第2章 进程控制

在学习操作系统课程中，我们学习了进程的基本概念、进程创建于撤销、进程调度、进程同步与通信等知识。本章先来观察体验 Linux 中的进程管理，下一章再讨论进程间同步与通信的内容。

我们用 file 命令检查输出的可执行文件，如屏显 2-1 所示，表明这是 x86-64 平台上的 ELF 格式可执行文件。

屏显 2-1 HelloWorld 可执行文件信息

```
[lqm@localhost ~]$ file HelloWorld
HelloWorld: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs),
for GNU/Linux 2.6.32, BuildID[sha1]=881c65d1c673694f0dad089dcb2a3c150c17e3f0, not stripped
[lqm@localhost ~]$
```

由于 ELF 文件内部包含了足够的信息，操作系统可以根据 ELF 可执行文件创建出进程影像，大致的示意图如图 2-1 所示。即将 HelloWorld 文件的代码部分拷贝到新进程虚存空间的低地址部分（对应图中 code），然后将可执行文件中的数据拷贝到代码的略高位置（对应图中 data），然后在用户进程高端位置建立用户态的堆栈（对应图中 stack），以及在内核空间中分配一些管理数据结构，例如进程控制块 PCB（图中的 struct task_struct）和内存描述符（图中的 struct mm_struct）等。

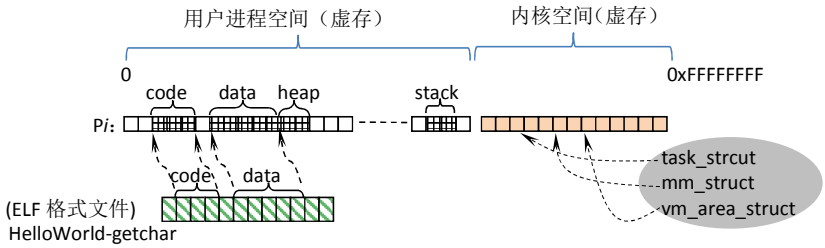

图 2-1 内存模型中一个进程的影像

进程映像以及各种管理数据结构构成了进程实体，另外进程还可能向系统申请各种资源——共同组成的进程时提的全部内涵。进程实体被调度运行就构成了进程的活动，如果系统中有多个活动的进程，那么它们各自有独立的进程空间（各自独立地使用自己的 0~0xFFFFFFFF 虚存空间）。

2.1. 进程基本概念

首先，进程是程序（静态概念）的一次运行（动态概念），每一次运行都需要操作系统为之分配必要的资源（内存、CPU、PCB 编号等）而构成进程实体。在命令行中输入可执行文件名然后回车，shell 程序就将创建子进程以便执行指定的可执行文件。

如果我们将 HelloWorld-getchar 运行三遍——为了方便观察每个在一个独立终端上运行，然后在第四个终端上用 ps 命令查看进程。如图 2-2 所示，根据 HelloWorld-getchar 可执行程序（磁盘文件）产生出来的第一个进程编号为 4502 并运行在伪终端 pts/0 上，其次有 4503 和 4549 两个进程分别运行在伪终端 pts/1 和 pts/2 上。进程编号就是操作系统教材上的 PCB 编号，本书后面将采用 Linux 的术语 PID（Process ID）来指代。

