Shlab report

罗思佳2021201679

一、实验目的

补全`tsh.c`中剩余的代码，编写一个简单的`shell`：

`eval`：解析并执行命令[70行]

`builtin_cmd`：检测命令是否为内建命令：`quit`、`fg`、`bg`、`jobs` [25行]

`do_bgfg`：实现`bg`、`fg`命令 [50行]

`waitfg`：等待前台命令执行完毕 [20行]

`sigchld_handler`：处理`SIGCHLD`信号[80行]

`sigint_handler`：处理`SIGINT`信号，即`ctrl-c` [15行]

`sigstp_handler`：处理`SIGSTP`信号，即`ctrl-z` [15行]

二、前期准备

（1）阅读已有代码

首先分析一下`tsh.c`里原有的代码。

结构体`job_t`代表一个工作，包含了pid、jid、状态和命令行，结构体数组`jobs`用于记录所有`job`的信息。

代码里还给出了许多**辅助函数**：

`parseline`函数用于解析命令行，返回参数列表和bg的值（是否是后台任务）

`sigquit_handler`函数用于接受`SIGQUIT`信号并终止程序

`clearjob`用于清空一个job_t结构体，`initjobs`用于初始化job list

`addjob`函数用于将一个job添加到 job list

`deletejob`函数用于删除指定的job

`fgpid`函数用于得到前台任务的pid

`listjobs`函数用于列出当前的job

（还有一些比较一目了然的函数不必多说）

（2）参考tshref的输出

shlab提供了样例程序，所以可以先试着运行一下看看是什么效果：

输入`quit`，程序直接退出

```
[2021201679@work122 shlab-handout]$ ./tshref
tsh> quit
[2021201679@work122 shlab-handout]$ ./tshref
```

运行一个后台程序，shell会输出这个后台任务的信息；运行一个前台程序，shell会等待这个任务完成。

```
[2021201679@work122 shlab-handout]$ ./tshref
tsh> ./myspin 5 &
[1] (8075) ./myspin 5 &
tsh> jobs
[1] (8075) Running ./myspin 5 &
tsh> ./myint 5
Job [1] (8463) terminated by signal 2
```

实验还提供了16个trace文件用于测试，可以一个一个运行看看每一步会输出什么结果，比如说测试trace09.txt：

```
[2021201679@work122 shlab-handout]$ make rtest09
./sdriver.pl -t trace09.txt -s ./tshref -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (12843) ./myspin 4 &
tsh> ./myspin 5
Job [2] (12845) stopped by signal 20
tsh> jobs
[1] (12843) Running ./myspin 4 &
[2] (12845) Stopped ./myspin 5
tsh> bg %2
[2] (12845) ./myspin 5
tsh> jobs
[1] (12843) Running ./myspin 4 &
[2] (12845) Running ./myspin 5
[2021201679@work122 shlab-handout]$
```

三、实验过程

对shell的运行流程有了一个大致印象之后，然后我开始思考代码的实现。

CMU的实验指导文件里标明了每个函数的预期行数，因此我打算先完成行数较少、容易实现的函数。(以下内容按照完成的顺序进行描述)

1.sigint_handler和 sigstp_handler

我首先选择完成`sigint_handler`和`sigstp_handler`函数，这两个函数都是给前台任务发送信号，只是发送的信号不一样，但在函数里的体现基本一样，都是先需要获取前台任务的pid，然后使用`kill`发送信号。

```
void sigint_handler(int sig)
{
    pid_t pid = fgpid(jobs);//获取前台job的pid
    if(pid != 0)
    {
     kill(-pid, sig);//发送信号
    }

    return;
}
```

```
void sigtstp_handler(int sig)
{
    pid_t pid = fgpid(jobs);
    if(pid != 0)
    {
     kill(-pid, sig);
    }
    return;
}
```

