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, Bruce Lowekamp, lOstman, 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 6 14:26 2026 blank Page 1

```
0950 typedef unsigned int uint;
0951 typedef unsigned short ushort;
0952 typedef unsigned char uchar;
0953
0954 typedef unsigned char uint8;
0955 typedef unsigned short uint16;
0956 typedef unsigned int uint32;
0957 typedef unsigned long uint64;
0958
0959 typedef uint64 pde_t;
0960
0961
0962
0963
0964
0965
0966
0967
0968
0969
0970
0971
0972
0973
0974
0975
0976
0977
0978
0979
0980
0981
0982
0983
0984
0985
0986
0987
0988
0989
0990
0991
0992
0993
0994
0995
0996
0997
0998
0999
```



```
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 d
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 // core-local interrupt controller (CLINT)
0228 #define CLINT_BASE  0x02000000L
0229 #define CLINT(hart) (CLINT_BASE + (hart) * 4)
0230
0231 // qemu puts platform-level interrupt controller (PLIC) here.
0232 #define PLIC                 0x0c000000L
0233 #define PLIC_PRIORITY        (PLIC + 0x0)
0234 #define PLIC_PENDING         (PLIC + 0x1000)
0235 #define PLIC_SENABLE(hart)   (PLIC + 0x2080 + (hart) * 0x100)
0236 #define PLIC_SPRIORITY(hart) (PLIC + 0x201000 + (hart) * 0x2000)
0237 #define PLIC_SCLAIM(hart)    (PLIC + 0x201004 + (hart) * 0x2000)
0238
0239 // the kernel expects there to be RAM
0240 // for use by the kernel and user pages
0241 // from physical address 0x80000000 to PHYSTOP.
0242 #define KERNBASE 0x80000000L
0243 #define PHYSTOP  (KERNBASE + 128 * 1024 * 1024)
0244
0245 // map the trampoline page to the highest address,
0246 // in both user and kernel space.
0247 #define TRAMPOLINE (MAXVA - PGSIZE)
0248
0249
```



```
0250 // map kernel stacks beneath the trampoline,
0251 // each surrounded by invalid guard pages.
0252 #define KSTACK(p) (TRAMPOLINE - ((p) + 1) * 2 * PGSIZE)
0253
0254 // User memory layout.
0255 // Address zero first:
0256 //   text
0257 //   original data and bss
0258 //   fixed-size stack
0259 //   expandable heap
0260 //   ...
0261 //   TRAPFRAME (p->trapframe, used by the trampoline)
0262 //   TRAMPOLINE (the same page as in the kernel)
0263 #define TRAPFRAME (TRAMPOLINE - PGSIZE)
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 // clang-format off
0301 struct buf;
0302 struct context;
0303 struct file;
0304 struct inode;
0305 struct pipe;
0306 struct proc;
0307 struct spinlock;
0308 struct sleeplock;
0309 struct stat;
0310 struct superblock;
0311
0312 // bio.c
0313 void            binit(void);
0314 struct buf*     bread(uint, uint);
0315 void            brelse(struct buf*);
0316 void            bwrite(struct buf*);
0317 void            bpin(struct buf*);
0318 void            bunpin(struct buf*);
0319
0320 // console.c
0321 void            consoleinit(void);
0322 void            consoleintr(int);
0323 void            consputc(int);
0324
0325 // exec.c
0326 int             kexec(char*, char**);
0327
0328 // file.c
0329 struct file*    filealloc(void);
0330 void            fileclose(struct file*);
0331 struct file*    filedup(struct file*);
0332 void            fileinit(void);
0333 int             fileread(struct file*, uint64, int n);
0334 int             filestat(struct file*, uint64 addr);
0335 int             filewrite(struct file*, uint64, int n);
0336
0337 // fs.c
0338 void            fsinit(int);
0339 int             dirlink(struct inode*, char*, uint);
0340 struct inode*   dirlookup(struct inode*, char*, uint*);
0341 struct inode*   ialloc(uint, short);
0342 struct inode*   idup(struct inode*);
0343 void            iinit();
0344 void            ilock(struct inode*);
0345 void            iput(struct inode*);
0346 void            iunlock(struct inode*);
0347 void            iunlockput(struct inode*);
0348 void            iupdate(struct inode*);
0349 int             namecmp(const char*, const char*);
```



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

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



```
0450 // uart.c
0451 void            uartinit(void);
0452 void            uartintr(void);
0453 void            uartwrite(char [], int);
0454 void            uartputc_sync(int);
0455
0456 // vm.c
0457 void            kvminit(void);
0458 void            kvminithart(void);
0459 void            kvmmap(pagetable_t, uint64, uint64, uint64, int);
0460 int             mappages(pagetable_t, uint64, uint64, uint64, int);
0461 pagetable_t     uvmcreate(void);
0462 uint64          uvmalloc(pagetable_t, uint64, uint64, int);
0463 uint64          uvmdealloc(pagetable_t, uint64, uint64);
0464 int             uvmcopy(pagetable_t, pagetable_t, uint64);
0465 void            uvmfree(pagetable_t, uint64);
0466 void            uvmunmap(pagetable_t, uint64, uint64, int);
0467 void            uvmclear(pagetable_t, uint64);
0468 pte_t *         walk(pagetable_t, uint64, int);
0469 uint64          walkaddr(pagetable_t, uint64);
0470 int             copyout(pagetable_t, uint64, char *, uint64);
0471 int             copyin(pagetable_t, char *, uint64, uint64);
0472 int             copyinstr(pagetable_t, char *, uint64, uint64);
0473 int             ismapped(pagetable_t, uint64);
0474 uint64          vmfault(pagetable_t, uint64, int);
0475
0476 // plic.c
0477 void            plicinit(void);
0478 void            plicinithart(void);
0479 int             plic_claim(void);
0480 void            plic_complete(int);
0481
0482 // virtio_disk.c
0483 void            virtio_disk_init(void);
0484 void            virtio_disk_rw(struct buf *, int);
0485 void            virtio_disk_intr(void);
0486
0487 // number of elements in fixed-size array
0488 #define NELEM(x) (sizeof(x) / sizeof((x)[0]))
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 static inline void
0573 s_sstatus(uint64 x)
0574 {
0575   __asm__ __volatile__("csrs sstatus, %0" ::"rK"(x) : "memory");
0576 }
0577
0578 static inline void
0579 c_sstatus(uint64 x)
0580 {
0581   __asm__ __volatile__("csrc sstatus, %0" ::"rK"(x) : "memory");
0582 }
0583
0584 static inline uint64
0585 rc_sstatus(uint64 x)
0586 {
0587   __asm__ __volatile__("csrrc %0, sstatus, %1" : "=r"(x) : "rK"(x) : "memory
0588   return x;
0589 }
0590
0591 // Supervisor Interrupt Pending
0592 static inline uint64
0593 r_sip()
0594 {
0595   uint64 x;
0596   asm volatile("csrr %0, sip" : "=r"(x));
0597   return x;
0598 }
0599
```

```
0600 static inline void
0601 w_sip(uint64 x)
0602 {
0603   asm volatile("csrw sip, %0" : : "r"(x));
0604 }
0605
0606 // Supervisor Interrupt Enable
0607 #define SIE_SEIE (1L << 9) // external
0608 #define SIE_STIE (1L << 5) // timer
0609 static inline uint64
0610 r_sie()
0611 {
0612   uint64 x;
0613   asm volatile("csrr %0, sie" : "=r"(x));
0614   return x;
0615 }
0616
0617 static inline void
0618 w_sie(uint64 x)
0619 {
0620   asm volatile("csrw sie, %0" : : "r"(x));
0621 }
0622
0623 // Machine-mode Interrupt Enable
0624 #define MIE_STIE (1L << 5) // supervisor timer
0625 static inline uint64
0626 r_mie()
0627 {
0628   uint64 x;
0629   asm volatile("csrr %0, mie" : "=r"(x));
0630   return x;
0631 }
0632
0633 static inline void
0634 w_mie(uint64 x)
0635 {
0636   asm volatile("csrw mie, %0" : : "r"(x));
0637 }
0638
0639 // supervisor exception program counter, holds the
0640 // instruction address to which a return from
0641 // exception will go.
0642 static inline void
0643 w_sepc(uint64 x)
0644 {
0645   asm volatile("csrw sepc, %0" : : "r"(x));
0646 }
0647
0648
0649
```



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

```
0700 static inline uint64
0701 r_stvec()
0702 {
0703   uint64 x;
0704   asm volatile("csrr %0, stvec" : "=r"(x));
0705   return x;
0706 }
0707
0708 // Supervisor Timer Comparison Register
0709 static inline uint64
0710 r_stimecmp()
0711 {
0712   uint64 x;
0713   // asm volatile("csrr %0, stimecmp" : "=r" (x) );
0714   asm volatile("csrr %0, 0x14d" : "=r"(x));
0715   return x;
0716 }
0717
0718 static inline void
0719 w_stimecmp(uint64 x)
0720 {
0721   // asm volatile("csrw stimecmp, %0" : : "r" (x));
0722   asm volatile("csrw 0x14d, %0" : : "r"(x));
0723 }
0724
0725 // Machine Environment Configuration Register
0726
0727 #define MENVCFG_STCE (1L << 63)
0728 #define MENVCFG_ADUE (1L << 61)
0729
0730 static inline uint64
0731 r_menvcfg()
0732 {
0733   uint64 x;
0734   // asm volatile("csrr %0, menvcfg" : "=r" (x) );
0735   asm volatile("csrr %0, 0x30a" : "=r"(x));
0736   return x;
0737 }
0738
0739 static inline void
0740 w_menvcfg(uint64 x)
0741 {
0742   // asm volatile("csrw menvcfg, %0" : : "r" (x));
0743   asm volatile("csrw 0x30a, %0" : : "r"(x));
0744 }
0745
0746
0747
0748
0749
```



```
0750 // Physical Memory Protection
0751 static inline void
0752 w_pmpcfg0(uint64 x)
0753 {
0754   asm volatile("csrw pmpcfg0, %0" : : "r"(x));
0755 }
0756
0757 static inline void
0758 w_pmpaddr0(uint64 x)
0759 {
0760   asm volatile("csrw pmpaddr0, %0" : : "r"(x));
0761 }
0762
0763 // use riscv's sv39 page table scheme.
0764 #define SATP_SV39 (8L << 60)
0765
0766 #define MAKE_SATP(pagetable) (SATP_SV39 | (((uint64)pagetable) >> 12))
0767
0768 // supervisor address translation and protection;
0769 // holds the address of the page table.
0770 static inline void
0771 w_satp(uint64 x)
0772 {
0773   asm volatile("csrw satp, %0" : : "r"(x));
0774 }
0775
0776 static inline uint64
0777 r_satp()
0778 {
0779   uint64 x;
0780   asm volatile("csrr %0, satp" : "=r"(x));
0781   return x;
0782 }
0783
0784 // Supervisor Trap Cause
0785 static inline uint64
0786 r_scause()
0787 {
0788   uint64 x;
0789   asm volatile("csrr %0, scause" : "=r"(x));
0790   return x;
0791 }
0792
0793
0794
0795
0796
0797
0798
0799
```

```
0800 // Supervisor Trap Value
0801 static inline uint64
0802 r_stval()
0803 {
0804   uint64 x;
0805   asm volatile("csrr %0, stval" : "=r"(x));
0806   return x;
0807 }
0808
0809 // Machine-mode Counter-Enable
0810 static inline void
0811 w_mcounteren(uint64 x)
0812 {
0813   asm volatile("csrw mcounteren, %0" : : "r"(x));
0814 }
0815
0816 static inline uint64
0817 r_mcounteren()
0818 {
0819   uint64 x;
0820   asm volatile("csrr %0, mcounteren" : "=r"(x));
0821   return x;
0822 }
0823
0824 // machine-mode cycle counter
0825 static inline uint64
0826 r_time()
0827 {
0828   uint64 x;
0829   asm volatile("csrr %0, time" : "=r"(x));
0830   return x;
0831 }
0832
0833 // enable device interrupts
0834 static inline void
0835 intr_on()
0836 {
0837   s_sstatus(SSTATUS_SIE);
0838 }
0839
0840 // disable device interrupts
0841 static inline void
0842 intr_off()
0843 {
0844   c_sstatus(SSTATUS_SIE);
0845 }
0846
0847
0848
0849
```



```
0850 // are device interrupts enabled?
0851 static inline int
0852 intr_get()
0853 {
0854   uint64 x = r_sstatus();
0855   return (x & SSTATUS_SIE) != 0;
0856 }
0857
0858 static inline uint64
0859 r_sp()
0860 {
0861   uint64 x;
0862   asm volatile("mv %0, sp" : "=r"(x));
0863   return x;
0864 }
0865
0866 // read and write tp, the thread pointer, which xv6 uses to hold
0867 // this core's hartid (core number), the index into cpus[].
0868 static inline uint64
0869 r_tp()
0870 {
0871   uint64 x;
0872   asm volatile("mv %0, tp" : "=r"(x));
0873   return x;
0874 }
0875
0876 static inline void
0877 w_tp(uint64 x)
0878 {
0879   asm volatile("mv tp, %0" : : "r"(x));
0880 }
0881
0882 static inline uint64
0883 r_ra()
0884 {
0885   uint64 x;
0886   asm volatile("mv %0, ra" : "=r"(x));
0887   return x;
0888 }
0889
0890 // flush the TLB.
0891 static inline void
0892 sfence_vma()
0893 {
0894   // the zero, zero means flush all TLB entries.
0895   asm volatile("sfence.vma zero, zero" ::: "memory");
0896 }
0897
0898
0899
```

```
0900 // fence for memory-mapped IO
0901 static inline void
0902 io_fence()
0903 {
0904   asm volatile("fence iorw, iorw" ::: "memory");
0905 }
0906
0907 // fence for icache
0908 static inline void
0909 icache_fence()
0910 {
0911   asm volatile("fence.i" ::: "memory");
0912 }
0913
0914 typedef uint64 pte_t;
0915 typedef uint64 *pagetable_t; // 512 PTEs
0916
0917 #endif // __ASSEMBLER__
0918
0919 #define PGSIZE  4096 // bytes per page
0920 #define PGSHIFT 12   // bits of offset within a page
0921
0922 #define PGROUNDUP(sz)  (((sz) + PGSIZE - 1) & ~(PGSIZE - 1))
0923 #define PGROUNDDOWN(a) (((a)) & ~(PGSIZE - 1))
0924
0925 #define PTE_V (1L << 0) // valid
0926 #define PTE_R (1L << 1)
0927 #define PTE_W (1L << 2)
0928 #define PTE_X (1L << 3)
0929 #define PTE_U (1L << 4) // user can access
0930
0931 // shift a physical address to the right place for a PTE.
0932 #define PA2PTE(pa) ((((uint64)pa) >> 12) << 10)
0933
0934 #define PTE2PA(pte) (((pte) >> 10) << 12)
0935
0936 #define PTE_FLAGS(pte) ((pte) & 0x3FF)
0937
0938 // extract the three 9-bit page table indices from a virtual address.
0939 #define PXMASK          0x1FF // 9 bits
0940 #define PXSHIFT(level) (PGSHIFT + (9 * (level)))
0941 #define PX(level, va)  ((((uint64)(va)) >> PXSHIFT(level)) & PXMASK)
0942
0943 // one beyond the highest possible virtual address.
0944 // MAXVA is actually one bit less than the max allowed by
0945 // Sv39, to avoid having to sign-extend virtual addresses
0946 // that have the high bit set.
0947 #define MAXVA (1L << (9 + 9 + 9 + 12 - 1))
0948
0949
```



```
0950 typedef unsigned int uint;
0951 typedef unsigned short ushort;
0952 typedef unsigned char uchar;
0953
0954 typedef unsigned char uint8;
0955 typedef unsigned short uint16;
0956 typedef unsigned int uint32;
0957 typedef unsigned long uint64;
0958
0959 typedef uint64 pde_t;
0960
0961
0962
0963
0964
0965
0966
0967
0968
0969
0970
0971
0972
0973
0974
0975
0976
0977
0978
0979
0980
0981
0982
0983
0984
0985
0986
0987
0988
0989
0990
0991
0992
0993
0994
0995
0996
0997
0998
0999
```

```
1000 // Format of an ELF executable file
1001
1002 #define ELF_MAGIC 0x464C457FU // "\x7FELF" in little endian
1003
1004 // File header
1005 struct elfhdr {
1006   uint magic; // must equal ELF_MAGIC
1007   uchar elf[12];
1008   ushort type;
1009   ushort machine;
1010   uint version;
1011   uint64 entry;
1012   uint64 phoff;
1013   uint64 shoff;
1014   uint flags;
1015   ushort ehsize;
1016   ushort phentsize;
1017   ushort phnum;
1018   ushort shentsize;
1019   ushort shnum;
1020   ushort shstrndx;
1021 };
1022
1023 // Program section header
1024 struct proghdr {
1025   uint32 type;
1026   uint32 flags;
1027   uint64 off;
1028   uint64 vaddr;
1029   uint64 paddr;
1030   uint64 filesz;
1031   uint64 memsz;
1032   uint64 align;
1033 };
1034
1035 // Values for Proghdr type
1036 #define ELF_PROG_LOAD 1
1037
1038 // Flag bits for Proghdr flags
1039 #define ELF_PROG_FLAG_EXEC  1
1040 #define ELF_PROG_FLAG_WRITE 2
1041 #define ELF_PROG_FLAG_READ  4
1042
1043
1044
1045
1046
1047
1048
1049
```



```
1050         # qemu -kernel loads the kernel at 0x80000000
1051         # and causes each hart (i.e. CPU) to jump there.
1052         # kernel.ld causes the following code to
1053         # be placed at 0x80000000.
1054 .section .text
1055 .global _entry
1056 _entry:
1057         # set up a stack for C.
1058         # stack0 is declared in start.c,
1059         # with a 4096-byte stack per CPU.
1060         # sp = stack0 + ((hartid + 1) * 4096)
1061         la sp, stack0
1062         li a0, 1024*4
1063         csrr a1, mhartid
1064         addi a1, a1, 1
1065         mul a0, a0, a1
1066         add sp, sp, a0
1067         # jump to start() in start.c
1068         call start
1069 spin:
1070         j spin
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
```

```
#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);

  // enable hardware updates of page table A and D bits
  w_menvcfg(r_menvcfg() | MENVCFG_ADUE);

  // 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 the sstc extension (i.e. stimecmp).
  w_menvcfg(r_menvcfg() | MENVCFG_STCE);

  // 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();
    printkinit();
    printk("\n");
    printk("xv6 kernel is booting\n");
    printk("\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

    __atomic_store_n(&started, 1, __ATOMIC_RELEASE);
  } else {
    while (__atomic_load_n(&started, __ATOMIC_ACQUIRE) == 0)
      ;

    printk("hart %d starting\n", cpuid());
    kvminithart();  // turn on paging
    trapinithart(); // install kernel trap vector
    plicinithart(); // ask PLIC for device interrupts
  }

  scheduler();
}
```



```
// Mutual exclusion lock.
struct spinlock {
  uint locked; // Is the lock held?

  // For debugging:
  char *name;      // Name of lock.
  struct cpu *cpu; // The cpu holding the lock.
};
```

```
// Mutual exclusion spin locks.

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

void
initlock(struct spinlock *lk, char *name)
{
  lk->name = name;
  lk->locked = 0;
  lk->cpu = 0;
}

// Acquire the lock.
// Loops (spins) until the lock is acquired.
void
acquire(struct spinlock *lk)
{
  push_off(); // disable interrupts to avoid deadlock.
  if (holding(lk))
    panic("acquire");

  // On RISC-V, __atomic_exchange_n turns into an atomic swap:
  //   a5 = 1
  //   s1 = &lk->locked
  //   amoswap.w.aq a5, a5, (s1)
  //
  // Passing __ATOMIC_ACQUIRE to __atomic_exchange_n tells
  // the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen strictly after the lock is acquired.
  while (__atomic_exchange_n(&lk->locked, 1, __ATOMIC_ACQUIRE) != 0)
    ;

  // Record info about lock acquisition for holding() and debugging.
  lk->cpu = mycpu();
}
```



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

  lk->cpu = 0;

  // 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, __atomic_store_n turns into a single atomic store:
  //   s1 = &lk->locked
  //   sw zero,0(s1)
  //
  // The __ATOMIC_RELEASE argument to __atomic_store_n tells the
  // 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 generates a fence instruction before the store:
  //   fence rw,w
  __atomic_store_n(&lk->locked, 0, __ATOMIC_RELEASE);

  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)
{
  // disable interrupts to prevent an involuntary context
  // switch while using mycpu().
  uint64 flags = rc_sstatus(SSTATUS_SIE);
  int old = !!(flags & SSTATUS_SIE);

  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 | PTE_W);

  // 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);
```

```
1500   return kpgtbl;
1501 }
1502
1503 // add a mapping to the kernel page table.
1504 // only used when booting.
1505 // does not flush TLB or enable paging.
1506 void
1507 kvmmap(pagetable_t kpgtbl, uint64 va, uint64 pa, uint64 sz, int perm)
1508 {
1509   if (mappages(kpgtbl, va, sz, pa, perm) != 0)
1510     panic("kvmmap");
1511 }
1512
1513 // Initialize the kernel_pagetable, shared by all CPUs.
1514 void
1515 kvminit(void)
1516 {
1517   kernel_pagetable = kvmmake();
1518 }
1519
1520 // Switch the current CPU's h/w page table register to
1521 // the kernel's page table, and enable paging.
1522 void
1523 kvminithart()
1524 {
1525   // wait for any previous writes to the page table memory to finish.
1526   sfence_vma();
1527
1528   w_satp(MAKE_SATP(kernel_pagetable));
1529
1530   // flush stale entries from the TLB.
1531   sfence_vma();
1532 }
1533
1534 // Return the address of the PTE in page table pagetable
1535 // that corresponds to virtual address va.  If alloc!=0,
1536 // create any required page-table pages.
1537 //
1538 // The risc-v Sv39 scheme has three levels of page-table
1539 // pages. A page-table page contains 512 64-bit PTEs.
1540 // A 64-bit virtual address is split into five fields:
1541 //   39..63 -- must be zero.
1542 //   30..38 -- 9 bits of level-2 index.
1543 //   21..29 -- 9 bits of level-1 index.
1544 //   12..20 -- 9 bits of level-0 index.
1545 //    0..11 -- 12 bits of byte offset within the page.
1546 pte_t *
1547 walk(pagetable_t pagetable, uint64 va, int alloc)
1548 {
1549   if (va >= MAXVA)
```



```
1550     panic("walk");
1551
1552   for (int level = 2; level > 0; level--) {
1553     pte_t *pte = &pagetable[PX(level, va)];
1554     if (*pte & PTE_V) {
1555       pagetable = (pagetable_t)PTE2PA(*pte);
1556     } else {
1557       if (!alloc || (pagetable = (pde_t *)kalloc()) == 0)
1558         return 0;
1559       memset(pagetable, 0, PGSIZE);
1560       *pte = PA2PTE(pagetable) | PTE_V;
1561     }
1562   }
1563   return &pagetable[PX(0, va)];
1564 }
1565
1566 // Look up a virtual address, return the physical address,
1567 // or 0 if not mapped.
1568 // Can only be used to look up user pages.
1569 uint64
1570 walkaddr(pagetable_t pagetable, uint64 va)
1571 {
1572   pte_t *pte;
1573   uint64 pa;
1574
1575   if (va >= MAXVA)
1576     return 0;
1577
1578   pte = walk(pagetable, va, 0);
1579   if (pte == 0)
1580     return 0;
1581   if ((*pte & PTE_V) == 0)
1582     return 0;
1583   if ((*pte & PTE_U) == 0)
1584     return 0;
1585   pa = PTE2PA(*pte);
1586   return pa;
1587 }
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
```

