实验要求

- 第八章：异常控制流 `ECF`
  - 应用是如何与操作系统交互的
  - 异常：硬件和操作系统交界的部分
  - 系统调用：为应用程序提供到操作系统入口点的异常
  - 进程和信号：应用和操作系统的交接之处
  - 非本地跳转：`ECF` 的一种应用层形式

- 实验要求
  - 命令行中的第一个单词是内置命令的名称或可执行文件的路径名， 剩下的词是命令行参数。
  - 如果第一个字是内置命令，则shell立即执行当前进程中的命令；否则，该词被假定为可执行程序的路径名。

    > 在这种情况下，shell 会派生一个子进程，然后在子进程的上下文中加载并运行程序。由于解释单个命令行而创建的子进程统称为作业（在这种情况下，术语job指的是这个初始子进程），通常一个作业可以由多个由 Unix 管道连接的子进程组成

  - 如果命令行以与号“&”结尾，则作业在后台运行，这意味着 shell 在打印提示和等待下一个命令行之前不会等待作业终止。
  - 如果命令行结尾没有符号“&”，作业在前台运行，这意味着 shell 在等待下一个命令行之前等待作业终止。

    > 因此，在任何时间点，最多只能有一个作业在前台运行。 但是，可以在后台运行任意数量的作业

实现一个`shell`

末尾附上代码

Ctrl+d 退出

测试

make test01

make rtest01

tshref.out文件给出了所有测试文件的输出

注意

- 按顺序一个个实现用例文件

- waitpid, kill, fork, execve, setpgid, and sigprocmask； WUNTRACED WNOHANG
- 在eval函数中：
    - 在fork之前，父进程要使用sigprocmask来阻塞SIGCHLD

要实现的函数

void eval(char *cmdline);

int builtin_cmd(char **argv);

- The `quit` command terminates the shell.
- The `jobs` command lists all background jobs.
- The `bg <job>` command restarts `<job>` by sending it a SIGCONT signal, and then runs it in the background. The `<job>` argument can be either a PID or a JID.
- The `fg <job>` command restarts `<job>` by sending it a SIGCONT signal, and then runs it in the foreground. The `<job>` argument can be either a PID or a JID.

void do_bgfg(char **argv);

void waitfg(pid_t pid);

void sigchld_handler(int sig);

void sigtstp_handler(int sig);

void sigint_handler(int sig);

- eval函数包含了shell的主要操作，读取命令行，fork子进程，执行
- builtin_cmd函数包含了处理内置命令行函数的操作，包括quit, fg, bg, jobs
- do_bgfg函数用来处理fg和bg操作，主要是对进程变换状态以及发送SIGCONT信号
- waitfg函数用来等待前台程序结束，因为回收子进程交给了sigchld_handler来做，所以waitfg只要用sleep写一个忙等待来等到前台进程结束。
- 三个信号的操作函数也是要重点实现的内容。

已经提供的

int parseline(const char *cmdline, char **argv); //解析命令行，判断是否是后台进程

void sigquit_handler(int sig);

void clearjob(struct job_t *job);

void initjobs(struct job_t *jobs);

int maxjid(struct job_t *jobs);

int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline);

int deletejob(struct job_t *jobs, pid_t pid);

pid_t fgpid(struct job_t *jobs); //获取前台进程的pid

struct job_t *getjobpid(struct job_t *jobs, pid_t pid);

struct job_t *getjobjid(struct job_t *jobs, int jid);

int pid2jid(pid_t pid);

void listjobs(struct job_t *jobs);

void usage(void);

void unix_error(char *msg);

void app_error(char *msg);

typedef void handler_t(int);

handler_t *Signal(int signum, handler_t *handler);

