OS lab 2 添加 Linux 系统调用

PB21081601 张芷苒

一 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

二 实验环境

- 安装在 Windows 的 Virtual Box
- OS: Ubuntu 20.04.4 LTS
- Linux内核版本: 4.9.263

三 实验内容

1. 实现一个Shell

> 要求：
>
> - 支持基本的单条命令运行、支持两条命令间的管道 `|` 、内建命令（只要求 `cd / exit / kill` ）
> - 选做：
> - 支持多条命令间的管道 `|` 操作
> - 支持重定向符 `>` 、 `>>` 、 `<` 和分号 `;`

2. 实现一个 top

> - 在linux4.9下创建适当的（可以是一个或多个）系统调用。利用新实现的系统调用，实现一个linux4.9下的进程状态信息统计程序 。
> - 输出的信息需要包括：
>   - 进程的PID
>   - 进程的COMMAND（进程名）
>   - 进程是否处于running状态
>   - 进程的CPU占用率
>   - 进程的总运行时间（单位秒）
> - 默认情况下每一秒刷新一次信息。输出的信息需要按CPU占用率排序。输出前20行即可，无需输出在两次刷新间隔之间新建/消失的进程。程序需要一直运行，无需考虑你写的 top 程序如何关闭。
> - 个性化定制输出，支持参数-d，每隔多少秒刷新。
> - 添加的系统调用在被调用时需要 printk 出自己的系统调用名和学号

四 实验过程/结果

1. 实现一个Shell

下面按照文档中描述的shell工作流程依次解释各部分代码，详细过程写在了注释里。

第一步：打印命令提示符（类似shell ->）。

```
/* 打印当前目录 */
char path_name[51];  // 存储当前路径的字符数组
getcwd(path_name, PATH_SIZE);  // 使用getcwd()函数获取当前目录路径
printf("shell: %s -> ", path_name);  // 打印当前目录路径
fflush(stdout);  // 刷新输出缓冲区，确保打印输出到终端

/* 读取用户输入 */
fgets(cmdline, 256, stdin);  // 从标准输入读取用户输入的命令行
strtok(cmdline, "\n");  // 去掉命令行末尾的换行符
```

第二步：把分隔符 ; 连接的各条命令分割开。（多命令选做内容）

```
/* 基于";"的多命令执行，请自行选择位置添加 */

int multi_cmd_num = split_string(cmdline, ";", multi_cmd);
// 使用分号分割命令行字符串，将多条命令存储到multi_cmd数组中，并返回多条命令的数量

for (int i = 0; i < multi_cmd_num; i++)  // 遍历多条命令
{
    strcpy(cmdline, multi_cmd[i]);  // 将每条命令存储到cmdline数组中

    // 此处为执行每条命令的代码
}
```

第三步：对于单条命令，把管道符 | 连接的各部分分割开。

```
cmd_count = split_string(cmdline, "|", commands);  // 使用竖线分割命令行字符串，将多个命令存储到commands数组中，并返回命令的数量
```

第四步：如果命令为单一命令没有管道，先根据命令设置标准输入和标准输出的重定向（重定向选做内容）；再检查是否是shell内置指令：是则处理内置指令后进入下一个循环；如果不是，则fork出一个子进程，然后在fork出的子进程中exec运行命令，等待运行结束。

