华东师范大学数据学院上机实践报告

课程名称：操作系统　　**年级：**2019 级　　**上机实践成绩：**

指导教师：翁楚良　　**姓名：**龚敬洋

上机实践名称：Shell 及系统调用　　**学号：**10195501436　　**上机实践日期：**2021/3/2

上机实践编号：

一、目的

在 MINIX 环境下通过系统调用实现一个基本的 Shell。

二、内容与设计思想

Shell 能解析的命令行如下:

1. 带参数的程序运行功能。

```
program arg1 arg2 ... argN
```

2. 重定向功能，将文件作为程序的输入/输出。

(1) “>”表示覆盖写

```
program arg1 arg2 ... argN > output-file
```

(2) “>>”表示追加写

```
program arg1 arg2 ... argN >> output-file
```

(3) “<”表示文件输入

```
program arg1 arg2 ... argN < input-file
```

3. 管道符号“|”，在程序间传递数据。

```
programA arg1 ... argN | programB arg1 ... argN
```

4. 后台符号&，表示此命令将以后台运行的方式执行。

```
program arg1 arg2 ... argN &
```

5. 工作路径移动命令 cd。
6. 程序运行统计 mytop。
7. shell 退出命令 exit。
8. history n 显示最近执行的 n 条指令。

三、使用环境

开发环境：Clion 2020.3

宿主机系统环境：Mac OS Big Sur 11.2.2
虚拟机应用：VirtualBox 6.1.18
虚拟机环境：Minix 3.3.0

四、实验过程

1. Shell 主体

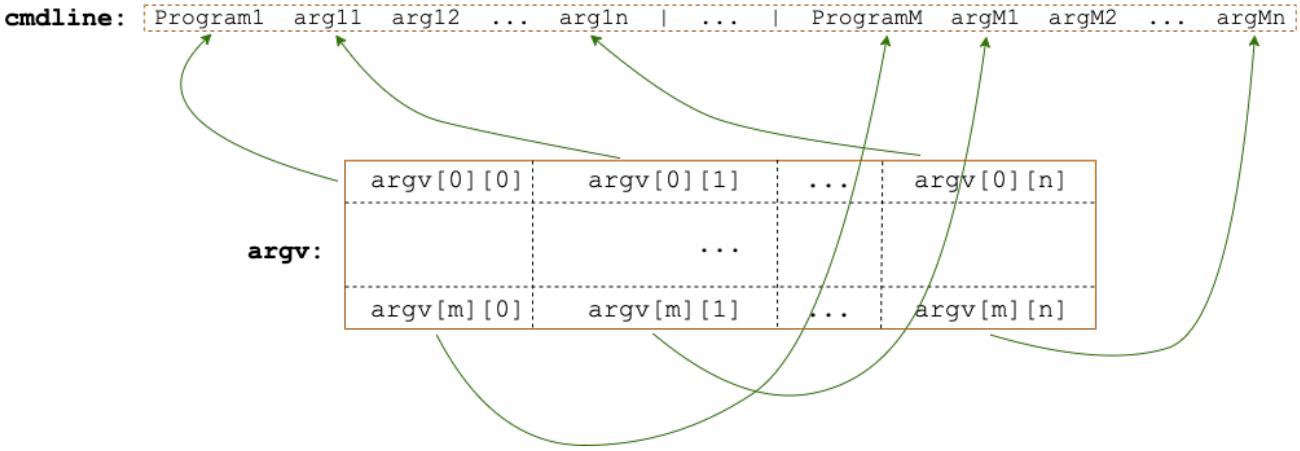
首先我们来实现 Shell 的基本结构。一个 Shell 的基本行为是不断接收用户的输入，并根据指令执行相应的操作。因此我们可以快速写出其 main 函数：

```c
#define MAXCMD 1024 //Max number of commands that can be recorded
#define MAXLINE 1024 //Max number of characters a command line can have

char cmd_list[MAXCMD][MAXLINE];

void eval(char *cmdline);

int main(int argc, char **argv){
    char cmdline[MAXLINE]; //Command line string

    cmd_cnt = 0;

    while(1){
        printf("shell: %s %% ", getcwd(NULL, NULL));
        //Read the command line from stdin
        if(fgets(cmdline, MAXLINE, stdin) == NULL) {
            print_message("Error occurred when reading command line!");
            continue;
        }

        strcpy(cmd_list[cmd_cnt], cmdline);
        cmd_cnt++;
        //Evaluate the command line
        eval(cmdline);
    }
    return 0;
}
```

