《计算机系统第四次实验》

班级：信安2302班

学号：202308060227

姓名：石云博

目录

[一. 实验描述 2](#_Toc198975006)

[二. 前期准备 4](#_Toc198975007)

[三. 实验过程 6](#_Toc198975008)

[四. 结果验证 23](#_Toc198975009)

[总结 38](#_Toc198975010)

# 实验描述

这个实验是大家在本课程第一次体验系统级编程，涉及过程，过程控制和信号的相关知识。

1.你需要干什么？

你需要构建一个简单的类Unix/Linux Shell。基于已经提供的“微Shell”框架tsh.c，完成部分函数和信号处理函数的编写工作。使用sdriver.pl可以评估你所完成的shell的相关功能。

2. 准备工作

使用命令tar xvf shelab-handout.tar 解压缩文件；

使用命令 make 去编译和链接一些测试例程；

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

你要实现的重要函数列出如下：

eval 主例程，用以分析和解释命令行（好消息：该函数原型在教材一第8章8.4节中可以找到！）；

builtin\_cmd 执行bg和fg内置命令；

waitfg 等待前台作业执行；

sigchld\_handler 响应处理SIGCHILD信号

sigint\_handler 响应处理SIGINT（ctrl-c）信号

sigtstp\_handler 相应处理SIGSTP（ctrl-z）信号

3.注意

每次修改了tsh.c文件，都需要make它以重新编译。在你的Linux终端中直接运行tsh（./tsh）就可以进入你所编写完成的tiny shell tsh>了。

4. 如何证明你完成了这个实验

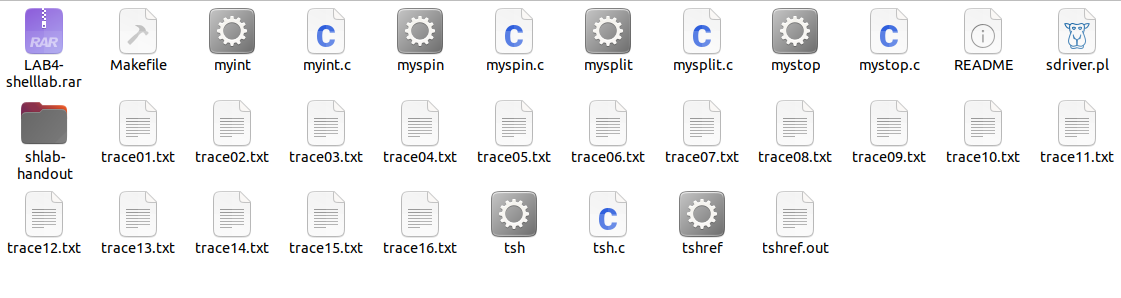
在你的Linux终端运行./tshref 这个已经实现的shell，将其输出结果与你所实现的./tsh 输出结果比较，是否一致。

相关比较命令行，参见shelab-overview文件。

5.请在实验报告体现你解决本实验目标的详细过程，仅仅贴图（图中只有代码）可能会导致“无个人工作，仅仅是复制粘贴”的极低分判定。

# 前期准备

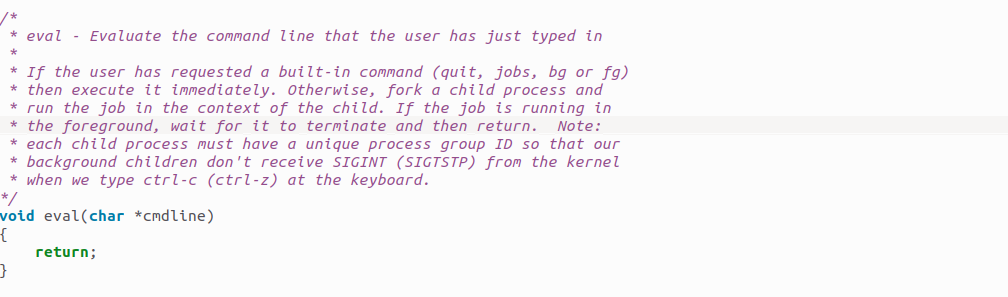
我们首先将文件进行解压，然后使用make命令来进行首次编译。这里的make的作用为自动判断重编译，当你在项目目录下输入 make，工具会扫描 Makefile，自动检测哪些源文件修改过，然后仅对这些文件进行编译／链接，无需全量重编译。同时可以简化命令，通过在 Makefile 中封装复杂的编译／链接命令，只需一次 make 调用，就能完成多步操作。除了源码编译，Makefile 也可定义其他任务（如单元测试、文档生成、打包发布），使项目管理更加一体化。

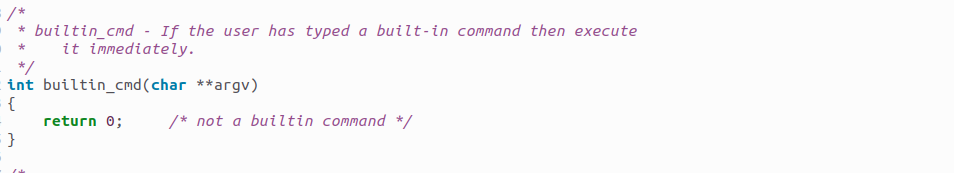


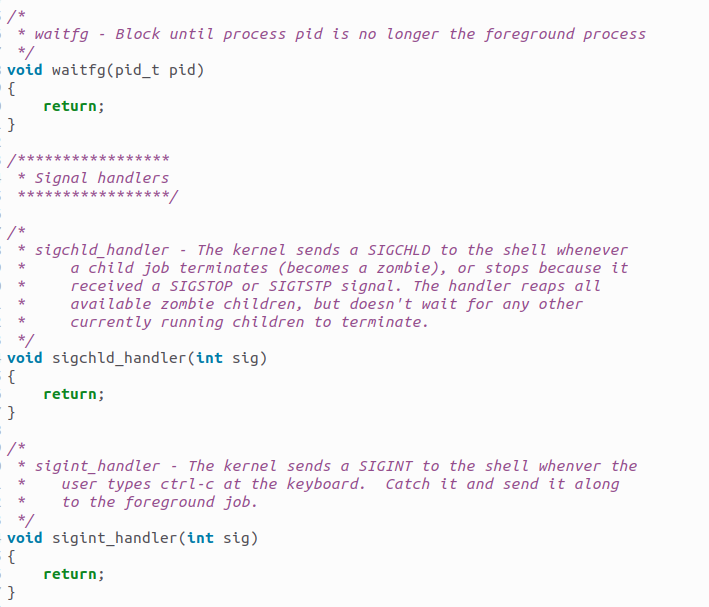
图表 1首次make后的结果

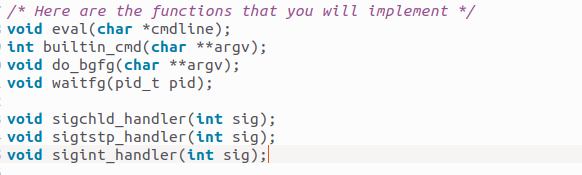
查看文件档中的这些文件，我们可以看到首先是 "my" 开头的四个 .c 文件——这些是测试 shell 时要用到的；还有 16 个 "trace" 开头的文件、一个 .pl 文件以及 Makefile，是测试 shell 用的。我们的代码要写在 [tsh.c](https://zhida.zhihu.com/search?content_id=182017372&content_type=Article&match_order=1&q=tsh.c&zhida_source=entity" \t "_blank) 里面。当然了，tsh.c 也不是一片空白，作者已经给我们定义好了用来描述作业的结构（job），用来描述作业状态的宏（UNDEF、FG、BG、ST），放作业的全局数组（jobs），以及一大堆乱七八糟的帮助函数。我们需要实现的函数有 7 个——三个是信号处理函数，余下四个是处理用户输入的函数。

