第一部分：内核向上接口

本部分主要聚焦于内核向上提供的接口，一般也被称为系统调用（system call）。

本部分并不会深入考察系统调用的底层实现，而只是关注其语义和表层实现（即系统调用完成了哪些功能？）。

本部分对应于 xv6 book 的 chapter 1 。

定义

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
```

```
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
```

```
extern uint64 sys_chdir(void);
extern uint64 sys_close(void);
extern uint64 sys_dup(void);
extern uint64 sys_exec(void);
extern uint64 sys_exit(void);
extern uint64 sys_fork(void);
extern uint64 sys_fstat(void);
extern uint64 sys_getpid(void);
extern uint64 sys_kill(void);
extern uint64 sys_link(void);
extern uint64 sys_mkdir(void);
extern uint64 sys_mknod(void);
extern uint64 sys_open(void);
extern uint64 sys_pipe(void);
extern uint64 sys_read(void);
extern uint64 sys_sbrk(void);
extern uint64 sys_sleep(void);
extern uint64 sys_unlink(void);
extern uint64 sys_wait(void);
extern uint64 sys_write(void);
extern uint64 sys_uptime(void);

static uint64 (*syscalls[])(void) = {
[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_sleep]   sys_sleep,
[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,
};
```

```
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;
  }
}
```

在 syscall.h 中，所有的系统调用被赋予编号；而在 syscall.c 中，编号与内核的函数声明通过 syscall() 函数连接起来。

```
uint64
sys_exit(void)
{ /*...*/ }

uint64
sys_getpid(void)
{ /*...*/ }

uint64
sys_fork(void)
{ /*...*/ }

uint64
sys_wait(void)
{ /*...*/ }

uint64
sys_sbrk(void)
{ /*...*/ }

uint64
sys_sleep(void)
{ /*...*/ }

uint64
sys_kill(void)
{ /*...*/ }

uint64
sys_uptime(void)
{ /*...*/ }
```

```
uint64
sys_dup(void)
{ /*...*/ }

uint64
```

```
sys_read(void)
{ /*...*/ }

uint64
sys_write(void)
{ /*...*/ }

uint64
sys_close(void)
{ /*...*/ }

uint64
sys_fstat(void)
{ /*...*/ }

uint64
sys_link(void)
{ /*...*/ }

uint64
sys_unlink(void)
{ /*...*/ }

uint64
sys_open(void)
{ /*...*/ }

uint64
sys_mkdir(void)
{ /*...*/ }

uint64
sys_mknod(void)
{ /*...*/ }

uint64
sys_chdir(void)
{ /*...*/ }

uint64
sys_exec(void)
{ /*...*/ }

uint64
sys_pipe(void)
{ /*...*/ }
```

在 `sysfile.c` 和 `sysproc.c` 中分别给出了系统调用的具体实现。

这些以 `sys_` 开头的函数，自身并不执行十分具体的操作。其主要工作包括：

1. 检查（用户）提供给系统调用的参数是否合法
2. 调用内核的相关函数，完成接口语义
3. return 正确的值

下面分为进程和文件两个部分，分别解释 `sysproc.c` 和 `sysfile.c` 中的系统调用的语义和实现。但在介绍系统调用之前，还是要稍微阐释系统调用的调用方法。

参数与调用规范

系统调用涉及到传参，但为了切换内核态和用户态，这里的调用并非直接通过 C 语言的调用函数，而是把参数通过 C 和 RISC-V 的调用规范传递。

系统调用的传参是通过 `trapframe` 来实现的，这点会在后面第二部分详细解释。系统调用传的整数参数通过读取 trapframe 的寄存器 `a0-a`7 来实现。

而字符串等变量，则通过传递地址变量，然后再读取对应地址找到一个字符串来实现（这些从内核与用户内存的沟通，是通过 `kernel/vm.c` 中的 `copyin`, `copyout` 函数等实现）。

xv6 提供了一些抽象这一过程的的函数，位于 `kernel/syscall.c` 中，具体来说是 `fetchstr` `fetchaddr`, ... `argraw`, `argint` 等函数。

在这里不再详细阐释以上函数的具体实现。

进程部分

fork

