操作系统第七次实验报告-Lab6

以下为学生闫思桥(19241091)的$Lab6$实验报告

$Part1$ 实验思考题

Thinking 6.1

```c
#include <stdlib.h>
#include <unistd.h>

int fildes[2];
/* buf size is 100 */
char buf[100];
int status;

int main(){
    status = pipe(fildes);

    if (status == -1 ) {
        /* an error occurred */
        printf("error\n");
    }

    switch (fork()) {
        case -1: /* Handle error */
            break;

        case 0: /* Child - reads from pipe */
            close(fildes[1]); /* Write end is unused */
            read(fildes[0], buf, 100); /* Get data from pipe */
            printf("child-process read:%s",buf); /* Print the data */
            close(fildes[0]); /* Finished with pipe */
            exit(EXIT_SUCCESS);

        default: /* Parent - writes to pipe */
            close(fildes[0]); /* Read end is unused */
            write(fildes[1], "Hello world\n", 12); /* Write data on pipe */
            close(fildes[1]); /* Child will see EOF */
            exit(EXIT_SUCCESS);
    }
}
```

在示例代码中，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?

MyAnswer

修改代码如下:

```c
#include <stdlib.h>
#include <unistd.h>

int fildes[2];
/* buf size is 100 */
char buf[100];
int status;

int main(){
```

```
    status = pipe(fildes);

    if (status == -1 ) {
        /* an error occurred */
        printf("error\n");
    }

    switch (fork()) {
        case -1: /* Handle error */
            break;

        case 0: /* Parent - reads to pipe */
            close(fildes[0]); /* Read end is unused */
            write(fildes[1], "Hello world\n", 12); /* Write data on pipe */
            close(fildes[1]); /* Child will see EOF */
            exit(EXIT_SUCCESS);

        default: /* Child - writes from pipe */
            close(fildes[1]); /* Write end is unused */
            read(fildes[0], buf, 100); /* Get data from pipe */
            printf("father-process read:%s",buf); /* Print the data */
            close(fildes[0]); /* Finished with pipe */
            exit(EXIT_SUCCESS);
    }
}
```

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/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?

MyAnswer

dup 函数未经修改前的代码如下：

```
int dup(int oldfdnum, int newfdnum)
{
    int i, r;
    u_int ova, nva, pte;
    struct Fd *oldfd, *newfd;

    if ((r = fd_lookup(oldfdnum, &oldfd)) < 0) {
        return r;
    }

    close(newfdnum);
    newfd = (struct Fd *)INDEX2FD(newfdnum);
    ova = fd2data(oldfd);
    nva = fd2data(newfd);

    if ((r = syscall_mem_map(0, (u_int)oldfd, 0, (u_int)newfd,
                             ((*vpt)[VPN(oldfd)]) & (PTE_V | PTE_R |
PTE_LIBRARY))) < 0){
        goto err;
    }

    if ((* vpd)[PDX(ova)]) {
        for (i = 0; i < PDMAP; i += BY2PG) {
            pte = (* vpt)[VPN(ova + i)];
```

```
            if (pte & PTE_V) {
                // should be no error here -- pd is already allocated
                if ((r = syscall_mem_map(0, ova + i, 0, nva + i,
                                         pte & (PTE_V | PTE_R | PTE_LIBRARY))) <
0) {
                    goto err;
                }
            }
        }
    }

    return newfdnum;

err:
    syscall_mem_unmap(0, (u_int)newfd);
    for (i = 0; i < PDMAP; i += BY2PG) {
        syscall_mem_unmap(0, nva + i);
    }
    return r;
}
```

很明显：dup函数最开始的这种写法是 先打开文件描述符，然后在打开缓冲区。如果在这两个操作中间被中断了，就会让另一端误以为pipe没有打开。

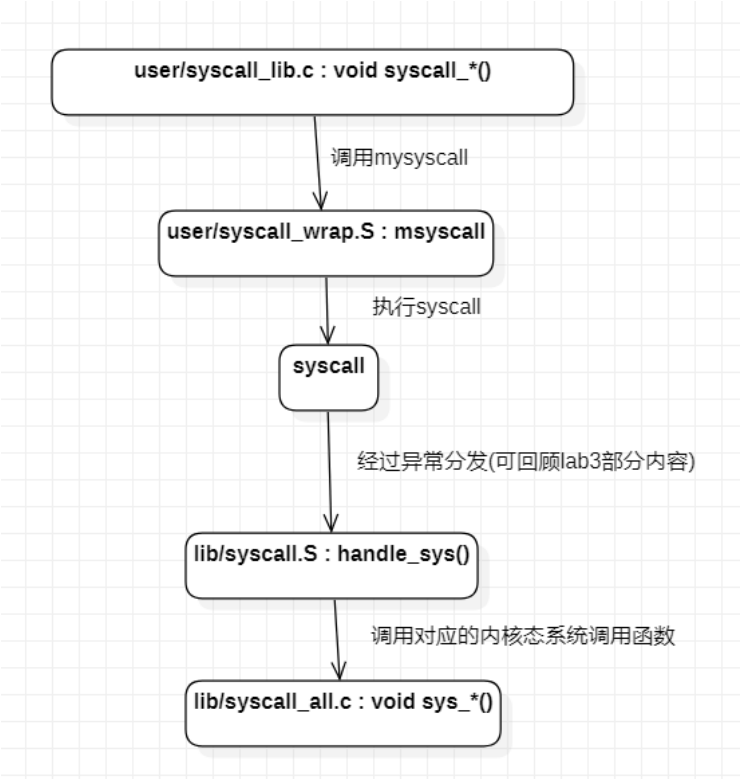
Thinking 6.3

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。

MyAnswer

原子操作的意思其实就是这个操作能够一次性完成而不被中断所打断。因为我们使用系统调用的时候关闭了对中断的响应，执行系统调用期间就不会被中断，所以系统调用是原子操作。

具体来说，所有的系统调用的流程就是：



用户态调用的系统调用函数 `syscall_*()` 的函数，实际上就是 `msyscall` 函数的套壳函数，而 `msyscall` 只是调用了syscall指令并返回。

syscall指令使得系统陷入了8号异常，系统立即进入内核态，并跳转到地址 0x8000 0080 处进行异常处理（TLB异常则是跳转到 0x8000 0000）。通过异常分发程序，选择 `handle_sys` 函数进行处理。而 `handle_sys` 函数在执行 `SAVE_ALL` 就会执行 `CLI` 禁用中断（使得SR寄存器最后一位为0）。

这样，在整个内核态下，CPU是不会响应中断的。

因此，系统调用的执行过程是不会被打断的。因此是原子操作。

Thinking 6.4

> 我们考虑之前那个预想之外的情景，它出现的最关键原因在于：pipe 的引用次数总比 fd 要高。当管道的 close 进行到一半时，若先解除 pipe 的映射，再解除 fd 的映射，就会使得 pipe 的引用次数的-1 先于 fd。这就导致在两个 unmap 的间隙，会出现 pageref(pipe) == pageref(fd) 的情况。那么若调换 fd 和 pipe 在 close 中的 unmap 顺序，能否解决这个问题呢?

仔细阅读上面这段话，并思考下列问题：

- **按照上述说法控制 pipeclose 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，用于复制文件内容。试想，如果要复制的是一个管道，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。**

MyAnswer

首先，在 `pipeclose()` 函数中，使得解除 fd 的映射先于解除 pipe 的映射。这样可以保证：在两个 unmap 的间隙，**不会出现** pageref(pipe) == pageref(fd) 的情况。

因为 pipe 的引用次数始终是大于等于单个 fd 的，只要 fd 先解除，也就是 pageref(fd) 先减去1。这样，哪怕在两个 unmap 期间进行中断，也不可能会出现 pageref(pipe) == pageref(fd) 的情况。

这样就解决了我们所说的问题。具体的改动是 user/pipe.c 中的 pipeclose() 函数。

```
static int pipeclose(struct Fd *fd)
{
    syscall_mem_unmap(0, fd);
    syscall_mem_unmap(0, fd2data(fd));
    return 0;
}
```

然后，同理可得，在增加引用次数的时候，也可能会在次序不当的时候，出现pageref(pipe) == pageref(fd) 的情况。

dup函数将一个文件描述符对应的内容映射到另一个文件描述符，并把文件内容映射过去。

原来的执行顺序，会先把文件描述页面映射，再映射文件内容，这样如果在文件内容没有映射完毕时进程被中断并替换，另一程序会因为文件描述页已经存在认为文件映射完成，从而造成错误。

所以我们也需要先增加 pageref(pipe) ，再增加 pageref(fd) ，这样就使得两个 map 的间隙，**也不会出现** pageref(pipe) == pageref(fd) 的情况。具体的改动是 user/fd.c 中用于复制管道的 dup() 函数。

```
ova = fd2data(oldfd);
nva = fd2data(newfd);
// part1
if ((* vpd)[PDX(ova)]) {
    for (i = 0; i < PDMAP; i += BY2PG) {
        pte = (* vpt)[VPN(ova + i)];

        if (pte & PTE_V) {
            // should be no error here -- pd is already allocated
            if ((r = syscall_mem_map(0, ova + i, 0, nva + i,
                                     pte & (PTE_V | PTE_R | PTE_LIBRARY))) < 0)
{
                goto err;
            }
        }
    }
}
// part2  part2一开始在 part1 之前
if ((r = syscall_mem_map(0, (u_int)oldfd, 0, (u_int)newfd,
                         ((*vpt)[VPN(oldfd)]) & (PTE_V | PTE_R | PTE_LIBRARY)))
< 0) {
    goto err;
}
```

这样就实现了：两次map和两次unmap的间隙不会出现 pageref(pipe) == pageref(fd) 的情况。

Thinking 6.5

bss 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。请回答你设计的函数是如何实现上面这点的?

