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 RISC-V multiprocessor using ANSI C.

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14,
2000)). See also https://pdos.csail.mit.edu/6.1810/, which provides
pointers to on-line resources for v6.

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

We are also grateful for the bug reports and patches contributed by
Abhinavpatel00, Takahiro Aoyagi, Marcelo Arroyo, Hirbod Behnam, Silas
Boyd-Wickizer, Anton Burtsev, carlclone, Ian Chen, clivezeng, Dan
Cross, Cody Cutler, Mike CAT, Tej Chajed, Asami Doi,Wenyang Duan,
echtwerner, eyalz800, Nelson Elhage, Saar Ettinger, Alice Ferrazzi,
Nathaniel Filardo, flespark, Peter Froehlich, Yakir Goaron, Shivam
Handa, Matt Harvey, Bryan Henry, jaichenhengjie, Jim Huang, MatÃºÅ¡
JÃ³kay, John Jolly, Alexander Kapshuk, Anders Kaseorg, kehao95,
Wolfgang Keller, Jungwoo Kim, Jonathan Kimmitt, Eddie Kohler, Vadim
Kolontsov, Austin Liew, l0stman, Pavan Maddamsetti, Imbar Marinescu,
Yandong Mao, Matan Shabtay, Hitoshi Mitake, Carmi Merimovich,
mes900903, Mark Morrissey, mtasm, Joel Nider, Hayato Ohhashi,
OptimisticSide, papparapa, phosphagos, Harry Porter, Greg Price, Zheng
qhuo, Quancheng, RayAndrew, Jude Rich, segfault, Ayan Shafqat, Eldar
Sehayek, Yongming Shen, Fumiya Shigemitsu, snoire, Taojie, Cam Tenny,
tyfkda, Warren Toomey, Stephen Tu, Alissa Tung, Rafael Ubal, unicornx,
Amane Uehara, Pablo Ventura, Luc Videau, Xi Wang, WaheedHafez, Keiichi
Watanabe, Lucas Wolf, Nicolas Wolovick, wxdao, Grant Wu, x653, Andy
Zhang, Jindong Zhang, Icenowy Zheng, ZhUyU1997, and Zou Chang Wei.

ERROR REPORTS

Please send errors and suggestions to Frans Kaashoek and Robert Morris
(kaashoek,rtm@mit.edu). The main purpose of xv6 is as a teaching
operating system for MIT's 6.1810, so we are more interested in
simplifications and clarifications than new features.

BUILDING AND RUNNING XV6

You will need a RISC-V "newlib" tool chain from
https://github.com/riscv/riscv-gnu-toolchain, and qemu compiled for
riscv64-softmmu. Once they are installed, and in your shell
search path, you can run "make qemu".

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

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.

Aug 25 19:15 2025 blank Page 1

```
0900 typedef unsigned int   uint;
0901 typedef unsigned short ushort;
0902 typedef unsigned char  uchar;
0903
0904 typedef unsigned char uint8;
0905 typedef unsigned short uint16;
0906 typedef unsigned int  uint32;
0907 typedef unsigned long uint64;
0908
0909 typedef uint64 pde_t;
0910
0911
0912
0913
0914
0915
0916
0917
0918
0919
0920
0921
0922
0923
0924
0925
0926
0927
0928
0929
0930
0931
0932
0933
0934
0935
0936
0937
0938
0939
0940
0941
0942
0943
0944
0945
0946
0947
0948
0949
```



```
0150 #define NPROC        64  // maximum number of processes
0151 #define NCPU          8  // maximum number of CPUs
0152 #define NOFILE       16  // open files per process
0153 #define NFILE       100  // open files per system
0154 #define NINODE       50  // maximum number of active i-nodes
0155 #define NDEV         10  // maximum major device number
0156 #define ROOTDEV       1  // device number of file system root disk
0157 #define MAXARG       32  // max exec arguments
0158 #define MAXOPBLOCKS  10  // max # of blocks any FS op writes
0159 #define LOGBLOCKS    (MAXOPBLOCKS*3)  // max data blocks in on-disk log
0160 #define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
0161 #define FSSIZE       2000  // size of file system in blocks
0162 #define MAXPATH      128   // maximum file path name
0163 #define USERSTACK    1     // user stack pages
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 // Physical memory layout
0201
0202 // qemu -machine virt is set up like this,
0203 // based on qemu's hw/riscv/virt.c:
0204 //
0205 // 00001000 -- boot ROM, provided by qemu
0206 // 02000000 -- CLINT
0207 // 0C000000 -- PLIC
0208 // 10000000 -- uart0
0209 // 10001000 -- virtio disk
0210 // 80000000 -- qemu's boot ROM loads the kernel here,
0211 //             then jumps here.
0212 // unused RAM after 80000000.
0213
0214 // the kernel uses physical memory thus:
0215 // 80000000 -- entry.S, then kernel text and data
0216 // end -- start of kernel page allocation area
0217 // PHYSTOP -- end RAM used by the kernel
0218
0219 // qemu puts UART registers here in physical memory.
0220 #define UART0 0x10000000L
0221 #define UART0_IRQ 10
0222
0223 // virtio mmio interface
0224 #define VIRTIO0 0x10001000
0225 #define VIRTIO0_IRQ 1
0226
0227 // qemu puts platform-level interrupt controller (PLIC) here.
0228 #define PLIC 0x0c000000L
0229 #define PLIC_PRIORITY (PLIC + 0x0)
0230 #define PLIC_PENDING (PLIC + 0x1000)
0231 #define PLIC_SENABLE(hart) (PLIC + 0x2080 + (hart)*0x100)
0232 #define PLIC_SPRIORITY(hart) (PLIC + 0x201000 + (hart)*0x2000)
0233 #define PLIC_SCLAIM(hart) (PLIC + 0x201004 + (hart)*0x2000)
0234
0235 // the kernel expects there to be RAM
0236 // for use by the kernel and user pages
0237 // from physical address 0x80000000 to PHYSTOP.
0238 #define KERNBASE 0x80000000L
0239 #define PHYSTOP (KERNBASE + 128*1024*1024)
0240
0241 // map the trampoline page to the highest address,
0242 // in both user and kernel space.
0243 #define TRAMPOLINE (MAXVA - PGSIZE)
0244
0245 // map kernel stacks beneath the trampoline,
0246 // each surrounded by invalid guard pages.
0247 #define KSTACK(p) (TRAMPOLINE - ((p)+1)* 2*PGSIZE)
0248
0249
```



```
0250 // User memory layout.
0251 // Address zero first:
0252 //   text
0253 //   original data and bss
0254 //   fixed-size stack
0255 //   expandable heap
0256 //   ...
0257 //   TRAPFRAME (p->trapframe, used by the trampoline)
0258 //   TRAMPOLINE (the same page as in the kernel)
0259 #define TRAPFRAME (TRAMPOLINE - PGSIZE)
0260
0261
0262
0263
0264
0265
0266
0267
0268
0269
0270
0271
0272
0273
0274
0275
0276
0277
0278
0279
0280
0281
0282
0283
0284
0285
0286
0287
0288
0289
0290
0291
0292
0293
0294
0295
0296
0297
0298
0299
```

```
0300 struct buf;
0301 struct context;
0302 struct file;
0303 struct inode;
0304 struct pipe;
0305 struct proc;
0306 struct spinlock;
0307 struct sleeplock;
0308 struct stat;
0309 struct superblock;
0310
0311 // bio.c
0312 void            binit(void);
0313 struct buf*     bread(uint, uint);
0314 void            brelse(struct buf*);
0315 void            bwrite(struct buf*);
0316 void            bpin(struct buf*);
0317 void            bunpin(struct buf*);
0318
0319 // console.c
0320 void            consoleinit(void);
0321 void            consoleintr(int);
0322 void            consputc(int);
0323
0324 // exec.c
0325 int             kexec(char*, char**);
0326
0327 // file.c
0328 struct file*    filealloc(void);
0329 void            fileclose(struct file*);
0330 struct file*    filedup(struct file*);
0331 void            fileinit(void);
0332 int             fileread(struct file*, uint64, int n);
0333 int             filestat(struct file*, uint64 addr);
0334 int             filewrite(struct file*, uint64, int n);
0335
0336 // fs.c
0337 void            fsinit(int);
0338 int             dirlink(struct inode*, char*, uint);
0339 struct inode*   dirlookup(struct inode*, char*, uint*);
0340 struct inode*   ialloc(uint, short);
0341 struct inode*   idup(struct inode*);
0342 void            iinit();
0343 void            ilock(struct inode*);
0344 void            iput(struct inode*);
0345 void            iunlock(struct inode*);
0346 void            iunlockput(struct inode*);
0347 void            iupdate(struct inode*);
0348 int             namecmp(const char*, const char*);
0349 struct inode*   namei(char*);
```



```
0350 struct inode*   nameiparent(char*, char*);
0351 int             readi(struct inode*, int, uint64, uint, uint);
0352 void            stati(struct inode*, struct stat*);
0353 int             writei(struct inode*, int, uint64, uint, uint);
0354 void            itrunc(struct inode*);
0355 void            ireclaim(int);
0356
0357 // kalloc.c
0358 void*           kalloc(void);
0359 void            kfree(void *);
0360 void            kinit(void);
0361
0362 // log.c
0363 void            initlog(int, struct superblock*);
0364 void            log_write(struct buf*);
0365 void            begin_op(void);
0366 void            end_op(void);
0367
0368 // pipe.c
0369 int             pipealloc(struct file**, struct file**);
0370 void            pipeclose(struct pipe*, int);
0371 int             piperead(struct pipe*, uint64, int);
0372 int             pipewrite(struct pipe*, uint64, int);
0373
0374 // printf.c
0375 int             printf(char*, ...) __attribute__ ((format (printf, 1, 2)));
0376 void            panic(char*) __attribute__((noreturn));
0377 void            printfinit(void);
0378
0379 // proc.c
0380 int             cpuid(void);
0381 void            kexit(int);
0382 int             kfork(void);
0383 int             growproc(int);
0384 void            proc_mapstacks(pagetable_t);
0385 pagetable_t     proc_pagetable(struct proc *);
0386 void            proc_freepagetable(pagetable_t, uint64);
0387 int             kkill(int);
0388 int             killed(struct proc*);
0389 void            setkilled(struct proc*);
0390 struct cpu*     mycpu(void);
0391 struct proc*    myproc();
0392 void            procinit(void);
0393 void            scheduler(void) __attribute__((noreturn));
0394 void            sched(void);
0395 void            sleep(void*, struct spinlock*);
0396 void            userinit(void);
0397 int             kwait(uint64);
0398 void            wakeup(void*);
0399 void            yield(void);
```

```
0400 int             either_copyout(int user_dst, uint64 dst, void *src, uint64 l
0401 int             either_copyin(void *dst, int user_src, uint64 src, uint64 le
0402 void            procdump(void);
0403
0404 // swtch.S
0405 void            swtch(struct context*, struct context*);
0406
0407 // spinlock.c
0408 void            acquire(struct spinlock*);
0409 int             holding(struct spinlock*);
0410 void            initlock(struct spinlock*, char*);
0411 void            release(struct spinlock*);
0412 void            push_off(void);
0413 void            pop_off(void);
0414
0415 // sleeplock.c
0416 void            acquiresleep(struct sleeplock*);
0417 void            releasesleep(struct sleeplock*);
0418 int             holdingsleep(struct sleeplock*);
0419 void            initsleeplock(struct sleeplock*, char*);
0420
0421 // string.c
0422 int             memcmp(const void*, const void*, uint);
0423 void*           memmove(void*, const void*, uint);
0424 void*           memset(void*, int, uint);
0425 char*           safestrcpy(char*, const char*, int);
0426 int             strlen(const char*);
0427 int             strncmp(const char*, const char*, uint);
0428 char*           strncpy(char*, const char*, int);
0429
0430 // syscall.c
0431 void            argint(int, int*);
0432 int             argstr(int, char*, int);
0433 void            argaddr(int, uint64 *);
0434 int             fetchstr(uint64, char*, int);
0435 int             fetchaddr(uint64, uint64*);
0436 void            syscall();
0437
0438 // trap.c
0439 extern uint     ticks;
0440 void            trapinit(void);
0441 void            trapinithart(void);
0442 extern struct spinlock tickslock;
0443 void            prepare_return(void);
0444
0445 // uart.c
0446 void            uartinit(void);
0447 void            uartintr(void);
0448 void            uartwrite(char [], int);
0449 void            uartputc_sync(int);
```



```
0450 int             uartgetc(void);
0451
0452 // vm.c
0453 void            kvminit(void);
0454 void            kvminithart(void);
0455 void            kvmmap(pagetable_t, uint64, uint64, uint64, int);
0456 int             mappages(pagetable_t, uint64, uint64, uint64, int);
0457 pagetable_t     uvmcreate(void);
0458 uint64          uvmalloc(pagetable_t, uint64, uint64, int);
0459 uint64          uvmdealloc(pagetable_t, uint64, uint64);
0460 int             uvmcopy(pagetable_t, pagetable_t, uint64);
0461 void            uvmfree(pagetable_t, uint64);
0462 void            uvmunmap(pagetable_t, uint64, uint64, int);
0463 void            uvmclear(pagetable_t, uint64);
0464 pte_t *         walk(pagetable_t, uint64, int);
0465 uint64          walkaddr(pagetable_t, uint64);
0466 int             copyout(pagetable_t, uint64, char *, uint64);
0467 int             copyin(pagetable_t, char *, uint64, uint64);
0468 int             copyinstr(pagetable_t, char *, uint64, uint64);
0469 int             ismapped(pagetable_t, uint64);
0470 uint64          vmfault(pagetable_t, uint64, int);
0471
0472 // plic.c
0473 void            plicinit(void);
0474 void            plicinithart(void);
0475 int             plic_claim(void);
0476 void            plic_complete(int);
0477
0478 // virtio_disk.c
0479 void            virtio_disk_init(void);
0480 void            virtio_disk_rw(struct buf *, int);
0481 void            virtio_disk_intr(void);
0482
0483 // number of elements in fixed-size array
0484 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
0485
0486
0487
0488
0489
0490
0491
0492
0493
0494
0495
0496
0497
0498
0499
```

```
0500 #ifndef __ASSEMBLER__
0501
0502 // which hart (core) is this?
0503 static inline uint64
0504 r_mhartid()
0505 {
0506   uint64 x;
0507   asm volatile("csrr %0, mhartid" : "=r" (x) );
0508   return x;
0509 }
0510
0511 // Machine Status Register, mstatus
0512
0513 #define MSTATUS_MPP_MASK (3L << 11) // previous mode.
0514 #define MSTATUS_MPP_M (3L << 11)
0515 #define MSTATUS_MPP_S (1L << 11)
0516 #define MSTATUS_MPP_U (0L << 11)
0517
0518 static inline uint64
0519 r_mstatus()
0520 {
0521   uint64 x;
0522   asm volatile("csrr %0, mstatus" : "=r" (x) );
0523   return x;
0524 }
0525
0526 static inline void
0527 w_mstatus(uint64 x)
0528 {
0529   asm volatile("csrw mstatus, %0" : : "r" (x));
0530 }
0531
0532 // machine exception program counter, holds the
0533 // instruction address to which a return from
0534 // exception will go.
0535 static inline void
0536 w_mepc(uint64 x)
0537 {
0538   asm volatile("csrw mepc, %0" : : "r" (x));
0539 }
0540
0541
0542
0543
0544
0545
0546
0547
0548
0549
```



```
0550 // Supervisor Status Register, sstatus
0551
0552 #define SSTATUS_SPP (1L << 8)  // Previous mode, 1=Supervisor, 0=User
0553 #define SSTATUS_SPIE (1L << 5) // Supervisor Previous Interrupt Enable
0554 #define SSTATUS_UPIE (1L << 4) // User Previous Interrupt Enable
0555 #define SSTATUS_SIE (1L << 1)  // Supervisor Interrupt Enable
0556 #define SSTATUS_UIE (1L << 0)  // User Interrupt Enable
0557
0558 static inline uint64
0559 r_sstatus()
0560 {
0561   uint64 x;
0562   asm volatile("csrr %0, sstatus" : "=r" (x) );
0563   return x;
0564 }
0565
0566 static inline void
0567 w_sstatus(uint64 x)
0568 {
0569   asm volatile("csrw sstatus, %0" : : "r" (x));
0570 }
0571
0572 // Supervisor Interrupt Pending
0573 static inline uint64
0574 r_sip()
0575 {
0576   uint64 x;
0577   asm volatile("csrr %0, sip" : "=r" (x) );
0578   return x;
0579 }
0580
0581 static inline void
0582 w_sip(uint64 x)
0583 {
0584   asm volatile("csrw sip, %0" : : "r" (x));
0585 }
0586
0587 // Supervisor Interrupt Enable
0588 #define SIE_SEIE (1L << 9) // external
0589 #define SIE_STIE (1L << 5) // timer
0590 static inline uint64
0591 r_sie()
0592 {
0593   uint64 x;
0594   asm volatile("csrr %0, sie" : "=r" (x) );
0595   return x;
0596 }
0597
0598
0599
```

```
0600 static inline void
0601 w_sie(uint64 x)
0602 {
0603   asm volatile("csrw sie, %0" : : "r" (x));
0604 }
0605
0606 // Machine-mode Interrupt Enable
0607 #define MIE_STIE (1L << 5)  // supervisor timer
0608 static inline uint64
0609 r_mie()
0610 {
0611   uint64 x;
0612   asm volatile("csrr %0, mie" : "=r" (x) );
0613   return x;
0614 }
0615
0616 static inline void
0617 w_mie(uint64 x)
0618 {
0619   asm volatile("csrw mie, %0" : : "r" (x));
0620 }
0621
0622 // supervisor exception program counter, holds the
0623 // instruction address to which a return from
0624 // exception will go.
0625 static inline void
0626 w_sepc(uint64 x)
0627 {
0628   asm volatile("csrw sepc, %0" : : "r" (x));
0629 }
0630
0631 static inline uint64
0632 r_sepc()
0633 {
0634   uint64 x;
0635   asm volatile("csrr %0, sepc" : "=r" (x) );
0636   return x;
0637 }
0638
0639 // Machine Exception Delegation
0640 static inline uint64
0641 r_medeleg()
0642 {
0643   uint64 x;
0644   asm volatile("csrr %0, medeleg" : "=r" (x) );
0645   return x;
0646 }
0647
0648
0649
```



```
0650 static inline void
0651 w_medeleg(uint64 x)
0652 {
0653   asm volatile("csrw medeleg, %0" : : "r" (x));
0654 }
0655
0656 // Machine Interrupt Delegation
0657 static inline uint64
0658 r_mideleg()
0659 {
0660   uint64 x;
0661   asm volatile("csrr %0, mideleg" : "=r" (x) );
0662   return x;
0663 }
0664
0665 static inline void
0666 w_mideleg(uint64 x)
0667 {
0668   asm volatile("csrw mideleg, %0" : : "r" (x));
0669 }
0670
0671 // Supervisor Trap-Vector Base Address
0672 // low two bits are mode.
0673 static inline void
0674 w_stvec(uint64 x)
0675 {
0676   asm volatile("csrw stvec, %0" : : "r" (x));
0677 }
0678
0679 static inline uint64
0680 r_stvec()
0681 {
0682   uint64 x;
0683   asm volatile("csrr %0, stvec" : "=r" (x) );
0684   return x;
0685 }
0686
0687 // Supervisor Timer Comparison Register
0688 static inline uint64
0689 r_stimecmp()
0690 {
0691   uint64 x;
0692   // asm volatile("csrr %0, stimecmp" : "=r" (x) );
0693   asm volatile("csrr %0, 0x14d" : "=r" (x) );
0694   return x;
0695 }
0696
0697
0698
0699
```

```
0700 static inline void
0701 w_stimecmp(uint64 x)
0702 {
0703   // asm volatile("csrw stimecmp, %0" : : "r" (x));
0704   asm volatile("csrw 0x14d, %0" : : "r" (x));
0705 }
0706
0707 // Machine Environment Configuration Register
0708 static inline uint64
0709 r_menvcfg()
0710 {
0711   uint64 x;
0712   // asm volatile("csrr %0, menvcfg" : "=r" (x) );
0713   asm volatile("csrr %0, 0x30a" : "=r" (x) );
0714   return x;
0715 }
0716
0717 static inline void
0718 w_menvcfg(uint64 x)
0719 {
0720   // asm volatile("csrw menvcfg, %0" : : "r" (x));
0721   asm volatile("csrw 0x30a, %0" : : "r" (x));
0722 }
0723
0724 // Physical Memory Protection
0725 static inline void
0726 w_pmpcfg0(uint64 x)
0727 {
0728   asm volatile("csrw pmpcfg0, %0" : : "r" (x));
0729 }
0730
0731 static inline void
0732 w_pmpaddr0(uint64 x)
0733 {
0734   asm volatile("csrw pmpaddr0, %0" : : "r" (x));
0735 }
0736
0737 // use riscv's sv39 page table scheme.
0738 #define SATP_SV39 (8L << 60)
0739
0740 #define MAKE_SATP(pagetable) (SATP_SV39 | (((uint64)pagetable) >> 12))
0741
0742 // supervisor address translation and protection;
0743 // holds the address of the page table.
0744 static inline void
0745 w_satp(uint64 x)
0746 {
0747   asm volatile("csrw satp, %0" : : "r" (x));
0748 }
0749
```



```
0750 static inline uint64
0751 r_satp()
0752 {
0753   uint64 x;
0754   asm volatile("csrr %0, satp" : "=r" (x) );
0755   return x;
0756 }
0757
0758 // Supervisor Trap Cause
0759 static inline uint64
0760 r_scause()
0761 {
0762   uint64 x;
0763   asm volatile("csrr %0, scause" : "=r" (x) );
0764   return x;
0765 }
0766
0767 // Supervisor Trap Value
0768 static inline uint64
0769 r_stval()
0770 {
0771   uint64 x;
0772   asm volatile("csrr %0, stval" : "=r" (x) );
0773   return x;
0774 }
0775
0776 // Machine-mode Counter-Enable
0777 static inline void
0778 w_mcounteren(uint64 x)
0779 {
0780   asm volatile("csrw mcounteren, %0" : : "r" (x));
0781 }
0782
0783 static inline uint64
0784 r_mcounteren()
0785 {
0786   uint64 x;
0787   asm volatile("csrr %0, mcounteren" : "=r" (x) );
0788   return x;
0789 }
0790
0791 // machine-mode cycle counter
0792 static inline uint64
0793 r_time()
0794 {
0795   uint64 x;
0796   asm volatile("csrr %0, time" : "=r" (x) );
0797   return x;
0798 }
0799
```

```
0800 // enable device interrupts
0801 static inline void
0802 intr_on()
0803 {
0804   w_sstatus(r_sstatus() | SSTATUS_SIE);
0805 }
0806
0807 // disable device interrupts
0808 static inline void
0809 intr_off()
0810 {
0811   w_sstatus(r_sstatus() & ~SSTATUS_SIE);
0812 }
0813
0814 // are device interrupts enabled?
0815 static inline int
0816 intr_get()
0817 {
0818   uint64 x = r_sstatus();
0819   return (x & SSTATUS_SIE) != 0;
0820 }
0821
0822 static inline uint64
0823 r_sp()
0824 {
0825   uint64 x;
0826   asm volatile("mv %0, sp" : "=r" (x) );
0827   return x;
0828 }
0829
0830 // read and write tp, the thread pointer, which xv6 uses to hold
0831 // this core's hartid (core number), the index into cpus[].
0832 static inline uint64
0833 r_tp()
0834 {
0835   uint64 x;
0836   asm volatile("mv %0, tp" : "=r" (x) );
0837   return x;
0838 }
0839
0840 static inline void
0841 w_tp(uint64 x)
0842 {
0843   asm volatile("mv tp, %0" : : "r" (x));
0844 }
0845
0846
0847
0848
0849
```



```
0850 static inline uint64
0851 r_ra()
0852 {
0853   uint64 x;
0854   asm volatile("mv %0, ra" : "=r" (x) );
0855   return x;
0856 }
0857
0858 // flush the TLB.
0859 static inline void
0860 sfence_vma()
0861 {
0862   // the zero, zero means flush all TLB entries.
0863   asm volatile("sfence.vma zero, zero");
0864 }
0865
0866 typedef uint64 pte_t;
0867 typedef uint64 *pagetable_t; // 512 PTEs
0868
0869 #endif // __ASSEMBLER__
0870
0871 #define PGSIZE 4096 // bytes per page
0872 #define PGSHIFT 12  // bits of offset within a page
0873
0874 #define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
0875 #define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
0876
0877 #define PTE_V (1L << 0) // valid
0878 #define PTE_R (1L << 1)
0879 #define PTE_W (1L << 2)
0880 #define PTE_X (1L << 3)
0881 #define PTE_U (1L << 4) // user can access
0882
0883 // shift a physical address to the right place for a PTE.
0884 #define PA2PTE(pa) ((((uint64)pa) >> 12) << 10)
0885
0886 #define PTE2PA(pte) (((pte) >> 10) << 12)
0887
0888 #define PTE_FLAGS(pte) ((pte) & 0x3FF)
0889
0890 // extract the three 9-bit page table indices from a virtual address.
0891 #define PXMASK          0x1FF // 9 bits
0892 #define PXSHIFT(level)  (PGSHIFT+(9*(level)))
0893 #define PX(level, va) ((((uint64) (va)) >> PXSHIFT(level)) & PXMASK)
0894
0895 // one beyond the highest possible virtual address.
0896 // MAXVA is actually one bit less than the max allowed by
0897 // Sv39, to avoid having to sign-extend virtual addresses
0898 // that have the high bit set.
0899 #define MAXVA (1L << (9 + 9 + 9 + 12 - 1))
```

```
0900 typedef unsigned int   uint;
0901 typedef unsigned short ushort;
0902 typedef unsigned char  uchar;
0903
0904 typedef unsigned char uint8;
0905 typedef unsigned short uint16;
0906 typedef unsigned int  uint32;
0907 typedef unsigned long uint64;
0908
0909 typedef uint64 pde_t;
0910
0911
0912
0913
0914
0915
0916
0917
0918
0919
0920
0921
0922
0923
0924
0925
0926
0927
0928
0929
0930
0931
0932
0933
0934
0935
0936
0937
0938
0939
0940
0941
0942
0943
0944
0945
0946
0947
0948
0949
```



```
0950 // 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   uint64 entry;
0962   uint64 phoff;
0963   uint64 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   uint32 type;
0976   uint32 flags;
0977   uint64 off;
0978   uint64 vaddr;
0979   uint64 paddr;
0980   uint64 filesz;
0981   uint64 memsz;
0982   uint64 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
```

```
        # qemu -kernel loads the kernel at 0x80000000
        # and causes each hart (i.e. CPU) to jump there.
        # kernel.ld causes the following code to
        # be placed at 0x80000000.
.section .text
.global _entry
_entry:
        # set up a stack for C.
        # stack0 is declared in start.c,
        # with a 4096-byte stack per CPU.
        # sp = stack0 + ((hartid + 1) * 4096)
        la sp, stack0
        li a0, 1024*4
        csrr a1, mhartid
        addi a1, a1, 1
        mul a0, a0, a1
        add sp, sp, a0
        # jump to start() in start.c
        call start
spin:
        j spin
```



```
#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "riscv.h"
#include "defs.h"

void main();
void timerinit();

// entry.S needs one stack per CPU.
__attribute__ ((aligned (16))) char stack0[4096 * NCPU];

// entry.S jumps here in machine mode on stack0.
void
start()
{
  // set M Previous Privilege mode to Supervisor, for mret.
  unsigned long x = r_mstatus();
  x &= ~MSTATUS_MPP_MASK;
  x |= MSTATUS_MPP_S;
  w_mstatus(x);

  // set M Exception Program Counter to main, for mret.
  // requires gcc -mcmodel=medany
  w_mepc((uint64)main);

  // disable paging for now.
  w_satp(0);

  // delegate all interrupts and exceptions to supervisor mode.
  w_medeleg(0xffff);
  w_mideleg(0xffff);
  w_sie(r_sie() | SIE_SEIE | SIE_STIE);

  // configure Physical Memory Protection to give supervisor mode
  // access to all of physical memory.
  w_pmpaddr0(0x3fffffffffffffull);
  w_pmpcfg0(0xf);

  // ask for clock interrupts.
  timerinit();

  // keep each CPU's hartid in its tp register, for cpuid().
  int id = r_mhartid();
  w_tp(id);

  // switch to supervisor mode and jump to main().
  asm volatile("mret");
}
```

```
// ask each hart to generate timer interrupts.
void
timerinit()
{
  // enable supervisor-mode timer interrupts.
  w_mie(r_mie() | MIE_STIE);

  // enable the sstc extension (i.e. stimecmp).
  w_menvcfg(r_menvcfg() | (1L << 63));

  // allow supervisor to use stimecmp and time.
  w_mcounteren(r_mcounteren() | 2);

  // ask for the very first timer interrupt.
  w_stimecmp(r_time() + 1000000);
}
```



```
#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "riscv.h"
#include "defs.h"

volatile static int started = 0;

// start() jumps here in supervisor mode on all CPUs.
void
main()
{
  if(cpuid() == 0){
    consoleinit();
    printfinit();
    printf("\n");
    printf("xv6 kernel is booting\n");
    printf("\n");
    kinit();         // physical page allocator
    kvminit();       // create kernel page table
    kvminithart();   // turn on paging
    procinit();      // process table
    trapinit();      // trap vectors
    trapinithart();  // install kernel trap vector
    plicinit();      // set up interrupt controller
    plicinithart();  // ask PLIC for device interrupts
    binit();         // buffer cache
    iinit();         // inode table
    fileinit();      // file table
    virtio_disk_init(); // emulated hard disk
    userinit();      // first user process
    __sync_synchronize();
    started = 1;
  } else {
    while(started == 0)
      ;
    __sync_synchronize();
    printf("hart %d starting\n", cpuid());
    kvminithart();    // turn on paging
    trapinithart();   // install kernel trap vector
    plicinithart();   // ask PLIC for device interrupts
  }

  scheduler();
}
```

```
1200 // Mutual exclusion lock.
1201 struct spinlock {
1202   uint locked;       // Is the lock held?
1203
1204   // For debugging:
1205   char *name;        // Name of lock.
1206   struct cpu *cpu;   // The cpu holding the lock.
1207 };
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
```



```
1250 // Mutual exclusion spin locks.
1251
1252 #include "types.h"
1253 #include "param.h"
1254 #include "memlayout.h"
1255 #include "spinlock.h"
1256 #include "riscv.h"
1257 #include "proc.h"
1258 #include "defs.h"
1259
1260 void
1261 initlock(struct spinlock *lk, char *name)
1262 {
1263   lk->name = name;
1264   lk->locked = 0;
1265   lk->cpu = 0;
1266 }
1267
1268 // Acquire the lock.
1269 // Loops (spins) until the lock is acquired.
1270 void
1271 acquire(struct spinlock *lk)
1272 {
1273   push_off(); // disable interrupts to avoid deadlock.
1274   if(holding(lk))
1275     panic("acquire");
1276
1277   // On RISC-V, sync_lock_test_and_set turns into an atomic swap:
1278   //   a5 = 1
1279   //   s1 = &lk->locked
1280   //   amoswap.w.aq a5, a5, (s1)
1281   while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
1282     ;
1283
1284   // Tell the C compiler and the processor to not move loads or stores
1285   // past this point, to ensure that the critical section's memory
1286   // references happen strictly after the lock is acquired.
1287   // On RISC-V, this emits a fence instruction.
1288   __sync_synchronize();
1289
1290   // Record info about lock acquisition for holding() and debugging.
1291   lk->cpu = mycpu();
1292 }
1293
1294
1295
1296
1297
1298
1299
```

```
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->cpu = 0;

  // Tell the C compiler and the CPU to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other CPUs before the lock is released,
  // and that loads in the critical section occur strictly before
  // the lock is released.
  // On RISC-V, this emits a fence instruction.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code doesn't use a C assignment, since the C standard
  // implies that an assignment might be implemented with
  // multiple store instructions.
  // On RISC-V, sync_lock_release turns into an atomic swap:
  //   s1 = &lk->locked
  //   amoswap.w zero, zero, (s1)
  __sync_lock_release(&lk->locked);

  pop_off();
}

// Check whether this cpu is holding the lock.
// Interrupts must be off.
int
holding(struct spinlock *lk)
{
  int r;
  r = (lk->locked && lk->cpu == mycpu());
  return r;
}
```



```
// push_off/pop_off are like intr_off()/intr_on() except that they are match
// it takes two pop_off()s to undo two push_off()s.  Also, if interrupts
// are initially off, then push_off, pop_off leaves them off.

void
push_off(void)
{
  int old = intr_get();

  // disable interrupts to prevent an involuntary context
  // switch while using mycpu().
  intr_off();

  if(mycpu()->noff == 0)
    mycpu()->intena = old;
  mycpu()->noff += 1;
}

void
pop_off(void)
{
  struct cpu *c = mycpu();
  if(intr_get())
    panic("pop_off - interruptible");
  if(c->noff < 1)
    panic("pop_off");
  c->noff -= 1;
  if(c->noff == 0 && c->intena)
    intr_on();
}
```

```
#include "param.h"
#include "types.h"
#include "memlayout.h"
#include "elf.h"
#include "riscv.h"
#include "defs.h"
#include "spinlock.h"
#include "proc.h"
#include "fs.h"

/*
 * the kernel's page table.
 */
pagetable_t kernel_pagetable;

extern char etext[];  // kernel.ld sets this to end of kernel code.

extern char trampoline[]; // trampoline.S

// Make a direct-map page table for the kernel.
pagetable_t
kvmmake(void)
{
  pagetable_t kpgtbl;

  kpgtbl = (pagetable_t) kalloc();
  memset(kpgtbl, 0, PGSIZE);

  // uart registers
  kvmmap(kpgtbl, UART0, UART0, PGSIZE, PTE_R | PTE_W);

  // virtio mmio disk interface
  kvmmap(kpgtbl, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

  // PLIC
  kvmmap(kpgtbl, PLIC, PLIC, 0x4000000, PTE_R | PTE_W);

  // map kernel text executable and read-only.
  kvmmap(kpgtbl, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);

  // map kernel data and the physical RAM we'll make use of.
  kvmmap(kpgtbl, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R

  // map the trampoline for trap entry/exit to
  // the highest virtual address in the kernel.
  kvmmap(kpgtbl, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);

  // allocate and map a kernel stack for each process.
  proc_mapstacks(kpgtbl);

