操作系统 实验报告1

温兆和 10205501432

实验背景

在本次实验中，我们要实现一个Unix Shell，它要能够解析如下命令或符号：

- 带参数的程序运行功能
- 重定向功能（覆盖写 `>` 、追加写 `>>` 、文件输入 `<` ）
- 管道符号 `|`
- 后台符号 `&`
- 工作路径移动命令 `cd`
- 程序运行统计 `mytop`
- 退出命令 `exit`
- 显示最近执行的若干条指令 `history n`

实验过程

下面，我们来展示 `eval` 、 `waitpd` 和 `builtin_cmd` 三个函数的设计过程、遇到的bug和收获的知识点。

`eval`

`eval` 函数的作用是解析并命令行：如果命令是内置命令，就转交给 `builtin_cmd` 函数执行；否则就开一个新的子进程并交由子进程运行相应程序，最后再根据命令是前台作业还是后台作业来决定我们是等待子进程在前台运行完还是仅仅在前台输出相应信息，一边让进程在后台执行一边等待用户输入下一条指令。此外，在运行程序命令的前提下， `eval` 函数还要解析重定向和管道符号，并由此执行 `dup2` 和 `pipe` 系统调用，改变当前进程的标准输入输出，使进程的输入或者输出变成另一个文件或者另一个进程（通过管道）。

首先，解析命令行并判断它是前台作业还是后台作业：

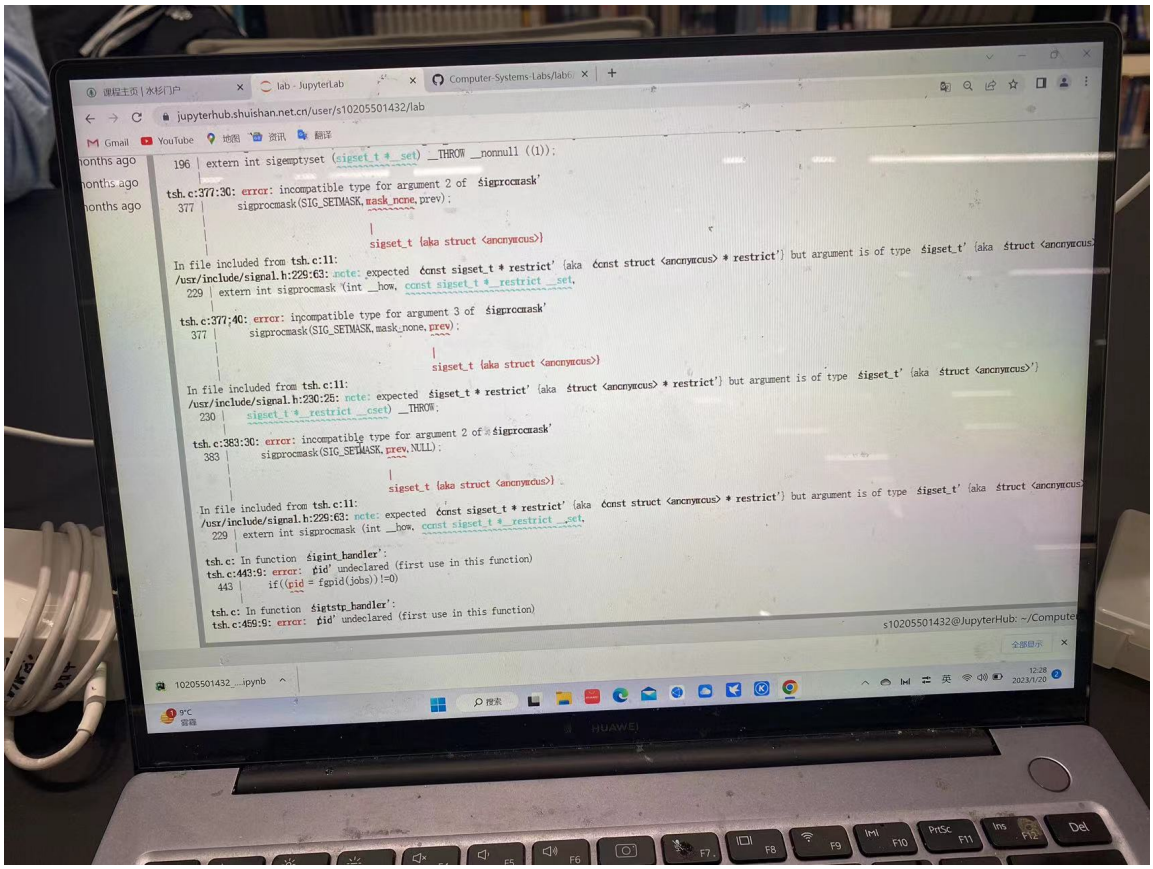
In [ ]:
```
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    char buf[MAXLINE];
    int bg;
    int state;
    pid_t pid;
    sigset_t mask_all,mask_one,prev;
    strcpy(buf,cmdline);
    bg = parseline(buf,argv);
```

```
    if(argv[0]==NULL)
    {
        return;
    }
    state = bg ? BG : FG;
```

然后，设置三个信号集合：全集、`SIGCHILD`单信号集和之前被阻塞的信号的集合。接下去我们会用到信号阻塞的方法，需要阻塞信号的原因待会儿会讲：

In [ ]:
```
sigfillset(&mask_all);
    sigemptyset(&mask_one);
    sigaddset(&mask_one,SIGCHLD);
```

值得注意的是，《异常控制流》这一章中所有有关信号的函数在输入信号集合时都要用到引用传值。这就引出了我在整个实验中遇到最多的bug，就是把诸如`sigfillset(&mask_all);`写成`sigfillset(mask_all);`，导致最后`make`这段代码的时候出现了很多很多这样的报错：



如果是内置命令，就直接交给`builtin_cmd`函数处理；否则就创建一个子进程，创建一个新的进程组并设置其id为进程pid。这里有两点值得注意：

- `builtin_cmd`函数有两个功能：执行内置命令和返回命令是否是内置命令，这一点很巧妙；
- 在创建子进程之前需要把`SIGCHILD`信号阻塞掉，这样做的原因是防止子进程在父进程运行到`addjob`之前就结束并发送`SIGCHILD`信号回收子进程，这在整个shell里被大量

用到。

此后，处理重定向中的覆盖写符号。如果覆盖写符号后没有任何参数（即目标文件的文件名），就直接报错；否则，我们就直接创建相应文件（原有同名文件会被覆盖），把进程的标准输出指向这个文件。

In [ ]:
```
if(!builtin_cmd(argv))
    {
        sigprocmask(SIG_BLOCK,&mask_one,&prev);
        if((pid = fork())==0)
        {
            sigprocmask(SIG_SETMASK,&prev,NULL);
            if(setpgid(0,0)<0)
            {
                unix_error("Setpid error");
                exit(0);
            }
            for (int i=0;argv[i]!=NULL;i++)
            {
                if (!strcmp(argv[i],">"))
                {
                    if(argv[i+1]==NULL)
                    {
                        perror("No output file!");
                        exit(1);
                    }
                    else
                    {
                        argv[i]=NULL;
                        int fd = creat(argv[i+1],0666);
                        dup2(fd,1);
                        close(fd);
                        break;
                    }
                }
            }
```

