xv6 代码阅读报告 #1

王集 2020012389

操作系统接口

xv6 中提供了大量的系统调用函数，在这一节我将叙述其中频繁使用的几个。

fork

使用 fork() 函数创建一个新的进程。这个函数调用非同寻常，对于父进程，它返回子进程的pid，对于子进程，它返回0。我们通过它的返回值来判断目前位于哪个进程，然后针对性地展开新的操作。

```
int pid = fork()
if(pid > 0){
    printf("parent: child=%d\n", pid);
    pid = wait((int *) 0);
    printf("child %d is done\n", pid);
} else if(pid == 0){
    printf("child: exiting\n");
    exit(0);
} else {
    printf("fork error\n");
}
```

例如上面这段代码，父进程会返回子进程的id，这个值是大于零的，因此会进入if的第一个分支里。随后打出 parent: child=%d 。

之后，调用 wait(int* status) 系统调用。这个调用使父进程等待子进程结束，返回值为结束的进程pid，status为子进程exit(int)传入的值。子进程正常结束会传入0，异常的时候传入程序员精心设计的值。这样在调试的时候，我们可以知道出了什么问题。

另外，父进程可以等子进程，子进程等不了父进程。

轮到子进程的时候，子进程会打出 child: exiting\n 然后以exit(0)退出。

子进程和父进程谁先执行，谁后执行是不确定的。因此那两个printf的输出顺序不是完全确定的。

exec

exec 系统调用用于启动一个新的程序。它接受两个参数，第一个是可执行文件的名字，第二个是传进程序的参数——一个字符串的数组argv。

argv中的第一个元素通常为运行的程序的名字，之后的元素才是真正的参数。argv的最后一个元素要是0，标志着参数的终结。运行echo的一段代码示例如下：

```
char *argv[3];
argv[0] = "echo";
argv[1] = "hello";
argv[2] = 0;
exec("/bin/echo", argv);
printf("exec error\n");
```

假如exec成功的话，进程的内存会被新打开的代码片段所覆盖。因此，exec后面的代码将不会被执行。如果被执行，一定是exec不成功。

I/O

文件标识符是个小整数，用来标示一个进程读写的对象。默认0代表标准输入，进程可以从标准输入里读取信息；1代表标准输出，就是printf默认输出的地方；2代表标准错误。对于不同的进程，文件标识符的意义是不同的。

使用 open(char* file, int flags) 打开一个文件，它会创建/打开一个文件，然后返回一个文件标识符（-1代表错误）。这个文件标识符会占用目前最小的未被占用的文件标识符。
flags可以指定打开文件的类型，列举如下：

1）O_RDONLY：以只读方式打开；
2）O_WRONLY：以只写方式打开；
3）O_RDWR ：以读写方式打开。
可选模式（用按位或 | 来操作）：
4）O_APPEND：把写入数据追加在文件的末尾；
5）O_TRUNC：把文件长度设置为零，丢弃已有的内容；

6）O_CREAT：如果需要，就按照参数 mode 中给出的访问模式创建文件；
7）O_EXCL：与 O_CREAT 一起使用，确保调用者创建出文件。使用这个模式可以防止两个程序同时创建同一个文件，如果文件已经存在，open 调用将失败。

使用 close() 可以关闭一个文件。

使用 write(int fd, char *buf, int n) 可以向fd中写入n个字符，这n个字符是字符串buf。

使用 int read(int fd, char *buf, int n) 可以从fd中读入n个字符，这n个字符会被存进字符串buf。

使用 dup(int fd) 可以复制新建返回一个文件标识符，指向的文件与fd相同。

重定向在后面Shell部分讲。

pipe

管道用于两个进程之间的文件传输。pipe[1]是写端，而pipe[0]是读端。

示例代码：

```
int p[2];
char *argv[2];
argv[0] = "wc";
argv[1] = 0;
pipe(p); // 新建管道
if(fork() == 0) { // 子进程
    close(0); // 关闭标准输入
    dup(p[0]); // 复制管道的读端，返回0，意味着程序会从管道里读取数据，而不是从标准输入
    close(p[0]); // 关闭管道的读端，因为0已经是读端了，不需要保存两份fd
    close(p[1]); // 关闭管道的写端，为了确保read可以return，否则当无数据读入的时候，read有可能会卡住，一直读取。
    exec("/bin/wc", argv);
} else {
    close(p[0]); // 关闭管道的读端
    write(p[1], "hello world\n", 12); // 写入
    close(p[1]); // 关闭管道的写端
}
```