```



```
  return kpgtbl;
}

// add a mapping to the kernel page table.
// only used when booting.
// does not flush TLB or enable paging.
void
kvmmap(pagetable_t kpgtbl, uint64 va, uint64 pa, uint64 sz, int perm)
{
  if(mappages(kpgtbl, va, sz, pa, perm) != 0)
    panic("kvmmap");
}

// Initialize the kernel_pagetable, shared by all CPUs.
void
kvminit(void)
{
  kernel_pagetable = kvmmake();
}

// Switch the current CPU's h/w page table register to
// the kernel's page table, and enable paging.
void
kvminithart()
{
  // wait for any previous writes to the page table memory to finish.
  sfence_vma();

  w_satp(MAKE_SATP(kernel_pagetable));

  // flush stale entries from the TLB.
  sfence_vma();
}

// Return the address of the PTE in page table pagetable
// that corresponds to virtual address va.  If alloc!=0,
// create any required page-table pages.
//
// The risc-v Sv39 scheme has three levels of page-table
// pages. A page-table page contains 512 64-bit PTEs.
// A 64-bit virtual address is split into five fields:
//   39..63 -- must be zero.
//   30..38 -- 9 bits of level-2 index.
//   21..29 -- 9 bits of level-1 index.
//   12..20 -- 9 bits of level-0 index.
//    0..11 -- 12 bits of byte offset within the page.
pte_t *
walk(pagetable_t pagetable, uint64 va, int alloc)
{
  if(va >= MAXVA)
```

```
    panic("walk");

  for(int level = 2; level > 0; level--) {
    pte_t *pte = &pagetable[PX(level, va)];
    if(*pte & PTE_V) {
      pagetable = (pagetable_t)PTE2PA(*pte);
    } else {
      if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  return &pagetable[PX(0, va)];
}

// Look up a virtual address, return the physical address,
// or 0 if not mapped.
// Can only be used to look up user pages.
uint64
walkaddr(pagetable_t pagetable, uint64 va)
{
  pte_t *pte;
  uint64 pa;

  if(va >= MAXVA)
    return 0;

  pte = walk(pagetable, va, 0);
  if(pte == 0)
    return 0;
  if((*pte & PTE_V) == 0)
    return 0;
  if((*pte & PTE_U) == 0)
    return 0;
  pa = PTE2PA(*pte);
  return pa;
}
```



```
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa.
// va and size MUST be page-aligned.
// Returns 0 on success, -1 if walk() couldn't
// allocate a needed page-table page.
int
mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)
{
  uint64 a, last;
  pte_t *pte;

  if((va % PGSIZE) != 0)
    panic("mappages: va not aligned");

  if((size % PGSIZE) != 0)
    panic("mappages: size not aligned");

  if(size == 0)
    panic("mappages: size");

  a = va;
  last = va + size - PGSIZE;
  for(;;){
    if((pte = walk(pagetable, a, 1)) == 0)
      return -1;
    if(*pte & PTE_V)
      panic("mappages: remap");
    *pte = PA2PTE(pa) | perm | PTE_V;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}

// create an empty user page table.
// returns 0 if out of memory.
pagetable_t
uvmcreate()
{
  pagetable_t pagetable;
  pagetable = (pagetable_t) kalloc();
  if(pagetable == 0)
    return 0;
  memset(pagetable, 0, PGSIZE);
  return pagetable;
}
```

```
1600 // Remove npages of mappings starting from va. va must be
1601 // page-aligned. It's OK if the mappings don't exist.
1602 // Optionally free the physical memory.
1603 void
1604 uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
1605 {
1606   uint64 a;
1607   pte_t *pte;
1608
1609   if((va % PGSIZE) != 0)
1610     panic("uvmunmap: not aligned");
1611
1612   for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
1613     if((pte = walk(pagetable, a, 0)) == 0) // leaf page table entry allocate
1614       continue;
1615     if((*pte & PTE_V) == 0)  // has physical page been allocated?
1616       continue;
1617     if(do_free){
1618       uint64 pa = PTE2PA(*pte);
1619       kfree((void*)pa);
1620     }
1621     *pte = 0;
1622   }
1623 }
1624
1625 // Allocate PTEs and physical memory to grow a process from oldsz to
1626 // newsz, which need not be page aligned.  Returns new size or 0 on error.
1627 uint64
1628 uvmalloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz, int xperm)
1629 {
1630   char *mem;
1631   uint64 a;
1632
1633   if(newsz < oldsz)
1634     return oldsz;
1635
1636   oldsz = PGROUNDUP(oldsz);
1637   for(a = oldsz; a < newsz; a += PGSIZE){
1638     mem = kalloc();
1639     if(mem == 0){
1640       uvmdealloc(pagetable, a, oldsz);
1641       return 0;
1642     }
1643     memset(mem, 0, PGSIZE);
1644     if(mappages(pagetable, a, PGSIZE, (uint64)mem, PTE_R|PTE_U|xperm) != 0){
1645       kfree(mem);
1646       uvmdealloc(pagetable, a, oldsz);
1647       return 0;
1648     }
1649   }
```



```
1650   return newsz;
1651 }
1652
1653 // Deallocate user pages to bring the process size from oldsz to
1654 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
1655 // need to be less than oldsz.  oldsz can be larger than the actual
1656 // process size.  Returns the new process size.
1657 uint64
1658 uvmdealloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz)
1659 {
1660   if(newsz >= oldsz)
1661     return oldsz;
1662
1663   if(PGROUNDUP(newsz) < PGROUNDUP(oldsz)){
1664     int npages = (PGROUNDUP(oldsz) - PGROUNDUP(newsz)) / PGSIZE;
1665     uvmunmap(pagetable, PGROUNDUP(newsz), npages, 1);
1666   }
1667
1668   return newsz;
1669 }
1670
1671 // Recursively free page-table pages.
1672 // All leaf mappings must already have been removed.
1673 void
1674 freewalk(pagetable_t pagetable)
1675 {
1676   // there are 2^9 = 512 PTEs in a page table.
1677   for(int i = 0; i < 512; i++){
1678     pte_t pte = pagetable[i];
1679     if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
1680       // this PTE points to a lower-level page table.
1681       uint64 child = PTE2PA(pte);
1682       freewalk((pagetable_t)child);
1683       pagetable[i] = 0;
1684     } else if(pte & PTE_V){
1685       panic("freewalk: leaf");
1686     }
1687   }
1688   kfree((void*)pagetable);
1689 }
1690
1691 // Free user memory pages,
1692 // then free page-table pages.
1693 void
1694 uvmfree(pagetable_t pagetable, uint64 sz)
1695 {
1696   if(sz > 0)
1697     uvmunmap(pagetable, 0, PGROUNDUP(sz)/PGSIZE, 1);
1698   freewalk(pagetable);
1699 }
```

```
1700 // Given a parent process's page table, copy
1701 // its memory into a child's page table.
1702 // Copies both the page table and the
1703 // physical memory.
1704 // returns 0 on success, -1 on failure.
1705 // frees any allocated pages on failure.
1706 int
1707 uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
1708 {
1709   pte_t *pte;
1710   uint64 pa, i;
1711   uint flags;
1712   char *mem;
1713
1714   for(i = 0; i < sz; i += PGSIZE){
1715     if((pte = walk(old, i, 0)) == 0)
1716       continue;   // page table entry hasn't been allocated
1717     if((*pte & PTE_V) == 0)
1718       continue;   // physical page hasn't been allocated
1719     pa = PTE2PA(*pte);
1720     flags = PTE_FLAGS(*pte);
1721     if((mem = kalloc()) == 0)
1722       goto err;
1723     memmove(mem, (char*)pa, PGSIZE);
1724     if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
1725       kfree(mem);
1726       goto err;
1727     }
1728   }
1729   return 0;
1730
1731  err:
1732   uvmunmap(new, 0, i / PGSIZE, 1);
1733   return -1;
1734 }
1735
1736 // mark a PTE invalid for user access.
1737 // used by exec for the user stack guard page.
1738 void
1739 uvmclear(pagetable_t pagetable, uint64 va)
1740 {
1741   pte_t *pte;
1742
1743   pte = walk(pagetable, va, 0);
1744   if(pte == 0)
1745     panic("uvmclear");
1746   *pte &= ~PTE_U;
1747 }
1748
1749
```



```
1750 // Copy from kernel to user.
1751 // Copy len bytes from src to virtual address dstva in a given page table.
1752 // Return 0 on success, -1 on error.
1753 int
1754 copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
1755 {
1756   uint64 n, va0, pa0;
1757   pte_t *pte;
1758
1759   while(len > 0){
1760     va0 = PGROUNDDOWN(dstva);
1761     if(va0 >= MAXVA)
1762       return -1;
1763
1764     pa0 = walkaddr(pagetable, va0);
1765     if(pa0 == 0) {
1766       if((pa0 = vmfault(pagetable, va0, 0)) == 0) {
1767         return -1;
1768       }
1769     }
1770
1771     pte = walk(pagetable, va0, 0);
1772     // forbid copyout over read-only user text pages.
1773     if((*pte & PTE_W) == 0)
1774       return -1;
1775
1776     n = PGSIZE - (dstva - va0);
1777     if(n > len)
1778       n = len;
1779     memmove((void *)(pa0 + (dstva - va0)), src, n);
1780
1781     len -= n;
1782     src += n;
1783     dstva = va0 + PGSIZE;
1784   }
1785   return 0;
1786 }
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
```

```
1800 // Copy from user to kernel.
1801 // Copy len bytes to dst from virtual address srcva in a given page table.
1802 // Return 0 on success, -1 on error.
1803 int
1804 copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
1805 {
1806   uint64 n, va0, pa0;
1807
1808   while(len > 0){
1809     va0 = PGROUNDDOWN(srcva);
1810     pa0 = walkaddr(pagetable, va0);
1811     if(pa0 == 0) {
1812       if((pa0 = vmfault(pagetable, va0, 0)) == 0) {
1813         return -1;
1814       }
1815     }
1816     n = PGSIZE - (srcva - va0);
1817     if(n > len)
1818       n = len;
1819     memmove(dst, (void *)(pa0 + (srcva - va0)), n);
1820
1821     len -= n;
1822     dst += n;
1823     srcva = va0 + PGSIZE;
1824   }
1825   return 0;
1826 }
1827
1828 // Copy a null-terminated string from user to kernel.
1829 // Copy bytes to dst from virtual address srcva in a given page table,
1830 // until a '\0', or max.
1831 // Return 0 on success, -1 on error.
1832 int
1833 copyinstr(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max)
1834 {
1835   uint64 n, va0, pa0;
1836   int got_null = 0;
1837
1838   while(got_null == 0 && max > 0){
1839     va0 = PGROUNDDOWN(srcva);
1840     pa0 = walkaddr(pagetable, va0);
1841     if(pa0 == 0)
1842       return -1;
1843     n = PGSIZE - (srcva - va0);
1844     if(n > max)
1845       n = max;
1846
1847
1848
1849
```



```
1850     char *p = (char *) (pa0 + (srcva - va0));
1851     while(n > 0){
1852       if(*p == '\0'){
1853         *dst = '\0';
1854         got_null = 1;
1855         break;
1856       } else {
1857         *dst = *p;
1858       }
1859       --n;
1860       --max;
1861       p++;
1862       dst++;
1863     }
1864
1865     srcva = va0 + PGSIZE;
1866   }
1867   if(got_null){
1868     return 0;
1869   } else {
1870     return -1;
1871   }
1872 }
1873
1874 // allocate and map user memory if process is referencing a page
1875 // that was lazily allocated in sys_sbrk().
1876 // returns 0 if va is invalid or already mapped, or if
1877 // out of physical memory, and physical address if successful.
1878 uint64
1879 vmfault(pagetable_t pagetable, uint64 va, int read)
1880 {
1881   uint64 mem;
1882   struct proc *p = myproc();
1883
1884   if (va >= p->sz)
1885     return 0;
1886   va = PGROUNDDOWN(va);
1887   if(ismapped(pagetable, va)) {
1888     return 0;
1889   }
1890   mem = (uint64) kalloc();
1891   if(mem == 0)
1892     return 0;
1893   memset((void *) mem, 0, PGSIZE);
1894   if (mappages(p->pagetable, va, PGSIZE, mem, PTE_W|PTE_U|PTE_R) != 0) {
1895     kfree((void *)mem);
1896     return 0;
1897   }
1898   return mem;
1899 }
```

```
1900 int
1901 ismapped(pagetable_t pagetable, uint64 va)
1902 {
1903   pte_t *pte = walk(pagetable, va, 0);
1904   if (pte == 0) {
1905     return 0;
1906   }
1907   if (*pte & PTE_V){
1908     return 1;
1909   }
1910   return 0;
1911 }
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
```



```
1950 // Saved registers for kernel context switches.
1951 struct context {
1952   uint64 ra;
1953   uint64 sp;
1954
1955   // callee-saved
1956   uint64 s0;
1957   uint64 s1;
1958   uint64 s2;
1959   uint64 s3;
1960   uint64 s4;
1961   uint64 s5;
1962   uint64 s6;
1963   uint64 s7;
1964   uint64 s8;
1965   uint64 s9;
1966   uint64 s10;
1967   uint64 s11;
1968 };
1969
1970 // Per-CPU state.
1971 struct cpu {
1972   struct proc *proc;          // The process running on this cpu, or null.
1973   struct context context;     // swtch() here to enter scheduler().
1974   int noff;                   // Depth of push_off() nesting.
1975   int intena;                 // Were interrupts enabled before push_off()?
1976 };
1977
1978 extern struct cpu cpus[NCPU];
1979
1980 // per-process data for the trap handling code in trampoline.S.
1981 // sits in a page by itself just under the trampoline page in the
1982 // user page table. not specially mapped in the kernel page table.
1983 // uservec in trampoline.S saves user registers in the trapframe,
1984 // then initializes registers from the trapframe's
1985 // kernel_sp, kernel_hartid, kernel_satp, and jumps to kernel_trap.
1986 // usertrapret() and userret in trampoline.S set up
1987 // the trapframe's kernel_*, restore user registers from the
1988 // trapframe, switch to the user page table, and enter user space.
1989 // the trapframe includes callee-saved user registers like s0-s11 because th
1990 // return-to-user path via usertrapret() doesn't return through
1991 // the entire kernel call stack.
1992 struct trapframe {
1993   /*   0 */ uint64 kernel_satp;   // kernel page table
1994   /*   8 */ uint64 kernel_sp;     // top of process's kernel stack
1995   /*  16 */ uint64 kernel_trap;   // usertrap()
1996   /*  24 */ uint64 epc;           // saved user program counter
1997   /*  32 */ uint64 kernel_hartid; // saved kernel tp
1998   /*  40 */ uint64 ra;
1999   /*  48 */ uint64 sp;
```

```
  /*  56 */ uint64 gp;
  /*  64 */ uint64 tp;
  /*  72 */ uint64 t0;
  /*  80 */ uint64 t1;
  /*  88 */ uint64 t2;
  /*  96 */ uint64 s0;
  /* 104 */ uint64 s1;
  /* 112 */ uint64 a0;
  /* 120 */ uint64 a1;
  /* 128 */ uint64 a2;
  /* 136 */ uint64 a3;
  /* 144 */ uint64 a4;
  /* 152 */ uint64 a5;
  /* 160 */ uint64 a6;
  /* 168 */ uint64 a7;
  /* 176 */ uint64 s2;
  /* 184 */ uint64 s3;
  /* 192 */ uint64 s4;
  /* 200 */ uint64 s5;
  /* 208 */ uint64 s6;
  /* 216 */ uint64 s7;
  /* 224 */ uint64 s8;
  /* 232 */ uint64 s9;
  /* 240 */ uint64 s10;
  /* 248 */ uint64 s11;
  /* 256 */ uint64 t3;
  /* 264 */ uint64 t4;
  /* 272 */ uint64 t5;
  /* 280 */ uint64 t6;
};

enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wai
  int pid;                     // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process
```



```
  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

```
2100 #include "types.h"
2101 #include "param.h"
2102 #include "memlayout.h"
2103 #include "riscv.h"
2104 #include "spinlock.h"
2105 #include "proc.h"
2106 #include "defs.h"
2107
2108 struct cpu cpus[NCPU];
2109
2110 struct proc proc[NPROC];
2111
2112 struct proc *initproc;
2113
2114 int nextpid = 1;
2115 struct spinlock pid_lock;
2116
2117 extern void forkret(void);
2118 static void freeproc(struct proc *p);
2119
2120 extern char trampoline[]; // trampoline.S
2121
2122 // helps ensure that wakeups of wait()ing
2123 // parents are not lost. helps obey the
2124 // memory model when using p->parent.
2125 // must be acquired before any p->lock.
2126 struct spinlock wait_lock;
2127
2128 // Allocate a page for each process's kernel stack.
2129 // Map it high in memory, followed by an invalid
2130 // guard page.
2131 void
2132 proc_mapstacks(pagetable_t kpgtbl)
2133 {
2134   struct proc *p;
2135
2136   for(p = proc; p < &proc[NPROC]; p++) {
2137     char *pa = kalloc();
2138     if(pa == 0)
2139       panic("kalloc");
2140     uint64 va = KSTACK((int) (p - proc));
2141     kvmmap(kpgtbl, va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
2142   }
2143 }
2144
2145
2146
2147
2148
2149
```



