

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

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14, 2000)). See also <http://pdos.csail.mit.edu/6.828/2012/v6.html>, which provides pointers to on-line resources for v6.

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

The following people have made contributions:

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

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

The code in the files that constitute xv6 is
 Copyright 2006-2012 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/6.828/2012/tools.html>. Then run "make TOOLPREFIX=i386-jos-elf-".

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

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

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

# basic headers	# system calls	# string operations
01 types.h	29 traps.h	61 string.c
01 param.h	29 vectors.pl	
02 memlayout.h	30 trapasm.S	# low-level hardware
02 defs.h	30 trap.c	63 mp.h
04 x86.h	32 syscall.h	64 mp.c
06 asm.h	32 syscall.c	66 lapic.c
07 mmu.h	34 sysproc.c	68 ioapic.c
09 elf.h		69 picirq.c
	# file system	70 kbd.h
# entering xv6	35 buf.h	71 kbd.c
10 entry.S	35 fcntl.h	72 console.c
11 entryother.S	36 stat.h	75 timer.c
12 main.c	36 fs.h	76 uart.c
	37 file.h	
# locks	38 ide.c	# user-level
14 spinlock.h	40 bio.c	77 initcode.S
14 spinlock.c	41 log.c	77 usys.S
	44 fs.c	78 init.c
# processes	52 file.c	78 sh.c
16 vm.c	54 sysfile.c	
20 proc.h	59 exec.c	# bootloader
21 proc.c		84 bootasm.S
27 swtch.S	# pipes	85 bootmain.c
27 kalloc.c	60 pipe.c	

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

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

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

acquire 1474 8468 8517
 0377 1474 1478 2210 2373 2408 BPB 3692
 2467 2524 2568 2583 2616 2629 3692 3695 4462 4464 4489
 2826 2843 3116 3472 3492 3907 bread 4102
 3965 4070 4130 4279 4310 4658 0262 4102 4226 4227 4239 4256
 4691 4711 4740 4758 4768 5229 4339 4431 4442 4463 4488 4613
 5254 5268 6063 6084 6105 7260 4634 4718 4826 4870 4919 4969
 7416 7458 7506 bre1se 4125
 allocproc 2205 0263 4125 4128 4230 4231 4246
 2205 2257 2310 4264 4342 4433 4445 4469 4474
 allocuvn 1853 4495 4619 4622 4643 4726 4832
 0422 1853 1867 2287 5943 5953 4876 4922 4973
 alltraps 3004 BSIZE 3661
 2959 2967 2980 2985 3003 3004 3661 3672 3686 3692 4207 4228
 ALT 7010 4340 4443 4919 4920 4921 4965
 7010 7038 7040 4969 4970 4971
 argfd 5419 buf 3500
 5419 5456 5471 5483 5494 5506 0250 0262 0263 0264 0306 0332
 argint 3295 2020 2023 2032 2034 3500 3504
 0395 3295 3308 3324 3432 3456 3505 3506 3811 3826 3829 3875
 3470 5424 5471 5483 5708 5776 3904 3954 3956 3959 4026 4030
 5777 5826 4034 4040 4053 4065 4068 4101
 argptr 3304 4104 4114 4125 4155 4226 4227
 0396 3304 5471 5483 5506 5857 4239 4240 4246 4256 4257 4263
 argstr 3321 4264 4325 4339 4418 4429 4440
 0397 3321 5518 5608 5708 5757 4457 4483 4606 4631 4705 4813
 5775 5806 5826 4859 4905 4955 7229 7240 7244
 BACK 7861 7247 7403 7424 7438 7468 7501
 7861 7974 8120 8389 7508 7984 7987 7988 7989 8003
 backcmd 7896 8114 8015 8016 8019 8020 8021 8025
 7896 7909 7975 8114 8116 8242 bwrite 4114
 8355 8390 0264 4114 4117 4229 4263 4341
 BACKSPACE 7350 bzero 4438
 7350 7367 7394 7426 7432 4438 4470
 balloc 4454 B_BUSY 3509
 4454 4476 4817 4825 4829 3509 3958 4076 4077 4088 4091
 BBLOCK 3695 4116 4127 4139
 3695 4463 4488 B_DIRTY 3511
 begin_trans 4277 3511 3887 3916 3921 3960 3978
 0333 4277 5283 5374 5523 5613 4088 4118 4345
 5711 5756 5774 B_VALID 3510
 bfree 4481 3510 3920 3960 3978 4107
 4481 4864 4874 4877 C 7031 7409
 bget 4066 7031 7079 7104 7105 7106 7107
 4066 4096 4106 7108 7110 7409 7419 7422 7429
 binit 4038 7440 7469
 0261 1231 4038 CAPSLOCK 7012
 bmap 4810 7012 7045 7186
 4810 4836 4919 4969 cgaputc 7355
 bootmain 8517 7355 7398

clearpteu 1929 0430 2018 5963 5974
 0431 1929 1935 5955 copyvm 1953
 cli 0557 0427 1953 1964 1966 2314
 0557 0559 1126 1560 7310 7389 cprintf 7252
 8412 0268 1224 1264 1867 2676 2680
 cmd 7865 2682 3140 3153 3158 3383 6519
 7865 7877 7886 7887 7892 7893 6539 6711 6862 7252 7312 7313
 7898 7902 7906 7915 7918 7923 7314 7317
 7931 7937 7941 7951 7975 7977 cpu 2054
 8052 8055 8057 8058 8059 8060 0309 1224 1264 1266 1278 1406
 8063 8064 8066 8068 8069 8070 1466 1487 1508 1546 1561 1562
 8071 8072 8073 8074 8075 8076 1570 1572 1618 1631 1637 1776
 8079 8080 8082 8084 8085 8086 1777 1778 1779 2054 2064 2068
 8087 8088 8089 8100 8101 8103 2079 2478 2509 2515 2516 2517
 8105 8106 8107 8108 8109 8110 3115 3140 3141 3153 3154 3158
 8113 8114 8116 8118 8119 8120 3160 6413 6414 6711 7312
 8121 8122 8212 8213 8214 8215 cpunum 6701
 8217 8221 8224 8230 8231 8234 0323 1288 1624 6701 6873 6882
 8237 8239 8242 8246 8248 8250 CR0_PE 0727
 8253 8255 8258 8260 8263 8264 0727 1135 1171 8443
 8275 8278 8281 8285 8300 8303 CR0_PG 0737
 8308 8312 8313 8316 8321 8322 0737 1050 1171
 8328 8337 8338 8344 8345 8351 CR0_WP 0733
 8352 8361 8364 8366 8372 8373 0733 1050 1171
 8378 8384 8390 8391 8394 CR4_PSE 0739
 COM1 7613 0739 1043 1164
 7613 7623 7626 7627 7628 7629 create 5657
 7630 7631 7634 7640 7641 7657 5657 5677 5690 5694 5712 5757
 7659 7667 7669 5778
 commit_trans 4301 CRTPORT 7351
 0334 4301 5285 5379 5528 5546 7351 7360 7361 7362 7363 7378
 5555 5645 5652 5713 5758 5762 7379 7380 7381
 5779 5783 CTL 7009
 CONSOLE 3787 7009 7035 7039 7185
 3787 7521 7522 deallocvm 1882
 consoleinit 7516 0423 1868 1882 1916 2290
 0267 1227 7516 DEVSPACE 0204
 consoleintr 7412 0204 1732 1745
 0269 7198 7412 7675 devsw 3780
 consoleread 7451 3780 3785 4908 4910 4958 4960
 7451 7522 5211 7521 7522
 consolewrite 7501 dinode 3676
 7501 7521 3676 3686 4607 4614 4632 4635
 consputc 7386 4706 4719
 7216 7247 7268 7286 7289 7293 dirent 3700
 7294 7386 7426 7432 7439 7508 3700 5014 5055 5564 5604
 context 2093 dirlink 5052
 0251 0374 2056 2093 2111 2238 0286 5021 5052 5067 5075 5539
 2239 2240 2241 2478 2516 2678 5689 5693 5694
 copyout 2018 dirlookup 5011

```

0287 5011 5017 5059 5174 5621
5667
DIRSIZ 3698
3698 3702 5005 5072 5128 5129
5191 5515 5605 5661
DPL_USER 0779
0779 1627 1628 2264 2265 3073
3168 3177
EOESC 7016
7016 7170 7174 7175 7177 7180
elfhdr 0955
0955 5915 8519 8524
ELF_MAGIC 0952
0952 5928 8530
ELF_PROG_LOAD 0986
0986 5939
entry 1040
0961 1036 1039 1040 2952 2953
5987 6321 8521 8545 8546
EOI 6614
6614 6684 6725
ERROR 6635
6635 6677
ESR 6617
6617 6680 6681
EXEC 7857
7857 7922 8059 8365
exec 5910
0273 5842 5910 7768 7829 7830
7926 7927
execcmd 7869 8053
7869 7910 7923 8053 8055 8321
8327 8328 8356 8366
exit 2354
0359 2354 2390 3105 3109 3169
3178 3417 7716 7719 7761 7826
7831 7916 7925 7935 7980 8028
8035
EXTMEM 0202
0202 0208 1729
fdalloc 5438
5438 5458 5726 5862
fetchint 3267
0398 3267 3297 5833
fetchstr 3279
0399 3279 3326 5839
file 3750
0252 0276 0277 0278 0280 0281
0282 0351 2114 3750 4420 5208
5214 5224 5227 5230 5251 5252
5264 5266 5302 5315 5352 5413
5419 5422 5438 5453 5467 5479
5492 5503 5705 5854 6006 6021
7210 7608 7878 7933 7934 8064
8072 8272
filealloc 5225
0276 5225 5726 6027
fileclose 5264
0277 2365 5264 5270 5497 5728
5865 5866 6054 6056
filedup 5252
0278 2329 5252 5256 5460
fileinit 5218
0279 1232 5218
fileread 5315
0280 5315 5330 5473
filestat 5302
0281 5302 5508
filewrite 5352
0282 5352 5384 5389 5485
FL_IF 0710
0710 1562 1568 2268 2513 6708
fork 2304
0360 2304 3411 7760 7823 7825
8043 8045
fork1 8039
7900 7942 7954 7961 7976 8024
8039
forkret 2533
2167 2241 2533
freerange 2801
2761 2784 2790 2801
freevm 1910
0424 1910 1915 1977 2421 5990
5995
gatedesc 0901
0523 0526 0901 3061
getcallerpcs 1526
0378 1488 1526 2678 7315
getcmd 7984
7984 8015
gettoken 8156
8156 8241 8245 8257 8270 8271
8307 8311 8333
growproc 2281
0361 2281 3459
havedisk1 3828
3828 3864 3962
holding 1544
0379 1477 1504 1544 2507

```

```

ialloc 4603
0288 4603 4624 5676 5677
IBLOCK 3689
3689 4613 4634 4718
ICRHI 6628
6628 6687 6756 6768
ICRLO 6618
6618 6688 6689 6757 6759 6769
ID 6611
6611 6647 6716
ideinit 3851
0304 1234 3851
ideintr 3902
0305 3124 3902
idelock 3825
3825 3855 3907 3909 3928 3965
3979 3982
iderw 3954
0306 3954 3959 3961 3963 4108
4119
idestart 3875
3829 3875 3878 3926 3975
idewait 3833
3833 3858 3880 3916
IDE_BSY 3813
3813 3837
IDE_CMD_READ 3818
3818 3891
IDE_CMD_WRITE 3819
3819 3888
IDE_DF 3815
3815 3839
IDE_DRDY 3814
3814 3837
IDE_ERR 3816
3816 3839
idtinit 3079
0406 1265 3079
idup 4689
0289 2330 4689 5161
iget 4654
4573 4620 4654 4674 5029 5159
iinit 4568
0290 1233 4568
ilock 4703
0291 4703 4709 4729 5164 5305
5324 5375 5525 5538 5551 5615
5623 5665 5669 5679 5719 5808
5922 7463 7483 7510
inb 0453
0453 3837 3863 6554 7164 7167
7361 7363 7634 7640 7641 7657
7667 7669 8423 8431 8554
initlock 1462
0380 1462 2175 2782 3075 3855
4042 4211 4570 5220 6035 7518
7519
initlog 4205
0331 2544 4205 4208
initvm 1803
0425 1803 1808 2261
inode 3762
0253 0286 0287 0288 0289 0291
0292 0293 0294 0295 0297 0298
0299 0300 0301 0426 1818 2115
3756 3762 3781 3782 4423 4564
4573 4602 4629 4653 4656 4662
4688 4689 4703 4735 4756 4778
4810 4856 4887 4902 4952 5010
5011 5052 5056 5153 5156 5188
5195 5516 5561 5603 5656 5660
5706 5754 5769 5804 5916 7451
7501
INPUT_BUF 7400
7400 7403 7424 7436 7438 7440
7468
insl 0462
0462 0464 3917 8573
install_trans 4221
4221 4271 4305
INT_DISABLED 6819
6819 6867
IOAPIC 6808
6808 6858
ioapic 6827
6507 6529 6530 6824 6827 6836
6837 6843 6844 6858
ioapicenable 6873
0309 3857 6873 7526 7643
ioapicid 6417
0310 6417 6530 6547 6861 6862
ioapicinit 6851
0311 1226 6851 6862
ioapicread 6834
6834 6859 6860
ioapicwrite 6841
6841 6867 6868 6881 6882
IO_PIC1 6907
6907 6920 6935 6944 6947 6952
6962 6976 6977

```

IO_PIC2 6908	kalloc 2838
6908 6921 6936 6965 6966 6967	0314 1294 1663 1742 1809 1865
6970 6979 6980	1968 2223 2838 6029
IO_RTC 6735	KBDATAP 7004
6735 6748 6749	7004 7167
IO_TIMER1 7559	kbdgetc 7156
7559 7568 7578 7579	7156 7198
IPB 3686	kbdintr 7196
3686 3689 3695 4614 4635 4719	0320 3131 7196
iput 4756	KBSTATP 7002
0292 2370 4756 4762 4781 5060	7002 7164
5182 5284 5544 5814	KBS_DIB 7003
IRQ_COM1 2933	7003 7165
2933 3134 7642 7643	KERNBASE 0207
IRQ_ERROR 2935	0207 0208 0212 0213 0217 0218
2935 6677	0220 0221 1315 1533 1729 1858
IRQ_IDE 2934	1916
2934 3123 3127 3856 3857	KERNLINK 0208
IRQ_KBD 2932	0208 1730
2932 3130 7525 7526	KEY_DEL 7028
IRQ_SLAVE 6910	7028 7069 7091 7115
6910 6914 6952 6967	KEY_DN 7022
IRQ_SPURIOUS 2936	7022 7065 7087 7111
2936 3139 6657	KEY_END 7020
IRQ_TIMER 2931	7020 7068 7090 7114
2931 3114 3173 6664 7580	KEY_HOME 7019
isdiempty 5561	7019 7068 7090 7114
5561 5568 5627	KEY_INS 7027
ismp 6415	7027 7069 7091 7115
0337 1235 6415 6512 6520 6540	KEY_LF 7023
6543 6855 6875	7023 7067 7089 7113
itrunc 4856	KEY_PGDN 7026
4423 4765 4856	7026 7066 7088 7112
iunlock 4735	KEY_PGUP 7025
0293 4735 4738 4780 5171 5307	7025 7066 7088 7112
5327 5378 5534 5732 5813 7456	KEY_RT 7024
7505	7024 7067 7089 7113
iunlockput 4778	KEY_UP 7021
0294 4778 5166 5175 5178 5527	7021 7065 7087 7111
5540 5543 5554 5628 5639 5643	kfree 2815
5651 5668 5672 5696 5721 5729	0315 1898 1900 1920 1923 2315
5761 5782 5810 5948 5997	2419 2806 2815 2820 6052 6073
iupdate 4629	kill 2625
0295 4629 4767 4882 4978 5533	0362 2625 3159 3434 7767
5553 5637 5642 5683 5687	kinit1 2780
I_BUSY 3775	0316 1219 2780
3775 4712 4714 4737 4741 4761	kinit2 2788
4763	0317 1238 2788
I_INVALID 3776	KSTACKSIZE 0151
3776 4717 4727 4759	0151 1054 1063 1295 1779 2227

kvmalloc 1757	mappages 1679
0418 1220 1757	1679 1748 1811 1872 1971
lapiceoi 6722	MAXARG 0159
0325 3121 3125 3132 3136 3142	0159 5822 5914 5960
6722	MAXARGS 7863
lapicinit 6651	7863 7871 7872 8340
0326 1222 1256 6651	MAXFILE 3673
lapicstartap 6740	3673 4965
0327 1299 6740	memcmp 6165
lapicw 6644	0386 6165 6445 6488
6644 6657 6663 6664 6665 6668	memmove 6181
6669 6674 6677 6680 6681 6684	0387 1285 1812 1970 2032 4228
6687 6688 6693 6725 6756 6757	4340 4432 4641 4725 4921 4971
6759 6768 6769	5129 5131 6181 6204 7373
lcr3 0590	memset 6154
0590 1768 1783	0388 1666 1744 1810 1871 2240
lgdt 0512	2263 2823 4443 4616 5632 5829
0512 0520 1133 1633 8441	6154 7375 7987 8058 8069 8085
lidt 0526	8106 8119
0526 0534 3081	microdelay 6731
LINT0 6633	0328 6731 6758 6760 6770 7658
6633 6668	min 4422
LINT1 6634	4422 4920 4970
6634 6669	mp 6302
LIST 7860	6302 6408 6437 6444 6445 6446
7860 7940 8107 8383	6455 6460 6464 6465 6468 6469
listcmd 7890 8101	6480 6483 6485 6487 6494 6504
7890 7911 7941 8101 8103 8246	6510 6550
8357 8384	mpbcpu 6420
loadgs 0551	0338 6420
0551 1634	MPBUS 6352
loaduvm 1818	6352 6533
0426 1818 1824 1827 5945	mpconf 6313
log 4190 4200	6313 6479 6482 6487 6505
4190 4200 4211 4213 4214 4215	mpconfig 6480
4225 4226 4227 4239 4242 4243	6480 6510
4244 4256 4259 4260 4261 4272	mpenter 1252
4279 4280 4281 4283 4284 4303	1252 1296
4306 4310 4311 4312 4313 4329	mpinit 6501
4331 4334 4335 4338 4339 4343	0339 1221 6501 6519 6539
4344	MPIOAPIC 6353
logheader 4185	6353 6528
4185 4196 4207 4208 4240 4257	mpioapic 6339
LOGSIZE 0160	6339 6507 6529 6531
0160 4187 4329 5367	MPIOINTR 6354
log_write 4325	6354 6534
0332 4325 4444 4468 4494 4618	MPLINTR 6355
4642 4830 4972	6355 6535
ltr 0538	mpmain 1262
0538 0540 1780	1209 1241 1257 1262

MPPROC 6351
 6351 6516
 mpproc 6328
 6328 6506 6517 6526
 mpsearch 6456
 6456 6485
 mpsearch1 6438
 6438 6464 6468 6471
 multiboot_header 1025
 1024 1025
 namecmp 5003
 0296 5003 5024 5618
 namei 5189
 0297 2273 5189 5520 5717 5806
 5920
 nameiparent 5196
 0298 5154 5169 5181 5196 5536
 5610 5663
 namex 5154
 5154 5192 5198
 NBUF 0155
 0155 4030 4053
 NCPU 0152
 0152 2068 6413
 ncpu 6416
 1224 1287 2069 3857 6416 6518
 6519 6523 6524 6525 6545
 NDEV 0157
 0157 4908 4958 5211
 NDIRECT 3671
 3671 3673 3682 3773 4815 4820
 4824 4825 4862 4869 4870 4877
 4878
 NELEM 0434
 0434 1747 2672 3380 5831
 nextpid 2166
 2166 2219
 NFILE 0154
 0154 5214 5230
 NINDIRECT 3672
 3672 3673 4822 4872
 NINODE 0156
 0156 4564 4662
 NO 7006
 7006 7052 7055 7057 7058 7059
 7060 7062 7074 7077 7079 7080
 7081 7082 7084 7102 7103 7105
 7106 7107 7108
 NOFILE 0153
 0153 2114 2327 2363 5426 5442

NPENTRIES 0821
 0821 1311 1917
 NPROC 0150
 0150 2161 2211 2379 2412 2468
 2607 2630 2669
 NPENTRIES 0822
 0822 1894
 NSEGS 2051
 1611 2051 2058
 nulterminate 8352
 8215 8230 8352 8373 8379 8380
 8385 8386 8391
 NUMLOCK 7013
 7013 7046
 outb 0471
 0471 3861 3870 3881 3882 3883
 3884 3885 3886 3888 3891 6553
 6554 6748 6749 6920 6921 6935
 6936 6944 6947 6952 6962 6965
 6966 6967 6970 6976 6977 6979
 6980 7360 7362 7378 7379 7380
 7381 7577 7578 7579 7623 7626
 7627 7628 7629 7630 7631 7659
 8428 8436 8564 8565 8566 8567
 8568 8569
 outsl 0483
 0483 0485 3889
 outw 0477
 0477 1181 1183 8474 8476
 O_CREATE 3553
 3553 5710 8278 8281
 O_RDONLY 3550
 3550 5720 8275
 O_RDWR 3552
 3552 5738 7814 7816 8007
 O_WRONLY 3551
 3551 5737 5738 8278 8281
 P2V 0218
 0218 1219 1238 6462 6750 7352
 panic 7305 8032
 0270 1478 1505 1569 1571 1690
 1746 1782 1808 1824 1827 1898
 1915 1935 1964 1966 2260 2360
 2390 2508 2510 2512 2514 2556
 2559 2820 3155 3878 3959 3961
 3963 4096 4117 4128 4208 4330
 4332 4476 4492 4624 4674 4709
 4729 4738 4762 4836 5017 5021
 5067 5075 5256 5270 5330 5384
 5389 5568 5626 5634 5677 5690

 5694 7263 7305 7312 7901 7920
 7953 8032 8045 8228 8272 8306
 8310 8336 8341
 panicked 7218
 7218 7318 7388
 parseblock 8301
 8301 8306 8325
 parsecmd 8218
 7902 8025 8218
 parseexec 8317
 8214 8255 8317
 parseline 8235
 8212 8224 8235 8246 8308
 parsepipe 8251
 8213 8239 8251 8258
 parseredirs 8264
 8264 8312 8331 8342
 PCINT 6632
 6632 6674
 pde_t 0103
 0103 0420 0421 0422 0423 0424
 0425 0426 0427 0430 0431 1210
 1270 1311 1610 1654 1656 1679
 1736 1739 1742 1803 1818 1853
 1882 1910 1929 1952 1953 1955
 2002 2018 2105 5918
 PDX 0812
 0812 1659
 PDXSHIFT 0827
 0812 0818 0827 1315
 peek 8201
 8201 8225 8240 8244 8256 8269
 8305 8309 8324 8332
 PGROUNDOWN 0830
 0830 1684 1685 2025
 PGROUNDUP 0829
 0829 1863 1890 2804 5952
 PGSIZE 0823
 0823 0829 0830 1310 1666 1694
 1695 1744 1807 1810 1811 1823
 1825 1829 1832 1864 1871 1872
 1891 1894 1962 1970 1971 2029
 2035 2262 2269 2805 2819 2823
 5953 5955
 PHYSTOP 0203
 0203 1238 1731 1745 1746 2819
 picenable 6925
 0343 3856 6925 7525 7580 7642
 picinit 6932
 0344 1225 6932

picsetmask 6917
 6917 6927 6983
 pinit 2173
 0363 1229 2173
 PIPE 7859
 7859 7950 8086 8377
 pipe 6011
 0254 0352 0353 0354 3755 5281
 5322 5359 6011 6023 6029 6035
 6039 6043 6061 6080 6101 7763
 7952 7953
 pipealloc 6021
 0351 5859 6021
 pipeclose 6061
 0352 5281 6061
 pipecmd 7884 8080
 7884 7912 7951 8080 8082 8258
 8358 8378
 piperead 6101
 0353 5322 6101
 PIPESIZE 6009
 6009 6013 6086 6094 6116
 pipewrite 6080
 0354 5359 6080
 popcli 1566
 0383 1521 1566 1569 1571 1784
 printint 7226
 7226 7276 7280
 proc 2103
 0255 0358 0428 1205 1458 1606
 1638 1773 1779 2065 2080 2103
 2109 2156 2161 2164 2204 2207
 2211 2254 2285 2287 2290 2293
 2294 2307 2314 2320 2321 2322
 2328 2329 2330 2334 2356 2359
 2364 2365 2366 2370 2371 2376
 2379 2380 2388 2405 2412 2413
 2433 2439 2460 2468 2475 2478
 2483 2511 2516 2525 2555 2573
 2574 2578 2605 2607 2627 2630
 2665 2669 3055 3104 3106 3108
 3151 3159 3160 3162 3168 3173
 3177 3255 3269 3283 3286 3297
 3310 3379 3381 3384 3385 3406
 3440 3458 3475 3807 4416 5161
 5411 5426 5443 5444 5496 5814
 5815 5864 5904 5981 5984 5985
 5986 5987 5988 5989 6004 6087
 6107 6411 6506 6517 6518 6519
 6522 7213 7461 7610

```

procdump 2654          6810 6860
0364 2654 7420        REG_TABLE 6812
proghdr 0974          6812 6867 6868 6881 6882
0974 5917 8520 8534    REG_VER 6811
PTE_ADDR 0844         6811 6859
0844 1661 1828 1896 1919 1967 release 1502
2011                  0381 1502 1505 2214 2220 2427
PTE_P 0833           2434 2485 2527 2537 2569 2582
0833 1313 1315 1660 1670 1689 2618 2636 2640 2831 2848 3119
1691 1895 1918 1965 2007 3476 3481 3494 3909 3928 3982
PTE_PS 0840          4078 4092 4142 4284 4313 4665
0840 1313 1315        4681 4693 4715 4743 4764 4773
pte_t 0847           5233 5237 5258 5272 5278 6072
0847 1653 1657 1661 1663 1682 6075 6088 6097 6108 6119 7301
1821 1884 1931 1956 2004 7448 7462 7482 7509
PTE_U 0835          ROOTDEV 0158
0835 1670 1811 1872 1936 1971 0158 4212 4215 5159
2009                ROOTINO 3660
PTE_W 0834          3660 5159
0834 1313 1315 1670 1729 1731 run 2764
1732 1811 1872 1971    2661 2764 2765 2771 2817 2827
PTX 0815            2840
0815 1672            runcmd 7906
PTXSHIFT 0826        7906 7920 7937 7943 7945 7959
0815 0818 0826        7966 7977 8025
pushcli 1555         RUNNING 2100
0382 1476 1555 1775    2100 2477 2511 2661 3173
rcr2 0582            safestrcpy 6232
0582 3154 3161        0389 2272 2334 5981 6232
readeflags 0544      sched 2503
0544 1559 1568 2513 6708 0366 2389 2503 2508 2510 2512
readi 4902           2514 2526 2575
0299 1833 4902 5020 5066 5325 scheduler 2458
5567 5568 5926 5937    0365 1267 2056 2458 2478 2516
readsb 4427          SCROLLLOCK 7014
0285 4212 4427 4461 4487 4610 7014 7047
readsect 8560        SECTSIZE 8512
8560 8595            8512 8573 8586 8589 8594
readseg 8579         SEG 0769
8514 8527 8538 8579    0769 1625 1626 1627 1628 1631
read_head 4237       SEG16 0773
4237 4270            0773 1776
recover_from_log 4268 segdesc 0752
4202 4216 4268        0509 0512 0752 0769 0773 1611
REDIR 7858           2058
7858 7930 8070 8371    seginit 1616
redircmd 7875 8064    0417 1223 1255 1616
7875 7913 7931 8064 8066 8275 SEG_ASM 0660
8278 8281 8359 8372    0660 1190 1191 8484 8485
REG_ID 6810          REG_KCODE 0741

