609 8675 8814 8816
  8942 9055 9090
BACKSPACE 8050
  8050 8067 8109 8142 8148
ballocc 4804
  4804 4824 5167 5175 5179
BBLOCK 3960
  3960 4811 4835
B_BUSY 3759
  3759 4258 4376 4377 4390
  4393 4417 4428 4440
B_DIRTY 3761
  3761 4195 4219 4224 4260
  4278 4390 4419 4739
begin_op 4628
  0335 2620 4628 5683 5774
  5921 6011 6111 6156 6173
  6206 6320
bfree 4829
  4829 5214 5224 5227
bget 4366
  4366 4398 4406
binit 4339
  0262 1331 4339
bmap 5160
  4923 5160 5186 5269 5319
bootmain 9217
  9168 9217
BPB 3957
  3957 3960 4810 4812 4836
bread 4402
  0263 4402 4577 4578 4590
  4606 4688 4689 4784 4795
  4811 4835 4960 4981 5068
  5176 5220 5269 5319
brelse 4426
  0264 4426 4429 4581 4582
  4597 4614 4692 4693 4786
  4798 4817 4822 4842 4966
  4969 4990 5076 5182 5226
  5272 5323
BSIZE 3905
  3757 3905 3924 3951 3957
  4181 4197 4220 4558 4579
  4690 4796 5269 5270 5271
  5315 5319 5320 5321
buf 3750
  0250 0263 0264 0265 0307
  0334 2120 2123 2132 2134
  3750 3754 3755 3756 4112
  4130 4133 4175 4207 4254
  4256 4259 4327 4331 4335
  4341 4353 4365 4368 4401
  4404 4415 4426 4505 4577
  4578 4590 4591 4597 4606
  4607 4613 4614 4688 4689
  4722 4769 4782 4793 4807
  4831 4956 4978 5055 5163
  5209 5255 5305 7929 7940
  7944 7947 8118 8140 8154
  8188 8209 8216 8684 8687
  8688 8689 8703 8715 8716

```

```

  8718 8719 8720 8724
B_VALID 3760
  3760 4223 4260 4278 4407
bwrite 4415
  0265 4415 4418 4580 4613
  4691
bzero 4791
  4791 4818
C 7731 8124
  7731 7779 7804 7805 7806
  7807 7808 7810 8124 8134
  8138 8145 8156 8189
CAPSLOCK 7712
  7712 7745 7886
cgaputc 8055
  8055 8113
clearpteu 2034
  0428 2034 2040 6364
cli 0557
  0557 0559 1224 1660 8010
  8104 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 7300
  7300 7314 7315 7363
CMOS_RETURN 7301
  7301 7366
CMOS_STATA 7350
  7350 7392
CMOS_STATB 7351
  7351 7385
CMOS_UIP 7352
  7352 7392
COM1 8313
  8313 8323 8326 8327 8328
  8329 8330 8331 8334 8340
  8341 8357 8359 8367 8369
commit 4701
  4553 4673 4701
CONSOLE 4037
  4037 8228 8229
consoleinit 8224
  0268 1327 8224
consoleintr 8127
  0270 7898 8127 8375
consoleread 8171
  8171 8229
consolewrite 8209
  8209 8228
consputc 8101
  7916 7947 7968 7986 7989
  7993 7994 8101 8142 8148
  8155 8216
context 2340
  0251 0373 2303 2340 2361
  2486 2487 2488 2489 2728
  2771 2928
CONV 7402
  7402 7403 7404 7405 7406
  7407 7408 7409
copyout 2118
  0427 2118 6372 6383
copyuvn 2053
  0424 2053 2064 2066 2570
cprintf 7952
  0269 1324 1364 1967 1973
  2926 2930 2932 3390 3403
  3408 3633 4922 7263 7512
  7952 8012 8013 8014 8017
cpu 2301
  0310 1364 1366 1378 1506
  1566 1590 1608 1645 1661
  1662 1663 1671 1673 1717
  1730 1736 1876 1877 1878
  1879 1882 2301 2311 2315
  2326 2728 2764 2770 2771
  2772 3390 3403 3408 6913
  7263 8012

```

```

cpunum 7251
    0325 1324 1364 1388 1723
    3365 3391 3404 3410 7251
    7523 7532
CRO_PE 0727
    0727 1237 1270 9143
CRO_PG 0737
    0737 1154 1270
CRO_WP 0733
    0733 1154 1270
CR4_PSE 0739
    0739 1147 1263
create 6057
    6057 6077 6090 6094 6114
    6157 6177
CRTPORT 8051
    8051 8060 8061 8062 8063
    8081 8082 8083 8084
CTL 7709
    7709 7735 7739 7885
DAY 7357
    7357 7374
deallocvm 1987
    0420 1968 1974 1987 2021
    2544
DEVSPACE 0204
    0204 1832 1845
devsw 4030
    4030 4035 5258 5260 5308
    5310 5611 8228 8229
dinode 3928
    3928 3951 4957 4961 4979
    4982 5056 5069
dirent 3965
    3965 5364 5405 5966 6004
dirlink 5402
    0287 5371 5402 5417 5425
    5941 6089 6093 6094
dirlookup 5361
    0288 5361 5367 5409 5525
    6023 6067
DIRSIZ 3963
    3963 3967 5355 5422 5478
    5479 5542 5915 6005 6061
DPL_USER 0829
    0829 1726 1727 2516 2517
    3323 3418 3427
EOESC 7716
    7716 7870 7874 7875 7877
    7880
elfhdr 1005
    1005 6315 9219 9224
ELF_MAGIC 1002
    1002 6331 9230
ELF_PROG_LOAD 1036
    1036 6342
end_op 4653
    0336 2622 4653 5685 5779
    5923 5930 5948 5957 6013
    6047 6052 6116 6121 6127
    6136 6140 6158 6162 6178
    6182 6208 6214 6219 6322
    6356 6407
entry 1144
    1011 1140 1143 1144 3202
    3203 6396 6771 9221 9245
    9246
EOI 7117
    7117 7234 7283
ERROR 7138
    7138 7227
ESR 7120
    7120 7230 7231
exec 6310
    0274 6247 6310 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 2604
    0358 2604 2642 3355 3359
    3419 3428 3668 8417 8420
    8461 8526 8531 8616 8625
    8635 8680 8727 8734
EXTMEM 0202
    0202 0208 1829
fdalloc 5838
    5838 5858 6132 6262
fetchint 3517
    0397 3517 3547 6238
fetchstr 3529
    0398 3529 3576 6244
file 4000
    0252 0277 0278 0279 0281
    0282 0283 0351 2364 4000
    4770 5608 5614 5624 5627
    5630 5651 5652 5664 5666
    5702 5715 5752 5813 5819

```

```

    5822 5838 5853 5867 5879
    5892 5903 6105 6254 6456
    6471 7910 8308 8578 8633
    8634 8764 8772 8972
filealloc 5625
    0277 5625 6132 6477
fileclose 5664
    0278 2615 5664 5670 5897
    6134 6265 6266 6504 6506
filedup 5652
    0279 2586 5652 5656 5860
fileinit 5618
    0280 1332 5618
fileread 5715
    0281 5715 5730 5873
filestat 5702
    0282 5702 5908
filewrite 5752
    0283 5752 5784 5789 5885
FL_IF 0710
    0710 1662 1669 2520 2768
    7260
fork 2556
    0359 2556 3662 8460 8523
    8525 8742 8744
fork1 8738
    8600 8642 8654 8661 8676
    8723 8738
forkret 2788
    2417 2489 2788
freerange 3051
    3011 3034 3040 3051
freevm 2015
    0421 2015 2020 2078 2671
    6399 6404
FSSIZE 0162
    0162 4179
gatedesc 0951
    0523 0526 0951 3311
getcallerpcs 1625
    0377 1591 1625 2928 8015
getcmd 8684
    8684 8715
gettoken 8856
    8856 8941 8945 8957 8970
    8971 9007 9011 9033
growproc 2535
    0360 2535 3709
havedisk1 4132
    4132 4164 4262
holding 1643
    0378 1577 1604 1643 2762
HOURS 7356
    7356 7373
ialloc 4953
    0289 4953 4971 6076 6077
IBLOCK 3954
    3954 4960 4981 5068
I_BUSY 4025
    4025 5062 5064 5087 5091
    5113 5115
ICRHI 7131
    7131 7237 7322 7334
ICRLO 7121
    7121 7238 7239 7323 7325
    7335
ID 7114
    7114 7154 7270
IDE_BSY 4115
    4115 4141
IDE_CMD_RDMUL 4122
    4122 4183
IDE_CMD_READ 4120
    4120 4183
IDE_CMD_WRITE 4121
    4121 4184
IDE_CMD_WRMUL 4123
    4123 4184
IDE_DF 4117
    4117 4143
IDE_DRDY 4116
    4116 4141
IDE_ERR 4118
    4118 4143
ideinit 4151
    0305 1333 4151
ideintr 4205
    0306 3374 4205
idelock 4129
    4129 4155 4210 4212 4231
    4265 4279 4282
iderw 4254
    0307 4254 4259 4261 4263
    4408 4420
idestart 4175
    4133 4175 4178 4186 4229
    4275
idewait 4137
    4137 4158 4188 4219
idtinit 3329

```

```

0404 1365 3329
idup 5039
0290 2587 5039 5512
iget 5004
4928 4967 5004 5024 5379
5510
iinit 4918
0291 2799 4918
ilock 5053
0292 5053 5059 5079 5515
5705 5724 5775 5927 5940
5953 6017 6025 6065 6069
6079 6124 6211 6325 8183
8203 8218
inb 0453
0453 4141 4163 7054 7366
7864 7867 8061 8063 8334
8340 8341 8357 8367 8369
9123 9131 9254
initlock 1562
0379 1562 2425 3032 3325
4155 4343 4562 4920 5620
6485 8226
initlog 4556
0333 2800 4556 4559
initvm 1903
0422 1903 1908 2513
inode 4012
0253 0287 0288 0289 0290
0292 0293 0294 0295 0296
0298 0299 0300 0301 0302
0423 1918 2365 4006 4012
4031 4032 4773 4914 4928
4952 4976 5003 5006 5012
5038 5039 5053 5085 5108
5130 5160 5206 5237 5252
5302 5360 5361 5402 5406
5504 5507 5539 5550 5916
5963 6003 6056 6060 6106
6154 6169 6204 6316 8171
8209
INPUT_BUF 8116
8116 8118 8140 8152 8154
8156 8188
insl 0462
0462 0464 4220 9273
install_trans 4572
4572 4621 4706
INT_DISABLED 7469
7469 7517
ioapic 7477
7007 7024 7025 7474 7477
7486 7487 7493 7494 7508
IOAPIC 7458
7458 7508
ioapicenable 7523
0310 4157 7523 8233 8343
ioapicid 6916
0311 6916 7025 7042 7511
7512
ioapicinit 7501
0312 1326 7501 7512
ioapicread 7484
7484 7509 7510
ioapicwrite 7491
7491 7517 7518 7531 7532
IO_PIC1 7557
7557 7570 7585 7594 7597
7602 7612 7626 7627
IO_PIC2 7558
7558 7571 7586 7615 7616
7617 7620 7629 7630
IO_TIMER1 8259
8259 8268 8278 8279
IPB 3951
3951 3954 4961 4982 5069
iput 5108
0293 2621 5108 5114 5133
5410 5533 5684 5946 6218
IRQ_COM1 3183
3183 3384 8342 8343
IRQ_ERROR 3185
3185 7227
IRQ_IDE 3184
3184 3373 3377 4156 4157
IRQ_KBD 3182
3182 3380 8232 8233
IRQ_SLAVE 7560
7560 7564 7602 7617
IRQ_SPURIOUS 3186
3186 3389 7207
IRQ_TIMER 3181
3181 3364 3423 7214 8280
isdirempty 5963
5963 5970 6029
ismp 6914
0339 1334 6914 7011 7034
7038 7505 7525
itrunc 5206
4773 5117 5206

```

```

iunlock 5085
0294 5085 5088 5132 5522
5707 5727 5778 5936 6139
6217 8176 8213
iunlockput 5130
0295 5130 5517 5526 5529
5929 5942 5945 5956 6030
6041 6045 6051 6068 6072
6096 6126 6135 6161 6181
6213 6355 6406
iupdate 4976
0296 4976 5119 5232 5328
5935 5955 6039 6044 6083
6087
I_INVALID 4026
4026 5067 5077 5111
kalloc 3088
0315 1394 1763 1842 1909
1965 2069 2471 3088 6479
KBDATAP 7704
7704 7867
kbdgetc 7856
7856 7898
kbdintr 7896
0321 3381 7896
KBS_DIB 7703
7703 7865
KBSTATP 7702
7702 7864
KERNBASE 0207
0207 0208 0210 0211 0213
0214 1416 1632 1829 1958
2021
KERNLINK 0208
0208 1830
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 3065
0316 1975 2003 2005 2025
2028 2571 2669 3056 3065
3070 6502 6523
kill 2875
0361 2875 3409 3685 8467
kinit1 3030
0317 1319 3030
kinit2 3038
0318 1337 3038
KSTACKSIZE 0151
0151 1158 1167 1395 1879
2475
kvmalloc 1857
0416 1320 1857
lapiceoi 7280
0327 3371 3375 3382 3386
3392 7280
lapicinit 7201
0328 1322 1356 7201
lapicstartap 7306
0329 1399 7306
lapicw 7151
7151 7207 7213 7214 7215
7218 7219 7224 7227 7230
7231 7234 7237 7238 7243
7283 7322 7323 7325 7334
7335
lcr3 0590
0590 1868 1886
lgdt 0512
0512 0520 1235 1732 9141
lidt 0526
0526 0534 3331
LINT0 7136
7136 7218
LINT1 7137
7137 7219
LIST 8560
8560 8640 8807 9083
listcmd 8590 8801
8590 8611 8641 8801 8803
8946 9057 9084
loadgs 0551
0551 1733

```

