操作系统大作业实验报告

计算机实验班 81 王烨 2186113483

实验目的：

目的为更加熟悉过程控制和信号通知的概念。编写一个简单的 unix shell 程序来支持 job 的控制。

Shell 简介：

Shell 是一个交互式的命令行解释器，它代表用户运行程序。Shell 反复打印提示，在 stdin 上等待命令行，然后按照命令行内容的指示执行一些操作。

命令行是由空格分隔的 ASCII 文本单词序列。命令行中的第一个按此是内置命令的名称或者可执行文件的路径名。其余的单词是命令行参数。如果第一个单词是内置命令，则 shell 将在当前进程中立即执行该命令。否则，假定该单词为可执行程序的路径名。在这种情况下，shell 将派生出一个子进程，然后在该子进程的上下文中加载并运行该程序（利用 execve 函数），shell 为解释单个命令行而创建的子进程统称为序列 job（job 由 shell 管理）。通常一个作业可以包含多个通过 unix 管道连接的子进程。（unix 管道的实现超过课本知识讲述，这里我们实现的每个作业只包含一个进程）。

如果命令行以“&”结尾，则表示作业将在后台运行（称为后台作业），此时 shell 在打印提示并等待下一个命令行之前不会等待作业终止。否则 shell 将在前台运行，此时 shell 将会等待该作业结束并回收后展示下一个命令行。因此在任何时间点，前台只能至多运行一个作业。但是可以在后台运行任意数量的作业。

Unix shell 支持 *job-control*，此时永续用户在前台和后台之间来回移动作业，同时更改作业中进程的状态（运行，停止或终止）。键入 ctrl-c 会使 SIGINT 信号传递到前台作业中的每个进程。SIGINT 的默认操作是终止该进程。键入 ctrl-z 会使 SIGTSTP 信号传递到前台作业中的每个进程。SIGTSTP 的默认操作时将进程置于停止状态，该进程会一直保持到收到 SIGCONT 信号将其唤醒为止。Unix shell 还提供了各种支持作业控制的内置命令。Such as：

- Jobs：展示所有运行中的和停止的后台进程。
- bg <job>：改变停止的后台进程到一个运行的后台进程。
- fg <job>：改变一个运行着的后台进程变成一个运行的前台进程。
- kill<job>：终止一个进程（后台）。

实验要求：

1．提示应为字符串“ tsh>”。

2．用户键入的命令行应由名称和零个或多个参数组成，所有参数均由一个或多个空格分隔。 如果 name 是内置命令，则 tsh 应该立即处理它并等待下一个命令行。 否则，tsh 应该假定名称是可执行文件的路径，该文件将在初始子进程的上下文中加载并运行。

3．Tsh 不需要支持管道(|)或者 I/O 重定向(<and>)。

4．键入 ctrl-c（ctrl-z）应该会导致 SIGINT（SIGTSTP）信号发送到当前前台作业以及该作业的任何后代（例如，它派生的任何子进程）。 如果没有前台作业，则该信号应该无效。

5．如果命令行以＆结束，则 tsh 应该在后台运行作业。否则它将会在前台运行作业。

6．可以通过进程 ID（PID）或者 job ID（JID）标识每个作业，该 ID 时 tsh 分配的正整数。JID 应该在命令行上用前缀“%”表示。例如“%5”表示 JID 为 5，“5”表示 PID 为 5.（此时处理 job 的相关程序已经给出）

7．Tsh 有以下内置的命令：

a) quit：终止 shell。

b) jobs：打印出所有后台作业。

c) bg <job>：发送 SIGCONT 信号给 job，同时重启该 job 在后台运行。Job 可以表示为 PID 或者 JID。

d) fg <job>：发送 SIGCONT 信号给 job，同时重启该 job 在前台执行。（能执行 fg 内置命令表明目前前台没有作业），job 可以时 PID 或者 JID。

