

# CSAPP : shellab 实验报告

姓名：施楚峰

学号：PB18000335

## 实验介绍

题目要求我们完成一个简单的 Linux Shell，其中大部分函数已经写好，我们只需完成剩余的下部分程序。涉及到第八章异常控制流中信号接收处理相关的内容。

## 实验准备

一定要对书本上的内容充分理解，不然做 lab 的时候就会手足无措，一定要详细了解各个信号相关的函数参数的含义，以及返回值的意义。

## 待完成函数

函数名	功能
void eval(char *cmdline);	处理输入命令行
int builtin_cmd(char **argv);	执行内置命令，如果不是内置命令则返回 0
void do_bgfg(char **argv);	执行内置的 bg fg 命令
void waitfg(pid_t pid);	显式地等待进程结束
void sigchld_handler(int sig);	处理 SIGCHLD 信号
void sigtstp_handler(int sig);	处理 SIGTSTP 信号
void sigint_handler(int sig);	处理 SIGINT 信号

## 函数及参数说明

# waitpid

```
pid_t waitpid(pid_t pid,int * status,int options);
```

参数 options 提供了一些额外的选项来控制 waitpid

- WNOHANG 如果等待集合中的任何子进程都还没有终止，那么就立即返回 0。
- WUNTRACED 挂起调用进程的执行，直到等待集合中的一个进程变成已终止或者被停止。
- WCONTINUED 挂起调用进程的执行，直到等待集合中一个正在进行的进程终止或集合中一个被停止的进程收到 SIGCONT 信号重新开始执行。
- WNOHANG | WUNTRACED 立即返回，如果等待集合中的子进程都没有被停止或终止，则返回 0；如果有一个停止或终止，则返回子进程的 PID。

判断进程退出状态 status 的宏

宏名	功能
WIFEXITED(status)	如果子进程正常终止则为非0值
WEXITSTATUS(status)	取得子进程 exit() 的返回值，只有 WIFEXITED() 返回为真时才定义这个状态
WIFSIGNALED(status)	子进程因信号终止时宏值为真
WTERMSIG(status)	返回导致子进程终止的信号编号，只有 WIFSIGNALED() 返回为真时，才定义这个状态
WIFSTOPPED(status)	子进程处于停止状态时宏值为真
WSTOPSIG(status)	返回导致子进程停止的信号编号，只有 WIFSTOPPED() 返回为真时，才定义这个状态

# kill

```
int kill(pid_t pid,int sig);
```

kill() 可以用来送参数 sig 指定的信号给参数 pid 指定的进程。参数 pid 有几种情况

- pid > 0 将信号传给进程识别码为 pid 的进程。
- pid = 0 将信号传给和目前进程相同进程组的所有进程
- pid = -1 将信号广播传送给系统内所有的进程
- pid < 0 将信号传给进程组识别码为 pid 绝对值的所有进程

执行成功则返回 0 ,如果有错误则返回 -1 。

# 阻塞和解除阻塞信号

函数名	功能
<code>int sigemptyset(sigset_t* set);</code>	初始化信号集（清空）
<code>int sigfillset(sigset_t* set);</code>	初始化信号集（指定位置 1）
<code>int sigaddset(sigset_t* set,int signo);</code>	添加有效信号 signo
<code>int sigdelset(sigset_t* set,int signo);</code>	删除有效信号 signo
<code>int sigismember(const sigset_t* set,int signo);</code>	判断信号集中的有效信号是否包含编号为 signo 的信号
<code>int sigprocmask(int how,const sigset_t* set,sigset_t* oset);</code>	读取或更改进程的阻塞信号集

### 参数说明

how

如何操作信号屏蔽字，此参数有三个可选值：

- SIG\_BLOCK 阻塞该信号
- SIG\_UNBLOCK 解除信号阻塞
- SIG\_SETMASK 设置当前信号屏蔽字为set所指向的值

set/oset

- 如果 oset 是非空指针，则读取进程的当前信号屏蔽字通过oset参数传出
- 如果 set 是非空指针，则更改当前进程的信号屏蔽字

# 实验过程

刚上手程序的时候看了一下内置的 helper routines 发现好像存在一些 bug，它对 `jobid` 的分配存在一些问题。

```
afool@ubuntu: ~/Documents/CS:APP_LAB/shelllab
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[18] (18663) ./myint 10000 &
tsh> ./myint 10000 &
[1] (18664) ./myint 10000 &
tsh> jobs
[17] (18661) Running ./myint 10000 &
[2] (18634) Running ./myint 10000 &
[3] (18637) Running ./myint 10000 &
[4] (18638) Running ./myint 10000 &
[1] (18662) Running ./myint 10000 &
[18] (18663) Running ./myint 10000 &
[7] (18641) Running ./myint 10000 &
[8] (18642) Running ./myint 10000 &
[9] (18643) Running ./myint 10000 &
[10] (18644) Running ./myint 10000 &
[11] (18645) Running ./myint 10000 &
[12] (18646) Running ./myint 10000 &
[1] (18664) Running ./myint 10000 &
[14] (18648) Running ./myint 10000 &
[15] (18649) Running ./myint 10000 &
[16] (18650) Running ./myint 10000 &
tsh>
```