第五步：如果只有一个管道，创建一个管道，并将子进程1的标准输出重定向到管道写端，然后fork出一个子进程，根据命令重新设置标准输入和标准输出的重定向（重定向选做内容），在子进程中先检查是否为内置指令，是则处理内置指令，否则exec运行命令；子进程2的标准输入重定向到管道读端，同子进程1的运行思路。

```
if (cmd_count == 0) {
    continue;  // 如果没有命令，则继续下一次循环
}
else if (cmd_count == 1) {  // 没有管道的单一命令
    char *argv[MAX_CMD_ARG_NUM];  // 保存命令行参数的数组
    int argc;
    int fd[2];  // 用于管道通信的文件描述符数组

    /* TODO: 处理参数，分出命令名和参数 */

    argc = split_string(cmdline, " ", argv);  // 使用空格分割命令行字符串，将参数存储到argv数组中，并返回参数的数量

    /* 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 */
    if (exec_builtin(argc, argv, fd) == 0) {
        continue;  // 如果是内建命令，则在主进程中完成并继续下一次循环
    }

    /* TODO: 创建子进程，运行命令，等待命令运行结束 */

    pid_t pid = fork();  // 创建子进程
    if (pid == 0) {  // 子进程
        if (execute(argc, argv) < 0) {  // 执行命令，如果出错则打印错误信息并退出
            printf("Command not found.\n", argv[0]);
```

```
            exit(0);
        }
    }
    while (wait(NULL) > 0);  // 等待子进程结束，并回收资源
}
// 两个命令之间有管道
else if (cmd_count == 2) {
    int pipefd[2];
    int ret = pipe(pipefd);
    if (ret < 0) {
        printf("pipe error!\n");
        continue;
    }
    // 子进程1
    pid_t pid = fork();
    if (pid == 0) {
        // 将子进程1的标准输出重定向到管道
        close(pipefd[READ_END]);
        dup2(pipefd[WRITE_END], STDOUT_FILENO);
        close(pipefd[WRITE_END]);
        // 将命令行拆分成命令名和参数，并使用execute函数运行命令
        char *argv[MAX_CMD_ARG_NUM];
        int argc = split_string(commands[0], " ", argv);
        execute(argc, argv);
        exit(255);
    }
    // 子进程2
    pid = fork();
    if (pid == 0) {
        // 将子进程2的标准输入重定向到管道
        close(pipefd[WRITE_END]);
        dup2(pipefd[READ_END], STDIN_FILENO);
        close(pipefd[READ_END]);
        // 将命令行拆分成命令名和参数，并使用execute函数运行命令
        char *argv[MAX_CMD_ARG_NUM];
        int argc = split_string(commands[1], " ", argv);
        execute(argc, argv);
        exit(255);
    }
    close(pipefd[WRITE_END]);
    close(pipefd[READ_END]);
    // 父进程等待子进程结束
    while (wait(NULL) > 0);
}
```

第六步：如果有多个管道，参考第三步，n个进程创建n-1个管道，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。（多管道选做内容）

第七步：根据第二步结果确定是否有剩余命令未执行，如果有，返回第三步执行（多命令选做内容）；否则进入下一步。

```
else {    // 选做：三个以上的命令
    int read_fd;    // 上一个管道的读端口（出口）
    for(int i=0; i<cmd_count; i++) {
        int pipefd[2];
        // 创建管道，n条命令只需要n-1个管道，所以有一次循环中是不用创建管道的
        if(i != cmd_count - 1) {
            int ret = pipe(pipefd);
            if(ret < 0) {
                printf("Pipe error!\n");
                continue;
            }
        }
        int pid = fork();
        if(pid == 0) {
            // 除了最后一条命令外，都将标准输出重定向到当前管道入口
            if(i != cmd_count - 1) {
                close(pipefd[0]);
                dup2(pipefd[1], STDOUT_FILENO);
                close(pipefd[1]);
            }
            // 除了第一条命令外，都将标准输入重定向到上一个管道出口
            if(i != 0) {
                close(pipefd[1]);
                dup2(read_fd, STDIN_FILENO);
                close(read_fd);
                if(i == cmd_count - 1)
                    close(pipefd[0]);
            }
            // 处理参数，分出命令名和参数，并使用execute运行
            char *argv[MAX_CMD_ARG_NUM];
            int argc = split_string(commands[i], " ", argv);
            execute(argc, argv);
            exit(255);
        }
        // 父进程除了第一条命令，都需要关闭当前命令用完的上一个管道读端口
        // 父进程除了最后一条命令，都需要保存当前命令的管道读端口
        // 记得关闭父进程没用的管道写端口
        if(i != 0) close(read_fd);
        if(i != cmd_count - 1) read_fd = pipefd[0];
        close(pipefd[1]);
        // 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
    }
    // 等待所有子进程结束
    while(wait(NULL) > 0);
}
```

