lab6 - 实验报告

思考题

T1

> Thinking 6.1 示例代码中，父进程操作管道的写端，子进程操作管道的读端。如果现在想让父进程作为“读者”，代码应当如何修改？ ■

在原有代码实现中，作为读者，子进程关掉了管道写端；相应的，父进程关掉了管道读端，我们只要将关闭的端交换，并修改写入 / 读取语句即可实现要求即可

```

switch (fork()) {
  case -1:
    break;
  case 0:
    close(fildes[0]);
    write(fildes[1], "Hello world\n", 12);
    close(fildes[1]);
    exit(EXIT_SUCCESS);

  default:
    close(fildes[1]);
    read(fildes[0], buf, 100);
    printf("child-process read:%s",buf);
    close(fildes[0]);
    exit(EXIT_SUCCESS);
}
```

T2

> Thinking 6.2 上面这种不同步修改 pp_ref 而导致的进程竞争问题在 user/lib/fd.c 中的 dup 函数中也存在。请结合代码模仿上述情景，分析一下我们的 dup 函数中为什么会出现预想之外的情况？ ■

当我们调用 `dup` 函数时，会在进程中创建一个新的文件描述符 `newfd` ，这个文件描述符指向 `oldfd` 所拥有的文件表项，也就是在用户态中复制了一个文件的描述符，而实际上在执行复制的过程中，我们并不能一步把所有的数据都复制完，实际上是先对 `fd` 使用 `syscall_mem_map` 进行复制，再对它所属的 `data` 复制

现在假设一个情景：子进程 `dup(pipe[1])` 后 `read(pipe[0])`，父进程 `dup(pipe[0])` 后 `write(pipe[1])` ：

1. 先令子进程执行：顺序执行至 dup 完成后发生时钟中断，此时 `pageref(pipe[1]) = 1`，`pageref(pipe) = 1`
2. 随后父进程开始执行：执行至 dup 函数中 fd 和 data 的 map **之间**，此时 `pageref(pipe[0]) = 1`，`pageref(pipe) == 1`
3. 子进程再次开始执行：进入 read 函数，判断发现 `pageref(pipe[0]) == pageref(pipe)`

这个非同步更改的 `pageref` 和管道关闭时的等式一致，这里会让 `read` 函数认为管道中已经没有了写者，于是关闭了管道的读端

T3

> **Thinking 6.3** 阅读上述材料并思考：为什么系统调用一定是原子操作呢？如果你觉得不是所有的系统调用都是原子操作，请给出反例。希望能结合相关代码进行分析说明。 ■

结论：系统调用是原子操作

因为系统调用开始前，通过修改 SR 寄存器的值，关闭了外部中断，而在执行内核代码时，合理的内核设计应保证不出现其它类型的异常。所以这使得系统调用成为了原子操作

T4

> **Thinking 6.4** 仔细阅读上面这段话，并思考下列问题
>
> - 按照上述说法控制 **pipe_close** 中 **fd** 和 **pipe unmap** 的顺序，是否可以解决上述场景的进程竞争问题？给出你的分析过程。
> - 我们只分析了 **close** 时的情形，在 **fd.c** 中有一个 **dup** 函数，用于复制文件描述符。试想，如果要复制的文件描述符指向一个管道，那么是否会出现与 **close** 类似的问题？请模仿上述材料写写你的理解。
>
> ■

可以解决进程竞争的问题：

- 最初 `pageref(pipe[0]) = 2`，`pageref(pipe[1]) = 2`，`pageref(pipe) = 4`
- 子进程先运行，执行 `close` 解除了 `pipe[1]` 的文件描述符映射
- 发生时钟中断，此时 `pageref(pipe[0]) = 2`，`pageref(pipe[1]) = 1`，`pageref(pipe) = 4`
- 父进程执行完`close(pipe[0])` 后，`pageref(pipe[0]) = 1`，`pageref(pipe[1]) = 1`，`pageref(pipe) = 3`
- 可以发现此过程中不满足写端关闭的条件

在 `Thinking 6.2` 中用到的样例就体现了问题发生的原理：

- 如果先映射作为 `fd` 的 `pipe[0]`，就会暂时产生 `pageref(pipe) == pageref(pipe[0])` 的情况，会出现类似问题

T5

Thinking 6.5 思考以下三个问题。

- 认真回看 Lab5 文件系统相关代码，弄清打开文件的过程。
- 回顾 Lab1 与 Lab3，思考如何读取并加载 ELF 文件。
- 在 Lab1 中我们介绍了 `data` `text` `bss` 段及它们的含义，`data` 段存放初始化过的全局变量，`bss` 段存放未初始化的全局变量。关于 `memsize` 和 `filesize` ，我们在 Note 1.3.4中也解释了它们的含义与特点。关于 Note 1.3.4，注意其中关于“`bss` 段并不在文件中占数据”表述的含义。回顾 Lab3 并思考:`elf_load_seg()` 和 `load_icode_mapper()` 函数是如何确保加载 ELF 文件时，`bss` 段数据被正确加载进虚拟内存空间。`bss` 段在 ELF 中并不占空间，但 ELF 加载进内存后，`bss` 段的数据占据了空间，并且初始值都是 0。请回顾 `elf_load_seg()` 和 `load_icode_mapper()` 的实现，思考这一点是如何实现的?

下面给出一些对于上述问题的提示，以便大家更好地把握加载内核进程和加载用户进程的区别与联系，类比完成 `spawn` 函数。

关于第一个问题，在 Lab3 中我们创建进程，并且通过 `ENV_CREATE(...)` 在内核态加载了初始进程，而我们的 `spawn` 函数则是通过和文件系统交互，取得文件描述块，进而找到 ELF 在“硬盘”中的位置，进而读取。

关于第二个问题，各位已经在 Lab3 中填写了 `load_icode` 函数，实现了 ELF 可执行文件中读取数据并加载到内存空间，其中通过调用 `elf_load_seg` 函数来加载各个程序段。在 Lab3 中我们要填写 `load_icode_mapper` 回调函数，在内核态下加载 ELF 数据到内存空间；相应地，在 Lab6 中 `spawn` 函数也需要在用户态下使用系统调用为 ELF 数据分配空间。 ■

打开文件的过程：

- 根据文件名，调用用户态的 `open` 函数，其申请了一个文件描述符，并且调用了服务函数 `fsipc_open` ，利用 `fsipc` 包装后向文件服务进程发起请求
- 文件服务进程接收到请求后分发给 `serve_open` 函数，创建 `Open` 并调用 `file_open` 函数从磁盘中加载到内存中，返回共享的信息，文件打开

加载 ELF 文件：

- 在进程中打开 ELF 文件后，先创建子进程，初始化其堆栈，做好前置工作
- 按段（Segment）解析 ELF 文件，利用 `elf_load_seg` 函数将每个段映射到子进程的对应地址空间中，在函数执行过程中，会对在文件中不占大小、在内存中需要补 0 的 `.bss` 段数据进行额外的映射（总文件大小与已经映射的大小的差值即为 `.bss` 段大小，追加在文件部分之后，并填充为 0）
- 实际的映射函数是 `spwan_mapper`，它利用 `syscall_mem_map` 将数据从父进程映射到子进程中，完成 ELF 文件的加载

