华东师范大学数据科学与工程学院实验报告

课程名称：操作系统	年级：21 级	上机实践成绩：
指导教师：翁楚良	姓名：杨茜雅	学号：10215501435
上机实践名称：shell 及系统调用		上机实践日期：2023.3
上机实践编号：1	组号：	上机实践时间：2023.3

一、 实验目的

学习 Shell，系统编程，实现一个基本的 Shell

二、 实验任务

实现一个基本的 Shell，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

物理机：Windows10

虚拟机：Minix

虚拟机软件：VMware Workstation pro 17

四、 实验过程

1、 先验知识

Shell 是什么？

Shell 主体结构是一个 while 循环，不断地接受用户键盘输入行并给出反馈。Shell 将输入行分解成单词序列，根据命令名称分为二类分别处理，即 shell 内置命令（例如 cd，history，exit）和 program 命令（例如/bin/目录下的 ls，grep 等）。识别为 shell 内置命令后，执行对应操作。接受 program 命令后，利用 minix 自带的程序创建一个或多个新进程，并等待进程结束。如果末尾包含&参数，Shell 可以不等待进程结束，直接返回。

2、实验思路：

（1）内置命令：

1、cd：调用 chdir 函数改变目录，将当前工作目录切换到从命令行传进来的路径。

2、exit：exit（0）退出 shell 的 while 循环。

3、history：用一个二维数组 his[][]保存 shell 中输入的命令，然后根据 history n 的参数输出相应的历史命令。

4、mytop：在 Minix 系统的 proc 文件夹中通过 fopen/fscanf 获取进程信息，输出内存使用情况和 CPU 使用百分比。

（2）program 命令：

1、重定向 覆盖写 > ： 调用 open 函数得到文件描述符 fd（以清空文件内容的方式），调用 dup2(fd,1)函数将文件描述符映射到标准输出。

2、重定向 追加写>> ：调用 open 函数得到文件描述符 fd （以保留文件内容的方式），调用 dup2(fd,1)的方式将文件描述符映射到标准输出。

3、重定向 文件输入 < ：调用 open 函数得到文件描述符 fd，调用 dup2(fd,0)函数将文件描述符映射到标准输入。

4、后台 ： 将子进程的标准输入、标准输出、标准错误输出映射到/dev/null，屏蔽键盘和控制台。调用 signall(SIGCHLD,SIG_IGN)函数是的 Minix 接管此进程，shell 忽略不用等待子进程结束直接运行下一条命令。

5、管道 利用 pipe 函数创建一个管道 fd[2]，在子进程中调用 close(1)和 dup(fd[1])将管道的写端映射到标准输出，将进程的输出写入管道，执行管道的前部分指令；在父进程中等待子进程结束并回收，调用 close(0)和 dup(fd[0])将管道的读端映射到标准输入，从管道中读入数据，再执行管道的后部分指令。

3、**代码实现**

（1）宏定义，定义全局变量以及函数声明

```
#define MAXLINE 100 //最大命令数量
#define MAXARGS 100//最大输入命令字符数
#define M 100 //每条命令的最大长度

char his[M][M];//用二维数组保存 shell 中输入的命令
int his_cnt = 0; //历史命令计数
char *path = NULL;//初始化指向 NULL

void doCommand(char *cmdline);
int parseline(const char *cmdline, char **argv);
int builtin_cmd(char **argv);
void pipe_line(char *process1[], char *process2[]);
void mytop();
```

(2)主函数

获取当前路径，打印 shell 提示符，将输入命令到 cmdline 数组中并且将命令储存到历史命令二维数组 his[M][M]中，再调用 doCommand 函数解析命令（看是内置还是 program 命令），最后刷新缓冲区。

```
int main(int argc, char **argv)
{
	char c;
	char cmdline[MAXLINE];

	while (1)
```

```
	{
		path = getcwd(NULL, 0);//获取当前工作路径
		printf("10215501435_shell>%s# ", path);
		fflush(stdout);//打印 shell 提示符
		if (fgets(cmdline, MAXLINE, stdin) == NULL)
		{//输入命令到 cmdline 数组中
			continue;
		}
		for (int i = 0; i < M; i++)
		{//将命令储存到历史命令二维数组 his 中
			his[his_cnt][i] = cmdline[i];
		}
		his_cnt = his_cnt + 1;
		doCommand(cmdline);//解析命令
		fflush(stdout);
	}
	exit(0);
}
```