接着，处理重定向中的追加写符号。这里的总体实现思路与覆盖写相同，唯一的不同点在于这里调用的函数是 `int fd = open(argv[i+1],O_RDWR|O_APPEND,0666);` ，把 `open` 函数的参数设置为 `O_RDWR|O_APPEND` ，这样进程的输出结果就会被写在原有文件的最后，而不是覆盖原来的文件的内容。

In [ ]:
```
for (int i=0;argv[i]!=NULL;i++)
            {
                if (!strcmp(argv[i],">>"))
                {
                    if(argv[i+1]==NULL)
                    {
                        perror("No output file!");
                        exit(1);
                    }
                    else
                    {
```

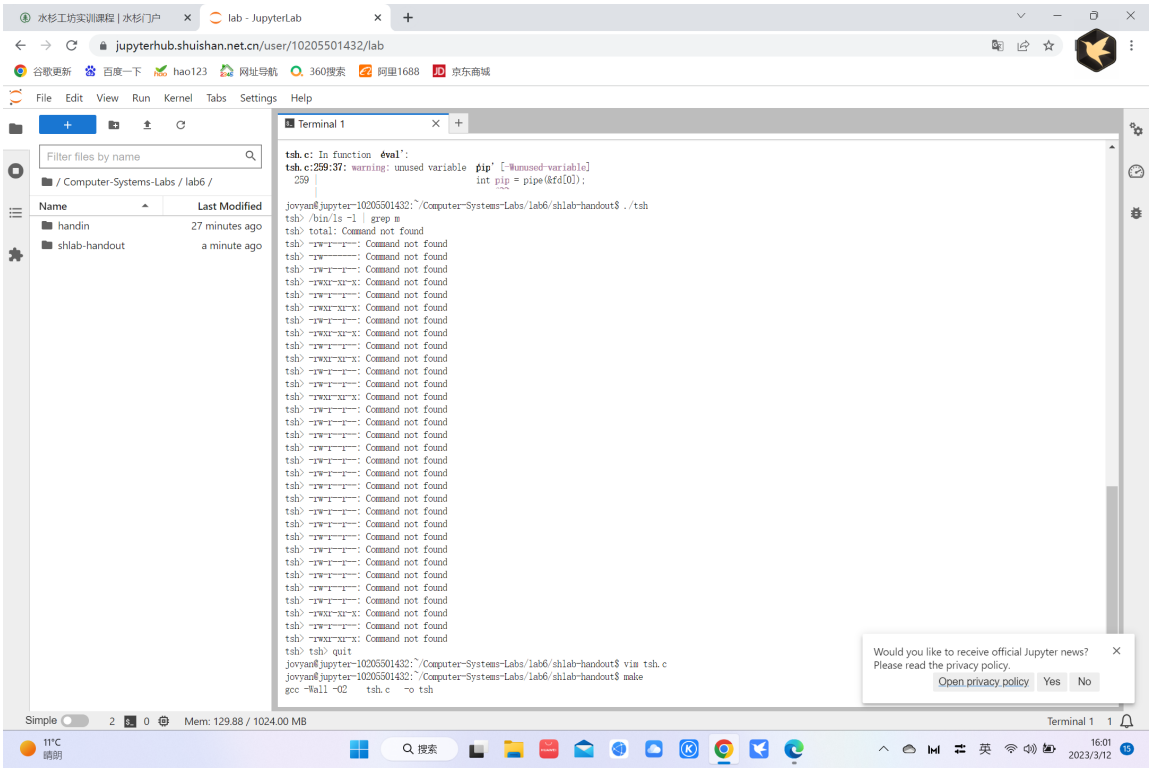
```
                        argv[i]=NULL;
                        int fd = open(argv[i+1],O_RDWR|O_APPEND,0666);
                        dup2(fd,1);
                        close(fd);
                        break;
                    }
                }
            }
```

再处理重定向中的文件输入符号。这里的总体实现思路与覆盖写相同，唯一的不同点在于它调用 `dup2` 系统调用是把进程的标准输入改成目标文件，而不是标准输出。此外，它不是靠 `creat` 创建文件，而是通过 `open` 系统调用以只读方式打开现有文件。

In [ ]:
```
for (int i=0;argv[i]!=NULL;i++)
                    {
                            if (!strcmp(argv[i],"<"))
                            {
                                    if(argv[i+1]==NULL)
                                    {
                                            perror("No output file!");
                                            exit(1);
                                    }
                                    else
                                    {
                                            argv[i]=NULL;
                                            int fd = open(argv[i+1],O_RDONLY);
                                            dup2(fd,0);
                                            close(fd);
                                            break;
                                    }
                            }
                    }
```

最后还要处理管道符号。首先，定义两个存放字符串的数组，把管道符号前面的进程参数和管道符号后面的进程参数分别存放在里面。然后，调用 `pipe` 系统调用建立管道。接下来开一个子进程，把子进程的标准输出指向管道，并在子进程中执行管道符号前面的那个进程。而父进程需要等待子进程执行完之后，将其标准输入指向管道并运行管道符号后面的进程。这样，前一个进程的输出就能作为后一个进程的输入了。

值得注意的是，这里在加载并运行程序时最好使用 `execvp` 而不是 `execve`，否则就会出现下面这样的情况：



In [ ]:

```
for (int i=0;argv[i]!=NULL;i++)
            {
                char *program1[MAXARGS];
                char *program2[MAXARGS];
                for (int ii=0;ii<MAXARGS;ii++)
                {
                    program1[ii]=NULL;
                    program2[ii]=NULL;
                }
                if (!strcmp(argv[i],"|"))
                {
                    is_pipe++;
                    for (int ii=0;ii<i;ii++)
                    {
                        program1[ii]=argv[ii];
                    }
                    for (int ii=i+1;argv[ii]!=NULL;ii++)
                    {
                        program2[ii-i-1]=argv[ii];
                    }
                    int fd[2];
                    int pip = pipe(&fd[0]);
                    pid_t pid2;
                    if((pid2=fork())==0)
                    {
                        close(fd[0]);
                        dup2(fd[1],1);
                        execvp(program1[0],program1);
                    }
                    else
                    {
                        wait(NULL);
```

```
                    }
                    close(fd[1]);
                    dup2(fd[0],0);
                    execvp(program2[0],program2);
                    break;
                }
            }
```

在没有管道符号的前提下（有管道符号的情况已经直接在处理管道符号时加载并执行了相应程序），加载并执行目标程序。

In [ ]:
```
if(!is_pipe)
            {
                if(execvp(argv[0],argv)<0)
                {
                    printf("%s: Command not found\n", argv[0]);
                    exit(0);
                }
            }
        }
```

