OSlab2 实验报告

牛庆源 PB21111733

实验目的

用 shell 实现一个系统调用

实验环境

VMware

实验内容

第二部分

1. TODO：添加和实现内置指令

```
if (strcmp(argv[0], "cd") == 0) {                       // 实现 cd 指令
        char buf[MAX_BUF_SIZE + 1];                     // 指令
    memset(buf, 0, MAX_BUF_SIZE + 1);                   // 初始化
        if (argv[1][0] != '.' && argv[1][0] != '/')     // 处理特殊
        {
            getcwd(buf, MAX_BUF_SIZE);
            strncat(buf, "/", MAX_BUF_SIZE - strlen(buf));
    }
        strncat(buf, argv[1], MAX_BUF_SIZE - strlen(buf));   // 拼接
        chdir(buf);                                          // 跳转
        return 0;
    } else if (strcmp(argv[0], "exit") == 0){           // 实现 exit 指令
        exit(0);
    // kill 直接采用内置指令了
    } else {
        // 不是内置指令时
        return -1;
}
```

2. TODO：execute 的运行命令与结束

```
execvp(argv[0] ,argv);              // 运行
exit(0);                            // 结束
```

3. TODO：打印当前目录

```
char buf[MAX_BUF_SIZE + 1];             // 目录
memset(buf, 0, MAX_BUF_SIZE + 1);       // 初始化
getcwd(buf , MAX_BUF_SIZE);             // 取指令
printf("shell:%s-> ", buf);             // 打印
```

4. TODO：单一命令

```
// TODO:处理参数，分出命令名和参数
            argc = split_string(cmdline, " ", argv);            // 分隔
//  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) {             // 没有管道
                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);
                exit(255);
            }

            while (wait(NULL) > 0);
```

5. TODO：标准输出重新定向，标准输入重新定向

```
close(pipefd[READ_END]);
dup2(pipefd[WRITE_END], STDOUT_FILENO);
close(pipefd[WRITE_END]);

close(pipefd[WRITE_END]);
dup2(pipefd[READ_END], STDIN_FILENO);
close(pipefd[READ_END]);
//  分出命令名和参数  运行
int argc = split_string(commands[1], " ", argv);
execute(argc, argv);
exit(255);
```

6. TODO：三个以上的命令

```
// 建 n-1 条管道，特判没有管道输入的第一条命令和没有管道输出的最后一条命令，运行
if (i != cmd_count - 1)
{
    int ret = pipe(pipefd);
    if(ret < 0) {
        printf("pipe error!\n");
        continue;
    }
}
int pid = fork();
if(pid == 0) {
// TODO: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[READ_END]);
        dup2(pipefd[WRITE_END], STDOUT_FILENO);
        close(pipefd[WRITE_END]);
    }
// TODO: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[WRITE_END]);
        dup2(read_fd, STDIN_FILENO);
        close(read_fd);
    }
// TODO: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);


// TODO:等待所有子进程结束
while (wait(NULL) > 0);
```

测试

```
cd ..
cd oslab
kill -9 pid
ps aux | wc -l
ps aux | grep qyniu | wc -l
```

```
qyniu@ubuntu:~/code$ ./lab2_sh
shell:/home/qyniu/code-> cd ..
shell:/home/qyniu-> cd oslab
shell:/home/qyniu/oslab-> ps
    PID TTY          TIME CMD
  19216 pts/0    00:00:00 bash
  19225 pts/0    00:00:00 lab2_sh
  19226 pts/0    00:00:00 ps
shell:/home/qyniu/oslab-> kill -9 19225
已杀死
qyniu@ubuntu:~/code$ ./lab2_sh
shell:/home/qyniu/code-> ps aux | wc -l
292
shell:/home/qyniu/code-> ps aux | grep qyniu | wc -l
74
shell:/home/qyniu/code->
```

第三部分

注册系统调用

333 common ps_info sys_ps_info

声明内核函数原型

asmlinkage long sys_ps_info(int __user * num,char __user * comm,int __user * isrun,long long __user * stime);

实现内核函数

```
SYSCALL_DEFINE4(ps_info, int __user *, num, char __user *, comm, int __user *, isrun, long long __user *, stime)
{
    struct task_struct* task;
    long long task_time[100];          // 任务开始时刻
    int pids[100];                     // pid
    int isruns[100];                   // 运行情况
    char comms[1600];                  // 指令
    int counter = 0;
```

```
    printk("[Syscall] ps_info\n[StuID] PB21111733\n");          // 在内核内打印
    int i;
    for (i = 0; i < 100;i++)  // 初始化
    {
        task_time[i] = 0;
        pids[i] = 0;
        isruns[i] = 0;
    }
    for (i = 0;i < 1600;i++) // 初始化
    {
        comms[i] = ' ';
    }
    for_each_process(task)
    {    // 内核态向用户态输信息
        task_time[counter] = task->se.sum_exec_runtime;
        pids[counter] = task->pid;
        int j;
        for (j = 0;j < 16 && j < strlen(task->comm);j++)
        {    // 处理指令
            comms[counter*16+j] = task->comm[j];
        }
        if (task->state == 0) isruns[counter] = 1; // 判断运行
        else isruns[counter] = 0;
        counter++;
    }
    // copy to user
    copy_to_user(num, pids, 100*sizeof(int));
    copy_to_user(stime, task_time, 100*sizeof(long));
    copy_to_user(isrun, isruns, 100*sizeof(int));
    copy_to_user(comm, comms,1600*sizeof(char));
    return 0;
}
```