第八步：打印新的命令提示符，进入下一轮循环。

2. 实现一个 top

用户空间代码: `get_num_ps.c`，具体实现过程写在了注释里。这段代码是获取并显示当前系统中运行的进程信息，包括进程的PID、名称、是否正在运行、执行时间和CPU占用率等信息，并按照CPU占用率从高到低排序，最后输出前20个占用CPU最高的进程的信息。其中使用了系统调用 `syscall` 来获取进程信息。

```
#include<stdio.h>
#include<unistd.h>
#include<sys/syscall.h>
#include<string.h>

#define max_len 500

struct task_exist{
    int PID;
    char command[16];
    int running;
    double time;
    double CPU;
};

int main(void)
{
    int sum1, sum2;                             // 进程数量
    int PID1[max_len], PID2[max_len];           // 进程的PID
    char COMMAND1[max_len][16], COMMAND2[max_len][16];   // 进程的名称
    long running1[max_len], running2[max_len];  // 进程是否正在运行
    unsigned long long time1[max_len], time2[max_len]; // 进程执行时间
    struct task_exist task[max_len];            // 存储进程信息的结构体
    int i, j;
    int k = 0;

    syscall(332, &sum1, PID1, COMMAND1, running1, time1); // 获取上一次的进程信息

    while (1) {
        sleep(1);
        system("clear");
        syscall(332, &sum2, PID2, COMMAND2, running2, time2); // 获取当前的进程信息

        k = 0;
        // 找到上一次和当前都存在的进程，将其信息存储到结构体中
        for (i = 0; i < sum2; i++) {
            for (j = 0; j < sum1; j++) {
                if (PID2[i] == PID1[j]) {
                    task[k].PID = PID2[i];
                    task[k].running = (int)running2[i] ? 0 : 1;
                    task[k].time = time2[i] / 1000000000.0;
                    task[k].CPU = (time2[i] - time1[j]) / 10000000.0;
                    strcpy(task[k].command, COMMAND2[i]);
                    k++;
                    break;
                }
```

```
            }
        }

        // 根据CPU占用率排序
        int max_num;
        double max;
        int flag[max_len]; // 标记某个进程是否已经被打印过
        for (i = 0; i < max_len; i++)
            flag[i] = 1;
        printf("%-5s%-16s%-10s%-10s%-10s\n", "PID", "COMM", "ISRUNNING", "%CPU","TIME");
        // 找到占用CPU最高的前20个进程
        for (i = 0; i < 20 && i < k; i++) {
            for (j = 0; j < k; j++) {
                if (flag[j]) {
                    max = task[j].CPU;
                    max_num = j;
                    break;
                }
            }
            for (; j < k; j++) {
                if (task[j].CPU > max && flag[j]) {
                    max = task[j].CPU;
                    max_num = j;
                }
            }
            printf("%-5d%-16s%-10d%-10.2lf%-10.2lf\n", task[max_num].PID,
task[max_num].command, task[max_num].running, task[max_num].CPU, task[max_num].time);
            flag[max_num] = 0;
        }

        // 更新上一次的进程信息为当前进程信息
        sum1 = sum2;
        for (i = 0; i < sum2; i++) {
            PID1[i] = PID2[i];
            strcpy(COMMAND1[i], COMMAND2[i]);
            running1[i] = running2[i];
            time1[i] = time2[i];
        }
    }
    return 0;
}
```

