山东大学 计算机科学与技术 学院

操作系统 课程实验报告

学号：201800130005	姓名：张畅	班级：2018 级计科三班
实验题目：MSH(SHELL 实验)		
实验学时：2	实验日期：2020.11.24	

实验目的：

理解 UNIX 或者 LINUX 系统中的 shell 是如何进行系统调用的。

实验过程中遇到和解决的问题：

（记录实验过程中遇到的问题，以及解决过程和实验结果。可以适当配以关键代码辅助说明，但不要大段贴代码。）

本实验考验了对进程及管道操作的理解和应用，需要对这些相关的机制相当熟悉才能够编写出相应的程序。同时因为 Shell 的执行对 Linux 中的指令熟练程度也有较高的要求，只有熟悉每条指令的执行内容和执行结果，才能在调试中更好的确定程序的正确性，快速地找到程序中的问题。

在实验调试的过程中，出现了调试结果正确但是测试程序无法通过的情况，多次检查无果，于是在网站上查询了 Linux 相关的指令调用方法和执行内容，发现自己对该指令的理解存在一定的偏差，导致了错误地认为开始的程序调试结果是正确的。经过重新的排查错误的修改之后，程序通过。

实验步骤与内容：

步骤一：复习并熟悉关于进程和管道的相关操作：

进程的使用：

fork()

返回一个子进程的进程号 pid：

execve()

path 绝对路径的文件名字符串

argv 参数列表

envp 环境列表，可以为空

wait()

用于等待任何一个子进程执行成功

waitpid()

用于等待某一个特定的子进程执行完成

getpid()

获得当前进程的进程号

getppid()

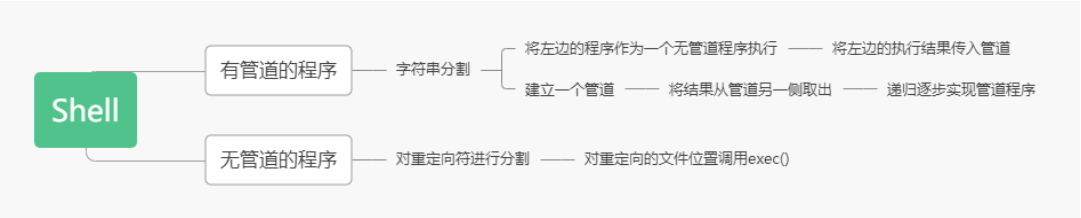
获得当前的进程的父进程的进程号

signal()

