

# 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/要求

### 3.1 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 `;`

### 3.2 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

### 4.1 实现一个 Shell

#### 4.1.1 内置命令的实现

这段代码编写一个函数，名为 `callkill`，用于实现类Unix操作系统中 `kill` 系统调用的功能。`kill` 系统调用可以向指定的进程或进程组发送信号，用于控制或终止进程。

`callkill` 函数有两个或三个参数：`argc`（一个整数，表示传递给函数的参数数量），以及 `argv`（一个指向包含传递给函数的参数的字符指针数组）。

如果 `argc` 等于2，则函数使用第二个参数指定的进程ID（`atoi(argv[1])`）和信号号码15（`SIGTERM`）调用 `kill` 函数，从而终止该进程。

如果 `argc` 等于3，则函数使用第二个参数指定的进程ID和第三个参数指定的信号号码（`atoi(argv[2])`）调用 `kill` 函数。

如果 `argc` 不等于2或3，则函数输出错误信息并返回-1。

```
/*
    内置指令kill:
*/

int callkill(int argc, char **argv) {
    if(argc == 2) {
        kill(atoi(argv[1]), 15);
        printf("Success.");
    }
    else if(argc == 3){
        kill(atoi(argv[1]), atoi(argv[2]));
    }
    else {
        printf("Error.");
        return -1;
    }
    return 0;
}
```

## 4.1.2 执行内置命令

```
/*
    执行内置命令
    arguments:
        argc: 输入，命令的参数个数
        argv: 输入，依次代表每个参数，注意第一个参数就是要执行的命令，
        若执行"ls a b c"命令，则argc=4, argv={"ls", "a", "b", "c"}
        fd: 输出，命令输入和输出的文件描述符 (Deprecated)
    return:
        int, 若执行成功返回0，否则返回值非零
*/

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

    if (strcmp(argv[0], "cd") == 0) {
        // 如果命令是cd，则尝试切换当前工作目录
        if (argv[1] == NULL) {
            // 如果没有指定目录，则切换到用户的家目录
            if (chdir(getenv("HOME")) != 0) {
                // 切换失败，输出错误信息并返回-1
                perror("cd");
                return -1;
            }
        } else { // 尝试切换到指定的目录
            if (chdir(argv[1]) != 0) {
                // 切换失败，输出错误信息并返回-1
                perror("cd");
                return -1;
            }
        }
    } else if (strcmp(argv[0], "exit") == 0) {
        exit(0);
    } else if (strcmp(argv[0], "kill") == 0) {
        callkill(argc, argv);
    } else {
        // 不是内置指令时
        return -1;
    }
}
```

### 4.1.3 重定向

这段代码实现了Linux/Unix系统下的重定向功能，用于将输入输出重定向到指定的文件。该函数接受三个参数：`argc`和`argv`是传递给该程序的参数列表，`fd`是一个指向整型数组的指针，用于存储重定向后的输入输出文件描述符。

在该函数中，首先将`fd[READ_END]`和`fd[WRITE_END]`分别设置为标准输入文件描述符`STDIN_FILENO`和标准输出文件描述符`STDOUT_FILENO`。然后遍历参数列表，查找是否存在重定向符号`<`、`>`或`>>`。如果存在，则打开指定的输入输出文件，并将文件描述符保存到`fd`数组中，以便后续使用。

如果找到`>`重定向符号，则使用`open`系统调用打开指定的输出文件，并将文件描述符保存到`fd[WRITE_END]`中，以便后续将输出重定向到该文件。如果找到`>>`重定向符号，则打开指定的输出文件，并使用`O_APPEND`选项表示以追加模式打开文件。找到`<`重定向符号时，则使用`open`系统调用打开指定的输入文件，并将文件描述符保存到`fd[READ_END]`中，以便后续从该文件读取输入。

对于没有重定向符号的参数，则将它们保存到新的参数列表中，以便后续执行。最后，函数将新的参数列表中的最后一个参数后面的所有参数都设置为`NULL`，并返回新的参数个数`j`，以便在调用`execvp`函数时使用。