MyAnswer

下面详细解释我在 lab3 中完成的 `load_icode_mapper()` 函数。

```
static int load_icode_mapper(u_long va, u_int32_t sgsize,
                             u_char *bin, u_int32_t bin_size, void *user_data)
{
```

```
    struct Env *env = (struct Env *)user_data;
    struct Page *p = NULL;
    u_long i;
    int r;
    u_long offset = va - ROUNDDOWN(va, BY2PG); // 如果 va 没有和BY2PG对齐，则势必会产生不为 0 的offset

    /* Step 1: load all content of bin into memory. */
    r = 0;
    if (offset > 0){
        p = page_lookup(env->env_pgdir, va, NULL);  // 先确认起始地址va所在的页（进程空间的页）是不是已经有了对应的物理内存控制块（即已经申请过对应的物理页面）
        if (p == 0){// 说明并不存在上面说的这种情况
            // 那就需要为 未能整页对齐的开始部分新申请一页物理内存
            if ( (r = page_alloc(&p))<0 ){
                return r;
            }
             page_insert(env->env_pgdir, p, va, PTE_R);
            // 在内核中虚拟空间复制完毕后，就需要建立起虚拟空间和物理空间的映射关系
            //将一级页表基地址 pgdir 对应的两级页表结构中 va 这一虚拟地址 所对应的二级页表项中
            //填入p这个物理内存控制块对应的物理页面的页号，并设置页表权限为可写
            //还是要注意，进程的一级页表和二级页表都在内核虚拟空间中有对应的内存分配
        }
        /* 如果说要加载到的位置前面有前一个段申请好的页面，就不需要申请页和建立映射 */
        r = MIN(BY2PG - offset, bin_size); // 这里需要比较 va占用的第一页的 整页 减去offset后的大小 和整个 ELF 文件 大小，防止 ELF 只能占据不到一页
        // 将开头部分复制到申请的物理控制块所映射的 内核虚拟空间 。
        bcopy((void *)bin, (void *)(page2kva(p)+offset), r);
    }

    // 这里的 r 考虑到了两种情况：offset为 0时，r就是0；offset不为 0时，r是 MIN(BY2PG - offset, bin_size)。总之是让开始遍历的地址页对齐。
    for (i = r; i < bin_size; i += BY2PG) {
        if ((r = page_alloc(&p)) < 0) {// 经过上面的调整，i=r以后，i就已经整页对齐了
            return r;
        }
        bcopy((void *)(bin + i), (void *)page2kva(p), MIN(BY2PG, bin_size - i));// 这里要时刻留意二进制文件剩下的部分够不够一页
        page_insert(env->env_pgdir, p, va + i, PTE_R);
    }
    /* Step 2: alloc pages to reach `sgsize` when `bin_size` < `sgsize`.
     * hint: variable `i` has the value of `bin_size` now! */
    // 注意 .bss是不需要复制到内存中的 所以，即使这里有可能 ELF 文件在最后一页的残余部分 加上 .bss部分也不能超过最后一页，此时 i >= sgsize，但是因为不需要复制
    // 并且要求文件占不够的直接置零就好了，所以这样跳过了下面的代码，也是没问题的。
    // 而如果 ELF 文件在最后一页的残余部分 加上 .bss部分 后超过了最后一页，那么 i < sgsize，执行代码
    while (i < sgsize) {
        if ((r = page_alloc(&p)) != 0) {
            return r;
        }
        page_insert(env->env_pgdir, p, va + i, PTE_R);
        i += BY2PG;
    }
    return 0;
}
```

通过最后一部分 while 循环，就可以实现，为 .bss 段在内存中申请空间，并且置零。

Thinking 6.6

为什么我们的 *.b 的 text 段偏移值都是一样的，为固定值?

MyAnswer

因为在链接器里我们把text的段开头的偏移量都设为了相同的值。

文件中是分段管理的，各段起始地址固定。

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 指令。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 指令是内部指令而不是外部指令?

MyAnswer

先放一些我搜集到的基础知识。

> linux中的命令大致可分为两类，内部命令和外部命令。
>
> 内部命令：也称shell内嵌命令；
>
> 外部命令：存放在一个文件中，使用时需要去文件中查找，这些文件被定义在$PATH。
>
> 所谓的内外之分是相对shell来分的。内部命令在系统启动时就调入内存，是常驻内存的，所以执行效率高。外部命令是系统的软件功能，用户需要时才从硬盘中读入内存。
>
> (1) 内部命令实际上是shell程序的一部分，其中包含的是一些比较简单的linux系统命令，这些命令由shell程序识别并在shell程序内部完成运行，通常在linux系统加载运行时shell就被加载并驻留在系统内存中。内部命令是写在bashy源码里面的，其执行速度比外部命令快，因为解析内部命令shell不需要创建子进程。比如：exit，history，cd，echo等。
> (2) 外部命令是linux系统中的实用程序部分，因为实用程序的功能通常都比较强大，所以其包含的程序量也会很大，在系统加载时并不随系统一起被加载到内存中，而是在需要时才将其调用内存。通常外部命令的实体并不包含在shell中，但是其命令执行过程是由shell程序控制的。shell程序管理外部命令执行的路径查找、加载存放，并控制命令的执行。外部命令是在bash之外额外安装的，通常放在/bin，/usr/bin，/sbin，/usr/sbin……等等。可通过“echo $PATH”命令查看外部命令的存储路径，比如：ls、vi等。
>
> type命令可以查看命令类型，以区别是内部命令还是外部命令
>
> ```
> [root@centos7 ~]# type cd
> cd is a shell builtin
> [root@centos7 ~]# type ls
> ls is aliased to `ls --color=auto'
> [root@centos7 ~]# type ifconfig
> ifconfig is /usr/sbin/ifconfig
> ```
>
> 可以看到，cd为shell内嵌命令，ls命令为ls --color=auto的别名，ifconfig命令为外部命令在文件/usr/sbin/ifconfig中。

在MOS中，我们用到的所有shell命令应该**几乎都是 外部命令**。因为除了两种情况以外，剩下的都需要执行fork函数生成一个子shell进程。从下面的代码里可以验证这一点。

```
for(;;){
    if (interactive)
```

```
        fwritef(1, "\n$ ");
    readline(buf, sizeof buf);

    if (buf[0] == '#')
        continue;
    if (echocmds)
        fwritef(1, "# %s\n", buf);
    if ((r = fork()) < 0)
        user_panic("fork: %e", r);
    if (r == 0) {
        runcmd(buf);
        exit();
        return;
    } else
        wait(r);
}
```

下面简单说说我个人对于为什么 Linux 的 cd 指令是内部指令的看法

Linux 的 cd 指令是内部指令，一方面，cd 指令的使用频率很高，如果是写在shell主进程的源码里，这样不需要fork一个子shell进程来响应，无疑会提高shell的性能；另一方面，cd指令本身是将当前进程所处的环境切换至一个新的目录，是切实地改变了当前的工作路径。因此当cd指令执行完毕时，这个切换效果必须作用到主shell进程上。所以，cd指令作为内部指令更好。

不过我们的shell似乎不支持 cd 指令?

Thinking 6.8

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。

MyAnswer

我们的测试进程从 user/icode 开始执行，里面调用了 spawn(init.b), 在 完成了 spawn 后，创建了 init.b 进程。init.b 进程调用 spawn(sh.b)，创建了 sh.b 进程，也就是我们的 shell。

因此，我们可以首先来看看 user/init.c，果不其然发现了一些端倪：`opencons()` 函数。

跳转到该函数定义的地方，发现一句代码：`fd->fd_dev_id = devcons.dev_id;` ，这就是

```
// user/init.c
    close(0);
    if ((r = opencons()) < 0)
        user_panic("opencons: %e", r);
    if (r != 0)
        user_panic("first opencons used fd %d", r);
    if ((r = dup(0, 1)) < 0)
        user_panic("dup: %d", r);
write(1,"LALA",4);

    for (;;) {
        writef("init: starting sh\n");
        r = spawnl("sh.b", "sh", (char*)0);
        if (r < 0) {
            writef("init: spawn sh: %e\n", r);
            continue;
        }
        wait(r);
// user/console.c
int opencons(void)
{
```

```
    int r;
    struct Fd *fd;

    if ((r = fd_alloc(&fd)) < 0)
        return r;
    if ((r = syscall_mem_alloc(0, (u_int)fd, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
        return r;
    fd->fd_dev_id = devcons.dev_id; // 这一句将文件描述符的设备id定为了控制台id
    fd->fd_omode = O_RDWR;
    return fd2num(fd);
}
```

init.b 进程通过 spawn 生成 shell 进程。init.b 申请了 console（控制台）作为标准 输入输出，而这个 console 就是通过共享页面映射给 shell 进程，使得 shell 进程可以通过控制台与用户交互。

顺便复习一下不同的设备。

在 user/fd.c 中记录了实验中所有设备的数组 devtable。

```
static struct Dev *devtab[] = {
    &devfile,
    &devcons,
    &devpipe,
    0
};
```

三个设备对应的定义分别为：

```
// user/file.c
struct Dev devfile = {
    .dev_id =    'f',
    .dev_name =  "file",
    .dev_read =  file_read,
    .dev_write = file_write,
    .dev_close = file_close,
    .dev_stat =  file_stat,
};
// user/console.c
struct Dev devcons =
{
    .dev_id=    'c',
    .dev_name=  "cons",
    .dev_read=  cons_read,
    .dev_write= cons_write,
    .dev_close= cons_close,
    .dev_stat=  cons_stat,
};
// user/pipe.c
struct Dev devpipe =
{
    .dev_id=    'p',
    .dev_name=  "pipe",
    .dev_read=  piperead,
    .dev_write= pipewrite,
    .dev_close= pipeclose,
    .dev_stat=  pipestat,
};
```

Thinking 6.9

在你的 shell 中输入指令 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？**

MyAnswer

- **请问你可以在你的 shell 中观察到几次 spawn？分别对应哪个进程？**

2次，如下图所示：

```
$ ls.b | cat.b > motd

[00001c03] pipecreate

[00001c03] SPAWN: ls.b

serve_open 00001c03 ffff000 0x0

serve_open 00002404 ffff000 0x1

[00002404] SPAWN: cat.b

serve_open 00002404 ffff000 0x0


::::::::::spawn size : 20  sp : 7f3fdfe8::::::::


::::::::::spawn size : 20  sp : 7f3fdfe8::::::::
```