```
1600 // Create PTEs for virtual addresses starting at va that refer to
1601 // physical addresses starting at pa.
1602 // va and size MUST be page-aligned.
1603 // Returns 0 on success, -1 if walk() couldn't
1604 // allocate a needed page-table page.
1605 int
1606 mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)
1607 {
1608   uint64 a, last;
1609   pte_t *pte;
1610
1611   if ((va % PGSIZE) != 0)
1612     panic("mappages: va not aligned");
1613
1614   if ((size % PGSIZE) != 0)
1615     panic("mappages: size not aligned");
1616
1617   if (size == 0)
1618     panic("mappages: size");
1619
1620   a = va;
1621   last = va + size - PGSIZE;
1622   for (;;) {
1623     if ((pte = walk(pagetable, a, 1)) == 0)
1624       return -1;
1625     if (*pte & PTE_V)
1626       panic("mappages: remap");
1627     *pte = PA2PTE(pa) | perm | PTE_V;
1628     if (a == last)
1629       break;
1630     a += PGSIZE;
1631     pa += PGSIZE;
1632   }
1633   return 0;
1634 }
1635
1636 // create an empty user page table.
1637 // returns 0 if out of memory.
1638 pagetable_t
1639 uvmcreate()
1640 {
1641   pagetable_t pagetable;
1642   pagetable = (pagetable_t)kalloc();
1643   if (pagetable == 0)
1644     return 0;
1645   memset(pagetable, 0, PGSIZE);
1646   return pagetable;
1647 }
1648
1649
```



```
1650 // Remove npages of mappings starting from va. va must be
1651 // page-aligned. It's OK if the mappings don't exist.
1652 // Optionally free the physical memory.
1653 void
1654 uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
1655 {
1656   uint64 a;
1657   pte_t *pte;
1658
1659   if ((va % PGSIZE) != 0)
1660     panic("uvmunmap: not aligned");
1661
1662   for (a = va; a < va + npages * PGSIZE; a += PGSIZE) {
1663     if ((pte = walk(pagetable, a, 0)) == 0) // leaf page table entry allocat
1664       continue;
1665     if ((*pte & PTE_V) == 0) // has physical page been allocated?
1666       continue;
1667     if (do_free) {
1668       uint64 pa = PTE2PA(*pte);
1669       kfree((void *)pa);
1670     }
1671     *pte = 0;
1672   }
1673 }
1674
1675 // Allocate PTEs and physical memory to grow a process from oldsz to
1676 // newsz, which need not be page aligned.  Returns new size or 0 on error.
1677 uint64
1678 uvmalloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz, int xperm)
1679 {
1680   char *mem;
1681   uint64 a;
1682
1683   if (newsz < oldsz)
1684     return oldsz;
1685
1686   oldsz = PGROUNDUP(oldsz);
1687   for (a = oldsz; a < newsz; a += PGSIZE) {
1688     mem = kalloc();
1689     if (mem == 0) {
1690       uvmdealloc(pagetable, a, oldsz);
1691       return 0;
1692     }
1693     memset(mem, 0, PGSIZE);
1694     if (mappages(pagetable, a, PGSIZE, (uint64)mem, PTE_R | PTE_U | xperm) !
1695         0) {
1696       kfree(mem);
1697       uvmdealloc(pagetable, a, oldsz);
1698       return 0;
1699     }
```

```
1700   }
1701   return newsz;
1702 }
1703
1704 // Deallocate user pages to bring the process size from oldsz to
1705 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
1706 // need to be less than oldsz.  oldsz can be larger than the actual
1707 // process size.  Returns the new process size.
1708 uint64
1709 uvmdealloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz)
1710 {
1711   if (newsz >= oldsz)
1712     return oldsz;
1713
1714   if (PGROUNDUP(newsz) < PGROUNDUP(oldsz)) {
1715     int npages = (PGROUNDUP(oldsz) - PGROUNDUP(newsz)) / PGSIZE;
1716     uvmunmap(pagetable, PGROUNDUP(newsz), npages, 1);
1717   }
1718
1719   return newsz;
1720 }
1721
1722 // Recursively free page-table pages.
1723 // All leaf mappings must already have been removed.
1724 void
1725 freewalk(pagetable_t pagetable)
1726 {
1727   // there are 2^9 = 512 PTEs in a page table.
1728   for (int i = 0; i < 512; i++) {
1729     pte_t pte = pagetable[i];
1730     if ((pte & PTE_V) && (pte & (PTE_R | PTE_W | PTE_X)) == 0) {
1731       // this PTE points to a lower-level page table.
1732       uint64 child = PTE2PA(pte);
1733       freewalk((pagetable_t)child);
1734       pagetable[i] = 0;
1735     } else if (pte & PTE_V) {
1736       panic("freewalk: leaf");
1737     }
1738   }
1739   kfree((void *)pagetable);
1740 }
1741
1742
1743
1744
1745
1746
1747
1748
1749
```



```
1750 // Free user memory pages,
1751 // then free page-table pages.
1752 void
1753 uvmfree(pagetable_t pagetable, uint64 sz)
1754 {
1755   if (sz > 0)
1756     uvmunmap(pagetable, 0, PGROUNDUP(sz) / PGSIZE, 1);
1757   freewalk(pagetable);
1758 }
1759
1760 // Given a parent process's page table, copy
1761 // its memory into a child's page table.
1762 // Copies both the page table and the
1763 // physical memory.
1764 // returns 0 on success, -1 on failure.
1765 // frees any allocated pages on failure.
1766 int
1767 uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
1768 {
1769   pte_t *pte;
1770   uint64 pa, i;
1771   uint flags;
1772   char *mem;
1773
1774   for (i = 0; i < sz; i += PGSIZE) {
1775     if ((pte = walk(old, i, 0)) == 0)
1776       continue; // page table entry hasn't been allocated
1777     if ((*pte & PTE_V) == 0)
1778       continue; // physical page hasn't been allocated
1779     pa = PTE2PA(*pte);
1780     flags = PTE_FLAGS(*pte);
1781     if ((mem = kalloc()) == 0)
1782       goto err;
1783     memmove(mem, (char *)pa, PGSIZE);
1784     if (mappages(new, i, PGSIZE, (uint64)mem, flags) != 0) {
1785       kfree(mem);
1786       goto err;
1787     }
1788   }
1789   return 0;
1790
1791 err:
1792   uvmunmap(new, 0, i / PGSIZE, 1);
1793   return -1;
1794 }
1795
1796
1797
1798
1799
```

```
1800 // mark a PTE invalid for user access.
1801 // used by exec for the user stack guard page.
1802 void
1803 uvmclear(pagetable_t pagetable, uint64 va)
1804 {
1805   pte_t *pte;
1806
1807   pte = walk(pagetable, va, 0);
1808   if (pte == 0)
1809     panic("uvmclear");
1810   *pte &= ~PTE_U;
1811 }
1812
1813 // Copy from kernel to user.
1814 // Copy len bytes from src to virtual address dstva in a given page table.
1815 // Return 0 on success, -1 on error.
1816 int
1817 copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
1818 {
1819   uint64 n, va0, pa0;
1820   pte_t *pte;
1821
1822   while (len > 0) {
1823     va0 = PGROUNDDOWN(dstva);
1824     if (va0 >= MAXVA)
1825       return -1;
1826
1827     pa0 = walkaddr(pagetable, va0);
1828     if (pa0 == 0) {
1829       if ((pa0 = vmfault(pagetable, va0, 0)) == 0) {
1830         return -1;
1831       }
1832     }
1833
1834     pte = walk(pagetable, va0, 0);
1835     // forbid copyout over read-only user text pages.
1836     if ((*pte & PTE_W) == 0)
1837       return -1;
1838
1839     n = PGSIZE - (dstva - va0);
1840     if (n > len)
1841       n = len;
1842     memmove((void *)(pa0 + (dstva - va0)), src, n);
1843
1844     len -= n;
1845     src += n;
1846     dstva = va0 + PGSIZE;
1847   }
1848   return 0;
1849 }
```



```
1850 // Copy from user to kernel.
1851 // Copy len bytes to dst from virtual address srcva in a given page table.
1852 // Return 0 on success, -1 on error.
1853 int
1854 copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
1855 {
1856   uint64 n, va0, pa0;
1857
1858   while (len > 0) {
1859     va0 = PGROUNDDOWN(srcva);
1860     pa0 = walkaddr(pagetable, va0);
1861     if (pa0 == 0) {
1862       if ((pa0 = vmfault(pagetable, va0, 1)) == 0) {
1863         return -1;
1864       }
1865     }
1866     n = PGSIZE - (srcva - va0);
1867     if (n > len)
1868       n = len;
1869     memmove(dst, (void *)(pa0 + (srcva - va0)), n);
1870
1871     len -= n;
1872     dst += n;
1873     srcva = va0 + PGSIZE;
1874   }
1875   return 0;
1876 }
1877
1878 // Copy a null-terminated string from user to kernel.
1879 // Copy bytes to dst from virtual address srcva in a given page table,
1880 // until a '\0', or max.
1881 // Return 0 on success, -1 on error.
1882 int
1883 copyinstr(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max)
1884 {
1885   uint64 n, va0, pa0;
1886   int got_null = 0;
1887
1888   while (got_null == 0 && max > 0) {
1889     va0 = PGROUNDDOWN(srcva);
1890     pa0 = walkaddr(pagetable, va0);
1891     if (pa0 == 0)
1892       return -1;
1893     n = PGSIZE - (srcva - va0);
1894     if (n > max)
1895       n = max;
1896
1897
1898
1899
```

```
    char *p = (char *)(pa0 + (srcva - va0));
    while (n > 0) {
      if (*p == '\0') {
        *dst = '\0';
        got_null = 1;
        break;
      } else {
        *dst = *p;
      }
      --n;
      --max;
      p++;
      dst++;
    }

    srcva = va0 + PGSIZE;
  }
  if (got_null) {
    return 0;
  } else {
    return -1;
  }
}

// allocate and map user memory if process is referencing a page
// that was lazily allocated in sys_sbrk().
// returns 0 if va is invalid or already mapped, or if
// out of physical memory, and physical address if successful.
uint64
vmfault(pagetable_t pagetable, uint64 va, int read)
{
  uint64 mem;
  struct proc *p = myproc();

  if (va >= p->sz)
    return 0;
  va = PGROUNDDOWN(va);
  if (ismapped(pagetable, va)) {
    return 0;
  }
  mem = (uint64)kalloc();
  if (mem == 0)
    return 0;
  memset((void *)mem, 0, PGSIZE);
  if (mappages(p->pagetable, va, PGSIZE, mem, PTE_W | PTE_U | PTE_R) != 0) {
    kfree((void *)mem);
    return 0;
  }
  return mem;
}
```



```
int
ismapped(pagetable_t pagetable, uint64 va)
{
  pte_t *pte = walk(pagetable, va, 0);
  if (pte == 0) {
    return 0;
  }
  if (*pte & PTE_V) {
    return 1;
  }
  return 0;
}
```

```
2000 // Saved registers for kernel context switches.
2001 struct context {
2002   uint64 ra;
2003   uint64 sp;
2004
2005   // callee-saved
2006   uint64 s0;
2007   uint64 s1;
2008   uint64 s2;
2009   uint64 s3;
2010   uint64 s4;
2011   uint64 s5;
2012   uint64 s6;
2013   uint64 s7;
2014   uint64 s8;
2015   uint64 s9;
2016   uint64 s10;
2017   uint64 s11;
2018 };
2019
2020 // Per-CPU state.
2021 struct cpu {
2022   struct proc *proc;      // The process running on this cpu, or null.
2023   struct context context; // swtch() here to enter scheduler().
2024   int noff;               // Depth of push_off() nesting.
2025   int intena;             // Were interrupts enabled before push_off()?
2026 };
2027
2028 extern struct cpu cpus[NCPU];
2029
2030 // per-process data for the trap handling code in trampoline.S.
2031 // sits in a page by itself just under the trampoline page in the
2032 // user page table. not specially mapped in the kernel page table.
2033 // uservec in trampoline.S saves user registers in the trapframe,
2034 // then initializes registers from the trapframe's
2035 // kernel_sp, kernel_hartid, kernel_satp, and jumps to kernel_trap.
2036 // prepare_return() and userret in trampoline.S set up
2037 // the trapframe's kernel_*, restore user registers from the
2038 // trapframe, switch to the user page table, and enter user space.
2039 struct trapframe {
2040   /*   0 */ uint64 kernel_satp;   // kernel page table
2041   /*   8 */ uint64 kernel_sp;     // top of process's kernel stack
2042   /*  16 */ uint64 kernel_trap;   // usertrap()
2043   /*  24 */ uint64 epc;           // saved user program counter
2044   /*  32 */ uint64 kernel_hartid; // saved kernel tp
2045   /*  40 */ uint64 ra;
2046   /*  48 */ uint64 sp;
2047   /*  56 */ uint64 gp;
2048   /*  64 */ uint64 tp;
2049   /*  72 */ uint64 t0;
```



```
2050   /*  80 */ uint64 t1;
2051   /*  88 */ uint64 t2;
2052   /*  96 */ uint64 s0;
2053   /* 104 */ uint64 s1;
2054   /* 112 */ uint64 a0;
2055   /* 120 */ uint64 a1;
2056   /* 128 */ uint64 a2;
2057   /* 136 */ uint64 a3;
2058   /* 144 */ uint64 a4;
2059   /* 152 */ uint64 a5;
2060   /* 160 */ uint64 a6;
2061   /* 168 */ uint64 a7;
2062   /* 176 */ uint64 s2;
2063   /* 184 */ uint64 s3;
2064   /* 192 */ uint64 s4;
2065   /* 200 */ uint64 s5;
2066   /* 208 */ uint64 s6;
2067   /* 216 */ uint64 s7;
2068   /* 224 */ uint64 s8;
2069   /* 232 */ uint64 s9;
2070   /* 240 */ uint64 s10;
2071   /* 248 */ uint64 s11;
2072   /* 256 */ uint64 t3;
2073   /* 264 */ uint64 t4;
2074   /* 272 */ uint64 t5;
2075   /* 280 */ uint64 t6;
2076 };
2077
2078 enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
2079
2080 // Per-process state
2081 struct proc {
2082   struct spinlock lock;
2083
2084   // p->lock must be held when using these:
2085   enum procstate state; // Process state
2086   void *chan;           // If non-zero, sleeping on chan
2087   int killed;           // If non-zero, have been killed
2088   int xstate;           // Exit status to be returned to parent's wait
2089   int pid;              // Process ID
2090
2091   // wait_lock must be held when using this:
2092   struct proc *parent; // Parent process
2093
2094   // these are private to the process, so p->lock need not be held.
2095   uint64 kstack;               // Virtual address of kernel stack
2096   uint64 sz;                   // Size of process memory (bytes)
2097   pagetable_t pagetable;       // User page table
2098   struct trapframe *trapframe; // data page for trampoline.S
2099   struct context context;      // swtch() here to run process
```

```
2100   struct file *ofile[NOFILE];  // Open files
2101   struct inode *cwd;           // Current directory
2102   char name[16];               // Process name (debugging)
2103 };
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
```



```
2150 #include "types.h"
2151 #include "param.h"
2152 #include "memlayout.h"
2153 #include "riscv.h"
2154 #include "spinlock.h"
2155 #include "proc.h"
2156 #include "defs.h"
2157
2158 struct cpu cpus[NCPU];
2159
2160 struct proc proc[NPROC];
2161
2162 struct proc *initproc;
2163
2164 int nextpid = 1;
2165 struct spinlock pid_lock;
2166
2167 extern void forkret(void);
2168 static void freeproc(struct proc *p);
2169
2170 extern char trampoline[]; // trampoline.S
2171
2172 // helps ensure that wakeups of wait()ing
2173 // parents are not lost. helps obey the
2174 // memory model when using p->parent.
2175 // must be acquired before any p->lock.
2176 struct spinlock wait_lock;
2177
2178 // Allocate a page for each process's kernel stack.
2179 // Map it high in memory, followed by an invalid
2180 // guard page.
2181 void
2182 proc_mapstacks(pagetable_t kpgtbl)
2183 {
2184   struct proc *p;
2185
2186   for (p = proc; p < &proc[NPROC]; p++) {
2187     char *pa = kalloc();
2188     if (pa == 0)
2189       panic("kalloc");
2190     uint64 va = KSTACK((int)(p - proc));
2191     kvmmap(kpgtbl, va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
2192   }
2193 }
2194
2195
2196
2197
2198
2199
```

```
2200 // initialize the proc table.
2201 void
2202 procinit(void)
2203 {
2204   struct proc *p;
2205
2206   initlock(&pid_lock, "nextpid");
2207   initlock(&wait_lock, "wait_lock");
2208   for (p = proc; p < &proc[NPROC]; p++) {
2209     initlock(&p->lock, "proc");
2210     p->state = UNUSED;
2211     p->kstack = KSTACK((int)(p - proc));
2212   }
2213 }
2214
2215 // Must be called with interrupts disabled,
2216 // to prevent race with process being moved
2217 // to a different CPU.
2218 int
2219 cpuid()
2220 {
2221   int id = r_tp();
2222   return id;
2223 }
2224
2225 // Return this CPU's cpu struct.
2226 // Interrupts must be disabled.
2227 struct cpu *
2228 mycpu(void)
2229 {
2230   int id = cpuid();
2231   struct cpu *c = &cpus[id];
2232   return c;
2233 }
2234
2235 // Return the current struct proc *, or zero if none.
2236 struct proc *
2237 myproc(void)
2238 {
2239   push_off();
2240   struct cpu *c = mycpu();
2241   struct proc *p = c->proc;
2242   pop_off();
2243   return p;
2244 }
2245
2246
2247
2248
2249
```



```
2250 int
2251 allocpid()
2252 {
2253   int pid;
2254
2255   acquire(&pid_lock);
2256   pid = nextpid;
2257   nextpid = nextpid + 1;
2258   release(&pid_lock);
2259
2260   return pid;
2261 }
2262
2263 // Look in the process table for an UNUSED proc.
2264 // If found, initialize state required to run in the kernel,
2265 // and return with p->lock held.
2266 // If there are no free procs, or a memory allocation fails, return 0.
2267 static struct proc *
2268 allocproc(void)
2269 {
2270   struct proc *p;
2271
2272   for (p = proc; p < &proc[NPROC]; p++) {
2273     acquire(&p->lock);
2274     if (p->state == UNUSED) {
2275       goto found;
2276     } else {
2277       release(&p->lock);
2278     }
2279   }
2280   return 0;
2281
2282 found:
2283   p->pid = allocpid();
2284   p->state = USED;
2285
2286   // Allocate a trapframe page.
2287   if ((p->trapframe = (struct trapframe *)kalloc()) == 0) {
2288     freeproc(p);
2289     release(&p->lock);
2290     return 0;
2291   }
2292
2293   // An empty user page table.
2294   p->pagetable = proc_pagetable(p);
2295   if (p->pagetable == 0) {
2296     freeproc(p);
2297     release(&p->lock);
2298     return 0;
2299   }
```

```
2300   // Set up new context to start executing at forkret,
2301   // which returns to user space.
2302   memset(&p->context, 0, sizeof(p->context));
2303   p->context.ra = (uint64)forkret;
2304   p->context.sp = p->kstack + PGSIZE;
2305
2306   return p;
2307 }
2308
2309 // free a proc structure and the data hanging from it,
2310 // including user pages.
2311 // p->lock must be held.
2312 static void
2313 freeproc(struct proc *p)
2314 {
2315   if (p->trapframe)
2316     kfree((void *)p->trapframe);
2317   p->trapframe = 0;
2318   if (p->pagetable)
2319     proc_freepagetable(p->pagetable, p->sz);
2320   p->pagetable = 0;
2321   p->sz = 0;
2322   p->pid = 0;
2323   p->name[0] = 0;
2324   p->chan = 0;
2325   p->killed = 0;
2326   p->xstate = 0;
2327   p->state = UNUSED;
2328 }
2329
2330 // Create a user page table for a given process, with no user memory,
2331 // but with trampoline and trapframe pages.
2332 pagetable_t
2333 proc_pagetable(struct proc *p)
2334 {
2335   pagetable_t pagetable;
2336
2337   // An empty page table.
2338   pagetable = uvmcreate();
2339   if (pagetable == 0)
2340     return 0;
2341
2342   // map the trampoline code (for system call return)
2343   // at the highest user virtual address.
2344   // only the supervisor uses it, on the way
2345   // to/from user space, so not PTE_U.
2346   if (mappages(pagetable, TRAMPOLINE, PGSIZE, (uint64)trampoline,
2347                PTE_R | PTE_X) < 0) {
2348     uvmfree(pagetable, 0);
2349     return 0;
```



```
2350   }
2351
2352   // map the trapframe page just below the trampoline page, for
2353   // trampoline.S.
2354   if (mappages(pagetable, TRAPFRAME, PGSIZE, (uint64)(p->trapframe),
2355                PTE_R | PTE_W) < 0) {
2356     uvmunmap(pagetable, TRAMPOLINE, 1, 0);
2357     uvmfree(pagetable, 0);
2358     return 0;
2359   }
2360
2361   return pagetable;
2362 }
2363
2364 // Free a process's page table, and free the
2365 // physical memory it refers to.
2366 void
2367 proc_freepagetable(pagetable_t pagetable, uint64 sz)
2368 {
2369   uvmunmap(pagetable, TRAMPOLINE, 1, 0);
2370   uvmunmap(pagetable, TRAPFRAME, 1, 0);
2371   uvmfree(pagetable, sz);
2372 }
2373
2374 // Set up first user process.
2375 void
2376 userinit(void)
2377 {
2378   struct proc *p;
2379
2380   p = allocproc();
2381   initproc = p;
2382
2383   p->cwd = namei("/");
2384
2385   p->state = RUNNABLE;
2386
2387   release(&p->lock);
2388 }
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
```

```
2400 // Grow or shrink user memory by n bytes.
2401 // Return 0 on success, -1 on failure.
2402 int
2403 growproc(int n)
2404 {
2405   uint64 sz;
2406   struct proc *p = myproc();
2407
2408   sz = p->sz;
2409   if (n > 0) {
2410     if (sz + n > TRAPFRAME) {
2411       return -1;
2412     }
2413     if ((sz = uvmalloc(p->pagetable, sz, sz + n, PTE_W)) == 0) {
2414       return -1;
2415     }
2416   } else if (n < 0) {
2417     sz = uvmdealloc(p->pagetable, sz, sz + n);
2418   }
2419   p->sz = sz;
2420   return 0;
2421 }
2422
2423 // Create a new process, copying the parent.
2424 // Sets up child kernel stack to return as if from fork() system call.
2425 int
2426 kfork(void)
2427 {
2428   int i, pid;
2429   struct proc *np;
2430   struct proc *p = myproc();
2431
2432   // Allocate process.
2433   if ((np = allocproc()) == 0) {
2434     return -1;
2435   }
2436
2437   // Copy user memory from parent to child.
2438   if (uvmcopy(p->pagetable, np->pagetable, p->sz) < 0) {
2439     freeproc(np);
2440     release(&np->lock);
2441     return -1;
2442   }
2443   np->sz = p->sz;
2444
2445   // copy saved user registers.
2446   *(np->trapframe) = *(p->trapframe);
2447
2448   // Cause fork to return 0 in the child.
2449   np->trapframe->a0 = 0;
```



