实验名称：添加 Linux 系统调用

PB21111681 朱炜荣

实验目的：

1.学习如何使用 Linux 系统调用，并利用简单的系统调用来实现一个简单的 shell 程序。

2.学习如何添加 Linux 系统调用，并基于自己的系统调用来实现一个简单的 top 程序。

实验环境：

（虚拟机）OS: Ubuntu 20.04.4 LTS ； Linux 内核版本: 4.9.263

实验内容（描述实验步骤并介绍代码原理）：

Part1：“实现一个 Shell”

首先我先阅读了实验文档中的相关知识介绍，同时阅读了助教给出的代码框架大致理解了 shell 程序的各个部分的逻辑和实现方式。

shell 程序的主体为一个 while（1）的死循环，在循环的最开始指令会读入用户输入的指令，在读入之前有一个输出类似 shell->的模块，代码实现逻辑为使用系统调用函数 getcwd 来获取当前的工作路径：

```
    char pwd[255];
    getcwd(pwd, 255);
    printf("shell:%s -> ",pwd);
    fflush(stdout);
```

然后我的代码实现逻辑会调用助教给出的 split_string（）函数对读入的命令按照“；”进行第一次切割，然后进入一个新的循环用来处理多命令的指令输入，直到这个新循环结束再读取新的命令行。

然后对于单条指令的处理，框架的逻辑为判断命令中是否存在管道符“|”，然后对于管道符的个数进行分类，如果个数为 0 则只需要判断是否为内建命令，内建命令只需要在主程序中运行即可，若为其它命令则在子进程中执行。

```
char *argv[MAX_CMD_ARG_NUM];
int argc;
int fd[2];
/* TODO:处理参数，分出命令名和参数
*
*
*
*/
/* 在没有管道时，内建命令直接在主进程中完成，外部命令通过创建子进程完成 */
argc = split_string(commands[0], " ", argv);
//
if(exec_builtin(argc, argv, fd) == 0) {
continue;
}
```

```
/* TODO:创建子进程，运行命令，等待命令运行结束
*
*
*
*
*/
int pid = fork();
if(pid == 0){
//execute(argc,argv);
//execvp(argv[0],argv);
fd[READ_END] = STDIN_FILENO;
fd[WRITE_END] = STDOUT_FILENO;
argc = process_redirect(argc, argv, fd);
dup2(fd[READ_END], STDIN_FILENO);
dup2(fd[WRITE_END], STDOUT_FILENO);
execvp(argv[0],argv);
exit(255);
}
else{
wait(NULL);
}
```

如果管道数为 1 时，主程序会先创建出一个管道，再 fork（）出两个子进程来分别执行管道符前后的命令，通过关闭子进程中的读口和写口来确定管道的传输顺序。顺序确定之后，我们借助封装好的 execute（）函数来执行命令：

```
int pipefd[2];
int ret = pipe(pipefd);
if(ret < 0) {
printf("pipe error!\n");
continue;
}
// 子进程 1
int pid = fork();
if(pid == 0) {
/*TODO:子进程 1 将标准输出重定向到管道，注意这里数组的下标被挖空了要补全*/
close(pipefd[READ_END]);
dup2(pipefd[WRITE_END], STDOUT_FILENO);
close(pipefd[WRITE_END]);
/*
在使用管道时，为了可以并发运行，所以内建命令也在子进程中运行
因此我们用了一个封装好的 execute 函数
*/
char *argv[MAX_CMD_ARG_NUM];
```

```
int argc = split_string(commands[0], " ", argv);
execute(argc, argv);
exit(255);
}
// 因为在 shell 的设计中，管道是并发执行的，所以我们不在每个子进程结束后才运行下一个
// 而是直接创建下一个子进程
// 子进程 2
pid = fork();
if(pid == 0) {
/* TODO:子进程 2 将标准输入重定向到管道，注意这里数组的下标被挖空了要补全 */
close(pipefd[WRITE_END]);
dup2(pipefd[READ_END], STDIN_FILENO);
close(pipefd[READ_END]);
```

```
char *argv[MAX_CMD_ARG_NUM];
/* TODO:处理参数，分出命令名和参数，并使用 execute 运行
* 在使用管道时，为了可以并发运行，所以内建命令也在子进程中运行
* 因此我们用了一个封装好的 execute 函数
*
*
*/
int argc = split_string(commands[1], " ", argv);
execute(argc, argv);
exit(255);
}
close(pipefd[WRITE_END]);
close(pipefd[READ_END]);
while (wait(NULL) > 0);
```

