

# Shell lab

## 实验总结

整体来说还是比较轻松的，第八章很大的篇幅都可以用在这个实验上，并且实验给的Hint也足够详细。

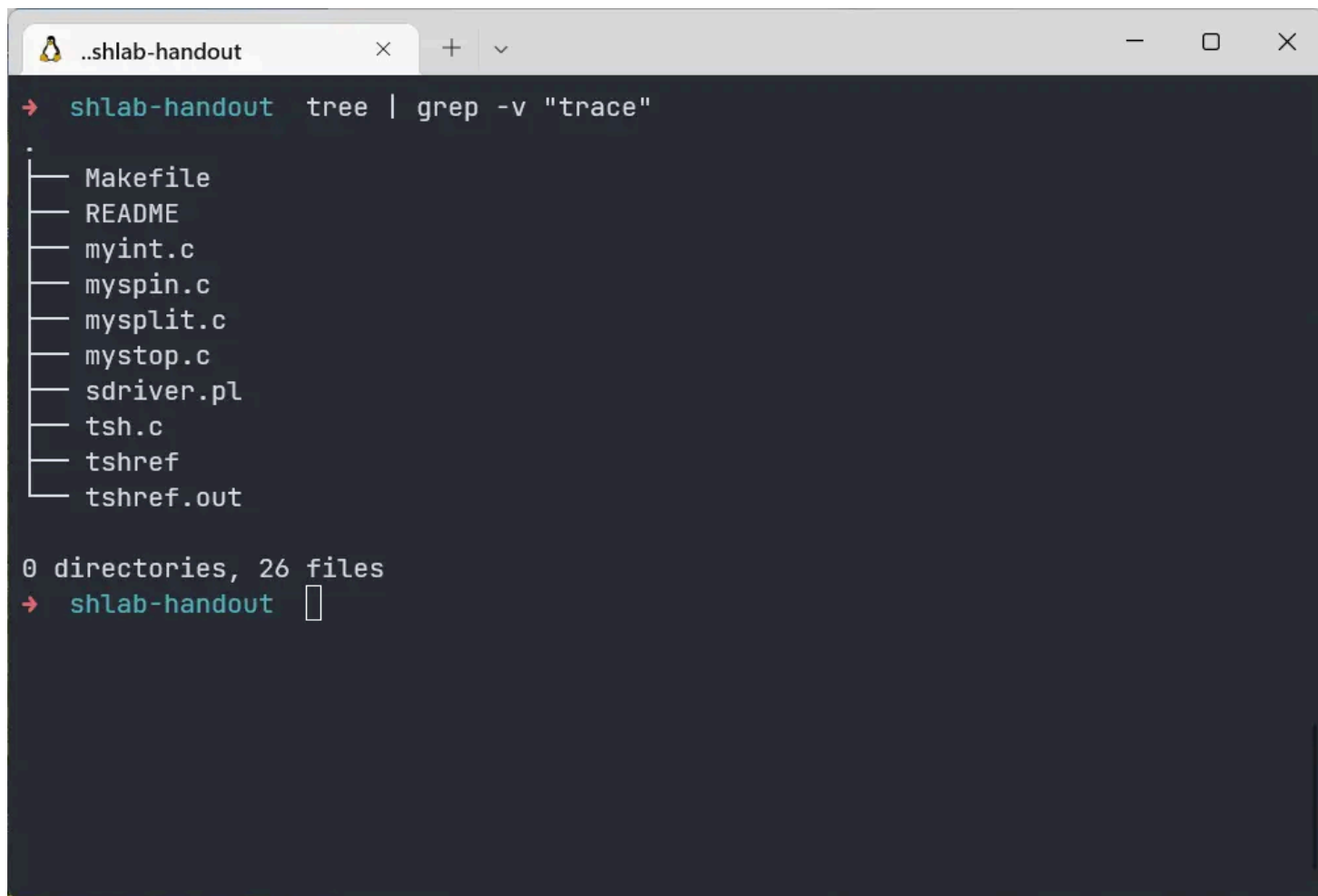
但是感觉这个实验的阶梯性并没有设计的很好，读完第八章之后写出来的简易代码，存在一次性能过很多个trace，或者前面后面trace都能过，但是中间的过不了的现象。

