Os 实验二 添加Linux系统调用

总述

本次实验主要关注Linux系统调用的使用及实现这两部分内容。

为熟悉Linux系统调用的使用，我们在已有完善已有架构，搭建了一个简单的shell。我们实现了当前所在的目录的显示，单条命令的运行，含管道符命令或“；”的多条命令运行，同时我们也实现了exit, cd, kill三个shell内置指令和含有重定向符命令的运行。

此后我们实现了自己的Linux系统调用mytop。我们在内核中添加了系统调用的实现函数，这个函数将被用户空间的程序调用。随后，我们将系统调用添加到了系统调用表中，以便用户空间的程序可以通过系统调用号来调用相应的函数。最后，我们需要重新编译内核并重启系统，在qemu中实现了调用了mytop的top程序。

总而言之，本次实验是一个较为综合的实验，在过程中我们学习并熟练掌握了系统调用的相关内容，深入理解了在Linux源码中添加自己的系统调用的原理和实现方法，此外，通过本次实验，我们得以从一个较为直观的视角了解用户空间是如何通过系统调用与内核空间相连结的。

实验目的

- 学习如何使用Linux系统调用：实现一个简单的shell
- 学习如何添加Linux系统调用：实现一个简单的top

实验环境

- 操作系统：Ubuntu 20.04 LTS
- Linux内核版本: 4.9.263
- 开发环境：Vscode、GCC编译器
- 开发语言：C语言

实验内容

实验一 实现一个Linux Shell

实验过程

1.实现打印命令提示符（类似shell ->）：

主要思路：利用getcwd系统调用，将当前工作目录的绝对路径复制到参数buf所指的内存空间中,参数size为buf的空间大小。

```
char * cwd_now = getcwd(buf,MAX_BUF_SIZE);
printf("shell:%s -> ",cwd_now);
```

2.把分隔符 ; 连接的各条命令分割开：

主要思路：利用split_string()函数分割各条指令并存储，逐步执行各条命令。

```
split_string(commands[0], ";", commandsa)；
```

注意事项：这里应该关注“；”处理程序的位置，因为原有架构中已经有根据“|”进行了分段，我们需要考虑当“；”和“|”同时出现时二者的优先级。（理论上应该先处理；，再接着处理|，但由于实验架构已经给定了，为了不做太大变动，我在实际实现中将”；“和”|“分开处理。）

3.处理含管道符的单条命令：

主要思路：创建管道，重定向子进程1的标准输出到，fork另一一个子进程，将子进程2的标准输入重定向到管道读端。

```
int pipefd[2];
int ret = pipe(pipefd);
if(ret < 0) {
                printf("pipe error!\n");
                free(buf);
                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]);//关闭读端
                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]);
```

注意事项：

1）各命令并发执行，整体不是顺序执行的，为了让多个内置指令可以并发，需要在fork出子进程后才执行内置指令

2）在管道重定向的过程中无论是否关闭已经重定向了对应的管道，程序均可以正常运行。

4.实现多管道命令：

主要思路：每次将子进程的标准输出重定向到管道写端，父进程保存对应管道的读端（上一个子进程向管道写入的内容），并使得下一个进程的标准输入重定向到保存的读端，直到最后一个进程使用标准输出将结果打印到终端。

注意事项：

1）n个进程只需创建n-1个管道

2）注意保存上一个管道的读端口int read_fd

3）需要关注父子进程管道关闭的对应时间，由于调用关系，需要在所有管道均建立完成后再逐一关闭相应的管道。

```
if( i != 0){
                    close(read_fd);
                }
```

```
                if( i != (cmd_count -1)){
                    read_fd = pipefd[READ_END];
                    close(pipefd[WRITE_END]);
                }
```