```

```

0741 1150 1625 3072 3073 8453 sti 0563
SEG_KCPU 0743        0563 0565 1573 2464
0743 1631 1634 3016    stosb 0492
SEG_KDATA 0742       0492 0494 6160 8540
0742 1154 1626 1778 3013 8458 stosl 0501
SEG_NULLASM 0654     0501 0503 6158
0654 1189 8483        strlen 6251
SEG_TSS 0746         0390 5962 5963 6251 8019 8223
0746 1776 1777 1780    strncmp 6208
SEG_UCODE 0744       0391 5005 6208
0744 1627 2264        strncpy 6218
SEG_UDATA 0745       0392 5072 6218
0745 1628 2265        STS_IG32 0800
SETGATE 0921         0800 0927
0921 3072 3073        STS_T32A 0797
setupkvm 1737        0797 1776
0420 1737 1759 1960 2259 5931 STS_TG32 0801
SHIFT 7008           0801 0927
7008 7036 7037 7185    sum 6426
skipelem 5115        6426 6428 6430 6432 6433 6445
5115 5163            6492
sleep 2553           superbloc 3664
0367 2439 2553 2556 2559 2659 0258 0285 3664 4210 4427 4458
3479 3979 4081 4281 4713 6092 4484 4608
6111 7466 7779        SVR 6615
spinlock 1401        6615 6657
0256 0367 0377 0379 0380 0381 switchkvm 1766
0409 1401 1459 1462 1474 1502 0429 1254 1760 1766 2479
1544 2157 2160 2553 2759 2769 switchvum 1773
3058 3063 3810 3825 4025 4029 0428 1773 1782 2294 2476 5989
4153 4191 4417 4563 5209 5213 swtch 2708
6007 6012 7208 7221 7402 7606 0374 2478 2516 2707 2708
start 1125 7708 8411 SYSCALL 7753 7760 7761 7762 7763 77
1124 1125 1167 1175 1177 4192 7760 7761 7762 7763 7764 7765
4213 4226 4239 4256 4339 7707 7766 7767 7768 7769 7770 7771
7708 8410 8411 8467    7772 7773 7774 7775 7776 7777
startothers 1274     7778 7779 7780
1208 1237 1274        syscall 3375
stat 3604            0400 3107 3257 3375
0257 0281 0300 3604 4414 4887 SYS_chdir 3209
5302 5409 5504 7803    3209 3359
stat1 4887           sys_chdir 5801
0300 4887 5306        3329 3359 5801
STA_R 0669 0786      SYS_close 3221
0669 0786 1190 1625 1627 8484 3221 3371
STA_W 0668 0785      sys_close 5489
0668 0785 1191 1626 1628 1631 3330 3371 5489
8485                SYS_dup 3210
STA_X 0665 0782      3210 3360
0665 0782 1190 1625 1627 8484 sys_dup 5451

```

```

3331 3360 5451
SYS_exec 3207
3207 3357 7712
sys_exec 5820
3332 3357 5820
SYS_exit 3202
3202 3352 7717
sys_exit 3415
3333 3352 3415
SYS_fork 3201
3201 3351
sys_fork 3409
3334 3351 3409
SYS_fstat 3208
3208 3358
sys_fstat 5501
3335 3358 5501
SYS_getpid 3211
3211 3361
sys_getpid 3438
3336 3361 3438
SYS_kill 3206
3206 3356
sys_kill 3428
3337 3356 3428
SYS_link 3219
3219 3369
sys_link 5513
3338 3369 5513
SYS_mkdir 3220
3220 3370
sys_mkdir 5751
3339 3370 5751
SYS_mknod 3217
3217 3367
sys_mknod 5767
3340 3367 5767
SYS_open 3215
3215 3365
sys_open 5701
3341 3365 5701
SYS_pipe 3204
3204 3354
sys_pipe 5851
3342 3354 5851
SYS_read 3205
3205 3355
sys_read 5465
3343 3355 5465
SYS_sbrk 3212
3212 3362
sys_sbrk 3451
3344 3362 3451
SYS_sleep 3213
3213 3363
sys_sleep 3465
3345 3363 3465
SYS_unlink 3218
3218 3368
sys_unlink 5601
3346 3368 5601
SYS_uptime 3214
3214 3364
sys_uptime 3488
3349 3364 3488
SYS_wait 3203
3203 3353
sys_wait 3422
3347 3353 3422
SYS_write 3216
3216 3366
sys_write 5477
3348 3366 5477
taskstate 0851
0851 2057
TDCR 6639
6639 6663
ticks 3064
0407 3064 3117 3118 3473 3474
3479 3493
tickslock 3063
0409 3063 3075 3116 3119 3472
3476 3479 3481 3492 3494
TICR 6637
6637 6665
TIMER 6629
6629 6664
timerinit 7574
0403 1236 7574
TIMER_16BIT 7571
7571 7577
TIMER_DIV 7566
7566 7578 7579
TIMER_FREQ 7565
7565 7566
TIMER_MODE 7568
7568 7577
TIMER_RATEGEN 7570
7570 7577
TIMER_SELO 7569

```

```

7569 7577
TPR 6613
6613 6693
trap 3101
2952 2954 3022 3101 3153 3155
3158
trapframe 0602
0602 2110 2231 3101
trapret 3027
2168 2236 3026 3027
tvinit 3067
0408 1230 3067
T_DEV 3602
3602 4907 4957 5778
T_DIR 3600
3600 5016 5165 5526 5627 5635
5685 5720 5757 5809
T_FILE 3601
3601 5670 5712
T_IRQ0 2929
2929 3114 3123 3127 3130 3134
3138 3139 3173 6657 6664 6677
6867 6881 6947 6966
T_SYSCALL 2926
2926 3073 3103 7713 7718 7757
uart 7615
7615 7636 7655 7665
uartgetc 7663
7663 7675
uartinit 7618
0412 1228 7618
uartintr 7673
0413 3135 7673
uartputc 7651
0414 7395 7397 7647 7651
userinit 2252
0368 1239 2252 2260
uva2ka 2002
0421 2002 2026
V2P 0217
0217 1730 1731
V2P_WO 0220
0220 1036 1046
VER 6612
6612 6673
wait 2403
0369 2403 3424 7762 7833 7944
7970 7971 8026
waitdisk 8551
8551 8563 8572
wakeup 2614
0370 2614 3118 3922 4140 4312
4742 4770 6066 6069 6091 6096
6118 7442
wakeup1 2603
2170 2376 2383 2603 2617
walkpgdir 1654
1654 1687 1826 1892 1933 1963
2006
writei 4952
0301 4952 5074 5376 5633 5634
write_head 4254
4254 4273 4304 4307
xchg 0569
0569 1266 1483 1519
yield 2522
0371 2522 3174
__attribute__ 1310
0270 0365 1209 1310

```

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

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



```

0200 // Memory layout
0201
0202 #define EXTMEM 0x100000          // Start of extended memory
0203 #define PHYSTOP 0xE000000      // Top physical memory
0204 #define DEVSPACE 0xFE000000    // Other devices are at high addresses
0205
0206 // Key addresses for address space layout (see kmap in vm.c for layout)
0207 #define KERNBASE 0x80000000     // First kernel virtual address
0208 #define KERNLINK (KERNBASE+EXTMEM) // Address where kernel is linked
0209
0210 #ifndef __ASSEMBLER__
0211
0212 static inline uint v2p(void *a) { return ((uint) (a)) - KERNBASE; }
0213 static inline void *p2v(uint a) { return (void *) ((a) + KERNBASE); }
0214
0215 #endif
0216
0217 #define V2P(a) (((uint) (a)) - KERNBASE)
0218 #define P2V(a) (((void *) (a)) + KERNBASE)
0219
0220 #define V2P_WO(x) ((x) - KERNBASE) // same as V2P, but without casts
0221 #define P2V_WO(x) ((x) + KERNBASE) // same as V2P, but without casts
0222
0223
0224
0225
0226
0227
0228
0229
0230
0231
0232
0233
0234
0235
0236
0237
0238
0239
0240
0241
0242
0243
0244
0245
0246
0247
0248
0249

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

0800 #define STS_IC32    0xE    // 32-bit Interrupt Gate
0801 #define STS_TG32    0xF    // 32-bit Trap Gate
0802
0803 // A virtual address 'la' has a three-part structure as follows:
0804 //
0805 // +-----10-----+-----10-----+-----12-----+
0806 // | Page Directory | Page Table | Offset within Page |
0807 // |      Index      |      Index      |                |
0808 // +-----+-----+-----+
0809 // \--- PDX(va) ---/ \--- PTX(va) ---/
0810
0811 // page directory index
0812 #define PDX(va)      (((uint)(va) >> PDXSHIFT) & 0x3FF)
0813
0814 // page table index
0815 #define PTX(va)      (((uint)(va) >> PTXSHIFT) & 0x3FF)
0816
0817 // construct virtual address from indexes and offset
0818 #define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
0819
0820 // Page directory and page table constants.
0821 #define NPDENTRIES    1024    // # directory entries per page directory
0822 #define NPTENTRIES    1024    // # PTEs per page table
0823 #define PGSIZE        4096    // bytes mapped by a page
0824
0825 #define PGSHIFT        12      // log2(PGSIZE)
0826 #define PTXSHIFT      12      // offset of PTX in a linear address
0827 #define PDXSHIFT      22      // offset of PDX in a linear address
0828
0829 #define PGROUNDUP(sz) (((sz)+PGSIZE-1) & ~(PGSIZE-1))
0830 #define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
0831
0832 // Page table/directory entry flags.
0833 #define PTE_P          0x001    // Present
0834 #define PTE_W          0x002    // Writeable
0835 #define PTE_U          0x004    // User
0836 #define PTE_PWT        0x008    // Write-Through
0837 #define PTE_PCD        0x010    // Cache-Disable
0838 #define PTE_A          0x020    // Accessed
0839 #define PTE_D          0x040    // Dirty
0840 #define PTE_PS         0x080    // Page Size
0841 #define PTE_MBZ        0x180    // Bits must be zero
0842
0843 // Address in page table or page directory entry
0844 #define PTE_ADDR(pte)  ((uint)(pte) & ~0xFFF)
0845
0846 #ifndef __ASSEMBLER__
0847 typedef uint pte_t;
0848
0849

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

1550 // Pushcli/popcli are like cli/sti except that they are matched:
1551 // it takes two popcli to undo two pushcli. Also, if interrupts
1552 // are off, then pushcli, popcli leaves them off.
1553
1554 void
1555 pushcli(void)
1556 {
1557     int eflags;
1558
1559     eflags = readeflags();
1560     cli();
1561     if(cpu->ncli++ == 0)
1562         cpu->intena = eflags & FL_IF;
1563 }
1564
1565 void
1566 popcli(void)
1567 {
1568     if(readeflags() & FL_IF)
1569         panic("popcli - interruptible");
1570     if(--cpu->ncli < 0)
1571         panic("popcli");
1572     if(cpu->ncli == 0 && cpu->intena)
1573         sti();
1574 }
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599

```

```

1600 #include "param.h"
1601 #include "types.h"
1602 #include "defs.h"
1603 #include "x86.h"
1604 #include "memlayout.h"
1605 #include "mmu.h"
1606 #include "proc.h"
1607 #include "elf.h"
1608
1609 extern char data[]; // defined by kernel.ld
1610 pde_t *kpgdir; // for use in scheduler()
1611 struct segdesc gdt[NSEGS];
1612
1613 // Set up CPU's kernel segment descriptors.
1614 // Run once on entry on each CPU.
1615 void
1616 seginit(void)
1617 {
1618     struct cpu *c;
1619
1620     // Map "logical" addresses to virtual addresses using identity map.
1621     // Cannot share a CODE descriptor for both kernel and user
1622     // because it would have to have DPL_USR, but the CPU forbids
1623     // an interrupt from CPL=0 to DPL=3.
1624     c = &cpu[cpunum()];
1625     c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
1626     c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1627     c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
1628     c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1629
1630     // Map cpu, and curproc
1631     c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
1632
1633     lgdt(c->gdt, sizeof(c->gdt));
1634     loadgs(SEG_KCPU << 3);
1635
1636     // Initialize cpu-local storage.
1637     cpu = c;
1638     proc = 0;
1639 }
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649

```

```

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

```

```

1700 // There is one page table per process, plus one that's used when
1701 // a CPU is not running any process (kpgdir). The kernel uses the
1702 // current process's page table during system calls and interrupts;
1703 // page protection bits prevent user code from using the kernel's
1704 // mappings.
1705 //
1706 // setupkvm() and exec() set up every page table like this:
1707 //
1708 // 0..KERNBASE: user memory (text+data+stack+heap), mapped to
1709 // phys memory allocated by the kernel
1710 // KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
1711 // KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
1712 // for the kernel's instructions and r/o data
1713 // data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
1714 // rw data + free physical memory
1715 // 0xfe000000..0: mapped direct (devices such as ioapic)
1716 //
1717 // The kernel allocates physical memory for its heap and for user memory
1718 // between V2P(end) and the end of physical memory (PHYSTOP)
1719 // (directly addressable from end..P2V(PHYSTOP)).
1720 //
1721 // This table defines the kernel's mappings, which are present in
1722 // every process's page table.
1723 static struct kmap {
1724     void *virt;
1725     uint phys_start;
1726     uint phys_end;
1727     int perm;
1728 } kmap[] = {
1729     { (void*)KERNBASE, 0,          EXTMEM,    PTE_W}, // I/O space
1730     { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},    // kern text+rodata
1731     { (void*)data,     V2P(data),   PHYSTOP,   PTE_W}, // kern data+memory
1732     { (void*)DEVSPACE, DEVSPACE,    0,        PTE_W}, // more devices
1733 };
1734
1735 // Set up kernel part of a page table.
1736 pde_t*
1737 setupkvm(void)
1738 {
1739     pde_t *pgdir;
1740     struct kmap *k;
1741
1742     if((pgdir = (pde_t*)kalloc()) == 0)
1743         return 0;
1744     memset(pgdir, 0, PGSIZE);
1745     if (p2v(PHYSTOP) > (void*)DEVSPACE)
1746         panic("PHYSTOP too high");
1747     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1748         if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1749             (uint)k->phys_start, k->perm) < 0)

```

```

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

```



```

1800 // Load the initcode into address 0 of pgdir.
1801 // sz must be less than a page.
1802 void
1803 inituvm(pde_t *pgdir, char *init, uint sz)
1804 {
1805     char *mem;
1806
1807     if(sz >= PGSIZE)
1808         panic("inituvm: more than a page");
1809     mem = kalloc();
1810     memset(mem, 0, PGSIZE);
1811     mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
1812     memmove(mem, init, sz);
1813 }
1814
1815 // Load a program segment into pgdir. addr must be page-aligned
1816 // and the pages from addr to addr+sz must already be mapped.
1817 int
1818 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
1819 {
1820     uint i, pa, n;
1821     pte_t *pte;
1822
1823     if((uint) addr % PGSIZE != 0)
1824         panic("loaduvm: addr must be page aligned");
1825     for(i = 0; i < sz; i += PGSIZE){
1826         if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
1827             panic("loaduvm: address should exist");
1828         pa = PTE_ADDR(*pte);
1829         if(sz - i < PGSIZE)
1830             n = sz - i;
1831         else
1832             n = PGSIZE;
1833         if(readi(ip, p2v(pa), offset+i, n) != n)
1834             return -1;
1835     }
1836     return 0;
1837 }
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849

```

```

1850 // Allocate page tables and physical memory to grow process from oldsz to
1851 // newsz, which need not be page aligned. Returns new size or 0 on error.
1852 int
1853 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1854 {
1855     char *mem;
1856     uint a;
1857
1858     if(newsz >= KERNBASE)
1859         return 0;
1860     if(newsz < oldsz)
1861         return oldsz;
1862
1863     a = PGROUNDUP(oldsz);
1864     for(; a < newsz; a += PGSIZE){
1865         mem = kalloc();
1866         if(mem == 0){
1867             cprintf("allocuvm out of memory\n");
1868             deallocuvm(pgdir, newsz, oldsz);
1869             return 0;
1870         }
1871         memset(mem, 0, PGSIZE);
1872         mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U);
1873     }
1874     return newsz;
1875 }
1876
1877 // Deallocate user pages to bring the process size from oldsz to
1878 // newsz. oldsz and newsz need not be page-aligned, nor does newsz
1879 // need to be less than oldsz. oldsz can be larger than the actual
1880 // process size. Returns the new process size.
1881 int
1882 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1883 {
1884     pte_t *pte;
1885     uint a, pa;
1886
1887     if(newsz >= oldsz)
1888         return oldsz;
1889
1890     a = PGROUNDUP(newsz);
1891     for(; a < oldsz; a += PGSIZE){
1892         pte = walkpgdir(pgdir, (char*)a, 0);
1893         if(!pte)
1894             a += (NPENTRIES - 1) * PGSIZE;
1895         else if((*pte & PTE_P) != 0){
1896             pa = PTE_ADDR(*pte);
1897             if(pa == 0)
1898                 panic("kfree");
1899             char *v = p2v(pa);

```

```

1900     kfree(v);
1901     *pte = 0;
1902 }
1903 }
1904 return newsz;
1905 }
1906
1907 // Free a page table and all the physical memory pages
1908 // in the user part.
1909 void
1910 freevm(pde_t *pgdir)
1911 {
1912     uint i;
1913
1914     if(pgdir == 0)
1915         panic("freevm: no pgdir");
1916     deallocvm(pgdir, KERNBASE, 0);
1917     for(i = 0; i < NPENTRIES; i++){
1918         if(pgdir[i] & PTE_P){
1919             char *v = p2v(PTE_ADDR(pgdir[i]));
1920             kfree(v);
1921         }
1922     }
1923     kfree((char*)pgdir);
1924 }
1925
1926 // Clear PTE_U on a page. Used to create an inaccessible
1927 // page beneath the user stack.
1928 void
1929 clearpteu(pde_t *pgdir, char *uva)
1930 {
1931     pte_t *pte;
1932
1933     pte = walkpgdir(pgdir, uva, 0);
1934     if(pte == 0)
1935         panic("clearpteu");
1936     *pte &= ~PTE_U;
1937 }
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949

```

```

1950 // Given a parent process's page table, create a copy
1951 // of it for a child.
1952 pde_t*
1953 copyvm(pde_t *pgdir, uint sz)
1954 {
1955     pde_t *d;
1956     pte_t *pte;
1957     uint pa, i;
1958     char *mem;
1959
1960     if((d = setupkvm()) == 0)
1961         return 0;
1962     for(i = 0; i < sz; i += PGSIZE){
1963         if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
1964             panic("copyvm: pte should exist");
1965         if(!(*pte & PTE_P))
1966             panic("copyvm: page not present");
1967         pa = PTE_ADDR(*pte);
1968         if((mem = kalloc()) == 0)
1969             goto bad;
1970         memmove(mem, (char*)p2v(pa), PGSIZE);
1971         if(mappages(d, (void*)i, PGSIZE, v2p(mem), PTE_W|PTE_U) < 0)
1972             goto bad;
1973     }
1974     return d;
1975
1976 bad:
1977     freevm(d);
1978     return 0;
1979 }
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999

```

```

2000 // Map user virtual address to kernel address.
2001 char*
2002 uva2ka(pde_t *pgdir, char *uva)
2003 {
2004     pte_t *pte;
2005
2006     pte = walkpgdir(pgdir, uva, 0);
2007     if((*pte & PTE_P) == 0)
2008         return 0;
2009     if((*pte & PTE_U) == 0)
2010         return 0;
2011     return (char*)p2v(PTE_ADDR(*pte));
2012 }
2013
2014 // Copy len bytes from p to user address va in page table pgdir.
2015 // Most useful when pgdir is not the current page table.
2016 // uva2ka ensures this only works for PTE_U pages.
2017 int
2018 copyout(pde_t *pgdir, uint va, void *p, uint len)
2019 {
2020     char *buf, *pa0;
2021     uint n, va0;
2022
2023     buf = (char*)p;
2024     while(len > 0){
2025         va0 = (uint)PGROUNDDOWN(va);
2026         pa0 = uva2ka(pgdir, (char*)va0);
2027         if(pa0 == 0)
2028             return -1;
2029         n = PGSIZE - (va - va0);
2030         if(n > len)
2031             n = len;
2032         memmove(pa0 + (va - va0), buf, n);
2033         len -= n;
2034         buf += n;
2035         va = va0 + PGSIZE;
2036     }
2037     return 0;
2038 }
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049

```

```

2050 // Segments in proc->gdt.
2051 #define NSEGS      7
2052
2053 // Per-CPU state
2054 struct cpu {
2055     uchar id;                    // Local APIC ID; index into cpus[] below
2056     struct context *scheduler;   // swtch() here to enter scheduler
2057     struct taskstate ts;         // Used by x86 to find stack for interrupt
2058     struct segdesc gdt[NSEGS];  // x86 global descriptor table
2059     volatile uint started;       // Has the CPU started?
2060     int ncli;                    // Depth of pushcli nesting.
2061     int intena;                  // Were interrupts enabled before pushcli?
2062
2063     // Cpu-local storage variables; see below
2064     struct cpu *cpu;
2065     struct proc *proc;           // The currently-running process.
2066 };
2067
2068 extern struct cpu cpus[NCPU];
2069 extern int ncpu;
2070
2071 // Per-CPU variables, holding pointers to the
2072 // current cpu and to the current process.
2073 // The asm suffix tells gcc to use "%gs:0" to refer to cpu
2074 // and "%gs:4" to refer to proc. seginit sets up the
2075 // %gs segment register so that %gs refers to the memory
2076 // holding those two variables in the local cpu's struct cpu.
2077 // This is similar to how thread-local variables are implemented
2078 // in thread libraries such as Linux pthreads.
2079 extern struct cpu *cpu asm("%gs:0"); // &cpus[cpunum()]
2080 extern struct proc *proc asm("%gs:4"); // cpus[cpunum()].proc
2081
2082
2083 // Saved registers for kernel context switches.
2084 // Don't need to save all the segment registers (%cs, etc),
2085 // because they are constant across kernel contexts.
2086 // Don't need to save %eax, %ecx, %edx, because the
2087 // x86 convention is that the caller has saved them.
2088 // Contexts are stored at the bottom of the stack they
2089 // describe; the stack pointer is the address of the context.
2090 // The layout of the context matches the layout of the stack in swtch.S
2091 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
2092 // but it is on the stack and allocproc() manipulates it.
2093 struct context {
2094     uint edi;
2095     uint esi;
2096     uint ebx;
2097     uint ebp;
2098     uint eip;
2099 };

```

```

2100 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2101
2102 // Per-process state
2103 struct proc {
2104     uint sz;                // Size of process memory (bytes)
2105     pde_t* pgdir;           // Page table
2106     char *kstack;           // Bottom of kernel stack for this process
2107     enum procstate state;    // Process state
2108     volatile int pid;        // Process ID
2109     struct proc *parent;     // Parent process
2110     struct trapframe *tf;    // Trap frame for current syscall
2111     struct context *context; // swtch() here to run process
2112     void *chan;              // If non-zero, sleeping on chan
2113     int killed;              // If non-zero, have been killed
2114     struct file *ofile[NOFILE]; // Open files
2115     struct inode *cwd;        // Current directory
2116     char name[16];           // Process name (debugging)
2117 };
2118
2119 // Process memory is laid out contiguously, low addresses first:
2120 //   text
2121 //   original data and bss
2122 //   fixed-size stack
2123 //   expandable heap
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149

```

```

2150 #include "types.h"
2151 #include "defs.h"
2152 #include "param.h"
2153 #include "memlayout.h"
2154 #include "mmu.h"
2155 #include "x86.h"
2156 #include "proc.h"
2157 #include "spinlock.h"
2158
2159 struct {
2160     struct spinlock lock;
2161     struct proc proc[NPROC];
2162 } ptable;
2163
2164 static struct proc *initproc;
2165
2166 int nextpid = 1;
2167 extern void forkret(void);
2168 extern void trapret(void);
2169
2170 static void wakeup1(void *chan);
2171
2172 void
2173 pinit(void)
2174 {
2175     initlock(&ptable.lock, "ptable");
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 // Look in the process table for an UNUSED proc.
2201 // If found, change state to EMBRYO and initialize
2202 // state required to run in the kernel.
2203 // Otherwise return 0.
2204 static struct proc*
2205 allocproc(void)
2206 {
2207     struct proc *p;
2208     char *sp;
2209
2210     acquire(&ptable.lock);
2211     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2212         if(p->state == UNUSED)
2213             goto found;
2214         release(&ptable.lock);
2215         return 0;
2216     found:
2217     p->state = EMBRYO;
2218     p->pid = nextpid++;
2219     release(&ptable.lock);
2220
2221     // Allocate kernel stack.
2222     if((p->kstack = kalloc()) == 0){
2223         p->state = UNUSED;
2224         return 0;
2225     }
2226     sp = p->kstack + KSTACKSIZE;
2227
2228     // Leave room for trap frame.
2229     sp -= sizeof *p->tf;
2230     p->tf = (struct trapframe*)sp;
2231
2232     // Set up new context to start executing at forkret,
2233     // which returns to trapret.
2234     sp -= 4;
2235     *(uint*)sp = (uint)trapret;
2236
2237     sp -= sizeof *p->context;
2238     p->context = (struct context*)sp;
2239     memset(p->context, 0, sizeof *p->context);
2240     p->context->eip = (uint)forkret;
2241
2242     return p;
2243 }
2244
2245
2246
2247
2248
2249