我们可以看看tsh.c文件，里面作者已经把我们需要完成的函数表明了：



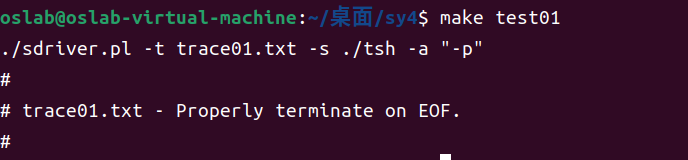






同时函数的声明也全部添加了，我们需要做的就是把他们的具体实现补全。

# 实验过程

我们可以先对于没有经过修改的程序运行我们的测试文件：  


可以看到回显了“# trace01.txt - Properly terminate on EOF.”该句的意思是在输入流（这里由 trace01.txt 模拟）到达文件末尾（EOF，End-Of-File）时，你实现的 Shell 应当“正确地终止”即退出，而不是继续等待新命令或进入死循环，也就是说sdriver.pl 脚本会依次读取 trace01.txt 中的每一行指令，并将其“喂”给我们自己做的 tsh。当文件读到末尾时，脚本期望你的 Tiny Shell 能检测到 EOF 并正常退出，否则测试会判定为未通过“Properly terminate on EOF”这一项。由此我们可以知道，我们需要做的就是对tsh程序进行实验所期望的修改，使得其能通过所有的测试。我们需要使得所有tsh程序运行的测试文件的输出结果和使用tshref的输出结果一致。

我们先从比较简单的两个信号处理函数开始。

首先是第一个函数**sigint\_handler**

该函数的作用是：处理由终端输入ctrl-c引起的异常，将前台作业结束（实现由操作系统内核完成，这里的作用是以内核特权向进程发送SIGINT信号，这个是kill函数做到的）。这两个函数的主要任务，是在收到 shell 传来的信号时，将这个信号“转发”给在 shell 中运行的进程。这个过程很好办——先用 fgpid（作者提供的帮助函数）获取前台进程（因为 SIGTSTP 和 SIGINT 信号是只发给前台进程的）的 pid，之后走 kill 调用，向这个子进程发对应的信号。其中还是有一些需要注意的点的。fgpid 是不是访问了全局数组呢？那么我们是否要在它执行的时候暂时阻塞所有信号呢？为了保证信号处理函数之外依赖于 errno 的代码正常运行，我们是不是应该暂存 errno 呢？  
其实还有一个比较隐蔽的点——不过作者已经在 writeup 中提示给我们了——就是 kill 的第一个参数应该填 "-pid" （“负的 pid”），而不是 "pid"，从而保证”转发“的信号能够被传递给整个子进程组。为什么是“子进程组**”**呢？如果 shell fork 出来的子进程，没有再 fork 它自己的子进程的话，填 "pid" 没有任何问题；但是，如果它 fork 了的话（shell 就有孙进程了），这时候子进程和孙进程的 pid 是不一样的。填正的 pid，只能保证子进程能被结束；但是孙进程就没那么幸运了——它会“丧父”（变成孤儿进程），直到操作系统将他回收。所以为了避免这些麻烦事，填成负的 pid 就好了。所以我们可以首先获取前台进程的pid，若pid合法，调用kill函数发送SIGINT信号给前台进程。基于这个思路，我们可以写出该函数的具体实现：

**void** **sigint\_handler**(**int** sig)

{

int olderrno = errno;

// 暂存 errno

sigset\_t mask\_all, prev\_all;

sigfillset(&mask\_all);

sigprocmask(SIG\_BLOCK, &mask\_all, &prev\_all);

// 阻塞所有信号

pid\_t pid = fgpid(jobs);

if (pid == 0)

{

// 没有就收拾收拾返回

errno = olderrno;

sigprocmask(SIG\_SETMASK, &prev\_all, NULL);

return;

}

kill(-pid, SIGINT);

sigprocmask(SIG\_SETMASK, &prev\_all, NULL);

errno = olderrno;

return;

}

该段代码中，我们为了不打断 shell 正在进行的系统调用或库函数的错误返回,暂存 errno，接着阻塞所有信号,所以先用 sigprocmask 屏蔽所有信号，并保存原来的屏蔽字到 prev\_all，然后fgpid(jobs) 遍历全局 jobs[]，返回 state==FG 的那项的 pid（即进程组 ID），注意的是，kill函数中使用负数，用负号 -pid，表示把信号送到以 pid 为进程组 ID 的组中，接着复信号屏蔽字，允许信号继续递送，最后恢复 errno，不改变被中断的上下文的错误状态。

接着是**sigtstp\_handler**

这个函数和上面那个差不多，我们可以直接写出类似的实现出来：

**void** **sigtstp\_handler**(**int** sig)

{

int olderrno = errno;

sigset\_t mask\_all, prev\_all;

sigfillset(&mask\_all);

sigprocmask(SIG\_BLOCK, &mask\_all, &prev\_all);

pid\_t pid = fgpid(jobs);

if (pid == 0)

{

errno = olderrno;

sigprocmask(SIG\_SETMASK, &prev\_all, NULL);

return;

}

kill(-pid, SIGTSTP);

sigprocmask(SIG\_SETMASK, &prev\_all, NULL);

errno = olderrno;

}

上面我们实现了处理 SIGINT 和 SIGTSTP 信号的函数。那么，当子进程收到信号之后停止了，又会发生什么呢？我们知道，这时候内核会发 SIGCHLD 信号给 shell，让它处理子进程的停止（或者是“终止”）事件。所以接下来我们要写的 sigchld\_handler 就是做这件事的。

**void** **sigchld\_handler**(**int** sig)

{

int olderrno = errno;

// 存储 errno ——编写安全的信号处理函数要求之一

int status;

pid\_t pid;

sigset\_t mask\_all, prev\_all;

sigfillset(&mask\_all);

while ((pid = waitpid(-1, &status, WUNTRACED | WNOHANG)) > 0)

{

sigprocmask(SIG\_BLOCK, &mask\_all, &prev\_all);

// 阻塞所有信号——编写安全的信号处理函数要求之二

if (WIFSIGNALED(status))

{

printf("Job [%d] (%d) terminated by signal %d\n",

pid2jid(pid), pid, WTERMSIG(status));

// 按照 writeup 中的要求打印一条消息

// csapp 里面的要求是，不在信号处理函数里面调用异步信号不安全的函数（比如 sprintf 和 printf）。但是，查一下 tshref 的反汇编，你会发现官方的实现在这里调用了 sprintf。可见，官方也没有认真遵守自己在书中提到的要求。

deletejob(jobs, pid);

}

else if (WIFSTOPPED(status))

// 子进程被 Ctrl + Z 阻塞了

{

printf("Job [%d] (%d) stopped by signal %d\n",

pid2jid(pid), pid, WSTOPSIG(status));

struct job\_t\* job = getjobpid(jobs, pid);

job->state = ST;

// 改下子进程的状态就好，不要把它从列表中删除

}

else // 子进程寿终正寝的情况

deletejob(jobs, pid);

// 直接删除就好

}

sigprocmask(SIG\_SETMASK, &prev\_all, NULL);

errno = olderrno;

}

这个函数我们可以先照着图 8.40 里的示例代码先抄上，之后再做一些相关的改动。第一处修改就是把 waitpid 的第三个参数从 0 改成了 WUNTRACED | WNOHANG因为第三个参数为 0 的含义是，当还有子进程存在时，就挂起父进程，直到子进程终止（停止）。如果这个 shell 没有运行后台进程的话，那么其实还好；但是一旦有一个前台进程和一个后台进程同时运行的话，当前台进程终止（停止），shell 就会开始执行上面那个 while 循环，处理前台进程；处理完之后，因为后台进程还在执行，所以 shell 会被 waitpid 卡住，直到后台进程结束，才会继续向下执行。而改成 WUNTRACED | WNOHANG 就不会出现这个问题了——此时的 waitpid，只是去看一眼有没有已经终止（停止）的后台进程，而不会等待进程执行完才返回。  
其他改动还有一些小的改动，看上面的注释就可以了。

写完了和子进程打交道的信号处理函数，我们把目光收回到 shell 本身上来。也就是eval 和 waitfg函数。

首先是第一个函数，eval()，我们先来看看该函数的说明，主例程，用以分析和解释命令行，解析和解释命令行。

eval首先解析命令行，如果用户请求了一个内置命令（quit、jobs、bg或fg），则立即执行。

否则，派生一个子进程并在该子进程的上下文中运行作业。

如果作业正在前台运行，请等待它终止，然后返回。

注意：每个子进程都必须有一个唯一的进程组ID，这样当我们在键盘上键入ctrl-c（ctrl-z）时，我们的后台子进程就不会从内核接收SIGINT（SIGTSTP）。

这里有个好消息是，关于 eval 函数呢，其实书上的图 8.24 里已经给了我们一个半成品，所以我们可以直接在原型基础上进行修改以符合我们的实验要求，修改之后的eval函数实现如下：