5.实现命令的执行

主要思路：命令可以分为内置命令和外部命令，对于外部命令可以直接使用execvp（）函数执行，但内部命令需要自行实现。

注意事项：exec有很多种不同的实现，现将其罗列如下，应根据需要的功能对应选择。

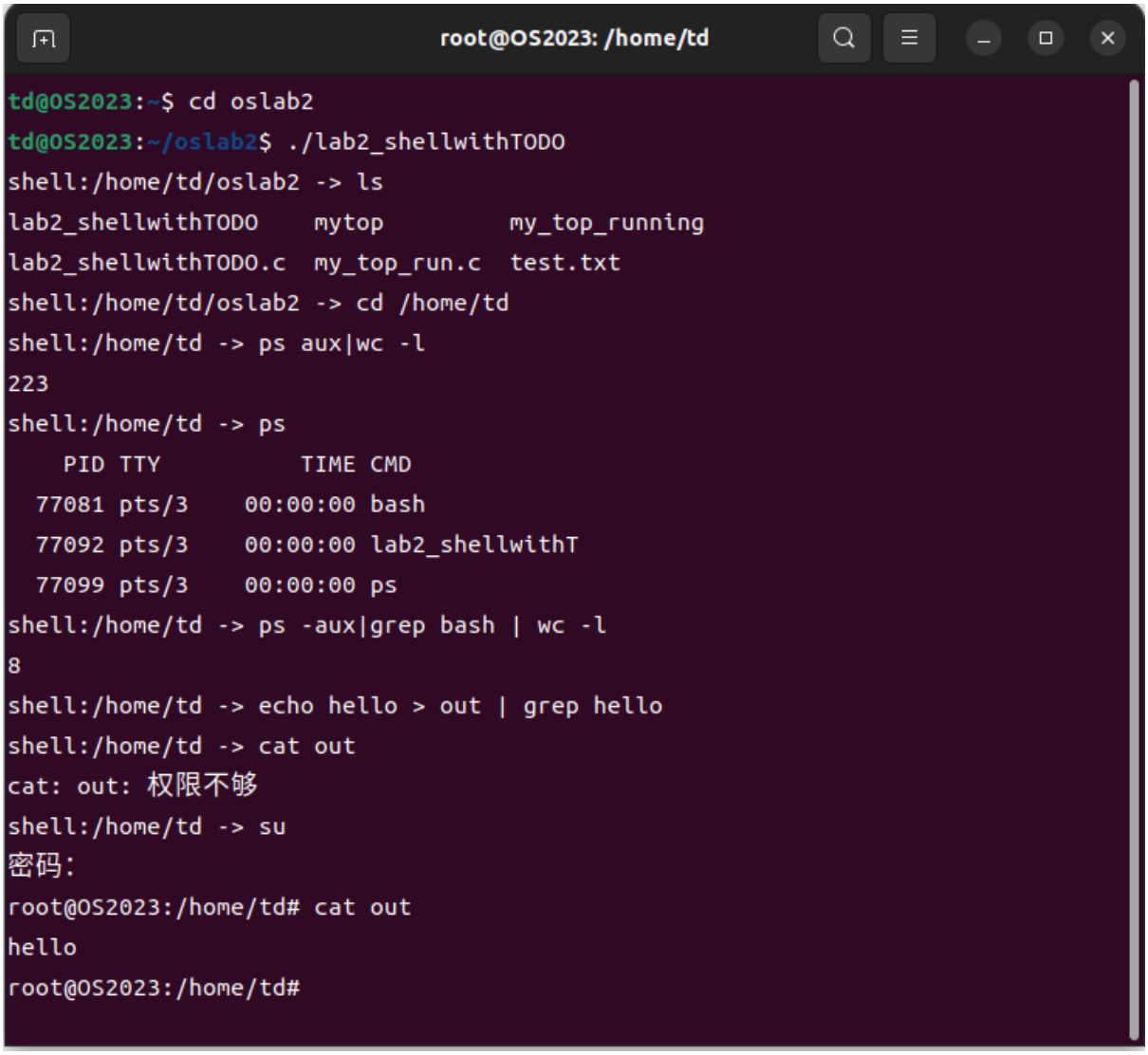
函数族：

```
#include <unistd.h>
extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg,..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],char *const envp[]);
```

实验结果

部分命令展示：



实验二 编写系统调用实现一个Linux top

实验过程

1.注册系统调用：

主要过程：我们只需要修改 syscall_64.tbl即可。（在编译时，脚本 arch/x86/entry/syscalls/syscalltbl.sh 会被运行，将 syscall_64.tbl 文件中登记过的系统调用都生成到syscalls_64.h 文件中）

```
555 common  trans_info    sys_trans_info
```

