Shell Lab

前言

shell lab实验是对于CSAPP第八章的内容的代码补充,需要在开始之前阅读掌握异常控制流(ECF)这一部分内容.中文版第三版524页开始

实验内容

本次实验推荐大家在本机WSL中完成

开始之前

在正式开始实验之前我们先从一个宏观的角度来审视一下这次实验

```
(base) kamilu@LZX:~/shlab$ ls
Makefile   mysplit.c    trace02.txt  trace06.txt  trace10.txt  trace14.txt  tshref
README     mystop.c     trace03.txt  trace07.txt  trace11.txt  trace15.txt  tshref.out
myint.c    sdriver.pl   trace04.txt  trace08.txt  trace12.txt  trace16.txt
myspin.c   trace01.txt  trace05.txt  trace09.txt  trace13.txt  tsh.c
```

其中出现了几个my开头的c文件,`mysplit.c` `mystop.c` `myspin.c` `myint.c` 这些文件的代码相当简单,正如其文件名所示.这些函数并没有参与到本次shell的编译之中,你可以在Makefile中看到它们会被单独编译得到对应的可执行文件,这些文件会被用于测试

接着你可以看到trace开头的16个文件,这些文件是本次实验使用的测试文件,用于验证你是否正确的实现了shell.从最简单的trace01.txt到复杂的trace16.txt,在逐步完成每一个阶段的任务之后,你会得到一个更加强大,更加全面的shell.

然后你可以发现一个可执行文件 `tshref`,这是一个shell的参考文件,如果你不确定shell的运行结果或者输出结果,你可以通过这个已经编写好的shell来辅助你判断正确的结果

最后是`tsh.c`,这是你本次实验需要的主文件,**你只需要编写修改这个文件中的代码来完成一个shell**

接下来编译整个实验代码

```
make
```

运行之后你得到了`tsh`,这是一个最初始的shell,运行此文件你会进入一个命令行,尝试输入一些文字,没有什么反应.使用 `ctrl + d` 退出

除此之外还得到了提到的 `my`开头的几个可执行文件,我们暂时先不去管他们,当然你也可以尝试运行一下,不过没什么输出效果就是了.

接下来我们浏览一下本次实验的核心 `tsh.c` 这个文件

开头引入了几个C库文件,接着可以看到定义的宏.最大行数,参数上限,任务上限,任务的ID上限等等.

```c
/* Misc manifest constants */
#define MAXLINE    1024   /* max line size */
#define MAXARGS     128   /* max args on a command line */
#define MAXJOBS      16   /* max jobs at any point in time */
#define MAXJID    1<<16   /* max job ID */
```

接着是对任务JOB state的一些宏,我们放到后面去说明

接着是一些全局变量

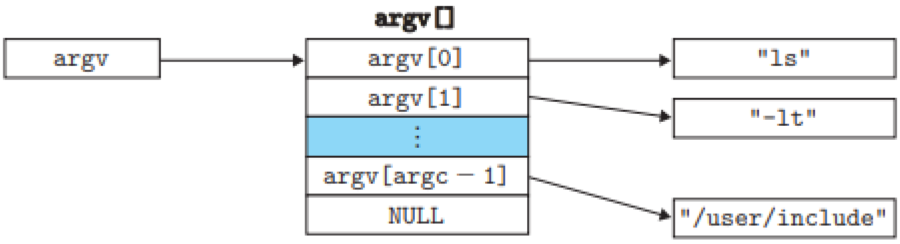
```c
/* Global variables */
extern char **environ;      /* defined in libc */
char prompt[] = "tsh> ";    /* command line prompt (DO NOT CHANGE) */
int verbose = 0;            /* if true, print additional output */
int nextjid = 1;            /* next job ID to allocate */
char sbuf[MAXLINE];         /* for composing sprintf messages */
struct job_t {              /* The job struct */
    pid_t pid;              /* job PID */
    int jid;                /* job ID [1, 2, ...] */
    int state;              /* UNDEF, BG, FG, or ST */
    char cmdline[MAXLINE];  /* command line */
};
struct job_t jobs[MAXJOBS]; /* The job list */
```

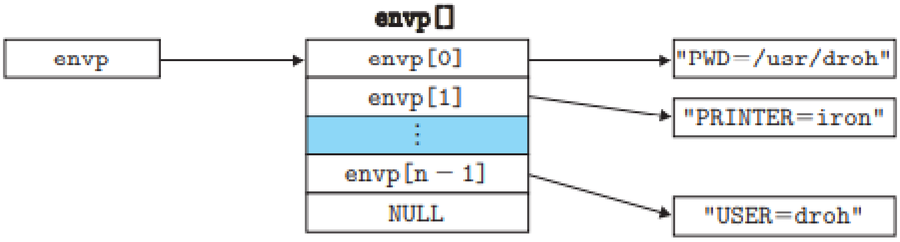
其中 `extern char **environ` 这个变量是环境变量,大概率会在 `execve` 函数调用的时候被作为第三个参数传入;jobs则是任务列表

```c
#include <unistd.h>

int execve(const char *filename, const char *argv, const char *envp[]);
// 如果成功不返回,如果错误返回-1
```

execve函数加载并运行可执行目标文件filename,并且带参数列表argv和环境变量列表envp,与fork返回两次不同.execve调用一次并且不返回,只有出现错误,比如找到不到filename才会返回到调用程序