```
uint64
sys_fork(void)
{
  return fork();
}
```

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

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

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

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

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

  // increment reference counts on open file descriptors.
  for(i = 0; i < NOFILE; i++)
```

```
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);

  safestrcpy(np->name, p->name, sizeof(p->name));

  pid = np->pid;

  release(&np->lock);

  // 维护父子关系
  acquire(&wait_lock);
  np->parent = p;
  release(&wait_lock);

  // 维护进程状态
  acquire(&np->lock);
  np->state = RUNNABLE;
  release(&np->lock);

  return pid;
}
```

语义

进程（父进程）调用 `fork` 函数，会新建一个新进程（子进程）。【进程真的会维护父进程的 PID 。】

父进程与子进程完全相同（包括内存，自然也包括指令），但是他们从 `fork` 之后就相互独立，任何修改都不会影响彼此。

`fork` 函数会在调用父进程和子进程中“分别” return ，在父进程返回子进程的 PID，在子进程中返回 0 。借此可以区分子进程和父进程，执行进一步操作。

实现

主要进行了以下操作：

1. 分配一个新的进程表给子进程（通过 `allocproc` 函数）
2. 复制父进程的内存、寄存器、文件描述符给子进程
3. 设置子进程的返回值为 0 （通过改变 `trapframe` 的 `a0` 寄存器，也就是返回值放置的地方）
4. 维护子进程的父指针（parent）为父进程。
5. 标记子进程的状态（state）为可以执行（RUNNABLE）。

exit

```
uint64
sys_exit(void)
{
  int n;
  if(argint(0, &n) < 0)
    return -1;
  exit(n);
  return 0;  // not reached
}
```

```
// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait().
void
exit(int status)
{
  struct proc *p = myproc();

  if(p == initproc)
    panic("init exiting");

  // Close all open files.
  for(int fd = 0; fd < NOFILE; fd++){
    if(p->ofile[fd]){
      struct file *f = p->ofile[fd];
      fileclose(f);
      p->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(p->cwd);
  end_op();
  p->cwd = 0;

  acquire(&wait_lock);

  // Give any children to init.
  reparent(p);

  // Parent might be sleeping in wait().
  wakeup(p->parent);

  acquire(&p->lock);

  p->xstate = status;
  p->state = ZOMBIE;

  release(&wait_lock);

  // Jump into the scheduler, never to return.
  sched();
  panic("zombie exit");
}
```

语义

一个进程调用 `exit(status)`，会使当前进程退出运行，将控制权交还给调度器去进行调度。

惯例上，`status=0` 代表成功，`status=1` 代表失败。

实现

1. 关闭、释放所有打开的文件和已经分配的内存
2. 调整进程父子关系
3. 唤醒父进程
4. 交还控制权给调度器

5. 标记进程状态为死亡（`ZOMBIE`）

wait

```
uint64
sys_wait(void)
{
  uint64 p;
  if(argaddr(0, &p) < 0)
    return -1;
  return wait(p);
}
```

```
// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
int
wait(uint64 addr)
{
  struct proc *np;
  int havekids, pid;
  struct proc *p = myproc();

  acquire(&wait_lock);

  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(np = proc; np < &proc[NPROC]; np++){
      if(np->parent == p){
        // make sure the child isn't still in exit() or swtch().
        acquire(&np->lock);

        havekids = 1;
        if(np->state == ZOMBIE){
          // Found one.
          pid = np->pid;
          if(addr != 0 && copyout(p->pagetable, addr, (char *)&np->xstate,
                                  sizeof(np->xstate)) < 0) {
            release(&np->lock);
            release(&wait_lock);
            return -1;
          }
          freeproc(np);
          release(&np->lock);
          release(&wait_lock);
          return pid;
        }
        release(&np->lock);
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || p->killed){
      release(&wait_lock);
      return -1;
    }
```

```
    // Wait for a child to exit.
    sleep(p, &wait_lock);  //DOC: wait-sleep
  }
}
```

语义

进程调用 `wait` 函数，会等待**（任意）一个**子进程退出运行（`exit`）或被强制停止运行（`kill`）（如果没有子进程会返回 -1），将子进程返回的状态放到调用参数给出的 `status` 地址上。

实现

wait 函数具体完成了以下的内容：

1. 轮询进程表
2. 如果找到已经执行完成的子进程：把 status 放到指定位置，返回
3. 如果找到了没有执行完成的子进程：进入 sleep，等待被子进程唤醒
4. 如果压根没有子进程：直接 return -1.