由于后续需要通过 history 指令查询用户输入过的指令，这里我们设置了一个全局变量来记录用户每次输入的指令字符串。

2. 指令输入解析

接下来我们对用户输入的指令进行结构化解析。我们使用一个二维字符串数组来存储用户输入的指令及参数流，这个数组的结构如下：



随后我们通过一个结构体来存储每行指令的全局特性（输入输出重定向，是否后台运行等）。

```c
#define MAXPROG 16 //Max program that can be connected by pipe
#define MAXARGS 128 //Max number of arguments a command can have

void parseline(char *cmdline, char *argv[MAXPROG][MAXARGS], struct cmd_feature *feature);

struct cmd_feature{
    int is_error; //Error flag, 0 - No error, 1 - Error in cmdline
    int bg; //Background flag, 1 - Background, 0 - Foreground
    int prog_num; //Number of program connected by pipe (0 means no pipe feature)
    int input_mode; //Overall input mode, 0 - Standard Input, 1 - File Input
```

```
    int output_mode; //Overall output mode, 0 - Standard Output, 1 - File Output (Overrride), 2 - File Output (Append)
    char *infile; //Path of input file
    char *outfile; //Path of output file
};
void eval(char *cmdline){
    char *argv[MAXPROG][MAXARGS]; //Argument strings
    struct cmd_feature line_feature; //Feature of the cmdline

    //Parse the command line
    parseline(cmdline, argv, &line_feature);
}
```

现在我们来实现具体的指令解析逻辑。由于用户的指令是一次性输入的，因此我们需要考虑所有可能的情况。总的来说，对于一行指令，可能出现的内容有如下几个：

(1) 由管道连接符相连接的一组待执行的程序
(2) 每个待执行程序附带的若干参数
(3) 输入重定向
(4) 输出重定向
(5) 后台运行标识符

我们分情况讨论所有的这些情况，并将它们结构化存储到相应的数组和结构体中：

```
void parseline(char *cmdline, char *argv[MAXPROG][MAXARGS], struct cmd_feature *feature){
    char *delim; // Points to first space delimiter
    int argc; // Number of arguments
    int prog_cnt; //Counter of programs

    cmdline[strlen(cmdline) - 1] = ' '; // Replace trailing '\n' with space
    while (*cmdline && (*cmdline == ' ')) // Ignore leading spaces
        cmdline++;

    // Build the argv list
    feature->is_error = 0;
    feature->input_mode = 0;
    feature->output_mode = 0;
    prog_cnt = 0;
    argc = 0;
    delim = strchr(cmdline, ' ');
    while (delim) {
        char *arg_tmp = strtok(cmdline, " ");
        if(!strcmp(arg_tmp, "<")){
            *delim = '\0';
            cmdline = delim + 1;
            while (*cmdline && (*cmdline == ' ')) // Ignore redundant spaces
                cmdline++;
            delim = strchr(cmdline, ' ');
            if(!delim){
                print_message("Missing I/O File Path!");
                feature->is_error = 1;
                return;
            }
            feature->input_mode = 1;
            feature->infile = strtok(cmdline, " ");
        }
        else if(!strcmp(arg_tmp, ">")){
            *delim = '\0';
            cmdline = delim + 1;
            while (*cmdline && (*cmdline == ' ')) // Ignore redundant spaces
                cmdline++;
            delim = strchr(cmdline, ' ');
            if(!delim){
```

```
                print_message("Missing I/O File Path!");
                feature->is_error = 1;
                return;
            }
            feature->output_mode = 1;
            feature->outfile = strtok(cmdline, " ");
        }
        else if(!strcmp(arg_tmp, ">>")){
            *delim = '\0';
            cmdline = delim + 1;
            while (*cmdline && (*cmdline == ' ')) // Ignore redundant spaces
                cmdline++;
            delim = strchr(cmdline, ' ');
            if(!delim){
                print_message("Missing I/O File Path!");
                feature->is_error = 1;
                return;
            }
            feature->output_mode = 2;
            feature->outfile = strtok(cmdline, " ");
        }
        else if(!strcmp(arg_tmp, "|")){
            argc++;
            argv[prog_cnt][argc] = NULL;
            prog_cnt++;
            argc = 0;
        }
        else{
            argv[prog_cnt][argc] = arg_tmp;
            argc++;
        }
        *delim = '\0';
        cmdline = delim + 1;
        while (*cmdline && (*cmdline == ' ')) // Ignore redundant spaces
            cmdline++;
        delim = strchr(cmdline, ' ');
    }
    argv[prog_cnt][argc] = NULL;
    feature->prog_num = prog_cnt + 1;

    // Figure out whether the job should be run in foreground or background
    if(argc == 0){
        feature->bg = 0;
    }
    else if(!strcmp(argv[prog_cnt][argc - 1], "&"))
        feature->bg = 1;
    else
        feature->bg = 0;
    if (feature->bg != 0) {
        argc--;
        argv[prog_cnt][argc] = NULL;
    }
}
```

3. 内置指令

我们的 Shell 中包含 4 条内置指令，当检测到用户输入这四条指令时，我们需要立即执行并给出相应的反馈。这里我们分别使用函数对 4 条内置指令的实现进行封装，并将判断是否为内置指令的过程封装为另一个函数以供主函数调用：

```
//Judge whether the command is a builtin command, 1 - Builtin command, 0 - Other command
 builtin_cmd(char **argv){
 if(!strcmp(argv[0], "cd")){
     change_dir(argv);
```

```
        return 1;
    }
    else if(!strcmp(argv[0], "history")){
        print_history();
        return 1;
    }
    else if(!strcmp(argv[0], "mytop")){
        perform_top();
        return 1;
    }
    else if(!strcmp(argv[0], "exit")){
        exit(0);
    }
    else
        return 0;
```