图中两个不同 `job` 的 `jobid` 均为 `1`

## eval

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    char buf[MAXLINE];
    int bg;
    pid_t pid = 0;

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return;

    sigset_t mask_all, mask_one, prev_one;

    sigfillset(&mask_all);
    sigemptyset(&mask_one);
    sigaddset(&mask_one, SIGCHLD);

    if (builtin_cmd(argv) == 0)
    {
        sigprocmask(SIG_BLOCK, &mask_one, &prev_one);
        if ((pid = fork()) == 0)
        {
            setpgid(0, 0);
            sigprocmask(SIG_SETMASK, &prev_one, NULL);

            if (execve(argv[0], argv, environ) == -1) {
```

```

        printf("%s: Command not found\n", argv[0]);
        _exit(0);
    }
}
sigprocmask(SIG_BLOCK, &mask_all, NULL);

if (bg == 1) addjob(jobs, pid, BG, cmdline);
else addjob(jobs, pid, FG, cmdline);

sigprocmask(SIG_SETMASK, &prev_one, NULL);
}
else return;

if (bg == 1) printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
else waitfg(pid);

return;
}

```

书本上有类似的代码，为了避免竞争引起错误，要在 addjob 前阻塞信号，同时由于子进程继承父进程的信号阻塞，所以要在 execve 前解除对信号的阻塞。利用 setpgid 设置子进程所属工作组，以保证在 shell 的前台进程组中只有自己的 tsh 进程。

## builtin\_cmd

```

int builtin_cmd(char **argv)
{
    if (strcmp(argv[0], "quit") == 0)
    {
        _exit(0);
    }
    else if (strcmp(argv[0], "jobs") == 0)
    {
        listjobs(jobs);
    }
    else if (strcmp(argv[0], "bg") == 0)
    {
        do_bgfg(argv);
    }
    else if (strcmp(argv[0], "fg") == 0)
    {
        do_bgfg(argv);
    }
    else return 0;      /* not a builtin command */

    return 1;
}

```

仅仅是对命令的解析与函数调用。

## do\_bgfg

```
void do_bgfg(char **argv)
{
    if (argv[1] == NULL)
    {
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }

    int isjid, id;
    char *buf, *stop;
    isjid = 0;
    buf = argv[1];

    if (*buf == '%')
    {
        isjid = 1;
        ++buf;
    }
    id = strtol(buf, &stop, 10);
    if (buf == stop)
    {
        printf("%s bg: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }

    struct job_t *job;
    if (isjid == 1) job = getjobjid(jobs, id);
        else job = getjobpid(jobs, id);
    if (job == NULL)
    {
        if (isjid == 1) printf("%s: No such job\n", argv[1]);
            else printf("(%d): No such process\n", id);
        return;
    }