（3）doCommand(char *cmdline)函数

调用 parseline 函数对命令进行分割，获取命令和参数并判断是不是后台命令。调用 builtin_cmd（argv）函数，如果是内置命令则直接返回，如果不是内置命令则根据参数列表 argv 判断所属情况，将 program 命令分成六个 case，每个情况有一个专门的 case_command 方便后续跳转到对应的 switch-case 结构里去。

```
void doCommand(char *cmdline)
//内置命令、program 命令、后台运行
{
	char *argv[MAXARGS];
	char buf[MAXLINE];
	int bg;
	pid_t pid;
	char *file;
	int fd;
	int status;
	int case_command = 0;
	strcpy(buf, cmdline);
```

```
//parseline 函数可以对命令进行分割，获取命令和参数来判断是不是后台命令
	if ((bg = parseline(buf, argv)) == 1)
	{ //后台
		case_command = 4;
	}
	if (argv[0] == NULL)
	{
		return;
	}

	if (builtin_cmd(argv))
		return; //如果是内置命令则直接返回
//根据参数列表 argv 判断所属情况，给一个 case_command 去到 switch-case 结构中
	int i = 0;
	for (i = 0; argv[i] != NULL; i++)
	{
		if (strcmp(argv[i], ">") == 0)
		{
			if (strcmp(argv[i + 1], ">") == 0)
			{
				case_command = 5;
				break;
			}
			case_command = 1;
			file = argv[i + 1];
			argv[i] = NULL;
			break;
		}
	}

	for (i = 0; argv[i] != NULL; i++)
	{
		if (strcmp(argv[i], "<") == 0)
		{
```

```
                case_command = 2;
                file = argv[i + 1];//利用参数列表得到符号后的文件名
                printf("filename=%s\n", file);
                argv[i] = NULL;//执行 null 之前的所有命令
                break;
            }
        }
        char *leftargv[MAXARGS];
        for (i = 0; argv[i] != NULL; i++)
        {
            if (strcmp(argv[i], "|") == 0)
            {
                case_command = 3;
                argv[i] = NULL;
                int j;
                for (j = i + 1; argv[j] != NULL; j++)
                {
                    leftargv[j - i - 1] = argv[j];
                }//得到管道前面和后面的命令参数
                leftargv[j - i - 1] = NULL;
                break;
            }
        }
```

六个 case 分别为：

Case 0:未出现管道、重定向、后台等命令，则直接 fork 一个子进程然后 execvp 运行，对 shell 中的命令进行处理，处理完以后新进程结束，waitpid 等待回收进程。

```
//program 命令有六种 case
        switch (case_command)
        {
        case 0://未出现管道，重定向，后台命令
            if ((pid = fork()) == 0)
            {//fork 一个子进程然后 execvp 运行
                execvp(argv[0], argv);
            //对 shell 中的命令进行处理
```

```
                exit(0);
            }//处理完以后新进程结束，waitpid 等待回收进程
            if (waitpid(pid, &status, 0) == -1)
            {
                printf("error\n");
            }
            break;
```

Case 1:重定向输出-覆盖写>:fork 子进程，调用 open 函数的同属得到 file 文件的文件描述符 fd，参数中运用 O_TRUNC（以该数学打开文件时，如果这个文件本来是有内容的，则本来的内容会被丢弃，相当于清空）。7 相当于 4+2+1 即拥有读、写和执行的权限。接着调用 dup2(fd，1)函数将 file 文件的文件描述符映射到标准输出，close(fd)函数关闭该文件的文件描述符，execvp 函数执行重定向前的指令，最后父进程中调用 waitpid 函数等待子进程结束并回收。

```
case 1:
            /*包含重定向输出 覆盖写>*/
            if ((pid = fork()) == 0)
            {//fork 一个子进程
            //调用 open 函数得到 file 文件描述符 fd
            //文件描述符 0、1、2 与进程的标准输入，标准输出，标准错误输出相对应
            //open 函数参数：需要读取的文件所在路径，参数 2 以读、写、读写的方式打开文件/当文件不存在时，创建文件，在文件末尾追加
            //Linux 中每一个文件被创建出来都自带权限，分别表示用户权限、组权限、其他权限
            //O_TRUNC 属性去打开文件时，如果这个文件的本来是有内容的，则原来的内容会被丢弃。
                fd = open(file, O_RDWR | O_CREAT | O_TRUNC, 7777);
                //可读可写，若文件存在，则长度被截为 0，属性不变（将打开文件长度截短为零，相当于清空，不能与 O_RDONLY 搭配使用）
                //7777 最大的权限

                if (fd == -1)
                {
                    printf("open %s error!\n", file);
                }
                dup2(fd, 1);//将结果输出到 file，将 file 文件描述符映射到标准输出
                close(fd);//close 函数参数指需要关闭的文件的文件描述符
                execvp(argv[0], argv);//执行重定向前的指令
```

```
            exit(0);
        }
        if (waitpid(pid, &status, 0) == -1)
        {//父进程中 waitpid 等待子进程结束并且回收
            printf("error\n");
        }
        break;
```

