OS Lab2

Part 1 :

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#include <paths.h>

#define MAX_CMDLINE_LENGTH  1024    /* max cmdline length in a line*/
#define MAX_BUF_SIZE        4096    /* max buffer size */
#define MAX_CMD_ARG_NUM     32      /* max number of single command args */
#define WRITE_END 1     // pipe write end
#define READ_END 0      // pipe read end

#define PATH_SIZE 50
extern char **__environ;
/*
 * 需要大家完成的代码已经用注释`TODO:`标记
 * 可以编辑器搜索找到
 * 使用支持TODO高亮编辑器（如vscode装TODO highlight插件）的同学可以轻松找到要添加内容的地方。
 */

/*
    int split_string(char* string, char *sep, char** string_clips);

    基于分隔符sep对于string做分割，并去掉头尾的空格

    arguments:      char* string, 输入, 待分割的字符串
                    char* sep, 输入, 分割符
                    char** string_clips, 输出, 分割好的字符串数组

    return:   分割的段数
*/

int split_string(char* string, char *sep, char** string_clips) {

    char string_dup[MAX_BUF_SIZE];
    string_clips[0] = strtok(string, sep);
    int clip_num=0;

    do {
```

```
        char *head, *tail;
        head = string_clips[clip_num];
        tail = head + strlen(string_clips[clip_num]) - 1;
        while(*head == ' ' && head != tail)
            head ++;
        while(*tail == ' ' && tail != head)
            tail --;
        *(tail + 1) = '\0';
        string_clips[clip_num] = head;
        clip_num ++;
    }while(string_clips[clip_num]=strtok(NULL, sep));
    return clip_num;
}

/*
    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: 添加和实现内置指令 */

    if (strcmp(argv[0], "cd") == 0) {
        if(chdir(argv[1]) != 0){
            printf("cd: no such file or directory: %s", argv[1]);
            return -1;
        }
    } else if (strcmp(argv[0], "exit") == 0){
       exit(0);
    } else {
        // 不是内置指令时
        return -1;
    }
}

/*
    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
}



/*
    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);
    return 0;
}

int main() {
    /* 输入的命令行 */
    char cmdline[MAX_CMDLINE_LENGTH];

    char *commands[128];
    char *multi_cmd[128];
    int cmd_count;
    while (1) {
        /* TODO: 增加打印当前目录，格式类似"shell:/home/oslab ->"，你需要改下面的printf */
        char path_name[51];
        getcwd(path_name, PATH_SIZE);
        printf("shell:%s -> ", path_name);
        fflush(stdout);

        fgets(cmdline, 256, stdin);
        strtok(cmdline, "\n");

        /* TODO: 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);
        for(int i = 0; i < multi_cmd_num; i++){
            strcpy(cmdline, multi_cmd[i]);

            /* 由管道操作符'|'分割的命令行各个部分，每个部分是一条命令 */
            /* 拆解命令行 */
            cmd_count = split_string(cmdline, "|", commands);

            if(cmd_count == 0) {
```

```
            continue;
        } else if(cmd_count == 1) {     // 没有管道的单一命令
            char *argv[MAX_CMD_ARG_NUM];
            int argc;
            int fd[2];
            /* TODO:处理参数，分出命令名和参数
            *
            *
            *
            */
            argc = split_string(cmdline, " ", argv);

            /* 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;
            }
            /* TODO:创建子进程，运行命令，等待命令运行结束
            *
            *
            *
            *
            */
            pid_t pid = fork();
            if(pid == 0) {
                if(execute(argc, argv) < 0) {
                    printf("%s : Command not found.\n",argv[0]);
                    exit(0);
                }
            }
            while(wait(NULL) > 0);

        } else if(cmd_count == 2) {     // 两个命令间的管道
            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[0]);
                dup2(pipefd[1], STDOUT_FILENO);
                close(pipefd[1]);
                /*
                    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);
```

```
252                     execute(argc, argv);
253                     exit(255);
254 
255                 }
256                 // 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
257                 // 而是直接创建下一个子进程
258                 // 子进程2
259                 pid = fork();
260                 if(pid == 0) {
261                     /* 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 */
262                     close(pipefd[1]);
263                     dup2(pipefd[0], STDIN_FILENO);
264                     close(pipefd[0]);
265 
266                     char *argv[MAX_CMD_ARG_NUM];
267                     /* TODO:处理参数，分出命令名和参数，并使用execute运行
268                     * 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
269                     * 因此我们用了一个封装好的execute函数
270                     *
271                     *
272                     */
273                     int argc = split_string(commands[1], " ", argv);
274                     execute(argc, argv);
275                     exit(255);
276                 }
277                 close(pipefd[WRITE_END]);
278                 close(pipefd[READ_END]);
279 
280                 while (wait(NULL) > 0);
281             } else {    // 选做：三个以上的命令
282                 int read_fd;    // 上一个管道的读端口（出口）
283                 for(int i = 0; i < cmd_count; i++) {
284                     int pipefd[2];
285                     /* TODO: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
286                     *
287                     *
288                     *
289                     */
290                     if(i != cmd_count - 1){
291                         int ret = pipe(pipefd);
292                         if(ret < 0) {
293                             printf("pipe error!\n");
294                             continue;
295                         }
296                     }
297 
298                     int pid = fork();
299                     if(pid == 0) {
300                         /* 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
301                         *
302                         *
303                         *
```