**void** **eval**(**char\*** cmdline)

{

char\* argv[MAXARGS];

char buf[MAXLINE];

int bg;

pid\_t pid;

sigset\_t mask\_all, prev\_all;

strcpy(buf, cmdline);

bg = parseline(cmdline, argv);

if (argv[0] == NULL)

return;

if (builtin\_cmd(argv))

// 修改了判断条件——避免 if 语句嵌套

return;

sigfillset(&mask\_all);

sigprocmask(SIG\_BLOCK, &mask\_all, &prev\_all);

// 加上了阻塞信号的代码

if ((pid = fork()) == 0) // 子进程

{

sigprocmask(SIG\_SETMASK, &prev\_all, NULL);

setpgid(0, 0);

// 对 setpgid 的调用；下面会介绍它

if (execve(argv[0], argv, environ) < 0)

{

printf("%s: Command not found.\n", argv[0]);

exit(0);

}

}

// 子进程不会执行到这里

if (!bg)

{

addjob(jobs, pid, FG, cmdline);

// 添上去的一行——向全局作业表中添加作业

sigprocmask(SIG\_SETMASK, &prev\_all, NULL);

waitfg(pid);

// 等待前台进程结束；waitfg 我们一会儿就写

}

else // 后台进程的情况

{

addjob(jobs, pid, BG, cmdline);

sigprocmask(SIG\_SETMASK, &prev\_all, NULL);

// 这里就不等子进程了——收拾收拾就返回

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

// 输出相关信息——就像 bash 那样

}

}

我们可以看到该函数说明了它在一个带作业控制（job control）的简易 Shell 中如何安全地创建和管理前台/后台作业。我们来仔细看看其逻辑：

首先，在准备阶段里，该函数进行了解析命令行与内建命令的判断，“strcpy(buf, cmdline);” 把 cmdline 复制到本地缓冲区 buf，以免后面 parseline 修改原始字符串导致打印或存储出错。“bg = parseline(cmdline, argv);”中parseline 将命令行按空格／引号拆成 argv[]，并返回 bg（0 = 前台，1 = 后台）。这里如果用户只敲回车（argv[0] == NULL），直接返回，不做任何处理。同时，如果是 cd、jobs、fg、bg 等内建命令，builtin\_cmd 在这里执行并返回 true，Shell 直接返回，不走后续 fork/exec 分支。

下面是第二个阶段，阻塞所有可中断信号，避免竞态，“sigprocmask(SIG\_BLOCK, &mask\_all, &prev\_all);”会在执行 fork() 之前，先把像 SIGCHLD、SIGINT、SIGTSTP 这些信号全部阻塞。这里这么实现的目的是通过在关键区（fork 及 addjob）将信号暂时屏蔽，保证对全局作业表的更新是“原子”的、不会被中断的。

再往下，函数创建子进程并执行程序，igprocmask(SIG\_SETMASK, &prev\_all, NULL)：恢复到 fork 前的信号屏蔽字，允许接收 SIGINT、SIGTSTP 等信号，否则子进程会继承父进程的全部阻塞，导致自己无法被控制台信号中断。“setpgid(0, 0)” 将子进程放入以自己 PID 为组 ID（PGID）的新进程组。Execve函数则是，用 argv[0] 程序替换子进程映像。若失败，打印“Command not found”并退出子进程。

再往下，我们看到了父进程：登记作业并恢复信号。if (!bg)进入前台作业分支，addjob(jobs, pid, FG, cmdline);：把子进程的 PID、PGID、命令行、作业状态（FG）记录到全局作业表中。后台作业中，addjob(jobs, pid, BG, cmdline);：同样登记，但状态是 BG（后台）。

就这样，我们实现了该实验的第一个函数。

接下来我们实现waitfg函数，函数的作用是实现阻塞等待前台程序运行结束。我们可以通过pid获取该pid对应的job本体，新设立一个wait信号集并清空该信号集，若前台进程未结束，使用空信号集替换信号掩码，即信号掩码为空，此时任何信号都会唤醒该进程。

基于上述思路，我们可以写出该函数的具体实现：

**void** **waitfg**(pid\_t pid)

{

sigset\_t mask\_all, prev\_all;

sigfillset(&mask\_all);

while (1)

{

// 访问全局变量记得阻塞有关的信号

sigprocmask(SIG\_BLOCK, &mask\_all, &prev\_all);

struct job\_t\* foreground\_job = getjobpid(jobs, pid);

sigprocmask(SIG\_SETMASK, &prev\_all, NULL);

if (!foreground\_job || foreground\_job->state != FG)

// 当找不到前台进程，或者前台进程已经挂起，就退出循环

break;

sleep(1);

}

}

接下来，builtin\_cmd，这个函数相比于第一个函数来说比较简单，其功能大概就是执行bg和fg内置命令；

对于该函数，我们的实现思路为：若命令为quit，调用exit实现退出；若命令为bg或fg，调用do\_fgbg函数执行相关操作；若命令为jobs，调用listjobs函数打印后台作业；

若都不满足，则不是内置命令，以0返回eval，eval会做相应的处理。

对于该函数，书上也提供了参考（图 8.24）。所以我们能写出函数如下：

**int** **builtin\_cmd**(**char\*\*** argv)

{

**if** (strcmp(argv[0], "quit") **==** 0)

exit(0);

**else** **if** (strcmp(argv[0], "jobs") **==** 0)

{

listjobs(jobs);

*// 借用作者给我们写好的 listjobs*

**return** 1;

}

**else** **if** (strcmp(argv[0], "fg") **==** 0 **||** strcmp(argv[0], "bg") **==** 0)

{

do\_bgfg(argv);

*// 这个函数是专门处理 bg 和 fg 命令的*

**return** 1;

}

**else** **if** (strcmp(argv[0], "&") **==** 0)

*// 不处理单独的 '&'*

**return** 1;

**return** 0; */\* not a builtin command \*/*

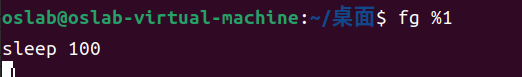
}

再下一个函数，void do\_bgfg(char \*\*argv); 函数作用，完成内置指令bg或fg的切换进程前后台状态的操作。

在开写之前，我们可以先亲身体验一下 bash 中的 bg 和 fg 命令。

首先，我们打开（操作系统的）终端输入 "sleep 100 &"—— 创建一个会在 100 秒之后关闭的进程——当然它是在后台运行的。 现在，终端应该显示 "[1]" 后跟进程的 pid. 我们输入 "fg %1", 把这个进程推到前台。





回车之后你应该就能看到效果了。终端输出 "sleep 100", 之后就不再响应我们的输入了——因为前台在 sleep…… 这时按 Ctrl + Z，把前台进程停掉，再输入 "bg %1", 就可以把 sleep 拉回后台执行；或者输入 "fg %1"，就能再次在前台启动进程。

现在我们已经对 fg 和 bg 有了感性的了解。那么具体应该怎么实现这两个内建命令呢？我们把关注的目光放到最后那个 if 块中：

对于 bg 命令，我们只是向目标进程发送 SIGCONT 信号，让它继续执行；对于 fg 命令呢，我们先判断目标进程是不是已经暂停了（如果是，就先启动它）；之后调用 waitfg 等待进程结束。注意，这里的 kill 函数的第一个参数也是要填负值的。根据上面的思路，我们可以试着写出该函数的具体实现：

void do\_bgfg(char \*\*argv)

{

if (argv[1] == NULL)

// 没有额外参数的情况

{

printf("%s command requires PID or %%jobid argument\n", argv[0]);

return;

}

sigset\_t mask\_all, prev\_all;

sigfillset(&mask\_all);

sigprocmask(SIG\_BLOCK, &mask\_all, &prev\_all);

// 阻塞信号——下面要访问全局数组了

struct job\_t\* job;

int jid = 0, pid = 0;

// 获取 jid 或是 pid

if (argv[1][0] == '%')

jid = atoi(argv[1] + 1);

else

pid = atoi(argv[1]);

// 乱填 jid 或是 pid 的情况之一

if ((jid == 0 && argv[1][0] == '%') || (pid == 0 && argv[1][0] != '%'))

{

printf("%s: argument must be a PID or %%jobid\n", argv[0]);

sigprocmask(SIG\_SETMASK, &prev\_all, NULL);

return;

}

if (jid == 0)

jid = pid2jid(pid);

job = getjobjid(jobs, jid);

