哈爾濱工業大學

实验报告

实 验（四）

题　　目　微壳(TinyShell)

专　　业　人工智能

学　　号　2021112845

班　　级　2103601

学　　生　张智雄

指 导 老 师　郑贵滨

实 验 地 点　G.709

实 验 日 期　2023.4.30

计算学部

目 录

第 1 章 实验基本信息

1.1 实验目的

理解现代计算机系统进程与并发的基本知识

掌握 linux 异常控制流和信号机制的基本原理和相关系统函数

掌握 shell 的基本原理和实现方法

深入理解 Linux 信号响应可能导致的并发冲突及解决方法

培养 Linux 下的软件系统开发与测试能力

1.2 实验环境与工具

1.2.1 硬件环境

X64 CPU；2.30GHz；16G RAM；1.5THD disk

1.2.2 软件环境

Windows11 64 位；Vmware Workstation 17 Pro；Ubuntu 22.10

1.2.3 开发工具

Visual Studio 2019 64 位；CodeBlocks 64 位；vim+gcc

1.3 实验预习

上实验课前，必须认真预习实验指导书（PPT 或 PDF）

了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。

了解进程、作业、信号的基本概念和原理

了解 shell 的基本原理

熟知进程创建、回收的方法和相关系统函数

熟知信号机制和信号处理相关的系统函数

第 2 章 实验预习

总分 20 分

2.1 进程的概念、创建和回收方法（5 分）

进程：一个执行程序中的实例，系统中的每个程序都运行在某个进程的上下文(context)中。

每次当用户通过向 shell 输入一个可执行目标文件的名字，运行程序时，shell 就会创建一个新的进程，然后就在这个新进程的上下文中运行这个可执行目标文件。应用程序也能够创建新进程，并且在这个新进程的上下文中运行它们自己的代码或其他应用程序。

创建进程：父进程调用 fork 函数创建一个新的运行的子进程。新创建的子进程几乎但不完全与父进程相同。子进程将得到与父进程用户级虚拟地址空间相同的(但是独立的)一份部分，包括代码和数据段、堆、共享库以及用户栈。而父进程与新创建的子进程之间最大的区别在于它们有不同的 PID。

回收方法：当进程终止时，他仍然消耗系统资源，因而需要主动回收。父进程使用 wait 或 waitpid 函数来等待子进程终止或者停止。具体过程为：父进程执行回收（wait 或 waitpid）、父进程挂起直至收到子进程的退出状态、内核从系统中删除僵死子进程。

2.2 信号的机制、种类（5 分）

信号：一个信号就是一条小消息，它通知进程系统中发生了一个某种类型的事件。每种信号类型都对应某种系统的事件。

低层的硬件异常是由内核异常处理程序处理的，通常用户进程不可见，信号提供了一种机制，通知用户进程发生了这些异常。

信号可以被进程阻塞（延迟接收处理）、忽略、捕获并自定义处理或执行默认处理行为。但信号不能累计，每一种信号最多只能被发送 1 次直到被接收并清空标志位。

下图 2-1 展示了 Linux 系统上支持的 30 中不同类型的信号，其类型使用小整数 ID 来标识的，信号中唯一的信息是它的 ID 和它的到达。

序号	名称	默认行为	相应事件
1	SIGHUP	终止	终端线挂断
2	SIGINT	终止	来自键盘的中断
3	SIGQUIT	终止	来自键盘的退出
4	SIGILL	终止	非法指令
5	SIGTRAP	终止并转储内存①	跟踪陷阱
6	SIGABRT	终止并转储内存①	来自 abort 函数的终止信号
7	SIGBUS	终止	总线错误
8	SIGFPE	终止并转储内存①	浮点异常
9	SIGKILL	终止②	杀死程序
10	SIGUSR1	终止	用户定义的信号 1
11	SIGSEGV	终止并转储内存①	无效的内存引用（段故障）
12	SIGUSR2	终止	用户定义的信号 2
13	SIGPIPE	终止	向一个没有读用户的管道做写操作
14	SIGALRM	终止	来自 alarm 函数的定时器信号
15	SIGTERM	终止	软件终止信号
16	SIGSTKFLT	终止	协处理器上的栈故障
17	SIGCHLD	忽略	一个子进程停止或者终止
18	SIGCONT	忽略	继续进程如果该进程停止
19	SIGSTOP	停止直到下一个 SIGCONT②	不是来自终端的停止信号
20	SIGTSTP	停止直到下一个 SIGCONT	来自终端的停止信号
21	SIGTTIN	停止直到下一个 SIGCONT	后台进程从终端读
22	SIGTTOU	停止直到下一个 SIGCONT	后台进程向终端写
23	SIGURG	忽略	套接字上的紧急情况
24	SIGXCPU	终止	CPU 时间限制超出
25	SIGXFSZ	终止	文件大小限制超出
26	SIGVTALRM	终止	虚拟定时器期满
27	SIGPROF	终止	剖析定时器期满
28	SIGWINCH	忽略	窗口大小变化
29	SIGIO	终止	在某个描述符上可执行 I/O 操作
30	SIGPWR	终止	电源故障