由 `runcmd()` 函数的相关代码可以看出，这两次spawn分别对应着 进程 `00001c03` 和 进程 `00002404` 。创建管道的进程是前者。

- **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？**

```
::::::::::spawn size : 20  sp : 7f3fdfe8::::::::
pageout:        @@@___0x40a400___@@@  ins a page
serve_open 00003406 ffff000 0x0
pageout:        @@@___0x40c000___@@@  ins a page
serve_open 00003406 ffff000 0x0
[00003406] destroying 00003406
[00003406] free env 00003406
i am killed ...
[00002c05] destroying 00002c05
[00002c05] free env 00002c05
i am killed ...
[00002404] destroying 00002404
[00002404] free env 00002404
i am killed ...
[00001c03] destroying 00001c03
[00001c03] free env 00001c03
i am killed ...
```

4次，这4次进程销毁分别对应着进程 `0000 3406` `0000 2c05` `0000 2404` `00001c03` 。

我们的shell是通过：主shell进程在遇到有命令输入的时候，运行 fork() 函数，创建一个子shell进程，执行 `runcmd()` 函数来完成对应命令。

Part2 实验难点图示

Part2.1 管道

初窥管道

在 lab4 中，我们已经学习过一种进程间通信 (IPC,Inter-Process Communication)的方式——共享内存。而这次实验中的管道，其实也是进程间通信的一种方式。

通俗来讲，管道就像家里的自来水管：一端用于注入水，一端用于放出水，且水只能在一个方向上流动，而不能双向流动，所以说管道是典型的单向通信。管道又叫做匿名管道，**只能用在具有公共祖先的进程之间使用，通常使用在父子进程之间通信。**

创建管道

在 Unix 中，管道由 pipe 函数创建，函数原型如下：

```
#include<unistd.h>
int  pipe(int fd[2]);    //成功返回0，否则返回-1;
```

参数fd 返回两个文件描述符，fd[0]对应读端，fd[1]对应写端。

为了更好地理解管道实现的原理，同样，我们先来做实验亲自体会一下。

（实验代码参考http://pubs.opengroup.org/onlinepubs/9699919799/functions/pipe.html）

```c
#include <stdlib.h>
#include <unistd.h>

int fildes[2];
/* buf size is 100 */
char buf[100];
int status;

int main(){
    status = pipe(fildes);

    if (status == -1 ) {
        /* an error occurred */
        printf("error\n");
    }

    switch (fork()) {
        case -1: /* Handle error */
            break;

        case 0: /* Child - reads from pipe */
            close(fildes[1]); /* Write end is unused */
            read(fildes[0], buf, 100); /* Get data from pipe */
            printf("child-process read:%s",buf); /* Print the data */
            close(fildes[0]); /* Finished with pipe */
            exit(EXIT_SUCCESS);

        default: /* Parent - writes to pipe */
            close(fildes[0]); /* Read end is unused */
            write(fildes[1], "Hello world\n", 12); /* Write data on pipe */
            close(fildes[1]); /* Child will see EOF */
            exit(EXIT_SUCCESS);
    }
}
```

示例代码实现了从父进程向子进程发送消息”Hello,world”，并且在子进程中打印到屏幕上。它演示了管道在父子进程之间通信的基本用法：

在 pipe 函数之后，调用 fork 来产生一个子进程，之后在父子进程中执行不同的操作。

在示例代码中，父进程操作写端，而子进程操作读端。同时，示例代码也为我们演示了使用 pipe 系统调用的习惯：

fork 之后，进程在开始读或写管道之前都会关掉不会用到的管道端。

管道的本质

从本质上说，管道是一种只在内存中的文件。

在 UNIX 中使用 pipe 系统调用时，进程中会打开两个新的文件描述符：一个只读端和一个只写端，而这两个文件描述符都映射到了同一片内存区域。但这样建立的管道的两端都在同一进程中，而且构建出的管道两端是两个匿名的文件描述符，这就让其他进程无法连接该管道。在 fork 的配合下，才能在父子进程间建立起进程间通信管道，这也是匿名管道只能在具有亲缘关系的进程间通信的原因。

管道的测试

我们下面就来填充函数实现匿名管道的功能。思考刚才的代码样例，要实现匿名管道，至少需要有两个功能：管道读取、管道写入。

要想实现管道，首先我们来看看本次实验我们将如何测试。lab6 关于管道的测试有两个，分别是 `user/testpipe.c` 与 `user/testpiperace.c`。

首先我们来观察 `testpipe.c` 的内容。

```
#include "lib.h"

char *msg = "Now is the time for all good men to come to the aid of their
party.";

void umain(void)
{
    char buf[100];
    int i, pid, p[2];

    if ((i = pipe(p)) < 0) {
        user_panic("pipe: %e", i);
    }

    if ((pid = fork()) < 0) {
        user_panic("fork: %e", i);
    }

    if (pid == 0) { // 子进程读取管道
        writef("[%08x] pipereadeof close %d\n", env->env_id, p[1]);
        close(p[1]); // 关闭管道写端
        writef("[%08x] pipereadeof readn %d\n", env->env_id, p[0]);
        i = readn(p[0], buf, sizeof buf - 1); // 从管道中读出数据
        if (i < 0) {
            user_panic("read: %e", i);
        }

        buf[i] = 0;
        if (strcmp(buf, msg) == 0) {
            writef("\npipe read closed properly\n");
        } else {
            writef("\ngot %d bytes: %s\n", i, buf);
        }
        exit();
    } else {    // 父进程写入管道
        writef("[%08x] pipereadeof close %d\n", env->env_id, p[0]);
        close(p[0]); // 关闭管道读端
        writef("[%08x] pipereadeof write %d\n", env->env_id, p[1]);
        if ((i = write(p[1], msg, strlen(msg))) != strlen(msg)) {
            user_panic("write: %e", i);
        }
        close(p[1]);
    }

    wait(pid);
    if ((i = pipe(p)) < 0) {
        user_panic("pipe: %e", i);
    }

    if ((pid = fork()) < 0) {
        user_panic("fork: %e", i);
    }

    if (pid == 0) {
        close(p[0]);
```

```
        for (;;) {
            writef(".");
            if (write(p[1], "x", 1) != 1) {
                break;
            }
        }
        writef("\npipe write closed properly\n");
    }

    close(p[0]);
    close(p[1]);
    wait(pid);
    writef("pipe tests passed\n");
}
```

实际上可以看出，测试文件使用 pipe 的流程和示例代码是一致的。

先使用函数 pipe(int p[2])创建了管道，读端的文件控制块编号2为 p[0]，写端的文件控制块编号为 p[1]。之后使fork() 创建子进程，**注意这时父子进程使用 p[0] 和 p[1] 访问到的内存区域一致**。之后子进程关闭了 p[1]，从 p[0] 读；父进程关闭了 p[0]，从 p[1] 写入管道。

先复习一些宏的定义

```
#define MAXFD 32
#define FILEBASE 0x60000000
#define FDTABLE (FILEBASE-PDMAP)    // 文件描述符的表格

#define INDEX2FD(i) (FDTABLE+(i)*BY2PG)
// 由i是0-31，则具体的范围就是 0x6000 0000 - 4M, 0x6000 0000 - 4M + 4K * 32
#define INDEX2DATA(i)   (FILEBASE+(i)*PDMAP)
// 由i是0-31，则具体的范围就是 0x60000000 - 0x6C000000
// 一个文件描述符对应着一个PDMAP = 4M的大小，同时我们记得，在上个实验中，我们可以推出，单个文件的最大大小就是4M
```

工具函数 fd_alloc()

从0到 $\mathrm{MAXFD}-1=31$ 之间最小的 i 使得该第 i 个文件描述符还没有映射到对应的 文件描述符的页面 上。

返回分配到的第 i 个文件描述符应该被映射到的虚拟地址。

每一个文件描述符有存放它本身的一个页面，还有它控制的一个数据区域 PDMAP 。

```
int fd_alloc(struct Fd **fd)
{
    // Find the smallest i from 0 to MAXFD-1 that doesn't have
    // its fd page mapped.  Set *fd to the fd page virtual address.
    // (Do not allocate a page.  It is up to the caller to allocate
    // the page.  This means that if someone calls fd_alloc twice
    // in a row without allocating the first page we return, we'll
    // return the same page the second time.)
    // Return 0 on success, or an error code on error.
    u_int va;
    u_int fdno;

    for (fdno = 0; fdno < MAXFD - 1; fdno++) {
        va = INDEX2FD(fdno);
```

```
        if (((* vpd)[va / PDMAP] & PTE_V) == 0) {
            *fd = (struct Fd *)va;
            return 0;
        }

        if (((* vpt)[va / BY2PG] & PTE_V) == 0) {   //the fd is not used
            *fd = (struct Fd *)va;
            return 0;
        }
    }
    return -E_MAX_OPEN;
}
```

lab4 的实验中，我们的 fork 实现是完全遵循 Copy-On-Write 原则的，即对于所有用户态的地址空间都进行了 PTE_COW 的设置。但实际上写时复制并不完全适用，至少在我们当前情景下是不允许写时拷贝。

为什么呢？我们来看看 pipe 函数中的关键部分就能知晓答案：

```
int pipe(int pfd[2])
{
    int r, va;
    struct Fd *fd0, *fd1;

    // allocate the file descriptor table entries
    if ((r = fd_alloc(&fd0)) < 0
    ||  (r = syscall_mem_alloc(0, (u_int)fd0, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
        goto err;

    if ((r = fd_alloc(&fd1)) < 0
    ||  (r = syscall_mem_alloc(0, (u_int)fd1, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
        goto err1;

    // allocate the pipe structure as first data page in both
    va = fd2data(fd0);
    if ((r = syscall_mem_alloc(0, va, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
        goto err2;
    if ((r = syscall_mem_map(0, va, 0, fd2data(fd1), PTE_V|PTE_R|PTE_LIBRARY)) <
0)
        goto err3;
    ...
}
```