- `int parseline(const char *cmdline,char **argv)`：获取参数列表`char **argv`，返回是否为后台运行命令（`true`）。
- `void clearjob(struct job_t *job)`：清除`job`结构。
- `void initjobs(struct job_t *jobs)`：初始化`jobs`链表。
- `void maxjid(struct job_t *jobs)`：返回`jobs`链表中最大的`jid`号。
- `int addjob(struct job_t *jobs,pid_t pid,int state,char *cmdline)`：在`jobs`链表中添加`job`
- `int deletejob(struct job_t *jobs,pid_t pid)`：在`jobs`链表中删除`pid`的`job`。
- `pid_t fgpid(struct job_t *jobs)`：返回当前前台运行`job`的`pid`号。
- `struct job_t *getjobpid(struct job_t *jobs,pid_t pid)`：返回`pid`号的`job`。
- `struct job_t *getjobjid(struct job_t *jobs,int jid)`：返回`jid`号的`job`。
- `int pid2jid(pid_t pid)`：将`pid`号转化为`jid`。
- `void listjobs(struct job_t *jobs)`：打印`jobs`。
- `void sigquit_handler(int sig)`：处理`SIGQUIT`信号。

main()函数中，通过while循环调用eval()来**处理**输入的每一行命令

```
int main(int argc, char **argv)
{
    /* Execute the shell's read/eval loop */
    while (1) {
        /* Evaluate the command line */
        eval(cmdline);
    }
    exit(0); /* control never reaches here */
}
```

trace01

Properly terminate on EOF.

正常退出，无需修改

trace02

Process builtin quit command.

实现内置命令quit

> 不需要创建新进程，直接执行即可，也**不会进入**到`eval()`的`if(!builtin_cmd(argv))`当中

```
int builtin_cmd(char **argv)    //如果不是内置命令则返回0
{
    if(!strcmp(argv[0],"quit"))     //strcmp字符串比较函数，如果匹配则返回0
    {
        exit(0);
    }
    return 0;
}
```

trace03

Run a foreground job.

运行一个前台进程/bin/echo

在eval()函数中，`bg=parseline(buf,argv)`判断要执行的命令是否是后台进程（是为1，不是为0），**如果是bg=1则父进程打印子进程id和这条命令本身，如果bg=0则父进程等待子进程结束**

> **执行逻辑**
>
> `bg=parseline(buf,argv)`得到返回值为0，说明该条命令是前台进程
>
> `if(!builtin_cmd(argv))`，该条命令不是内置命令，因此进入`if()`
>
> 创建子进程，并使用`execve()`函数执行`/bin/echo`程序
>
> 父进程执行`addjob()`将其加入job全局列表，要屏蔽所有信号
>
> 由于是前台进程，父进程使用`waitfg(pid)`等待其结束
>
> 当子进程结束后，信号处理程序`sigchld_handler()`执行`deletejob()`，`waitfg()`退出

```
void waitfg(pid_t pid)
{
    //fgpid(jobs)会查找状态为FG的job，前台子进程结束，父进程deletejob()，将前台子进程对应的job进行初始化，fgpid(jobs)找不到前台进程，会返回0，因此会跳出循环
    while(pid==fgpid(jobs)){
        sleep(0);
    }
    return;
}
```

等待前台进程执行结束的函数：调用自带的`fgpid()`函数就能知道当前前台进程号，因为不需要负责子进程回收，只要判断**当前子进程号**是否是**前台进程**，如果不相同，就说明结束了，跳出循环，否则忙等待。

```
void sigchld_handler(int sig)
{
    pid_t pid;
    int status;
    // WNOHANG|WUNTRACED 立即返回：子进程有一个停止或终止，则返回子进程pid，否则返回0
    while((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0){
        if(WIFEXITED(status))    //子进程调用exit或return正常终止(terminate)
        {
```

```
            deletejob(jobs, pid);
        }
        if(WIFSIGNALED(status)) //子进程因一个未被捕获的信号而终止(terminate)
        {
            printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid,
WTERMSIG(status));
            deletejob(jobs, pid);
        }
        if(WIFSTOPPED(status))  //引起返回的子进程当前是停止的(stop)
        {
            printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid,
WSTOPSIG(status));
            struct job_t *job=getjobpid(jobs, pid); //找到pid对应的job
            job->state=ST;  //修改job的状态
        }
    }
    return;
}
```

处理一个子进程结束时发出`SIGCHLD`信号的函数，对于一个子进程结束，主要有3种原因，**正常运行结束**，**收到`SIGINT`终止**，**收到`SIGTSTP`停止**。

对于这三种情况应该要有不同的处理：

- 首先正常结束的肯定要delete。
- 对于终止的进程，，也要delete，并且输出一些信息 。
- 对于停止的，可能会再启动，所以只要修改其state就行了。

trace04

Run a background job.