```
2150 // initialize the proc table.
2151 void
2152 procinit(void)
2153 {
2154   struct proc *p;
2155
2156   initlock(&pid_lock, "nextpid");
2157   initlock(&wait_lock, "wait_lock");
2158   for(p = proc; p < &proc[NPROC]; p++) {
2159       initlock(&p->lock, "proc");
2160       p->state = UNUSED;
2161       p->kstack = KSTACK((int) (p - proc));
2162   }
2163 }
2164
2165 // Must be called with interrupts disabled,
2166 // to prevent race with process being moved
2167 // to a different CPU.
2168 int
2169 cpuid()
2170 {
2171   int id = r_tp();
2172   return id;
2173 }
2174
2175 // Return this CPU's cpu struct.
2176 // Interrupts must be disabled.
2177 struct cpu*
2178 mycpu(void)
2179 {
2180   int id = cpuid();
2181   struct cpu *c = &cpus[id];
2182   return c;
2183 }
2184
2185 // Return the current struct proc *, or zero if none.
2186 struct proc*
2187 myproc(void)
2188 {
2189   push_off();
2190   struct cpu *c = mycpu();
2191   struct proc *p = c->proc;
2192   pop_off();
2193   return p;
2194 }
2195
2196
2197
2198
2199
```

```
2200 int
2201 allocpid()
2202 {
2203   int pid;
2204
2205   acquire(&pid_lock);
2206   pid = nextpid;
2207   nextpid = nextpid + 1;
2208   release(&pid_lock);
2209
2210   return pid;
2211 }
2212
2213 // Look in the process table for an UNUSED proc.
2214 // If found, initialize state required to run in the kernel,
2215 // and return with p->lock held.
2216 // If there are no free procs, or a memory allocation fails, return 0.
2217 static struct proc*
2218 allocproc(void)
2219 {
2220   struct proc *p;
2221
2222   for(p = proc; p < &proc[NPROC]; p++) {
2223     acquire(&p->lock);
2224     if(p->state == UNUSED) {
2225       goto found;
2226     } else {
2227       release(&p->lock);
2228     }
2229   }
2230   return 0;
2231
2232 found:
2233   p->pid = allocpid();
2234   p->state = USED;
2235
2236   // Allocate a trapframe page.
2237   if((p->trapframe = (struct trapframe *)kalloc()) == 0){
2238     freeproc(p);
2239     release(&p->lock);
2240     return 0;
2241   }
2242
2243   // An empty user page table.
2244   p->pagetable = proc_pagetable(p);
2245   if(p->pagetable == 0){
2246     freeproc(p);
2247     release(&p->lock);
2248     return 0;
2249   }
```



```
2250   // Set up new context to start executing at forkret,
2251   // which returns to user space.
2252   memset(&p->context, 0, sizeof(p->context));
2253   p->context.ra = (uint64)forkret;
2254   p->context.sp = p->kstack + PGSIZE;
2255
2256   return p;
2257 }
2258
2259 // free a proc structure and the data hanging from it,
2260 // including user pages.
2261 // p->lock must be held.
2262 static void
2263 freeproc(struct proc *p)
2264 {
2265   if(p->trapframe)
2266     kfree((void*)p->trapframe);
2267   p->trapframe = 0;
2268   if(p->pagetable)
2269     proc_freepagetable(p->pagetable, p->sz);
2270   p->pagetable = 0;
2271   p->sz = 0;
2272   p->pid = 0;
2273   p->parent = 0;
2274   p->name[0] = 0;
2275   p->chan = 0;
2276   p->killed = 0;
2277   p->xstate = 0;
2278   p->state = UNUSED;
2279 }
2280
2281 // Create a user page table for a given process, with no user memory,
2282 // but with trampoline and trapframe pages.
2283 pagetable_t
2284 proc_pagetable(struct proc *p)
2285 {
2286   pagetable_t pagetable;
2287
2288   // An empty page table.
2289   pagetable = uvmcreate();
2290   if(pagetable == 0)
2291     return 0;
2292
2293   // map the trampoline code (for system call return)
2294   // at the highest user virtual address.
2295   // only the supervisor uses it, on the way
2296   // to/from user space, so not PTE_U.
2297   if(mappages(pagetable, TRAMPOLINE, PGSIZE,
2298               (uint64)trampoline, PTE_R | PTE_X) < 0){
2299     uvmfree(pagetable, 0);
```

```
    return 0;
  }

  // map the trapframe page just below the trampoline page, for
  // trampoline.S.
  if(mappages(pagetable, TRAPFRAME, PGSIZE,
              (uint64)(p->trapframe), PTE_R | PTE_W) < 0){
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmfree(pagetable, 0);
    return 0;
  }

  return pagetable;
}

// Free a process's page table, and free the
// physical memory it refers to.
void
proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  uvmfree(pagetable, sz);
}

// Set up first user process.
void
userinit(void)
{
  struct proc *p;

  p = allocproc();
  initproc = p;

  p->cwd = namei("/");

  p->state = RUNNABLE;

  release(&p->lock);
}
```



```
// Shrink user memory by n bytes.
// Return 0 on success, -1 on failure.
int
growproc(int n)
{
  uint64 sz;
  struct proc *p = myproc();

  sz = p->sz;
  if(n > 0){
    if((sz = uvmalloc(p->pagetable, sz, sz + n, PTE_W)) == 0) {
      return -1;
    }
  } else if(n < 0){
    sz = uvmdealloc(p->pagetable, sz, sz + n);
  }
  p->sz = sz;
  return 0;
}

// Create a new process, copying the parent.
// Sets up child kernel stack to return as if from fork() system call.
int
kfork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz;

  // copy saved user registers.
  *(np->trapframe) = *(p->trapframe);

  // Cause fork to return 0 in the child.
  np->trapframe->a0 = 0;
```

```
2400   // increment reference counts on open file descriptors.
2401   for(i = 0; i < NOFILE; i++)
2402     if(p->ofile[i])
2403       np->ofile[i] = filedup(p->ofile[i]);
2404   np->cwd = idup(p->cwd);
2405
2406   safestrcpy(np->name, p->name, sizeof(p->name));
2407
2408   pid = np->pid;
2409
2410   release(&np->lock);
2411
2412   acquire(&wait_lock);
2413   np->parent = p;
2414   release(&wait_lock);
2415
2416   acquire(&np->lock);
2417   np->state = RUNNABLE;
2418   release(&np->lock);
2419
2420   return pid;
2421 }
2422
2423 // Pass p's abandoned children to init.
2424 // Caller must hold wait_lock.
2425 void
2426 reparent(struct proc *p)
2427 {
2428   struct proc *pp;
2429
2430   for(pp = proc; pp < &proc[NPROC]; pp++){
2431     if(pp->parent == p){
2432       pp->parent = initproc;
2433       wakeup(initproc);
2434     }
2435   }
2436 }
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
```



```
2450 // Exit the current process.  Does not return.
2451 // An exited process remains in the zombie state
2452 // until its parent calls wait().
2453 void
2454 kexit(int status)
2455 {
2456   struct proc *p = myproc();
2457
2458   if(p == initproc)
2459     panic("init exiting");
2460
2461   // Close all open files.
2462   for(int fd = 0; fd < NOFILE; fd++){
2463     if(p->ofile[fd]){
2464       struct file *f = p->ofile[fd];
2465       fileclose(f);
2466       p->ofile[fd] = 0;
2467     }
2468   }
2469
2470   begin_op();
2471   iput(p->cwd);
2472   end_op();
2473   p->cwd = 0;
2474
2475   acquire(&wait_lock);
2476
2477   // Give any children to init.
2478   reparent(p);
2479
2480   // Parent might be sleeping in wait().
2481   wakeup(p->parent);
2482
2483   acquire(&p->lock);
2484
2485   p->xstate = status;
2486   p->state = ZOMBIE;
2487
2488   release(&wait_lock);
2489
2490   // Jump into the scheduler, never to return.
2491   sched();
2492   panic("zombie exit");
2493 }
2494
2495
2496
2497
2498
2499
```

```
2500 // Wait for a child process to exit and return its pid.
2501 // Return -1 if this process has no children.
2502 int
2503 kwait(uint64 addr)
2504 {
2505   struct proc *pp;
2506   int havekids, pid;
2507   struct proc *p = myproc();
2508
2509   acquire(&wait_lock);
2510
2511   for(;;){
2512     // Scan through table looking for exited children.
2513     havekids = 0;
2514     for(pp = proc; pp < &proc[NPROC]; pp++){
2515       if(pp->parent == p){
2516         // make sure the child isn't still in exit() or swtch().
2517         acquire(&pp->lock);
2518
2519         havekids = 1;
2520         if(pp->state == ZOMBIE){
2521           // Found one.
2522           pid = pp->pid;
2523           if(addr != 0 && copyout(p->pagetable, addr, (char *)&pp->xstate,
2524                                   sizeof(pp->xstate)) < 0) {
2525             release(&pp->lock);
2526             release(&wait_lock);
2527             return -1;
2528           }
2529           freeproc(pp);
2530           release(&pp->lock);
2531           release(&wait_lock);
2532           return pid;
2533         }
2534         release(&pp->lock);
2535       }
2536     }
2537
2538     // No point waiting if we don't have any children.
2539     if(!havekids || killed(p)){
2540       release(&wait_lock);
2541       return -1;
2542     }
2543
2544     // Wait for a child to exit.
2545     sleep(p, &wait_lock);
2546   }
2547 }
2548
2549
```



```
2550 // Per-CPU process scheduler.
2551 // Each CPU calls scheduler() after setting itself up.
2552 // Scheduler never returns.  It loops, doing:
2553 //  - choose a process to run.
2554 //  - swtch to start running that process.
2555 //  - eventually that process transfers control
2556 //    via swtch back to the scheduler.
2557 void
2558 scheduler(void)
2559 {
2560   struct proc *p;
2561   struct cpu *c = mycpu();
2562
2563   c->proc = 0;
2564   for(;;){
2565     // The most recent process to run may have had interrupts
2566     // turned off; enable them to avoid a deadlock if all
2567     // processes are waiting. Then turn them back off
2568     // to avoid a possible race between an interrupt
2569     // and wfi.
2570     intr_on();
2571     intr_off();
2572
2573     int found = 0;
2574     for(p = proc; p < &proc[NPROC]; p++) {
2575       acquire(&p->lock);
2576       if(p->state == RUNNABLE) {
2577         // Switch to chosen process.  It is the process's job
2578         // to release its lock and then reacquire it
2579         // before jumping back to us.
2580         p->state = RUNNING;
2581         c->proc = p;
2582         swtch(&c->context, &p->context);
2583
2584         // Process is done running for now.
2585         // It should have changed its p->state before coming back.
2586         c->proc = 0;
2587         found = 1;
2588       }
2589       release(&p->lock);
2590     }
2591     if(found == 0) {
2592       // nothing to run; stop running on this core until an interrupt.
2593       asm volatile("wfi");
2594     }
2595   }
2596 }
2597
2598
2599
```

```
2600 // Switch to scheduler.  Must hold only p->lock
2601 // and have changed proc->state. Saves and restores
2602 // intena because intena is a property of this
2603 // kernel thread, not this CPU. It should
2604 // be proc->intena and proc->noff, but that would
2605 // break in the few places where a lock is held but
2606 // there's no process.
2607 void
2608 sched(void)
2609 {
2610   int intena;
2611   struct proc *p = myproc();
2612
2613   if(!holding(&p->lock))
2614     panic("sched p->lock");
2615   if(mycpu()->noff != 1)
2616     panic("sched locks");
2617   if(p->state == RUNNING)
2618     panic("sched RUNNING");
2619   if(intr_get())
2620     panic("sched interruptible");
2621
2622   intena = mycpu()->intena;
2623   swtch(&p->context, &mycpu()->context);
2624   mycpu()->intena = intena;
2625 }
2626
2627 // Give up the CPU for one scheduling round.
2628 void
2629 yield(void)
2630 {
2631   struct proc *p = myproc();
2632   acquire(&p->lock);
2633   p->state = RUNNABLE;
2634   sched();
2635   release(&p->lock);
2636 }
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
```



```
2650 // A fork child's very first scheduling by scheduler()
2651 // will swtch to forkret.
2652 void
2653 forkret(void)
2654 {
2655   extern char userret[];
2656   static int first = 1;
2657   struct proc *p = myproc();
2658
2659   // Still holding p->lock from scheduler.
2660   release(&p->lock);
2661
2662   if (first) {
2663     // File system initialization must be run in the context of a
2664     // regular process (e.g., because it calls sleep), and thus cannot
2665     // be run from main().
2666     fsinit(ROOTDEV);
2667
2668     first = 0;
2669     // ensure other cores see first=0.
2670     __sync_synchronize();
2671
2672     // We can invoke kexec() now that file system is initialized.
2673     // Put the return value (argc) of kexec into a0.
2674     p->trapframe->a0 = kexec("/init", (char *[]){ "/init", 0 });
2675     if (p->trapframe->a0 == -1) {
2676       panic("exec");
2677     }
2678   }
2679
2680   // return to user space, mimicing usertrap()'s return.
2681   prepare_return();
2682   uint64 satp = MAKE_SATP(p->pagetable);
2683   uint64 trampoline_userret = TRAMPOLINE + (userret - trampoline);
2684   ((void (*)(uint64))trampoline_userret)(satp);
2685 }
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
```

```
2700 // Sleep on channel chan, releasing condition lock lk.
2701 // Re-acquires lk when awakened.
2702 void
2703 sleep(void *chan, struct spinlock *lk)
2704 {
2705   struct proc *p = myproc();
2706
2707   // Must acquire p->lock in order to
2708   // change p->state and then call sched.
2709   // Once we hold p->lock, we can be
2710   // guaranteed that we won't miss any wakeup
2711   // (wakeup locks p->lock),
2712   // so it's okay to release lk.
2713
2714   acquire(&p->lock);
2715   release(lk);
2716
2717   // Go to sleep.
2718   p->chan = chan;
2719   p->state = SLEEPING;
2720
2721   sched();
2722
2723   // Tidy up.
2724   p->chan = 0;
2725
2726   // Reacquire original lock.
2727   release(&p->lock);
2728   acquire(lk);
2729 }
2730
2731 // Wake up all processes sleeping on channel chan.
2732 // Caller should hold the condition lock.
2733 void
2734 wakeup(void *chan)
2735 {
2736   struct proc *p;
2737
2738   for(p = proc; p < &proc[NPROC]; p++) {
2739     if(p != myproc()){
2740       acquire(&p->lock);
2741       if(p->state == SLEEPING && p->chan == chan) {
2742         p->state = RUNNABLE;
2743       }
2744       release(&p->lock);
2745     }
2746   }
2747 }
2748
2749
```



```
2750 // Kill the process with the given pid.
2751 // The victim won't exit until it tries to return
2752 // to user space (see usertrap() in trap.c).
2753 int
2754 kkill(int pid)
2755 {
2756   struct proc *p;
2757
2758   for(p = proc; p < &proc[NPROC]; p++){
2759     acquire(&p->lock);
2760     if(p->pid == pid){
2761       p->killed = 1;
2762       if(p->state == SLEEPING){
2763         // Wake process from sleep().
2764         p->state = RUNNABLE;
2765       }
2766       release(&p->lock);
2767       return 0;
2768     }
2769     release(&p->lock);
2770   }
2771   return -1;
2772 }
2773
2774 void
2775 setkilled(struct proc *p)
2776 {
2777   acquire(&p->lock);
2778   p->killed = 1;
2779   release(&p->lock);
2780 }
2781
2782 int
2783 killed(struct proc *p)
2784 {
2785   int k;
2786
2787   acquire(&p->lock);
2788   k = p->killed;
2789   release(&p->lock);
2790   return k;
2791 }
2792
2793
2794
2795
2796
2797
2798
2799
```

```
2800 // Copy to either a user address, or kernel address,
2801 // depending on usr_dst.
2802 // Returns 0 on success, -1 on error.
2803 int
2804 either_copyout(int user_dst, uint64 dst, void *src, uint64 len)
2805 {
2806   struct proc *p = myproc();
2807   if(user_dst){
2808     return copyout(p->pagetable, dst, src, len);
2809   } else {
2810     memmove((char *)dst, src, len);
2811     return 0;
2812   }
2813 }
2814
2815 // Copy from either a user address, or kernel address,
2816 // depending on usr_src.
2817 // Returns 0 on success, -1 on error.
2818 int
2819 either_copyin(void *dst, int user_src, uint64 src, uint64 len)
2820 {
2821   struct proc *p = myproc();
2822   if(user_src){
2823     return copyin(p->pagetable, dst, src, len);
2824   } else {
2825     memmove(dst, (char*)src, len);
2826     return 0;
2827   }
2828 }
2829
2830 // Print a process listing to console.  For debugging.
2831 // Runs when user types ^P on console.
2832 // No lock to avoid wedging a stuck machine further.
2833 void
2834 procdump(void)
2835 {
2836   static char *states[] = {
2837   [UNUSED]    "unused",
2838   [USED]      "used",
2839   [SLEEPING]  "sleep ",
2840   [RUNNABLE]  "runble",
2841   [RUNNING]   "run   ",
2842   [ZOMBIE]    "zombie"
2843   };
2844   struct proc *p;
2845   char *state;
2846
2847   printf("\n");
2848   for(p = proc; p < &proc[NPROC]; p++){
2849     if(p->state == UNUSED)
```



```
2850       continue;
2851     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
2852       state = states[p->state];
2853     else
2854       state = "???";
2855     printf("%d %s %s", p->pid, state, p->name);
2856     printf("\n");
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
```

```
# Context switch
#
#   void swtch(struct context *old, struct context *new);
#
# Save current registers in old. Load from new.


.globl swtch
swtch:
        sd ra, 0(a0)
        sd sp, 8(a0)
        sd s0, 16(a0)
        sd s1, 24(a0)
        sd s2, 32(a0)
        sd s3, 40(a0)
        sd s4, 48(a0)
        sd s5, 56(a0)
        sd s6, 64(a0)
        sd s7, 72(a0)
        sd s8, 80(a0)
        sd s9, 88(a0)
        sd s10, 96(a0)
        sd s11, 104(a0)

        ld ra, 0(a1)
        ld sp, 8(a1)
        ld s0, 16(a1)
        ld s1, 24(a1)
        ld s2, 32(a1)
        ld s3, 40(a1)
        ld s4, 48(a1)
        ld s5, 56(a1)
        ld s6, 64(a1)
        ld s7, 72(a1)
        ld s8, 80(a1)
        ld s9, 88(a1)
        ld s10, 96(a1)
        ld s11, 104(a1)

        ret
```



```
// Physical memory allocator, for user processes,
// kernel stacks, page-table pages,
// and pipe buffers. Allocates whole 4096-byte pages.

#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "spinlock.h"
#include "riscv.h"
#include "defs.h"

void freerange(void *pa_start, void *pa_end);

extern char end[]; // first address after kernel.
                   // defined by kernel.ld.

struct run {
  struct run *next;
};

struct {
  struct spinlock lock;
  struct run *freelist;
} kmem;

void
kinit()
{
  initlock(&kmem.lock, "kmem");
  freerange(end, (void*)PHYSTOP);
}

void
freerange(void *pa_start, void *pa_end)
{
  char *p;
  p = (char*)PGROUNDUP((uint64)pa_start);
  for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
    kfree(p);
}
```

```
3000 // Free the page of physical memory pointed at by pa,
3001 // which normally should have been returned by a
3002 // call to kalloc().  (The exception is when
3003 // initializing the allocator; see kinit above.)
3004 void
3005 kfree(void *pa)
3006 {
3007   struct run *r;
3008
3009   if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
3010     panic("kfree");
3011
3012   // Fill with junk to catch dangling refs.
3013   memset(pa, 1, PGSIZE);
3014
3015   r = (struct run*)pa;
3016
3017   acquire(&kmem.lock);
3018   r->next = kmem.freelist;
3019   kmem.freelist = r;
3020   release(&kmem.lock);
3021 }
3022
3023 // Allocate one 4096-byte page of physical memory.
3024 // Returns a pointer that the kernel can use.
3025 // Returns 0 if the memory cannot be allocated.
3026 void *
3027 kalloc(void)
3028 {
3029   struct run *r;
3030
3031   acquire(&kmem.lock);
3032   r = kmem.freelist;
3033   if(r)
3034     kmem.freelist = r->next;
3035   release(&kmem.lock);
3036
3037   if(r)
3038     memset((char*)r, 5, PGSIZE); // fill with junk
3039   return (void*)r;
3040 }
3041
3042
3043
3044
3045
3046
3047
3048
3049
```



```
3050         #
3051         # low-level code to handle traps from user space into
3052         # the kernel, and returns from kernel to user.
3053         #
3054         # the kernel maps the page holding this code
3055         # at the same virtual address (TRAMPOLINE)
3056         # in user and kernel space so that it continues
3057         # to work when it switches page tables.
3058         # kernel.ld causes this code to start at
3059         # a page boundary.
3060         #
3061
3062 #include "riscv.h"
3063 #include "memlayout.h"
3064
3065 .section trampsec
3066 .globl trampoline
3067 .globl usertrap
3068 trampoline:
3069 .align 4
3070 .globl uservec
3071 uservec:
3072    #
3073         # trap.c sets stvec to point here, so
3074         # traps from user space start here,
3075         # in supervisor mode, but with a
3076         # user page table.
3077         #
3078
3079         # save user a0 in sscratch so
3080         # a0 can be used to get at TRAPFRAME.
3081         csrw sscratch, a0
3082
3083         # each process has a separate p->trapframe memory area,
3084         # but it's mapped to the same virtual address
3085         # (TRAPFRAME) in every process's user page table.
3086         li a0, TRAPFRAME
3087
3088         # save the user registers in TRAPFRAME
3089         sd ra, 40(a0)
3090         sd sp, 48(a0)
3091         sd gp, 56(a0)
3092         sd tp, 64(a0)
3093         sd t0, 72(a0)
3094         sd t1, 80(a0)
3095         sd t2, 88(a0)
3096         sd s0, 96(a0)
3097         sd s1, 104(a0)
3098         sd a1, 120(a0)
3099         sd a2, 128(a0)
```

```
3100        sd a3, 136(a0)
3101        sd a4, 144(a0)
3102        sd a5, 152(a0)
3103        sd a6, 160(a0)
3104        sd a7, 168(a0)
3105        sd s2, 176(a0)
3106        sd s3, 184(a0)
3107        sd s4, 192(a0)
3108        sd s5, 200(a0)
3109        sd s6, 208(a0)
3110        sd s7, 216(a0)
3111        sd s8, 224(a0)
3112        sd s9, 232(a0)
3113        sd s10, 240(a0)
3114        sd s11, 248(a0)
3115        sd t3, 256(a0)
3116        sd t4, 264(a0)
3117        sd t5, 272(a0)
3118        sd t6, 280(a0)
3119
3120    # save the user a0 in p->trapframe->a0
3121        csrr t0, sscratch
3122        sd t0, 112(a0)
3123
3124        # initialize kernel stack pointer, from p->trapframe->kernel_sp
3125        ld sp, 8(a0)
3126
3127        # make tp hold the current hartid, from p->trapframe->kernel_hartid
3128        ld tp, 32(a0)
3129
3130        # load the address of usertrap(), from p->trapframe->kernel_trap
3131        ld t0, 16(a0)
3132
3133        # fetch the kernel page table address, from p->trapframe->kernel_sat
3134        ld t1, 0(a0)
3135
3136        # wait for any previous memory operations to complete, so that
3137        # they use the user page table.
3138        sfence.vma zero, zero
3139
3140        # install the kernel page table.
3141        csrw satp, t1
3142
3143        # flush now-stale user entries from the TLB.
3144        sfence.vma zero, zero
3145
3146        # call usertrap()
3147        jalr t0
3148
3149
```



```
3150 .globl userret
3151 userret:
3152        # usertrap() returns here, with user satp in a0.
3153        # return from kernel to user.
3154
3155        # switch to the user page table.
3156        sfence.vma zero, zero
3157        csrw satp, a0
3158        sfence.vma zero, zero
3159
3160        li a0, TRAPFRAME
3161
3162        # restore all but a0 from TRAPFRAME
3163        ld ra, 40(a0)
3164        ld sp, 48(a0)
3165        ld gp, 56(a0)
3166        ld tp, 64(a0)
3167        ld t0, 72(a0)
3168        ld t1, 80(a0)
3169        ld t2, 88(a0)
3170        ld s0, 96(a0)
3171        ld s1, 104(a0)
3172        ld a1, 120(a0)
3173        ld a2, 128(a0)
3174        ld a3, 136(a0)
3175        ld a4, 144(a0)
3176        ld a5, 152(a0)
3177        ld a6, 160(a0)
3178        ld a7, 168(a0)
3179        ld s2, 176(a0)
3180        ld s3, 184(a0)
3181        ld s4, 192(a0)
3182        ld s5, 200(a0)
3183        ld s6, 208(a0)
3184        ld s7, 216(a0)
3185        ld s8, 224(a0)
3186        ld s9, 232(a0)
3187        ld s10, 240(a0)
3188        ld s11, 248(a0)
3189        ld t3, 256(a0)
3190        ld t4, 264(a0)
3191        ld t5, 272(a0)
3192        ld t6, 280(a0)
3193
3194    # restore user a0
3195        ld a0, 112(a0)
3196
3197        # return to user mode and user pc.
3198        # usertrapret() set up sstatus and sepc.
3199        sret
```