2.builtin_cmd

该函数用于执行内建命令，并且“当用户输入内建命令时，需要立即执行”。因为一共有四个内建命令，所以可以用if-else语句来判断。

如果是`quit`命令，直接用`exit(0)`退出；如果是`bg`或`fg`命令，则执行`do_bgfg`函数进一步处理；如果是`jobs`命令，则执行`listjobs`函数列出所有job。

```
int builtin_cmd(char **argv)
{
    if(strcmp(argv[0], "quit") == 0)
    {
        exit(0);//立即退出
    }
    else if((strcmp(argv[0], "fg")==0) || (strcmp(argv[0],"bg")==0))
    {
        do_bgfg(argv);//进行前后台切换工作
        return 1;
    }
    else if(strcmp(argv[0], "jobs") == 0)
    {
        listjobs(jobs);
        return 1;
    }

    return 0;     /* not a builtin command */
}
```

3.waitfg

“Block until process pid is no longer the foreground process”，也就是需要等待前台工作完成。因此函数逻辑比较简单，需要用一个while循环，调用`fgpid`函数，如果是前台任务的话就等待，等待可以用`sleep(0)`函数，表示挂起0毫秒。

```
void waitfg(pid_t pid)
{
    while(fgpid(jobs) == pid)
    {
        sleep(0);//等待

    }
    return;
}
```

4.do_bgfg

该函数处理后台和前台的命令。由于`<job>`可以由`JID`或者`PID`进行标识，所以需要根据数字前是否有`%`来判断是`JID`还是`PID`。因为要对joblist中的job进行改变，所以需要设置一个结构体指针，指向结构体数组`jobs`中对应的job。如果输入的是`JID`,可以调用辅助函数`getjobjid`获取对应的`job`，如果是`PID`则调用`getjobpid`。

之后，因为不管是前台还是后台任务，都会编程或者保持`Running`状态，所以可以直接`kill(-(job->pid), SIGCONT)`,然后再判断是前台还是后台，前台的话需要调用`waitfg`函数等待，后台则打印该任务的信息，注意需要修改`job`的`state`属性。

然后再判断是前台还是后台，前台的话，先给阻塞态的job发送`SIGCONT`信号，改为Running状态，然后修改`state`为`FG`，前台任务还需要调用`waitfg`函数等待。

如果是后台，同样先给阻塞态的job发送`SIGCONT`信号，改为Running状态，然后修改`state`为`BG`，输出相关信息。

```
void do_bgfg(char **argv)
{
    char* id_str = argv[1];
    struct job_t* job;
    if(id_str[0] == '%')//JID
    {
        //char* jid_str = id_str;
        int jid = atoi(id_str+1);
        job = getjobjid(jobs, jid);
        if(job == NULL)//没找到
        {
            printf("%s: No such job\n", id_str);
            return;
        }
    }
    else
    {
        int pid = atoi(id_str);
        job = getjobpid(jobs, pid);
        if(job == NULL)
        {
```

```
            printf("(%s): No such process\n", id_str);
            return;
        }

    }

    if(strcmp(argv[0], "fg")==0)//前台
    {
        if(job->state == ST)
            kill(-(job->pid), SIGCONT);//改为Running状态

        job->state = FG;
        waitfg(job->pid);//等待前台任务执行完
    }
    else//后台
    {
        if(job->state == ST)
            kill(-(job->pid), SIGCONT);//改为Running状态
        job->state = BG;
        printf("[%d] (%d) %s",job->jid, job->pid, job->cmdline);

    }
    return;
}
```

然而使用trace14.txt进行测试的时候发现，里面有很多错误输入，因此还需要考虑`bg`和`fg`命令输入格式不正确的情况

```
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 4 &
[1] (26326) ./myspin 4 &
tsh> fg
fg command requires PID or %jobid argument
tsh> bg
bg command requires PID or %jobid argument
tsh> fg a
fg: argument must be a PID or %jobid
tsh> bg a
bg: argument must be a PID or %jobid
tsh> fg 9999999
(9999999): No such process
tsh> bg 9999999
(9999999): No such process
tsh> fg %2
%2: No such job
```