8．tsh 应该处理所有的 zombie 子进程。如果任何作业由于接收到未捕获的信号而终止（比如 SIGINT），则 tsh 应该识别此事件同时打印一条带有该作业的 PID 消息以及对该问题的信号描述。

检测方法：

见 shlab 文档。

实验相关知识：

该实验考察课本第八章的 ECF 主要内容，首先我们对相关知识以及函数做一个简单的了解：

1．exception：

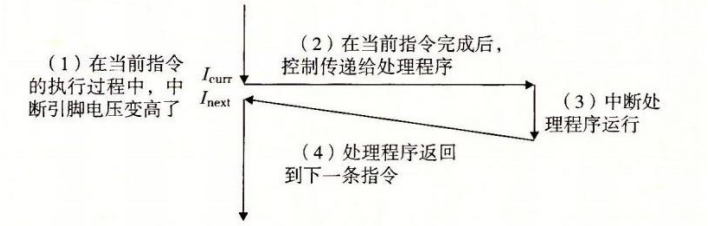
异常分为 4 类，为：

- 中断（interrupt）来自 I/O 设备的信号，异步，总是返回到下一条指令。
- 陷阱（trap）有意的异常，同步，总是返回到下一条指令。
- 故障（fault）潜在的可恢复的错误，同步，可能返回到当前指令

- 终止（abort）不可恢复的错误，同步，不可返回。

中断：

处理信号时的操作。流程图如图：

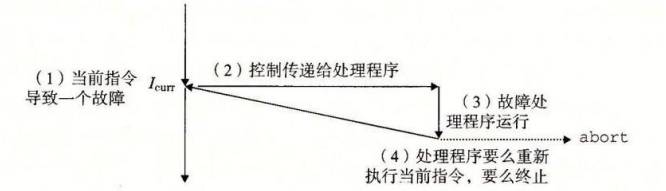


陷阱和系统调用：

陷阱是有意的异常，是执行一条指令的结果。像中断处理程序一样，陷阱处理程序将控制返回到下一条指令，陷阱最重要的用途是在用户程序和内核之间提供一个像过程一样的接口，叫做系统调用。（用户程序向内核请求的服务，例如 fork()，exit()等），从程序员的角度来看，系统调用和普通的函数调用时一样的。但是实现方式不同，普通的函数运行在用户模式中，用户模式显示了函数可以执行的指令的类型，而且它们只能访问和调用相同的栈。系统调用运行在内核模式中，内核模式永续系统调用执行特权指令，并访问定义在内核中的栈。

故障：

处理错误的情况，当故障发生时，处理器会控制转移给故障处理程序，流程如下：



2．进程：

对于进程我们需要关注下列概念和函数：（操作系统如何实现进程超范围不予考虑）

1．关注进程抽象：

1．逻辑控制流：提供假象，好像我们的程序在独占处理器。

2． 私有的地址空间：提供假象，好像我们的程序独占的使用内存系统。

2．并发/并行：

1．并发：多个流并发的执行（一般在一个 CPU 上）的现象为并发。

2． 并行：多个流并发的运行在不同的 CPU 上的现象为并行。

3．用户模式和内核模式：

处理器用某个控制寄存器中的一个模式位提供相关功能，设置了模式位进程就运行在内核模式中（也就是超级用户模式），当异常发生时，控制传递到

异常处理程序此时处理器将模式从用户模式转变为内核模式。内核模式可以执行指令集中的任何指令同时可以访问系统中的任何内存设置。

4．上下文切换：

内核为每个进程维持一个上下文，上下文就是内核重新启动一个被抢占的进程所需的状态（寄存器值，用户栈，状态寄存器等），当在进程执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个先前被抢占了的进程（这个机制和子进程停止直到 SIGCONT 信号到来的机制不太相同，调度维持一定规则比如时间规则为同步，但是信号为异步，但是发生上下文的切换却是接收信号的时间点）。这种决策叫做调度。