对于 exit 指令，我们只需要直接退出程序即可。

对于 cd 指令，我们可以通过使用 chdir 系统调用来更改工作目录：

```
void change_dir(char **argv){
    if(chdir(argv[1]) < 0){
        print_message("Error when changing working directory!");
    };
}
```

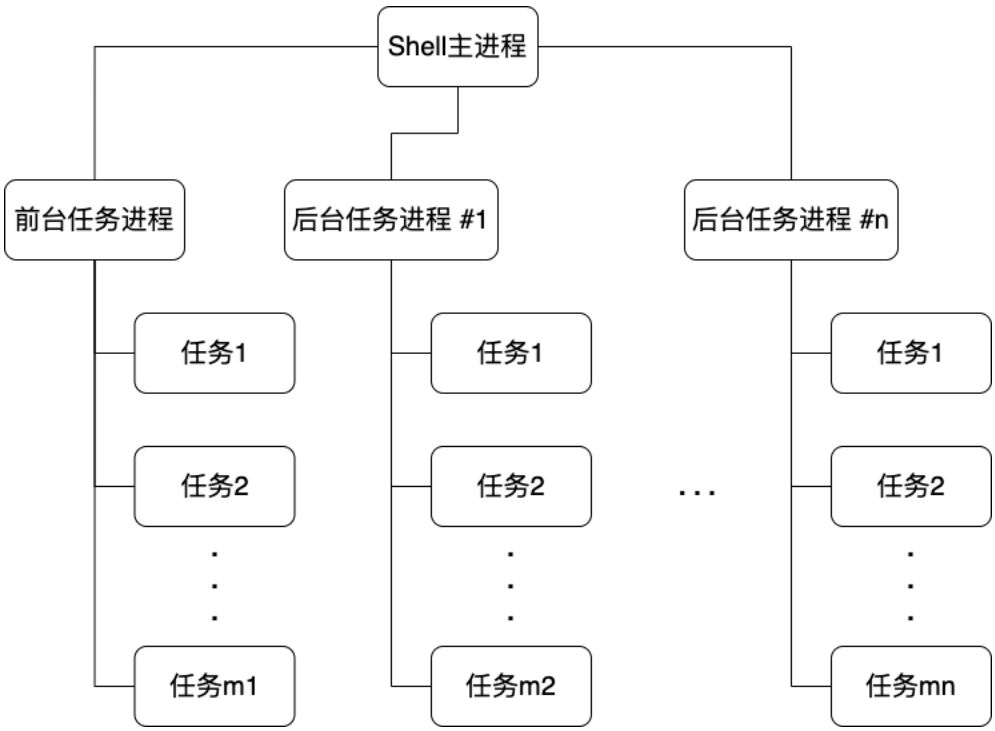
对于 history 指令，由于前面我们已经通过 cmd_list 全局变量保存了每次输入的指令，这里我们只需要遍历这一数组打印指令即可：

```
void print_history(){
    for(int i = 0; i < cmd_cnt; i++){
        fprintf(stdout, "%s", cmd_list[i]);
    }
}
```

对于 mytop 指令，由于逻辑相对较为复杂，我们放在后文中讨论。

4. 任务流管理

若用户输入的不为内置指令，则 Shell 需要执行相应的任务序列。由于允许后台程序，Shell 中可能会同时存在多个子任务，因此我们需要将每个任务相互隔离开来。考虑到后续管道的建立，我们将使用如下的进程结构图：



首先我们使用 fork 函数创建一个子进程用于托管当前指令流的任务，随后对于任务序列中的每条指令，我们再分别 fork 一个单独的进程用于执行程序。由于任务序列中的任务为顺序执行的，因此我们需要使用 waitpid 函数等待上一任务执行完毕再新建下一任务进程。这里需要注意的是，由于我们是在托管进程中再新开子进程执行程序的，托管进程在子任务全部结束后并不会自动退出，这样父进程中的后续代码同样会被托管进程执行，造成无限嵌套，因此我们必须在子任务结束后手动结束托管进程。相关代码实现如下：

```
1.  void waitfg(pid_t pid){
2.      int state;
3.      waitpid(pid, &state, 0);
4.  }
5.  void eval(char *cmdline){
6.      char *argv[MAXPROG][MAXARGS]; //Argument strings
7.      struct cmd_feature line_feature; //Feature of the cmdline
8.      pid_t pid; //PID of the latest child
9.
10.     //Parse the command line
11.     parseline(cmdline, argv, &line_feature);
12.
13.     //Command line error
14.     if(line_feature.is_error == 1)
15.         return;
16.
17.     //Blank space
18.     if(argv[0][0] == NULL)
19.         return;
20.
21.     if (!builtin_cmd(argv[0])) {
22.         if ((pid = fork()) == 0) {
23.             for (int i = 0; i < line_feature.prog_num; i++) {
24.                 if ((pid = fork()) == 0) {
25.                     //Subsequent logic goes here
26.
27.                     if (execvp(argv[i][0], argv[i]) < 0) {
28.                         fprintf(stdout, "%s: Program not found.\n", argv[i][0]);
29.                         exit(0);
30.                     }
```

```
                } else {
                    waitfg(pid);
                }
            }
        exit(0);
        //Subsequent logic goes here
        }
    }
}
```