图 2-1　Linux 信号

2.3 信号的发送方法、阻塞方法、处理程序的设置方法（5 分）

发送信号：内核能通过更新目的进程上下文中的某个状态项 struct sigpending pending)，来实现发送（递送）一个信号给目的进程。其方法包括以下四种：

1) 用/bin/kill 程序发送信号。/bin/kill 程序可以向另外的进程发送任意的信号。

2) 从键盘发送信号。在键盘上输入 Ctrl+C/Ctrl+Z 会导致内核发送一个 SIGINT /SIGTSTP 信号到前台进程组中的每个作业。

3) 用 kill 函数发送信号。进程通过调用 kill 函数发送信号给其他进程（包括它们自己）。

4) 用 alarm 函数发送信号。进程可以通过调用 alarm 函数从而向它自己发送 SIGALRM 信号

阻塞信号：一个进程可以有选择性地阻塞接收某种信号。当一种信号被阻塞时，它仍可以被发送，但是产生的待处理信号不会被接受。Linux 提供阻塞信号的隐式和显式的机制。

1) 隐式阻塞机制：内核默认阻塞任何当前处理程序正在处理信号类型的待处理信号。

2) 显式阻塞机制：应用程序可以使用 sigprocmask 函数和它的辅助函数，明确地阻塞和解除阻塞选定的信号。

处理程序的设置：进程可以通过使用 signal 函数修改和信号相关联的默认行为。唯一的例外是 SIGSTOP 和 SIGKILL,它们的默认行为是不能修改的。

1) 函数原型 sighandler_t *signal(int sig, sighandler_t *handler)
2) 功能：设置信号的处理函数或恢复默认函数行为。
3) handler 的不同取值：
   - SIG_IGN：忽略类型为 sig 的信号
   - SIG_DFL：类型为 sig 的信号行为恢复为默认行为
   - handler 是用户自定义函数的地址，这个函数称为信号处理程序
4) 返回值：
   - 设置成功，返回原处理函数指针
   - 设置失败，返回 SIG_ERR
   - #define SIG_ERR(void (*)()) -1

2.4 什么是 shell，简述其功能和处理流程（5 分）

Shell：一个交互型用户程序，代表用户运行其他程序。具体而言，shell 是指为使用者提供操作界面的软件（command interpreter，命令解析器）。它类似于 DOS 下的 COMMAND.COM 和后来的 cmd.exe。它接收用户命令，然后调用相应的应用程序。

功能：shell 独立于内核，是连接内核和应用程序的桥梁。其提供了一个图形化或命令行式界面，用户能够通过访问这个界面访问操作系统内核的服务。

处理流程：

1) 读取从键盘输入的命令。
2) 判断命令是否正确，且将命令行的参数改造为系统调用 execve()内部处理所要求的形式。
3) 终端进程调用 fork()来创建子进程，自身调用 wait()来等待子进程完成。
4) 当子进程运行时，调用 execve()函数，同时根据命令的名字指定的文件到目录中查找可行性文件，调入内存并执行这个命令。
5) 当子进程完成处理后，向父进程报告，此时终端进程被唤醒，做完必要的判别工作后，再发提示符，让用户输入新命令。

第 3 章 TinyShell 的设计与实现

总分 45 分

3.1 设计

3.1.1 void eval(char *cmdline)函数（10 分）

函数功能：解析和解释命令行的主例程

参　　数：char *cmdline 命令行字符串的起始指针

处理流程：

1. 提取命令参数：调用 parseline 函数将输入命令行字符串 cmdline 分割为参数数组 argv（忽略空命令行），并根据结尾字符是否为'&'返回标志量 bg，决定该操作后台运行还是前台运行。

2. 判断内置函数：调用 buildin_cmd 函数判断输入指令是否为内置命令，若为内置命令则立即执行，执行后返回。

3. 设置阻塞信号：若不为内置命令，阻塞 SIGCHLD、SIGINT 和 SIGTSTP 信号，避免目标命令与上述阻塞信号到达之间的竞争。

4. 创建子进程：创建一个子进程，解除子进程对上述信号的阻塞，并赋予子进程一个新的进程组 ID，而后在子进程中调用 execve 函数载入并运行该作业。

5. 父进程控制：同时父进程将此子进程加入 jobs 列表，此时再解除父进程对上述信号的阻塞。根据标志量 bg 判断此作业是否后台运行，若否，父进程挂起，执行 waitfg 直至作业完成；若是，打印该作业 pid 等信息，等待下一命令的输入。

要点分析：

1. 将内置命令集中判断并封装，增加了代码可读性以及健壮性。

2. 在父进程中，阻塞 SIGCHLD、SIGINT 和 SIGTSTP 信号，直至将作业添加到作业列表中。避免在将作业添加到作业列表时，shell 捕获并处理上述信号。

3. 每个新作业必须获得一个新的进程组 ID，这样内核就不会向所有 shell 作业（包括 tsh 和 tsh 创建的进程）发送 ctrl-c 和 ctrl-z 信号。