5．进程控制：

从程序员角度来看程序总处于以下三个状态之一：

1．运行：进程要么在 CPU 上执行，要么等待被执行且最终被内核调度。

2．停止：进程的执行被挂起，且不会被调度。当收到 SIGSTOP,SIGTSTP 等信号时，进程就停止，并且保持停止直到它收到一个 SIGCONT 信号，这个时刻进程再次开始运行。(pause 和 sleep 是主动挂起，直到收到信号或者时间到达，取消阻塞到达 ready 状态)。

3．终止：进程永远的终止了。进程会因为三种原因终止：

a） 收到一个信号，该信号的默认行为为终止进程。

b） 从主程序返回。

c） 调用 ***exit ()*** 函数。（系统调用，在内核中执行）

6．进程相关系统调用

1．exit()

不返回，以 status 退出状态来终止进程（也就是终止后在进程中留下退出状态值 status），此时在内核中仍由进程的相关信息比如调度信息等，留到 waitpid 再讨论。

2．fork()

创建子进程时使用，调用一次返回两次，新创建的子进程几乎与父进程完全相同，父进程中返回的为子进程的 PID，而子进程中返回 0，然后两者并发执行，拥有相同但是独立的地址空间。

3． waitpid(pid_t pid, int *statusp, int options)

进程终止时，内核并不是立即将他清除，相反，进程被保存再一种已经终止的状态中，直到被父进程回收。当父进程回收已经终止的子进程时，内核将子进程的退出状态传递给父进程，然后抛弃已经终止的进程，一个终止但未回收的进程未僵尸进程。

默认情况下(options=0 时),waitpid 挂起调用进程的执行，直到它的等待集合中的一个子进程终止（非停止）。如果等待集合中的一个进程已经终止了，那么 waitpid 立即返回。在着两种情况下 waitpid 返回已终止的子进程的 pid。此时子进程已经被回收。

参数分析：

1．判定集合成员：

Pid>0： 等待集合为一个单独的子进程，它的进程 ID 为 pid。

Pid=-1：等待集合由父进程的全体子进程构成。

2．修改默认行为（只挑重点）

WNOHANG：如果等待集合重点任何子进程都没有终止，那么立即返回（返回值为 0）。默认的行为时挂起调用进程，直到由子进程终止。在等待子进程终止的同时，如果还想做些有用的工作，这个选项会有用。

WUNTRACED：挂起调用进程的执行，直到等待集合中的一个进程变成已终止或者被停止。返回的 PID 为导致返回的已终止或被停止的子进程的 PID。当要检查已终止和被停止的子进程时，这个选项会有用。

WNOHANG | WUNTRACED：立即返回。如果等待集合中的子进程都没有被停止或终止，则返回值为 0；如果有一个停止或终止，则返回值为该子进程的 PID。

3．检查已回收子进程的退出状态

如果 statusp = NULL，那么不做处理，如果不为空，则 waitpid 会在 status 中放上关于导致返回的子进程的状态信息，status 时 statusp 指向的值，有如下几个宏可以调用来处理 status：

WIFEXITED(status)：子进程调用 exit 或者 return 终止返回真

WEXITSTATUS(status)：返回正常终止的子进程的退出状态。只有在上面返回为真的时候才会定义这个状态

WIFSIGNALED(status)：子进程因未捕获的信号终止返回真（例如 SIGINT）

WTERMSIG(status)：返回导致子进程终止的编号。只有上面返回为真时才定义这个状态。

WIFSTOPPED(status)：如果引起返回的子进程目前停止，那么返回真

WSTOPSIG(status)：返回导致子进程停止的信号的编号。

4．错误条件

如果调用没有子进程，那么 waitpid 返回-1，同时设置 errno 为 ECHLD。如果 waitpid 函数被中断，那么返回-1，设置 errno 为 EINTR.