5. 输入输出重定向

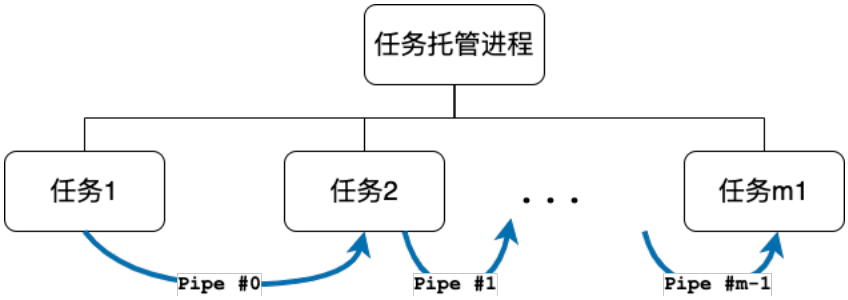
当用户输入的指令带有<、>或>>时，则需要将程序的输入输出流重定向到用户指定的文件中。由于在前面的解析中我们已经将需要的参数记录了下来，这里我们只需要根据结构化的信息打开文件，并利用 dup 函数将其文件描述符赋给标准输入/输出即可：

```
void eval(char *cmdline){
    struct cmd_feature line_feature; //Feature of the cmdline
    pid_t pid; //PID of the latest child
    //Other codes omitted
    if (!builtin_cmd(argv[0])) {
        //Other codes omitted
        if ((pid = fork()) == 0) {
            for (int i = 0; i < line_feature.prog_num; i++) {
                //Other codes omitted
                if ((pid = fork()) == 0) {
                    //Input redirection
                    if (i == 0 && line_feature.input_mode == 1) {
                        int fd = open(line_feature.infile, O_RDONLY);
                        close(STDIN_FILENO);
                        dup(fd);
                    }
                    //Output redirection(Override file)
                    else if (i == (line_feature.prog_num - 1) && line_feature.output_mode == 1) {
                        int fd = open(line_feature.outfile, O_CREAT | O_WRONLY | O_TRUNC, S_IREAD | S_IWRITE);
                        close(STDOUT_FILENO);
                        dup(fd);
                    }
                    //Output redirection(Append to the end of file)
                    else if (i == (line_feature.prog_num - 1) && line_feature.output_mode == 2) {
                        int fd = open(line_feature.outfile, O_CREAT | O_WRONLY | O_APPEND, S_IREAD | S_IWRITE);
                        close(STDOUT_FILENO);
                        dup(fd);
                    }
                    //Other codes omitted
                }
                //Other codes omitted
            }
            //Other codes omitted
        }
        //Other codes omitted
    }
}
```

6. 管道连接

当用户输入的指令中包含多于一个程序时，我们需要通过管道将前一个程序的输出与后一个程序的输入相连接。我们可以使用 pipe 函数来创建管道。对于一根管道，我们需要传

入一个大小为 2 的一维数组作为管道的输入和输出端文件描述符，由于一条指令中可能含有多个程序，因此我们使用一个二维数组来为每两个程序间分别开辟一根管道以实现程序通讯的隔离。



现在，我们只需要使用上文中输入输出重定向的方式将程序的输入输出分别重定向到管道的入口和出口即可。需要注意的是，在任务托管进程中，我们需要关闭管道的输入端，以防止父子进程间的管道交互：

```
#define MAXPROG 16 //Max program that can be connected by pipe

void eval(char *cmdline){
    struct cmd_feature line_feature; //Feature of the cmdline
    pid_t pid; //PID of the latest child
    int pipe_gate[MAXPROG][2];
    //Other codes omitted
    if (!builtin_cmd(argv[0])) {
        //Other codes omitted
        if ((pid = fork()) == 0) {
            for (int i = 0; i < line_feature.prog_num; i++) {
                pipe(pipe_gate[i]);
                if ((pid = fork()) == 0) {
                    //Other codes omitted
                    //Pipe connection
                    if (line_feature.prog_num > 1) {
                        if (i == 0) {
                            close(STDOUT_FILENO);
                            close(pipe_gate[i][0]);
                            dup(pipe_gate[i][1]);
                        } else if (i == line_feature.prog_num - 1) {
                            close(STDIN_FILENO);
                            close(pipe_gate[i - 1][1]);
                            dup(pipe_gate[i - 1][0]);
                        } else {
                            close(STDIN_FILENO);
                            close(pipe_gate[i - 1][1]);
                            dup(pipe_gate[i - 1][0]);
                            close(STDOUT_FILENO);
                            close(pipe_gate[i][0]);
                            dup(pipe_gate[i][1]);
                        }
                    }
                    //Other codes omitted
                } else {
                    close(pipe_gate[i][1]);
                    waitfg(pid);
                }
            }
            //Other codes omitted
        }
        //Other codes omitted
    }
}
```