```

```

2250 // Set up first user process.
2251 void
2252 userinit(void)
2253 {
2254     struct proc *p;
2255     extern char _binary_initcode_start[], _binary_initcode_size[];
2256
2257     p = allocproc();
2258     initproc = p;
2259     if((p->pgdir = setupkvm()) == 0)
2260         panic("userinit: out of memory?");
2261     inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2262     p->sz = PGSIZE;
2263     memset(p->tf, 0, sizeof(*p->tf));
2264     p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2265     p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2266     p->tf->es = p->tf->ds;
2267     p->tf->ss = p->tf->ds;
2268     p->tf->eflags = FL_IF;
2269     p->tf->esp = PGSIZE;
2270     p->tf->eip = 0; // beginning of initcode.S
2271
2272     safestrcpy(p->name, "initcode", sizeof(p->name));
2273     p->cwd = namei("/");
2274
2275     p->state = RUNNABLE;
2276 }
2277
2278 // Grow current process's memory by n bytes.
2279 // Return 0 on success, -1 on failure.
2280 int
2281 growproc(int n)
2282 {
2283     uint sz;
2284
2285     sz = proc->sz;
2286     if(n > 0){
2287         if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
2288             return -1;
2289     } else if(n < 0){
2290         if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
2291             return -1;
2292     }
2293     proc->sz = sz;
2294     switchuvm(proc);
2295     return 0;
2296 }
2297
2298
2299

```

```

2300 // Create a new process copying p as the parent.
2301 // Sets up stack to return as if from system call.
2302 // Caller must set state of returned proc to RUNNABLE.
2303 int
2304 fork(void)
2305 {
2306     int i, pid;
2307     struct proc *np;
2308
2309     // Allocate process.
2310     if((np = allocproc()) == 0)
2311         return -1;
2312
2313     // Copy process state from p.
2314     if((np->pgdir = copyvm(proc->pgdir, proc->sz)) == 0){
2315         kfree(np->kstack);
2316         np->kstack = 0;
2317         np->state = UNUSED;
2318         return -1;
2319     }
2320     np->sz = proc->sz;
2321     np->parent = proc;
2322     *np->tf = *proc->tf;
2323
2324     // Clear %eax so that fork returns 0 in the child.
2325     np->tf->eax = 0;
2326
2327     for(i = 0; i < NOFILE; i++)
2328         if(proc->ofile[i])
2329             np->ofile[i] = filedup(proc->ofile[i]);
2330     np->cwd = idup(proc->cwd);
2331
2332     pid = np->pid;
2333     np->state = RUNNABLE;
2334     safestrcpy(np->name, proc->name, sizeof(proc->name));
2335     return pid;
2336 }
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349

```

```

2350 // Exit the current process. Does not return.
2351 // An exited process remains in the zombie state
2352 // until its parent calls wait() to find out it exited.
2353 void
2354 exit(void)
2355 {
2356     struct proc *p;
2357     int fd;
2358
2359     if(proc == initproc)
2360         panic("init exiting");
2361
2362     // Close all open files.
2363     for(fd = 0; fd < NOFILE; fd++){
2364         if(proc->ofile[fd]){
2365             fclose(proc->ofile[fd]);
2366             proc->ofile[fd] = 0;
2367         }
2368     }
2369
2370     iput(proc->cwd);
2371     proc->cwd = 0;
2372
2373     acquire(&ptable.lock);
2374
2375     // Parent might be sleeping in wait().
2376     wakeup1(proc->parent);
2377
2378     // Pass abandoned children to init.
2379     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2380         if(p->parent == proc){
2381             p->parent = initproc;
2382             if(p->state == ZOMBIE)
2383                 wakeup1(initproc);
2384         }
2385     }
2386
2387     // Jump into the scheduler, never to return.
2388     proc->state = ZOMBIE;
2389     sched();
2390     panic("zombie exit");
2391 }
2392
2393
2394
2395
2396
2397
2398
2399

```

```

2400 // Wait for a child process to exit and return its pid.
2401 // Return -1 if this process has no children.
2402 int
2403 wait(void)
2404 {
2405     struct proc *p;
2406     int havekids, pid;
2407
2408     acquire(&ptable.lock);
2409     for(;;){
2410         // Scan through table looking for zombie children.
2411         havekids = 0;
2412         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2413             if(p->parent != proc)
2414                 continue;
2415             havekids = 1;
2416             if(p->state == ZOMBIE){
2417                 // Found one.
2418                 pid = p->pid;
2419                 kfree(p->kstack);
2420                 p->kstack = 0;
2421                 freevm(p->pgdir);
2422                 p->state = UNUSED;
2423                 p->pid = 0;
2424                 p->parent = 0;
2425                 p->name[0] = 0;
2426                 p->killed = 0;
2427                 release(&ptable.lock);
2428                 return pid;
2429             }
2430         }
2431
2432         // No point waiting if we don't have any children.
2433         if(!havekids || proc->killed){
2434             release(&ptable.lock);
2435             return -1;
2436         }
2437
2438         // Wait for children to exit. (See wakeup1 call in proc_exit.)
2439         sleep(proc, &ptable.lock);
2440     }
2441 }
2442
2443
2444
2445
2446
2447
2448
2449

```

```

2450 // Per-CPU process scheduler.
2451 // Each CPU calls scheduler() after setting itself up.
2452 // Scheduler never returns. It loops, doing:
2453 // - choose a process to run
2454 // - swtch to start running that process
2455 // - eventually that process transfers control
2456 //   via swtch back to the scheduler.
2457 void
2458 scheduler(void)
2459 {
2460     struct proc *p;
2461
2462     for(;;){
2463         // Enable interrupts on this processor.
2464         sti();
2465
2466         // Loop over process table looking for process to run.
2467         acquire(&ptable.lock);
2468         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2469             if(p->state != RUNNABLE)
2470                 continue;
2471
2472             // Switch to chosen process. It is the process's job
2473             // to release ptable.lock and then reacquire it
2474             // before jumping back to us.
2475             proc = p;
2476             switchvm(p);
2477             p->state = RUNNING;
2478             swtch(&cpu->scheduler, proc->context);
2479             switchkvmm();
2480
2481             // Process is done running for now.
2482             // It should have changed its p->state before coming back.
2483             proc = 0;
2484         }
2485         release(&ptable.lock);
2486     }
2487 }
2488 }
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499

```

```

2500 // Enter scheduler. Must hold only ptable.lock
2501 // and have changed proc->state.
2502 void
2503 sched(void)
2504 {
2505     int intena;
2506
2507     if(!holding(&ptable.lock))
2508         panic("sched ptable.lock");
2509     if(cpu->ncli != 1)
2510         panic("sched locks");
2511     if(proc->state == RUNNING)
2512         panic("sched running");
2513     if(readeflags() & FL_IF)
2514         panic("sched interruptible");
2515     intena = cpu->intena;
2516     swtch(&proc->context, cpu->scheduler);
2517     cpu->intena = intena;
2518 }
2519
2520 // Give up the CPU for one scheduling round.
2521 void
2522 yield(void)
2523 {
2524     acquire(&ptable.lock);
2525     proc->state = RUNNABLE;
2526     sched();
2527     release(&ptable.lock);
2528 }
2529
2530 // A fork child's very first scheduling by scheduler()
2531 // will swtch here. "Return" to user space.
2532 void
2533 forkret(void)
2534 {
2535     static int first = 1;
2536     // Still holding ptable.lock from scheduler.
2537     release(&ptable.lock);
2538
2539     if (first) {
2540         // Some initialization functions must be run in the context
2541         // of a regular process (e.g., they call sleep), and thus cannot
2542         // be run from main().
2543         first = 0;
2544         initlog();
2545     }
2546
2547     // Return to "caller", actually trapret (see allocproc).
2548 }
2549

```

```

2550 // Atomically release lock and sleep on chan.
2551 // Reacquires lock when awakened.
2552 void
2553 sleep(void *chan, struct spinlock *lk)
2554 {
2555     if(proc == 0)
2556         panic("sleep");
2557
2558     if(lk == 0)
2559         panic("sleep without lk");
2560
2561     // Must acquire ptable.lock in order to
2562     // change p->state and then call sched.
2563     // Once we hold ptable.lock, we can be
2564     // guaranteed that we won't miss any wakeup
2565     // (wakeup runs with ptable.lock locked),
2566     // so it's okay to release lk.
2567     if(lk != &ptable.lock){
2568         acquire(&ptable.lock);
2569         release(lk);
2570     }
2571
2572     // Go to sleep.
2573     proc->chan = chan;
2574     proc->state = SLEEPING;
2575     sched();
2576
2577     // Tidy up.
2578     proc->chan = 0;
2579
2580     // Reacquire original lock.
2581     if(lk != &ptable.lock){
2582         release(&ptable.lock);
2583         acquire(lk);
2584     }
2585 }
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599

```



```

2600 // Wake up all processes sleeping on chan.
2601 // The ptable lock must be held.
2602 static void
2603 wakeup1(void *chan)
2604 {
2605     struct proc *p;
2606     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2607         if(p->state == SLEEPING && p->chan == chan)
2608             p->state = RUNNABLE;
2609 }
2610
2611 // Wake up all processes sleeping on chan.
2612 void
2613 wakeup(void *chan)
2614 {
2615     acquire(&ptable.lock);
2616     wakeup1(chan);
2617     release(&ptable.lock);
2618 }
2619
2620 // Kill the process with the given pid.
2621 // Process won't exit until it returns
2622 // to user space (see trap in trap.c).
2623 int
2624 kill(int pid)
2625 {
2626     struct proc *p;
2627     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2628         if(p->pid == pid){
2629             p->killed = 1;
2630             // Wake process from sleep if necessary.
2631             if(p->state == SLEEPING)
2632                 p->state = RUNNABLE;
2633             release(&ptable.lock);
2634             return 0;
2635         }
2636     }
2637     release(&ptable.lock);
2638     return -1;
2639 }
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649

```

```

2650 // Print a process listing to console. For debugging.
2651 // Runs when user types ^P on console.
2652 // No lock to avoid wedging a stuck machine further.
2653 void
2654 procdump(void)
2655 {
2656     static char *states[] = {
2657         [UNUSED]    "unused",
2658         [EMBRYO]    "embryo",
2659         [SLEEPING]  "sleep ",
2660         [RUNNABLE]  "runble",
2661         [RUNNING]   "run   ",
2662         [ZOMBIE]    "zombie"
2663     };
2664     int i;
2665     struct proc *p;
2666     char *state;
2667     uint pc[10];
2668     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2669         if(p->state == UNUSED)
2670             continue;
2671         if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
2672             state = states[p->state];
2673         else
2674             state = "???";
2675         cprintf("%d %s %s", p->pid, state, p->name);
2676         if(p->state == SLEEPING){
2677             getcallerpcs((uint*)p->context->ebp+2, pc);
2678             for(i=0; i<10 && pc[i] != 0; i++)
2679                 cprintf(" %p", pc[i]);
2680         }
2681         cprintf("\n");
2682     }
2683 }
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699

```

```

2700 # Context switch
2701 #
2702 # void switch(struct context **old, struct context *new);
2703 #
2704 # Save current register context in old
2705 # and then load register context from new.
2706
2707 .globl switch
2708 switch:
2709     movl 4(%esp), %eax
2710     movl 8(%esp), %edx
2711
2712 # Save old callee-save registers
2713     pushl %ebp
2714     pushl %ebx
2715     pushl %esi
2716     pushl %edi
2717
2718 # Switch stacks
2719     movl %esp, (%eax)
2720     movl %edx, %esp
2721
2722 # Load new callee-save registers
2723     popl %edi
2724     popl %esi
2725     popl %ebx
2726     popl %ebp
2727     ret
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749

```

```

2750 // Physical memory allocator, intended to allocate
2751 // memory for user processes, kernel stacks, page table pages,
2752 // and pipe buffers. Allocates 4096-byte pages.
2753
2754 #include "types.h"
2755 #include "defs.h"
2756 #include "param.h"
2757 #include "memlayout.h"
2758 #include "mmu.h"
2759 #include "spinlock.h"
2760
2761 void freerange(void *vstart, void *vend);
2762 extern char end[]; // first address after kernel loaded from ELF file
2763
2764 struct run {
2765     struct run *next;
2766 };
2767
2768 struct {
2769     struct spinlock lock;
2770     int use_lock;
2771     struct run *freelist;
2772 } kmem;
2773
2774 // Initialization happens in two phases.
2775 // 1. main() calls kinit1() while still using entrypgdir to place just
2776 // the pages mapped by entrypgdir on free list.
2777 // 2. main() calls kinit2() with the rest of the physical pages
2778 // after installing a full page table that maps them on all cores.
2779 void
2780 kinit1(void *vstart, void *vend)
2781 {
2782     initlock(&kmem.lock, "kmem");
2783     kmem.use_lock = 0;
2784     freerange(vstart, vend);
2785 }
2786
2787 void
2788 kinit2(void *vstart, void *vend)
2789 {
2790     freerange(vstart, vend);
2791     kmem.use_lock = 1;
2792 }
2793
2794
2795
2796
2797
2798
2799

```

```

2800 void
2801 freerange(void *vstart, void *vend)
2802 {
2803     char *p;
2804     p = (char*)PGROUNDUP((uint)vstart);
2805     for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
2806         kfree(p);
2807 }
2808
2809
2810 // Free the page of physical memory pointed at by v,
2811 // which normally should have been returned by a
2812 // call to kalloc(). (The exception is when
2813 // initializing the allocator; see kinit above.)
2814 void
2815 kfree(char *v)
2816 {
2817     struct run *r;
2818
2819     if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)
2820         panic("kfree");
2821
2822     // Fill with junk to catch dangling refs.
2823     memset(v, 1, PGSIZE);
2824
2825     if(kmem.use_lock)
2826         acquire(&kmem.lock);
2827     r = (struct run*)v;
2828     r->next = kmem.freelist;
2829     kmem.freelist = r;
2830     if(kmem.use_lock)
2831         release(&kmem.lock);
2832 }
2833
2834 // Allocate one 4096-byte page of physical memory.
2835 // Returns a pointer that the kernel can use.
2836 // Returns 0 if the memory cannot be allocated.
2837 char*
2838 kalloc(void)
2839 {
2840     struct run *r;
2841
2842     if(kmem.use_lock)
2843         acquire(&kmem.lock);
2844     r = kmem.freelist;
2845     if(r)
2846         kmem.freelist = r->next;
2847     if(kmem.use_lock)
2848         release(&kmem.lock);
2849     return (char*)r;

```

```

2850 }
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899

```

```

2900 // x86 trap and interrupt constants.
2901
2902 // Processor-defined:
2903 #define T_DIVIDE      0      // divide error
2904 #define T_DEBUG      1      // debug exception
2905 #define T_NMI        2      // non-maskable interrupt
2906 #define T_BRKPT      3      // breakpoint
2907 #define T_OFLOW      4      // overflow
2908 #define T_BOUND      5      // bounds check
2909 #define T_ILLOP      6      // illegal opcode
2910 #define T_DEVICE      7      // device not available
2911 #define T_DBLFLT      8      // double fault
2912 // #define T_COPROC    9      // reserved (not used since 486)
2913 #define T_TSS        10     // invalid task switch segment
2914 #define T_SEGNP      11     // segment not present
2915 #define T_STACK      12     // stack exception
2916 #define T_GPFLT      13     // general protection fault
2917 #define T_PGFLT      14     // page fault
2918 // #define T_RES       15     // reserved
2919 #define T_FPERR      16     // floating point error
2920 #define T_ALIGN      17     // alignment check
2921 #define T_MCHK       18     // machine check
2922 #define T_SIMDERR     19     // SIMD floating point error
2923
2924 // These are arbitrarily chosen, but with care not to overlap
2925 // processor defined exceptions or interrupt vectors.
2926 #define T_SYSCALL     64     // system call
2927 #define T_DEFAULT     500    // catchall
2928
2929 #define T_IRQ0        32     // IRQ 0 corresponds to int T_IRQ
2930
2931 #define IRQ_TIMER      0
2932 #define IRQ_KBD        1
2933 #define IRQ_COM1       4
2934 #define IRQ_IDE       14
2935 #define IRQ_ERROR      19
2936 #define IRQ_SPURIOUS   31
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949

```

```

2950 #!/usr/bin/perl -w
2951
2952 # Generate vectors.S, the trap/interrupt entry points.
2953 # There has to be one entry point per interrupt number
2954 # since otherwise there's no way for trap() to discover
2955 # the interrupt number.
2956
2957 print "# generated by vectors.pl - do not edit\n";
2958 print "# handlers\n";
2959 print ".globl alltraps\n";
2960 for(my $i = 0; $i < 256; $i++){
2961     print ".globl vector$i\n";
2962     print "vector$i:\n";
2963     if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
2964         print "    pushl \\\$0\n";
2965     }
2966     print "    pushl \\\$i\n";
2967     print "    jmp alltraps\n";
2968 }
2969
2970 print "\n# vector table\n";
2971 print ".data\n";
2972 print ".globl vectors\n";
2973 print "vectors:\n";
2974 for(my $i = 0; $i < 256; $i++){
2975     print "    .long vector$i\n";
2976 }
2977
2978 # sample output:
2979 # # handlers
2980 # .globl alltraps
2981 # .globl vector0
2982 # vector0:
2983 #     pushl $0
2984 #     pushl $0
2985 #     jmp alltraps
2986 # ...
2987 #
2988 # # vector table
2989 # .data
2990 # .globl vectors
2991 # vectors:
2992 #     .long vector0
2993 #     .long vector1
2994 #     .long vector2
2995 # ...
2996
2997
2998
2999

```

```

3000 #include "mmu.h"
3001
3002 # vectors.S sends all traps here.
3003 .globl alltraps
3004 alltraps:
3005 # Build trap frame.
3006 pushl %ds
3007 pushl %es
3008 pushl %fs
3009 pushl %gs
3010 pushal
3011
3012 # Set up data and per-cpu segments.
3013 movw $(SEG_KDATA<<3), %ax
3014 movw %ax, %ds
3015 movw %ax, %es
3016 movw $(SEG_KCPU<<3), %ax
3017 movw %ax, %fs
3018 movw %ax, %gs
3019
3020 # Call trap(tf), where tf=%esp
3021 pushl %esp
3022 call trap
3023 addl $4, %esp
3024
3025 # Return falls through to trapret...
3026 .globl trapret
3027 trapret:
3028 popal
3029 popl %gs
3030 popl %fs
3031 popl %es
3032 popl %ds
3033 addl $0x8, %esp # trapno and errcode
3034 iret
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049

```

```

3050 #include "types.h"
3051 #include "defs.h"
3052 #include "param.h"
3053 #include "memlayout.h"
3054 #include "mmu.h"
3055 #include "proc.h"
3056 #include "x86.h"
3057 #include "traps.h"
3058 #include "spinlock.h"
3059
3060 // Interrupt descriptor table (shared by all CPUs).
3061 struct gatedesc idt[256];
3062 extern uint vectors[]; // in vectors.S: array of 256 entry pointers
3063 struct spinlock tickslock;
3064 uint ticks;
3065
3066 void
3067 tvinit(void)
3068 {
3069     int i;
3070
3071     for(i = 0; i < 256; i++)
3072         SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3073     SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3074
3075     initlock(&tickslock, "time");
3076 }
3077
3078 void
3079 idtinit(void)
3080 {
3081     lidt(idt, sizeof(idt));
3082 }
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099

```

```

3100 void
3101 trap(struct trapframe *tf)
3102 {
3103     if(tf->trapno == T_SYSCALL){
3104         if(proc->killed)
3105             exit();
3106         proc->tf = tf;
3107         syscall();
3108         if(proc->killed)
3109             exit();
3110         return;
3111     }
3112
3113     switch(tf->trapno){
3114     case T_IRQ0 + IRQ_TIMER:
3115         if(cpu->id == 0){
3116             acquire(&tickslock);
3117             ticks++;
3118             wakeup(&ticks);
3119             release(&tickslock);
3120         }
3121         lapiceoi();
3122         break;
3123     case T_IRQ0 + IRQ_IDE:
3124         ideintr();
3125         lapiceoi();
3126         break;
3127     case T_IRQ0 + IRQ_IDE+1:
3128         // Bochs generates spurious IDE1 interrupts.
3129         break;
3130     case T_IRQ0 + IRQ_KBD:
3131         kbdintr();
3132         lapiceoi();
3133         break;
3134     case T_IRQ0 + IRQ_COM1:
3135         uartintr();
3136         lapiceoi();
3137         break;
3138     case T_IRQ0 + 7:
3139     case T_IRQ0 + IRQ_SPURIOUS:
3140         cprintf("cpu%d: spurious interrupt at %x:%x\n",
3141             cpu->id, tf->cs, tf->eip);
3142         lapiceoi();
3143         break;
3144
3145
3146
3147
3148
3149

```

```

3150     default:
3151         if(proc == 0 || (tf->cs&3) == 0){
3152             // In kernel, it must be our mistake.
3153             cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
3154                 tf->trapno, cpu->id, tf->eip, rcr2());
3155             panic("trap");
3156         }
3157         // In user space, assume process misbehaved.
3158         cprintf("pid %d %s: trap %d err %d on cpu %d "
3159             "eip 0x%x addr 0x%x--kill proc\n",
3160             proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip,
3161             rcr2());
3162         proc->killed = 1;
3163     }
3164
3165     // Force process exit if it has been killed and is in user space.
3166     // (If it is still executing in the kernel, let it keep running
3167     // until it gets to the regular system call return.)
3168     if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3169         exit();
3170
3171     // Force process to give up CPU on clock tick.
3172     // If interrupts were on while locks held, would need to check nlock.
3173     if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
3174         yield();
3175
3176     // Check if the process has been killed since we yielded
3177     if(proc && proc->killed && (tf->cs&3) == DPL_USER)
3178         exit();
3179 }
3180
3181
3182
3183
3184
3185
3186
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199

```

```

3200 // System call numbers
3201 #define SYS_fork    1
3202 #define SYS_exit    2
3203 #define SYS_wait    3
3204 #define SYS_pipe    4
3205 #define SYS_read    5
3206 #define SYS_kill    6
3207 #define SYS_exec    7
3208 #define SYS_fstat   8
3209 #define SYS_chdir   9
3210 #define SYS_dup    10
3211 #define SYS_getpid 11
3212 #define SYS_sbrk   12
3213 #define SYS_sleep  13
3214 #define SYS_uptime 14
3215 #define SYS_open   15
3216 #define SYS_write  16
3217 #define SYS_mknod  17
3218 #define SYS_unlink 18
3219 #define SYS_link   19
3220 #define SYS_mkdir  20
3221 #define SYS_close  21
3222
3223
3224
3225
3226
3227
3228
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249

```

```

3250 #include "types.h"
3251 #include "defs.h"
3252 #include "param.h"
3253 #include "memlayout.h"
3254 #include "mmu.h"
3255 #include "proc.h"
3256 #include "x86.h"
3257 #include "syscall.h"
3258
3259 // User code makes a system call with INT T_SYSCALL.
3260 // System call number in %eax.
3261 // Arguments on the stack, from the user call to the C
3262 // library system call function. The saved user %esp points
3263 // to a saved program counter, and then the first argument.
3264
3265 // Fetch the int at addr from the current process.
3266 int
3267 fetchint(uint addr, int *ip)
3268 {
3269     if(addr >= proc->sz || addr+4 > proc->sz)
3270         return -1;
3271     *ip = *(int*)(addr);
3272     return 0;
3273 }
3274
3275 // Fetch the nul-terminated string at addr from the current process.
3276 // Doesn't actually copy the string - just sets *pp to point at it.
3277 // Returns length of string, not including nul.
3278 int
3279 fetchstr(uint addr, char **pp)
3280 {
3281     char *s, *ep;
3282
3283     if(addr >= proc->sz)
3284         return -1;
3285     *pp = (char*)addr;
3286     ep = (char*)proc->sz;
3287     for(s = *pp; s < ep; s++)
3288         if(*s == 0)
3289             return s - *pp;
3290     return -1;
3291 }
3292
3293 // Fetch the nth 32-bit system call argument.
3294 int
3295 argint(int n, int *ip)
3296 {
3297     return fetchint(proc->tf->esp + 4 + 4*n, ip);
3298 }
3299

```

```

3300 // Fetch the nth word-sized system call argument as a pointer
3301 // to a block of memory of size n bytes. Check that the pointer
3302 // lies within the process address space.
3303 int
3304 argptr(int n, char **pp, int size)
3305 {
3306     int i;
3307
3308     if(argint(n, &i) < 0)
3309         return -1;
3310     if((uint)i >= proc->sz || (uint)i+size > proc->sz)
3311         return -1;
3312     *pp = (char*)i;
3313     return 0;
3314 }
3315
3316 // Fetch the nth word-sized system call argument as a string pointer.
3317 // Check that the pointer is valid and the string is nul-terminated.
3318 // (There is no shared writable memory, so the string can't change
3319 // between this check and being used by the kernel.)
3320 int
3321 argstr(int n, char **pp)
3322 {
3323     int addr;
3324     if(argint(n, &addr) < 0)
3325         return -1;
3326     return fetchstr(addr, pp);
3327 }
3328
3329 extern int sys_chdir(void);
3330 extern int sys_close(void);
3331 extern int sys_dup(void);
3332 extern int sys_exec(void);
3333 extern int sys_exit(void);
3334 extern int sys_fork(void);
3335 extern int sys_fstat(void);
3336 extern int sys_getpid(void);
3337 extern int sys_kill(void);
3338 extern int sys_link(void);
3339 extern int sys_mkdir(void);
3340 extern int sys_mknod(void);
3341 extern int sys_open(void);
3342 extern int sys_pipe(void);
3343 extern int sys_read(void);
3344 extern int sys_sbrk(void);
3345 extern int sys_sleep(void);
3346 extern int sys_unlink(void);
3347 extern int sys_wait(void);
3348 extern int sys_write(void);
3349 extern int sys_uptime(void);