在 pipe 中，首先分配两个文件描述符并为其分配空间，

然后将一个管道作为这两个文件描述符数据区的第一页数据，从而使得这两个文件描述符能够共享一个管道的数据缓冲区。

（具体来说，首先是为fd0对应的数据区PDMAP申请一页空间，然后把这一页空间的数据用 `syscall_mem_map()` 函数共享给fd1对应的数据区PDMAP）

> 复习：一些转换函数
>
> fd2data()

```
#define FILEBASE 0x60000000
#define INDEX2FD(i) (FDTABLE+(i)*BY2PG)
#define INDEX2DATA(i)   (FILEBASE+(i)*PDMAP)
u_int fd2data(struct Fd *fd){
    return INDEX2DATA(fd2num(fd));
}
```

父子进程和管道的数据缓冲区的关系

下面我们使用一张图来表示父子进程与管道的数据缓冲区的关系：

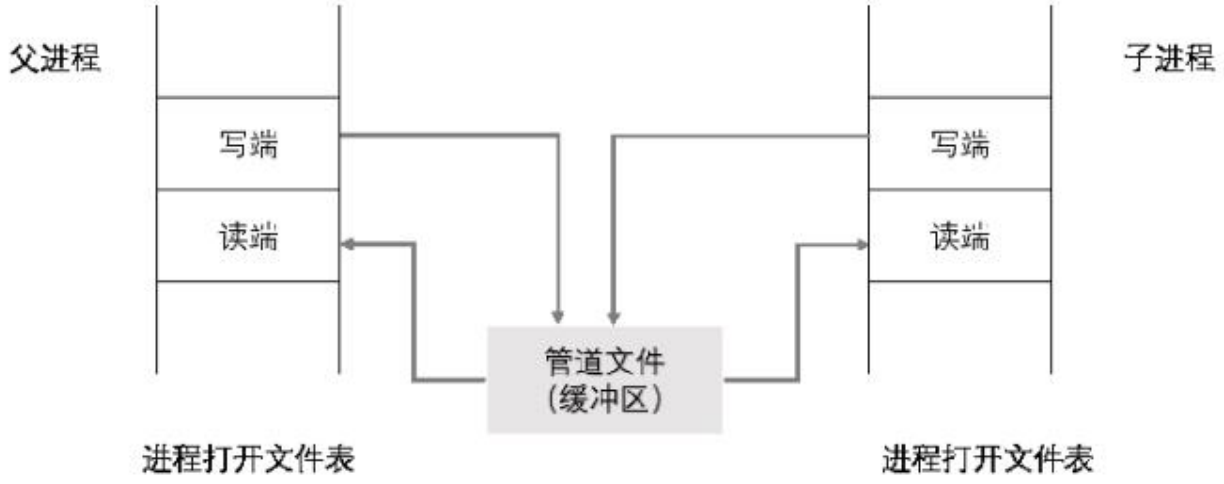


图 6.1: 父子进程与管道缓冲区

实际上，在父子进程中各自 close 掉不再使用的端口后，父子进程与管道缓冲区的关系如下图：

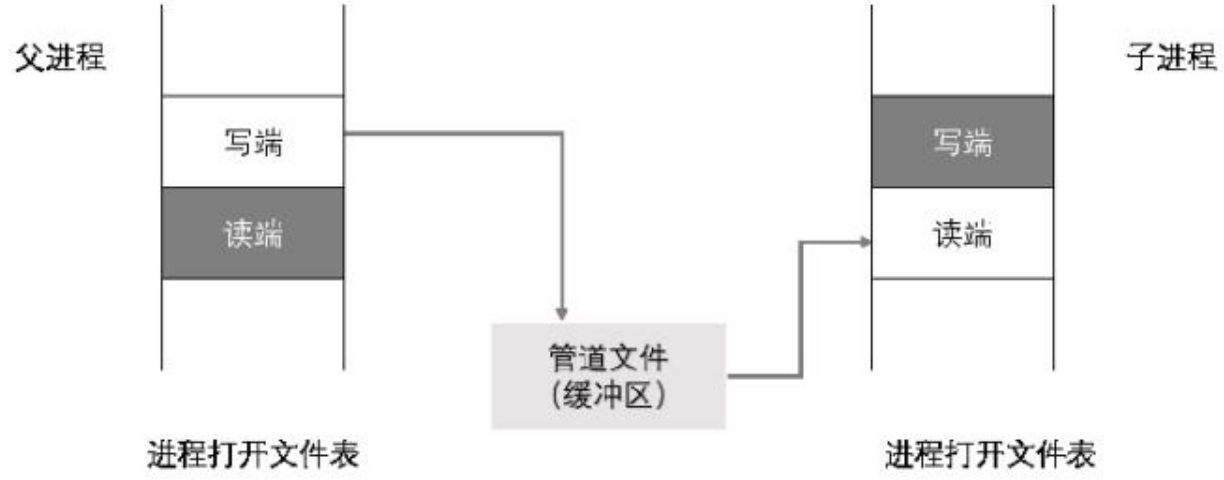


图 6.2: 关闭不使用的端口后

管道的读写

struct Pipe

在 `user/pipe.c` 定义了Pipe结构体。

```
struct Pipe {
    u_int p_rpos;        // read position
    u_int p_wpos;        // write position
    u_char p_buf[BY2PIPE];  // data buffer
};
```

在 Pipe 结构体中，p_rpos 给出了下一个将要从管道读的数据的位置，而 p_wpos 给出了下一个将要向管道写的数据的位置。只有读者可以更新 p_rpos，同样，只有写者可以更新 p_wpos，读者和写者通过这两个变量的值进行协调读写。

一个管道有 BY2PIPE(32Byte)大小的缓冲区。这个只有 BY2PIPE 大小的缓冲区发挥的作用类似于环形缓冲区，所以下一个要读或写的位置 i 实际上是 $i\%\mathrm{BY2PIPE}$。

读者写者切换

读者在从管道读取数据时，要将 p_buf[p_rpos%BY2PIPE] 的数据拷贝走，然后读指针自增 1。但是需要注意的是，管道的缓冲区此时可能还没有被写入数据。所以如果管道数据为空，即当 p_rpos >= p_wpos 时，应该进程切换到写者运行。

类似于读者，写者在向管道写入数据时，也是将数据存入 p_buf[p_wpos%BY2PIPE]，然后写指针自增 1。需要注意管道的缓冲区可能出现满溢的情况，所以写者必须得在p_wpos - p_rpos < BY2PIPE 时方可运行，否则要一直挂起。

上面这些还不足以使得读者写者一定能顺利完成管道操作。假设这样的情景：管道写端已经全部关闭，读者读到缓冲区有效数据的末尾，此时有 p_rpos = p_wpos。按照上面的做法，我们这里应当切换到写者运行。但写者进程已经结束，进程切换就造成了死循环，这时候读者进程如何知道应当退出了呢?

为了解决上面提出的问题，我们必须得知道管道的另一端是否已经关闭。不论是在读者还是在写者中，我们都需要对另一端的状态进行判断：当出现缓冲区空或满的情况时，要根据另一端是否关闭来判断是否要返回。如果另一端已经关闭，进程返回 0 即可；如果没有关闭，则切换到其他进程运行。

> **Note**
>
> 管道的关闭涉及到以下几个函数：fd.c 中的 close,fd_close 以及 pipe.c 中的 pipeclose。
>
> 如果管道的写端相关的所有的文件描述符都已经关闭，那么管道读端将 会读到文件结尾并返回 0。link : http://linux.die.net/man/7/pip

判断管道另一端是否关闭

那么我们该如何知晓管道的另一端是否已经关闭了呢? 这时就要用到我们的 `static int _pipeisclosed(struct Fd *fd, struct Pipe *p)` 函数。而这个函数的核心，就是下面我们要讲的恒成立等式了。

在之前的图6.2中我们没有明确画出文件描述符所占的页，但实际上，对于每一个匿名管道而言，我们分配了三页空间：一页是读数据的文件描述符 rfd，一页是写数据的文件描述符 wfd，剩下一页是被两个文件描述符共享的管道数据缓冲区。既然管道数据缓冲区 h 是被两个文件描述符所共享的，我们很直观地就能得到一个结论：如果有 1 个读者，1 个写者，那么管道将被引用 2 次，就如同上图所示。pageref 函数能得到页的引用次数，所以实际上有下面这个等式成立：

$$pageref(rfd) + pageref(wfd) = pageref(pipe)$$

> **Note**
>
> 内核会对 pages 数组成员维护一个页引用变量 pp_ref 来记录指向该物 理页的虚页数量。pageref 的实现实际上就是查询虚页 P 对应的实际物理页，然后 返回其 pp_ref 变量的值。

```
int pageref(void *v)
{
    u_int pte;
    if (!((* vpd)[PDX(v)] & PTE_V)) {
        return 0;
    }
    pte = (* vpt)[VPN(v)];
    if (!(pte & PTE_V)) {
        return 0;
    }
    return pages[PPN(pte)].pp_ref;
}
```

这个等式对我们而言有什么用呢？假设我们现在在运行读者进程，而进行管道写入 的进程都已经结束了，那么此时就应该有：$pageref(wfd) = 0$。

所以就有 $pageref(rfd) = pageref(pipe)$。所以我们只要判断这个等式是否成立 就可以得知写端是否关闭，对写者来说同理。

> **Note**
>
> 注意在本次实验中由于文件系统服务所在进程已经默认为 1 号进程 (起始进程为 0 号进程)，在测试时想启用文件系统需要注意 ENV_CREATE(fs_serv) 在 init.c 中的位置。

_pipeisclosed()

参数fd是管道的读端、写端之一。这个函数用于判断除了fd的另一个管道读写端是否已经关闭。

```
static int _pipeisclosed(struct Fd *fd, struct Pipe *p)
{
    // Check pageref(fd) and pageref(p),
    // returning 1 if they're the same, 0 otherwise.
    //
    // The logic here is that pageref(p) is the total
    // number of readers *and* writers, whereas pageref(fd)
    // is the number of file descriptors like fd (readers if fd is
    // a reader, writers if fd is a writer).
    //
    // If the number of file descriptors like fd is equal
    // to the total number of readers and writers, then
    // everybody left is what fd is.  So the other end of
    // the pipe is closed.
    int pfd,pfp,runs;

    do {    // 在比较的前面，需要确认 两次读取之间进程没有切换
        runs = env->env_runs;
        pfd = pageref(fd);
        pfp = pageref(p);
    } while (runs != env->env_runs); // 如果有切换，最后切换回来原进程时，env_runs就会比原来多1次

    if (pfd == pfp) {
        // writef("pfd is %d, pfp is %d\n", pfd, pfp);
        return 1;
    } else {
        return 0;
    }

    user_panic("_pipeisclosed not implemented");
}
```

pipeisclosed()