7. 后台运行

当用户输入的指令最后含有&时，表明用户希望该任务流在后台执行。对于 Shell 主进程来说，我们只需要让其不等待子进程结束直接进入下一轮循环即可。但若直接这样做，则当子进程运行结束后将成为僵尸进程占用系统资源，因此我们需要使用信号系统将其交给 MINIX 系统托管。此外，为了防止子进程受到终端输入输出的影响，我们需要将其输入输出流重定向到/dev/null 下。同样的，由于前面我们已经保存了当前指令是否要后台执行的信息，这里我们只需要直接读取前文结构体中的相关字段即可：

```
1.  void eval(char *cmdline){
2.   struct cmd_feature line_feature; //Feature of the cmdline
3.   pid_t pid; //PID of the latest child
4.   //Other codes omitted
5.   if (!builtin_cmd(argv[0])) {
6.       if(line_feature.bg){
7.           signal(SIGCHLD, SIG_IGN);
8.       }
9.       else{
10.          signal(SIGCHLD, SIG_DFL);
11.      }
12.      if ((pid = fork()) == 0) {
13.          for (int i = 0; i < line_feature.prog_num; i++) {
14.              //Other codes omitted
15.              if ((pid = fork()) == 0) {
16.                  //Other codes omitted
17.                  //Background command
18.                  if (i == (line_feature.prog_num - 1) && line_feature.bg) {
19.                      int fd = open("/dev/null", O_RDWR);
20.                      close(STDIN_FILENO);
21.                      dup(fd);
22.                      close(STDOUT_FILENO);
23.                      dup(fd);
24.                  }
25.                  //Other codes omitted
26.              }
27.              //Other codes omitted
28.          }
29.          //Other codes omitted
30.      } else {
31.          if (!line_feature.bg) {
32.              waitfg(pid);
33.          }
34.      }
35.  }
```

8. 系统资源查看

现在我们来实现 mytop 指令的逻辑。当用户输入 mytop 指令时，Shell 需要打印出当前系统的内存使用情况及 CPU 占用率。我们可以在 MINIX 系统的/proc 目录下查询到这些信息。

对于内存使用情况，我们需要打开 proc 目录下的 meminfo 文件。该文件中共有 5 个参数，分别对应页面大小、总页数量、空闲页数量、最大页数量以及缓存页数量。我们只需将其依次读入，并将页面大小与其他几个参数依次向乘，即可得到内存的相应使用情况：

```
1.  #define MAXBUF 1024 //Max buffer size for an file read operation
2.
3.  void get_memory(int *total_size, int *free_size, int *cached_size){
4.      int mem_f; //File descriptor of memory info
5.      int bufsize; //Actual size of bytes read
6.      char buf[MAXBUF]; //Buffer for reading from file
```

```
    int page_size, total_page, free_page, largest_page, cached_page;

    mem_f = open(MEMPATH, O_RDONLY); //Open memory info file
    bufsize = read(mem_f, buf, sizeof(buf)); //Read memory info
    if(bufsize == -1){
        print_message("Error reading memory info!");
    }
    else{
        page_size = atoi(strtok(buf, " "));
        total_page = atoi(strtok(NULL, " "));
        free_page = atoi(strtok(NULL, " "));
        largest_page = atoi(strtok(NULL, " "));
        cached_page = atoi(strtok(NULL, " "));
        *total_size = (page_size * total_page) / 1024;
        *free_size = (page_size * free_page) / 1024;
        *cached_size = (page_size * cached_page) / 1024;
    }
}

void perform_top(){
    int total_size, free_size, cached_size; //Info of memory

    get_memory(&total_size, &free_size, &cached_size);
    fprintf(stdout, "Total: %dK, Free: %dK, Cached: %dK\n", total_size, free_size, cached_
size);
    //Other codes omitted
}
```

CPU 占用率的获取方式相对较为复杂。首先我们需要从 proc 目录下的 kinfo 文件中获取进程和任务的总数量：

```
#define PROCPATH "/proc/kinfo"
unsigned int nr_procs, nr_tasks;
int nr_total; //Number of process + task

void getkinfo()
{
    int fd; //File descriptor of kinfo
    int bufsize; //Actual buffer size
    char buf[MAXBUF], pathbuf[MAXBUF]; //Buffer for file reading

    fd = open(PROCPATH, O_RDONLY);
    if (fd == -1) {
        print_message("Reading kinfo file error!");
        exit(1);
    }
    bufsize = read(fd, buf, sizeof(buf)); //Read process info
    if(bufsize == -1){
        print_message("Error reading total process info!");
    }
    else {
        nr_procs = (unsigned int) atoi(strtok(buf, " ")); //Number of process
        nr_tasks = (unsigned int) atoi(strtok(NULL, " ")); //Number of tasks
        close(fd);
        nr_total = (int) (nr_procs + nr_tasks);
    }
}
```