```

```

3350 static int (*syscalls[])(void) = {
3351     [SYS_fork]    sys_fork,
3352     [SYS_exit]    sys_exit,
3353     [SYS_wait]    sys_wait,
3354     [SYS_pipe]    sys_pipe,
3355     [SYS_read]    sys_read,
3356     [SYS_kill]    sys_kill,
3357     [SYS_exec]    sys_exec,
3358     [SYS_fstat]   sys_fstat,
3359     [SYS_chdir]   sys_chdir,
3360     [SYS_dup]     sys_dup,
3361     [SYS_getpid]  sys_getpid,
3362     [SYS_sbrk]    sys_sbrk,
3363     [SYS_sleep]   sys_sleep,
3364     [SYS_uptime]  sys_uptime,
3365     [SYS_open]    sys_open,
3366     [SYS_write]   sys_write,
3367     [SYS_mknod]   sys_mknod,
3368     [SYS_unlink]  sys_unlink,
3369     [SYS_link]    sys_link,
3370     [SYS_mkdir]   sys_mkdir,
3371     [SYS_close]   sys_close,
3372 };
3373
3374 void
3375 syscall(void)
3376 {
3377     int num;
3378
3379     num = proc->tf->eax;
3380     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3381         proc->tf->eax = syscalls[num]();
3382     } else {
3383         cprintf("%d %s: unknown sys call %d\n",
3384             proc->pid, proc->name, num);
3385         proc->tf->eax = -1;
3386     }
3387 }
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399

```



```

3400 #include "types.h"
3401 #include "x86.h"
3402 #include "defs.h"
3403 #include "param.h"
3404 #include "memlayout.h"
3405 #include "mmu.h"
3406 #include "proc.h"
3407
3408 int
3409 sys_fork(void)
3410 {
3411     return fork();
3412 }
3413
3414 int
3415 sys_exit(void)
3416 {
3417     exit();
3418     return 0; // not reached
3419 }
3420
3421 int
3422 sys_wait(void)
3423 {
3424     return wait();
3425 }
3426
3427 int
3428 sys_kill(void)
3429 {
3430     int pid;
3431
3432     if(argint(0, &pid) < 0)
3433         return -1;
3434     return kill(pid);
3435 }
3436
3437 int
3438 sys_getpid(void)
3439 {
3440     return proc->pid;
3441 }
3442
3443
3444
3445
3446
3447
3448
3449

```

```

3450 int
3451 sys_sbrk(void)
3452 {
3453     int addr;
3454     int n;
3455
3456     if(argint(0, &n) < 0)
3457         return -1;
3458     addr = proc->sz;
3459     if(growproc(n) < 0)
3460         return -1;
3461     return addr;
3462 }
3463
3464 int
3465 sys_sleep(void)
3466 {
3467     int n;
3468     uint ticks0;
3469
3470     if(argint(0, &n) < 0)
3471         return -1;
3472     acquire(&tickslock);
3473     ticks0 = ticks;
3474     while(ticks - ticks0 < n){
3475         if(proc->killed){
3476             release(&tickslock);
3477             return -1;
3478         }
3479         sleep(&ticks, &tickslock);
3480     }
3481     release(&tickslock);
3482     return 0;
3483 }
3484
3485 // return how many clock tick interrupts have occurred
3486 // since start.
3487 int
3488 sys_uptime(void)
3489 {
3490     uint xticks;
3491
3492     acquire(&tickslock);
3493     xticks = ticks;
3494     release(&tickslock);
3495     return xticks;
3496 }
3497
3498
3499

```

```
3500 struct buf {
3501     int flags;
3502     uint dev;
3503     uint sector;
3504     struct buf *prev; // LRU cache list
3505     struct buf *next;
3506     struct buf *qnext; // disk queue
3507     uchar data[512];
3508 };
3509 #define B_BUSY 0x1 // buffer is locked by some process
3510 #define B_VALID 0x2 // buffer has been read from disk
3511 #define B_DIRTY 0x4 // buffer needs to be written to disk
3512
3513
3514
3515
3516
3517
3518
3519
3520
3521
3522
3523
3524
3525
3526
3527
3528
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
```

```
3550 #define O_RDONLY 0x000
3551 #define O_WRONLY 0x001
3552 #define O_RDWR 0x002
3553 #define O_CREATE 0x200
3554
3555
3556
3557
3558
3559
3560
3561
3562
3563
3564
3565
3566
3567
3568
3569
3570
3571
3572
3573
3574
3575
3576
3577
3578
3579
3580
3581
3582
3583
3584
3585
3586
3587
3588
3589
3590
3591
3592
3593
3594
3595
3596
3597
3598
3599
```

```

3600 #define T_DIR 1 // Directory
3601 #define T_FILE 2 // File
3602 #define T_DEV 3 // Device
3603
3604 struct stat {
3605     short type; // Type of file
3606     int dev; // File system's disk device
3607     uint ino; // Inode number
3608     short nlink; // Number of links to file
3609     uint size; // Size of file in bytes
3610 };
3611
3612
3613
3614
3615
3616
3617
3618
3619
3620
3621
3622
3623
3624
3625
3626
3627
3628
3629
3630
3631
3632
3633
3634
3635
3636
3637
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649

```

```

3650 // On-disk file system format.
3651 // Both the kernel and user programs use this header file.
3652
3653 // Block 0 is unused.
3654 // Block 1 is super block.
3655 // Blocks 2 through sb.ninodes/IPB hold inodes.
3656 // Then free bitmap blocks holding sb.size bits.
3657 // Then sb.nblocks data blocks.
3658 // Then sb.nlog log blocks.
3659
3660 #define ROOTINO 1 // root i-number
3661 #define BSIZE 512 // block size
3662
3663 // File system super block
3664 struct superblock {
3665     uint size; // Size of file system image (blocks)
3666     uint nblocks; // Number of data blocks
3667     uint ninodes; // Number of inodes.
3668     uint nlog; // Number of log blocks
3669 };
3670
3671 #define NDIRECT 12
3672 #define NINDIRECT (BSIZE / sizeof(uint))
3673 #define MAXFILE (NDIRECT + NINDIRECT)
3674
3675 // On-disk inode structure
3676 struct dinode {
3677     short type; // File type
3678     short major; // Major device number (T_DEV only)
3679     short minor; // Minor device number (T_DEV only)
3680     short nlink; // Number of links to inode in file system
3681     uint size; // Size of file (bytes)
3682     uint addrs[NDIRECT+1]; // Data block addresses
3683 };
3684
3685 // Inodes per block.
3686 #define IPB (BSIZE / sizeof(struct dinode))
3687
3688 // Block containing inode i
3689 #define IBLOCK(i) ((i) / IPB + 2)
3690
3691 // Bitmap bits per block
3692 #define BPB (BSIZE*8)
3693
3694 // Block containing bit for block b
3695 #define BBLOCK(b, ninodes) (b/BPB + (ninodes)/IPB + 3)
3696
3697 // Directory is a file containing a sequence of dirent structures.
3698 #define DIRSIZ 14
3699

```

```

3700 struct dirent {
3701     ushort inum;
3702     char name[DIRSIZ];
3703 };
3704
3705
3706
3707
3708
3709
3710
3711
3712
3713
3714
3715
3716
3717
3718
3719
3720
3721
3722
3723
3724
3725
3726
3727
3728
3729
3730
3731
3732
3733
3734
3735
3736
3737
3738
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3749

```

```

3750 struct file {
3751     enum { FD_NONE, FD_PIPE, FD_INODE } type;
3752     int ref; // reference count
3753     char readable;
3754     char writable;
3755     struct pipe *pipe;
3756     struct inode *ip;
3757     uint off;
3758 };
3759
3760
3761 // in-memory copy of an inode
3762 struct inode {
3763     uint dev;           // Device number
3764     uint inum;          // Inode number
3765     int ref;            // Reference count
3766     int flags;          // I_BUSY, I_VALID
3767
3768     short type;         // copy of disk inode
3769     short major;
3770     short minor;
3771     short nlink;
3772     uint size;
3773     uint addrs[NDIRECT+1];
3774 };
3775 #define I_BUSY 0x1
3776 #define I_VALID 0x2
3777
3778 // table mapping major device number to
3779 // device functions
3780 struct devsw {
3781     int (*read)(struct inode*, char*, int);
3782     int (*write)(struct inode*, char*, int);
3783 };
3784
3785 extern struct devsw devsw[];
3786
3787 #define CONSOLE 1
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3799

```

```

3800 // Simple PIO-based (non-DMA) IDE driver code.
3801
3802 #include "types.h"
3803 #include "defs.h"
3804 #include "param.h"
3805 #include "memlayout.h"
3806 #include "mmu.h"
3807 #include "proc.h"
3808 #include "x86.h"
3809 #include "traps.h"
3810 #include "spinlock.h"
3811 #include "buf.h"
3812
3813 #define IDE_BSY      0x80
3814 #define IDE_DRDY     0x40
3815 #define IDE_DF       0x20
3816 #define IDE_ERR      0x01
3817
3818 #define IDE_CMD_READ 0x20
3819 #define IDE_CMD_WRITE 0x30
3820
3821 // idequeue points to the buf now being read/written to the disk.
3822 // idequeue->qnext points to the next buf to be processed.
3823 // You must hold idelock while manipulating queue.
3824
3825 static struct spinlock idelock;
3826 static struct buf *idequeue;
3827
3828 static int havdisk1;
3829 static void idestart(struct buf*);
3830
3831 // Wait for IDE disk to become ready.
3832 static int
3833 idewait(int checkerr)
3834 {
3835     int r;
3836
3837     while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
3838         ;
3839     if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
3840         return -1;
3841     return 0;
3842 }
3843
3844
3845
3846
3847
3848
3849

```

```

3850 void
3851 ideinit(void)
3852 {
3853     int i;
3854
3855     initlock(&idelock, "ide");
3856     picenable(IRQ_IDE);
3857     ioapicenable(IRQ_IDE, ncpu - 1);
3858     idewait(0);
3859
3860     // Check if disk 1 is present
3861     outb(0x1f6, 0xe0 | (1<<4));
3862     for(i=0; i<1000; i++){
3863         if(inb(0x1f7) != 0){
3864             havdisk1 = 1;
3865             break;
3866         }
3867     }
3868
3869     // Switch back to disk 0.
3870     outb(0x1f6, 0xe0 | (0<<4));
3871 }
3872
3873 // Start the request for b. Caller must hold idelock.
3874 static void
3875 idestart(struct buf *b)
3876 {
3877     if(b == 0)
3878         panic("idestart");
3879
3880     idewait(0);
3881     outb(0x3f6, 0); // generate interrupt
3882     outb(0x1f2, 1); // number of sectors
3883     outb(0x1f3, b->sector & 0xff);
3884     outb(0x1f4, (b->sector >> 8) & 0xff);
3885     outb(0x1f5, (b->sector >> 16) & 0xff);
3886     outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
3887     if(b->flags & B_DIRTY){
3888         outb(0x1f7, IDE_CMD_WRITE);
3889         outsl(0x1f0, b->data, 512/4);
3890     } else {
3891         outb(0x1f7, IDE_CMD_READ);
3892     }
3893 }
3894
3895
3896
3897
3898
3899

```

```

3900 // Interrupt handler.
3901 void
3902 ideintr(void)
3903 {
3904     struct buf *b;
3905
3906     // First queued buffer is the active request.
3907     acquire(&idelock);
3908     if((b = idequeue) == 0){
3909         release(&idelock);
3910         // cprintf("spurious IDE interrupt\n");
3911         return;
3912     }
3913     idequeue = b->qnext;
3914
3915     // Read data if needed.
3916     if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
3917         insl(0x1f0, b->data, 512/4);
3918
3919     // Wake process waiting for this buf.
3920     b->flags |= B_VALID;
3921     b->flags &= ~B_DIRTY;
3922     wakeup(b);
3923
3924     // Start disk on next buf in queue.
3925     if(idequeue != 0)
3926         idestart(idequeue);
3927
3928     release(&idelock);
3929 }
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949

```

```

3950 // Sync buf with disk.
3951 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
3952 // Else if B_VALID is not set, read buf from disk, set B_VALID.
3953 void
3954 iderw(struct buf *b)
3955 {
3956     struct buf **pp;
3957
3958     if(!(b->flags & B_BUSY))
3959         panic("iderw: buf not busy");
3960     if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
3961         panic("iderw: nothing to do");
3962     if(b->dev != 0 && !havedisk1)
3963         panic("iderw: ide disk 1 not present");
3964
3965     acquire(&idelock);
3966
3967     // Append b to idequeue.
3968     b->qnext = 0;
3969     for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
3970         ;
3971     *pp = b;
3972
3973     // Start disk if necessary.
3974     if(idequeue == b)
3975         idestart(b);
3976
3977     // Wait for request to finish.
3978     while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
3979         sleep(b, &idelock);
3980     }
3981
3982     release(&idelock);
3983 }
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999

```

```

4000 // Buffer cache.
4001 //
4002 // The buffer cache is a linked list of buf structures holding
4003 // cached copies of disk block contents. Caching disk blocks
4004 // in memory reduces the number of disk reads and also provides
4005 // a synchronization point for disk blocks used by multiple processes.
4006 //
4007 // Interface:
4008 // * To get a buffer for a particular disk block, call bread.
4009 // * After changing buffer data, call bwrite to write it to disk.
4010 // * When done with the buffer, call brelse.
4011 // * Do not use the buffer after calling brelse.
4012 // * Only one process at a time can use a buffer,
4013 //   so do not keep them longer than necessary.
4014 //
4015 // The implementation uses three state flags internally:
4016 // * B_BUSY: the block has been returned from bread
4017 //   and has not been passed back to brelse.
4018 // * B_VALID: the buffer data has been read from the disk.
4019 // * B_DIRTY: the buffer data has been modified
4020 //   and needs to be written to disk.
4021
4022 #include "types.h"
4023 #include "defs.h"
4024 #include "param.h"
4025 #include "spinlock.h"
4026 #include "buf.h"
4027
4028 struct {
4029   struct spinlock lock;
4030   struct buf buf[NBUF];
4031
4032   // Linked list of all buffers, through prev/next.
4033   // head.next is most recently used.
4034   struct buf head;
4035 } bcache;
4036
4037 void
4038 binit(void)
4039 {
4040   struct buf *b;
4041
4042   initlock(&bcache.lock, "bcache");
4043
4044
4045
4046
4047
4048
4049

```

```

4050 // Create linked list of buffers
4051 bcache.head.prev = &bcache.head;
4052 bcache.head.next = &bcache.head;
4053 for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4054   b->next = bcache.head.next;
4055   b->prev = &bcache.head;
4056   b->dev = -1;
4057   bcache.head.next->prev = b;
4058   bcache.head.next = b;
4059 }
4060 }
4061
4062 // Look through buffer cache for sector on device dev.
4063 // If not found, allocate fresh block.
4064 // In either case, return B_BUSY buffer.
4065 static struct buf*
4066 bget(uint dev, uint sector)
4067 {
4068   struct buf *b;
4069
4070   acquire(&bcache.lock);
4071
4072   loop:
4073   // Is the sector already cached?
4074   for(b = bcache.head.next; b != &bcache.head; b = b->next){
4075     if(b->dev == dev && b->sector == sector){
4076       if(!(b->flags & B_BUSY)){
4077         b->flags |= B_BUSY;
4078         release(&bcache.lock);
4079         return b;
4080       }
4081       sleep(b, &bcache.lock);
4082       goto loop;
4083     }
4084   }
4085
4086   // Not cached; recycle some non-busy and clean buffer.
4087   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4088     if((b->flags & B_BUSY) == 0 && (b->flags & B_DIRTY) == 0){
4089       b->dev = dev;
4090       b->sector = sector;
4091       b->flags = B_BUSY;
4092       release(&bcache.lock);
4093       return b;
4094     }
4095   }
4096   panic("bget: no buffers");
4097 }
4098
4099

```

```

4100 // Return a B_BUSY buf with the contents of the indicated disk sector.
4101 struct buf*
4102 bread(uint dev, uint sector)
4103 {
4104     struct buf *b;
4105
4106     b = bget(dev, sector);
4107     if(!(b->flags & B_VALID))
4108         iderw(b);
4109     return b;
4110 }
4111
4112 // Write b's contents to disk. Must be B_BUSY.
4113 void
4114 bwrite(struct buf *b)
4115 {
4116     if((b->flags & B_BUSY) == 0)
4117         panic("bwrite");
4118     b->flags |= B_DIRTY;
4119     iderw(b);
4120 }
4121
4122 // Release a B_BUSY buffer.
4123 // Move to the head of the MRU list.
4124 void
4125 brelse(struct buf *b)
4126 {
4127     if((b->flags & B_BUSY) == 0)
4128         panic("brelse");
4129
4130     acquire(&bcache.lock);
4131
4132     b->next->prev = b->prev;
4133     b->prev->next = b->next;
4134     b->next = bcache.head.next;
4135     b->prev = &bcache.head;
4136     bcache.head.next->prev = b;
4137     bcache.head.next = b;
4138
4139     b->flags &= ~B_BUSY;
4140     wakeup(b);
4141
4142     release(&bcache.lock);
4143 }
4144
4145
4146
4147
4148
4149

```

```

4150 #include "types.h"
4151 #include "defs.h"
4152 #include "param.h"
4153 #include "spinlock.h"
4154 #include "fs.h"
4155 #include "buf.h"
4156
4157 // Simple logging. Each system call that might write the file system
4158 // should be surrounded with begin_trans() and commit_trans() calls.
4159 //
4160 // The log holds at most one transaction at a time. Commit forces
4161 // the log (with commit record) to disk, then installs the affected
4162 // blocks to disk, then erases the log. begin_trans() ensures that
4163 // only one system call can be in a transaction; others must wait.
4164 //
4165 // Allowing only one transaction at a time means that the file
4166 // system code doesn't have to worry about the possibility of
4167 // one transaction reading a block that another one has modified,
4168 // for example an i-node block.
4169 //
4170 // Read-only system calls don't need to use transactions, though
4171 // this means that they may observe uncommitted data. I-node and
4172 // buffer locks prevent read-only calls from seeing inconsistent data.
4173 //
4174 // The log is a physical re-do log containing disk blocks.
4175 // The on-disk log format:
4176 //   header block, containing sector #s for block A, B, C, ...
4177 //   block A
4178 //   block B
4179 //   block C
4180 //   ...
4181 // Log appends are synchronous.
4182
4183 // Contents of the header block, used for both the on-disk header block
4184 // and to keep track in memory of logged sector #s before commit.
4185 struct logheader {
4186     int n;
4187     int sector[LOGSIZE];
4188 };
4189
4190 struct log {
4191     struct spinlock lock;
4192     int start;
4193     int size;
4194     int busy; // a transaction is active
4195     int dev;
4196     struct logheader lh;
4197 };
4198
4199

```



```

4200 struct log log;
4201
4202 static void recover_from_log(void);
4203
4204 void
4205 initlog(void)
4206 {
4207     if (sizeof(struct logheader) >= BSIZE)
4208         panic("initlog: too big logheader");
4209
4210     struct superblock sb;
4211     initlock(&log.lock, "log");
4212     readsb(ROOTDEV, &sb);
4213     log.start = sb.size - sb.nlog;
4214     log.size = sb.nlog;
4215     log.dev = ROOTDEV;
4216     recover_from_log();
4217 }
4218
4219 // Copy committed blocks from log to their home location
4220 static void
4221 install_trans(void)
4222 {
4223     int tail;
4224
4225     for (tail = 0; tail < log.lh.n; tail++) {
4226         struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
4227         struct buf *dbuf = bread(log.dev, log.lh.sector[tail]); // read dst
4228         memmove(dbuf->data, lbuf->data, BSIZE); // copy block to dst
4229         bwrite(dbuf); // write dst to disk
4230         brelse(lbuf);
4231         brelse(dbuf);
4232     }
4233 }
4234
4235 // Read the log header from disk into the in-memory log header
4236 static void
4237 read_head(void)
4238 {
4239     struct buf *buf = bread(log.dev, log.start);
4240     struct logheader *lh = (struct logheader *) (buf->data);
4241     int i;
4242     log.lh.n = lh->n;
4243     for (i = 0; i < log.lh.n; i++) {
4244         log.lh.sector[i] = lh->sector[i];
4245     }
4246     brelse(buf);
4247 }
4248
4249

```

```

4250 // Write in-memory log header to disk.
4251 // This is the true point at which the
4252 // current transaction commits.
4253 static void
4254 write_head(void)
4255 {
4256     struct buf *buf = bread(log.dev, log.start);
4257     struct logheader *hb = (struct logheader *) (buf->data);
4258     int i;
4259     hb->n = log.lh.n;
4260     for (i = 0; i < log.lh.n; i++) {
4261         hb->sector[i] = log.lh.sector[i];
4262     }
4263     bwrite(buf);
4264     brelse(buf);
4265 }
4266
4267 static void
4268 recover_from_log(void)
4269 {
4270     read_head();
4271     install_trans(); // if committed, copy from log to disk
4272     log.lh.n = 0;
4273     write_head(); // clear the log
4274 }
4275
4276 void
4277 begin_trans(void)
4278 {
4279     acquire(&log.lock);
4280     while (log.busy) {
4281         sleep(&log, &log.lock);
4282     }
4283     log.busy = 1;
4284     release(&log.lock);
4285 }
4286
4287
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299

```

```

4300 void
4301 commit_trans(void)
4302 {
4303     if (log.lh.n > 0) {
4304         write_head(); // Write header to disk -- the real commit
4305         install_trans(); // Now install writes to home locations
4306         log.lh.n = 0;
4307         write_head(); // Erase the transaction from the log
4308     }
4309
4310     acquire(&log.lock);
4311     log.busy = 0;
4312     wakeup(&log);
4313     release(&log.lock);
4314 }
4315
4316 // Caller has modified b->data and is done with the buffer.
4317 // Append the block to the log and record the block number,
4318 // but don't write the log header (which would commit the write).
4319 // log_write() replaces bwrite(); a typical use is:
4320 //   bp = bread(...)
4321 //   modify bp->data[]
4322 //   log_write(bp)
4323 //   brelse(bp)
4324 void
4325 log_write(struct buf *b)
4326 {
4327     int i;
4328
4329     if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
4330         panic("too big a transaction");
4331     if (!log.busy)
4332         panic("write outside of trans");
4333
4334     for (i = 0; i < log.lh.n; i++) {
4335         if (log.lh.sector[i] == b->sector) // log absorbtion?
4336             break;
4337     }
4338     log.lh.sector[i] = b->sector;
4339     struct buf *lbuf = bread(b->dev, log.start+i+1);
4340     memmove(lbuf->data, b->data, BSIZE);
4341     bwrite(lbuf);
4342     brelse(lbuf);
4343     if (i == log.lh.n)
4344         log.lh.n++;
4345     b->flags |= B_DIRTY; // XXX prevent eviction
4346 }
4347
4348
4349

```

```

4350 // Blank page.
4351
4352
4353
4354
4355
4356
4357
4358
4359
4360
4361
4362
4363
4364
4365
4366
4367
4368
4369
4370
4371
4372
4373
4374
4375
4376
4377
4378
4379
4380
4381
4382
4383
4384
4385
4386
4387
4388
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399

```

```

4400 // File system implementation. Five layers:
4401 //   + Blocks: allocator for raw disk blocks.
4402 //   + Log: crash recovery for multi-step updates.
4403 //   + Files: inode allocator, reading, writing, metadata.
4404 //   + Directories: inode with special contents (list of other inodes!)
4405 //   + Names: paths like /usr/rtn/xv6/fs.c for convenient naming.
4406 //
4407 // This file contains the low-level file system manipulation
4408 // routines. The (higher-level) system call implementations
4409 // are in sysfile.c.
4410
4411 #include "types.h"
4412 #include "defs.h"
4413 #include "param.h"
4414 #include "stat.h"
4415 #include "mmu.h"
4416 #include "proc.h"
4417 #include "spinlock.h"
4418 #include "buf.h"
4419 #include "fs.h"
4420 #include "file.h"
4421
4422 #define min(a, b) ((a) < (b) ? (a) : (b))
4423 static void itrunc(struct inode*);
4424
4425 // Read the super block.
4426 void
4427 readsb(int dev, struct superblock *sb)
4428 {
4429     struct buf *bp;
4430
4431     bp = bread(dev, 1);
4432     memmove(sb, bp->data, sizeof(*sb));
4433     brelse(bp);
4434 }
4435
4436 // Zero a block.
4437 static void
4438 bzero(int dev, int bno)
4439 {
4440     struct buf *bp;
4441
4442     bp = bread(dev, bno);
4443     memset(bp->data, 0, BSIZE);
4444     log_write(bp);
4445     brelse(bp);
4446 }
4447
4448
4449

```

```

4450 // Blocks.
4451
4452 // Allocate a zeroed disk block.
4453 static uint
4454 balloc(uint dev)
4455 {
4456     int b, bi, m;
4457     struct buf *bp;
4458     struct superblock sb;
4459
4460     bp = 0;
4461     readsb(dev, &sb);
4462     for(b = 0; b < sb.size; b += BPB){
4463         bp = bread(dev, BBLOCK(b, sb.ninodes));
4464         for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
4465             m = 1 << (bi % 8);
4466             if((bp->data[bi/8] & m) == 0){ // Is block free?
4467                 bp->data[bi/8] |= m; // Mark block in use.
4468                 log_write(bp);
4469                 brelse(bp);
4470                 bzero(dev, b + bi);
4471                 return b + bi;
4472             }
4473         }
4474         brelse(bp);
4475     }
4476     panic("balloc: out of blocks");
4477 }
4478
4479 // Free a disk block.
4480 static void
4481 bfree(int dev, uint b)
4482 {
4483     struct buf *bp;
4484     struct superblock sb;
4485     int bi, m;
4486
4487     readsb(dev, &sb);
4488     bp = bread(dev, BBLOCK(b, sb.ninodes));
4489     bi = b % BPB;
4490     m = 1 << (bi % 8);
4491     if((bp->data[bi/8] & m) == 0)
4492         panic("freeing free block");
4493     bp->data[bi/8] &= ~m;
4494     log_write(bp);
4495     brelse(bp);
4496 }
4497
4498
4499

