Shell Lab 实验报告

李梓童 2017202121

- 实验目标

  完成一个简易的 shell 程序

- 实验过程

编写程序的时候，按照 trace01 – trace16 的顺序来对已有的程序进行增删修改，最终完成能实现所有功能的版本。在实验报告中也按照 trace 的顺序，对解题思路进行描述。

Trace01

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

这个基本上不用我们做什么，刚解压出来的 tsh.c 文件已具有正确退出的功能（在 main 函数的循环语句中，if (feof(stdin))语句中的 exit(0)便可使 shell 顺利退出）。

Trace02

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

这一步要求我们写一个 quit 的内建命令。

这里我们要开始写枢纽作用的 eval 函数，基本思路并不难：利用已经提供的 parseline 函数，将输入的字符串指令切割出来。判断切割出来的指令是不是内建命令，这里涉及到另一个需要我们编写的函数 builtin_cmd。Eval 过程中，若先在 builtin_cmd 函数里找到了内建命令 quit，则直接用内建命令退出（exit(0)）。

参考 CS:APP 中的图 8-24，可得 eval 函数和 builtin_cmd 函数的大体框架，剩余的内容要在接下来的 trace 中补充完整。

Trace03

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

这一步中，我们要写一个 fg 命令。

由于“/bin/echo”不是内建命令、也不是 bg 命令（shell 不需要等待 bg 命令完成，但需要等待 fg 命令完成），故 shell 会另开一个子进程，将该指令行放入 execve 过程执行。父进程等待子进程执行完毕后，用 wait 将其回收。

Trace04

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

这一步中，我们要写一个 bg 命令，即 shell 不需要等待这个命令。

上一步中写到“if(bg==0) wait”,这一步中则要接下去写“else …”。参考 tshref 中的输出格式，可知 else 中应当 print 的语句格式为“‘[%d] (%d) %s’, pid2jid(pid), pid, cmdline”。

这一题中遇到的问题有：

1. 一开始看不出 tshref 输出的东西，不知道[ ]里的是 jid、( )里的是 pid。后来根据书里图 8-24 的代码和已有的 tsh 代码（其中有 jid 的部分）判断。
2. 没有注意到 jid 的累加，导致刚开始的时候出来的 pid2jid 结果总是 0。解决：在父进程 wait 之前先添加 addjob 函数。这个时候的代码是这样的：

```
1.  if(bg==0) addjob(jobs, pid, FG, cmdline);
2.  else addjob(jobs, pid, BG, cmdline);
3.
4.  if(bg==0){
5.      int status;
6.      if(waitpid(pid,&status,0)<0) printf("error");
7.  }
8.  else{
9.      //trace04.txt:output format.
10.     printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
11. }
```

3，加上 addjob 之后，发现出来本该是[1]的 jid 结果总是[2]。在 builtin_cmd 里添加了 jobs 命令后，可以用 listjobs 来查看当前任务列表，如下：

```
tsh> jobs
[1] (24224) Foreground /bin/echo -e tsh> ./myspin 1 \046
[2] (24227) Running ./myspin 1 &
```

我们需要的是第二个任务，而第一个任务还在任务列表中，没有被 delete 掉，所以可以推测是某个地方应该 deletejob，也就是 sigchld_handler。在其中添加语句如下：

```
1.  while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
2.    if (WIFEXITED(status)){
3.          deletejob(jobs, pid);
4.      }
5.  }
```

这里用了 WNOHANG，则当子进程结束的时候便直接返回 0，刚开始用的不是 WNOHANG 而是 0，后期发现用 WNOHANG 可以方便判断输出条件，故改用 WNOHANG。这个时候重新跑，发现 jid 依然是 2，结果与上面输出相同。检查之后，发现是 eval 函数中

父进程的 waitpid 出了问题。采用原来的 `if(waitpid(pid,&status,0)<0)`，则任一子进程终止，父进程便继续运行，而子进程用的是 execve 函数，是不会返回的，也就无从 deletejob。于是此处还需用上 waitfg：

```
1.  void waitfg(pid_t pid)
2.  {
3.    struct job_t* job2;
4.    job2 = getjobpid(jobs,pid);
5.
6.    if(pid == 0) return;
7.    if(job2 != NULL){
8.      while(pid==fgpid(jobs)){}
9.    }
10.   return;
11. }
```