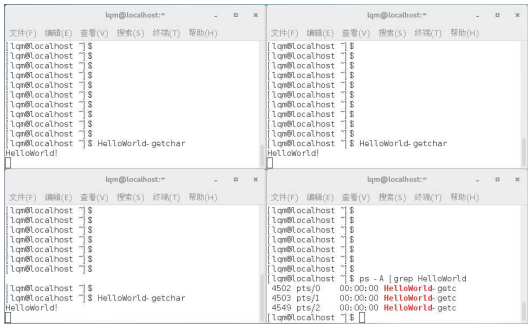

图 2-2 HelloWorld 运行三次创建三个进程

如果用 ps –aux 命令可以看到完整的可执行文件名，如屏显 2-2 所示。此时还可以看到这三个进程都处于 S 阻塞睡眠状态（后面会讨论进程状态）。

屏显 2-2 察看 HelloWorld-gecha 的三个进程

```
[lqm@localhost ~]$ ps -aux |grep HelloWorld
lqm        4502  0.0  0.0   4164   344 pts/0    S+   16:56   0:00 HelloWorld-getchar
lqm        4503  0.0  0.0   4164   340 pts/1    S+   16:56   0:00 HelloWorld-getchar
lqm        4549  0.0  0.0   4164   344 pts/2    S+   16:56   0:00 HelloWorld-getchar
lqm        5025  0.0  0.0 112664   976 pts/3    R+   17:11   0:00 grep --color=auto HelloWorld
[lqm@localhost ~]$
```

因此，我们首先建立起进程是程序的一次动态运行的直观感受，其次认识到一个程序（可执行文件）可以创建多个进程。

另外，要认识到 Linux 终端上运行着 shell 进程，该 shell 进程打印出命令行提示符（例如[lqm@localhost~]），然后等待用户的输入。我们在命令行上输入可执行文件名（例如 HelloWorld-getchar）并输入回车后，shell 进程检查输入字符串，如果不是内部命令则创建一个子进程来执行相应的程序。shell 进程在子进程运行时，自己将阻塞，直到子程序结束或者转入后台，shell 进程才再次获得运行并打印新的提示符。

2.1.1. 进程间组织关系

Linux 系统中所有进程通过进程控制块 PCB（struct task_struct）形成多种组织关系。根据进程创建过程可以有亲缘关系，通过进程间的父子关系组织成一个进程树（如图 2-3 和屏显 2-3 所示）；根据用户登录活动有会话、进程组等关系（如图 2-4 和屏显 2-4 所示）；根据调度策略和进程状态则会归入到不同队列中，将实时进程组织成位图队列，而将普通进程组织成红黑树，进程阻塞时还会被挂入事件的等待队列中。

进程在创建的时候就需要建立起这些关系，并且在执行中可能会动态调整。

亲缘关系

一个进程通过创建一个子进程则形成父子关系，如果创建多个子进程，那么这些新创建的进程间属于兄弟关系。这些关系可以利用各自的 task_struct 中的 parent、children 和 sibling 成员（都是 list_head 结构体，用 prev 和 next 指针指向前驱和后继）组织这些关系，可用图 2-3 表示，图中 P0 创建了三个子进程 P1、P2 和 P3，而 P3 创建了 P4。子进程在结束时需要向父进程发送 SIGCHLD 信号，如果父进程提前结束那么子进程将把 1 号 init 进程作为父进程。

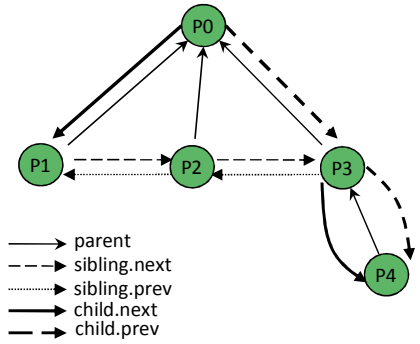

图 2-3 进程间亲缘关系 （P4 的 child.next/prev 是这样的吗？）

通过 pstree 命令，我们可以看到全系统的进程树结构。我们的系统上的全部进程构成的进程树如屏显 2-3 所示。可以看出我们在 GNOME 的图形桌面上的三个伪终端上的 bash 各自创建了一个 HelloWorld-getchar 子进程，另一个终端上的 bash 进程创建了三个子进程（more、pstree 和 top）

