杜佳男 21210110

操作系统lab6实验报告

1、实验思考题

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?

答：首先来看源代码：

```
#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);
    }
}
```

分析代码逻辑，在之前的学习中我们已经知道，`fork()`函数有三种返回值：

- 返回`-1`代表`Handle error`，此时直接跳出；
- 返回`0`代表子进程，此时进行读操作后退出；
- 返回`子进程的进程号`对于`default`，此时进行写操作后退出。

如果想让父进程作为“读者”，则应该把`default`后写操作的部分换为读操作，即：

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

同时让子进程作为"写者"则改`case 0`后的操作为写操作：

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

Thinking 6.2

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?

```
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;
}
```

答：主要原因是`dup`函数为非原子性的。它首先复制了`fd`描述符所在的页，接着逐页复制`fd`所对应的数据区(4MB)。这两步并不是原子操作，若`fd`所在页刚被复制之后还未复制数据区就发生了一个时钟中断，切换到其他进程运行，。这时`fd`描述符所对应的文件数据复制尚未完成，对`fd`描述符对应文件的操作就会出错。

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。

答：系统调用一定是原子操作。

- 从使用的角度来看，系统调用是操作系统内核为各种用户进程提供服务的接口，不应该被来自用户进程的时钟中断打断。
- 从代码实现角度来看，在进行系统调用时，首先会进入`handle_sys`函数，在函数的头使用`CLI`宏来关闭了中断位。

  lib/syscall.S -- handle_sys：

```
/*** exercise 4.2 ***/
NESTED(handle_sys,TF_SIZE, sp)
    SAVE_ALL                        // Macro used to save trapframe
    CLI                             // Clean Interrupt Mask
    nop
    .set at                         // Resume use of $at
```

  include/stackframe.h -- CLI：

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

.macro SAVE_ALL
```

Thinking 6.4

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，那么对于 dup 中出现的情况又该如何解决？请模仿上述材料写写你的理解。**

答：

- 可以解决上述场景的进程竞争问题。
  - `fork`结束后，子进程先执行，时钟中断产生在了`close(p[1])`与`read`之间，`p[1]`已经解除了对于`pipe`的映射，同时也解除了对于`p[1]`的映射。
  - 父进程在`close(p[0])`中产生时钟中断，此时已经解除了对`p[0]`的映射，但是还没有来得及解除`p[0]`对于pipe的映射。
  - 此时分析每个页的实际引用情况，`pageref(p[0])=1`，`pageref(p[1])=1`，`pageref(pipe)=1`，此时子进程执行`read`函数，不会判断写端已经关闭而直接退出。
- 对于`dup`中出现的情况，只需要交换`fd`与`fd数据区`的复制顺序。只要我们将复制数据区的操作放在复制`fd`描述符对映页的操作之前，就可以保证`fd`文件描述符完成复制时它所对应的内容也完成复制，从而解决`fd`复制而对应文件内容未复制的问题。

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?

```
        while (i < sgsize) {
                ret = syscall_mem_alloc(child_envid, va + i, PTE_R | PTE_V);
                if (ret) return ret;
                i += BY2PG;
    }
```

答：在`usr_load_elf`函数中，将ELF的text端加载全部加载进内存后，对于未到`p_memsz`（在上图代码中已被赋值给`sgsize`）大小的部分，使用`syscall_mem_alloc`函数在子进程的对应地址处分配一页的大小，并设置标志位为可读、可写，这个函数所调用的`page_alloc`函数会主动调用`bzero`将这一页地址数据清0，既分配了空间，又实现了bss端初始值设置为0的操作。

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?

答：代码段为`UTEXT`以下的区域，因此任意elf文件的偏移量都应该是一样的，这样方便设置指针时统一地将其设置在`UTEXT`处。

Thinking 6.7

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。

答：标准输入与标准输出的安排是通过`user/init.c`中调用函数`opencons`来实现的。

user/init.c -- umain：

```
    // being run directly from kernel, so no file descriptors open yet
    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);
```

user/console.c -- opencons:

```
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;
    fd->fd_omode = O_RDWR;
    return fd2num(fd);
}
```

此时直接从内核运行代码，还没有完成对文件描述符的相关设置，故通过opencons函数获取的就是0号和1号文件描述符。

2、实验难点

- `spawn.c`，简直有毒……`spawn`函数的作用是帮助我们调用文件系统中的可执行文件并执行。从文件系统中读出可执行文件并加载到子进程的地址空间中，是在用户态下进行的，加载可执行文件到内存需要通过系统调用。
- 前面思考题中关于竞争的部分。

3、体会与感受

- 首先是OS实验系列完结撒花，有一说一整个系列实验占用时间长的离谱，特别是debug……
- 再就是疯狂吐槽，`spawn.c`部分提示太少了，完全没有思路
- 最后，感谢助教

4、指导书反馈

- 希望增加更多有关spawn实现部分的提示