其中参数列表的数据结构如上图所示

argv变量指向一个null结尾的指针数组,每个指针指向一个参数字符串,通常来说argv[0]是可执行目标文件的名字

envp变量指向一个null结尾的指针数据,每个指针指向一个字符串,每个串都是类似"name=value"的键值对

接下来是我们本次实验需要完成的一些函数,这里已经体现列了出来

```
/* Here are the functions that you will implement */
void eval(char *cmdline);
int builtin_cmd(char **argv);
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);
```

然后是一些本次实验中作者提供的一些辅助函数,我们需要在完成这几个函数的同时利用到作者提供的这些辅助函数来执行或者或许信息,具体的函数内容我们放到对应的出现位置再去讲解

接下来就是main函数了,先判断一下有没有携带`-h -v -p`参数,如果使用了那么执行相关函数.接下来使用`Signal`将四个中断信号注册到对应的方法.`initjob`将所有的任务初始化,进入while(1)死循环,输出prompt即`tsh>` ,刷新标准输出流然后进入`eval`,接着再次刷新,进入循环.

现在我们的eval函数暂时未完成,所以整个程序运行的流程大致如此.

除此之外我们注意到Makefile中提供了很方便的测试指令,比如想要测试trace11是否通过,可以使用

```
make test11
```

检测是否正确,可以使用

```
make rtest11
```

这里由于两次pid不同所以输出会有一点点差异,我们可以对比两次结果来判断是否正确实现了shell

trace01

```
#
# trace01.txt - Properly terminate on EOF.
#
CLOSE
WAIT
```

第一关的输入是CLOSE和WAIT,这两个命令并没有什么作用,所以什么也不需要改动

trace02

```
#
# trace02.txt - Process builtin quit command.
#
quit
WAIT
```

第二关需要我们实现内置命令"quit",当输入quit之后退出当前shell.我们完善一下eval函数解析命令行

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    char buf[MAXLINE];
    int bg;
    strcpy(buf,cmdline);
    bg=parseline(buf,argv);
    if (argv[0] == NULL) return;
    builtin_cmd(argv);//读取到quit就退出了
    return;
}
int builtin_cmd(char **argv)
{
    if(!strcmp(argv[0],"quit")) exit(0);
    return 0;     /* not a builtin command */
}
```

这里的parseline是一个解析命令行的函数,可以将传入的字符串通过空格拆分,并将拆分后的结果保存在argv中

它的返回值用于判断是否是一个后台进程,如果是返回1否则返回0

然后判断一下是否是空输入,是则直接返回

接下来实现builtin_cmd对于内置命令的解析,判断第一个参数是否是"quit",如果是则直接退出

trace03

```
#
# trace03.txt - Run a foreground job.
#
/bin/echo tsh> quit
quit
```

第三关加入了一个新的测试命令,其中`/bin/echo`是Linux中一个文件,一个可执行程序.它的作用是输出后面的字符串.你可以直接在命令行中使用这个指令

```
echo hello world
```

> 这里直接使用echo是因为默认会把/bin目录加入到环境变量之中,它与使用/bin/echo是完全等价的

所以如果想要在我们的shell中执行/bin/echo,我们需要在解析完成之后创建一个子进程,然后交由操作系统去执行

```c
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    char buf[MAXLINE];
    int bg;
    pid_t pid;
    strcpy(buf,cmdline);
    bg=parseline(buf,argv);
    if (argv[0] == NULL) return;
    if (!builtin_cmd(argv)) {
        if ((pid = fork() == 0)) {
            if (execve(argv[0],argv,environ) < 0) {
                printf("%s: Command not found\n",argv[0]);
                exit(0);
            }
        }
    }
    return;
}
```

先使用builtin_cmd判断是否是内置的指令,如果不是则fork一个子进程,使用execve函数去执行当前的参数.execve函数的三个参数分别是argv[0]即"/bin/echo"这个程序的名字,argv其余参数包括需要输出的字符串,environ环境变量信息,这个值直接使用整个shell中的环境变量信息

trace04

```
#
# trace04.txt - Run a background job.
#
/bin/echo -e tsh> ./myspin 1 \046
./myspin 1 &
```

第四关首先是-e,这是echo的一个参数,用于激活转义字符,这里我们不需要考虑太多

然后是运行了一个程序myspin,参数是1,并且在后台执行.

> 这里的myspin函数实现很简单,就是休眠n秒,n为传入的参数.这里是休眠1s

这里我先放出修改后的代码,然后我们一点一点看

