Lab6

Thinking

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?

父进程先关闭写通道。

```
father_process
    close(fildes[1]);
    read(fildes[0], buf, 100);
    printf("father-process read:%s",buf);
    close(fildes[0]);
    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/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?
> `dup` 函数：将一个文件描述符(`fd0`)所对应的内容，映射到另一个文件描述符(`fd1`)中；最终会将 `fd0` 和 `pipe` 的引用次数都增加1，将 `fd1` 的引用次数变为 `fd0` 的引用次数。
> 若在复制了文件描述符页面后，产生时钟中断，pipe的引用次数尚未增加，可能会导致另一进程调用 `pipeisclosed` ，发现 `pageref(fd[0]) = pageref(pipe)` ，误以为读/写端已经关闭。

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。
> 原因：系统调用时，陷入内核，关闭时钟中断，因此系统调用一定是原子操作。以下代码实现关中断：

```
.macro CLI
```

```
    mfc0 t0, CP0_STATUS
    li t1, (STATUS_CU0 | 0x1)
    or t0, t1
    xor t0, 0x1
    mtc0 t0, CP0_STATUS
.endm
```

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。
>   可以解决。

- 当`pageref(pipe) > pageref(fd)`时，无上述问题；
- 当`pageref(pipe) = pageref(fd)`时，即读缓冲区为空，写缓冲区为满时，会再次循环直到进程切换两者全部unmap为止。

> - 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。
>   `dup`也会出现类似问题。
>   解决方案：先对`pipe`进行`map`，再对`fd`进行`map`。

Thinking 6.5

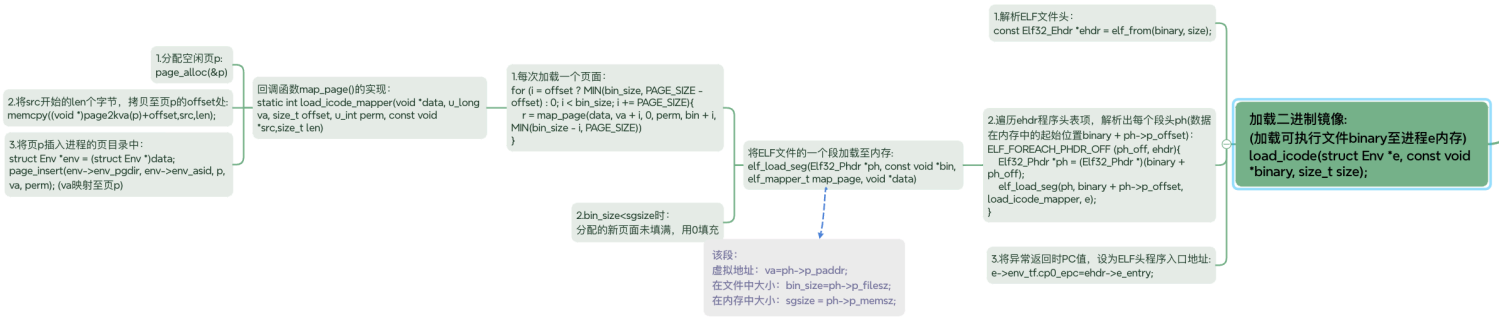
> 思考以下三个问题。
> • 认真回看 Lab5 文件系统相关代码，弄清打开文件的过程。

- 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`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。

> • 回顾 Lab1 与 Lab3，思考如何读取并加载 ELF 文件。

- `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，其中：`load_icode_mapper`回调函数，在**内核态**下加载 ELF 数据到内存空间；
- `spawn`函数在**用户态**下使用系统调用为 ELF 数据分配空间。

> 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 Note1.3.4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?



- `static int load_icode_mapper(void *data, u_long va, size_t offset, u_int perm, const void *src, size_t len);`

> `load_icode_mapper()` 函数：当 `src` 为 `NULL` 时，它不会执行 `memcpy()` 来复制数据（这正是我们想要的，因为 `bss` 段在文件中没有数据）。然后，它使用 `page_insert()` 来将页面插入到进程的页表中，并将其标记为有效（PTE_V）和其他必要的权限。
> 由于 `bss` 段在文件中没有数据，因此 `load_icode_mapper()` 在这种情况下不会实际复制任何数据到内存中。但是，由于调用了 `page_insert()`，操作系统会分配一个新的页面，并将其初始化为零（在大多数操作系统中，新分配的页面默认内容就是零）。因此，当 bss 段被映射到虚拟地址空间时，它实际上占据了一定的空间，并且这些空间的内容是零。