```

loaduvm 1918
  0423 1918 1924 1927 6352
log 4537 4550
  4537 4550 4562 4564 4565
  4566 4576 4577 4578 4590
  4593 4594 4595 4606 4609
  4610 4611 4622 4630 4632
  4633 4634 4636 4638 4639
  4657 4658 4659 4660 4661
  4663 4666 4668 4674 4675
  4676 4677 4687 4688 4689
  4703 4707 4726 4728 4731
  4732 4733 4736 4737 4738
  4740
logheader 4532
  4532 4544 4558 4559 4591
  4607
LOGSIZE 0160
  0160 4534 4634 4726 5767
log_write 4722
  0334 4722 4729 4797 4816
  4841 4965 4989 5180 5322
ltr 0538
  0538 0540 1883
mappages 1779
  1779 1848 1911 1972 2072
MAXARG 0158
  0158 6227 6314 6369
MAXARGS 8563
  8563 8571 8572 9040
MAXFILE 3925
  3925 5315
MAXOPBLOCKS 0159
  0159 0160 0161 4634
memcmp 6615
  0385 6615 6938 6988 7395
memmove 6631
  0386 1385 1912 2071 2132
  4579 4690 4785 4988 5075
  5271 5321 5479 5481 6631
  6654 8076
memset 6604
  0387 1766 1844 1910 1971
  2488 2515 3073 4796 4963
  6034 6234 6604 8078 8687
  8758 8769 8785 8806 8819
microdelay 7289
  0330 7289 7324 7326 7336
  7364 8358
min 4772

```

```

  4772 5270 5320
MINS 7355
  7355 7372
MONTH 7358
  7358 7375
mp 6752
  6752 6908 6930 6937 6938
  6939 6955 6960 6964 6965
  6968 6969 6980 6983 6985
  6987 6994 7004 7009 7050
MPBUS 6802
  6802 7028
mpconf 6763
  6763 6979 6982 6987 7005
mpconfig 6980
  6980 7009
mpenter 1352
  1352 1396
mpinit 7001
  0340 1321 7001
mpioapic 6789
  6789 7007 7024 7026
MPIOPIC 6803
  6803 7023
MPIOINTR 6804
  6804 7029
MPLINTR 6805
  6805 7030
mpmain 1362
  1309 1339 1357 1362
mpproc 6778
  6778 7006 7016 7021
MPPROC 6801
  6801 7015
mpsearch 6956
  6956 6985
mpsearch1 6931
  6931 6964 6968 6971
multiboot_header 1129
  1128 1129
namecmp 5353
  0297 5353 5374 6020
namei 5540
  0298 2525 5540 5922 6120
  6207 6321
nameiparent 5551
  0299 5505 5520 5532 5551
  5938 6012 6063
namex 5505
  5505 5543 5553

```

```

NBUF 0161
  0161 4331 4353
ncpu 6915
  1324 1387 2316 4157 6915
  7017 7018 7019 7040 7271
NCPU 0152
  0152 2315 6913 7017
NDEV 0156
  0156 5258 5308 5611
NDIRECT 3923
  3923 3925 3934 4023 5165
  5170 5174 5175 5212 5219
  5220 5227 5228
NELEM 0431
  0431 1847 2922 3630 6236
nextpid 2416
  2416 2468
NFILE 0154
  0154 5614 5630
NINDIRECT 3924
  3924 3925 5172 5222
NINODE 0155
  0155 4914 5012
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 2364 2584 2613 5826
  5842
NPENTRIES 0871
  0871 1412 2022
NPROC 0150
  0150 2411 2461 2631 2662
  2718 2857 2880 2919
NPTENTRIES 0872
  0872 1999
NSEGS 0751
  0751 2305
nulterminate 9052
  8915 8930 9052 9073 9079
  9080 9085 9086 9091
NUMLOCK 7713
  7713 7746
O_CREATE 3803
  3803 6113 8978 8981
O_RDONLY 3800
  3800 6125 8975

```

```

O_RDWR 3802
  3802 6146 8514 8516 8707
outb 0471
  0471 4161 4170 4189 4190
  4191 4192 4193 4194 4196
  4199 7053 7054 7314 7315
  7363 7570 7571 7585 7586
  7594 7597 7602 7612 7615
  7616 7617 7620 7626 7627
  7629 7630 8060 8062 8081
  8082 8083 8084 8277 8278
  8279 8323 8326 8327 8328
  8329 8330 8331 8359 9128
  9136 9264 9265 9266 9267
  9268 9269
outsl 0483
  0483 0485 4197
outw 0477
  0477 1280 1282 9174 9176
O_WRONLY 3801
  3801 6145 6146 8978 8981
P2V 0211
  0211 1319 1337 1384 1761
  1845 1933 2004 2024 2071
  2111 6935 6962 6987 7316
  8052
panic 8005 8731
  0271 1578 1605 1670 1672
  1790 1846 1885 1908 1924
  1927 2003 2020 2040 2064
  2066 2512 2610 2642 2763
  2765 2767 2769 2812 2815
  3070 3405 4178 4180 4186
  4259 4261 4263 4398 4418
  4429 4559 4660 4727 4729
  4824 4839 4971 5024 5059
  5079 5088 5114 5186 5367
  5371 5417 5425 5656 5670
  5730 5784 5789 5970 6028
  6036 6077 6090 6094 7275
  7963 8005 8012 8073 8601
  8620 8653 8731 8744 8928
  8972 9006 9010 9036 9041
panicked 7918
  7918 8018 8103
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 7135
7135 7224
pde_t 0103
0103 0417 0418 0419 0420
0421 0422 0423 0424 0427
0428 1310 1370 1412 1710
1754 1756 1779 1836 1839
1842 1903 1918 1953 1987
2015 2034 2052 2053 2055
2102 2118 2355 6318
PDX 0862
0862 1759
PDXSHIFT 0877
0862 0868 0877 1416
peek 8901
8901 8925 8940 8944 8956
8969 9005 9009 9024 9032
PGROUNDDOWN 0880
0880 1784 1785 2125
PGROUNDUP 0879
0879 1963 1995 3054 6361
PGSIZE 0873
0873 0879 0880 1411 1766
1794 1795 1844 1907 1910
1911 1923 1925 1929 1932
1964 1971 1972 1996 1999
2062 2071 2072 2129 2135
2514 2521 3055 3069 3073
6350 6362 6364
PHYSTOP 0203
0203 1337 1831 1845 1846
3069
picenable 7575
0343 4156 7575 8232 8280
8342
picinit 7582
0344 1325 7582
picsetmask 7567
7567 7577 7633
pinit 2423
0362 1329 2423
pipe 6461
0254 0352 0353 0354 4005

```

```

5681 5722 5759 6461 6473
6479 6485 6489 6493 6511
6530 6551 8463 8652 8653
PIPE 8559
8559 8650 8786 9077
pipealloc 6471
0351 6259 6471
pipeclose 6511
0352 5681 6511
pipecmd 8584 8780
8584 8612 8651 8780 8782
8958 9058 9078
piperead 6551
0353 5722 6551
PIPESIZE 6459
6459 6463 6536 6544 6566
pipewrite 6530
0354 5759 6530
popcli 1667
0382 1620 1667 1670 1672
1887
printint 7926
7926 7976 7980
proc 2353
0255 0425 1305 1558 1706
1737 1873 1879 2312 2327
2353 2359 2406 2411 2414
2455 2458 2461 2504 2539
2541 2544 2547 2548 2559
2570 2577 2578 2579 2585
2586 2587 2589 2606 2609
2614 2615 2616 2621 2623
2628 2631 2632 2640 2655
2662 2663 2683 2689 2710
2718 2725 2733 2766 2771
2780 2811 2829 2830 2834
2855 2857 2877 2880 2915
2919 3305 3354 3356 3358
3401 3409 3410 3412 3418
3423 3427 3505 3519 3533
3536 3547 3560 3629 3631
3634 3635 3657 3691 3708
3725 4107 4766 5512 5811
5826 5843 5844 5896 6218
6220 6264 6304 6390 6393
6394 6395 6396 6397 6398
6454 6537 6557 6911 7006
7016 7018 7111 7913 8181
8310
procdump 2904

```

```

0363 2904 8166
proghdr 1024
1024 6317 9220 9234
PTE_ADDR 0894
0894 1761 1928 2001 2024
2067 2111
PTE_FLAGS 0895
0895 2068
PTE_P 0883
0883 1414 1416 1760 1770
1789 1791 2000 2023 2065
2107
PTE_PS 0890
0890 1414 1416
pte_t 0898
0898 1753 1757 1761 1763
1782 1921 1989 2036 2056
2104
PTE_U 0885
0885 1770 1911 1972 2041
2109
PTE_W 0884
0884 1414 1416 1770 1829
1831 1832 1911 1972
PTX 0865
0865 1772
PTXSHIFT 0876
0865 0868 0876
pushcli 1655
0381 1576 1655 1875
rcr2 0582
0582 3404 3411
readeflags 0544
0544 1659 1669 2768 7260
read_head 4588
4588 4620
readi 5252
0300 1933 5252 5370 5416
5725 5969 5970 6329 6340
readsb 4780
0286 4563 4780 4834 4921
readsect 9260
9260 9295
readseg 9279
9214 9227 9238 9279
recover_from_log 4618
4552 4567 4618
REDIR 8558
8558 8630 8770 9071
redircmd 8575 8764

```

```

8575 8613 8631 8764 8766
8975 8978 8981 9059 9072
REG_ID 7460
7460 7510
REG_TABLE 7462
7462 7517 7518 7531 7532
REG_VER 7461
7461 7509
release 1602
0380 1602 1605 2529 2565
2574 2595 2677 2684 2735
2782 2792 2825 2838 2868
2886 2890 3081 3098 3369
3726 3731 3744 4212 4231
4282 4378 4394 4443 4639
4668 4677 4740 5015 5031
5043 5065 5093 5116 5125
5633 5637 5658 5672 5678
6522 6525 6538 6547 6558
6569 8001 8164 8182 8202
8217
ROOTDEV 0157
0157 2799 2800 5510
ROOTINO 3904
3904 5510
run 3014
2911 3014 3015 3021 3067
3077 3090
runcmd 8606
8606 8620 8637 8643 8645
8659 8666 8677 8724
RUNNING 2350
2350 2727 2766 2911 3423
safestrcpy 6682
0388 2524 2589 6390 6682
sb 4776
0286 3954 3960 4561 4563
4564 4565 4776 4780 4785
4810 4811 4812 4834 4835
4921 4922 4923 4924 4925
4959 4960 4981 5068 7383
7385 7387
sched 2758
0365 2641 2758 2763 2765
2767 2769 2781 2831
scheduler 2708
0364 1367 2303 2708 2728
2771
SCROLLLOCK 7714
7714 7747

```

SECS 7354	0380 0407 1501 1559 1562
7354 7371	1574 1602 1643 2407 2410
SECTOR_SIZE 4114	2809 3009 3019 3308 3313
4114 4181	4110 4129 4325 4330 4503
SECTSIZE 9212	4538 4767 4913 5609 5613
9212 9273 9286 9289 9294	6457 6462 7908 7921 8306
SEG 0819	STA_R 0669 0836
0819 1724 1725 1726 1727	0669 0836 1289 1724 1726
1730	9184
SEG16 0823	start 1223 8409 9111
0823 1876	1222 1223 1266 1274 1276
SEG_ASM 0660	4539 4564 4577 4590 4606
0660 1289 1290 9184 9185	4688 4923 8408 8409 9110
segdesc 0802	9111 9167
0509 0512 0802 0819 0823	startothers 1374
2305	1308 1336 1374
seginit 1715	stat 3854
0415 1323 1355 1715	0258 0282 0301 3854 4764
SEG_KCODE 0742	5237 5702 5809 5904 8503
0742 1243 1724 3322 3323	stati 5237
9153	0301 5237 5706
SEG_KCPU 0744	STA_W 0668 0835
0744 1730 1733 3266	0668 0835 1290 1725 1727
SEG_KDATA 0743	1730 9185
0743 1253 1725 1878 3263	STA_X 0665 0832
9158	0665 0832 1289 1724 1726
SEG_NULLASM 0654	9184
0654 1288 9183	sti 0563
SEG_TSS 0747	0563 0565 1674 2714
0747 1876 1877 1883	stosb 0492
SEG_UCODE 0745	0492 0494 6610 9240
0745 1726 2516	stosl 0501
SEG_UDATA 0746	0501 0503 6608
0746 1727 2517	strlen 6701
SETGATE 0971	0389 6371 6372 6701 8718
0971 3322 3323	8923
setupkvm 1837	strncmp 6658
0417 1837 1859 2060 2511	0390 5355 6658
6334	strncpy 6668
SHIFT 7708	0391 5422 6668
7708 7736 7737 7885	STS_IG32 0850
skipelem 5465	0850 0977
5465 5514	STS_T32A 0847
sleep 2809	0847 1876
0366 2689 2809 2812 2815	STS_TG32 0851
2909 3729 4279 4381 4633	0851 0977
4636 5063 6542 6561 8186	sum 6919
8479	6919 6921 6923 6925 6926
spinlock 1501	6938 6992
0257 0366 0376 0378 0379	superblock 3913

0259 0286 3913 4561 4776	3461 3611
4780	sys_kill 3679
SVR 7118	3587 3606 3679
7118 7207	SYS_kill 3456
switchkvm 1866	3456 3606
0426 1354 1860 1866 2729	sys_link 5913
switchvm 1873	3588 3619 5913
0425 1873 1885 2548 2726	SYS_link 3469
6398	3469 3619
swtch 2958	sys_mkdir 6151
0373 2728 2771 2957 2958	3589 3620 6151
syscall 3625	SYS_mkdir 3470
0399 3357 3507 3625	3470 3620
SYSCALL 8453 8460 8461 8462 8463 84	sys_mknod 6167
8460 8461 8462 8463 8464	3590 3617 6167
8465 8466 8467 8468 8469	SYS_mknod 3467
8470 8471 8472 8473 8474	3467 3617
8475 8476 8477 8478 8479	sys_open 6101
8480	3591 3615 6101
sys_chdir 6201	SYS_open 3465
3579 3609 6201	3465 3615
SYS_chdir 3459	sys_pipe 6251
3459 3609	3592 3604 6251
sys_close 5889	SYS_pipe 3454
3580 3621 5889	3454 3604
SYS_close 3471	sys_read 5865
3471 3621	3593 3605 5865
sys_dup 5851	SYS_read 3455
3581 3610 5851	3455 3605
SYS_dup 3460	sys_sbrk 3701
3460 3610	3594 3612 3701
sys_exec 6225	SYS_sbrk 3462
3582 3607 6225	3462 3612
SYS_exec 3457	sys_sleep 3715
3457 3607 8413	3595 3613 3715
sys_exit 3666	SYS_sleep 3463
3583 3602 3666	3463 3613
SYS_exit 3452	sys_unlink 6001
3452 3602 8418	3596 3618 6001
sys_fork 3660	SYS_unlink 3468
3584 3601 3660	3468 3618
SYS_fork 3451	sys_uptime 3738
3451 3601	3599 3614 3738
sys_fstat 5901	SYS_uptime 3464
3585 3608 5901	3464 3614
SYS_fstat 3458	sys_wait 3673
3458 3608	3597 3603 3673
sys_getpid 3689	SYS_wait 3453
3586 3611 3689	3453 3603
SYS_getpid 3461	sys_write 5877

3598 3616 5877
 SYS_write 3466
 3466 3616
 taskstate 0901
 0901 2304
 TDCR 7142
 7142 7213
 T_DEV 3852
 3852 5257 5307 6177
 T_DIR 3850
 3850 5366 5516 5928 6029
 6037 6085 6125 6157 6212
 T_FILE 3851
 3851 6070 6114
 ticks 3314
 0405 3314 3367 3368 3723
 3724 3729 3743
 tickslock 3313
 0407 3313 3325 3366 3369
 3722 3726 3729 3731 3742
 3744
 TICR 7140
 7140 7215
 TIMER 7132
 7132 7214
 TIMER_16BIT 8271
 8271 8277
 TIMER_DIV 8266
 8266 8278 8279
 TIMER_FREQ 8265
 8265 8266
 timerinit 8274
 0401 1335 8274
 TIMER_MODE 8268
 8268 8277
 TIMER_RATEGEN 8270
 8270 8277
 TIMER_SELO 8269
 8269 8277
 T_IRQ0 3179
 3179 3364 3373 3377 3380
 3384 3388 3389 3423 7207
 7214 7227 7517 7531 7597
 7616
 TPR 7116
 7116 7243
 trap 3351
 3202 3204 3272 3351 3403
 3405 3408
 trapframe 0602

0602 2360 2479 3351
 trapret 3277
 2418 2484 3276 3277
 T_SYSCALL 3176
 3176 3323 3353 8414 8419
 8457
 tvinit 3317
 0406 1330 3317
 uart 8315
 8315 8336 8355 8365
 uartgetc 8363
 8363 8375
 uartinit 8318
 0410 1328 8318
 uartintr 8373
 0411 3385 8373
 uartputc 8351
 0412 8110 8112 8347 8351
 userinit 2502
 0367 1338 2502 2512
 uva2ka 2102
 0418 2102 2126
 V2P 0210
 0210 1397 1399 1770 1830
 1831 1868 1886 1911 1972
 2072 3069
 V2P_WO 0213
 0213 1140 1150
 VER 7115
 7115 7223
 wait 2653
 0368 2653 3675 8462 8533
 8644 8670 8671 8725
 waitdisk 9251
 9251 9263 9272
 wakeup 2864
 0369 2864 3368 4225 4441
 4666 4676 5092 5122 6516
 6519 6541 6546 6568 8158
 wakeup1 2853
 2420 2628 2635 2853 2867
 walkpgdir 1754
 1754 1787 1926 1997 2038
 2063 2106
 write_head 4604
 4604 4623 4705 4708
 writei 5302
 0302 5302 5424 5776 6035
 6036
 write_log 4683

4683 4704
 xchg 0569
 0569 1366 1581
 YEAR 7359

7359 7376
 yield 2777
 0370 2777 3424