```
3200         #
3201         # interrupts and exceptions while in supervisor
3202         # mode come here.
3203         #
3204         # the current stack is a kernel stack.
3205         # push registers, call kerneltrap().
3206         # when kerneltrap() returns, restore registers, return.
3207         #
3208 .globl kerneltrap
3209 .globl kernelvec
3210 .align 4
3211 kernelvec:
3212         # make room to save registers.
3213         addi sp, sp, -256
3214
3215         # save caller-saved registers.
3216         sd ra, 0(sp)
3217         # sd sp, 8(sp)
3218         sd gp, 16(sp)
3219         sd tp, 24(sp)
3220         sd t0, 32(sp)
3221         sd t1, 40(sp)
3222         sd t2, 48(sp)
3223         sd a0, 72(sp)
3224         sd a1, 80(sp)
3225         sd a2, 88(sp)
3226         sd a3, 96(sp)
3227         sd a4, 104(sp)
3228         sd a5, 112(sp)
3229         sd a6, 120(sp)
3230         sd a7, 128(sp)
3231         sd t3, 216(sp)
3232         sd t4, 224(sp)
3233         sd t5, 232(sp)
3234         sd t6, 240(sp)
3235
3236         # call the C trap handler in trap.c
3237         call kerneltrap
3238
3239         # restore registers.
3240         ld ra, 0(sp)
3241         # ld sp, 8(sp)
3242         ld gp, 16(sp)
3243         # not tp (contains hartid), in case we moved CPUs
3244         ld t0, 32(sp)
3245         ld t1, 40(sp)
3246         ld t2, 48(sp)
3247         ld a0, 72(sp)
3248         ld a1, 80(sp)
3249         ld a2, 88(sp)
```



```
3250         ld a3, 96(sp)
3251         ld a4, 104(sp)
3252         ld a5, 112(sp)
3253         ld a6, 120(sp)
3254         ld a7, 128(sp)
3255         ld t3, 216(sp)
3256         ld t4, 224(sp)
3257         ld t5, 232(sp)
3258         ld t6, 240(sp)
3259
3260         addi sp, sp, 256
3261
3262         # return to whatever we were doing in the kernel.
3263         sret
3264
3265
3266
3267
3268
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
```

```
#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "riscv.h"
#include "spinlock.h"
#include "proc.h"
#include "defs.h"

struct spinlock tickslock;
uint ticks;

extern char trampoline[], uservec[];

// in kernelvec.S, calls kerneltrap().
void kernelvec();

extern int devintr();

void
trapinit(void)
{
  initlock(&tickslock, "time");
}

// set up to take exceptions and traps while in the kernel.
void
trapinithart(void)
{
  w_stvec((uint64)kernelvec);
}

//
// handle an interrupt, exception, or system call from user space.
// called from, and returns to, trampoline.S
// return value is user satp for trampoline.S to switch to.
//
uint64
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

```



```
  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(killed(p))
      kexit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sepc, scause, and sstatus,
    // so enable only now that we're done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else if((r_scause() == 15 || r_scause() == 13) &&
            vmfault(p->pagetable, r_stval(), (r_scause() == 13)? 1 : 0) != 0
    // page fault on lazily-allocated page
  } else {
    printf("usertrap(): unexpected scause 0x%lx pid=%d\n", r_scause(), p->pi
    printf("            sepc=0x%lx stval=0x%lx\n", r_sepc(), r_stval());
    setkilled(p);
  }

  if(killed(p))
    kexit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  prepare_return();

  // the user page table to switch to, for trampoline.S
  uint64 satp = MAKE_SATP(p->pagetable);

  // return to trampoline.S; satp value in a0.
  return satp;
}


```

```
3400 //
3401 // set up trapframe and control registers for a return to user space
3402 //
3403 void
3404 prepare_return(void)
3405 {
3406   struct proc *p = myproc();
3407
3408   // we're about to switch the destination of traps from
3409   // kerneltrap() to usertrap(). because a trap from kernel
3410   // code to usertrap would be a disaster, turn off interrupts.
3411   intr_off();
3412
3413   // send syscalls, interrupts, and exceptions to uservec in trampoline.S
3414   uint64 trampoline_uservec = TRAMPOLINE + (uservec - trampoline);
3415   w_stvec(trampoline_uservec);
3416
3417   // set up trapframe values that uservec will need when
3418   // the process next traps into the kernel.
3419   p->trapframe->kernel_satp = r_satp();         // kernel page table
3420   p->trapframe->kernel_sp = p->kstack + PGSIZE; // process's kernel stack
3421   p->trapframe->kernel_trap = (uint64)usertrap;
3422   p->trapframe->kernel_hartid = r_tp();         // hartid for cpuid()
3423
3424   // set up the registers that trampoline.S's sret will use
3425   // to get to user space.
3426
3427   // set S Previous Privilege mode to User.
3428   unsigned long x = r_sstatus();
3429   x &= ~SSTATUS_SPP; // clear SPP to 0 for user mode
3430   x |= SSTATUS_SPIE; // enable interrupts in user mode
3431   w_sstatus(x);
3432
3433   // set S Exception Program Counter to the saved user pc.
3434   w_sepc(p->trapframe->epc);
3435 }
3436
3437
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
```



```
3450 // interrupts and exceptions from kernel code go here via kernelvec,
3451 // on whatever the current kernel stack is.
3452 void
3453 kerneltrap()
3454 {
3455   int which_dev = 0;
3456   uint64 sepc = r_sepc();
3457   uint64 sstatus = r_sstatus();
3458   uint64 scause = r_scause();
3459
3460   if((sstatus & SSTATUS_SPP) == 0)
3461     panic("kerneltrap: not from supervisor mode");
3462   if(intr_get() != 0)
3463     panic("kerneltrap: interrupts enabled");
3464
3465   if((which_dev = devintr()) == 0){
3466     // interrupt or trap from an unknown source
3467     printf("scause=0x%lx sepc=0x%lx stval=0x%lx\n", scause, r_sepc(), r_stva
3468     panic("kerneltrap");
3469   }
3470
3471   // give up the CPU if this is a timer interrupt.
3472   if(which_dev == 2 && myproc() != 0)
3473     yield();
3474
3475   // the yield() may have caused some traps to occur,
3476   // so restore trap registers for use by kernelvec.S's sepc instruction.
3477   w_sepc(sepc);
3478   w_sstatus(sstatus);
3479 }
3480
3481 void
3482 clockintr()
3483 {
3484   if(cpuid() == 0){
3485     acquire(&tickslock);
3486     ticks++;
3487     wakeup(&ticks);
3488     release(&tickslock);
3489   }
3490
3491   // ask for the next timer interrupt. this also clears
3492   // the interrupt request. 1000000 is about a tenth
3493   // of a second.
3494   w_stimecmp(r_time() + 1000000);
3495 }
3496
3497
3498
3499
```

```
// check if it's an external interrupt or software interrupt,
// and handle it.
// returns 2 if timer interrupt,
// 1 if other device,
// 0 if not recognized.
int
devintr()
{
  uint64 scause = r_scause();

  if(scause == 0x8000000000000009L){
    // this is a supervisor external interrupt, via PLIC.

    // irq indicates which device interrupted.
    int irq = plic_claim();

    if(irq == UART0_IRQ){
      uartintr();
    } else if(irq == VIRTIO0_IRQ){
      virtio_disk_intr();
    } else if(irq){
      printf("unexpected interrupt irq=%d\n", irq);
    }

    // the PLIC allows each device to raise at most one
    // interrupt at a time; tell the PLIC the device is
    // now allowed to interrupt again.
    if(irq)
      plic_complete(irq);

    return 1;
  } else if(scause == 0x8000000000000005L){
    // timer interrupt.
    clockintr();
    return 2;
  } else {
    return 0;
  }
}
```



```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_pause  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
```

```
3600 #include "types.h"
3601 #include "param.h"
3602 #include "memlayout.h"
3603 #include "riscv.h"
3604 #include "spinlock.h"
3605 #include "proc.h"
3606 #include "syscall.h"
3607 #include "defs.h"
3608
3609 // Fetch the uint64 at addr from the current process.
3610 int
3611 fetchaddr(uint64 addr, uint64 *ip)
3612 {
3613   struct proc *p = myproc();
3614   if(addr >= p->sz || addr+sizeof(uint64) > p->sz) // both tests needed, in
3615     return -1;
3616   if(copyin(p->pagetable, (char *)ip, addr, sizeof(*ip)) != 0)
3617     return -1;
3618   return 0;
3619 }
3620
3621 // Fetch the nul-terminated string at addr from the current process.
3622 // Returns length of string, not including nul, or -1 for error.
3623 int
3624 fetchstr(uint64 addr, char *buf, int max)
3625 {
3626   struct proc *p = myproc();
3627   if(copyinstr(p->pagetable, buf, addr, max) < 0)
3628     return -1;
3629   return strlen(buf);
3630 }
3631
3632 static uint64
3633 argraw(int n)
3634 {
3635   struct proc *p = myproc();
3636   switch (n) {
3637   case 0:
3638     return p->trapframe->a0;
3639   case 1:
3640     return p->trapframe->a1;
3641   case 2:
3642     return p->trapframe->a2;
3643   case 3:
3644     return p->trapframe->a3;
3645   case 4:
3646     return p->trapframe->a4;
3647   case 5:
3648     return p->trapframe->a5;
3649   }
```



```
3650   panic("argraw");
3651   return -1;
3652 }
3653
3654 // Fetch the nth 32-bit system call argument.
3655 void
3656 argint(int n, int *ip)
3657 {
3658   *ip = argraw(n);
3659 }
3660
3661 // Retrieve an argument as a pointer.
3662 // Doesn't check for legality, since
3663 // copyin/copyout will do that.
3664 void
3665 argaddr(int n, uint64 *ip)
3666 {
3667   *ip = argraw(n);
3668 }
3669
3670 // Fetch the nth word-sized system call argument as a null-terminated string
3671 // Copies into buf, at most max.
3672 // Returns string length if OK (including nul), -1 if error.
3673 int
3674 argstr(int n, char *buf, int max)
3675 {
3676   uint64 addr;
3677   argaddr(n, &addr);
3678   return fetchstr(addr, buf, max);
3679 }
3680
3681 // Prototypes for the functions that handle system calls.
3682 extern uint64 sys_fork(void);
3683 extern uint64 sys_exit(void);
3684 extern uint64 sys_wait(void);
3685 extern uint64 sys_pipe(void);
3686 extern uint64 sys_read(void);
3687 extern uint64 sys_kill(void);
3688 extern uint64 sys_exec(void);
3689 extern uint64 sys_fstat(void);
3690 extern uint64 sys_chdir(void);
3691 extern uint64 sys_dup(void);
3692 extern uint64 sys_getpid(void);
3693 extern uint64 sys_sbrk(void);
3694 extern uint64 sys_pause(void);
3695 extern uint64 sys_uptime(void);
3696 extern uint64 sys_open(void);
3697 extern uint64 sys_write(void);
3698 extern uint64 sys_mknod(void);
3699 extern uint64 sys_unlink(void);
```

```
3700 extern uint64 sys_link(void);
3701 extern uint64 sys_mkdir(void);
3702 extern uint64 sys_close(void);
3703
3704 // An array mapping syscall numbers from syscall.h
3705 // to the function that handles the system call.
3706 static uint64 (*syscalls[])(void) = {
3707 [SYS_fork]    sys_fork,
3708 [SYS_exit]    sys_exit,
3709 [SYS_wait]    sys_wait,
3710 [SYS_pipe]    sys_pipe,
3711 [SYS_read]    sys_read,
3712 [SYS_kill]    sys_kill,
3713 [SYS_exec]    sys_exec,
3714 [SYS_fstat]   sys_fstat,
3715 [SYS_chdir]   sys_chdir,
3716 [SYS_dup]     sys_dup,
3717 [SYS_getpid]  sys_getpid,
3718 [SYS_sbrk]    sys_sbrk,
3719 [SYS_pause]   sys_pause,
3720 [SYS_uptime]  sys_uptime,
3721 [SYS_open]    sys_open,
3722 [SYS_write]   sys_write,
3723 [SYS_mknod]   sys_mknod,
3724 [SYS_unlink]  sys_unlink,
3725 [SYS_link]    sys_link,
3726 [SYS_mkdir]   sys_mkdir,
3727 [SYS_close]   sys_close,
3728 };
3729
3730 void
3731 syscall(void)
3732 {
3733   int num;
3734   struct proc *p = myproc();
3735
3736   num = p->trapframe->a7;
3737   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3738     // Use num to lookup the system call function for num, call it,
3739     // and store its return value in p->trapframe->a0
3740     p->trapframe->a0 = syscalls[num]();
3741   } else {
3742     printf("%d %s: unknown sys call %d\n",
3743             p->pid, p->name, num);
3744     p->trapframe->a0 = -1;
3745   }
3746 }
3747
3748
3749
```



```
3750 #include "types.h"
3751 #include "riscv.h"
3752 #include "defs.h"
3753 #include "param.h"
3754 #include "memlayout.h"
3755 #include "spinlock.h"
3756 #include "proc.h"
3757 #include "vm.h"
3758
3759 uint64
3760 sys_exit(void)
3761 {
3762   int n;
3763   argint(0, &n);
3764   kexit(n);
3765   return 0;  // not reached
3766 }
3767
3768 uint64
3769 sys_getpid(void)
3770 {
3771   return myproc()->pid;
3772 }
3773
3774 uint64
3775 sys_fork(void)
3776 {
3777   return kfork();
3778 }
3779
3780 uint64
3781 sys_wait(void)
3782 {
3783   uint64 p;
3784   argaddr(0, &p);
3785   return kwait(p);
3786 }
3787
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3799
```

```
3800 uint64
3801 sys_sbrk(void)
3802 {
3803   uint64 addr;
3804   int t;
3805   int n;
3806
3807   argint(0, &n);
3808   argint(1, &t);
3809   addr = myproc()->sz;
3810
3811   if(t == SBRK_EAGER || n < 0) {
3812     if(growproc(n) < 0) {
3813       return -1;
3814     }
3815   } else {
3816     // Lazily allocate memory for this process: increase its memory
3817     // size but don't allocate memory. If the processes uses the
3818     // memory, vmfault() will allocate it.
3819     if(addr + n < addr)
3820       return -1;
3821     myproc()->sz += n;
3822   }
3823   return addr;
3824 }
3825
3826 uint64
3827 sys_pause(void)
3828 {
3829   int n;
3830   uint ticks0;
3831
3832   argint(0, &n);
3833   if(n < 0)
3834     n = 0;
3835   acquire(&tickslock);
3836   ticks0 = ticks;
3837   while(ticks - ticks0 < n){
3838     if(killed(myproc())){
3839       release(&tickslock);
3840       return -1;
3841     }
3842     sleep(&ticks, &tickslock);
3843   }
3844   release(&tickslock);
3845   return 0;
3846 }
3847
3848
3849
```



```
3850 uint64
3851 sys_kill(void)
3852 {
3853   int pid;
3854
3855   argint(0, &pid);
3856   return kkill(pid);
3857 }
3858
3859 // return how many clock tick interrupts have occurred
3860 // since start.
3861 uint64
3862 sys_uptime(void)
3863 {
3864   uint xticks;
3865
3866   acquire(&tickslock);
3867   xticks = ticks;
3868   release(&tickslock);
3869   return xticks;
3870 }
3871
3872
3873
3874
3875
3876
3877
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
```

```
3900 struct buf {
3901   int valid;   // has data been read from disk?
3902   int disk;    // does disk "own" buf?
3903   uint dev;
3904   uint blockno;
3905   struct sleeplock lock;
3906   uint refcnt;
3907   struct buf *prev; // LRU cache list
3908   struct buf *next;
3909   uchar data[BSIZE];
3910 };
3911
3912
3913
3914
3915
3916
3917
3918
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
```



```
3950 // Long-term locks for processes
3951 struct sleeplock {
3952   uint locked;       // Is the lock held?
3953   struct spinlock lk; // spinlock protecting this sleep lock
3954
3955   // For debugging:
3956   char *name;        // Name of lock.
3957   int pid;           // Process holding lock
3958 };
3959
3960
3961
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
```

```
4000 #define O_RDONLY  0x000
4001 #define O_WRONLY  0x001
4002 #define O_RDWR    0x002
4003 #define O_CREATE  0x200
4004 #define O_TRUNC   0x400
4005
4006
4007
4008
4009
4010
4011
4012
4013
4014
4015
4016
4017
4018
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
```



```
4050 #define T_DIR     1   // Directory
4051 #define T_FILE    2   // File
4052 #define T_DEVICE  3   // Device
4053
4054 struct stat {
4055   int dev;     // File system's disk device
4056   uint ino;    // Inode number
4057   short type;  // Type of file
4058   short nlink; // Number of links to file
4059   uint64 size; // Size of file in bytes
4060 };
4061
4062
4063
4064
4065
4066
4067
4068
4069
4070
4071
4072
4073
4074
4075
4076
4077
4078
4079
4080
4081
4082
4083
4084
4085
4086
4087
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
```