```
2450   // increment reference counts on open file descriptors.
2451   for (i = 0; i < NOFILE; i++)
2452     if (p->ofile[i])
2453       np->ofile[i] = filedup(p->ofile[i]);
2454   np->cwd = idup(p->cwd);
2455
2456   safestrcpy(np->name, p->name, sizeof(p->name));
2457
2458   pid = np->pid;
2459
2460   release(&np->lock);
2461
2462   acquire(&wait_lock);
2463   np->parent = p;
2464   release(&wait_lock);
2465
2466   acquire(&np->lock);
2467   np->state = RUNNABLE;
2468   release(&np->lock);
2469
2470   return pid;
2471 }
2472
2473 // Pass p's abandoned children to init.
2474 // Caller must hold wait_lock.
2475 void
2476 reparent(struct proc *p)
2477 {
2478   struct proc *pp;
2479
2480   for (pp = proc; pp < &proc[NPROC]; pp++) {
2481     if (pp->parent == p) {
2482       pp->parent = initproc;
2483       wakeup(initproc);
2484     }
2485   }
2486 }
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
```

```
2500 // Exit the current process.  Does not return.
2501 // An exited process remains in the zombie state
2502 // until its parent calls wait().
2503 void
2504 kexit(int status)
2505 {
2506   struct proc *p = myproc();
2507
2508   if (p == initproc)
2509     panic("init exiting");
2510
2511   // Close all open files.
2512   for (int fd = 0; fd < NOFILE; fd++) {
2513     if (p->ofile[fd]) {
2514       struct file *f = p->ofile[fd];
2515       fileclose(f);
2516       p->ofile[fd] = 0;
2517     }
2518   }
2519
2520   begin_op();
2521   iput(p->cwd);
2522   end_op();
2523   p->cwd = 0;
2524
2525   acquire(&wait_lock);
2526
2527   // Give any children to init.
2528   reparent(p);
2529
2530   // Parent might be sleeping in wait().
2531   wakeup(p->parent);
2532
2533   acquire(&p->lock);
2534
2535   p->xstate = status;
2536   p->state = ZOMBIE;
2537
2538   release(&wait_lock);
2539
2540   // Jump into the scheduler, never to return.
2541   sched();
2542   panic("zombie exit");
2543 }
2544
2545
2546
2547
2548
2549
```



```
2550 // Wait for a child process to exit and return its pid.
2551 // Return -1 if this process has no children.
2552 int
2553 kwait(uint64 addr)
2554 {
2555   struct proc *pp;
2556   int havekids, pid;
2557   struct proc *p = myproc();
2558
2559   acquire(&wait_lock);
2560
2561   for (;;) {
2562     // Scan through table looking for exited children.
2563     havekids = 0;
2564     for (pp = proc; pp < &proc[NPROC]; pp++) {
2565       if (pp->parent == p) {
2566         // make sure the child isn't still in exit() or swtch().
2567         acquire(&pp->lock);
2568
2569         havekids = 1;
2570         if (pp->state == ZOMBIE) {
2571           // Found one.
2572           pid = pp->pid;
2573           if (addr != 0 && copyout(p->pagetable, addr, (char *)&pp->xstate,
2574                                    sizeof(pp->xstate)) < 0) {
2575             release(&pp->lock);
2576             release(&wait_lock);
2577             return -1;
2578           }
2579           pp->parent = 0;
2580           freeproc(pp);
2581           release(&pp->lock);
2582           release(&wait_lock);
2583           return pid;
2584         }
2585         release(&pp->lock);
2586       }
2587     }
2588
2589     // No point waiting if we don't have any children.
2590     if (!havekids || killed(p)) {
2591       release(&wait_lock);
2592       return -1;
2593     }
2594
2595     // Wait for a child to exit.
2596     sleep(p, &wait_lock);
2597   }
2598 }
2599
```

```
2600 // Per-CPU process scheduler.
2601 // Each CPU calls scheduler() after setting itself up.
2602 // Scheduler never returns.  It loops, doing:
2603 //  - choose a process to run.
2604 //  - swtch to start running that process.
2605 //  - eventually that process transfers control
2606 //    via swtch back to the scheduler.
2607 void
2608 scheduler(void)
2609 {
2610   struct proc *p;
2611   struct cpu *c = mycpu();
2612
2613   c->proc = 0;
2614   for (;;) {
2615     // The most recent process to run may have had interrupts
2616     // turned off; enable them to avoid a deadlock if all
2617     // processes are waiting. Then turn them back off
2618     // to avoid a possible race between an interrupt
2619     // and wfi.
2620     intr_on();
2621     intr_off();
2622
2623     int found = 0;
2624     for (p = proc; p < &proc[NPROC]; p++) {
2625       acquire(&p->lock);
2626       if (p->state == RUNNABLE) {
2627         // Switch to chosen process.  It is the process's job
2628         // to release its lock and then reacquire it
2629         // before jumping back to us.
2630         p->state = RUNNING;
2631         c->proc = p;
2632         swtch(&c->context, &p->context);
2633
2634         // Process is done running for now.
2635         // It should have changed its p->state before coming back.
2636         c->proc = 0;
2637         found = 1;
2638       }
2639       release(&p->lock);
2640     }
2641     if (found == 0) {
2642       // nothing to run; stop running on this core until an interrupt.
2643       asm volatile("wfi");
2644     }
2645   }
2646 }
2647
2648
2649
```



```
2650 // Switch to scheduler.  Must hold only p->lock
2651 // and have changed proc->state. Saves and restores
2652 // intena because intena is a property of this
2653 // kernel thread, not this CPU. It should
2654 // be proc->intena and proc->noff, but that would
2655 // break in the few places where a lock is held but
2656 // there's no process.
2657 void
2658 sched(void)
2659 {
2660   int intena;
2661   struct proc *p = myproc();
2662
2663   if (!holding(&p->lock))
2664     panic("sched p->lock");
2665   if (mycpu()->noff != 1)
2666     panic("sched locks");
2667   if (p->state == RUNNING)
2668     panic("sched RUNNING");
2669   if (intr_get())
2670     panic("sched interruptible");
2671
2672   intena = mycpu()->intena;
2673   swtch(&p->context, &mycpu()->context);
2674   mycpu()->intena = intena;
2675 }
2676
2677 // Give up the CPU for one scheduling round.
2678 void
2679 yield(void)
2680 {
2681   struct proc *p = myproc();
2682   acquire(&p->lock);
2683   p->state = RUNNABLE;
2684   sched();
2685   release(&p->lock);
2686 }
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
```

```
2700 // A fork child's very first scheduling by scheduler()
2701 // will swtch to forkret.
2702 void
2703 forkret(void)
2704 {
2705   extern char userret[];
2706   static int first = 1;
2707   struct proc *p = myproc();
2708
2709   // Still holding p->lock from scheduler.
2710   release(&p->lock);
2711
2712   if (__atomic_load_n(&first, __ATOMIC_ACQUIRE)) {
2713     // File system initialization must be run in the context of a
2714     // regular process (e.g., because it calls sleep), and thus cannot
2715     // be run from main().
2716     fsinit(ROOTDEV);
2717
2718     // ensure other cores see first=0.
2719     __atomic_store_n(&first, 0, __ATOMIC_RELEASE);
2720
2721     // We can invoke kexec() now that file system is initialized.
2722     // Put the return value (argc) of kexec into a0.
2723     p->trapframe->a0 = kexec("/init", (char *[]){"/init", 0});
2724     if (p->trapframe->a0 == -1) {
2725       panic("exec");
2726     }
2727   }
2728
2729   // return to user space, mimicing usertrap()'s return.
2730   prepare_return();
2731   uint64 satp = MAKE_SATP(p->pagetable);
2732   uint64 trampoline_userret = TRAMPOLINE + (userret - trampoline);
2733   ((void (*)(uint64))trampoline_userret)(satp);
2734 }
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
```



```
2750 // Sleep on channel chan, releasing condition lock lk.
2751 // Re-acquires lk when awakened.
2752 void
2753 sleep(void *chan, struct spinlock *lk)
2754 {
2755   struct proc *p = myproc();
2756
2757   // Must acquire p->lock in order to
2758   // change p->state and then call sched.
2759   // Once we hold p->lock, we can be
2760   // guaranteed that we won't miss any wakeup
2761   // (wakeup locks p->lock),
2762   // so it's okay to release lk.
2763
2764   acquire(&p->lock);
2765   release(lk);
2766
2767   // Go to sleep.
2768   p->chan = chan;
2769   p->state = SLEEPING;
2770
2771   sched();
2772
2773   // Tidy up.
2774   p->chan = 0;
2775
2776   // Reacquire original lock.
2777   release(&p->lock);
2778   acquire(lk);
2779 }
2780
2781 // Wake up all processes sleeping on channel chan.
2782 // Caller should hold the condition lock.
2783 void
2784 wakeup(void *chan)
2785 {
2786   struct proc *p;
2787
2788   for (p = proc; p < &proc[NPROC]; p++) {
2789     if (p != myproc()) {
2790       acquire(&p->lock);
2791       if (p->state == SLEEPING && p->chan == chan) {
2792         p->state = RUNNABLE;
2793       }
2794       release(&p->lock);
2795     }
2796   }
2797 }
2798
2799
```

```
2800 // Kill the process with the given pid.
2801 // The victim won't exit until it tries to return
2802 // to user space (see usertrap() in trap.c).
2803 int
2804 kkill(int pid)
2805 {
2806   struct proc *p;
2807
2808   for (p = proc; p < &proc[NPROC]; p++) {
2809     acquire(&p->lock);
2810     if (p->pid == pid) {
2811       p->killed = 1;
2812       if (p->state == SLEEPING) {
2813         // Wake process from sleep().
2814         p->state = RUNNABLE;
2815       }
2816       release(&p->lock);
2817       return 0;
2818     }
2819     release(&p->lock);
2820   }
2821   return -1;
2822 }
2823
2824 void
2825 setkilled(struct proc *p)
2826 {
2827   acquire(&p->lock);
2828   p->killed = 1;
2829   release(&p->lock);
2830 }
2831
2832 int
2833 killed(struct proc *p)
2834 {
2835   int k;
2836
2837   acquire(&p->lock);
2838   k = p->killed;
2839   release(&p->lock);
2840   return k;
2841 }
2842
2843
2844
2845
2846
2847
2848
2849
```



```
2850 // Copy to either a user address, or kernel address,
2851 // depending on usr_dst.
2852 // Returns 0 on success, -1 on error.
2853 int
2854 either_copyout(int user_dst, uint64 dst, void *src, uint64 len)
2855 {
2856   struct proc *p = myproc();
2857   if (user_dst) {
2858     return copyout(p->pagetable, dst, src, len);
2859   } else {
2860     memmove((char *)dst, src, len);
2861     return 0;
2862   }
2863 }
2864
2865 // Copy from either a user address, or kernel address,
2866 // depending on usr_src.
2867 // Returns 0 on success, -1 on error.
2868 int
2869 either_copyin(void *dst, int user_src, uint64 src, uint64 len)
2870 {
2871   struct proc *p = myproc();
2872   if (user_src) {
2873     return copyin(p->pagetable, dst, src, len);
2874   } else {
2875     memmove(dst, (char *)src, len);
2876     return 0;
2877   }
2878 }
2879
2880 // Print a process listing to console.  For debugging.
2881 // Runs when user types ^P on console.
2882 // No lock to avoid wedging a stuck machine further.
2883 void
2884 procdump(void)
2885 {
2886   static char *states[] = {
2887     // clang-format off
2888     [UNUSED]    "unused",
2889     [USED]      "used",
2890     [SLEEPING]  "sleep ",
2891     [RUNNABLE]  "runble",
2892     [RUNNING]   "run   ",
2893     [ZOMBIE]    "zombie"
2894     // clang-format on
2895   };
2896   struct proc *p;
2897   char *state;
2898
2899   printk("\n");
```

```
2900   for (p = proc; p < &proc[NPROC]; p++) {
2901     if (p->state == UNUSED)
2902       continue;
2903     if (p->state >= 0 && p->state < NELEM(states) && states[p->state])
2904       state = states[p->state];
2905     else
2906       state = "???";
2907     printk("%d %s %s", p->pid, state, p->name);
2908     printk("\n");
2909   }
2910 }
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
```



```
2950 # Context switch
2951 #
2952 #   void swtch(struct context *old, struct context *new);
2953 #
2954 # Save current registers in old. Load from new.
2955
2956
2957 .globl swtch
2958 swtch:
2959         sd ra, 0(a0)
2960         sd sp, 8(a0)
2961         sd s0, 16(a0)
2962         sd s1, 24(a0)
2963         sd s2, 32(a0)
2964         sd s3, 40(a0)
2965         sd s4, 48(a0)
2966         sd s5, 56(a0)
2967         sd s6, 64(a0)
2968         sd s7, 72(a0)
2969         sd s8, 80(a0)
2970         sd s9, 88(a0)
2971         sd s10, 96(a0)
2972         sd s11, 104(a0)
2973
2974         ld ra, 0(a1)
2975         ld sp, 8(a1)
2976         ld s0, 16(a1)
2977         ld s1, 24(a1)
2978         ld s2, 32(a1)
2979         ld s3, 40(a1)
2980         ld s4, 48(a1)
2981         ld s5, 56(a1)
2982         ld s6, 64(a1)
2983         ld s7, 72(a1)
2984         ld s8, 80(a1)
2985         ld s9, 88(a1)
2986         ld s10, 96(a1)
2987         ld s11, 104(a1)
2988
2989         ret
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
```

```
3000 // Physical memory allocator, for user processes,
3001 // kernel stacks, page-table pages,
3002 // and pipe buffers. Allocates whole 4096-byte pages.
3003
3004 #include "types.h"
3005 #include "param.h"
3006 #include "memlayout.h"
3007 #include "spinlock.h"
3008 #include "riscv.h"
3009 #include "defs.h"
3010
3011 void freerange(void *pa_start, void *pa_end);
3012
3013 extern char end[]; // first address after kernel.
3014                    // defined by kernel.ld.
3015
3016 struct run {
3017   struct run *next;
3018 };
3019
3020 struct {
3021   struct spinlock lock;
3022   struct run *freelist;
3023 } kmem;
3024
3025 void
3026 kinit()
3027 {
3028   initlock(&kmem.lock, "kmem");
3029   freerange(end, (void *)PHYSTOP);
3030 }
3031
3032 void
3033 freerange(void *pa_start, void *pa_end)
3034 {
3035   char *p;
3036   p = (char *)PGROUNDUP((uint64)pa_start);
3037   for (; p + PGSIZE <= (char *)pa_end; p += PGSIZE)
3038     kfree(p);
3039 }
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
```



```
3050 // Free the page of physical memory pointed at by pa,
3051 // which normally should have been returned by a
3052 // call to kalloc().  (The exception is when
3053 // initializing the allocator; see kinit above.)
3054 void
3055 kfree(void *pa)
3056 {
3057   struct run *r;
3058
3059   if (((uint64)pa % PGSIZE) != 0 || (char *)pa < end || (uint64)pa >= PHYSTO
3060     panic("kfree");
3061
3062   // Fill with junk to catch dangling refs.
3063   memset(pa, 1, PGSIZE);
3064
3065   r = (struct run *)pa;
3066
3067   acquire(&kmem.lock);
3068   r->next = kmem.freelist;
3069   kmem.freelist = r;
3070   release(&kmem.lock);
3071 }
3072
3073 // Allocate one 4096-byte page of physical memory.
3074 // Returns a pointer that the kernel can use.
3075 // Returns 0 if the memory cannot be allocated.
3076 void *
3077 kalloc(void)
3078 {
3079   struct run *r;
3080
3081   acquire(&kmem.lock);
3082   r = kmem.freelist;
3083   if (r)
3084     kmem.freelist = r->next;
3085   release(&kmem.lock);
3086
3087   if (r)
3088     memset((char *)r, 5, PGSIZE); // fill with junk
3089   return (void *)r;
3090 }
3091
3092
3093
3094
3095
3096
3097
3098
3099
```

```
3100         #
3101         # low-level code to handle traps from user space into
3102         # the kernel, and returns from kernel to user.
3103         #
3104         # the kernel maps the page holding this code
3105         # at the same virtual address (TRAMPOLINE)
3106         # in user and kernel space so that it continues
3107         # to work when it switches page tables.
3108         # kernel.ld causes this code to start at
3109         # a page boundary.
3110         #
3111
3112 #include "riscv.h"
3113 #include "memlayout.h"
3114
3115 .section trampsec
3116 .globl trampoline
3117 .globl usertrap
3118 trampoline:
3119 .align 4
3120 .globl uservec
3121 uservec:
3122    #
3123         # trap.c sets stvec to point here, so
3124         # traps from user space start here,
3125         # in supervisor mode, but with a
3126         # user page table.
3127         #
3128
3129         # save user a0 in sscratch so
3130         # a0 can be used to get at TRAPFRAME.
3131         csrw sscratch, a0
3132
3133         # each process has a separate p->trapframe memory area,
3134         # but it's mapped to the same virtual address
3135         # (TRAPFRAME) in every process's user page table.
3136         li a0, TRAPFRAME
3137
3138         # save the user registers in TRAPFRAME
3139         sd ra, 40(a0)
3140         sd sp, 48(a0)
3141         sd gp, 56(a0)
3142         sd tp, 64(a0)
3143         sd t0, 72(a0)
3144         sd t1, 80(a0)
3145         sd t2, 88(a0)
3146         sd s0, 96(a0)
3147         sd s1, 104(a0)
3148         sd a1, 120(a0)
3149         sd a2, 128(a0)
```



```
3150         sd a3, 136(a0)
3151         sd a4, 144(a0)
3152         sd a5, 152(a0)
3153         sd a6, 160(a0)
3154         sd a7, 168(a0)
3155         sd s2, 176(a0)
3156         sd s3, 184(a0)
3157         sd s4, 192(a0)
3158         sd s5, 200(a0)
3159         sd s6, 208(a0)
3160         sd s7, 216(a0)
3161         sd s8, 224(a0)
3162         sd s9, 232(a0)
3163         sd s10, 240(a0)
3164         sd s11, 248(a0)
3165         sd t3, 256(a0)
3166         sd t4, 264(a0)
3167         sd t5, 272(a0)
3168         sd t6, 280(a0)
3169
3170    # save the user a0 in p->trapframe->a0
3171         csrr t0, sscratch
3172         sd t0, 112(a0)
3173
3174         # initialize kernel stack pointer, from p->trapframe->kernel_sp
3175         ld sp, 8(a0)
3176
3177         # make tp hold the current hartid, from p->trapframe->kernel_hartid
3178         ld tp, 32(a0)
3179
3180         # load the address of usertrap(), from p->trapframe->kernel_trap
3181         ld t0, 16(a0)
3182
3183         # fetch the kernel page table address, from p->trapframe->kernel_sat
3184         ld t1, 0(a0)
3185
3186         # wait for any previous memory operations to complete, so that
3187         # they use the user page table.
3188         sfence.vma zero, zero
3189
3190         # install the kernel page table.
3191         csrw satp, t1
3192
3193         # flush now-stale user entries from the TLB.
3194         sfence.vma zero, zero
3195
3196         # call usertrap()
3197         jalr t0
3198
3199
```

```
.globl userret
userret:
        # usertrap() returns here, with user satp in a0.
        # return from kernel to user.

        # flush icache, in case this is the first time
        # we're running this proc on this hart.
        fence.i

        # switch to the user page table.
        sfence.vma zero, zero
        csrw satp, a0
        sfence.vma zero, zero

        li a0, TRAPFRAME

        # restore all but a0 from TRAPFRAME
        ld ra, 40(a0)
        ld sp, 48(a0)
        ld gp, 56(a0)
        ld tp, 64(a0)
        ld t0, 72(a0)
        ld t1, 80(a0)
        ld t2, 88(a0)
        ld s0, 96(a0)
        ld s1, 104(a0)
        ld a1, 120(a0)
        ld a2, 128(a0)
        ld a3, 136(a0)
        ld a4, 144(a0)
        ld a5, 152(a0)
        ld a6, 160(a0)
        ld a7, 168(a0)
        ld s2, 176(a0)
        ld s3, 184(a0)
        ld s4, 192(a0)
        ld s5, 200(a0)
        ld s6, 208(a0)
        ld s7, 216(a0)
        ld s8, 224(a0)
        ld s9, 232(a0)
        ld s10, 240(a0)
        ld s11, 248(a0)
        ld t3, 256(a0)
        ld t4, 264(a0)
        ld t5, 272(a0)
        ld t6, 280(a0)

   # restore user a0
        ld a0, 112(a0)
```



```
        # return to user mode and user pc.
        # prepare_return() sets up sstatus and sepc.
        sret
```

```
        #
        # interrupts and exceptions while in supervisor
        # mode come here.
        #
        # the current stack is a kernel stack.
        # push registers, call kerneltrap().
        # when kerneltrap() returns, restore registers, return.
        #
.globl kerneltrap
.globl kernelvec
.align 4
kernelvec:
        # make room to save registers.
        addi sp, sp, -256

        # save caller-saved registers.
        sd ra, 0(sp)
        # sd sp, 8(sp)
        sd gp, 16(sp)
        # sd tp, 24(sp)
        sd t0, 32(sp)
        sd t1, 40(sp)
        sd t2, 48(sp)
        sd a0, 72(sp)
        sd a1, 80(sp)
        sd a2, 88(sp)
        sd a3, 96(sp)
        sd a4, 104(sp)
        sd a5, 112(sp)
        sd a6, 120(sp)
        sd a7, 128(sp)
        sd t3, 216(sp)
        sd t4, 224(sp)
        sd t5, 232(sp)
        sd t6, 240(sp)

        # call the C trap handler in trap.c
        call kerneltrap

        # restore registers.
        ld ra, 0(sp)
        # ld sp, 8(sp)
        ld gp, 16(sp)
        # not tp (contains hartid), in case we moved CPUs
        ld t0, 32(sp)
        ld t1, 40(sp)
        ld t2, 48(sp)
        ld a0, 72(sp)
        ld a1, 80(sp)
        ld a2, 88(sp)
```



```
        ld a3, 96(sp)
        ld a4, 104(sp)
        ld a5, 112(sp)
        ld a6, 120(sp)
        ld a7, 128(sp)
        ld t3, 216(sp)
        ld t4, 224(sp)
        ld t5, 232(sp)
        ld t6, 240(sp)

        addi sp, sp, 256

        # return to whatever we were doing in the kernel.
        sret
```

```
3400 #include "types.h"
3401 #include "param.h"
3402 #include "memlayout.h"
3403 #include "riscv.h"
3404 #include "spinlock.h"
3405 #include "proc.h"
3406 #include "defs.h"
3407
3408 struct spinlock tickslock;
3409 uint ticks;
3410
3411 extern char trampoline[], uservec[];
3412
3413 // in kernelvec.S, calls kerneltrap().
3414 void kernelvec();
3415
3416 extern int devintr();
3417
3418 void
3419 trapinit(void)
3420 {
3421   initlock(&tickslock, "time");
3422 }
3423
3424 // set up to take exceptions and traps while in the kernel.
3425 void
3426 trapinithart(void)
3427 {
3428   w_stvec((uint64)kernelvec);
3429 }
3430
3431 //
3432 // handle an interrupt, exception, or system call from user space.
3433 // called from, and returns to, trampoline.S
3434 // return value is user satp for trampoline.S to switch to.
3435 //
3436 uint64
3437 usertrap(void)
3438 {
3439   int which_dev = 0;
3440
3441   if ((r_sstatus() & SSTATUS_SPP) != 0)
3442     panic("usertrap: not from user mode");
3443
3444   // send interrupts and exceptions to kerneltrap(),
3445   // since we're now in the kernel.
3446   w_stvec((uint64)kernelvec);
3447
3448   struct proc *p = myproc();
3449
```



```
3450   // save user program counter.
3451   p->trapframe->epc = r_sepc();
3452
3453   if (r_scause() == 8) {
3454     // system call
3455
3456     if (killed(p))
3457       kexit(-1);
3458
3459     // sepc points to the ecall instruction,
3460     // but we want to return to the next instruction.
3461     p->trapframe->epc += 4;
3462
3463     // an interrupt will change sepc, scause, and sstatus,
3464     // so enable only now that we're done with those registers.
3465     intr_on();
3466
3467     syscall();
3468   } else if ((which_dev = devintr()) != 0) {
3469     // ok
3470   } else if ((r_scause() == 15 || r_scause() == 13) &&
3471              vmfault(p->pagetable, r_stval(), (r_scause() == 13) ? 1 : 0) !=
3472                0) {
3473     // page fault on lazily-allocated page
3474   } else {
3475     printk("usertrap(): unexpected scause 0x%lx pid=%d\n", r_scause(), p->pi
3476     printk("            sepc=0x%lx stval=0x%lx\n", r_sepc(), r_stval());
3477     setkilled(p);
3478   }
3479
3480   if (killed(p))
3481     kexit(-1);
3482
3483   // give up the CPU if this is a timer interrupt.
3484   if (which_dev == 2)
3485     yield();
3486
3487   prepare_return();
3488
3489   // the user page table to switch to, for trampoline.S
3490   uint64 satp = MAKE_SATP(p->pagetable);
3491
3492   // return to trampoline.S; satp value in a0.
3493   return satp;
3494 }
3495
3496
3497
3498
3499
```

```
//
// set up trapframe and control registers for a return to user space
//
void
prepare_return(void)
{
  struct proc *p = myproc();

  // we're about to switch the destination of traps from
  // kerneltrap() to usertrap(). because a trap from kernel
  // code to usertrap would be a disaster, turn off interrupts.
  intr_off();

  // send syscalls, interrupts, and exceptions to uservec in trampoline.S
  uint64 trampoline_uservec = TRAMPOLINE + (uservec - trampoline);
  w_stvec(trampoline_uservec);

  // set up trapframe values that uservec will need when
  // the process next traps into the kernel.
  p->trapframe->kernel_satp = r_satp();         // kernel page table
  p->trapframe->kernel_sp = p->kstack + PGSIZE; // process's kernel stack
  p->trapframe->kernel_trap = (uint64)usertrap;
  p->trapframe->kernel_hartid = r_tp(); // hartid for cpuid()

  // set up the registers that trampoline.S's sret will use
  // to get to user space.