```
void eval(char *cmdline) {
  char *argv[MAXARGS];
  char buf[MAXLINE];
  int bg;
  pid_t pid;
  strcpy(buf, cmdline);
  bg = parseline(buf, argv);
  if (argv[0] == NULL)
    return;

  sigset_t mask_all, mask_one, prev;
  sigfillset(&mask_all);
  sigemptyset(&mask_one);
  sigaddset(&mask_one, SIGCHLD);
  if (!builtin_cmd(argv)) {
    sigprocmask(SIG_BLOCK, &mask_one, &prev);
    if ((pid = fork()) == 0) {
      sigprocmask(SIG_SETMASK, &prev, NULL);
      setpgid(0, 0);
      if (execve(argv[0], argv, environ) < 0) {
        printf("%s: Command not found\n", argv[0]);
        exit(0);
      }
    }
    if (!bg) {
      // 对于前台进程，添加job后解除阻塞，并通过waitfg等待子进程结束后回收
      sigprocmask(SIG_BLOCK, &mask_all, NULL);
      addjob(jobs, pid, FG, cmdline);
      sigprocmask(SIG_SETMASK, &prev, NULL);
      waitfg(pid);
    } else {
      // 后台进程不需要等待子进程，进程结束之后收到SIGCHLD信号回收即可
      sigprocmask(SIG_BLOCK, &mask_all, NULL);
      addjob(jobs, pid, BG, cmdline);
      sigprocmask(SIG_SETMASK, &prev, NULL);
```

```c
        printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
      }
    }

    return;
  }
  void waitfg(pid_t pid) {
    sigset_t mask;
    sigemptyset(&mask);
    while (pid == fgpid(jobs))
      sigsuspend(&mask);
    return;
  }
  void sigchld_handler(int sig) {
    pid_t pid;
    int status;
    sigset_t mask_all, prev;
    sigfillset(&mask_all);
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
      if (WIFEXITED(status)) // 正常退出 delete
      {
        sigprocmask(SIG_BLOCK, &mask_all, &prev);
        deletejob(jobs, pid);
        sigprocmask(SIG_SETMASK, &prev, NULL);
      }
    }
    return;
  }
```

> 如果你对这部分代码感到疑惑,请重新阅读第八章

首先是加入了信号和阻塞,在fork之前阻塞SIGCHLD信号,以防子进程和父进程竞争.判断该进程是前台进程还是后台进程之后使用addjob添加对应的job类型,然后解除阻塞,再去响应子进程结束信号SIGCHLD,确保addjob操作deletejob操作顺序对应

如果是前台进程比如echo,那么使用waitfg等待前台进程结束,waitfg的具体实现就是调用fgpid找到当前的前台进程,然后判断处于前台的进程pid是否是该pid,即前台进程仍然没有执行结束,那么挂起当前进程sigsuspend等待结束

如果后台进程比如myspin,那么使用addjob加入任务队列之后就不需要关心了,使用printf输出一句话即可(与make rtest04的结果相同)

> 这里使用了开头定义的宏BG FG

对于子进程结束的信号SIGCHLD的处理则完成对应的sigchld_handler,这里的waitpid使用的第三个参数位置是`WNOHANG | WUNTRACED`代表立即返回,然后判断一下返回的状态,如果是正常退出那么deletejob删除对应任务即可

除此之外还需要注意在创建子进程之后使用了`setpgid(0, 0)`,这里的作用是将子进程的gpid组设置与父进程pid相同

运行`make test04`后结果如下,其中pid可能不同,导致pid2jid的结果不同

> 这里简单解释一下,2同英文to,所以日常编程之中经常使用2来代指to表示转换,这里的pid2jid类似pid_to_jid

```
./sdriver.pl -t trace04.txt -s ./tsh -a "-p"
#
# trace04.txt - Run a background job.
#
tsh> ./myspin 1 &
[1] (980) ./myspin 1 &
```

trace05

```
#
# trace05.txt - Process jobs builtin command.
#
/bin/echo -e tsh> ./myspin 2 \046
./myspin 2 &

/bin/echo -e tsh> ./myspin 3 \046
./myspin 3 &

/bin/echo tsh> jobs
jobs
```

第五关是分别运行了前台echo,后台myspin,前台echo,后台myspin,前台echo,前台jobs

这里需要实现一个内置命令jobs,功能是显示目前任务列表中的所有任务以及所有属性,这里可以利用listjob函数

```
int builtin_cmd(char **argv) {
  if (!strcmp(argv[0], "quit")) {
      exit(0);
  }
  if (!strcmp(argv[0],"jobs")) {
      listjobs(jobs);
      return 1;
  }
  return 0; /* not a builtin command */
}
```

trace06

```
#
# trace06.txt - Forward SIGINT to foreground job.
#
/bin/echo -e tsh> ./myspin 4
```

```
./myspin 4

SLEEP 2
INT
```

第六关加入了INT,表示接收到了中断信号SIGINT(即CTRL_C),那么结束前台进程,此时需要我们完成sigint_handler

```c
void sigchld_handler(int sig) {
  pid_t pid;
  int status;
  sigset_t mask_all, prev;
  sigfillset(&mask_all);
  while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
    if (WIFEXITED(status)) // 正常退出 delete
    {
      sigprocmask(SIG_BLOCK, &mask_all, &prev);
      deletejob(jobs, pid);
      sigprocmask(SIG_SETMASK, &prev, NULL);
    } else if (WIFSIGNALED(status)) // 信号退出 delete
    {
      sigprocmask(SIG_BLOCK, &mask_all, &prev);
      printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid,
WTERMSIG(status));
      deletejob(jobs, pid);
      sigprocmask(SIG_SETMASK, &prev, NULL);
    }
  }
  return;
}