需要注意的是，在使用`open`系统调用打开文件时，需要进行错误检查，并在打开失败时输出错误信息。另外，对于重定向符号的参数，需要将它们从参数列表中删除，以便后续执行时不会将它们当作命令的一部分。

```
/*
从argv中删除重定向符和随后的参数，并打开对应的文件，将文件描述符放在fd数组中。
运行后，fd[0]读端的文件描述符，fd[1]是写端的文件描述符
arguments:
    argc: 输入，命令的参数个数
    argv: 输入，依次代表每个参数，注意第一个参数就是要执行的命令，
    若执行"ls a b c"命令，则argc=4, argv={"ls", "a", "b", "c"}
    fd: 输出，命令输入和输出使用的文件描述符
return:
    int, 返回处理过重定向后命令的参数个数
*/

int 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_RDWR | O_CREAT | O_TRUNC, 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_RDWR | 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);
        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
}

```

#### 4.1.4 在本进程中执行给定的命令

这段代码实现了一个函数 `execute`，用于在本进程中执行给定的命令，并在命令执行完毕后结束进程。该函数接受两个参数：`argc` 表示命令的参数个数，`argv` 是一个指向包含命令参数的字符指针数组。

函数中使用了 `process_redirect` 函数，该函数用于处理输入输出重定向符号 `<`、`>` 和 `>>`，并将重定向后的输入输出文件描述符存储在 `fd` 数组中，以便在后续执行命令时使用。如果不需要支持输入输出重定向，则可以注释掉 `process_redirect` 函数的调用。

接着，函数使用 `exec_builtin` 函数尝试执行内置命令。如果命令是内置命令，将在 `exec_builtin` 函数中直接执行该命令，并使用 `exit(0)` 函数结束进程。如果不是内置命令，则继续执行下面的代码。

在处理完内置命令后，函数将标准输入和标准输出的文件描述符修改为 `fd` 数组中的相应值。这样，执行命令时将会从重定向后的文件中读取输入，将输出写入到重定向后的文件中。

最后，函数使用 `execvp` 函数执行命令。`execvp` 函数将根据给定的命令名和参数列表在当前进程中执行指定的程序，如果执行成功，则该函数不会返回。如果执行失败，则 `execvp` 函数返回-1，函数返回0表示执行成功。

```

/*
    在本进程中执行，且执行完毕后结束进程。

```

```

arguments:
    argc: 命令的参数个数
    argv: 依次代表每个参数, 注意第一个参数就是要执行的命令,
    若执行"ls a b c"命令, 则argc=4, argv={"ls", "a", "b", "c"}
return:
    int, 若执行成功则不会返回(进程直接结束), 否则返回非零
*/
int execute(int argc, char** argv) {
    int fd[2];
    // 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;
    // 处理重定向符, 如果不做本部分内容, 请注释掉process_redirect的调用
    argc = process_redirect(argc, argv, fd);
    if(exec_builtin(argc, argv, fd) == 0) {
        exit(0);
    }
    // 将标准输入输出STDIN_FILENO和STDOUT_FILENO修改为fd对应的文件
    dup2(fd[READ_END], STDIN_FILENO);
    dup2(fd[WRITE_END], STDOUT_FILENO);
    /* TODO :运行命令与结束 */
    execvp(argv[0], argv); add below
    return 0;
}

```

#### 4.1.5 shell 的具体实现 (main)

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。

## 4.2 实现一个 top

### 4.2.1 修改过的kernel源码

#### 注册系统调用 `syscall_64.tbl`

`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` 和 `sys_ps_info`。

```
332 common ps_counter sys_ps_counter
333 common ps_info sys_ps_info
```

#### 声明内核函数原型 `syscalls.h`

`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);
```

#### 实现内核函数 `sys.c`

`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。

这段代码实现了一个自定义的系统调用 `ps_info`，用于获取当前正在运行的进程信息，包括进程的 PID、进程名称、执行时间、CPU 时间和进程状态。具体实现细节如下：

1. 遍历进程列表，获取每个进程的 PID、执行时间、CPU 时间和状态。
2. 使用空格作为分隔符，将进程名称存储到字符数组 `name_a` 中。
3. 将进程名称和 PID 拷贝到用户空间 `user_name` 和 `user_pid` 中。
4. 将 CPU 时间和进程状态拷贝到用户空间 `user_time` 和 `user_state` 中。
5. 返回 0，表示系统调用执行成功。

```
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;

    // 输出学号和系统调用名称
    printk("[Syscall] ps_info\n");
    printk("[StuID] PB21081601\n");

    // 遍历进程列表
    for_each_process(task) {
        pid_a[i] = task -> pid;

        // 从用户空间拷贝上一次的进程执行时间
        err = copy_from_user(&(old_time), user_old_time + i, sizeof(unsigned long
long));

        // 计算当前进程执行时间和 CPU 时间
        cpu_time = task -> se.sum_exec_runtime - old_time;