```
4100 // On-disk file system format.
4101 // Both the kernel and user programs use this header file.
4102
4103
4104 #define ROOTINO  1   // root i-number
4105 #define BSIZE 1024  // block size
4106
4107 // Disk layout:
4108 // [ boot block | super block | log | inode blocks |
4109 //                                          free bit map | data blocks]
4110 //
4111 // mkfs computes the super block and builds an initial file system. The
4112 // super block describes the disk layout:
4113 struct superblock {
4114   uint magic;        // Must be FSMAGIC
4115   uint size;         // Size of file system image (blocks)
4116   uint nblocks;      // Number of data blocks
4117   uint ninodes;      // Number of inodes.
4118   uint nlog;         // Number of log blocks
4119   uint logstart;     // Block number of first log block
4120   uint inodestart;   // Block number of first inode block
4121   uint bmapstart;    // Block number of first free map block
4122 };
4123
4124 #define FSMAGIC 0x10203040
4125
4126 #define NDIRECT 12
4127 #define NINDIRECT (BSIZE / sizeof(uint))
4128 #define MAXFILE (NDIRECT + NINDIRECT)
4129
4130 // On-disk inode structure
4131 struct dinode {
4132   short type;           // File type
4133   short major;          // Major device number (T_DEVICE only)
4134   short minor;          // Minor device number (T_DEVICE only)
4135   short nlink;          // Number of links to inode in file system
4136   uint size;            // Size of file (bytes)
4137   uint addrs[NDIRECT+1];   // Data block addresses
4138 };
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```



```
4150 // Inodes per block.
4151 #define IPB           (BSIZE / sizeof(struct dinode))
4152
4153 // Block containing inode i
4154 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
4155
4156 // Bitmap bits per block
4157 #define BPB           (BSIZE*8)
4158
4159 // Block of free map containing bit for block b
4160 #define BBLOCK(b, sb) ((b)/BPB + sb.bmapstart)
4161
4162 // Directory is a file containing a sequence of dirent structures.
4163 #define DIRSIZ 14
4164
4165 struct dirent {
4166   ushort inum;
4167   char name[DIRSIZ];
4168 };
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
```

```
4200 struct file {
4201   enum { FD_NONE, FD_PIPE, FD_INODE, FD_DEVICE } type;
4202   int ref; // reference count
4203   char readable;
4204   char writable;
4205   struct pipe *pipe; // FD_PIPE
4206   struct inode *ip;  // FD_INODE and FD_DEVICE
4207   uint off;          // FD_INODE
4208   short major;       // FD_DEVICE
4209 };
4210
4211 #define major(dev)  ((dev) >> 16 & 0xFFFF)
4212 #define minor(dev)  ((dev) & 0xFFFF)
4213 #define    mkdev(m,n)  ((uint)((m)<<16| (n)))
4214
4215 // in-memory copy of an inode
4216 struct inode {
4217   uint dev;           // Device number
4218   uint inum;          // Inode number
4219   int ref;            // Reference count
4220   struct sleeplock lock; // protects everything below here
4221   int valid;          // inode has been read from disk?
4222
4223   short type;         // copy of disk inode
4224   short major;
4225   short minor;
4226   short nlink;
4227   uint size;
4228   uint addrs[NDIRECT+1];
4229 };
4230
4231 // map major device number to device functions.
4232 struct devsw {
4233   int (*read)(int, uint64, int);
4234   int (*write)(int, uint64, int);
4235 };
4236
4237 extern struct devsw devsw[];
4238
4239 #define CONSOLE 1
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
```



```
4250 // Buffer cache.
4251 //
4252 // The buffer cache is a linked list of buf structures holding
4253 // cached copies of disk block contents.  Caching disk blocks
4254 // in memory reduces the number of disk reads and also provides
4255 // a synchronization point for disk blocks used by multiple processes.
4256 //
4257 // Interface:
4258 // * To get a buffer for a particular disk block, call bread.
4259 // * After changing buffer data, call bwrite to write it to disk.
4260 // * When done with the buffer, call brelse.
4261 // * Do not use the buffer after calling brelse.
4262 // * Only one process at a time can use a buffer,
4263 //     so do not keep them longer than necessary.
4264
4265
4266 #include "types.h"
4267 #include "param.h"
4268 #include "spinlock.h"
4269 #include "sleeplock.h"
4270 #include "riscv.h"
4271 #include "defs.h"
4272 #include "fs.h"
4273 #include "buf.h"
4274
4275 struct {
4276   struct spinlock lock;
4277   struct buf buf[NBUF];
4278
4279   // Linked list of all buffers, through prev/next.
4280   // Sorted by how recently the buffer was used.
4281   // head.next is most recent, head.prev is least.
4282   struct buf head;
4283 } bcache;
4284
4285 void
4286 binit(void)
4287 {
4288   struct buf *b;
4289
4290   initlock(&bcache.lock, "bcache");
4291
4292   // Create linked list of buffers
4293   bcache.head.prev = &bcache.head;
4294   bcache.head.next = &bcache.head;
4295   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
4296     b->next = bcache.head.next;
4297     b->prev = &bcache.head;
4298     initsleeplock(&b->lock, "buffer");
4299     bcache.head.next->prev = b;
```

```
4300     bcache.head.next = b;
4301   }
4302 }
4303
4304 // Look through buffer cache for block on device dev.
4305 // If not found, allocate a buffer.
4306 // In either case, return locked buffer.
4307 static struct buf*
4308 bget(uint dev, uint blockno)
4309 {
4310   struct buf *b;
4311
4312   acquire(&bcache.lock);
4313
4314   // Is the block already cached?
4315   for(b = bcache.head.next; b != &bcache.head; b = b->next){
4316     if(b->dev == dev && b->blockno == blockno){
4317       b->refcnt++;
4318       release(&bcache.lock);
4319       acquiresleep(&b->lock);
4320       return b;
4321     }
4322   }
4323
4324   // Not cached.
4325   // Recycle the least recently used (LRU) unused buffer.
4326   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
4327     if(b->refcnt == 0) {
4328       b->dev = dev;
4329       b->blockno = blockno;
4330       b->valid = 0;
4331       b->refcnt = 1;
4332       release(&bcache.lock);
4333       acquiresleep(&b->lock);
4334       return b;
4335     }
4336   }
4337   panic("bget: no buffers");
4338 }
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
```



```
4350 // Return a locked buf with the contents of the indicated block.
4351 struct buf*
4352 bread(uint dev, uint blockno)
4353 {
4354   struct buf *b;
4355
4356   b = bget(dev, blockno);
4357   if(!b->valid) {
4358     virtio_disk_rw(b, 0);
4359     b->valid = 1;
4360   }
4361   return b;
4362 }
4363
4364 // Write b's contents to disk.  Must be locked.
4365 void
4366 bwrite(struct buf *b)
4367 {
4368   if(!holdingsleep(&b->lock))
4369     panic("bwrite");
4370   virtio_disk_rw(b, 1);
4371 }
4372
4373 // Release a locked buffer.
4374 // Move to the head of the most-recently-used list.
4375 void
4376 brelse(struct buf *b)
4377 {
4378   if(!holdingsleep(&b->lock))
4379     panic("brelse");
4380
4381   releasesleep(&b->lock);
4382
4383   acquire(&bcache.lock);
4384   b->refcnt--;
4385   if (b->refcnt == 0) {
4386     // no one is waiting for it.
4387     b->next->prev = b->prev;
4388     b->prev->next = b->next;
4389     b->next = bcache.head.next;
4390     b->prev = &bcache.head;
4391     bcache.head.next->prev = b;
4392     bcache.head.next = b;
4393   }
4394
4395   release(&bcache.lock);
4396 }
4397
4398
4399
```

```
4400 void
4401 bpin(struct buf *b) {
4402   acquire(&bcache.lock);
4403   b->refcnt++;
4404   release(&bcache.lock);
4405 }
4406
4407 void
4408 bunpin(struct buf *b) {
4409   acquire(&bcache.lock);
4410   b->refcnt--;
4411   release(&bcache.lock);
4412 }
4413
4414
4415
4416
4417
4418
4419
4420
4421
4422
4423
4424
4425
4426
4427
4428
4429
4430
4431
4432
4433
4434
4435
4436
4437
4438
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
```



```
4450 // Sleeping locks
4451
4452 #include "types.h"
4453 #include "riscv.h"
4454 #include "defs.h"
4455 #include "param.h"
4456 #include "memlayout.h"
4457 #include "spinlock.h"
4458 #include "proc.h"
4459 #include "sleeplock.h"
4460
4461 void
4462 initsleeplock(struct sleeplock *lk, char *name)
4463 {
4464   initlock(&lk->lk, "sleep lock");
4465   lk->name = name;
4466   lk->locked = 0;
4467   lk->pid = 0;
4468 }
4469
4470 void
4471 acquiresleep(struct sleeplock *lk)
4472 {
4473   acquire(&lk->lk);
4474   while (lk->locked) {
4475     sleep(lk, &lk->lk);
4476   }
4477   lk->locked = 1;
4478   lk->pid = myproc()->pid;
4479   release(&lk->lk);
4480 }
4481
4482 void
4483 releasesleep(struct sleeplock *lk)
4484 {
4485   acquire(&lk->lk);
4486   lk->locked = 0;
4487   lk->pid = 0;
4488   wakeup(lk);
4489   release(&lk->lk);
4490 }
4491
4492
4493
4494
4495
4496
4497
4498
4499
```

```
4500 int
4501 holdingsleep(struct sleeplock *lk)
4502 {
4503   int r;
4504
4505   acquire(&lk->lk);
4506   r = lk->locked && (lk->pid == myproc()->pid);
4507   release(&lk->lk);
4508   return r;
4509 }
4510
4511
4512
4513
4514
4515
4516
4517
4518
4519
4520
4521
4522
4523
4524
4525
4526
4527
4528
4529
4530
4531
4532
4533
4534
4535
4536
4537
4538
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4549
```



```
4550 #include "types.h"
4551 #include "riscv.h"
4552 #include "defs.h"
4553 #include "param.h"
4554 #include "spinlock.h"
4555 #include "sleeplock.h"
4556 #include "fs.h"
4557 #include "buf.h"
4558
4559 // Simple logging that allows concurrent FS system calls.
4560 //
4561 // A log transaction contains the updates of multiple FS system
4562 // calls. The logging system only commits when there are
4563 // no FS system calls active. Thus there is never
4564 // any reasoning required about whether a commit might
4565 // write an uncommitted system call's updates to disk.
4566 //
4567 // A system call should call begin_op()/end_op() to mark
4568 // its start and end. Usually begin_op() just increments
4569 // the count of in-progress FS system calls and returns.
4570 // But if it thinks the log is close to running out, it
4571 // sleeps until the last outstanding end_op() commits.
4572 //
4573 // The log is a physical re-do log containing disk blocks.
4574 // The on-disk log format:
4575 //   header block, containing block #s for block A, B, C, ...
4576 //   block A
4577 //   block B
4578 //   block C
4579 //   ...
4580 // Log appends are synchronous.
4581
4582 // Contents of the header block, used for both the on-disk header block
4583 // and to keep track in memory of logged block# before commit.
4584 struct logheader {
4585   int n;
4586   int block[LOGBLOCKS];
4587 };
4588
4589 struct log {
4590   struct spinlock lock;
4591   int start;
4592   int outstanding; // how many FS sys calls are executing.
4593   int committing;  // in commit(), please wait.
4594   int dev;
4595   struct logheader lh;
4596 };
4597
4598
4599
```

```
4600 struct log log;
4601
4602 static void recover_from_log(void);
4603 static void commit();
4604
4605 void
4606 initlog(int dev, struct superblock *sb)
4607 {
4608   if (sizeof(struct logheader) >= BSIZE)
4609     panic("initlog: too big logheader");
4610
4611   initlock(&log.lock, "log");
4612   log.start = sb->logstart;
4613   log.dev = dev;
4614   recover_from_log();
4615 }
4616
4617 // Copy committed blocks from log to their home location
4618 static void
4619 install_trans(int recovering)
4620 {
4621   int tail;
4622
4623   for (tail = 0; tail < log.lh.n; tail++) {
4624     if(recovering) {
4625       printf("recovering tail %d dst %d\n", tail, log.lh.block[tail]);
4626     }
4627     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
4628     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
4629     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
4630     bwrite(dbuf);  // write dst to disk
4631     if(recovering == 0)
4632       bunpin(dbuf);
4633     brelse(lbuf);
4634     brelse(dbuf);
4635   }
4636 }
4637
4638
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649
```



```
4650 // Read the log header from disk into the in-memory log header
4651 static void
4652 read_head(void)
4653 {
4654   struct buf *buf = bread(log.dev, log.start);
4655   struct logheader *lh = (struct logheader *) (buf->data);
4656   int i;
4657   log.lh.n = lh->n;
4658   for (i = 0; i < log.lh.n; i++) {
4659     log.lh.block[i] = lh->block[i];
4660   }
4661   brelse(buf);
4662 }
4663
4664 // Write in-memory log header to disk.
4665 // This is the true point at which the
4666 // current transaction commits.
4667 static void
4668 write_head(void)
4669 {
4670   struct buf *buf = bread(log.dev, log.start);
4671   struct logheader *hb = (struct logheader *) (buf->data);
4672   int i;
4673   hb->n = log.lh.n;
4674   for (i = 0; i < log.lh.n; i++) {
4675     hb->block[i] = log.lh.block[i];
4676   }
4677   bwrite(buf);
4678   brelse(buf);
4679 }
4680
4681 static void
4682 recover_from_log(void)
4683 {
4684   read_head();
4685   install_trans(1); // if committed, copy from log to disk
4686   log.lh.n = 0;
4687   write_head(); // clear the log
4688 }
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4699
```

```
4700 // called at the start of each FS system call.
4701 void
4702 begin_op(void)
4703 {
4704   acquire(&log.lock);
4705   while(1){
4706     if(log.committing){
4707       sleep(&log, &log.lock);
4708     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGBLOCKS){
4709       // this op might exhaust log space; wait for commit.
4710       sleep(&log, &log.lock);
4711     } else {
4712       log.outstanding += 1;
4713       release(&log.lock);
4714       break;
4715     }
4716   }
4717 }
4718
4719 // called at the end of each FS system call.
4720 // commits if this was the last outstanding operation.
4721 void
4722 end_op(void)
4723 {
4724   int do_commit = 0;
4725
4726   acquire(&log.lock);
4727   log.outstanding -= 1;
4728   if(log.committing)
4729     panic("log.committing");
4730   if(log.outstanding == 0){
4731     do_commit = 1;
4732     log.committing = 1;
4733   } else {
4734     // begin_op() may be waiting for log space,
4735     // and decrementing log.outstanding has decreased
4736     // the amount of reserved space.
4737     wakeup(&log);
4738   }
4739   release(&log.lock);
4740
4741   if(do_commit){
4742     // call commit w/o holding locks, since not allowed
4743     // to sleep with locks.
4744     commit();
4745     acquire(&log.lock);
4746     log.committing = 0;
4747     wakeup(&log);
4748     release(&log.lock);
4749   }
```



```
4750 }
4751
4752 // Copy modified blocks from cache to log.
4753 static void
4754 write_log(void)
4755 {
4756   int tail;
4757
4758   for (tail = 0; tail < log.lh.n; tail++) {
4759     struct buf *to = bread(log.dev, log.start+tail+1); // log block
4760     struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
4761     memmove(to->data, from->data, BSIZE);
4762     bwrite(to);  // write the log
4763     brelse(from);
4764     brelse(to);
4765   }
4766 }
4767
4768 static void
4769 commit()
4770 {
4771   if (log.lh.n > 0) {
4772     write_log();     // Write modified blocks from cache to log
4773     write_head();    // Write header to disk -- the real commit
4774     install_trans(0); // Now install writes to home locations
4775     log.lh.n = 0;
4776     write_head();    // Erase the transaction from the log
4777   }
4778 }
4779
4780 // Caller has modified b->data and is done with the buffer.
4781 // Record the block number and pin in the cache by increasing refcnt.
4782 // commit()/write_log() will do the disk write.
4783 //
4784 // log_write() replaces bwrite(); a typical use is:
4785 //   bp = bread(...)
4786 //   modify bp->data[]
4787 //   log_write(bp)
4788 //   brelse(bp)
4789 void
4790 log_write(struct buf *b)
4791 {
4792   int i;
4793
4794   acquire(&log.lock);
4795   if (log.lh.n >= LOGBLOCKS)
4796     panic("too big a transaction");
4797   if (log.outstanding < 1)
4798     panic("log_write outside of trans");
4799
```

```
4800   for (i = 0; i < log.lh.n; i++) {
4801     if (log.lh.block[i] == b->blockno)   // log absorption
4802       break;
4803   }
4804   log.lh.block[i] = b->blockno;
4805   if (i == log.lh.n) {  // Add new block to log?
4806     bpin(b);
4807     log.lh.n++;
4808   }
4809   release(&log.lock);
4810 }
4811
4812
4813
4814
4815
4816
4817
4818
4819
4820
4821
4822
4823
4824
4825
4826
4827
4828
4829
4830
4831
4832
4833
4834
4835
4836
4837
4838
4839
4840
4841
4842
4843
4844
4845
4846
4847
4848
4849
```



```
4850 // File system implementation.  Five layers:
4851 //   + Blocks: allocator for raw disk blocks.
4852 //   + Log: crash recovery for multi-step updates.
4853 //   + Files: inode allocator, reading, writing, metadata.
4854 //   + Directories: inode with special contents (list of other inodes!)
4855 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
4856 //
4857 // This file contains the low-level file system manipulation
4858 // routines.  The (higher-level) system call implementations
4859 // are in sysfile.c.
4860
4861 #include "types.h"
4862 #include "riscv.h"
4863 #include "defs.h"
4864 #include "param.h"
4865 #include "stat.h"
4866 #include "spinlock.h"
4867 #include "proc.h"
4868 #include "sleeplock.h"
4869 #include "fs.h"
4870 #include "buf.h"
4871 #include "file.h"
4872
4873 #define min(a, b) ((a) < (b) ? (a) : (b))
4874 // there should be one superblock per disk device, but we run with
4875 // only one device
4876 struct superblock sb;
4877
4878 // Read the super block.
4879 static void
4880 readsb(int dev, struct superblock *sb)
4881 {
4882   struct buf *bp;
4883
4884   bp = bread(dev, 1);
4885   memmove(sb, bp->data, sizeof(*sb));
4886   brelse(bp);
4887 }
4888
4889 // Init fs
4890 void
4891 fsinit(int dev) {
4892   readsb(dev, &sb);
4893   if(sb.magic != FSMAGIC)
4894     panic("invalid file system");
4895   initlog(dev, &sb);
4896   ireclaim(dev);
4897 }
4898
4899
```

```
4900 // Zero a block.
4901 static void
4902 bzero(int dev, int bno)
4903 {
4904   struct buf *bp;
4905
4906   bp = bread(dev, bno);
4907   memset(bp->data, 0, BSIZE);
4908   log_write(bp);
4909   brelse(bp);
4910 }
4911
4912 // Blocks.
4913
4914 // Allocate a zeroed disk block.
4915 // returns 0 if out of disk space.
4916 static uint
4917 balloc(uint dev)
4918 {
4919   int b, bi, m;
4920   struct buf *bp;
4921
4922   bp = 0;
4923   for(b = 0; b < sb.size; b += BPB){
4924     bp = bread(dev, BBLOCK(b, sb));
4925     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
4926       m = 1 << (bi % 8);
4927       if((bp->data[bi/8] & m) == 0){  // Is block free?
4928         bp->data[bi/8] |= m;  // Mark block in use.
4929         log_write(bp);
4930         brelse(bp);
4931         bzero(dev, b + bi);
4932         return b + bi;
4933       }
4934     }
4935     brelse(bp);
4936   }
4937   printf("balloc: out of blocks\n");
4938   return 0;
4939 }
4940
4941
4942
4943
4944
4945
4946
4947
4948
4949
```



```
4950 // Free a disk block.
4951 static void
4952 bfree(int dev, uint b)
4953 {
4954   struct buf *bp;
4955   int bi, m;
4956
4957   bp = bread(dev, BBLOCK(b, sb));
4958   bi = b % BPB;
4959   m = 1 << (bi % 8);
4960   if((bp->data[bi/8] & m) == 0)
4961     panic("freeing free block");
4962   bp->data[bi/8] &= ~m;
4963   log_write(bp);
4964   brelse(bp);
4965 }
4966
4967 // Inodes.
4968 //
4969 // An inode describes a single unnamed file.
4970 // The inode disk structure holds metadata: the file's type,
4971 // its size, the number of links referring to it, and the
4972 // list of blocks holding the file's content.
4973 //
4974 // The inodes are laid out sequentially on disk at block
4975 // sb.inodestart. Each inode has a number, indicating its
4976 // position on the disk.
4977 //
4978 // The kernel keeps a table of in-use inodes in memory
4979 // to provide a place for synchronizing access
4980 // to inodes used by multiple processes. The in-memory
4981 // inodes include book-keeping information that is
4982 // not stored on disk: ip->ref and ip->valid.
4983 //
4984 // An inode and its in-memory representation go through a
4985 // sequence of states before they can be used by the
4986 // rest of the file system code.
4987 //
4988 // * Allocation: an inode is allocated if its type (on disk)
4989 //   is non-zero. ialloc() allocates, and iput() frees if
4990 //   the reference and link counts have fallen to zero.
4991 //
4992 // * Referencing in table: an entry in the inode table
4993 //   is free if ip->ref is zero. Otherwise ip->ref tracks
4994 //   the number of in-memory pointers to the entry (open
4995 //   files and current directories). iget() finds or
4996 //   creates a table entry and increments its ref; iput()
4997 //   decrements ref.
4998 //
4999 // * Valid: the information (type, size, &c) in an inode
```

```
5000 //   table entry is only correct when ip->valid is 1.
5001 //   ilock() reads the inode from
5002 //   the disk and sets ip->valid, while iput() clears
5003 //   ip->valid if ip->ref has fallen to zero.
5004 //
5005 // * Locked: file system code may only examine and modify
5006 //   the information in an inode and its content if it
5007 //   has first locked the inode.
5008 //
5009 // Thus a typical sequence is:
5010 //   ip = iget(dev, inum)
5011 //   ilock(ip)
5012 //   ... examine and modify ip->xxx ...
5013 //   iunlock(ip)
5014 //   iput(ip)
5015 //
5016 // ilock() is separate from iget() so that system calls can
5017 // get a long-term reference to an inode (as for an open file)
5018 // and only lock it for short periods (e.g., in read()).
5019 // The separation also helps avoid deadlock and races during
5020 // pathname lookup. iget() increments ip->ref so that the inode
5021 // stays in the table and pointers to it remain valid.
5022 //
5023 // Many internal file system functions expect the caller to
5024 // have locked the inodes involved; this lets callers create
5025 // multi-step atomic operations.
5026 //
5027 // The itable.lock spin-lock protects the allocation of itable
5028 // entries. Since ip->ref indicates whether an entry is free,
5029 // and ip->dev and ip->inum indicate which i-node an entry
5030 // holds, one must hold itable.lock while using any of those fields.
5031 //
5032 // An ip->lock sleep-lock protects all ip-> fields other than ref,
5033 // dev, and inum.  One must hold ip->lock in order to
5034 // read or write that inode's ip->valid, ip->size, ip->type, &c.
5035
5036 struct {
5037   struct spinlock lock;
5038   struct inode inode[NINODE];
5039 } itable;
5040
5041 void
5042 iinit()
5043 {
5044   int i = 0;
5045
5046   initlock(&itable.lock, "itable");
5047   for(i = 0; i < NINODE; i++) {
5048     initsleeplock(&itable.inode[i].lock, "inode");
5049   }
```



```
5050 }
5051
5052 static struct inode* iget(uint dev, uint inum);
5053
5054 // Allocate an inode on device dev.
5055 // Mark it as allocated by  giving it type type.
5056 // Returns an unlocked but allocated and referenced inode,
5057 // or NULL if there is no free inode.
5058 struct inode*
5059 ialloc(uint dev, short type)
5060 {
5061   int inum;
5062   struct buf *bp;
5063   struct dinode *dip;
5064
5065   for(inum = 1; inum < sb.ninodes; inum++){
5066     bp = bread(dev, IBLOCK(inum, sb));
5067     dip = (struct dinode*)bp->data + inum%IPB;
5068     if(dip->type == 0){  // a free inode
5069       memset(dip, 0, sizeof(*dip));
5070       dip->type = type;
5071       log_write(bp);   // mark it allocated on the disk
5072       brelse(bp);
5073       return iget(dev, inum);
5074     }
5075     brelse(bp);
5076   }
5077   printf("ialloc: no inodes\n");
5078   return 0;
5079 }
5080
5081 // Copy a modified in-memory inode to disk.
5082 // Must be called after every change to an ip->xxx field
5083 // that lives on disk.
5084 // Caller must hold ip->lock.
5085 void
5086 iupdate(struct inode *ip)
5087 {
5088   struct buf *bp;
5089   struct dinode *dip;
5090
5091   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5092   dip = (struct dinode*)bp->data + ip->inum%IPB;
5093   dip->type = ip->type;
5094   dip->major = ip->major;
5095   dip->minor = ip->minor;
5096   dip->nlink = ip->nlink;
5097   dip->size = ip->size;
5098   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
5099   log_write(bp);
```

```
5100   brelse(bp);
5101 }
5102
5103 // Find the inode with number inum on device dev
5104 // and return the in-memory copy. Does not lock
5105 // the inode and does not read it from disk.
5106 static struct inode*
5107 iget(uint dev, uint inum)
5108 {
5109   struct inode *ip, *empty;
5110
5111   acquire(&itable.lock);
5112
5113   // Is the inode already in the table?
5114   empty = 0;
5115   for(ip = &itable.inode[0]; ip < &itable.inode[NINODE]; ip++){
5116     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
5117       ip->ref++;
5118       release(&itable.lock);
5119       return ip;
5120     }
5121     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
5122       empty = ip;
5123   }
5124
5125   // Recycle an inode entry.
5126   if(empty == 0)
5127     panic("iget: no inodes");
5128
5129   ip = empty;
5130   ip->dev = dev;
5131   ip->inum = inum;
5132   ip->ref = 1;
5133   ip->valid = 0;
5134   release(&itable.lock);
5135
5136   return ip;
5137 }
5138
5139 // Increment reference count for ip.
5140 // Returns ip to enable ip = idup(ip1) idiom.
5141 struct inode*
5142 idup(struct inode *ip)
5143 {
5144   acquire(&itable.lock);
5145   ip->ref++;
5146   release(&itable.lock);
5147   return ip;
5148 }
5149
```



```
5150 // Lock the given inode.
5151 // Reads the inode from disk if necessary.
5152 void
5153 ilock(struct inode *ip)
5154 {
5155   struct buf *bp;
5156   struct dinode *dip;
5157
5158   if(ip == 0 || ip->ref < 1)
5159     panic("ilock");
5160
5161   acquiresleep(&ip->lock);
5162
5163   if(ip->valid == 0){
5164     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
5165     dip = (struct dinode*)bp->data + ip->inum%IPB;
5166     ip->type = dip->type;
5167     ip->major = dip->major;
5168     ip->minor = dip->minor;
5169     ip->nlink = dip->nlink;
5170     ip->size = dip->size;
5171     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
5172     brelse(bp);
5173     ip->valid = 1;
5174     if(ip->type == 0)
5175       panic("ilock: no type");
5176   }
5177 }
5178
5179 // Unlock the given inode.
5180 void
5181 iunlock(struct inode *ip)
5182 {
5183   if(ip == 0 || !holdingsleep(&ip->lock) || ip->ref < 1)
5184     panic("iunlock");
5185
5186   releasesleep(&ip->lock);
5187 }
5188
5189
5190
5191
5192
5193
5194
5195
5196
5197
5198
5199
```

```
5200 // Drop a reference to an in-memory inode.
5201 // If that was the last reference, the inode table entry can
5202 // be recycled.
5203 // If that was the last reference and the inode has no links
5204 // to it, free the inode (and its content) on disk.
5205 // All calls to iput() must be inside a transaction in
5206 // case it has to free the inode.
5207 void
5208 iput(struct inode *ip)
5209 {
5210   acquire(&itable.lock);
5211
5212   if(ip->ref == 1 && ip->valid && ip->nlink == 0){
5213     // inode has no links and no other references: truncate and free.
5214
5215     // ip->ref == 1 means no other process can have ip locked,
5216     // so this acquiresleep() won't block (or deadlock).
5217     acquiresleep(&ip->lock);
5218
5219     release(&itable.lock);
5220
5221     itrunc(ip);
5222     ip->type = 0;
5223     iupdate(ip);
5224     ip->valid = 0;
5225
5226     releasesleep(&ip->lock);
5227
5228     acquire(&itable.lock);
5229   }
5230
5231   ip->ref--;
5232   release(&itable.lock);
5233 }
5234
5235 // Common idiom: unlock, then put.
5236 void
5237 iunlockput(struct inode *ip)
5238 {
5239   iunlock(ip);
5240   iput(ip);
5241 }
5242
5243
5244
5245
5246
5247
5248
5249
```



```
5250 void
5251 ireclaim(int dev)
5252 {
5253   for (int inum = 1; inum < sb.ninodes; inum++) {
5254     struct inode *ip = 0;
5255     struct buf *bp = bread(dev, IBLOCK(inum, sb));
5256     struct dinode *dip = (struct dinode *)bp->data + inum % IPB;
5257     if (dip->type != 0 && dip->nlink == 0) {  // is an orphaned inode
5258       printf("ireclaim: orphaned inode %d\n", inum);
5259       ip = iget(dev, inum);
5260     }
5261     brelse(bp);
5262     if (ip) {
5263       begin_op();
5264       ilock(ip);
5265       iunlock(ip);
5266       iput(ip);
5267       end_op();
5268     }
5269   }
5270 }
5271
5272 // Inode content
5273 //
5274 // The content (data) associated with each inode is stored
5275 // in blocks on the disk. The first NDIRECT block numbers
5276 // are listed in ip->addrs[].  The next NINDIRECT blocks are
5277 // listed in block ip->addrs[NDIRECT].
5278
5279 // Return the disk block address of the nth block in inode ip.
5280 // If there is no such block, bmap allocates one.
5281 // returns 0 if out of disk space.
5282 static uint
5283 bmap(struct inode *ip, uint bn)
5284 {
5285   uint addr, *a;
5286   struct buf *bp;
5287
5288   if(bn < NDIRECT){
5289     if((addr = ip->addrs[bn]) == 0){
5290       addr = balloc(ip->dev);
5291       if(addr == 0)
5292         return 0;
5293       ip->addrs[bn] = addr;
5294     }
5295     return addr;
5296   }
5297   bn -= NDIRECT;
5298
5299
```

```
5300   if(bn < NINDIRECT){
5301     // Load indirect block, allocating if necessary.
5302     if((addr = ip->addrs[NDIRECT]) == 0){
5303       addr = balloc(ip->dev);
5304       if(addr == 0)
5305         return 0;
5306       ip->addrs[NDIRECT] = addr;
5307     }
5308     bp = bread(ip->dev, addr);
5309     a = (uint*)bp->data;
5310     if((addr = a[bn]) == 0){
5311       addr = balloc(ip->dev);
5312       if(addr){
5313         a[bn] = addr;
5314         log_write(bp);
5315       }
5316     }
5317     brelse(bp);
5318     return addr;
5319   }
5320
5321   panic("bmap: out of range");
5322 }
5323
5324 // Truncate inode (discard contents).
5325 // Caller must hold ip->lock.
5326 void
5327 itrunc(struct inode *ip)
5328 {
5329   int i, j;
5330   struct buf *bp;
5331   uint *a;
5332
5333   for(i = 0; i < NDIRECT; i++){
5334     if(ip->addrs[i]){
5335       bfree(ip->dev, ip->addrs[i]);
5336       ip->addrs[i] = 0;
5337     }
5338   }
5339
5340   if(ip->addrs[NDIRECT]){
5341     bp = bread(ip->dev, ip->addrs[NDIRECT]);
5342     a = (uint*)bp->data;
5343     for(j = 0; j < NINDIRECT; j++){
5344       if(a[j])
5345         bfree(ip->dev, a[j]);
5346     }
5347     brelse(bp);
5348     bfree(ip->dev, ip->addrs[NDIRECT]);
5349     ip->addrs[NDIRECT] = 0;
```



```
5350   }
5351
5352   ip->size = 0;
5353   iupdate(ip);
5354 }
5355
5356 // Copy stat information from inode.
5357 // Caller must hold ip->lock.
5358 void
5359 stati(struct inode *ip, struct stat *st)
5360 {
5361   st->dev = ip->dev;
5362   st->ino = ip->inum;
5363   st->type = ip->type;
5364   st->nlink = ip->nlink;
5365   st->size = ip->size;
5366 }
5367
5368 // Read data from inode.
5369 // Caller must hold ip->lock.
5370 // If user_dst==1, then dst is a user virtual address;
5371 // otherwise, dst is a kernel address.
5372 int
5373 readi(struct inode *ip, int user_dst, uint64 dst, uint off, uint n)
5374 {
5375   uint tot, m;
5376   struct buf *bp;
5377
5378   if(off > ip->size || off + n < off)
5379     return 0;
5380   if(off + n > ip->size)
5381     n = ip->size - off;
5382
5383   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
5384     uint addr = bmap(ip, off/BSIZE);
5385     if(addr == 0)
5386       break;
5387     bp = bread(ip->dev, addr);
5388     m = min(n - tot, BSIZE - off%BSIZE);
5389     if(either_copyout(user_dst, dst, bp->data + (off % BSIZE), m) == -1) {
5390       brelse(bp);
5391       tot = -1;
5392       break;
5393     }
5394     brelse(bp);
5395   }
5396   return tot;
5397 }
5398
5399
```

```
5400 // Write data to inode.
5401 // Caller must hold ip->lock.
5402 // If user_src==1, then src is a user virtual address;
5403 // otherwise, src is a kernel address.
5404 // Returns the number of bytes successfully written.
5405 // If the return value is less than the requested n,
5406 // there was an error of some kind.
5407 int
5408 writei(struct inode *ip, int user_src, uint64 src, uint off, uint n)
5409 {
5410   uint tot, m;
5411   struct buf *bp;
5412
5413   if(off > ip->size || off + n < off)
5414     return -1;
5415   if(off + n > MAXFILE*BSIZE)
5416     return -1;
5417
5418   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
5419     uint addr = bmap(ip, off/BSIZE);
5420     if(addr == 0)
5421       break;
5422     bp = bread(ip->dev, addr);
5423     m = min(n - tot, BSIZE - off%BSIZE);
5424     if(either_copyin(bp->data + (off % BSIZE), user_src, src, m) == -1) {
5425       brelse(bp);
5426       break;
5427     }
5428     log_write(bp);
5429     brelse(bp);
5430   }
5431
5432   if(off > ip->size)
5433     ip->size = off;
5434
5435   // write the i-node back to disk even if the size didn't change
5436   // because the loop above might have called bmap() and added a new
5437   // block to ip->addrs[].
5438   iupdate(ip);
5439
5440   return tot;
5441 }
5442
5443 // Directories
5444
5445 int
5446 namecmp(const char *s, const char *t)
5447 {
5448   return strncmp(s, t, DIRSIZ);
5449 }
```



```
5450 // Look for a directory entry in a directory.
5451 // If found, set *poff to byte offset of entry.
5452 struct inode*
5453 dirlookup(struct inode *dp, char *name, uint *poff)
5454 {
5455   uint off, inum;
5456   struct dirent de;
5457
5458   if(dp->type != T_DIR)
5459     panic("dirlookup not DIR");
5460
5461   for(off = 0; off < dp->size; off += sizeof(de)){
5462     if(readi(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
5463       panic("dirlookup read");
5464     if(de.inum == 0)
5465       continue;
5466     if(namecmp(name, de.name) == 0){
5467       // entry matches path element
5468       if(poff)
5469         *poff = off;
5470       inum = de.inum;
5471       return iget(dp->dev, inum);
5472     }
5473   }
5474
5475   return 0;
5476 }
5477
5478 // Write a new directory entry (name, inum) into the directory dp.
5479 // Returns 0 on success, -1 on failure (e.g. out of disk blocks).
5480 int
5481 dirlink(struct inode *dp, char *name, uint inum)
5482 {
5483   int off;
5484   struct dirent de;
5485   struct inode *ip;
5486
5487   // Check that name is not present.
5488   if((ip = dirlookup(dp, name, 0)) != 0){
5489     iput(ip);
5490     return -1;
5491   }
5492
5493   // Look for an empty dirent.
5494   for(off = 0; off < dp->size; off += sizeof(de)){
5495     if(readi(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
5496       panic("dirlink read");
5497     if(de.inum == 0)
5498       break;
5499   }
```

```
5500   strncpy(de.name, name, DIRSIZ);
5501   de.inum = inum;
5502   if(writei(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
5503     return -1;
5504
5505   return 0;
5506 }
5507
5508 // Paths
5509
5510 // Copy the next path element from path into name.
5511 // Return a pointer to the element following the copied one.
5512 // The returned path has no leading slashes,
5513 // so the caller can check *path=='\0' to see if the name is the last one.
5514 // If no name to remove, return 0.
5515 //
5516 // Examples:
5517 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5518 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5519 //   skipelem("a", name) = "", setting name = "a"
5520 //   skipelem("", name) = skipelem("////", name) = 0
5521 //
5522 static char*
5523 skipelem(char *path, char *name)
5524 {
5525   char *s;
5526   int len;
5527
5528   while(*path == '/')
5529     path++;
5530   if(*path == 0)
5531     return 0;
5532   s = path;
5533   while(*path != '/' && *path != 0)
5534     path++;
5535   len = path - s;
5536   if(len >= DIRSIZ)
5537     memmove(name, s, DIRSIZ);
5538   else {
5539     memmove(name, s, len);
5540     name[len] = 0;
5541   }
5542   while(*path == '/')
5543     path++;
5544   return path;
5545 }
5546
5547
5548
5549
```



```
5550 // Look up and return the inode for a path name.
5551 // If parent != 0, return the inode for the parent and copy the final
5552 // path element into name, which must have room for DIRSIZ bytes.
5553 // Must be called inside a transaction since it calls iput().
5554 static struct inode*
5555 namex(char *path, int nameiparent, char *name)
5556 {
5557   struct inode *ip, *next;
5558
5559   if(*path == '/')
5560     ip = iget(ROOTDEV, ROOTINO);
5561   else
5562     ip = idup(myproc()->cwd);
5563
5564   while((path = skipelem(path, name)) != 0){
5565     ilock(ip);
5566     if(ip->type != T_DIR){
5567       iunlockput(ip);
5568       return 0;
5569     }
5570     if(nameiparent && *path == '\0'){
5571       // Stop one level early.
5572       iunlock(ip);
5573       return ip;
5574     }
5575     if((next = dirlookup(ip, name, 0)) == 0){
5576       iunlockput(ip);
5577       return 0;
5578     }
5579     iunlockput(ip);
5580     ip = next;
5581   }
5582   if(nameiparent){
5583     iput(ip);
5584     return 0;
5585   }
5586   return ip;
5587 }
5588
5589 struct inode*
5590 namei(char *path)
5591 {
5592   char name[DIRSIZ];
5593   return namex(path, 0, name);
5594 }
5595
5596
5597
5598
5599
```

```
5600 struct inode*
5601 nameiparent(char *path, char *name)
5602 {
5603   return namex(path, 1, name);
5604 }
5605
5606
5607
5608
5609
5610
5611
5612
5613
5614
5615
5616
5617
5618
5619
5620
5621
5622
5623
5624
5625
5626
5627
5628
5629
5630
5631
5632
5633
5634
5635
5636
5637
5638
5639
5640
5641
5642
5643
5644
5645
5646
5647
5648
5649
```



```
5650 //
5651 // Support functions for system calls that involve file descriptors.
5652 //
5653
5654 #include "types.h"
5655 #include "riscv.h"
5656 #include "defs.h"
5657 #include "param.h"
5658 #include "fs.h"
5659 #include "spinlock.h"
5660 #include "sleeplock.h"
5661 #include "file.h"
5662 #include "stat.h"
5663 #include "proc.h"
5664
5665 struct devsw devsw[NDEV];
5666 struct {
5667   struct spinlock lock;
5668   struct file file[NFILE];
5669 } ftable;
5670
5671 void
5672 fileinit(void)
5673 {
5674   initlock(&ftable.lock, "ftable");
5675 }
5676
5677 // Allocate a file structure.
5678 struct file*
5679 filealloc(void)
5680 {
5681   struct file *f;
5682
5683   acquire(&ftable.lock);
5684   for(f = ftable.file; f < ftable.file + NFILE; f++){
5685     if(f->ref == 0){
5686       f->ref = 1;
5687       release(&ftable.lock);
5688       return f;
5689     }
5690   }
5691   release(&ftable.lock);
5692   return 0;
5693 }
5694
5695
5696
5697
5698
5699
```

```
5700 // Increment ref count for file f.
5701 struct file*
5702 filedup(struct file *f)
5703 {
5704   acquire(&ftable.lock);
5705   if(f->ref < 1)
5706     panic("filedup");
5707   f->ref++;
5708   release(&ftable.lock);
5709   return f;
5710 }
5711
5712 // Close file f.  (Decrement ref count, close when reaches 0.)
5713 void
5714 fileclose(struct file *f)
5715 {
5716   struct file ff;
5717
5718   acquire(&ftable.lock);
5719   if(f->ref < 1)
5720     panic("fileclose");
5721   if(--f->ref > 0){
5722     release(&ftable.lock);
5723     return;
5724   }
5725   ff = *f;
5726   f->ref = 0;
5727   f->type = FD_NONE;
5728   release(&ftable.lock);
5729
5730   if(ff.type == FD_PIPE){
5731     pipeclose(ff.pipe, ff.writable);
5732   } else if(ff.type == FD_INODE || ff.type == FD_DEVICE){
5733     begin_op();
5734     iput(ff.ip);
5735     end_op();
5736   }
5737 }
5738
5739
5740
5741
5742
5743
5744
5745
5746
5747
5748
5749
```



```
5750 // Get metadata about file f.
5751 // addr is a user virtual address, pointing to a struct stat.
5752 int
5753 filestat(struct file *f, uint64 addr)
5754 {
5755   struct proc *p = myproc();
5756   struct stat st;
5757
5758   if(f->type == FD_INODE || f->type == FD_DEVICE){
5759     ilock(f->ip);
5760     stati(f->ip, &st);
5761     iunlock(f->ip);
5762     if(copyout(p->pagetable, addr, (char *)&st, sizeof(st)) < 0)
5763       return -1;
5764     return 0;
5765   }
5766   return -1;
5767 }
5768
5769 // Read from file f.
5770 // addr is a user virtual address.
5771 int
5772 fileread(struct file *f, uint64 addr, int n)
5773 {
5774   int r = 0;
5775
5776   if(f->readable == 0)
5777     return -1;
5778
5779   if(f->type == FD_PIPE){
5780     r = piperead(f->pipe, addr, n);
5781   } else if(f->type == FD_DEVICE){
5782     if(f->major < 0 || f->major >= NDEV || !devsw[f->major].read)
5783       return -1;
5784     r = devsw[f->major].read(1, addr, n);
5785   } else if(f->type == FD_INODE){
5786     ilock(f->ip);
5787     if((r = readi(f->ip, 1, addr, f->off, n)) > 0)
5788       f->off += r;
5789     iunlock(f->ip);
5790   } else {
5791     panic("fileread");
5792   }
5793
5794   return r;
5795 }
5796
5797
5798
5799
```

```
5800 // Write to file f.
5801 // addr is a user virtual address.
5802 int
5803 filewrite(struct file *f, uint64 addr, int n)
5804 {
5805   int r, ret = 0;
5806
5807   if(f->writable == 0)
5808     return -1;
5809
5810   if(f->type == FD_PIPE){
5811     ret = pipewrite(f->pipe, addr, n);
5812   } else if(f->type == FD_DEVICE){