运行一个后台进程`./myspin 1 &`

> `/bin/echo -e tsh> ./myspin 1 \046` 命令行输入的命令，\046代表**&**的ASCII码表示，防止和**后台进程符号&**冲突

myspin程序会让进程sleep一段时间

```
root@ubuntu:/home/csapp/Desktop/Lab5/shlab-handout# make rtest04
./sdriver.pl -t trace04.txt -s ./tshref -a "-p"
#
# trace04.txt - Run a background job.
#
tsh> ./myspin 1 &
[1] (3446) ./myspin 1 &
```

运行后台进程时，父进程shell要输出一条信息，依次为jid, pid, cmdline

此时，我们用到了定义的结构体job

在**父进程**中，我们要使用`addjob()`将新创建的job加入到jobs列表中

> 如果是内置命令则直接执行，不会创建子进程，也不会区分前台还是后台进程，更不会加入jobs列表

```
// int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline);  state表示是前台还是后台进程UNDEF, BG, FG, or ST
if(bg)      //后台进程BG
{
    addjob(jobs,pid,BG,cmdline);
}
else        //前台进程FG
{
    addjob(jobs,pid,FG,cmdline);
}
```

之后判断该进程是后台进程，执行以下逻辑（输出job的id，进程的id，和命令行内容）

```
if(!bg)         //非后台进程，即前台进程，父进程等待
{
    waitfg(pid);        //等待前台子进程结束
}
else    //后台进程
{
    printf("[%d] (%d) %s",pid2jid(pid), pid, cmdline);
        //printf("%d %s",pid,cmdline);
}
```

trace05

Process jobs builtin command.（处理内置命令jobs）

jobs: List the running and stopped background jobs.

> 如何只显示后台进程，而不显示前台进程?
>
> 前台进程执行完就`deletejob()`，自然不会显示

修改builtin_cmd()函数，添加jobs内置命令

```
if(!strcmp(argv[0],"jobs"))
{
    listjobs(jobs);
    return 1;       //内置命令返回1，非内置命令返回0
}
```

判断子进程为前台进程，调用waitfg()函数

```
void waitfg(pid_t pid)
{
    while(pid==fgpid(jobs)){    //判断子进程pid是否为当前前台进程的id
        sleep(0);
    }
    return;
}
```

sigchld_handler()为子进程结束的信号处理程序，子进程正常退出

```c
void sigchld_handler(int sig)
{
    pid_t pid;
    int status;
    while((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0){
        if(WIFEXITED(status))
        {
            deletejob(jobs, pid);
        }
    }
    return;
}
```

注意!!!

在`fork()`之后，`execve()`之前，子进程执行`setpgid(0, 0)`

会将子进程放入到一个新的进程组，该进程组的`ID`就是此进程的`pid`

仅fork时子进程会继承父进程fork之前所设置的信号处理方式。

当有exec加载新程序时
（1）子进程继承的处理方式是 忽略 或 默认 处理方式时，exec新程序后设置依然有效。
（2）如果子进程继承是捕获处理方式时，exec新程序后将被还原为默认处理方式。（**此情况**）

https://blog.csdn.net/qq_43648751/article/details/104623880

经测试：父进程会接收到`SIGINT`信号，而子进程不会接收

`sleep(1)`和`sleep(0)`

https://blog.csdn.net/u012107806/article/details/44154805

---

trace06

Forward SIGINT to foreground job.

经测试：父进程会接收到`SIGINT`信号，而子进程不会接收

sigint_handler()为`Ctrl+C`的信号处理程序，利用`kill`终止**前台进程**

```
void sigint_handler(int sig)
{
    pid_t pid = fgpid(jobs);    //获取当前前台进程pid
    if(pid!=0)
    {
        kill(-pid,sig);     //发送信号SIGINT给进程组pid中的每个进程
    }
    return;
}
```

sigchld_handler()对`来自键盘的中断Ctrl+C`而**终止**(terminate)的子进程进行处理

```
if(WIFSIGNALED(status))
{
    printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid,
WTERMSIG(status));
    deletejob(jobs, pid);
}
```

> **过程**
>
> 执行`./tsh`程序，产生一个父进程，执行main函数，进入while(1)循环，读取命令行
>
> 执行`./myspin`程序，产生一个子进程，在前台运行
>
> 父进程shell在进入while(1)循环之前就已经完成了绑定`Signal(SIGINT, sigint_handler)`
>
> 父进程shell接收到`Ctrl+C`的信号，执行`sigint_handler()`，通过kill给前台子进程发送`sig`
>
> 子进程收到信号，执行默认处理方式，即终止
>
> 父进程收到子进程终止的信号，执行`sigchld_handler()`，打印信息，并删除对应的job

经测试：父进程会接收到`SIGINT`信号，而子进程不会接收

trace07

Forward SIGINT only to foreground job.