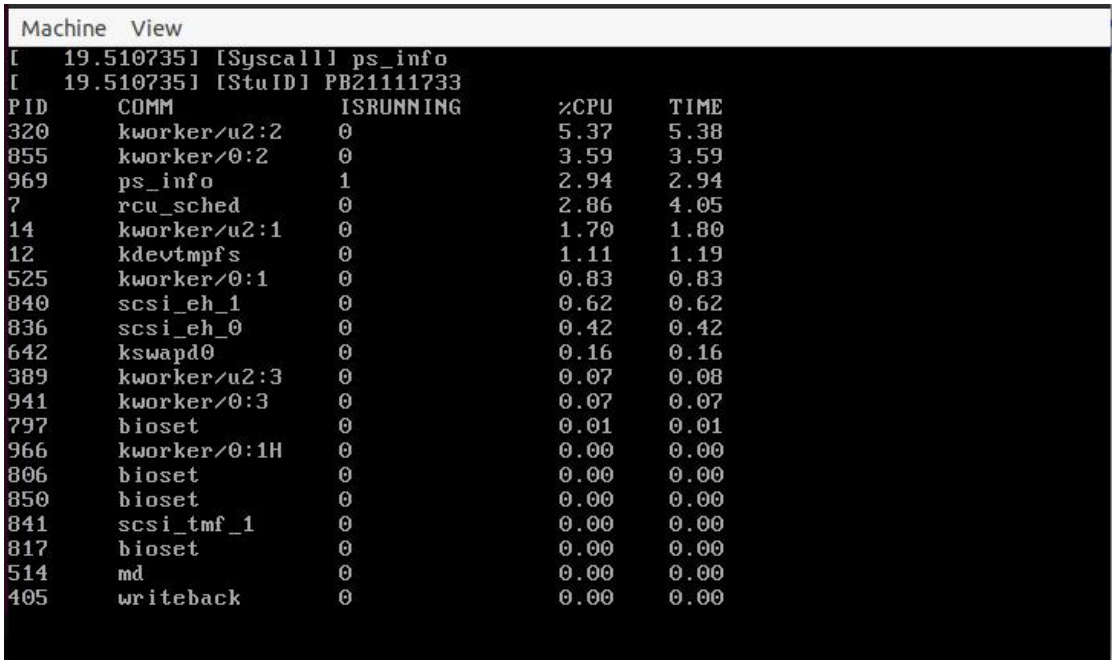
测试代码

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

int main(void)
{
```

```
int pid_ls[100];
long long lstime[100];  // 上一任务的开始时刻
long long stime[100];   // 当前任务的开始时刻
char comm[1600];        // 指令
int isrun[100];         // 运行与否
double cpuo[100];       // cpu 占用率
double nstime[100];     // 持续时间
int i;
int k;
while (1)
{
    syscall(333, pid_ls, comm, isrun, stime); // 系统调用，传参
    printf("PID\tCOMM\t\tISRUNNING\t%%CPU\tTIME\n");
    for (i = 0; i < 100; i++) // init
    {
        nstime[i] = (double)stime[i]/(double)10000000;
        cpuo[i] = (double)(stime[i]-lstime[i])/(double)10000000;      // ns 为 10 的-9
次，cpu 按照百分比计算所以 10 的-7 次
        lstime[i] = stime[i];
    }
    for (i = 0; i < 100; i++)
    {   // 根据 cpu 占用率排序
        int j;
        for (j = i+1; j < 100; j++)
        {
            if (cpuo[i]<cpuo[j])//exchange position
            {
                double temp1;
                int temp2;
                char temp3;

                temp2 = pid_ls[i];
                pid_ls[i] = pid_ls[j];
                pid_ls[j] = temp2;

                temp2 = lstime[i];
                lstime[i] = lstime[j];
                lstime[j] = temp2;

                temp2 = isrun[i];
                isrun[i] = isrun[j];
                isrun[j] = temp2;

                temp1 = cpuo[i];
```

```
                cpuo[i] = cpuo[j];
                cpuo[j] = temp1;

                temp1 = nstime[i];
                nstime[i] = nstime[j];
                nstime[j] = temp1;

                int k;
                for (k = 0;k < 16;k++)
                {
                    temp3 = comm[16*i+k];
                    comm[16*i+k] = comm[16*j+k];
                    comm[16*j+k] = temp3;
                }
            }
        }
    }
    k = 20;
    for (i = 0; i < k; i++)
    {   // 把自己进程去掉
        if(pid_ls[i] == 1)
        {
            k = 21;
            continue;
        }
        else
        {
            int j;
            printf("%d\t",pid_ls[i]);
            for (j = 0; j < 16; j++)
            {
                printf("%c",comm[i * 16 + j]);
            }
            printf("%d\t\t",isrun[i]);
            printf("%.2lf\t%.2lf",cpuo[i],nstime[i]);
            printf("\n");
        }
    }
    sleep(1); // 每隔 1s 刷新一次
    system("clear");
  }
  return 0;
}
```

测试结果



实验总结

实验内容较新，需要参考除实验文档之外的内容。检查之后对模糊的地方也有了明确的理解，对 os 提升较大。