  // set S Previous Privilege mode to User.
  unsigned long x = r_sstatus();
  x &= ~SSTATUS_SPP; // clear SPP to 0 for user mode
  x |= SSTATUS_SPIE; // enable interrupts in user mode
  w_sstatus(x);

  // set S Exception Program Counter to the saved user pc.
  w_sepc(p->trapframe->epc);
}
```



```
// interrupts and exceptions from kernel code go here via kernelvec,
// on whatever the current kernel stack is.
void
kerneltrap()
{
  int which_dev = 0;
  uint64 sepc = r_sepc();
  uint64 sstatus = r_sstatus();
  uint64 scause = r_scause();

  if ((sstatus & SSTATUS_SPP) == 0)
    panic("kerneltrap: not from supervisor mode");
  if (intr_get() != 0)
    panic("kerneltrap: interrupts enabled");

  if ((which_dev = devintr()) == 0) {
    // interrupt or trap from an unknown source
    printk("scause=0x%lx sepc=0x%lx stval=0x%lx\n", scause, r_sepc(),
           r_stval());
    panic("kerneltrap");
  }

  // give up the CPU if this is a timer interrupt.
  if (which_dev == 2 && myproc() != 0)
    yield();

  // the yield() may have caused some traps to occur,
  // so restore trap registers for use by kernelvec.S's sepc instruction.
  w_sepc(sepc);
  w_sstatus(sstatus);
}

void
clockintr()
{
  if (cpuid() == 0) {
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }

  // ask for the next timer interrupt. this also clears
  // the interrupt request. 1000000 is about a tenth
  // of a second.
  w_stimecmp(r_time() + 1000000);
}
```

```
3600 // check if it's an external interrupt or software interrupt,
3601 // and handle it.
3602 // returns 2 if timer interrupt,
3603 // 1 if other device,
3604 // 0 if not recognized.
3605 int
3606 devintr()
3607 {
3608   uint64 scause = r_scause();
3609
3610   if (scause == 0x8000000000000009L) {
3611     // this is a supervisor external interrupt, via PLIC.
3612
3613     // irq indicates which device interrupted.
3614     int irq = plic_claim();
3615
3616     if (irq == UART0_IRQ) {
3617       uartintr();
3618     } else if (irq == VIRTIO0_IRQ) {
3619       virtio_disk_intr();
3620     } else if (irq) {
3621       printk("unexpected interrupt irq=%d\n", irq);
3622     }
3623
3624     // the PLIC allows each device to raise at most one
3625     // interrupt at a time; tell the PLIC the device is
3626     // now allowed to interrupt again.
3627     if (irq)
3628       plic_complete(irq);
3629
3630     return 1;
3631   } else if (scause == 0x8000000000000005L) {
3632     // timer interrupt.
3633     clockintr();
3634     return 2;
3635   } else {
3636     return 0;
3637   }
3638 }
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649
```



```
3650 // System call numbers
3651 #define SYS_fork   1
3652 #define SYS_exit   2
3653 #define SYS_wait   3
3654 #define SYS_pipe   4
3655 #define SYS_read   5
3656 #define SYS_kill   6
3657 #define SYS_exec   7
3658 #define SYS_fstat  8
3659 #define SYS_chdir  9
3660 #define SYS_dup    10
3661 #define SYS_getpid 11
3662 #define SYS_sbrk   12
3663 #define SYS_pause  13
3664 #define SYS_uptime 14
3665 #define SYS_open   15
3666 #define SYS_write  16
3667 #define SYS_mknod  17
3668 #define SYS_unlink 18
3669 #define SYS_link   19
3670 #define SYS_mkdir  20
3671 #define SYS_close  21
3672 #define SYS_sync   22
3673
3674
3675
3676
3677
3678
3679
3680
3681
3682
3683
3684
3685
3686
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699
```

```
3700 #include "types.h"
3701 #include "param.h"
3702 #include "memlayout.h"
3703 #include "riscv.h"
3704 #include "spinlock.h"
3705 #include "proc.h"
3706 #include "syscall.h"
3707 #include "defs.h"
3708
3709 // Fetch the uint64 at addr from the current process.
3710 int
3711 fetchaddr(uint64 addr, uint64 *ip)
3712 {
3713   struct proc *p = myproc();
3714   if (addr >= p->sz ||
3715       addr + sizeof(uint64) > p->sz) // both tests needed, in case of overfl
3716     return -1;
3717   if (copyin(p->pagetable, (char *)ip, addr, sizeof(*ip)) != 0)
3718     return -1;
3719   return 0;
3720 }
3721
3722 // Fetch the nul-terminated string at addr from the current process.
3723 // Returns length of string, not including nul, or -1 for error.
3724 int
3725 fetchstr(uint64 addr, char *buf, int max)
3726 {
3727   struct proc *p = myproc();
3728   if (copyinstr(p->pagetable, buf, addr, max) < 0)
3729     return -1;
3730   return strlen(buf);
3731 }
3732
3733 static uint64
3734 argraw(int n)
3735 {
3736   struct proc *p = myproc();
3737   switch (n) {
3738   case 0:
3739     return p->trapframe->a0;
3740   case 1:
3741     return p->trapframe->a1;
3742   case 2:
3743     return p->trapframe->a2;
3744   case 3:
3745     return p->trapframe->a3;
3746   case 4:
3747     return p->trapframe->a4;
3748   case 5:
3749     return p->trapframe->a5;
```



```
3750   }
3751   panic("argraw");
3752   return -1;
3753 }
3754
3755 // Fetch the nth 32-bit system call argument.
3756 void
3757 argint(int n, int *ip)
3758 {
3759   *ip = argraw(n);
3760 }
3761
3762 // Retrieve an argument as a pointer.
3763 // Doesn't check for legality, since
3764 // copyin/copyout will do that.
3765 void
3766 argaddr(int n, uint64 *ip)
3767 {
3768   *ip = argraw(n);
3769 }
3770
3771 // Fetch the nth word-sized system call argument as a null-terminated string
3772 // Copies into buf, at most max.
3773 // Returns string length if OK (not including nul), -1 if error.
3774 int
3775 argstr(int n, char *buf, int max)
3776 {
3777   uint64 addr;
3778   argaddr(n, &addr);
3779   return fetchstr(addr, buf, max);
3780 }
3781
3782 // Prototypes for the functions that handle system calls.
3783 extern uint64 sys_fork(void);
3784 extern uint64 sys_exit(void);
3785 extern uint64 sys_wait(void);
3786 extern uint64 sys_pipe(void);
3787 extern uint64 sys_read(void);
3788 extern uint64 sys_kill(void);
3789 extern uint64 sys_exec(void);
3790 extern uint64 sys_fstat(void);
3791 extern uint64 sys_chdir(void);
3792 extern uint64 sys_dup(void);
3793 extern uint64 sys_getpid(void);
3794 extern uint64 sys_sbrk(void);
3795 extern uint64 sys_pause(void);
3796 extern uint64 sys_uptime(void);
3797 extern uint64 sys_open(void);
3798 extern uint64 sys_write(void);
3799 extern uint64 sys_mknod(void);
```

```
extern uint64 sys_unlink(void);
extern uint64 sys_link(void);
extern uint64 sys_mkdir(void);
extern uint64 sys_close(void);
extern uint64 sys_sync(void);

// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
static uint64 (*syscalls[])(void) = {
  // clang-format off
  [SYS_fork]    sys_fork,
  [SYS_exit]    sys_exit,
  [SYS_wait]    sys_wait,
  [SYS_pipe]    sys_pipe,
  [SYS_read]    sys_read,
  [SYS_kill]    sys_kill,
  [SYS_exec]    sys_exec,
  [SYS_fstat]   sys_fstat,
  [SYS_chdir]   sys_chdir,
  [SYS_dup]     sys_dup,
  [SYS_getpid]  sys_getpid,
  [SYS_sbrk]    sys_sbrk,
  [SYS_pause]   sys_pause,
  [SYS_uptime]  sys_uptime,
  [SYS_open]    sys_open,
  [SYS_write]   sys_write,
  [SYS_mknod]   sys_mknod,
  [SYS_unlink]  sys_unlink,
  [SYS_link]    sys_link,
  [SYS_mkdir]   sys_mkdir,
  [SYS_close]   sys_close,
  [SYS_sync]    sys_sync,
  // clang-format on
};

void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // Use num to lookup the system call function for num, call it,
    // and store its return value in p->trapframe->a0
    p->trapframe->a0 = syscalls[num]();
  } else {
    printk("%d %s: unknown sys call %d\n", p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
```



```
}
```

```
3900 #include "types.h"
3901 #include "riscv.h"
3902 #include "defs.h"
3903 #include "param.h"
3904 #include "memlayout.h"
3905 #include "spinlock.h"
3906 #include "proc.h"
3907 #include "vm.h"
3908
3909 uint64
3910 sys_exit(void)
3911 {
3912   int n;
3913   argint(0, &n);
3914   kexit(n);
3915   return 0; // not reached
3916 }
3917
3918 uint64
3919 sys_getpid(void)
3920 {
3921   return myproc()->pid;
3922 }
3923
3924 uint64
3925 sys_fork(void)
3926 {
3927   return kfork();
3928 }
3929
3930 uint64
3931 sys_wait(void)
3932 {
3933   uint64 p;
3934   argaddr(0, &p);
3935   return kwait(p);
3936 }
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
```



```
3950 uint64
3951 sys_sbrk(void)
3952 {
3953   uint64 addr;
3954   int t;
3955   int n;
3956
3957   argint(0, &n);
3958   argint(1, &t);
3959   addr = myproc()->sz;
3960
3961   if (t == SBRK_EAGER || n < 0) {
3962     if (growproc(n) < 0) {
3963       return -1;
3964     }
3965   } else {
3966     // Lazily allocate memory for this process: increase its memory
3967     // size but don't allocate memory. If the processes uses the
3968     // memory, vmfault() will allocate it.
3969     if (addr + n < addr)
3970       return -1;
3971     if (addr + n > TRAPFRAME)
3972       return -1;
3973     myproc()->sz += n;
3974   }
3975   return addr;
3976 }
3977
3978 uint64
3979 sys_pause(void)
3980 {
3981   int n;
3982   uint ticks0;
3983
3984   argint(0, &n);
3985   if (n < 0)
3986     n = 0;
3987   acquire(&tickslock);
3988   ticks0 = ticks;
3989   while (ticks - ticks0 < n) {
3990     if (killed(myproc())) {
3991       release(&tickslock);
3992       return -1;
3993     }
3994     sleep(&ticks, &tickslock);
3995   }
3996   release(&tickslock);
3997   return 0;
3998 }
3999
```

```
4000 uint64
4001 sys_kill(void)
4002 {
4003   int pid;
4004
4005   argint(0, &pid);
4006   return kkill(pid);
4007 }
4008
4009 // return how many clock tick interrupts have occurred
4010 // since start.
4011 uint64
4012 sys_uptime(void)
4013 {
4014   uint xticks;
4015
4016   acquire(&tickslock);
4017   xticks = ticks;
4018   release(&tickslock);
4019   return xticks;
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 struct buf {
4051   int valid; // has data been read from disk?
4052   int disk;  // does disk "own" buf?
4053   uint dev;
4054   uint blockno;
4055   struct sleeplock lock;
4056   uint refcnt;
4057   struct buf *prev; // LRU cache list
4058   struct buf *next;
4059   uchar data[BSIZE];
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 // Long-term locks for processes
4101 struct sleeplock {
4102   uint locked;        // Is the lock held?
4103   struct spinlock lk; // spinlock protecting this sleep lock
4104
4105   // For debugging:
4106   char *name; // Name of lock.
4107   int pid;    // Process holding lock
4108 };
4109
4110
4111
4112
4113
4114
4115
4116
4117
4118
4119
4120
4121
4122
4123
4124
4125
4126
4127
4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
```



```
4150 #define O_RDONLY 0x000
4151 #define O_WRONLY 0x001
4152 #define O_RDWR   0x002
4153 #define O_CREATE 0x200
4154 #define O_TRUNC  0x400
4155
4156
4157
4158
4159
4160
4161
4162
4163
4164
4165
4166
4167
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 #define T_DIR    1 // Directory
4201 #define T_FILE   2 // File
4202 #define T_DEVICE 3 // Device
4203
4204 struct stat {
4205   int dev;     // File system's disk device
4206   uint ino;    // Inode number
4207   short type;  // Type of file
4208   short nlink; // Number of links to file
4209   uint64 size; // Size of file in bytes
4210 };
4211
4212
4213
4214
4215
4216
4217
4218
4219
4220
4221
4222
4223
4224
4225
4226
4227
4228
4229
4230
4231
4232
4233
4234
4235
4236
4237
4238
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
```



```
4250 // On-disk file system format.
4251 // Both the kernel and user programs use this header file.
4252
4253 #define ROOTINO 1    // root i-number
4254 #define BSIZE   1024 // block size
4255
4256 // Disk layout:
4257 // [ boot block | super block | log | inode blocks |
4258 //                                          free bit map | data blocks]
4259 //
4260 // mkfs computes the super block and builds an initial file system. The
4261 // super block describes the disk layout:
4262 struct superblock {
4263   uint magic;      // Must be FSMAGIC
4264   uint size;       // Size of file system image (blocks)
4265   uint nblocks;    // Number of data blocks
4266   uint ninodes;    // Number of inodes.
4267   uint nlog;       // Number of log blocks
4268   uint logstart;   // Block number of first log block
4269   uint inodestart; // Block number of first inode block
4270   uint bmapstart;  // Block number of first free map block
4271 };
4272
4273 #define FSMAGIC 0x10203040
4274
4275 #define NDIRECT   12
4276 #define NINDIRECT (BSIZE / sizeof(uint))
4277 #define MAXFILE   (NDIRECT + NINDIRECT)
4278
4279 // On-disk inode structure
4280 struct dinode {
4281   short type;              // File type
4282   short major;             // Major device number (T_DEVICE only)
4283   short minor;             // Minor device number (T_DEVICE only)
4284   short nlink;             // Number of links to inode in file system
4285   uint size;               // Size of file (bytes)
4286   uint addrs[NDIRECT + 1]; // Data block addresses
4287 };
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
```

```
4300 // Inodes per block.
4301 #define IPB (BSIZE / sizeof(struct dinode))
4302
4303 // Block containing inode i
4304 #define IBLOCK(i, sb) ((i) / IPB + sb.inodestart)
4305
4306 // Bitmap bits per block
4307 #define BPB (BSIZE * 8)
4308
4309 // Block of free map containing bit for block b
4310 #define BBLOCK(b, sb) ((b) / BPB + sb.bmapstart)
4311
4312 // Directory is a file containing a sequence of dirent structures.
4313 #define DIRSIZ 14
4314
4315 // The name field may have DIRSIZ characters and not end in a NUL
4316 // character.
4317 struct dirent {
4318   ushort inum;
4319   char name[DIRSIZ] __attribute__((nonstring));
4320 };
4321
4322
4323
4324
4325
4326
4327
4328
4329
4330
4331
4332
4333
4334
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
```



```
4350 struct file {
4351   enum { FD_NONE, FD_PIPE, FD_INODE, FD_DEVICE } type;
4352   int ref; // reference count
4353   char readable;
4354   char writable;
4355   struct pipe *pipe; // FD_PIPE
4356   struct inode *ip;  // FD_INODE and FD_DEVICE
4357   uint off;          // FD_INODE
4358   short major;       // FD_DEVICE
4359 };
4360
4361 #define major(dev)  ((dev) >> 16 & 0xFFFF)
4362 #define minor(dev)  ((dev) & 0xFFFF)
4363 #define mkdev(m, n) ((uint)((m) << 16 | (n)))
4364
4365 // in-memory copy of an inode
4366 struct inode {
4367   uint dev;              // Device number
4368   uint inum;             // Inode number
4369   int ref;               // Reference count
4370   struct sleeplock lock; // protects everything below here
4371   int valid;             // inode has been read from disk?
4372
4373   short type; // copy of disk inode
4374   short major;
4375   short minor;
4376   short nlink;
4377   uint size;
4378   uint addrs[NDIRECT + 1];
4379 };
4380
4381 // map major device number to device functions.
4382 struct devsw {
4383   int (*read)(int, uint64, int);
4384   int (*write)(int, uint64, int);
4385 };
4386
4387 extern struct devsw devsw[];
4388
4389 #define CONSOLE 1
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
```

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



```
4450   }
4451 }
4452
4453 // Look through buffer cache for block on device dev.
4454 // If not found, allocate a buffer.
4455 // In either case, return locked buffer.
4456 static struct buf *
4457 bget(uint dev, uint blockno)
4458 {
4459   struct buf *b;
4460
4461   acquire(&bcache.lock);
4462
4463   // Is the block already cached?
4464   for (b = bcache.head.next; b != &bcache.head; b = b->next) {
4465     if (b->dev == dev && b->blockno == blockno) {
4466       b->refcnt++;
4467       release(&bcache.lock);
4468       acquiresleep(&b->lock);
4469       return b;
4470     }
4471   }
4472
4473   // Not cached.
4474   // Recycle the least recently used (LRU) unused buffer.
4475   for (b = bcache.head.prev; b != &bcache.head; b = b->prev) {
4476     if (b->refcnt == 0) {
4477       b->dev = dev;
4478       b->blockno = blockno;
4479       b->valid = 0;
4480       b->refcnt = 1;
4481       release(&bcache.lock);
4482       acquiresleep(&b->lock);
4483       return b;
4484     }
4485   }
4486   panic("bget: no buffers");
4487 }
4488
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4499
```

```
4500 // Return a locked buf with the contents of the indicated block.
4501 struct buf *
4502 bread(uint dev, uint blockno)
4503 {
4504   struct buf *b;
4505
4506   b = bget(dev, blockno);
4507   if (!b->valid) {
4508     virtio_disk_rw(b, 0);
4509     b->valid = 1;
4510   }
4511   return b;
4512 }
4513
4514 // Write b's contents to disk.  Must be locked.
4515 // Only the log calls bwrite.
4516 void
4517 bwrite(struct buf *b)
4518 {
4519   if (!holdingsleep(&b->lock))
4520     panic("bwrite");
4521   virtio_disk_rw(b, 1);
4522 }
4523
4524 // Release a locked buffer.
4525 // Move to the head of the most-recently-used list.
4526 void
4527 brelse(struct buf *b)
4528 {
4529   if (!holdingsleep(&b->lock))
4530     panic("brelse");
4531
4532   releasesleep(&b->lock);
4533
4534   acquire(&bcache.lock);
4535   b->refcnt--;
4536   if (b->refcnt == 0) {
4537     // no one is waiting for it.
4538     b->next->prev = b->prev;
4539     b->prev->next = b->next;
4540     b->next = bcache.head.next;
4541     b->prev = &bcache.head;
4542     bcache.head.next->prev = b;
4543     bcache.head.next = b;
4544   }
4545
4546   release(&bcache.lock);
4547 }
4548
4549
```



```
4550 void
4551 bpin(struct buf *b)
4552 {
4553   acquire(&bcache.lock);
4554   b->refcnt++;
4555   release(&bcache.lock);
4556 }
4557
4558 void
4559 bunpin(struct buf *b)
4560 {
4561   acquire(&bcache.lock);
4562   b->refcnt--;
4563   release(&bcache.lock);
4564 }
4565
4566
4567
4568
4569
4570
4571
4572
4573
4574
4575
4576
4577
4578
4579
4580
4581
4582
4583
4584
4585
4586
4587
4588
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4599
```

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



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

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



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

```
4800 // Read the log header from disk into the in-memory log header
4801 static void
4802 read_head(void)
4803 {
4804   struct buf *buf = bread(log.dev, log.start);
4805   struct logheader *lh = (struct logheader *)(buf->data);
4806   int i;
4807   log.lh.n = lh->n;
4808   for (i = 0; i < log.lh.n; i++) {
4809     log.lh.block[i] = lh->block[i];
4810   }
4811   brelse(buf);
4812 }
4813
4814 // Write in-memory log header to disk.
4815 // This is the true point at which the
4816 // current transaction commits.
4817 static void
4818 write_head(void)
4819 {
4820   struct buf *buf = bread(log.dev, log.start);
4821   struct logheader *hb = (struct logheader *)(buf->data);
4822   int i;
4823   hb->n = log.lh.n;
4824   for (i = 0; i < log.lh.n; i++) {
4825     hb->block[i] = log.lh.block[i];
4826   }
4827   bwrite(buf);
4828   brelse(buf);
4829 }
4830
4831 static void
4832 recover_from_log(void)
4833 {
4834   read_head();
4835   install_trans(1); // if committed, copy from log to disk
4836   log.lh.n = 0;
4837   write_head(); // clear the log
4838 }
4839
4840
4841
4842
4843
4844
4845
4846
4847
4848
4849
```