同上：kill只发送信号给前台进程，后台进程继续运行

`kill`应该是负数

trace08

Forward SIGTSTP only to foreground job.

sigtstp_handler()同sigint_handler()

```
void sigtstp_handler(int sig)
{
    pid_t pid = fgpid(jobs);
    if(pid!=0)
    {
        kill(-pid,sig);     //发送信号SIGTSTP给进程组pid中的每个进程
    }
    return;
}
```

sigchld_handler()对`来自终端的停止信号Ctrl+Z`而**停止**(stop)的子进程进行处理

不需要删除job，只需将其状态改为ST

```
if(WIFSTOPPED(status))
{
    printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid,
WSTOPSIG(status));
    struct job_t *job=getjobpid(jobs, pid);
    job->state=ST;
}
```

trace09 & trace10

Process bg builtin command

实现内置命令`bg`和`fg`，将stop变为run，并分别改为后台进程和前台进程

先在`builtin_cmd()`中添加对应的内置命令

```
if(!strcmp(argv[0], "bg")){
    do_bgfg(argv);
    return 1;
}
if(!strcmp(argv[0], "fg")){
    do_bgfg(argv);
    return 1;
}
```

实现`do_bgfg()`

> `SIGCONT`信号：如果该进程停止则继续运行，即将stop变为run

```
void do_bgfg(char **argv)
{
    int id;     //记录jid或pid
    struct job_t *job;  //记录jid或pid对应的job

    //命令行格式如下
    // bg %1 或 bg 1000
    // fg %1 或 fg 1000
```

```
if(argv[1] == NULL) //没有pid或jobID
{
    printf("%s command requires PID or %%jobid argument\n", argv[0]);
    return;
}

if(argv[1][0] == '%')   //参数开头有%，是jobID
{
    // bg %1
    if(argv[1][1] >= '0' && argv[1][1] <= '9')
    {
        //argv[1] 是参数 %1 的地址，也即指针，指针+1得到%后面的字符串，转为整数
        id = atoi(argv[1]+1);   //此时id为job的id，即jid
        job = getjobjid(jobs, id);  //通过jid找到对应的job
        if(job == NULL)
        {   //没有对应的job
            printf("%s: No such job\n",argv[1]);
            return;
        }
    }
    else    //参数错误
    {
        printf("%s: argument must be a PID or %%jobid\n",argv[0]);
        return;
    }
}
else    //参数开头没有%，是pid
{
    // bg 1000
    if(argv[1][0] >='0'&& argv[1][0] <='9'){
        //argv[1] 是参数 1000 的地址，也即指针，直接将其转为整数
        id = atoi(argv[1]);     //此时id为进程的id，即pid
        job = getjobpid(jobs, id);  //通过pid找到对应的job
        if(job == NULL)
        {   //没有对应的jod
            printf("(%s): No such process\n",argv[1]);
            return;
        }
    }
    else
    {
        //参数错误
        printf("%s: argument must be a PID or %%jobid\n",argv[0]);
        return;
    }
}

//此时我们得到了要操作的job
//SIGCONT信号的相应事件：继续进程如果该进程停止，即将stop变为run
kill(-(job->pid), SIGCONT); //将SIGCONT信号通过kill命令传递给job对应的进程组

//接下来判断是在前台运行，还是在后台运行
if(!strcmp(argv[0], "bg"))
{
    job->state = BG;    //修改为后台进程，输出该命令行
    printf("[%d] (%d) %s",job->jid, job->pid, job->cmdline);
}
```

```
    else
    {
        job->state = FG;    //修改为前台进程，等待其运行结束
        waitfg(job->pid);
    }

    return;
}
```

处理 `bg` 和 `fg` 命令的函数，主要是解析参数，判断是否会出现命令错误。找到需要操作的进程，然后用 `kill` 函数对它发出 `SIGCONT` 信号。因为前面创建进程的时候都是用 `setgpid()`，每个进程在单独一个进程组，所以使用 `kill` 非常方便。