T6

Thinking 6.6 通过阅读代码空白段的注释我们知道，将标准输入或输出定向到文件，需要我们将其 `dup` 到 0 或 1 号文件描述符 (fd)。那么问题来了：在哪步，0 和 1 被“安排”为标准输入和标准输出？请分析代码执行流程，给出答案。 ■

用于 `reading` 的文件描述符会被 `dup` 到 `fd[0]`，过程如下：

```
// Open 't' for reading, dup it onto fd 0, and then close the original fd.
  /* Exercise 6.5: Your code here. (1/3) */
  if ((r = open(t, O_RDONLY)) < 0) {
    user_panic("redirction_1: open file in shell failed!");
  }
  fd = r;
  dup(fd, 0);
  close(fd);
```

映射 write 的描述符操作类似，这里把 0 和 1 分别映射成为标准输入和输出

T7

> **Thinking 6.7** 在 shell 中执行的命令分为内置命令和外部命令。在执行内置命令时 shell 不需要 `fork` 一个子 shell，如 Linux 系统中的 `cd` 命令。在执行外部命令时 shell 需要 `fork` 一个子 shell，然后子 shell 去执行这条命令。
>
> 据此判断，在 `MOS` 中我们用到的 shell 命令是内置命令还是外部命令？请思考为什么 Linux 的 `cd` 命令是内部命令而不是外部命令？ ■

我们用到的 shell 命令均属于外部命令，在 shell 运行过程中，我们对指令调用 `runcmd` 进行处理，其内部调用了 `parsecmd` 进行解析，在指令解析后直接利用这个指令 `spwan` 了一个子进程

```
int child = spawn(argv[0], argv);
```

这也就是说，无论执行任何指令，MOS 中的 shell 都会将这个流程解析为：创建子进程、运行指令所指向的文件、完成所需功能

T8

> **Thinking 6.8** 在你的 shell 中输入命令 `ls.b | cat.b > motd`。
>
> - 请问你可以在你的 shell 中观察到几次 `spawn` ？分别对应哪个进程？
> - 请问你可以在你的 shell 中观察到几次进程销毁？分别对应哪个进程？
>
> ■

终端输出如下：

```
[00002803] pipecreate
[00003805] destroying 00003805
[00003805] free env 00003805
i am killed ...
[00004006] destroying 00004006
[00004006] free env 00004006
i am killed ...
[00003004] destroying 00003004
[00003004] free env 00003004
i am killed ...
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...
```

- 可以观察到2次 `spawn`：4006 和 3805 进程，这是 ls.b 命令和 cat.b 命令通过 shell 创建的进程
- 可以观察到4次进程销毁：3805、4006、3004、2803，按顺序是 ls.b 命令、cat.b 命令 spawn 出的进程、通过管道创建的 shell 进程和 main 函数的 shell 进程

管道

管道是一种典型的进程间单向通信的方式，管道分有名管道和匿名管道两种，匿名管道只能在具有公共祖先的进程之间使用，且通常使用在父子进程之间，在 MOS 中，我们要实现匿名管道

后面所使用的管道，不特殊说明，指的都是匿名管道

```
#include <stdlib.h>
#include <unistd.h>
int fildes[2];
char buf[100];
int status;
int main(){
  status = pipe(fildes);
  if (status == -1) {
    printf("error\n");
  }
  switch (fork()) {
    case -1: break;
    case 0: /* 子进程 - 作为管道的读者 */
      close(fildes[1]); /* 关闭不用的写端 */
      read(fildes[0], buf, 100); /* 从管道中读数据 */
      printf("child-process read:%s",buf); /* 打印读到的数据 */
      close(fildes[0]); /* 读取结束，关闭读端 */
      exit(EXIT_SUCCESS);
    default: /* 父进程 - 作为管道的写者 */
      close(fildes[0]);  /* 关闭不用的读端 */
      write(fildes[1], "Hello world\n", 12); /* 向管道中写数据 */
      close(fildes[1]); /* 写入结束，关闭写入端 */
      exit(EXIT_SUCCESS);
  }
}
```

该代码实现了从父进程向管道中写入消息 Hello,world，子进程从管道中读出数据并打印到屏幕的功能

它演示了管道在父子进程之间通信的基本用法：

1. 父进程在 pipe 函数之后，调用 fork 来产生一个子进程，之后在父子进程中各自执行不同的操作：关掉自己不会用到的管道端，然后进行相应的读写操作
2. 在示例代码中，父进程操作写端，而子进程操作读端，从本质上说，管道是一种只存在于内存中的文件，在 UNIX 以及 MOS 中，父进程调用 pipe函数时，会打开两个新的文件描述符：一个表示只读端，另一个表示只写端，两个描述符都映射到了同一片内存区域
3. 在 fork 的配合下，子进程复制父进程的两个文件描述符，从而在父子进程间形成了四个（父子各拥有一读一写）指向同一片内存区域的文件描述符，父子进程可根据需要关掉自己不用的一个，从而实现父子进程间的单向通信管道，这也是匿名管道只能用在具有亲缘关系的进程间通信的原因

user / lib / pipe.c

相关数据结构

```
struct Dev devpipe = {
    .dev_id = 'p',
    .dev_name = "pipe",
    .dev_read = pipe_read,
    .dev_write = pipe_write,
    .dev_close = pipe_close,
```

```
    .dev_stat = pipe_stat,
};

// 由 devpipe 所能索引到的函数指针
static int pipe_close(struct Fd *);
static int pipe_read(struct Fd *fd, void *buf, u_int n, u_int offset);
static int pipe_stat(struct Fd *, struct Stat *);
static int pipe_write(struct Fd *fd, const void *buf, u_int n, u_int
offset);

#define PIPE_SIZE 32 // 将缓冲区大小设为较小的 32 字节，用于激发并发竞争（race
condition）

struct Pipe {
        u_int p_rpos;             // 当前读取位置（读指针）
        u_int p_wpos;             // 当前写入位置（写指针）
        u_char p_buf[PIPE_SIZE]; // 实际的数据缓冲区
};
```

devpipe 结构体实例：声明了一个叫做**管道**的设备结构体，管道设备的操作接口，并且为其设置了相关函数操作的接口

字段名	含义
`dev_id = 'p'`	设备 ID，字符 `'p'` 用来唯一标识这个设备是“pipe”
`dev_name = "pipe"`	设备名称，便于调试和识别
`dev_read = pipe_read`	管道的读取函数指针
`dev_write = pipe_write`	管道的写入函数指针
`dev_close = pipe_close`	管道关闭时调用的函数指针
`dev_stat = pipe_stat`	查询管道状态时使用的函数指针（如缓冲区中剩余数据等）

