实验思考题

Thinking6.1

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？

只需要将 `case 0` 和 `default` 两个部分的代码交换即可。

```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 - 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);
        default: /* Parent - reads from 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);
    }
}
```

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？

dup 函数的功能是将文件描述符 oldfdnum 所对应的内容映射到文件描述符 newfdnum 中，会将 oldfdnum 和 pipe 的引用次数都增加 1，将 newfdnum 的引用次数变为 oldfdnum 的引用次数。

当我们将一个管道的读/写端对应的文件描述符（记作 fd[0]）映射到另一个文件描述符。在进行映射之前，f[0]，f[1] 与 pipe 的引用次数分别为 1，1，2。按照 dup 函数的执行顺序，会先将 fd[0] 引用次数 +1，再将 pipe 引用次数 +1，如果 fd[0] 的引用次数 +1 后恰好发生了一次时钟中断，进程切换后，另一进程调用 _pipeisclosed 函数判断管道写端是否关闭，此时 `pageref(fd[0]) = pageref(pipe) = 2`，所以会误认为写端关闭，从而出现判断错误。

Thinking 6.3

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。

在进行系统调用时，系统陷入内核，会关闭时钟中断，以保证系统调用不会被打断，因此系统调用都是原子操作。

```
.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 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。

  可以解决。如果程序正常运行，pipe 的 pageref 是要大于 fd 的，在执行 unmap 操作时，优先解除 fd 的映射，这样就可保证严格大于关系恒成立，即使发生了时钟中断，也不会出现运行错误。

- 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。

  同样的道理，在 dup 使引用次数增加时，先将 pipe 的引用次数增加，保证不会出现两者相等的情况。

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？

处理 bss 段的函数是 lab3 中的 load_icode_mapper。在这个函数中，我们要对 bss 段进行内存分配，但不进行初始化。当 `bin_size < sgsize` 时，会将空位填 0，在这段过程中为 bss 段的数据全部赋上了默认值 0。

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 user.lds 文件中对 .text 段的地址做了统一的约定。

Thinking 6.7

- 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?

  在 MOS 中，我们用到的 shell 命令是外部命令，需要 fork 一个子 shell 来执行命令。

- 请思考 为什么 Linux 的 cd 指令是内部指令而不是外部指令?

  Linux 的 cd 指令是改变当前的工作目录，如果在子 shell 中执行，则改变的是子 shell 的工作目录，无法改变当前 shell 的工作目录。

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。

在 user 文件夹的 init.c 文件中的 umain 函数当中有下面的部分。

```
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);
```

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?

  两次 spawn，结果如下所示。

  ```
  $ 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
  ```

- 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?

  四次进程销毁，结果如下所示。

```
[00002c05] destroying 00002c05
[00002c05] free env 00002c05
i am killed ...
[00003406] destroying 00003406
[00003406] free env 00003406
i am killed ...
[00002404] destroying 00002404
[00002404] free env 00002404
i am killed ...
[00001c03] destroying 00001c03
[00001c03] free env 00001c03
i am killed ...
```

实验难点

填写有关管道的函数，即 `piperead`，`pipewrite` 和 `_pipeisclosed` 函数。

- `_pipeisclosed`：这是一个判断函数，这个函数的判断机制主要是看 `pageref(fd)` 和 `pageref(page)` 的大小，为保证这两个值在读取过程中没有切换，需要不断进行 `env_runs` 的判断，直到稳定为止。
- `piperead`：在 `p_rpos ⩾ p_wpos` 时不能立刻返回，而是应该根据 `_pipeisclosed()` 的返回值判断管道是否关闭，若未关闭，应执行进程切换。
- `pipewrite`：在 `p_wpos - p_rpos ⩾ BY2PIPE` 时不能立刻返回，而是应该根据 `_pipeisclosed()` 的返回值判断管道是否关闭，若未关闭，应执行进程切换。

体会与感想

Lab6 的内容相对较少，但理解上的难度较高，很多函数都给人一种难以下手的感觉，在弄懂函数原理的过程中，花费了很多的功夫。不过这是最后一个 Lab 了，OS 实验课到这里也要结束了。这应该是我接触计算机以来，第一次接触有关内核的知识，第一次去一步步探索计算机的底层代码如何执行，虽然说中间不管是课下实验还是课上实验都遇到了些许困难，但最终的收获还是很大的。