随后我们需要遍历整个 proc 目录以获取各个进程的具体信息。对于一个进程号为 PID 的进程，其信息被保存在/proc/PID/psinfo 的文件下，该文件中的前 13 个参数分别为版本号、类型（T - Task，S - System，U - User）、端点、进程名字、进程状态（S - Sleep，W - Wait，Zombie - Z，R - Run，T - Stop）、阻塞状态、动态优先级、滴答、高

周期、低周期、内存、有效用户和静态优先级。为了方便起见我们定义一个结构体用于保存每个进程的各项参数：

```
const char *cputimenames[] = { "user", "ipc", "kernelcall" }; //CPU cycle types

#define CPUTIMENAMES (sizeof(cputimenames)/sizeof(cputimenames[0]))

struct proc {
    int p_flags;
    int p_endpoint;
    pid_t p_pid;
    uint64_t p_cpucycles[CPUTIMENAMES];
    int p_priority;
    int p_blocked;
    time_t p_user_time;
    long unsigned int p_memory;
    uid_t p_effuid;
    int p_nice;
    char p_name[17];
};
```

并定义一组宏方便后续对各项参数的处理：

```
#define INFOPATH "/proc"

#define CPUTIME(m, i) (m & (1L << (i)))

#define  USED       0x1
#define  IS_TASK    0x2
#define  IS_SYSTEM  0x4
#define  BLOCKED    0x8

/* Process types. */
#define TYPE_TASK   'T'
#define TYPE_SYSTEM 'S'
#define TYPE_USER   'U'

/* General process states. */
#define STATE_SLEEP 'S'
#define STATE_WAIT  'W'
#define STATE_ZOMBIE    'Z'
#define STATE_RUN   'R'
#define STATE_STOP  'T'

/* Kernel tasks. These all run in the same address space. */
#define ASYNCM  ((int) -5) /* notifies about finished async sends */
#define IDLE    ((int) -4) /* runs when no one else can run */
#define CLOCK   ((int) -3) /* alarms and other clock functions */
#define SYSTEM  ((int) -2) /* request system functionality */
#define KERNEL  ((int) -1) /* pseudo-process for IPC and scheduling */
#define HARDWARE     KERNEL /* for hardware interrupt handlers */
```

由于每个进程中保存的信息均为自该程序启动之时到当前时间戳的统计信息，因此为了获取即时的占用信息，我们需要在极短的时间内对每个进程信息文件读取两次，并通过计算差值的方式来获得即时的占用信息。为此我们分别申明两个结构体用于存储两次读取到的信息，并在主函数中读取两次进程信息：

```
struct proc *proc = NULL, *prev_proc = NULL;

void parse_file(pid_t pid);
void parse_dir();
```

```
5.
6.  void get_procs()
7.  {
8.      struct proc *p;
9.      int i;
10.
11.     p = prev_proc;
12.     prev_proc = proc;
13.     proc = p;
14.
15.     if (proc == NULL) {
16.         proc = malloc(nr_total * sizeof(proc[0])); //Allocate a new process structure
17.         //Allocate failed
18.         if (proc == NULL) {
19.             fprintf(stderr, "Out of memory!\n");
20.             exit(1);
21.         }
22.     }
23.     //Initialize all the entry ranging in the total process+task num
24.     for (i = 0; i < nr_total; i++)
25.         proc[i].p_flags = 0;
26.
27.     parse_dir();
28. }
29.
30. void perform_top(){
31.     //Other codes omitted
32.     getkinfo();
33.     get_procs();
34.     if (prev_proc == NULL)
35.         get_procs();
36.     //Other codes omitted
37. }
```

一个进程的 CPU 占用时间可以从两次读取到的 CPU 周期之差得到，不过需要注意的是，MINIX 进程信息文件中保存的分别是周期的高 32 位及低 32 位，我们需要首先将他们拼接成一个完整的 64 位整数才能进行后续的计算：

```
1.  uint64_t make_cycle(unsigned long lo, unsigned long hi)
2.  {
3.      return ((uint64_t)hi << 32) | (uint64_t)lo;
4.  }
```