3.1.2 int builtin_cmd(char **argv)函数（5 分）

函数功能：识别并解释内置命令(quit, jobs, bg or fg)

参　　数：char **argv 指向参数数组 argv 的指针

处理流程：利用 if-else 语句将输入命令与内置命令逐一比较，若匹配则直接运行相应的指令后，返回 1（quit 指令直接退出）；反之，则返回 0。

要点分析：

1. 可以用 else if 取代连续 if，减少冗余匹配次数。
2. 注意到 bg 和 fg 的执行函数相同，因而可以合并到一个 if 语句中。

```
304 int builtin_cmd(char **argv)
305 {
306     /*
307      * 判断是否是内置函数，如果是，执行并返回1；如果不是，返回0
308      * 内置函数：quit, jobs, bg or fg
309      */
310     if(!strcmp(argv[0],"quit")){
311         exit(0);                /* 退出quit */
312     }
313 
314     else if(!strcmp(argv[0],"jobs")){
315         listjobs(jobs);         /* 打印当前所有暂停的进程信息 */
316         return 1;
317     }
318 
319     else if(!strcmp(argv[0],"bg")||!strcmp(argv[0],"fg")){
320         do_bgfg(argv);          /* 将作业放到后台或前台运行 */
321         return 1;
322     }
323     else
324         return 0;               /* not a builtin command */
325 }
```

图 3-1-2　builtin_cmd 函数代码

3.1.3 void do_bgfg(char **argv) 函数（5 分）

函数功能：实现内置命令 bg 和 fg

参　　数：char **argv 指向参数数组 argv 的指针

处理流程：

1. 参数检查：检查 bg 或 fg 之后是否有 PID 或%JID 等参数，若为空输出提示信息后返回；若不为空则继续执行。
2. 解析参数：判断参数属于 PID 还是%JID，提取信息后在进程列表 jobs 中寻找对应工作，若找不到则输出“No such process”后返回。
3. 具体执行：判断执行 bg 还是 fg。利用 kill 函数向 PID 或%JID 对应进程发送 SIGCONT 信号，将其工作状态置为相应的 BG 或 FG。其中对于 fg 指令，需要通过 waitfg 函数等待目标进程结束后再返回。

要点分析：

1. 需要正确区分 PID 或%JID，最直接的方法是根据第一位是否为‘%’。
2. 需在程序各个阶段对输入进行检查，对非法输入的处理考虑要周全。
3. 利用 kill 函数实现一个进程向另一个进程发送信号。
4. 对于 fg 指令，需调用 waitfg 等待目标程序在前台运行结束后才返回。

3.1.4 void waitfg(pid_t pid) 函数（5 分）

函数功能：等待一个前台作业结束

参　　数：pid_t pid 进程识别号

处理流程：

1. 调用 fgpid 函数检查当前前台作业的 PID 是否改变
2. 若改变即当前前台作业已结束，跳出循环，waitfg 函数返回；反之则使用 sigsuspend 函数将当前进程挂起，等待下次循环条件判断。

要点分析：

1. fgpid 函数返回当前前台作业的 PID，如果没有，则返回 0；利用其返回值是否改变判断目标进程是否结束。
2. 可以采取在循环体内插入 pause，但 pause 会引入竞争，在 while 测试后和 pause 之前 pid 改变，pause 会永远睡眠。
3. 也可选择采用 sleep 函数，但因为没有很好的办法确定休眠的间隔，仍存在资源的浪费。
4. 因而采用 sigsuspend 函数，不设置对任何信号的阻塞（在此与 pause 方法几乎等价），但增强了代码的可移植性。

```
387  void waitfg(pid_t pid)
388  {
389      /*
390       * 调用fgpid函数，判断目标进程pid是否改变
391       * 通过sigsuspend函数将当前进程挂起
392       */
393      sigset_t mask,prev;
394      sigemptyset(&mask);
395      sigprocmask(SIG_BLOCK, &mask, &prev);
396      while(fgpid(jobs)==pid)
397      {
398          sigsuspend(&prev);
399      }
400      sigprocmask(SIG_BLOCK, &prev, NULL);
401      return;
402  }
```

图 3-1-4　waitfg 函数代码

3.1.5 void sigchld_handler(int sig) 函数（10 分）

函数功能：捕获并处理 SIGCHILD 函数

参　　数：int sig 信号小整数 ID

处理流程：

1. 保护现场：存储程序在运行中的错误代码 errno，防止处理过程中发生其他系统级的错误导致 errno 被修改，而影响进程上下文的正常运行。

2. 进程回收：调用 waitpid 函数回收僵死进程，参数 WHOANG|WUNTRACED 代表立即返回，等待集合中的一个进程终止或收到 SIGCONT 而从停止状态重新开始，若无子进程终止返回 0 值。

3. 状态检查：用 WIFEXITED、WIFSIGNALED、WIFSTOPPED 宏函数判断进程结束状态，正常结束则直接从工作列表删除此工作即可；异常情况或处于暂停状态则显示异常信息及造成原因，进程意外终止也需从工作列表删除。