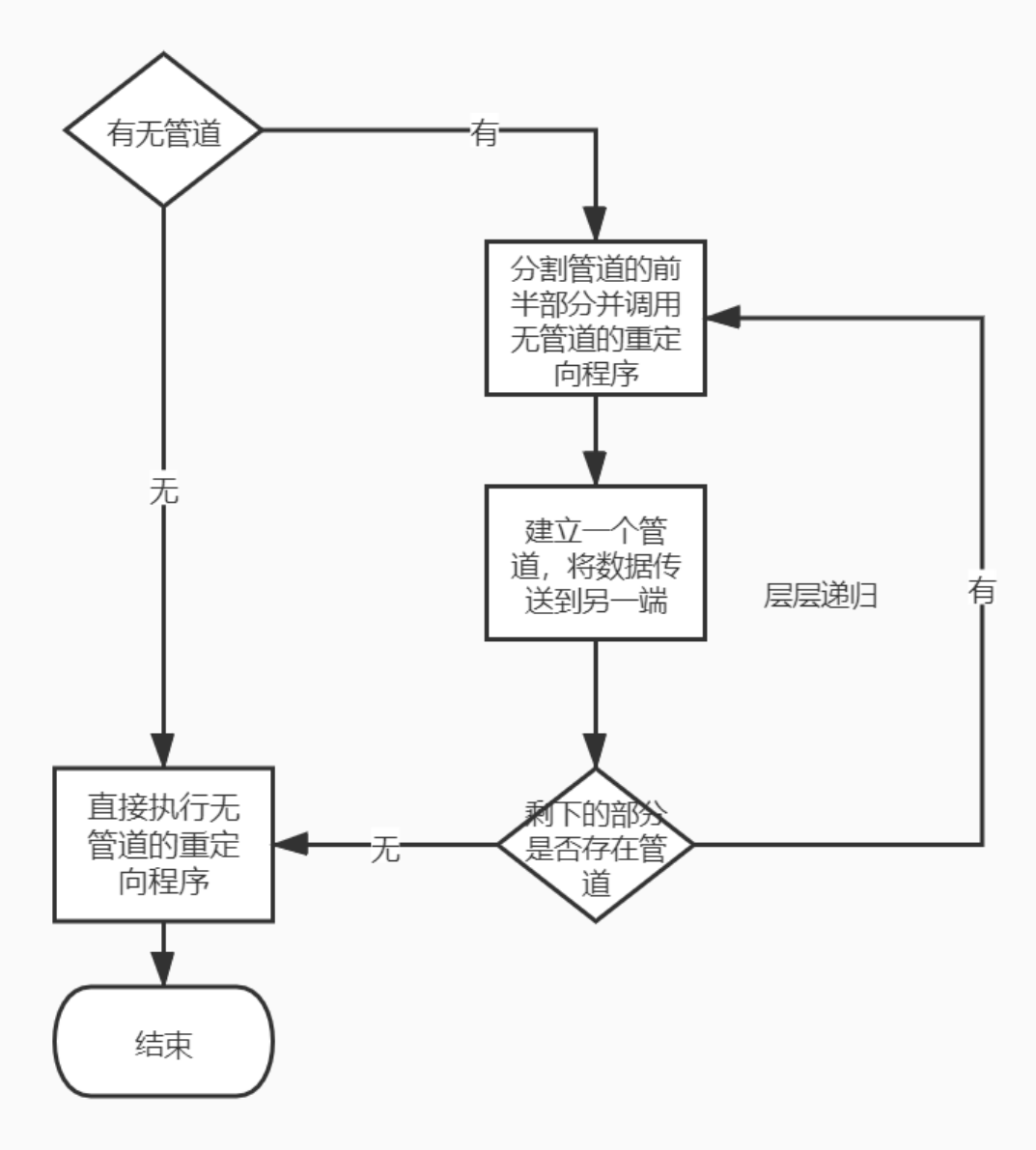
注册某个信号的信号处理函数

示例：

kill()
向某个进程发送一个信号 sig
失败返回-1，否则>0
pause()
挂起当前的进程，如果当前的进程捕获了一个信号，回调进程继续执行
sleep()
使当前的系统睡眠一段时间，执行睡眠的秒数
管道：
pipe()
使用 pipe 将建立一个无名管道。成功返回 0，失败返回-1.
文件的读写操作
dup()
dup 能够复制参数 oldfd 指向的文件描述符并分配一个新的文件描述符指向 oldfd，将其返回。调用之后，新分配的文件描述符与 oldfd 指向的文件描述符指向同一个文件，共享所有权限。
dup2 ()
dup2 所复制的文件描述符与原来的文件描述符共享各种文件状态。共享所有的锁定，读写位置和各项权限或 flags 等。
步骤二：设计 MSH 的逻辑结构
经过思考，可以将 MSH 的设计分成两个部分，管道的实现和无管道程序的调用，两个模块的组合调用就可以实现。



无管道程序的调用：无管道程序调用的核心问题就是解决重定向问题，即如何识别“<”和“>”，并且将执行的程序结果输出到指定的文件中。
确定重定向符号的方法：
利用 strcmp()函数去匹配需要的字符串，即重定向输出"<"和重定向输入符">"；出现了重定向符号之后肯定就是输入和输出的目标文件，也需要相应的变量去存储；对于重定向操作，需要执行的指令到重定向符之前，需要把指令执行的下标进行修改；
处理管道指令的方法：
（1） 考虑特殊情况，‘|’后面没有指令，那么就等于没后续的指令可以执行；
（2） 先定义好管道，然后申请子进程；
（3） 需要对输入、输入端进行重定向；
步骤三：实现 MSH 并调试
分成两个函数分别实现无管道的程序重定向执行和管道功能，封装之后组合调用实现了 MSH，经过调试功能正常。整体的实现逻辑思路如下所示：



步骤四：测试实验结果并使用测试程序运行：
对上述实现的代码进行了测试，运行测试程序后执行结果如下所示：

```
tang@ubuntu:~/OS-exp/OS/OS-resources/lab3$ make grade
+ grading...
simple echo: PASS
simple grep: PASS
two commands: PASS
output redirection: PASS
input redirection: PASS
both redirection: PASS
simple pipe: PASS
pipe and redirections: PASS
lots of commands: PASS
two commands with multiple pipes: PASS
signal SIGINT: PASS
All tests passed :)
```