Pipe 结构体：管道**内部**的实际数据结构，保存通信数据与读写状态

字段名	类型	含义
`p_rpos`	`u_int`	当前**读指针位置**，标记下一个将从哪里读取数据（索引）
`p_wpos`	`u_int`	当前**写指针位置**，标记下一个将从哪里写入数据（索引）
`p_buf`	`u_char[PIPE_SIZE]`	实际的数据缓冲区，大小为 32 字

pipe

```
/* Overview:
  函数功能 : 创建一个管道
  后置条件 : 成功时返回 0，并且把管道的读端设置为 pfd[0]，写端设置为 pfd[0]，失败时返回
相应的错误码
  入参含义 : 返回时用来赋值为文件描述符，0 - 读；1 - 写
 */
int pipe(int pfd[2]) {
        int r;
```

```
        void *va;
        struct Fd *fd0, *fd1;

// 分配文件描述符
/*
  使用 fd_alloc 申请一个空闲的 Fd 结构，并且保存地址在 fd0 中
  通过 syscall_mem_alloc 为该结构分配用户空间的内存页，并设置权限位可写+可共享
*/
        if ((r = fd_alloc(&fd0)) < 0 || (r = syscall_mem_alloc(0, fd0, PTE_D | PTE_LIBRARY)) < 0) {
                goto err;
        }
// 为 fd1 的申请同理
        if ((r = fd_alloc(&fd1)) < 0 || (r = syscall_mem_alloc(0, fd1, PTE_D | PTE_LIBRARY)) < 0) {
                goto err1;
        }

// 分配并映射 pipe 数据页
/*
  获取 fd0 所对应的用来保存数据的 data 地址，对应的 Pipe 结构存在此处，为 va 分配一个实际物理页，用来保存管道缓冲区，也就是 Pipe 结构
*/
        va = fd2data(fd0);
        if ((r = syscall_mem_alloc(0, (void *)va, PTE_D | PTE_LIBRARY)) < 0) {
                goto err2;
        }
// 把 fd0 的数据区映射到 fd1 的数据区上，二者共享同一页，这一步使得 fd0 和 fd1 的数据区共享一块实际的物理内存，实现通信
        if ((r = syscall_mem_map(0, (void *)va, 0, (void *)fd2data(fd1), PTE_D | PTE_LIBRARY)) <
            0) {
                goto err3;
        }

// 设置文件描述符属性
/*
  fd0 为只读端，打开模式是只读
  fd1 为只写端，打开模式是只写
  devpipe.dev_id 是管道设备的 ID，表示 Fd 和 Pipe 设备是相互绑定的
*/
        fd0->fd_dev_id = devpipe.dev_id;
        fd0->fd_omode = O_RDONLY;

        fd1->fd_dev_id = devpipe.dev_id;
        fd1->fd_omode = O_WRONLY;

        debugf("[%08x] pipecreate \n", env->env_id, vpt[VPN(va)]);

// 设置返回值，0 - 0，1 - 1，并且返回
        pfd[0] = fd2num(fd0);
        pfd[1] = fd2num(fd1);
        return 0;
// 如果发生错误，需要解除已经建立的页面映射
err3:
        syscall_mem_unmap(0, (void *)va);
err2:
```

```
        syscall_mem_unmap(0, fd1);
err1:
        syscall_mem_unmap(0, fd0);
err:
        return r;
}
```

pipe_is_closed 和 _pipe_is_closed

pipe_is_closed：该函数其实是对 _pipe_is_closed 这个函数的上层封装

```
/* Overview:
  函数功能 : 检查文件描述符 fdnum 所对应的管道是否已经关闭
  返回值 : 如果已关闭，返回 1，否则返回 0
 */
int pipe_is_closed(int fdnum) {
        struct Fd *fd;
        struct Pipe *p;
        int r;
// 先根据 fdnum 索引到 fd，再根据 fd 索引到 data(也就是对应的 pipe)，接着直接调用 _pipe_is_closed 即可
        if ((r = fd_lookup(fdnum, &fd)) < 0) {
                return r;
        }
        p = (struct Pipe *)fd2data(fd);
        return _pipe_is_closed(fd, p);
}
```

_pipe_is_closed:

```
/* Overview:
  函数功能 : 检查管道是否已经关闭
  后置条件 : 如果管道已经关闭，返回 1；如果管道没有关闭，返回 0
  提示 : 使用 pageref 来获取由虚拟页映射的物理页的引用计数
 */
static int _pipe_is_closed(struct Fd *fd, struct Pipe *p) {
/*
  pageref(p) : 映射该管道物理页的读者和写者的总数
  pageref(fd) : 打开该 fd 的环境数目(如果 fd 是读者则是读者数，如果 fd 是写者则是写者数)
  如果二者相同，则管道已经关闭，反之则没有关闭
*/

        int fd_ref, pipe_ref, runs;
/*
  使用 pageref 获取 fd 和 p 的引用计数，分别存入 fd_ref 和 pipe_ref 中，读取这两个引用计数的时候，需要保证 env->env_uns 在读取前后没有变化，否则需要重新获取，保持数据一致性
*/
        do {
                runs = env->env_runs;
                fd_ref = pageref(fd);
                pipe_ref = pageref(p);
        } while (runs != env->env_runs);

        return fd_ref == pipe_ref;
}
```

实现机理：

1. **正常情况下(管道未关闭)**:
   - `pageref(p)` 会大于 `pageref(fd)`
   - 因为 `pageref(p)` 包含所有读写端的引用，而 `pageref(fd)` 只包含当前端(读或写)的引用
   - 例如：有一个读端和一个写端时，`pageref(p)=2`，而 `pageref(fd)=1`(取决于fd是读端还是写端)
2. **管道关闭时**:
   - 当管道的另一端关闭时，`pageref(p)` 会减少
   - 最终 `pageref(p)` 将等于 `pageref(fd)`
   - 这意味着只有当前端还在引用管道，另一端已经关闭，此时说明管道使用完毕，也就是已经关闭

如果不考虑多个进程共享管道，那么最后管道关闭的时候，fd_ref 和 p_ref 应该都是 1，也就是只有管道的创建端（既可以是读端也可以是写端）这一个端口持有管道

pipe_read

```
/* Overview:
  函数功能 : 从 fd 所指向的管道中最多读取 n 个字节到 vbuf 中
  后置条件 : 返回从管道中读出的字节数，返回值必须大于 0，除非管道已经关闭且自上次读取后没有写入任何数据
  提示 :
    使用 fd2data 获取 fd 所指向的 Pipe 结构
    使用 _pipe_is_closed 检查管道是否已经关闭
    该函数不使用 offset 参数
 */
static int pipe_read(struct Fd *fd, void *vbuf, u_int n, u_int offset) {
        int i;
        struct Pipe *p;
        char *rbuf;
// 把文件描述符 fd 转换为底层管道结构，同时设置用户缓冲区指针 rbuf
        p = (struct Pipe *)fd2data(fd);
        rbuf = (char *)vbuf;
// 最多读取 n 个字节
        for (i = 0; i < n; ++i) {
          // 读写位置相等的时候，表示缓冲区为空，rpos 为读位置，wpos 为写位置，此时需要等待或检查关闭状态，如果管道已经关闭(先判断)，或者已经读取了部分字节(读了 i 个)，则直接返回已经读取的字节数
                while (p->p_rpos == p->p_wpos) {
                        if (_pipe_is_closed(fd, p) || i > 0) {
                                return i;
                        }
                  // 否则挂起进程
                        syscall_yield();
                }
          // 可以正常读，从唤醒缓冲区读取一个字节，同时更新读取位置的指针，保证下一次读取可以从当前读取的下一位开始读取
                rbuf[i] = p->p_buf[p->p_rpos % PIPE_SIZE];
                p->p_rpos++;
        }
        return n;
}
```