## 前言

在做shelllab之前，请确保已经熟读了第八章，shelllab的很多内容都是课本上原原本本的代码。如果第八章看的不仔细，这个lab很难完成。

**在熟读第八章的基础上，writeup文件中的Hints部分对做实验非常有帮助。**

# 实验介绍



```
..shlab-handout x + v
→ shlab-handout tree | grep -v "trace"
.
├── Makefile
├── README
├── myint.c
├── myspin.c
├── mysplit.c
├── mystop.c
├── sdriver.pl
├── tsh.c
├── tshref
└── tshref.out

0 directories, 26 files
→ shlab-handout
```

实验的目录如图所示:

1. tsh.c是tsh的源文件，本实验的过程就是根据trace文件的提示，在tsh.c中编写一些函数，实现tsh的一些功能。
2. Makefile可以编译我们tsh.c，生成对应的二进制文件tsh，同时也可以用于测试我们的tsh功能，下文会介绍。
3. myxxx.c, [sdriver.pl](#) 是用于测试我们tsh的文件。
4. tshref是老师写好的一个tsh，只要我们的程序的行为跟老师的tsh程序行为一致，就算我们通关了。同时也可……………以用tshref运行trace文件，得到本次trace的正常输出应该是什么。
5. 如果不想运行，可以直接查看tshref.out文件来获得本次trace的正常输出。
6. trace文件比较多，为了显示比较好看，被我过滤掉了。trace中会有提示，指引你下一步应该实现什么功能。

```
..shlab-handout × + v - □ ×
→ shlab-handout make
gcc -Wall -O2 tsh.c -o tsh
gcc -Wall -O2 myspin.c -o myspin
gcc -Wall -O2 mysplit.c -o mysplit
gcc -Wall -O2 mystop.c -o mystop
gcc -Wall -O2 myint.c -o myint
→ shlab-handout make test01
./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
→ shlab-handout make rtest01
./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
→ shlab-handout █
```

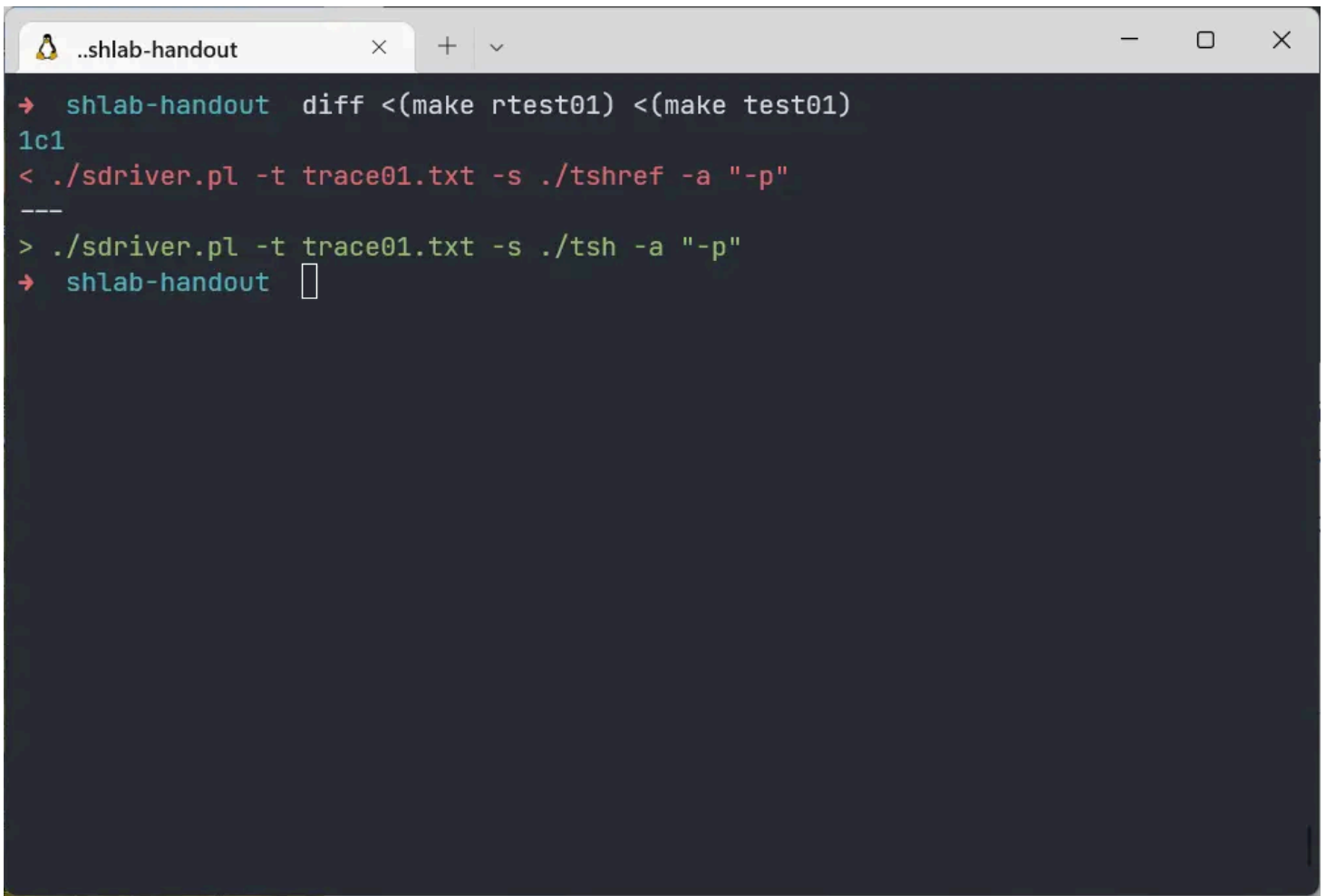
在编译完成(make)tsh.c后，可以通过make test01来进行第一个测试。同时，我们也可以运行make rtest01 来了解功能正常的tsh运行此测试文件应该会产生什么样的输入。本实验共有16个trace文件，只要这16个文件产生的输出与tshref都相同，就算成功。