屏显 2-3 pstree 输出

```
[lqm@localhost ~]$ pstree
systemd─┬─ModemManager───2*[{ModemManager}]
        ├─NetworkManager───2*[{NetworkManager}]
        ├─2*[VBoxClient───VBoxClient───{VBoxClient}]
        ├─VBoxClient───VBoxClient
        ├─VBoxClient───VBoxClient───2*[{VBoxClient}]
        ├─VBoxService───7*[{VBoxService}]
        ├─2*[abrt-watch-log]
        ├─abrtd
        ├─accounts-daemon───2*[{accounts-daemon}]
        ├─alsactl
        ...
        ├─gnome-terminal─┬─┬─3*[bash───HelloWorld-getc]
        │                ├─bash─┬─more
        │                │      ├─pstree
        │                │      └─top
        │                ├─gnome-pty-helpe
        │                └─3*[{gnome-terminal-}]
        ├─goa-daemon───3*[{goa-daemon}]
        ...
        ├─sshd
        ├─systemd-hostnam
        ├─systemd-journal
        ├─systemd-logind
        ├─systemd-udevd
        ├─tracker-store───7*[{tracker-store}]
        ├─tuned───4*[{tuned}]
```

```
├─udisksd─────4*[{udisksd}]
├─upowerd─────2*[{upowerd}]
└─wpa_supplicant
```

进程的父子关系还可以通过父进程的 PID（Parent PID）来展示。例如屏显 2-4 中展示了 shell 上先后运行“top | more&”和“ps j |more”命令后，PID 为 2350（top）、2351（more）、2352（ps -j）和 2353（more）进程的 PPID（父进程 PID）都是 1858，即它们都是由父进程 1858（bash）创建的。而 1858 进程的 PPID 表明它是由 1857 进程创建的。

会话、进程组、线程组及控制终端

“会话/进程组/线程组”几个概念呈现层级关系，Linux 系统中可以有多个会话（session），每个会话可以包含多个进程组（process group），每个进程组可以包含多个进程，每个进程构成一个线程组——一个线程组由一个进程内的一个或多个线程组成，这些对象都有各自的 PID。

会话是用户登陆系统到退出前的全部活动，不同帐户的登录属于不同会话，而同一帐户的多次登录也构成不同的会话。而进程组主要出于作业控制的目的，有些 Shell 没有作用控制能力，就将整个会话的所有进程构成一个进程组。。

每当用户登录到 Unix 主机时，系统将打开一个终端运行 login 程序等待输入用户名和密码，通过密码验证后则创建一个 shell 程序从该终端读取用户命令和输出显示信息，由该 shell 创建的所有进程都使用这个终端，即同一会话所有进程使用相同终端。从登陆到结束前创建的所有进程都属于这次会话（session），第一个被创建的进程（通常是 shell）称为会话的领头进程（session leader），而系统打开的那个终端就是这些进程的控制终端。进程打开/dev/tty 可以获得当前进程的控制终端文件描述符 fd（没有控制终端的进程则会失败），用库函数 tcgetsid(fd)可以获得会话 ID（SID）。除了用户登录而产生的会话外，守护进程利用 setsid()系统调用（linux-3.13/kernel/sys.c）及其它相关操作建立自己的会话和进程组，这类会话与用户登录会话相互独立避免相互影响，而且守护进程的会话是没有控制终端的。

屏显 2-4 查看会话、进程组和控制终端的例子

```
[lqm@localhost ~]$ top | more&
[1] 2351
[lqm@localhost ~]$ ps j |more
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 1857  1858  1858  1858 pts/1     2352 Ss    1000   0:00 -bash
 1858  2350  2350  1858 pts/1     2352 T     1000   0:00 top
 1858  2351  2350  1858 pts/1     2352 T     1000   0:00 more
 1858  2352  2352  1858 pts/1     2352 R+    1000   0:00 ps j
 1858  2353  2352  1858 pts/1     2352 S+    1000   0:00 more
```

上面的命令显式通过“top |more &”创建 top 进程和 more 进程，并且在后台运行。然后通过“ps j |more”命令创建 ps 进程和第二个 more 进程，其中 ps 的参数 j 表示用任务格式来显示进程。显示 bash 是这次会话的领头进程，此会话的所有进程的 SID 都为 1858（会话领头进程 bash 的 PID）。