在父进程当中，把新的作业添加到工作列表中。如果是前台作业，就调用 `waitfg` 函数等待前台作业结束；如果是后台作业，就直接在屏幕上输出相关信息。

这部分给我带来的受收获是对 `fork` 函数的切身体会：乍一看 `if((pid = fork())==0)` 和 `else` 中的代码只有一段会被执行，但实际上 `fork` 函数会返回两次，分别返回父进程pid和子进程pid。所以，这两部分代码都会被执行。在不熟悉 `fork` 函数特性的情况下阅读这部分代码可能会有理解上的困难。

In [ ]:
```
else
        {
            if(state==BG)
            {
                sigprocmask(SIG_BLOCK,&mask_all,NULL);
                addjob(jobs, pid, state, cmdline);
                sigprocmask(SIG_SETMASK,&mask_one,NULL);
                printf("[%d] (%d) %s",pid2jid(pid), pid, cmdline);
            }
            else
            {
                sigprocmask(SIG_BLOCK,&mask_all,NULL);
                addjob(jobs, pid, state, cmdline);
                sigprocmask(SIG_SETMASK,&mask_one,NULL);
                waitfg(pid);
            }
            sigprocmask(SIG_SETMASK, &prev, NULL);
        }
    }
    return;
}
```

`builtin_cmd`

`builtin_cmd`函数的作用是处理shell的内置命令。这里仅介绍这个函数中本次实验作了要求的部分。

首先，如果用户输入`exit`命令，就调用`exit`退出shell进程。

In [ ]:
```
int builtin_cmd(char **argv)
{
    if(!strcmp(argv[0],"exit"))
    {
        exit(0);
    }
```

对单独的`&`不处理。

In [ ]:
```
else if (!strcmp(argv[0],"&"))
    {
        return 1;
    }
```

在shell程序中，我们已经定义了如下全局变量：

In [ ]:
```
char commandHistory[1000][MAXLINE];/* storing command history */
int commandHistoryNumber = 0;/* how many records are stored in commandHistory */
```

分别用来存储之前输入过的命令以及记录的命令的数量。如果输入的命令条数小于0，系统会报错。否则，就按照用户输入的数量，把相应数量的历史记录打印出来。如果记录的数目没有用户要求的那么多，就把记录在案的所有命令都打印出来。如果保存在`commandHistory`中的命令超过了1000条，系统会将目前最早的一条记录删除，并将最新输入的一条命令保存在`commandHistory`的最后。

In [ ]:
```
else if(!strcmp(argv[0],"history"))
    {
        int commandToPrint = atoi(argv[1]);
        if (commandToPrint<=0)
        {
            printf("No command history record will be printed!\n");
            return 1;
        }
        else
        {
            for(int i=0;i<(commandToPrint<commandHistoryNumber?commandToPrint:comma
            {
                printf("%s",commandHistory[i]);
            }
            return 1;
        }
    }
```

如果用户想要移动工作路径，就直接调用`chdir`系统调用来调整工作路径。如果没有这条路径，就报错。

In [ ]:
```
else if (!strcmp(argv[0],"cd"))
    {
        int result = chdir(argv[1]);
        if(result!=0)
        {
            printf("%s : No such directory!\n",argv[1]);
        }
        return 1;
    }
```

如果用户想要查看CPU和内存的使用情况，就输入 `mytop` 命令。随后，调用 `print_memory` 函数打印内存信息。之后，调用 `get_procs` 函数获取当前所有进程。由于需要比较当前状态和上一次输入 `mytop` 命令时的进程状态而第一次输入 `mytop` 命令时 `prev_proc` 指针为空，所以要再调用一次 `get_procs` 函数将当前进程状态记录进 `prev_proc` 。此后，`get_procs` 函数会调用 `parse_dir` 函数读取每个进程的进程号，再调用 `parse_file` 函数将每个进程的信息读取出来，再调用 `cputicks` 函数计算每个进程的滴答，最后调用 `print_procs` 函数打印CPU中运行的进程的信息。

In [ ]:
```
else if(!strcmp(argv[0],"mytop"))
    {
        print_memory();
        getkinfo();
        get_procs();
        if(prev_proc==NULL)
        {
            get_procs();
        }
        print_procs(prev_proc,proc,1);
        return 1;
    }
```

如果不是shell内置命令，就直接返回0。

In [ ]:
```
else
    {
        return 0;
    }
}
```

waitfg

`waitfg` 函数的作用是等待前台作业运行结束。我们用死循环和 `sleep` 函数的组合来实现父进程的休眠。如果前台运行的作业的pid变了，就直接跳出循环。在这个函数的设计过程中，我主要学会了如何使用 `sleep` 函数让进程休眠。

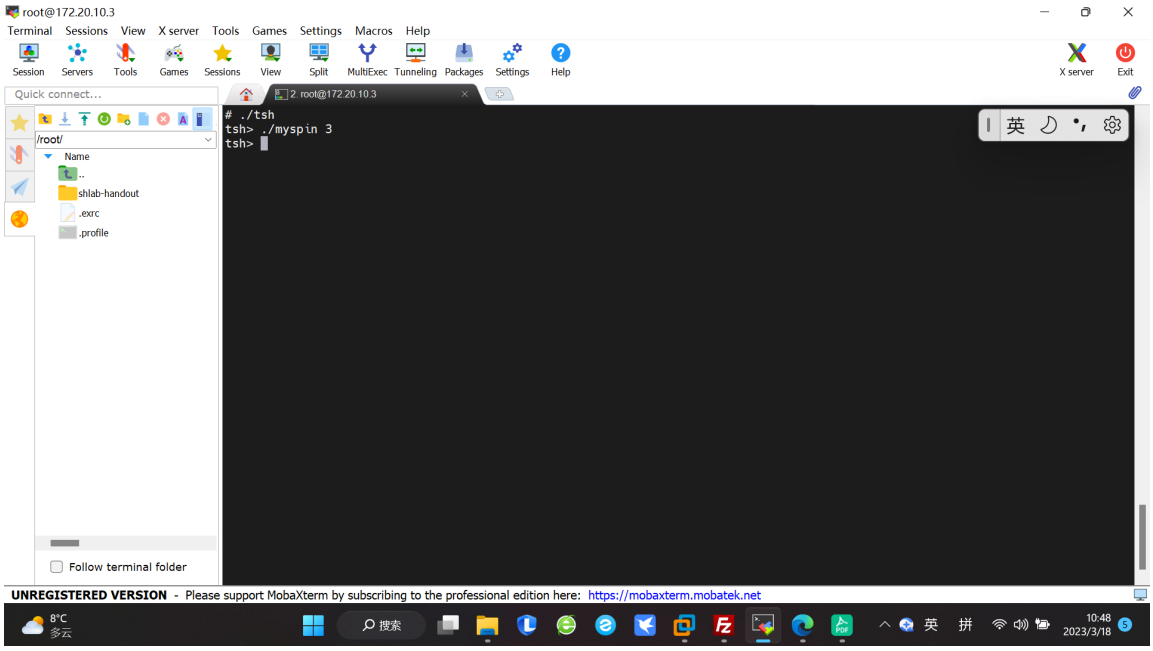
In [ ]:
```
void waitfg(pid_t pid)
{
    sigset_t mask_none,prev;
    sigemptyset(&mask_none);
    while(1)
```

```
    {
        sigprocmask(SIG_SETMASK,&mask_none,&prev);
        if(fgpid(jobs)==0)
        {
            break;
        }
        sleep(1);
        sigprocmask(SIG_SETMASK,&prev,NULL);
    }
    return;
}
```