4. 恢复现场：还原 errno，函数返回。

要点分析：

1. 对处理前的进程信息进行保护，处理完成后恢复，以免发生意外错误。

2. waitpid 函数立即返回，目的是实时回收，获取所有可用的僵尸子进程，但不等待任何其他当前正在运行的子进程终止。

3. 对异常情况使用 WTERMSIG，WSTOPSIG 追溯造成原因。

4. 为避免删除任务时 shell 捕获并处理其他信号，删除时阻塞所有信号的接收，直至任务被删除完成。

```
415 void sigchld_handler(int sig)
416 {
417     int olderrno = errno;              /* 保存程序在运行中的错误代码 */
418     int status;
419     pid_t pid;
420     sigset_t mask, prev;
421     sigfillset(&mask);
422     while((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0){
423         /*
424          *     waitpid回收僵死进程，参数WHOANG|WUNTRACED代表立即返回，
425          *     等待集合中的一个进程终止或收到SIGCONT而从停止状态重新开始，若无子进程终止返回0值
426          *     用WIFEXITED、WIFSIGNALED、WIFSTOPPED宏函数判断进程结束状态
427          *     显示异常情况及造成原因
428          */
429         if (WIFEXITED(status))             /* 进程正常结束 */
430         {
431             /*
432              *     为避免删除任务时shell捕获并处理其他信号
433              *     阻塞所有信号的接收，直至任务被删除
434              */
435             sigprocmask(SIG_BLOCK, &mask, &prev);
436             deletejob(jobs, pid);
437             sigprocmask(SIG_SETMASK, &prev, NULL);
438         }
439 
440         else if (WIFSIGNALED(status))      /* 进程异常结束 */
441         {
442             printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(status));
443             sigprocmask(SIG_BLOCK, &mask, &prev);
```

```
444             deletejob(jobs, pid);
445             sigprocmask(SIG_SETMASK, &prev, NULL);
446         }
447 
448         else if (WIFSTOPPED(status))         /* 进程处于暂停状态 */
449         {
450             printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, WSTOPSIG(status));
451         }
452     }
453     errno = olderrno;
454     return;
455 }
```

图 3-1-5　sigchld_handler 函数代码

3.2 程序实现（tsh.c 的全部内容）（10 分）

```
462 void sigint_handler(int sig)
463 {
464     int olderrno = errno;
465 
466     pid_t pid = fgpid(jobs);
467     if (pid == 0)
468         return;
469     kill(-pid, SIGINT);       /* 向所有前台运行的进程所处进程组发送SIGINT信号 */
470 
471     errno = olderrno;
472     return;
473 }
474 
475 /*
476  * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
477  *     the user types ctrl-z at the keyboard. Catch it and suspend the
478  *     foreground job by sending it a SIGTSTP.
479  */
480 void sigtstp_handler(int sig)
481 {
482     int olderrno = errno;
483 
484     pid_t pid = fgpid(jobs);
485     if(pid == 0)
486     return;
487     struct job_t *p = getjobpid(jobs, pid)
488     p->state = ST;              /* 设置相应工作状态为ST */
489     kill(-pid, SIGTSTP);        /* 向所有前台运行的进程所处进程组发送SIGTSTP信号 */
490     printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, sig);
491                                 /*显示终止信息*/
492     errno = olderrno;
493     return;
494 }
```

图 3-2　其余作业要求函数代码

重点检查代码风格：

（1）　用较好的代码注释说明——5 分

（2）　检查每个系统调用的返回值——5 分

第 4 章 TinyShell 测试

总分 15 分

4.1 测试方法

针对 tsh 和参考 shell 程序 tshref，完成测试项目 4.3.1-4.3.15 的对比测试，并将测试结果截图或者通过重定向保存到文本文件(例如：./sdriver.pl -t trace01.txt -s ./tsh -a "-p" > tshresult01.txt)。

4.2 测试结果评价

tsh 与 tshref 的输出在一下两个方面可以不同：

1. PID

2. 测试文件 trace11.txt, trace12.txt 和 trace13.txt 中的/bin/ps 命令，每次运行的输出都会不同，但每个 mysplit 进程的运行状态应该相同。

除了上述两方面允许的差异，tsh 与 tshref 的输出相同则判为正确，如不同则给出原因分析。

4.3 自测试结果

4.3.1 测试用例 trace01.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
zzxiong@zzxiong-virtual-machine:~/xxxx/lab4/ make test01 ./sdriver.pl -t trace01.txt -s ./tsh -a "-p" # # trace01.txt - Properly terminate on EOF. #	zzxiong@zzxiong-virtual-machine:~/xxxx/lab4/shl make rtest01 ./sdriver.pl -t trace01.txt -s ./tshref -a "-p" # # trace01.txt - Properly terminate on EOF. #
测试结论	相同