pipe_write

和 pipe_read 完全对偶的函数，一个负责从管道中读取，一个负责向管道中写入

```
/* Overview:
  函数功能 : 把用户缓冲区 vbuf 中的数据，最多取前 n 个字节，写入到 fd 所指向的管道中，返回值是成功写入的字节数
  终止条件 : n 个字节写完，或者管道写满
 */
static int pipe_write(struct Fd *fd, const void *vbuf, u_int n, u_int offset) {
        int i;
        struct Pipe *p;
        char *wbuf;
        p = (struct Pipe *)fd2data(fd);
        wbuf = (char *)vbuf;
        for (i = 0; i < n; ++i) {
          // 这里注意管道满的条件，只需要两个指针相差为 PIGE_SIZE 即可，因为这里模拟的是循环队列
                while (p->p_wpos - p->p_rpos == PIPE_SIZE) {
                        if (_pipe_is_closed(fd, p)) {
                                return i;
                        }
                        syscall_yield();
                }
                p->p_buf[p->p_wpos % PIPE_SIZE] = wbuf[i];
                p->p_wpos++;
        }
        return n;
}
```

pipe_close

```
/* Overview:
  函数功能 : 关闭 fd 所对应的管道
  返回值 : 成功时返回 0
 */
static int pipe_close(struct Fd *fd) {
// 分别取消 fd 和 fd 所指向的 data 在物理内存所能索引到的物理页面的映射，使用 syscall_mem_unmap 实现即可
        syscall_mem_unmap(0, fd);
        syscall_mem_unmap(0, (void *)fd2data(fd));
        return 0;
}
```

shell

shell 是指为使用者提供操作界面的软件（命令解析器），它接收用户命令，然后调用相应的应用程序

user / lib / spawn.c

spawn 系列的函数用来调用文件系统中的可执行文件并执行，这是 shell 解释器和真正执行相关命令的子进程之间**直接产生联系**的部分，子进程无法得知命令的具体含义，只能通过 spawn 给出的 ELF 文件路径，不透明地解析并执行相应的 ELF 文件

init_stack

函数功能：为子进程初始化栈空间，具体分为以下四个步骤

1. 计算命令行参数的总长度
2. 在临时页面上布置参数字符串和指针数组
3. 设置argc / argv等启动参数
4. 将初始化好的栈页面映射到子进程地址空间

```
/*
  函数功能 : 子进程栈空间初始化
  入参含义 : child - 子进程的 envid，argv - 命令行参数，以 NULL 结尾的字符指针数组，init_sp - 存返回值，表示子进程初始栈指针位置
*/
int init_stack(u_int child, char **argv, u_int *init_sp) {
// argc - 命令行参数个数，tot - 命令行参数总长度，strings - 参数字符串在栈中起始位置(栈页面的高地址区)，args - argv 指针数组位置(参数字符串下方)
        int argc, i, r, tot;
        char *strings;
        u_int *args;
        tot = 0;
        for (argc = 0; argv[argc]; argc++) {
                tot += strlen(argv[argc]) + 1; // 总长度要包含结尾 0
        }

// 检查参数字符串和指针数组是否会超出页面
        if (ROUND(tot, 4) + 4 * (argc + 3) > PAGE_SIZE) {
                return -E_NO_MEM;
        }

// 在 UTEMP 位置分配临时页面
        strings = (char *)(UTEMP + PAGE_SIZE) - tot;
        args = (u_int *)(UTEMP + PAGE_SIZE - ROUND(tot, 4) - 4 * (argc + 1));

        if ((r = syscall_mem_alloc(0, (void *)UTEMP, PTE_D)) < 0) {
                return r;
        }

// 把参数字符串拷贝到对应的栈页面
        char *ctemp, *argv_temp;
        u_int j;
        ctemp = strings;
        for (i = 0; i < argc; i++) {
                argv_temp = argv[i];
                for (j = 0; j < strlen(argv[i]); j++) {
                        *ctemp = *argv_temp;
                        ctemp++;
                        argv_temp++;
                }
                *ctemp = 0;
                ctemp++;
        }

// 指针数组设置
        ctemp = (char *)(USTACKTOP - UTEMP - PAGE_SIZE + (u_int)strings);
        for (i = 0; i < argc; i++) {
```

```
                args[i] = (u_int)ctemp;
                ctemp += strlen(argv[i]) + 1;
        }
        ctemp--;
        args[argc] = (u_int)ctemp;

// 启动参数布置，argc 在最底部，然后是 argv 指针
        u_int *pargv_ptr;
        pargv_ptr = args - 1;
        *pargv_ptr = USTACKTOP - UTEMP - PAGE_SIZE + (u_int)args;
        pargv_ptr--;
        *pargv_ptr = argc;

// 完成内存映射
        *init_sp = USTACKTOP - UTEMP - PAGE_SIZE + (u_int)pargv_ptr;

        if ((r = syscall_mem_map(0, (void *)UTEMP, child, (void *)(USTACKTOP
- PAGE_SIZE), PTE_D)) <
            0) {
                goto error;
        }
        if ((r = syscall_mem_unmap(0, (void *)UTEMP)) < 0) {
                goto error;
        }

        return 0;

// 发生错误的时候释放掉已经映射的页面
error:
        syscall_mem_unmap(0, (void *)UTEMP);
        return r;
}
```