实验结果

下面，我们将展示这个Unix shell在minix3虚拟机中的运行结果。

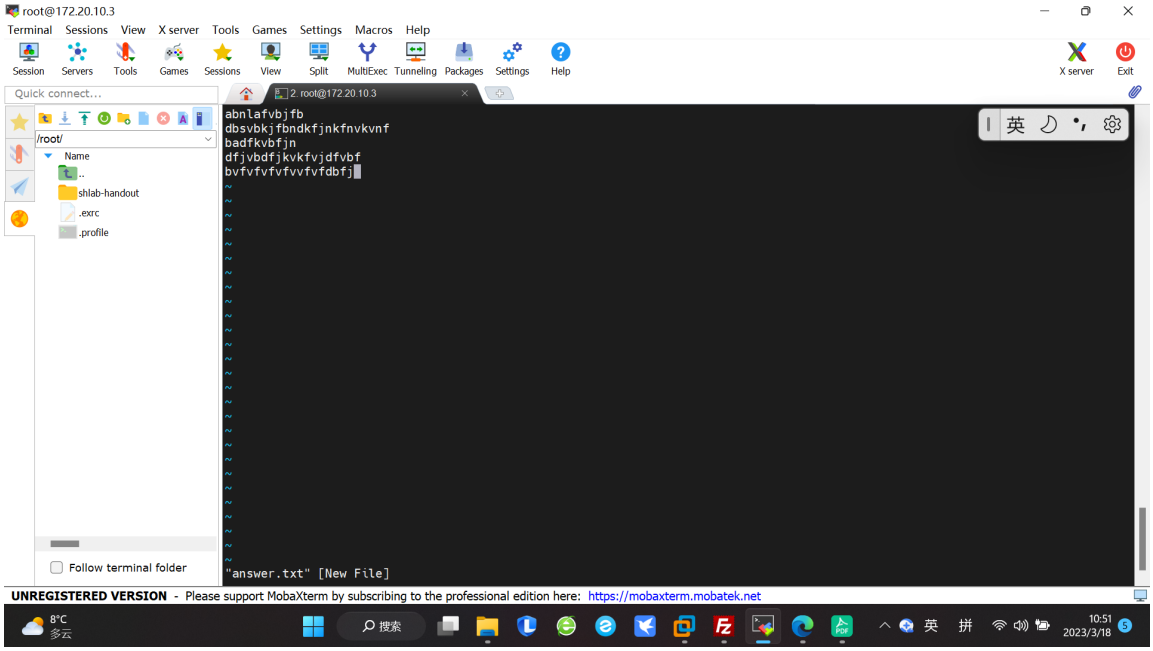
- 带参数的程序运行功能



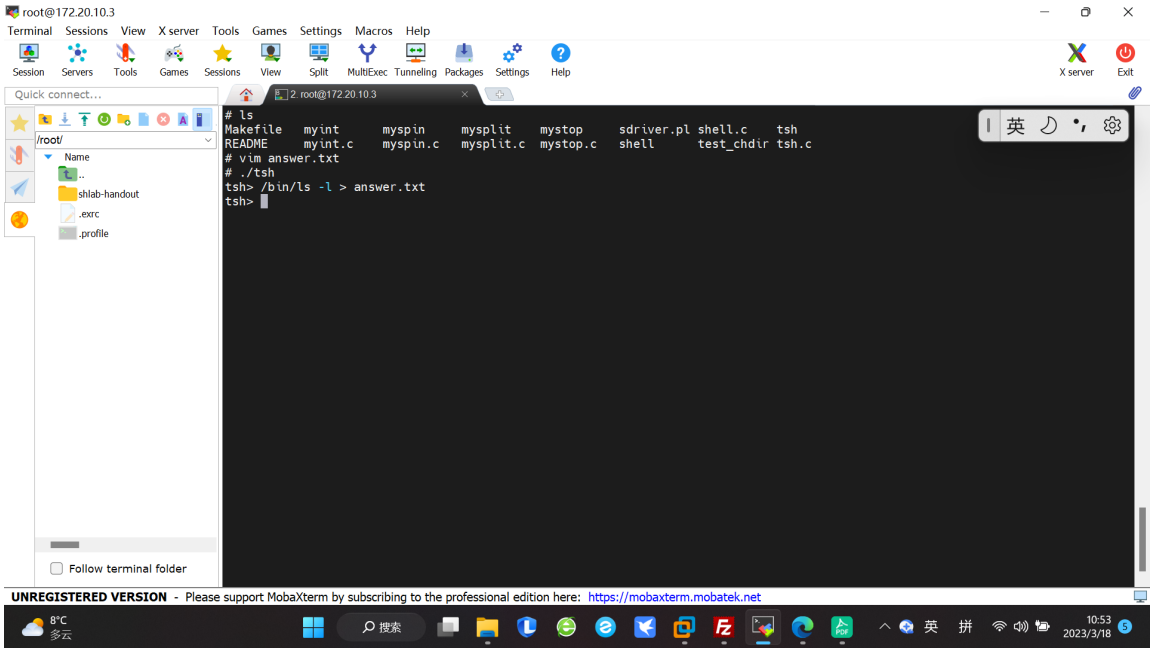
shell程序在前台运行了`myspin`程序，参数为3，没有出现`Command noy found!`之类的报错。