首先可能会出现只有bg或fg而没有第二个参数的情况，然后第二个参数还可能是其他字符，需要单独处理。

这里判断是否是`PID`时，我最开时用的是`isdigit`函数，但是考虑到输入的数字可能是两位数或更多，虽然trace文件中并没有这种情况，但还是严谨一点，判断第二个参数的整个字符串是否是数字，使用`strspn(id_str,"0123456789")==strlen(id_str)`进行判断，这里的方法参考了一篇博客：[C语言：判断输入字符串是否为数字*c语言判断字符串是否为数字*那个"谁谁谁"的博客-CSDN博客]

```
void do_bgfg(char **argv)
```

```
{
    if(argv[1] == NULL)//没有第二个参数
    {
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }
    char* id_str = argv[1];
    struct job_t* job;
    if(id_str[0] == '%')//JID
    {
        //char* jid_str = id_str;
        int jid = atoi(id_str+1);
        job = getjobjid(jobs, jid);
        if(job == NULL)//没找到
        {
            printf("%s: No such job\n", id_str);
            return;
        }
    }
    else if(strspn(id_str,"0123456789")==strlen(id_str))//PID
    {
        int pid = atoi(id_str);
        job = getjobpid(jobs, pid);
        if(job == NULL)
        {
            printf("(%s): No such process\n", id_str);
            return;
        }

    }
    else//其他输入
    {
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }

    if(strcmp(argv[0], "fg")==0)//前台
    {
        if(job->state == ST)
            kill(-(job->pid), SIGCONT);//改为Running状态

        job->state = FG;
        waitfg(job->pid);//等待前台任务执行完
    }
    else//后台
    {
        if(job->state == ST)
            kill(-(job->pid), SIGCONT);//改为Running状态
        job->state = BG;
        printf("[%d] (%d) %s",job->jid, job->pid, job->cmdline);

    }
    return;
}
```

5.sigchld_handler

当子任务终止（成为僵尸进程）时，或者因为收到了`SIGSTOP`或`SIGSTP`信号而停滞时，`kernel`会发送`SIGCHLD`信号给`shell`，该函数需要回收所有的僵尸进程，但不要等待其他正在运行的子进程终止。

基本操作是while循环加 pid=waitpid()>0 的判断条件，复习一下`waitpid`函数的定义：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid,int *status,int options);
```

对于参数pid，pid=-1表示等待任何子进程

以下表格参考自(97条消息) Linux中waitpid()函数的用法_Roland_Sun的博客-CSDN博客

对于参数`options`，主要包括两个选项：`WNOHANG`和`WUNTRACED`

参数	说明
WNOHANG	如果pid指定的子进程没有结束，则waitpid()函数立即返回0，而不是阻塞在这个函数上等待；如果结束了，则返回该子进程的进程号。
WUNTRACED	如果子进程进入暂停状态，则马上返回。

然后因为题目要求不要等待其他正在运行的子进程终止，也不用等待被阻滞的进程，所以需要加上这两个选项，用“|”连接。

对于`status`，这个参数将保存子进程的状态信息，有了这个信息父进程就可以知道子进程退出的原因，以及是正常退出还是异常退出。

宏	说明
WIFEXITED(status)	如果子进程正常结束，它就返回真；否则返回假。
WEXITSTATUS(status)	如果WIFEXITED(status)为真，则可以用该宏取得子进程exit()返回的结束代码。
WIFSIGNALED(status)	如果子进程因为一个未捕获的信号而终止，它就返回真；否则返回假。
WTERMSIG(status)	如果WIFSIGNALED(status)为真，则可以用该宏获得导致子进程终止的信号代码。
WIFSTOPPED(status)	如果当前子进程被暂停了，则返回真；否则返回假。
WSTOPSIG(status)	如果WIFSTOPPED(status)为真，则可以使用该宏获得导致子进程暂停的信号代码。

`WIFEXITED(status)`：子进程正常退出，使用`deletejob`函数从job list中删除job。

`WIFSIGNALED(status)`：如果子进程因为一个未捕获的信号而终止，不仅要删除job，还要打印详细信息。

`WIFSTOPPED(status)`：子进程被阻滞，需要修改job的`state`，并打印详细信息。