```

```

4500 // Inodes.
4501 //
4502 // An inode describes a single unnamed file.
4503 // The inode disk structure holds metadata: the file's type,
4504 // its size, the number of links referring to it, and the
4505 // list of blocks holding the file's content.
4506 //
4507 // The inodes are laid out sequentially on disk immediately after
4508 // the superblock. Each inode has a number, indicating its
4509 // position on the disk.
4510 //
4511 // The kernel keeps a cache of in-use inodes in memory
4512 // to provide a place for synchronizing access
4513 // to inodes used by multiple processes. The cached
4514 // inodes include book-keeping information that is
4515 // not stored on disk: ip->ref and ip->flags.
4516 //
4517 // An inode and its in-memory representative go through a
4518 // sequence of states before they can be used by the
4519 // rest of the file system code.
4520 //
4521 // * Allocation: an inode is allocated if its type (on disk)
4522 //   is non-zero. ialloc() allocates, iput() frees if
4523 //   the link count has fallen to zero.
4524 //
4525 // * Referencing in cache: an entry in the inode cache
4526 //   is free if ip->ref is zero. Otherwise ip->ref tracks
4527 //   the number of in-memory pointers to the entry (open
4528 //   files and current directories). iget() to find or
4529 //   create a cache entry and increment its ref, iput()
4530 //   to decrement ref.
4531 //
4532 // * Valid: the information (type, size, &c) in an inode
4533 //   cache entry is only correct when the I_VALID bit
4534 //   is set in ip->flags. ilock() reads the inode from
4535 //   the disk and sets I_VALID, while iput() clears
4536 //   I_VALID if ip->ref has fallen to zero.
4537 //
4538 // * Locked: file system code may only examine and modify
4539 //   the information in an inode and its content if it
4540 //   has first locked the inode. The I_BUSY flag indicates
4541 //   that the inode is locked. ilock() sets I_BUSY,
4542 //   while iunlock clears it.
4543 //
4544 // Thus a typical sequence is:
4545 //   ip = iget(dev, inum)
4546 //   ilock(ip)
4547 //   ... examine and modify ip->xxx ...
4548 //   iunlock(ip)
4549 //   iput(ip)

```

```

4550 //
4551 // ilock() is separate from iget() so that system calls can
4552 // get a long-term reference to an inode (as for an open file)
4553 // and only lock it for short periods (e.g., in read()).
4554 // The separation also helps avoid deadlock and races during
4555 // pathname lookup. iget() increments ip->ref so that the inode
4556 // stays cached and pointers to it remain valid.
4557 //
4558 // Many internal file system functions expect the caller to
4559 // have locked the inodes involved; this lets callers create
4560 // multi-step atomic operations.
4561 //
4562 struct {
4563   struct spinlock lock;
4564   struct inode inode[NINODE];
4565 } icache;
4566
4567 void
4568 iinit(void)
4569 {
4570   initlock(&icache.lock, "icache");
4571 }
4572
4573 static struct inode* iget(uint dev, uint inum);
4574
4575
4576
4577
4578
4579
4580
4581
4582
4583
4584
4585
4586
4587
4588
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4599

```

```

4600 // Allocate a new inode with the given type on device dev.
4601 // A free inode has a type of zero.
4602 struct inode*
4603 ialloc(uint dev, short type)
4604 {
4605     int inum;
4606     struct buf *bp;
4607     struct dinode *dip;
4608     struct superblock sb;
4609
4610     readsb(dev, &sb);
4611
4612     for(inum = 1; inum < sb.ninodes; inum++){
4613         bp = bread(dev, IBLOCK(inum));
4614         dip = (struct dinode*)bp->data + inum%IPB;
4615         if(dip->type == 0){ // a free inode
4616             memset(dip, 0, sizeof(*dip));
4617             dip->type = type;
4618             log_write(bp); // mark it allocated on the disk
4619             brelse(bp);
4620             return iget(dev, inum);
4621         }
4622         brelse(bp);
4623     }
4624     panic("ialloc: no inodes");
4625 }
4626
4627 // Copy a modified in-memory inode to disk.
4628 void
4629 iupdate(struct inode *ip)
4630 {
4631     struct buf *bp;
4632     struct dinode *dip;
4633
4634     bp = bread(ip->dev, IBLOCK(ip->inum));
4635     dip = (struct dinode*)bp->data + ip->inum%IPB;
4636     dip->type = ip->type;
4637     dip->major = ip->major;
4638     dip->minor = ip->minor;
4639     dip->nlink = ip->nlink;
4640     dip->size = ip->size;
4641     memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
4642     log_write(bp);
4643     brelse(bp);
4644 }
4645
4646
4647
4648
4649

```

```

4650 // Find the inode with number inum on device dev
4651 // and return the in-memory copy. Does not lock
4652 // the inode and does not read it from disk.
4653 static struct inode*
4654 iget(uint dev, uint inum)
4655 {
4656     struct inode *ip, *empty;
4657
4658     acquire(&icache.lock);
4659
4660     // Is the inode already cached?
4661     empty = 0;
4662     for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
4663         if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
4664             ip->ref++;
4665             release(&icache.lock);
4666             return ip;
4667         }
4668         if(empty == 0 && ip->ref == 0) // Remember empty slot.
4669             empty = ip;
4670     }
4671
4672     // Recycle an inode cache entry.
4673     if(empty == 0)
4674         panic("iget: no inodes");
4675
4676     ip = empty;
4677     ip->dev = dev;
4678     ip->inum = inum;
4679     ip->ref = 1;
4680     ip->flags = 0;
4681     release(&icache.lock);
4682
4683     return ip;
4684 }
4685
4686 // Increment reference count for ip.
4687 // Returns ip to enable ip = idup(ip1) idiom.
4688 struct inode*
4689 idup(struct inode *ip)
4690 {
4691     acquire(&icache.lock);
4692     ip->ref++;
4693     release(&icache.lock);
4694     return ip;
4695 }
4696
4697
4698
4699

```

```

4700 // Lock the given inode.
4701 // Reads the inode from disk if necessary.
4702 void
4703 ilock(struct inode *ip)
4704 {
4705     struct buf *bp;
4706     struct dinode *dip;
4707
4708     if(ip == 0 || ip->ref < 1)
4709         panic("ilock");
4710
4711     acquire(&icache.lock);
4712     while(ip->flags & I_BUSY)
4713         sleep(ip, &icache.lock);
4714     ip->flags |= I_BUSY;
4715     release(&icache.lock);
4716
4717     if(!(ip->flags & I_INVALID)){
4718         bp = bread(ip->dev, IBLOCK(ip->inum));
4719         dip = (struct dinode*)bp->data + ip->inum%IPB;
4720         ip->type = dip->type;
4721         ip->major = dip->major;
4722         ip->minor = dip->minor;
4723         ip->nlink = dip->nlink;
4724         ip->size = dip->size;
4725         memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
4726         brelse(bp);
4727         ip->flags |= I_INVALID;
4728         if(ip->type == 0)
4729             panic("ilock: no type");
4730     }
4731 }
4732
4733 // Unlock the given inode.
4734 void
4735 iunlock(struct inode *ip)
4736 {
4737     if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
4738         panic("iunlock");
4739
4740     acquire(&icache.lock);
4741     ip->flags &= ~I_BUSY;
4742     wakeup(ip);
4743     release(&icache.lock);
4744 }
4745
4746
4747
4748
4749

```

```

4750 // Drop a reference to an in-memory inode.
4751 // If that was the last reference, the inode cache entry can
4752 // be recycled.
4753 // If that was the last reference and the inode has no links
4754 // to it, free the inode (and its content) on disk.
4755 void
4756 iput(struct inode *ip)
4757 {
4758     acquire(&icache.lock);
4759     if(ip->ref == 1 && (ip->flags & I_INVALID) && ip->nlink == 0){
4760         // inode has no links: truncate and free inode.
4761         if(ip->flags & I_BUSY)
4762             panic("iput busy");
4763         ip->flags |= I_BUSY;
4764         release(&icache.lock);
4765         itrunc(ip);
4766         ip->type = 0;
4767         iupdate(ip);
4768         acquire(&icache.lock);
4769         ip->flags = 0;
4770         wakeup(ip);
4771     }
4772     ip->ref--;
4773     release(&icache.lock);
4774 }
4775
4776 // Common idiom: unlock, then put.
4777 void
4778 iunlockput(struct inode *ip)
4779 {
4780     iunlock(ip);
4781     iput(ip);
4782 }
4783
4784
4785
4786
4787
4788
4789
4790
4791
4792
4793
4794
4795
4796
4797
4798
4799

```

```

4800 // Inode content
4801 //
4802 // The content (data) associated with each inode is stored
4803 // in blocks on the disk. The first NDIRECT block numbers
4804 // are listed in ip->addrs[]. The next NINDIRECT blocks are
4805 // listed in block ip->addrs[NDIRECT].
4806
4807 // Return the disk block address of the nth block in inode ip.
4808 // If there is no such block, bmap allocates one.
4809 static uint
4810 bmap(struct inode *ip, uint bn)
4811 {
4812     uint addr, *a;
4813     struct buf *bp;
4814
4815     if(bn < NDIRECT){
4816         if((addr = ip->addrs[bn]) == 0)
4817             ip->addrs[bn] = addr = balloc(ip->dev);
4818         return addr;
4819     }
4820     bn -= NDIRECT;
4821
4822     if(bn < NINDIRECT){
4823         // Load indirect block, allocating if necessary.
4824         if((addr = ip->addrs[NDIRECT]) == 0)
4825             ip->addrs[NDIRECT] = addr = balloc(ip->dev);
4826         bp = bread(ip->dev, addr);
4827         a = (uint*)bp->data;
4828         if((addr = a[bn]) == 0){
4829             a[bn] = addr = balloc(ip->dev);
4830             log_write(bp);
4831         }
4832         brelse(bp);
4833         return addr;
4834     }
4835
4836     panic("bmap: out of range");
4837 }
4838
4839
4840
4841
4842
4843
4844
4845
4846
4847
4848
4849

```

```

4850 // Truncate inode (discard contents).
4851 // Only called when the inode has no links
4852 // to it (no directory entries referring to it)
4853 // and has no in-memory reference to it (is
4854 // not an open file or current directory).
4855 static void
4856 itrunc(struct inode *ip)
4857 {
4858     int i, j;
4859     struct buf *bp;
4860     uint *a;
4861
4862     for(i = 0; i < NDIRECT; i++){
4863         if(ip->addrs[i]){
4864             bfree(ip->dev, ip->addrs[i]);
4865             ip->addrs[i] = 0;
4866         }
4867     }
4868
4869     if(ip->addrs[NDIRECT]){
4870         bp = bread(ip->dev, ip->addrs[NDIRECT]);
4871         a = (uint*)bp->data;
4872         for(j = 0; j < NINDIRECT; j++){
4873             if(a[j])
4874                 bfree(ip->dev, a[j]);
4875         }
4876         brelse(bp);
4877         bfree(ip->dev, ip->addrs[NDIRECT]);
4878         ip->addrs[NDIRECT] = 0;
4879     }
4880
4881     ip->size = 0;
4882     iupdate(ip);
4883 }
4884
4885 // Copy stat information from inode.
4886 void
4887 stati(struct inode *ip, struct stat *st)
4888 {
4889     st->dev = ip->dev;
4890     st->ino = ip->inum;
4891     st->type = ip->type;
4892     st->nlink = ip->nlink;
4893     st->size = ip->size;
4894 }
4895
4896
4897
4898
4899

```

```

4900 // Read data from inode.
4901 int
4902 readi(struct inode *ip, char *dst, uint off, uint n)
4903 {
4904     uint tot, m;
4905     struct buf *bp;
4906
4907     if(ip->type == T_DEV){
4908         if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
4909             return -1;
4910         return devsw[ip->major].read(ip, dst, n);
4911     }
4912
4913     if(off > ip->size || off + n < off)
4914         return -1;
4915     if(off + n > ip->size)
4916         n = ip->size - off;
4917
4918     for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
4919         bp = bread(ip->dev, bmap(ip, off/BSIZE));
4920         m = min(n - tot, BSIZE - off%BSIZE);
4921         memmove(dst, bp->data + off%BSIZE, m);
4922         brelse(bp);
4923     }
4924     return n;
4925 }
4926
4927
4928
4929
4930
4931
4932
4933
4934
4935
4936
4937
4938
4939
4940
4941
4942
4943
4944
4945
4946
4947
4948
4949

```

```

4950 // Write data to inode.
4951 int
4952 writei(struct inode *ip, char *src, uint off, uint n)
4953 {
4954     uint tot, m;
4955     struct buf *bp;
4956
4957     if(ip->type == T_DEV){
4958         if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
4959             return -1;
4960         return devsw[ip->major].write(ip, src, n);
4961     }
4962
4963     if(off > ip->size || off + n < off)
4964         return -1;
4965     if(off + n > MAXFILE*BSIZE)
4966         return -1;
4967
4968     for(tot=0; tot<n; tot+=m, off+=m, src+=m){
4969         bp = bread(ip->dev, bmap(ip, off/BSIZE));
4970         m = min(n - tot, BSIZE - off%BSIZE);
4971         memmove(bp->data + off%BSIZE, src, m);
4972         log_write(bp);
4973         brelse(bp);
4974     }
4975
4976     if(n > 0 && off > ip->size){
4977         ip->size = off;
4978         iupdate(ip);
4979     }
4980     return n;
4981 }
4982
4983
4984
4985
4986
4987
4988
4989
4990
4991
4992
4993
4994
4995
4996
4997
4998
4999

```



```

5000 // Directories
5001
5002 int
5003 namecmp(const char *s, const char *t)
5004 {
5005     return strncmp(s, t, DIRSIZ);
5006 }
5007
5008 // Look for a directory entry in a directory.
5009 // If found, set *poff to byte offset of entry.
5010 struct inode*
5011 dirlookup(struct inode *dp, char *name, uint *poff)
5012 {
5013     uint off, inum;
5014     struct dirent de;
5015
5016     if(dp->type != T_DIR)
5017         panic("dirlookup not DIR");
5018
5019     for(off = 0; off < dp->size; off += sizeof(de)){
5020         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5021             panic("dirlink read");
5022         if(de.inum == 0)
5023             continue;
5024         if(namecmp(name, de.name) == 0){
5025             // entry matches path element
5026             if(poff)
5027                 *poff = off;
5028             inum = de.inum;
5029             return iget(dp->dev, inum);
5030         }
5031     }
5032
5033     return 0;
5034 }
5035
5036
5037
5038
5039
5040
5041
5042
5043
5044
5045
5046
5047
5048
5049

```

```

5050 // Write a new directory entry (name, inum) into the directory dp.
5051 int
5052 dirlink(struct inode *dp, char *name, uint inum)
5053 {
5054     int off;
5055     struct dirent de;
5056     struct inode *ip;
5057
5058     // Check that name is not present.
5059     if((ip = dirlookup(dp, name, 0)) != 0){
5060         iput(ip);
5061         return -1;
5062     }
5063
5064     // Look for an empty dirent.
5065     for(off = 0; off < dp->size; off += sizeof(de)){
5066         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5067             panic("dirlink read");
5068         if(de.inum == 0)
5069             break;
5070     }
5071
5072     strncpy(de.name, name, DIRSIZ);
5073     de.inum = inum;
5074     if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5075         panic("dirlink");
5076
5077     return 0;
5078 }
5079
5080
5081
5082
5083
5084
5085
5086
5087
5088
5089
5090
5091
5092
5093
5094
5095
5096
5097
5098
5099

```

```

5100 // Paths
5101
5102 // Copy the next path element from path into name.
5103 // Return a pointer to the element following the copied one.
5104 // The returned path has no leading slashes,
5105 // so the caller can check *path=='\0' to see if the name is the last one.
5106 // If no name to remove, return 0.
5107 //
5108 // Examples:
5109 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5110 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5111 //   skipelem("a", name) = "", setting name = "a"
5112 //   skipelem("", name) = skipelem("///", name) = 0
5113 //
5114 static char*
5115 skipelem(char *path, char *name)
5116 {
5117     char *s;
5118     int len;
5119
5120     while(*path == '/')
5121         path++;
5122     if(*path == 0)
5123         return 0;
5124     s = path;
5125     while(*path != '/' && *path != 0)
5126         path++;
5127     len = path - s;
5128     if(len >= DIRSIZ)
5129         memmove(name, s, DIRSIZ);
5130     else {
5131         memmove(name, s, len);
5132         name[len] = 0;
5133     }
5134     while(*path == '/')
5135         path++;
5136     return path;
5137 }
5138
5139
5140
5141
5142
5143
5144
5145
5146
5147
5148
5149

```

```

5150 // Look up and return the inode for a path name.
5151 // If parent != 0, return the inode for the parent and copy the final
5152 // path element into name, which must have room for DIRSIZ bytes.
5153 static struct inode*
5154 namex(char *path, int nameiparent, char *name)
5155 {
5156     struct inode *ip, *next;
5157
5158     if(*path == '/')
5159         ip = iget(ROOTDEV, ROOTINO);
5160     else
5161         ip = idup(proc->cwd);
5162
5163     while((path = skipelem(path, name)) != 0){
5164         ilock(ip);
5165         if(ip->type != T_DIR){
5166             iunlockput(ip);
5167             return 0;
5168         }
5169         if(nameiparent && *path == '\0'){
5170             // Stop one level early.
5171             iunlock(ip);
5172             return ip;
5173         }
5174         if((next = dirlookup(ip, name, 0)) == 0){
5175             iunlockput(ip);
5176             return 0;
5177         }
5178         iunlockput(ip);
5179         ip = next;
5180     }
5181     if(nameiparent){
5182         iput(ip);
5183         return 0;
5184     }
5185     return ip;
5186 }
5187
5188 struct inode*
5189 namei(char *path)
5190 {
5191     char name[DIRSIZ];
5192     return namex(path, 0, name);
5193 }
5194
5195 struct inode*
5196 nameiparent(char *path, char *name)
5197 {
5198     return namex(path, 1, name);
5199 }

```

```

5200 //
5201 // File descriptors
5202 //
5203
5204 #include "types.h"
5205 #include "defs.h"
5206 #include "param.h"
5207 #include "fs.h"
5208 #include "file.h"
5209 #include "spinlock.h"
5210
5211 struct devsw devsw[NDEV];
5212 struct {
5213   struct spinlock lock;
5214   struct file file[NFILE];
5215 } ftable;
5216
5217 void
5218 fileinit(void)
5219 {
5220   initlock(&ftable.lock, "ftable");
5221 }
5222
5223 // Allocate a file structure.
5224 struct file*
5225 filealloc(void)
5226 {
5227   struct file *f;
5228
5229   acquire(&ftable.lock);
5230   for(f = ftable.file; f < ftable.file + NFILE; f++){
5231     if(f->ref == 0){
5232       f->ref = 1;
5233       release(&ftable.lock);
5234       return f;
5235     }
5236   }
5237   release(&ftable.lock);
5238   return 0;
5239 }
5240
5241
5242
5243
5244
5245
5246
5247
5248
5249

```

```

5250 // Increment ref count for file f.
5251 struct file*
5252 filedup(struct file *f)
5253 {
5254   acquire(&ftable.lock);
5255   if(f->ref < 1)
5256     panic("filedup");
5257   f->ref++;
5258   release(&ftable.lock);
5259   return f;
5260 }
5261
5262 // Close file f. (Decrement ref count, close when reaches 0.)
5263 void
5264 fileclose(struct file *f)
5265 {
5266   struct file ff;
5267
5268   acquire(&ftable.lock);
5269   if(f->ref < 1)
5270     panic("fileclose");
5271   if(--f->ref > 0){
5272     release(&ftable.lock);
5273     return;
5274   }
5275   ff = *f;
5276   f->ref = 0;
5277   f->type = FD_NONE;
5278   release(&ftable.lock);
5279
5280   if(ff.type == FD_PIPE)
5281     pipeclose(ff.pipe, ff.writable);
5282   else if(ff.type == FD_INODE){
5283     begin_trans();
5284     iput(ff.ip);
5285     commit_trans();
5286   }
5287 }
5288
5289
5290
5291
5292
5293
5294
5295
5296
5297
5298
5299

```

```

5300 // Get metadata about file f.
5301 int
5302 filestat(struct file *f, struct stat *st)
5303 {
5304     if(f->type == FD_INODE){
5305         ilock(f->ip);
5306         stati(f->ip, st);
5307         iunlock(f->ip);
5308         return 0;
5309     }
5310     return -1;
5311 }
5312
5313 // Read from file f.
5314 int
5315 fileread(struct file *f, char *addr, int n)
5316 {
5317     int r;
5318
5319     if(f->readable == 0)
5320         return -1;
5321     if(f->type == FD_PIPE)
5322         return piperead(f->pipe, addr, n);
5323     if(f->type == FD_INODE){
5324         ilock(f->ip);
5325         if((r = readi(f->ip, addr, f->off, n)) > 0)
5326             f->off += r;
5327         iunlock(f->ip);
5328         return r;
5329     }
5330     panic("fileread");
5331 }
5332
5333
5334
5335
5336
5337
5338
5339
5340
5341
5342
5343
5344
5345
5346
5347
5348
5349

```

```

5350 // Write to file f.
5351 int
5352 filewrite(struct file *f, char *addr, int n)
5353 {
5354     int r;
5355
5356     if(f->writable == 0)
5357         return -1;
5358     if(f->type == FD_PIPE)
5359         return pipewrite(f->pipe, addr, n);
5360     if(f->type == FD_INODE){
5361         // write a few blocks at a time to avoid exceeding
5362         // the maximum log transaction size, including
5363         // i-node, indirect block, allocation blocks,
5364         // and 2 blocks of slop for non-aligned writes.
5365         // this really belongs lower down, since writei()
5366         // might be writing a device like the console.
5367         int max = ((LOGSIZE-1-1-2) / 2) * 512;
5368         int i = 0;
5369         while(i < n){
5370             int n1 = n - i;
5371             if(n1 > max)
5372                 n1 = max;
5373
5374             begin_trans();
5375             ilock(f->ip);
5376             if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
5377                 f->off += r;
5378             iunlock(f->ip);
5379             commit_trans();
5380
5381             if(r < 0)
5382                 break;
5383             if(r != n1)
5384                 panic("short filewrite");
5385             i += r;
5386         }
5387         return i == n ? n : -1;
5388     }
5389     panic("filewrite");
5390 }
5391
5392
5393
5394
5395
5396
5397
5398
5399

```

```

5400 //
5401 // File-system system calls.
5402 // Mostly argument checking, since we don't trust
5403 // user code, and calls into file.c and fs.c.
5404 //
5405
5406 #include "types.h"
5407 #include "defs.h"
5408 #include "param.h"
5409 #include "stat.h"
5410 #include "mmu.h"
5411 #include "proc.h"
5412 #include "fs.h"
5413 #include "file.h"
5414 #include "fcntl.h"
5415
5416 // Fetch the nth word-sized system call argument as a file descriptor
5417 // and return both the descriptor and the corresponding struct file.
5418 static int
5419 argfd(int n, int *pfd, struct file **pf)
5420 {
5421     int fd;
5422     struct file *f;
5423
5424     if(argint(n, &fd) < 0)
5425         return -1;
5426     if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
5427         return -1;
5428     if(pfd)
5429         *pfd = fd;
5430     if(pf)
5431         *pf = f;
5432     return 0;
5433 }
5434
5435 // Allocate a file descriptor for the given file.
5436 // Takes over file reference from caller on success.
5437 static int
5438 fdalloc(struct file *f)
5439 {
5440     int fd;
5441
5442     for(fd = 0; fd < NOFILE; fd++){
5443         if(proc->ofile[fd] == 0){
5444             proc->ofile[fd] = f;
5445             return fd;
5446         }
5447     }
5448     return -1;
5449 }

```

```

5450 int
5451 sys_dup(void)
5452 {
5453     struct file *f;
5454     int fd;
5455
5456     if(argfd(0, 0, &f) < 0)
5457         return -1;
5458     if((fd=fdalloc(f)) < 0)
5459         return -1;
5460     filedup(f);
5461     return fd;
5462 }
5463
5464 int
5465 sys_read(void)
5466 {
5467     struct file *f;
5468     int n;
5469     char *p;
5470
5471     if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5472         return -1;
5473     return fileread(f, p, n);
5474 }
5475
5476 int
5477 sys_write(void)
5478 {
5479     struct file *f;
5480     int n;
5481     char *p;
5482
5483     if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5484         return -1;
5485     return filewrite(f, p, n);
5486 }
5487
5488 int
5489 sys_close(void)
5490 {
5491     int fd;
5492     struct file *f;
5493
5494     if(argfd(0, &fd, &f) < 0)
5495         return -1;
5496     proc->ofile[fd] = 0;
5497     fileclose(f);
5498     return 0;
5499 }

```

```

5500 int
5501 sys_fstat(void)
5502 {
5503     struct file *f;
5504     struct stat *st;
5505
5506     if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
5507         return -1;
5508     return filestat(f, st);
5509 }
5510
5511 // Create the path new as a link to the same inode as old.
5512 int
5513 sys_link(void)
5514 {
5515     char name[DIRSIZ], *new, *old;
5516     struct inode *dp, *ip;
5517
5518     if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
5519         return -1;
5520     if((ip = namei(old)) == 0)
5521         return -1;
5522
5523     begin_trans();
5524
5525     ilock(ip);
5526     if(ip->type == T_DIR){
5527         iunlockput(ip);
5528         commit_trans();
5529         return -1;
5530     }
5531
5532     ip->nlink++;
5533     iupdate(ip);
5534     iunlock(ip);
5535
5536     if((dp = nameiparent(new, name)) == 0)
5537         goto bad;
5538     ilock(dp);
5539     if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
5540         iunlockput(dp);
5541         goto bad;
5542     }
5543     iunlockput(dp);
5544     iput(ip);
5545
5546     commit_trans();
5547
5548     return 0;
5549