4.3.2 测试用例 trace02.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
zzxiong@zzxiong-virtual-machine:~/xxxx/lab4/sh make test02 ./sdriver.pl -t trace02.txt -s ./tsh -a "-p" # # trace02.txt - Process builtin quit command. #	zzxiong@zzxiong-virtual-machine:~/xxxx/lab4/shla make rtest02 ./sdriver.pl -t trace02.txt -s ./tshref -a "-p" # # trace02.txt - Process builtin quit command. #
测试结论	相同

4.3.3 测试用例 trace03.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
zzxiong@zzxiong-virtual-machine:~/xxxx/lab4/s make test03 ./sdriver.pl -t trace03.txt -s ./tsh -a "-p" # # trace03.txt - Run a foreground job. # tsh> quit	zzxiong@zzxiong-virtual-machine:~/xxxx/lab4/shlab make rtest03 ./sdriver.pl -t trace03.txt -s ./tshref -a "-p" # # trace03.txt - Run a foreground job. # tsh> quit
测试结论	相同

4.3.4 测试用例 trace04.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
zzxiong@zzxiong-virtual-machine:~/xxxx/lab4/s make test04 ./sdriver.pl -t trace04.txt -s ./tsh -a "-p" # # trace04.txt - Run a background job. # tsh> ./myspin 1 & [1] (5518) ./myspin 1 &	zzxiong@zzxiong-virtual-machine:~/xxxx/lab4/shla make rtest04 ./sdriver.pl -t trace04.txt -s ./tshref -a "-p" # # trace04.txt - Run a background job. # tsh> ./myspin 1 & [1] (5525) ./myspin 1 &
测试结论	相同

4.3.5 测试用例 trace05.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
zzxiong@zzxiong-virtual-machine:~/xxxx/lab4/sh make test05 ./sdriver.pl -t trace05.txt -s ./tsh -a "-p" # # trace05.txt - Process jobs builtin command. # tsh> ./myspin 2 & [1] (5535) ./myspin 2 & tsh> ./myspin 3 & [2] (5537) ./myspin 3 & tsh> jobs [1] (5535) Running ./myspin 2 & [2] (5537) Running ./myspin 3 &	zzxiong@zzxiong-virtual-machine:~/xxxx/lab4/shla make rtest05 ./sdriver.pl -t trace05.txt -s ./tshref -a "-p" # # trace05.txt - Process jobs builtin command. # tsh> ./myspin 2 & [1] (5544) ./myspin 2 & tsh> ./myspin 3 & [2] (5546) ./myspin 3 & tsh> jobs [1] (5544) Running ./myspin 2 & [2] (5546) Running ./myspin 3 &
测试结论	相同

4.3.6 测试用例 trace06.txt 的输出截图（1 分）

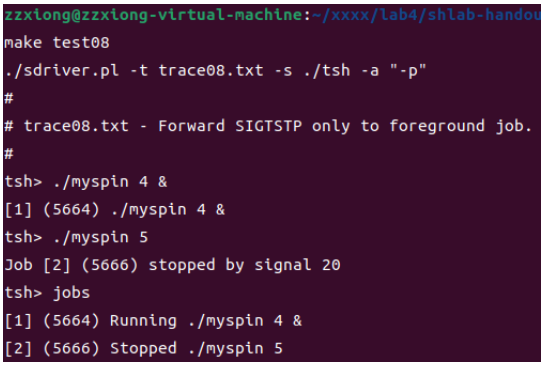
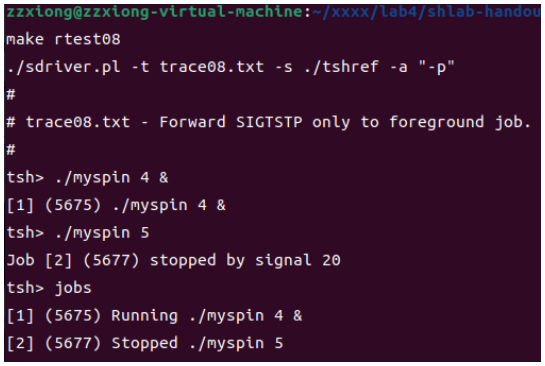
tsh 测试结果	tshref 测试结果
zzxiong@zzxiong-virtual-machine:~/xxxx/lab4/shlab- make test06 ./sdriver.pl -t trace06.txt -s ./tsh -a "-p" # # trace06.txt - Forward SIGINT to foreground job. # tsh> ./myspin 4 Job [1] (5555) terminated by signal 2	zzxiong@zzxiong-virtual-machine:~/xxxx/lab4/shlab- make rtest06 ./sdriver.pl -t trace06.txt -s ./tshref -a "-p" # # trace06.txt - Forward SIGINT to foreground job. # tsh> ./myspin 4 Job [1] (5561) terminated by signal 2
测试结论	相同