```

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

```

```

0150 #define NPROC          64 // maximum number of processes
0151 #define KSTACKSIZE 4096 // size of per-process kernel stack
0152 #define NCPU           8 // maximum number of CPUs
0153 #define NOFILE         16 // open files per process
0154 #define NFILE          100 // open files per system
0155 #define NINODE          50 // maximum number of active i-nodes
0156 #define NDEV           10 // maximum major device number
0157 #define ROOTDEV         1 // device number of file system root disk
0158 #define MAXARG          32 // max exec arguments
0159 #define MAXOPBLOCKS    10 // max # of blocks any FS op writes
0160 #define LOGSIZE         (MAXOPBLOCKS*3) // max data 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 *) (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 stat;
0259 struct superblock;
0260
0261 // bio.c
0262 void          binit(void);
0263 struct buf*   bread(uint, uint);
0264 void          brelease(struct buf*);
0265 void          bwrite(struct buf*);
0266
0267 // console.c
0268 void          consoleinit(void);
0269 void          cprintf(char*, ...);
0270 void          consoleintr(int (*)(void));
0271 void          panic(char*) __attribute__((noreturn));
0272
0273 // exec.c
0274 int           exec(char*, char**);
0275
0276 // file.c
0277 struct file*  filealloc(void);
0278 void          fileclose(struct file*);
0279 struct file*  filedup(struct file*);
0280 void          fileinit(void);
0281 int           fileread(struct file*, char*, int n);
0282 int           filestat(struct file*, struct stat*);
0283 int           filewrite(struct file*, char*, int n);
0284
0285 // fs.c
0286 void          readsb(int dev, struct superblock *sb);
0287 int           dirlink(struct inode*, char*, uint);
0288 struct inode* dirlookup(struct inode*, char*, uint*);
0289 struct inode* ialloc(uint, short);
0290 struct inode* idup(struct inode*);
0291 void          iinit(int dev);
0292 void          ilock(struct inode*);
0293 void          iput(struct inode*);
0294 void          iunlock(struct inode*);
0295 void          iunlockput(struct inode*);
0296 void          iupdate(struct inode*);
0297 int           namecmp(const char*, const char*);
0298 struct inode* namei(char*);
0299 struct inode* nameiparent(char*, char*);

```

```

0300 int      readi(struct inode*, char*, uint, uint);
0301 void      stati(struct inode*, struct stat*);
0302 int      writei(struct inode*, char*, uint, uint);
0303
0304 // ide.c
0305 void      ideinit(void);
0306 void      ideintr(void);
0307 void      iderw(struct buf*);
0308
0309 // ioapic.c
0310 void      ioapicenable(int irq, int cpu);
0311 extern uchar ioapicid;
0312 void      ioapicinit(void);
0313
0314 // kalloc.c
0315 char*      kalloc(void);
0316 void      kfree(char*);
0317 void      kinit1(void*, void*);
0318 void      kinit2(void*, void*);
0319
0320 // kbd.c
0321 void      kbdtintr(void);
0322
0323 // lapic.c
0324 void      cmostime(struct rtcdate *r);
0325 int      cpunum(void);
0326 extern volatile uint* lapic;
0327 void      lapiceoi(void);
0328 void      lapicinit(void);
0329 void      lapicstartap(uchar, uint);
0330 void      microdelay(int);
0331
0332 // log.c
0333 void      initlog(int dev);
0334 void      log_write(struct buf*);
0335 void      begin_op();
0336 void      end_op();
0337
0338 // mp.c
0339 extern int ismp;
0340 void      mpinit(void);
0341
0342 // picirq.c
0343 void      picenable(int);
0344 void      picinit(void);
0345
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 void      exit(void);
0359 int      fork(void);
0360 int      growproc(int);
0361 int      kill(int);
0362 void      pinit(void);
0363 void      procdump(void);
0364 void      scheduler(void) __attribute__((noreturn));
0365 void      sched(void);
0366 void      sleep(void*, struct spinlock*);
0367 void      userinit(void);
0368 int      wait(void);
0369 void      wakeup(void*);
0370 void      yield(void);
0371
0372 // swtch.S
0373 void      swtch(struct context**, struct context*);
0374
0375 // spinlock.c
0376 void      acquire(struct spinlock*);
0377 void      getcallerpcs(void*, uint*);
0378 int      holding(struct spinlock*);
0379 void      initlock(struct spinlock*, char*);
0380 void      release(struct spinlock*);
0381 void      pushcli(void);
0382 void      popcli(void);
0383
0384 // string.c
0385 int      memcmp(const void*, const void*, uint);
0386 void*      memmove(void*, const void*, uint);
0387 void*      memset(void*, int, uint);
0388 char*      safestrcpy(char*, const char*, int);
0389 int      strlen(const char*);
0390 int      strncmp(const char*, const char*, uint);
0391 char*      strncpy(char*, const char*, int);
0392
0393 // syscall.c
0394 int      argint(int, int*);
0395 int      argptr(int, char**, int);
0396 int      argstr(int, char**);
0397 int      fetchint(uint, int*);
0398 int      fetchstr(uint, char**);
0399 void      syscall(void);

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

0700 // This file contains definitions for the
0701 // x86 memory management unit (MMU).
0702
0703 // Eflags register
0704 #define FL_CF      0x00000001    // Carry Flag
0705 #define FL_PF      0x00000004    // Parity Flag
0706 #define FL_AF      0x00000010    // Auxiliary carry Flag
0707 #define FL_ZF      0x00000040    // Zero Flag
0708 #define FL_SF      0x00000080    // Sign Flag
0709 #define FL_TF      0x00000100    // Trap Flag
0710 #define FL_IF      0x00000200    // Interrupt Enable
0711 #define FL_DF      0x00000400    // Direction Flag
0712 #define FL_OF      0x00000800    // Overflow Flag
0713 #define FL_IOPL_MASK 0x00003000    // I/O Privilege Level bitmask
0714 #define FL_IOPL_0    0x00000000    // IOPL == 0
0715 #define FL_IOPL_1    0x00001000    // IOPL == 1
0716 #define FL_IOPL_2    0x00002000    // IOPL == 2
0717 #define FL_IOPL_3    0x00003000    // IOPL == 3
0718 #define FL_NT      0x00004000    // Nested Task
0719 #define FL_RF      0x00010000    // Resume Flag
0720 #define FL_VM      0x00020000    // Virtual 8086 mode
0721 #define FL_AC      0x00040000    // Alignment Check
0722 #define FL_VIF      0x00080000    // Virtual Interrupt Flag
0723 #define FL_VIP      0x00100000    // Virtual Interrupt Pending
0724 #define FL_ID      0x00200000    // ID flag
0725
0726 // Control Register flags
0727 #define CR0_PE      0x00000001    // Protection Enable
0728 #define CR0_MP      0x00000002    // Monitor coProcessor
0729 #define CR0_EM      0x00000004    // Emulation
0730 #define CR0_TS      0x00000008    // Task Switched
0731 #define CR0_ET      0x00000010    // Extension Type
0732 #define CR0_NE      0x00000020    // Numeric Error
0733 #define CR0_WP      0x00010000    // Write Protect
0734 #define CR0_AM      0x00040000    // Alignment Mask
0735 #define CR0_NW      0x02000000    // Not Writethrough
0736 #define CR0_CD      0x40000000    // Cache Disable
0737 #define CR0_PG      0x80000000    // Paging
0738
0739 #define CR4_PSE      0x00000010    // Page size extension
0740
0741 // various segment selectors.
0742 #define SEG_KCODE 1 // kernel code
0743 #define SEG_KDATA 2 // kernel data+stack
0744 #define SEG_KCPU  3 // kernel per-cpu data
0745 #define SEG_UCODE 4 // user code
0746 #define SEG_UDATA 5 // user data+stack
0747 #define SEG_TSS   6 // this process's task state
0748
0749

```

```

0750 // cpu->gdt[NSEGS] holds the above segments.
0751 #define NSEGS      7
0752
0753
0754
0755
0756
0757
0758
0759
0760
0761
0762
0763
0764
0765
0766
0767
0768
0769
0770
0771
0772
0773
0774
0775
0776
0777
0778
0779
0780
0781
0782
0783
0784
0785
0786
0787
0788
0789
0790
0791
0792
0793
0794
0795
0796
0797
0798
0799

```

```

0800 #ifndef __ASSEMBLER__
0801 // Segment Descriptor
0802 struct segdesc {
0803     uint lim_15_0 : 16; // Low bits of segment limit
0804     uint base_15_0 : 16; // Low bits of segment base address
0805     uint base_23_16 : 8; // Middle bits of segment base address
0806     uint type : 4;       // Segment type (see STS_ constants)
0807     uint s : 1;         // 0 = system, 1 = application
0808     uint dpl : 2;       // Descriptor Privilege Level
0809     uint p : 1;         // Present
0810     uint lim_19_16 : 4; // High bits of segment limit
0811     uint avl : 1;       // Unused (available for software use)
0812     uint rsv1 : 1;      // Reserved
0813     uint db : 1;        // 0 = 16-bit segment, 1 = 32-bit segment
0814     uint g : 1;         // Granularity: limit scaled by 4K when set
0815     uint base_31_24 : 8; // High bits of segment base address
0816 };
0817
0818 // Normal segment
0819 #define SEG(type, base, lim, dpl) (struct segdesc) \
0820 { ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff, \
0821   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
0822   (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
0823 #define SEG16(type, base, lim, dpl) (struct segdesc) \
0824 { (lim) & 0xffff, (uint)(base) & 0xffff, \
0825   ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
0826   (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
0827 #endif
0828
0829 #define DPL_USER 0x3 // User DPL
0830
0831 // Application segment type bits
0832 #define STA_X 0x8 // Executable segment
0833 #define STA_E 0x4 // Expand down (non-executable segments)
0834 #define STA_C 0x4 // Conforming code segment (executable only)
0835 #define STA_W 0x2 // Writeable (non-executable segments)
0836 #define STA_R 0x2 // Readable (executable segments)
0837 #define STA_A 0x1 // Accessed
0838
0839 // System segment type bits
0840 #define STS_T16A 0x1 // Available 16-bit TSS
0841 #define STS_LDT 0x2 // Local Descriptor Table
0842 #define STS_T16B 0x3 // Busy 16-bit TSS
0843 #define STS_CG16 0x4 // 16-bit Call Gate
0844 #define STS_TG 0x5 // Task Gate / Coum Transmissions
0845 #define STS_IG16 0x6 // 16-bit Interrupt Gate
0846 #define STS_TG16 0x7 // 16-bit Trap Gate
0847 #define STS_T32A 0x9 // Available 32-bit TSS
0848 #define STS_T32B 0xB // Busy 32-bit TSS
0849 #define STS_CG32 0xC // 32-bit Call Gate

```