先将参数代表的文件描述符的序号 fdnum 转为文件表述符 fd，找到 fd 对应的数据区域，再调用 `_pipeisclosed(fd, p)` 函数。

```
int pipeisclosed(int fdnum)
{
    struct Fd *fd;
    struct Pipe *p;
    int r;

    if ((r = fd_lookup(fdnum, &fd)) < 0)
        return r;
    p = (struct Pipe*)fd2data(fd);
    return _pipeisclosed(fd, p);
}
```

piperead()

实现读出管道，将管道的缓冲区中长度为n的数据复制到参数 vbuf 指向的空间中。

```
static int piperead(struct Fd *fd, void *vbuf, u_int n, u_int offset)
{
    int i = 0;
    struct Pipe *p;
    char *rbuf = (char *)vbuf;

    p = (struct Pipe *)fd2data(fd);
    while (1) {
        // 如果管道为空，并且管道的写端已经关闭。如果还没有读出任何数据，要返回0
        if (_pipeisclosed(fd, p)) {
            return i;
        }
        // 读位置已经赶上了写位置，说明此时管道已经空了
        if (p->p_rpos == p->p_wpos) {
            // 如果还没有读出任何bytes
            if (i == 0) {
                syscall_yield();
            } else { // 如果已经读出（复制）了一些bytes，直接返回读出的btyes数量
                return i;
            }
        }
        while (p->p_rpos < p->p_wpos) { // 单次循环读出（复制）一个字节bytes
            if (i >= n) {
                return i;
            }
            rbuf[i] = p->p_buf[p->p_rpos % BY2PIPE];
            i++;
            p->p_rpos++;
        }
    }
    user_panic("piperead reaches end");
    return -E_INVAL;
}
```

pipewrite()

实现写入管道，将参数 vbuf 指向的空间中长度为n的数据写入到管道的缓冲区中。

```
static int pipewrite(struct Fd *fd, const void *vbuf, u_int n, u_int offset)
{
    int i;
    struct Pipe *p;
    char *wbuf = (char *)vbuf;

    p = (struct Pipe *)fd2data(fd);
    for (i = 0; i < n; i++) {
        // 这里说明管道已经满了，写位置已经超过读位置 一个管道缓冲区的长度了。
        while (p->p_wpos == p->p_rpos + BY2PIPE) {
            // 如果管道已经满了，并且管道的读端已经关闭，返回0
            if (_pipeisclosed(fd, p)) {
                writef("writing on a closed pipe\n");
                return 0;
            }
            // 如果管道已经满了，并且已经向管道中写入了一部分数据
            // 则挂起进程，等待管道空的时候再继续写入
            syscall_yield();
        }
        p->p_buf[p->p_wpos % BY2PIPE] = wbuf[i];
        p->p_wpos++;
    }
    return n;
}
```

管道的竞争

我们的 MOS 操作系统采用的是时间片轮转调度的进程调度算法。这种抢占式的进程管理就意味着，用户进程随时有可能会被打断。

当然，如果进程间是孤立的，随时打断也没有关系。但当多个进程共享同一个变量 时，执行同一段代码，不同的进程执行顺序有可能产生完全不同的结果，造成运行结果 的不确定性。而进程通信需要共享（不论是管道还是共享内存），所以我们要对进程中共 享变量的读写操作有足够高的警惕。

实际上，因为管道本身的共享性质，所以在管道中有一系列的竞争情况。在当前这种不加锁控制的情况下，我们无法保证 `_pipeisclosed` 用于管道另一端关闭的判断一定返回正确的结果。

回顾 `_pipeisclosed` 函数。在这个函数中我们对 `pageref(fd)` 与 `pageref(pipe)` 进行了等价关系的判断。假如不考虑进程竞争，不论是在读者还是写者进程中，我们会认为：

- 对 fd 和对 pipe 的 pp_ref 的**写入**是同步的。
- 对 fd 和对 pipe 的 pp_ref 的**读取**是同步的。

但现在我们处于进程竞争、执行顺序不定的情景下，上述两种情况现在都会出现不同步的现象。如果在下面这种场景下，我们前面得出的等式$pageref(rfd) + pageref(wfd) = pageref(pipe)$ 就不一定成立了。

```
pipe(p);
if(fork()==0){
    close(p[1]);
    read(p[0], buf, sizeof buf);
}else{
    close(p[0]);
    write(p[1], "hello", 5);
}
```

- fork 结束后，子进程先执行。时钟中断产生在 close(p[1]) 与 read 之间，父进程开始执行。
- 父进程在 close(p[0]) 中，p[0] 已经解除了对 pipe 的映射 (unmap)，还没有来得及解除对 p[0] 的映射，时钟中断产生，子进程接着执行。
- 注意此时各个页的引用情况：pageref(p[0]) = 2(因为父进程还没有解除对 p[0] 的 映射)，而 pageref(p[1]) = 1(因为子进程已经关闭了 p[1])。但注意，此时 pipe 的 pageref 是 2，子进程中 p[0] 引用了 pipe，同时父进程中 p[0] 刚解除对 pipe 的映射，所以在父进程中也只有 p[1] 引用了 pipe。
- 子进程执行 read，read 中首先判断写者是否关闭。比较 pageref(pipe) 与 pageref(p[0]) 之后发现它们都是 2，说明写端已经关闭，于是子进程退出。

> **Thinking Link**
>
> 上面这种不同步修改 pp_ref 而导致的进程竞争问题在 user/fd.c 中的 dup 函数中也存在。在Thinking 6.2 中会解决。

在 close 中，既然问题出现在两次 unmap 之间，那么我们为什么不能使两次 unmap 统一起来是一个原子操作呢？要注意，在我们的 MOS 操作 系统中，只有 syscall_ 开头的系统调用函数是原子操作，其他所有包括 fork 这些函数都是可能会被打断的。一次系统调用只能 unmap 一页，所以我们是不能保持两次 unmap 为一个原子操作的。那是不是一定要两次 unmap 是原子操作才能使得 _pipeisclosed一 定返回正确结果呢？

答案当然是否定的，_pipeisclosed函数返回正确结果的条件其实只是：

- 写端关闭当且仅当 $pageref(p[0]) = pageref(pipe)$；
- 读端关闭当且仅当 $pageref(p[1]) = pageref(pipe)$；

比如说第一个条件，写端关闭时，当然有 $pageref(p[0]) = pageref(pipe)$。但是由于进程切换的存在，我们无法确保 当 $pageref(p[0]) = pageref(pipe)$ 时，写端关闭。正面如果不好解决问题，我们可以考虑从其逆否命题着手，即我们要确保：当写端没有关闭的时候，$pageref(p[0]) \neq pageref(pipe)$。

我们考虑之前那个预想之外的情景，它出现的最关键原因在于：pipe 的引用次数总比 fd 要高。当管道的 close 进行到一半时，若先解除 pipe 的映射，再解除 fd 的映射，就会使得 pipe 的引用次数的-1 先于 fd。这就导致在两个 unmap 的间隙，会出现 $pageref(pipe) == pageref(fd)$ 的情况。那么若调换 fd 和 pipe 在 close 中的 unmap 顺序，能否解决这个问题呢？

> **Thinking Link**
>
> 按照上述说法控制 pipeclose 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。包括 dup 函数也存在类似的问题。
>
> 在上面的Thinking 6.4解决

根据上面的描述我们其实已经能够得出一个结论：控制 fd 与 pipe 的 map/unmap 的顺序可以解决上述情景中出现的进程竞争问题。

修改 user/pipe.c 中 pipeclose 函数中的 unmap 顺序与 user/fd.c 中 dup 函数中的 map 顺序以避免上述情景中的进程竞争情况。

dup()

dup函数将一个 oldfdnum 代表的文件描述符对应的文件内容区域（ `fd2data(oldfd)` ） 映射到 newfdnum 代表的文件描述符，再把前者文件描述符对应的页面（ `oldfd` 被分配的页面），映射到后者文件描述符。

注意，这里的映射都是指共享。

```
int dup(int oldfdnum, int newfdnum)
{
    int i, r;
    u_int ova, nva, pte;
    struct Fd *oldfd, *newfd;

    if ((r = fd_lookup(oldfdnum, &oldfd)) < 0) {
        return r;
    }
    close(newfdnum);
    newfd = (struct Fd *)INDEX2FD(newfdnum);
    ova = fd2data(oldfd);
    nva = fd2data(newfd);

    if ((* vpd)[PDX(ova)]) {
        for (i = 0; i < PDMAP; i += BY2PG) {
            pte = (* vpt)[VPN(ova + i)];
            if (pte & PTE_V) {
                // should be no error here -- pd is already allocated
                if ((r = syscall_mem_map(0, ova + i, 0, nva + i,
                                         pte & (PTE_V | PTE_R | PTE_LIBRARY))) <
0) {
                    goto err;
                }
            }
        }
    }

    if ((r = syscall_mem_map(0, (u_int)oldfd, 0, (u_int)newfd,
                             ((*vpt)[VPN(oldfd)]) & (PTE_V | PTE_R |
PTE_LIBRARY))) < 0)
        goto err;

    return newfdnum;

err:
    syscall_mem_unmap(0, (u_int)newfd);
    for (i = 0; i < PDMAP; i += BY2PG) {
        syscall_mem_unmap(0, nva + i);
    }
    return r;
}
```