这样琢磨一遍，就能跑出想要的结果了。后来随着 sigchld_handler 和 do_bgfg 的编写，waitfg 也作了修改。由于 fg_reaped 变量（标记 fgjob 是否被回收）的出现，这里变成了通过 fg_reaped 变量来判断是否等待。若 fgjob 被回收，则 fg_reaped=1，可进行下一个输入的处理。

Trace04 是遇到的第一个较大的坎，从一开始不太懂 trace 的输入方法（以为它只输入一条“./myspin 1 &”，所以 shell 只输出一个 jid），到考虑修改 SIGCHLD_handler，到加入 waitfg 函数，每一步都是试错很多次之后才想“是不是得加点新的东西”。

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
```

trace05 主要添加一个内建命令 jobs，而这一步在 trace04 调试时已经做掉了，所以直接进行 trace05 测试，结果和答案一样。

Trace06/Trace07

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

SLEEP 2
INT

#
# trace07.txt - Forward SIGINT only to foreground job.
#
/bin/echo -e tsh> ./myspin 4 \046
./myspin 4 &

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

SLEEP 2
INT

/bin/echo tsh> jobs
jobs
```

这两步都和 SIGINT_handler 有关，且 trace07 是 trace06 的进阶要求，总而言之就是要让 SIGINT 只传给 fgjobs。于是可以在相应的 handler 里加入以下代码：

```
1.  pid_t pid = fgpid(jobs);  //get fgjob pid
2.  kill(pid, sig);  //sending SIGINT
```

跑出来是这样：

```
2017202121@VM-0-17-ubuntu:~/shlab$ ./sdriver.pl -t trace06.txt -s ./tshref
#
# trace06.txt - Forward SIGINT to foreground job.
#
tsh> tsh> ./myspin 4
tsh> Job [1] (2501) terminated by signal 2
tsh> 2017202121@VM-0-17-ubuntu:~/shlab$ ./sdriver.pl -t trace06.txt -s ./tsh
#
# trace06.txt - Forward SIGINT to foreground job.
#
tsh> tsh> ./myspin 4
tsh> tsh> 2017202121@VM-0-17-ubuntu:~/shlab$
```

和答案相比少了一行输出信息，原来子进程被 terminated 的时候没有设置。在 SIGCHLD 的 handler 里加上：

```
1.  if (WIFSIGNALED(status)){
2.      printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(status));
3.      deletejob(jobs, pid);
4.  }
```

结果就正确了。开始时不知道可以用 WTERMSIG，对 signal 后面跟着的整数也思考了一段时间，后来在网上查找函数使用样例解决。

- 此处还应注意一个问题，即进程竞争。fork 以后会在 job 列表里添加 job，

sigchld_handler 回收进程后会在 job 列表中删除，如果信号来的比较早，那么就可能会发生先删除后添加的情况。所以在这个过程中要先阻塞 SIGCHLD 再还原。

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
```

本题要求只向前台任务发送 SIGTSTP。参考前面 trace07 的做法，并不太难。

编写过程中遇到的 bug：1，忘了把 job 的状态改成 ST，结果 listjobs 上面打印不出来。2，在 sigchld_handler 中的 waitpid 上应当在 options 上新增 WUNTRACED，这样便能在子进程暂停的时候返回。(后期对 sigchld_handler 进行了修改，加入变量 stopped_child，用以恢复暂停的子进程)

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
```

这一步要求支持输入“bg”后，暂停的后台指令可以继续运行。

开始添加 do_bgfg 函数：

```
1.  ......
2.  tmp = argv[1];
3.
4.  //jid
5.  if(argv[1] [0] == '%') {
6.     int jid = atoi(argv[1] + 1);
7.     struct job_t *job2 = getjobjid(jobs, jid);
8.     stopped_child = job2->pid;
9.      kill(job2->pid, SIGCONT);
10. }
11. ......
```

这里完成的是用 jid 来寻找 job 的部分，用到了 atoi 函数来进行字符串到整数的转换。通过 kill 向 jid 对应的进程发送 SIGCONT，使之继续运行。

Trace10

```
#
# trace10.txt - Process fg builtin command.
#
/bin/echo -e tsh> ./myspin 4 \046
./myspin 4 &

SLEEP 1
/bin/echo tsh> fg %1
fg %1

SLEEP 1
TSTP

/bin/echo tsh> jobs
jobs

/bin/echo tsh> fg %1
fg %1

/bin/echo tsh> jobs
jobs
```