4.3.7 测试用例 trace07.txt 的输出截图（1 分）

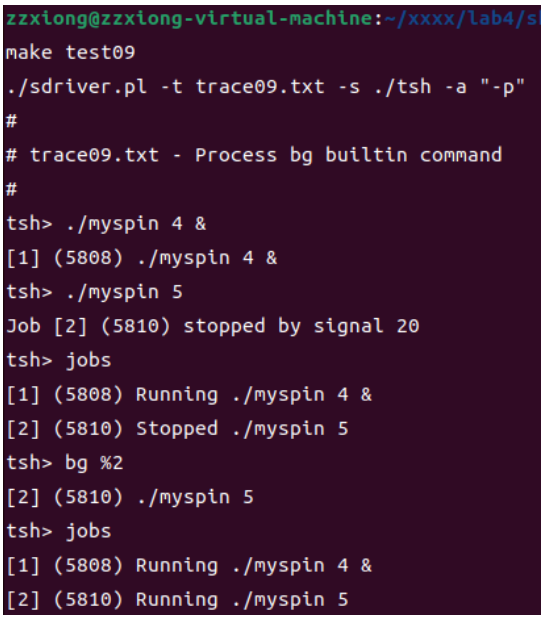
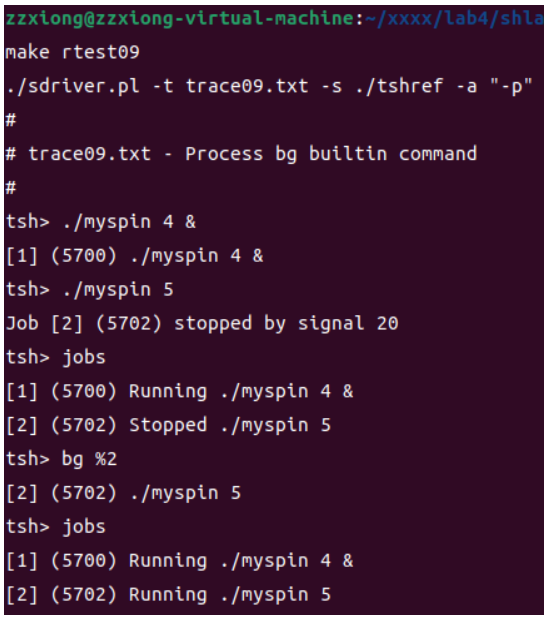
tsh 测试结果	tshref 测试结果
zzxiong@zzxiong-virtual-machine:~/xxxx/lab4/shlab-hand make test07 ./sdriver.pl -t trace07.txt -s ./tsh -a "-p" # # trace07.txt - Forward SIGINT only to foreground job. # tsh> ./myspin 4 & [1] (5571) ./myspin 4 & tsh> ./myspin 5 Job [2] (5573) terminated by signal 2 tsh> jobs [1] (5571) Running ./myspin 4 &	zzxiong@zzxiong-virtual-machine:~/xxxx/lab4/shlab-hand make rtest07 ./sdriver.pl -t trace07.txt -s ./tshref -a "-p" # # trace07.txt - Forward SIGINT only to foreground job. # tsh> ./myspin 4 & [1] (5580) ./myspin 4 & tsh> ./myspin 5 Job [2] (5582) terminated by signal 2 tsh> jobs [1] (5580) Running ./myspin 4 &

测试结论	相同

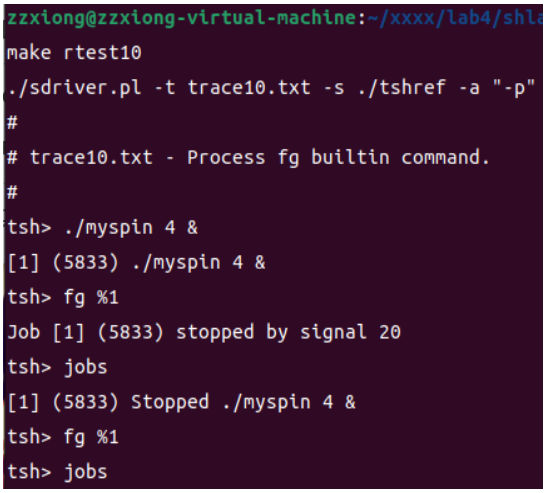
4.3.8 测试用例 trace08.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
	
测试结论	相同

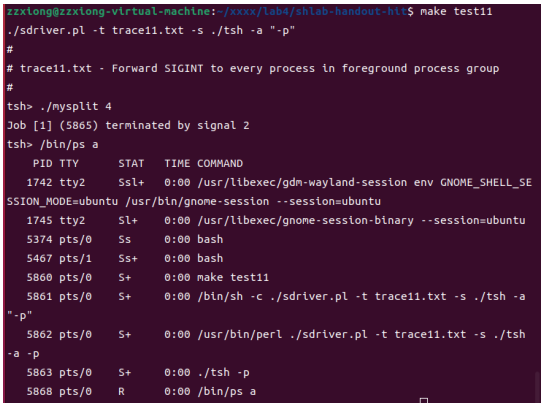
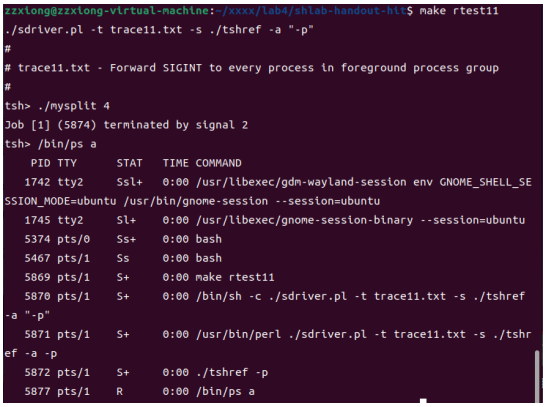
4.3.9 测试用例 trace09.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
	