管道的同步

我们通过控制修改 pp_ref 的前后顺序避免了“写数据”导致的错觉，但是我们还得 解决第二个问题：读取 pp_ref 的同步问题。

同样是上面的代码，我们思考下面的情景：

- fork 结束后，子进程先执行。执行完 close(p[1]) 后，执行 read，要从 p[0] 读取数据。但由于此时管道数据缓冲区为空，所以 read 函数要判断父进程中的写端是否关闭，进入到 _pipeisclosed 函数，pageref(fd) 值为 2(父进程和子进程都打开了 p[0])，此时，时钟中断产生。
- 内核切换到父进程执行，父进程 close(p[0])，之后向管道缓冲区写数据。要写的数据较多，写到一半时钟中断产生，内核切换到子进程运行。
- 子进程继续运行，获取到 pageref(pipe) 值为 2(父进程打开了 p[1]，子进程打开了 p[0])，引用值相等，于是认为父进程的写端已经关闭，子进程退出。

上述现象的根源是什么？fd 是一个父子进程共享的变量，但子进程中的 pageref(fd) 没有随父进程对 fd 的修改而同步，这就造成了子进程读到的 pageref(fd) 成为了“脏数据”。

> **MyNote**：用数据库理论的话来说，就是不可重复读。

为了保证读的同步性，子进程应当重新读取 pageref(fd) 和 pageref(pipe)，并且要 **在确认两次读取之间进程没有切换后**，才能返回正确的结果。为了实现这一点，我们要使用到之前一直都没用到的变量：env_runs。

env_runs 记录了一个进程 env_run 的次数，这样我们就可以根据某个操作 do() 前 后进程 env_runs 值是否相等，来判断在 do() 中进程是否发生了切换。

根据上面的表述， `_pipeisclosed()` 函数中必须有相应的代码，使得它满足 “同步读” 的要求。注意 env_runs 变量是需要维护的。

这也就是为什么在 `_pipeisclosed()` 函数中，会有如下的代码段

```
do {    // 在比较的前面，需要确认 两次读取之间进程没有切换
    runs = env->env_runs;
    pfd = pageref(fd);
    pfp = pageref(p);
} while (runs != env->env_runs); // 如果有切换，最后切换回来原进程时，env_runs就会比原来多1次

if (pfd == pfp) {
    return 1;
} else {
    return 0;
}
```

$Part2.2$ shell

什么是 shell？在计算机科学中，Shell 俗称壳（用来区别于核），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。基本上 shell 分两大类：

一是图形界面 shell（Graphical User Interface shell 即 GUI shell）。例如：应用最为广泛的 Windows Explorer （微软的 windows 系列操作系统），还有也包括广为人知 的 Linux shell，其中 linux shell 包括 X window manager (BlackBox 和 FluxBox），以 及功能更强大的 CDE、GNOME、KDE、XFCE。

二是命令行式 shell（Command Line Interface shell ，即 CLI shell），也就是我们 MOS 操作系统最后即将实现的 shell 模式。

常见的 shell 命令在 lab0 已经介绍过了，这里就不赘述，接下来让我们一步一步揭开 shell 背后的神秘面纱。

Spawn 函数

spawn 的意思是 “产卵”，其作用是帮助我们调用文件系统中的可执行文件并执行。

spawn 的流程可以分解如下：

- 从文件系统打开对应的文件 (2 进制 ELF，在我们的 OS 里是 *.b 文件)。
- 申请新的进程描述符；
- 将目标程序加载到子进程的地址空间中，并为它们分配物理页面；
- 为子进程初始化堆、栈空间，并设置栈顶指针, 以及重定向、管道的文件描述符，对于栈空间，因为我们的调用可能是有参数的，所以要将参数也安排进用户栈中。

- 设置子进程的寄存器 (栈寄存器 sp 设置为 esp。程序入口地址 pc 设置为 UTEXT)
- 将父进程的共享页面映射到子进程的地址空间中。
- 这些都做完后，设置子进程可执行。

在动手填写 spawn 函数前，需要：

1. 认真回看 lab5 文件系统相关代码，弄清打开文件的过程。
2. 思考如何读取 elf 文件或者说如何知道二进制文件中 text 段。

关于后一个问题，在 lab3 中填写了 load_icode 函数，实现了 elf 中读取数据并写入内存空间。而 text 段的位置，可以借助 readelf 的帮助。

我们需要思考下面几个问题：

- 在 lab1 中我们介绍了 data text bss 段及它们的含义，data 存放初始化过的全局变量，bss 存放未初始化的全局变量。关于 memsize 和 filesize ，这里摘录一段 lab1 里面指导书的Note。注意其中关于 “bss 并不在文件中占数据” 的表述。

  > MemSiz 永远大于等于 FileSiz。若 MemSiz 大于 FileSiz，则操作系统 在加载程序的时候，会首先将文件中记录的数据加载到对应的 VirtAddr 处。之后， 向内存中填 0, 直到该段在内存中的大小达到 MemSiz 为止。那么为什么 MemSiz 有时候会大于 FileSiz 呢？这里举这样一个例子：C 语言中未初始化的全局变量，我们需要为其分配内存，但它又不需要被初始化成特定数据。因此，在可执行文件中也只记录它需要占用内存 (MemSiz)，但在文件中却没有相应的数据（因为它并不需要初始化成特定数据）。故而在这种情况下，MemSiz 会大于 FileSiz。这也解释了， 为什么 C 语言中全局变量会有默认值 0。这是因为操作系统在加载时将所有未初始 化的全局变量所占的内存统一填了 0。

- 在 lab3 中我们创建进程，并且在内核态加载了初始进程（ENV_CREATE(...)), 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。

在 lab3 中我们要填写 load_icode_mapper 函数，分为两部分填写，第二部分则是 处理 msize 和 fsize 不相等时的情况。

总的来说就是：**bss 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。因此，我们需要回顾一下 lab3 设计的函数 `load_icode_mapper()` 函数是如何实现这点的。**

关于如何为子进程初始化栈空间，则需要仔细阅读 `init_stack()` 函数。

因为我们无法直接操作子进程的栈空间，所以该函数首先将需要准备的参数填充到 本进程的 TMPPAGE 这个页面处，然后将 TMPPAGE 映射到子进程的栈空间中。

首先将 argc 个字符串填到栈上，并且不要忘记在每个字符串的末尾要加上 '\0' 表示结束，然后将 argc+1 个指针填到栈上，第 argc+1 个指针指的是一个空字符串表示参数的结束。 最后将 argc 和 argv 填到栈上，argv 将指向那 argc+1 个字符指针。

这里给出一张 spawn 准备的栈空间的示意图。

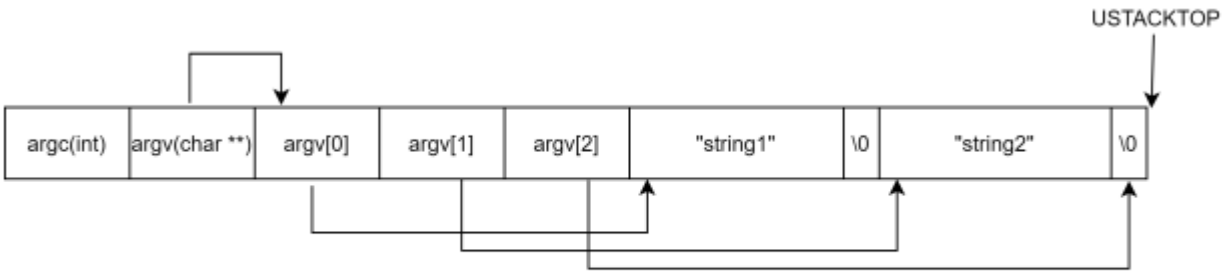


图 6.3: 子进程栈空间示意图

init_stack()

这个函数用于初始化子进程的用户栈。

注意，上面图中反应的内容最多只占据了一页大小。用 TMPPAGETOP 是为了倒着来依次编排，实际上在父进程中，内容是在 $[\mathrm{BY2PG}, \mathrm{TMPPAGETOP} = 2 * \mathrm{BY2PG})$ 之间的。

首先先把每一个参数字符串复制到 $[\mathrm{TMPPAGETOP} - \mathrm{tot}, \mathrm{TMPPAGETOP})$ 之间。其中 tot 是 每一个字符串长度+1后 的和；

然后在 $[\mathrm{TMPPAGETOP} - \mathrm{ROUND}(\mathrm{tot}, 4) - 4 * (\mathrm{argc} + 1), \mathrm{TMPPAGETOP} - \mathrm{tot})$ 之间，计算出 **在子进程的对应页面中** 每一个参数字符串的首地址后，将这些值赋给指针数组的成员。

> 值得注意的是：在初始化参数字符串的指针数组的时候， `args[0..argc-1]` 是被初始化为 **在子进程中，参数字符串应该在的地址。具体来说，是在子进程的进程空间里，地址 `USTACKTOP-BY2PG` 处对应的页面上，各个字符串的地址。**