文件部分

文件部分并不只包括文件，而是文件系统、设备、管道等一系列可以输入输出的设备。

文件描述符

文件描述符（显然）并非一个系统调用，而是抽暴露给上层的一个指针/handle，代表一个具有读/写功能的抽象对象。文件描述符表示的读/写的具体对象可以是：文件、屏幕、管道、设备等等。

文件描述符在实现上是从 0 开始的一系列整数。

dup

```
uint64
sys_dup(void)
{
  struct file *f;
  int fd;

  if(argfd(0, 0, &f) < 0)
    return -1;
  if((fd=fdalloc(f)) < 0)
    return -1;
  filedup(f);
  return fd;
}
```

```
// Increment ref count for file f.
struct file*
filedup(struct file *f)
{
  acquire(&ftable.lock);
  if(f->ref < 1)
    panic("filedup");
  f->ref++;
  release(&ftable.lock);
  return f;
}
```

语义

dup 系统调用会复制一个已有的文件标识符，相当于新建了指向同一个对象的的指针

实现

需要提及的是，这里的 file* 并非文件系统的“文件”，只是文件描述符这个指针的底层实现。

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE, FD_DEVICE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe; // FD_PIPE
  struct inode *ip;  // FD_INODE and FD_DEVICE
  uint off;          // FD_INODE
  short major;       // FD_DEVICE
};
```

可以看到第二行就是一个枚举类型，而 INODE 才是文件系统中对应的文件。这在本文中不再提及。

```
// Allocate a file descriptor for the given file.
// Takes over file reference from caller on success.
static int
fdalloc(struct file *f)
{
  int fd;
  struct proc *p = myproc();

  for(fd = 0; fd < NOFILE; fd++){
    if(p->ofile[fd] == 0){
      p->ofile[fd] = f;
      return fd;
    }
  }
  return -1;
}
```

```
// Fetch the nth word-sized system call argument as a file descriptor
// and return both the descriptor and the corresponding struct file.
static int
argfd(int n, int *pfd, struct file **pf)
{
```

```
  int fd;
  struct file *f;

  if(argint(n, &fd) < 0)
    return -1;
  if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
    return -1;
  if(pfd)
    *pfd = fd;
  if(pf)
    *pf = f;
  return 0;
}
```

这块的函数调用颇为复杂。但这几个函数都写了比较充分的注释。这里按照 dup 的调用链解释：

1. `sys_dup`：
2. `argfd(...)`：根据调用规范规定的第 1 个参数（也就是 fd ）取回一个文件指针
3. `fdalloc(...)`：找到第一个非被占用的的文件描述符，并将文件描述符（下标）的文件指针指向参数里的文件指针，并返回文件描述符
4. `filedup(...)`：把文件指针的 ref 自增。也就是当前维护了多少个文件描述符（存疑）。

read, write

```
uint64
sys_read(void)
{
  struct file *f;
  int n;
  uint64 p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argaddr(1, &p) < 0)
    return -1;
  return fileread(f, p, n);
}

uint64
sys_write(void)
{
  struct file *f;
  int n;
  uint64 p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argaddr(1, &p) < 0)
    return -1;

  return filewrite(f, p, n);
}
```

`read` 从某个文件描述符读取 n 字节到内存.

`write` 将内存的数据向某个文件描述符写入 n 字节。

这里不再分析 `fileread` 函数和 `filewrite` 函数，因为涉及到文件描述符背后的具体实现（文件、管道等等）。

pipe

```
uint64
sys_pipe(void)
{
  uint64 fdarray; // user pointer to array of two integers
  struct file *rf, *wf;
  int fd0, fd1;
  struct proc *p = myproc();