因为我选择完成的是重定向以及多命令的处理，所以有关多管道的相关函数我并没有填写，所以至此 main 函数的逻辑已经结束。

这里专门介绍一下重定向函数和内置指令实现的逻辑：

```
nt process_redirect(int argc, char** argv, int *fd) {
/* 默认输入输出到命令行，即输入 STDIN_FILENO，输出 STDOUT_FILENO */
fd[READ_END] = STDIN_FILENO;
fd[WRITE_END] = STDOUT_FILENO;
int i = 0, j = 0;
while(i < argc) {
int tfd;
if(strcmp(argv[i], ">") == 0) {
//TODO: 打开输出文件从头写入
tfd = open(argv[i+1], O_WRONLY | O_CREAT, 0666);
if(tfd < 0) {
printf("open '%s' error: %s\n", argv[i+1], strerror(errno));
} else {
```

```
//TODO: 输出重定向
fd[WRITE_END] = tfd;
}
i += 2;
} else if(strcmp(argv[i], ">>") == 0) {
//TODO: 打开输出文件追加写入
tfd = open(argv[i+1], O_WRONLY | O_CREAT | O_APPEND, 0666);
if(tfd < 0) {
printf("open '%s' error: %s\n", argv[i+1], strerror(errno));
} else {
//TODO:输出重定向
fd[WRITE_END] = tfd;
}
i += 2;
} else if(strcmp(argv[i], "<") == 0) {
//TODO: 读输入文件
tfd = open(argv[i+1], O_RDONLY, 0666);
if(tfd < 0) {
printf("open '%s' error: %s\n", argv[i+1], strerror(errno));
} else {
//TODO:输出重定向
fd[READ_END] = tfd;
}
i += 2;
} else {
argv[j++] = argv[i++];
}
}
argv[j] = NULL;
return j; // 新的 argc
}
```

对于不同的重定向类型“>”“>>”“<”，我们需要处理不同的文件打开类型。“>”直接覆写的文件给予 0666 的文件权限即都可以读写，flag 信号为写信号和创建信号。“>>”唯一的不同是 flag 里加入了附加写入的信号。而“<”则仅需要文件的读取信号。而循环的意义在于对于多个修改输入输出我们后面的重定向符会对前面的重定向位置进行覆写。

```
nt exec_builtin(int argc, char**argv, int *fd) {
if(argc == 0) {
return 0;
}
/* TODO: 添加和实现内置指令 */

if (strcmp(argv[0], "cd") == 0) {
if(argc == 1){
```

```
chdir("/home/auv");
return 0;
}
else if(argc == 2){
chdir(argv[1]);
return 0;
}
else{
printf("cd error\n");
return 1;
}
} else if (strcmp(argv[0], "exit") == 0){
if(argc == 2){
exit(atoi(argv[1]));
}
else{
exit(0);
}
return 0;
} else if(strcmp(argv[0], "kill") == 0){
if(argc == 2){
kill(atoi(argv[1]),15);
return 0;
}
else if(argc == 3){
kill(atoi(argv[1]),atoi(argv[2]));
return 0;
}
else{
return 1;
}
}
else {
// 不是内置指令时
return -1;
}
}
```

而内置命令的实现是依靠特殊处理每一个小指令，并通过判断指令的长度来进行不同的调用参数设置。主要依靠了 chdir、kill 和 exit 的调用。

Part2：“编写系统调用”

内核程序代码：

```
//new add
SYSCALL_DEFINE4(my_top,int __user *, pid,char __user *, name,int __user *, status, unsigned long long __user
* ,time){
```

```
struct task_struct* task;
int count = 0;
printk("[Syscall] my_top\n");
printk("[StuID] PB21111681\n");
for_each_process(task){
//copy_to_user(num, &counter, sizeof(int));

copy_to_user(pid + count, &task->pid, sizeof(int));
copy_to_user(name + 16 * count, &task->comm, sizeof(char) * 16);
copy_to_user(status + count, &task->state, sizeof(int));
copy_to_user(time + count, &task->se.sum_exec_runtime, sizeof(unsigned long long));
count++;
}
return 0;
}
```

主要是传出了 4 个参数，分别为用户态空间代码需要的 pid，进程名，运行状态和每一个进程目前运行的总时间。因为在用户态我们用数组的形式进行保存，所以我的思路是通过内核态的偏移量来实现用户态的对应位置存储。而进程名较为特殊的原因是进程名本身就是一个一维 char 数组，而我们无法定义 char ** 来传入，所以查询源码得知进程名的最大长度为 16，所以我采用 16 倍的偏移量来进行传输。