现在我们可以来实现遍历 proc 目录的函数了：

```
1.  void parse_file(pid_t pid)
2.  {
3.      char path[MAXBUF], name[256], type, state;
4.      int version, endpt, effuid;
5.      unsigned long cycles_hi, cycles_lo;
6.      FILE *fp;
7.      struct proc *p;
8.      int i;
9.
10.     sprintf(path, "/proc/%d/psinfo", pid);
11.
12.     if ((fp = fopen(path, "r")) == NULL)
13.         return;
14.
15.     if (fscanf(fp, "%d", &version) != 1) {
16.         fclose(fp);
17.         return;
18.     }
```

```
19.
20.     if (fscanf(fp, " %c %d", &type, &endpt) != 2) {
21.         fclose(fp);
22.         return;
23.     }
24.
25.     slot++;
26.
27.     if(slot < 0 || slot >= nr_total) {
28.         fprintf(stderr, "Unreasonable endpoint number %d\n", endpt);
29.         fclose(fp);
30.         return;
31.     }
32.
33.     p = &proc[slot];
34.
35.     if (type == TYPE_TASK)
36.         p->p_flags |= IS_TASK;
37.     else if (type == TYPE_SYSTEM)
38.         p->p_flags |= IS_SYSTEM;
39.
40.     p->p_endpoint = endpt;
41.     p->p_pid = pid;
42.
43.     if (fscanf(fp, " %255s %c %d %d %lu %*u %lu %lu",
44.                name, &state, &p->p_blocked, &p->p_priority,
45.                &p->p_user_time, &cycles_hi, &cycles_lo) != 7) {
46.
47.         fclose(fp);
48.         return;
49.     }
50.
51.     strncpy(p->p_name, name, sizeof(p->p_name)-1);
52.     p->p_name[sizeof(p->p_name)-1] = 0;
53.
54.     if (state != STATE_RUN)
55.         p->p_flags |= BLOCKED;
56.     p->p_cpucycles[0] = make_cycle(cycles_lo, cycles_hi);
57.     p->p_memory = 0L;
58.
59.     if (!(p->p_flags & IS_TASK)) {
60.         int j;
61.         if ((j=fscanf(fp, " %lu %*u %*u %*c %*d %*u %u %*u %d %*c %*d %*u",
62.                       &p->p_memory, &effuid, &p->p_nice)) != 3) {
63.
64.             fclose(fp);
65.             return;
66.         }
67.
68.         p->p_effuid = effuid;
69.     } else p->p_effuid = 0;
70.
71.     for(i = 1; i < CPUTIMENAMES; i++) {
72.         if(fscanf(fp, " %lu %lu",
73.                   &cycles_hi, &cycles_lo) == 2) {
74.             p->p_cpucycles[i] = make_cycle(cycles_lo, cycles_hi);
75.         } else  {
76.             p->p_cpucycles[i] = 0;
77.         }
78.     }
79.
80.     if ((p->p_flags & IS_TASK)) {
81.         if(fscanf(fp, " %lu", &p->p_memory) != 1) {
82.             p->p_memory = 0;
83.         }
84.     }
```

```
85.
86.     p->p_flags |= USED;
87.
88.     fclose(fp);
89. }
90.
91. void parse_dir()
92. {
93.     DIR *p_dir; //Pointer of directory
94.     struct dirent *p_ent; //Info of the directory
95.     pid_t pid; //Name of sub directory(PID)
96.     char *end;
97.
98.     if ((p_dir = opendir(INFOPATH)) == NULL) {
99.         exit(1);
100.           }
101.
102.          //Traverse the directory
103.          for (p_ent = readdir(p_dir); p_ent != NULL; p_ent = readdir(p_dir)) {
104.              pid = strtol(p_ent->d_name, &end, 10); //Get the name of sub directory
105.
106.              if (!end[0] && pid != 0)
107.                  parse_file(pid);
108.          }
109.
110.          closedir(p_dir);
111.      }
```

到此，我们已经将所需要的进程信息结构化存储至了结构体中，接下来我们来计算 CPU 占用率。CPU 的总使用时间为用户进程、系统进程及空闲进程的占用时间之和，由此可知要得到 CPU 占用率，我们只要统计出总使用时间、用户进程占用时间和系统进程占用时间即可。对于每个进程，我们可以利用其 CPU 周期之差算出其滴答：

```
1.  uint64_t cputicks(struct proc *p1, struct proc *p2, int timemode)
2.  {
3.      int i;
4.      uint64_t t = 0;
5.      for(i = 0; i < CPUTIMENAMES; i++) {
6.          if(!CPUTIME(timemode, i))
7.              continue;
8.          if(p1->p_endpoint == p2->p_endpoint) {
9.              t = t + p2->p_cpucycles[i] - p1->p_cpucycles[i];
10.         } else {
11.             t = t + p2->p_cpucycles[i];
12.         }
13.     }
14.
15.     return t;
16. }
```

随后我们遍历所有的有效进程，再结合进程类型及进程状态，即可得到所要的三个信息，再通过用户进程占用时间加系统进程占用时间与 CPU 总使用时间做比值，即可得到最终的 CPU 占用率：

```
1.  float print_procs(struct proc *proc1, struct proc *proc2, int cputimemode) {
2.      int p, nprocs;
3.      uint64_t systemticks = 0;
4.      uint64_t userticks = 0;
5.      uint64_t total_ticks = 0;
6.      static struct tp *tick_procs = NULL;
7.
```

```
    if (tick_procs == NULL) {
        tick_procs = malloc(nr_total * sizeof(tick_procs[0]));

        if (tick_procs == NULL) {
            fprintf(stderr, "Out of memory!\n");
            exit(1);
        }
    }

    for (p = nprocs = 0; p < nr_total; p++) {
        uint64_t uticks;
        if (!(proc2[p].p_flags & USED))
            continue;
        tick_procs[nprocs].p = proc2 + p;
        tick_procs[nprocs].ticks = cputicks(&proc1[p], &proc2[p], cputimemode);
        uticks = cputicks(&proc1[p], &proc2[p], 1);
        total_ticks = total_ticks + uticks;
        if(!(proc2[p].p_flags & IS_TASK)) {
            if(proc2[p].p_flags & IS_SYSTEM)
                systemticks = systemticks + tick_procs[nprocs].ticks;
            else
                userticks = userticks + tick_procs[nprocs].ticks;
        }

        nprocs++;
    }

    if (total_ticks == 0)
        return 0.0;

    return 100.0 * (systemticks + userticks) / total_ticks;
}

void perform_top(){
    //Other codes omitted
    float idle = print_procs(prev_proc, proc, 1);
    fprintf(stdout, "CPU Usage: %f%%\n", idle);
}
```