进程继续执行之后，只需要给后台进程改一下 `state`，输出对应的命令行；前台进程的话，也要改 `state`，还要 `waitfg()`，等待其运行结束。

`myspin.c`

`sleep` 一段时间后，自己执行 `exit(0)`，正常退出

`mysplit.c`

fork一个子进程，子进程sleep一段时间，父进程等待子进程运行结束后，自己执行 `exit(0)`，正常退出

`myint.c`

`sleep` 一段时间后，通过 `kill(pid, SIGINT)`，给自己发送 `SIGINT` 信号

`mystop.c`

`sleep` 一段时间后，通过 `kill(-pid, SIGTSTP)`，给自己发送 `SIGTSTP` 信号

---

trace11

Forward SIGINT to every process in foreground process group

使前台进程组里的每个进程都能处理 `SIGINT` 信号

> **过程**
>
> 父进程shell创建子进程 `./mysplit`，在 `fork()` 之后和 `execve()` 之前，执行 `setpgid(0, 0)`
>
> 使其单独成为一个进程组，进程组id就是它自己的id

> 接着子进程`./mysplit`有执行一次`fork()`，这两个`./mysplit`进程同属于一个进程组
>
> shell收到`SIGINT`信号后，执行`sigint_handler()`，对前台进程执行`kill(-pid,sig)`
>
> **所以进程组中的两个`./mysplit`进程都会结束**

job和进程组的关系

shell为每个作业创建一个独立的进程组

```
if((pid=fork())==0)
{
 if(setpgid(0, 0) < 0)  //这一步很关键
 {
     printf("setpgid error");
     exit(0);
    }

    if(execve(argv[0],argv,environ)<0)
    {
     printf("%s:Command not found.\n",argv[0]);
     exit(0);
    }
}
```

刚开始，进程组只有这一个进程，作业也只包含一个进程，

如果该进程又创建了新的子进程，则所有进程都属于同一个进程组，也同属于一个作业

3. 从键盘发送信号

Unix shell 使用作业(job)这个抽象概念来表示为对一条命令行求值而创建的进程。在任何时刻，至多只有一个前台作业和 0 个或多个后台作业。比如，键入

```
linux> ls | sort
```

会创建一个由两个进程组成的前台作业，这两个进程是通过 Unix 管道连接起来的：一个进程运行 ls 程序，另一个运行 sort 程序。shell 为每个作业创建一个独立的进程组。进程组 ID 通常取自作业中父进程中的一个。比如，图 8-28 展示了有一个前台作业和两个后台作业的 shell。前台作业中的父进程 PID 为 20，进程组 ID 也为 20。父进程创建两个子进程，每个也都是进程组 20 的成员。

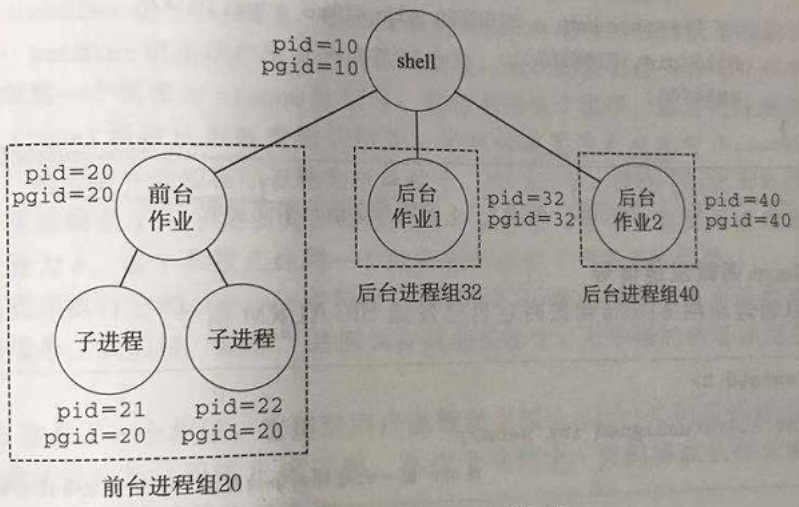

图 8-28　前台和后台进程组

在键盘上输入 Ctrl+C 会导致内核发送一个 SIGINT 信号到前台进程组中的每个进程，默认情况下，结果是终止前台作业。类似地，输入 Ctrl+Z 会发送一个 SIGTSTP 信号到前台进程组中的每个进程。默认情况下，结果是停止(挂起)前台作业。