实验总结：
1、输入的命令的识别
实际上对输入命令的处理就是一个对字符串的处理，用“ ”进行分割，然后利用 strtok_r 函数进行分割，获取要执行的命令和之后的操作；需要注意的就是函数每次切割之后得到的剩余字符串的存储方式；
2、没有管道指令的命令处理
这一段实际上才是处理程序段的重点。因为其实管道指令也就是一个个命令的处理，只不过是换到了相连的管道中去执行。处理的关键在于是否有重定向操作，如果没有，直接利用 execvp 函数执行就可以了；如果有的话，那么还需要先用特定的模式操作将目标文件打开，再执行操作。
3、管道指令的处理
重点即在于对于输出和输入端的定义，需要考虑清楚对于在执行操作的管道，它的输出和输入究竟是需要定向为标准的，还是管道的。最终实现的目标情况就是命令会通过管道将左侧程序的输出作为输入传递给右侧程序。管道支持多级连接，即左侧的标准输出内容会被一次 传递给右侧的下一级，作为它们的标准输入。
4、shell 程序的实现过程
建立一个新的进程，然后在那个进程中运行一个程序（如完成 ls 操作）然后等待那个进程执行结束。然后 shell 便可读取新的一行输入，建立一个新的进程，在这个进程中运行程序 并等待这个进程结束。所以要写一个 shell，需要循环以下过程：
（1）获取命令行
（2）解析命令行
（3）建立一个子进程（fork）
（4）替换子进程（execvp）
（5）父进程等待子进程退出（wait）

附录：程序源代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <wait.h>
#include <time.h>
#include <fcntl.h>

pid_t parentpid, childpid;
typedef void (*sighandler_t) (int);

//信号处理功能：如果是父进程收到了一个 SIGINT，就直接发送一个 SIGINT 给子进程，从而关闭掉这个执行命令的进程
```

```
void sigcat(){
    if(getpid() == parentpid)
        kill(childpid, SIGINT);
}

//获取输入的信息放入到 str 字符串中
void getstr(char *str){
    char ch;
    int i = 0;
    while(scanf("%c", &ch) != EOF){
        if(ch != '\0' && ch != '\n')
            str[i++] = ch;
        else
            break;
    }
}

//解析命令
int parse(char *cmd, char **args){
    //分割字符串的初始化
    memset(args, 0, sizeof(char*)*64);//清除掉命令行的数据
    int cnt = 0;//记录切割后的参数数目个数
    char *str = cmd;//保存输入的命令
    char *tmp_str = NULL;//保存每次分割后的剩余字符串

    //对输入命令字符串按照中间的空格进行切割，切割结果放在 args 数组里
    while((args[cnt] = strtok_r(str, " ", &tmp_str)) != NULL){
        cnt++;
        str = NULL;
    }

    //返回一个切割的参数数量
    return cnt;
}

//执行无管道的指令
void exec_cmd(char *cmd){

    //进程使用
    size_t pid;
    int status;

    char *in = NULL;
    char *out = NULL;
```

```
//在重定向系统中输入和输出文件的名称
int in_num = 0;
int out_num = 0;
//重定向输入和输出的个数，方便判断使用者的指令是否存在错误

char temp[256];
strcpy(temp, cmd);
 char *args[256];
//存储命令行的参数
int argc = parse(temp, args);
//argc 分割后得到的字符串的数目

int eidx = argc;//对于重定向操作的终止下标

for(int i = 0; i < argc; i++){
    //确定重定向符“>”和“<”的数目
    if(!strcmp(args[i], "<")){
        if(i + 1 >= argc){
            printf("File name dose not exist.\n");
            exit(1);
        }
        else
            in = args[++i];//存储重定向输入文件名

        in_num++;
        if(eidx == argc) eidx = i - 1;
    }

    if(!strcmp(args[i], ">")){
        if(i + 1 >= argc){
            printf("File name dose not exist.\n");
            exit(1);
        }
        else
            out = args[++i];//存储重定向输出文件名

        out_num++;