最后，将指针数组再压栈两个 u_int 大小的空间处，分别存放参数字符串数组的首地址、参数个数。

到这里，父进程中对应的信息就已经填写好了，然后映射到子进程对应的空间里（具体来说是 $[\mathrm{USTACKTOP} - \mathrm{BY2PG}, \mathrm{USTACKTOP})$ 处）。

```
#define TMPPAGE     (BY2PG)
#define TMPPAGETOP  (TMPPAGE+BY2PG)
int init_stack(u_int child, char **argv, u_int *init_esp)
{
    int argc, i, r, tot;
    char *strings;
    u_int *args;

    /* 计算参数的个数 (argc) 和 用于存放参数字符串的总空间长度 (tot) */
    tot = 0;
    for (argc=0; argv[argc]; argc++)
        tot += strlen(argv[argc])+1;

    /* 确保用于存放参数字符串（在进行字对齐以后的大小）、arg+1个参数字符串指针argv[0]、指向这些指针的二级指针argv(char **)、参数个数argc(int)，这些是这个用户栈页面放得下的 */
    if (ROUND(tot, 4)+4*(argc+3) > BY2PG)
        return -E_NO_MEM;

    /* 计算出 存放参数字符串的的起始地址 */
    strings = (char*)TMPPAGETOP - tot;
    /* 计算出 argc+1个参数字符串指针（其实就是指针数组）的基地址 */
    args = (u_int*)(TMPPAGETOP - ROUND(tot, 4) - 4*(argc+1));
    /* 在当前进程的进程空间内的 TMPPAGE 地址处，申请一页实际的物理内存 */
    if ((r = syscall_mem_alloc(0, TMPPAGE, PTE_V|PTE_R)) < 0)
        return r;

    /* 将各个参数字符串 复制进入 用户栈所在页的 strings 地址处 */
    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++;
    }

    // 下面是 将 args[0..argc-1] 初始化为 **在子进程中，参数字符串应该在的地址**
```

```
    // 计算出 **在子进程的（USTACKTOP-BY2PG处）对应的页面中** 每一个参数字符串的首地址后，
将这些值赋给指针数组的成员。
    ctemp = (char *)(USTACKTOP - TMPPAGETOP + (u_int)strings);
    for(i = 0;i < argc;i++)
    {   // 将每个参数字符串的地址赋给指针数组
        args[i] = (u_int)ctemp;
        ctemp += strlen(argv[i])+1;
    }
    /*  将最后一个指针数组成员 args[argc] 赋值为0，用空指针代表终结。*/
    ctemp--;
    args[argc] = ctemp;
    /* 令 pargv_ptr 指向的位置是args 再往下一个 u_int 大小的位置（指针加减法）
        然后令 *pargv_ptr 指向 参数字符串指针数组的基地址 */
    u_int *pargv_ptr;
    pargv_ptr = args - 1;
    *pargv_ptr = USTACKTOP - TMPPAGETOP + (u_int)args;
    /* 令 pargv_ptr 指向的位置是args 再往下两个 u_int 大小的位置（指针加减法）
        然后令 *pargv_ptr 记录 参数个数argc */
    pargv_ptr--;
    *pargv_ptr = argc;

    //  将 *init_esp 设置为 子进程的初始化栈指针
    *init_esp = USTACKTOP - TMPPAGETOP + (u_int)pargv_ptr;

    if ((r = syscall_mem_map(0, TMPPAGE, child, USTACKTOP-BY2PG, PTE_V|PTE_R)) <
0)
        goto error;
    if ((r = syscall_mem_unmap(0, TMPPAGE)) < 0)
        goto error;

    return 0;
error:
    syscall_mem_unmap(0, TMPPAGE);
    return r;
}
```

解释 shell 命令

接下来，我们将通过一个实例，再次吸收理解上面 lab1、lab3 联动的内容。

```
#include "lib.h"
#define ARRAYSIZE (1024*10)
int bigarray[ARRAYSIZE]={0};

void umain(int argc, char **argv)
{
    int i;
    writef("Making sure bss works right...\n");
    for(i = 0;i < ARRAYSIZE; i++)
        if(bigarray[i]!=0)
            user_panic("bigarray[%d] isn't cleared!\n",i);

    for (i = 0; i < ARRAYSIZE; i++)
        bigarray[i] = i;

    for (i = 0; i < ARRAYSIZE; i++)
        if (bigarray[i] != i)
            user_panic("bigarray[%d] didn't hold its value!\n", i);

    writef("Yes, good. Now doing a wild write off the end...\n");
    bigarray[ARRAYSIZE+1024] = 0;
```

```
    user_panic("SHOULD HAVE TRAPPED!!!");
}
```

在 user/文件夹下创建上面的文件，并在 Makefile 中添加相应信息，使得生成相应 的.b 文件，在 init/init.c 中创建相应的初始进程，观察相应的实验现象。（具体可以参考 lab4 中 pingpong 和 fktest 是如何添加到 makefile 中的）如果能正确运行，则说明我们的 load_icode 系列函数正确地保证了 bss 段的初始化。

使用 readelf 命令解析我们的 testbss.b 文件，看看 bss 段的大小并分析其原因。学习并使用 size 命令，看看我们的 testbss.b 文件的大小。

修改代码，将数组初始化为 array[SIZE]=0；重新编译（记得常 make clean），再次分析 bss 段。再次使用 size 命令。

再次修改代码，将数组初始化为 array[SIZE]=1；再次重新编译，再次分析。

在 Lab5 中我们实现了文件系统，lab6 的 shell 部分我们提供了几个可执行二进制 文件，模拟 linux 的命令：ls.b cat.b ……。上面提到的 spawn 函数实现方法，是打开相应的文件并执行。请你思考我们是如何将文件 “烧录” 到 fs.img 中的（阅读 fs/Makefile）。

如果你并没有看懂这段话，请回看 Exercise 5.4

你可以尝试将生成的 testbss.b 加载进 fs.img 中.

这节 “补课” 下课以后，大家再去补全 spawn 函数，相信能如鱼得水了。

usr_load_elf()

这个函数就是用户态的 `load_icode_mapper()` 函数。

整体上的逻辑是：

1. 如果 `offset = va - ROUNDDOWN(va, BY2PG), offser!=0` ，说明开头存在没有页对齐的部分；
2. 那么，为子进程的进程空间里的地址 va 处，申请一页实际的物理内存；
3. 将子进程刚刚为 va 处申请的物理内存共享给 父进程的 $[\mathrm{USTACKTOP}, \mathrm{USTACKTOP} + \mathrm{BY2PG})$ 这一页空间。
4. 如果 `offset = va - ROUNDDOWN(va, BY2PG), offser!=0` ，说明开头存在没有页对齐的部分，那么就拷贝到父进程的 $[\mathrm{USTACKTOP} + \mathrm{offset}, \mathrm{USTACKTOP} + \mathrm{BY2PG})$ ；
5. 解除父进程的 $[\mathrm{USTACKTOP}, \mathrm{USTACKTOP} + \mathrm{BY2PG})$ 到原来那页共享的物理页的映射。此时，父子进程共享的物理页上就存放了相应的二进制文件的内容。
6. 在页对齐以后，以页为单位进行复制。索引是i。
7. 那么，为子进程的进程空间里的地址 va+i 处，申请一页实际的物理内存；
8. 将子进程刚刚为 va+i 处申请的物理内存共享给 父进程的 $[\mathrm{USTACKTOP}, \mathrm{USTACKTOP} + \mathrm{BY2PG})$ 这一页空间。
9. 将二进制文件的相应部分拷贝到父进程的 $[\mathrm{USTACKTOP}, \mathrm{USTACKTOP} + \mathrm{BY2PG})$ 。此时，父子进程共享的物理页上就存放了相应的二进制文件的内容。
10. 解除父进程的 $[\mathrm{USTACKTOP}, \mathrm{USTACKTOP} + \mathrm{BY2PG})$ 到原来那页共享的物理页的映射。此时，父子进程共享的物理页上就存放了相应的二进制文件的内容。

总的来说就是：用父进程的 $[\mathrm{USTACKTOP}, \mathrm{USTACKTOP} + \mathrm{BY2PG})$ 这页空间去共享子进程申请的物理页，用在父进程的进程空间里写入数据的方式，在子进程对应的物理页中写入elf文件。

这是因为：**不能直接操作子进程的用户空间栈。**

```
int usr_load_elf(int fd, Elf32_Phdr *ph, int child_envid)
{
    u_long va = ph->p_vaddr;
    u_int32_t sgsize = ph->p_memsz;
```

```
    u_int32_t bin_size = ph->p_filesz;
    u_char *bin;
    u_long i;
    int r;
    u_long offset = va - ROUNDDOWN(va, BY2PG);
    r = read_map(fd, ph->p_offset, &bin);
    if (r < 0)
        return r;
    if (offset != 0)
    {
        if ((r = syscall_mem_alloc(child_envid, va, PTE_V | PTE_R)) < 0)
            return r;
        if ((r = syscall_mem_map(child_envid, va, 0, USTACKTOP, PTE_V | PTE_R))
< 0)
            return r;
        user_bcopy(bin, USTACKTOP + offset, MIN(BY2PG - offset, bin_size));
        if ((r = syscall_mem_unmap(0, USTACKTOP)) < 0)
            return r;
    }
    for (i = offset ? MIN(BY2PG - offset, bin_size) : 0; i < bin_size; i +=
BY2PG)
    {
        if ((r = syscall_mem_alloc(child_envid, va + i, PTE_V | PTE_R)) < 0)
            return r;
        if ((r = syscall_mem_map(child_envid, va + i, 0, USTACKTOP, PTE_V |
PTE_R)) < 0)
            return r;
        user_bcopy(bin + i, USTACKTOP, MIN(BY2PG, bin_size - i));
        if ((r = syscall_mem_unmap(0, USTACKTOP)) < 0)
            return r;
    }
    while (i < sgsize)
    {
        if ((r = syscall_mem_alloc(child_envid, va + i, PTE_V | PTE_R)) < 0)
            return r;
        i += BY2PG;
    }
    return 0;
}
```

spawn()