5813     if(f->major < 0 || f->major >= NDEV || !devsw[f->major].write)
5814       return -1;
5815     ret = devsw[f->major].write(1, addr, n);
5816   } else if(f->type == FD_INODE){
5817     // write a few blocks at a time to avoid exceeding
5818     // the maximum log transaction size, including
5819     // i-node, indirect block, allocation blocks,
5820     // and 2 blocks of slop for non-aligned writes.
5821     int max = ((MAXOPBLOCKS-1-1-2) / 2) * BSIZE;
5822     int i = 0;
5823     while(i < n){
5824       int n1 = n - i;
5825       if(n1 > max)
5826         n1 = max;
5827
5828       begin_op();
5829       ilock(f->ip);
5830       if ((r = writei(f->ip, 1, addr + i, f->off, n1)) > 0)
5831         f->off += r;
5832       iunlock(f->ip);
5833       end_op();
5834
5835       if(r != n1){
5836         // error from writei
5837         break;
5838       }
5839       i += r;
5840     }
5841     ret = (i == n ? n : -1);
5842   } else {
5843     panic("filewrite");
5844   }
5845
5846   return ret;
5847 }
5848
5849
```



```
5850 //
5851 // File-system system calls.
5852 // Mostly argument checking, since we don't trust
5853 // user code, and calls into file.c and fs.c.
5854 //
5855
5856 #include "types.h"
5857 #include "riscv.h"
5858 #include "defs.h"
5859 #include "param.h"
5860 #include "stat.h"
5861 #include "spinlock.h"
5862 #include "proc.h"
5863 #include "fs.h"
5864 #include "sleeplock.h"
5865 #include "file.h"
5866 #include "fcntl.h"
5867
5868 // Fetch the nth word-sized system call argument as a file descriptor
5869 // and return both the descriptor and the corresponding struct file.
5870 static int
5871 argfd(int n, int *pfd, struct file **pf)
5872 {
5873   int fd;
5874   struct file *f;
5875
5876   argint(n, &fd);
5877   if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
5878     return -1;
5879   if(pfd)
5880     *pfd = fd;
5881   if(pf)
5882     *pf = f;
5883   return 0;
5884 }
5885
5886
5887
5888
5889
5890
5891
5892
5893
5894
5895
5896
5897
5898
5899
```

```
5900 // Allocate a file descriptor for the given file.
5901 // Takes over file reference from caller on success.
5902 static int
5903 fdalloc(struct file *f)
5904 {
5905   int fd;
5906   struct proc *p = myproc();
5907
5908   for(fd = 0; fd < NOFILE; fd++){
5909     if(p->ofile[fd] == 0){
5910       p->ofile[fd] = f;
5911       return fd;
5912     }
5913   }
5914   return -1;
5915 }
5916
5917 uint64
5918 sys_dup(void)
5919 {
5920   struct file *f;
5921   int fd;
5922
5923   if(argfd(0, 0, &f) < 0)
5924     return -1;
5925   if((fd=fdalloc(f)) < 0)
5926     return -1;
5927   filedup(f);
5928   return fd;
5929 }
5930
5931 uint64
5932 sys_read(void)
5933 {
5934   struct file *f;
5935   int n;
5936   uint64 p;
5937
5938   argaddr(1, &p);
5939   argint(2, &n);
5940   if(argfd(0, 0, &f) < 0)
5941     return -1;
5942   return fileread(f, p, n);
5943 }
5944
5945
5946
5947
5948
5949
```



```
5950 uint64
5951 sys_write(void)
5952 {
5953   struct file *f;
5954   int n;
5955   uint64 p;
5956
5957   argaddr(1, &p);
5958   argint(2, &n);
5959   if(argfd(0, 0, &f) < 0)
5960     return -1;
5961
5962   return filewrite(f, p, n);
5963 }
5964
5965 uint64
5966 sys_close(void)
5967 {
5968   int fd;
5969   struct file *f;
5970
5971   if(argfd(0, &fd, &f) < 0)
5972     return -1;
5973   myproc()->ofile[fd] = 0;
5974   fileclose(f);
5975   return 0;
5976 }
5977
5978 uint64
5979 sys_fstat(void)
5980 {
5981   struct file *f;
5982   uint64 st; // user pointer to struct stat
5983
5984   argaddr(1, &st);
5985   if(argfd(0, 0, &f) < 0)
5986     return -1;
5987   return filestat(f, st);
5988 }
5989
5990
5991
5992
5993
5994
5995
5996
5997
5998
5999
```

```
6000 // Create the path new as a link to the same inode as old.
6001 uint64
6002 sys_link(void)
6003 {
6004   char name[DIRSIZ], new[MAXPATH], old[MAXPATH];
6005   struct inode *dp, *ip;
6006
6007   if(argstr(0, old, MAXPATH) < 0 || argstr(1, new, MAXPATH) < 0)
6008     return -1;
6009
6010   begin_op();
6011   if((ip = namei(old)) == 0){
6012     end_op();
6013     return -1;
6014   }
6015
6016   ilock(ip);
6017   if(ip->type == T_DIR){
6018     iunlockput(ip);
6019     end_op();
6020     return -1;
6021   }
6022
6023   ip->nlink++;
6024   iupdate(ip);
6025   iunlock(ip);
6026
6027   if((dp = nameiparent(new, name)) == 0)
6028     goto bad;
6029   ilock(dp);
6030   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
6031     iunlockput(dp);
6032     goto bad;
6033   }
6034   iunlockput(dp);
6035   iput(ip);
6036
6037   end_op();
6038
6039   return 0;
6040
6041 bad:
6042   ilock(ip);
6043   ip->nlink--;
6044   iupdate(ip);
6045   iunlockput(ip);
6046   end_op();
6047   return -1;
6048 }
6049
```



```
6050 // Is the directory dp empty except for "." and ".." ?
6051 static int
6052 isdirempty(struct inode *dp)
6053 {
6054   int off;
6055   struct dirent de;
6056
6057   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
6058     if(readi(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
6059       panic("isdirempty: readi");
6060     if(de.inum != 0)
6061       return 0;
6062   }
6063   return 1;
6064 }
6065
6066 uint64
6067 sys_unlink(void)
6068 {
6069   struct inode *ip, *dp;
6070   struct dirent de;
6071   char name[DIRSIZ], path[MAXPATH];
6072   uint off;
6073
6074   if(argstr(0, path, MAXPATH) < 0)
6075     return -1;
6076
6077   begin_op();
6078   if((dp = nameiparent(path, name)) == 0){
6079     end_op();
6080     return -1;
6081   }
6082
6083   ilock(dp);
6084
6085   // Cannot unlink "." or "..".
6086   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6087     goto bad;
6088
6089   if((ip = dirlookup(dp, name, &off)) == 0)
6090     goto bad;
6091   ilock(ip);
6092
6093   if(ip->nlink < 1)
6094     panic("unlink: nlink < 1");
6095   if(ip->type == T_DIR && !isdirempty(ip)){
6096     iunlockput(ip);
6097     goto bad;
6098   }
6099
```

```
6100   memset(&de, 0, sizeof(de));
6101   if(writei(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
6102     panic("unlink: writei");
6103   if(ip->type == T_DIR){
6104     dp->nlink--;
6105     iupdate(dp);
6106   }
6107   iunlockput(dp);
6108
6109   ip->nlink--;
6110   iupdate(ip);
6111   iunlockput(ip);
6112
6113   end_op();
6114
6115   return 0;
6116
6117 bad:
6118   iunlockput(dp);
6119   end_op();
6120   return -1;
6121 }
6122
6123 static struct inode*
6124 create(char *path, short type, short major, short minor)
6125 {
6126   struct inode *ip, *dp;
6127   char name[DIRSIZ];
6128
6129   if((dp = nameiparent(path, name)) == 0)
6130     return 0;
6131
6132   ilock(dp);
6133
6134   if((ip = dirlookup(dp, name, 0)) != 0){
6135     iunlockput(dp);
6136     ilock(ip);
6137     if(type == T_FILE && (ip->type == T_FILE || ip->type == T_DEVICE))
6138       return ip;
6139     iunlockput(ip);
6140     return 0;
6141   }
6142
6143   if((ip = ialloc(dp->dev, type)) == 0){
6144     iunlockput(dp);
6145     return 0;
6146   }
6147
6148
6149
```



```
6150   ilock(ip);
6151   ip->major = major;
6152   ip->minor = minor;
6153   ip->nlink = 1;
6154   iupdate(ip);
6155
6156   if(type == T_DIR){  // Create . and .. entries.
6157     // No ip->nlink++ for ".": avoid cyclic ref count.
6158     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6159       goto fail;
6160   }
6161
6162   if(dirlink(dp, name, ip->inum) < 0)
6163     goto fail;
6164
6165   if(type == T_DIR){
6166     // now that success is guaranteed:
6167     dp->nlink++;  // for ".."
6168     iupdate(dp);
6169   }
6170
6171   iunlockput(dp);
6172
6173   return ip;
6174
6175  fail:
6176   // something went wrong. de-allocate ip.
6177   ip->nlink = 0;
6178   iupdate(ip);
6179   iunlockput(ip);
6180   iunlockput(dp);
6181   return 0;
6182 }
6183
6184 uint64
6185 sys_open(void)
6186 {
6187   char path[MAXPATH];
6188   int fd, omode;
6189   struct file *f;
6190   struct inode *ip;
6191   int n;
6192
6193   argint(1, &omode);
6194   if((n = argstr(0, path, MAXPATH)) < 0)
6195     return -1;
6196
6197   begin_op();
6198
6199
```

```
6200   if(omode & O_CREATE){
6201     ip = create(path, T_FILE, 0, 0);
6202     if(ip == 0){
6203       end_op();
6204       return -1;
6205     }
6206   } else {
6207     if((ip = namei(path)) == 0){
6208       end_op();
6209       return -1;
6210     }
6211     ilock(ip);
6212     if(ip->type == T_DIR && omode != O_RDONLY){
6213       iunlockput(ip);
6214       end_op();
6215       return -1;
6216     }
6217   }
6218
6219   if(ip->type == T_DEVICE && (ip->major < 0 || ip->major >= NDEV)){
6220     iunlockput(ip);
6221     end_op();
6222     return -1;
6223   }
6224
6225   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6226     if(f)
6227       fileclose(f);
6228     iunlockput(ip);
6229     end_op();
6230     return -1;
6231   }
6232
6233   if(ip->type == T_DEVICE){
6234     f->type = FD_DEVICE;
6235     f->major = ip->major;
6236   } else {
6237     f->type = FD_INODE;
6238     f->off = 0;
6239   }
6240   f->ip = ip;
6241   f->readable = !(omode & O_WRONLY);
6242   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6243
6244   if((omode & O_TRUNC) && ip->type == T_FILE){
6245     itrunc(ip);
6246   }
6247
6248   iunlock(ip);
6249   end_op();
```



```
6250   return fd;
6251 }
6252
6253 uint64
6254 sys_mkdir(void)
6255 {
6256   char path[MAXPATH];
6257   struct inode *ip;
6258
6259   begin_op();
6260   if(argstr(0, path, MAXPATH) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
6261     end_op();
6262     return -1;
6263   }
6264   iunlockput(ip);
6265   end_op();
6266   return 0;
6267 }
6268
6269 uint64
6270 sys_mknod(void)
6271 {
6272   struct inode *ip;
6273   char path[MAXPATH];
6274   int major, minor;
6275
6276   begin_op();
6277   argint(1, &major);
6278   argint(2, &minor);
6279   if((argstr(0, path, MAXPATH)) < 0 ||
6280      (ip = create(path, T_DEVICE, major, minor)) == 0){
6281     end_op();
6282     return -1;
6283   }
6284   iunlockput(ip);
6285   end_op();
6286   return 0;
6287 }
6288
6289
6290
6291
6292
6293
6294
6295
6296
6297
6298
6299
```

```
6300 uint64
6301 sys_chdir(void)
6302 {
6303   char path[MAXPATH];
6304   struct inode *ip;
6305   struct proc *p = myproc();
6306
6307   begin_op();
6308   if(argstr(0, path, MAXPATH) < 0 || (ip = namei(path)) == 0){
6309     end_op();
6310     return -1;
6311   }
6312   ilock(ip);
6313   if(ip->type != T_DIR){
6314     iunlockput(ip);
6315     end_op();
6316     return -1;
6317   }
6318   iunlock(ip);
6319   iput(p->cwd);
6320   end_op();
6321   p->cwd = ip;
6322   return 0;
6323 }
6324
6325 uint64
6326 sys_exec(void)
6327 {
6328   char path[MAXPATH], *argv[MAXARG];
6329   int i;
6330   uint64 uargv, uarg;
6331
6332   argaddr(1, &uargv);
6333   if(argstr(0, path, MAXPATH) < 0) {
6334     return -1;
6335   }
6336   memset(argv, 0, sizeof(argv));
6337   for(i=0;; i++){
6338     if(i >= NELEM(argv)){
6339       goto bad;
6340     }
6341     if(fetchaddr(uargv+sizeof(uint64)*i, (uint64*)&uarg) < 0){
6342       goto bad;
6343     }
6344     if(uarg == 0){
6345       argv[i] = 0;
6346       break;
6347     }
6348     argv[i] = kalloc();
6349     if(argv[i] == 0)
```



```
6350       goto bad;
6351     if(fetchstr(uarg, argv[i], PGSIZE) < 0)
6352       goto bad;
6353   }
6354
6355   int ret = kexec(path, argv);
6356
6357   for(i = 0; i < NELEM(argv) && argv[i] != 0; i++)
6358     kfree(argv[i]);
6359
6360   return ret;
6361
6362  bad:
6363   for(i = 0; i < NELEM(argv) && argv[i] != 0; i++)
6364     kfree(argv[i]);
6365   return -1;
6366 }
6367
6368 uint64
6369 sys_pipe(void)
6370 {
6371   uint64 fdarray; // user pointer to array of two integers
6372   struct file *rf, *wf;
6373   int fd0, fd1;
6374   struct proc *p = myproc();
6375
6376   argaddr(0, &fdarray);
6377   if(pipealloc(&rf, &wf) < 0)
6378     return -1;
6379   fd0 = -1;
6380   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6381     if(fd0 >= 0)
6382       p->ofile[fd0] = 0;
6383     fileclose(rf);
6384     fileclose(wf);
6385     return -1;
6386   }
6387   if(copyout(p->pagetable, fdarray, (char*)&fd0, sizeof(fd0)) < 0 ||
6388      copyout(p->pagetable, fdarray+sizeof(fd0), (char *)&fd1, sizeof(fd1)) <
6389     p->ofile[fd0] = 0;
6390     p->ofile[fd1] = 0;
6391     fileclose(rf);
6392     fileclose(wf);
6393     return -1;
6394   }
6395   return 0;
6396 }
6397
6398
6399
```

```
6400 #include "types.h"
6401 #include "param.h"
6402 #include "memlayout.h"
6403 #include "riscv.h"
6404 #include "spinlock.h"
6405 #include "proc.h"
6406 #include "defs.h"
6407 #include "elf.h"
6408
6409 static int loadseg(pde_t *, uint64, struct inode *, uint, uint);
6410
6411 // map ELF permissions to PTE permission bits.
6412 int flags2perm(int flags)
6413 {
6414     int perm = 0;
6415     if(flags & 0x1)
6416       perm = PTE_X;
6417     if(flags & 0x2)
6418       perm |= PTE_W;
6419     return perm;
6420 }
6421
6422 //
6423 // the implementation of the exec() system call
6424 //
6425 int
6426 kexec(char *path, char **argv)
6427 {
6428   char *s, *last;
6429   int i, off;
6430   uint64 argc, sz = 0, sp, ustack[MAXARG], stackbase;
6431   struct elfhdr elf;
6432   struct inode *ip;
6433   struct proghdr ph;
6434   pagetable_t pagetable = 0, oldpagetable;
6435   struct proc *p = myproc();
6436
6437   begin_op();
6438
6439   // Open the executable file.
6440   if((ip = namei(path)) == 0){
6441     end_op();
6442     return -1;
6443   }
6444   ilock(ip);
6445
6446   // Read the ELF header.
6447   if(readi(ip, 0, (uint64)&elf, 0, sizeof(elf)) != sizeof(elf))
6448     goto bad;
6449
```



```
6450   // Is this really an ELF file?
6451   if(elf.magic != ELF_MAGIC)
6452     goto bad;
6453
6454   if((pagetable = proc_pagetable(p)) == 0)
6455     goto bad;
6456
6457   // Load program into memory.
6458   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6459     if(readi(ip, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
6460       goto bad;
6461     if(ph.type != ELF_PROG_LOAD)
6462       continue;
6463     if(ph.memsz < ph.filesz)
6464       goto bad;
6465     if(ph.vaddr + ph.memsz < ph.vaddr)
6466       goto bad;
6467     if(ph.vaddr % PGSIZE != 0)
6468       goto bad;
6469     uint64 sz1;
6470     if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz, flags2perm(ph.fla
6471       goto bad;
6472     sz = sz1;
6473     if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0)
6474       goto bad;
6475   }
6476   iunlockput(ip);
6477   end_op();
6478   ip = 0;
6479
6480   p = myproc();
6481   uint64 oldsz = p->sz;
6482
6483   // Allocate some pages at the next page boundary.
6484   // Make the first inaccessible as a stack guard.
6485   // Use the rest as the user stack.
6486   sz = PGROUNDUP(sz);
6487   uint64 sz1;
6488   if((sz1 = uvmalloc(pagetable, sz, sz + (USERSTACK+1)*PGSIZE, PTE_W)) == 0)
6489     goto bad;
6490   sz = sz1;
6491   uvmclear(pagetable, sz-(USERSTACK+1)*PGSIZE);
6492   sp = sz;
6493   stackbase = sp - USERSTACK*PGSIZE;
6494
6495   // Copy argument strings into new stack, remember their
6496   // addresses in ustack[].
6497   for(argc = 0; argv[argc]; argc++) {
6498     if(argc >= MAXARG)
6499       goto bad;
```

```
    sp -= strlen(argv[argc]) + 1;
    sp -= sp % 16; // riscv sp must be 16-byte aligned
    if(sp < stackbase)
      goto bad;
    if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
      goto bad;
    ustack[argc] = sp;
  }
  ustack[argc] = 0;

  // push a copy of ustack[], the array of argv[] pointers.
  sp -= (argc+1) * sizeof(uint64);
  sp -= sp % 16;
  if(sp < stackbase)
    goto bad;
  if(copyout(pagetable, sp, (char *)ustack, (argc+1)*sizeof(uint64)) < 0)
    goto bad;

  // a0 and a1 contain arguments to user main(argc, argv)
  // argc is returned via the system call return
  // value, which goes in a0.
  p->trapframe->a1 = sp;

  // Save program name for debugging.
  for(last=s=path; *s; s++)
    if(*s == '/')
      last = s+1;
  safestrcpy(p->name, last, sizeof(p->name));

  // Commit to the user image.
  oldpagetable = p->pagetable;
  p->pagetable = pagetable;
  p->sz = sz;
  p->trapframe->epc = elf.entry;  // initial program counter = main
  p->trapframe->sp = sp; // initial stack pointer
  proc_freepagetable(oldpagetable, oldsz);

  return argc; // this ends up in a0, the first argument to main(argc, argv)

 bad:
  if(pagetable)
    proc_freepagetable(pagetable, sz);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
}
```



```
// Load an ELF program segment into pagetable at virtual address va.
// va must be page-aligned
// and the pages from va to va+sz must already be mapped.
// Returns 0 on success, -1 on failure.
static int
loadseg(pagetable_t pagetable, uint64 va, struct inode *ip, uint offset, uin
{
  uint i, n;
  uint64 pa;

  for(i = 0; i < sz; i += PGSIZE){
    pa = walkaddr(pagetable, va + i);
    if(pa == 0)
      panic("loadseg: address should exist");
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, 0, (uint64)pa, offset+i, n) != n)
      return -1;
  }

  return 0;
}
```

```
6600 #include "types.h"
6601 #include "riscv.h"
6602 #include "defs.h"
6603 #include "param.h"
6604 #include "spinlock.h"
6605 #include "proc.h"
6606 #include "fs.h"
6607 #include "sleeplock.h"
6608 #include "file.h"
6609
6610 #define PIPESIZE 512
6611
6612 struct pipe {
6613   struct spinlock lock;
6614   char data[PIPESIZE];
6615   uint nread;     // number of bytes read
6616   uint nwrite;    // number of bytes written
6617   int readopen;   // read fd is still open
6618   int writeopen;  // write fd is still open
6619 };
6620
6621 int
6622 pipealloc(struct file **f0, struct file **f1)
6623 {
6624   struct pipe *pi;
6625
6626   pi = 0;
6627   *f0 = *f1 = 0;
6628   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
6629     goto bad;
6630   if((pi = (struct pipe*)kalloc()) == 0)
6631     goto bad;
6632   pi->readopen = 1;
6633   pi->writeopen = 1;
6634   pi->nwrite = 0;
6635   pi->nread = 0;
6636   initlock(&pi->lock, "pipe");
6637   (*f0)->type = FD_PIPE;
6638   (*f0)->readable = 1;
6639   (*f0)->writable = 0;
6640   (*f0)->pipe = pi;
6641   (*f1)->type = FD_PIPE;
6642   (*f1)->readable = 0;
6643   (*f1)->writable = 1;
6644   (*f1)->pipe = pi;
6645   return 0;
6646
6647
6648
6649
```



```
6650  bad:
6651   if(pi)
6652     kfree((char*)pi);
6653   if(*f0)
6654     fileclose(*f0);
6655   if(*f1)
6656     fileclose(*f1);
6657   return -1;
6658 }
6659
6660 void
6661 pipeclose(struct pipe *pi, int writable)
6662 {
6663   acquire(&pi->lock);
6664   if(writable){
6665     pi->writeopen = 0;
6666     wakeup(&pi->nread);
6667   } else {
6668     pi->readopen = 0;
6669     wakeup(&pi->nwrite);
6670   }
6671   if(pi->readopen == 0 && pi->writeopen == 0){
6672     release(&pi->lock);
6673     kfree((char*)pi);
6674   } else
6675     release(&pi->lock);
6676 }
6677
6678 int
6679 pipewrite(struct pipe *pi, uint64 addr, int n)
6680 {
6681   int i = 0;
6682   struct proc *pr = myproc();
6683
6684   acquire(&pi->lock);
6685   while(i < n){
6686     if(pi->readopen == 0 || killed(pr)){
6687       release(&pi->lock);
6688       return -1;
6689     }
6690     if(pi->nwrite == pi->nread + PIPESIZE){
6691       wakeup(&pi->nread);
6692       sleep(&pi->nwrite, &pi->lock);
6693     } else {
6694       char ch;
6695       if(copyin(pr->pagetable, &ch, addr + i, 1) == -1)
6696         break;
6697       pi->data[pi->nwrite++ % PIPESIZE] = ch;
6698       i++;
6699     }
```

```
6700   }
6701   wakeup(&pi->nread);
6702   release(&pi->lock);
6703
6704   return i;
6705 }
6706
6707 int
6708 piperead(struct pipe *pi, uint64 addr, int n)
6709 {
6710   int i;
6711   struct proc *pr = myproc();
6712   char ch;
6713
6714   acquire(&pi->lock);
6715   while(pi->nread == pi->nwrite && pi->writeopen){
6716     if(killed(pr)){
6717       release(&pi->lock);
6718       return -1;
6719     }
6720     sleep(&pi->nread, &pi->lock);
6721   }
6722   for(i = 0; i < n; i++){
6723     if(pi->nread == pi->nwrite)
6724       break;
6725     ch = pi->data[pi->nread++ % PIPESIZE];
6726     if(copyout(pr->pagetable, addr + i, &ch, 1) == -1)
6727       break;
6728   }
6729   wakeup(&pi->nwrite);
6730   release(&pi->lock);
6731   return i;
6732 }
6733
6734
6735
6736
6737
6738
6739
6740
6741
6742
6743
6744
6745
6746
6747
6748
6749
```



```
6750 #include "types.h"
6751
6752 void*
6753 memset(void *dst, int c, uint n)
6754 {
6755   char *cdst = (char *) dst;
6756   int i;
6757   for(i = 0; i < n; i++){
6758     cdst[i] = c;
6759   }
6760   return dst;
6761 }
6762
6763 int
6764 memcmp(const void *v1, const void *v2, uint n)
6765 {
6766   const uchar *s1, *s2;
6767
6768   s1 = v1;
6769   s2 = v2;
6770   while(n-- > 0){
6771     if(*s1 != *s2)
6772       return *s1 - *s2;
6773     s1++, s2++;
6774   }
6775
6776   return 0;
6777 }
6778
6779 void*
6780 memmove(void *dst, const void *src, uint n)
6781 {
6782   const char *s;
6783   char *d;
6784
6785   if(n == 0)
6786     return dst;
6787
6788   s = src;
6789   d = dst;
6790   if(s < d && s + n > d){
6791     s += n;
6792     d += n;
6793     while(n-- > 0)
6794       *--d = *--s;
6795   } else
6796     while(n-- > 0)
6797       *d++ = *s++;
6798
6799
```

```
6800   return dst;
6801 }
6802
6803 // memcpy exists to placate GCC.  Use memmove.
6804 void*
6805 memcpy(void *dst, const void *src, uint n)
6806 {
6807   return memmove(dst, src, n);
6808 }
6809
6810 int
6811 strncmp(const char *p, const char *q, uint n)
6812 {
6813   while(n > 0 && *p && *p == *q)
6814     n--, p++, q++;
6815   if(n == 0)
6816     return 0;
6817   return (uchar)*p - (uchar)*q;
6818 }
6819
6820 char*
6821 strncpy(char *s, const char *t, int n)
6822 {
6823   char *os;
6824
6825   os = s;
6826   while(n-- > 0 && (*s++ = *t++) != 0)
6827     ;
6828   while(n-- > 0)
6829     *s++ = 0;
6830   return os;
6831 }
6832
6833 // Like strncpy but guaranteed to NUL-terminate.
6834 char*
6835 safestrcpy(char *s, const char *t, int n)
6836 {
6837   char *os;
6838
6839   os = s;
6840   if(n <= 0)
6841     return os;
6842   while(--n > 0 && (*s++ = *t++) != 0)
6843     ;
6844   *s = 0;
6845   return os;
6846 }
6847
6848
6849
```



```
6850 int
6851 strlen(const char *s)
6852 {
6853   int n;
6854
6855   for(n = 0; s[n]; n++)
6856     ;
6857   return n;
6858 }
6859
6860
6861
6862
6863
6864
6865
6866
6867
6868
6869
6870
6871
6872
6873
6874
6875
6876
6877
6878
6879
6880
6881
6882
6883
6884
6885
6886
6887
6888
6889
6890
6891
6892
6893
6894
6895
6896
6897
6898
6899
```

```
6900 #include "types.h"
6901 #include "param.h"
6902 #include "memlayout.h"
6903 #include "riscv.h"
6904 #include "defs.h"
6905
6906 //
6907 // the riscv Platform Level Interrupt Controller (PLIC).
6908 //
6909
6910 void
6911 plicinit(void)
6912 {
6913   // set desired IRQ priorities non-zero (otherwise disabled).
6914   *(uint32*)(PLIC + UART0_IRQ*4) = 1;
6915   *(uint32*)(PLIC + VIRTIO0_IRQ*4) = 1;
6916 }
6917
6918 void
6919 plicinithart(void)
6920 {
6921   int hart = cpuid();
6922
6923   // set enable bits for this hart's S-mode
6924   // for the uart and virtio disk.
6925   *(uint32*)PLIC_SENABLE(hart) = (1 << UART0_IRQ) | (1 << VIRTIO0_IRQ);
6926
6927   // set this hart's S-mode priority threshold to 0.
6928   *(uint32*)PLIC_SPRIORITY(hart) = 0;
6929 }
6930
6931 // ask the PLIC what interrupt we should serve.
6932 int
6933 plic_claim(void)
6934 {
6935   int hart = cpuid();
6936   int irq = *(uint32*)PLIC_SCLAIM(hart);
6937   return irq;
6938 }
6939
6940 // tell the PLIC we've served this IRQ.
6941 void
6942 plic_complete(int irq)
6943 {
6944   int hart = cpuid();
6945   *(uint32*)PLIC_SCLAIM(hart) = irq;
6946 }
6947
6948
6949
```



```
6950 //
6951 // Console input and output, to the uart.
6952 // Reads are line at a time.
6953 // Implements special input characters:
6954 //   newline -- end of line
6955 //   control-h -- backspace
6956 //   control-u -- kill line
6957 //   control-d -- end of file
6958 //   control-p -- print process list
6959 //
6960
6961 #include <stdarg.h>
6962
6963 #include "types.h"
6964 #include "param.h"
6965 #include "spinlock.h"
6966 #include "sleeplock.h"
6967 #include "fs.h"
6968 #include "file.h"
6969 #include "memlayout.h"
6970 #include "riscv.h"
6971 #include "defs.h"
6972 #include "proc.h"
6973
6974 #define BACKSPACE 0x100
6975 #define C(x)  ((x)-'@')  // Control-x
6976
6977 //
6978 // send one character to the uart.
6979 // called by printf(), and to echo input characters,
6980 // but not from write().
6981 //
6982 void
6983 consputc(int c)
6984 {
6985   if(c == BACKSPACE){
6986     // if the user typed backspace, overwrite with a space.
6987     uartputc_sync('\b'); uartputc_sync(' '); uartputc_sync('\b');
6988   } else {
6989     uartputc_sync(c);
6990   }
6991 }
6992
6993
6994
6995
6996
6997
6998
6999
```

```
7000 struct {
7001   struct spinlock lock;
7002
7003   // input
7004 #define INPUT_BUF_SIZE 128
7005   char buf[INPUT_BUF_SIZE];
7006   uint r;  // Read index
7007   uint w;  // Write index
7008   uint e;  // Edit index
7009 } cons;
7010
7011 //
7012 // user write()s to the console go here.
7013 //
7014 int
7015 consolewrite(int user_src, uint64 src, int n)
7016 {
7017   char buf[32];
7018   int i = 0;
7019
7020   while(i < n){
7021     int nn = sizeof(buf);
7022     if(nn > n - i)
7023       nn = n - i;
7024     if(either_copyin(buf, user_src, src+i, nn) == -1)
7025       break;
7026     uartwrite(buf, nn);
7027     i += nn;
7028   }
7029
7030   return i;
7031 }
7032
7033 //
7034 // user read()s from the console go here.
7035 // copy (up to) a whole input line to dst.
7036 // user_dist indicates whether dst is a user
7037 // or kernel address.
7038 //
7039 int
7040 consoleread(int user_dst, uint64 dst, int n)
7041 {
7042   uint target;
7043   int c;
7044   char cbuf;
7045
7046   target = n;
7047   acquire(&cons.lock);
7048   while(n > 0){
7049     // wait until interrupt handler has put some
```



```
7050     // input into cons.buffer.
7051     while(cons.r == cons.w){
7052       if(killed(myproc())){
7053         release(&cons.lock);
7054         return -1;
7055       }
7056       sleep(&cons.r, &cons.lock);
7057     }
7058
7059     c = cons.buf[cons.r++ % INPUT_BUF_SIZE];
7060
7061     if(c == C('D')){  // end-of-file
7062       if(n < target){
7063         // Save ^D for next time, to make sure
7064         // caller gets a 0-byte result.
7065         cons.r--;
7066       }
7067       break;
7068     }
7069
7070     // copy the input byte to the user-space buffer.
7071     cbuf = c;
7072     if(either_copyout(user_dst, dst, &cbuf, 1) == -1)
7073       break;
7074
7075     dst++;
7076     --n;
7077
7078     if(c == '\n'){
7079       // a whole line has arrived, return to
7080       // the user-level read().
7081       break;
7082     }
7083   }
7084   release(&cons.lock);
7085
7086   return target - n;
7087 }
7088
7089
7090
7091
7092
7093
7094
7095
7096
7097
7098
7099
```

```
7100 //
7101 // the console input interrupt handler.
7102 // uartintr() calls this for input character.
7103 // do erase/kill processing, append to cons.buf,
7104 // wake up consoleread() if a whole line has arrived.
7105 //
7106 void
7107 consoleintr(int c)
7108 {
7109   acquire(&cons.lock);
7110
7111   switch(c){
7112   case C('P'):  // Print process list.
7113     procdump();
7114     break;
7115   case C('U'):  // Kill line.
7116     while(cons.e != cons.w &&
7117           cons.buf[(cons.e-1) % INPUT_BUF_SIZE] != '\n'){
7118       cons.e--;
7119       consputc(BACKSPACE);
7120     }
7121     break;
7122   case C('H'): // Backspace
7123   case '\x7f': // Delete key
7124     if(cons.e != cons.w){
7125       cons.e--;
7126       consputc(BACKSPACE);
7127     }
7128     break;
7129   default:
7130     if(c != 0 && cons.e-cons.r < INPUT_BUF_SIZE){
7131       c = (c == '\r') ? '\n' : c;
7132
7133       // echo back to the user.
7134       consputc(c);
7135
7136       // store for consumption by consoleread().
7137       cons.buf[cons.e++ % INPUT_BUF_SIZE] = c;
7138
7139       if(c == '\n' || c == C('D') || cons.e-cons.r == INPUT_BUF_SIZE){
7140         // wake up consoleread() if a whole line (or end-of-file)
7141         // has arrived.
7142         cons.w = cons.e;
7143         wakeup(&cons.r);
7144       }
7145     }
7146     break;
7147   }
7148
7149
```



```
7150   release(&cons.lock);
7151 }
7152
7153 void
7154 consoleinit(void)
7155 {
7156   initlock(&cons.lock, "cons");
7157
7158   uartinit();
7159
7160   // connect read and write system calls
7161   // to consoleread and consolewrite.
7162   devsw[CONSOLE].read = consoleread;
7163   devsw[CONSOLE].write = consolewrite;
7164 }
7165
7166
7167
7168
7169
7170
7171
7172
7173
7174
7175
7176
7177
7178
7179
7180
7181
7182
7183
7184
7185
7186
7187
7188
7189
7190
7191
7192
7193
7194
7195
7196
7197
7198
7199
```

```
7200 //
7201 // low-level driver routines for 16550a UART.
7202 //
7203
7204 #include "types.h"
7205 #include "param.h"
7206 #include "memlayout.h"
7207 #include "riscv.h"
7208 #include "spinlock.h"
7209 #include "proc.h"
7210 #include "defs.h"
7211
7212 // the UART control registers are memory-mapped
7213 // at address UART0. this macro returns the
7214 // address of one of the registers.
7215 #define Reg(reg) ((volatile unsigned char *)(UART0 + (reg)))
7216
7217 // the UART control registers.
7218 // some have different meanings for
7219 // read vs write.
7220 // see http://byterunner.com/16550.html
7221 #define RHR 0                 // receive holding register (for input bytes)
7222 #define THR 0                 // transmit holding register (for output bytes
7223 #define IER 1                 // interrupt enable register
7224 #define IER_RX_ENABLE (1<<0)
7225 #define IER_TX_ENABLE (1<<1)
7226 #define FCR 2                 // FIFO control register
7227 #define FCR_FIFO_ENABLE (1<<0)
7228 #define FCR_FIFO_CLEAR (3<<1) // clear the content of the two FIFOs
7229 #define ISR 2                 // interrupt status register
7230 #define LCR 3                 // line control register
7231 #define LCR_EIGHT_BITS (3<<0)
7232 #define LCR_BAUD_LATCH (1<<7) // special mode to set baud rate
7233 #define LSR 5                 // line status register
7234 #define LSR_RX_READY (1<<0)   // input is waiting to be read from RHR
7235 #define LSR_TX_IDLE (1<<5)    // THR can accept another character to send
7236
7237 #define ReadReg(reg) (*(Reg(reg)))
7238 #define WriteReg(reg, v) (*(Reg(reg)) = (v))
7239
7240 // for transmission.
7241 static struct spinlock tx_lock;
7242 static int tx_busy;           // is the UART busy sending?
7243 static int tx_chan;           // &tx_chan is the "wait channel"
7244
7245 extern volatile int panicking; // from printf.c
7246 extern volatile int panicked; // from printf.c
7247
7248
7249
```



```
7250 void
7251 uartinit(void)
7252 {
7253   // disable interrupts.
7254   WriteReg(IER, 0x00);
7255
7256   // special mode to set baud rate.
7257   WriteReg(LCR, LCR_BAUD_LATCH);
7258
7259   // LSB for baud rate of 38.4K.
7260   WriteReg(0, 0x03);
7261
7262   // MSB for baud rate of 38.4K.
7263   WriteReg(1, 0x00);
7264
7265   // leave set-baud mode,
7266   // and set word length to 8 bits, no parity.
7267   WriteReg(LCR, LCR_EIGHT_BITS);
7268
7269   // reset and enable FIFOs.
7270   WriteReg(FCR, FCR_FIFO_ENABLE | FCR_FIFO_CLEAR);
7271
7272   // enable transmit and receive interrupts.
7273   WriteReg(IER, IER_TX_ENABLE | IER_RX_ENABLE);
7274
7275   initlock(&tx_lock, "uart");
7276 }
7277
7278 // transmit buf[] to the uart. it blocks if the
7279 // uart is busy, so it cannot be called from
7280 // interrupts, only from write() system calls.
7281 void
7282 uartwrite(char buf[], int n)
7283 {
7284   acquire(&tx_lock);
7285
7286   int i = 0;
7287   while(i < n){
7288     while(tx_busy != 0){
7289       // wait for a UART transmit-complete interrupt
7290       // to set tx_busy to 0.
7291       sleep(&tx_chan, &tx_lock);
7292     }
7293
7294     WriteReg(THR, buf[i]);
7295     i += 1;
7296     tx_busy = 1;
7297   }
7298
7299
```

```
7300   release(&tx_lock);
7301 }
7302
7303
7304 // write a byte to the uart without using
7305 // interrupts, for use by kernel printf() and
7306 // to echo characters. it spins waiting for the uart's
7307 // output register to be empty.
7308 void
7309 uartputc_sync(int c)
7310 {
7311   if(panicking == 0)
7312     push_off();
7313
7314   if(panicked){
7315     for(;;)
7316       ;
7317   }
7318
7319   // wait for Transmit Holding Empty to be set in LSR.
7320   while((ReadReg(LSR) & LSR_TX_IDLE) == 0)
7321     ;
7322   WriteReg(THR, c);
7323
7324   if(panicking == 0)
7325     pop_off();
7326 }
7327
7328 // read one input character from the UART.
7329 // return -1 if none is waiting.
7330 int
7331 uartgetc(void)
7332 {
7333   if(ReadReg(LSR) & LSR_RX_READY){
7334     // input data is ready.
7335     return ReadReg(RHR);
7336   } else {
7337     return -1;
7338   }
7339 }
7340
7341
7342
7343
7344
7345
7346
7347
7348
7349
```



```
7350 // handle a uart interrupt, raised because input has
7351 // arrived, or the uart is ready for more output, or
7352 // both. called from devintr().
7353 void
7354 uartintr(void)
7355 {
7356   ReadReg(ISR); // acknowledge the interrupt
7357
7358   acquire(&tx_lock);
7359   if(ReadReg(LSR) & LSR_TX_IDLE){
7360     // UART finished transmitting; wake up sending thread.
7361     tx_busy = 0;
7362     wakeup(&tx_chan);
7363   }
7364   release(&tx_lock);
7365
7366   // read and process incoming characters.
7367   while(1){
7368     int c = uartgetc();
7369     if(c == -1)
7370       break;
7371     consoleintr(c);
7372   }
7373 }
7374
7375
7376
7377
7378
7379
7380
7381
7382
7383
7384
7385
7386
7387
7388
7389
7390
7391
7392
7393
7394
7395
7396
7397
7398
7399
```

```
7400 //
7401 // driver for qemu's virtio disk device.
7402 // uses qemu's mmio interface to virtio.
7403 //
7404 // qemu ... -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-d
7405 //
7406
7407 #include "types.h"
7408 #include "riscv.h"
7409 #include "defs.h"
7410 #include "param.h"
7411 #include "memlayout.h"
7412 #include "spinlock.h"
7413 #include "sleeplock.h"
7414 #include "fs.h"
7415 #include "buf.h"
7416 #include "virtio.h"
7417
7418 // the address of virtio mmio register r.
7419 #define R(r) ((volatile uint32 *)(VIRTIO0 + (r)))
7420
7421 static struct disk {
7422   // a set (not a ring) of DMA descriptors, with which the
7423   // driver tells the device where to read and write individual
7424   // disk operations. there are NUM descriptors.
7425   // most commands consist of a "chain" (a linked list) of a couple of
7426   // these descriptors.
7427   struct virtq_desc *desc;
7428
7429   // a ring in which the driver writes descriptor numbers
7430   // that the driver would like the device to process.  it only
7431   // includes the head descriptor of each chain. the ring has
7432   // NUM elements.
7433   struct virtq_avail *avail;
7434
7435   // a ring in which the device writes descriptor numbers that
7436   // the device has finished processing (just the head of each chain).
7437   // there are NUM used ring entries.
7438   struct virtq_used *used;
7439
7440   // our own book-keeping.
7441   char free[NUM];  // is a descriptor free?
7442   uint16 used_idx; // we've looked this far in used[2..NUM].
7443
7444   // track info about in-flight operations,
7445   // for use when completion interrupt arrives.
7446   // indexed by first descriptor index of chain.
7447   struct {
7448     struct buf *b;
7449     char status;
```



```
7450   } info[NUM];
7451
7452   // disk command headers.
7453   // one-for-one with descriptors, for convenience.
7454   struct virtio_blk_req ops[NUM];
7455
7456   struct spinlock vdisk_lock;
7457
7458 } disk;
7459
7460 void
7461 virtio_disk_init(void)
7462 {
7463   uint32 status = 0;
7464
7465   initlock(&disk.vdisk_lock, "virtio_disk");
7466
7467   if(*R(VIRTIO_MMIO_MAGIC_VALUE) != 0x74726976 ||
7468      *R(VIRTIO_MMIO_VERSION) != 2 ||
7469      *R(VIRTIO_MMIO_DEVICE_ID) != 2 ||
7470      *R(VIRTIO_MMIO_VENDOR_ID) != 0x554d4551){
7471     panic("could not find virtio disk");
7472   }
7473
7474   // reset device
7475   *R(VIRTIO_MMIO_STATUS) = status;
7476
7477   // set ACKNOWLEDGE status bit
7478   status |= VIRTIO_CONFIG_S_ACKNOWLEDGE;
7479   *R(VIRTIO_MMIO_STATUS) = status;
7480
7481   // set DRIVER status bit
7482   status |= VIRTIO_CONFIG_S_DRIVER;
7483   *R(VIRTIO_MMIO_STATUS) = status;
7484
7485   // negotiate features
7486   uint64 features = *R(VIRTIO_MMIO_DEVICE_FEATURES);
7487   features &= ~(1 << VIRTIO_BLK_F_RO);
7488   features &= ~(1 << VIRTIO_BLK_F_SCSI);
7489   features &= ~(1 << VIRTIO_BLK_F_CONFIG_WCE);
7490   features &= ~(1 << VIRTIO_BLK_F_MQ);
7491   features &= ~(1 << VIRTIO_F_ANY_LAYOUT);
7492   features &= ~(1 << VIRTIO_RING_F_EVENT_IDX);
7493   features &= ~(1 << VIRTIO_RING_F_INDIRECT_DESC);
7494   *R(VIRTIO_MMIO_DRIVER_FEATURES) = features;
7495
7496   // tell device that feature negotiation is complete.
7497   status |= VIRTIO_CONFIG_S_FEATURES_OK;
7498   *R(VIRTIO_MMIO_STATUS) = status;
7499
```

```
7500   // re-read status to ensure FEATURES_OK is set.
7501   status = *R(VIRTIO_MMIO_STATUS);
7502   if(!(status & VIRTIO_CONFIG_S_FEATURES_OK))
7503     panic("virtio disk FEATURES_OK unset");
7504
7505   // initialize queue 0.
7506   *R(VIRTIO_MMIO_QUEUE_SEL) = 0;
7507
7508   // ensure queue 0 is not in use.
7509   if(*R(VIRTIO_MMIO_QUEUE_READY))
7510     panic("virtio disk should not be ready");
7511
7512   // check maximum queue size.
7513   uint32 max = *R(VIRTIO_MMIO_QUEUE_NUM_MAX);
7514   if(max == 0)
7515     panic("virtio disk has no queue 0");
7516   if(max < NUM)
7517     panic("virtio disk max queue too short");
7518
7519   // allocate and zero queue memory.
7520   disk.desc = kalloc();
7521   disk.avail = kalloc();
7522   disk.used = kalloc();
7523   if(!disk.desc || !disk.avail || !disk.used)
7524     panic("virtio disk kalloc");
7525   memset(disk.desc, 0, PGSIZE);
7526   memset(disk.avail, 0, PGSIZE);
7527   memset(disk.used, 0, PGSIZE);
7528
7529   // set queue size.
7530   *R(VIRTIO_MMIO_QUEUE_NUM) = NUM;
7531
7532   // write physical addresses.
7533   *R(VIRTIO_MMIO_QUEUE_DESC_LOW) = (uint64)disk.desc;
7534   *R(VIRTIO_MMIO_QUEUE_DESC_HIGH) = (uint64)disk.desc >> 32;
7535   *R(VIRTIO_MMIO_DRIVER_DESC_LOW) = (uint64)disk.avail;
7536   *R(VIRTIO_MMIO_DRIVER_DESC_HIGH) = (uint64)disk.avail >> 32;
7537   *R(VIRTIO_MMIO_DEVICE_DESC_LOW) = (uint64)disk.used;
7538   *R(VIRTIO_MMIO_DEVICE_DESC_HIGH) = (uint64)disk.used >> 32;
7539
7540   // queue is ready.
7541   *R(VIRTIO_MMIO_QUEUE_READY) = 0x1;
7542
7543   // all NUM descriptors start out unused.
7544   for(int i = 0; i < NUM; i++)
7545     disk.free[i] = 1;
7546
7547   // tell device we're completely ready.
7548   status |= VIRTIO_CONFIG_S_DRIVER_OK;
7549   *R(VIRTIO_MMIO_STATUS) = status;
```



```
7550   // plic.c and trap.c arrange for interrupts from VIRTIOO_IRQ.
7551 }
7552
7553 // find a free descriptor, mark it non-free, return its index.
7554 static int
7555 alloc_desc()
7556 {
7557   for(int i = 0; i < NUM; i++){
7558     if(disk.free[i]){
7559       disk.free[i] = 0;
7560       return i;
7561     }
7562   }
7563   return -1;
7564 }
7565
7566 // mark a descriptor as free.
7567 static void
7568 free_desc(int i)
7569 {
7570   if(i >= NUM)
7571     panic("free_desc 1");
7572   if(disk.free[i])
7573     panic("free_desc 2");
7574   disk.desc[i].addr = 0;
7575   disk.desc[i].len = 0;
7576   disk.desc[i].flags = 0;
7577   disk.desc[i].next = 0;
7578   disk.free[i] = 1;
7579   wakeup(&disk.free[0]);
7580 }
7581
7582 // free a chain of descriptors.
7583 static void
7584 free_chain(int i)
7585 {
7586   while(1){
7587     int flag = disk.desc[i].flags;
7588     int nxt = disk.desc[i].next;
7589     free_desc(i);
7590     if(flag & VRING_DESC_F_NEXT)
7591       i = nxt;
7592     else
7593       break;
7594   }
7595 }
7596
7597
7598
7599
```

```
// allocate three descriptors (they need not be contiguous).
// disk transfers always use three descriptors.
static int
alloc3_desc(int *idx)
{
  for(int i = 0; i < 3; i++){
    idx[i] = alloc_desc();
    if(idx[i] < 0){
      for(int j = 0; j < i; j++)
        free_desc(idx[j]);
      return -1;
    }
  }
  return 0;
}

void
virtio_disk_rw(struct buf *b, int write)
{
  uint64 sector = b->blockno * (BSIZE / 512);

  acquire(&disk.vdisk_lock);

  // the spec's Section 5.2 says that legacy block operations use
  // three descriptors: one for type/reserved/sector, one for the
  // data, one for a 1-byte status result.

  // allocate the three descriptors.
  int idx[3];
  while(1){
    if(alloc3_desc(idx) == 0) {
      break;
    }
    sleep(&disk.free[0], &disk.vdisk_lock);
  }

  // format the three descriptors.
  // qemu's virtio-blk.c reads them.

  struct virtio_blk_req *buf0 = &disk.ops[idx[0]];

  if(write)
    buf0->type = VIRTIO_BLK_T_OUT; // write the disk
  else
    buf0->type = VIRTIO_BLK_T_IN; // read the disk
  buf0->reserved = 0;
  buf0->sector = sector;