对shell比较熟悉的同学或许有好点子来与tshref做比较，比如采用如下命令：

```
diff <(make rtest01) <(make test01)
```

将rtest放在前面仅仅是因为在我的shell中，diff的第一个输入如果与第二个不同，会产生一个红色的输出，比较醒目。这样看到单独红色输出就知道自己可能错了。

这其实是个冷知识，叫做进程替换(*process substitution*)：<( Command )会执行Command并将结果输出到一个临时文件中，并将<( Command )替换成临时文件名。

A terminal window titled '..shlab-handout' with standard window controls. It shows a series of commands and their outputs. The first command is 'diff <(make rtest01) <(make test01)', which outputs '1c1'. The second command is './sdriver.pl -t trace01.txt -s ./tshref -a "-p"', followed by a separator '---'. The third command is './sdriver.pl -t trace01.txt -s ./tsh -a "-p"', and the final prompt is 'shlab-handout' with a cursor.

```
→ shlab-handout diff <(make rtest01) <(make test01)
1c1
< ./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
---
> ./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
→ shlab-handout █
```

可以看到我们与标准的tsh产生的输出只有名字不一样而已，证明我们通过了这个测试。

## trace01(Properly terminate on EOF.)

skip这个trace，-.-，发现刚下载完没有编写的tsh.c都可以通过这个测试。

## trace02(trace02.txt - Process builtin quit command.)

这个trace让我们实现内置的quit命令。

首先，我们需要编写eval()函数，此函数的**部分作用**是区分命令是否是内置命令，如果是内置命令，就调用builtin\_cmd()函数，通过此函数来执行内置命令。

这两个函数都在课本的8.4.6节有介绍，我们只需要找到tsh.c中的eval()，与builtin\_cmd()，然后抄上去就可以了。

```

1  /* eval - Evaluate a command line */
2  void eval(char *cmdline)
3  {
4      char *argv[MAXARGS]; /* Argument list execve() */
5      char buf[MAXLINE];    /* Holds modified command line */
6      int bg;               /* Should the job run in bg or fg? */
7      pid_t pid;           /* Process id */
8
9      strcpy(buf, cmdline);
10     bg = parseline(buf, argv);
11     if (argv[0] == NULL)
12         return; /* Ignore empty lines */
13
14     if (!builtin_command(argv)) {
15         if ((pid = Fork()) == 0) { /* Child runs user job */
16             if (execve(argv[0], argv, environ) < 0) {
17                 printf("%s: Command not found.\n", argv[0]);
18                 exit(0);
19             }
20         }
21
22         /* Parent waits for foreground job to terminate */
23         if (!bg) {
24             int status;
25             if (waitpid(pid, &status, 0) < 0)
26                 unix_error("waitfg: waitpid error");
27         }
28         else
29             printf("%d %s", pid, cmdline);
30     }
31     return;
32 }
33
34 /* If first arg is a builtin command, run it and return true */
35 int builtin_command(char **argv)
36 {
37     if (!strcmp(argv[0], "quit")) /* quit command */
38         exit(0);
39     if (!strcmp(argv[0], "&")) /* Ignore singleton & */
40         return 1;
41     return 0; /* Not a builtin command */
42 }

```

Figure 8.24 eval evaluates the shell command line.