文件系统

文件目录就像一棵树。

chdir(char* dir) 调用可以改变目前的工作目录。

mkdir(char* dir) 调用可以创建新的目录

mknod(*char* file, int, int) 可以创建一个新的设备

int fstat(int fd, struct stat *st) 向st中存入一个fd对应文件相关的信息。

int link(char *file1, char* file2) 让字符串file1和file2指向同一个文件。

int unlink(char *file) 解除字符串file和文件的绑定关系

Shell

Shell的主体运行结构非常简单，就是无限循环，等待命令，然后运行命令。然而，其中有不少细节值得注意。

```
//这个panic函数是一个辅助函数，用来报错，退出程序。
void panic(char *s)
...

int
main(void)
{
  static char buf[100];
  int fd;

  // 确保打开了三个文件标识符（012）
  while((fd = open("console", O_RDWR)) >= 0){
    if(fd >= 3){
      close(fd);
      break;
    }
  }

  // 读取和运行指令
```

```
  while(getcmd(buf, sizeof(buf)) >= 0){
    if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
      // Chdir 必须被父进程，也就是Shell调用，因为是Shell自己改变工作路径
      buf[strlen(buf)-1] = 0;  // 切掉\n
      if(chdir(buf+3) < 0) // 切掉"cd "
        fprintf(2, "cannot cd %s\n", buf+3);
      continue;
    }
    if(fork1() == 0) // 包装过的fork，其效果就是错误的时候panic
      runcmd(parsecmd(buf)); //解析，然后运行命令
    wait(0); //等待命令运行完成
  }
  exit(0);
}
```

Shell运行命令

说明内嵌在下面的代码中。

```
void
runcmd(struct cmd *cmd)
{
// 一些初始化
  int p[2];
  struct backcmd *bcmd;
  struct execcmd *ecmd;
  struct listcmd *lcmd;
  struct pipecmd *pcmd;
  struct redircmd *rcmd;

// 处理异常情况
  if(cmd == 0)
    exit(1);

  switch(cmd->type){
  default:
    panic("runcmd");

/*****************************************************************
*****
*  Shell一共处理五种命令，在这里枚举为 EXEC, REDIR, LIST, PIPE, BACK 五
种。
```

```
*  EXEC: 普通的命令，只需照常执行。
*  REDIR: 重定向命令，符号 A > B 代表把A的输出重定向到文件B
*         符号 A < B 代表把A的输入重定向为文件B
*  LIST: 并列执行的命令，符号 A ; B 代表命令A和B会同时执行
*  PIPE: 管道。A | B 意味着会把A的输出作为B的输入
*  BACK: 后台执行。 &A 意味着把命令A放到后台执行
***************************************************************/
  case EXEC:
    ecmd = (struct execcmd*)cmd; // 转换类型
    if(ecmd->argv[0] == 0) // 处理错误 没给参数
      exit(1);
    exec(ecmd->argv[0], ecmd->argv); // 直接执行
    fprintf(2, "exec %s failed\n", ecmd->argv[0]); // 失败，则输出这一
行
    break;

  case REDIR:
    rcmd = (struct redircmd*)cmd; // 转换类型
    close(rcmd->fd); // 关闭默认的文件标识符
    if(open(rcmd->file, rcmd->mode) < 0){
        // 把重定向的那个文件标识符打开。因为上面关掉了默认的文件标识符，所以
现在open会选择那个文件标识符指向新的文件，这样就起到了“覆盖”的效果。
      fprintf(2, "open %s failed\n", rcmd->file);
      exit(1);
    }
    //直接运行
    runcmd(rcmd->cmd);
    break;

  case LIST:
  // 获取左右两个命令，然后分别执行。
    lcmd = (struct listcmd*)cmd;
    if(fork1() == 0)
      runcmd(lcmd->left);
    wait(0);
    runcmd(lcmd->right);
    break;

  case PIPE:
    pcmd = (struct pipecmd*)cmd;
    if(pipe(p) < 0)
      panic("pipe");

    // 父进程先运行左边的那个命令
```

```
      if(fork1() == 0){
      // 关闭标准输出
        close(1);
      // 把p[1]复制到0，这样每当程序向0写入信息，就会写到p[1]去
        dup(p[1]);
      // 关闭读端
        close(p[0]);
      // 关闭冗余的fd（已经是0了）
        close(p[1]);
        runcmd(pcmd->left);
      }
      if(fork1() == 0){
      // 关闭标准输入
        close(0);
      // 把标准输入重定向到p[0]，这样程序从p[0]读入信息
        dup(p[0]);
      // 关闭冗余
        close(p[0]);
      // 关闭写端
        close(p[1]);
        runcmd(pcmd->right);
      }
      close(p[0]);
      close(p[1]);
      // wait两次，因为有两个子进程
      wait(0);
      wait(0);
      break;

    case BACK:
    // 创建一个新的进程，也就是后台执行。
      bcmd = (struct backcmd*)cmd;
      if(fork1() == 0)
        runcmd(bcmd->cmd);
      break;
    }
    exit(0);
  }
```