        // 将 CPU 时间拷贝到用户空间
        err = copy_to_user(user_time + i, &(cpu_time), sizeof(unsigned long long));

        // 将进程状态拷贝到用户空间
        err = copy_to_user(user_state + i, &(task -> state), sizeof(long));

        // 使用空格作为分隔符, 将进程名称存储到字符数组中
        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++;
    }

    // 将进程名称和 PID 拷贝到用户空间
    err = copy_to_user(user_name, name_a, sizeof(name_a));
    err = copy_to_user(user_pid, pid_a, sizeof(pid_a));
}

```

```
    return 0;
}
```

## 4.2.2 用户空间代码 `get_ps_info.c`

这段代码实现了一个进程监控程序，会每秒钟获取当前正在运行的进程的信息，并按照 CPU 占用率从高到低排序，输出前 20 个进程的信息。具体实现细节如下：

1. 定义了一个结构体 `task_exist`，用于存储进程的相关信息，包括 PID、进程名称、是否正在运行、执行时间和 CPU 占用率。
2. 使用系统调用 `syscall(333, &sum1, PID1, COMMAND1, running1, time1)` 获取上一次的进程信息，其中 `sum1` 表示进程的数量，`PID1` 数组存储 PID，`COMMAND1` 数组存储进程名称，`running1` 数组存储进程是否正在运行，`time1` 数组存储进程执行时间。
3. 通过 `syscall(333, &sum2, PID2, COMMAND2, running2, time2)` 获取当前的进程信息。
4. 找到上一次和当前都存在的进程，并将其信息存储到结构体 `task` 中，包括进程的 PID、进程名称、是否正在运行、执行时间和 CPU 占用率。
5. 根据 CPU 占用率从高到低排序，找到占用 CPU 最高的前 20 个进程，并输出它们的 PID、进程名称、是否正在运行、执行时间和 CPU 占用率。
6. 更新上一次的进程信息为当前进程信息。
7. 循环执行步骤 3-6，每秒钟更新一次进程信息并输出前 20 个进程的信息。

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

while (1) {
    sleep(1);
    system("clear");
    syscall(333, &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;
}

```

总体来说：

`get_num_ps.c` 和 `get_ps_info.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。

注意到，`ps_counter`，`SYSCALL_DEFINE1`，还有 `asm linkage long sys_ps_counter(int __user * num)`；即是模仿实验文档中的例子，写在这里只是为了更好的对比和理解系统调用的机制，与实际完成实验要求的部分无关。

## 4.3 实验结果

按照要求编译、运行，具体步骤详见实验文档。

### 4.3.1 shell

在当前目录下使用 `gcc -o shell lab2_shell.c` 编译，使用 `./shell` 运行，得到一个自制的 shell。

尝试使用以下指令进行测试，均通过：

- 基本单条命令 `ls`
- 两条命令之间的管道 `ps aux | wc -l`
- 内建命令 `cd`, `kill`, `exit`
- 多条命令之间的管道 `ps aux | grep bash | wc -l`
- 支持重定向符 `>`, `<`, `>>` 和分号 `;`，此部分在代码中有所体现，但是没有进行案例测试。

```

minerva@minerva-VirtualBox:~/桌面$ cd ~/桌面/winshare/lab2/src/part1
minerva@minerva-VirtualBox:~/桌面/winshare/lab2/src/part1$ gcc -o shell lab2_shell.c
minerva@minerva-VirtualBox:~/桌面/winshare/lab2/src/part1$ ./shell
shell: /home/minerva/桌面/winshare/lab2/src/part1 -> pwd
/home/minerva/桌面/winshare/lab2/src/part1
shell: /home/minerva/桌面/winshare/lab2/src/part1 -> cd /home
shell: /home -> cd /home/minerva/桌面/winshare/lab2/src/part1
shell: /home/minerva/桌面/winshare/lab2/src/part1 -> ls
lab2_shell.c  shell
shell: /home/minerva/桌面/winshare/lab2/src/part1 -> ps aux | wc -l
231
shell: /home/minerva/桌面/winshare/lab2/src/part1 -> ps aux | grep bash | wc -l
2