测试结论	相同

4.3.10 测试用例 trace10.txt 的输出截图（1 分）

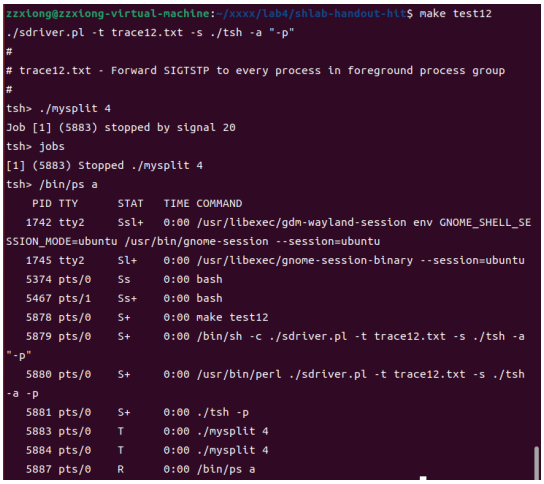
tsh 测试结果	tshref 测试结果
	
测试结论	相同

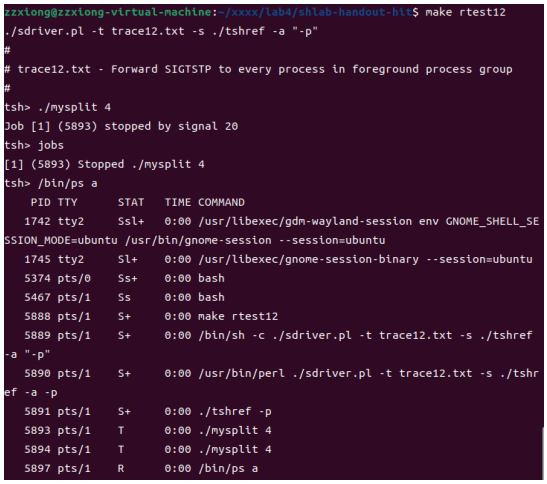
4.3.11 测试用例 trace11.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
	
测试结论	相同

4.3.12 测试用例 trace12.txt 的输出截图（1 分）

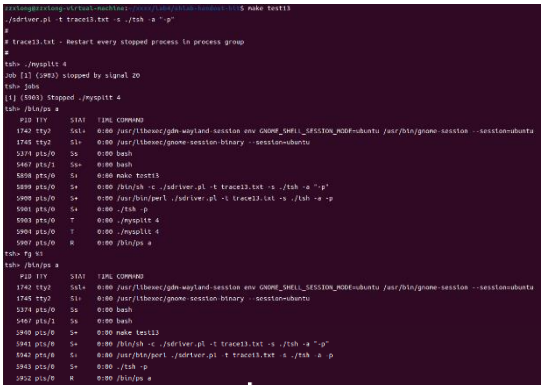
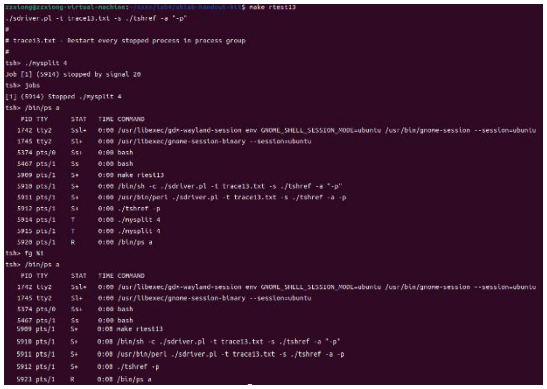
tsh 测试结果	tshref 测试结果





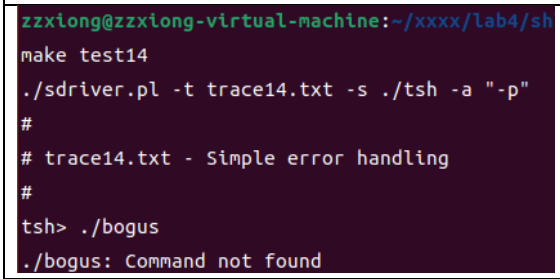
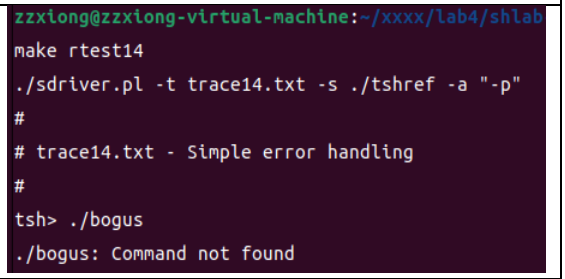
测试结论	相同