重头戏：解析Shell命令

```
char whitespace[] = " \t\r\n\v";
char symbols[] = "<|>&;()";
/*************************************************
*  下面是两个辅助函数
*   peek(char** ps, char* es, char* tok) 的作用有两个：
*       1. 去除**ps中的空格，把指针*ps指向非空格字符
*       2. 如果s非空而且在**ps中找到任何一个给定字符tok，返回这个字符的位置
*   gettoken(char **ps, char *es, char **q, char **eq) 的作用：
*     从*ps到*es提取一段字符串，
*     这段字符串的开头为指针*q，结尾为指针*eq。
*************************************************/
int
gettoken(char **ps, char *es, char **q, char **eq);

int
peek(char **ps, char *es, char *toks);

/*************************************************
*  解析命令正式开始
*  parsecmd 把解析命令的任务交给 parseline
*  parseline 能够自己处理BACK和LIST类型的解析，管道类型交给 parsepipe
*  parsepipe 处理管道类型的解析，然后交给parseexec 处理其他的
*  parseexec 交给parseblock处理"()"类型的命令（括号中的命令是一个整体）
*            交给parseredirs处理重定向命令
*            自己处理普通的命令
*
*************************************************/
struct cmd*
parsecmd(char *s);

// 解析一行代码
struct cmd*
parseline(char **ps, char *es);

struct cmd*
parsepipe(char **ps, char *es);

struct cmd*
parseredirs(struct cmd *cmd, char **ps, char *es);

struct cmd*
parseblock(char **ps, char *es);
```

```
struct cmd*
parseexec(char **ps, char *es);


/*
*   构造命令，用于构造所有的命令结构。
*   这里类似面向对象程序设计语言的多态性，一个“虚”的cmd，很多子类，非常灵活
*/

struct cmd*
execcmd(void);

struct cmd*
redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int
fd);

struct cmd*
pipecmd(struct cmd *left, struct cmd *right);

struct cmd*
listcmd(struct cmd *left, struct cmd *right);

struct cmd*
backcmd(struct cmd *subcmd);
```