4. execve(const char *filename, const char **argv, const char **envp);

函数加载并运行可执行目标文件 filename，且带参数列表 argv 和环境变量 envp。只有出现错误时，execve 才会返回到调用程序。代表了与 fork 不同，execve 调用一次从不返回。

参数：

argv：指向一个 null 结尾的指针数组，其中每个指针都指向一个参数字符串。按照惯例，argv[0]为可执行文件的名字。

envp：指向一个以 null 结尾的指针数组，其中每个指针都指向一个环境变量字符串，每个串都时形如“name=value“的名字-值对。

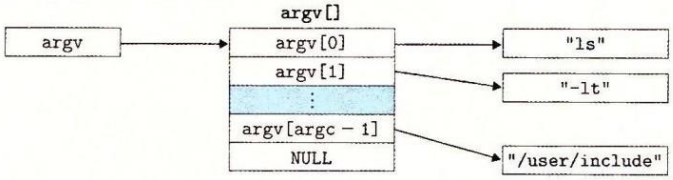


图 8-20 参数列表的组织结构

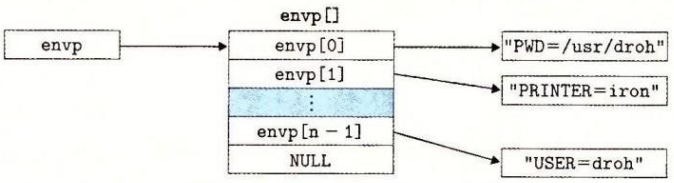


图 8-21 环境变量列表的组织结构

（其他不在我们这里赘述，只选取关乎我们实现的部分。）

3．信号

信号允许内核或进程中断其他进程。信号也就是小消息。我们关注信号的以下方面：

1．相关信号了解

SIGINT（2）：ctrl+c，内核会对当前前台进程组中的所有进程发送。默认行为终止。

SIGTSTP（20）：ctrl+z，默认行为为停止直到下一个 SIGCONT。

SIGCONT（18）：继续进程如果该进程目前停止。

SIGKILL(9)：一个进程可以发送 SIGKILL 信号给另一个进程去强制终止它。

SIGCHLD(17)：当子进程终止或停止时（很重要），内核会发送一个 SIGCHLD 信号给父进程。

2．发送信号

内核根据更新目的进程上下文的状态，发送一个信号给目的进程。发送信号有两种原因。1）内核检测到某个事件（ctrl+z，子进程终止等）2）一个进程调用了 kill 函数。

相关概念：

1．进程组

每个进程只属于一个进程组，进程组是由进程组 ID 标识，默认一个子进程和他的父进程为同一进程组，可以用 setpgid 函数改变进程的进程组。

* ***setpgid(pid_t pid, pid_t pgid);***

setpgid 函数将进程 pid 的进程组改为 pgid。有如下实例：

setpgid(0, 0);此时会创建一个新的进程组，组 ID 和此时调用的进程的 PID 相同同时把这个进程加入这个进程组。

2．从键盘发送信号

在键盘输入 ctrl+c 会导致内核发送一个 SIGINT 信号给前台进程组中的所有进程。默认情况下是终止前台作业。类似的 ctrl+z 发送一个 SIGTSTP 的信号给前台进程组中的每个进程。默认情况下为挂起前台作业。

3．作业（job）

Linux 用 job 来表示为一条输入的命令行执行而创建的进程。在任何时刻，之多只有一个前台作业和 0 到多个后台作业。Shell 为每个作业创建一个独立的进程组，每个进程组的 ID 通常取作业中父进程的一个。

4．Kill 发送信号

**** int kill(pid_t pid, int sig);***

参数条件：

Pid>0:: kill 发送信号号码为 sig 给进程 pid。

Pid=0：kill 发送 sig 给调用进程所在进程组中所有进程，包括自己。

Pid<0：kill 发送 sig 给进程组|pid|（pid 的绝对值）中的每个进程。

3．接收信号

当进程 p 完成系统调用或上下文切换从内核模式转变为用户模式时，这时检查进程 p 的未阻塞的待处理信号集合，（内核默认会为每个进程设置待处理信号集合和阻塞信号集合，一个代表了该进程还未处理的信号，一个代表了该进程可忽略的信号）如果集合为空，则内核将控制传递到 p 的逻辑控制流的下一条指令，如果集合非空，那么内核选择集合中的某个信号 k（通常为最小的 k），并强制让 p 接收信号 k。收到信号会出发进程采取某种行为。一旦进程完成了这个行为（行为不为终止或停止），那么控制传递回 p 的逻辑控制流