修改过的kernel源码：

`sys.c` 是一个 Linux 系统调用，用于获取进程信息，包括进程的 PID、名称、执行时间、旧的执行时间、状态等。其中 `ps_counter` 用于获取系统中运行的进程数量， `ps_info` 用于获取进程的详细信息。

```
SYSCALL_DEFINE1(ps_counter, int __user *, num) {
    struct task_struct* task;
    int counter = 0, err;
    printk("[Syscall] ps_counter\n");
    for_each_process(task) {
        counter ++;
```

```
    }
    err = copy_to_user(num, &counter, sizeof(int));
    return 0;
}

SYSCALL_DEFINE5(ps_info, pid_t * __user *, user_pid, char* __user * , user_name, unsigned
long long * __user *, user_time, unsigned long long * __user *, user_old_time,  long *
__user *, user_state) {
    struct task_struct* task;
    int i = 0, j = 0, k = 0, cnt = 0, err = 0;
    char name_a[1024];
    pid_t pid_a[128];
    unsigned long long old_time, cpu_time;
    for(k = 0; k < 1024; k++) name_a[k] = ' ';

    printk("[StuID] pb21081601\n");
    printk("[Syscall] ps_info\n");

    for_each_process(task) {

        // err = copy_to_user(user_pid + i, &(task -> pid) , sizeof(pid_t)); // This line
has unknown bug. You may try and find out why.
        pid_a[i] = task -> pid;
        err = copy_from_user(&(old_time), user_old_time + i, sizeof(unsigned long long));
        cpu_time = task -> se.sum_exec_runtime - old_time;
        //Pass the data one by one to save the stack space.
        err = copy_to_user(user_time + i, &(cpu_time), sizeof(unsigned long long));
        err = copy_to_user(user_state + i, &(task -> state), sizeof(long));
        //Use space as delimiter to store process names in a char array.
        for(j = 0; j < 16; j++) {
            if(task -> comm[j] != ' ' && task -> comm[j] != '\0') {
                name_a[cnt + j] = task -> comm[j];
            }
            else {
                name_a[cnt + j] = ' ';
                cnt += j + 1;
                break;
            }
        }

        i++;
    }

    err = copy_to_user(user_name, name_a, sizeof(name_a));
    err = copy_to_user(user_pid, pid_a, sizeof(pid_a));

    return 0;
}
```

`syscalls.h` 是一个头文件，定义了两个 Linux 系统调用的函数原型，分别是 `sys_ps_counter` 和 `sys_ps_info`，它们分别对应了上一个代码段中的两个系统调用。

```
asmlinkage long sys_ps_counter(int __user * num);

asmlinkage long sys_ps_info(pid_t* __user * user_pid, char* __user * user_name, unsigned long long * __user * user_time, unsigned long long * __user * old_user_time, long * __user * user_state);
```

`syscall_64.tbl` 是一个系统调用表文件，定义了两个新的系统调用`ps_counter`和`ps_info`，它们的系统调用号分别为332和333，并指定了对应的函数名称`sys_ps_counter_pb21081601`和`sys_ps_info_pb21081601`。

```
332 common  ps_counter      sys_ps_counter_pb21081601
333 common  ps_info         sys_ps_info_pb21081601
```

总体来说：

`get_num_ps.c`中使用了系统调用`syscall`来获取进程信息，而`sys.c`中实现了两个系统调用`ps_counter`和`ps_info`，并使用`SYSCALL_DEFINE1`和`SYSCALL_DEFINE5`分别将它们与系统调用号关联起来。而`syscall.tbl`文件中则定义了这两个系统调用的系统调用号，并将它们与对应的函数名关联起来。最后，`syscalls.h`文件中定义了这两个系统调用的函数原型。

3. 实验结果

按照要求编译、运行后，基本可以满足实验文档的要求，具体展示在检查实验时进行。

五 实验总结

- 本次实验难度较大，操作性较强，好在实验文档比较详细，只要堆砌足够多的时间就可以解决大部分问题。