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

所以`fgpid(jobs)`获取得到的前台进程`pid`，也是**前台进程组（前台作业）**的进程组`id`

使用`kill(-pid,sig)`，就可以使前台进程组里的每个进程都能收到`SIGINT`信号了

```
root@ubuntu:/home/csapp/Desktop/Lab5/shlab-handout# make test11
./sdriver.pl -t trace11.txt -s ./tsh -a "-p"
#
# trace11.txt - Forward SIGINT to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (10685) terminated by signal 2
tsh> /bin/ps a
   PID TTY      STAT   TIME COMMAND
   944 tty7     Ssl+   0:23 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/run/lightdm/root/:
0 -nolisten tcp vt7 -novtswitch
  1362 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
  3294 pts/18   Ss+    0:00 /bin/bash
  3430 pts/4    Ss     0:00 bash
  3452 pts/4    S      0:00 su
  3457 pts/4    S      0:00 bash
 10680 pts/4    S+     0:00 make test11
 10681 pts/4    S+     0:00 /bin/sh -c ./sdriver.pl -t trace11.txt -s ./tsh -a "-p"
 10682 pts/4    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace11.txt -s ./tsh -a -p
 10683 pts/4    S+     0:00 ./tsh -p
 10688 pts/4    R      0:00 /bin/ps a
root@ubuntu:/home/csapp/Desktop/Lab5/shlab-handout# ./tsh
```

可以看到，两个`./mysplit 4`都**消失了**（与`trace12`进行对比）

trace12

trace1

trace12

Forward SIGTSTP to every process in foreground process group

使前台进程组里的每个进程都能处理`SIGTSTP`信号

同上

利用`kill(-pid,sig)`，使前台进程组里的每个进程都能收到`SIGTSTP`信号

```
root@ubuntu:/home/csapp/Desktop/Lab5/shlab-handout# make test12
./sdriver.pl -t trace12.txt -s ./tsh -a "-p"
#
# trace12.txt - Forward SIGTSTP to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (10800) stopped by signal 20
tsh> jobs
[1] (10800) Stopped ./mysplit 4
tsh> /bin/ps a
   PID TTY      STAT   TIME COMMAND
   944 tty7     Ssl+   0:24 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/run/lightdm/root/:
0 -nolisten tcp vt7 -novtswitch
  1362 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
  3294 pts/18   Ss+    0:00 /bin/bash
  3430 pts/4    Ss     0:00 bash
  3452 pts/4    S      0:00 su
  3457 pts/4    S      0:00 bash
 10795 pts/4    S+     0:00 make test12
 10796 pts/4    S+     0:00 /bin/sh -c ./sdriver.pl -t trace12.txt -s ./tsh -a "-p"
 10797 pts/4    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace12.txt -s ./tsh -a -p
 10798 pts/4    S+     0:00 ./tsh -p
 10800 pts/4    T      0:00 ./mysplit 4
 10801 pts/4    T      0:00 ./mysplit 4
 10804 pts/4    R      0:00 /bin/ps a
root@ubuntu:/home/csapp/Desktop/Lab5/shlab-handout#
```