中的下一条指令。

信号的默认行为有以下几种：

* 进程终止

* 进程停止直到 SIGCONT 信号到来重启

* 进程忽略该信号

设置信号处理程序：

sighandler_t signal(int signum, sighandler_t handler);

此函数的作用为改变*调用的进程*接收到响应信号的默认处理，通过把处理程序的地址传递到 signal 函数从而改变默认行为。这叫做*设置信号处理程序*。调用信号处理程序被称为*捕获信号*。执行信号处理程序被称为*处理信号*。

信号处理程序根据信号的号大小不同可能会发生中断，也就是进程 p 正在执行。

一个信号处理程序时，若捕获到了另一个号码大小更小的信号，则会被另一个信号中断，此时如果另一个信号的行为为停止，则会导致很差的效果，因为另一个信号处理程序未能执行结束。（这里的机制为源程序压入内核栈然后执行相应的中断处理程序（不能算单独进程）和上下文切换的概念（进程之间）又有些不同）。

4．阻塞和接触阻塞信号

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的信号。

2．显示阻塞机制

**** int sigfillset(sigset_t* set);*** 将所有信号加入 set 中

**** int sigaddset(sigset_t* set, int signum);*** 将 sig 信号加入 set 中

**** int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)***

How=SIG_BLOCK:把 set 中的信号添加到 blocked 中

How=SIG_SETMASK: block=set

以上行为都将会把原阻塞表赋给 oldset

5．编写信号处理程序

信号处理程序很难，有以下原因 1）处理程序与主程序并发运行（指上下文切换时如果有信号则会中断进行处理，简单来讲就是不知道何时进行中断）2）如何和何时接收信号的规则有违人的直觉。所以我们要保证以下几点：

1．安全的信号处理

0．处理程序尽量简单

1．在处理程序中只调用异步信号安全函数（简单而言就是不被信号处理程序中断）

2．保存和恢复异常错误 errno

3．当访问全局变量时，阻塞所有的信号（有先后顺序的访问逻辑）

4．用 volatile 声明全局变量，此时每次调用 volatile 时都会从内存中调。

5．用 sig_atomic_t 声明标志。对它的读写（不保证运算）是原子的。

2．正确的信号处理

信号是不排队的。因为 pending 位向量（待处理信号向量）中每种类型的信号只对应有以为，所以每种类型最多只能由一个未处理的信号。因此，如果两个类型 k 的信号发送给了同一个目的进程，而因为目的进程当前正在执行信号 k 的吃力程序，所以信号 k 被阻塞了，第二个信号也会被丢弃。

3．同步流以避免并发错误

主要是进程调度（上下文切换）导致对某些全局量的访问没有按一定的顺序，要通过设置相关的屏蔽位来阻止接收该信号（因为不知道什么时候会处理信号，但是可以设置屏蔽位来保证在执行某一块代码时绝对不会来触发中断去处理信号。）

4．显式的等待信号

有时候主程序需要显式的等待某个信号处理程序的运行。例如 shell 创建一个前台作业时，在接收下一条用户命令之前，它必须等待作业终止，被 SIGCHLD 处理程序回收。

**** sigsuspend(const sigset_t *mask); sigsuspend（const sigset_t * mask）;***

Sigsuspend 函数暂时用 mask 替换当前的阻塞集合，然后挂起该进程（shell 挂起后就不会显示新的 tsh>，已经挂起不向下运行），直到收到一个信号，其行为要么时运行一个中断处理程序，要么时终止该进程。