```

0850 #define STS_IG32 0xE // 32-bit Interrupt Gate
0851 #define STS_TG32 0xF // 32-bit Trap Gate
0852
0853 // A virtual address 'la' has a three-part structure as follows:
0854 //
0855 // +-----10-----+-----10-----+-----12-----+
0856 // | Page Directory | Page Table | Offset within Page |
0857 // | Index          | Index      |                      |
0858 // +-----+-----+-----+
0859 // \--- PDX(va) --/ \--- PTX(va) --/
0860
0861 // page directory index
0862 #define PDX(va) (((uint)(va) >> PDXSHIFT) & 0x3FF)
0863
0864 // page table index
0865 #define PTX(va) (((uint)(va) >> PTXSHIFT) & 0x3FF)
0866
0867 // construct virtual address from indexes and offset
0868 #define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
0869
0870 // Page directory and page table constants.
0871 #define NPENTRIES 1024 // # directory entries per page directory
0872 #define NPTENTRIES 1024 // # PTEs per page table
0873 #define PGSIZE 4096 // bytes mapped by a page
0874
0875 #define PGSHIFT 12 // log2(PGSIZE)
0876 #define PTXSHIFT 12 // offset of PTX in a linear address
0877 #define PDXSHIFT 22 // offset of PDX in a linear address
0878
0879 #define PGROUNDUP(sz) (((sz)+PGSIZE-1) & ~(PGSIZE-1))
0880 #define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
0881
0882 // Page table/directory entry flags.
0883 #define PTE_P 0x001 // Present
0884 #define PTE_W 0x002 // Writeable
0885 #define PTE_U 0x004 // User
0886 #define PTE_PWT 0x008 // Write-Through
0887 #define PTE_PCD 0x010 // Cache-Disable
0888 #define PTE_A 0x020 // Accessed
0889 #define PTE_D 0x040 // Dirty
0890 #define PTE_PS 0x080 // Page Size
0891 #define PTE_MBZ 0x180 // Bits must be zero
0892
0893 // Address in page table or page directory entry
0894 #define PTE_ADDR(pte) ((uint)(pte) & ~0xFFF)
0895 #define PTE_FLAGS(pte) ((uint)(pte) & 0xFFF)
0896
0897 #ifndef __ASSEMBLER__
0898 typedef uint pte_t;
0899

```



```

0900 // Task state segment format
0901 struct taskstate {
0902     uint link;           // Old ts selector
0903     uint esp0;           // Stack pointers and segment selectors
0904     ushort ss0;         // after an increase in privilege level
0905     ushort padding1;
0906     uint *esp1;
0907     ushort ss1;
0908     ushort padding2;
0909     uint *esp2;
0910     ushort ss2;
0911     ushort padding3;
0912     void *cr3;           // Page directory base
0913     uint *eip;           // Saved state from last task switch
0914     uint eflags;
0915     uint eax;           // More saved state (registers)
0916     uint ecx;
0917     uint edx;
0918     uint ebx;
0919     uint *esp;
0920     uint *ebp;
0921     uint esi;
0922     uint edi;
0923     ushort es;          // Even more saved state (segment selectors)
0924     ushort padding4;
0925     ushort cs;
0926     ushort padding5;
0927     ushort ss;
0928     ushort padding6;
0929     ushort ds;
0930     ushort padding7;
0931     ushort fs;
0932     ushort padding8;
0933     ushort gs;
0934     ushort padding9;
0935     ushort ldt;
0936     ushort padding10;
0937     ushort t;           // Trap on task switch
0938     ushort iomb;        // I/O map base address
0939 };
0940
0941
0942
0943
0944
0945
0946
0947
0948
0949

```

```

0950 // Gate descriptors for interrupts and traps
0951 struct gatedesc {
0952     uint off_15_0 : 16;  // low 16 bits of offset in segment
0953     uint cs : 16;         // code segment selector
0954     uint args : 5;        // # args, 0 for interrupt/trap gates
0955     uint rsv1 : 3;        // reserved(should be zero I guess)
0956     uint type : 4;        // type(STS_{TG,IG32,TG32})
0957     uint s : 1;          // must be 0 (system)
0958     uint dpl : 2;        // descriptor(meaning new) privilege level
0959     uint p : 1;          // Present
0960     uint off_31_16 : 16; // high bits of offset in segment
0961 };
0962
0963 // Set up a normal interrupt/trap gate descriptor.
0964 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
0965 // - interrupt gate clears FL_IF, trap gate leaves FL_IF alone
0966 // - sel: Code segment selector for interrupt/trap handler
0967 // - off: Offset in code segment for interrupt/trap handler
0968 // - dpl: Descriptor Privilege Level -
0969 //       the privilege level required for software to invoke
0970 //       this interrupt/trap gate explicitly using an int instruction.
0971 #define SETGATE(gate, istrap, sel, off, d) \
0972 { \
0973     (gate).off_15_0 = (uint)(off) & 0xffff; \
0974     (gate).cs = (sel); \
0975     (gate).args = 0; \
0976     (gate).rsv1 = 0; \
0977     (gate).type = (istrap) ? STS_TG32 : STS_IG32; \
0978     (gate).s = 0; \
0979     (gate).dpl = (d); \
0980     (gate).p = 1; \
0981     (gate).off_31_16 = (uint)(off) >> 16; \
0982 }
0983
0984 #endif
0985
0986
0987
0988
0989
0990
0991
0992
0993
0994
0995
0996
0997
0998
0999

```

```

1000 // Format of an ELF executable file
1001
1002 #define ELF_MAGIC 0x464C457FU // "\x7FELF" in little endian
1003
1004 // File header
1005 struct elfhdr {
1006     uint magic; // must equal ELF_MAGIC
1007     uchar elf[12];
1008     ushort type;
1009     ushort machine;
1010     uint version;
1011     uint entry;
1012     uint phoff;
1013     uint shoff;
1014     uint flags;
1015     ushort ehsize;
1016     ushort phentsize;
1017     ushort phnum;
1018     ushort shentsize;
1019     ushort shnum;
1020     ushort shstrndx;
1021 };
1022
1023 // Program section header
1024 struct proghdr {
1025     uint type;
1026     uint off;
1027     uint vaddr;
1028     uint paddr;
1029     uint filesz;
1030     uint memsz;
1031     uint flags;
1032     uint align;
1033 };
1034
1035 // Values for Proghdr type
1036 #define ELF_PROG_LOAD 1
1037
1038 // Flag bits for Proghdr flags
1039 #define ELF_PROG_FLAG_EXEC 1
1040 #define ELF_PROG_FLAG_WRITE 2
1041 #define ELF_PROG_FLAG_READ 4
1042
1043
1044
1045
1046
1047
1048
1049

```

```

1050 // Blank page.
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099

```

```

1100 # The xv6 kernel starts executing in this file. This file is linked with
1101 # the kernel C code, so it can refer to kernel symbols such as main().
1102 # The boot block (bootasm.S and bootmain.c) jumps to entry below.
1103
1104 # Multiboot header, for multiboot boot loaders like GNU Grub.
1105 # http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
1106 #
1107 # Using GRUB 2, you can boot xv6 from a file stored in a
1108 # Linux file system by copying kernel or kernelmemfs to /boot
1109 # and then adding this menu entry:
1110 #
1111 # menuentry "xv6" {
1112 #   insmod ext2
1113 #   set root='(hd0,msdos1)'
1114 #   set kernel='/boot/kernel'
1115 #   echo "Loading ${kernel}..."
1116 #   multiboot ${kernel} ${kernel}
1117 #   boot
1118 # }
1119
1120 #include "asm.h"
1121 #include "memlayout.h"
1122 #include "mmu.h"
1123 #include "param.h"
1124
1125 # Multiboot header. Data to direct multiboot loader.
1126 .p2align 2
1127 .text
1128 .globl multiboot_header
1129 multiboot_header:
1130     #define magic 0x1badb002
1131     #define flags 0
1132     .long magic
1133     .long flags
1134     .long (-magic-flags)
1135
1136 # By convention, the _start symbol specifies the ELF entry point.
1137 # Since we haven't set up virtual memory yet, our entry point is
1138 # the physical address of 'entry'.
1139 .globl _start
1140 _start = V2P_W0(entry)
1141
1142 # Entering xv6 on boot processor, with paging off.
1143 .globl entry
1144 entry:
1145     # Turn on page size extension for 4Mbyte pages
1146     movl    %cr4, %eax
1147     orl     $(CR4_PSE), %eax
1148     movl    %eax, %cr4
1149     # Set page directory

```

```

1150     movl    $(V2P_W0(entrypgdir)), %eax
1151     movl    %eax, %cr3
1152     # Turn on paging.
1153     movl    %cr0, %eax
1154     orl     $(CR0_PG|CR0_WP), %eax
1155     movl    %eax, %cr0
1156
1157     # Set up the stack pointer.
1158     movl    $(stack + KSTACKSIZE), %esp
1159
1160     # Jump to main(), and switch to executing at
1161     # high addresses. The indirect call is needed because
1162     # the assembler produces a PC-relative instruction
1163     # for a direct jump.
1164     mov     $main, %eax
1165     jmp     *%eax
1166
1167 .comm stack, KSTACKSIZE
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199

```

```

1200 #include "asm.h"
1201 #include "memlayout.h"
1202 #include "mmu.h"
1203
1204 # Each non-boot CPU ("AP") is started up in response to a STARTUP
1205 # IPI from the boot CPU. Section B.4.2 of the Multi-Processor
1206 # Specification says that the AP will start in real mode with CS:IP
1207 # set to XY00:0000, where XY is an 8-bit value sent with the
1208 # STARTUP. Thus this code must start at a 4096-byte boundary.
1209 #
1210 # Because this code sets DS to zero, it must sit
1211 # at an address in the low 2^16 bytes.
1212 #
1213 # Startothers (in main.c) sends the STARTUPs one at a time.
1214 # It copies this code (start) at 0x7000. It puts the address of
1215 # a newly allocated per-core stack in start-4, the address of the
1216 # place to jump to (mpenter) in start-8, and the physical address
1217 # of entrypgdir in start-12.
1218 #
1219 # This code combines elements of bootasm.S and entry.S.
1220
1221 .code16
1222 .globl start
1223 start:
1224     cli
1225
1226     # Zero data segment registers DS, ES, and SS.
1227     xorw    %ax,%ax
1228     movw    %ax,%ds
1229     movw    %ax,%es
1230     movw    %ax,%ss
1231
1232     # Switch from real to protected mode. Use a bootstrap GDT that makes
1233     # virtual addresses map directly to physical addresses so that the
1234     # effective memory map doesn't change during the transition.
1235     lgdt    gdt desc
1236     movl    %cr0,%eax
1237     orl     $CRO_PE,%eax
1238     movl    %eax,%cr0
1239
1240     # Complete the transition to 32-bit protected mode by using a long jmp
1241     # to reload %cs and %eip. The segment descriptors are set up with no
1242     # translation, so that the mapping is still the identity mapping.
1243     ljmpl   $(SEG_KCODE<<3), $(start32)
1244
1245
1246
1247
1248
1249

```

```

1250 .code32 # Tell assembler to generate 32-bit code now.
1251 start32:
1252     # Set up the protected-mode data segment registers
1253     movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
1254     movw    %ax,%ds                # -> DS: Data Segment
1255     movw    %ax,%es                # -> ES: Extra Segment
1256     movw    %ax,%ss                # -> SS: Stack Segment
1257     movw    $0,%ax                 # Zero segments not ready for use
1258     movw    %ax,%fs                # -> FS
1259     movw    %ax,%gs                # -> GS
1260
1261     # Turn on page size extension for 4Mbyte pages
1262     movl    %cr4,%eax
1263     orl     $(CR4_PSE), %eax
1264     movl    %eax,%cr4
1265     # Use entrypgdir as our initial page table
1266     movl    (start-12), %eax
1267     movl    %eax,%cr3
1268     # Turn on paging.
1269     movl    %cr0,%eax
1270     orl     $(CRO_PE|CRO_PG|CRO_WP), %eax
1271     movl    %eax,%cr0
1272
1273     # Switch to the stack allocated by startothers()
1274     movl    (start-4), %esp
1275     # Call mpenter()
1276     call    *(start-8)
1277
1278     movw    $0x8a00,%ax
1279     movw    %ax,%dx
1280     outw    %ax,%dx
1281     movw    $0x8ae0,%ax
1282     outw    %ax,%dx
1283 spin:
1284     jmp     spin
1285
1286 .p2align 2
1287 gdt:
1288     SEG_NULLASM
1289     SEG_ASM(STA_X|STA_R, 0, 0xffffffff)
1290     SEG_ASM(STA_W, 0, 0xffffffff)
1291
1292
1293 gdt desc:
1294     .word   (gdt desc - gdt - 1)
1295     .long   gdt
1296
1297
1298
1299

```

```

1300 #include "types.h"
1301 #include "defs.h"
1302 #include "param.h"
1303 #include "memlayout.h"
1304 #include "mmu.h"
1305 #include "proc.h"
1306 #include "x86.h"
1307
1308 static void startothers(void);
1309 static void mpmain(void)  __attribute__((noreturn));
1310 extern pde_t *kpgdir;
1311 extern char end[]; // first address after kernel loaded from ELF file
1312
1313 // Bootstrap processor starts running C code here.
1314 // Allocate a real stack and switch to it, first
1315 // doing some setup required for memory allocator to work.
1316 int
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
1322     lapicinit(); // interrupt controller
1323     seginit(); // segment descriptors
1324     cprintf("\ncpu%d: starting xv6\n\n", cpunum());
1325     picinit(); // another interrupt controller
1326     ioapicinit(); // another interrupt controller
1327     consoleinit(); // console hardware
1328     uartinit(); // serial port
1329     pinit(); // process table
1330     tvinit(); // trap vectors
1331     binit(); // buffer cache
1332     fileinit(); // file table
1333     ideinit(); // disk
1334     if(!ismp)
1335         timerinit(); // uniprocessor timer
1336     startothers(); // start other processors
1337     kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
1338     userinit(); // first user process
1339     mpmain(); // finish this processor's setup
1340 }
1341
1342
1343
1344
1345
1346
1347
1348
1349

```