        //对于重定向操作,重定向符所在的位置在文件名前一个字符上，最后一个字符是文件名
        if(eidx == argc) eidx = i - 1;
    }
}

//重定向输入输出文件数量过多，提示
```

```
if(in_num > 1){
    printf("Too many redirection input files\n");
}
else if(out_num > 1){
    printf("Too many redirection output files.\n");
}

//如果是输入操作，需要考虑文件不存在的情况
else if(in_num == 1){
    FILE *fp = fopen(in, "r");
    if(fp == NULL)
        printf("The input file dose not exist.\n");
    fclose(fp);
}

//处理指令
childpid = fork();
if(childpid < 0){
    printf("Create Process fail!\n");
    exit(1);
}
//子进程
else if(childpid == 0){
    //有重定向输入
    if(in_num == 1){
        close(0);//如果有重定向输入，需要关掉 0，open 打开的新文件描述符会负责输入
        if(open(in, O_RDONLY) != 0){
            fprintf(stderr, "testsh: open != 0\n");
            exit(-1);
         }

    }

    //有重定向输出
    if(out_num == 1){
        close(1);//如果有重定向输出，需要关掉 1，open 打开的新文件描述符会负责输出
        if(open(out, O_CREAT|O_WRONLY|O_TRUNC, 0644) != 1){
            fprintf(stderr, "testsh: open != 1\n");
            exit(-1);
         }
    }

    char *tmp[256];
    for(int i = 0; i < eidx; i++)
```

```
            tmp[i] = args[i];
        tmp[eidx] = NULL;
        //如果是有重定向的这里就是将重定向符置为NULL
        //否则就是把argc位置置为NULL

        if(execvp(tmp[0], tmp) < 0)//执行操作
            printf("ERROR: invalid command.\n");
        exit(1);
    }
    //父进程
    else{
        waitpid(childpid, &status, 0);
    }
}

//处理管道指令
void pipe_cmd(char *cmd){

    size_t pid;

    char *tmp_str;
    //存放多级管道指令中当前需要执行的指令的字符串
    char *next = NULL;
    //存储分割后的剩余指令字符串
    char *args[256];

    //无管道指令，直接调用exec执行
    if(strstr(cmd, "|") == NULL){
        exec_cmd(cmd);
        return;
    }

    //有管道指令,对指令进行一次切割
    tmp_str = strtok_r(cmd, "|", &next);
    if(*next == 0){
        printf("There are no follow-up commands after |.\n");
        exit(1);
    }

    int pipe1[2];
    if(pipe(pipe1) < 0){
        printf("create pipe failed.\n");
    }
    //利用进程的并发和管道来实现管道的数据传输
```

```
    childpid = fork();
    if(childpid < 0){
        printf("create fork failed.\n");
        exit(1);
    }
    //子进程实现左边部分，并且将结果输出到管道的一端
    else if(childpid == 0){
        //关掉管道的读
        close(pipe1[0]);
        //将标准输出重定向到管道的输出
        dup2(pipe1[1], 1);
        //执行前半部分指令
        exec_cmd(tmp_str);
    }
    //父进程实现右边部分，将子进程左边传入的结果读取并作为右边指令的输入
    else{
        int status;
        waitpid(childpid, &status, 0);//父进程需要等子进程执行完成
        //关掉管道的写
        close(pipe1[1]);
        //将标准输入重定向到管道的输入
        dup2(pipe1[0], 0);

        pipe_cmd(next);//处理后半部分指令指令

        close(pipe1[0]);
    }
}

int main(){
    parentpid = getpid();
    signal(SIGINT, (sighandler_t)sigcat);
    char cmd[256];
    char *argclist[256];

    while(1){
        memset(cmd, 0, sizeof(cmd));
        //获取输入的命令并且刷洗缓冲区
        getstr(cmd);
        fflush(stdin);

        if(*cmd == '\n') continue;
        if(*cmd == '\0') exit(1);
```

```
        char tep_cmd[256];
        strcpy(tep_cmd, cmd);
        parse(tep_cmd, argclist);//对输入的指令进行解析

        //备份原标准输入输出标识符
        int stdin_FLAG = dup(0);
        int stdout_FLAG = dup(1);

        pipe_cmd(cmd);

        dup2(stdin_FLAG, 0);  //恢复标准输入输出标识符
        dup2(stdout_FLAG, 1);
    }
    return 0;
}
```