```

```

5550 bad:
5551     ilock(ip);
5552     ip->nlink--;
5553     iupdate(ip);
5554     iunlockput(ip);
5555     commit_trans();
5556     return -1;
5557 }
5558
5559 // Is the directory dp empty except for "." and ".." ?
5560 static int
5561 isdirempty(struct inode *dp)
5562 {
5563     int off;
5564     struct dirent de;
5565
5566     for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
5567         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5568             panic("isdirempty: readi");
5569         if(de.inum != 0)
5570             return 0;
5571     }
5572     return 1;
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 int
5601 sys_unlink(void)
5602 {
5603     struct inode *ip, *dp;
5604     struct dirent de;
5605     char name[DIRSIZ], *path;
5606     uint off;
5607
5608     if(argstr(0, &path) < 0)
5609         return -1;
5610     if((dp = nameiparent(path, name)) == 0)
5611         return -1;
5612
5613     begin_trans();
5614
5615     ilock(dp);
5616
5617     // Cannot unlink "." or "..".
5618     if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
5619         goto bad;
5620
5621     if((ip = dirlookup(dp, name, &off)) == 0)
5622         goto bad;
5623     ilock(ip);
5624
5625     if(ip->nlink < 1)
5626         panic("unlink: nlink < 1");
5627     if(ip->type == T_DIR && !isdirempty(ip)){
5628         iunlockput(ip);
5629         goto bad;
5630     }
5631
5632     memset(&de, 0, sizeof(de));
5633     if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
5634         panic("unlink: writei");
5635     if(ip->type == T_DIR){
5636         dp->nlink--;
5637         iupdate(dp);
5638     }
5639     iunlockput(dp);
5640
5641     ip->nlink--;
5642     iupdate(ip);
5643     iunlockput(ip);
5644
5645     commit_trans();
5646
5647     return 0;
5648
5649

```

```

5650 bad:
5651     iunlockput(dp);
5652     commit_trans();
5653     return -1;
5654 }
5655
5656 static struct inode*
5657 create(char *path, short type, short major, short minor)
5658 {
5659     uint off;
5660     struct inode *ip, *dp;
5661     char name[DIRSIZ];
5662
5663     if((dp = nameiparent(path, name)) == 0)
5664         return 0;
5665     ilock(dp);
5666
5667     if((ip = dirlookup(dp, name, &off)) != 0){
5668         iunlockput(dp);
5669         ilock(ip);
5670         if(type == T_FILE && ip->type == T_FILE)
5671             return ip;
5672         iunlockput(ip);
5673         return 0;
5674     }
5675
5676     if((ip = ialloc(dp->dev, type)) == 0)
5677         panic("create: ialloc");
5678
5679     ilock(ip);
5680     ip->major = major;
5681     ip->minor = minor;
5682     ip->nlink = 1;
5683     iupdate(ip);
5684
5685     if(type == T_DIR){ // Create . and .. entries.
5686         dp->nlink++; // for ".."
5687         iupdate(dp);
5688         // No ip->nlink++ for ".": avoid cyclic ref count.
5689         if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
5690             panic("create dots");
5691     }
5692
5693     if(dirlink(dp, name, ip->inum) < 0)
5694         panic("create: dirlink");
5695
5696     iunlockput(dp);
5697
5698     return ip;
5699 }

```

```

5700 int
5701 sys_open(void)
5702 {
5703     char *path;
5704     int fd, omode;
5705     struct file *f;
5706     struct inode *ip;
5707
5708     if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
5709         return -1;
5710     if(omode & O_CREATE){
5711         begin_trans();
5712         ip = create(path, T_FILE, 0, 0);
5713         commit_trans();
5714         if(ip == 0)
5715             return -1;
5716     } else {
5717         if((ip = namei(path)) == 0)
5718             return -1;
5719         ilock(ip);
5720         if(ip->type == T_DIR && omode != O_RDONLY){
5721             iunlockput(ip);
5722             return -1;
5723         }
5724     }
5725
5726     if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
5727         if(f)
5728             fileclose(f);
5729         iunlockput(ip);
5730         return -1;
5731     }
5732     iunlock(ip);
5733
5734     f->type = FD_INODE;
5735     f->ip = ip;
5736     f->off = 0;
5737     f->readable = !(omode & O_WRONLY);
5738     f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
5739     return fd;
5740 }
5741
5742
5743
5744
5745
5746
5747
5748
5749

```

```

5750 int
5751 sys_mkdir(void)
5752 {
5753     char *path;
5754     struct inode *ip;
5755
5756     begin_trans();
5757     if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
5758         commit_trans();
5759         return -1;
5760     }
5761     iunlockput(ip);
5762     commit_trans();
5763     return 0;
5764 }
5765
5766 int
5767 sys_mknod(void)
5768 {
5769     struct inode *ip;
5770     char *path;
5771     int len;
5772     int major, minor;
5773
5774     begin_trans();
5775     if((len=argstr(0, &path)) < 0 ||
5776         argint(1, &major) < 0 ||
5777         argint(2, &minor) < 0 ||
5778         (ip = create(path, T_DEV, major, minor)) == 0){
5779         commit_trans();
5780         return -1;
5781     }
5782     iunlockput(ip);
5783     commit_trans();
5784     return 0;
5785 }
5786
5787
5788
5789
5790
5791
5792
5793
5794
5795
5796
5797
5798
5799

```



```

5800 int
5801 sys_chdir(void)
5802 {
5803     char *path;
5804     struct inode *ip;
5805
5806     if(argstr(0, &path) < 0 || (ip = namei(path)) == 0)
5807         return -1;
5808     ilock(ip);
5809     if(ip->type != T_DIR){
5810         iunlockput(ip);
5811         return -1;
5812     }
5813     iunlock(ip);
5814     iput(proc->cwd);
5815     proc->cwd = ip;
5816     return 0;
5817 }
5818
5819 int
5820 sys_exec(void)
5821 {
5822     char *path, *argv[MAXARG];
5823     int i;
5824     uint uargv, uarg;
5825
5826     if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
5827         return -1;
5828     }
5829     memset(argv, 0, sizeof(argv));
5830     for(i=0;; i++){
5831         if(i >= NELEM(argv))
5832             return -1;
5833         if(fetchint(uargv+4*i, (int*)&uarg) < 0)
5834             return -1;
5835         if(uarg == 0){
5836             argv[i] = 0;
5837             break;
5838         }
5839         if(fetchstr(uarg, &argv[i]) < 0)
5840             return -1;
5841     }
5842     return exec(path, argv);
5843 }
5844
5845
5846
5847
5848
5849

```

```

5850 int
5851 sys_pipe(void)
5852 {
5853     int *fd;
5854     struct file *rf, *wf;
5855     int fd0, fd1;
5856
5857     if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
5858         return -1;
5859     if(pipealloc(&rf, &wf) < 0)
5860         return -1;
5861     fd0 = -1;
5862     if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
5863         if(fd0 >= 0)
5864             proc->ofile[fd0] = 0;
5865         fileclose(rf);
5866         fileclose(wf);
5867         return -1;
5868     }
5869     fd[0] = fd0;
5870     fd[1] = fd1;
5871     return 0;
5872 }
5873
5874
5875
5876
5877
5878
5879
5880
5881
5882
5883
5884
5885
5886
5887
5888
5889
5890
5891
5892
5893
5894
5895
5896
5897
5898
5899

```

```

5900 #include "types.h"
5901 #include "param.h"
5902 #include "memlayout.h"
5903 #include "mmu.h"
5904 #include "proc.h"
5905 #include "defs.h"
5906 #include "x86.h"
5907 #include "elf.h"
5908
5909 int
5910 exec(char *path, char **argv)
5911 {
5912     char *s, *last;
5913     int i, off;
5914     uint argc, sz, sp, ustack[3+MAXARG+1];
5915     struct elfhdr elf;
5916     struct inode *ip;
5917     struct proghdr ph;
5918     pde_t *pgdir, *oldpgdir;
5919
5920     if((ip = namei(path)) == 0)
5921         return -1;
5922     ilock(ip);
5923     pgdir = 0;
5924
5925     // Check ELF header
5926     if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
5927         goto bad;
5928     if(elf.magic != ELF_MAGIC)
5929         goto bad;
5930
5931     if((pgdir = setupkvm()) == 0)
5932         goto bad;
5933
5934     // Load program into memory.
5935     sz = 0;
5936     for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
5937         if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
5938             goto bad;
5939         if(ph.type != ELF_PROG_LOAD)
5940             continue;
5941         if(ph.memsz < ph.filesz)
5942             goto bad;
5943         if((sz = allocvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
5944             goto bad;
5945         if(loadvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
5946             goto bad;
5947     }
5948     iunlockput(ip);
5949     ip = 0;

```

```

5950 // Allocate two pages at the next page boundary.
5951 // Make the first inaccessible. Use the second as the user stack.
5952 sz = PGROUNDUP(sz);
5953 if((sz = allocvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
5954     goto bad;
5955 clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
5956 sp = sz;
5957
5958 // Push argument strings, prepare rest of stack in ustack.
5959 for(argc = 0; argv[argc]; argc++) {
5960     if(argc >= MAXARG)
5961         goto bad;
5962     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
5963     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
5964         goto bad;
5965     ustack[3+argc] = sp;
5966 }
5967 ustack[3+argc] = 0;
5968
5969 ustack[0] = 0xffffffff; // fake return PC
5970 ustack[1] = argc;
5971 ustack[2] = sp - (argc+1)*4; // argv pointer
5972
5973 sp -= (3+argc+1) * 4;
5974 if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
5975     goto bad;
5976
5977 // Save program name for debugging.
5978 for(last=s=path; *s; s++)
5979     if(*s == '/')
5980         last = s+1;
5981 safestrcpy(proc->name, last, sizeof(proc->name));
5982
5983 // Commit to the user image.
5984 oldpgdir = proc->pgdir;
5985 proc->pgdir = pgdir;
5986 proc->sz = sz;
5987 proc->tf->eip = elf.entry; // main
5988 proc->tf->esp = sp;
5989 switchvm(proc);
5990 freevm(oldpgdir);
5991 return 0;
5992
5993 bad:
5994 if(pgdir)
5995     freevm(pgdir);
5996 if(ip)
5997     iunlockput(ip);
5998 return -1;
5999 }

```

```

6000 #include "types.h"
6001 #include "defs.h"
6002 #include "param.h"
6003 #include "mmu.h"
6004 #include "proc.h"
6005 #include "fs.h"
6006 #include "file.h"
6007 #include "spinlock.h"
6008
6009 #define PIPESIZE 512
6010
6011 struct pipe {
6012     struct spinlock lock;
6013     char data[PIPESIZE];
6014     uint nread;    // number of bytes read
6015     uint nwrite;   // number of bytes written
6016     int readopen;  // read fd is still open
6017     int writeopen; // write fd is still open
6018 };
6019
6020 int
6021 pipealloc(struct file **f0, struct file **f1)
6022 {
6023     struct pipe *p;
6024
6025     p = 0;
6026     *f0 = *f1 = 0;
6027     if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
6028         goto bad;
6029     if((p = (struct pipe*)kalloc()) == 0)
6030         goto bad;
6031     p->readopen = 1;
6032     p->writeopen = 1;
6033     p->nwrite = 0;
6034     p->nread = 0;
6035     initlock(&p->lock, "pipe");
6036     (*f0)->type = FD_PIPE;
6037     (*f0)->readable = 1;
6038     (*f0)->writable = 0;
6039     (*f0)->pipe = p;
6040     (*f1)->type = FD_PIPE;
6041     (*f1)->readable = 0;
6042     (*f1)->writable = 1;
6043     (*f1)->pipe = p;
6044     return 0;
6045
6046
6047
6048
6049

```

```

6050 bad:
6051     if(p)
6052         kfree((char*)p);
6053     if(*f0)
6054         fileclose(*f0);
6055     if(*f1)
6056         fileclose(*f1);
6057     return -1;
6058 }
6059
6060 void
6061 pipeclose(struct pipe *p, int writable)
6062 {
6063     acquire(&p->lock);
6064     if(writable){
6065         p->writeopen = 0;
6066         wakeup(&p->nread);
6067     } else {
6068         p->readopen = 0;
6069         wakeup(&p->nwrite);
6070     }
6071     if(p->readopen == 0 && p->writeopen == 0){
6072         release(&p->lock);
6073         kfree((char*)p);
6074     } else
6075         release(&p->lock);
6076 }
6077
6078 int
6079 pipewrite(struct pipe *p, char *addr, int n)
6080 {
6081     int i;
6082
6083     acquire(&p->lock);
6084     for(i = 0; i < n; i++){
6085         while(p->nwrite == p->nread + PIPESIZE){
6086             if(p->readopen == 0 || proc->killed){
6087                 release(&p->lock);
6088                 return -1;
6089             }
6090             wakeup(&p->nread);
6091             sleep(&p->nwrite, &p->lock);
6092         }
6093         p->data[p->nwrite++ % PIPESIZE] = addr[i];
6094     }
6095     wakeup(&p->nread);
6096     release(&p->lock);
6097     return n;
6098 }
6099

```

```

6100 int
6101 piperead(struct pipe *p, char *addr, int n)
6102 {
6103     int i;
6104
6105     acquire(&p->lock);
6106     while(p->nread == p->nwrite && p->writeopen){
6107         if(proc->killed){
6108             release(&p->lock);
6109             return -1;
6110         }
6111         sleep(&p->nread, &p->lock);
6112     }
6113     for(i = 0; i < n; i++){
6114         if(p->nread == p->nwrite)
6115             break;
6116         addr[i] = p->data[p->nread++ % PIPESIZE];
6117     }
6118     wakeup(&p->nwrite);
6119     release(&p->lock);
6120     return i;
6121 }
6122
6123
6124
6125
6126
6127
6128
6129
6130
6131
6132
6133
6134
6135
6136
6137
6138
6139
6140
6141
6142
6143
6144
6145
6146
6147
6148
6149

```

```

6150 #include "types.h"
6151 #include "x86.h"
6152
6153 void*
6154 memset(void *dst, int c, uint n)
6155 {
6156     if ((int)dst%4 == 0 && n%4 == 0){
6157         c &= 0xFF;
6158         stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
6159     } else
6160         stosb(dst, c, n);
6161     return dst;
6162 }
6163
6164 int
6165 memcmp(const void *v1, const void *v2, uint n)
6166 {
6167     const uchar *s1, *s2;
6168
6169     s1 = v1;
6170     s2 = v2;
6171     while(n-- > 0){
6172         if(*s1 != *s2)
6173             return *s1 - *s2;
6174         s1++, s2++;
6175     }
6176
6177     return 0;
6178 }
6179
6180 void*
6181 memmove(void *dst, const void *src, uint n)
6182 {
6183     const char *s;
6184     char *d;
6185
6186     s = src;
6187     d = dst;
6188     if(s < d && s + n > d){
6189         s += n;
6190         d += n;
6191         while(n-- > 0)
6192             *--d = *--s;
6193     } else
6194         while(n-- > 0)
6195             *d++ = *s++;
6196
6197     return dst;
6198 }
6199

```

```

6200 // memcpy exists to placate GCC. Use memmove.
6201 void*
6202 memcpy(void *dst, const void *src, uint n)
6203 {
6204     return memmove(dst, src, n);
6205 }
6206
6207 int
6208 strncmp(const char *p, const char *q, uint n)
6209 {
6210     while(n > 0 && *p && *p == *q)
6211         n--, p++, q++;
6212     if(n == 0)
6213         return 0;
6214     return (uchar)*p - (uchar)*q;
6215 }
6216
6217 char*
6218 strncpy(char *s, const char *t, int n)
6219 {
6220     char *os;
6221
6222     os = s;
6223     while(n-- > 0 && (*s++ = *t++) != 0)
6224         ;
6225     while(n-- > 0)
6226         *s++ = 0;
6227     return os;
6228 }
6229
6230 // Like strncpy but guaranteed to NUL-terminate.
6231 char*
6232 safestrcpy(char *s, const char *t, int n)
6233 {
6234     char *os;
6235
6236     os = s;
6237     if(n <= 0)
6238         return os;
6239     while(--n > 0 && (*s++ = *t++) != 0)
6240         ;
6241     *s = 0;
6242     return os;
6243 }
6244
6245
6246
6247
6248
6249

```

```

6250 int
6251 strlen(const char *s)
6252 {
6253     int n;
6254
6255     for(n = 0; s[n]; n++)
6256         ;
6257     return n;
6258 }
6259
6260
6261
6262
6263
6264
6265
6266
6267
6268
6269
6270
6271
6272
6273
6274
6275
6276
6277
6278
6279
6280
6281
6282
6283
6284
6285
6286
6287
6288
6289
6290
6291
6292
6293
6294
6295
6296
6297
6298
6299

```

```

6300 // See MultiProcessor Specification Version 1.[14]
6301
6302 struct mp {          // floating pointer
6303     uchar signature[4];    // "_MP_"
6304     void *physaddr;        // phys addr of MP config table
6305     uchar length;          // 1
6306     uchar specrev;         // [14]
6307     uchar checksum;        // all bytes must add up to 0
6308     uchar type;            // MP system config type
6309     uchar imcrp;
6310     uchar reserved[3];
6311 };
6312
6313 struct mpconf {       // configuration table header
6314     uchar signature[4];    // "PCMP"
6315     ushort length;         // total table length
6316     uchar version;         // [14]
6317     uchar checksum;        // all bytes must add up to 0
6318     uchar product[20];     // product id
6319     uint *oemtable;        // OEM table pointer
6320     ushort oemlength;      // OEM table length
6321     ushort entry;          // entry count
6322     uint *lapicaddr;       // address of local APIC
6323     ushort xlength;        // extended table length
6324     uchar xchecksum;       // extended table checksum
6325     uchar reserved;
6326 };
6327
6328 struct mpproc {        // processor table entry
6329     uchar type;            // entry type (0)
6330     uchar apicid;          // local APIC id
6331     uchar version;         // local APIC verison
6332     uchar flags;           // CPU flags
6333     #define MPBOOT 0x02    // This proc is the bootstrap processor.
6334     uchar signature[4];    // CPU signature
6335     uint feature;          // feature flags from CPUID instruction
6336     uchar reserved[8];
6337 };
6338
6339 struct mpioapic {      // I/O APIC table entry
6340     uchar type;           // entry type (2)
6341     uchar apicno;         // I/O APIC id
6342     uchar version;        // I/O APIC version
6343     uchar flags;          // I/O APIC flags
6344     uint *addr;           // I/O APIC address
6345 };
6346
6347
6348
6349

```

```

6350 // Table entry types
6351 #define MPPROC 0x00 // One per processor
6352 #define MPBUS 0x01 // One per bus
6353 #define MPIOAPIC 0x02 // One per I/O APIC
6354 #define MPIOINTR 0x03 // One per bus interrupt source
6355 #define MPLINTR 0x04 // One per system interrupt source
6356
6357
6358
6359
6360
6361
6362
6363
6364
6365
6366
6367
6368
6369
6370
6371
6372
6373
6374
6375
6376
6377
6378
6379
6380
6381
6382
6383
6384
6385
6386
6387
6388
6389
6390
6391
6392
6393
6394
6395
6396
6397
6398
6399

```

```

6400 // Multiprocessor support
6401 // Search memory for MP description structures.
6402 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
6403
6404 #include "types.h"
6405 #include "defs.h"
6406 #include "param.h"
6407 #include "memlayout.h"
6408 #include "mp.h"
6409 #include "x86.h"
6410 #include "mmu.h"
6411 #include "proc.h"
6412
6413 struct cpu cpus[NCPU];
6414 static struct cpu *bcpu;
6415 int ismp;
6416 int ncpu;
6417 uchar ioapicid;
6418
6419 int
6420 mpbcpu(void)
6421 {
6422     return bcpu-cpus;
6423 }
6424
6425 static uchar
6426 sum(uchar *addr, int len)
6427 {
6428     int i, sum;
6429
6430     sum = 0;
6431     for(i=0; i<len; i++)
6432         sum += addr[i];
6433     return sum;
6434 }
6435
6436 // Look for an MP structure in the len bytes at addr.
6437 static struct mp*
6438 mpsearch1(uint a, int len)
6439 {
6440     uchar *e, *p, *addr;
6441
6442     addr = p2v(a);
6443     e = addr+len;
6444     for(p = addr; p < e; p += sizeof(struct mp))
6445         if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
6446             return (struct mp*)p;
6447     return 0;
6448 }
6449

```

```

6450 // Search for the MP Floating Pointer Structure, which according to the
6451 // spec is in one of the following three locations:
6452 // 1) in the first KB of the EBDA;
6453 // 2) in the last KB of system base memory;
6454 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
6455 static struct mp*
6456 mpsearch(void)
6457 {
6458     uchar *bda;
6459     uint p;
6460     struct mp *mp;
6461
6462     bda = (uchar *) P2V(0x400);
6463     if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
6464         if((mp = mpsearch1(p, 1024)))
6465             return mp;
6466     } else {
6467         p = ((bda[0x14]<<8)| bda[0x13])*1024;
6468         if((mp = mpsearch1(p-1024, 1024)))
6469             return mp;
6470     }
6471     return mpsearch1(0xF0000, 0x10000);
6472 }
6473
6474 // Search for an MP configuration table. For now,
6475 // don't accept the default configurations (physaddr == 0).
6476 // Check for correct signature, calculate the checksum and,
6477 // if correct, check the version.
6478 // To do: check extended table checksum.
6479 static struct mpconf*
6480 mpconfig(struct mp **pmp)
6481 {
6482     struct mpconf *conf;
6483     struct mp *mp;
6484
6485     if((mp = mpsearch()) == 0 || mp->physaddr == 0)
6486         return 0;
6487     conf = (struct mpconf*) p2v((uint) mp->physaddr);
6488     if(memcmp(conf, "PCMP", 4) != 0)
6489         return 0;
6490     if(conf->version != 1 && conf->version != 4)
6491         return 0;
6492     if(sum((uchar*)conf, conf->length) != 0)
6493         return 0;
6494     *pmp = mp;
6495     return conf;
6496 }
6497
6498
6499

```

```

6500 void
6501 mpinit(void)
6502 {
6503     uchar *p, *e;
6504     struct mp *mp;
6505     struct mpconf *conf;
6506     struct mpproc *proc;
6507     struct mpioapic *ioapic;
6508
6509     bcpu = &cpus[0];
6510     if((conf = mpconfig(&mp)) == 0)
6511         return;
6512     ismp = 1;
6513     lapic = (uint*)conf->lapicaddr;
6514     for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
6515         switch(*p){
6516             case MPPROC:
6517                 proc = (struct mpproc*)p;
6518                 if(ncpu != proc->apicid){
6519                     cprintf("mpinit: ncpu=%d apicid=%d\n", ncpu, proc->apicid);
6520                     ismp = 0;
6521                 }
6522                 if(proc->flags & MPBOOT)
6523                     bcpu = &cpus[ncpu];
6524                 cpus[ncpu].id = ncpu;
6525                 ncpu++;
6526                 p += sizeof(struct mpproc);
6527                 continue;
6528             case MPIOAPIC:
6529                 ioapic = (struct mpioapic*)p;
6530                 ioapicid = ioapic->apicno;
6531                 p += sizeof(struct mpioapic);
6532                 continue;
6533             case MPBUS:
6534             case MPIOINTR:
6535             case MPLINTR:
6536                 p += 8;
6537                 continue;
6538             default:
6539                 cprintf("mpinit: unknown config type %x\n", *p);
6540                 ismp = 0;
6541         }
6542     }
6543     if(!ismp){
6544         // Didn't like what we found; fall back to no MP.
6545         ncpu = 1;
6546         lapic = 0;
6547         ioapicid = 0;
6548         return;
6549     }

```

```

6550     if(mp->imcrp){
6551         // Bochs doesn't support IMCR, so this doesn't run on Bochs.
6552         // But it would on real hardware.
6553         outb(0x22, 0x70); // Select IMCR
6554         outb(0x23, inb(0x23) | 1); // Mask external interrupts.
6555     }
6556 }
6557
6558
6559
6560
6561
6562
6563
6564
6565
6566
6567
6568
6569
6570
6571
6572
6573
6574
6575
6576
6577
6578
6579
6580
6581
6582
6583
6584
6585
6586
6587
6588
6589
6590
6591
6592
6593
6594
6595
6596
6597
6598
6599

```



```

6600 // The local APIC manages internal (non-I/O) interrupts.
6601 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
6602
6603 #include "types.h"
6604 #include "defs.h"
6605 #include "memlayout.h"
6606 #include "traps.h"
6607 #include "mmu.h"
6608 #include "x86.h"
6609
6610 // Local APIC registers, divided by 4 for use as uint[] indices.
6611 #define ID      (0x0020/4) // ID
6612 #define VER     (0x0030/4) // Version
6613 #define TPR     (0x0080/4) // Task Priority
6614 #define EOI     (0x00B0/4) // EOI
6615 #define SVR     (0x00F0/4) // Spurious Interrupt Vector
6616 #define ENABLE  (0x00000100 // Unit Enable
6617 #define ESR     (0x0280/4) // Error Status
6618 #define ICRLO   (0x0300/4) // Interrupt Command
6619 #define INIT     0x00000500 // INIT/RESET
6620 #define STARTUP 0x00000600 // Startup IPI
6621 #define DELIVS   0x00001000 // Delivery status
6622 #define ASSERT   0x00004000 // Assert interrupt (vs deassert)
6623 #define DEASSERT 0x00000000
6624 #define LEVEL    0x00008000 // Level triggered
6625 #define BCAST    0x00080000 // Send to all APICs, including self.
6626 #define BUSY     0x00001000
6627 #define FIXED     0x00000000
6628 #define ICRHI   (0x0310/4) // Interrupt Command [63:32]
6629 #define TIMER   (0x0320/4) // Local Vector Table 0 (TIMER)
6630 #define X1      0x0000000B // divide counts by 1
6631 #define PERIODIC 0x00020000 // Periodic
6632 #define PCINT    (0x0340/4) // Performance Counter LVT
6633 #define LINT0    (0x0350/4) // Local Vector Table 1 (LINT0)
6634 #define LINT1    (0x0360/4) // Local Vector Table 2 (LINT1)
6635 #define ERROR    (0x0370/4) // Local Vector Table 3 (ERROR)
6636 #define MASKED   0x00010000 // Interrupt masked
6637 #define TICR     (0x0380/4) // Timer Initial Count
6638 #define TCCR     (0x0390/4) // Timer Current Count
6639 #define TDCR     (0x03E0/4) // Timer Divide Configuration
6640
6641 volatile uint *lapic; // Initialized in mp.c
6642
6643 static void
6644 lapicw(int index, int value)
6645 {
6646     lapic[index] = value;
6647     lapic[ID]; // wait for write to finish, by reading
6648 }
6649