2.声明内核函数原型

主要思路：利用split_string()函数分割各条指令并存储，逐步执行各条命令。打开 linux-4.9.263/include/linux/syscalls.h ，里面是对于系统调用函数原型的定义，在最后面加上我们创建的新的系统调用函数原型，格式为 asmlinkage long sys_xxx(...) 。注意，如果传入了用户空间的地址，需要加入 __user 宏来说明。

```
// my_define
asmlinkage long sys_trans_info(int __user * num, void __user * info);
```

注意事项：这里有位需要注意，zhe'shi一个.h文件，需要通过”//“来进行注释，最开始用成”#“了，虽然不会影响程序的运行，但也值得注意。附：在上一个.tbl文件中是可以使用”#“进行注释的。

3.实现内核函数

主要思路：利用struct task_struct将更进程信息传出内核。过程中应注意内核空间较小，所以应当将内容传出内核再进行处理。这里重点要熟悉struct task_struct结构的掌握。

这里我将die值默认设为零，以和上层程序作用，达到显示更新的效果。

```
SYSCALL_DEFINE2(trans_info, int __user *, num, void __user *, info)
{
    struct task_struct *task;
    int counter = 0;
    struct topinfo
    {
        pid_t pid;
        char comm[TASK_COMM_LEN];
        volatile long state;
        u64 sum_exec_runtime;
        int die;
    };
    struct topinfo tinfo[128];
    printk("[Syscall] ] trans_info\n");
    printk("[StuID] PB21061287\n");
    for_each_process(task)
    {
        tinfo[counter].pid = task->pid;
        tinfo[counter].state = task->state;
        memcpy(tinfo[counter].comm, task->comm, TASK_COMM_LEN);
        tinfo[counter].sum_exec_runtime = task->se.sum_exec_runtime;// CPU运行时间
        tinfo[counter].die = 0;
        counter++;
    }
    copy_to_user(num, &counter, sizeof(int));
    copy_to_user(info, &tinfo, 128*sizeof(struct topinfo));
    return 0;
}
```

注意事项：

1. **当再次编译的时候，应当注意使用make clean命令，以保证编译为最新的文件。**
2. **这里的task引用需要用”→”而不能用“.“**

5.编写测试代码

主要思路：

1）利用clear和sleep函数进行刷新

2）建立输出需要的结构体和临时结构体数组，从而可以灵活存储多次出现的进程。如果只使用一个结构体数组难以同时实现上一次进程信息的记录和下一次信息的传入（系统调用需要直接传入void *类型指针）。

3）CPU计算：可以利用前后两次运行时间差来表示，但是应注意，上一次的时间应当被存储下来，不可以直接计算两次之差。

4）**排序算法：这里为了节省时间，我们只对前20个进程进行排序，其余不做处理。排序使用单独的数组记录下标。**

5.输出：注意对齐

```
struct topinfo
{
    pid_t pid;
    char comm[TASK_COMM_LEN];
    volatile long state;
    u64 sum_exec_runtime;
    int die ;//判断该行数据是否有效 0：有效/1：无效
};
```