实验分析：

参考教材 p525 页的残缺 shell，目前要实现的函数为：

1．eval：解析和解释命令行的主例程（70 lines）

2．builtin_cmd：识别和解释内置命令（25 lines）

3．do_bgfg：实现 bg 和 fg 内置命令（50 lines）

4．waitfg：等待前台工作完成（20 lines）

5．sigchld_handler：处理 SIGCHILD 信号（80 lines）

6．sigint_handler：处理 SIGINT 信号（15 lines）

7．sigtstp_handler：处理 SIGTSTP 信号（15 lines）

此时根据题目中的引导 test 一个个分析即可。（见实验设计）

实验设计：

Test1~4：

首先实现一个不含信号控制的框架（参考 P525），也就是不会回收后台作业，首先进行命令行分析得到相关的 argv（通过 parseline 函数实现，题目中已经实现这里不深入探究其中的具体实现）.

同时分析的命令行会得出为前台作业还是后台作业。解析完成后 eval 调用 builtin_cmd 函数，该函数检查第一个命令是不是内置的 shell 命令，如果是，它就立刻解释这个命令并且返回值 1（表明处理完成，如果能返回的话就直接又有新的 tsh>出现等待新的命令行输入），此时我们设置 shell 只有简单的 quit 内置命令只需要直接终止父进程 shell 调用 exit（0）即可）此时如果 Builtin_cmd 返回为 0，表明此时不是内置函数，那么 shell 创建一个子进程，并且在子进程中执行所请求的程序。如果用户要求在后台运行该程序，那么 shell 返回到循环的顶部，tsh>等待下一个命令输入，否则，shell 由于我们先不加信号控制的终结，此时我们调用 waitpid 使用默认值来等待作业停止。当作业停止时会向父进程 shell 发送信号，此时 shell 从挂起状态再次从上次挂起的地方开始执行（这里挂起为动作，停止为状态），shell 开始下一轮迭代.

此时改动的相关函数为 eval，Builtin_cmd 即可完成 1~4。

Test5：

此时首先我们要注意要添加 jobs 内置命令，同时得修改 eval 分析函数，使之能加入 job，但是加入 job 就涉及到在 job 终止后的回收，这可能会导致同步流的并发错误（参考前面的相关知识），处理同步流并发错误的具体方法为（参考 P543），改变 eval 和 sigchld_handler 使用显性屏蔽使之能正确的加入和删除任务。同时如果要处理前台程序要改变 waitfg，使之显性等待子进程的结束（参考上知识储备），使用 sigsuspend（const sigset_t * mask）显式的等待子进程结束后发的 SIGCHLD，再转 sigchld_handler 处理正常终止后的输出。

此时改动相关函数为 eval，Builtin_cmd，sigchld_handler，waitfg。

Test6-7：

此时要求我们能处理 SIGINT（也就是 ctrl+c）来停止前台进程，值得注意的 SIGINT 信号在 shell 父进程接收到后转 sigint_handler，sigint_handler 的作用是要给前台进程组内的所有进程都发 SIGINT 来提前终止，子进程（此时为前台作业）终止后会向它的父进程 shell 发送 SIGCHLD 信号此时在 sigchld_handler 中得添加子进程因意外终止的相关处理。

此时改动相关函数为 sigint_handler，sigchld_handler

Test8：

此时要求处理 SIGTSTP，和 test6 基本相同，除了 kill 发送的信号量不相同，在 sigchld_handler 处理函数中增加子进程因停止而发的 SIGCHLD 的相关处理。

此时改动的相关函数为 sigtstp_handler，sigchld_handler

Test9-10：

此时增加内置功能 bg 和 fg，bg 和 fg 的要实现功能在实验要求中，此时首先分析传入的为进程号 pid 还是作业号 jid。然后对于 bg 而言，此时实现的功能为在后台重启，设

