Lab 2.2 Linux Shell 提示

首先首先，system() 这个函数实在是太开挂了，不推荐使用。根据 system(3) 的手册，这个函数内部其实是运行了 `/bin/sh -c <command>` 来执行传进去的命令，也就是调用了系统自带的 Shell（一个完整的 Shell）。如果你要用 system() 的话，我觉得完全没必要做这个实验了。（个人观点，具体以助教为准）

关于 system(3) 中的括号和数字 3 的意思，见附录 A。

好了下面开始正文，其中运行命令的部分假设你用 fork() 和 execvp() 来实现，因为*我就是这么实现的*。

1. 命令执行

按逻辑顺序来讲读进来的命令是先处理（分割）再执行的，但是考虑到我们执行的方式依赖于其他函数（也就是 execvp()）所以这里有必要先考虑一下怎么执行命令。

根据 execvp(3) 的手册，它有两个参数。第一个参数是一个字符串，表示你需要运行的命令。这里 execvp() 和 execv()（助教给的）有一个很重要的区别，能省不少事。第二个参数是一个字符指针数组，每个元素（字符串）表示被执行程序的一个参数。这个数组传递到被执行的程序那里就相当于 `int main(int argc, char** argv)` 里面的那个 argv 了，也就是命令行参数。需要注意的是，根据 C 语言的标准，`argv[argc]` 是一个空指针（而不是数组越界）。所以如果你需要运行 `ls -l /` 这样一个命令，需要用类似下面的代码：

```
char cmd[3][4] = {"ls", "-l", "/"};
char *argv[4] = {cmd[0], cmd[1], cmd[2], NULL}; // 注意结尾的空指针
execvp(argv[0], argv); // 可以直接用 argv[0] 作为命令
```

然后是 fork() 的用法。fork() 会把当前进程复制一份，产生一个新进程。对于父进程（运行 fork() 的那个），返回值是子进程的 PID；而对于子进程，返回值是零。所以使用 fork + exec 运行命令的基本思路是先 fork() 一个，然后判断是父进程还是子进程，子进程 exec 到要运行的命令，而父进程就用 waitpid() 等待子进程结束。代码类似这样：

```
pid_t child_pid = fork();
if (child_pid == 0) {
    // 子进程，准备 exec
} else {
    // 父进程，等待子进程结束
    int status;
    status = waitpid(child_pid, &status, 0);
    // status 的用法详见 [man 2 wait]
}
```

知道了我们需要怎样执行命令，也就是知道了我们对命令分割的需要的结果。

2. 命令分割

字符串处理，数据结构课学过的。这个其实比较简单，直接把输入字符串扫描一遍，每个单词的开头处用一个指针指着，按照上面介绍的 execvp() 的需要，遇到空格什么的直接替换成空字符（`\0`）就行了。可以参考下面这个示意图：

```
Input: "ls -l /"
cmdline: 'l', 's', ' ', '-', 'l', ' ', '/', 0
replace: 'l', 's',   0, '-', 'l',   0, '/', 0
          ^             ^             ^
        p[0]          p[1]          p[2]    p[3]=NULL

execvp(p[0], p);
```

基本上做到这一步就可以了，最后那个 execvp() 就可以正常运行命令了。

关于分号（子命令）的处理，其实也非常简单。把分号当做空格，如果遇到了分号就同样替换成零字符，但是这时候需要跳出循环去先把已经分割好的命令执行掉，并清空指针数组 p 准备重新分割后面的命令。

对于管道也是类似，只不过这里需要注意的一点是，遇到管道符号 `|` 的时候表示当前命令已经处理完毕，但是这里需要先把当前命令存起来，把下一条命令也分割好，再一起进入管道处理的部分。

3. 管道处理

popen(3) 这个操作我其实不是很懂为什么，因为这里开了两个管道，两个子进程通过父进程交换数据，而不是直接传输，多了一层完全没必要的工作，不过既然助教给了这个函数，我们就用吧。

具体操作不难想，就是将两个命令分别用 popen 执行，得到两个文件指针，其中一读一写。父进程将数据从一个文件指针写到另一个文件指针中，然后两边都 pclose 一下，就算完成了。大体思路如下：

```
FILE *fr = popen("left command", "r");
FILE *fw = popen("right command", "w");
fread(fr, ...);
fwrite(fw, ...);
pclose(fr);
pclose(fw);
```