```
4850 // called at the start of each FS system call.
4851 void
4852 begin_op(void)
4853 {
4854   acquire(&log.lock);
4855   while (1) {
4856     if (log.committing) {
4857       sleep(&log, &log.lock);
4858     } else if (log.lh.n + (log.outstanding + 1) * MAXOPBLOCKS > LOGBLOCKS) {
4859       // this op might exhaust log space; wait for commit.
4860       sleep(&log, &log.lock);
4861     } else {
4862       log.outstanding += 1;
4863       release(&log.lock);
4864       break;
4865     }
4866   }
4867 }
4868
4869 // called at the end of each FS system call.
4870 // commits if this was the last outstanding operation.
4871 void
4872 end_op(void)
4873 {
4874   int do_commit = 0;
4875
4876   acquire(&log.lock);
4877   log.outstanding -= 1;
4878   if (log.committing)
4879     panic("log.committing");
4880   if (log.outstanding == 0) {
4881     do_commit = 1;
4882     log.committing = 1;
4883   } else {
4884     // begin_op() may be waiting for log space,
4885     // and decrementing log.outstanding has decreased
4886     // the amount of reserved space.
4887     wakeup(&log);
4888   }
4889   release(&log.lock);
4890
4891   if (do_commit) {
4892     // call commit w/o holding locks, since not allowed
4893     // to sleep with locks.
4894     commit();
4895     acquire(&log.lock);
4896     log.committing = 0;
4897     log.ncommit += 1;
4898     wakeup(&log);
4899     release(&log.lock);
```

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



```
4950   for (i = 0; i < log.lh.n; i++) {
4951     if (log.lh.block[i] == b->blockno) // log absorption
4952       break;
4953   }
4954   log.lh.block[i] = b->blockno;
4955   if (i == log.lh.n) { // Add new block to log?
4956     bpin(b);
4957     log.lh.n++;
4958   }
4959   release(&log.lock);
4960 }
4961
4962 uint64
4963 sys_sync(void)
4964 {
4965   acquire(&log.lock);
4966   if (log.committing || log.outstanding > 0) {
4967     int n = log.ncommit + 1;
4968     while (log.ncommit < n) {
4969       sleep(&log, &log.lock);
4970     }
4971   }
4972   release(&log.lock);
4973   return 0;
4974 }
4975
4976
4977
4978
4979
4980
4981
4982
4983
4984
4985
4986
4987
4988
4989
4990
4991
4992
4993
4994
4995
4996
4997
4998
4999
```

```
5000 // File system implementation.  Five layers:
5001 //   + Blocks: allocator for raw disk blocks.
5002 //   + Log: crash recovery for multi-step updates.
5003 //   + Files: inode allocator, reading, writing, metadata.
5004 //   + Directories: inode with special contents (list of other inodes!)
5005 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
5006 //
5007 // This file contains the low-level file system manipulation
5008 // routines.  The (higher-level) system call implementations
5009 // are in sysfile.c.
5010
5011 #include "types.h"
5012 #include "riscv.h"
5013 #include "defs.h"
5014 #include "param.h"
5015 #include "stat.h"
5016 #include "spinlock.h"
5017 #include "proc.h"
5018 #include "sleeplock.h"
5019 #include "fs.h"
5020 #include "buf.h"
5021 #include "file.h"
5022
5023 #define min(a, b) ((a) < (b) ? (a) : (b))
5024 // there should be one superblock per disk device, but we run with
5025 // only one device
5026 struct superblock sb;
5027
5028 // Read the super block.
5029 static void
5030 readsb(int dev, struct superblock *sb)
5031 {
5032   struct buf *bp;
5033
5034   bp = bread(dev, 1);
5035   memmove(sb, bp->data, sizeof(*sb));
5036   brelse(bp);
5037 }
5038
5039 // Init fs
5040 void
5041 fsinit(int dev)
5042 {
5043   readsb(dev, &sb);
5044   if (sb.magic != FSMAGIC)
5045     panic("invalid file system");
5046   initlog(dev, &sb);
5047   ireclaim(dev);
5048 }
5049
```



```
5050 // Zero a block.
5051 static void
5052 bzero(int dev, int bno)
5053 {
5054   struct buf *bp;
5055
5056   bp = bread(dev, bno);
5057   memset(bp->data, 0, BSIZE);
5058   log_write(bp);
5059   brelse(bp);
5060 }
5061
5062 // Blocks.
5063
5064 // Allocate a zeroed disk block.
5065 // returns 0 if out of disk space.
5066 static uint
5067 balloc(uint dev)
5068 {
5069   int b, bi, m;
5070   struct buf *bp;
5071
5072   bp = 0;
5073   for (b = 0; b < sb.size; b += BPB) {
5074     bp = bread(dev, BBLOCK(b, sb));
5075     for (bi = 0; bi < BPB && b + bi < sb.size; bi++) {
5076       m = 1 << (bi % 8);
5077       if ((bp->data[bi / 8] & m) == 0) { // Is block free?
5078         bp->data[bi / 8] |= m;           // Mark block in use.
5079         log_write(bp);
5080         brelse(bp);
5081         bzero(dev, b + bi);
5082         return b + bi;
5083       }
5084     }
5085     brelse(bp);
5086   }
5087   printk("balloc: out of blocks\n");
5088   return 0;
5089 }
5090
5091
5092
5093
5094
5095
5096
5097
5098
5099
```

```
5100 // Free a disk block.
5101 static void
5102 bfree(int dev, uint b)
5103 {
5104   struct buf *bp;
5105   int bi, m;
5106
5107   bp = bread(dev, BBLOCK(b, sb));
5108   bi = b % BPB;
5109   m = 1 << (bi % 8);
5110   if ((bp->data[bi / 8] & m) == 0)
5111     panic("freeing free block");
5112   bp->data[bi / 8] &= ~m;
5113   log_write(bp);
5114   brelse(bp);
5115 }
5116
5117 // Inodes.
5118 //
5119 // An inode describes a single unnamed file.
5120 // The inode disk structure holds metadata: the file's type,
5121 // its size, the number of links referring to it, and the
5122 // list of blocks holding the file's content.
5123 //
5124 // The inodes are laid out sequentially on disk at block
5125 // sb.inodestart. Each inode has a number, indicating its
5126 // position on the disk.
5127 //
5128 // The kernel keeps a table of in-use inodes in memory
5129 // to provide a place for synchronizing access
5130 // to inodes used by multiple processes. The in-memory
5131 // inodes include book-keeping information that is
5132 // not stored on disk: ip->ref and ip->valid.
5133 //
5134 // An inode and its in-memory representation go through a
5135 // sequence of states before they can be used by the
5136 // rest of the file system code.
5137 //
5138 // * Allocation: an inode is allocated if its type (on disk)
5139 //   is non-zero. ialloc() allocates, and iput() frees if
5140 //   the reference and link counts have fallen to zero.
5141 //
5142 // * Referencing in table: an entry in the inode table
5143 //   is free if ip->ref is zero. Otherwise ip->ref tracks
5144 //   the number of in-memory pointers to the entry (open
5145 //   files and current directories). iget() finds or
5146 //   creates a table entry and increments its ref; iput()
5147 //   decrements ref.
5148 //
5149 // * Valid: the information (type, size, &c) in an inode
```



```
5150 //   table entry is only correct when ip->valid is 1.
5151 //   ilock() reads the inode from
5152 //   the disk and sets ip->valid, while iput() clears
5153 //   ip->valid if ip->ref has fallen to zero.
5154 //
5155 // * Locked: file system code may only examine and modify
5156 //   the information in an inode and its content if it
5157 //   has first locked the inode.
5158 //
5159 // Thus a typical sequence is:
5160 //   ip = iget(dev, inum)
5161 //   ilock(ip)
5162 //   ... examine and modify ip->xxx ...
5163 //   iunlock(ip)
5164 //   iput(ip)
5165 //
5166 // ilock() is separate from iget() so that system calls can
5167 // get a long-term reference to an inode (as for an open file)
5168 // and only lock it for short periods (e.g., in read()).
5169 // The separation also helps avoid deadlock and races during
5170 // pathname lookup. iget() increments ip->ref so that the inode
5171 // stays in the table and pointers to it remain valid.
5172 //
5173 // Many internal file system functions expect the caller to
5174 // have locked the inodes involved; this lets callers create
5175 // multi-step atomic operations.
5176 //
5177 // The itable.lock spin-lock protects the allocation of itable
5178 // entries. Since ip->ref indicates whether an entry is free,
5179 // and ip->dev and ip->inum indicate which i-node an entry
5180 // holds, one must hold itable.lock while using any of those fields.
5181 //
5182 // An ip->lock sleep-lock protects all ip-> fields other than ref,
5183 // dev, and inum.  One must hold ip->lock in order to
5184 // read or write that inode's ip->valid, ip->size, ip->type, &c.
5185
5186 struct {
5187   struct spinlock lock;
5188   struct inode inode[NINODE];
5189 } itable;
5190
5191 void
5192 iinit()
5193 {
5194   int i = 0;
5195
5196   initlock(&itable.lock, "itable");
5197   for (i = 0; i < NINODE; i++) {
5198     initsleeplock(&itable.inode[i].lock, "inode");
5199   }
```

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



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

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



```
5350 // Drop a reference to an in-memory inode.
5351 // If that was the last reference, the inode table entry can
5352 // be recycled.
5353 // If that was the last reference and the inode has no links
5354 // to it, free the inode (and its content) on disk.
5355 // All calls to iput() must be inside a transaction in
5356 // case it has to free the inode.
5357 void
5358 iput(struct inode *ip)
5359 {
5360   acquire(&itable.lock);
5361
5362   if (ip->ref == 1 && ip->valid && ip->nlink == 0) {
5363     // inode has no links and no other references: truncate and free.
5364
5365     // ip->ref == 1 means no other process can have ip locked,
5366     // so this acquiresleep() won't block (or deadlock).
5367     acquiresleep(&ip->lock);
5368
5369     release(&itable.lock);
5370
5371     itrunc(ip);
5372     ip->type = 0;
5373     iupdate(ip);
5374     ip->valid = 0;
5375
5376     releasesleep(&ip->lock);
5377
5378     acquire(&itable.lock);
5379   }
5380
5381   ip->ref--;
5382   release(&itable.lock);
5383 }
5384
5385 // Common idiom: unlock, then put.
5386 void
5387 iunlockput(struct inode *ip)
5388 {
5389   iunlock(ip);
5390   iput(ip);
5391 }
5392
5393
5394
5395
5396
5397
5398
5399
```

```
5400 void
5401 ireclaim(int dev)
5402 {
5403   for (int inum = 1; inum < sb.ninodes; inum++) {
5404     struct inode *ip = 0;
5405     struct buf *bp = bread(dev, IBLOCK(inum, sb));
5406     struct dinode *dip = (struct dinode *)bp->data + inum % IPB;
5407     if (dip->type != 0 && dip->nlink == 0) { // is an orphaned inode
5408       printk("ireclaim: orphaned inode %d\n", inum);
5409       ip = iget(dev, inum);
5410     }
5411     brelse(bp);
5412     if (ip) {
5413       begin_op();
5414       ilock(ip);
5415       iunlock(ip);
5416       iput(ip);
5417       end_op();
5418     }
5419   }
5420 }
5421
5422 // Inode content
5423 //
5424 // The content (data) associated with each inode is stored
5425 // in blocks on the disk. The first NDIRECT block numbers
5426 // are listed in ip->addrs[].  The next NINDIRECT blocks are
5427 // listed in block ip->addrs[NDIRECT].
5428
5429 // Return the disk block address of the nth block in inode ip.
5430 // If there is no such block, bmap allocates one.
5431 // returns 0 if out of disk space.
5432 static uint
5433 bmap(struct inode *ip, uint bn)
5434 {
5435   uint addr, *a;
5436   struct buf *bp;
5437
5438   if (bn < NDIRECT) {
5439     if ((addr = ip->addrs[bn]) == 0) {
5440       addr = balloc(ip->dev);
5441       if (addr == 0)
5442         return 0;
5443       ip->addrs[bn] = addr;
5444     }
5445     return addr;
5446   }
5447   bn -= NDIRECT;
5448
5449
```



```
5450   if (bn < NINDIRECT) {
5451     // Load indirect block, allocating if necessary.
5452     if ((addr = ip->addrs[NDIRECT]) == 0) {
5453       addr = balloc(ip->dev);
5454       if (addr == 0)
5455         return 0;
5456       ip->addrs[NDIRECT] = addr;
5457     }
5458     bp = bread(ip->dev, addr);
5459     a = (uint *)bp->data;
5460     if ((addr = a[bn]) == 0) {
5461       addr = balloc(ip->dev);
5462       if (addr) {
5463         a[bn] = addr;
5464         log_write(bp);
5465       }
5466     }
5467     brelse(bp);
5468     return addr;
5469   }
5470
5471   panic("bmap: out of range");
5472 }
5473
5474 // Truncate inode (discard contents).
5475 // Caller must hold ip->lock.
5476 void
5477 itrunc(struct inode *ip)
5478 {
5479   int i, j;
5480   struct buf *bp;
5481   uint *a;
5482
5483   for (i = 0; i < NDIRECT; i++) {
5484     if (ip->addrs[i]) {
5485       bfree(ip->dev, ip->addrs[i]);
5486       ip->addrs[i] = 0;
5487     }
5488   }
5489
5490   if (ip->addrs[NDIRECT]) {
5491     bp = bread(ip->dev, ip->addrs[NDIRECT]);
5492     a = (uint *)bp->data;
5493     for (j = 0; j < NINDIRECT; j++) {
5494       if (a[j])
5495         bfree(ip->dev, a[j]);
5496     }
5497     brelse(bp);
5498     bfree(ip->dev, ip->addrs[NDIRECT]);
5499     ip->addrs[NDIRECT] = 0;
```

```
5500   }
5501
5502   ip->size = 0;
5503   iupdate(ip);
5504 }
5505
5506 // Copy stat information from inode.
5507 // Caller must hold ip->lock.
5508 void
5509 stati(struct inode *ip, struct stat *st)
5510 {
5511   st->dev = ip->dev;
5512   st->ino = ip->inum;
5513   st->type = ip->type;
5514   st->nlink = ip->nlink;
5515   st->size = ip->size;
5516 }
5517
5518 // Read data from inode.
5519 // Caller must hold ip->lock.
5520 // If user_dst==1, then dst is a user virtual address;
5521 // otherwise, dst is a kernel address.
5522 int
5523 readi(struct inode *ip, int user_dst, uint64 dst, uint off, uint n)
5524 {
5525   uint tot, m;
5526   struct buf *bp;
5527
5528   if (off > ip->size || off + n < off)
5529     return 0;
5530   if (off + n > ip->size)
5531     n = ip->size - off;
5532
5533   for (tot = 0; tot < n; tot += m, off += m, dst += m) {
5534     uint addr = bmap(ip, off / BSIZE);
5535     if (addr == 0)
5536       break;
5537     bp = bread(ip->dev, addr);
5538     m = min(n - tot, BSIZE - off % BSIZE);
5539     if (either_copyout(user_dst, dst, bp->data + (off % BSIZE), m) == -1) {
5540       brelse(bp);
5541       tot = -1;
5542       break;
5543     }
5544     brelse(bp);
5545   }
5546   return tot;
5547 }
5548
5549
```



```
5550 // Write data to inode.
5551 // Caller must hold ip->lock.
5552 // If user_src==1, then src is a user virtual address;
5553 // otherwise, src is a kernel address.
5554 // Returns the number of bytes successfully written.
5555 // If the return value is less than the requested n,
5556 // there was an error of some kind.
5557 int
5558 writei(struct inode *ip, int user_src, uint64 src, uint off, uint n)
5559 {
5560   uint tot, m;
5561   struct buf *bp;
5562
5563   if (off > ip->size || off + n < off)
5564     return -1;
5565   if (off + n > MAXFILE * BSIZE)
5566     return -1;
5567
5568   for (tot = 0; tot < n; tot += m, off += m, src += m) {
5569     uint addr = bmap(ip, off / BSIZE);
5570     if (addr == 0)
5571       break;
5572     bp = bread(ip->dev, addr);
5573     m = min(n - tot, BSIZE - off % BSIZE);
5574     if (either_copyin(bp->data + (off % BSIZE), user_src, src, m) == -1) {
5575       // Might have partially updated the block, so we need to log it.
5576       log_write(bp);
5577       brelse(bp);
5578       break;
5579     }
5580     log_write(bp);
5581     brelse(bp);
5582   }
5583
5584   if (off > ip->size)
5585     ip->size = off;
5586
5587   // write the i-node back to disk even if the size didn't change
5588   // because the loop above might have called bmap() and added a new
5589   // block to ip->addrs[].
5590   iupdate(ip);
5591
5592   return tot;
5593 }
5594
5595
5596
5597
5598
5599
```

```
5600 // Directories
5601
5602 int
5603 namecmp(const char *s, const char *t)
5604 {
5605   return strncmp(s, t, DIRSIZ);
5606 }
5607
5608 // Look for a directory entry in a directory.
5609 // If found, set *poff to byte offset of entry.
5610 struct inode *
5611 dirlookup(struct inode *dp, char *name, uint *poff)
5612 {
5613   uint off, inum;
5614   struct dirent de;
5615
5616   if (dp->type != T_DIR)
5617     panic("dirlookup not DIR");
5618
5619   for (off = 0; off < dp->size; off += sizeof(de)) {
5620     if (readi(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
5621       panic("dirlookup read");
5622     if (de.inum == 0)
5623       continue;
5624     if (namecmp(name, de.name) == 0) {
5625       // entry matches path element
5626       if (poff)
5627         *poff = off;
5628       inum = de.inum;
5629       return iget(dp->dev, inum);
5630     }
5631   }
5632
5633   return 0;
5634 }
5635
5636
5637
5638
5639
5640
5641
5642
5643
5644
5645
5646
5647
5648
5649
```



```
5650 // Write a new directory entry (name, inum) into the directory dp.
5651 // Returns 0 on success, -1 on failure (e.g. out of disk blocks).
5652 int
5653 dirlink(struct inode *dp, char *name, uint inum)
5654 {
5655   int off;
5656   struct dirent de;
5657   struct inode *ip;
5658
5659   // Check that name is not present.
5660   if ((ip = dirlookup(dp, name, 0)) != 0) {
5661     iput(ip);
5662     return -1;
5663   }
5664
5665   // Look for an empty dirent.
5666   for (off = 0; off < dp->size; off += sizeof(de)) {
5667     if (readi(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
5668       panic("dirlink read");
5669     if (de.inum == 0)
5670       break;
5671   }
5672
5673   strncpy(de.name, name, DIRSIZ);
5674   de.inum = inum;
5675   if (writei(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
5676     return -1;
5677
5678   return 0;
5679 }
5680
5681 // Paths
5682
5683 // Copy the next path element from path into name.
5684 // Return a pointer to the element following the copied one.
5685 // The returned path has no leading slashes,
5686 // so the caller can check *path=='\0' to see if the name is the last one.
5687 // If no name to remove, return 0.
5688 //
5689 // Examples:
5690 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
5691 //   skipelem("///a//bb", name) = "bb", setting name = "a"
5692 //   skipelem("a", name) = "", setting name = "a"
5693 //   skipelem("", name) = skipelem("////", name) = 0
5694 //
5695 static char *
5696 skipelem(char *path, char *name)
5697 {
5698   char *s;
5699   int len;
```

```
  while (*path == '/')
    path++;
  if (*path == 0)
    return 0;
  s = path;
  while (*path != '/' && *path != 0)
    path++;
  len = path - s;
  if (len >= DIRSIZ)
    memmove(name, s, DIRSIZ);
  else {
    memmove(name, s, len);
    name[len] = 0;
  }
  while (*path == '/')
    path++;
  return path;
}

// Look up and return the inode for a path name.
// If parent != 0, return the inode for the parent and copy the final
// path element into name, which must have room for DIRSIZ bytes.
// Must be called inside a transaction since it calls iput().
static struct inode *
namex(char *path, int nameiparent, char *name)
{
  struct inode *ip, *next;

  if (*path == '/')
    ip = iget(ROOTDEV, ROOTINO);
  else
    ip = idup(myproc()->cwd);

  while ((path = skipelem(path, name)) != 0) {
    ilock(ip);
    if (ip->type != T_DIR) {
      iunlockput(ip);
      return 0;
    }
    if (nameiparent && *path == '\0') {
      // Stop one level early.
      iunlock(ip);
      return ip;
    }
    if ((next = dirlookup(ip, name, 0)) == 0) {
      iunlockput(ip);
      return 0;
    }
    iunlockput(ip);
    ip = next;
```



```
  }
  if (nameiparent) {
    iput(ip);
    return 0;
  }
  return ip;
}

struct inode *
namei(char *path)
{
  char name[DIRSIZ];
  return namex(path, 0, name);
}

struct inode *
nameiparent(char *path, char *name)
{
  return namex(path, 1, name);
}
```

```
5800 //
5801 // Support functions for system calls that involve file descriptors.
5802 //
5803
5804 #include "types.h"
5805 #include "riscv.h"
5806 #include "defs.h"
5807 #include "param.h"
5808 #include "fs.h"
5809 #include "spinlock.h"
5810 #include "sleeplock.h"
5811 #include "file.h"
5812 #include "stat.h"
5813 #include "proc.h"
5814
5815 struct devsw devsw[NDEV];
5816 struct {
5817   struct spinlock lock;
5818   struct file file[NFILE];
5819 } ftable;
5820
5821 void
5822 fileinit(void)
5823 {
5824   initlock(&ftable.lock, "ftable");
5825 }
5826
5827 // Allocate a file structure.
5828 struct file *
5829 filealloc(void)
5830 {
5831   struct file *f;
5832
5833   acquire(&ftable.lock);
5834   for (f = ftable.file; f < ftable.file + NFILE; f++) {
5835     if (f->ref == 0) {
5836       f->ref = 1;
5837       release(&ftable.lock);
5838       return f;
5839     }
5840   }
5841   release(&ftable.lock);
5842   return 0;
5843 }
5844
5845
5846
5847
5848
5849
```



```
5850 // Increment ref count for file f.
5851 struct file *
5852 filedup(struct file *f)
5853 {
5854   acquire(&ftable.lock);
5855   if (f->ref < 1)
5856     panic("filedup");
5857   f->ref++;
5858   release(&ftable.lock);
5859   return f;
5860 }
5861
5862 // Close file f.  (Decrement ref count, close when reaches 0.)
5863 void
5864 fileclose(struct file *f)
5865 {
5866   struct file ff;
5867
5868   acquire(&ftable.lock);
5869   if (f->ref < 1)
5870     panic("fileclose");
5871   if (--f->ref > 0) {
5872     release(&ftable.lock);
5873     return;
5874   }
5875   ff = *f;
5876   f->ref = 0;
5877   f->type = FD_NONE;
5878   release(&ftable.lock);
5879
5880   if (ff.type == FD_PIPE) {
5881     pipeclose(ff.pipe, ff.writable);
5882   } else if (ff.type == FD_INODE || ff.type == FD_DEVICE) {
5883     begin_op();
5884     iput(ff.ip);
5885     end_op();
5886   }
5887 }
5888
5889
5890
5891
5892
5893
5894
5895
5896
5897
5898
5899
```