可以看到，两个`./mysplit 4`都是`ST`**状态**（与`trace11`进行对比）

trace13

Restart every stopped process in process group

将进程组中的每个进程，都由stop变为run

同上

在函数`do_bgfg()`中，执行`kill(-(job->pid), SIGCONT)`

将`SIGCONT`信号通过`kill`命令传递给job对应的**进程组**

```
root@ubuntu:/home/csapp/Desktop/Lab5/shlab-handout# make test13
./sdriver.pl -t trace13.txt -s ./tsh -a "-p"
#
# trace13.txt - Restart every stopped process in process group
#
tsh> ./mysplit 4
Job [1] (10866) stopped by signal 20
tsh> jobs
[1] (10866) Stopped ./mysplit 4
tsh> /bin/ps a
   PID TTY      STAT   TIME COMMAND
   944 tty7     Ssl+   0:25 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/run/lightdm/root/:
0 -nolisten tcp vt7 -novtswitch
  1362 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
  3294 pts/18   Ss+    0:00 /bin/bash
  3430 pts/4    Ss     0:00 bash
  3452 pts/4    S      0:00 su
  3457 pts/4    S      0:00 bash
 10861 pts/4    S+     0:00 make test13
 10862 pts/4    S+     0:00 /bin/sh -c ./sdriver.pl -t trace13.txt -s ./tsh -a "-p"
 10863 pts/4    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tsh -a -p
 10864 pts/4    S+     0:00 ./tsh -p
 10866 pts/4    T      0:00 ./mysplit 4
 10867 pts/4    T      0:00 ./mysplit 4
 10870 pts/4    R      0:00 /bin/ps a
tsh> fg %1
tsh> /bin/ps a
   PID TTY      STAT   TIME COMMAND
   944 tty7     Ssl+   0:25 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/run/lightdm/root/:
0 -nolisten tcp vt7 -novtswitch
  1362 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
  3294 pts/18   Ss+    0:00 /bin/bash
  3430 pts/4    Ss     0:00 bash
  3452 pts/4    S      0:00 su
  3457 pts/4    S      0:00 bash
 10861 pts/4    S+     0:00 make test13
 10862 pts/4    S+     0:00 /bin/sh -c ./sdriver.pl -t trace13.txt -s ./tsh -a "-p"
 10863 pts/4    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tsh -a -p
 10864 pts/4    S+     0:00 ./tsh -p
 10873 pts/4    R      0:00 /bin/ps a
root@ubuntu:/home/csapp/Desktop/Lab5/shlab-handout#
```

可以看到，执行`fg %1`之前，两个`./mysplit 4`进程都是`ST`状态

执行`fg %1`之后，两个`./mysplit 4`进程都由`stop`变成了`run`，都成功运行结束

trace14 & trace15

一些测试（包含错误命令）

trace16

Tests whether the shell can handle `SIGTSTP` and `SIGINT` signals that come from other processes instead of the terminal.

子进程自己给自己通过`kill(-pid, SIGTSTP)`和`kill(pid, SIGINT)`发送信号，而不是来自终端的信号

> **过程**
>
> 子进程给自己发送信号后，它的信号处理程序执行默认行为，进行**停止(stop)**或**终止(terminate)**
>
> 父进程收到子进程的信号，执行`sigchld_handler()`信号处理程序

> （此过程并没有调用`sigtstp_handler()`和`sigint_handler()`）

注意

- 处理来自终端的信号，父进程`shell`通过`sigtstp_handler()`和`sigint_handler()`这两个信号处理函数，使用`kill()`函数对相关的**前台进程组**发送对应的信号，接着父进程收到子进程停止或终止的信号后，执行`sigchld_handler()`信号处理函数
- 处理来自进程的信号，前台进程通过`kill()`给自己发送停止或终止的信号，而子进程并没有重写信号处理程序，所以会执行默认行为，也就是停止或终止，接着父进程收到子进程停止或终止的信号后，执行`sigchld_handler()`信号处理函数

代码

`void eval(char *cmdline)`

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

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

    strcpy(buf,cmdline);
    bg=parseline(buf,argv);      //如果是后台进程则bg=1
    if(argv[0]==NULL)
        return;

    if(!builtin_cmd(argv))       //builtin_cmd()判断是否为内置命令，如果不是返回0，即进入该if语句(如果是的话builtin_cmd()内部就执行了)
    {       //进入if语句的都不是内置命令
        sigprocmask(SIG_BLOCK, &mask_one, &prev);   //屏蔽SIGCHLD
        if((pid=fork())==0)
        {
            sigprocmask(SIG_SETMASK, &prev, NULL);  //子进程恢复prev，不再屏蔽SIGCHLD

            if(setpgid(0, 0) < 0){
                printf("setpgid error");
                exit(0);
            }

            if(execve(argv[0],argv,environ)<0)
            {
                printf("%s:Command not found.\n",argv[0]);
                exit(0);
```