  if(argaddr(0, &fdarray) < 0)
    return -1;
  if(pipealloc(&rf, &wf) < 0)
    return -1;
  fd0 = -1;
  if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
    if(fd0 >= 0)
      p->ofile[fd0] = 0;
    fileclose(rf);
    fileclose(wf);
    return -1;
  }
  if(copyout(p->pagetable, fdarray, (char*)&fd0, sizeof(fd0)) < 0 ||
     copyout(p->pagetable, fdarray+sizeof(fd0), (char *)&fd1, sizeof(fd1)) < 0){
    p->ofile[fd0] = 0;
    p->ofile[fd1] = 0;
    fileclose(rf);
    fileclose(wf);
    return -1;
  }
  return 0;
}
```

语义

管道是在内核中实现的一个小缓冲区，会给用户进程提供一对两个文件描述符，一个用来读，一个用来写。管道提供了一种进程间通信的方法。

调用 pipe 函数，并提供一个参数地址 `p` ， `pipe` 函数会在 `p[0]` 和 `p[1]` 分别放置一个文件描述符，分别对应读和写的文件描述符。

实现

关键的函数是 `pipealloc` ，如下：

```
int
pipealloc(struct file **f0, struct file **f1)
{
  struct pipe *pi;

  pi = 0;
  *f0 = *f1 = 0;
  if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
    goto bad;
  if((pi = (struct pipe*)kalloc()) == 0)
    goto bad;
```

```
  pi->readopen = 1;
  pi->writeopen = 1;
  pi->nwrite = 0;
  pi->nread = 0;
  initlock(&pi->lock, "pipe");
  (*f0)->type = FD_PIPE;
  (*f0)->readable = 1;
  (*f0)->writable = 0;
  (*f0)->pipe = pi;
  (*f1)->type = FD_PIPE;
  (*f1)->readable = 0;
  (*f1)->writable = 1;
  (*f1)->pipe = pi;
  return 0;

 bad:
  if(pi)
    kfree((char*)pi);
  if(*f0)
    fileclose(*f0);
  if(*f1)
    fileclose(*f1);
  return -1;
}
```

这里便是调用 `filealloc`，获取了两个全新的文件指针，并且把它们的类型均设置为 `FD_PIPE`（这会在读写时调用针对 PIPE 的函数）。

然后再利用 `fdalloc` 分配两个文件描述符并返回。

open, close

```
uint64
sys_open(void)
{
  char path[MAXPATH];
  int fd, omode;
  struct file *f;
  struct inode *ip;
  int n;

  if((n = argstr(0, path, MAXPATH)) < 0 || argint(1, &omode) < 0)
    return -1;

  begin_op();

  if(omode & O_CREATE){
    ip = create(path, T_FILE, 0, 0);
    if(ip == 0){
      end_op();
      return -1;
    }
  } else {
    if((ip = namei(path)) == 0){
      end_op();
      return -1;
    }
```

```
    ilock(ip);
    if(ip->type == T_DIR && omode != O_RDONLY){
      iunlockput(ip);
      end_op();
      return -1;
    }
  }

  if(ip->type == T_DEVICE && (ip->major < 0 || ip->major >= NDEV)){
    iunlockput(ip);
    end_op();
    return -1;
  }

  if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
    if(f)
      fileclose(f);
    iunlockput(ip);
    end_op();
    return -1;
  }

  if(ip->type == T_DEVICE){
    f->type = FD_DEVICE;
    f->major = ip->major;
  } else {
    f->type = FD_INODE;
    f->off = 0;
  }
  f->ip = ip;
  f->readable = !(omode & O_WRONLY);
  f->writable = (omode & O_WRONLY) || (omode & O_RDWR);

  if((omode & O_TRUNC) && ip->type == T_FILE){
    itrunc(ip);
  }

  iunlock(ip);
  end_op();

  return fd;
}
```

```
uint64
sys_close(void)
{
  int fd;
  struct file *f;

  if(argfd(0, &fd, &f) < 0)
    return -1;
  myproc()->ofile[fd] = 0;
  fileclose(f);
  return 0;
}
```

语义

open 接受一个文件路径的参数，和一个表示打开方式（读/写，新建...）的参数，返回一个用于对这个文件读写的文件描述符。

close 调用接受一个文件描述符，关闭这个文件描述符，以待后面的读写。

文件在 open 和 close 之间会有一个锁的存在，防止其他进程对同一文件的读写。

具体的实现在文件系统章节中体现，这里不加

exec