这个方法有一个明显的缺点，就是只支持一个管道，不支持串联的多个管道。如果想要支持多个管道，需要使用 pipe(2) 和 dup2(2) 这两个系统调用。考虑到这并不是本实验的要求，有兴趣的同学可以自己查资料尝试。

4. 错误处理（选做）

只想交 Lab 的同学这一节不用看了，毕竟连我都给它标上了*选做*。

本实验中用到的几个系统调用都会在成功时返回 0（exec 和 fork 的父进程除外），在失败时返回 -1。所以判断系统调用是否成功，可以检查返回值是否大于等于零。如果返回值是 -1，那么还有一个额外的变量 `errno` 会被设置为错误代码，可以用 strerror(3) 来将错误代码转换成可以输出的文字。变量 `errno` 在头文件 `errno.h` 中，而函数 `strerror` 在头文件 `string.h` 中。

参考以下代码：

```
#include <errno.h>
#include <string.h>

dup2(-1, 0);
printf("%s\n", strerror(errno));

```

```c
execl("/asdfg", "", NULL);
printf("%s\n", strerror(errno));

execl("/bin", "", NULL);
printf("%s\n", strerror(errno));

execl("/dev/ram", "", NULL);
printf("%s\n", strerror(errno));
```

该示例代码的一个可能的输出是：

```
Bad file descriptor
No such file or directory
Permission denied
Permission denied
```

这样可以很简便地产生可读的错误信息。

附录

A. 查看系统自带的手册

*几乎*所有的终端命令、系统调用、C 语言库函数等内容在 Linux 系统中都有自带的 “用户手册” 可以查看，方法是直接运行 `man <something>`，其中 something 是你想查看的命令或者函数，比如 `man fopen` 就能看到 C 语言 fopen() 函数的手册。你的终端会变成一个可以上下滚动的页面显示这个用户手册。按键盘的上下键或者（有时候）用鼠标滚轮可以翻页查看，按 Q 退出这个手册。

有时候一个条目可能出现在多个 “分区” 里，例如 stat 就有 [stat(1)][1-stat] 和 [stat(2)][2-stat] 两个版本（命令行命令和系统调用），直接运行 `man stat` 只会给出第一个结果，这时候可以运行 `man 2 stat` 来告诉 man 你要查看 stat(2) 而不是搜索到的第一个页面。

也许你已经发现了，数字指的就是条目对应的 man 文档所属的 “分区”，所以 chmod(1), chown(1), stat(1) 都指代命令行的命令，chmod(2), chown(2), stat(2) 都是系统调用。常见的几个分类如下：

```
Section 1
    user commands (introduction)
Section 2
    system calls (introduction)
Section 3
    library functions (introduction)
Section 4
    special files (introduction)
Section 5
    file formats (introduction)
Section 6
    games (introduction)
Section 7
    conventions and miscellany (introduction)
Section 8
    administration and privileged commands (introduction)
```

另外，这种公开的文档网上也有一大把，例如你也可以在 https://linux.die.net/man/ 上用同样的方式查看手册。善用 **Google**。

B. 吐槽处

关于*系统调用*，以下是助教 PDF 中原文

> 熟悉以下系统调用的用法
>
> ```
> pid_t fork();   //创建进程
> pid_t waitpid(pid_t pid,int* status,int options);   //等待指定pid的子进程结束
> int execv(const char *path, char *const argv[]);     //根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除了进程号外，其他全部被新程序的内容替换了
> int system(const char* command);     //调用fork()产生子进程，在子进程执行参数command字符串所代表的命令，此命令执行完后随即返回原调用的进程
> FILE* popen(const char* command, const char* mode); //popen函数先执行fork，然后调用exec以执行command，并且根据mode的值（"r"或"w"）返回一个指向子进程的stdout或指向stdin的文件指针
> int pclose(FILE* stream);   //关闭标准I/O流，等待命令执行结束
> ```

具体*系统调用*的定义和解释可以看 syscalls(2)。上面列出来的 6 个函数中，只有最上面的两个是真 · 系统调用（它们的 man 文档都在第二部分）。第三个 execv(3) 是 execve(2) 的“前端”，实际上属于 exec(3) 系列，经常容易搞混。

至于后面三个嘛，实在想吐槽了。大家学 C 的时候都知道有这么个函数了，还是*系统调用*？ popen(3) 也是一个 C 语言的库函数，其实是包装过的 pipe(2), fork(2) 和 dup2(2) 的 “前端” 函数。把 execv 当做系统调用就算了，这几个函数*也*当做系统调用列出来，emmmm……