// 乱填 jid 或是 pid 的情况之二

if (job == NULL)

{

if (argv[1][0] != '%')

printf("(%s): ", argv[1]);

else

printf("%s: ", argv[1]);

printf("No such %s\n", argv[1][0] == '%' ? "job" : "process");

sigprocmask(SIG\_SETMASK, &prev\_all, NULL);

return;

}

// 最重要的部分就是下面两块了

if (strcmp(argv[0], "bg") == 0)

{

job->state = BG;

printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);

kill(-(job->pid), SIGCONT);

sigprocmask(SIG\_SETMASK, &prev\_all, NULL);

}

else

{

if (job->state == ST)

kill(-(job->pid), SIGCONT);

job->state = FG;

sigprocmask(SIG\_SETMASK, &prev\_all, NULL);

waitfg(job->pid);

}

return;

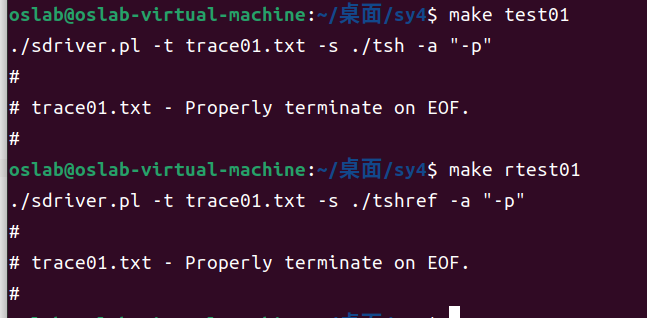
}

实现 bg 和 fg 命令，把指定作业从停止（ST）态恢复执行，要么放后台（BG），要么切到前台（FG）并等待它结束或再次停止。

关键点在于：首先，对于信号屏蔽我们需要在访问/修改全局 jobs 时要阻塞所有信号，防止竞态。其次，参数解析与合法性检查，我们要区分 %jid 和 pid，检查格式和作业是否存在。然后，状态转换，需要正确设置 job->state，并用 kill(-pid, SIGCONT) 唤醒进程组。最后，针对前台与后台差异，前台需要 waitfg 等待，后台只需打印信息然后立即返回。