```

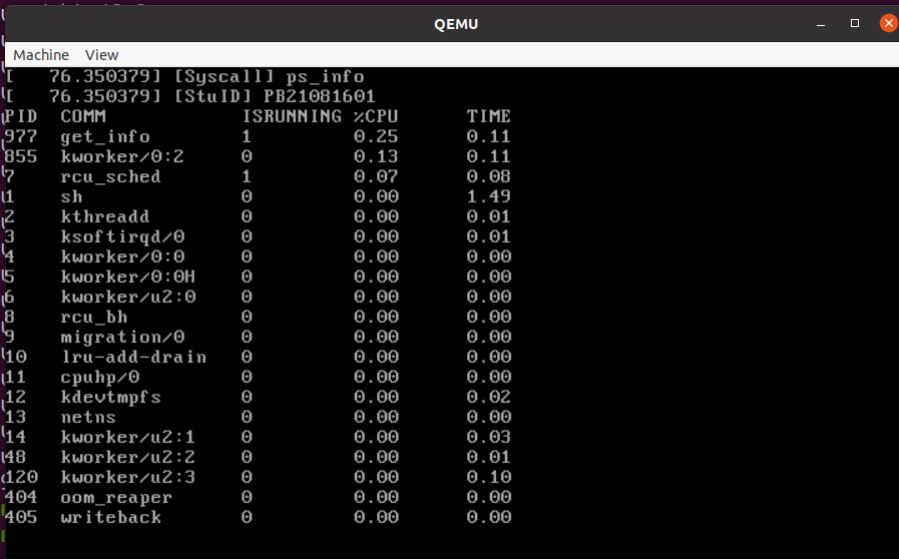
```

shell: /home/minerva/桌面/winshare/lab2/src/part1 -> ps
  PID TTY          TIME CMD
  5010 pts/0    00:00:00 bash
  5034 pts/0    00:00:00 shell
  5103 pts/0    00:00:00 ps
shell: /home/minerva/桌面/winshare/lab2/src/part1 -> kill 5010
Success.shell: /home/minerva/桌面/winshare/lab2/src/part1 -> exit
minerva@minerva-VirtualBox:~/桌面/winshare/lab2/src/part1$

```

### 4.3.2 top

- 程序在【用户态】每秒打印一次各进程的CPU占用统计，并按CPU占用统计排序
- 程序在【用户态】每秒打印一次各进程的PID、进程名、进程是否处于running状态、进程运行时间



PID	COMM	ISRUNNING	%CPU	TIME
1	get_info	1	0.25	0.11
855	kworker/0:2	0	0.13	0.11
7	rcu_sched	1	0.07	0.08
1	sh	0	0.00	1.49
2	kthreadd	0	0.00	0.01
3	ksoftirqd/0	0	0.00	0.01
4	kworker/0:0	0	0.00	0.00
5	kworker/0:0H	0	0.00	0.00
6	kworker/u2:0	0	0.00	0.00
8	rcu_bh	0	0.00	0.00
9	migration/0	0	0.00	0.00
10	lru-add-drain	0	0.00	0.00
11	cpuhp/0	0	0.00	0.00
12	kdevtmpfs	0	0.00	0.02
13	netns	0	0.00	0.00
14	kworker/u2:1	0	0.00	0.03
48	kworker/u2:2	0	0.00	0.01
120	kworker/u2:3	0	0.00	0.10
404	oom_reaper	0	0.00	0.00
405	writeback	0	0.00	0.00

## 五 实验总结

### 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。

## 遇到的问题

1. 在lab1中为了省事，没有配置增强功能、粘贴板共享等，在做实验时感觉虚拟机和宿主机之间共享文件很麻烦，所以回头配置增强功能，共享文件夹等，耗费了大量时间。以后要吸取教训，不能因为图方便就在任务轻松时偷懒。
2. 在写用户空间的测试代码的时候，头文件少写了一个，导致调用的函数用不了。
3. 将3.4.2编译出的可执行文件 `get_ps_num` 放到 `busybox-1.32.1/_install` 下面，重新制作 `initramfs` 文件（重新执行Lab1的3.2.6.6的find操作），这样我们才能在qemu中看见编译好的 `get_ps_num` 可执行文件。  
  
执行这一步的时候，忘记了 lab1 中的细节，没在 `_install` 下面打开，导致程序跑不出来，最后试了好几遍才发现问题。
4. 一开始启动qemu的时候，按照lab1调试gdb的时候加了 `-s -s`，发现弹出的窗口不动，最后查阅实验文档，发现自己没有理解 `-s -s` 的含义。

P.S. 另外在此说明一下bb系统上第一份提交的报告（ddl内提交）不是最终版本的报告，有不完善的地方；今天（5.8）根据助教上课所讲的报告要求再次修改完善了报告，不知道这样是不是要算迟交扣分（？）希望助教能通融一下QAQ...