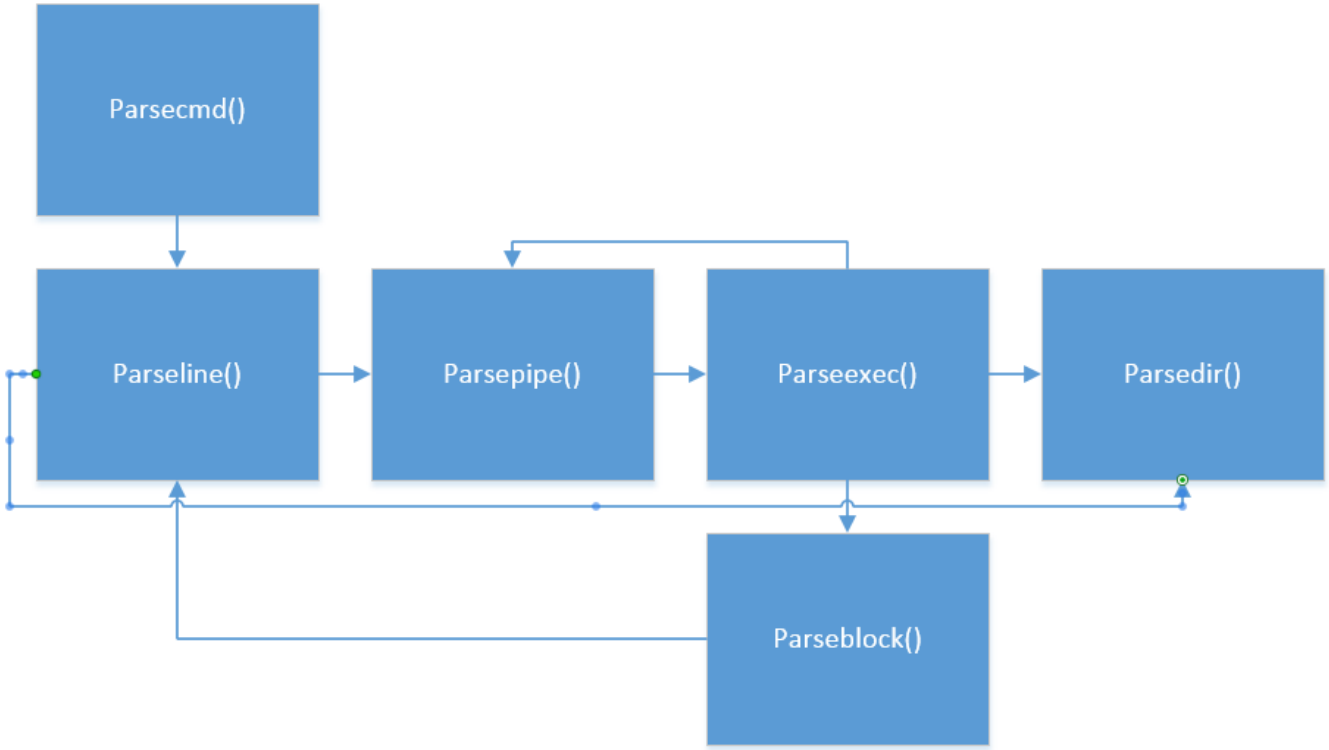
参考资料：

*xv6 shell*实现源代码分析

https://wjqwsp.github.io/2017/06/04/xv6-shell%E5%AE%9E%E7%8E%B0%E6%BA%90%E4%BB%A3%E7%A0%81%E5%88%86%E6%9E%90/

这篇文章写的非常好，非常清楚。

里面有两张非常有参考意义的图，解析环节的。



操作系统组织

为什么需要操作系统？重要的原因是提供强隔离，让运行的程序以为自己独占所有的资源，同时不让它访问到其他程序的资源。
隔离则需要抽象物理资源，把它们虚拟化。这些服务都由操作系统提供。
除此以外，还有并行、互动等服务，无论如何，这需要在操作系统和普通应用程序之间建筑一道边界。使一些命令只有操作系统可以使用，其他应用无法使用。假如普通的应用可以到处读写，操作系统就不能正常工作了。
在RISC-V中，CPU一共有三种执行命令的方式：机器模式、用户模式和监督模式(supervisor mode)。用户模式的进程通过系统调用(syscall)的方式实现一些特殊功能，如上一节“操作系统接口”所述。
xv6是一种monolithic的操作系统，它把整个操作系统都实现在监督模式。

xv6的代码组织结构如下所示

File	Description
bio.c	Disk block cache for the file system.
console.c	Connect to the user keyboard and screen.
entry.S	Very first boot instructions.
exec.c	exec() system call.
file.c	File descriptor support.
fs.c	File system.
kalloc.c	Physical page allocator.
kernelvec.S	Handle traps from kernel, and timer interrupts.
log.c	File system logging and crash recovery.
main.c	Control initialization of other modules during boot.
pipe.c	Pipes.
plic.c	RISC-V interrupt controller.
printf.c	Formatted output to the console.
proc.c	Processes and scheduling.
sleeplock.c	Locks that yield the CPU.
spinlock.c	Locks that don't yield the CPU.
start.c	Early machine-mode boot code.
string.c	C string and byte-array library.
swtch.S	Thread switching.
syscall.c	Dispatch system calls to handling function.
sysfile.c	File-related system calls.
sysproc.c	Process-related system calls.
trampoline.S	Assembly code to switch between user and kernel.
trap.c	C code to handle and return from traps and interrupts.
uart.c	Serial-port console device driver.
virtio_disk.c	Disk device driver.
vm.c	Manage page tables and address spaces.

Figure 2.2: Xv6 kernel source files.