Case 2:重定向输入 文件输入 <: file = argv[i+1]利用参数列表得到符号后的文件名，在 fork 的子进程中调用 open 函数获得 file 的文件描述符 fd，调用 dup2(fd,0)将 file 映射到标准输入，close(fd)关闭闲置的文件描述符防止混乱，调用 execvp 执行符号前面的指令，最后父进程中调用 waitpid 函数等待子进程结束并回收。

```
case 2:
        /*包含重定向输入    文件输入<*/
        if ((pid = fork()) == 0)
        {//调用 open 得到 file 的文件描述符
            fd = open(file, O_RDONLY);
            dup2(fd, 0);//将 file 映射到标准输入
            close(fd);//关闭闲置的文件描述防止混乱
            execvp(argv[0], argv);//执行指令
            exit(0);
        }
        if (waitpid(pid, &status, 0) == -1)
        {//父进程 waitpid 等待子进程结束并回收
            printf("error\n");
        }
        break;
```

Case 3:管道 | ：在之前得到 case_command=3 的代码段中已经实现了分别得到管道前面和后面的命令参数的功能。在 case 3 中 argv 和 leftargv 在子进程中通过 pipeline 函数实现管道。

```
    case 3:
        /*命令包含管道    |*/
        if ((pid = fork()) == 0)
        {//前面得到管道前面和后面的命名参数然后在子进程中 pipeline 函数实现管道
            pipe_line(argv, leftargv);
        }
```

```
        else
        {
            if (waitpid(pid, &status, 0) == -1)
            {//老样子回收回收
                printf("error\n");
            }
        }
        break;
```

Case 4:后台 &：fork 一个子进程，子进程调用 signal(SIGCHLD,SIG_IGN)函数使 Minix 接管此进程（函数作用为子进程发送一个信号给父进程，忽略子进程不用等他终止），调用 open 函数得到/dev/null/的文件描述符，将其映射到标准输入标准输出和标准错误输出，最后执行命令。

```
    case 4: //后台运行 &
        if ((pid = fork()) == 0)
        {
            int fd1 = open("/dev/null", O_RDONLY);
            dup2(fd1, 0);//映射到标准输入
            dup2(fd1, 1);//映射到标准输出
            dup2(fd1, 2);
            execvp(argv[0], argv);//执行命令
            signal(SIGCHLD, SIG_IGN);//minix 接管此进程，子进程结束了发一个信号给父进程，忽略子进程不用等待它终止
            exit(0);
            }
        else {
            printf("[process id %d]\n", pid);            //若为后台程序，则输出进程号
        }
        //不等待结束
        break;
```

Case 5:追加写>>:该功能与覆盖写的实现逻辑基本一致，最重要的不同点是在 open 函数中以 O_APPEND 属性去打开文件（作用是保留不是清空，即如果这个文件中本来就是有内容的，则新写入的内容会被追加在原来内容的后面），接着调用 dup2(fd,1)函数将 file 映射到标准输出，execvp 函数执行重定向符号前的指令，父进程中 waitpid 函数等待子进程结束并回收。

```
    case 5: //追加写>>，与覆盖写显著的区别就是文件的方式是保留不是清空
        if ((pid = fork()) == 0)
```

```
            {//open 得到文件描述符 fd
                  fd = open(file, O_RDWR | O_CREAT | O_APPEND, 7777);
                  //O_APPEND 属性去打开文件时，如果这个文件中本来就是有内容的，则新写入的内
容会被接续到原来内容的后面
                  if (fd == -1)
                  {
                        printf("open %s error!\n", file);
                  }
                  dup2(fd, 1);//映射到标准输出
                  close(fd);
                  execvp(argv[0], argv);
                  exit(0);
            }
            if (waitpid(pid, &status, 0) == -1)
            {
                  printf("error\n");
            }
            break;

      default:
            break;
      }
      return;
}
```