```
uint64
sys_exec(void)
{
  char path[MAXPATH], *argv[MAXARG];
  int i;
  uint64 uargv, uarg;

  if(argstr(0, path, MAXPATH) < 0 || argaddr(1, &uargv) < 0){
    return -1;
  }
  memset(argv, 0, sizeof(argv));
  for(i=0;; i++){
    if(i >= NELEM(argv)){
      goto bad;
    }
    if(fetchaddr(uargv+sizeof(uint64)*i, (uint64*)&uarg) < 0){
      goto bad;
    }
    if(uarg == 0){
      argv[i] = 0;
      break;
    }
    argv[i] = kalloc();
    if(argv[i] == 0)
      goto bad;
    if(fetchstr(uarg, argv[i], PGSIZE) < 0)
      goto bad;
  }

  int ret = exec(path, argv);

  for(i = 0; i < NELEM(argv) && argv[i] != 0; i++)
    kfree(argv[i]);

  return ret;

 bad:
  for(i = 0; i < NELEM(argv) && argv[i] != 0; i++)
    kfree(argv[i]);
  return -1;
}
```

```
int
exec(char *path, char **argv)
```

```
{
  char *s, *last;
  int i, off;
  uint64 argc, sz = 0, sp, ustack[MAXARG], stackbase;
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pagetable_t pagetable = 0, oldpagetable;
  struct proc *p = myproc();

  begin_op();

  if((ip = namei(path)) == 0){
    end_op();
    return -1;
  }
  ilock(ip);

  // Check ELF header
  if(readi(ip, 0, (uint64)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;

  if((pagetable = proc_pagetable(p)) == 0)
    goto bad;

  // Load program into memory.
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
      goto bad;
    uint64 sz1;
    if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
    sz = sz1;
    if((ph.vaddr % PGSIZE) != 0)
      goto bad;
    if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
  }
  iunlockput(ip);
  end_op();
  ip = 0;

  p = myproc();
  uint64 oldsz = p->sz;

  // Allocate two pages at the next page boundary.
  // Use the second as the user stack.
  sz = PGROUNDUP(sz);
  uint64 sz1;
  if((sz1 = uvmalloc(pagetable, sz, sz + 2*PGSIZE)) == 0)
```

```
    goto bad;
  sz = sz1;
  uvmclear(pagetable, sz-2*PGSIZE);
  sp = sz;
  stackbase = sp - PGSIZE;

  // Push argument strings, prepare rest of stack in ustack.
  for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
      goto bad;
    sp -= strlen(argv[argc]) + 1;
    sp -= sp % 16; // riscv sp must be 16-byte aligned
    if(sp < stackbase)
      goto bad;
    if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
      goto bad;
    ustack[argc] = sp;
  }
  ustack[argc] = 0;

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

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

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

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

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

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

语义

`fork` 函数虽然可以创建新进程，但只能创建“子进程”，功能受限。

进程调用 `exec` 函数，会将当前进程程序替换为 `path` 指定的文件的程序（也包括内存等）。

指定的文件必须具有 ELF 格式。

`exec` 函数失败会返回 -1；`exec` 函数成功执行的话并不会返回，而是会从 ELF 文件头中找到起始地址，“继续”运行。

`exec` 命令也支持携带参数。

实现

在第二部分中解释。

练习

https://github.com/ChenQiqian/xv6-code-report/compare/riscv...ex1

增加了 `pingpong.c` 。

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

void test_once(){
    int p_1[2], p_2[2];
    char a[2]; // read and write buffers
    pipe(p_1);
    pipe(p_2);
    a[0] = 'a';
    write(p_1[1], a, 1);
    close(p_1[1]);
    int pid = fork();
    if(pid == 0) { // new process
        read(p_1[0], a, 1);
        close(p_1[1]);
        close(p_1[0]);
        // printf("p_1:%d %d\n",p_1[0],p_1[1]);
        write(p_2[1], a, 1);
        close(p_2[1]);
        exit(0);
    }
    // parent process
    read(p_2[0], a, 1);
    close(p_2[1]);
    close(p_2[0]);
    // printf("finished\n");
    wait(0);
}

int main(int argc, char *argv[]) {
    int ticks_before = uptime();
    for(int i = 0;i < 100000;i++){
        test_once();
```

```
    }
    int ticks_after = uptime();
    int ticks_diff = ticks_after - ticks_before;
    printf("%d\n", ticks_diff);
    exit(0);
}
```

阅读心得

这段代码主要解释了系统调用是如何实现的，我也见识到了为了实现内核的安全性所付出的在复杂性上的巨大代价。