抄写完成后，通过运行我们前面介绍的测试命令，来测试这个trace。（编写完成后，不要忘记再次make一下编译tsh.c）

```
..shlab-handout x + v - □ ×
→ shlab-handout make
gcc -Wall -O2 tsh.c -o tsh
→ shlab-handout diff <(make rtest02) <(make test02)
1c1
< ./sdriver.pl -t trace02.txt -s ./tshref -a "-p"
---
> ./sdriver.pl -t trace02.txt -s ./tsh -a "-p"
→ shlab-handout
```

trace02  
成功过关。

## trace03-04

- trace03: run a foreground job.

在抄写完成后，我们可以直接通过trace03，(偷鸡了，原本应该是需要实现job control的，放在trace04，一起介绍)

- trace04: run a background job

```
..shlab-handout x + v - □ ×
→ shlab-handout diff <(make rtest03) <(make test03)
1c1
< ./sdriver.pl -t trace03.txt -s ./tshref -a "-p"
---
> ./sdriver.pl -t trace03.txt -s ./tsh -a "-p"
→ shlab-handout diff <(make rtest04) <(make test04)
1c1
< ./sdriver.pl -t trace04.txt -s ./tshref -a "-p"
---
> ./sdriver.pl -t trace04.txt -s ./tsh -a "-p"
6c6
< [1] (1189) ./myspin 1 &
---
> 1188 ./myspin 1 &
→ shlab-handout
```

trace03-04

与tshref的输出比较发现，我们的输出格式不对，因此，需要修改printf中的输出格式。除此之外，此trace开始需要作业号了，因此我们必须开始完成我们的job control部分了。

## job control

书上的代码是不断迭代成最终版本的，受限于篇幅，我们不展示中间过程版本，直接是最终版本了。**这部分需要看书到非本地跳转之前的所有部分。**

首先，为了写起来方便直观，定义几个宏：

```
#define BLOCK(set, old_set) Sigprocmask(SIG_BLOCK, &(set), &(old_set))
#define BLOCK_NOT_SAVE_OLD_SET(set) Sigprocmask(SIG_BLOCK, &(set), NULL)
#define UNBLOCK(old_set) Sigprocmask(SIG_SETMASK, &(old_set), NULL)
volatile sig_atomic_t FG_PID_GLOBALS;
```

修改后的eval()最终版本

```

void eval(char *cmdline)
{
    char *argv[MAXARGS]; // argument list exec()
    char buf[MAXLINE]; // Holds modified command line.
    int bg; // should the job run in bg or fg?
    pid_t pid;

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    // constructing argv[MAXARGS] And return true if job is bg. iow, the last char is &.

    if (argv[0] == NULL)
        return; // Ignore empty lines.

    if (!builtin_cmd(argv)){

        sigset_t mask_all;
        sigset_t mask_one, prev_one;

        Sigfillset(&mask_all);
        Sigemptyset(&mask_one);
        Sigaddset(&mask_one, SIGCHLD);

        BLOCK(mask_one, prev_one); // Block SIGCHLD.

        pid = Fork();

        if (pid == 0){
            // child runs user job.
            UNBLOCK(prev_one); //Unblock SIGCHLD.
            setpgid(0, 0);
            if (execve(argv[0], argv, environ) < 0){
                printf("%s: Command not found. \n", argv[0]);
                exit(0);
            }
        }
        else {
            int state = bg ? BG : FG;
            BLOCK_NOT_SAVE_OLD_SET(mask_all);
            addjob(jobs, pid, state, cmdline);
            UNBLOCK(prev_one); // Unblock SIGCHLD.
        }
    }
}

```



```

    /* Parent waits for fg job to terminate.*/
    if (!bg){
        waitfg(pid);
    }
    else {
        // the job is bg.
        BLOCK_NOT_SAVE_OLD_SET(mask_all);
        printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
    }
    UNBLOCK(prev_one); // Unblock all signal
}
}
}