```
void sigchld_handler(int sig)
{
    pid_t pid;
```

```
    int status;
    while((pid=waitpid(-1,&status, WNOHANG|WUNTRACED))>0)
    {

        if(WIFEXITED(status))//子进程正常结束
        {
            deletejob(jobs, pid);
        }
        else if(WIFSIGNALED(status))//子进程因为一个未捕获的信号而终止
        {
            int jid = pid2jid(pid);
            deletejob(jobs, pid);
            printf("Job [%d] (%d) terminated by signal %d\n", jid, pid,
WTERMSIG(status));

        }
        else if(WIFSTOPPED(status))//子进程被暂停了
        {
            int jid = pid2jid(pid);
            struct job_t* job = getjobjid(jobs,jid);
            job->state = ST;//修改状态
            printf("Job [%d] (%d) stopped by signal %d\n", jid, pid,
WSTOPSIG(status));
        }

    }
    return;
}
```

6.eval

评估用户输入的命令行，可以看做是最后汇总的函数，所以放在最后写。

对于这个函数，CMU的指导文件里有明确的说明：

- In `eval`, the parent must use `sigprocmask` to block `SIGCHLD` signals before it forks the child, and then unblock these signals, again using `sigprocmask` after it adds the child to the job list by calling `addjob`. Since children inherit the `blocked` vectors of their parents, the child must be sure to then unblock `SIGCHLD` signals before it execs the new program.

翻译过来是：在评估中，父进程必须在fork子进程之前使用`sigprocmask`来阻止`SIGCHLD`信号，
然后解除这些信号，在它通过调用`addjob`将子任务添加到任务列表中后，再次使用`sigprocmask`。调用`addjob`后，再次使用`sigprocmask`解除这些信号。由于子程序继承了父程序的阻塞向量，子程序必须确保
解除对`SIGCHLD`信号的封锁，然后再执行新的程序。

所以在函数中需要用到信号阻塞。信号阻塞和解除阻塞的写法参考的是课件：

Explicitly Blocking Signals

```
1       sigset_t mask, prev_mask;
2
3       Sigemptyset(&mask);
4       Sigaddset(&mask, SIGINT);
5
6       /* Block SIGINT and save previous blocked set */
7       Sigprocmask(SIG_BLOCK, &mask, &prev_mask);
8       .
        .  // Code region that will not be interrupted by SIGINT
        .
9       /* Restore previous blocked set, unblocking SIGINT */
10      Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
11
```

首先需要调用辅助函数`parseline`解析命令行获得参数列表，以及返回值。如果第一个参数为空，说明是空行，直接返回。

根据函数前的注释中的提示信息，接着需要判断是否是内建命令，如果是则直接执行`builtin_cmd`，然后返回。如果不是，需要fork出一个子进程来运行job，注意在此之前需要阻塞`SIGCHLD`信号。

然后接下来，如果`pid==0`，说明是子进程，首先解除对`SIGCHLD`信号的阻塞。根据**CMU的指导文件**，在fork之后子进程还需要调用`setpgid(0,0)`函数，改进程组与自己pid一样，确保在前台进程组中只有一个进程；然后调用`execve`函数，在子进程中运行程序，如果函数执行失败，返回值为-1，这时输出“Command not found”的提示信息。

> Here is the workaround: After the `fork`, but before the `execve`, the child process should call `setpgid(0, 0)`, which puts the child in a new process group whose group ID is identical to the child's PID. This ensures that there will be only one process, your shell, in the foreground process group. When you type `ctrl-c`, the shell should catch the resulting SIGINT and then forward it to the appropriate foreground job (or more precisely, the process group that contains the foreground job).

如果是父进程，再判断是`bg`还是`fg`，如果是后台作业，添加job list，并且解除阻塞；如果是前台作业，在加入到job list和解除阻塞后，还需要等待作业完成。