置该要求进程 state 为 BG，同时向该进程（job）发送 SIGCONT 来重启作业。该重启作业结束后自然会发送 SIGCHLD 给父进程 shell 然后转 SIGCHLD 处理程序。(此时转的时候父进程并没有停留在 sigsuspend 显式调用，只是有 SIGCHLD 信号在信号能被接收的时间处理，所有后台进程的回收都是利用这个机制)。对于 fg 而言，此时需要显示等待该进程的停止,所以使用 waitfg 函数来等待前台作业的终止然后接收 SIGCHLD 转 SIGCHLD 处理程序。

此时改动的相关函数为 do_bgfg。

Test11-16：

是以上的组合设计，要完美实现以上功能还需注意以下细节：

1． 在 sigint_handler 和 sigtstp_handler 向前台作业发送相关信号时，需注意需使用-pid 来发给前台进程组的每一个进程（虽然没有管道注定每一个作业只有一个进程)。

2． 在 eval 中，父进程 shell 必须在派生子进程之前使用 sigprocmask 来阻止 SIGCHLD 信号，然后通过调用 addjob 将子进程添加到作业列表中后，然后再使用 sigprocmask 取消之前阻止的信号。由于子进程会继承父进程的阻止向量，所以必须在 execve 之前小心的切除掉 SIGCHLD 信号。父进程 shell 需要以这种方式阻塞信号的原因是为了避免在父级调用 addjob 之前，子进程就会被 SIGCHLD 处理程序处理（如果没有 sigprocmask 那么处理信号将会在接收信号的时间点处理。可能就会导致在 add 之前先发生子进程 SIGCHLD 终止中断，见相关知识->接收信号)。

· 3．在 SIGCHLD 信号处理时既要处理终止的子进程发出的 SIGCHLD 又要处理停止(挂起）的子进程发出的 SIGCHLD，此时在 waitpid 的 option 中得使用 SIGINT and SIGTSTP 参数来辅助处理。

4．当您从标准 Unix shell 运行 shell 时，shell 在前台进程组中运行。 如果您的 shell 随后创建了一个子进程，则默认情况下，该子进程也将成为标准 shell 前台进程组的成员。 由于输入 ctrl-c 会将 SIGINT 发送到前台组中的每个进程，因此输入 ctrl-c 会将 SIGINT 发送到自己的 shell 以及自己的 shell 创建的每个进程，这显然是不正确的。

解决方法是：在派生之后，但在执行之前，子进程应调用 setpgid（0，0），这会将子进程放入一个新的进程组中，该组的组 ID 与该子进程的 PID 相同。 这样可以确保在前台进程组中只有一个进程，即 Shell。 键入 ctrl-c 时，外壳程序应捕获结果 SIGINT，然后将其转发到适当的前台作业（或更准确地说，是包含前台作业的进程组)。

相关问题：

1．在测试 test04 时总显示 jid 为 2 实际应该为 1？

解决：因为实现时还没有实现 SIGCHLD 信号的处理导致无用行（前面的导言）会被当成路径，但是并不是会被简单丢弃设置 exit（0），但是此时没有信号处理无法丢弃，所以此时这个无用行也会被当成一个前台作业留在其中（因为导言后没有&)。

2．在测试 test10 时 fg %1 会卡死？

解决：如果在 eval 中阻塞信号过早，在 built_cmd 之前阻塞，会让 do_bgfg 中的

prev 也阻塞 SIGCHLD 从而调用 fg %1 会一直卡死在 waitfg 函数，解决方法就是在 built_cmd 之后一行来实现。

实验结果：

见 tsh_out.txt

总结：

总体实现了简单 shell 的基本功能，能按实验要求实现相关的内置命令和加载并运行新的前台/后台作业。如果要实现管道等相关其他实现得进一步了解并行编程同时了解 linux 内核。

相关实验环境：

Liunx - deepin2.0

Cpu - inter i7-9750h 2.6GHz

Complier - gcc

相关实现：

见 tsh.c

具体代码分析见 README.md