（4）parseline 函数
作用：解析命令得到参数序列，判断是前台作业还是后台作业。
Char *strtok（char *s, char *delim）实现原理：将分隔符出现的地方改为’\0’，根据空格对命令进行分割，得到 argv 参数序列，根据最后一个字符是不是&判断是否是后台命令。

```
int parseline(const char *cmdline, char **argv)
{
      //解析命令
      static char array[MAXLINE];
      char *buf = array;
      int argc = 0;
      int bg;
```

```
	strcpy(buf, cmdline);
	buf[strlen(buf) - 1] = ' ';//将行末尾的回车改为空格
	while (*buf && (*buf == ' '))
		buf++;
//char *strtok(char *s,char *delim) 实现原理：将分隔符出现的地方改为'\0'
//根据空格对命令进行分割
	char *s = strtok(buf, " ");
	if (s == NULL)
	{
		exit(0);
	}
	argv[argc] = s;//得到 argv 参数序列
	argc++;
	while ((s = strtok(NULL, " ")) != NULL)
	{//参数设置为 NULL，从上一次读取的地方继续
		argv[argc] = s;
		argc++;
	}
	argv[argc] = NULL;

	if (argc == 0)
		return 1;
//根据最后一个字符是不是&判断是不是后台命令
	if ((bg = (*argv[(argc)-1] == '&')) != 0)
	{
		argv[--(argc)] = NULL;
	}
	return bg;
}
```

（5）builtin_cmd 函数
作用：实现内置命令

1、 exit：直接退出 main 函数的 while 循环
2、 mytop：调用实现的 mytop 函数
3、 cd：调用 chdir 函数，根据参数改变工作目录，将当前工作目录切换到其参数
getcwd 函数获取当前的工作路径，用 path 指向当前的工作目录
4、 history:如果只输入 history 而无参数的话则会提醒输入参数，获取参数以后输出二维数组中的命令历史。

```
int builtin_cmd(char **argv)
{
	//内置命令
	//关于 exit 退出功能
	if (!strcmp(argv[0], "exit"))
	{
		exit(0);//直接退出 main 函数的 while 循环
	}
	//关于 mytop 功能
	if (!strcmp(argv[0], "mytop"))
	{
		mytop();
		return 1;
	}
	//关于 cd 功能
	if (!strcmp(argv[0], "cd"))
	{
		if (!argv[1])
		{ // cd 后面没有任何输入
			argv[1] = ".";
		}
		int ret = chdir(argv[1]); //根据参数改变工作目录,将当前工作目录切换到 argv[1](从命令行传进来的路径)
		//函数原型 int chdir(const char *path)
		if (ret < 0)
		{
			printf("No such directory!\n");
		}
		else
		{
```

```
                path = getcwd(NULL, 0); //path 指向当前的工作目录，调用 getcwd 函数获取当前
的工作路径
                //char *getcwd(char *buf, size_t size);
            }
            return 1;
      }
//对于 history
      if (!strcmp(argv[0], "history"))
      {
            if (!argv[1])
            { //只输入 history
                for (int j = 1; j <= his_cnt; j++)
                {
                      printf("%d ", j);
                      puts(his[j - 1]);
                }
            }
            else
            {
                int t = atoi(argv[1]);
                if (his_cnt - t < 0)
                {//输入 history 之后的数字
                      printf("please confirm the number below %d\n", his_cnt);
                }
                else
                {
                      for (int j = his_cnt - t; j < his_cnt; j++)
                      {
                            printf("%d ", j + 1);
                            puts(his[j]);//输出二维数组中的命令历史
                      }
                }
            }
            return 1;
```

```
	}


	return 0;
}
```

（6）pipe_line 函数

调用 pipe 函数创建一个管道 fd[2]，在 fork 的子进程中关闭管道读端 fd[0]和文件描述符 1，将管道的写端映射到标准输出，再关闭管道写端以免堵塞，execvp 函数执行管道前部分指令，将进程的输出写入管道。