```

1350 // Other CPUs jump here from entryother.S.
1351 static void
1352 mpenter(void)
1353 {
1354     switchkvm();
1355     seginit();
1356     lapicinit();
1357     mpmain();
1358 }
1359
1360 // Common CPU setup code.
1361 static void
1362 mpmain(void)
1363 {
1364     cprintf("cpu%d: starting\n", cpunum());
1365     idtinit(); // load idt register
1366     xchg(&cpu->started, 1); // tell startothers() we're up
1367     scheduler(); // start running processes
1368 }
1369
1370 pde_t entrypgdir[]; // For entry.S
1371
1372 // Start the non-boot (AP) processors.
1373 static void
1374 startothers(void)
1375 {
1376     extern uchar _binary_entryother_start[], _binary_entryother_size[];
1377     uchar *code;
1378     struct cpu *c;
1379     char *stack;
1380
1381     // Write entry code to unused memory at 0x7000.
1382     // The linker has placed the image of entryother.S in
1383     // _binary_entryother_start.
1384     code = P2V(0x7000);
1385     memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
1386
1387     for(c = cpus; c < cpus+ncpu; c++){
1388         if(c == cpus+cpunum()) // We've started already.
1389             continue;
1390
1391         // Tell entryother.S what stack to use, where to enter, and what
1392         // pgdir to use. We cannot use kpgdir yet, because the AP processor
1393         // is running in low memory, so we use entrypgdir for the APs too.
1394         stack = kalloc();
1395         *(void**)(code-4) = stack + KSTACKSIZE;
1396         *(void**)(code-8) = mpenter;
1397         *(int**)(code-12) = (void *) V2P(entrypgdir);
1398
1399         lapicstartap(c->apicid, V2P(code));

```

```

1400 // wait for cpu to finish mpmain()
1401 while(c->started == 0)
1402     ;
1403 }
1404 }
1405
1406 // The boot page table used in entry.S and entryother.S.
1407 // Page directories (and page tables) must start on page boundaries,
1408 // hence the __aligned__ attribute.
1409 // PTE_PS in a page directory entry enables 4Mbyte pages.
1410
1411 __attribute__((__aligned__(PGSIZE)))
1412 pde_t entrypgdir[NPDENTRIES] = {
1413     // Map VA's [0, 4MB) to PA's [0, 4MB)
1414     [0] = (0) | PTE_P | PTE_W | PTE_PS,
1415     // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1416     [KERNBASE >> PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
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 = cpu;
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 to not re-order.
1615     __sync_synchronize();
1616
1617     // Release the lock.
1618     lk->locked = 0;
1619
1620     popcli();
1621 }
1622
1623 // Record the current call stack in pcs[] by following the %ebp chain.
1624 void
1625 getcallerpcs(void *v, uint pcs[])
1626 {
1627     uint *ebp;
1628     int i;
1629
1630     ebp = (uint*)v - 2;
1631     for(i = 0; i < 10; i++){
1632         if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
1633             break;
1634         pcs[i] = ebp[1]; // saved %eip
1635         ebp = (uint*)ebp[0]; // saved %ebp
1636     }
1637     for(; i < 10; i++)
1638         pcs[i] = 0;
1639 }
1640
1641 // Check whether this cpu is holding the lock.
1642 int
1643 holding(struct spinlock *lock)
1644 {
1645     return lock->locked && lock->cpu == cpu;
1646 }
1647
1648
1649

```

```

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

```



```

1700 #include "param.h"
1701 #include "types.h"
1702 #include "defs.h"
1703 #include "x86.h"
1704 #include "memlayout.h"
1705 #include "mmu.h"
1706 #include "proc.h"
1707 #include "elf.h"
1708
1709 extern char data[]; // defined by kernel.ld
1710 pde_t *kpgdir; // for use in scheduler()
1711
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 = &cpus[cpunum()];
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
1729     // Map cpu and proc -- these are private per cpu.
1730     c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
1731
1732     lgdt(c->gdt, sizeof(c->gdt));
1733     loadgs(SEG_KCPU << 3);
1734
1735     // Initialize cpu-local storage.
1736     cpu = c;
1737     proc = 0;
1738 }
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749

```

```

1750 // Return the address of the PTE in page table pgdir
1751 // that corresponds to virtual address va.  If alloc!=0,
1752 // create any required page table pages.
1753 static pte_t *
1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1755 {
1756     pde_t *pde;
1757     pte_t *pgtab;
1758
1759     pde = &pgdir[PDX(va)];
1760     if(*pde & PTE_P){
1761         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1762     } else {
1763         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1764             return 0;
1765         // Make sure all those PTE_P bits are zero.
1766         memset(pgtab, 0, PGSIZE);
1767         // The permissions here are overly generous, but they can
1768         // be further restricted by the permissions in the page table
1769         // entries, if necessary.
1770         *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1771     }
1772     return &pgtab[PTX(va)];
1773 }
1774
1775 // Create PTEs for virtual addresses starting at va that refer to
1776 // physical addresses starting at pa. va and size might not
1777 // be page-aligned.
1778 static int
1779 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1780 {
1781     char *a, *last;
1782     pte_t *pte;
1783
1784     a = (char*)PGROUNDDOWN((uint)va);
1785     last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1786     for(;;){
1787         if((pte = walkpgdir(pgdir, a, 1)) == 0)
1788             return -1;
1789         if(*pte & PTE_P)
1790             panic("remap");
1791         *pte = pa | perm | PTE_P;
1792         if(a == last)
1793             break;
1794         a += PGSIZE;
1795         pa += PGSIZE;
1796     }
1797     return 0;
1798 }
1799

```

```

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

```

```

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

```

```

1900 // Load the initcode into address 0 of pgdir.
1901 // sz must be less than a page.
1902 void
1903 inituvm(pde_t *pgdir, char *init, uint sz)
1904 {
1905     char *mem;
1906
1907     if(sz >= PGSIZE)
1908         panic("inituvm: more than a page");
1909     mem = kalloc();
1910     memset(mem, 0, PGSIZE);
1911     mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W|PTE_U);
1912     memmove(mem, init, sz);
1913 }
1914
1915 // Load a program segment into pgdir.  addr must be page-aligned
1916 // and the pages from addr to addr+sz must already be mapped.
1917 int
1918 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
1919 {
1920     uint i, pa, n;
1921     pte_t *pte;
1922
1923     if((uint) addr % PGSIZE != 0)
1924         panic("loaduvm: addr must be page aligned");
1925     for(i = 0; i < sz; i += PGSIZE){
1926         if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1927             panic("loaduvm: address should exist");
1928         pa = PTE_ADDR(*pte);
1929         if(sz - i < PGSIZE)
1930             n = sz - i;
1931         else
1932             n = PGSIZE;
1933         if(readi(ip, P2V(pa), offset+i, n) != n)
1934             return -1;
1935     }
1936     return 0;
1937 }
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949

```

```

1950 // Allocate page tables and physical memory to grow process from oldsz to
1951 // newsz, which need not be page aligned.  Returns new size or 0 on error.
1952 int
1953 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1954 {
1955     char *mem;
1956     uint a;
1957
1958     if(newsz >= KERNBASE)
1959         return 0;
1960     if(newsz < oldsz)
1961         return oldsz;
1962
1963     a = PGROUNDUP(oldsz);
1964     for(; a < newsz; a += PGSIZE){
1965         mem = kalloc();
1966         if(mem == 0){
1967             cprintf("allocuvm out of memory\n");
1968             deallocuvm(pgdir, newsz, oldsz);
1969             return 0;
1970         }
1971         memset(mem, 0, PGSIZE);
1972         if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
1973             cprintf("allocuvm out of memory (2)\n");
1974             deallocuvm(pgdir, newsz, oldsz);
1975             kfree(mem);
1976             return 0;
1977         }
1978     }
1979     return newsz;
1980 }
1981
1982 // Deallocate user pages to bring the process size from oldsz to
1983 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
1984 // need to be less than oldsz.  oldsz can be larger than the actual
1985 // process size.  Returns the new process size.
1986 int
1987 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1988 {
1989     pte_t *pte;
1990     uint a, pa;
1991
1992     if(newsz >= oldsz)
1993         return oldsz;
1994
1995     a = PGROUNDUP(newsz);
1996     for(; a < oldsz; a += PGSIZE){
1997         pte = walkpgdir(pgdir, (char*)a, 0);
1998         if(!pte)
1999             a += (NPENTRIES - 1) * PGSIZE;

```

```

2000     else if((*pte & PTE_P) != 0){
2001         pa = PTE_ADDR(*pte);
2002         if(pa == 0)
2003             panic("kfree");
2004         char *v = P2V(pa);
2005         kfree(v);
2006         *pte = 0;
2007     }
2008 }
2009 return newsz;
2010 }
2011
2012 // Free a page table and all the physical memory pages
2013 // in the user part.
2014 void
2015 freevm(pde_t *pgdir)
2016 {
2017     uint i;
2018
2019     if(pgdir == 0)
2020         panic("freevm: no pgdir");
2021     deallocvm(pgdir, KERNBASE, 0);
2022     for(i = 0; i < NPENTRIES; i++){
2023         if(pgdir[i] & PTE_P){
2024             char *v = P2V(PTE_ADDR(pgdir[i]));
2025             kfree(v);
2026         }
2027     }
2028     kfree((char*)pgdir);
2029 }
2030
2031 // Clear PTE_U on a page. Used to create an inaccessible
2032 // page beneath the user stack.
2033 void
2034 clearpteu(pde_t *pgdir, char *uva)
2035 {
2036     pte_t *pte;
2037
2038     pte = walkpgdir(pgdir, uva, 0);
2039     if(pte == 0)
2040         panic("clearpteu");
2041     *pte &= ~PTE_U;
2042 }
2043
2044
2045
2046
2047
2048
2049

```

```

2050 // Given a parent process's page table, create a copy
2051 // of it for a child.
2052 pde_t*
2053 copyvm(pde_t *pgdir, uint sz)
2054 {
2055     pde_t *d;
2056     pte_t *pte;
2057     uint pa, i, flags;
2058     char *mem;
2059
2060     if((d = setupkvm()) == 0)
2061         return 0;
2062     for(i = 0; i < sz; i += PGSIZE){
2063         if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
2064             panic("copyvm: pte should exist");
2065         if(!(*pte & PTE_P))
2066             panic("copyvm: page not present");
2067         pa = PTE_ADDR(*pte);
2068         flags = PTE_FLAGS(*pte);
2069         if((mem = kalloc()) == 0)
2070             goto bad;
2071         memmove(mem, (char*)P2V(pa), PGSIZE);
2072         if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
2073             goto bad;
2074     }
2075     return d;
2076
2077 bad:
2078     freevm(d);
2079     return 0;
2080 }
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099

```

```

2100 // Map user virtual address to kernel address.
2101 char*
2102 uva2ka(pde_t *pgdir, char *uva)
2103 {
2104     pte_t *pte;
2105
2106     pte = walkpgdir(pgdir, uva, 0);
2107     if((*pte & PTE_P) == 0)
2108         return 0;
2109     if((*pte & PTE_U) == 0)
2110         return 0;
2111     return (char*)P2V(PTE_ADDR(*pte));
2112 }
2113
2114 // Copy len bytes from p to user address va in page table pgdir.
2115 // Most useful when pgdir is not the current page table.
2116 // uva2ka ensures this only works for PTE_U pages.
2117 int
2118 copyout(pde_t *pgdir, uint va, void *p, uint len)
2119 {
2120     char *buf, *pa0;
2121     uint n, va0;
2122
2123     buf = (char*)p;
2124     while(len > 0){
2125         va0 = (uint)PGROUNDDOWN(va);
2126         pa0 = uva2ka(pgdir, (char*)va0);
2127         if(pa0 == 0)
2128             return -1;
2129         n = PGSIZE - (va - va0);
2130         if(n > len)
2131             n = len;
2132         memmove(pa0 + (va - va0), buf, n);
2133         len -= n;
2134         buf += n;
2135         va = va0 + PGSIZE;
2136     }
2137     return 0;
2138 }
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149

```

```

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

```

2200 // Blank page.

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

2250 // Blank page.

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

```

2300 // 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
2310     // Cpu-local storage variables; see below
2311     struct cpu *cpu;
2312     struct proc *proc;       // The currently-running process.
2313 };
2314
2315 extern struct cpu cpus[NCPU];
2316 extern int ncpu;
2317
2318 // Per-CPU variables, holding pointers to the
2319 // current cpu and to the current process.
2320 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2321 // and "%gs:4" to refer to proc.  seginit sets up the
2322 // %gs segment register so that %gs refers to the memory
2323 // holding those two variables in the local cpu's struct cpu.
2324 // This is similar to how thread-local variables are implemented
2325 // in thread libraries such as Linux pthreads.
2326 extern struct cpu *cpu asm("%gs:0"); // &cpus[cpunum()]
2327 extern struct proc *proc asm("%gs:4"); // cpus[cpunum()].proc
2328
2329
2330 // Saved registers for kernel context switches.
2331 // Don't need to save all the segment registers (%cs, etc),
2332 // because they are constant across kernel contexts.
2333 // Don't need to save %eax, %ecx, %edx, because the
2334 // x86 convention is that the caller has saved them.
2335 // Contexts are stored at the bottom of the stack they
2336 // describe; the stack pointer is the address of the context.
2337 // The layout of the context matches the layout of the stack in swtch.S
2338 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2339 // but it is on the stack and allocproc() manipulates it.
2340 struct context {
2341     uint edi;
2342     uint esi;
2343     uint ebx;
2344     uint ebp;
2345     uint eip;
2346 };
2347
2348
2349