进程组用于作业控制，它允许一个终端上启动多个作业（进程组），并且控制哪个作业在前台并可以访问终端，哪些作业在后台运行没有控制终端。

上面的输出还显示出有三个进程组，即 PGID（process group ID，进程组 ID）为 1858 的“bash”、2350 的“top | more&”和 2352 的“ps j | more”进程组。在 shell 上的一条命令所

产生的所有进程形成一个进程组，每个进程组内第一个进程往往成为领头进程（process group leader），并以领头进程的 PID 为组内进程的 PGID。进程组的生命周期持续到组中最后一个进程终止，或加入其他进程组为止。可以利用 getpgrp()系统调用来获取进程组 ID。但是如果 shell 没有作业控制（job control）能力则不管理进程组，例如 ash 中创建的进程都属于同一个会话、同一个进程组。

一个带有控制终端的会话有一个前台进程组（ foreground process group），还可以有一个或多个后台进程组（background process group），其关系如图 2-4 所示。

就用我们的例子来绘制它们的关系图，具体如图 2-4 所示，此时前台进程组为 2352（ps j 和 more）。

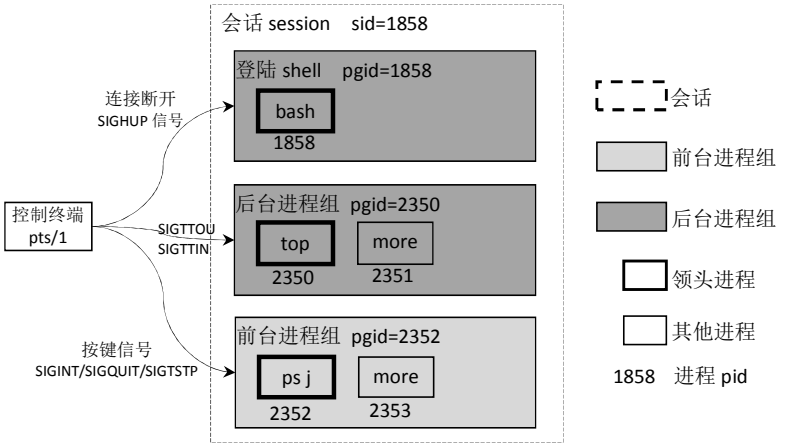

图 2-4 会话、进程组以及前后台关系例子

在这里除了 bash 的父进程号为 1857 外，其余四个进程是兄弟进程且父进程号都是这个 bash 的 PID——1858。

进程刚创建时只有一个主线程，此时 task_struct->group_leader 指向自己，其后创建的线程 task_struct->group_leader 指向线程组长（主线程）。主线程 task_struct->thread_group 作为链表头，其他子线程通过 task_struct->thread_group 构成链表。

进程组提供了一个机制，让信号可以发送给组内的所有进程，这使得作业控制和其他 shell 活动变得简单。其中前台进程连接到控制终端，若在控制终端键入中断键（Ctrl-C 或 Delete）或退出键（Ctrl-\），就会造成将中断信号 SIGINT 或退出信号 SIGQUIT 送至前台进程组的所有进程。挂起键（Ctrl+Z）对应的 SIGTSTP 也发送给前台进程组的所有进程。而检测到终端的 modem 或网络断线时，SIGHUP 将发给会话领头进程，当会话领头进程终止后内核还会将此信号送往前台进程组，进程组的默认动作是终止进程。如果禁止后台进程组操作终端后，后台进程进行终端读写操作则引发 SIGTTOU 或 SIGTTIN 信号。这些信号的发送关系如图 2-4 所示。

2.1.2. 进程控制命令

这里只讨论通过 shell 命令方式创建进程的方是，至于通过编程接口来创建进程的方法将在 2.2 小节讨论。

创建进程的命令

除了在 shell 命令行输入可执行文件名的方式来创建进程外，还可以用脚本方式创建进程，以及也可以通过 at、cron 命令定时地创建进程。脚本方式本质上和命名行方式差不多，其实都是由 shell 创建进程，只不过提前将命名输入到一个脚本文件中。例如我们编辑一个 shell 脚本如代码 2-1 所示。这个 shell 脚本只有三行，第一行用于指出该脚本需要用/usr/bin/bash 来解释执行，第二行 shell 内部命令 echo 用来显示，第三行 HelloWorld 是外部命令（磁盘上的可执行文件）。