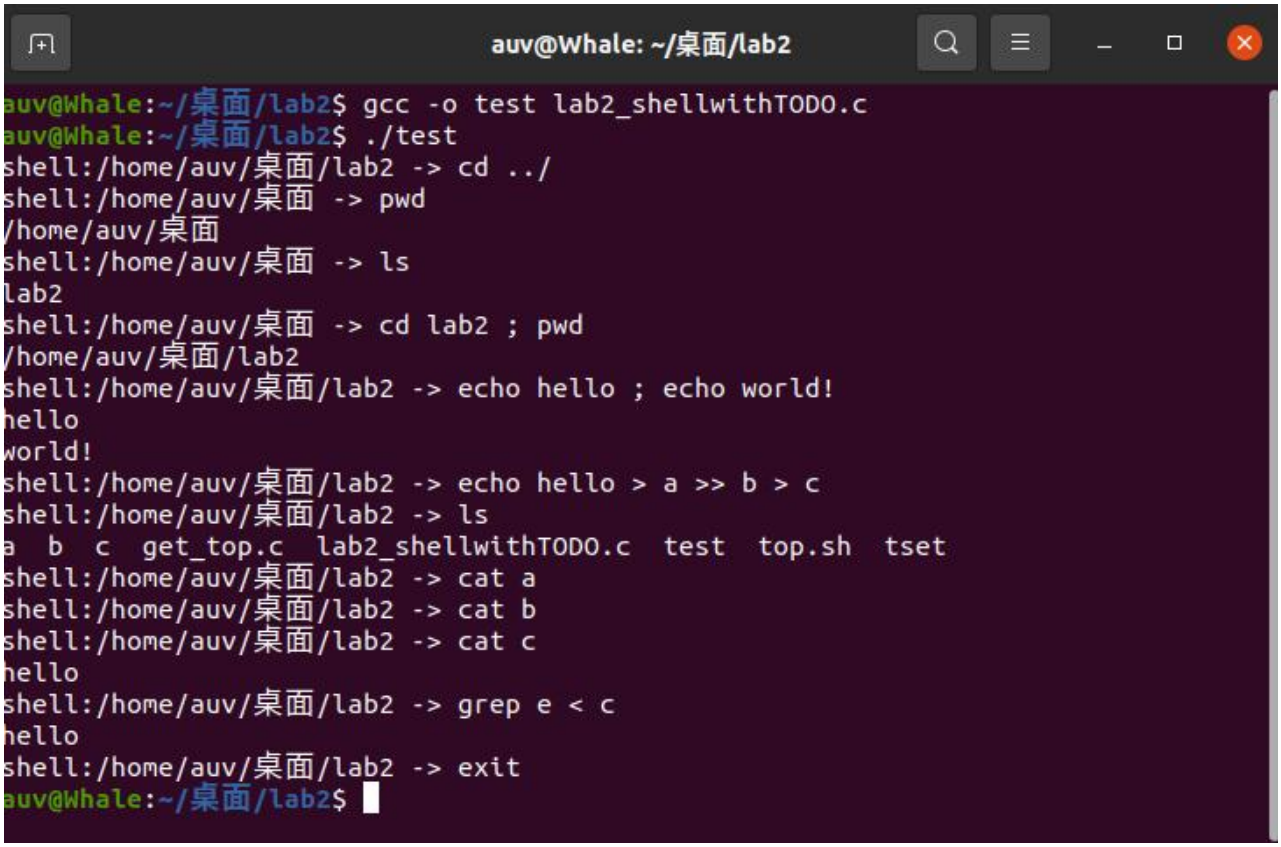
```
#include<unistd.h>
#include<stdio.h>
#include<sys/syscall.h>
#include<stdlib.h>
#define MAX 100
int main(){
int i,j,k,count=0,flag;
int pid1[MAX];
int pid2[MAX];
char name[MAX][16] ;
int status[MAX];
unsigned long long time1[MAX];
unsigned long long time2[MAX];
int index[MAX] = {0};
double time[MAX] = {-1};
syscall(333,pid1,name,status,time1);
while(1){
count = flag =0;
sleep(1);
system("clear");
syscall(333,pid2,name,status,time2);
printf("PID\tCOMM\tISRUNNING\t%%CPU\tTIME\n");
for(i=0; i<MAX; i++){
```

```
for(j=0; j<MAX; j++){
if(pid1[i] == pid2[j] && pid2[j] != 0){
time[j] = (time2[j] - time1[i])*1e-9;
index[count++] = j;
}
}
}
for(i=0;i<count-1;i++){
for(j=i;j<count;j++){
if(time[index[i]] - time[index[j]] < 1e-6){
int temp2 = index[i];
index[i] = index[j];
index[j] = temp2;
}
}
}
for(i=0;i<19;i++){
printf("%d\t%s \t%d\t%lf\t%lf\n",pid2[index[i]],
name[index[i]],
(status[index[i]]==0),
time[index[i]]*100,
time2[index[i]]*1e-9);
}
for(i=0;i<MAX;i++){
pid1[i] = pid2[i];
time1[i] = time2[i];
}
}
return 0;
}
```