```
5900 // Get metadata about file f.
5901 // addr is a user virtual address, pointing to a struct stat.
5902 int
5903 filestat(struct file *f, uint64 addr)
5904 {
5905   struct proc *p = myproc();
5906   struct stat st;
5907
5908   if (f->type == FD_INODE || f->type == FD_DEVICE) {
5909     ilock(f->ip);
5910     stati(f->ip, &st);
5911     iunlock(f->ip);
5912     if (copyout(p->pagetable, addr, (char *)&st, sizeof(st)) < 0)
5913       return -1;
5914     return 0;
5915   }
5916   return -1;
5917 }
5918
5919 // Read from file f.
5920 // addr is a user virtual address.
5921 int
5922 fileread(struct file *f, uint64 addr, int n)
5923 {
5924   int r = 0;
5925
5926   if (f->readable == 0)
5927     return -1;
5928
5929   if (f->type == FD_PIPE) {
5930     r = piperead(f->pipe, addr, n);
5931   } else if (f->type == FD_DEVICE) {
5932     if (f->major < 0 || f->major >= NDEV || !devsw[f->major].read)
5933       return -1;
5934     r = devsw[f->major].read(1, addr, n);
5935   } else if (f->type == FD_INODE) {
5936     ilock(f->ip);
5937     if ((r = readi(f->ip, 1, addr, f->off, n)) > 0)
5938       f->off += r;
5939     iunlock(f->ip);
5940   } else {
5941     panic("fileread");
5942   }
5943
5944   return r;
5945 }
5946
5947
5948
5949
```



```
5950 // Write to file f.
5951 // addr is a user virtual address.
5952 int
5953 filewrite(struct file *f, uint64 addr, int n)
5954 {
5955   int r, ret = 0;
5956
5957   if (f->writable == 0)
5958     return -1;
5959
5960   if (f->type == FD_PIPE) {
5961     ret = pipewrite(f->pipe, addr, n);
5962   } else if (f->type == FD_DEVICE) {
5963     if (f->major < 0 || f->major >= NDEV || !devsw[f->major].write)
5964       return -1;
5965     ret = devsw[f->major].write(1, addr, n);
5966   } else if (f->type == FD_INODE) {
5967     // write a few blocks at a time to avoid exceeding
5968     // the maximum log transaction size, including
5969     // i-node, indirect block, allocation blocks,
5970     // and 2 blocks of slop for non-aligned writes.
5971     int max = ((MAXOPBLOCKS - 1 - 1 - 2) / 2) * BSIZE;
5972     int i = 0;
5973     while (i < n) {
5974       int n1 = n - i;
5975       if (n1 > max)
5976         n1 = max;
5977
5978       begin_op();
5979       ilock(f->ip);
5980       if ((r = writei(f->ip, 1, addr + i, f->off, n1)) > 0)
5981         f->off += r;
5982       iunlock(f->ip);
5983       end_op();
5984
5985       if (r != n1) {
5986         // error from writei
5987         break;
5988       }
5989       i += r;
5990     }
5991     ret = (i == n ? n : -1);
5992   } else {
5993     panic("filewrite");
5994   }
5995
5996   return ret;
5997 }
5998
5999
```

```
6000 //
6001 // File-system system calls.
6002 // Mostly argument checking, since we don't trust
6003 // user code, and calls into file.c and fs.c.
6004 //
6005
6006 #include "types.h"
6007 #include "riscv.h"
6008 #include "defs.h"
6009 #include "param.h"
6010 #include "stat.h"
6011 #include "spinlock.h"
6012 #include "proc.h"
6013 #include "fs.h"
6014 #include "sleeplock.h"
6015 #include "file.h"
6016 #include "fcntl.h"
6017
6018 // Fetch the nth word-sized system call argument as a file descriptor
6019 // and return both the descriptor and the corresponding struct file.
6020 static int
6021 argfd(int n, int *pfd, struct file **pf)
6022 {
6023   int fd;
6024   struct file *f;
6025
6026   argint(n, &fd);
6027   if (fd < 0 || fd >= NOFILE || (f = myproc()->ofile[fd]) == 0)
6028     return -1;
6029   if (pfd)
6030     *pfd = fd;
6031   if (pf)
6032     *pf = f;
6033   return 0;
6034 }
6035
6036
6037
6038
6039
6040
6041
6042
6043
6044
6045
6046
6047
6048
6049
```



```
6050 // Allocate a file descriptor for the given file.
6051 // Takes over file reference from caller on success.
6052 static int
6053 fdalloc(struct file *f)
6054 {
6055   int fd;
6056   struct proc *p = myproc();
6057
6058   for (fd = 0; fd < NOFILE; fd++) {
6059     if (p->ofile[fd] == 0) {
6060       p->ofile[fd] = f;
6061       return fd;
6062     }
6063   }
6064   return -1;
6065 }
6066
6067 uint64
6068 sys_dup(void)
6069 {
6070   struct file *f;
6071   int fd;
6072
6073   if (argfd(0, 0, &f) < 0)
6074     return -1;
6075   if ((fd = fdalloc(f)) < 0)
6076     return -1;
6077   filedup(f);
6078   return fd;
6079 }
6080
6081 uint64
6082 sys_read(void)
6083 {
6084   struct file *f;
6085   int n;
6086   uint64 p;
6087
6088   argaddr(1, &p);
6089   argint(2, &n);
6090   if (argfd(0, 0, &f) < 0)
6091     return -1;
6092   return fileread(f, p, n);
6093 }
6094
6095
6096
6097
6098
6099
```

```
6100 uint64
6101 sys_write(void)
6102 {
6103   struct file *f;
6104   int n;
6105   uint64 p;
6106
6107   argaddr(1, &p);
6108   argint(2, &n);
6109   if (argfd(0, 0, &f) < 0)
6110     return -1;
6111
6112   return filewrite(f, p, n);
6113 }
6114
6115 uint64
6116 sys_close(void)
6117 {
6118   int fd;
6119   struct file *f;
6120
6121   if (argfd(0, &fd, &f) < 0)
6122     return -1;
6123   myproc()->ofile[fd] = 0;
6124   fileclose(f);
6125   return 0;
6126 }
6127
6128 uint64
6129 sys_fstat(void)
6130 {
6131   struct file *f;
6132   uint64 st; // user pointer to struct stat
6133
6134   argaddr(1, &st);
6135   if (argfd(0, 0, &f) < 0)
6136     return -1;
6137   return filestat(f, st);
6138 }
6139
6140
6141
6142
6143
6144
6145
6146
6147
6148
6149
```



```
6150 // Create the path new as a link to the same inode as old.
6151 uint64
6152 sys_link(void)
6153 {
6154   char name[DIRSIZ], new[MAXPATH], old[MAXPATH];
6155   struct inode *dp, *ip;
6156
6157   if (argstr(0, old, MAXPATH) < 0 || argstr(1, new, MAXPATH) < 0)
6158     return -1;
6159
6160   begin_op();
6161   if ((ip = namei(old)) == 0) {
6162     end_op();
6163     return -1;
6164   }
6165
6166   ilock(ip);
6167   if (ip->type == T_DIR) {
6168     iunlockput(ip);
6169     end_op();
6170     return -1;
6171   }
6172
6173   ip->nlink++;
6174   iupdate(ip);
6175   iunlock(ip);
6176
6177   if ((dp = nameiparent(new, name)) == 0)
6178     goto bad;
6179   ilock(dp);
6180   if (dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0) {
6181     iunlockput(dp);
6182     goto bad;
6183   }
6184   iunlockput(dp);
6185   iput(ip);
6186
6187   end_op();
6188
6189   return 0;
6190
6191 bad:
6192   ilock(ip);
6193   ip->nlink--;
6194   iupdate(ip);
6195   iunlockput(ip);
6196   end_op();
6197   return -1;
6198 }
6199
```

```
6200 // Is the directory dp empty except for "." and ".." ?
6201 static int
6202 isdirempty(struct inode *dp)
6203 {
6204   int off;
6205   struct dirent de;
6206
6207   for (off = 2 * sizeof(de); off < dp->size; off += sizeof(de)) {
6208     if (readi(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
6209       panic("isdirempty: readi");
6210     if (de.inum != 0)
6211       return 0;
6212   }
6213   return 1;
6214 }
6215
6216 uint64
6217 sys_unlink(void)
6218 {
6219   struct inode *ip, *dp;
6220   struct dirent de;
6221   char name[DIRSIZ], path[MAXPATH];
6222   uint off;
6223
6224   if (argstr(0, path, MAXPATH) < 0)
6225     return -1;
6226
6227   begin_op();
6228   if ((dp = nameiparent(path, name)) == 0) {
6229     end_op();
6230     return -1;
6231   }
6232
6233   ilock(dp);
6234
6235   // Cannot unlink "." or "..".
6236   if (namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6237     goto bad;
6238
6239   if ((ip = dirlookup(dp, name, &off)) == 0)
6240     goto bad;
6241   ilock(ip);
6242
6243   if (ip->nlink < 1)
6244     panic("unlink: nlink < 1");
6245   if (ip->type == T_DIR && !isdirempty(ip)) {
6246     iunlockput(ip);
6247     goto bad;
6248   }
6249
```



```
6250   memset(&de, 0, sizeof(de));
6251   if (writei(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
6252     panic("unlink: writei");
6253   if (ip->type == T_DIR) {
6254     dp->nlink--;
6255     iupdate(dp);
6256   }
6257   iunlockput(dp);
6258
6259   ip->nlink--;
6260   iupdate(ip);
6261   iunlockput(ip);
6262
6263   end_op();
6264
6265   return 0;
6266
6267 bad:
6268   iunlockput(dp);
6269   end_op();
6270   return -1;
6271 }
6272
6273 static struct inode *
6274 create(char *path, short type, short major, short minor)
6275 {
6276   struct inode *ip, *dp;
6277   char name[DIRSIZ];
6278
6279   if ((dp = nameiparent(path, name)) == 0)
6280     return 0;
6281
6282   ilock(dp);
6283
6284   if ((ip = dirlookup(dp, name, 0)) != 0) {
6285     iunlockput(dp);
6286     ilock(ip);
6287     if (type == T_FILE && (ip->type == T_FILE || ip->type == T_DEVICE))
6288       return ip;
6289     iunlockput(ip);
6290     return 0;
6291   }
6292
6293   if ((ip = ialloc(dp->dev, type)) == 0) {
6294     iunlockput(dp);
6295     return 0;
6296   }
6297
6298
6299
```

```
6300   ilock(ip);
6301   ip->major = major;
6302   ip->minor = minor;
6303   ip->nlink = 1;
6304   iupdate(ip);
6305
6306   if (type == T_DIR) { // Create . and .. entries.
6307     // No ip->nlink++ for ".": avoid cyclic ref count.
6308     if (dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
6309       goto fail;
6310   }
6311
6312   if (dirlink(dp, name, ip->inum) < 0)
6313     goto fail;
6314
6315   if (type == T_DIR) {
6316     // now that success is guaranteed:
6317     dp->nlink++; // for ".."
6318     iupdate(dp);
6319   }
6320
6321   iunlockput(dp);
6322
6323   return ip;
6324
6325 fail:
6326   // something went wrong. de-allocate ip.
6327   ip->nlink = 0;
6328   iupdate(ip);
6329   iunlockput(ip);
6330   iunlockput(dp);
6331   return 0;
6332 }
6333
6334 uint64
6335 sys_open(void)
6336 {
6337   char path[MAXPATH];
6338   int fd, omode;
6339   struct file *f;
6340   struct inode *ip;
6341   int n;
6342
6343   argint(1, &omode);
6344   if ((n = argstr(0, path, MAXPATH)) < 0)
6345     return -1;
6346
6347   begin_op();
6348
6349
```



```
6350   if (omode & O_CREATE) {
6351     ip = create(path, T_FILE, 0, 0);
6352     if (ip == 0) {
6353       end_op();
6354       return -1;
6355     }
6356   } else {
6357     if ((ip = namei(path)) == 0) {
6358       end_op();
6359       return -1;
6360     }
6361     ilock(ip);
6362     if (ip->type == T_DIR && omode != O_RDONLY) {
6363       iunlockput(ip);
6364       end_op();
6365       return -1;
6366     }
6367   }
6368
6369   if (ip->type == T_DEVICE && (ip->major < 0 || ip->major >= NDEV)) {
6370     iunlockput(ip);
6371     end_op();
6372     return -1;
6373   }
6374
6375   if ((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0) {
6376     if (f)
6377       fileclose(f);
6378     iunlockput(ip);
6379     end_op();
6380     return -1;
6381   }
6382
6383   if (ip->type == T_DEVICE) {
6384     f->type = FD_DEVICE;
6385     f->major = ip->major;
6386   } else {
6387     f->type = FD_INODE;
6388     f->off = 0;
6389   }
6390   f->ip = ip;
6391   f->readable = !(omode & O_WRONLY);
6392   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6393
6394   if ((omode & O_TRUNC) && ip->type == T_FILE) {
6395     itrunc(ip);
6396   }
6397
6398   iunlock(ip);
6399   end_op();
```

```
6400   return fd;
6401 }
6402
6403 uint64
6404 sys_mkdir(void)
6405 {
6406   char path[MAXPATH];
6407   struct inode *ip;
6408
6409   begin_op();
6410   if (argstr(0, path, MAXPATH) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0) {
6411     end_op();
6412     return -1;
6413   }
6414   iunlockput(ip);
6415   end_op();
6416   return 0;
6417 }
6418
6419 uint64
6420 sys_mknod(void)
6421 {
6422   struct inode *ip;
6423   char path[MAXPATH];
6424   int major, minor;
6425
6426   begin_op();
6427   argint(1, &major);
6428   argint(2, &minor);
6429   if ((argstr(0, path, MAXPATH)) < 0 ||
6430       (ip = create(path, T_DEVICE, major, minor)) == 0) {
6431     end_op();
6432     return -1;
6433   }
6434   iunlockput(ip);
6435   end_op();
6436   return 0;
6437 }
6438
6439
6440
6441
6442
6443
6444
6445
6446
6447
6448
6449
```



```
6450 uint64
6451 sys_chdir(void)
6452 {
6453   char path[MAXPATH];
6454   struct inode *ip;
6455   struct proc *p = myproc();
6456
6457   begin_op();
6458   if (argstr(0, path, MAXPATH) < 0 || (ip = namei(path)) == 0) {
6459     end_op();
6460     return -1;
6461   }
6462   ilock(ip);
6463   if (ip->type != T_DIR) {
6464     iunlockput(ip);
6465     end_op();
6466     return -1;
6467   }
6468   iunlock(ip);
6469   iput(p->cwd);
6470   end_op();
6471   p->cwd = ip;
6472   return 0;
6473 }
6474
6475 uint64
6476 sys_exec(void)
6477 {
6478   char path[MAXPATH], *argv[MAXARG];
6479   int i;
6480   uint64 uargv, uarg;
6481
6482   argaddr(1, &uargv);
6483   if (argstr(0, path, MAXPATH) < 0) {
6484     return -1;
6485   }
6486   memset(argv, 0, sizeof(argv));
6487   for (i = 0;; i++) {
6488     if (i >= NELEM(argv)) {
6489       goto bad;
6490     }
6491     if (fetchaddr(uargv + sizeof(uint64) * i, (uint64 *)&uarg) < 0) {
6492       goto bad;
6493     }
6494     if (uarg == 0) {
6495       argv[i] = 0;
6496       break;
6497     }
6498     argv[i] = kalloc();
6499     if (argv[i] == 0)
```

```
6500       goto bad;
6501     if (fetchstr(uarg, argv[i], PGSIZE) < 0)
6502       goto bad;
6503   }
6504
6505   int ret = kexec(path, argv);
6506
6507   for (i = 0; i < NELEM(argv) && argv[i] != 0; i++)
6508     kfree(argv[i]);
6509
6510   return ret;
6511
6512 bad:
6513   for (i = 0; i < NELEM(argv) && argv[i] != 0; i++)
6514     kfree(argv[i]);
6515   return -1;
6516 }
6517
6518 uint64
6519 sys_pipe(void)
6520 {
6521   uint64 fdarray; // user pointer to array of two integers
6522   struct file *rf, *wf;
6523   int fd0, fd1;
6524   struct proc *p = myproc();
6525
6526   argaddr(0, &fdarray);
6527   if (pipealloc(&rf, &wf) < 0)
6528     return -1;
6529   fd0 = -1;
6530   if ((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0) {
6531     if (fd0 >= 0)
6532       p->ofile[fd0] = 0;
6533     fileclose(rf);
6534     fileclose(wf);
6535     return -1;
6536   }
6537   if (copyout(p->pagetable, fdarray, (char *)&fd0, sizeof(fd0)) < 0 ||
6538       copyout(p->pagetable, fdarray + sizeof(fd0), (char *)&fd1, sizeof(fd1)) <
6539         0) {
6540     p->ofile[fd0] = 0;
6541     p->ofile[fd1] = 0;
6542     fileclose(rf);
6543     fileclose(wf);
6544     return -1;
6545   }
6546   return 0;
6547 }
6548
6549
```



```
6550 #include "types.h"
6551 #include "param.h"
6552 #include "memlayout.h"
6553 #include "riscv.h"
6554 #include "spinlock.h"
6555 #include "proc.h"
6556 #include "defs.h"
6557 #include "elf.h"
6558
6559 static int loadseg(pde_t *, uint64, struct inode *, uint, uint);
6560
6561 // map ELF permissions to PTE permission bits.
6562 int
6563 flags2perm(int flags)
6564 {
6565   int perm = 0;
6566   if (flags & 0x1)
6567     perm = PTE_X;
6568   if (flags & 0x2)
6569     perm |= PTE_W;
6570   return perm;
6571 }
6572
6573 //
6574 // the implementation of the exec() system call
6575 //
6576 int
6577 kexec(char *path, char **argv)
6578 {
6579   char *s, *last;
6580   int i, off;
6581   uint64 argc, sz = 0, sp, ustack[MAXARG], stackbase;
6582   struct elfhdr elf;
6583   struct inode *ip;
6584   struct proghdr ph;
6585   pagetable_t pagetable = 0, oldpagetable;
6586   struct proc *p = myproc();
6587
6588   begin_op();
6589
6590   // Open the executable file.
6591   if ((ip = namei(path)) == 0) {
6592     end_op();
6593     return -1;
6594   }
6595   ilock(ip);
6596
6597   // Read the ELF header.
6598   if (readi(ip, 0, (uint64)&elf, 0, sizeof(elf)) != sizeof(elf))
6599     goto bad;
```

```
6600   // Is this really an ELF file?
6601   if (elf.magic != ELF_MAGIC)
6602     goto bad;
6603
6604   if ((pagetable = proc_pagetable(p)) == 0)
6605     goto bad;
6606
6607   // Load program into memory.
6608   for (i = 0, off = elf.phoff; i < elf.phnum; i++, off += sizeof(ph)) {
6609     if (readi(ip, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
6610       goto bad;
6611     if (ph.type != ELF_PROG_LOAD)
6612       continue;
6613     if (ph.memsz < ph.filesz)
6614       goto bad;
6615     if (ph.vaddr + ph.memsz < ph.vaddr)
6616       goto bad;
6617     if (ph.vaddr % PGSIZE != 0)
6618       goto bad;
6619     uint64 sz1;
6620     if ((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz,
6621                         flags2perm(ph.flags))) == 0)
6622       goto bad;
6623     sz = sz1;
6624     if (loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0)
6625       goto bad;
6626   }
6627   iunlockput(ip);
6628   end_op();
6629   ip = 0;
6630
6631   p = myproc();
6632   uint64 oldsz = p->sz;
6633
6634   // Allocate some pages at the next page boundary.
6635   // Make the first inaccessible as a stack guard.
6636   // Use the rest as the user stack.
6637   sz = PGROUNDUP(sz);
6638   uint64 sz1;
6639   if ((sz1 = uvmalloc(pagetable, sz, sz + (USERSTACK + 1) * PGSIZE, PTE_W)) =
6640       0)
6641     goto bad;
6642   sz = sz1;
6643   uvmclear(pagetable, sz - (USERSTACK + 1) * PGSIZE);
6644   sp = sz;
6645   stackbase = sp - USERSTACK * PGSIZE;
6646
6647
6648
6649
```



```
6650   // Copy argument strings into new stack, remember their
6651   // addresses in ustack[].
6652   for (argc = 0; argv[argc]; argc++) {
6653     if (argc >= MAXARG)
6654       goto bad;
6655     sp -= strlen(argv[argc]) + 1;
6656     sp -= sp % 16; // riscv sp must be 16-byte aligned
6657     if (sp < stackbase)
6658       goto bad;
6659     if (copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
6660       goto bad;
6661     ustack[argc] = sp;
6662   }
6663   ustack[argc] = 0;
6664
6665   // push a copy of ustack[], the array of argv[] pointers.
6666   sp -= (argc + 1) * sizeof(uint64);
6667   sp -= sp % 16;
6668   if (sp < stackbase)
6669     goto bad;
6670   if (copyout(pagetable, sp, (char *)ustack, (argc + 1) * sizeof(uint64)) < 
6671     goto bad;
6672
6673   // a0 and a1 contain arguments to user main(argc, argv)
6674   // argc is returned via the system call return
6675   // value, which goes in a0.
6676   p->trapframe->a1 = sp;
6677
6678   // Save program name for debugging.
6679   for (last = s = path; *s; s++)
6680     if (*s == '/')
6681       last = s + 1;
6682   safestrcpy(p->name, last, sizeof(p->name));
6683
6684   // Commit to the user image.
6685   oldpagetable = p->pagetable;
6686   p->pagetable = pagetable;
6687   p->sz = sz;
6688   p->trapframe->epc = elf.entry; // initial program counter = ulib.c:start()
6689   p->trapframe->sp = sp;         // initial stack pointer
6690   proc_freepagetable(oldpagetable, oldsz);
6691
6692   return argc; // this ends up in a0, the first argument to main(argc, argv)
6693
6694 bad:
6695   if (pagetable)
6696     proc_freepagetable(pagetable, sz);
6697   if (ip) {
6698     iunlockput(ip);
6699     end_op();
```

```
6700   }
6701   return -1;
6702 }
6703
6704 // Load an ELF program segment into pagetable at virtual address va.
6705 // va must be page-aligned
6706 // and the pages from va to va+sz must already be mapped.
6707 // Returns 0 on success, -1 on failure.
6708 static int
6709 loadseg(pagetable_t pagetable, uint64 va, struct inode *ip, uint offset,
6710         uint sz)
6711 {
6712   uint i, n;
6713   uint64 pa;
6714
6715   for (i = 0; i < sz; i += PGSIZE) {
6716     pa = walkaddr(pagetable, va + i);
6717     if (pa == 0)
6718       panic("loadseg: address should exist");
6719     if (sz - i < PGSIZE)
6720       n = sz - i;
6721     else
6722       n = PGSIZE;
6723     if (readi(ip, 0, (uint64)pa, offset + i, n) != n)
6724       return -1;
6725   }
6726
6727   return 0;
6728 }
6729
6730
6731
6732
6733
6734
6735
6736
6737
6738
6739
6740
6741
6742
6743
6744
6745
6746
6747
6748
6749
```



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

```
6800 bad:
6801   if (pi)
6802     kfree((char *)pi);
6803   if (*f0)
6804     fileclose(*f0);
6805   if (*f1)
6806     fileclose(*f1);
6807   return -1;
6808 }
6809
6810 void
6811 pipeclose(struct pipe *pi, int writable)
6812 {
6813   acquire(&pi->lock);
6814   if (writable) {
6815     pi->writeopen = 0;
6816     wakeup(&pi->nread);
6817   } else {
6818     pi->readopen = 0;
6819     wakeup(&pi->nwrite);
6820   }
6821   if (pi->readopen == 0 && pi->writeopen == 0) {
6822     release(&pi->lock);
6823     kfree((char *)pi);
6824   } else
6825     release(&pi->lock);
6826 }
6827
6828 int
6829 pipewrite(struct pipe *pi, uint64 addr, int n)
6830 {
6831   int i = 0;
6832   struct proc *pr = myproc();
6833
6834   acquire(&pi->lock);
6835   while (i < n) {
6836     if (pi->readopen == 0 || killed(pr)) {
6837       release(&pi->lock);
6838       return -1;
6839     }
6840     if (pi->nwrite == pi->nread + PIPESIZE) {
6841       wakeup(&pi->nread);
6842       sleep(&pi->nwrite, &pi->lock);
6843     } else {
6844       char ch;
6845       if (copyin(pr->pagetable, &ch, addr + i, 1) == -1)
6846         break;
6847       pi->data[pi->nwrite++ % PIPESIZE] = ch;
6848       i++;
6849     }
```