```

```

6650 void
6651 lapicinit(void)
6652 {
6653     if(!lapic)
6654         return;
6655
6656     // Enable local APIC; set spurious interrupt vector.
6657     lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
6658
6659     // The timer repeatedly counts down at bus frequency
6660     // from lapic[TICR] and then issues an interrupt.
6661     // If xv6 cared more about precise timekeeping,
6662     // TICR would be calibrated using an external time source.
6663     lapicw(TDCR, X1);
6664     lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
6665     lapicw(TICR, 10000000);
6666
6667     // Disable logical interrupt lines.
6668     lapicw(LINT0, MASKED);
6669     lapicw(LINT1, MASKED);
6670
6671     // Disable performance counter overflow interrupts
6672     // on machines that provide that interrupt entry.
6673     if(((lapic[VER]>>16) & 0xFF) >= 4)
6674         lapicw(PCINT, MASKED);
6675
6676     // Map error interrupt to IRQ_ERROR.
6677     lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
6678
6679     // Clear error status register (requires back-to-back writes).
6680     lapicw(ESR, 0);
6681     lapicw(ESR, 0);
6682
6683     // Ack any outstanding interrupts.
6684     lapicw(EOI, 0);
6685
6686     // Send an Init Level De-Assert to synchronise arbitration ID's.
6687     lapicw(ICRHI, 0);
6688     lapicw(ICRLO, BCAST | INIT | LEVEL);
6689     while(lapic[ICRLO] & DELIVS)
6690         ;
6691
6692     // Enable interrupts on the APIC (but not on the processor).
6693     lapicw(TPR, 0);
6694 }
6695
6696
6697
6698
6699

```

```

6700 int
6701 cpunum(void)
6702 {
6703     // Cannot call cpu when interrupts are enabled:
6704     // result not guaranteed to last long enough to be used!
6705     // Would prefer to panic but even printing is chancy here:
6706     // almost everything, including cprintf and panic, calls cpu,
6707     // often indirectly through acquire and release.
6708     if(readeflags() & FL_IF){
6709         static int n;
6710         if(n++ == 0)
6711             cprintf("cpu called from %x with interrupts enabled\n",
6712                 __builtin_return_address(0));
6713     }
6714
6715     if(lapic)
6716         return lapic[ID]>>24;
6717     return 0;
6718 }
6719
6720 // Acknowledge interrupt.
6721 void
6722 lapiceoi(void)
6723 {
6724     if(lapic)
6725         lapicw(EOI, 0);
6726 }
6727
6728 // Spin for a given number of microseconds.
6729 // On real hardware would want to tune this dynamically.
6730 void
6731 microdelay(int us)
6732 {
6733 }
6734
6735 #define IO_RTC 0x70
6736
6737 // Start additional processor running entry code at addr.
6738 // See Appendix B of MultiProcessor Specification.
6739 void
6740 lapicstartap(uchar apicid, uint addr)
6741 {
6742     int i;
6743     ushort *wrv;
6744
6745     // "The BSP must initialize CMOS shutdown code to 0AH
6746     // and the warm reset vector (DWORD based at 40:67) to point at
6747     // the AP startup code prior to the [universal startup algorithm]."
6748     outb(IO_RTC, 0xF); // offset 0xF is shutdown code
6749     outb(IO_RTC+1, 0x0A);

```

```

6750     wrv = (ushort*)P2V((0x40<<4 | 0x67)); // Warm reset vector
6751     wrv[0] = 0;
6752     wrv[1] = addr >> 4;
6753
6754     // "Universal startup algorithm."
6755     // Send INIT (level-triggered) interrupt to reset other CPU.
6756     lapicw(ICRHI, apicid<<24);
6757     lapicw(ICRLO, INIT | LEVEL | ASSERT);
6758     microdelay(200);
6759     lapicw(ICRLO, INIT | LEVEL);
6760     microdelay(100); // should be 10ms, but too slow in Bochs!
6761
6762     // Send startup IPI (twice!) to enter code.
6763     // Regular hardware is supposed to only accept a STARTUP
6764     // when it is in the halted state due to an INIT. So the second
6765     // should be ignored, but it is part of the official Intel algorithm.
6766     // Bochs complains about the second one. Too bad for Bochs.
6767     for(i = 0; i < 2; i++){
6768         lapicw(ICRHI, apicid<<24);
6769         lapicw(ICRLO, STARTUP | (addr>>12));
6770         microdelay(200);
6771     }
6772 }
6773
6774
6775
6776
6777
6778
6779
6780
6781
6782
6783
6784
6785
6786
6787
6788
6789
6790
6791
6792
6793
6794
6795
6796
6797
6798
6799

```

```

6800 // The I/O APIC manages hardware interrupts for an SMP system.
6801 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
6802 // See also picirq.c.
6803
6804 #include "types.h"
6805 #include "defs.h"
6806 #include "traps.h"
6807
6808 #define IOAPIC 0xFEC00000 // Default physical address of IO APIC
6809
6810 #define REG_ID 0x00 // Register index: ID
6811 #define REG_VER 0x01 // Register index: version
6812 #define REG_TABLE 0x10 // Redirection table base
6813
6814 // The redirection table starts at REG_TABLE and uses
6815 // two registers to configure each interrupt.
6816 // The first (low) register in a pair contains configuration bits.
6817 // The second (high) register contains a bitmask telling which
6818 // CPUs can serve that interrupt.
6819 #define INT_DISABLED 0x00010000 // Interrupt disabled
6820 #define INT_LEVEL 0x00008000 // Level-triggered (vs edge-)
6821 #define INT_ACTIVELOW 0x00002000 // Active low (vs high)
6822 #define INT_LOGICAL 0x00000800 // Destination is CPU id (vs APIC ID)
6823
6824 volatile struct ioapic *ioapic;
6825
6826 // IO APIC MMIO structure: write reg, then read or write data.
6827 struct ioapic {
6828     uint reg;
6829     uint pad[3];
6830     uint data;
6831 };
6832
6833 static uint
6834 ioapicread(int reg)
6835 {
6836     ioapic->reg = reg;
6837     return ioapic->data;
6838 }
6839
6840 static void
6841 ioapicwrite(int reg, uint data)
6842 {
6843     ioapic->reg = reg;
6844     ioapic->data = data;
6845 }
6846
6847
6848
6849

```

```

6850 void
6851 ioapicinit(void)
6852 {
6853     int i, id, maxintr;
6854
6855     if(!ismp)
6856         return;
6857
6858     ioapic = (volatile struct ioapic*)IOAPIC;
6859     maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
6860     id = ioapicread(REG_ID) >> 24;
6861     if(id != ioapicid)
6862         cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
6863
6864     // Mark all interrupts edge-triggered, active high, disabled,
6865     // and not routed to any CPUs.
6866     for(i = 0; i <= maxintr; i++){
6867         ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
6868         ioapicwrite(REG_TABLE+2*i+1, 0);
6869     }
6870 }
6871
6872 void
6873 ioapicenable(int irq, int cpunum)
6874 {
6875     if(!ismp)
6876         return;
6877
6878     // Mark interrupt edge-triggered, active high,
6879     // enabled, and routed to the given cpunum,
6880     // which happens to be that cpu's APIC ID.
6881     ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
6882     ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
6883 }
6884
6885
6886
6887
6888
6889
6890
6891
6892
6893
6894
6895
6896
6897
6898
6899

```

```

6900 // Intel 8259A programmable interrupt controllers.
6901
6902 #include "types.h"
6903 #include "x86.h"
6904 #include "traps.h"
6905
6906 // I/O Addresses of the two programmable interrupt controllers
6907 #define IO_PIC1      0x20    // Master (IRQs 0-7)
6908 #define IO_PIC2      0xA0    // Slave (IRQs 8-15)
6909
6910 #define IRQ_SLAVE     2      // IRQ at which slave connects to master
6911
6912 // Current IRQ mask.
6913 // Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
6914 static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);
6915
6916 static void
6917 picsetmask(ushort mask)
6918 {
6919     irqmask = mask;
6920     outb(IO_PIC1+1, mask);
6921     outb(IO_PIC2+1, mask >> 8);
6922 }
6923
6924 void
6925 picenable(int irq)
6926 {
6927     picsetmask(irqmask & ~(1<<irq));
6928 }
6929
6930 // Initialize the 8259A interrupt controllers.
6931 void
6932 picinit(void)
6933 {
6934     // mask all interrupts
6935     outb(IO_PIC1+1, 0xFF);
6936     outb(IO_PIC2+1, 0xFF);
6937
6938     // Set up master (8259A-1)
6939
6940     // ICW1: 0001g0hi
6941     //   g: 0 = edge triggering, 1 = level triggering
6942     //   h: 0 = cascaded PICs, 1 = master only
6943     //   i: 0 = no ICW4, 1 = ICW4 required
6944     outb(IO_PIC1, 0x11);
6945
6946     // ICW2: Vector offset
6947     outb(IO_PIC1+1, T_IRQ0);
6948
6949

```

```

6950 // ICW3: (master PIC) bit mask of IR lines connected to slaves
6951 //        (slave PIC) 3-bit # of slave's connection to master
6952 outb(IO_PIC1+1, 1<<IRQ_SLAVE);
6953
6954 // ICW4: 000nbmap
6955 //   n: 1 = special fully nested mode
6956 //   b: 1 = buffered mode
6957 //   m: 0 = slave PIC, 1 = master PIC
6958 //        (ignored when b is 0, as the master/slave role
6959 //        can be hardwired).
6960 //   a: 1 = Automatic EOI mode
6961 //   p: 0 = MCS-80/85 mode, 1 = intel x86 mode
6962 outb(IO_PIC1+1, 0x3);
6963
6964 // Set up slave (8259A-2)
6965 outb(IO_PIC2, 0x11);           // ICW1
6966 outb(IO_PIC2+1, T_IRQ0 + 8);   // ICW2
6967 outb(IO_PIC2+1, IRQ_SLAVE);    // ICW3
6968 // NB Automatic EOI mode doesn't tend to work on the slave.
6969 // Linux source code says it's "to be investigated".
6970 outb(IO_PIC2+1, 0x3);          // ICW4
6971
6972 // OCW3: 0ef01prs
6973 //   ef: 0x = NOP, 10 = clear specific mask, 11 = set specific mask
6974 //   p: 0 = no polling, 1 = polling mode
6975 //   rs: 0x = NOP, 10 = read IRR, 11 = read ISR
6976 outb(IO_PIC1, 0x68);           // clear specific mask
6977 outb(IO_PIC1, 0x0a);           // read IRR by default
6978
6979 outb(IO_PIC2, 0x68);           // OCW3
6980 outb(IO_PIC2, 0x0a);           // OCW3
6981
6982 if(irqmask != 0xFFFF)
6983     picsetmask(irqmask);
6984 }
6985
6986
6987
6988
6989
6990
6991
6992
6993
6994
6995
6996
6997
6998
6999

```

```

7000 // PC keyboard interface constants
7001
7002 #define KBSTATP      0x64    // kbd controller status port(I)
7003 #define KBS_DIB      0x01    // kbd data in buffer
7004 #define KBDATAP      0x60    // kbd data port(I)
7005
7006 #define NO            0
7007
7008 #define SHIFT         (1<<0)
7009 #define CTL           (1<<1)
7010 #define ALT           (1<<2)
7011
7012 #define CAPSLOCK      (1<<3)
7013 #define NUMLOCK      (1<<4)
7014 #define SCROLLLOCK   (1<<5)
7015
7016 #define E0ESC        (1<<6)
7017
7018 // Special keycodes
7019 #define KEY_HOME      0xE0
7020 #define KEY_END       0xE1
7021 #define KEY_UP        0xE2
7022 #define KEY_DN        0xE3
7023 #define KEY_LF        0xE4
7024 #define KEY_RT        0xE5
7025 #define KEY_PGUP      0xE6
7026 #define KEY_PGDN      0xE7
7027 #define KEY_INS       0xE8
7028 #define KEY_DEL       0xE9
7029
7030 // C('A') == Control-A
7031 #define C(x) (x - '@')
7032
7033 static uchar shiftcode[256] =
7034 {
7035     [0x1D] CTL,
7036     [0x2A] SHIFT,
7037     [0x36] SHIFT,
7038     [0x38] ALT,
7039     [0x9D] CTL,
7040     [0xB8] ALT
7041 };
7042
7043 static uchar togglecode[256] =
7044 {
7045     [0x3A] CAPSLOCK,
7046     [0x45] NUMLOCK,
7047     [0x46] SCROLLLOCK
7048 };
7049

```

```

7050 static uchar normalmap[256] =
7051 {
7052     NO,    0x1B, '1', '2', '3', '4', '5', '6', // 0x00
7053     '7', '8', '9', '0', '-', '=', '\b', '\t',
7054     'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', // 0x10
7055     'o', 'p', '[', ']', '\n', NO, 'a', 's',
7056     'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', // 0x20
7057     '\'', ',', NO, '\\', 'z', 'x', 'c', 'v',
7058     'b', 'n', 'm', ',', '.', '/', NO, '*', // 0x30
7059     NO, ' ', NO, NO, NO, NO, NO, NO,
7060     NO, NO, NO, NO, NO, NO, NO, '7', // 0x40
7061     '8', '9', '-', '4', '5', '6', '+', '1',
7062     '2', '3', '0', '.', NO, NO, NO, NO, // 0x50
7063     [0x9C] '\n', // KP_Enter
7064     [0xB5] '/', // KP_Div
7065     [0xC8] KEY_UP, [0xD0] KEY_DN,
7066     [0xC9] KEY_PGUP, [0xD1] KEY_PGDN,
7067     [0xCB] KEY_LF, [0xCD] KEY_RT,
7068     [0x97] KEY_HOME, [0xCF] KEY_END,
7069     [0xD2] KEY_INS, [0xD3] KEY_DEL
7070 };
7071
7072 static uchar shiftmap[256] =
7073 {
7074     NO,    033, '!', '@', '#', '$', '%', '^', // 0x00
7075     '&', '*', '(', ')', '-', '+', '\b', '\t',
7076     'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', // 0x10
7077     'O', 'P', '[', ']', '\n', NO, 'A', 'S',
7078     'D', 'F', 'G', 'H', 'J', 'K', 'L', ';', // 0x20
7079     '"', '~', NO, '|', 'Z', 'X', 'C', 'V',
7080     'B', 'N', 'M', '<', '>', '?', NO, '*', // 0x30
7081     NO, ' ', NO, NO, NO, NO, NO, NO,
7082     NO, NO, NO, NO, NO, NO, NO, '7', // 0x40
7083     '8', '9', '-', '4', '5', '6', '+', '1',
7084     '2', '3', '0', '.', NO, NO, NO, NO, // 0x50
7085     [0x9C] '\n', // KP_Enter
7086     [0xB5] '/', // KP_Div
7087     [0xC8] KEY_UP, [0xD0] KEY_DN,
7088     [0xC9] KEY_PGUP, [0xD1] KEY_PGDN,
7089     [0xCB] KEY_LF, [0xCD] KEY_RT,
7090     [0x97] KEY_HOME, [0xCF] KEY_END,
7091     [0xD2] KEY_INS, [0xD3] KEY_DEL
7092 };
7093
7094
7095
7096
7097
7098
7099

```

```

7100 static uchar ctlmap[256] =
7101 {
7102     NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7103     NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
7104     C('Q'), C('W'), C('E'), C('R'), C('T'), C('Y'), C('U'), C('I'),
7105     C('O'), C('P'), NO,      NO,      '\r', NO,      C('A'), C('S'),
7106     C('D'), C('F'), C('G'), C('H'), C('J'), C('K'), C('L'), NO,
7107     NO,      NO,      NO,      C('\\'), C('Z'), C('X'), C('C'), C('V'),
7108     C('B'), C('N'), C('M'), NO,      NO,      C('/'), NO,      NO,
7109     [0x9C] '\r',      // KP_Enter
7110     [0xB5] C('/'),    // KP_Div
7111     [0xC8] KEY_UP,    [0xD0] KEY_DN,
7112     [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
7113     [0xCB] KEY_LF,    [0xCD] KEY_RT,
7114     [0x97] KEY_HOME,  [0xCF] KEY_END,
7115     [0xD2] KEY_INS,   [0xD3] KEY_DEL
7116 };
7117
7118
7119
7120
7121
7122
7123
7124
7125
7126
7127
7128
7129
7130
7131
7132
7133
7134
7135
7136
7137
7138
7139
7140
7141
7142
7143
7144
7145
7146
7147
7148
7149

```

```

7150 #include "types.h"
7151 #include "x86.h"
7152 #include "defs.h"
7153 #include "kbd.h"
7154
7155 int
7156 kbdgetc(void)
7157 {
7158     static uint shift;
7159     static uchar *charcode[4] = {
7160         normalmap, shiftmap, ctlmap, ctlmap
7161     };
7162     uint st, data, c;
7163
7164     st = inb(KBSTATP);
7165     if((st & KBS_DIB) == 0)
7166         return -1;
7167     data = inb(KBDATAP);
7168
7169     if(data == 0xE0){
7170         shift |= E0ESC;
7171         return 0;
7172     } else if(data & 0x80){
7173         // Key released
7174         data = (shift & E0ESC ? data : data & 0x7F);
7175         shift &= ~(shiftcode[data] | E0ESC);
7176         return 0;
7177     } else if(shift & E0ESC){
7178         // Last character was an E0 escape; or with 0x80
7179         data |= 0x80;
7180         shift &= ~E0ESC;
7181     }
7182
7183     shift |= shiftcode[data];
7184     shift ^= togglecode[data];
7185     c = charcode[shift & (CTL | SHIFT)][data];
7186     if(shift & CAPSLOCK){
7187         if('a' <= c && c <= 'z')
7188             c += 'A' - 'a';
7189         else if('A' <= c && c <= 'Z')
7190             c += 'a' - 'A';
7191     }
7192     return c;
7193 }
7194
7195 void
7196 kbdintr(void)
7197 {
7198     consoleintr(kbdgetc);
7199 }

```

```

7200 // Console input and output.
7201 // Input is from the keyboard or serial port.
7202 // Output is written to the screen and serial port.
7203
7204 #include "types.h"
7205 #include "defs.h"
7206 #include "param.h"
7207 #include "traps.h"
7208 #include "spinlock.h"
7209 #include "fs.h"
7210 #include "file.h"
7211 #include "memlayout.h"
7212 #include "mmu.h"
7213 #include "proc.h"
7214 #include "x86.h"
7215
7216 static void consputc(int);
7217
7218 static int panicked = 0;
7219
7220 static struct {
7221   struct spinlock lock;
7222   int locking;
7223 } cons;
7224
7225 static void
7226 printint(int xx, int base, int sign)
7227 {
7228   static char digits[] = "0123456789abcdef";
7229   char buf[16];
7230   int i;
7231   uint x;
7232
7233   if(sign && (sign = xx < 0))
7234     x = -xx;
7235   else
7236     x = xx;
7237
7238   i = 0;
7239   do{
7240     buf[i++] = digits[x % base];
7241   }while((x /= base) != 0);
7242
7243   if(sign)
7244     buf[i++] = '-';
7245
7246   while(--i >= 0)
7247     consputc(buf[i]);
7248 }
7249

```

```

7250 // Print to the console. only understands %d, %x, %p, %s.
7251 void
7252 cprintf(char *fmt, ...)
7253 {
7254   int i, c, locking;
7255   uint *argp;
7256   char *s;
7257
7258   locking = cons.locking;
7259   if(locking)
7260     acquire(&cons.lock);
7261
7262   if (fmt == 0)
7263     panic("null fmt");
7264
7265   argp = (uint*)(void*)&fmt + 1;
7266   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
7267     if(c != '%'){
7268       consputc(c);
7269       continue;
7270     }
7271     c = fmt[++i] & 0xff;
7272     if(c == 0)
7273       break;
7274     switch(c){
7275     case 'd':
7276       printint(*argp++, 10, 1);
7277       break;
7278     case 'x':
7279     case 'p':
7280       printint(*argp++, 16, 0);
7281       break;
7282     case 's':
7283       if((s = (char*)*argp++) == 0)
7284         s = "(null)";
7285       for(; *s; s++)
7286         consputc(*s);
7287       break;
7288     case '%':
7289       consputc('%');
7290       break;
7291     default:
7292       // Print unknown % sequence to draw attention.
7293       consputc('%');
7294       consputc(c);
7295       break;
7296     }
7297   }
7298
7299

```

```

7300  if(locking)
7301      release(&cons.lock);
7302  }
7303
7304  void
7305  panic(char *s)
7306  {
7307      int i;
7308      uint pcs[10];
7309
7310      cli();
7311      cons.locking = 0;
7312      cprintf("cpu%d: panic: ", cpu->id);
7313      cprintf(s);
7314      cprintf("\n");
7315      getcallerpcs(&s, pcs);
7316      for(i=0; i<10; i++)
7317          cprintf(" %p", pcs[i]);
7318      panicked = 1; // freeze other CPU
7319      for(;;)
7320          ;
7321  }
7322
7323
7324
7325
7326
7327
7328
7329
7330
7331
7332
7333
7334
7335
7336
7337
7338
7339
7340
7341
7342
7343
7344
7345
7346
7347
7348
7349

```

```

7350  #define BACKSPACE 0x100
7351  #define CRTPORT 0x3d4
7352  static ushort *crt = (ushort*)P2V(0xb8000); // CGA memory
7353
7354  static void
7355  cgaputc(int c)
7356  {
7357      int pos;
7358
7359      // Cursor position: col + 80*row.
7360      outb(CRTPORT, 14);
7361      pos = inb(CRTPORT+1) << 8;
7362      outb(CRTPORT, 15);
7363      pos |= inb(CRTPORT+1);
7364
7365      if(c == '\n')
7366          pos += 80 - pos%80;
7367      else if(c == BACKSPACE){
7368          if(pos > 0) --pos;
7369      } else
7370          crt[pos++] = (c&0xff) | 0x0700; // black on white
7371
7372      if((pos/80) >= 24){ // Scroll up.
7373          memmove(crt, crt+80, sizeof(crt[0])*23*80);
7374          pos -= 80;
7375          memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
7376      }
7377
7378      outb(CRTPORT, 14);
7379      outb(CRTPORT+1, pos>>8);
7380      outb(CRTPORT, 15);
7381      outb(CRTPORT+1, pos);
7382      crt[pos] = ' ' | 0x0700;
7383  }
7384
7385  void
7386  consputc(int c)
7387  {
7388      if(panicked){
7389          cli();
7390          for(;;)
7391              ;
7392      }
7393
7394      if(c == BACKSPACE){
7395          uartputc('\b'); uartputc(' '); uartputc('\b');
7396      } else
7397          uartputc(c);
7398      cgaputc(c);
7399  }

```



```

7400 #define INPUT_BUF 128
7401 struct {
7402     struct spinlock lock;
7403     char buf[INPUT_BUF];
7404     uint r; // Read index
7405     uint w; // Write index
7406     uint e; // Edit index
7407 } input;
7408
7409 #define C(x) ((x)-'@') // Control-x
7410
7411 void
7412 consoleintr(int (*getc)(void))
7413 {
7414     int c;
7415
7416     acquire(&input.lock);
7417     while((c = getc()) >= 0){
7418         switch(c){
7419             case C('P'): // Process listing.
7420                 procdump();
7421                 break;
7422             case C('U'): // Kill line.
7423                 while(input.e != input.w &&
7424                     input.buf[(input.e-1) % INPUT_BUF] != '\n'){
7425                     input.e--;
7426                     consputc(BACKSPACE);
7427                 }
7428                 break;
7429             case C('H'): case '\x7f': // Backspace
7430                 if(input.e != input.w){
7431                     input.e--;
7432                     consputc(BACKSPACE);
7433                 }
7434                 break;
7435             default:
7436                 if(c != 0 && input.e-input.r < INPUT_BUF){
7437                     c = (c == '\r') ? '\n' : c;
7438                     input.buf[input.e++ % INPUT_BUF] = c;
7439                     consputc(c);
7440                     if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
7441                         input.w = input.e;
7442                         wakeup(&input.r);
7443                     }
7444                 }
7445                 break;
7446         }
7447     }
7448     release(&input.lock);
7449 }

```

```

7450 int
7451 consoleread(struct inode *ip, char *dst, int n)
7452 {
7453     uint target;
7454     int c;
7455
7456     iunlock(ip);
7457     target = n;
7458     acquire(&input.lock);
7459     while(n > 0){
7460         while(input.r == input.w){
7461             if(proc->killed){
7462                 release(&input.lock);
7463                 ilock(ip);
7464                 return -1;
7465             }
7466             sleep(&input.r, &input.lock);
7467         }
7468         c = input.buf[input.r++ % INPUT_BUF];
7469         if(c == C('D')){ // EOF
7470             if(n < target){
7471                 // Save ^D for next time, to make sure
7472                 // caller gets a 0-byte result.
7473                 input.r--;
7474             }
7475             break;
7476         }
7477         *dst++ = c;
7478         --n;
7479         if(c == '\n')
7480             break;
7481     }
7482     release(&input.lock);
7483     ilock(ip);
7484
7485     return target - n;
7486 }
7487
7488
7489
7490
7491
7492
7493
7494
7495
7496
7497
7498
7499

```

```

7500 int
7501 consolewrite(struct inode *ip, char *buf, int n)
7502 {
7503     int i;
7504
7505     iunlock(ip);
7506     acquire(&cons.lock);
7507     for(i = 0; i < n; i++)
7508         consputc(buf[i] & 0xff);
7509     release(&cons.lock);
7510     ilock(ip);
7511
7512     return n;
7513 }
7514
7515 void
7516 consoleinit(void)
7517 {
7518     initlock(&cons.lock, "console");
7519     initlock(&input.lock, "input");
7520
7521     devsw[CONSOLE].write = consolewrite;
7522     devsw[CONSOLE].read = consoleread;
7523     cons.locking = 1;
7524
7525     picenable(IRQ_KBD);
7526     ioapicenable(IRQ_KBD, 0);
7527 }
7528
7529
7530
7531
7532
7533
7534
7535
7536
7537
7538
7539
7540
7541
7542
7543
7544
7545
7546
7547
7548
7549

```

```

7550 // Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
7551 // Only used on uniprocessors;
7552 // SMP machines use the local APIC timer.
7553
7554 #include "types.h"
7555 #include "defs.h"
7556 #include "traps.h"
7557 #include "x86.h"
7558
7559 #define IO_TIMER1      0x040          // 8253 Timer #1
7560
7561 // Frequency of all three count-down timers;
7562 // (TIMER_FREQ/freq) is the appropriate count
7563 // to generate a frequency of freq Hz.
7564
7565 #define TIMER_FREQ      1193182
7566 #define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))
7567
7568 #define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
7569 #define TIMER_SELO      0x00          // select counter 0
7570 #define TIMER_RATEGEN    0x04          // mode 2, rate generator
7571 #define TIMER_16BIT      0x30          // r/w counter 16 bits, LSB first
7572
7573 void
7574 timerinit(void)
7575 {
7576     // Interrupt 100 times/sec.
7577     outb(TIMER_MODE, TIMER_SELO | TIMER_RATEGEN | TIMER_16BIT);
7578     outb(IO_TIMER1, TIMER_DIV(100) % 256);
7579     outb(IO_TIMER1, TIMER_DIV(100) / 256);
7580     picenable(IRQ_TIMER);
7581 }
7582
7583
7584
7585
7586
7587
7588
7589
7590
7591
7592
7593
7594
7595
7596
7597
7598
7599