spawn

该函数用来加载并启动一个新的用户程序，有点类似于 fork 函数，但 spwan 会向子进程中加载新的 ELF 文件，包括以下六个步骤：

1. 读取 ELF 文件（这个 ELF 文件其实就是某个 shell 指令文件编译后的结果）
2. 创建新的子进程进程
3. 加载程序段到子进程的内存空间（包括程序头的解析）
4. 初始化栈空间
5. 共享页面的设置
6. 设置子进程为可运行状态

```
/* Overview:
  函数功能 : 加载并启动一个新的用户程序
  加载完成后必须执行 D-cache（数据缓存）和 I-cache（指令缓存）的写回/失效操作，以维持缓存一致性，而 MOS 并未实现这些操作
  入参含义 : prog - 待加载程序的路径名，argv - 程序入参
  返回值 : 若成功则返回子进程 envid
 */
int spawn(char *prog, char **argv) {
// 以只读方式打开程序路径 prog 所对应的可执行文件，返回文件描述符 fd，得到相应的可执行文件的文件描述符
        int fd;
        if ((fd = open(prog, O_RDONLY)) < 0) {
```

```
                return fd;
        }

// 读取 ELF 文件头，elfbuf 用来存放 ELF 文件头数据，如果读取字节数不足，则跳转到 err 处理
        int r;
        u_char elfbuf[512];
        if ((r = readn(fd, elfbuf, sizeof(Elf32_Ehdr))) != sizeof(Elf32_Ehdr)) {
                goto err;
        }

// 解析 ELF 文件头，elf_from 用来解析 ELF 文件头，返回 Elf32_Ehdr 指针，entrypoint 保存 ELF 程序的入口地址，即 e_entry 字段
        const Elf32_Ehdr *ehdr = elf_from(elfbuf, sizeof(Elf32_Ehdr));
        if (!ehdr) {
                r = -E_NOT_EXEC;
                goto err;
        }
        u_long entrypoint = ehdr->e_entry;

// 创建子环境，使用 syscall_exofork 函数创建子进程，返回其 envid
        u_int child;
        child = syscall_exofork();
        if (child < 0) {
                r = child;
                goto err;
        }

// 调用 init_stack 为子进程初始化用户栈，初始化并返回栈顶指针 sp
        u_int sp;
        if ((r = init_stack(child, argv, &sp))) {
                goto err1;
        }

// 加载程序段(ELF 段)，使用 ELF_FOREACH_PHDR_OFF 宏遍历每一个 program header，对于可加载段(PT_LOAD)，使用 elf_head_seg 把该段内容加载到子环境地址空间，同时使用 spawn_mapper 实现地址映射回调，把页面映射回子进程
        size_t ph_off;
        ELF_FOREACH_PHDR_OFF (ph_off, ehdr) {
                if ((r = seek(fd, ph_off)) < 0) {
                        goto err1;
                }
                if ((r = readn(fd, elfbuf, ehdr->e_phentsize)) != ehdr->e_phentsize) {
                        goto err1;
                }
                Elf32_Phdr *ph = (Elf32_Phdr *)elfbuf;
                if (ph->p_type == PT_LOAD) {
                        void *bin;
                        r = read_map(fd, ph->p_offset, &bin);
                        if (r != 0) {
                                goto err1;
                        }
                        r = elf_load_seg(ph, bin, spawn_mapper, &child);
                        if (r != 0) {
                                goto err1;
                        }
```

```
                }
        }
// 加载完毕后关闭文件即可，因为 ELF 文件已经加载完毕，不再需要文件描述符
        close(fd);

// 设置子进程上下文 Trapframe，通过 ENVX(child) 先取得进程原有上下文，再设置入口指令地址 epc 和栈指针 sp，最后系统调用 set_trapframe 写入新的上下文
        struct Trapframe tf = envs[ENVX(child)].env_tf;
        tf.cp0_epc = entrypoint;
        tf.regs[29] = sp;
        if ((r = syscall_set_trapframe(child, &tf)) != 0) {
                goto err2;
        }

// 共享库内存映射(遍历所有带有 PTE_LIBRARY 标志的页进行映射)
        for (u_int pdeno = 0; pdeno <= PDX(USTACKTOP); pdeno++) {
                if (!(vpd[pdeno] & PTE_V)) {
                        continue;
                }
                for (u_int pteno = 0; pteno <= PTX(~0); pteno++) {
                        u_int pn = (pdeno << 10) + pteno;
                        u_int perm = vpt[pn] & ((1 << PGSHIFT) - 1);
                        if ((perm & PTE_V) && (perm & PTE_LIBRARY)) {
                                void *va = (void *)(pn << PGSHIFT);

                                if ((r = syscall_mem_map(0, va, child, va, perm)) < 0) {
                                        debugf("spawn: syscall_mem_map %x %x: %d\n", va, child, r);
                                        goto err2;
                                }
                        }
                }
        }

// 子进程的启动，设置其状态为 ENV_RUNNABLE，使得子进程可以被调度器安排执行
        if ((r = syscall_set_env_status(child, ENV_RUNNABLE)) < 0) {
                debugf("spawn: syscall_set_env_status %x: %d\n", child, r);
                goto err2;
        }
        return child;

err2:
        syscall_env_destroy(child);
        return r;
err1:
        syscall_env_destroy(child);
err:
        close(fd);
        return r;
}
```

user / sh.c

sh.c 是我们实现的 shell 解释器的模拟，用于从脚本文件或者用户标准输入中读取指令，解析指令，把相应指令对应的 ELF 文件路径传送给子进程

函数依赖关系大致为：

main -> readline -> runcmd -> parsecmd -> gettoken -> spawn

parsecmd

```
/* Overview :
  函数功能：解析命令行输入，包括普通参数（命令与其对应的参数），输入重定向 >，输出重定向 <，管道 |，并且根据命令的结构设置合适的文件描述符
  入参含义 : argv - 参数数组，保存解析出来的命令参数，比如 ls -l 会被解析成 argv[0] = "ls", argv[1] = "-l"; rightpipe - 如果命令中存在管道 |，用于存放管道右侧子进程的 envid，用于实现后续的调度
*/
int parsecmd(char **argv, int *rightpipe) {
// argc - 参数个数
        int argc = 0;
        while (1) {
                char *t;
                int fd, r;
// 使用 gettoken 获得一个单词，其返回值即为 case 中的情况，‘w’ 表示普通单词，特殊符号返回对应字符，0 表示输入结尾
                int c = gettoken(0, &t);
                switch (c) {
                case 0:
                        return argc;
                case 'w':
                        if (argc >= MAXARGS) {
                                debugf("too many arguments\n");
                                exit();
                        }
                        argv[argc++] = t;
                        break;
// 输入重定向，< 的后面应该紧跟一个文件名，也就是 w 类型，该文件名存在 t 中，根据文件名以只读方式打开文件，把文件描述符 fd 复制到标准输入 0 上，关闭文件描述符
                case '<':
                        if (gettoken(0, &t) != 'w') {
                                debugf("syntax error: < not followed by word\n");
                                exit();
                        }

                        fd = open(t, O_RDONLY);
                        if (fd < 0) {
                                debugf("failed to open '%s'\n", t);
                                exit();
                        }
                        dup(fd, 0);
                        close(fd);
// 实现功能 : 把标准输入重定向为文件 t 的内容
                        break;
// 输出重定向，> 的后面应该紧跟一个文件名，也就是 w 类型，该文件名存在 t 中，根据文件名以写入，创建，截断方式打开文件，把文件描述符 fd 复制到标准输入 1 上，关闭文件描述符
                case '>':
```

```
                        if (gettoken(0, &t) != 'w') {
                                debugf("syntax error: > not followed by
word\n");
                                exit();
                        }
                        fd = open(t, O_WRONLY | O_CREAT | O_TRUNC);
                        if (fd < 0) {
                                debugf("failed to open '%s'\n", t);
                                exit();
                        }
                        dup(fd, 1);
                        close(fd);
// 实现功能 : 把标准输出重定向为文件 t 的输出
                        break;
// 管道处理，使用 pipe 创建一对管道文件描述符，p[0] - 读，p[1] - 写，fork 用来创建子进
程
                case '|':
                        int p[2];
                        r = pipe(p);
                        if (r != 0) {
                                debugf("pipe: %d\n", r);
                                exit();
                        }
                        r = fork();
                        if (r < 0) {
                                debugf("fork: %d\n", r);
                                exit();
                        }
// 如果是父进程，记录右侧子进程 ID，如果是子进程
// 对于子进程，通过 dup 把管道读端设置为标准输入，关闭对应的无用端口，递归调用 parsecmd
继续处理管道右侧的命令
                        *rightpipe = r;
                        if (r == 0) {
                                dup(p[0], 0);
                                close(p[0]);
                                close(p[1]);
                                return parsecmd(argv, rightpipe);
 // 对于父进程，通过 dup 把管道写端设置为标准输出，关闭对应的无用端口，返回参数数量，让主
程序执行当前命令左边的部分
                        } else {
                                dup(p[1], 1);
                                close(p[1]);
                                close(p[0]);
                                return argc;
                        }
// 实现功能 : 两个命令之间的管道通信，比如 ls | grep txt
                        break;
                }
        }

        return argc;
}
```