```

SIGCHLD信号处理程序。其中，关于waitpid的参数，参考了writeup中的Hints部分。

```

void sigchld_handler(int sig)
{
    int old_errno = errno;
    pid_t pid;
    int status;
    sigset_t mask_all, prev_all;

    sigfillset(&mask_all);

    while((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)
    {
        BLOCK(mask_all, prev_all);
        struct job_t *job = getjobpid(jobs, pid);
        if (pid == fgpid(jobs)){
            FG_PID_G = pid;
        }
        deletejob(jobs, pid);
        UNBLOCK(prev_all);
    }
    errno = old_errno;
}

```

最后，waitfg()函数我们顺手也能写出来了--

```

void waitfg(pid_t pid)
{
    sigset_t mask;
    sigemptyset(&mask);

    FG_PID_GLOBALS = 0;
    while (!FG_PID_GLOBALS)
        sigsuspend(&mask);
}

```

The screenshot shows the Visual Studio Code editor with the file `tsh.c` open. The code defines a `sigchld_handler` function and a `waitfg` function. The terminal window at the bottom shows the following commands and output:

```

shlab git:(master) x make
gcc -Wall -O2 tsh.c -o tsh
shlab git:(master) x diff <(make rtest04) <(make test04)
1c1
< ./sdriver.pl -t trace04.txt -s ./tshref -a "-p"
---
> ./sdriver.pl -t trace04.txt -s ./tsh -a "-p"
6c6
< [1] (4164) ./myspin 1 &
---
> [1] (4163) ./myspin 1 &
shlab git:(master) x

```

trace04

最后，我们成功产生了相同的输出

## trace05(Process jobs builtin command)

trace05与trace02类似，都是处理内置命令。修改**builtin\_cmd()**函数，添加对应命令即可。

```

int builtin_cmd(char **argv)
{

    if (!strcmp(argv[0], "quit"))// quit command.
        exit(0);
    else if (!strcmp(argv[0], "&")) // Ignore singleton &. nothing is happen.
        return 1;
    else if (!strcmp(argv[0], "jobs")){// jobs command.
        sigset_t mask_all, prev_all;
        Sigfillset(&mask_all);

        BLOCK(mask_all, prev_all);
        listjobs(jobs);
        UNBLOCK(prev_all);

        return 1;
    }
    return 0;    /* not a builtin command */
}

```

```

→ shlab git:(master) ✕ diff <(make rtest05) <(make test05)
1c1
< ./sdriver.pl -t trace05.txt -s ./tshref -a "-p"
---
> ./sdriver.pl -t trace05.txt -s ./tsh -a "-p"
6c6
< [1] (4449) ./myspin 2 &
---
> [1] (4447) ./myspin 2 &
8c8
< [2] (4453) ./myspin 3 &
---
> [2] (4451) ./myspin 3 &
10,11c10,11
< [1] (4449) Running ./myspin 2 &
< [2] (4453) Running ./myspin 3 &
---
> [1] (4447) Running ./myspin 2 &
> [2] (4451) Running ./myspin 3 &

```

trace05

轻松通过。

## trace06-08

- trace06: Forward SIGINT to foreground job.
- trace07: Forward SIGINT only to foreground job.
- trace08: Forward SIGTSTP only to foreground job.

这三个trace有个问题需要注意，在我们上面的eval()函数中我们有一句setpid(0,0)，这是课本代码所没有的。

写这个语句的原因是，如果我们在我们的shell中运行tsh，对于我们的shell来说，由于tsh与其子进程都属于同一个进程组，我们如果按下control+C，会把tsh本身与其前台进程全部杀死(回忆关于信号的章节，信号的机制是基于进程组的)。

所以我们需要setpgid(0,0),这个语句会让tsh中的前台进程的进程组id修改成自身的进程id相同。这样对于shell来说，它的前台进程只有tsh一个，然后我们在tsh中小心编写对应的信号处理程序，即可只杀死

前台进程，而不是连同tsh本身都被干掉。

**以上内容也来自writeup中的Hint部分。**

如果不注意区分tsh与其前台进程，trace05-06会发生明明写好了对应的信号处理程序，却看不到输出。

原因是tsh本身也被杀死了or休眠了。

```
4
3     if (pid == 0){
2         // child runs user job.
1         UNBLOCK(prev_one); //Unblock SIGCHLD.
220     .  setpgid(0, 0);           You, 5 hours ago • tsh v1 ...
1         if (execve(argv[0], argv, environ) < 0){
2             printf("%s: Command not found. \n", argv[0]);
3             exit(0);
4         }
5     }
6     else {
7         int state = bg ? BG : FG;
```

setpgid(0, 0)