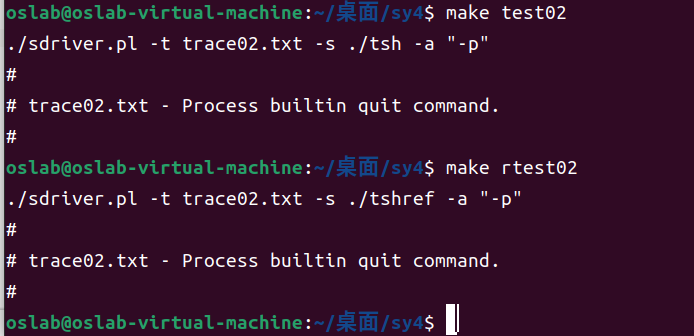
# 结果验证

当我们实现了所有的函数之后，我们就可以运行测试文件了，看我们自己实现的tsh和作者给我们的shell实现有什么区别：  
test01：



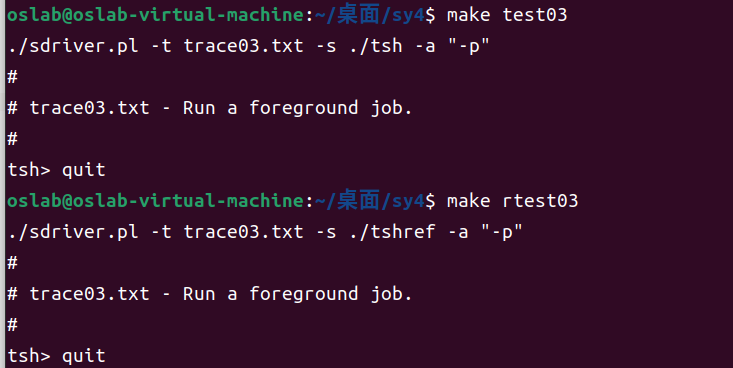
没有区别，测试通过。

Test02：



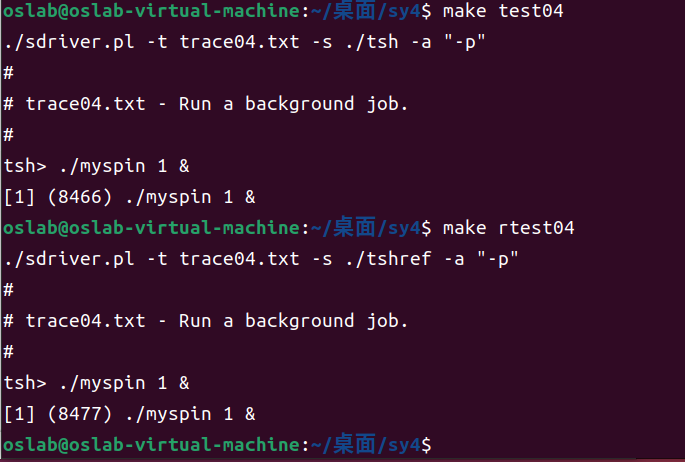
没有区别，测试通过。

Test03：



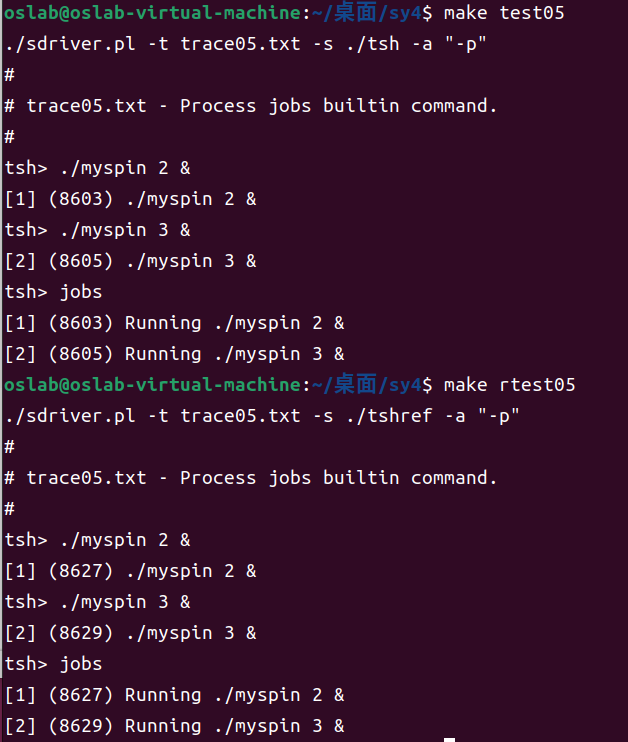
通过。

Test04：



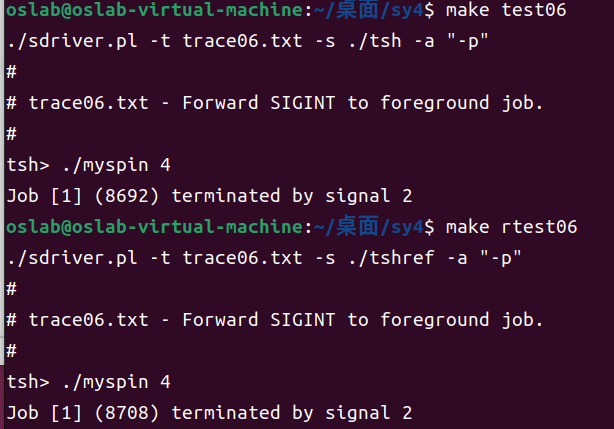
只有运行信息数字有一点区别，测试通过。

Test05：



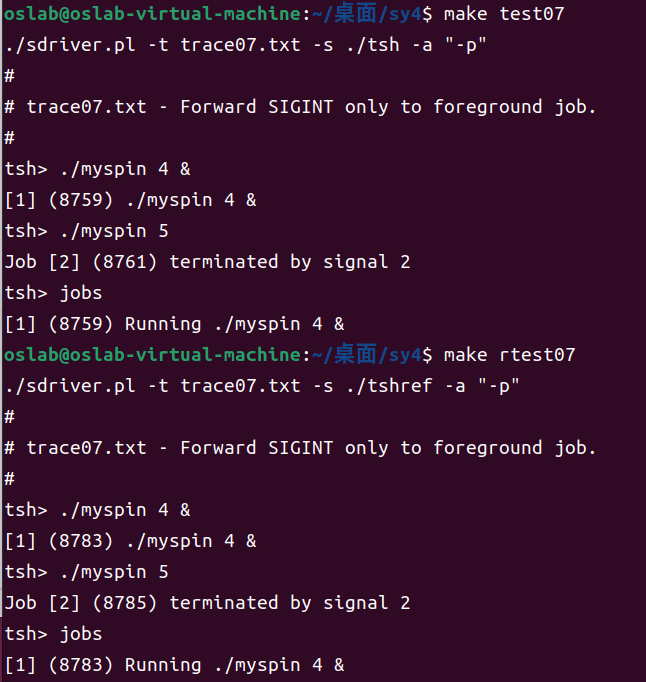
通过。

Test06：



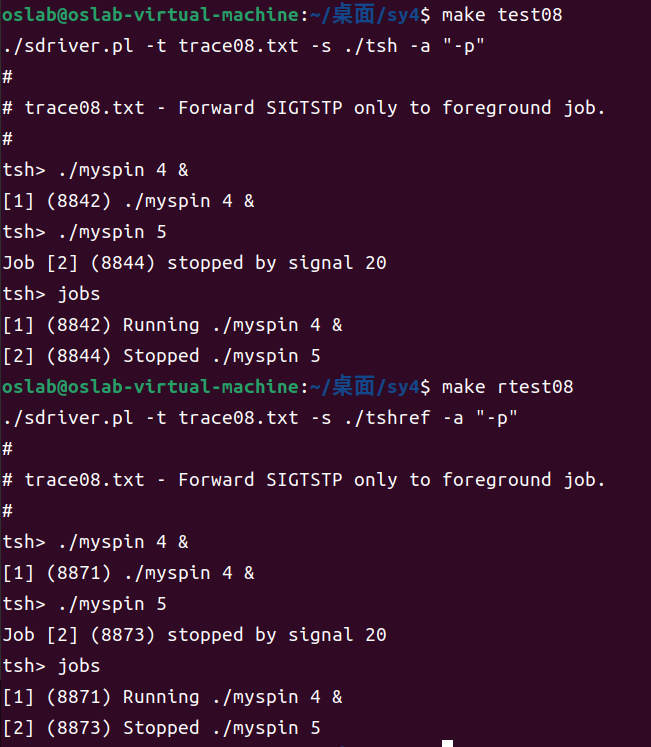
通过。

Test07：