```
                        */
                        if(i != cmd_count - 1) {
                            close(pipefd[0]);
                            dup2(pipefd[1], STDOUT_FILENO);
                            close(pipefd[1]);
                        }

                        /* 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[1]);
                            dup2(read_fd, STDIN_FILENO);
                            close(read_fd);
                            if(i == cmd_count - 1) close(pipefd[0]);
                        }

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

    }
}
```

Part 2 :

```c
//get_ps_num.c
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>
int main()
{
    int result;
    syscall(332, &result);
    printf("process number is %d\n", result);
    return 0;
}
```

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

typedef struct ps_array
{
    pid_t pid_a[128];
    char name_a[1024];
    unsigned long long cpu_time_a[128];
    long state_a[128];
}ps_array;

int split_string(char* string, char *sep, char** string_clips) {

    char string_dup[1024];
    string_clips[0] = strtok(string, sep);
    int clip_num=0;

    do {
        char *head, *tail;
        head = string_clips[clip_num];
        tail = head + strlen(string_clips[clip_num]) - 1;
        while(*head == ' ' && head != tail)
            head ++;
        while(*tail == ' ' && tail != head)
            tail --;
        *(tail + 1) = '\0';
        string_clips[clip_num] = head;
        clip_num ++;
    }while(string_clips[clip_num]=strtok(NULL, sep));
```

```
    return clip_num;
}

int main()
{
    ps_array user_array;
    int  i = 0, j = 0, cnt;
    int p_out[128];
    char* ps_name[128];
    char out_name[1024];
    char* n_out[128];
    unsigned long long old_time[128];
    long ps_state[128], s_out[128];
    double ps_time[128], t_out[128];

    for(i = 0; i < 128; i++) old_time[i] = 0;

    while(1){
        syscall(332, &cnt);
        syscall(333, &(user_array.pid_a), &(user_array.name_a), &(user_array.cpu_time_a), &(old_time), &(user_array.state_a));
        strcpy(out_name, user_array.name_a);
        int pieces = split_string(user_array.name_a, " ", ps_name);
        int pieces_out = split_string(out_name," ", n_out);
        for(i = 0; i < cnt; i++) {
            if(!user_array.state_a[i]) ps_state[i] = 1;
            else ps_state[i] = 0;
            old_time[i] = user_array.cpu_time_a[i] + old_time[i];
            ps_time[i] = (((double) (user_array.cpu_time_a[i]))/1000000000);

            s_out[i] = ps_state[i]; t_out[i] = ps_time[i]; p_out[i] = user_array.pid_a[i];
        }

      //冒泡排序 这里相当于每次都copy一份user_array之后用这个copy做排序 （如果用原数组排序而忽略内核态函数for_each_process生成process的顺序会导致混乱）

        for(i = 0; i < cnt; i++) {
            for(j = 0; j < cnt - i -1; j++){
                if(t_out[j] < t_out[j + 1]){
                    char * n_tmp = n_out[j + 1]; n_out[j + 1] = n_out[j]; n_out[j] = n_tmp;

                    long s_tmp = s_out[j + 1]; double t_tmp = t_out[j + 1]; pid_t p_tmp = p_out[j + 1];
                    s_out[j + 1] = s_out[j]; t_out[j + 1] = t_out[j]; p_out[j + 1] = p_out[j];
                    s_out[j] = s_tmp; t_out[j] = t_tmp; p_out[j] = p_tmp;
                }
            }
        }

        printf("PID              COMM        CPU          ISRUNNING\n");
```

```
        //for(i = 0; i < 20; i++) printf("%-5d %*s %10.5f%% %18ld\n", user_array.pid_a[i], 16, ps_name[i], ps_time[i], ps_state[i]);
        for(i = 0; i < 20; i++) printf("%-5d %*s %10.5f%% %18ld\n", p_out[i], 16, n_out[i], t_out[i], s_out[i]);

        sleep(1);
        system("clear");
    }
}
```

```
//sys.c

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("[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

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
332 common  ps_counter      sys_ps_counter
333 common  ps_info         sys_ps_info
```