```
6850   }
6851   wakeup(&pi->nread);
6852   release(&pi->lock);
6853
6854   return i;
6855 }
6856
6857 int
6858 piperead(struct pipe *pi, uint64 addr, int n)
6859 {
6860   int i;
6861   struct proc *pr = myproc();
6862   char ch;
6863
6864   acquire(&pi->lock);
6865   while (pi->nread == pi->nwrite && pi->writeopen) {
6866     if (killed(pr)) {
6867       release(&pi->lock);
6868       return -1;
6869     }
6870     sleep(&pi->nread, &pi->lock);
6871   }
6872   for (i = 0; i < n; i++) {
6873     if (pi->nread == pi->nwrite)
6874       break;
6875     ch = pi->data[pi->nread % PIPESIZE];
6876     if (copyout(pr->pagetable, addr + i, &ch, 1) == -1) {
6877       if (i == 0)
6878         i = -1;
6879       break;
6880     }
6881     pi->nread++;
6882   }
6883   wakeup(&pi->nwrite);
6884   release(&pi->lock);
6885   return i;
6886 }
6887
6888
6889
6890
6891
6892
6893
6894
6895
6896
6897
6898
6899
```

```
#include "types.h"

void *
memset(void *dst, int c, uint n)
{
  char *cdst = (char *)dst;
  int i;
  for (i = 0; i < n; i++) {
    cdst[i] = c;
  }
  return dst;
}

int
memcmp(const void *v1, const void *v2, uint n)
{
  const uchar *s1, *s2;

  s1 = v1;
  s2 = v2;
  while (n-- > 0) {
    if (*s1 != *s2)
      return *s1 - *s2;
    s1++, s2++;
  }

  return 0;
}

void *
memmove(void *dst, const void *src, uint n)
{
  const char *s;
  char *d;

  if (n == 0)
    return dst;

  s = src;
  d = dst;
  if (s < d && s + n > d) {
    s += n;
    d += n;
    while (n-- > 0)
      *--d = *--s;
  } else
    while (n-- > 0)
      *d++ = *s++;


```



```
  return dst;
}

// memcpy exists to placate GCC.  Use memmove.
void *
memcpy(void *dst, const void *src, uint n)
{
  return memmove(dst, src, n);
}

int
strncmp(const char *p, const char *q, uint n)
{
  while (n > 0 && *p && *p == *q)
    n--, p++, q++;
  if (n == 0)
    return 0;
  return (uchar)*p - (uchar)*q;
}

char *
strncpy(char *s, const char *t, int n)
{
  char *os;

  os = s;
  while (n-- > 0 && (*s++ = *t++) != 0)
    ;
  while (n-- > 0)
    *s++ = 0;
  return os;
}

// Like strncpy but guaranteed to NUL-terminate.
char *
safestrcpy(char *s, const char *t, int n)
{
  char *os;

  os = s;
  if (n <= 0)
    return os;
  while (--n > 0 && (*s++ = *t++) != 0)
    ;
  *s = 0;
  return os;
}


```

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



```
7050 #include "types.h"
7051 #include "param.h"
7052 #include "memlayout.h"
7053 #include "riscv.h"
7054 #include "defs.h"
7055
7056 //
7057 // the riscv Platform Level Interrupt Controller (PLIC).
7058 //
7059
7060 void
7061 plicinit(void)
7062 {
7063   // set desired IRQ priorities non-zero (otherwise disabled).
7064   *(uint32 *)(PLIC + UART0_IRQ * 4) = 1;
7065   *(uint32 *)(PLIC + VIRTIO0_IRQ * 4) = 1;
7066 }
7067
7068 void
7069 plicinithart(void)
7070 {
7071   int hart = cpuid();
7072
7073   // set enable bits for this hart's S-mode
7074   // for the uart and virtio disk.
7075   *(uint32 *)PLIC_SENABLE(hart) = (1 << UART0_IRQ) | (1 << VIRTIO0_IRQ);
7076
7077   // set this hart's S-mode priority threshold to 0.
7078   *(uint32 *)PLIC_SPRIORITY(hart) = 0;
7079 }
7080
7081 // ask the PLIC what interrupt we should serve.
7082 int
7083 plic_claim(void)
7084 {
7085   int hart = cpuid();
7086   int irq = *(uint32 *)PLIC_SCLAIM(hart);
7087   return irq;
7088 }
7089
7090 // tell the PLIC we've served this IRQ.
7091 void
7092 plic_complete(int irq)
7093 {
7094   int hart = cpuid();
7095   *(uint32 *)PLIC_SCLAIM(hart) = irq;
7096 }
7097
7098
7099
```

```
7100 //
7101 // Console input and output, to the uart.
7102 // Reads are line at a time.
7103 // Implements special input characters:
7104 //   newline -- end of line
7105 //   control-h -- backspace
7106 //   control-u -- kill line
7107 //   control-d -- end of file
7108 //   control-p -- print process list
7109 //
7110
7111 #include <stdarg.h>
7112
7113 #include "types.h"
7114 #include "param.h"
7115 #include "spinlock.h"
7116 #include "sleeplock.h"
7117 #include "fs.h"
7118 #include "file.h"
7119 #include "memlayout.h"
7120 #include "riscv.h"
7121 #include "defs.h"
7122 #include "proc.h"
7123
7124 #define BACKSPACE 0x100       // erase the last output character
7125 #define C(x)      ((x) - '@') // Control-x
7126
7127 //
7128 // send one character to the uart, but don't use
7129 // interrupts or sleep(). safe to be called from
7130 // interrupts, e.g. by printk and to echo input
7131 // characters.
7132 //
7133 void
7134 consputc(int c)
7135 {
7136   if (c == BACKSPACE) {
7137     // if the user typed backspace, overwrite with a space.
7138     uartputc_sync('\b');
7139     uartputc_sync(' ');
7140     uartputc_sync('\b');
7141   } else {
7142     uartputc_sync(c);
7143   }
7144 }
7145
7146
7147
7148
7149
```



```
7150 struct {
7151   struct spinlock lock;
7152
7153   // input circular buffer
7154 #define INPUT_BUF_SIZE 128
7155   char buf[INPUT_BUF_SIZE];
7156   uint r; // Read index
7157   uint w; // Write index
7158   uint e; // Edit index
7159 } cons;
7160
7161 //
7162 // user write() system calls to the console go here.
7163 // uses sleep() and UART interrupts.
7164 //
7165 int
7166 consolewrite(int user_src, uint64 src, int n)
7167 {
7168   char buf[32]; // move batches from user space to uart.
7169   int i = 0;
7170
7171   while (i < n) {
7172     int nn = sizeof(buf);
7173     if (nn > n - i)
7174       nn = n - i;
7175     if (either_copyin(buf, user_src, src + i, nn) == -1)
7176       break;
7177     uartwrite(buf, nn);
7178     i += nn;
7179   }
7180
7181   return i;
7182 }
7183
7184 //
7185 // user read()s from the console go here.
7186 // copy (up to) a whole input line to dst.
7187 // user_dst indicates whether dst is a user
7188 // or kernel address.
7189 //
7190 int
7191 consoleread(int user_dst, uint64 dst, int n)
7192 {
7193   uint target;
7194   int c;
7195   char cbuf;
7196
7197   target = n;
7198   acquire(&cons.lock);
7199   while (n > 0) {
```

```
7200     // wait until interrupt handler has put some
7201     // input into cons.buffer.
7202     while (cons.r == cons.w) {
7203       if (killed(myproc())) {
7204         release(&cons.lock);
7205         return -1;
7206       }
7207       sleep(&cons.r, &cons.lock);
7208     }
7209
7210     c = cons.buf[cons.r++ % INPUT_BUF_SIZE];
7211
7212     if (c == C('D')) { // end-of-file
7213       if (n < target) {
7214         // Save ^D for next time, to make sure
7215         // caller gets a 0-byte result.
7216         cons.r--;
7217       }
7218       break;
7219     }
7220
7221     // copy the input byte to the user-space buffer.
7222     cbuf = c;
7223     if (either_copyout(user_dst, dst, &cbuf, 1) == -1)
7224       break;
7225
7226     dst++;
7227     --n;
7228
7229     if (c == '\n') {
7230       // a whole line has arrived, return to
7231       // the user-level read().
7232       break;
7233     }
7234   }
7235   release(&cons.lock);
7236
7237   return target - n;
7238 }
7239
7240
7241
7242
7243
7244
7245
7246
7247
7248
7249
```



```
7250 //
7251 // the console input interrupt handler.
7252 // uartintr() calls this for each input character.
7253 // do erase/kill processing, append to cons.buf,
7254 // wake up consoleread() if a whole line has arrived.
7255 //
7256 void
7257 consoleintr(int c)
7258 {
7259   acquire(&cons.lock);
7260
7261   switch (c) {
7262   case C('P'): // Print process list.
7263     procdump();
7264     break;
7265   case C('U'): // Kill line.
7266     while (cons.e != cons.w &&
7267            cons.buf[(cons.e - 1) % INPUT_BUF_SIZE] != '\n') {
7268       cons.e--;
7269       consputc(BACKSPACE);
7270     }
7271     break;
7272   case C('H'): // Backspace
7273   case '\x7f': // Delete key
7274     if (cons.e != cons.w) {
7275       cons.e--;
7276       consputc(BACKSPACE);
7277     }
7278     break;
7279   default:
7280     if (c != 0 && cons.e - cons.r < INPUT_BUF_SIZE) {
7281       c = (c == '\r') ? '\n' : c;
7282
7283       // echo back to the user.
7284       consputc(c);
7285
7286       // store for consumption by consoleread().
7287       cons.buf[cons.e++ % INPUT_BUF_SIZE] = c;
7288
7289       if (c == '\n' || c == C('D') || cons.e - cons.r == INPUT_BUF_SIZE) {
7290         // wake up consoleread() if a whole line (or end-of-file)
7291         // has arrived.
7292         cons.w = cons.e;
7293         wakeup(&cons.r);
7294       }
7295     }
7296     break;
7297   }
7298
7299
```

```
7300   release(&cons.lock);
7301 }
7302
7303 void
7304 consoleinit(void)
7305 {
7306   initlock(&cons.lock, "cons");
7307
7308   uartinit();
7309
7310   // connect read and write system calls
7311   // to consoleread and consolewrite.
7312   devsw[CONSOLE].read = consoleread;
7313   devsw[CONSOLE].write = consolewrite;
7314 }
7315
7316
7317
7318
7319
7320
7321
7322
7323
7324
7325
7326
7327
7328
7329
7330
7331
7332
7333
7334
7335
7336
7337
7338
7339
7340
7341
7342
7343
7344
7345
7346
7347
7348
7349
```



```
7350 //
7351 // low-level driver for 16550a UART.
7352 //
7353
7354 #include "types.h"
7355 #include "param.h"
7356 #include "memlayout.h"
7357 #include "riscv.h"
7358 #include "spinlock.h"
7359 #include "proc.h"
7360 #include "defs.h"
7361
7362 // the UART control registers are memory-mapped
7363 // at address UART0. this macro returns the
7364 // address of one of the registers.
7365 #define Reg(reg) ((volatile unsigned char *)(UART0 + (reg)))
7366
7367 #define ReadReg(reg)     (*(Reg(reg)))
7368 #define WriteReg(reg, v) (*(Reg(reg)) = (v))
7369
7370 // the UART control registers.
7371 // some have different meanings for read vs write.
7372 // see http://byterunner.com/16550.html
7373 #define RHR             0        // receive holding register (for input byte
7374 #define THR             0        // transmit holding register (for output by
7375 #define IER             1        // interrupt enable register
7376 #define IER_RX_ENABLE   (1 << 0) // receiver interrupts
7377 #define IER_TX_ENABLE   (1 << 1) // transmit interrupts
7378 #define FCR             2        // FIFO control register
7379 #define FCR_FIFO_ENABLE (1 << 0)
7380 #define FCR_FIFO_CLEAR  (3 << 1) // clear the content of the two FIFOs
7381 #define ISR             2        // interrupt status register
7382 #define LCR             3        // line control register
7383 #define LCR_EIGHT_BITS  (3 << 0)
7384 #define LCR_BAUD_LATCH  (1 << 7) // special mode to set baud rate
7385 #define LSR             5        // line status register
7386 #define LSR_RX_READY    (1 << 0) // input is waiting to be read from RHR
7387 #define LSR_TX_IDLE     (1 << 5) // THR can accept another character to send
7388
7389 // for sending threads to synchronize with uart "ready" interrupts.
7390 static struct spinlock tx_lock;
7391 static int tx_busy; // is the UART busy sending?
7392 static int tx_chan; // &tx_chan is the "wait channel"
7393
7394 extern volatile int panicking; // from printk.c
7395 extern volatile int panicked;  // from printk.c
7396
7397
7398
7399
```

```
7400 void
7401 uartinit(void)
7402 {
7403   // disable interrupts.
7404   WriteReg(IER, 0x00);
7405
7406   // special mode to set baud rate.
7407   WriteReg(LCR, LCR_BAUD_LATCH);
7408
7409   // LSB for baud rate of 38.4K.
7410   WriteReg(0, 0x03);
7411
7412   // MSB for baud rate of 38.4K.
7413   WriteReg(1, 0x00);
7414
7415   // leave set-baud mode,
7416   // and set word length to 8 bits, no parity.
7417   WriteReg(LCR, LCR_EIGHT_BITS);
7418
7419   // reset and enable FIFOs.
7420   WriteReg(FCR, FCR_FIFO_ENABLE | FCR_FIFO_CLEAR);
7421
7422   // enable transmit and receive interrupts.
7423   WriteReg(IER, IER_TX_ENABLE | IER_RX_ENABLE);
7424
7425   initlock(&tx_lock, "uart");
7426 }
7427
7428 // transmit buf[] to the uart. it blocks if the
7429 // uart is busy, so it cannot be called from
7430 // interrupts, only from write() system calls.
7431 void
7432 uartwrite(char buf[], int n)
7433 {
7434   acquire(&tx_lock);
7435
7436   int i = 0;
7437   while (i < n) {
7438     while (tx_busy != 0) {
7439       // wait for a UART transmit-complete interrupt
7440       // to set tx_busy to 0.
7441       sleep(&tx_chan, &tx_lock);
7442     }
7443
7444     WriteReg(THR, buf[i]);
7445     i += 1;
7446     tx_busy = 1;
7447   }
7448
7449
```



```
7450   release(&tx_lock);
7451 }
7452
7453 // write a byte to the uart without using
7454 // interrupts, for use by kernel printk() and
7455 // to echo characters. it spins waiting for the uart's
7456 // output register to be empty.
7457 void
7458 uartputc_sync(int c)
7459 {
7460   if (panicking == 0)
7461     push_off();
7462
7463   if (panicked) {
7464     for (;;)
7465       ;
7466   }
7467
7468   // wait for UART to set Transmit Holding Empty in LSR.
7469   while ((ReadReg(LSR) & LSR_TX_IDLE) == 0)
7470     ;
7471   WriteReg(THR, c);
7472
7473   if (panicking == 0)
7474     pop_off();
7475 }
7476
7477 // try to read one input character from the UART.
7478 // return -1 if none is waiting.
7479 static int
7480 uartgetc(void)
7481 {
7482   // is input ready?
7483   if (ReadReg(LSR) & LSR_RX_READY) {
7484     return ReadReg(RHR);
7485   } else {
7486     return -1;
7487   }
7488 }
7489
7490
7491
7492
7493
7494
7495
7496
7497
7498
7499
```

```
7500 // handle a uart interrupt, raised because input has
7501 // arrived, or the uart is ready for more output, or
7502 // both. called from devintr().
7503 void
7504 uartintr(void)
7505 {
7506   ReadReg(ISR); // acknowledge the interrupt
7507
7508   acquire(&tx_lock);
7509   if (ReadReg(LSR) & LSR_TX_IDLE) {
7510     // UART finished transmitting; wake up sending thread.
7511     tx_busy = 0;
7512     wakeup(&tx_chan);
7513   }
7514   release(&tx_lock);
7515
7516   // read and process incoming characters, if any.
7517   while (1) {
7518     int c = uartgetc();
7519     if (c == -1)
7520       break;
7521     consoleintr(c);
7522   }
7523 }
7524
7525
7526
7527
7528
7529
7530
7531
7532
7533
7534
7535
7536
7537
7538
7539
7540
7541
7542
7543
7544
7545
7546
7547
7548
7549
```



```
7550 //
7551 // driver for qemu's virtio disk device.
7552 // uses qemu's mmio interface to virtio.
7553 //
7554 // qemu ... -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-d
7555 //
7556
7557 #include "types.h"
7558 #include "riscv.h"
7559 #include "defs.h"
7560 #include "param.h"
7561 #include "memlayout.h"
7562 #include "spinlock.h"
7563 #include "sleeplock.h"
7564 #include "fs.h"
7565 #include "buf.h"
7566 #include "virtio.h"
7567
7568 // the address of virtio mmio register r.
7569 #define R(r) ((volatile uint32 *)(VIRTIO0 + (r)))
7570
7571 static struct disk {
7572   // a set (not a ring) of DMA descriptors, with which the
7573   // driver tells the device where to read and write individual
7574   // disk operations. there are NUM descriptors.
7575   // most commands consist of a "chain" (a linked list) of a couple of
7576   // these descriptors.
7577   struct virtq_desc *desc;
7578
7579   // a ring in which the driver writes descriptor numbers
7580   // that the driver would like the device to process.  it only
7581   // includes the head descriptor of each chain. the ring has
7582   // NUM elements.
7583   struct virtq_avail *avail;
7584
7585   // a ring in which the device writes descriptor numbers that
7586   // the device has finished processing (just the head of each chain).
7587   // there are NUM used ring entries.
7588   struct virtq_used *used;
7589
7590   // our own book-keeping.
7591   char free[NUM];  // is a descriptor free?
7592   uint16 used_idx; // we've looked this far in used[2..NUM].
7593
7594   // track info about in-flight operations,
7595   // for use when completion interrupt arrives.
7596   // indexed by first descriptor index of chain.
7597   struct {
7598     struct buf *b;
7599     char status;
```

```
7600   } info[NUM];
7601
7602   // disk command headers.
7603   // one-for-one with descriptors, for convenience.
7604   struct virtio_blk_req ops[NUM];
7605
7606   struct spinlock vdisk_lock;
7607
7608 } disk;
7609
7610 void
7611 virtio_disk_init(void)
7612 {
7613   uint32 status = 0;
7614
7615   initlock(&disk.vdisk_lock, "virtio_disk");
7616
7617   if (*R(VIRTIO_MMIO_MAGIC_VALUE) != 0x74726976 ||
7618       *R(VIRTIO_MMIO_VERSION) != 2 || *R(VIRTIO_MMIO_DEVICE_ID) != 2 ||
7619       *R(VIRTIO_MMIO_VENDOR_ID) != 0x554d4551) {
7620     panic("could not find virtio disk");
7621   }
7622
7623   // reset device
7624   *R(VIRTIO_MMIO_STATUS) = status;
7625
7626   // set ACKNOWLEDGE status bit
7627   status |= VIRTIO_CONFIG_S_ACKNOWLEDGE;
7628   *R(VIRTIO_MMIO_STATUS) = status;
7629
7630   // set DRIVER status bit
7631   status |= VIRTIO_CONFIG_S_DRIVER;
7632   *R(VIRTIO_MMIO_STATUS) = status;
7633
7634   // negotiate features
7635   uint64 features = *R(VIRTIO_MMIO_DEVICE_FEATURES);
7636   features &= ~(1 << VIRTIO_BLK_F_RO);
7637   features &= ~(1 << VIRTIO_BLK_F_SCSI);
7638   features &= ~(1 << VIRTIO_BLK_F_CONFIG_WCE);
7639   features &= ~(1 << VIRTIO_BLK_F_MQ);
7640   features &= ~(1 << VIRTIO_F_ANY_LAYOUT);
7641   features &= ~(1 << VIRTIO_RING_F_EVENT_IDX);
7642   features &= ~(1 << VIRTIO_RING_F_INDIRECT_DESC);
7643   *R(VIRTIO_MMIO_DRIVER_FEATURES) = features;
7644
7645   // tell device that feature negotiation is complete.
7646   status |= VIRTIO_CONFIG_S_FEATURES_OK;
7647   *R(VIRTIO_MMIO_STATUS) = status;
7648
7649
```



```
7650   // re-read status to ensure FEATURES_OK is set.
7651   status = *R(VIRTIO_MMIO_STATUS);
7652   if (!(status & VIRTIO_CONFIG_S_FEATURES_OK))
7653     panic("virtio disk FEATURES_OK unset");
7654
7655   // initialize queue 0.
7656   *R(VIRTIO_MMIO_QUEUE_SEL) = 0;
7657
7658   // ensure queue 0 is not in use.
7659   if (*R(VIRTIO_MMIO_QUEUE_READY))
7660     panic("virtio disk should not be ready");
7661
7662   // check maximum queue size.
7663   uint32 max = *R(VIRTIO_MMIO_QUEUE_NUM_MAX);
7664   if (max == 0)
7665     panic("virtio disk has no queue 0");
7666   if (max < NUM)
7667     panic("virtio disk max queue too short");
7668
7669   // allocate and zero queue memory.
7670   disk.desc = kalloc();
7671   disk.avail = kalloc();
7672   disk.used = kalloc();
7673   if (!disk.desc || !disk.avail || !disk.used)
7674     panic("virtio disk kalloc");
7675   memset(disk.desc, 0, PGSIZE);
7676   memset(disk.avail, 0, PGSIZE);
7677   memset(disk.used, 0, PGSIZE);
7678
7679   // set queue size.
7680   *R(VIRTIO_MMIO_QUEUE_NUM) = NUM;
7681
7682   // write physical addresses.
7683   *R(VIRTIO_MMIO_QUEUE_DESC_LOW) = (uint64)disk.desc;
7684   *R(VIRTIO_MMIO_QUEUE_DESC_HIGH) = (uint64)disk.desc >> 32;
7685   *R(VIRTIO_MMIO_DRIVER_DESC_LOW) = (uint64)disk.avail;
7686   *R(VIRTIO_MMIO_DRIVER_DESC_HIGH) = (uint64)disk.avail >> 32;
7687   *R(VIRTIO_MMIO_DEVICE_DESC_LOW) = (uint64)disk.used;
7688   *R(VIRTIO_MMIO_DEVICE_DESC_HIGH) = (uint64)disk.used >> 32;
7689
7690   // queue is ready.
7691   *R(VIRTIO_MMIO_QUEUE_READY) = 0x1;
7692
7693   // all NUM descriptors start out unused.
7694   for (int i = 0; i < NUM; i++)
7695     disk.free[i] = 1;
7696
7697   // tell device we're completely ready.
7698   status |= VIRTIO_CONFIG_S_DRIVER_OK;
7699   *R(VIRTIO_MMIO_STATUS) = status;
```