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

2750 // Enter scheduler.  Must hold only ptable.lock
2751 // and have changed proc->state.  Saves and restores
2752 // intena because intena is a property of this
2753 // kernel thread, not this CPU.  It should
2754 // be proc->intena and proc->ncli, but that would
2755 // break in the few places where a lock is held but
2756 // there's no process.
2757 void
2758 sched(void) save the current process and load the context of the scheduler
2759 {
2760     int intena;
2761
2762     if(!holding(&ptable.lock))
2763         panic("sched ptable.lock");
2764     if(cpu->ncli != 1)
2765         panic("sched locks");
2766     if(proc->state == RUNNING)
2767         panic("sched running");
2768     if(readeflags() & FL_IF)
2769         panic("sched interruptible");
2770     intena = cpu->intena;
2771     swtch(&proc->context, cpu->scheduler);
2772     cpu->intena = intena;
2773 }
2774
2775 // Give up the CPU for one scheduling round.
2776 void
2777 yield(void) to yield it has to make it runnable, then calls a routine sched()
2778 {
2779     acquire(&ptable.lock);
2780     proc->state = RUNNABLE;
2781     sched();
2782     release(&ptable.lock);
2783 }
2784
2785 // A fork child's very first scheduling by scheduler()
2786 // will swtch here.  "Return" to user space.
2787 void
2788 forkret(void)
2789 {
2790     static int first = 1;
2791     // Still holding ptable.lock from scheduler.
2792     release(&ptable.lock);
2793
2794     if (first) {
2795         // Some initialization functions must be run in the context
2796         // of a regular process (e.g., they call sleep), and thus cannot
2797         // be run from main().
2798         first = 0;
2799         iinit(ROOTDEV);

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

3050 void
3051 freerange(void *vstart, void *vend)
3052 {
3053   char *p;
3054   p = (char*)PGROUNDUP((uint)vstart);
3055   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3056     kfree(p);
3057 }
3058
3059
3060 // Free the page of physical memory pointed at by v,
3061 // which normally should have been returned by a
3062 // call to kalloc(). (The exception is when
3063 // initializing the allocator; see kinit above.)
3064 void
3065 kfree(char *v)
3066 {
3067   struct run *r;
3068
3069   if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
3070     panic("kfree");
3071
3072   // Fill with junk to catch dangling refs.
3073   memset(v, 1, PGSIZE);
3074
3075   if(kmem.use_lock)
3076     acquire(&kmem.lock);
3077   r = (struct run*)v;
3078   r->next = kmem.freelist;
3079   kmem.freelist = r;
3080   if(kmem.use_lock)
3081     release(&kmem.lock);
3082 }
3083
3084 // Allocate one 4096-byte page of physical memory.
3085 // Returns a pointer that the kernel can use.
3086 // Returns 0 if the memory cannot be allocated.
3087 char*
3088 kalloc(void)
3089 {
3090   struct run *r;
3091
3092   if(kmem.use_lock)
3093     acquire(&kmem.lock);
3094   r = kmem.freelist;
3095   if(r)
3096     kmem.freelist = r->next;
3097   if(kmem.use_lock)
3098     release(&kmem.lock);
3099   return (char*)r;

```

```

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

```

```

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

```

```

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

```

```

3250 #include "mmu.h"
3251
3252 # vectors.S sends all traps here.
3253 .globl alltraps
3254 alltraps:
3255     # Build trap frame.
3256     pushl %ds
3257     pushl %es
3258     pushl %fs
3259     pushl %gs
3260     pushal
3261
3262     # Set up data and per-cpu segments.
3263     movw $(SEG_KDATA<<3), %ax
3264     movw %ax, %ds
3265     movw %ax, %es
3266     movw $(SEG_KCPU<<3), %ax
3267     movw %ax, %fs
3268     movw %ax, %gs
3269
3270     # Call trap(tf), where tf=%esp
3271     pushl %esp
3272     call trap
3273     addl $4, %esp
3274
3275     # Return falls through to trapret...
3276 .globl trapret
3277 trapret:
3278     popal
3279     popl %gs
3280     popl %fs
3281     popl %es
3282     popl %ds
3283     addl $0x8, %esp # trapno and errcode
3284     iret
3285
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299

```



```

3300 #include "types.h"
3301 #include "defs.h"
3302 #include "param.h"
3303 #include "memlayout.h"
3304 #include "mmu.h"
3305 #include "proc.h"
3306 #include "x86.h"
3307 #include "traps.h"
3308 #include "spinlock.h"
3309
3310 // Interrupt descriptor table (shared by all CPUs).
3311 struct gatedesc idt[256];
3312 extern uint vectors[]; // in vectors.S: array of 256 entry pointers
3313 struct spinlock tickslock;
3314 uint ticks;
3315
3316 void
3317 tvinit(void)
3318 {
3319     int i;
3320
3321     for(i = 0; i < 256; i++)
3322         SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3323     SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3324
3325     initlock(&tickslock, "time");
3326 }
3327
3328 void
3329 idtinit(void)
3330 {
3331     lidt(idt, sizeof(idt));
3332 }
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

3900 // On-disk file system format.
3901 // Both the kernel and user programs use this header file.
3902
3903
3904 #define ROOTINO 1 // root i-number
3905 #define BSIZE 512 // block size
3906
3907 // Disk layout:
3908 // [ boot block | super block | log | inode blocks |
3909 //                               free bit map | data blocks]
3910 //
3911 // mkfs computes the super block and builds an initial file system. The
3912 // super block describes the disk layout:
3913 struct superblock {
3914     uint size; // Size of file system image (blocks)
3915     uint nblocks; // Number of data blocks
3916     uint ninodes; // Number of inodes.
3917     uint nlog; // Number of log blocks
3918     uint logstart; // Block number of first log block
3919     uint inodestart; // Block number of first inode block
3920     uint bmapstart; // Block number of first free map block
3921 };
3922
3923 #define NDIRECT 12
3924 #define NINDIRECT (BSIZE / sizeof(uint))
3925 #define MAXFILE (NDIRECT + NINDIRECT)
3926
3927 // On-disk inode structure
3928 struct dinode {
3929     short type; // File type
3930     short major; // Major device number (T_DEV only)
3931     short minor; // Minor device number (T_DEV only)
3932     short nlink; // Number of links to inode in file system
3933     uint size; // Size of file (bytes)
3934     uint addrs[NDIRECT+1]; // Data block addresses
3935 };
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949

```

```

3950 // Inodes per block.
3951 #define IPB (BSIZE / sizeof(struct dinode))
3952
3953 // Block containing inode i
3954 #define IBLOCK(i, sb) ((i) / IPB + sb.inodestart)
3955
3956 // Bitmap bits per block
3957 #define BPB (BSIZE*8)
3958
3959 // Block of free map containing bit for block b
3960 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
3961
3962 // Directory is a file containing a sequence of dirent structures.
3963 #define DIRSIZ 14
3964
3965 struct dirent {
3966     ushort inum;
3967     char name[DIRSIZ];
3968 };
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999

```

```

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

```

```

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

```



```

4100 // Simple PIO-based (non-DMA) IDE driver code.
4101
4102 #include "types.h"
4103 #include "defs.h"
4104 #include "param.h"
4105 #include "memlayout.h"
4106 #include "mmu.h"
4107 #include "proc.h"
4108 #include "x86.h"
4109 #include "traps.h"
4110 #include "spinlock.h"
4111 #include "fs.h"
4112 #include "buf.h"
4113
4114 #define SECTOR_SIZE    512
4115 #define IDE_BSY        0x80
4116 #define IDE_DRDY        0x40
4117 #define IDE_DF          0x20
4118 #define IDE_ERR        0x01
4119
4120 #define IDE_CMD_READ    0x20
4121 #define IDE_CMD_WRITE  0x30
4122 #define IDE_CMD_RDMUL  0xc4
4123 #define IDE_CMD_WRMUL  0xc5
4124
4125 // idequeue points to the buf now being read/written to the disk.
4126 // idequeue->qnext points to the next buf to be processed.
4127 // You must hold idelock while manipulating queue.
4128
4129 static struct spinlock idelock;
4130 static struct buf *idequeue;
4131
4132 static int havedisk1;
4133 static void idestart(struct buf*);
4134
4135 // Wait for IDE disk to become ready.
4136 static int
4137 idewait(int checkerr)
4138 {
4139     int r;
4140
4141     while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
4142         ;
4143     if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
4144         return -1;
4145     return 0;
4146 }
4147
4148
4149

```

```

4150 void
4151 ideinit(void)
4152 {
4153     int i;
4154
4155     initlock(&idelock, "ide");
4156     picenable(IRQ_IDE);
4157     ioapicenable(IRQ_IDE, ncpu - 1);
4158     idewait(0);
4159
4160     // Check if disk 1 is present
4161     outb(0x1f6, 0xe0 | (1<<4));
4162     for(i=0; i<1000; i++){
4163         if(inb(0x1f7) != 0){
4164             havedisk1 = 1;
4165             break;
4166         }
4167     }
4168
4169     // Switch back to disk 0.
4170     outb(0x1f6, 0xe0 | (0<<4));
4171 }
4172
4173 // Start the request for b. Caller must hold idelock.
4174 static void
4175 idestart(struct buf *b)
4176 {
4177     if(b == 0)
4178         panic("idestart");
4179     if(b->blockno >= FSSIZE)
4180         panic("incorrect blockno");
4181     int sector_per_block = BSIZE/SECTOR_SIZE;
4182     int sector = b->blockno * sector_per_block;
4183     int read_cmd = (sector_per_block == 1) ? IDE_CMD_READ : IDE_CMD_RDMUL;
4184     int write_cmd = (sector_per_block == 1) ? IDE_CMD_WRITE : IDE_CMD_WRMUL;
4185
4186     if (sector_per_block > 7) panic("idestart");
4187
4188     idewait(0);
4189     outb(0x3f6, 0); // generate interrupt
4190     outb(0x1f2, sector_per_block); // number of sectors
4191     outb(0x1f3, sector & 0xff);
4192     outb(0x1f4, (sector >> 8) & 0xff);
4193     outb(0x1f5, (sector >> 16) & 0xff);
4194     outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
4195     if(b->flags & B_DIRTY){
4196         outb(0x1f7, write_cmd);
4197         outsl(0x1f0, b->data, BSIZE/4);
4198     } else {
4199         outb(0x1f7, read_cmd);

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

5750 // Write to file f.
5751 int
5752 filewrite(struct file *f, char *addr, int n)
5753 {
5754     int r;
5755
5756     if(f->writable == 0)
5757         return -1;
5758     if(f->type == FD_PIPE)
5759         return pipewrite(f->pipe, addr, n);
5760     if(f->type == FD_INODE){
5761         // write a few blocks at a time to avoid exceeding
5762         // the maximum log transaction size, including
5763         // i-node, indirect block, allocation blocks,
5764         // and 2 blocks of slop for non-aligned writes.
5765         // this really belongs lower down, since writei()
5766         // might be writing a device like the console.
5767         int max = ((LOGSIZE-1-1-2) / 2) * 512;
5768         int i = 0;
5769         while(i < n){
5770             int n1 = n - i;
5771             if(n1 > max)
5772                 n1 = max;
5773
5774             begin_op();
5775             ilock(f->ip);
5776             if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
5777                 f->off += r;
5778             iunlock(f->ip);
5779             end_op();
5780
5781             if(r < 0)
5782                 break;
5783             if(r != n1)
5784                 panic("short filewrite");
5785             i += r;
5786         }
5787         return i == n ? n : -1;
5788     }
5789     panic("filewrite");
5790 }
5791
5792
5793
5794
5795
5796
5797
5798
5799

```

```

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

```

```

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

```

```

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

```

```

5950     return 0;
5951
5952 bad:
5953     ilock(ip);
5954     ip->nlink--;
5955     iupdate(ip);
5956     iunlockput(ip);
5957     end_op();
5958     return -1;
5959 }
5960
5961 // Is the directory dp empty except for "." and ".." ?
5962 static int
5963 isdirempty(struct inode *dp)
5964 {
5965     int off;
5966     struct dirent de;
5967
5968     for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
5969         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5970             panic("isdirempty: readi");
5971         if(de.inum != 0)
5972             return 0;
5973     }
5974     return 1;
5975 }
5976
5977
5978
5979
5980
5981
5982
5983
5984
5985
5986
5987
5988
5989
5990
5991
5992
5993
5994
5995
5996
5997
5998
5999

```

```

6000 int
6001 sys_unlink(void)
6002 {
6003     struct inode *ip, *dp;
6004     struct dirent de;
6005     char name[DIRSIZ], *path;
6006     uint off;
6007
6008     if(argstr(0, &path) < 0)
6009         return -1;
6010
6011     begin_op();
6012     if((dp = nameiparent(path, name)) == 0){
6013         end_op();
6014         return -1;
6015     }
6016
6017     ilock(dp);
6018
6019     // Cannot unlink "." or "..".
6020     if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6021         goto bad;
6022
6023     if((ip = dirlookup(dp, name, &off)) == 0)
6024         goto bad;
6025     ilock(ip);
6026
6027     if(ip->nlink < 1)
6028         panic("unlink: nlink < 1");
6029     if(ip->type == T_DIR && !isdirempty(ip)){
6030         iunlockput(ip);
6031         goto bad;
6032     }
6033
6034     memset(&de, 0, sizeof(de));
6035     if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6036         panic("unlink: writei");
6037     if(ip->type == T_DIR){
6038         dp->nlink--;
6039         iupdate(dp);
6040     }
6041     iunlockput(dp);
6042
6043     ip->nlink--;
6044     iupdate(ip);
6045     iunlockput(ip);
6046
6047     end_op();
6048
6049     return 0;