代码 2-1 shell 示例（创建进程）

```
#!/usr/bin/bash
echo Running a shell script!
HelloWorld
```

将它保存为 shell-script，此使用 ls -l 查看发现它还不运行执行。用 chmod a+x shell-scrpit 命令为它增加执行权限，再用 ls –l 检查就发现有了“x”执行权限。在提示符下输入脚本文件名以运行该脚本程序，可以看出里面创建并运行了 HelloWorld 进程。以上操作具体如屏显 2-5 所示。

屏显 2-5 shell-script 脚本的执行

```
[lqm@localhost ~]$ ls -l shell-script
-rw-rw-r--. 1 lqm lqm 58 3 月  15 19:51 shell-script
[lqm@localhost ~]$ chmod a+x shell-script
[lqm@localhost ~]$ ls -l shell-script
-rwxrwxr-x. 1 lqm lqm 58 3 月  15 19:51 shell-script
[lqm@localhost ~]$ shell-script
Running a shell script!
HelloWorld!
```

虽然脚本里面写入可执行文件会创建进程并运行，但是 shell 脚本程序还可以编写其他更丰富的功能（比如，例子中用 echo 显示了一个行提示信息），这个不在本章讨论范围。at 和 cron 方式定时启动任务进程，请读者自行学习。

撤销进程的命令

命令行上撤销进程主要通过 kill 命令来完成。由于 kill 命令的输入需要进程号，通常先用 ps[1]命令查看到进程的进程号 PID，然后再用“kill -PID”命令杀死 PID 指定进程。例如我们运行得 HelloWorld-getchar，先通过“ps –ef |grep HelloWorld-getchar”查到进程号（3845，如屏显 2-6 所示），然后通过“kill 3485”即可以杀死这个进程。

屏显 2-6 通过 ps 查找进程的 PID

```
[lqm@localhost ~]$ ps -ef |grep HelloWorld-getchar
lqm       3845  3803  0 16:44 pts/0    00:00:00 HelloWorld-getchar
```

[1] 可用的相关命令有 ps、pidof、pstree 和 top 等都可以查看到进程号的信息，请根据具体情况选用不同命令。

```
lqm       4009  3847  0 16:52 pts/1    00:00:00 grep --color=auto HelloWorld-getchar
[lqm@localhost ~]$
```

常用的还有另一个命令，killall 用于杀死指定进程名的所有进程——它不是用 PID（例如“killall httpd”）。

不管是 kill 还是 killall 实际上都是通过发送信号给指定的进程，这些信号可以通过 kill –l 或 killall-l 列出。其中 9 号信号时最强的信号——无条件终止进进程且不能被进程所忽略。关于信号的内容，在后面进程间通信会讨论到。

2.2. 创建进程与撤销

前面的进程创建和撤销都是通过命令行由 shell 为我们创建或撤消进程的，现在我们学习通过 fork()函数来创建进程，并观察 Linux 进程的相关行为。

2.2.1. fork()创建子进程

为了创建子进程需要使用 fork()函数，fork()函数是 C 语言库的函数，它将进一步通过 Linux 系统调用经由操作系统内核完成子进程（child process）的创建。fork()函数的头文件是 unistd.h，函数原形为“int fork(void);”。

fork 函数被成功调用后将安装父进程的样子复制出一个完全相同的子进程。父进程 fork()函数结束时的返回值是子进程的 PID 编号。新创建的子进程则也是从 fork()返回处开始执行，但不同的是它获得的返回值是 0。也就是说父子进程执行上几乎相同，唯一区别是子进程中返回 0 值而父进程中返回子进程 PID。如果 fork()出错则只有父进程获得返回值-1 而子进程未能创建。

子进程是父进程的副本，它将获得父进程数据空间、堆、栈等资源的相同副本。注意，子进程持有的是上述存储空间的“副本”，这意味着父子进程间不共享这些存储空间，它们之间共享的存储空间只有代码段。