void sigint_handler(int sig) {
    sigset_t mask_all, prev;
    sigfillset(&mask_all);
    sigprocmask(SIG_SETMASK, &mask_all, &prev);
    pid_t pid = fgpid(jobs);
    sigprocmask(SIG_SETMASK, &prev, NULL);
    if (pid > 0) {
        kill(-pid,sig);
    }
    return;
}
```

这里修改了两个地方,首先是完成了sigint_handler,如果接收到了INT信号,则通过fgpid找到当前的前台进程,使用kill杀死这个进程.这里kill的第一个参数是负数,代表杀死这个进程组的所有进程,对子进程及其所有后代都发送终止信号

另外值得一提的是对于INT终止类型我们将printf的输出内容写在sigchld_handler中而不是sigint_handler,实际上如果你写在sigint_handler中对于这道题来说也是可以的,但是意义完全不同

- 如果写在sigint_handler,那么代表每次处理INT中断的时候输出这句话,与子进程无关

- 如果写在sigchld_handler,那么代表是子进程结束了,并且捕捉到是来自WIFSIGNALED的信号退出方式,对应INT,所以输出这句话

后者的写法才是正确的,前者的写法会导致来自外界的中断信号无法别识别,你会在trace16中看到差距

此外应该注意对于WIFSIGNALED的处理中应该先printf后delete,其实顺序无所谓,只是为了和tshref的输出相同

trace07

第七关没有任何新的命令,检测你之前是否是正确完成了所有的函数,有没有投机取巧

如果你之前都正确完成了,那么不需要改动直接通过

trace08

```
#
# trace08.txt - Forward SIGTSTP only to foreground job.
#
/bin/echo -e tsh> ./myspin 4 \046
./myspin 4 &

/bin/echo -e tsh> ./myspin 5
./myspin 5

SLEEP 2
TSTP

/bin/echo tsh> jobs
jobs
```

第八关加入了TSTP(CTRL_Z),当接收到了TSTP中断信号（即CTRL_Z),将前台进程挂起,然后输出被挂起的任务,和INT类似,完成两个函数

```c
void sigchld_handler(int sig) {
  pid_t pid;
  int status;
  sigset_t mask_all, prev;
  sigfillset(&mask_all);
  while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
    if (WIFEXITED(status)) // 正常退出 delete
    {
      sigprocmask(SIG_BLOCK, &mask_all, &prev);
      deletejob(jobs, pid);
      sigprocmask(SIG_SETMASK, &prev, NULL);
    } else if (WIFSIGNALED(status)) // 信号退出 delete
    {
      sigprocmask(SIG_BLOCK, &mask_all, &prev);
      printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid,
WTERMSIG(status));
      deletejob(jobs, pid);
```

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

void sigtstp_handler(int sig) {
    sigset_t mask_all, prev;
    sigfillset(&mask_all);
    sigprocmask(SIG_SETMASK, &mask_all, &prev);
    pid_t pid = fgpid(jobs);
    sigprocmask(SIG_SETMASK, &prev, NULL);
    if (pid > 0) {
        kill(-pid,sig);
    }
    return;
}
```

其中sigtstp_handler与sigint_handler实现完全相同,在sigchld_handler中找到对于TSTP(WIFSTOPPED)的信号,通过getjobpid找到对应任务并将其state修改为ST即可

trace09

```

#
# trace09.txt - Process bg builtin command
#
/bin/echo -e tsh> ./myspin 4 \046
./myspin 4 &

/bin/echo -e tsh> ./myspin 5
./myspin 5

SLEEP 2
TSTP

/bin/echo tsh> jobs
jobs

/bin/echo tsh> bg %2
bg %2

/bin/echo tsh> jobs
jobs
```

其中加入了bg命令,我们可以在开头的注释中找到对应命令所表示的含义

```
/*
 * Jobs states: FG (foreground), BG (background), ST (stopped)
 * Job state transitions and enabling actions:
 *     FG -> ST  : ctrl-z
 *     ST -> FG  : fg command
 *     ST -> BG  : bg command
 *     BG -> FG  : fg command
 * At most 1 job can be in the FG state.
 */
```

其中bg command代表将一个ST进程变为BG进程,即将挂起的进程调入后台执行,我们需要完成builtin_cmd和do_bgfg函数

```
int builtin_cmd(char **argv) {
  if (!strcmp(argv[0], "quit")) {
    exit(0);
  }
  if (!strcmp(argv[0],"jobs")) {
    listjobs(jobs);
    return 1;
  }
  if (!strcmp(argv[0], "bg")) {
    do_bgfg(argv);
    return 1;
  }
  return 0; /* not a builtin command */
}
void do_bgfg(char **argv) {

    int jid;
    struct job_t *job;
    if (!strcmp(argv[0], "bg")) {
        jid = atoi(&argv[1][1]);
        job=getjobjid(jobs,jid);
        job->state=BG;
        kill(-(job->pid),SIGCONT);
        printf("[%d] (%d) %s", jid, job->pid, job->cmdline);
    }
    return;
}
```

这里使用了atoi的C库函数,将字符串转为int类型,因为bg的命令的输入格式是 % + jid, 所以使用argv[1][1]获取该jid的字符串的起始地址

通过getjobjid找到对应的job后修改state为BG,同时发送SIGCONT信号使其执行.单纯修改state毫无意义,必须发送信号使其真正由挂起状态转入运行状态

trace10

第十关类似,只不过是fg命令

```
/*
 * Jobs states: FG (foreground), BG (background), ST (stopped)
 * Job state transitions and enabling actions:
 *     FG -> ST  : ctrl-z
 *     ST -> FG  : fg command
 *     ST -> BG  : bg command
 *     BG -> FG  : fg command
 * At most 1 job can be in the FG state.
 */
```