```
void eval(char *cmdline)
{
    char* argv[MAXARGS];
    int bg = parseline(cmdline, argv);//获得参数列表
    pid_t pid;
    sigset_t mask, prev_mask;

    if(argv[0] == NULL)//处理空行
      return;

    int bltin = builtin_cmd(argv);//是内建命令，直接执行
    if(bltin)
        return;
    //不是内建命令
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);//将SIGCHLD加入阻塞列表
    sigprocmask(SIG_BLOCK,&mask,&prev_mask);
```

```
    pid=fork();//创建子进程

    if(pid == 0)//子进程
    {
        sigprocmask(SIG_SETMASK, &prev_mask, NULL);//解除对SIGCHLD的阻塞

        setpgid(0,0);//改变子进程的组ID为子进程本身，确保在前台进程组中只有一个进程

        if(execve(argv[0], argv, environ)<0)//在子进程中运行程序
        {
            printf("%s: Command not found\n", argv[0]);
            exit(0);
        }
    }
    else//父进程
    {
        if(bg)//为后台作业，添加到joblist
        {
            addjob(jobs,pid,BG, cmdline);//加入jobs列表
            sigprocmask(SIG_SETMASK, &prev_mask, NULL);//解除阻塞
            int jid = pid2jid(pid);
            printf("[%d] (%d) %s", jid, pid, cmdline);
        }
        else//前台
        {
            addjob(jobs,pid,FG, cmdline);//加入jobs列表
            sigprocmask(SIG_SETMASK, &prev_mask, NULL);//解除阻塞
            waitfg(pid);//等待前台任务
        }
    }
    return;
}
```

四、实验结果

正确性：80分

```
Part 1: Correctness Tests

Checking trace01.txt...
Checking trace02.txt...
Checking trace03.txt...
Checking trace04.txt...
Checking trace05.txt...
Checking trace06.txt...
Checking trace07.txt...
Checking trace08.txt...
Checking trace09.txt...
Checking trace10.txt...
Checking trace11.txt...
Checking trace12.txt...
Checking trace13.txt...
Checking trace14.txt...
Checking trace15.txt...
Checking trace16.txt...

Preliminary correctness score: 80
```

五、遇到的问题

除了上面描述过的一些问题之外，还有一些细节上的小问题：

1.错误输入

最开始没有注意到要处理错误输入，测试trace14.txt的时候才发现，然后对代码进行补充，并且输出要和样例中的一致。

2.对waitpid函数使用不熟练

之前上课的时候这部分没听太仔细，因此最开始思考了很久要怎么进行判断，之后还是在csdn上找到了对waitpid函数的详细介绍，这才有了思路。

3.指针使用错误

在`eval`函数中，我最开始定义的用来存放参数列表的是二级指针，即`char** argv=NULL`，之后有些样例过不了，

```
Part 1: Correctness Tests

Checking trace01.txt...
rm: cannot remove ‘/tmp/tsh6905/*’: Permission denied
mkdir: cannot create directory ‘/tmp/tsh6905’: File exists
./checktsh.pl: ERROR: Couldn't create /tmp/tsh6905 directory
Checking trace02.txt...
Checking trace03.txt...
```

上网查了一下发现二级指针是一个指向指针的指针，和指针数组不一样，属于典型的“基础不牢，地动山摇”😥

六、实验总结

在本次实验中，我学到了很多关于进程和Shell的知识。我了解了Shell是如何解释和执行命令的，如何处理I/O以及如何管理进程。通过编写Shell程序，我深入理解了操作系统的运作方式。我学会了使用系统调用来完成各种任务。此外，我也掌握了Linux下常见的Shell命令和操作，这今后在Linux系统下的开发工作会有所帮助。

总的来说，本次实验让我深入理解了操作系统和Shell的运作原理，同时也提高了我的编程能力和解决问题的能力。