下面我们来执行代码 2-2 所示的代码，观察父进程创建子进程的过程。

代码 2-2 fork()示例代码 fork-demo.c

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char ** argv )
{
        int pid = fork();
        if(pid == -1 ) {
                printf("error!\n");
        } else if( pid ==0 ) {
                printf("This is the child process!\n");
                getchar();
        } else {
                printf("This is the parent process! child process id = %d\n", pid);
                getchar();
        }
        return 0;
}
```

然后编译后运行，输出入屏显 2-7 所示。

屏显 2-7 fork-demo 的输出

```
[lqm@localhost ~]$ gcc fork-demo.c -o fork-demo
[lqm@localhost ~]$ fork-demo
This is the parent process! child process id = 14411
This is the child process!
```

然后用 ps 命令和 pstree 命令查看其父子信息，屏显 2-8 表明两个 fork-demo 的进程号分别是 154422 和 1543，其中 15422 进程是 15423 的父进程。

屏显 2-8 fork-demo 的 ps/pstree 输出

```
[lqm@localhost ~]$ ps j
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
...
 4066 15422 15422  4066 pts/3    15422 S+    1000   0:00 fork-demo
15422 15423 15422  4066 pts/3    15422 S+    1000   0:00 fork-demo
 4030 15424 15424  4030 pts/2    15424 R+    1000   0:00 ps j
[lqm@localhost ~]$ pstree 15422
fork-demo———fork-demo
```

如果将 fork-demo.c 做个小修改，多做一次 fork（并且忽略的返回结果的检查），如代码 2-3 所示。

代码 2-3 fork-twice-demo.c

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char ** argv )
{
        int pid = fork();
        if(pid == -1 ) {
                printf("error!\n");
        } else if( pid ==0 ) {
                printf("This is the child process!\n");
                fork();
                getchar();
        } else {
                printf("This is the parent process! child process id = %d\n", pid);
                fork();
                getchar();
        }
        return 0;
}
```

请读者根据代码，判断子进程创建的情况以及所生成的进程树关系，并对照屏显 2-9 检查是否正确——最上层的进程是 15313，下一级的子进程有 15314 和 15315，而且 15314 还产生了更下一级的子进程（孙辈进程）15316。

屏显 2-9 fork-twice-demo 执行后的 ps/pstree 输出

```
[lqm@localhost ~]$ ps j
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
...
 4066 15313 15313  4066 pts/3    15313 S+    1000   0:00 fork-twice-demo
15313 15314 15313  4066 pts/3    15313 S+    1000   0:00 fork-twice-demo
15313 15315 15313  4066 pts/3    15313 S+    1000   0:00 fork-twice-demo
15314 15316 15313  4066 pts/3    15313 S+    1000   0:00 fork-twice-demo
 4030 15367 15367  4030 pts/2    15367 R+    1000   0:00 ps j
[lqm@localhost ~]$ pstree 15313
```

```
fork-twice-demo─┬─fork-twice-demo───fork-twice-demo
                └─fork-twice-demo
```

2.2.2. 孤儿进程和僵尸进程

下面来观察有关孤儿进程和僵尸进程的问题。首先看它们的定义：

（1） 孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被 init 进程（进程号为 1）所收养，并由 init 进程对它们完成状态收集工作。

（2） 僵尸进程：一个子进程在其父进程还没有调用 wait()或 waitpid()的情况下退出。这个子进程就是僵尸进程。

子进程结束后大部分资源都将直接回收，但是还将在 PCB（struct task_struct 和内核堆栈）中记录一下退出时的状态等信息，这些信息供父进程检查后才能销毁回收。因此，程僵尸进程因相应的数据没有父进程处理并回收，将会导致资源浪费，而孤儿则不会有这样的问题。