9. 运行结果

我们使用实验要求中的测试用例在 MINIX 环境下对 Shell 进行测试：

```
# ./shell
shell: /root % cd your
shell: /root/your % ls -a -l
total 3112
drwxr-xr-x  2 root  operator     576 Mar 20 21:52 .
drwxr-xr-x  3 root  operator    1984 Mar 21 21:47 ..
-rw-r--r--  1 root  operator       0 Mar 18 15:42 a.txt
-rwxr-xr-x  1 root  operator  774740 Mar 20 21:52 core.239
-rwxr-xr-x  1 root  operator  778836 Mar 20 21:52 core.241
-rw-------  1 root  operator     544 Mar 21 21:50 result.txt
-rw-r--r--  1 root  operator      19 Mar 20 19:37 sig_par.c
-rw-r--r--  1 root  operator     341 Mar 20 19:39 signal_test.c
-rw-r--r--  1 root  operator     345 Mar 20 19:39 signal_test_chld.c
shell: /root/your % ls -a -l > result.txt
shell: /root/your %
```

```
shell: /root/your % grep a < result.txt
total 3104
drwxr-xr-x  2 root  operator     576 Mar 20 21:52 .
drwxr-xr-x  3 root  operator    1984 Mar 21 21:47 ..
-rw-r--r--  1 root  operator       0 Mar 18 15:42 a.txt
-rwxr-xr-x  1 root  operator  774740 Mar 20 21:52 core.239
-rwxr-xr-x  1 root  operator  778836 Mar 20 21:52 core.241
-rw-------  1 root  operator       0 Mar 21 23:58 result.txt
-rw-r--r--  1 root  operator      19 Mar 20 19:37 sig_par.c
-rw-r--r--  1 root  operator     341 Mar 20 19:39 signal_test.c
-rw-r--r--  1 root  operator     345 Mar 20 19:39 signal_test_chld.c
shell: /root/your % ls -a -l | grep a
total 3112
drwxr-xr-x  2 root  operator     576 Mar 20 21:52 .
drwxr-xr-x  3 root  operator    1984 Mar 21 21:47 ..
-rw-r--r--  1 root  operator       0 Mar 18 15:42 a.txt
-rwxr-xr-x  1 root  operator  774740 Mar 20 21:52 core.239
-rwxr-xr-x  1 root  operator  778836 Mar 20 21:52 core.241
-rw-------  1 root  operator     544 Mar 21 23:58 result.txt
-rw-r--r--  1 root  operator      19 Mar 20 19:37 sig_par.c
-rw-r--r--  1 root  operator     341 Mar 20 19:39 signal_test.c
-rw-r--r--  1 root  operator     345 Mar 20 19:39 signal_test_chld.c
shell: /root/your % vi result.txt &
shell: /root/your % ex/vi: Vi's standard input and output must be a terminal

shell: /root/your % mytop
Total: 1048124K, Free: 1012984K, Cached: 11304K
CPU Usage: 0.356317%
shell: /root/your % history
cd your
ls -a -l
ls -a -l > result.txt
result.txt < grep a
grep a < result.txt
ls -a -l | grep a
vi result.txt &

vi result.txt &

mytop
history
shell: /root/your %
```

可以发现，程序的行为均与预期相同。除此之外，我们再结合多重管道、输出重定向及后台运行符对 Shell 进行更为复杂的测试：

```
shell: /root/your % ls -a -l | grep a | grep c | grep r
-rwxr-xr-x  1 root  operator  774740 Mar 20 21:52 core.239
-rwxr-xr-x  1 root  operator  778836 Mar 20 21:52 core.241
-rw-r--r--  1 root  operator      19 Mar 20 19:37 sig_par.c
-rw-r--r--  1 root  operator     341 Mar 20 19:39 signal_test.c
-rw-r--r--  1 root  operator     345 Mar 20 19:39 signal_test_chld.c
shell: /root/your % ls -a -l | grep a | grep c | grep r > b.txt &
shell: /root/your % _
```

Shell 同样可以正确的解析命令并执行相应的操作。

五、**总结**

在本实验中，我们从零开始在 MINIX 环境下完整实现了一个基本的 Shell 终端，这其中综合了系统调用、文件管理、进程管理及 I/O 交互的大量理论知识。通过该实验，极大的加深了我们对这些系统知识的理解。