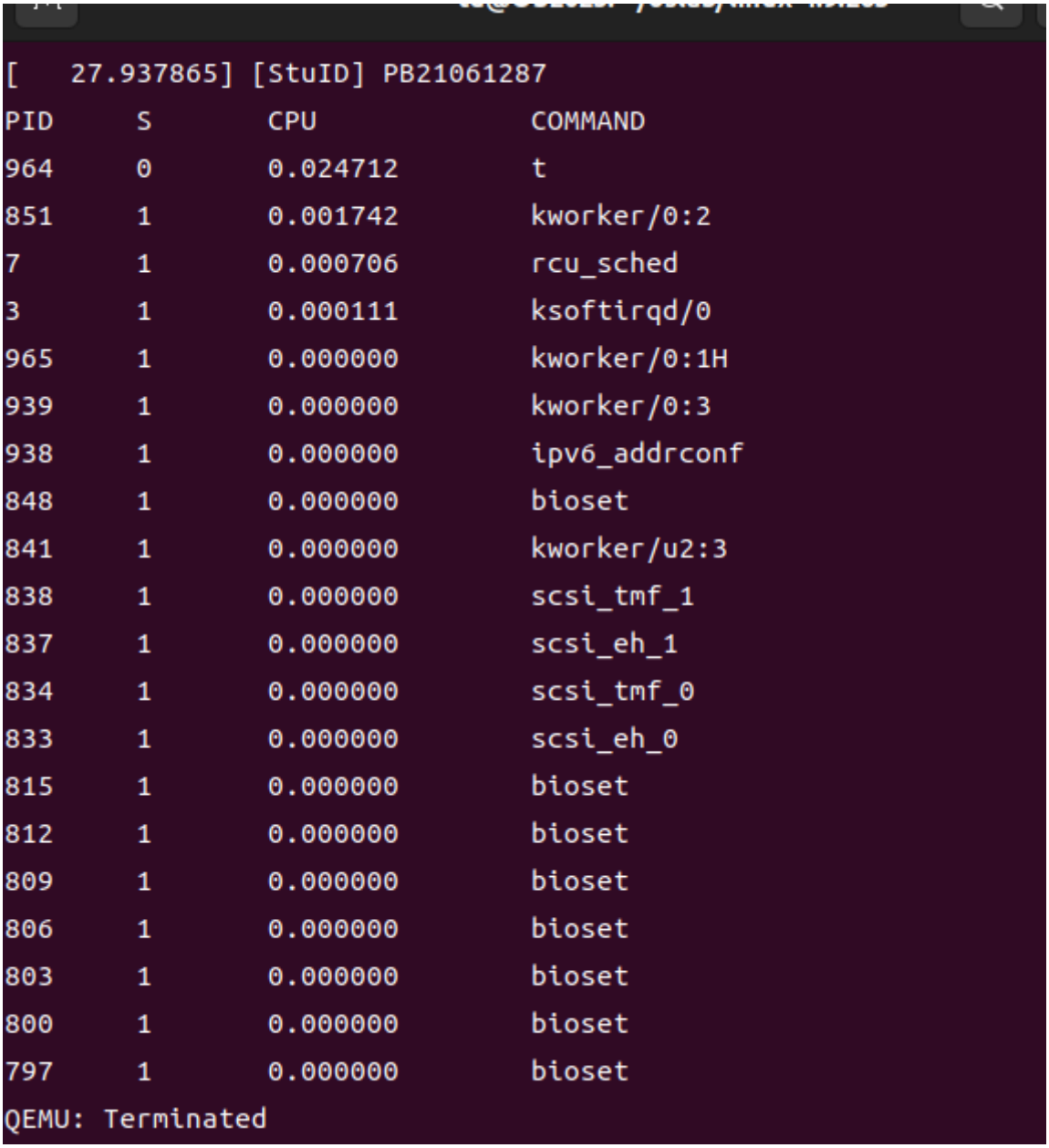
注意事项：

1）这里我选择引入die变量来表征进程是否再次出现

2）因为并不好实现将结构体数组指针传入内核，这里可以利用void *指针，再在用户空间对其作类型转换

3）尽可能以简单的结构来实现，不要为了节省空间大幅增加代码难度。

实验结果



实验总结

本次实验中，我们学习了操做系统调用，并实现了myshell和mytop。通过实验，我们深入理解了系统调用的原理和实现方法，并掌握了一些调试技巧。

此外，我发现自己在代码的编写过程中不够细致，以后尤其应当注意在没有高亮环境中代码的编写。比如在实验过程中，我误将common写成了commom，导致编译始终不通过。以及init文件中少复制了一个h找了三个小时的bug。

总体而言，本次实验收获颇丰，为我们今后的学习和工作打下了坚实的基础。

一点小建议：

1.可以增加一部分VsCode调试的配置，尤其是在shell的调试中使用VsCode DBG的图像化界面会方便很多，此外可以不用人为编译代码，对于需要部分修改的代码会方便很多。

2.在声明内核函数原型如果可以对代码内容加一点点解释就好了，虽然知道如何配置了，但是并不理解代码的含义。

3.使用一个数组实现时应格外注意，考虑数据覆盖的情况。（代码中踩过的坑）