现在再次回顾代码 2-2，其父进程并没有用 wait()或 waitpid()等待子进程的结束。因此我们再来运行它一次，更仔细的分析其运行过程。首先运行 fork-demo，共有两个进程（16016he 16017）呈现父子关系，此时如图 2-5 上部所示。然后在右边的终端中敲击回车键一次，此时父进程退出，子进程成为孤儿进程并被 init 进程收养（子进程 16017 的 PPID 修改为指向 1 号进程），如图 2-5 中部所示。最后再在右边的终端中点击回车键一次，此时子进程 16017 正常结束，没有造成任何资源浪费，如图 2-5 下部所示。

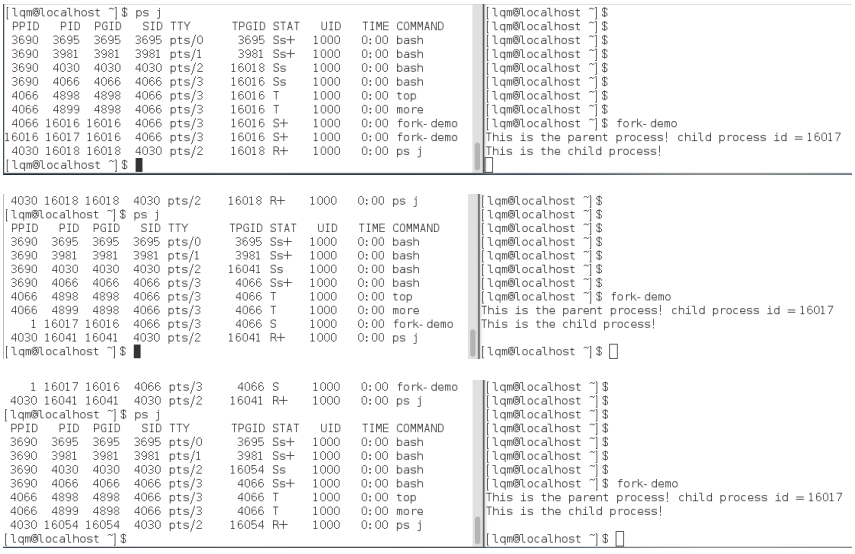

图 2-5 孤儿进程的示例

下面用代码 2-4 所示的 zombie-demo.c 代码展示僵尸进程的问题。子进程打印一行提示信息后将退出，但是此时父进程正在 sleep()还未准备处理字进程的 PCB 等遗留的数据对象，因此子进程处于僵尸状态。

代码 2-4 zombie-demo.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
    pid_t pid;
    pid = fork();
    if(pid < 0)
        printf("error occurred!\n");
    else if(pid == 0) {
        printf("Hi father! I'm a ZOMBIE\n");
        exit(0);      //no one waits for this process.
    }
    else {
        sleep(10);
        wait(NULL);   //the zombie process will be reaped now.
    }
}
```

编译运行 zombie-demo 程序，同时在另一终端上执行 ps j 命令可以看到字进程处于“Z+”状态（表明是僵尸状态，<defunct>），如图 2-6 上半部分所示。

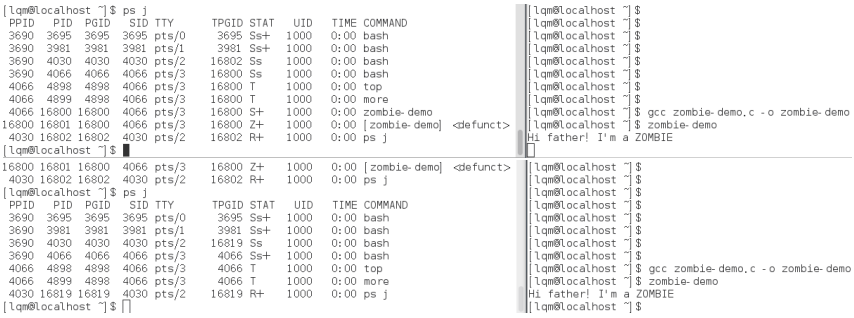

图 2-6 僵尸进程示例

在运行 10 秒之后，父进程将执行 wait()处理字进程的遗留数据对象，子进程将从僵尸状态转为完全消失状态，如图 2-6 下半部分所示。在父进程未执行 wait()之前，子进程的遗留数据空间（大致是一个 task_struct 结构体加上内核堆栈）将无法回收，如果这样的僵尸进程数量很大，将消耗很大的内存资源。

2.2.3. exec 函数族

注意到前面用 fork()创建的子进程是和父进程相同的，而实际上我们经常创建子进程已执行不同的任务从更具普遍意义，否则要以对 shell 程序干什么？fork()是通过将父进程复制一份而创建的子进程，exec 函数族的函数则可以用新的可执行程序的内容来更新进程影像，从而完成“变身”操作。在 Linux 中， exec 函数族包含一组函数，一共有 6 个，分别是：

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
```

```
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

其中只有 execve 是真正意义上的系统调用，其它都是在此基础上经过包装的库函数。下面我们在 fork-demo.c 的基础之上来编写一个 fork-exec-demo.c 如代码 2-5 所示。

代码 2-5 fork-exec-demo.c

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char ** argv )
{
        int pid = fork();
        if(pid == -1 ) {
                printf("error!\n");
        } else if( pid ==0 ) {
                printf("This is the child process!\n");

                char *argv[ ]={"ls", "-al", "/etc/passwd", NULL};
                char *envp[ ]={"PATH=/usr/bin", NULL};
                execve("/usr/bin/ls", argv, envp);


                printf("this printf()will not be exec,because it will be replaced by
/usr/bin/ls code!\n");


        } else {
                printf("This is the parent process! child process id = %d\n", pid);
                getchar();
        }
        return 0;
}
```