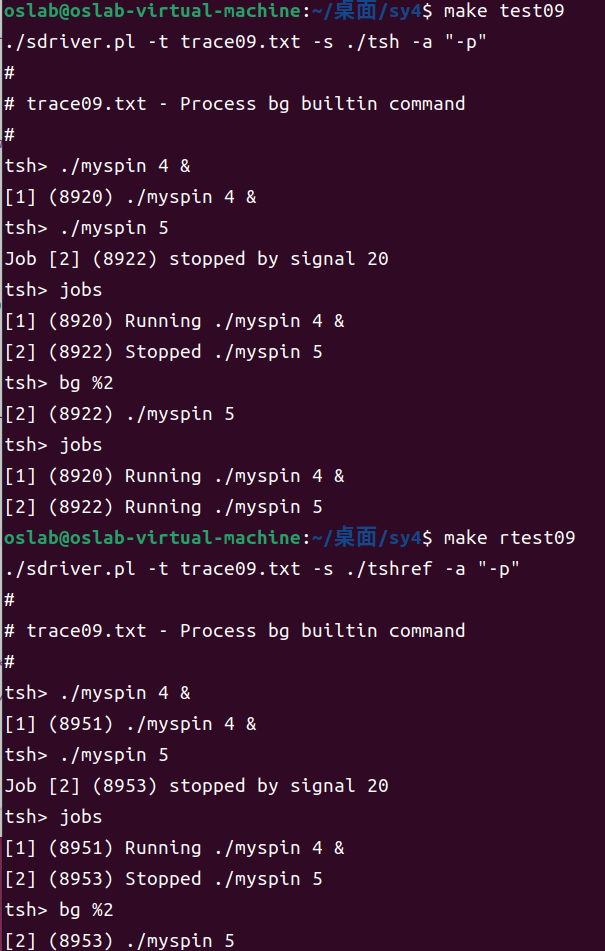
通过。

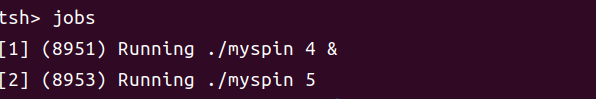
Test08：



通过。

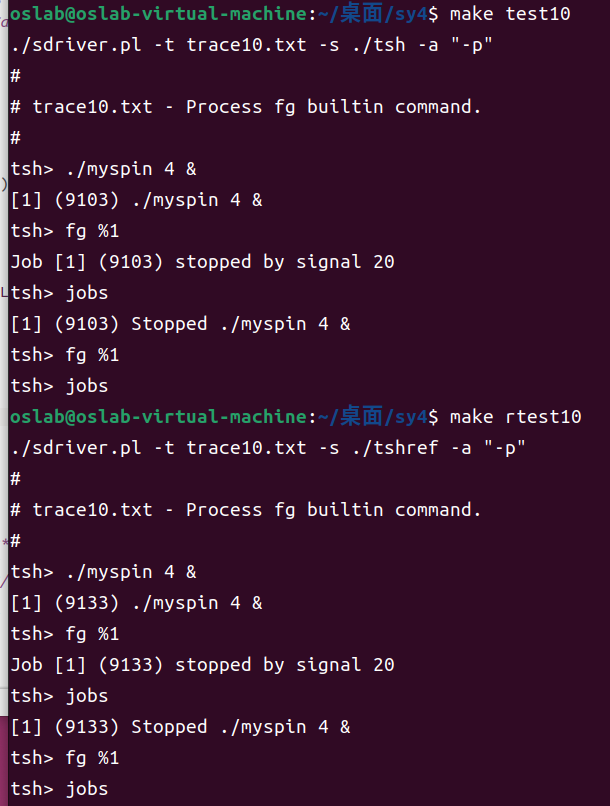
Test09：



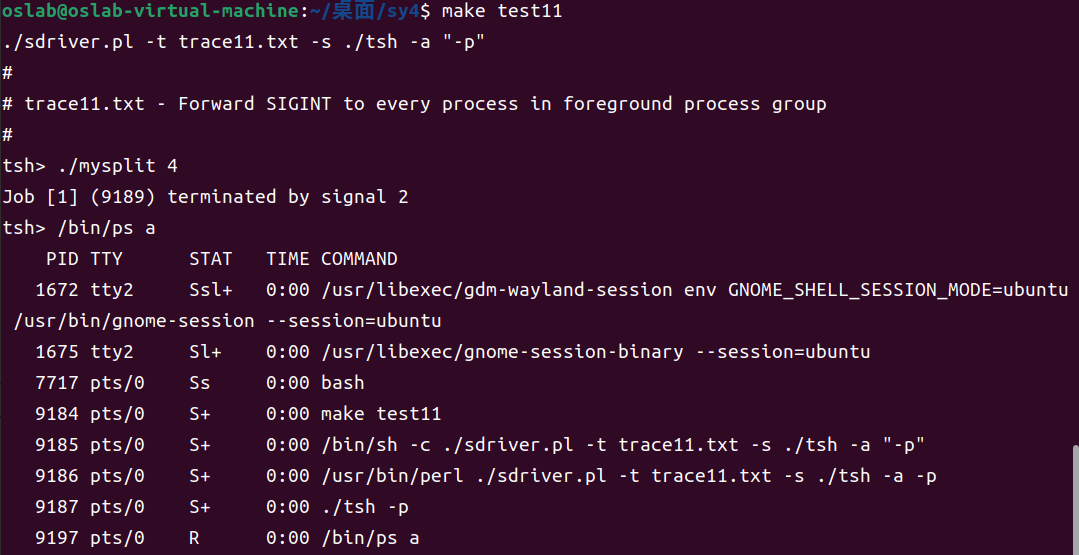


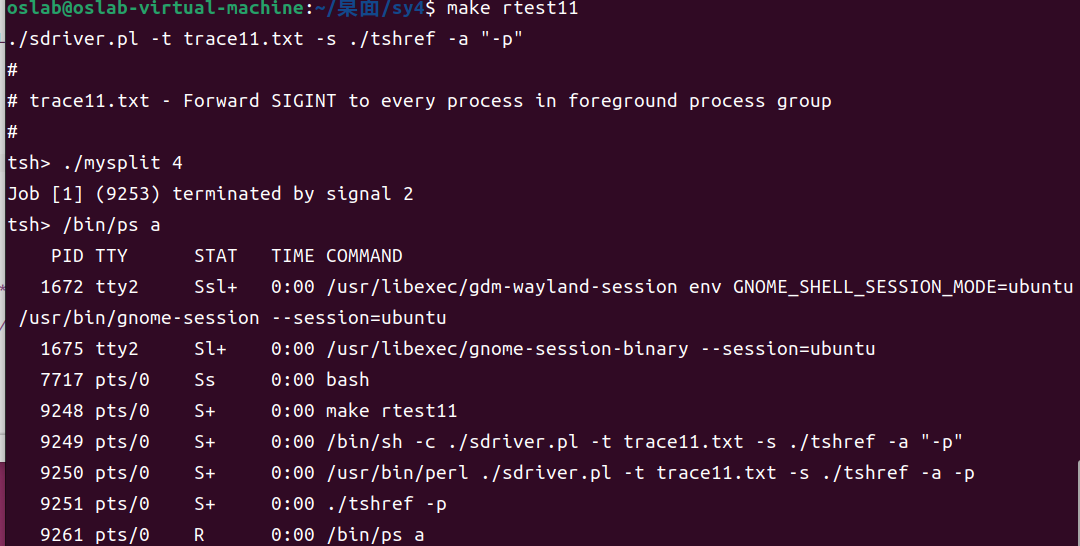
测试通过。

Test10：



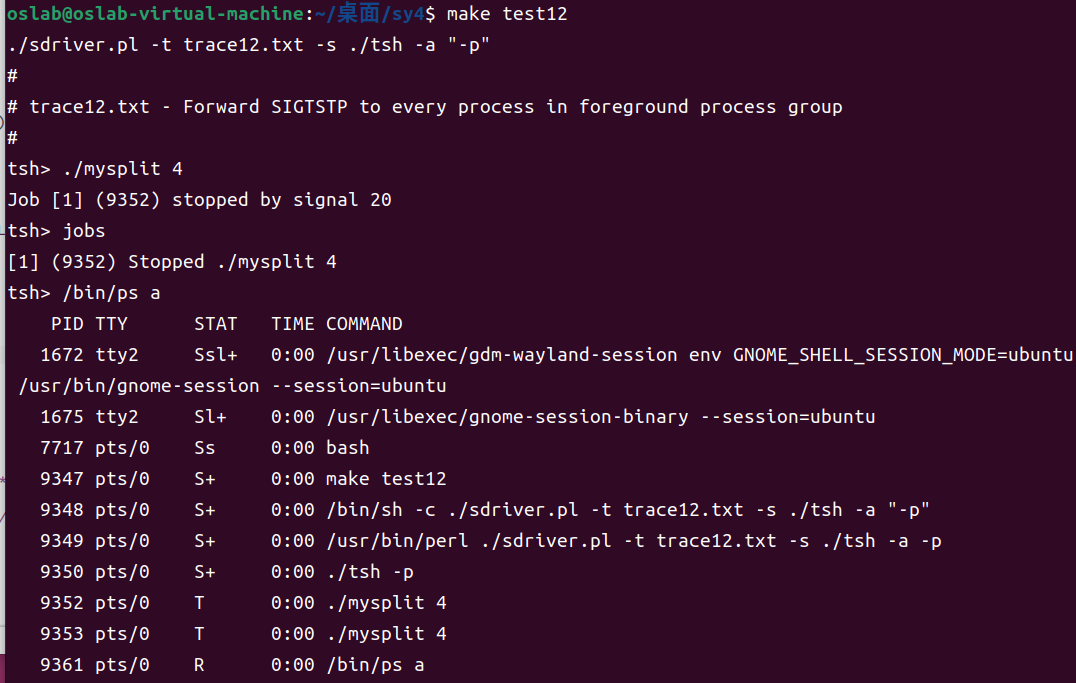
Test11：

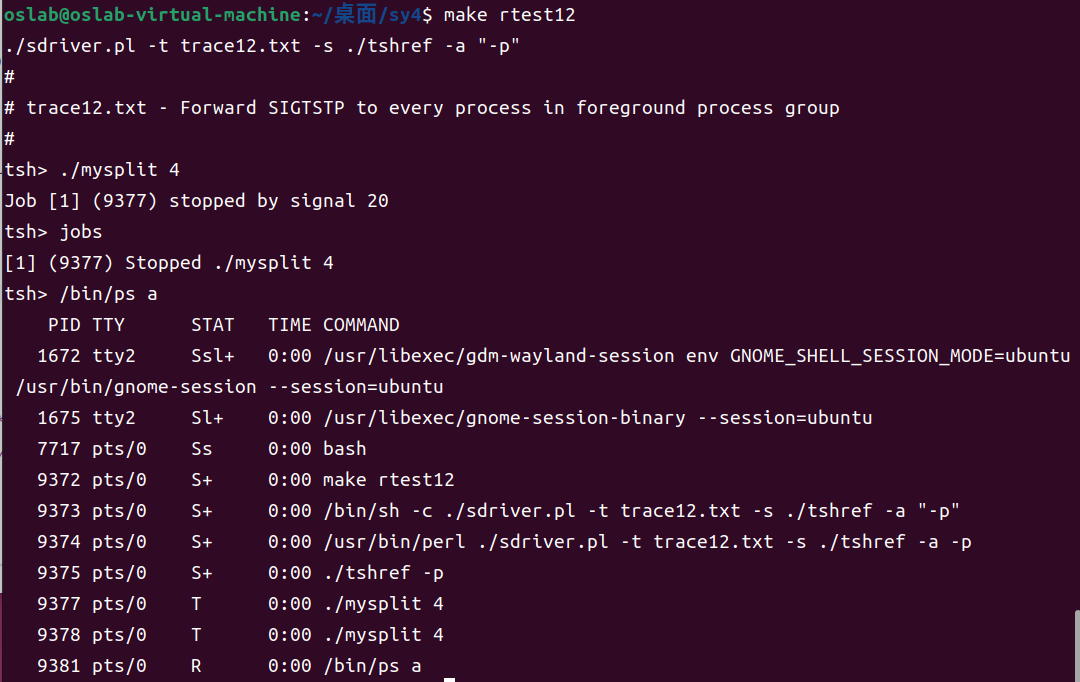




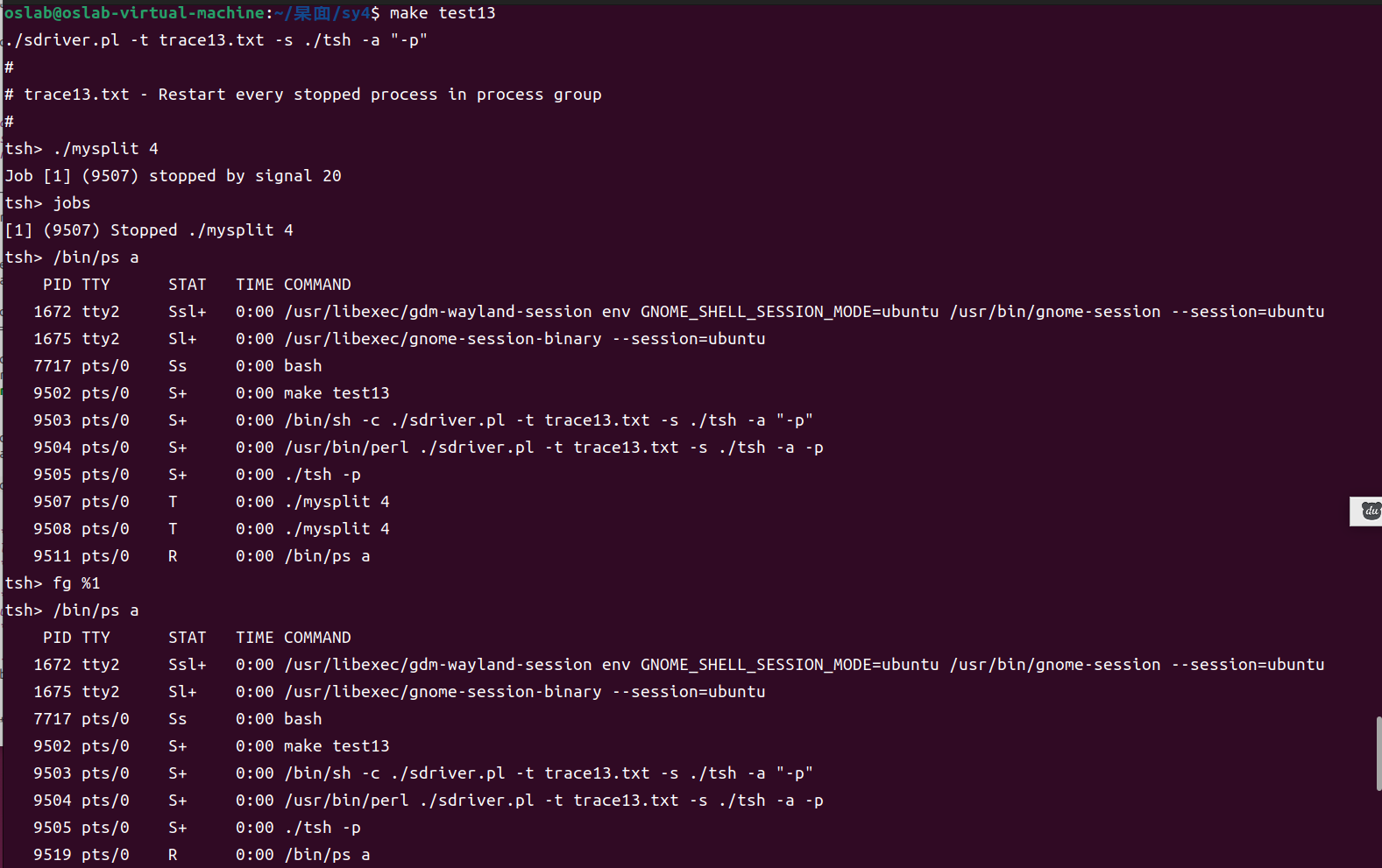
通过。

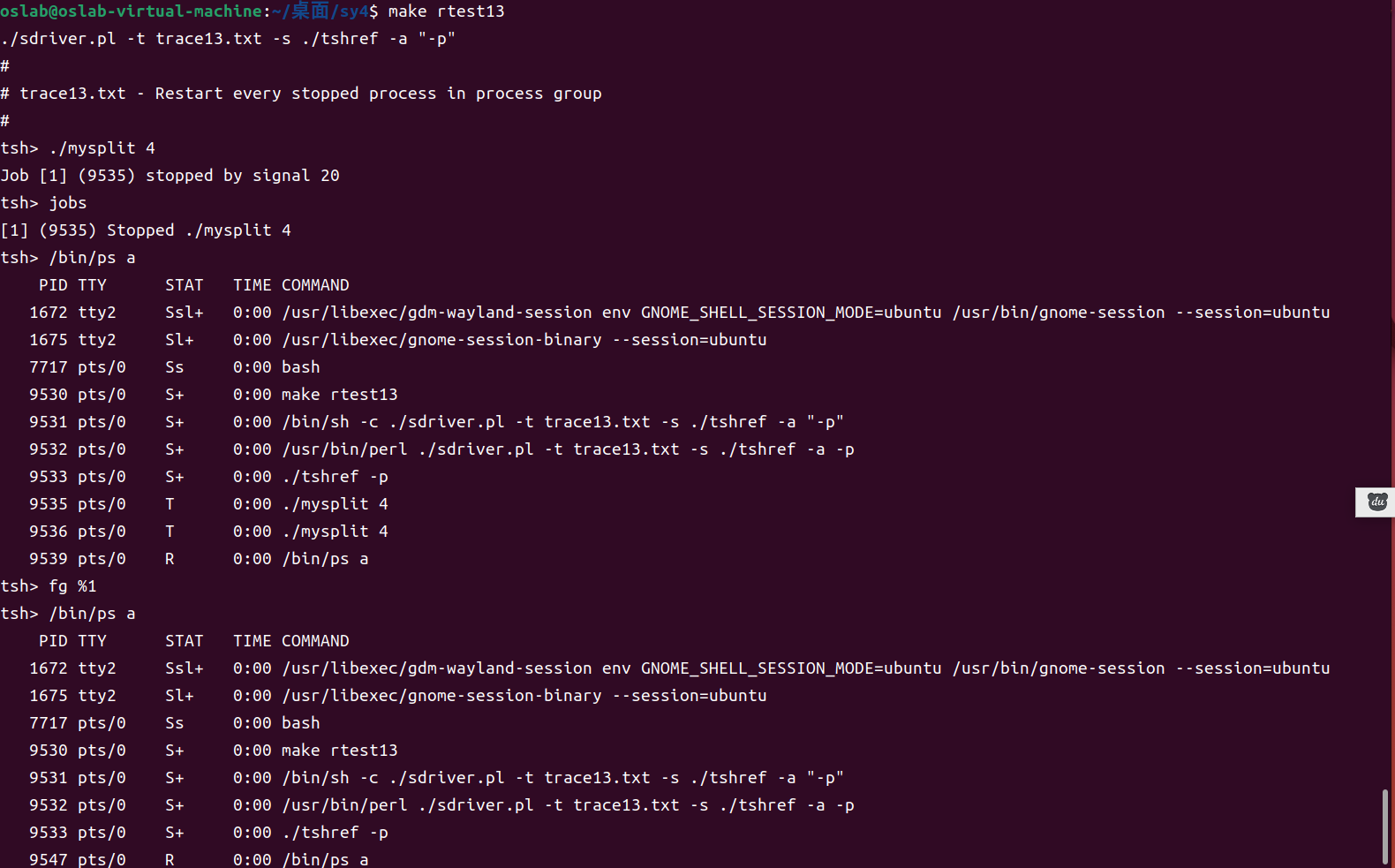
Test12：





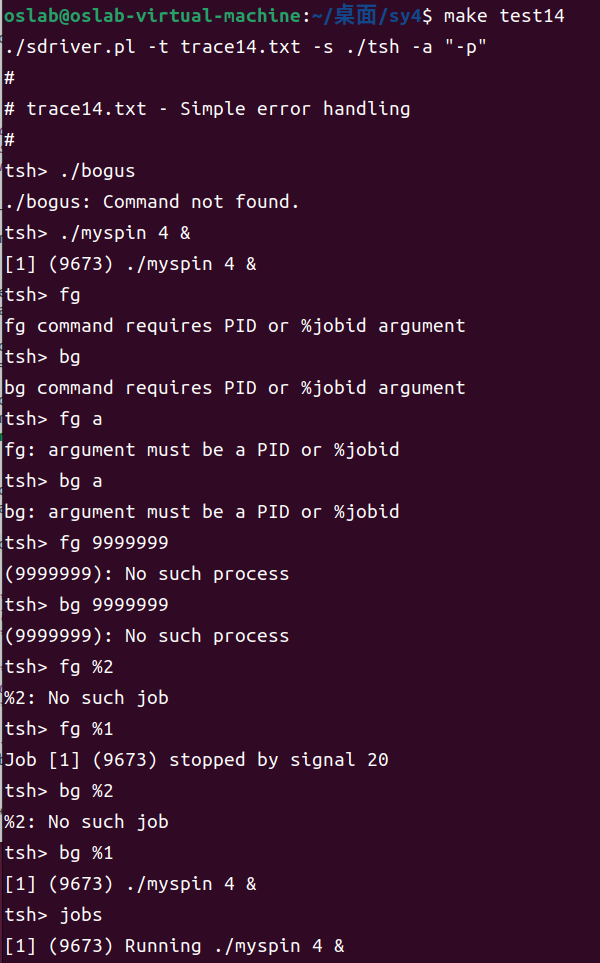