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

6300 #include "types.h"
6301 #include "param.h"
6302 #include "memlayout.h"
6303 #include "mmu.h"
6304 #include "proc.h"
6305 #include "defs.h"
6306 #include "x86.h"
6307 #include "elf.h"
6308
6309 int
6310 exec(char *path, char **argv)
6311 {
6312     char *s, *last;
6313     int i, off;
6314     uint argc, sz, sp, ustack[3+MAXARG+1];
6315     struct elfhdr elf;
6316     struct inode *ip;
6317     struct proghdr ph;
6318     pde_t *pgdir, *oldpgdir;
6319
6320     begin_op();
6321     if((ip = namei(path)) == 0){
6322         end_op();
6323         return -1;
6324     }
6325     ilock(ip);
6326     pgdir = 0;
6327
6328     // Check ELF header
6329     if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf)){
6330         goto bad;
6331     }
6332     if(elf.magic != ELF_MAGIC)
6333         goto bad;
6334
6335     if((pgdir = setupkvm()) == 0)
6336         goto bad;
6337
6338     // Load program into memory.
6339     sz = 0;
6340     for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6341         if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6342             goto bad;
6343         if(ph.type != ELF_PROG_LOAD)
6344             continue;
6345         if(ph.memsz < ph.filesz)
6346             goto bad;
6347         if(ph.vaddr + ph.memsz < ph.vaddr)
6348             goto bad;
6349         if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
6350             goto bad;

```

```

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

```

```

6400     return 0;
6401
6402 bad:
6403     if(pgdir)
6404         freevm(pgdir);
6405     if(ip){
6406         iunlockput(ip);
6407         end_op();
6408     }
6409     return -1;
6410 }
6411
6412
6413
6414
6415
6416
6417
6418
6419
6420
6421
6422
6423
6424
6425
6426
6427
6428
6429
6430
6431
6432
6433
6434
6435
6436
6437
6438
6439
6440
6441
6442
6443
6444
6445
6446
6447
6448
6449

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

6700 int
6701 strlen(const char *s)
6702 {
6703     int n;
6704
6705     for(n = 0; s[n]; n++)
6706         ;
6707     return n;
6708 }
6709
6710
6711
6712
6713
6714
6715
6716
6717
6718
6719
6720
6721
6722
6723
6724
6725
6726
6727
6728
6729
6730
6731
6732
6733
6734
6735
6736
6737
6738
6739
6740
6741
6742
6743
6744
6745
6746
6747
6748
6749

```

```

6750 // See MultiProcessor Specification Version 1.[14]
6751
6752 struct mp {                // floating pointer
6753     uchar signature[4];    // "_MP_"
6754     void *physaddr;        // phys addr of MP config table
6755     uchar length;          // 1
6756     uchar specrev;         // [14]
6757     uchar checksum;        // all bytes must add up to 0
6758     uchar type;            // MP system config type
6759     uchar imcrp;
6760     uchar reserved[3];
6761 };
6762
6763 struct mpconf {            // configuration table header
6764     uchar signature[4];    // "PCMP"
6765     ushort length;         // total table length
6766     uchar version;         // [14]
6767     uchar checksum;        // all bytes must add up to 0
6768     uchar product[20];     // product id
6769     uint *oemtable;        // OEM table pointer
6770     ushort oemlength;      // OEM table length
6771     ushort entry;          // entry count
6772     uint *lapicaddr;       // address of local APIC
6773     ushort xlength;        // extended table length
6774     uchar xchecksum;       // extended table checksum
6775     uchar reserved;
6776 };
6777
6778 struct mpproc {            // processor table entry
6779     uchar type;            // entry type (0)
6780     uchar apicid;          // local APIC id
6781     uchar version;         // local APIC version
6782     uchar flags;           // CPU flags
6783     #define MPBOOT 0x02    // This proc is the bootstrap processor.
6784     uchar signature[4];    // CPU signature
6785     uint feature;          // feature flags from CPUID instruction
6786     uchar reserved[8];
6787 };
6788
6789 struct mpioapic {          // I/O APIC table entry
6790     uchar type;            // entry type (2)
6791     uchar apicno;          // I/O APIC id
6792     uchar version;         // I/O APIC version
6793     uchar flags;           // I/O APIC flags
6794     uint *addr;            // I/O APIC address
6795 };
6796
6797
6798
6799

```

```

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

```

```

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

```

```

6900 // Multiprocessor support
6901 // Search memory for MP description structures.
6902 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
6903
6904 #include "types.h"
6905 #include "defs.h"
6906 #include "param.h"
6907 #include "memlayout.h"
6908 #include "mp.h"
6909 #include "x86.h"
6910 #include "mmu.h"
6911 #include "proc.h"
6912
6913 struct cpu cpus[NCPU];
6914 int ismp;
6915 int ncpu;
6916 uchar ioapicid;
6917
6918 static uchar
6919 sum(uchar *addr, int len)
6920 {
6921     int i, sum;
6922
6923     sum = 0;
6924     for(i=0; i<len; i++)
6925         sum += addr[i];
6926     return sum;
6927 }
6928
6929 // Look for an MP structure in the len bytes at addr.
6930 static struct mp*
6931 mpsearch1(uint a, int len)
6932 {
6933     uchar *e, *p, *addr;
6934
6935     addr = P2V(a);
6936     e = addr+len;
6937     for(p = addr; p < e; p += sizeof(struct mp))
6938         if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
6939             return (struct mp*)p;
6940     return 0;
6941 }
6942
6943
6944
6945
6946
6947
6948
6949

```

```

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

```

```

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

```

```

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

```

```

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

```

```

7150 static void
7151 lapicw(int index, int value)
7152 {
7153     lapic[index] = value;
7154     lapic[ID]; // wait for write to finish, by reading
7155 }
7156
7157
7158
7159
7160
7161
7162
7163
7164
7165
7166
7167
7168
7169
7170
7171
7172
7173
7174
7175
7176
7177
7178
7179
7180
7181
7182
7183
7184
7185
7186
7187
7188
7189
7190
7191
7192
7193
7194
7195
7196
7197
7198
7199

```

```

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

```

```

7250 int
7251 cpunum(void)
7252 {
7253     int apicid, i;
7254
7255     // Cannot call cpu when interrupts are enabled:
7256     // result not guaranteed to last long enough to be used!
7257     // Would prefer to panic but even printing is chancy here:
7258     // almost everything, including cprintf and panic, calls cpu,
7259     // often indirectly through acquire and release.
7260     if(readeflags() & FL_IF){
7261         static int n;
7262         if(n++ == 0)
7263             cprintf("cpu called from %x with interrupts enabled\n",
7264                     __builtin_return_address(0));
7265     }
7266
7267     if (!lapic)
7268         return 0;
7269
7270     apicid = lapic[ID] >> 24;
7271     for (i = 0; i < ncpu; ++i) {
7272         if (cpus[i].apicid == apicid)
7273             return i;
7274     }
7275     panic("unknown apicid\n");
7276 }
7277
7278 // Acknowledge interrupt.
7279 void
7280 lapiceoi(void)
7281 {
7282     if(lapic)
7283         lapicw(EOI, 0);
7284 }
7285
7286 // Spin for a given number of microseconds.
7287 // On real hardware would want to tune this dynamically.
7288 void
7289 microdelay(int us)
7290 {
7291 }
7292
7293
7294
7295
7296
7297
7298
7299

```



```

7300 #define CMOS_PORT    0x70
7301 #define CMOS_RETURN  0x71
7302
7303 // Start additional processor running entry code at addr.
7304 // See Appendix B of MultiProcessor Specification.
7305 void
7306 lapicstartap(uchar apicid, uint addr)
7307 {
7308     int i;
7309     ushort *wrv;
7310
7311     // "The BSP must initialize CMOS shutdown code to 0AH
7312     // and the warm reset vector (DWORD based at 40:67) to point at
7313     // the AP startup code prior to the [universal startup algorithm]."
7314     outb(CMOS_PORT, 0xF); // offset 0xF is shutdown code
7315     outb(CMOS_PORT+1, 0x0A);
7316     wrv = (ushort*)P2V((0x40<<4 | 0x67)); // Warm reset vector
7317     wrv[0] = 0;
7318     wrv[1] = addr >> 4;
7319
7320     // "Universal startup algorithm."
7321     // Send INIT (level-triggered) interrupt to reset other CPU.
7322     lapicw(ICRHI, apicid<<24);
7323     lapicw(ICRLO, INIT | LEVEL | ASSERT);
7324     microdelay(200);
7325     lapicw(ICRLO, INIT | LEVEL);
7326     microdelay(100); // should be 10ms, but too slow in Bochs!
7327
7328     // Send startup IPI (twice!) to enter code.
7329     // Regular hardware is supposed to only accept a STARTUP
7330     // when it is in the halted state due to an INIT. So the second
7331     // should be ignored, but it is part of the official Intel algorithm.
7332     // Bochs complains about the second one. Too bad for Bochs.
7333     for(i = 0; i < 2; i++){
7334         lapicw(ICRHI, apicid<<24);
7335         lapicw(ICRLO, STARTUP | (addr>>12));
7336         microdelay(200);
7337     }
7338 }
7339
7340
7341
7342
7343
7344
7345
7346
7347
7348
7349

```

```

7350 #define CMOS_STATA    0x0a
7351 #define CMOS_STATB    0x0b
7352 #define CMOS_UIP      (1 << 7) // RTC update in progress
7353
7354 #define SECS          0x00
7355 #define MINS          0x02
7356 #define HOURS         0x04
7357 #define DAY           0x07
7358 #define MONTH         0x08
7359 #define YEAR          0x09
7360
7361 static uint cmos_read(uint reg)
7362 {
7363     outb(CMOS_PORT, reg);
7364     microdelay(200);
7365
7366     return inb(CMOS_RETURN);
7367 }
7368
7369 static void fill_rtcdate(struct rtcdate *r)
7370 {
7371     r->second = cmos_read(SECS);
7372     r->minute = cmos_read(MINS);
7373     r->hour   = cmos_read(HOURS);
7374     r->day    = cmos_read(DAY);
7375     r->month  = cmos_read(MONTH);
7376     r->year   = cmos_read(YEAR);
7377 }
7378
7379 // qemu seems to use 24-hour GWT and the values are BCD encoded
7380 void cmostime(struct rtcdate *r)
7381 {
7382     struct rtcdate t1, t2;
7383     int sb, bcd;
7384
7385     sb = cmos_read(CMOS_STATB);
7386
7387     bcd = (sb & (1 << 2)) == 0;
7388
7389     // make sure CMOS doesn't modify time while we read it
7390     for(;;) {
7391         fill_rtcdate(&t1);
7392         if(cmos_read(CMOS_STATA) & CMOS_UIP)
7393             continue;
7394         fill_rtcdate(&t2);
7395         if(memcmp(&t1, &t2, sizeof(t1)) == 0)
7396             break;
7397     }
7398
7399

```

```

7400 // convert
7401 if(bcd) {
7402 #define CONV(x)      (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
7403     CONV(second);
7404     CONV(minute);
7405     CONV(hour );
7406     CONV(day );
7407     CONV(month );
7408     CONV(year );
7409 #undef CONV
7410 }
7411
7412 *r = t1;
7413 r->year += 2000;
7414 }
7415
7416
7417
7418
7419
7420
7421
7422
7423
7424
7425
7426
7427
7428
7429
7430
7431
7432
7433
7434
7435
7436
7437
7438
7439
7440
7441
7442
7443
7444
7445
7446
7447
7448
7449

```

```

7450 // The I/O APIC manages hardware interrupts for an SMP system.
7451 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
7452 // See also picirq.c.
7453
7454 #include "types.h"
7455 #include "defs.h"
7456 #include "traps.h"
7457
7458 #define IOAPIC 0xFEC00000 // Default physical address of IO APIC
7459
7460 #define REG_ID 0x00 // Register index: ID
7461 #define REG_VER 0x01 // Register index: version
7462 #define REG_TABLE 0x10 // Redirection table base
7463
7464 // The redirection table starts at REG_TABLE and uses
7465 // two registers to configure each interrupt.
7466 // The first (low) register in a pair contains configuration bits.
7467 // The second (high) register contains a bitmask telling which
7468 // CPUs can serve that interrupt.
7469 #define INT_DISABLED 0x00010000 // Interrupt disabled
7470 #define INT_LEVEL 0x00008000 // Level-triggered (vs edge-)
7471 #define INT_ACTIVELOW 0x00002000 // Active low (vs high)
7472 #define INT_LOGICAL 0x00000800 // Destination is CPU id (vs APIC ID)
7473
7474 volatile struct ioapic *ioapic;
7475
7476 // IO APIC MMIO structure: write reg, then read or write data.
7477 struct ioapic {
7478     uint reg;
7479     uint pad[3];
7480     uint data;
7481 };
7482
7483 static uint
7484 ioapicread(int reg)
7485 {
7486     ioapic->reg = reg;
7487     return ioapic->data;
7488 }
7489
7490 static void
7491 ioapicwrite(int reg, uint data)
7492 {
7493     ioapic->reg = reg;
7494     ioapic->data = data;
7495 }
7496
7497
7498
7499

```

```

7500 void
7501 ioapicinit(void)
7502 {
7503     int i, id, maxintr;
7504
7505     if(!ismp)
7506         return;
7507
7508     ioapic = (volatile struct ioapic*)IOAPIC;
7509     maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
7510     id = ioapicread(REG_ID) >> 24;
7511     if(id != ioapicid)
7512         cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
7513
7514     // Mark all interrupts edge-triggered, active high, disabled,
7515     // and not routed to any CPUs.
7516     for(i = 0; i <= maxintr; i++){
7517         ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
7518         ioapicwrite(REG_TABLE+2*i+1, 0);
7519     }
7520 }
7521
7522 void
7523 ioapicenable(int irq, int cpunum)
7524 {
7525     if(!ismp)
7526         return;
7527
7528     // Mark interrupt edge-triggered, active high,
7529     // enabled, and routed to the given cpunum,
7530     // which happens to be that cpu's APIC ID.
7531     ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
7532     ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
7533 }
7534
7535
7536
7537
7538
7539
7540
7541
7542
7543
7544
7545
7546
7547
7548
7549

```