这里调用了 dup 函数，用于实现输入输出的重定向

1. `dup(fd, 1)`：把标准输出(1)重定向到 fd
2. `dup(fd, 0)`：把标准输入(0)重定向到 fd

gettoken & _gettoken

```c
/* Overview:
  函数功能 : 从字符串 s 中解析出下一个 token 标记
  后置条件 : 把 p1 设置成当前找到的 token 的起始位置，把 p2 设置成该 token 结束后的位置(左闭右开区间)
 */
int _gettoken(char *s, char **p1, char **p2) {
        *p1 = 0;
        *p2 = 0;
// 空串
        if (s == 0) {
                return 0;
        }
// 跳过空白字符
        while (strchr(WHITESPACE, *s)) {
                *s++ = 0;
        }
        if (*s == 0) {
                return 0;
        }
// 找到特殊符号，管道等
        if (strchr(SYMBOLS, *s)) {
                int t = *s;
                *p1 = s;
                *s++ = 0;
                *p2 = s;
                return t;
        }
// 处理一个普通单词
        *p1 = s;
        while (*s && !strchr(WHITESPACE SYMBOLS, *s)) {
                s++;
        }
        *p2 = s;
        return 'w';
}

int gettoken(char *s, char **p1) {
        static int c, nc;
        static char *np1, *np2;
// 只有初始化的时候，会传一个 s 进去，其余时候调用的时候都是传一个 0 进去，此时会把第一个 token 解析到 nc 中，np2 指向下一个 token 的起始位置，np1 指向第一个 token 的起始位置
        if (s) {
                nc = _gettoken(s, &np1, &np2);
                return 0;
        }
// 正式调用的时候(以第一次为例)，先把第一个 token 从 nc 存入到 c 中，第一个 token 的起始位置从 np1 存入到 p1 中，接着去解析 np2 作为起始的字符串(也就是去掉第一个 token 的部分)，把解析到的第二个 token，第二个 token 的起始位置，第二个 token 结束后的下一个位置分别存入 nc，np1，np2 的位置中
        c = nc;
        *p1 = np1;
        nc = _gettoken(np2, &np1, &np2);
        return c;
}
```

runcmd

```
/* Overview :
  函数功能 : 执行用户输入的命令行字符串 s
*/
void runcmd(char *s) {
// 第一次调用，用 s 初始化 np1，并且获得第一个 token 存入 nc 中，但是实际返回的是 0
        gettoken(s, 0);
// argv 用于存储命令和参数列表，最大个数为 MAXARGS
        char *argv[MAXARGS];

// rightpipe 表示当前命令是否存在管道符（|）连接的右侧命令，初始为 0
        int rightpipe = 0;

// parsecmd 解析命令并填充 argv，同时检测是否有管道命令，若有则将 rightpipe 设置为右侧命令的进程 ID
        int argc = parsecmd(argv, &rightpipe);

// 如果命令为空（argc 为 0），则直接返回
        if (argc == 0) {
                return;
        }

// 给 argv 的最后一个元素设置为 NULL，符合 exec 系列函数要求的参数数组格式
        argv[argc] = 0;

// 使用 spawn 启动一个新的子进程来执行 argv[0]（即命令），并传入参数列表 argv
        int child = spawn(argv[0], argv);

// 调用 close_all 关闭所有不需要的文件描述符（如 pipe 用过的端口）
        close_all();

// 如果 spawn 返回的子进程 ID >= 0，说明成功启动子进程，调用 wait 等待其执行结束
        if (child >= 0) {
                wait(child);
        } else {
// 否则说明 spawn 失败，打印错误信息
                debugf("spawn %s: %d\n", argv[0], child);
        }

// 如果有管道右侧命令的进程（rightpipe 非零），也等待它完成
        if (rightpipe) {
                wait(rightpipe);
        }

// 当前父进程结束，退出 shell 命令执行函数
        exit();
}
```

readline

```
/* Overview :
  函数功能 : 从键盘读取一行用户输入，存储到 buf 中，直到读到换行符或读满指定的长度
  入参含义 : buf 用来接收读入的内容，n 表示最多读取的字符数量
*/
void readline(char *buf, u_int n) {
```

```
        int r;
// 实现逐个字符读取的循环，最多读取 n 个字符
        for (int i = 0; i < n; i++) {
                if ((r = read(0, buf + i, 1)) != 1) {
                        if (r < 0) {
                                debugf("read error: %d\n", r);
                        }
                        exit();
                }
// \b 是退格键，0x7f 是删除键，如果 i > 0 则要倒退两个位置，因为外层循环还要 i++，如果本来就在 i == 0，则设成 -1 用来消除外层循环的 i++ 即可
                if (buf[i] == '\b' || buf[i] == 0x7f) {
                        if (i > 0) {
                                i -= 2;
                        } else {
                                i = -1;
                        }
// 如果不是退格键，就在屏幕上打印一个退格键，使得光标真实向前移动一格位置
                        if (buf[i] != '\b') {
                                printf("\b");
                        }
                }
// 如果用户输入回车或者换行，就结束输入，并把该位置设成结尾 0
                if (buf[i] == '\r' || buf[i] == '\n') {
                        buf[i] = 0;
                        return;
                }
        }
// 如果输入超长，则丢弃掉后面的内容，并且清空缓冲区开头，防止上次没读完的数据影响下一次的读取
        debugf("line too long\n");
        while ((r = read(0, buf, 1)) == 1 && buf[0] != '\r' && buf[0] != '\n') {
                ;
        }
        buf[0] = 0;
}
```