```



```
  disk.desc[idx[0]].addr = (uint64) buf0;
  disk.desc[idx[0]].len = sizeof(struct virtio_blk_req);
  disk.desc[idx[0]].flags = VRING_DESC_F_NEXT;
  disk.desc[idx[0]].next = idx[1];

  disk.desc[idx[1]].addr = (uint64) b->data;
  disk.desc[idx[1]].len = BSIZE;
  if(write)
    disk.desc[idx[1]].flags = 0; // device reads b->data
  else
    disk.desc[idx[1]].flags = VRING_DESC_F_WRITE; // device writes b->data
  disk.desc[idx[1]].flags |= VRING_DESC_F_NEXT;
  disk.desc[idx[1]].next = idx[2];

  disk.info[idx[0]].status = 0xff; // device writes 0 on success
  disk.desc[idx[2]].addr = (uint64) &disk.info[idx[0]].status;
  disk.desc[idx[2]].len = 1;
  disk.desc[idx[2]].flags = VRING_DESC_F_WRITE; // device writes the status
  disk.desc[idx[2]].next = 0;

  // record struct buf for virtio_disk_intr().
  b->disk = 1;
  disk.info[idx[0]].b = b;

  // tell the device the first index in our chain of descriptors.
  disk.avail->ring[disk.avail->idx % NUM] = idx[0];

  __sync_synchronize();

  // tell the device another avail ring entry is available.
  disk.avail->idx += 1; // not % NUM ...

  __sync_synchronize();

  *R(VIRTIO_MMIO_QUEUE_NOTIFY) = 0; // value is queue number

  // Wait for virtio_disk_intr() to say request has finished.
  while(b->disk == 1) {
    sleep(b, &disk.vdisk_lock);
  }

  disk.info[idx[0]].b = 0;
  free_chain(idx[0]);

  release(&disk.vdisk_lock);
}