```

7550 // Intel 8259A programmable interrupt controllers.
7551
7552 #include "types.h"
7553 #include "x86.h"
7554 #include "traps.h"
7555
7556 // I/O Addresses of the two programmable interrupt controllers
7557 #define IO_PIC1        0x20    // Master (IRQs 0-7)
7558 #define IO_PIC2        0xA0    // Slave (IRQs 8-15)
7559
7560 #define IRQ_SLAVE      2        // IRQ at which slave connects to master
7561
7562 // Current IRQ mask.
7563 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
7564 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
7565
7566 static void
7567 picsetmask(ushort mask)
7568 {
7569     irqmask = mask;
7570     outb(IO_PIC1+1, mask);
7571     outb(IO_PIC2+1, mask >> 8);
7572 }
7573
7574 void
7575 picenable(int irq)
7576 {
7577     picsetmask(irqmask & ~(1<<irq));
7578 }
7579
7580 // Initialize the 8259A interrupt controllers.
7581 void
7582 picinit(void)
7583 {
7584     // mask all interrupts
7585     outb(IO_PIC1+1, 0xFF);
7586     outb(IO_PIC2+1, 0xFF);
7587
7588     // Set up master (8259A-1)
7589
7590     // ICW1: 0001g0hi
7591     //   g: 0 = edge triggering, 1 = level triggering
7592     //   h: 0 = cascaded PICs, 1 = master only
7593     //   i: 0 = no ICW4, 1 = ICW4 required
7594     outb(IO_PIC1, 0x11);
7595
7596     // ICW2: Vector offset
7597     outb(IO_PIC1+1, T_IRQ0);
7598
7599

```

```

7600 // ICW3: (master PIC) bit mask of IR lines connected to slaves
7601 //      (slave PIC) 3-bit # of slave's connection to master
7602 outb(IO_PIC1+1, 1<<IRQ_SLAVE);
7603
7604 // ICW4: 000nbmap
7605 //      n: 1 = special fully nested mode
7606 //      b: 1 = buffered mode
7607 //      m: 0 = slave PIC, 1 = master PIC
7608 //      (ignored when b is 0, as the master/slave role
7609 //      can be hardwired).
7610 //      a: 1 = Automatic EOI mode
7611 //      p: 0 = MCS-80/85 mode, 1 = intel x86 mode
7612 outb(IO_PIC1+1, 0x3);
7613
7614 // Set up slave (8259A-2)
7615 outb(IO_PIC2, 0x11);          // ICW1
7616 outb(IO_PIC2+1, T_IRQ0 + 8); // ICW2
7617 outb(IO_PIC2+1, IRQ_SLAVE);  // ICW3
7618 // NB Automatic EOI mode doesn't tend to work on the slave.
7619 // Linux source code says it's "to be investigated".
7620 outb(IO_PIC2+1, 0x3);        // ICW4
7621
7622 // OCW3: 0ef01prs
7623 //      ef: 0x = NOP, 10 = clear specific mask, 11 = set specific mask
7624 //      p: 0 = no polling, 1 = polling mode
7625 //      rs: 0x = NOP, 10 = read IRR, 11 = read ISR
7626 outb(IO_PIC1, 0x68);          // clear specific mask
7627 outb(IO_PIC1, 0x0a);          // read IRR by default
7628
7629 outb(IO_PIC2, 0x68);          // OCW3
7630 outb(IO_PIC2, 0x0a);          // OCW3
7631
7632 if(irqmask != 0xFFFF)
7633     picsetmask(irqmask);
7634 }
7635
7636
7637
7638
7639
7640
7641
7642
7643
7644
7645
7646
7647
7648
7649

```

```

7650 // Blank page.
7651
7652
7653
7654
7655
7656
7657
7658
7659
7660
7661
7662
7663
7664
7665
7666
7667
7668
7669
7670
7671
7672
7673
7674
7675
7676
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 |= E0ESC;
7871         return 0;
7872     } else if(data & 0x80){
7873         // Key released
7874         data = (shift & E0ESC ? data : data & 0x7F);
7875         shift &= ~(shiftcode[data] | E0ESC);
7876         return 0;
7877     } else if(shift & E0ESC){
7878         // Last character was an E0 escape; or with 0x80
7879         data |= 0x80;
7880         shift &= ~E0ESC;
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 kbdtintr(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 "fs.h"
7910 #include "file.h"
7911 #include "memlayout.h"
7912 #include "mmu.h"
7913 #include "proc.h"
7914 #include "x86.h"
7915
7916 static void consputc(int);
7917
7918 static int panicked = 0;
7919
7920 static struct {
7921   struct spinlock lock;
7922   int locking;
7923 } cons;
7924
7925 static void
7926 printint(int xx, int base, int sign)
7927 {
7928   static char digits[] = "0123456789abcdef";
7929   char buf[16];
7930   int i;
7931   uint x;
7932
7933   if(sign && (sign = xx < 0))
7934     x = -xx;
7935   else
7936     x = xx;
7937
7938   i = 0;
7939   do{
7940     buf[i++] = digits[x % base];
7941   }while((x /= base) != 0);
7942
7943   if(sign)
7944     buf[i++] = '-';
7945
7946   while(--i >= 0)
7947     consputc(buf[i]);
7948 }
7949
```

```

7950 // Print to the console. only understands %d, %x, %p, %s.
7951 void
7952 cprintf(char *fmt, ...)
7953 {
7954   int i, c, locking;
7955   uint *argp;
7956   char *s;
7957
7958   locking = cons.locking;
7959   if(locking)
7960     acquire(&cons.lock);
7961
7962   if (fmt == 0)
7963     panic("null fmt");
7964
7965   argp = (uint*)(void*)&fmt + 1;
7966   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
7967     if(c != '%'){
7968       consputc(c);
7969       continue;
7970     }
7971     c = fmt[++i] & 0xff;
7972     if(c == 0)
7973       break;
7974     switch(c){
7975     case 'd':
7976       printint(*argp++, 10, 1);
7977       break;
7978     case 'x':
7979     case 'p':
7980       printint(*argp++, 16, 0);
7981       break;
7982     case 's':
7983       if((s = (char*)*argp++) == 0)
7984         s = "(null)";
7985       for(; *s; s++)
7986         consputc(*s);
7987       break;
7988     case '%':
7989       consputc('%');
7990       break;
7991     default:
7992       // Print unknown % sequence to draw attention.
7993       consputc('%');
7994       consputc(c);
7995       break;
7996     }
7997   }
7998
7999
```

```

8000  if(locking)
8001      release(&cons.lock);
8002  }
8003
8004  void
8005  panic(char *s)
8006  {
8007      int i;
8008      uint pcs[10];
8009
8010      cli();
8011      cons.locking = 0;
8012      cprintf("cpu with apicid %d: panic: ", cpu->apicid);
8013      cprintf(s);
8014      cprintf("\n");
8015      getcallerpcs(&s, pcs);
8016      for(i=0; i<10; i++)
8017          cprintf(" %p", pcs[i]);
8018      panicked = 1; // freeze other CPU
8019      for(;;)
8020          ;
8021  }
8022
8023
8024
8025
8026
8027
8028
8029
8030
8031
8032
8033
8034
8035
8036
8037
8038
8039
8040
8041
8042
8043
8044
8045
8046
8047
8048
8049

```

```

8050  #define BACKSPACE 0x100
8051  #define CRTPORT 0x3d4
8052  static ushort *crt = (ushort*)P2V(0xb8000); // CGA memory
8053
8054  static void
8055  cgaputc(int c)
8056  {
8057      int pos;
8058
8059      // Cursor position: col + 80*row.
8060      outb(CRTPORT, 14);
8061      pos = inb(CRTPORT+1) << 8;
8062      outb(CRTPORT, 15);
8063      pos |= inb(CRTPORT+1);
8064
8065      if(c == '\n')
8066          pos += 80 - pos%80;
8067      else if(c == BACKSPACE){
8068          if(pos > 0) --pos;
8069      } else
8070          crt[pos++] = (c&0xff) | 0x0700; // black on white
8071
8072      if(pos < 0 || pos > 25*80)
8073          panic("pos under/overflow");
8074
8075      if((pos/80) >= 24){ // Scroll up.
8076          memmove(crt, crt+80, sizeof(crt[0])*23*80);
8077          pos -= 80;
8078          memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
8079      }
8080
8081      outb(CRTPORT, 14);
8082      outb(CRTPORT+1, pos>>8);
8083      outb(CRTPORT, 15);
8084      outb(CRTPORT+1, pos);
8085      crt[pos] = ' ' | 0x0700;
8086  }
8087
8088
8089
8090
8091
8092
8093
8094
8095
8096
8097
8098
8099

```



```

8100 void
8101 consputc(int c)
8102 {
8103     if(panicked){
8104         cli();
8105         for(;;)
8106             ;
8107     }
8108     if(c == BACKSPACE){
8109         uartputc('\b'); uartputc(' '); uartputc('\b');
8110     } else
8111         uartputc(c);
8112     cgaputc(c);
8113 }
8114
8115 #define INPUT_BUF 128
8116 struct {
8117     char buf[INPUT_BUF];
8118     uint r; // Read index
8119     uint w; // Write index
8120     uint e; // Edit index
8121 } input;
8122
8123 #define C(x) ((x)-'@') // Control-x
8124
8125 void
8126 consoleintr(int (*getc)(void))
8127 {
8128     int c, doprocdump = 0;
8129
8130     acquire(&cons.lock);
8131     while((c = getc()) >= 0){
8132         switch(c){
8133             case C('P'): // Process listing.
8134                 // procdump() locks cons.lock indirectly; invoke later
8135                 doprocdump = 1;
8136                 break;
8137             case C('U'): // Kill line.
8138                 while(input.e != input.w &&
8139                     input.buf[(input.e-1) % INPUT_BUF] != '\n'){
8140                     input.e--;
8141                     consputc(BACKSPACE);
8142                 }
8143                 break;
8144             case C('H'): case '\x7f': // Backspace
8145                 if(input.e != input.w){
8146                     input.e--;
8147                     consputc(BACKSPACE);
8148                 }
8149         }

```

```

8150         break;
8151     default:
8152         if(c != 0 && input.e-input.r < INPUT_BUF){
8153             c = (c == '\r') ? '\n' : c;
8154             input.buf[input.e++ % INPUT_BUF] = c;
8155             consputc(c);
8156             if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
8157                 input.w = input.e;
8158                 wakeup(&input.r);
8159             }
8160         }
8161     }
8162     break;
8163 }
8164 release(&cons.lock);
8165 if(doprocdump) {
8166     procdump(); // now call procdump() wo. cons.lock held
8167 }
8168 }
8169
8170 int
8171 consoleread(struct inode *ip, char *dst, int n)
8172 {
8173     uint target;
8174     int c;
8175
8176     iunlock(ip);
8177     target = n;
8178     acquire(&cons.lock);
8179     while(n > 0){
8180         while(input.r == input.w){
8181             if(proc->killed){
8182                 release(&cons.lock);
8183                 ilock(ip);
8184                 return -1;
8185             }
8186             sleep(&input.r, &cons.lock);
8187         }
8188         c = input.buf[input.r++ % INPUT_BUF];
8189         if(c == C('D')){ // EOF
8190             if(n < target){
8191                 // Save ^D for next time, to make sure
8192                 // caller gets a 0-byte result.
8193                 input.r--;
8194             }
8195             break;
8196         }
8197         *dst++ = c;
8198         --n;
8199         if(c == '\n')

```

```

8200     break;
8201 }
8202 release(&cons.lock);
8203 ilock(ip);
8204
8205 return target - n;
8206 }
8207
8208 int
8209 consolewrite(struct inode *ip, char *buf, int n)
8210 {
8211     int i;
8212
8213     iunlock(ip);
8214     acquire(&cons.lock);
8215     for(i = 0; i < n; i++)
8216         consputc(buf[i] & 0xff);
8217     release(&cons.lock);
8218     ilock(ip);
8219
8220     return n;
8221 }
8222
8223 void
8224 consoleinit(void)
8225 {
8226     initlock(&cons.lock, "console");
8227
8228     devsw[CONSOLE].write = consolewrite;
8229     devsw[CONSOLE].read = consoleread;
8230     cons.locking = 1;
8231
8232     picenable(IRQ_KBD);
8233     ioapicenable(IRQ_KBD, 0);
8234 }
8235
8236
8237
8238
8239
8240
8241
8242
8243
8244
8245
8246
8247
8248
8249

```

```

8250 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
8251 // Only used on uniprocessors;
8252 // SMP machines use the local APIC timer.
8253
8254 #include "types.h"
8255 #include "defs.h"
8256 #include "traps.h"
8257 #include "x86.h"
8258
8259 #define IO_TIMER1      0x040          // 8253 Timer #1
8260
8261 // Frequency of all three count-down timers;
8262 // (TIMER_FREQ/freq) is the appropriate count
8263 // to generate a frequency of freq Hz.
8264
8265 #define TIMER_FREQ      1193182
8266 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
8267
8268 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
8269 #define TIMER_SELO      0x00          // select counter 0
8270 #define TIMER_RATEGEN    0x04          // mode 2, rate generator
8271 #define TIMER_16BIT      0x30          // r/w counter 16 bits, LSB first
8272
8273 void
8274 timerinit(void)
8275 {
8276     // Interrupt 100 times/sec.
8277     outb(TIMER_MODE, TIMER_SELO | TIMER_RATEGEN | TIMER_16BIT);
8278     outb(IO_TIMER1, TIMER_DIV(100) % 256);
8279     outb(IO_TIMER1, TIMER_DIV(100) / 256);
8280     picenable(IRQ_TIMER);
8281 }
8282
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 "fs.h"
8308 #include "file.h"
8309 #include "mmu.h"
8310 #include "proc.h"
8311 #include "x86.h"
8312
8313 #define COM1    0x3f8
8314
8315 static int uart;    // is there a uart?
8316
8317 void
8318 uartinit(void)
8319 {
8320     char *p;
8321
8322     // Turn off the FIFO
8323     outb(COM1+2, 0);
8324
8325     // 9600 baud, 8 data bits, 1 stop bit, parity off.
8326     outb(COM1+3, 0x80);    // Unlock divisor
8327     outb(COM1+0, 115200/9600);
8328     outb(COM1+1, 0);
8329     outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
8330     outb(COM1+4, 0);
8331     outb(COM1+1, 0x01);    // Enable receive interrupts.
8332
8333     // If status is 0xFF, no serial port.
8334     if(inb(COM1+5) == 0xFF)
8335         return;
8336     uart = 1;
8337
8338     // Acknowledge pre-existing interrupt conditions;
8339     // enable interrupts.
8340     inb(COM1+2);
8341     inb(COM1+0);
8342     picenable(IRQ_COM1);
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    gdtdesc
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

```