在 trace09 的代码上加点小改动，在 builtin_cmd 和 do_bgfg 里加上 fg 的部分，需注意在 fg 的部分里加入对 fg_reaped 的修正。

```
#
# trace11.txt - Forward SIGINT to every process in foreground process group
#
/bin/echo -e tsh> ./mysplit 4
./mysplit 4

SLEEP 2
INT

/bin/echo tsh> /bin/ps a
/bin/ps a
```

```
#
# trace12.txt - Forward SIGTSTP to every process in foreground process
group
#
/bin/echo -e tsh> ./mysplit 4
./mysplit 4

SLEEP 2
TSTP

/bin/echo tsh> jobs
jobs

/bin/echo tsh> /bin/ps a
/bin/ps a
```

这两步要求可以向全部前台进程发送 SIGINT/SIGTSTP，主要用到了进程组设置，修改了 SIGINT 和 SIGTSTP_handler 中的 kill 函数。

跑出来的结果和 tshref 有一点小出入：

```
tsh> tsh> /bin/ps a
tsh>   PID TTY      STAT   TIME COMMAND
 1227 ttyS0    Ss+    0:00 /sbin/agetty -o -p -- \u --keep-baud 115200,38400,9600 ttyS0 vt220
 1265 tty1     Ss+    0:00 /sbin/agetty -o -p -- \u --noclear tty1 linux
 1660 pts/1    Ss     0:00 /bin/bash
 4755 pts/1    S+     0:00 sudo bash update.sh
 4756 pts/1    S+     0:00 bash update.sh
20346 pts/0    Ss+    0:00 -bash
23492 pts/2    Ss     0:00 -bash
26150 pts/3    Ss+    0:00 -bash
27000 pts/1    S+     0:00 sleep 1m
27003 pts/2    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace11.txt -s ./tsh
27004 pts/2    R+     0:02 ./tsh
27009 pts/2    R      0:00 /bin/ps a
```

这里遇到了解题过程中最耗时间的 bug 之一，./tsh 仍在运行。反复尝试后，从以下几个方面对代码做出了改进：1，采用了 stopped_child 变量，记录我们 fg 的 stopped 进程，在进入 sigchld_handler 后先检查这个变量是否大于零，如果是前台的暂停程序就直接退出。2，使用 while（waitpid）来回收进程时，由于后台中仍有可能正在运行的进程，waitpid 会一直等待这个进程结束。对 sigchld_handler 做出修改，while 改为 if，在回收过程中加入了 fg_reaped 变量。

Trace13

```
#
# trace13.txt - Restart every stopped process in process group
#
/bin/echo -e tsh> ./mysplit 4
./mysplit 4

SLEEP 2
TSTP

/bin/echo tsh> jobs
jobs

/bin/echo tsh> /bin/ps a
/bin/ps a

/bin/echo tsh> fg %1
fg %1

/bin/echo tsh> /bin/ps a
/bin/ps a
```

这一步主要对 do_fgbg 中的 kill 函数做了修改，从 kill 到 killpg。

Trace14

输入检验，主要在 do_bgfg 和 eval 中加入输入合法性检查。

Trace15/Trace16

对 tsh 作最后的检查。

- 实验心得

  我认为本次实验的重点是信号。当系统开始执行指令后，想让一个进程中断、终止或继续都要通过信号，信号把各个进程联系在一起，实现了它们之间的联动。而信号又是非常微观的一个东西，带有一定的隐蔽性，有时候都不知道从哪里发来了个信号，对进程产生了影响。

  难点是信号和 wait 函数的使用。信号的屏蔽和解除屏蔽、不同时刻一个进程给别的进程发送了什么信号，都需要思考。Waitpid 函数的参数设置、用 if 还是用 while，在编程时也作了不少尝试。

  优秀的地方：个人觉得没有特别新奇的地方，实现功能，发现问题后修正，编写代码的流程大致是这样。如果说有预见性的地方，大概就是提前看了还没有做的 trace，比如 trace05 的功能放到 trace04 中使用了、trace06 和 trace07 一起做了，诸如此类。

  通过本次实验，对 wait\fork\signal 等机制有了更深地了解。通过编写与系统相关的代码，熟悉了计算机进程的调用策略。循序渐进地编写代码，对已有代码进行修正，在这一过程中耐心和编程能力得到了提升。对于计算机系统来说，准确比效率更重要。