具体见注解。

```
int spawn(char *prog, char **argv)
{
    u_char elfbuf[512];
    int r;
    int fd;
    u_int child_envid;
    int size, text_start;
    u_int i, *blk;
    u_int esp;
    Elf32_Ehdr* elf;
    Elf32_Phdr* ph;
    // Note 0: some variable may be not used,you can cancel them as you like
    // Step 1: Open the file specified by `prog` (prog is the path of the
program)
    if((r=open(prog, O_RDONLY))<0){
        user_panic("spawn ::open line 102 RDONLY wrong !\n");
        return r;
    }
    // Your code begins here
```

```
    fd = r;
    r = readn(fd, elfbuf, sizeof(Elf32_Ehdr));
    if (r < 0)
        user_panic("read Ehdr failed!");

    elf = (Elf32_Ehdr*)elfbuf;

    //can't use ehdr after
    // Before Step 2 , You had better check the "target" spawned is a execute
bin
    /* 这一步就是检查拿到的是不是一个 可执行文件 elf */
    if ((!usr_is_elf_format((u_char*)elf)) || elf->e_type != 2) // #define
ET_EXEC 2
        user_panic("Not elf or exec!");

    // Step 2: Allocate an env (Hint: using syscall_env_alloc())
    r = syscall_env_alloc();
    if (r < 0)
        user_panic("Alloc env failed!");
    if (r == 0) {
        env = envs + ENVX(syscall_getenvid());
        return 0;
    }
    child_envid = r;
    // Step 3: Using init_stack(...) to initialize the stack of the allocated
env
    init_stack(child_envid, argv, &esp);
    // Step 3: Map file's content to new env's text segment
    //        Hint 1: what is the offset of the text segment in file? try to use
objdump to find out.
    //        Hint 2: using read_map(...)
    //        Hint 3: Important!!! sometimes ,its not safe to use read_map
,guess why
    //                If you understand, you can achieve the "load APP" with any
method
    // Note1: Step 1 and 2 need sanity check. In other words, you should check
whether
    //       the file is opened successfully, and env is allocated successfully.
    // Note2: You can achieve this func in any way , remember to ensure the
correctness
    //        Maybe you can review lab3
    /* 这一步总体上看，就是将打开的二进制文件的文本读入 子进程的进程空间（为之申请的物理内存
中） */
    text_start = elf->e_phoff;
    size = elf->e_phentsize;

    for (i = 0; i < elf->e_phnum; i++)
    {
        if ((r = seek(fd, text_start)) < 0)
            user_panic("seek failed!");
        if ((r = readn(fd, elfbuf, size)) < 0)
            user_panic("readn failed!");

        ph = (Elf32_Phdr *)elfbuf;
        if (ph->p_type == PT_LOAD)
        {
            r = usr_load_elf(fd, ph, child_envid);
            if (r < 0)
                user_panic("load faild %d!", r);
        }
        text_start += size;
    }
    // Your code ends here
    /* 设置子进程的寄存器（栈寄存器 sp 设置为 esp。程序入口地址 pc 设置为 UTEXT）
        esp 在执行init_stack()初始化子进程的用户栈空间的时候，就设置为了用户栈入口 */
    struct Trapframe *tf;
```

```
    writef("\n::::::::::spawn size : %x  sp : %x::::::::\n",size,esp);
    tf = &(envs[ENVX(child_envid)].env_tf);
    tf->pc = UTEXT;
    tf->regs[29]=esp;

    // Share memory
    u_int pdeno = 0;
    u_int pteno = 0;
    u_int pn = 0;
    u_int va = 0;
    for(pdeno = 0;pdeno<PDX(UTOP);pdeno++)
    {
        if(!((* vpd)[pdeno]&PTE_V))
            continue;
        for(pteno = 0;pteno<=PTX(~0);pteno++)
        {
            pn = (pdeno<<10)+pteno;
            if(((* vpt)[pn]&PTE_V)&&((* vpt)[pn]&PTE_LIBRARY))
            {
                va = pn*BY2PG;

                if((r = syscall_mem_map(0,va,child_envid,va,
(PTE_V|PTE_R|PTE_LIBRARY)))<0)
                {

                    writef("va: %x   child_envid: %x   \n",va,child_envid);
                    user_panic("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@");
                    return r;
                }
            }
        }
    }

    if((r = syscall_set_env_status(child_envid, ENV_RUNNABLE)) < 0)
    {
        writef("set child runnable is wrong\n");
        return r;
    }
    return child_envid;
}
```

shell 进程

通过观察 user/sh.c 里面的进程主函数，可以发现：**实际上就是 主shell进程 通过对每一条输入的shell指令，都创建一个 子shell进程 去执行该指令 的方式响应命令行。每一个 子shell 进程运行指令的核心就是 `runcmd()` 函数。**

接下来，我们需要在 shell 进程里实现对管道和重定向的解释功能。解释 shell 命令时：

1. 如果碰到重定向符号‘<’或者‘>’, 则读下一个单词，打开这个单词所代表的文 件，然后将其复制给标准输入或者标准输出。
2. 如果碰到管道符号‘|’, 则首先需要建立管道 pipe，然后 fork。
    - 对于父进程，需要将管道的写者复制给标准输出，然后关闭父进程的读者和 写者，运行‘|’左边的命令，获得输出，然后等待子进程运行。
    - 对于子进程，将管道的读者复制给标准输入，从管道中读取数据，然后关闭 子进程的读者和写者，继续读下一个单词。

在这里可以举一个使用管道符号的例子来方便大家理解，相信大家都使用过 linux 中的 ps 指令，也就是最基本的查看进程的命令，而直接使用 ps 会看到所有的进程，为了更方便的追踪某个进程，我们通常使用 ps aux|grep xxx 这条指令，这就是使用管道 的例子，ps aux 命令会将所有的进程按格式输出，而 grep xxx 命令作为子进程执行，所 有的进程作为他的输入，最后的输出将会筛选出含有 xxx 字符串的进程展示在屏幕上。

runcmd()

完成执行 shell 指令的核心函数。

```
#define MAXARGS 16
void runcmd(char *s)
{
    char *argv[MAXARGS], *t;
    int argc, c, i, r, p[2], fd, rightpipe;
    int fdnum;
    rightpipe = 0;
    gettoken(s, 0);
again:
    argc = 0;
    for(;;){
        c = gettoken(0, &t);
        switch(c){
        case 0:
            goto runit;
        case 'w':
            if(argc == MAXARGS){
                writef("too many arguments\n");
                exit();
            }
            argv[argc++] = t;
            break;
        case '<':
            if(gettoken(0, &t) != 'w'){
                writef("syntax error: < not followed by word\n");
                exit();
            }
            // Your code here -- open t for reading,
            // dup it onto fd 0, and then close the fd you got.
            r = open(t, O_RDONLY);
            if (r < 0)
                user_panic("< open file failed!");
            fd = r;
            dup(fd, 0);
            close(fd);
            break;
        case '>':
            // Your code here -- open t for writing,
            // dup it onto fd 1, and then close the fd you got.
            if(gettoken(0, &t) != 'w'){
                writef("syntax error: < not followed by word\n");
                exit();
            }
            r = open(t, O_WRONLY);
            if (r < 0)
                user_panic("> open file failed!");
            fd = r;
            dup(fd, 1);
            close(fd);
            break;
        case '|':
            // Your code here.
```

```
            //  First, allocate a pipe.
            //  Then fork.
            //  the child runs the right side of the pipe:
            //      dup the read end of the pipe onto 0
            //      close the read end of the pipe
            //      close the write end of the pipe
            //      goto again, to parse the rest of the command line
            //  the parent runs the left side of the pipe:
            //      dup the write end of the pipe onto 1
            //      close the write end of the pipe
            //      close the read end of the pipe
            //      set "rightpipe" to the child envid
            //      goto runit, to execute this piece of the pipeline
            //          and then wait for the right side to finish
            pipe(p);
            if ((rightpipe = fork()) == 0) {
                dup(p[0], 0);
                close(p[0]);
                close(p[1]);
                goto again;
            } else {
                dup(p[1], 1);
                close(p[1]);
                close(p[0]);
                goto runit;
            }
            break;
        }
    }

runit:
    if(argc == 0) {
        if (debug_) writef("EMPTY COMMAND\n");
        return;
    }
    argv[argc] = 0;
    if (1) {
        writef("[%08x] SPAWN:", env->env_id);
        for (i=0; argv[i]; i++)
            writef(" %s", argv[i]);
        writef("\n");
    }

    if ((r = spawn(argv[0], argv)) < 0)
        writef("spawn %s: %e\n", argv[0], r);
    close_all();
    if (r >= 0) {
        if (debug_) writef("[%08x] WAIT %s %08x\n", env->env_id, argv[0], r);
        wait(r);
    }
    if (rightpipe) {
        if (debug_) writef("[%08x] WAIT right-pipe %08x\n", env->env_id,
rightpipe);
        wait(rightpipe);
    }
    exit();
}
```

通过阅读 user/sh.c 中的 void runcmd(char *s)。代码空白段的注释我们知道，将文件复制给标准输入或输出，需要我们将其 dup 到 0 或 1 号文件描述符 (fd). 那么问题来了：

> **Thinking Link**

> 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。
>
> Thinking 6.8

在 spawn 函数中标记为 Share memory 一段的作用：这个函数将父进程所有的共享页面映射给了子进程。

想一下，进程空间中的哪些内存是共享内存?

在进程空间中，文件、管道、控制台以及文件描述符都是以共享页面的方式存在的。 有几处通过 spawn 产生新进程的位置。

- init.b 进程通过 spawn 生成 shell 进程。init.b 申请了 console（控制台）作为标准 输入输出，而这个 console 就是通过共享页面映射给 shell 进程，使得 shell 进程可以通过控制台与用户交互。
- 子 shell 进程负责解析命令行命令，并通过 spawn 生成可执行程序进程（对应 *.b 文件）。在解析命令行的命令时，子 shell 会将重定向的文件及管道等 dup 到子 shell 的标准输入或输出，然后 spawn 时将标准输入和输出通过共享内存映射给可执行程序，所以可执行程序可以从控制台、文件和管道等位置输入和输出数据。

> **Note**
>
> 我们的测试进程从 user/icode 开始执行，里面调用了 spawn(init.b), 在 完成了 spawn 后，创建了 init.b 进程。init.b 进程调用 spawn(sh.b)，创建了 sh.b 进程，也就是我们的 shell。

$Part3$ 体会与感想&指导书反馈

Lab6 作为整个实验的收尾，总体难度上是要比 Lab5 简单很多的。

主要还是和Lab5相关的部分不是很熟悉，由于期末时间很紧张+ Lab5 代码量十分庞大，所以 Lab5 的第二部分我确实没有完全搞清楚，只能说在假期里继续学习了。如果没有 Lab5-2-exam 帮助我理清楚关系，我可能还处于一片混沌之中。

所幸的是， Lab6 整体上难度不大，在阅读了指导书以后，基本上可以完成。

比较难的地方应该是 spawn() 函数以及辅助它实现功能的 `usr_load_elf()` 函数的书写。后者可以依照 `load_icode_mapper()` 函数来实现，但是前者真的碰了很多壁。最后是在前辈们的代码帮助下，勉强理解了需要我们进行填充的部分的代码逻辑，算是通过了。但是其他部分的代码还是有很多地方没有细品，这些只能是等到假期的时候再琢磨了。