main

main 函数是一个最小操作系统的 Shell 程序入口，它完成以下主要任务：

1. 打印欢迎信息
2. 支持两种运行模式：**交互式**（用户键盘输入）和**脚本模式**（从文件中读取命令）
3. 支持将命令行作为参数执行
4. 每条命令通过 `fork()` 创建子进程并调用 `runcmd()` 执行
5. 支持可选地打印命令（`-x` 参数）

```
/* Overview :
  Shell 命令解释器的主程序，用于实现用户交互，命令读取，命令解析，命令执行等功能
  入参含义 : argc - 参数个数；argv - 存命令本身的二维数组
*/
int main(int argc, char **argv) {
        int r;
// interactive 用来判断是否为控制台输入，echocmds 用来判断是否需要打印，也就是开启回显(类似于 cat 这样的指令)
        int interactive = iscons(0);
```

```
        int echocmds = 0;

 printf("\n:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::\n");
        printf("::                                                       ::\n");
        printf("::                     MOS Shell 2024                    ::\n");
        printf("::                                                       ::\n");

 printf("::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::\n");
// ARGBEGIN / ARGEND 是用于解析命令行参数的宏，i 表示强制设置为交互模式，x 表示开启命令回显，usage 是打印失败时的信息
        ARGBEGIN {
        case 'i':
                interactive = 1;
                break;
        case 'x':
                echocmds = 1;
                break;
        default:
                usage();
        }
        ARGEND

// main 函数参数个数超过 1，出错
        if (argc > 1) {
                usage();
        }
// 有 1 个参数，说明用户提供了一个脚本文件路径，需要从文件中读取命令
        if (argc == 1) {
                close(0);
// 使用 open 打开这个文件，代替标准读入 fd(0)
                if ((r = open(argv[0], O_RDONLY)) < 0) {
                        user_panic("open %s: %d", argv[0], r);
                }
                user_assert(r == 0);
        }
// 主循环，shell 不断接受用户输入的命令行进行处理，直到用户退出
        for (;;) {
// 如果是交互模式，interactive 为 1，则输出 $
                if (interactive) {
                        printf("\n$ ");
                }
// 从用户输入或者脚本中读取一行命令，存入 buf 中
                readline(buf, sizeof buf);
// 如果以 # 开头，说明是注释，跳过这一行
                if (buf[0] == '#') {
                        continue;
                }
// 如果开启了回显，echocmds 为 1，则在执行命令前先输出源命令内容
                if (echocmds) {
                        printf("# %s\n", buf);
                }
// 创建子进程执行命令
                if ((r = fork()) < 0) {
                        user_panic("fork: %d", r);
```

```
60                  }
61  // 如果是子进程，r == 0，则让他去执行这条命令，如果是父进程，则父进程需要等待子进程结束，
    wait 子进程的 envid 即可
62                  if (r == 0) {
63                          runcmd(buf);
64                          exit();
65                  } else {
66                          wait(r);
67                  }
68          }
69          return 0;
70  }
```

已经实现的 shell 指令

我们后面要做的就是接着写这样的指令文件，让他被编译成二进制文件后，能够被子进程调用

ls.c -> ls.b

用于模拟 ls 指令的功能

默认行为：列出目录内容

参数：

1. -d：显示目录本身而非其内容
2. -l：显示文件大小和类型信息（是否为目录）
3. -F：目录名后面添加 / 用来表示目录

```
1   #include <lib.h>
2
3   // 参数位标志数组，比如如果有参数 -lF，那么 flag 对应 l 和 F 的位置会变成 1
4   int flag[256];
5
6   void lsdir(char *, char *);
7   void ls1(char *, u_int, u_int, char *);
8
9   // 处理一个文件或目录
10  void ls(char *path, char *prefix) {
11          int r;
12          struct Stat st;
13  // 首先调用 stat 获取 path 路径下文件或目录的元信息，保存在 stat 结构体中
14          if ((r = stat(path, &st)) < 0) {
15                  user_panic("stat %s: %d", path, r);
16          }
17  // 如果当前文件是目录，且没有 -d 参数，也就是说用户希望展开目录内容而不是要看目录本身，就
    调用 lsdir 处理，否则就直接调用 ls1 打印这个目录本身的信息
18          if (st.st_isdir && !flag['d']) {
19                  lsdir(path, prefix);
20          } else {
21                  ls1(0, st.st_isdir, st.st_size, path);
22          }
23  }
24
25  // 函数功能 : 遍历目录，列出并打印目录内容
26  void lsdir(char *path, char *prefix) {
27          int fd, n;
28          struct File f;
```

```
// 打开文件目录
        if ((fd = open(path, O_RDONLY)) < 0) {
                user_panic("open %s: %d", path, fd);
        }
// 不断读取文件目录中的内容，readn 表示读取一个完整的 struct File，也就是文件项，如果得到的文件项 name 不为 0，则表示此项有效，直接调用 ls1 输出文件项的信息
        while ((n = readn(fd, &f, sizeof f)) == sizeof f) {
                if (f.f_name[0]) {
                        ls1(prefix, f.f_type == FTYPE_DIR, f.f_size, f.f_name);
                }
        }
// 检查读取完整性，必须从目录下读出所有的文件项
        if (n > 0) {
                user_panic("short read in directory %s", path);
        }
        if (n < 0) {
                user_panic("error reading directory %s: %d", path, n);
        }
}

// 函数功能 : 打印出单个文件或目录的信息
void ls1(char *prefix, u_int isdir, u_int size, char *name) {
// 用于决定是否要在 prefix 和 name 之间加分隔符 /
        char *sep;

// 如果是 -l 模式，打印文件的大小，文件类型，d 表示目录，- 表示文件
        if (flag['l']) {
                printf("%11d %c ", size, isdir ? 'd' : '-');
        }
// 如果有前缀路径 prefix，且前缀路径的 prefix 不是 / 结尾，需要手动加上 /，存入 sep 中
        if (prefix) {
                if (prefix[0] && prefix[strlen(prefix) - 1] != '/') {
                        sep = "/";
                } else {
                        sep = "";
                }
                printf("%s%s", prefix, sep);
        }
// 打印文件名
        printf("%s", name);
// 如果启用了 -F 参数，且当前文件是目录，则在末尾添加 /
        if (flag['F'] && isdir) {
                printf("/");
        }
        printf(" ");
}

// 用来提示错误信息的函数
void usage(void) {
        printf("usage: ls [-dFl] [file...]\n");
        exit();
}

// 主程序逻辑
int main(int argc, char **argv) {
        int i;
```

```
84
85          ARGBEGIN {
86          default:
87                  usage();
88  // 遇到 d F l 这些参数的时候，对应 flag 中该设置标记
89          case 'd':
90          case 'F':
91          case 'l':
92                  flag[(u_char)ARGC()]++;
93                  break;
94          }
95          ARGEND
96  // 如果不提供参数，就默认输出根目录，反之，就对每个参数都依次调用 ls 进行输出
97          if (argc == 0) {
98                  ls("/", "");
99          } else {
100                 for (i = 0; i < argc; i++) {
101                         ls(argv[i], argv[i]);
102                 }
103         }
104         printf("\n");
105         return 0;
106 }
```