    int pid, jid;
    pid = job->pid;
    jid = job->jid;
    if (strcmp(argv[0], "fg") == 0)
    {
        job->state = FG;
        kill(-pid, SIGCONT);
        waitfg(pid);
    }
    else
```

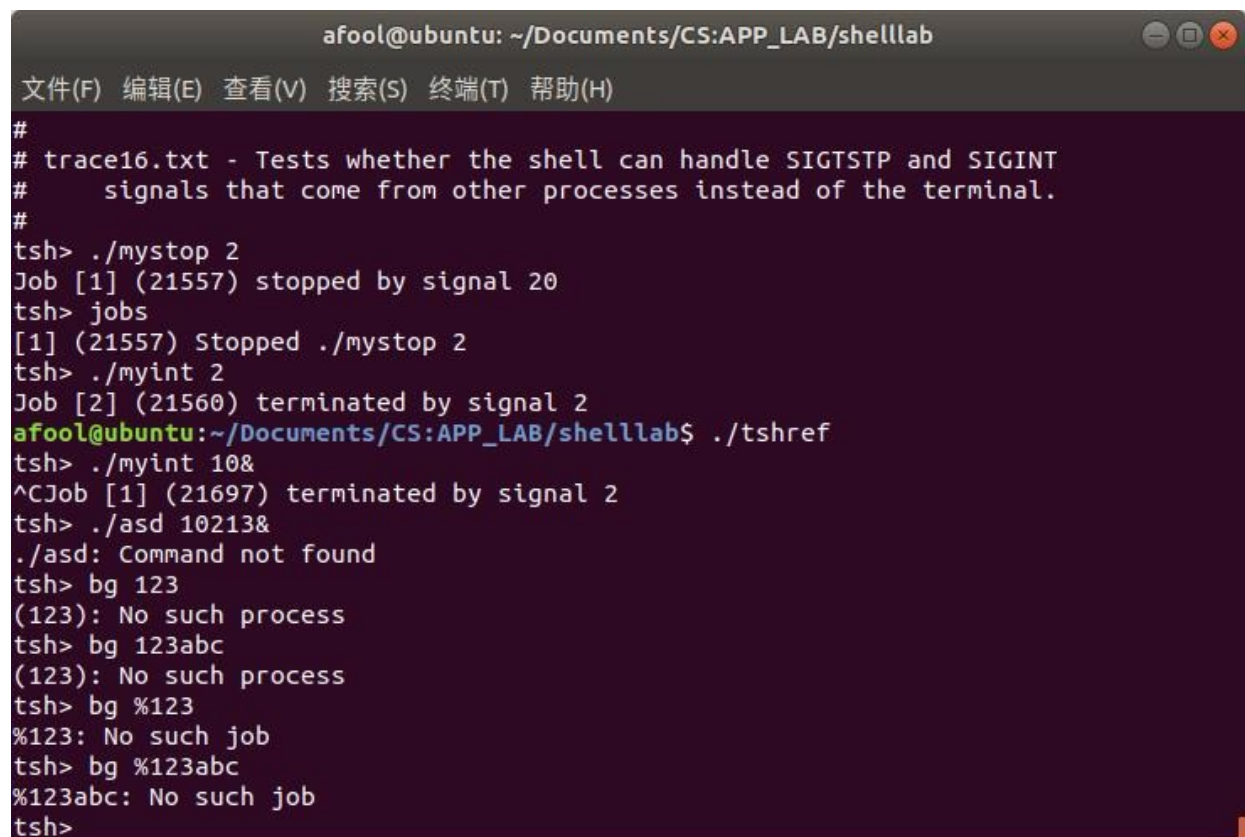
```

{
    job->state = BG;
    kill(-pid, SIGCONT);
    printf("[%d] (%d) %s", jid, pid, job->cmdline);
}

return;
}

```

要确保传入参数的合法性。参考程序输出信息不一致真的很让人头疼。



```

afool@ubuntu: ~/Documents/CS:APP_LAB/shelllab
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
#
# trace16.txt - Tests whether the shell can handle SIGTSTP and SIGINT
#   signals that come from other processes instead of the terminal.
#
tsh> ./mystop 2
Job [1] (21557) stopped by signal 20
tsh> jobs
[1] (21557) Stopped ./mystop 2
tsh> ./myint 2
Job [2] (21560) terminated by signal 2
afool@ubuntu:~/Documents/CS:APP_LAB/shelllab$ ./tshref
tsh> ./myint 10&
^CJob [1] (21697) terminated by signal 2
tsh> ./asd 10213&
./asd: Command not found
tsh> bg 123
(123): No such process
tsh> bg 123abc
(123): No such process
tsh> bg %123
%123: No such job
tsh> bg %123abc
%123abc: No such job
tsh>

```

## waitfg

```

void waitfg(pid_t pid)
{
    while (fgpid(jobs) == pid)
        sleep(1);