- 重定向功能————覆盖写

我们先打开一个文件`answer.txt`，并在里面输入一些内容。

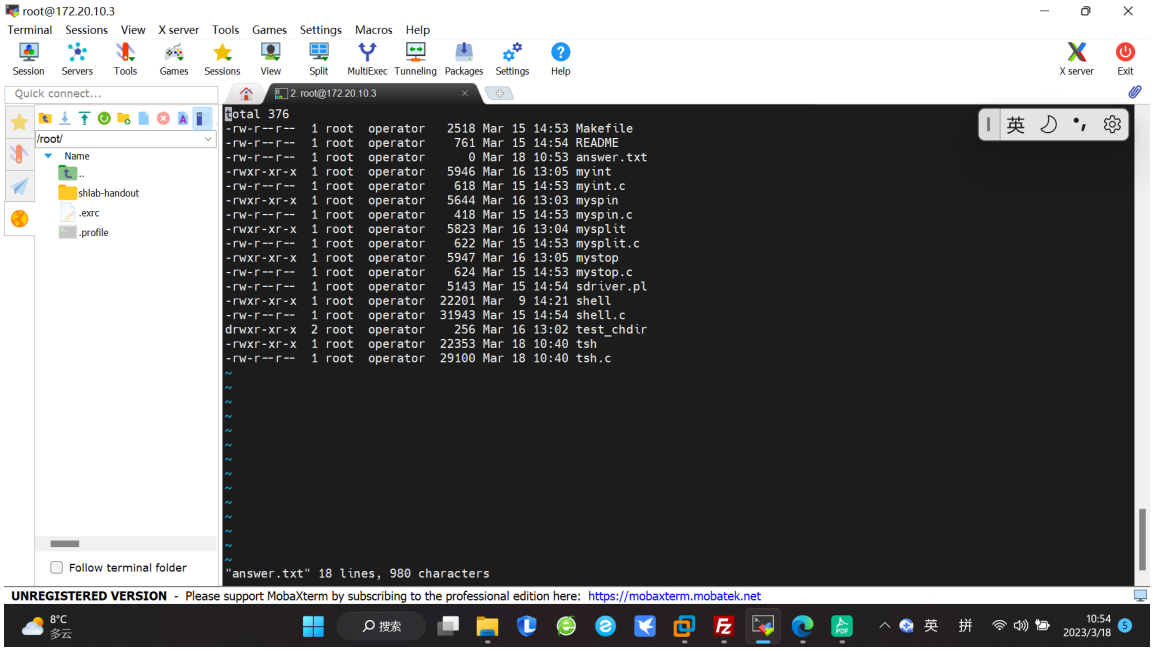


然后，运行shell，执行`/bin/ls -l > answer.txt`命令:



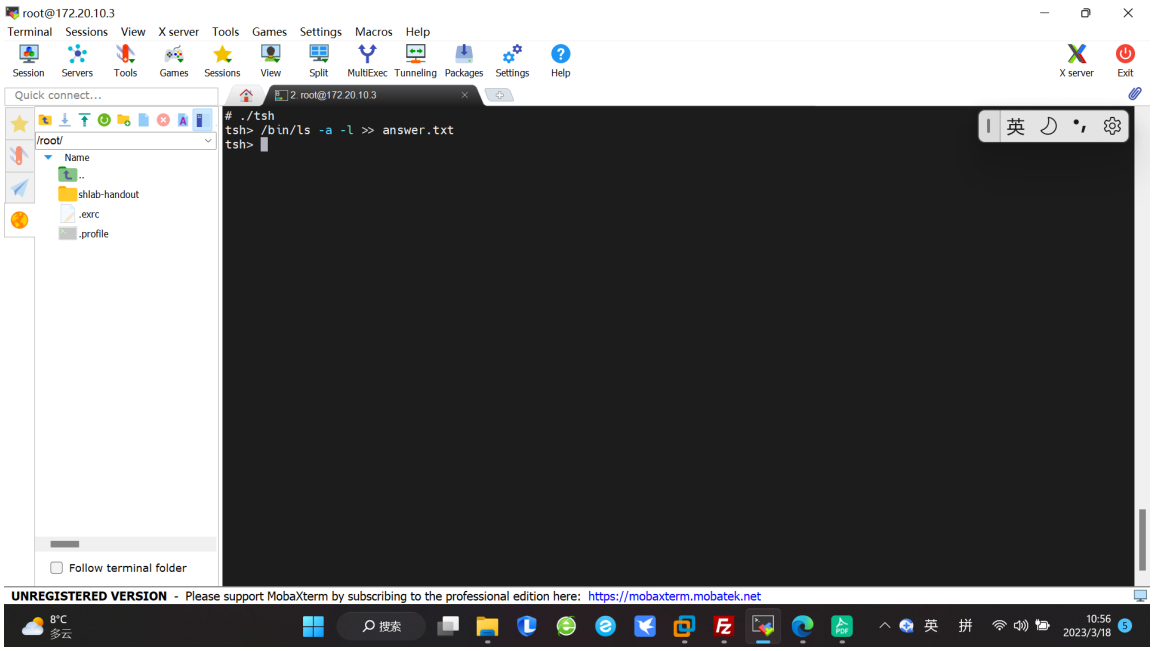
最后，再次打开文件`answer.txt`，先前手动输入的内容已经被`/bin/ls -l`的执行结果覆

盖掉了。

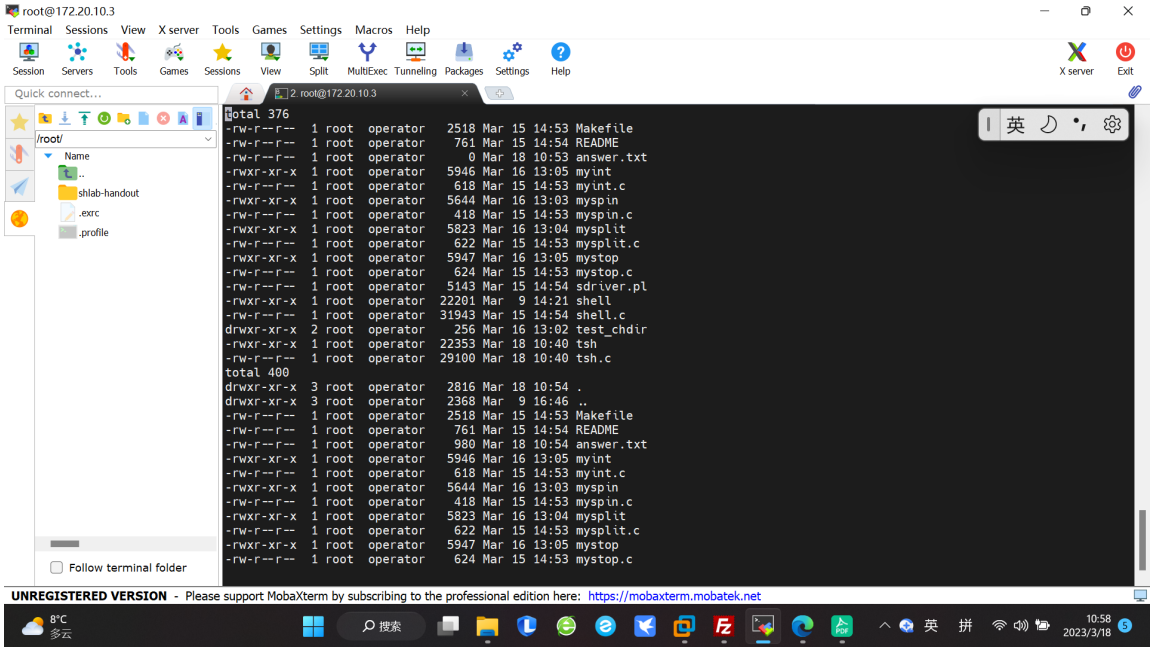


- 重定向功能————追加写

运行shell，执行 `/bin/ls -a -l >> answer.txt` 命令:

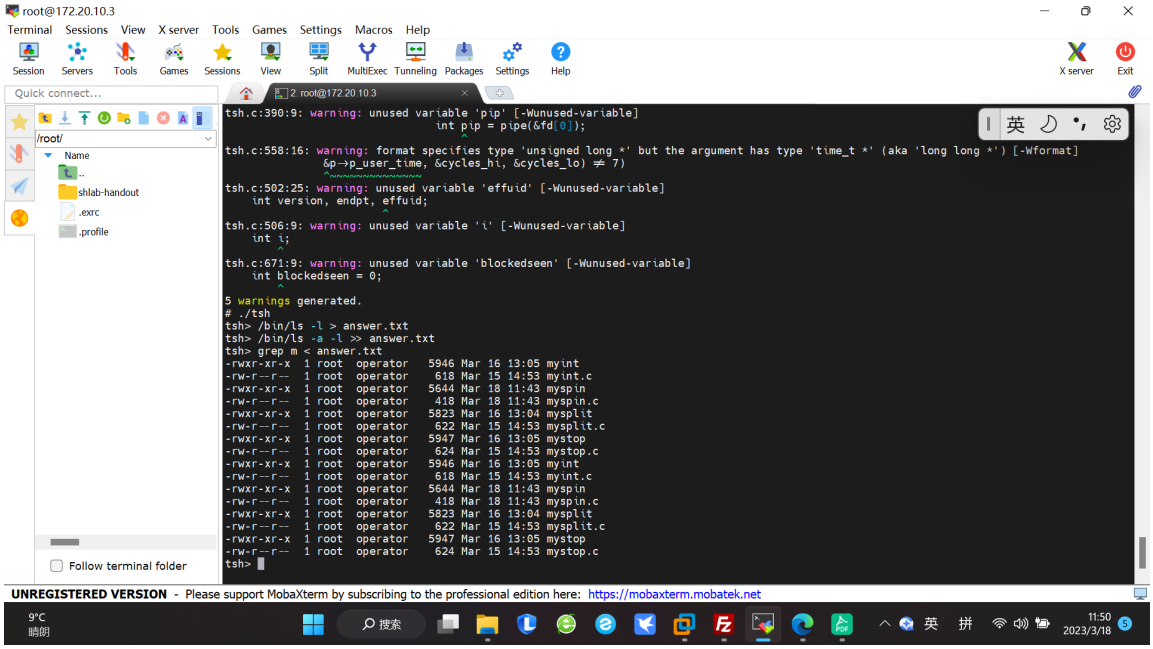


打开文件`answer.txt`，发现后面已经追加了`/bin/ls -a -l`的运行结果。



- 重定向功能————文件输入

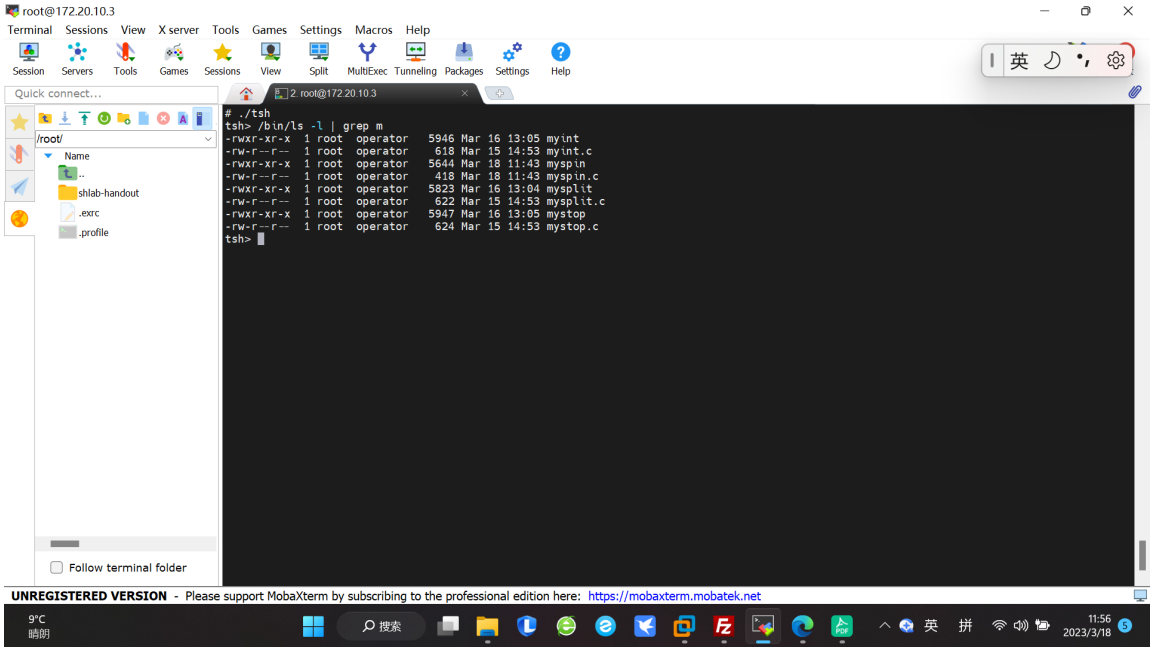
运行shell，执行`grep m < answer.txt`命令:



文件`answer.txt`中，所有带有`m`的行都被打印出来。

- 管道符号

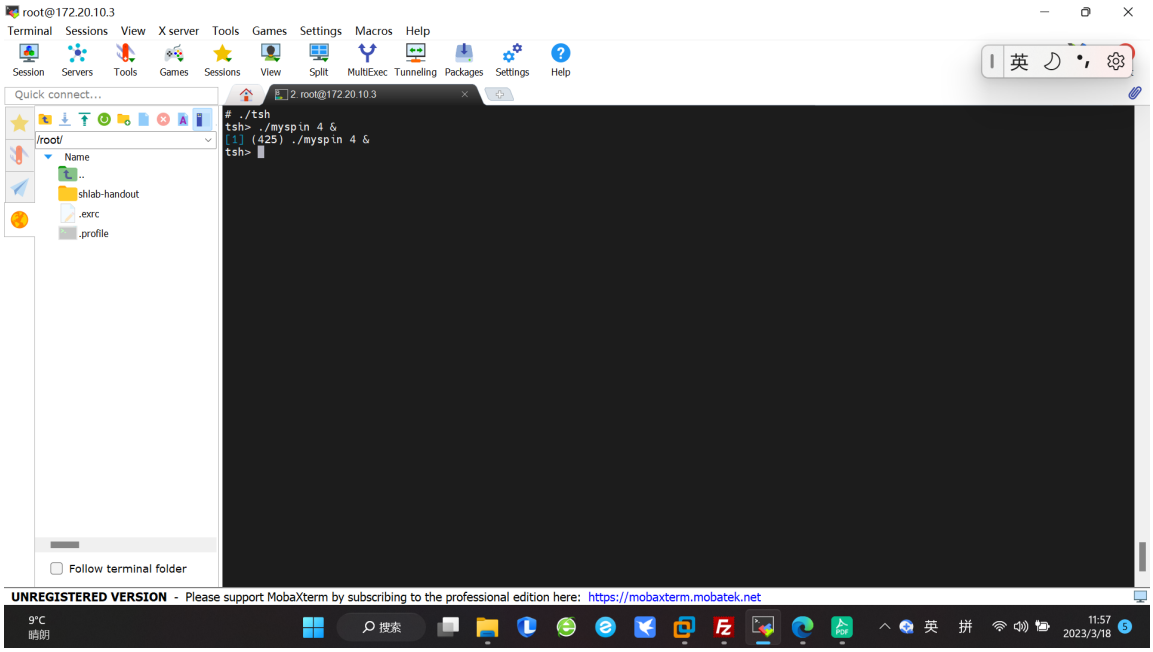
运行shell，执行 `/bin/ls -l | grep m` 命令:



前一个进程的输出被作为后一个进程的输入。

- 后台符号

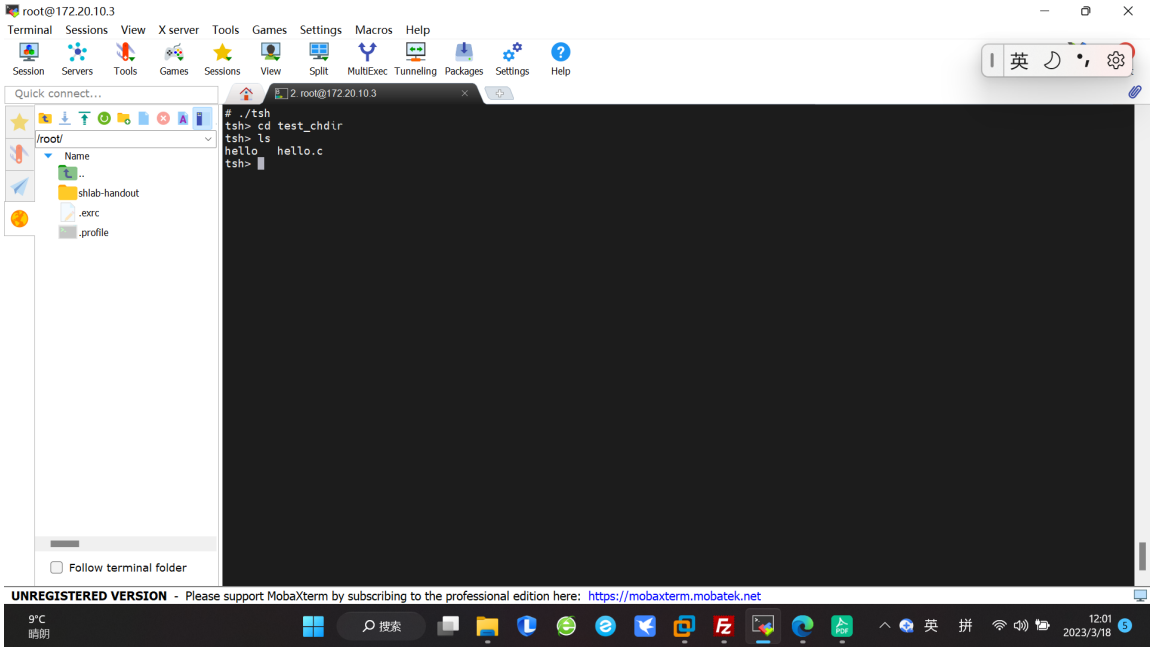
运行shell，执行 `./myspin 4 &` 命令:



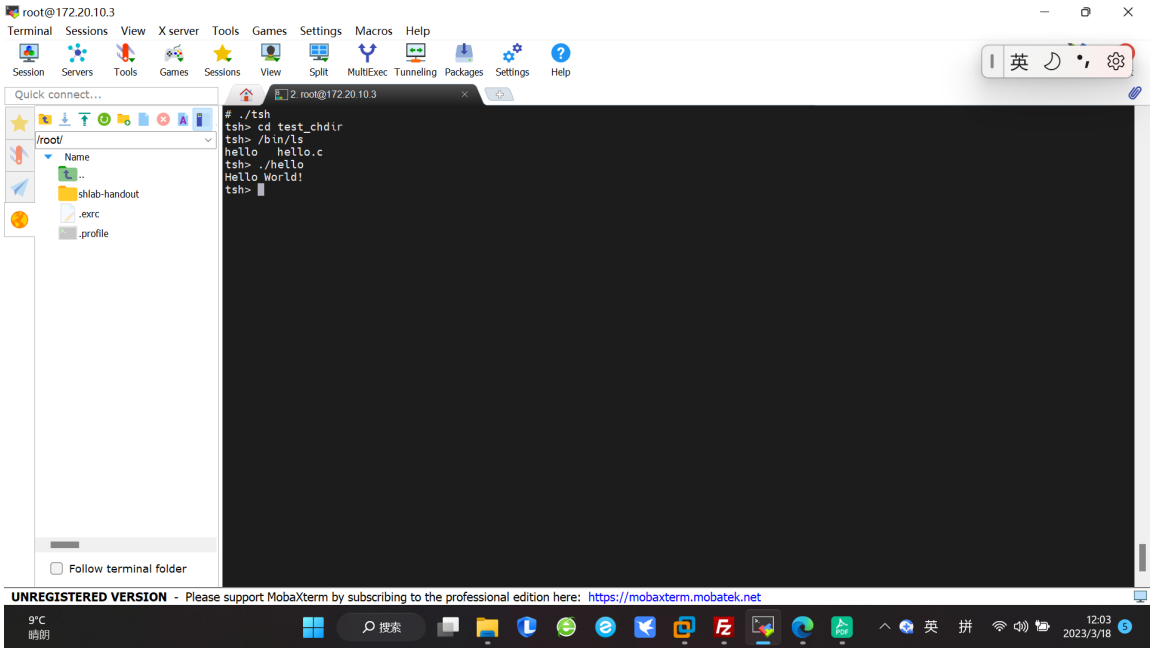
shell直接打印出一条信息，就在后台运行 `myspin` 程序，前台等待接收下一条指令。