4.3.13 测试用例 trace13.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
	

测试结论	相同

4.3.14 测试用例 trace14.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果
	

```
tsh> ./myspin 4 &
[1] (5981) ./myspin 4 &
tsh> fg
fg command requires PID or %jobid argument
tsh> bg
bg command requires PID or %jobid argument
tsh> fg a
fg: argument must be a PID or %jobid
tsh> bg a
bg: argument must be a PID or %jobid
tsh> fg 9999999
(9999999): No such process
tsh> bg 9999999
(9999999): No such process
tsh> fg %2
%2: No such job
tsh> fg %1
Job [1] (5981) stopped by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (5981) ./myspin 4 &
tsh> jobs
[1] (5981) Running ./myspin 4 &
```

```
tsh> ./myspin 4 &
[1] (6000) ./myspin 4 &
tsh> fg
fg command requires PID or %jobid argument
tsh> bg
bg command requires PID or %jobid argument
tsh> fg a
fg: argument must be a PID or %jobid
tsh> bg a
bg: argument must be a PID or %jobid
tsh> fg 9999999
(9999999): No such process
tsh> bg 9999999
(9999999): No such process
tsh> fg %2
%2: No such job
tsh> fg %1
Job [1] (6000) stopped by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (6000) ./myspin 4 &
tsh> jobs
[1] (6000) Running ./myspin 4 &
```

测试结论	相同

4.3.15 测试用例 trace15.txt 的输出截图（1 分）

tsh 测试结果	tshref 测试结果

```
zzxiong@zzxiong-virtual-machine:~/xxxx/lab4/sh
make test15
./sdriver.pl -t trace15.txt -s ./tsh -a "-p"
#
# trace15.txt - Putting it all together
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 10
Job [1] (6062) terminated by signal 2
tsh> ./myspin 3 &
[1] (6064) ./myspin 3 &
tsh> ./myspin 4 &
[2] (6066) ./myspin 4 &
tsh> jobs
[1] (6064) Running ./myspin 3 &
[2] (6066) Running ./myspin 4 &
tsh> fg %1
```

```
zzxiong@zzxiong-virtual-machine:~/xxxx/lab4/shlab
make rtest15
./sdriver.pl -t trace15.txt -s ./tshref -a "-p"
#
# trace15.txt - Putting it all together
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 10
Job [1] (6042) terminated by signal 2
tsh> ./myspin 3 &
[1] (6044) ./myspin 3 &
tsh> ./myspin 4 &
[2] (6046) ./myspin 4 &
tsh> jobs
[1] (6044) Running ./myspin 3 &
[2] (6046) Running ./myspin 4 &
tsh> fg %1
```

Job [1] (6064) stopped by signal 20 tsh> jobs [1] (6064) Stopped ./myspin 3 & [2] (6066) Running ./myspin 4 & tsh> bg %3 %3: No such job tsh> bg %1 [1] (6064) ./myspin 3 & tsh> jobs [1] (6064) Running ./myspin 3 & [2] (6066) Running ./myspin 4 & tsh> fg %1 tsh> quit	Job [1] (6044) stopped by signal 20 tsh> jobs [1] (6044) Stopped ./myspin 3 & [2] (6046) Running ./myspin 4 & tsh> bg %3 %3: No such job tsh> bg %1 [1] (6044) ./myspin 3 & tsh> jobs [1] (6044) Running ./myspin 3 & [2] (6046) Running ./myspin 4 & tsh> fg %1 tsh> quit
测试结论	相同

4.4 自测试评分

根据节 4.3 的自测试结果，程序的测试评分为：满分 15 分 。

第 5 章 总结

5.1 请总结本次实验的收获

1. 熟悉了进程 fork、execve、waitpid/wait、kill 等相关函数，对进程的工作原理有了更深的理解。

2. 熟悉了信号机制，信号发送与信号接收、信号处理的方法，尝试运用 signal、sigprocmask 等常见信号系统函数。

3. 对 shell 的工作原理进一步熟悉，了解了用户、shell、内核之间的交互过程。

5.2 请给出对本次实验内容的建议

1. 希望能增加对程序源码部分的讲解，对 sigchld_handler 函数补写部分的任务描述及要求不是很清晰。

2. 希望增加对 tinyshell 如何进行调试等相关内容的讲解。

注：本章为酌情加分项。

参考文献

[1]RANDALE.BRYANT, DAVIDR.O'HALLARON. 深入理解计算机系统[M]. 机械工业出版社, 2011.

[2]C++信号处理 signal(SIGINT, signalHandler) https://blog.csdn.net/u013288190/article/details/119957441

[3]LINUX C 中 sigprocmask()函数用法 https://blog.csdn.net/ShaoLiang_Ge/article/details/57984123

[4] C 语言中 errno 与 perror 函数 https://blog.csdn.net/hou09tian/article/details/910393

[5] linux c 学习笔记----信号(sigaction,sigaddset,sigprocmask) https://blog.csdn.net/weixin_43667308/article/details/85250003