启动xv6

接通 RISC-V 机器之后，boot loader 将 xv6 内核加载到地址0x80000000上。
执行的第一行xv6代码来自 entry.S
这段代码的功能是初始化堆栈，因为要运行c语言，必须要有一个栈。
在这之后，才能跳转到 c 代码的开始。

旁注： *RISC-V* 中的寄存器，指令
sp - x2的别名，stack pointer 栈指针，指向栈顶
la a b - 把地址从b加载到a
addi - 加常数
csrr a csr - 读取一个CSR（控制与状态寄存器），内容读取到a上
call a - 调用a

```
                # qemu -kernel 在地址 0x80000000 加载内核(< 0x80000000的
是外部设备，上面才是内存的地址)
        # 然后让每个CPU都跳转到那个地址.
        # kernel.ld 确保下面的代码
        # 被放到 0x80000000.
.section .text
.global _entry
_entry:
                # set up a stack for C.
        # stack0 is declared in start.c,
        # with a 4096-byte stack per CPU.
        # sp = stack0 + (hartid * 4096)
        la sp, stack0
        li a0, 1024*4
                csrr a1, mhartid
        addi a1, a1, 1
        mul a0, a0, a1
        add sp, sp, a0
                # jump to start() in start.c
        call start
spin:
        j spin
```

start.c 中的代码利用RISC-V的机器模式，做一些初始化的设定，打开硬件中断机制，然后用RISC-V的mret命令转入监督者模式。之后，进入 main.c。
start.c 中大部分代码都涉及汇编语言，而且还涉及许多之后的机制。所以不摘录start.c了。

main.c 开始了监督者模式的头几行代码。它初始化了若干子系统和设备，用 userinit() 创建了第一个进程。

```
void
main()
{
  if(cpuid() == 0){
    consoleinit(); // 初始化控制台
    printfinit(); // 初始化 prinf
    printf("\n");
    printf("xv6 kernel is booting\n");
    printf("\n");
    kinit();         // 初始化物理分页分配器
    kvminit();       // 创建内核页表
    kvminithart();   // 开启分页机制
    procinit();      // 初始化进程表
    trapinit();      // 初始化陷阱向量
    trapinithart();  // 安装内核陷阱向量
    plicinit();      // 设置中断控制器
    plicinithart();  // ask PLIC for device interrupts
    binit();         // 缓存
    iinit();         // inode表
    fileinit();      // 文件表
    virtio_disk_init(); // emulated hard disk
    userinit();      // 第一个进程!
    __sync_synchronize();
    started = 1;
  } else {
    while(started == 0)
      ;
    __sync_synchronize();
    printf("hart %d starting\n", cpuid());
    kvminithart();    // 开启分页
    trapinithart();   // install kernel trap vector
    plicinithart();   // ask PLIC for device interrupts
  }

  scheduler();
}
```

进程

下面，我们再来看看 xv6 中的进程。进程是对运行着的程序的抽象，是强隔离的实现。

比较重要的是：页表，内核栈和运行状态这三个成员变量。
xv6中的每个进程都有两个线程，一个线程是用户模式的线程，还有一个是内核模式的线程。所以有两个栈：kstack是内核模式的栈，而sp指向的是用户模式的栈，这个栈是由编译器控制的，无需显式维护。

```
struct proc {
  struct spinlock lock;

  // 用这些的时候必须上锁。p->lock must be held when using these:
  enum procstate state;        // 进程的状态 就绪或运行或阻塞
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to
parent's wait
  int pid;                     // 进程ID Process ID

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

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // 内核栈
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // 用户页表 User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // 上下文切换的时候，操作系统把寄存器存到
这个结构中 swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

系统调用

在调用 userinit() 之后，程序会执行 initcode.S 的机器码。这里的命令exec(init, argv) 是 xv6 的第一个系统调用。

```
# exec(init, argv)
.globl start
start:
        la a0, init
```

```
        la a1, argv
        li a7, SYS_exec
        ecall

# for(;;) exit();
exit:
        li a7, SYS_exit
        ecall
        jal exit

# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .long init
  .long 0
```

在这段代码后，用户程序将 exec 的代号 SYS_EXEC 压入寄存器a7，之后进入内核模式。用户模式不能直接用系统调用，必须进入内核模式才能用系统调用。

下面这段代码是 kernel/syscall.c 文件中的 syscall 函数。内核从用户进程中读取a7寄存器，然后用函数指针调用对应的系统调用函数。

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

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```