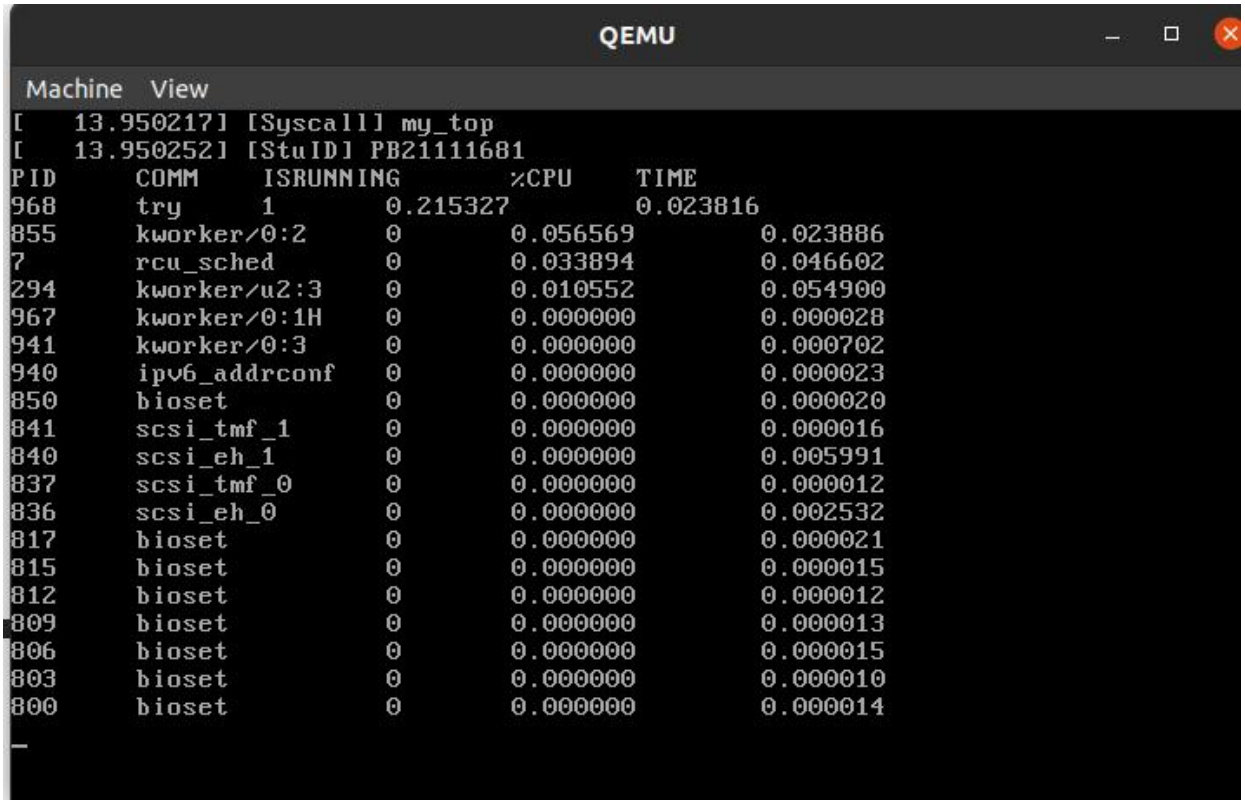
用户态代码中，主要处理的是两次读出数据的差别。主要差别存在于 pid 和运行时间。我的思路是将两次传出的 pid 进行比对，如果发现了有新建立的进程或是已经结束的老进程，我们不会对这些进程的运行时间进行处理，当然也不会打印出结果。然后将两次都存在的进程运行时间相减，得到的结果进行单位换算之后便得到了 CPU 的占用率（因为中间时间为 1s）。然后再对这些时间进行冒泡排序，把较大的结果排到更靠前的位置上，最后在输出时选择前 20 位进行输出打印。然后重复上述过程，并没有考虑进程该如何关闭。

实验结果：

如下为 shell 的运行和检测结果：



如下为 my_top 的运行结果：



实验总结：
本次实验算是操作系统的入门级别实验，在完成这个实验的过程中使用了课堂中学习到的知识，巩固了我对操作系统中有关系统调用的相关知识的理解，同时通过实操和添加系统调用我开始明白课本上的相关知识不仅仅是空话和无意义的思考，而是会真正出现错误的误区点。总之这次实验在一定程度上让我客服了恐惧心理，窥见了 Linux 操作系统的冰山一角。