可以看到fg命令有两种情况,分别是将一个挂起进程调入前台,和将一个后台进程调入前台

```
int builtin_cmd(char **argv) {
  if (!strcmp(argv[0], "quit")) {
    exit(0);
  }
  if (!strcmp(argv[0],"jobs")) {
    listjobs(jobs);
    return 1;
  }
  if (!strcmp(argv[0], "bg") || !strcmp(argv[0],"fg")) {
    do_bgfg(argv);
    return 1;
  }
  return 0; /* not a builtin command */
}

void do_bgfg(char **argv) {

    int jid;
    struct job_t *job;
    if (!strcmp(argv[0], "bg")) {
        jid = atoi(&argv[1][1]);
        job=getjobjid(jobs,jid);
        job->state=BG;
        kill(-(job->pid),SIGCONT);
        printf("[%d] (%d) %s", jid, job->pid, job->cmdline);
    } else if (!strcmp(argv[0], "fg")) {
        jid=atoi(&argv[1][1]);
        job=getjobjid(jobs,jid);
        if (job->state == ST) {
            //如果是挂起程序就重启并且转到前台，等待结束
            job->state=FG;
            kill(-(job->pid),SIGCONT);
            waitfg(job->pid);
        } else if (job->state == BG) {
```

```
                //如果是后台程序就转到前台并等待结束
                job->state=FG;
                waitfg(job->pid);
            }
        }
        return;
    }
```

前台程序和后台程序主要差异就是shell会不会等待你结束,前台shell主动等待waitfg,然后回收掉,后台就是什么时候结束什么时候返回SIGCHILD,然后回收掉

trace11

不需要修改

trace12

不需要修改

trace13

不需要修改

trace14

本关主要是测试所有的命令，判断是否正确。这一关没有技术难关，主要就是对照输出结果完善你的shell对于错误的处理

注意大小写！ 注意标点！ 注意空格！

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

void do_bgfg(char **argv) {

    int jid;
    struct job_t *job;
    if(argv[1]==NULL){
        printf("%s command requires PID or %%jobid argument\n",argv[0]);
        return ;
    }
    if(argv[1][0]=='%'){
        jid=atoi(&argv[1][1]);
        job=getjobjid(jobs,jid);
        if(job==NULL){
```

```
            printf("%%%d: No such job\n",jid);
            return ;
        }
    } else if(isdigit(argv[1][0])){
        jid = atoi(argv[1]);
        job=getjobjid(jobs,jid);
        if(job==NULL){
            printf("(%d): No such process\n",jid);
            return ;
        }

    } else {
        printf("%s: argument must be a PID or %%jobid\n",argv[0]);
        return ;
    }
    if (!strcmp(argv[0], "bg")) {
        job->state=BG;
        kill(-(job->pid),SIGCONT);
        printf("[%d] (%d) %s", jid, job->pid, job->cmdline);
    } else if (!strcmp(argv[0], "fg")) {
        if (job->state == ST) {
            //如果是挂起程序就重启并且转到前台，等待结束
            job->state=FG;
            kill(-(job->pid),SIGCONT);
            waitfg(job->pid);
        } else if (job->state == BG) {
            //如果是后台程序就转到前台并等待结束
            job->state=FG;
            waitfg(job->pid);
        }
    }
    return;
}
```

trace15

还是大小写和标点空格

trace16

这里使用了myint,对于INT handler的处理需要注意,这一点我们在前文trace06已经提及过了,这里不再赘述

下附完整代码