两个函数逻辑都很简单，比较容易写出。唯一要关注的kill的参数需要设置成-pid，因为按下control+C等，shell需要向整个进程组发送对应的信号。

```
void sigint_handler(int sig)
{
    int old_errno = errno;

    pid_t pid = fgpid(jobs);
    if (pid != 0)
        kill(-pid, sig);

    errno = old_errno;
}

void sigtstp_handler(int sig)
{
    int old_errno = errno;
    pid_t pid = fgpid(jobs);

    if (pid != 0)
        kill(-pid, sig);

    errno = old_errno;
}
```

最后，通过测试发现，我们还需要在tsh打印一些消息，来告知用户操作的结果。根据对应的输出格式，可以在sigchld\_handler()，通过对waitpid的status的一些宏操作(也是书上的内容)与printf来组合实现。

```

void sigchld_handler(int sig)
{
    int old_errno = errno;
    pid_t pid;
    int status;
    sigset_t mask_all, prev_all;

    sigfillset(&mask_all);

    while((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)
    {
        BLOCK(mask_all, prev_all);
        struct job_t *job = getjobpid(jobs, pid);

        if (pid == fgpid(jobs)){
            FG_PID_GLOBALS = pid;
        }

        if (WIFEXITED(status)) // Normal: delete job
        {
            deletejob(jobs, pid);
        }
        else if (WIFSIGNALED(status)) // C^C: print, and delete job
        {
            printf("Job [%d] (%d) terminated by signal %d\n", job->jid, job->pid, WTERMSIG(status));
            deletejob(jobs, pid);
        }
        else if (WIFSTOPPED(status)) // C^Z: print, and modify job state.
        {
            printf("Job [%d] (%d) stopped by signal %d\n", job->jid, job->pid, WSTOPSIG(status));
            job->state = ST;
        }
        else{
            deletejob(jobs, pid);
            printf("hahaha\n");
        }

        UNBLOCK(prev_all);
    }

    errno = old_errno;
}

```

The screenshot shows the Visual Studio Code interface with a terminal window open. The title bar indicates the user is running a shell command in WSL (Windows Subsystem for Linux) on Ubuntu.

The terminal output displays the results of a script execution, showing process IDs, job status, and signal termination messages. The script appears to be managing multiple background processes using `&` and `wait`.

```
tsh.c - shlab [WSL: Ubuntu] - Visual Studio Code  
PROBLEMS 1 OUTPUT DEBUG CONSOLE GITLENS JUPYTER TERMINAL  
zsh +  
6c6  
< [1] (4716) ./myspin 4 &  
---  
> [1] (4717) ./myspin 4 &  
8c8  
< Job [2] (4720) terminated by signal 2  
---  
> Job [2] (4721) terminated by signal 2  
10c10  
< [1] (4716) Running ./myspin 4 &  
---  
> [1] (4717) Running ./myspin 4 &  
→ shlab git:(master) ✕ diff <(make rtest08) <(make test08)  
1c1  
< ./sdriver.pl -t trace08.txt -s ./tshref -a "-p"  
---  
> ./sdriver.pl -t trace08.txt -s ./tsh -a "-p"  
6c6  
< [1] (4810) ./myspin 4 &  
---  
> [1] (4815) ./myspin 4 &  
8c8  
< Job [2] (4812) stopped by signal 20  
---  
> Job [2] (4817) stopped by signal 20  
10,11c10,11  
< [1] (4810) Running ./myspin 4 &  
< [2] (4812) Stopped ./myspin 5  
---  
> [1] (4815) Running ./myspin 4 &  
> [2] (4817) Stopped ./myspin 5  
→ shlab git:(master) ✕
```



```

void do_bgfg(char **argv)
{
    struct job_t *job;
    int id;
    sigset_t mask_all, prev_all;
    Sigfillset(&mask_all);

    if (argv[1] == NULL){
        // single fg or bg.
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }
    else if (sscanf(argv[1], "%%d", &id) > 0){
        BLOCK(mask_all, prev_all);
        job = getjobjid(jobs, id);
        UNBLOCK(prev_all);

        if (job == NULL){
            // not found the job.
            printf("%%d: No such job\n", id);
            return;
        }
    }
    else if(sscanf(argv[1], "%d", &id) > 0){
        BLOCK(mask_all, prev_all);
        job = getjobpid(jobs, id);
        UNBLOCK(prev_all);

        if (job == NULL){
            printf("(%d): No such process\n", id);
            return;
        }
    }
    else{
        // not pid or jid.
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }


    if (!strcmp(argv[0], "bg")){
        kill(-(job->pid), SIGCONT);
        job->state = BG;
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
    }
}

