Lab6

Thinking

Thinking 6.1

> **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？ ■

```
switch (fork()) {
    case -1:
        break;
    case 0:
        close(fides[0]);
        write(fides[1], "...", 12);
        close(fides[1]);
        exit(EXIT_SUCCESS);

    default:
        close(fides[1]);
        read(fides[0], buf, 100);
        printf("...", buf);
        close(fides[0]);
        exit(EXIT_SUCCESS);
}
```

Thinking 6.2

> **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？ ■

多进程环境下，由于 `dup()` 系统调用的非原子性操作（先复制文件描述符 `fd`，再复制底层数据 `data`），可能导致管道（pipe）的引用计数（`pageref`）出现不一致，进而引发 `read()` 函数的误判。

子进程 `dup(pipe[1])` 后 `read(pipe[0])`，父进程 `dup(pipe[0])` 后 `write(pipe[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`

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`

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。

Thinking 6.3

> **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。 ■

系统调用（syscall）通常被认为是“原子操作”，因为它们在**内核态执行期间不会被其他用户进程抢占**（即内核是非抢占式的）。然而，**原子性 ≠ 不可中断**，某些系统调用的执行可能涉及多步操作，或者依赖共享数据结构，从而在**内核内部**出现竞争条件。

例如`dup()`最终调用`do_dup2()`，其中会操作`file`结构：

```
static int do_dup2(struct files_struct *files, unsigned fd, unsigned newfd) {
    struct file *file = fcheck(fd);  // 获取原 fd 的 file 结构
    if (!file)
        return -EBADF;
    get_file(file);  // 增加 file->f_count（关键操作）
    return allocate_fd(newfd, file);  // 分配新 fd
}
```

- `get_file(file)`是原子操作（通过`atomic_inc(&file->f_count)`），但如果在`get_file()`之前发生中断，仍可能导致问题。

Thinking 6.4

> **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。
>
> ■

- 可以解决上述问题。
  - 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。

Thinking 6.5

> **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

Thinking 6.6

> **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。 ■

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

```
    if ((r = open(t, O_RDONLY)) < 0) {
        user_panic("redirction_1: open file in shell failed!");
    }
    fd = r;
    dup(fd, 0);
    close(fd);
```

Thinking 6.7

> **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？ ■

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。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

Thinking 6.8

> **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？
>
> ■

- 可以观察到2次 `spawn`：3805 和 4006 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。

难点分析

函数分析：

管道

pipe

```
int pipe(int pfd[2]) {
    int r; // 用于存储系统调用的返回值
    void *va; // 虚拟地址，用于映射共享内存
    struct Fd *fd0, *fd1; // 两个文件描述符结构体（读端和写端）

    if ((r = fd_alloc(&fd0)) < 0 || (r = syscall_mem_alloc(0, fd0, PTE_D | PTE_LIBRARY)) <
0) {
        goto err; // 分配一个文件描述符结构体 fd0（读端）,为 fd0 分配物理内存页，并设置权限
    }

    if ((r = fd_alloc(&fd1)) < 0 || (r = syscall_mem_alloc(0, fd1, PTE_D | PTE_LIBRARY)) <
0) {
        goto err1;
    }

    /* Step 2: Allocate and map the page for the 'Pipe' structure. */
    va = fd2data(fd0);  // 获取 fd0 对应的虚拟地址
    if ((r = syscall_mem_alloc(0, (void *)va, PTE_D | PTE_LIBRARY)) < 0) {
        goto err2; // 获取 fd0 对应的用户态虚拟地址 va,为 va 分配物理内存页（用于存储管道数据）。
    }
    if ((r = syscall_mem_map(0, (void *)va, 0, (void *)fd2data(fd1), PTE_D | PTE_LIBRARY)) <
0) { // 将 fd0 的虚拟地址 va 映射到 fd1 的虚拟地址（fd2data(fd1)），使两者共享同一块物理内存。
        goto err3;
    }

    /* Step 3: Set up 'Fd' structures. */
    fd0->fd_dev_id = devpipe.dev_id; // 设置读端的设备 ID
    fd0->fd_omode = O_RDONLY; // 设置为只读模式

    fd1->fd_dev_id = devpipe.dev_id; // 设置写端的设备 ID
    fd1->fd_omode = O_WRONLY; // 设置为只写模式

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

    /* Step 4: Save the result. */
    pfd[0] = fd2num(fd0); // 将读端 fd0 转换为文件描述符编号
    pfd[1] = fd2num(fd1); // 将写端 fd1 转换为文件描述符编号
    return 0; // 成功返回 0

err3:
    syscall_mem_unmap(0, (void *)va);
err2:
    syscall_mem_unmap(0, fd1);
err1:
    syscall_mem_unmap(0, fd0);
err:
    return r;
}
```

struct Pipe

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

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

```
/* 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_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;
}
```

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

spawn

```
/* 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;
}
```

实验体会

- 通过本次实验，我深刻体会到Unix系统"组合小工具"的设计哲学——看似简单的管道机制，通过将多个小程序以管道连接的方式组合使用，却能实现强大的功能。在实现Shell的管道功能过程中，我认识到精细控制文件描述符的重要性，特别是理解fork()创建子进程与exec()加载新程序之间的协作机制是关键所在。同时，实验中遇到的竞争条件问题也给我上了重要一课：在并发编程环境下，任何对共享资源（如管道）的访问都需要格外谨慎，必须通过合理的同步机制来保证数据一致性。这些经验不仅加深了我对操作系统进程通信机制的理解，更为今后进行更复杂的系统编程打下了坚实基础。