Test13：

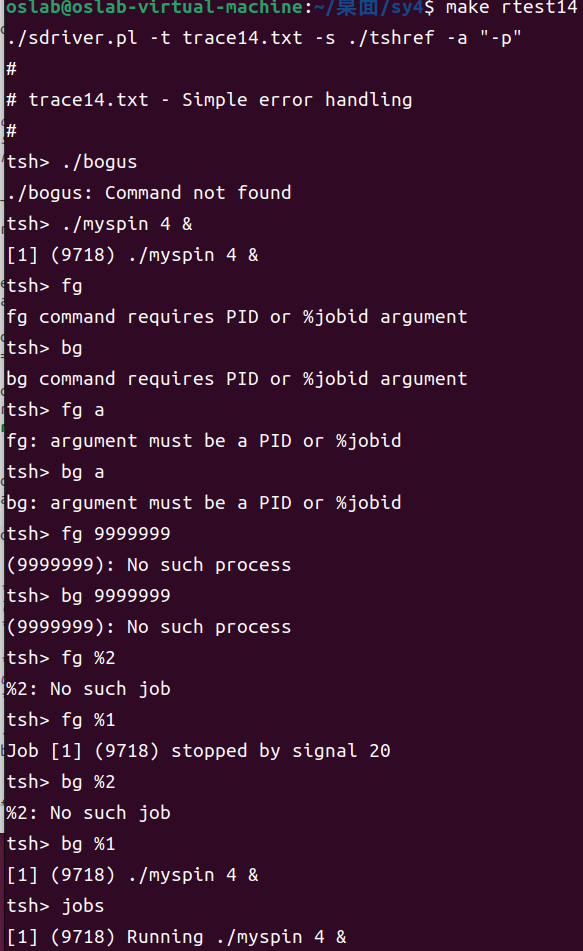




测试通过

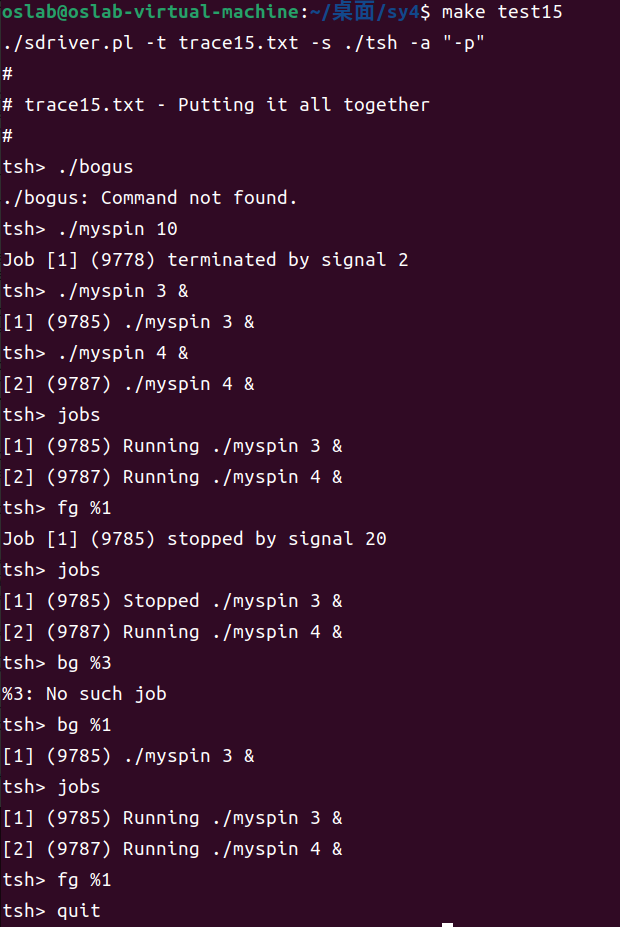
Test14：

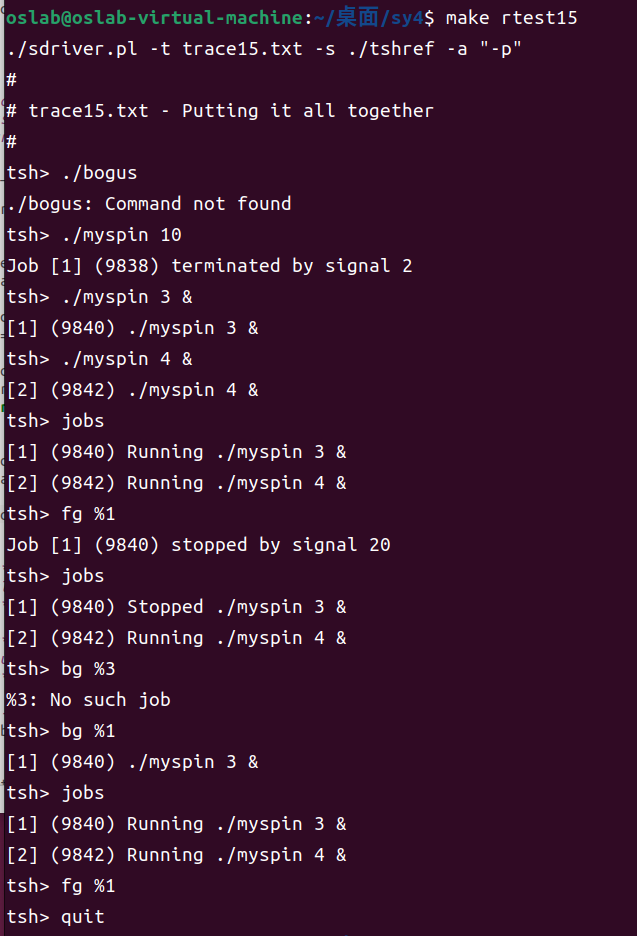


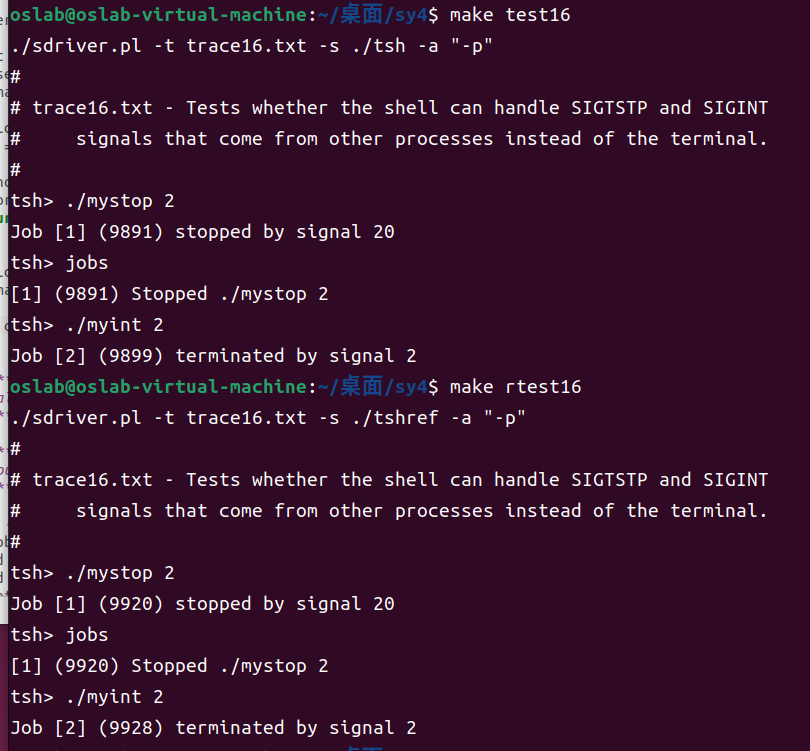


测试通过。

Test15：





Test16：  


测试通过。

至此，我们通过了所有的测试文件，实验成功。

# 总结

1. https://blog.csdn.net/u011563903/article/details/90116368. 详解一次完整的数据包传输过程 -- 层层递进. shufanhao.top 2023.11.25.

[2] https://blog.csdn.net/a595364628/article/details/64123565. 网络中的三种通讯模式：单播、广播、组播(多播). a595364628.2024.11.14.

[3] https://cloud.tencent.com/developer/article/2362927 网络协议性能优化：从HTTP到TCP、DNS、SSL/TLS的全面探究. bug菌.2021.01.24.

[4] https://developer.aliyun.com/article/1393349. 网络协议之性能优化与性能评估.Public IP List.2023.8.15.

[5] https://www.tance.cc/article/know/432. 网络协议栈优化：提升高并发环境下的数据传输效率. tance.cc.2022.3.24

[6] https://blog.csdn.net/weixin\_42839065/article/details/131375292. 干吃咖啡豆. 【TCP/IP】广播 - 定义、原理及编程实现. 2024.9.16.

[7] https://blog.csdn.net/xiaolei251990/article/details/83177165. 用udp实现广播通信. 【鹰击司马】. 2022.8.15.