实验日志 2.8

1、学会编译 tsh.c，调用 tsh 文件 traceXX.txt 的功能验证方法

1） 通过make clean和make指令来对tsh.c文件进行编译，如果没有语法错误将会如下图所示，编译成功。

```
lh@lh-VirtualBox:~/shlab-handout$ make clean
rm -f ./tsh ./myspin ./mysplit ./mystop ./myint *.o *~
lh@lh-VirtualBox:~/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
```

在tracexx.txt文件中都是一条一条已经写好的命令行语句，将会通过文件流输入的方式给予我们写好的shell代码，然后与给的tshref运行结果进行比较，如果相等则说明我们的代码是正确的。

2） 通过“make 跟踪文件”来对我们编写的 tsh.c 文件进行测试，正确的标准结果可以通过“make r 跟踪文件”来对老师给的 tshref 可执行文件进行测试， 如果相同则证明代码正确，实验成功。

```
lh@lh-VirtualBox:~/shlab-handout$ make test01
./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
lh@lh-VirtualBox:~/shlab-handout$ ./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
lh@lh-VirtualBox:~/shlab-handout$ make rtest01
./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
lh@lh-VirtualBox:~/shlab-handout$ ./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
#
# trace01.txt - Properly terminate on EOF.
```

2、用 trace01 和 trace02 比较 tsh 和 tshref 执行结果并分析

1） trace01.txt 执行 close 和 wait 命令行语句。首先是是读出文件中的数据； 然后读取 close 的时候，关闭文件；读取 wait 的时候，等待（在执行的时候并没有出现长时间的停顿）。对于这两个命令，tsh 和 tshref 并没有太大的区别。

```
1   #
2   # trace01.txt - Properly terminate on EOF.
3   #
4   CLOSE
5   WAIT
```

```
lh@lh-VirtualBox:~/shlab-handout$ make test01
./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
lh@lh-VirtualBox:~/shlab-handout$ make rtest01
./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
```

2） trace02.txt跟踪文件是执行quit和wait命令行语句，由于这个时候tsh.c文件并没有进行编写修改，所以tsh.c文件中没有对应quit命令行语句的函数，所以没有执行，也没有退出，停留在等待输入命令行语句的界面，而tshref执行文件的结果是在遇到quit语句后直接退出shell。

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

```
lh@lh-VirtualBox:~/shlab-handout$ make test02
./sdriver.pl -t trace02.txt -s ./tsh -a "-p"
#
# trace02.txt - Process builtin quit command.
#
```

```
lh@lh-VirtualBox:~/shlab-handout$ make rtest02
./sdriver.pl -t trace02.txt -s ./tshref -a "-p"
#
# trace02.txt - Process builtin quit command.
#
```

3、编程实现 quit 内置命令，补齐文件tsh.c 中的函数 eval() 和函数 builtin_cmd()与quit相关的部分

学习现有代码可以了解到主函数main中在满足条件：没有输入结束命令行ctrl-d之前，都会调用一个eval()函数，而函数的参数就是我们输入的命令行，所以如果想要实现quit语句，我们就需要先将eval函数写出来。

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    char buf[MAXLINE];
    sigset_t mask;
    pid_t pid;
    strcpy(buf,cmdline);

    parseline(buf,argv); //读取命令
    if(argv[0]==NULL) return; //忽略空行

    if(!builtin_cmd(argv)) //如果是内置命令那么结束，否则创建新的子进程
    {
        sigemptyset(&mask);
        sigaddset(&mask,SIGCHLD); //在派生子进程之前阻塞SIGCHLD信号
        sigprocmask(SIG_BLOCK,&mask,NULL);
        if((pid=fork())==0) //子进程
        {
            sigprocmask(SIG_UNBLOCK,&mask,NULL); //解除阻塞
            setpgid(0,0);
            //在子进程中通过execve寻找判断是否找到可执行文件
            if(execve(argv[0],argv,environ)<0)
            {
                printf("%s: Command not found.\n",argv[0]); //找不到
                exit(0);
            }
        }
    }
    return;
}
```

同时，我们需要在eval函数中调用builtin_cmd函数，这个函数的作用是判断是否是内置命令行语句，通过strcmp进行比较，如果是则进行相应的操作，函数的参数也应该是我们输入的命令行，由此也可以得到builin_cmd函数如下：

```
int builtin_cmd(char **argv)
{
    if(!strcmp(argv[0],"quit"))
    {
        exit(0);
    }
    return 0;
}
```

代码思路：首先利用 parseline()函数得到输入的命令行；再调用 builtin_cmd 函数进行判断，如果输入为内置命令行则返回 0；不是内置调用 fork()进入子进程，然后调用execve()进行判断是否找到可执行文件，没有找到则打印结果，并执行 exit(0)函数结束进程。在builin_cmd 函数内，判断命令行第一个参数是否为”quit”如果是则 exit(0)， 中止程序。

4. 使用 trace03 验证 quit命令

trace03.txt文件内容如下：

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

结果如下：

```
lh@lh-VirtualBox:~/shlab-handout$ make test03
./sdriver.pl -t trace03.txt -s ./tsh -a "-p"
#
# trace03.txt - Run a foreground job.
#
tsh> quit
lh@lh-VirtualBox:~/shlab-handout$ make rtest03
./sdriver.pl -t trace03.txt -s ./tshref -a "-p"
#
# trace03.txt - Run a foreground job.
#
tsh> quit
```

正确，test03 与 rtest03 结果一致。

5. 了解 eval()与 execve()执行流程和 fork()多进程运行方式

eval的首要任务是调用parseline函数，这个函数解析了以空格分隔的命令行参数，并构造最终会传给execve的向量 argv。第一个参数被假设为要么是一个内置的外壳命令，马上就执行这个命令，要么是一个可执行文件，会在一个新的子进程的上下文加载并允许这个文件。解析命令行之后eval函数调用builtin_cmd 函数。

fork函数在新的子进程中运行相同的程序，新的子进程是父进程的一个复制品。

execve函数在当前进程的上下文中加载并运行一个新的程序。它会覆盖当前进程的地址 空间，但并没有创建一个新的进程。新的程序任然有相同的PID，并且继承了调用execve函数时已打开的所有文件描述符。

exec*库函数在加载一个可执行文件的时候，先生成预处理文件.cpp(把include文件包含进来进行宏替换)、编译成汇编代码.s、编译成目标代码，得到二进制文件.o、再链接成可执行文件、最终运行。

在linux系统中，除了系统启动之后的第一个进程由系统来创建，其余的进程都必须由已存在的进程来创建，新创建的进程叫子进程，而创建子进程的进程为父进程。在系统启动及完成初始化之后，linux自动创建的进程叫做根进程。根进程也就是最初的父进程。linux中父进程荣国分裂的方式来创建子进程，创建一个子进程的系统调用叫做fork()，fork函数创建进程的实质是子进程对父进程的复制。