LAB6实验报告

扶星辰

19377251

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？

修改为如下：

```
switch (fork()) {
    case -1: /* Handle error */
        break;
    case 0: /* Child - reads from pipe */
        close(fildes[0]); /* Write end is unused */
        write(fildes[1], "xxx", n); /* Get data from pipe */
        printf("child-process write:%s",buf); /* Print the data */
        close(fildes[1]); /* Finished with pipe */
        exit(EXIT_SUCCESS);
    default: /* Parent - writes to pipe */
        close(fildes[1]); /* Read end is unused */
        read(fildes[0], buf, 100); /* Write data on pipe */
        close(fildes[0]); /* Child will see EOF */
        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？

关系式： `pageref(rfd) + pageref(wfd) = pageref(pipe)` 在非原子的close函数调用时不能保证成立。而为了使得另一个进程在判断管道某端关闭时的条件 `pageref(itself-fd)=pageref(pipe)` 不会出错，在close函数中需要先将`pagefref(itself-fd)`减小1，再将 `pageref(pipe)` 减小1，从而保证在该函数不会导致子进程发生错误判断。

`close(fd)`函数会将 `fd` 和 `pipe` 所在页面的ref均减小1，为保证管道另一端在真的关闭之前，当前端的读写不会发生误判，需要先`unmap fd` ，再`unmap pipe`，而`dup`函数则与之相反：将 `fd` 复制给另一个文件描述符，即 `fd` 所在的页面和 `pipe` 所在的页面的`pp_ref`数都要增加1，因此其`unmap`的次序也应当与`close`函数恰好相反，若不是这样，则以下实例可能发生同步的错误：

```
int pip[2] = {0};
pipe(pip);
p_id r = 0;
if((r=fork)==0) {
```

```
    close(pip[0]);
    write(pip[1], "a", 1);
}
if (r>0) {
    close(pip[0]);
    dup(STDOUT, pip[1]);
    close(pip[1]);
    execvp("ls", "-a", NULL);
    ...
}
```

此处本来应当由父进程调用 execvp 函数去执行 ls -a 的命令，将结果送入管道写端，但是，dup 函数若执行到恰好将stdout的文件描述符被替换(map)为pip[1]的文件描述符后就被切换为子进程此时 wfd=2， rfd=1，pipe=2 ,此时子进程将判断写端已经关闭，从而write函数执行出错。

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。

系统调用d都是原子操作，具体代码是在include/stackframe.h中定义的CLI宏，如下所示：

```
.macro
    CLImfc0 t0, CP0_STATUS
    li t1, (STATUS_CU0 | 0x1)
    or t0, t1
    xor t0, 0x1
    mtc0 t0, CP0_STATUS
.endm
```

CLI宏在handle_sys函数中出现，作用是设置CP0_STATUS寄存器，因此后面的中断无法发生，因此就无法发生嵌套中断，导致系统调用也不能被打断，因此系统调用是原子操作。对于比较特殊的 sys_ipc_receive ，也是在等待时设置为 ENV_NOT_RUNNABLE 后让步，待发送消息的进程将其接收消息的进程重新设置为env_runnable，由于 sys_ipc_can_send 是原子性的，因而其全过程也是原子性的。

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 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。

详情见Thinking6.2

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？

Load二进制文件时，根据bss段数据的 memsz 属性分配对应的内存空间并清零。

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?

在user的lds中规定了所有加载的可执行程序的text段的偏移值都是一样的。

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。

在user/init函数中可以看到如下步骤，它将0映射在1上，相当于就是把控制台的输入输出缓冲区当做管道。

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

实验难点图示

说一下较为复杂的函数spawn

下面简要说一下实现流程

- 首先从文件系统打开对应的文件，这些可执行文件以.b结尾 例如cat.b ls.b
- 用 fork 创建子进程，
- 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。大家下个学期学习编译原理后，会对这一点有更加深刻的认识。
- 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）
- 最后，把子进程设置为RUNNALBLE

此外，再简要说一下sh.c的主体逻辑，在umain中是一个无限循环，循环中会一直从命令行中读取命令，读取完命令后，fork出一个子进程运行runcmd，在runcmd中，主要的工作是对命令行接受来的命令做分词解析，其中定义了WHITESPACE " \r\n\t"和SYMBOL 符号等，其中用了gettoken和_gettoken函数，每条命令在第一次调用gettoken后会由其中的static变量存储这条指令的一些地址信息，随后使用_gettoken函数进行分词，不同的参数之间用\0隔开，传递给argv数组，在全部解析完成后就跳转到runit先fork一个子进程，子进程调用spawn函数来加载可执行文件完成对应的命令。

体会与感想

做完lab6就相当于os的实验快要画上句号了，这次的lab相较以前的lab难度稍微低了一些，但是遇到了一些以前lab的小bug，在debug中浪费了一些时间，此外这次写lab6时通过spawn函数体会到了程序加载的过程，对程序运行有了更深刻的理解。

指导书反馈

无

残留难点

无