```

```

7600 // Intel 8250 serial port (UART).
7601
7602 #include "types.h"
7603 #include "defs.h"
7604 #include "param.h"
7605 #include "traps.h"
7606 #include "spinlock.h"
7607 #include "fs.h"
7608 #include "file.h"
7609 #include "mmu.h"
7610 #include "proc.h"
7611 #include "x86.h"
7612
7613 #define COM1    0x3f8
7614
7615 static int uart;    // is there a uart?
7616
7617 void
7618 uartinit(void)
7619 {
7620     char *p;
7621
7622     // Turn off the FIFO
7623     outb(COM1+2, 0);
7624
7625     // 9600 baud, 8 data bits, 1 stop bit, parity off.
7626     outb(COM1+3, 0x80);    // Unlock divisor
7627     outb(COM1+0, 115200/9600);
7628     outb(COM1+1, 0);
7629     outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
7630     outb(COM1+4, 0);
7631     outb(COM1+1, 0x01);    // Enable receive interrupts.
7632
7633     // If status is 0xFF, no serial port.
7634     if(inb(COM1+5) == 0xFF)
7635         return;
7636     uart = 1;
7637
7638     // Acknowledge pre-existing interrupt conditions;
7639     // enable interrupts.
7640     inb(COM1+2);
7641     inb(COM1+0);
7642     picenable(IRQ_COM1);
7643     ioapicenable(IRQ_COM1, 0);
7644
7645     // Announce that we're here.
7646     for(p="xv6...\n"; *p; p++)
7647         uartputc(*p);
7648 }
7649

```

```

7650 void
7651 uartputc(int c)
7652 {
7653     int i;
7654
7655     if(!uart)
7656         return;
7657     for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
7658         microdelay(10);
7659     outb(COM1+0, c);
7660 }
7661
7662 static int
7663 uartgetc(void)
7664 {
7665     if(!uart)
7666         return -1;
7667     if(!(inb(COM1+5) & 0x01))
7668         return -1;
7669     return inb(COM1+0);
7670 }
7671
7672 void
7673 uartintr(void)
7674 {
7675     consoleintr(uartgetc);
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 # Initial process execs /init.
7701
7702 #include "syscall.h"
7703 #include "traps.h"
7704
7705
7706 # exec(init, argv)
7707 .globl start
7708 start:
7709     pushl $argv
7710     pushl $init
7711     pushl $0 // where caller pc would be
7712     movl $SYS_exec, %eax
7713     int $T_SYSCALL
7714
7715 # for(;;) exit();
7716 exit:
7717     movl $SYS_exit, %eax
7718     int $T_SYSCALL
7719     jmp exit
7720
7721 # char init[] = "/init\0";
7722 init:
7723     .string "/init\0"
7724
7725 # char *argv[] = { init, 0 };
7726 .p2align 2
7727 argv:
7728     .long init
7729     .long 0
7730
7731
7732
7733
7734
7735
7736
7737
7738
7739
7740
7741
7742
7743
7744
7745
7746
7747
7748
7749

```

```

7750 #include "syscall.h"
7751 #include "traps.h"
7752
7753 #define SYSCALL(name) \
7754     .globl name; \
7755     name: \
7756         movl $SYS_ ## name, %eax; \
7757         int $T_SYSCALL; \
7758         ret
7759
7760 SYSCALL(fork)
7761 SYSCALL(exit)
7762 SYSCALL(wait)
7763 SYSCALL(pipe)
7764 SYSCALL(read)
7765 SYSCALL(write)
7766 SYSCALL(close)
7767 SYSCALL(kill)
7768 SYSCALL(exec)
7769 SYSCALL(open)
7770 SYSCALL(mknod)
7771 SYSCALL(unlink)
7772 SYSCALL(fstat)
7773 SYSCALL(link)
7774 SYSCALL(mkdir)
7775 SYSCALL(chdir)
7776 SYSCALL(dup)
7777 SYSCALL(getpid)
7778 SYSCALL(sbrk)
7779 SYSCALL(sleep)
7780 SYSCALL(uptime)
7781
7782
7783
7784
7785
7786
7787
7788
7789
7790
7791
7792
7793
7794
7795
7796
7797
7798
7799

```

```

7800 // init: The initial user-level program
7801
7802 #include "types.h"
7803 #include "stat.h"
7804 #include "user.h"
7805 #include "fcntl.h"
7806
7807 char *argv[] = { "sh", 0 };
7808
7809 int
7810 main(void)
7811 {
7812     int pid, wpid;
7813
7814     if(open("console", O_RDWR) < 0){
7815         mknod("console", 1, 1);
7816         open("console", O_RDWR);
7817     }
7818     dup(0); // stdout
7819     dup(0); // stderr
7820
7821     for(;;){
7822         printf(1, "init: starting sh\n");
7823         pid = fork();
7824         if(pid < 0){
7825             printf(1, "init: fork failed\n");
7826             exit();
7827         }
7828         if(pid == 0){
7829             exec("sh", argv);
7830             printf(1, "init: exec sh failed\n");
7831             exit();
7832         }
7833         while((wpid=wait()) >= 0 && wpid != pid)
7834             printf(1, "zombie!\n");
7835     }
7836 }
7837
7838
7839
7840
7841
7842
7843
7844
7845
7846
7847
7848
7849

```

```

7850 // Shell.
7851
7852 #include "types.h"
7853 #include "user.h"
7854 #include "fcntl.h"
7855
7856 // Parsed command representation
7857 #define EXEC 1
7858 #define REDIR 2
7859 #define PIPE 3
7860 #define LIST 4
7861 #define BACK 5
7862
7863 #define MAXARGS 10
7864
7865 struct cmd {
7866     int type;
7867 };
7868
7869 struct execcmd {
7870     int type;
7871     char *argv[MAXARGS];
7872     char *eargv[MAXARGS];
7873 };
7874
7875 struct redircmd {
7876     int type;
7877     struct cmd *cmd;
7878     char *file;
7879     char *efile;
7880     int mode;
7881     int fd;
7882 };
7883
7884 struct pipecmd {
7885     int type;
7886     struct cmd *left;
7887     struct cmd *right;
7888 };
7889
7890 struct listcmd {
7891     int type;
7892     struct cmd *left;
7893     struct cmd *right;
7894 };
7895
7896 struct backcmd {
7897     int type;
7898     struct cmd *cmd;
7899 };

```

```

7900 int fork1(void); // Fork but panics on failure.
7901 void panic(char*);
7902 struct cmd *parsecmd(char*);
7903
7904 // Execute cmd. Never returns.
7905 void
7906 runcmd(struct cmd *cmd)
7907 {
7908     int p[2];
7909     struct backcmd *bcmd;
7910     struct execcmd *ecmd;
7911     struct listcmd *lcmd;
7912     struct pipecmd *pcmd;
7913     struct redircmd *rcmd;
7914
7915     if(cmd == 0)
7916         exit();
7917
7918     switch(cmd->type){
7919     default:
7920         panic("runcmd");
7921
7922     case EXEC:
7923         ecmd = (struct execcmd*)cmd;
7924         if(ecmd->argv[0] == 0)
7925             exit();
7926         exec(ecmd->argv[0], ecmd->argv);
7927         printf(2, "exec %s failed\n", ecmd->argv[0]);
7928         break;
7929
7930     case REDIR:
7931         rcmd = (struct redircmd*)cmd;
7932         close(rcmd->fd);
7933         if(open(rcmd->file, rcmd->mode) < 0){
7934             printf(2, "open %s failed\n", rcmd->file);
7935             exit();
7936         }
7937         runcmd(rcmd->cmd);
7938         break;
7939
7940     case LIST:
7941         lcmd = (struct listcmd*)cmd;
7942         if(fork1() == 0)
7943             runcmd(lcmd->left);
7944         wait();
7945         runcmd(lcmd->right);
7946         break;
7947
7948
7949

```

```

7950     case PIPE:
7951         pcmd = (struct pipecmd*)cmd;
7952         if(pipe(p) < 0)
7953             panic("pipe");
7954         if(fork1() == 0){
7955             close(1);
7956             dup(p[1]);
7957             close(p[0]);
7958             close(p[1]);
7959             runcmd(pcmd->left);
7960         }
7961         if(fork1() == 0){
7962             close(0);
7963             dup(p[0]);
7964             close(p[0]);
7965             close(p[1]);
7966             runcmd(pcmd->right);
7967         }
7968         close(p[0]);
7969         close(p[1]);
7970         wait();
7971         wait();
7972         break;
7973
7974     case BACK:
7975         bcmd = (struct backcmd*)cmd;
7976         if(fork1() == 0)
7977             runcmd(bcmd->cmd);
7978         break;
7979     }
7980     exit();
7981 }
7982
7983 int
7984 getcmd(char *buf, int nbuf)
7985 {
7986     printf(2, "$ ");
7987     memset(buf, 0, nbuf);
7988     gets(buf, nbuf);
7989     if(buf[0] == 0) // EOF
7990         return -1;
7991     return 0;
7992 }
7993
7994
7995
7996
7997
7998
7999

```

```

8000 int
8001 main(void)
8002 {
8003     static char buf[100];
8004     int fd;
8005
8006     // Assumes three file descriptors open.
8007     while((fd = open("console", O_RDWR)) >= 0){
8008         if(fd >= 3){
8009             close(fd);
8010             break;
8011         }
8012     }
8013
8014     // Read and run input commands.
8015     while(getcmd(buf, sizeof(buf)) >= 0){
8016         if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
8017             // Clumsy but will have to do for now.
8018             // Chdir has no effect on the parent if run in the child.
8019             buf[strlen(buf)-1] = 0; // chop \n
8020             if(chdir(buf+3) < 0)
8021                 printf(2, "cannot cd %s\n", buf+3);
8022             continue;
8023         }
8024         if(fork1() == 0)
8025             runcmd(parsecmd(buf));
8026         wait();
8027     }
8028     exit();
8029 }
8030
8031 void
8032 panic(char *s)
8033 {
8034     printf(2, "%s\n", s);
8035     exit();
8036 }
8037
8038 int
8039 fork1(void)
8040 {
8041     int pid;
8042
8043     pid = fork();
8044     if(pid == -1)
8045         panic("fork");
8046     return pid;
8047 }
8048
8049

```

```

8050 // Constructors
8051
8052 struct cmd*
8053 execcmd(void)
8054 {
8055     struct execcmd *cmd;
8056
8057     cmd = malloc(sizeof(*cmd));
8058     memset(cmd, 0, sizeof(*cmd));
8059     cmd->type = EXEC;
8060     return (struct cmd*)cmd;
8061 }
8062
8063 struct cmd*
8064 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
8065 {
8066     struct redircmd *cmd;
8067
8068     cmd = malloc(sizeof(*cmd));
8069     memset(cmd, 0, sizeof(*cmd));
8070     cmd->type = REDIR;
8071     cmd->cmd = subcmd;
8072     cmd->file = file;
8073     cmd->efile = efile;
8074     cmd->mode = mode;
8075     cmd->fd = fd;
8076     return (struct cmd*)cmd;
8077 }
8078
8079 struct cmd*
8080 pipecmd(struct cmd *left, struct cmd *right)
8081 {
8082     struct pipecmd *cmd;
8083
8084     cmd = malloc(sizeof(*cmd));
8085     memset(cmd, 0, sizeof(*cmd));
8086     cmd->type = PIPE;
8087     cmd->left = left;
8088     cmd->right = right;
8089     return (struct cmd*)cmd;
8090 }
8091
8092
8093
8094
8095
8096
8097
8098
8099

```

```

8100 struct cmd*
8101 listcmd(struct cmd *left, struct cmd *right)
8102 {
8103     struct listcmd *cmd;
8104
8105     cmd = malloc(sizeof(*cmd));
8106     memset(cmd, 0, sizeof(*cmd));
8107     cmd->type = LIST;
8108     cmd->left = left;
8109     cmd->right = right;
8110     return (struct cmd*)cmd;
8111 }
8112
8113 struct cmd*
8114 backcmd(struct cmd *subcmd)
8115 {
8116     struct backcmd *cmd;
8117
8118     cmd = malloc(sizeof(*cmd));
8119     memset(cmd, 0, sizeof(*cmd));
8120     cmd->type = BACK;
8121     cmd->cmd = subcmd;
8122     return (struct cmd*)cmd;
8123 }
8124
8125
8126
8127
8128
8129
8130
8131
8132
8133
8134
8135
8136
8137
8138
8139
8140
8141
8142
8143
8144
8145
8146
8147
8148
8149

```

```

8150 // Parsing
8151
8152 char whitespace[] = " \t\r\n\v";
8153 char symbols[] = "<|>&()";
8154
8155 int
8156 gettoken(char **ps, char *es, char **q, char **eq)
8157 {
8158     char *s;
8159     int ret;
8160
8161     s = *ps;
8162     while(s < es && strchr(whitespace, *s))
8163         s++;
8164     if(*s)
8165         *q = s;
8166     ret = *s;
8167     switch(*s){
8168     case 0:
8169         break;
8170     case '|':
8171     case '(':
8172     case ')':
8173     case ';':
8174     case '&':
8175     case '<':
8176         s++;
8177         break;
8178     case '>':
8179         s++;
8180         if(*s == '>'){
8181             ret = '+';
8182             s++;
8183         }
8184         break;
8185     default:
8186         ret = 'a';
8187         while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
8188             s++;
8189         break;
8190     }
8191     if(eq)
8192         *eq = s;
8193
8194     while(s < es && strchr(whitespace, *s))
8195         s++;
8196     *ps = s;
8197     return ret;
8198 }
8199

```



```

8200 int
8201 peek(char **ps, char *es, char *toks)
8202 {
8203     char *s;
8204
8205     s = *ps;
8206     while(s < es && strchr(whitespace, *s))
8207         s++;
8208     *ps = s;
8209     return *s && strchr(toks, *s);
8210 }
8211
8212 struct cmd *parseline(char**, char*);
8213 struct cmd *parsepipe(char**, char*);
8214 struct cmd *parseexec(char**, char*);
8215 struct cmd *nulterminate(struct cmd*);
8216
8217 struct cmd*
8218 parsecmd(char *s)
8219 {
8220     char *es;
8221     struct cmd *cmd;
8222
8223     es = s + strlen(s);
8224     cmd = parseline(&s, es);
8225     peek(&s, es, "");
8226     if(s != es){
8227         printf(2, "leftovers: %s\n", s);
8228         panic("syntax");
8229     }
8230     nulterminate(cmd);
8231     return cmd;
8232 }
8233
8234 struct cmd*
8235 parseline(char **ps, char *es)
8236 {
8237     struct cmd *cmd;
8238
8239     cmd = parsepipe(ps, es);
8240     while(peek(ps, es, "&")){
8241         gettoken(ps, es, 0, 0);
8242         cmd = backcmd(cmd);
8243     }
8244     if(peek(ps, es, ";")){
8245         gettoken(ps, es, 0, 0);
8246         cmd = listcmd(cmd, parseline(ps, es));
8247     }
8248     return cmd;
8249 }

```

```

8250 struct cmd*
8251 parsepipe(char **ps, char *es)
8252 {
8253     struct cmd *cmd;
8254
8255     cmd = parseexec(ps, es);
8256     if(peek(ps, es, "|")){
8257         gettoken(ps, es, 0, 0);
8258         cmd = pipecmd(cmd, parsepipe(ps, es));
8259     }
8260     return cmd;
8261 }
8262
8263 struct cmd*
8264 parseredirs(struct cmd *cmd, char **ps, char *es)
8265 {
8266     int tok;
8267     char *q, *eq;
8268
8269     while(peek(ps, es, "<>")){
8270         tok = gettoken(ps, es, 0, 0);
8271         if(gettoken(ps, es, &q, &eq) != 'a')
8272             panic("missing file for redirection");
8273         switch(tok){
8274             case '<':
8275                 cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
8276                 break;
8277             case '>':
8278                 cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8279                 break;
8280             case '+': // >>
8281                 cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8282                 break;
8283         }
8284     }
8285     return cmd;
8286 }
8287
8288
8289
8290
8291
8292
8293
8294
8295
8296
8297
8298
8299

```

```

8300 struct cmd*
8301 parseblock(char **ps, char *es)
8302 {
8303     struct cmd *cmd;
8304
8305     if(!peek(ps, es, "("))
8306         panic("parseblock");
8307     gettoken(ps, es, 0, 0);
8308     cmd = parseline(ps, es);
8309     if(!peek(ps, es, ")"))
8310         panic("syntax - missing )");
8311     gettoken(ps, es, 0, 0);
8312     cmd = parseredirs(cmd, ps, es);
8313     return cmd;
8314 }
8315
8316 struct cmd*
8317 parseexec(char **ps, char *es)
8318 {
8319     char *q, *eq;
8320     int tok, argc;
8321     struct execcmd *cmd;
8322     struct cmd *ret;
8323
8324     if(peek(ps, es, "("))
8325         return parseblock(ps, es);
8326
8327     ret = execcmd();
8328     cmd = (struct execcmd*)ret;
8329
8330     argc = 0;
8331     ret = parseredirs(ret, ps, es);
8332     while(!peek(ps, es, "|&");){
8333         if((tok=gettoken(ps, es, &q, &eq)) == 0)
8334             break;
8335         if(tok != 'a')
8336             panic("syntax");
8337         cmd->argv[argc] = q;
8338         cmd->eargv[argc] = eq;
8339         argc++;
8340         if(argc >= MAXARGS)
8341             panic("too many args");
8342         ret = parseredirs(ret, ps, es);
8343     }
8344     cmd->argv[argc] = 0;
8345     cmd->eargv[argc] = 0;
8346     return ret;
8347 }
8348
8349

```

```

8350 // NUL-terminate all the counted strings.
8351 struct cmd*
8352 nulterminate(struct cmd *cmd)
8353 {
8354     int i;
8355     struct backcmd *bcmd;
8356     struct execcmd *ecmd;
8357     struct listcmd *lcmd;
8358     struct pipecmd *pcmd;
8359     struct redircmd *rcmd;
8360
8361     if(cmd == 0)
8362         return 0;
8363
8364     switch(cmd->type){
8365     case EXEC:
8366         ecmd = (struct execcmd*)cmd;
8367         for(i=0; ecmd->argv[i]; i++)
8368             *ecmd->eargv[i] = 0;
8369         break;
8370
8371     case REDIR:
8372         rcmd = (struct redircmd*)cmd;
8373         nulterminate(rcmd->cmd);
8374         *rcmd->efile = 0;
8375         break;
8376
8377     case PIPE:
8378         pcmd = (struct pipecmd*)cmd;
8379         nulterminate(pcmd->left);
8380         nulterminate(pcmd->right);
8381         break;
8382
8383     case LIST:
8384         lcmd = (struct listcmd*)cmd;
8385         nulterminate(lcmd->left);
8386         nulterminate(lcmd->right);
8387         break;
8388
8389     case BACK:
8390         bcmd = (struct backcmd*)cmd;
8391         nulterminate(bcmd->cmd);
8392         break;
8393     }
8394     return cmd;
8395 }
8396
8397
8398
8399

```

```

8400 #include "asm.h"
8401 #include "memlayout.h"
8402 #include "mmu.h"
8403
8404 # Start the first CPU: switch to 32-bit protected mode, jump into C.
8405 # The BIOS loads this code from the first sector of the hard disk into
8406 # memory at physical address 0x7c00 and starts executing in real mode
8407 # with %cs=0 %ip=7c00.
8408
8409 .code16                                # Assemble for 16-bit mode
8410 .globl start
8411 start:
8412     cli                                # BIOS enabled interrupts; disable
8413
8414 # Zero data segment registers DS, ES, and SS.
8415     xorw    %ax,%ax                    # Set %ax to zero
8416     movw    %ax,%ds                    # -> Data Segment
8417     movw    %ax,%es                    # -> Extra Segment
8418     movw    %ax,%ss                    # -> Stack Segment
8419
8420 # Physical address line A20 is tied to zero so that the first PCs
8421 # with 2 MB would run software that assumed 1 MB. Undo that.
8422 seta20.1:
8423     inb     $0x64,%al                  # Wait for not busy
8424     testb   $0x2,%al
8425     jnz     seta20.1
8426
8427     movb    $0xd1,%al                  # 0xd1 -> port 0x64
8428     outb    %al,$0x64
8429
8430 seta20.2:
8431     inb     $0x64,%al                  # Wait for not busy
8432     testb   $0x2,%al
8433     jnz     seta20.2
8434
8435     movb    $0xdf,%al                  # 0xdf -> port 0x60
8436     outb    %al,$0x60
8437
8438 # Switch from real to protected mode. Use a bootstrap GDT that makes
8439 # virtual addresses map directly to physical addresses so that the
8440 # effective memory map doesn't change during the transition.
8441     lgdt    gdtdesc
8442     movl    %cr0,%eax
8443     orl     $CR0_PE,%eax
8444     movl    %eax,%cr0
8445
8446
8447
8448
8449

```

```

8450 # Complete transition to 32-bit protected mode by using long jmp
8451 # to reload %cs and %eip. The segment descriptors are set up with no
8452 # translation, so that the mapping is still the identity mapping.
8453     ljmp     $(SEG_KCODE<<3), $start32
8454
8455 .code32 # Tell assembler to generate 32-bit code now.
8456 start32:
8457 # Set up the protected-mode data segment registers
8458     movw    $(SEG_KDATA<<3), %ax      # Our data segment selector
8459     movw    %ax,%ds                    # -> DS: Data Segment
8460     movw    %ax,%es                    # -> ES: Extra Segment
8461     movw    %ax,%ss                    # -> SS: Stack Segment
8462     movw    $0,%ax                     # Zero segments not ready for use
8463     movw    %ax,%fs                    # -> FS
8464     movw    %ax,%gs                    # -> GS
8465
8466 # Set up the stack pointer and call into C.
8467     movl    $start,%esp
8468     call    bootmain
8469
8470 # If bootmain returns (it shouldn't), trigger a Bochs
8471 # breakpoint if running under Bochs, then loop.
8472     movw    $0x8a00,%ax                # 0x8a00 -> port 0x8a00
8473     movw    %ax,%dx
8474     outw    %ax,%dx
8475     movw    $0x8ae0,%ax                # 0x8ae0 -> port 0x8a00
8476     outw    %ax,%dx
8477 spin:
8478     jmp     spin
8479
8480 # Bootstrap GDT
8481 .p2align 2                                # force 4 byte alignment
8482 gdt:
8483     SEG_NULLASM                          # null seg
8484     SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg
8485     SEG_ASM(STA_W, 0x0, 0xffffffff)      # data seg
8486
8487 gdtdesc:
8488     .word    (gdtdesc - gdt - 1)        # sizeof(gdt) - 1
8489     .long    gdt                         # address gdt
8490
8491
8492
8493
8494
8495
8496
8497
8498
8499

```

```

8500 // Boot loader.
8501 //
8502 // Part of the boot sector, along with bootasm.S, which calls bootmain().
8503 // bootasm.S has put the processor into protected 32-bit mode.
8504 // bootmain() loads an ELF kernel image from the disk starting at
8505 // sector 1 and then jumps to the kernel entry routine.
8506
8507 #include "types.h"
8508 #include "elf.h"
8509 #include "x86.h"
8510 #include "memlayout.h"
8511
8512 #define SECTSIZE 512
8513
8514 void readseg(uchar*, uint, uint);
8515
8516 void
8517 bootmain(void)
8518 {
8519     struct elfhdr *elf;
8520     struct proghdr *ph, *eph;
8521     void (*entry)(void);
8522     uchar* pa;
8523
8524     elf = (struct elfhdr*)0x10000; // scratch space
8525
8526     // Read 1st page off disk
8527     readseg((uchar*)elf, 4096, 0);
8528
8529     // Is this an ELF executable?
8530     if(elf->magic != ELF_MAGIC)
8531         return; // let bootasm.S handle error
8532
8533     // Load each program segment (ignores ph flags).
8534     ph = (struct proghdr*)((uchar*)elf + elf->phoff);
8535     eph = ph + elf->phnum;
8536     for(; ph < eph; ph++){
8537         pa = (uchar*)ph->paddr;
8538         readseg(pa, ph->filesz, ph->off);
8539         if(ph->memsz > ph->filesz)
8540             stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
8541     }
8542
8543     // Call the entry point from the ELF header.
8544     // Does not return!
8545     entry = (void*)(void)(elf->entry);
8546     entry();
8547 }
8548
8549

```

```

8550 void
8551 waitdisk(void)
8552 {
8553     // Wait for disk ready.
8554     while((inb(0x1F7) & 0xC0) != 0x40)
8555         ;
8556 }
8557
8558 // Read a single sector at offset into dst.
8559 void
8560 readsect(void *dst, uint offset)
8561 {
8562     // Issue command.
8563     waitdisk();
8564     outb(0x1F2, 1); // count = 1
8565     outb(0x1F3, offset);
8566     outb(0x1F4, offset >> 8);
8567     outb(0x1F5, offset >> 16);
8568     outb(0x1F6, (offset >> 24) | 0xE0);
8569     outb(0x1F7, 0x20); // cmd 0x20 - read sectors
8570
8571     // Read data.
8572     waitdisk();
8573     insl(0x1F0, dst, SECTSIZE/4);
8574 }
8575
8576 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
8577 // Might copy more than asked.
8578 void
8579 readseg(uchar* pa, uint count, uint offset)
8580 {
8581     uchar* epa;
8582
8583     epa = pa + count;
8584
8585     // Round down to sector boundary.
8586     pa -= offset % SECTSIZE;
8587
8588     // Translate from bytes to sectors; kernel starts at sector 1.
8589     offset = (offset / SECTSIZE) + 1;
8590
8591     // If this is too slow, we could read lots of sectors at a time.
8592     // We'd write more to memory than asked, but it doesn't matter --
8593     // we load in increasing order.
8594     for(; pa < epa; pa += SECTSIZE, offset++){
8595         readsect(pa, offset);
8596     }
8597
8598
8599

```