```

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

代码整体比较简单，值得注意的是需要先判断是否是单独的fg，bg命令，之后再通过sscanf做格式解析。因为sscanf第一个参数不可以是NULL，不然会有段错误。

正常做到这个trace的时候应该写不成上述样子，因为trace09-10并没有测试如果fg/bg的参数非法会怎样。这部分正常应该是在trace14才开始做的。



```
File Edit Selection View Go Run Terminal Help
trace10.txt - shlab [WSL: Ubuntu] - Visual Studio Code

PROBLEMS 1 OUTPUT DEBUG CONSOLE GITLENS JUPYTER TERMINAL
zsh + - + - x

< [2] (5159) Stopped ./myspin 5
---
> [1] (5162) Running ./myspin 4 &
> [2] (5164) Stopped ./myspin 5
13c13
< [2] (5159) ./myspin 5
---
> [2] (5164) ./myspin 5
15,16c15,16
< [1] (5157) Running ./myspin 4 &
< [2] (5159) Running ./myspin 5
---
> [1] (5162) Running ./myspin 4 &
> [2] (5164) Running ./myspin 5
→ shlab git:(master) × diff <(make rtest10) <(make test10)
1c1
< ./sdriver.pl -t trace10.txt -s ./tshref -a "-p"
---
> ./sdriver.pl -t trace10.txt -s ./tsh -a "-p"
6c6
< [1] (5284) ./myspin 4 &
---
> [1] (5281) ./myspin 4 &
8c8
< Job [1] (5284) stopped by signal 20
---
> Job [1] (5281) stopped by signal 20
10c10
< [1] (5284) Stopped ./myspin 4 &
---
> [1] (5281) Stopped ./myspin 4 &
→ shlab git:(master) ×
```