```
7700   // plic.c and trap.c arrange for interrupts from VIRTIO0_IRQ.
7701 }
7702
7703 // find a free descriptor, mark it non-free, return its index.
7704 static int
7705 alloc_desc()
7706 {
7707   for (int i = 0; i < NUM; i++) {
7708     if (disk.free[i]) {
7709       disk.free[i] = 0;
7710       return i;
7711     }
7712   }
7713   return -1;
7714 }
7715
7716 // mark a descriptor as free.
7717 static void
7718 free_desc(int i)
7719 {
7720   if (i >= NUM)
7721     panic("free_desc 1");
7722   if (disk.free[i])
7723     panic("free_desc 2");
7724   disk.desc[i].addr = 0;
7725   disk.desc[i].len = 0;
7726   disk.desc[i].flags = 0;
7727   disk.desc[i].next = 0;
7728   disk.free[i] = 1;
7729   wakeup(&disk.free[0]);
7730 }
7731
7732 // free a chain of descriptors.
7733 static void
7734 free_chain(int i)
7735 {
7736   while (1) {
7737     int flag = disk.desc[i].flags;
7738     int nxt = disk.desc[i].next;
7739     free_desc(i);
7740     if (flag & VRING_DESC_F_NEXT)
7741       i = nxt;
7742     else
7743       break;
7744   }
7745 }
7746
7747
7748
7749
```



```
7750 // allocate three descriptors (they need not be contiguous).
7751 // disk transfers always use three descriptors.
7752 static int
7753 alloc3_desc(int *idx)
7754 {
7755   for (int i = 0; i < 3; i++) {
7756     idx[i] = alloc_desc();
7757     if (idx[i] < 0) {
7758       for (int j = 0; j < i; j++)
7759         free_desc(idx[j]);
7760       return -1;
7761     }
7762   }
7763   return 0;
7764 }
7765
7766 void
7767 virtio_disk_rw(struct buf *b, int write)
7768 {
7769   uint64 sector = b->blockno * (BSIZE / 512);
7770
7771   acquire(&disk.vdisk_lock);
7772
7773   // the spec's Section 5.2 says that legacy block operations use
7774   // three descriptors: one for type/reserved/sector, one for the
7775   // data, one for a 1-byte status result.
7776
7777   // allocate the three descriptors.
7778   int idx[3];
7779   while (1) {
7780     if (alloc3_desc(idx) == 0) {
7781       break;
7782     }
7783     sleep(&disk.free[0], &disk.vdisk_lock);
7784   }
7785
7786   // format the three descriptors.
7787   // qemu's virtio-blk.c reads them.
7788
7789   struct virtio_blk_req *buf0 = &disk.ops[idx[0]];
7790
7791   if (write)
7792     buf0->type = VIRTIO_BLK_T_OUT; // write the disk
7793   else
7794     buf0->type = VIRTIO_BLK_T_IN; // read the disk
7795   buf0->reserved = 0;
7796   buf0->sector = sector;
7797
7798
7799
```

```
7800   disk.desc[idx[0]].addr = (uint64)buf0;
7801   disk.desc[idx[0]].len = sizeof(struct virtio_blk_req);
7802   disk.desc[idx[0]].flags = VRING_DESC_F_NEXT;
7803   disk.desc[idx[0]].next = idx[1];
7804
7805   disk.desc[idx[1]].addr = (uint64)b->data;
7806   disk.desc[idx[1]].len = BSIZE;
7807   if (write)
7808     disk.desc[idx[1]].flags = 0; // device reads b->data
7809   else
7810     disk.desc[idx[1]].flags = VRING_DESC_F_WRITE; // device writes b->data
7811   disk.desc[idx[1]].flags |= VRING_DESC_F_NEXT;
7812   disk.desc[idx[1]].next = idx[2];
7813
7814   disk.info[idx[0]].status = 0xff; // device writes 0 on success
7815   disk.desc[idx[2]].addr = (uint64)&disk.info[idx[0]].status;
7816   disk.desc[idx[2]].len = 1;
7817   disk.desc[idx[2]].flags = VRING_DESC_F_WRITE; // device writes the status
7818   disk.desc[idx[2]].next = 0;
7819
7820   // record struct buf for virtio_disk_intr().
7821   b->disk = 1;
7822   disk.info[idx[0]].b = b;
7823
7824   // tell the device the first index in our chain of descriptors.
7825   disk.avail->ring[disk.avail->idx % NUM] = idx[0];
7826
7827   io_fence();
7828
7829   // tell the device another avail ring entry is available.
7830   disk.avail->idx += 1; // not % NUM ...
7831
7832   io_fence();
7833
7834   *R(VIRTIO_MMIO_QUEUE_NOTIFY) = 0; // value is queue number
7835
7836   // Wait for virtio_disk_intr() to say request has finished.
7837   while (b->disk == 1) {
7838     sleep(b, &disk.vdisk_lock);
7839   }
7840
7841   disk.info[idx[0]].b = 0;
7842   free_chain(idx[0]);
7843
7844   release(&disk.vdisk_lock);
7845 }
7846
7847
7848
7849
```



```
7850 void
7851 virtio_disk_intr()
7852 {
7853   acquire(&disk.vdisk_lock);
7854
7855   // the device won't raise another interrupt until we tell it
7856   // we've seen this interrupt, which the following line does.
7857   // this may race with the device writing new entries to
7858   // the "used" ring, in which case we may process the new
7859   // completion entries in this interrupt, and have nothing to do
7860   // in the next interrupt, which is harmless.
7861   *R(VIRTIO_MMIO_INTERRUPT_ACK) = *R(VIRTIO_MMIO_INTERRUPT_STATUS) & 0x3;
7862
7863   io_fence();
7864
7865   // the device increments disk.used->idx when it
7866   // adds an entry to the used ring.
7867
7868   while (disk.used_idx != disk.used->idx) {
7869     io_fence();
7870     int id = disk.used->ring[disk.used_idx % NUM].id;
7871
7872     if (disk.info[id].status != 0)
7873       panic("virtio_disk_intr status");
7874
7875     struct buf *b = disk.info[id].b;
7876     b->disk = 0; // disk is done with buf
7877     wakeup(b);
7878
7879     disk.used_idx += 1;
7880   }
7881
7882   release(&disk.vdisk_lock);
7883 }
7884
7885
7886
7887
7888
7889
7890
7891
7892
7893
7894
7895
7896
7897
7898
7899
```

```
// init: The initial user-level program

#include "kernel/types.h"
#include "kernel/stat.h"
#include "kernel/spinlock.h"
#include "kernel/sleeplock.h"
#include "kernel/fs.h"
#include "kernel/file.h"
#include "user/user.h"
#include "kernel/fcntl.h"

char *argv[] = {"sh", 0};

int
main(void)
{
  int pid, wpid;

  if (open("console", O_RDWR) < 0) {
    mknod("console", CONSOLE, 0);
    open("console", O_RDWR);
  }
  dup(0); // stdout
  dup(0); // stderr

  for (;;) {
    printf("init: starting sh\n");
    pid = fork();
    if (pid < 0) {
      printf("init: fork failed\n");
      exit(1);
    }
    if (pid == 0) {
      exec("sh", argv);
      printf("init: exec sh failed\n");
      exit(1);
    }

    for (;;) {
      // this call to wait() returns if the shell exits,
      // or if a parentless process exits.
      wpid = wait((int *)0);
      if (wpid == pid) {
        // the shell exited; restart it.
        break;
      } else if (wpid < 0) {
        printf("init: wait returned an error\n");
        exit(1);
      } else {
        // it was a parentless process; do nothing.
```



```
      }
    }
  }
}
```

```
8000 // Shell.
8001
8002 #include "kernel/types.h"
8003 #include "user/user.h"
8004 #include "kernel/fcntl.h"
8005
8006 // Parsed command representation
8007 #define EXEC  1
8008 #define REDIR 2
8009 #define PIPE  3
8010 #define LIST  4
8011 #define BACK  5
8012
8013 #define MAXARGS 10
8014
8015 struct cmd {
8016   int type;
8017 };
8018
8019 struct execcmd {
8020   int type;
8021   char *argv[MAXARGS];
8022   char *eargv[MAXARGS];
8023 };
8024
8025 struct redircmd {
8026   int type;
8027   struct cmd *cmd;
8028   char *file;
8029   char *efile;
8030   int mode;
8031   int fd;
8032 };
8033
8034 struct pipecmd {
8035   int type;
8036   struct cmd *left;
8037   struct cmd *right;
8038 };
8039
8040 struct listcmd {
8041   int type;
8042   struct cmd *left;
8043   struct cmd *right;
8044 };
8045
8046 struct backcmd {
8047   int type;
8048   struct cmd *cmd;
8049 };
```



```
8050 int fork1(void); // Fork but panics on failure.
8051 void panic(char *);
8052 struct cmd *parsecmd(char *);
8053 void runcmd(struct cmd *) __attribute__((noreturn));
8054
8055 // Execute cmd.  Never returns.
8056 void
8057 runcmd(struct cmd *cmd)
8058 {
8059   int p[2];
8060   struct backcmd *bcmd;
8061   struct execcmd *ecmd;
8062   struct listcmd *lcmd;
8063   struct pipecmd *pcmd;
8064   struct redircmd *rcmd;
8065
8066   if (cmd == 0)
8067     exit(1);
8068
8069   switch (cmd->type) {
8070   default:
8071     panic("runcmd");
8072
8073   case EXEC:
8074     ecmd = (struct execcmd *)cmd;
8075     if (ecmd->argv[0] == 0)
8076       exit(1);
8077     exec(ecmd->argv[0], ecmd->argv);
8078     fprintf(2, "exec %s failed\n", ecmd->argv[0]);
8079     break;
8080
8081   case REDIR:
8082     rcmd = (struct redircmd *)cmd;
8083     close(rcmd->fd);
8084     if (open(rcmd->file, rcmd->mode) < 0) {
8085       fprintf(2, "open %s failed\n", rcmd->file);
8086       exit(1);
8087     }
8088     runcmd(rcmd->cmd);
8089     break;
8090
8091   case LIST:
8092     lcmd = (struct listcmd *)cmd;
8093     if (fork1() == 0)
8094       runcmd(lcmd->left);
8095     wait(0);
8096     runcmd(lcmd->right);
8097     break;
8098
8099
```

```
8100   case PIPE:
8101     pcmd = (struct pipecmd *)cmd;
8102     if (pipe(p) < 0)
8103       panic("pipe");
8104     if (fork1() == 0) {
8105       close(1);
8106       dup(p[1]);
8107       close(p[0]);
8108       close(p[1]);
8109       runcmd(pcmd->left);
8110     }
8111     if (fork1() == 0) {
8112       close(0);
8113       dup(p[0]);
8114       close(p[0]);
8115       close(p[1]);
8116       runcmd(pcmd->right);
8117     }
8118     close(p[0]);
8119     close(p[1]);
8120     wait(0);
8121     wait(0);
8122     break;
8123
8124   case BACK:
8125     bcmd = (struct backcmd *)cmd;
8126     if (fork1() == 0)
8127       runcmd(bcmd->cmd);
8128     break;
8129   }
8130   exit(0);
8131 }
8132
8133 int
8134 getcmd(char *buf, int nbuf)
8135 {
8136   write(2, "$ ", 2);
8137   memset(buf, 0, nbuf);
8138   gets(buf, nbuf);
8139   if (buf[0] == 0) // EOF
8140     return -1;
8141   return 0;
8142 }
8143
8144
8145
8146
8147
8148
8149
```



```
8150 int
8151 main(void)
8152 {
8153   static char buf[100];
8154   int fd;
8155
8156   // Ensure that three file descriptors are open.
8157   while ((fd = open("console", O_RDWR)) >= 0) {
8158     if (fd >= 3) {
8159       close(fd);
8160       break;
8161     }
8162   }
8163
8164   // Read and run input commands.
8165   while (getcmd(buf, sizeof(buf)) >= 0) {
8166     char *cmd = buf;
8167     while (*cmd == ' ' || *cmd == '\t')
8168       cmd++;
8169     if (*cmd == '\n') // is a blank command
8170       continue;
8171     if (cmd[0] == 'c' && cmd[1] == 'd' && cmd[2] == ' ') {
8172       // Chdir must be called by the parent, not the child.
8173       cmd[strlen(cmd) - 1] = 0; // chop \n
8174       if (chdir(cmd + 3) < 0)
8175         fprintf(2, "cannot cd %s\n", cmd + 3);
8176     } else {
8177       if (fork1() == 0)
8178         runcmd(parsecmd(cmd));
8179       wait(0);
8180     }
8181   }
8182   exit(0);
8183 }
8184
8185 void
8186 panic(char *s)
8187 {
8188   fprintf(2, "%s\n", s);
8189   exit(1);
8190 }
8191
8192
8193
8194
8195
8196
8197
8198
8199
```

```
8200 int
8201 fork1(void)
8202 {
8203   int pid;
8204
8205   pid = fork();
8206   if (pid == -1)
8207     panic("fork");
8208   return pid;
8209 }
8210
8211
8212
8213
8214
8215
8216
8217
8218
8219
8220
8221
8222
8223
8224
8225
8226
8227
8228
8229
8230
8231
8232
8233
8234
8235
8236
8237
8238
8239
8240
8241
8242
8243
8244
8245
8246
8247
8248
8249
```



```
8250 // Constructors
8251
8252 struct cmd *
8253 execcmd(void)
8254 {
8255   struct execcmd *cmd;
8256
8257   cmd = malloc(sizeof(*cmd));
8258   memset(cmd, 0, sizeof(*cmd));
8259   cmd->type = EXEC;
8260   return (struct cmd *)cmd;
8261 }
8262
8263 struct cmd *
8264 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
8265 {
8266   struct redircmd *cmd;
8267
8268   cmd = malloc(sizeof(*cmd));
8269   memset(cmd, 0, sizeof(*cmd));
8270   cmd->type = REDIR;
8271   cmd->cmd = subcmd;
8272   cmd->file = file;
8273   cmd->efile = efile;
8274   cmd->mode = mode;
8275   cmd->fd = fd;
8276   return (struct cmd *)cmd;
8277 }
8278
8279 struct cmd *
8280 pipecmd(struct cmd *left, struct cmd *right)
8281 {
8282   struct pipecmd *cmd;
8283
8284   cmd = malloc(sizeof(*cmd));
8285   memset(cmd, 0, sizeof(*cmd));
8286   cmd->type = PIPE;
8287   cmd->left = left;
8288   cmd->right = right;
8289   return (struct cmd *)cmd;
8290 }
8291
8292
8293
8294
8295
8296
8297
8298
8299
```

```
8300 struct cmd *
8301 listcmd(struct cmd *left, struct cmd *right)
8302 {
8303   struct listcmd *cmd;
8304
8305   cmd = malloc(sizeof(*cmd));
8306   memset(cmd, 0, sizeof(*cmd));
8307   cmd->type = LIST;
8308   cmd->left = left;
8309   cmd->right = right;
8310   return (struct cmd *)cmd;
8311 }
8312
8313 struct cmd *
8314 backcmd(struct cmd *subcmd)
8315 {
8316   struct backcmd *cmd;
8317
8318   cmd = malloc(sizeof(*cmd));
8319   memset(cmd, 0, sizeof(*cmd));
8320   cmd->type = BACK;
8321   cmd->cmd = subcmd;
8322   return (struct cmd *)cmd;
8323 }
8324
8325
8326
8327
8328
8329
8330
8331
8332
8333
8334
8335
8336
8337
8338
8339
8340
8341
8342
8343
8344
8345
8346
8347
8348
8349
```



```
8350 // Parsing
8351
8352 char whitespace[] = " \t\r\n\v";
8353 char symbols[] = "<|>&;()";
8354
8355 int
8356 gettoken(char **ps, char *es, char **q, char **eq)
8357 {
8358   char *s;
8359   int ret;
8360
8361   s = *ps;
8362   while (s < es && strchr(whitespace, *s))
8363     s++;
8364   if (q)
8365     *q = s;
8366   ret = *s;
8367   switch (*s) {
8368   case 0:
8369     break;
8370   case '|':
8371   case '(':
8372   case ')':
8373   case ';':
8374   case '&':
8375   case '<':
8376     s++;
8377     break;
8378   case '>':
8379     s++;
8380     if (*s == '>') {
8381       ret = '+';
8382       s++;
8383     }
8384     break;
8385   default:
8386     ret = 'a';
8387     while (s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
8388       s++;
8389     break;
8390   }
8391   if (eq)
8392     *eq = s;
8393
8394   while (s < es && strchr(whitespace, *s))
8395     s++;
8396   *ps = s;
8397   return ret;
8398 }
8399
```

```
8400 int
8401 peek(char **ps, char *es, char *toks)
8402 {
8403   char *s;
8404
8405   s = *ps;
8406   while (s < es && strchr(whitespace, *s))
8407     s++;
8408   *ps = s;
8409   return *s && strchr(toks, *s);
8410 }
8411
8412 struct cmd *parseline(char **, char *);
8413 struct cmd *parsepipe(char **, char *);
8414 struct cmd *parseexec(char **, char *);
8415 struct cmd *nulterminate(struct cmd *);
8416
8417 struct cmd *
8418 parsecmd(char *s)
8419 {
8420   char *es;
8421   struct cmd *cmd;
8422
8423   es = s + strlen(s);
8424   cmd = parseline(&s, es);
8425   peek(&s, es, "");
8426   if (s != es) {
8427     fprintf(2, "leftovers: %s\n", s);
8428     panic("syntax");
8429   }
8430   nulterminate(cmd);
8431   return cmd;
8432 }
8433
8434 struct cmd *
8435 parseline(char **ps, char *es)
8436 {
8437   struct cmd *cmd;
8438
8439   cmd = parsepipe(ps, es);
8440   while (peek(ps, es, "&")) {
8441     gettoken(ps, es, 0, 0);
8442     cmd = backcmd(cmd);
8443   }
8444   if (peek(ps, es, ";")) {
8445     gettoken(ps, es, 0, 0);
8446     cmd = listcmd(cmd, parseline(ps, es));
8447   }
8448   return cmd;
8449 }
```



```
8450 struct cmd *
8451 parsepipe(char **ps, char *es)
8452 {
8453   struct cmd *cmd;
8454
8455   cmd = parseexec(ps, es);
8456   if (peek(ps, es, "|")) {
8457     gettoken(ps, es, 0, 0);
8458     cmd = pipecmd(cmd, parsepipe(ps, es));
8459   }
8460   return cmd;
8461 }
8462
8463 struct cmd *
8464 parseredirs(struct cmd *cmd, char **ps, char *es)
8465 {
8466   int tok;
8467   char *q, *eq;
8468
8469   while (peek(ps, es, "<>")) {
8470     tok = gettoken(ps, es, 0, 0);
8471     if (gettoken(ps, es, &q, &eq) != 'a')
8472       panic("missing file for redirection");
8473     switch (tok) {
8474     case '<':
8475       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
8476       break;
8477     case '>':
8478       cmd = redircmd(cmd, q, eq, O_WRONLY | O_CREATE | O_TRUNC, 1);
8479       break;
8480     case '+': // >>
8481       cmd = redircmd(cmd, q, eq, O_WRONLY | O_CREATE, 1);
8482       break;
8483     }
8484   }
8485   return cmd;
8486 }
8487
8488
8489
8490
8491
8492
8493
8494
8495
8496
8497
8498
8499
```

```
8500 struct cmd *
8501 parseblock(char **ps, char *es)
8502 {
8503   struct cmd *cmd;
8504
8505   if (!peek(ps, es, "("))
8506     panic("parseblock");
8507   gettoken(ps, es, 0, 0);
8508   cmd = parseline(ps, es);
8509   if (!peek(ps, es, ")"))
8510     panic("syntax - missing )");
8511   gettoken(ps, es, 0, 0);
8512   cmd = parseredirs(cmd, ps, es);
8513   return cmd;
8514 }
8515
8516 struct cmd *
8517 parseexec(char **ps, char *es)
8518 {
8519   char *q, *eq;
8520   int tok, argc;
8521   struct execcmd *cmd;
8522   struct cmd *ret;
8523
8524   if (peek(ps, es, "("))
8525     return parseblock(ps, es);
8526
8527   ret = execcmd();
8528   cmd = (struct execcmd *)ret;
8529
8530   argc = 0;
8531   ret = parseredirs(ret, ps, es);
8532   while (!peek(ps, es, "|)&;")) {
8533     if ((tok = gettoken(ps, es, &q, &eq)) == 0)
8534       break;
8535     if (tok != 'a')
8536       panic("syntax");
8537     cmd->argv[argc] = q;
8538     cmd->eargv[argc] = eq;
8539     argc++;
8540     if (argc >= MAXARGS)
8541       panic("too many args");
8542     ret = parseredirs(ret, ps, es);
8543   }
8544   cmd->argv[argc] = 0;
8545   cmd->eargv[argc] = 0;
8546   return ret;
8547 }
8548
8549
```



```
8550 // NUL-terminate all the counted strings.
8551 struct cmd *
8552 nulterminate(struct cmd *cmd)
8553 {
8554   int i;
8555   struct backcmd *bcmd;
8556   struct execcmd *ecmd;
8557   struct listcmd *lcmd;
8558   struct pipecmd *pcmd;
8559   struct redircmd *rcmd;
8560
8561   if (cmd == 0)
8562     return 0;
8563
8564   switch (cmd->type) {
8565   case EXEC:
8566     ecmd = (struct execcmd *)cmd;
8567     for (i = 0; ecmd->argv[i]; i++)
8568       *ecmd->eargv[i] = 0;
8569     break;
8570
8571   case REDIR:
8572     rcmd = (struct redircmd *)cmd;
8573     nulterminate(rcmd->cmd);
8574     *rcmd->efile = 0;
8575     break;
8576
8577   case PIPE:
8578     pcmd = (struct pipecmd *)cmd;
8579     nulterminate(pcmd->left);
8580     nulterminate(pcmd->right);
8581     break;
8582
8583   case LIST:
8584     lcmd = (struct listcmd *)cmd;
8585     nulterminate(lcmd->left);
8586     nulterminate(lcmd->right);
8587     break;
8588
8589   case BACK:
8590     bcmd = (struct backcmd *)cmd;
8591     nulterminate(bcmd->cmd);
8592     break;
8593   }
8594   return cmd;
8595 }
8596
8597
8598
8599
```

```
8600 OUTPUT_ARCH( "riscv" )
8601 ENTRY( _entry )
8602
8603 SECTIONS
8604 {
8605   /*
8606    * ensure that entry.S / _entry is at 0x80000000,
8607    * where qemu's -kernel jumps.
8608    */
8609   . = 0x80000000;
8610
8611   .text : {
8612     kernel/entry.o(_entry)
8613     *(.text .text.*)
8614     . = ALIGN(0x1000);
8615     _trampoline = .;
8616     *(trampsec)
8617     . = ALIGN(0x1000);
8618     ASSERT(. - _trampoline == 0x1000, "error: trampoline larger than one pag
8619     PROVIDE(etext = .);
8620   }
8621
8622   .rodata : {
8623     . = ALIGN(16);
8624     *(.srodata .srodata.*) /* do not need to distinguish this from .rodata *
8625     . = ALIGN(16);
8626     *(.rodata .rodata.*)
8627   }
8628
8629   .data : {
8630     . = ALIGN(16);
8631     *(.sdata .sdata.*) /* do not need to distinguish this from .data */
8632     . = ALIGN(16);
8633     *(.data .data.*)
8634   }
8635
8636   .bss : {
8637     . = ALIGN(16);
8638     *(.sbss .sbss.*) /* do not need to distinguish this from .bss */
8639     . = ALIGN(16);
8640     *(.bss .bss.*)
8641   }
8642
8643   PROVIDE(end = .);
8644 }
8645
8646
8647
8648
8649
```