```
/*
 * tsh - A tiny shell program with job control
 *
 * lzx 2019300003075
 */
#include <ctype.h>
#include <errno.h>
```

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

/* Misc manifest constants */
#define MAXLINE 1024   /* max line size */
#define MAXARGS 128    /* max args on a command line */
#define MAXJOBS 16     /* max jobs at any point in time */
#define MAXJID 1 << 16 /* max job ID */

/* Job states */
#define UNDEF 0 /* undefined */
#define FG 1    /* running in foreground */
#define BG 2    /* running in background */
#define ST 3    /* stopped */

/*
 * Jobs states: FG (foreground), BG (background), ST (stopped)
 * Job state transitions and enabling actions:
 *     FG -> ST  : ctrl-z
 *     ST -> FG  : fg command
 *     ST -> BG  : bg command
 *     BG -> FG  : fg command
 * At most 1 job can be in the FG state.
 */

/* Global variables */
extern char **environ;   /* defined in libc */
char prompt[] = "tsh> "; /* command line prompt (DO NOT CHANGE) */
int verbose = 0;         /* if true, print additional output */
int nextjid = 1;         /* next job ID to allocate */
char sbuf[MAXLINE];      /* for composing sprintf messages */

struct job_t {           /* The job struct */
  pid_t pid;             /* job PID */
  int jid;               /* job ID [1, 2, ...] */
  int state;             /* UNDEF, BG, FG, or ST */
  char cmdline[MAXLINE]; /* command line */
};
struct job_t jobs[MAXJOBS]; /* The job list */
/* End global variables */

/* Function prototypes */

/* Here are the functions that you will implement */
void eval(char *cmdline);
int builtin_cmd(char **argv);
void do_bgfg(char **argv);
void waitfg(pid_t pid);
```

```c
void sigchld_handler(int sig);
void sigtstp_handler(int sig);
void sigint_handler(int sig);

/* Here are helper routines that we've provided for you */
int parseline(const char *cmdline, char **argv);
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);
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);

/*
 * main - The shell's main routine
 */
int main(int argc, char **argv) {
  char c;
  char cmdline[MAXLINE];
  int emit_prompt = 1; /* emit prompt (default) */

  /* Redirect stderr to stdout (so that driver will get all output
   * on the pipe connected to stdout) */
  dup2(1, 2);

  /* Parse the command line */
  while ((c = getopt(argc, argv, "hvp")) != EOF) {
    switch (c) {
    case 'h': /* print help message */
      usage();
      break;
    case 'v': /* emit additional diagnostic info */
      verbose = 1;
      break;
    case 'p':         /* don't print a prompt */
      emit_prompt = 0; /* handy for automatic testing */
      break;
    default:
      usage();
    }
  }
```

```
  /* Install the signal handlers */

  /* These are the ones you will need to implement */
  Signal(SIGINT, sigint_handler);   /* ctrl-c */
  Signal(SIGTSTP, sigtstp_handler); /* ctrl-z */
  Signal(SIGCHLD, sigchld_handler); /* Terminated or stopped child */

  /* This one provides a clean way to kill the shell */
  Signal(SIGQUIT, sigquit_handler);

  /* Initialize the job list */
  initjobs(jobs);

  /* Execute the shell's read/eval loop */
  while (1) {

    /* Read command line */
    if (emit_prompt) {
      printf("%s", prompt);
      fflush(stdout);
    }
    if ((fgets(cmdline, MAXLINE, stdin) == NULL) && ferror(stdin))
      app_error("fgets error");
    if (feof(stdin)) { /* End of file (ctrl-d) */
      fflush(stdout);
      exit(0);
    }

    /* Evaluate the command line */
    eval(cmdline);
    fflush(stdout);
    fflush(stdout);
  }

  exit(0); /* control never reaches here */
}

/*
 * eval - Evaluate the command line that the user has just typed in
 *
 * If the user has requested a built-in command (quit, jobs, bg or fg)
 * then execute it immediately. Otherwise, fork a child process and
 * run the job in the context of the child. If the job is running in
 * the foreground, wait for it to terminate and then return.  Note:
 * each child process must have a unique process group ID so that our
 * background children don't receive SIGINT (SIGTSTP) from the kernel
 * when we type ctrl-c (ctrl-z) at the keyboard.
 */
void eval(char *cmdline) {
  char *argv[MAXARGS];
  char buf[MAXLINE];
  int bg;
  pid_t pid;
  strcpy(buf, cmdline);
```

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

  sigset_t mask_all, mask_one, prev;
  sigfillset(&mask_all);
  sigemptyset(&mask_one);
  sigaddset(&mask_one, SIGCHLD);
  if (!builtin_cmd(argv)) {
    sigprocmask(SIG_BLOCK, &mask_one, &prev);
    if ((pid = fork()) == 0) {
      sigprocmask(SIG_SETMASK, &prev, NULL);
      setpgid(0, 0);
      if (execve(argv[0], argv, environ) < 0) {
        printf("%s: Command not found\n", argv[0]);
        exit(0);
      }
    }
    if (!bg) {
      // 对于前台进程，添加job后解除阻塞，并通过waitfg等待子进程结束后回收
      sigprocmask(SIG_BLOCK, &mask_all, NULL);
      addjob(jobs, pid, FG, cmdline);
      sigprocmask(SIG_SETMASK, &prev, NULL);
      waitfg(pid);
    } else {
      // 后台进程不需要等待子进程，进程结束之后收到SIGCHLD信号回收即可
      sigprocmask(SIG_BLOCK, &mask_all, NULL);
      addjob(jobs, pid, BG, cmdline);
      sigprocmask(SIG_SETMASK, &prev, NULL);
      printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
    }
  }

  return;
}

/*
 * parseline - Parse the command line and build the argv array.
 *
 * Characters enclosed in single quotes are treated as a single
 * argument.  Return true if the user has requested a BG job, false if
 * the user has requested a FG job.
 */
int parseline(const char *cmdline, char **argv) {
  static char array[MAXLINE]; /* holds local copy of command line */
  char *buf = array;          /* ptr that traverses command line */
  char *delim;                /* points to first space delimiter */
  int argc;                   /* number of args */
  int bg;                     /* background job? */

  strcpy(buf, cmdline);
  buf[strlen(buf) - 1] = ' ';   /* replace trailing '\n' with space */
  while (*buf && (*buf == ' ')) /* ignore leading spaces */
    buf++;
```