在父进程中，关闭管道写端 fd[1]和文件描述符 0，dup(fd[0])恢复标准输入，将读端映射到标准输入，从管道中读入数据，再关闭读端以免堵塞，等待子进程结束并且执行管道后部分指令。

```
void pipe_line(char *process1[], char *process2[])
{
	int fd[2];
	pipe(&fd[0]);
//调用 pipe 函数创建一个管道 fd[2]
	int status;
	pid_t pid = fork();
	if (pid == 0)
	{
		close(fd[0]);
		close(1);
		//关闭管道读端 fd[0]和文件描述符 1
		dup(fd[1]);//将管道的写端映射到标准输出
		close(fd[1]);//关闭管道写端以免堵塞
		execvp(process1[0], process1);//执行管道前部分指令，将进程的输出写入管道
	}
	else
	{ //父进程中
		close(fd[1]);
		close(0);//关闭管道写端 fd[1]和文件描述符 0
		dup(fd[0]);//复制文件描述符，恢复标准输入，读端映射到标准输入，从管道中读入数据
		close(fd[0]);//关闭读端以免堵塞
		waitpid(pid, &status, 0);
		execvp(process2[0], process2);//执行管道后部分指令
	}
}
```

（7）mytop 函数

以只读的方式打开/proc/meminfo ，通过buff数组查看内存信息，依次是页面大小pagesize，总页数total，空闲页数量free，最大页数量largest，缓存页数量cached。可以计算出的有总体内存大小totalMemory、空闲内存大小freeMemory、缓存大小cachedMemory。以只读方式打开/proc/kinfo ，将文件内容存进buff可以获取进程数量和任务数量。

```
void mytop() {
	FILE *fp = NULL;
	char buff[255];

	fp = fopen("/proc/meminfo", "r");	// 以只读方式打开 meminfo 文件
	fgets(buff, 255, (FILE*)fp);		// 读取 meminfo 文件内容进 buff
	fclose(fp);

	// 获取 pagesize
	int i = 0, pagesize = 0;
	while (buff[i] != ' ') {
		pagesize = 10 * pagesize + buff[i] - 48;
		i++;
	}

	// 获取 页总数 total
	i++;
	int total = 0;
	while (buff[i] != ' ') {
		total = 10 * total + buff[i] - 48;
		i++;
	}

	// 获取空闲页数 free
	i++;
	int free = 0;
	while (buff[i] != ' ') {
		free = 10 * free + buff[i] - 48;
		i++;
	}
```

```
    // 获取最大页数量 largest
    i++;
    int largest = 0;
    while (buff[i] != ' ') {
        largest = 10 * largest + buff[i] - 48;
        i++;
    }

    // 获取缓存页数量 cached
    i++;
    int cached = 0;
    while (buff[i] >= '0' && buff[i] <= '9') {
        cached = 10 * cached + buff[i] - 48;
        i++;
    }

    // 总体内存大小 = (pagesize * total) / 1024 单位 KB
    int totalMemory   = pagesize / 1024 * total;
    // 空闲内存大小 = pagesize * free) / 1024 单位 KB
    int freeMemory    = pagesize / 1024 * free;
    // 缓存大小      = (pagesize * cached) / 1024 单位 KB
    int cachedMemory = pagesize / 1024 * cached;

    printf("totalMemory   is %d KB\n", totalMemory);
    printf("freeMemory    is %d KB\n", freeMemory);
    printf("cachedMemory is %d KB\n", cachedMemory);

    /* 2. 获取内容 2
        进程和任务数量
     */
    fp = fopen("/proc/kinfo", "r");        // 以只读方式打开 kinfo 文件
    memset(buff, 0x00, 255);                  // 格式化 buff 字符串
    fgets(buff, 255, (FILE*)fp);             // 读取 kinfo 文件内容进 buff
    fclose(fp);
```

```
    // 获取进程数量
    int processNumber = 0;
    i = 0;
    while (buff[i] != ' ') {
        processNumber = 10 * processNumber + buff[i] - 48;
        i++;
    }
    printf("processNumber = %d\n", processNumber);

    // 获取任务数量
    i++;
    int tasksNumber = 0;
    while (buff[i] >= '0' && buff[i] <= '9') {
        tasksNumber = 10 * tasksNumber + buff[i] - 48;
        i++;
    }
    printf("tasksNumber = %d\n", tasksNumber);
    return;
}
```

五、 代码运行结果截图

1、 编译&运行

```
# clang shell.c -o shell.o
# ./shell.o
10215501435_shell>/root# _
```