```
static int load_icode_mapper(void *data, u_long va, size_t offset, u_int
perm,const void *src, size_t len) {
    struct Env *env = (struct Env *)data;
    struct Page *p;
    //1.分配空闲页p
    try(page_alloc(&p));
    p->pp_ref++;
    //2.将src开始的len个字节，拷贝至页p的offset处
    if (src != NULL) {
        memcpy((void *)page2kva(p)+offset,src,len);
    }
    //3.将页p插入进程的页目录中:
    return page_insert(env->env_pgdir, env->env_asid, p, va, perm);
}
```

- `int elf_load_seg(Elf32_Phdr *ph, const void *bin, elf_mapper_t map_page, void *data);`

> `elf_load_seg()` 函数首先处理文件的实际内容（对应于 `p_filesz`），这部分内容对应于 `text` 和 `data` 段。对于每个页面，它使用 map_page() 函数来映射页面到虚拟地址空间，并将文件内容复制到对应的内存中（如果 `src` 非空）。
> 当处理完文件的实际内容后（即 `i` 等于 `bin_size`），`elf_load_seg` () 进入一个循环，该循环处理 `bss` 段。在这个循环中，它继续调用 `map_page()` 函数来映射页面到虚拟地址空间，但是这次 `src` 参数是 `NULL`（表示没有要从文件中复制的数据）。

```
//加载一个ELF文件，将其所有段映射至正确的虚拟地址
int elf_load_seg(Elf32_Phdr *ph, const void *bin, elf_mapper_t map_page, void
*data) {
        //从程序段头部ph,提取:虚拟地址va,文件大小bin_size,内存大小sgsize
        u_long va = ph->p_vaddr;
        size_t bin_size = ph->p_filesz;
        size_t sgsize = ph->p_memsz;
        //设置页面权限为有效位PTE_V;如果程序可写(PF_W),添加脏位(PTE_D)
        u_int perm = PTE_V;
        if (ph->p_flags & PF_W) {
                perm |= PTE_D;
        }

        int r;
        size_t i;
        //处理非页面对齐的虚拟地址
        u_long offset = va - ROUNDDOWN(va, PAGE_SIZE);
        if (offset != 0) {
                if ((r = map_page(data, va, offset, perm, bin,
                                  MIN(bin_size, PAGE_SIZE - offset))) != 0) {
                        return r;
                }
        }

        /* Step 1: load all content of bin into memory. */
        //循环映射整个bin_size大小的二进制文件至内存,每次映射一个页面
        for (i = offset ? MIN(bin_size, PAGE_SIZE - offset) : 0; i < bin_size; i
+= PAGE_SIZE) {
                if ((r = map_page(data, va + i, 0, perm, bin + i, MIN(bin_size -
i, PAGE_SIZE))) !=0) {
                        return r;
                }
        }

    /* Step 2: alloc pages to reach `sgsize` when `bin_size` < `sgsize`. */
```

```
        //若bin_size<sgsize,分配额外的页面,以达到sgsize
        while (i < sgsize) {
                if ((r = map_page(data, va + i, 0, perm, NULL, MIN(sgsize - i,
PAGE_SIZE))) != 0) {
                        return r;
                }
                i += PAGE_SIZE;
        }
        return 0;
}
```

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。

- 在`user/init`函数中，如下代码实现：

```
if ((r = dup(0, 1)) < 0) user_panic("dup: %d", r);
```

它将0映射在1上，实际上将控制台的输入输出缓冲区当做管道。

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?

- 外部命令。
- 因为我们的user文件夹中有cat.c ls.c文件，Linux下的cd指令没有对应的文件，使用时也不需要单独的创建一个子进程。cd 所做的是改变 shell 的 PWD。 因此倘若 cd 是一个外部命令，那么它改变的将会是子 shell 的 PWD，也不会向父 shell 返回任何东西。所以，当前 shell 的 PWD 就不会做任何改变。所有能对当前 shell的环境作出改变的命令都必须是内部命令。 因此如果我们将 cd 做成外部命令，就无法像原来一样改变当前目录了。

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?

- 两次，分别对应`[00001c03] SPAWN: ls.b`、`[00002404] SPAWN: cat.b`

> - 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?

- 四次，分别对应`[00003406] destroying 00003406`、`[00002c05] destroying 00002c05`、`[00002404] destroying 00002404`、`[00001c03] destroying 00001c03`.

难点

- 与Lab3对比的`load_icode_mapper`函数对比（内核态实现），Lab6`spawn`函数（用户态实现）

体会与感想

终于完成了本学期的OS实验部分（除挑战性任务外），收获挺多的。阅读了大量C源码，逐步分析过程，和理论课贯通，有了更深体会，感觉超棒!