```
  /* Build the argv list */
  argc = 0;
  if (*buf == '\'') {
    buf++;
    delim = strchr(buf, '\'');
  } else {
    delim = strchr(buf, ' ');
  }

  while (delim) {
    argv[argc++] = buf;
    *delim = '\0';
    buf = delim + 1;
    while (*buf && (*buf == ' ')) /* ignore spaces */
      buf++;

    if (*buf == '\'') {
      buf++;
      delim = strchr(buf, '\'');
    } else {
      delim = strchr(buf, ' ');
    }
  }
  argv[argc] = NULL;

  if (argc == 0) /* ignore blank line */
    return 1;

  /* should the job run in the background? */
  if ((bg = (*argv[argc - 1] == '&')) != 0) {
    argv[--argc] = NULL;
  }
  return bg;
}

/*
 * builtin_cmd - If the user has typed a built-in command then execute
 *    it immediately.
 */
int builtin_cmd(char **argv) {
  if (!strcmp(argv[0], "quit")) {
    exit(0);
  }
  if (!strcmp(argv[0],"jobs")) {
    listjobs(jobs);
    return 1;
  }
  if (!strcmp(argv[0], "bg") || !strcmp(argv[0],"fg")) {
    do_bgfg(argv);
    return 1;
  }
  return 0; /* not a builtin command */
}
```

```c
/*
 * do_bgfg - Execute the builtin bg and fg commands
 */
void do_bgfg(char **argv) {

    int jid;
    struct job_t *job;
    if(argv[1]==NULL){
        printf("%s command requires PID or %%jobid argument\n",argv[0]);
        return ;
    }
    if(argv[1][0]=='%'){
        jid=atoi(&argv[1][1]);
        job=getjobjid(jobs,jid);
        if(job==NULL){
            printf("%%%d: No such job\n",jid);
            return ;
        }
    } else if(isdigit(argv[1][0])){
        jid = atoi(argv[1]);
        job=getjobjid(jobs,jid);
        if(job==NULL){
            printf("(%d): No such process\n",jid);
            return ;
        }

    } else {
        printf("%s: argument must be a PID or %%jobid\n",argv[0]);
        return ;
    }
    if (!strcmp(argv[0], "bg")) {
        job->state=BG;
        kill(-(job->pid),SIGCONT);
        printf("[%d] (%d) %s", jid, job->pid, job->cmdline);
    } else if (!strcmp(argv[0], "fg")) {
        if (job->state == ST) {
            //如果是挂起程序就重启并且转到前台，等待结束
            job->state=FG;
            kill(-(job->pid),SIGCONT);
            waitfg(job->pid);
        } else if (job->state == BG) {
            //如果是后台程序就转到前台并等待结束
            job->state=FG;
            waitfg(job->pid);
        }
    }
    return;
}

/*
 * waitfg - Block until process pid is no longer the foreground process
 */
void waitfg(pid_t pid) {
```

```
  sigset_t mask;
  sigemptyset(&mask);
  while (pid == fgpid(jobs))
    sigsuspend(&mask);
  return;
}

/*****************
 * Signal handlers
 *****************/

/*
 * sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
 *     a child job terminates (becomes a zombie), or stops because it
 *     received a SIGSTOP or SIGTSTP signal. The handler reaps all
 *     available zombie children, but doesn't wait for any other
 *     currently running children to terminate.
 */
void sigchld_handler(int sig) {
  pid_t pid;
  int status;
  sigset_t mask_all, prev;
  sigfillset(&mask_all);
  while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
    if (WIFEXITED(status)) // 正常退出 delete
    {
      sigprocmask(SIG_BLOCK, &mask_all, &prev);
      deletejob(jobs, pid);
      sigprocmask(SIG_SETMASK, &prev, NULL);
    } else if (WIFSIGNALED(status)) // 信号退出 delete
    {
      sigprocmask(SIG_BLOCK, &mask_all, &prev);
      printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid,
WTERMSIG(status));
      deletejob(jobs, pid);
      sigprocmask(SIG_SETMASK, &prev, NULL);
    } else if (WIFSTOPPED(status)) {
      sigprocmask(SIG_BLOCK, &mask_all, &prev);
      struct job_t *job=getjobpid(jobs,pid);
      job->state=ST;
      printf("Job [%d] (%d) stopped by signal
%d\n",pid2jid(pid),pid,WSTOPSIG(status));
      sigprocmask(SIG_SETMASK, &prev, NULL);
    }
  }
  return;
}

/*
 * sigint_handler - The kernel sends a SIGINT to the shell whenver the
 *    user types ctrl-c at the keyboard.  Catch it and send it along
 *    to the foreground job.
 */
void sigint_handler(int sig) {
```

```
    sigset_t mask_all, prev;
    sigfillset(&mask_all);
    sigprocmask(SIG_SETMASK, &mask_all, &prev);
    pid_t pid = fgpid(jobs);
    sigprocmask(SIG_SETMASK, &prev, NULL);
    if (pid > 0) {
        kill(-pid,sig);
    }
    return;
}

/*
 * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
 *     the user types ctrl-z at the keyboard. Catch it and suspend the
 *     foreground job by sending it a SIGTSTP.
 */
void sigtstp_handler(int sig) {
    sigset_t mask_all, prev;
    sigfillset(&mask_all);
    sigprocmask(SIG_SETMASK, &mask_all, &prev);
    pid_t pid = fgpid(jobs);
    sigprocmask(SIG_SETMASK, &prev, NULL);
    if (pid > 0) {
        kill(-pid,sig);
    }
    return;
}