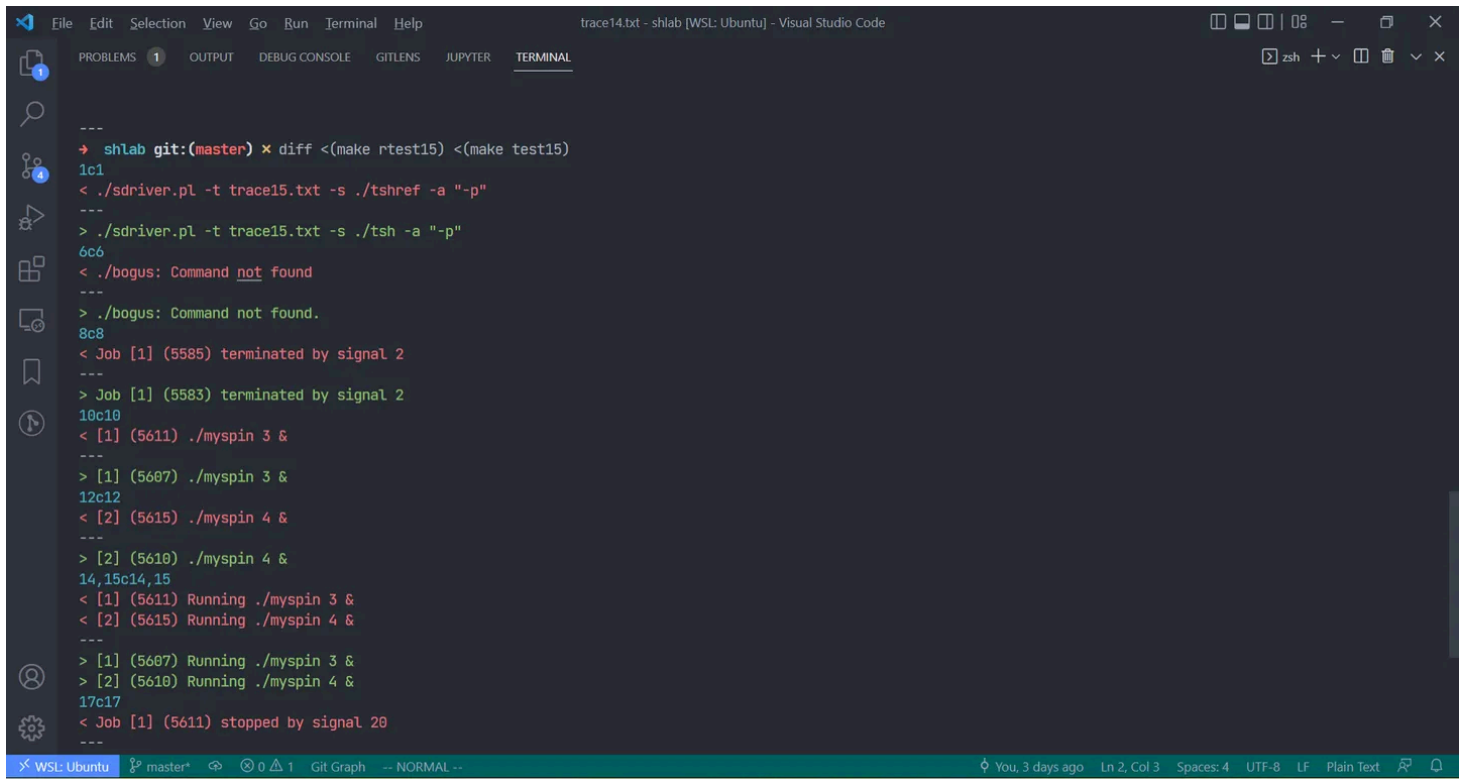
trace10

## trace11-16

- trace11: Forward SIGINT to every process in foreground process group

- trace12: Forward SIGTSTP to every process in foreground process group
- trace13: Restart every stopped process in process group
- trace14: Simple error handling
- trace15: Putting it all together
- trace16: Tests whether the shell can handle SIGTSTP and SIGINT signals that come from other processes instead of the terminal.

由于我们前面每个都快了一点点，导致我们后边6个trace，直接就结束了。  
图太多了，这里只展示下trace15与16的结果。



```
File Edit Selection View Go Run Terminal Help
trace14.txt - shlab [WSL: Ubuntu] - Visual Studio Code

PROBLEMS 1 OUTPUT DEBUG CONSOLE GITLENS JUPYTER TERMINAL

---
shlab git:(master) * diff <(make rtest15) <(make test15)
1c1
< ./sdriver.pl -t trace15.txt -s ./tshref -a "-p"
---
> ./sdriver.pl -t trace15.txt -s ./tsh -a "-p"
6c6
< ./bogus: Command not found
---
> ./bogus: Command not found.
8c8
< Job [1] (5585) terminated by signal 2
---
> Job [1] (5583) terminated by signal 2
10c10
< [1] (5611) ./myspin 3 &
---
> [1] (5607) ./myspin 3 &
12c12
< [2] (5615) ./myspin 4 &
---
> [2] (5610) ./myspin 4 &
14,15c14,15
< [1] (5611) Running ./myspin 3 &
< [2] (5615) Running ./myspin 4 &
---
> [1] (5607) Running ./myspin 3 &
> [2] (5610) Running ./myspin 4 &
17c17
< Job [1] (5611) stopped by signal 20
---
```

trace15

```
> [2] (5610) Running ./myspin 4 &
→ shlab git:(master) × diff <(make rtest16) <(make test16)
1c1
< ./sdriver.pl -t trace16.txt -s ./tshref -a "-p"
---
> ./sdriver.pl -t trace16.txt -s ./tsh -a "-p"
7c7
< Job [1] (5763) stopped by signal 20
---
> Job [1] (5764) stopped by signal 20
9c9
< [1] (5763) Stopped ./mystop 2
---
> [1] (5764) Stopped ./mystop 2
11c11
< Job [2] (5814) terminated by signal 2
---
> Job [2] (5815) terminated by signal 2
→ shlab git:(master) ×
```

trace16

## 后记

原本想写个python脚本，来方便测试的。因为哪怕采用diff <(make rtestxx) <(make testxx)，每次也要修改两个地方，不太方便。但是这个实验很快做完了，也就不了了之。