- 工作路径移动命令

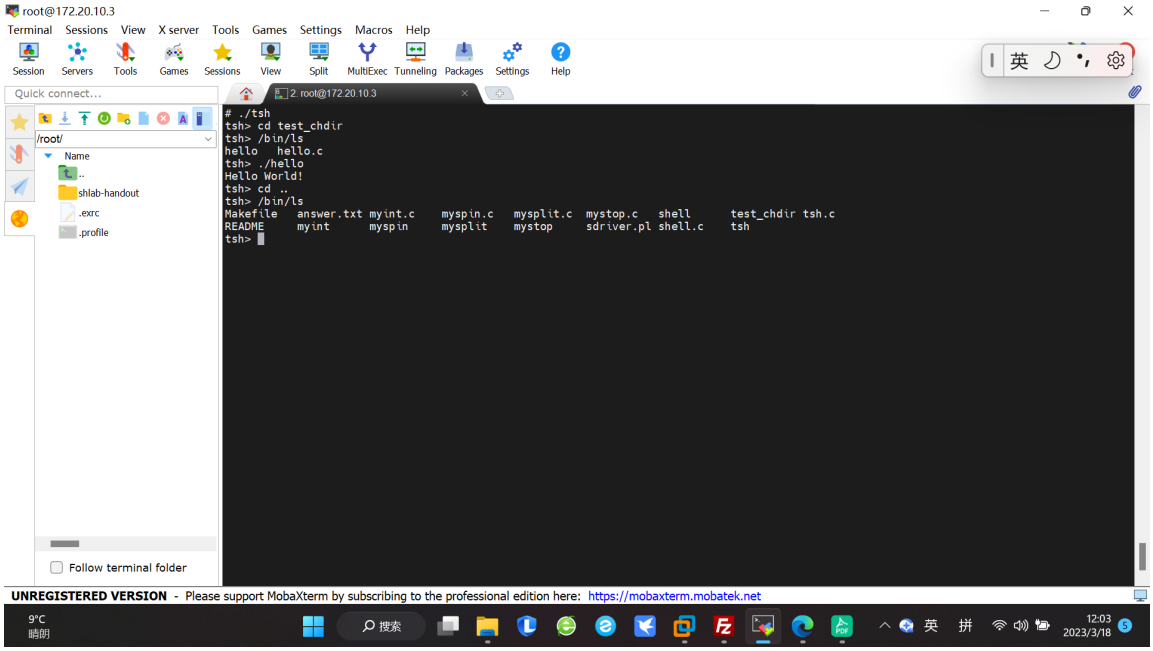
运行shell，进入`test_chdir`文件夹：



里面有一个编译好的`hello`程序。我们执行它：

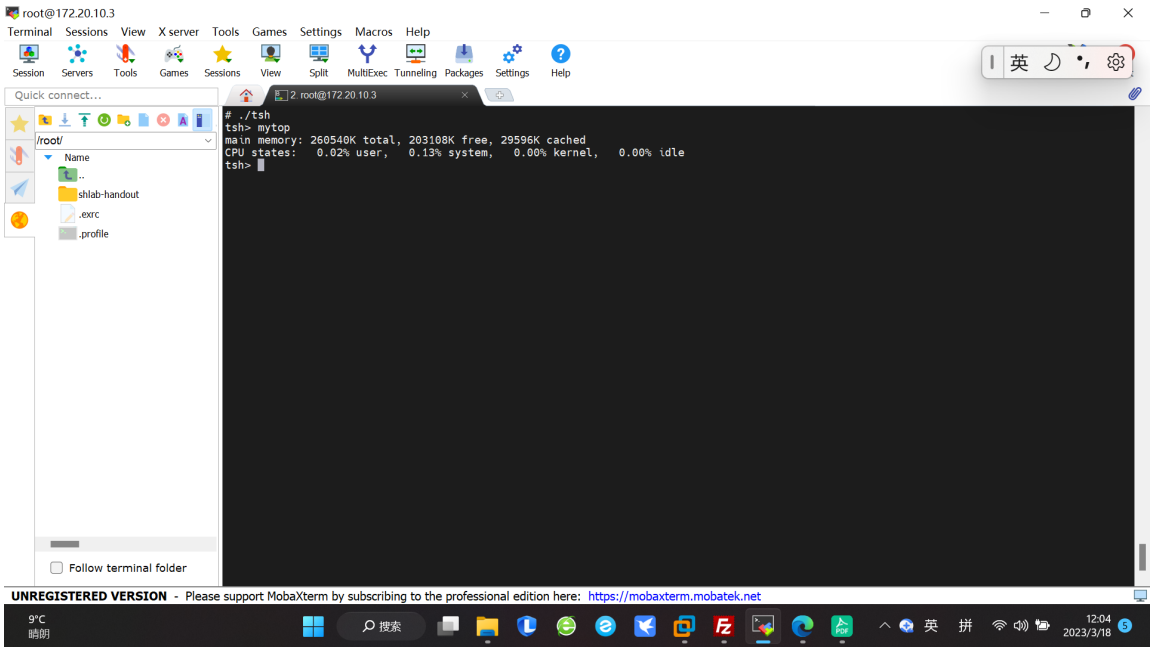


退回到上一级路径：



- 程序运行统计

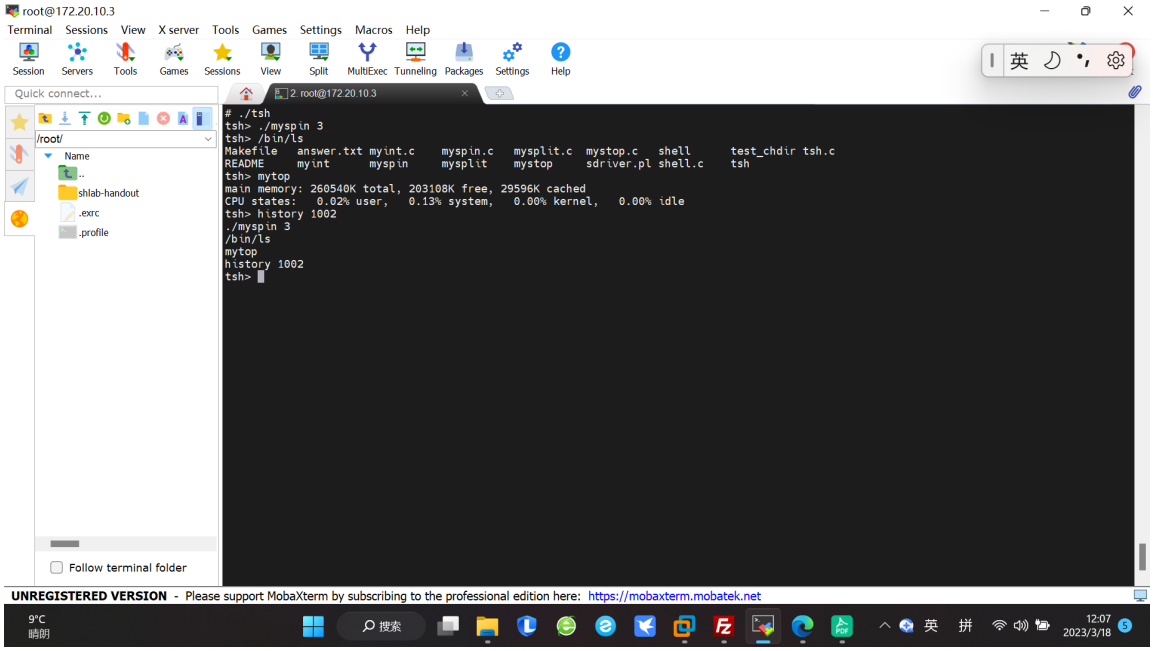
运行shell，输入 `mytop` 命令：



内心和CPU信息被打印在屏幕上。

- 显示最近执行的若干条指令

运行shell，先输入若干条命令，再输入 `history 1002` 命令：



最近执行的相应数量的指令被打印在屏幕上。

- shell退出命令

运行shell，输入 `exit` 命令，shell停止运行：