/*********************
 * End signal handlers
 *********************/

/***********************************************
 * Helper routines that manipulate the job list
 **********************************************/

/* clearjob - Clear the entries in a job struct */
void clearjob(struct job_t *job) {
  job->pid = 0;
  job->jid = 0;
  job->state = UNDEF;
  job->cmdline[0] = '\0';
}

/* initjobs - Initialize the job list */
void initjobs(struct job_t *jobs) {
  int i;

  for (i = 0; i < MAXJOBS; i++)
    clearjob(&jobs[i]);
}

/* maxjid - Returns largest allocated job ID */
int maxjid(struct job_t *jobs) {
```

```
  int i, max = 0;

  for (i = 0; i < MAXJOBS; i++)
    if (jobs[i].jid > max)
      max = jobs[i].jid;
  return max;
}

/* addjob - Add a job to the job list */
int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline) {
  int i;

  if (pid < 1)
    return 0;

  for (i = 0; i < MAXJOBS; i++) {
    if (jobs[i].pid == 0) {
      jobs[i].pid = pid;
      jobs[i].state = state;
      jobs[i].jid = nextjid++;
      if (nextjid > MAXJOBS)
        nextjid = 1;
      strcpy(jobs[i].cmdline, cmdline);
      if (verbose) {
        printf("Added job [%d] %d %s\n", jobs[i].jid, jobs[i].pid,
               jobs[i].cmdline);
      }
      return 1;
    }
  }
  printf("Tried to create too many jobs\n");
  return 0;
}

/* deletejob - Delete a job whose PID=pid from the job list */
int deletejob(struct job_t *jobs, pid_t pid) {
  int i;

  if (pid < 1)
    return 0;

  for (i = 0; i < MAXJOBS; i++) {
    if (jobs[i].pid == pid) {
      clearjob(&jobs[i]);
      nextjid = maxjid(jobs) + 1;
      return 1;
    }
  }
  return 0;
}

/* fgpid - Return PID of current foreground job, 0 if no such job */
pid_t fgpid(struct job_t *jobs) {
  int i;
```

```
  for (i = 0; i < MAXJOBS; i++)
    if (jobs[i].state == FG)
      return jobs[i].pid;
  return 0;
}

/* getjobpid  - Find a job (by PID) on the job list */
struct job_t *getjobpid(struct job_t *jobs, pid_t pid) {
  int i;

  if (pid < 1)
    return NULL;
  for (i = 0; i < MAXJOBS; i++)
    if (jobs[i].pid == pid)
      return &jobs[i];
  return NULL;
}

/* getjobjid  - Find a job (by JID) on the job list */
struct job_t *getjobjid(struct job_t *jobs, int jid) {
  int i;

  if (jid < 1)
    return NULL;
  for (i = 0; i < MAXJOBS; i++)
    if (jobs[i].jid == jid)
      return &jobs[i];
  return NULL;
}

/* pid2jid - Map process ID to job ID */
int pid2jid(pid_t pid) {
  int i;

  if (pid < 1)
    return 0;
  for (i = 0; i < MAXJOBS; i++)
    if (jobs[i].pid == pid) {
      return jobs[i].jid;
    }
  return 0;
}

/* listjobs - Print the job list */
void listjobs(struct job_t *jobs) {
  int i;

  for (i = 0; i < MAXJOBS; i++) {
    if (jobs[i].pid != 0) {
      printf("[%d] (%d) ", jobs[i].jid, jobs[i].pid);
      switch (jobs[i].state) {
      case BG:
        printf("Running ");
```

```
        break;
      case FG:
        printf("Foreground ");
        break;
      case ST:
        printf("Stopped ");
        break;
      default:
        printf("listjobs: Internal error: job[%d].state=%d ", i, jobs[i].state);
      }
      printf("%s", jobs[i].cmdline);
    }
  }
}
/******************************
 * end job list helper routines
 ******************************/

/***********************
 * Other helper routines
 ***********************/

/*
 * usage - print a help message
 */
void usage(void) {
  printf("Usage: shell [-hvp]\n");
  printf("   -h   print this message\n");
  printf("   -v   print additional diagnostic information\n");
  printf("   -p   do not emit a command prompt\n");
  exit(1);
}

/*
 * unix_error - unix-style error routine
 */
void unix_error(char *msg) {
  fprintf(stdout, "%s: %s\n", msg, strerror(errno));
  exit(1);
}

/*
 * app_error - application-style error routine
 */
void app_error(char *msg) {
  fprintf(stdout, "%s\n", msg);
  exit(1);
}

/*
 * Signal - wrapper for the sigaction function
 */
handler_t *Signal(int signum, handler_t *handler) {
  struct sigaction action, old_action;
```

```
  action.sa_handler = handler;
  sigemptyset(&action.sa_mask); /* block sigs of type being handled */
  action.sa_flags = SA_RESTART; /* restart syscalls if possible */

  if (sigaction(signum, &action, &old_action) < 0)
    unix_error("Signal error");
  return (old_action.sa_handler);
}

/*
 * sigquit_handler - The driver program can gracefully terminate the
 *    child shell by sending it a SIGQUIT signal.
 */
void sigquit_handler(int sig) {
  printf("Terminating after receipt of SIGQUIT signal\n");
  exit(1);
}
```