```

```
7700 void
7701 virtio_disk_intr()
7702 {
7703   acquire(&disk.vdisk_lock);
7704
7705   // the device won't raise another interrupt until we tell it
7706   // we've seen this interrupt, which the following line does.
7707   // this may race with the device writing new entries to
7708   // the "used" ring, in which case we may process the new
7709   // completion entries in this interrupt, and have nothing to do
7710   // in the next interrupt, which is harmless.
7711   *R(VIRTIO_MMIO_INTERRUPT_ACK) = *R(VIRTIO_MMIO_INTERRUPT_STATUS) & 0x3;
7712
7713   __sync_synchronize();
7714
7715   // the device increments disk.used->idx when it
7716   // adds an entry to the used ring.
7717
7718   while(disk.used_idx != disk.used->idx){
7719     __sync_synchronize();
7720     int id = disk.used->ring[disk.used_idx % NUM].id;
7721
7722     if(disk.info[id].status != 0)
7723       panic("virtio_disk_intr status");
7724
7725     struct buf *b = disk.info[id].b;
7726     b->disk = 0;   // disk is done with buf
7727     wakeup(b);
7728
7729     disk.used_idx += 1;
7730   }
7731
7732   release(&disk.vdisk_lock);
7733 }
7734
7735
7736
7737
7738
7739
7740
7741
7742
7743
7744
7745
7746
7747
7748
7749
```



```
7750 // init: The initial user-level program
7751
7752 #include "kernel/types.h"
7753 #include "kernel/stat.h"
7754 #include "kernel/spinlock.h"
7755 #include "kernel/sleeplock.h"
7756 #include "kernel/fs.h"
7757 #include "kernel/file.h"
7758 #include "user/user.h"
7759 #include "kernel/fcntl.h"
7760
7761 char *argv[] = { "sh", 0 };
7762
7763 int
7764 main(void)
7765 {
7766   int pid, wpid;
7767
7768   if(open("console", O_RDWR) < 0){
7769     mknod("console", CONSOLE, 0);
7770     open("console", O_RDWR);
7771   }
7772   dup(0);  // stdout
7773   dup(0);  // stderr
7774
7775   for(;;){
7776     printf("init: starting sh\n");
7777     pid = fork();
7778     if(pid < 0){
7779       printf("init: fork failed\n");
7780       exit(1);
7781     }
7782     if(pid == 0){
7783       exec("sh", argv);
7784       printf("init: exec sh failed\n");
7785       exit(1);
7786     }
7787
7788     for(;;){
7789       // this call to wait() returns if the shell exits,
7790       // or if a parentless process exits.
7791       wpid = wait((int *) 0);
7792       if(wpid == pid){
7793         // the shell exited; restart it.
7794         break;
7795       } else if(wpid < 0){
7796         printf("init: wait returned an error\n");
7797         exit(1);
7798       } else {
7799         // it was a parentless process; do nothing.
```

```
7800       }
7801     }
7802   }
7803 }
7804
7805
7806
7807
7808
7809
7810
7811
7812
7813
7814
7815
7816
7817
7818
7819
7820
7821
7822
7823
7824
7825
7826
7827
7828
7829
7830
7831
7832
7833
7834
7835
7836
7837
7838
7839
7840
7841
7842
7843
7844
7845
7846
7847
7848
7849
```



```
7850 // Shell.
7851
7852 #include "kernel/types.h"
7853 #include "user/user.h"
7854 #include "kernel/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 void runcmd(struct cmd*) __attribute__((noreturn));
7904
7905 // Execute cmd.  Never returns.
7906 void
7907 runcmd(struct cmd *cmd)
7908 {
7909   int p[2];
7910   struct backcmd *bcmd;
7911   struct execcmd *ecmd;
7912   struct listcmd *lcmd;
7913   struct pipecmd *pcmd;
7914   struct redircmd *rcmd;
7915
7916   if(cmd == 0)
7917     exit(1);
7918
7919   switch(cmd->type){
7920   default:
7921     panic("runcmd");
7922
7923   case EXEC:
7924     ecmd = (struct execcmd*)cmd;
7925     if(ecmd->argv[0] == 0)
7926       exit(1);
7927     exec(ecmd->argv[0], ecmd->argv);
7928     fprintf(2, "exec %s failed\n", ecmd->argv[0]);
7929     break;
7930
7931   case REDIR:
7932     rcmd = (struct redircmd*)cmd;
7933     close(rcmd->fd);
7934     if(open(rcmd->file, rcmd->mode) < 0){
7935       fprintf(2, "open %s failed\n", rcmd->file);
7936       exit(1);
7937     }
7938     runcmd(rcmd->cmd);
7939     break;
7940
7941   case LIST:
7942     lcmd = (struct listcmd*)cmd;
7943     if(fork1() == 0)
7944       runcmd(lcmd->left);
7945     wait(0);
7946     runcmd(lcmd->right);
7947     break;
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(0);
7971     wait(0);
7972     break;
7973
7974   case BACK:
7975     bcmd = (struct backcmd*)cmd;
7976     if(fork1() == 0)
7977       runcmd(bcmd->cmd);
7978     break;
7979   }
7980   exit(0);
7981 }
7982
7983 int
7984 getcmd(char *buf, int nbuf)
7985 {
7986   write(2, "$ ", 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   // Ensure that three file descriptors are 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     char *cmd = buf;
8017     while (*cmd == ' ' || *cmd == '\t')
8018       cmd++;
8019     if (*cmd == '\n') // is a blank command
8020       continue;
8021     if(cmd[0] == 'c' && cmd[1] == 'd' && cmd[2] == ' '){
8022       // Chdir must be called by the parent, not the child.
8023       cmd[strlen(cmd)-1] = 0;  // chop \n
8024       if(chdir(cmd+3) < 0)
8025         fprintf(2, "cannot cd %s\n", cmd+3);
8026     } else {
8027       if(fork1() == 0)
8028         runcmd(parsecmd(cmd));
8029       wait(0);
8030     }
8031   }
8032   exit(0);
8033 }
8034
8035 void
8036 panic(char *s)
8037 {
8038   fprintf(2, "%s\n", s);
8039   exit(1);
8040 }
8041
8042
8043
8044
8045
8046
8047
8048
8049
```



```
8050 int
8051 fork1(void)
8052 {
8053   int pid;
8054
8055   pid = fork();
8056   if(pid == -1)
8057     panic("fork");
8058   return pid;
8059 }
8060
8061
8062
8063
8064
8065
8066
8067
8068
8069
8070
8071
8072
8073
8074
8075
8076
8077
8078
8079
8080
8081
8082
8083
8084
8085
8086
8087
8088
8089
8090
8091
8092
8093
8094
8095
8096
8097
8098
8099
```

```
8100 // Constructors
8101
8102 struct cmd*
8103 execcmd(void)
8104 {
8105   struct execcmd *cmd;
8106
8107   cmd = malloc(sizeof(*cmd));
8108   memset(cmd, 0, sizeof(*cmd));
8109   cmd->type = EXEC;
8110   return (struct cmd*)cmd;
8111 }
8112
8113 struct cmd*
8114 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
8115 {
8116   struct redircmd *cmd;
8117
8118   cmd = malloc(sizeof(*cmd));
8119   memset(cmd, 0, sizeof(*cmd));
8120   cmd->type = REDIR;
8121   cmd->cmd = subcmd;
8122   cmd->file = file;
8123   cmd->efile = efile;
8124   cmd->mode = mode;
8125   cmd->fd = fd;
8126   return (struct cmd*)cmd;
8127 }
8128
8129 struct cmd*
8130 pipecmd(struct cmd *left, struct cmd *right)
8131 {
8132   struct pipecmd *cmd;
8133
8134   cmd = malloc(sizeof(*cmd));
8135   memset(cmd, 0, sizeof(*cmd));
8136   cmd->type = PIPE;
8137   cmd->left = left;
8138   cmd->right = right;
8139   return (struct cmd*)cmd;
8140 }
8141
8142
8143
8144
8145
8146
8147
8148
8149
```



```
8150 struct cmd*
8151 listcmd(struct cmd *left, struct cmd *right)
8152 {
8153   struct listcmd *cmd;
8154
8155   cmd = malloc(sizeof(*cmd));
8156   memset(cmd, 0, sizeof(*cmd));
8157   cmd->type = LIST;
8158   cmd->left = left;
8159   cmd->right = right;
8160   return (struct cmd*)cmd;
8161 }
8162
8163 struct cmd*
8164 backcmd(struct cmd *subcmd)
8165 {
8166   struct backcmd *cmd;
8167
8168   cmd = malloc(sizeof(*cmd));
8169   memset(cmd, 0, sizeof(*cmd));
8170   cmd->type = BACK;
8171   cmd->cmd = subcmd;
8172   return (struct cmd*)cmd;
8173 }
8174
8175
8176
8177
8178
8179
8180
8181
8182
8183
8184
8185
8186
8187
8188
8189
8190
8191
8192
8193
8194
8195
8196
8197
8198
8199
```

```
8200 // Parsing
8201
8202 char whitespace[] = " \t\r\n\v";
8203 char symbols[] = "<|>&;()";
8204
8205 int
8206 gettoken(char **ps, char *es, char **q, char **eq)
8207 {
8208   char *s;
8209   int ret;
8210
8211   s = *ps;
8212   while(s < es && strchr(whitespace, *s))
8213     s++;
8214   if(q)
8215     *q = s;
8216   ret = *s;
8217   switch(*s){
8218   case 0:
8219     break;
8220   case '|':
8221   case '(':
8222   case ')':
8223   case ';':
8224   case '&':
8225   case '<':
8226     s++;
8227     break;
8228   case '>':
8229     s++;
8230     if(*s == '>'){
8231       ret = '+';
8232       s++;
8233     }
8234     break;
8235   default:
8236     ret = 'a';
8237     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
8238       s++;
8239     break;
8240   }
8241   if(eq)
8242     *eq = s;
8243
8244   while(s < es && strchr(whitespace, *s))
8245     s++;
8246   *ps = s;
8247   return ret;
8248 }
8249
```



```
8250 int
8251 peek(char **ps, char *es, char *toks)
8252 {
8253   char *s;
8254
8255   s = *ps;
8256   while(s < es && strchr(whitespace, *s))
8257     s++;
8258   *ps = s;
8259   return *s && strchr(toks, *s);
8260 }
8261
8262 struct cmd *parseline(char**, char*);
8263 struct cmd *parsepipe(char**, char*);
8264 struct cmd *parseexec(char**, char*);
8265 struct cmd *nulterminate(struct cmd*);
8266
8267 struct cmd*
8268 parsecmd(char *s)
8269 {
8270   char *es;
8271   struct cmd *cmd;
8272
8273   es = s + strlen(s);
8274   cmd = parseline(&s, es);
8275   peek(&s, es, "");
8276   if(s != es){
8277     fprintf(2, "leftovers: %s\n", s);
8278     panic("syntax");
8279   }
8280   nulterminate(cmd);
8281   return cmd;
8282 }
8283
8284 struct cmd*
8285 parseline(char **ps, char *es)
8286 {
8287   struct cmd *cmd;
8288
8289   cmd = parsepipe(ps, es);
8290   while(peek(ps, es, "&")){
8291     gettoken(ps, es, 0, 0);
8292     cmd = backcmd(cmd);
8293   }
8294   if(peek(ps, es, ";")){
8295     gettoken(ps, es, 0, 0);
8296     cmd = listcmd(cmd, parseline(ps, es));
8297   }
8298   return cmd;
8299 }
```

```
8300 struct cmd*
8301 parsepipe(char **ps, char *es)
8302 {
8303   struct cmd *cmd;
8304
8305   cmd = parseexec(ps, es);
8306   if(peek(ps, es, "|")){
8307     gettoken(ps, es, 0, 0);
8308     cmd = pipecmd(cmd, parsepipe(ps, es));
8309   }
8310   return cmd;
8311 }
8312
8313 struct cmd*
8314 parseredirs(struct cmd *cmd, char **ps, char *es)
8315 {
8316   int tok;
8317   char *q, *eq;
8318
8319   while(peek(ps, es, "<>")){
8320     tok = gettoken(ps, es, 0, 0);
8321     if(gettoken(ps, es, &q, &eq) != 'a')
8322       panic("missing file for redirection");
8323     switch(tok){
8324     case '<':
8325       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
8326       break;
8327     case '>':
8328       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE|O_TRUNC, 1);
8329       break;
8330     case '+':  // >>
8331       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
8332       break;
8333     }
8334   }
8335   return cmd;
8336 }
8337
8338
8339
8340
8341
8342
8343
8344
8345
8346
8347
8348
8349
```



```
8350 struct cmd*
8351 parseblock(char **ps, char *es)
8352 {
8353   struct cmd *cmd;
8354
8355   if(!peek(ps, es, "("))
8356     panic("parseblock");
8357   gettoken(ps, es, 0, 0);
8358   cmd = parseline(ps, es);
8359   if(!peek(ps, es, ")"))
8360     panic("syntax - missing )");
8361   gettoken(ps, es, 0, 0);
8362   cmd = parseredirs(cmd, ps, es);
8363   return cmd;
8364 }
8365
8366 struct cmd*
8367 parseexec(char **ps, char *es)
8368 {
8369   char *q, *eq;
8370   int tok, argc;
8371   struct execcmd *cmd;
8372   struct cmd *ret;
8373
8374   if(peek(ps, es, "("))
8375     return parseblock(ps, es);
8376
8377   ret = execcmd();
8378   cmd = (struct execcmd*)ret;
8379
8380   argc = 0;
8381   ret = parseredirs(ret, ps, es);
8382   while(!peek(ps, es, "|)&;")){
8383     if((tok=gettoken(ps, es, &q, &eq)) == 0)
8384       break;
8385     if(tok != 'a')
8386       panic("syntax");
8387     cmd->argv[argc] = q;
8388     cmd->eargv[argc] = eq;
8389     argc++;
8390     if(argc >= MAXARGS)
8391       panic("too many args");
8392     ret = parseredirs(ret, ps, es);
8393   }
8394   cmd->argv[argc] = 0;
8395   cmd->eargv[argc] = 0;
8396   return ret;
8397 }
8398
8399
```

```
8400 // NUL-terminate all the counted strings.
8401 struct cmd*
8402 nulterminate(struct cmd *cmd)
8403 {
8404   int i;
8405   struct backcmd *bcmd;
8406   struct execcmd *ecmd;
8407   struct listcmd *lcmd;
8408   struct pipecmd *pcmd;
8409   struct redircmd *rcmd;
8410
8411   if(cmd == 0)
8412     return 0;
8413
8414   switch(cmd->type){
8415   case EXEC:
8416     ecmd = (struct execcmd*)cmd;
8417     for(i=0; ecmd->argv[i]; i++)
8418       *ecmd->eargv[i] = 0;
8419     break;
8420
8421   case REDIR:
8422     rcmd = (struct redircmd*)cmd;
8423     nulterminate(rcmd->cmd);
8424     *rcmd->efile = 0;
8425     break;
8426
8427   case PIPE:
8428     pcmd = (struct pipecmd*)cmd;
8429     nulterminate(pcmd->left);
8430     nulterminate(pcmd->right);
8431     break;
8432
8433   case LIST:
8434     lcmd = (struct listcmd*)cmd;
8435     nulterminate(lcmd->left);
8436     nulterminate(lcmd->right);
8437     break;
8438
8439   case BACK:
8440     bcmd = (struct backcmd*)cmd;
8441     nulterminate(bcmd->cmd);
8442     break;
8443   }
8444   return cmd;
8445 }
8446
8447
8448
8449
```



```
8450 OUTPUT_ARCH( "riscv" )
8451 ENTRY( _entry )
8452
8453 SECTIONS
8454 {
8455   /*
8456    * ensure that entry.S / _entry is at 0x80000000,
8457    * where qemu's -kernel jumps.
8458    */
8459   . = 0x80000000;
8460
8461   .text : {
8462     kernel/entry.o(_entry)
8463     *(.text .text.*)
8464     . = ALIGN(0x1000);
8465     _trampoline = .;
8466     *(trampsec)
8467     . = ALIGN(0x1000);
8468     ASSERT(. - _trampoline == 0x1000, "error: trampoline larger than one pag
8469     PROVIDE(etext = .);
8470   }
8471
8472   .rodata : {
8473     . = ALIGN(16);
8474     *(.srodata .srodata.*) /* do not need to distinguish this from .rodata *
8475     . = ALIGN(16);
8476     *(.rodata .rodata.*)
8477   }
8478
8479   .data : {
8480     . = ALIGN(16);
8481     *(.sdata .sdata.*) /* do not need to distinguish this from .data */
8482     . = ALIGN(16);
8483     *(.data .data.*)
8484   }
8485
8486   .bss : {
8487     . = ALIGN(16);
8488     *(.sbss .sbss.*) /* do not need to distinguish this from .bss */
8489     . = ALIGN(16);
8490     *(.bss .bss.*)
8491   }
8492
8493   PROVIDE(end = .);
8494 }
8495
8496
8497
8498
8499
```