cat.c -> cat.b

函数功能：读取文件内容并且写到标准输出（也就是终端），支持从标准输入中读取内容，也支持从一个或多个文件中读取内容

```
1  #include <lib.h>
2
3  // 用于暂存输出内容的数组，体现 I/O 缓冲机制
4  char buf[8192];
5
6  // 函数功能 : 把文件描述符 f 所指向的内容输出到标准输出
7  void cat(int f, char *s) {
8          long n;
9          int r;
10
11 // 每次调用 read 从文件描述符 f 所对应的文件中最多读取 8192 个字节，存入 buf 中，接着调用 write 把刚读取到的 n 个字节从 buf 写入到标准输出(标准输出对应的文件描述符为 1)
12         while ((n = read(f, buf, (long)sizeof buf)) > 0) {
13                 if ((r = write(1, buf, n)) != n) {
14                         user_panic("write error copying %s: %d", s, r);
15                 }
16         }
17         if (n < 0) {
18                 user_panic("error reading %s: %d", s, n);
19         }
20 // 如果读写字数不同，或读取失败，都会报错
21 }
22
23 // 主程序逻辑
24 int main(int argc, char **argv) {
25         int f, i;
26
```

```
// 如果用户没有传入文件名，则默认是标准输入(fd == 0)获取内容并输出，文件名显示为 stdin，此时 argc == 1 的参数是 cat 这条指令本身
        if (argc == 1) {
                cat(0, "<stdin>");
        } else {
                for (i = 1; i < argc; i++) {
// 反之如果有参数，就依次打开参数中所代表的文件，把这些文件的内容调用 cat 进行输出即可
                        f = open(argv[i], O_RDONLY);
                        if (f < 0) {
                                user_panic("can't open %s: %d", argv[i], f);
                        } else {
                                cat(f, argv[i]);
                                close(f);
                        }
                }
        }
        return 0;
}
```

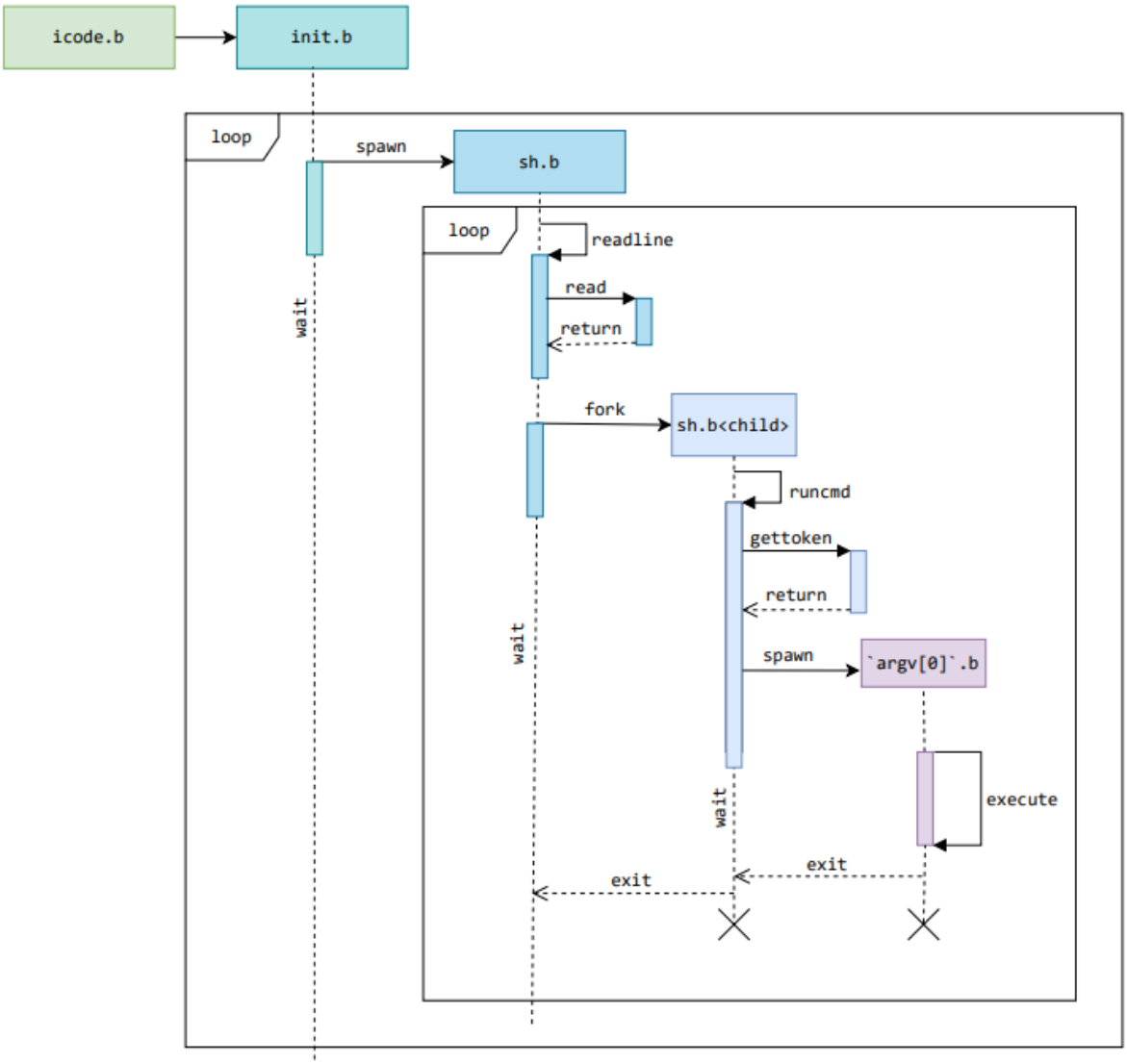
shell 指令执行流程

我们实现的 shell 是如何完成一条完整指令的执行过程的?

以 cat.b file.txt 的执行为例进行解析:

1. 首先 sh.c 中的 main 函数会通过 readline 函数从标准输入流中读取这条指令，以一个字符串的形式整体读入，此时指令还没有参数化，main 会调用 runcmd 尝试执行这条指令
2. runcmd 会调用 parsecmd 对指令流进行解析 ，这是对指令参数化的过程，该字符串会被拆成 argv 参数的形式，其中 argv [0] 和 argv [1] 分别表示 cat.b 和 file.txt 这两个参数
3. parsecmd 会先调用 gettoken，使用类似递归下降的处理流程，对 readline 获取的字符串进行解析，每解析出来一个参数就存入 argv 中，其中 argv [0] 最特殊，存储的是**指令**本身的名字，这个参数将来会用于子进程对目标 ELF 文件的索引
4. parsecmd 在解析完参数后，runcmd 会调用 spawn 创建子进程并为其加载 ELF 文件，而要加载的 ELF 文件的路径（spawn 第一个入参）就是 argv [0] 中的命令名称
5. 在 spawn 中会完成对目标 ELF 文件内容的加载，并且创建相关的子进程完成这个 ELF 文件内容的执行，而这个 ELF 文件对应的就是 cat.c 文件被编译成 cat.b 二进制文件的结果
6. 换言之，命令本身的解析是完全由 shell 部分完成的，子进程被创建后直接拿到了一个正确的 ELF 文件，这里的正确性是由待解析命令和 ELF 文件名一一对应所决定的，也就是说，其实子进程根本不知道自己执行的是什么指令，他只是执行了某个路径指向的特定的 ELF 文件

具体流程可以参考下面这张图:


icode.b
init.b
loop
spawn
sh.b
loop
readline
read
return
wait
fork
sh.b<child>
runcmd
gettoken
return
wait
spawn
`argv[0]`.b
execute
wait
exit
exit