    return;
}

```

使用 `busy_loop`，`handler` 中更改全局变量，一旦前台进程结束，就没有任何一个前台进程，于是停止等待。



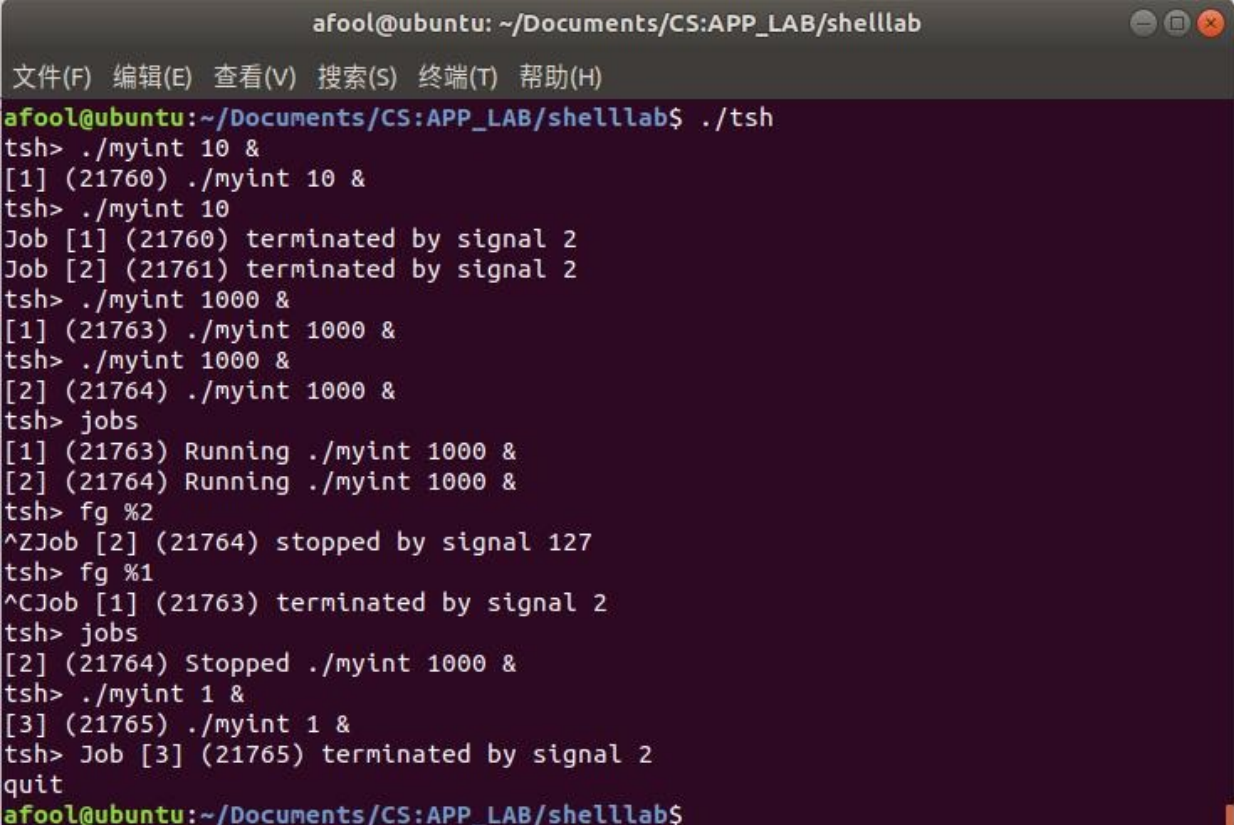
# 实验总结

其实这个实验还应该关注一下对于函数返回值的处理，以及并发错误的处理等方面，正如书上所写涉及到全局变量的函数都是不安全的，这次实验也没能写成一个“真正”安全的，不过如果真考虑起安全性来，感觉每个语句又都是不安全的了。不过通过这个实验，让我对信号的处理有了更直观的了解。如何创建新进程，如何在进程中执行另一个程序，程序的信号如何被传递，接收到信号又如何进一步处理，这些知识点在这一次实验中都有体现。

不过这次实验遇到了一个问题，在使用 Ctrl+Z 暂停前台进程时，却返回 stopped by signal 127，网上给出的说法是

- 1 如果fork失败了，或者waitpid返回除了EINTR之外的错误，system返回 -1；
- 2 exec执行失败，其返回值如同shell执行了“exit(127)”一样。
- 3 如果上述三步都执行成功，那么，system返回值是shell的终止状态。

不过还是搞不清为什么。



```
afool@ubuntu: ~/Documents/CS:APP_LAB/shelllab
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
afool@ubuntu:~/Documents/CS:APP_LAB/shelllab$ ./tsh
tsh> ./myint 10 &
[1] (21760) ./myint 10 &
tsh> ./myint 10
Job [1] (21760) terminated by signal 2
Job [2] (21761) terminated by signal 2
tsh> ./myint 1000 &
[1] (21763) ./myint 1000 &
tsh> ./myint 1000 &
[2] (21764) ./myint 1000 &
tsh> jobs
[1] (21763) Running ./myint 1000 &
[2] (21764) Running ./myint 1000 &
tsh> fg %2
^ZJob [2] (21764) stopped by signal 127
tsh> fg %1
^CJob [1] (21763) terminated by signal 2
tsh> jobs
[2] (21764) Stopped ./myint 1000 &
tsh> ./myint 1 &
[3] (21765) ./myint 1 &
tsh> Job [3] (21765) terminated by signal 2
quit
afool@ubuntu:~/Documents/CS:APP_LAB/shelllab$
```

## 其它