```
            }
        }

        sigprocmask(SIG_BLOCK, &mask_all, NULL);    //屏蔽所有信号
        // int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline);
state表示是前台还是后台进程UNDEF, BG, FG, or ST
        if(bg)      //后台进程
            addjob(jobs,pid,BG,cmdline);
        else        //前台进程
            addjob(jobs,pid,FG,cmdline);
        sigprocmask(SIG_SETMASK, &prev, NULL);      //父进程恢复prev

        if(!bg)         //非后台进程，即前台进程，父进程等待
        {
            waitfg(pid);        //等待前台子进程结束
        }
        else        //后台进程
        {
            printf("[%d] (%d) %s",pid2jid(pid), pid, cmdline);
            //printf("%d %s",pid,cmdline);
        }
    }
    return;
}
```

`int builtin_cmd(char **argv)`

```
int builtin_cmd(char **argv)
{
    if(!strcmp(argv[0],"quit"))     //strcmp字符串比较函数，如果匹配则返回0
    {
        exit(0);
    }
    if(!strcmp(argv[0],"jobs"))
    {
        listjobs(jobs);
        return 1;       //内置命令返回1，非内置命令返回0
    }
    if(!strcmp(argv[0], "bg")){
        do_bgfg(argv);
        return 1;
    }
    if(!strcmp(argv[0], "fg")){
        do_bgfg(argv);
        return 1;
    }
    return 0;     /* not a builtin command */
}
```

`void do_bgfg(char **argv)`

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

```
    int id;     //记录jid或pid
    struct job_t *job;  //记录jid或pid对应的job

    //命令行格式如下
    // bg %1 或 bg 1000
    // fg %1 或 fg 1000

    if(argv[1] == NULL) //没有pid或jobID
    {
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }

    if(argv[1][0] == '%')   //参数开头有%，是jobID
    {
        // bg %1
        if(argv[1][1] >= '0' && argv[1][1] <= '9')
        {
            //argv[1] 是参数 %1 的地址，也即指针，指针+1得到%后面的字符串，转为整数
            id = atoi(argv[1]+1);   //此时id为job的id，即jid
            job = getjobjid(jobs, id);  //通过jid找到对应的job
            if(job == NULL)
            {   //没有对应的job
                printf("%s: No such job\n",argv[1]);
                return;
            }
        }
        else    //参数错误
        {
            printf("%s: argument must be a PID or %%jobid\n",argv[0]);
            return;
        }
    }
    else    //参数开头没有%，是pid
    {
        // bg 1000
        if(argv[1][0] >='0'&& argv[1][0] <='9'){
            //argv[1] 是参数 1000 的地址，也即指针，直接将其转为整数
            id = atoi(argv[1]);     //此时id为进程的id，即pid
            job = getjobpid(jobs, id);  //通过pid找到对应的job
            if(job == NULL)
            {   //没有对应的jod
                printf("(%s): No such process\n",argv[1]);
                return;
            }
        }
        else
        {
            //参数错误
            printf("%s: argument must be a PID or %%jobid\n",argv[0]);
            return;
        }
    }

    //此时我们得到了要操作的job
    //SIGCONT信号的相应事件：继续进程如果该进程停止，即将stop变为run
    kill(-(job->pid), SIGCONT); //将SIGCONT信号通过kill命令传递给job对应的进程组
```

```
    //接下来判断是在前台运行，还是在后台运行
    if(!strcmp(argv[0], "bg"))
    {
        job->state = BG;    //修改为后台进程，输出该命令行
        printf("[%d] (%d) %s",job->jid, job->pid, job->cmdline);
    }
    else
    {
        job->state = FG;    //修改为前台进程，等待其运行结束
        waitfg(job->pid);
    }

    return;
}
```

`void waitfg(pid_t pid)`

```
void waitfg(pid_t pid)
{
    while(pid==fgpid(jobs)){    //判断子进程pid是否为当前前台进程的id
        sleep(0);
    }
    return;
}
```

`void sigchld_handler(int sig)`

```
void sigchld_handler(int sig)
{
    pid_t pid;
    int status;
    while((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0){
        if(WIFEXITED(status))
        {
            deletejob(jobs, pid);
        }
        if(WIFSIGNALED(status))
        {
            printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid,
WTERMSIG(status));
            deletejob(jobs, pid);
        }
        if(WIFSTOPPED(status))
        {
            printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid,
WSTOPSIG(status));
            struct job_t *job=getjobpid(jobs, pid);
            job->state=ST;      //将run改为stop
        }
    }
    return;
}
```

`void sigtstp_handler(int sig)`

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

`void sigint_handler(int sig)`

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