2、 ls -a -l

```
10215501435_shell>/root# ls -a -l
total 136
drwxr-xr-x   3 root  operator    640 Mar 26 15:55 .
drwxr-xr-x  17 root  operator   1408 Mar  2 13:51 ..
-rw-r--r--   1 root  operator     44 Sep 14  2014 .exrc
-rw-r--r--   1 root  operator    605 Sep 14  2014 .profile
-rwxr-xr-x   1 root  operator   5310 Mar 23 13:37 hello
-rw-r--r--   1 root  operator     75 Mar  2 15:17 hello.c
-rw-r--r--   1 root  operator  13801 Mar 26 15:53 shell.c
-rwxr-xr-x   1 root  operator  13260 Mar 26 15:55 shell.o
-rw-r--r--   1 root  operator      5 Mar  2 15:12 text.txt
drwxr-xr-x   3 root  operator    192 Mar 23 13:44 your
10215501435_shell>/root# _
```

3、 mytop

```
10215501435_shell>/root# mytop
totalMemory  is 260540 KB
freeMemory   is 210448 KB
cachedMemory is 23452 KB
processNumber = 256
tasksNumber = 5
10215501435_shell>/root#
```

4、 history 3

```
10215501435_shell>/root# history 3
1 ls -a -l

2 mytop

3 history 3

10215501435_shell>/root#
```

5、 cd your/path

```
10215501435_shell>/root# cd your/path
10215501435_shell>/root/your/path# ls -a -l
total 24
drwxr-xr-x  2 root  operator  192 Mar 23 15:09 .
drwxr-xr-x  3 root  operator  192 Mar 23 13:44 ..
---sr-S--t  1 root  operator  166 Mar 26 13:27 result.txt
10215501435_shell>/root/your/path# _
```

6、 ls -a -l > result.txt

```
10215501435_shell>/root# cd your/path
10215501435_shell>/root/your/path# ls -a -l
total 24
drwxr-xr-x  2 root  operator  192 Mar 23 15:09 .
drwxr-xr-x  3 root  operator  192 Mar 23 13:44 ..
---sr-S--t  1 root  operator  166 Mar 26 13:27 result.txt
10215501435_shell>/root/your/path# ls -a -l > result.txt
10215501435_shell>/root/your/path#
```

7、 vi result.txt 再 q 出去

```
total 16
drwxr-xr-x  2 root  operator  192 Mar 23 15:09 .
drwxr-xr-x  3 root  operator  192 Mar 23 13:44 ..
---sr-S--t  1 root  operator    0 Mar 26 15:58 result.txt
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
result.txt: unmodified, readonly: line 1
```

```
total 16
drwxr-xr-x  2 root  operator  192 Mar 23 15:09 .
drwxr-xr-x  3 root  operator  192 Mar 23 13:44 ..
---sr-S--t  1 root  operator    0 Mar 26 15:58 result.txt
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
:q_
```

8、 grep a <result.txt

```
~
~
~
~
~
:q
10215501435_shell>/root/your/path# grep a < result.txt
filename=result.txt
total 16
drwxr-xr-x  2 root  operator  192 Mar 23 15:09 .
drwxr-xr-x  3 root  operator  192 Mar 23 13:44 ..
---sr-S--t  1 root  operator    0 Mar 26 15:58 result.txt
10215501435_shell>/root/your/path# β
```

9、 ls -a -l | grep a

```
---sr-S--t  1 root  operator    0 Mar 26 15:58 result.txt
10215501435_shell>/root/your/path# ls -a -l | grep a
total 24
drwxr-xr-x  2 root  operator  192 Mar 23 15:09 .
drwxr-xr-x  3 root  operator  192 Mar 23 13:44 ..
---sr-S--t  1 root  operator  166 Mar 26 15:58 result.txt
10215501435_shell>/root/your/path# _
```

10、 vi result.txt &

```
---sr-S--t  1 root  operator  166 Mar 26 15:58 result.txt
10215501435_shell>/root/your/path# vi result.txt &
[process id 227]
10215501435_shell>/root/your/path#
```

11、 exit

```
[process id 227]
10215501435_shell>/root/your/path# exit
#
```

六、遇到的问题

在实现后台功能的时候，我遇到一个与 terminal 有关的 warning，后来通过调用三次 dup2 函数将其解决，为什么会遇到 warning 是因为在我映射的时候忘记映射到标准错误输出。

实验并没有实现多重管道功能，该实验中的管道部分代码有参考老师 os-2 课件最末尾的图例和代码。