execve()函数的第一个参数就是想要变成的“新”进程影像（可执行文件），第二个参数是命令行参数，第三个变量是环境变量。这里可以看到字进程通过 execve()将自己变身为“/usr/bin/ls”，因此不再执行与父进程里的代码——即后面的“printf("this printf()will not be exec,because …”是没有机会得到执行的。真是的运行结果如所示，可以看到字进程变成了 ls。

屏显 2-10 fork-exec-demo 的执行结果

```
[lqm@localhost ~]$ gcc fork-exec-demo.c -o fork-exec-demo
[lqm@localhost ~]$ fork-exec-demo
This is the parent process! child process id = 5819
This is the child process!
-rw-r--r--. 1 root root 2274 Mar 12 16:45 /etc/passwd

[lqm@localhost ~]$
```

我们在 shell 命令提示符下键入 HelloWorld-getchar 并回车后，shell 首先是 fork()创建出一个字进程来（进程映像仍是 shell），然后再利用 exec 函数族的函数变身为 HelloWorld-getchar。fork()和 exec 函数族构成了创建新进程的完整操作，如果父子进程功能相同则只用 fork()如果父子进程不同则利用 exec 函数族的来实现。

2.2.4. 通过 kill()撤销进程

前面我们通过 kill 命令通过发送信号来撤销进程，现在来看看等效的 kill()函数如何撤销进程。该函数及其参数需要头文件有如下：

```
#include <sys/types.h>
#include <signal.h>
```

函数原型为“int kill(pid_t pid, int sig);”，第一个参数是目标进程的 PID，第二个参数是发送的信号。另有 raise()函数用于自己发送信号，相当于 kill 给自己发信号 kill(getpid(), *SIGNAL*)[2]。如果 kill()或 raise()的信号取值为 SIGKILL 可用于杀死进程，更多细节参见信号章节。

2.2.5. 创建守护进程

本学期先不做。

http://blog.chinaunix.net/uid-25365622-id-3055635.html

批注 [PR4]: 需要扩充完善

2.3. 创建 pthread 线程

操作系统课程中指出进程既是调度执行的基本单位，由是资源分配基本单位。考虑一下一个场景：一个网络服务器程序以N个独立进程形式服务于N个客户端，此时各个服务器进程实体是长得一抹一样，具有的功能也一模一样，但是各自运行的执行流（指令执行）却是各异的——有的已经将响应返回给客户端、有的还在连接建立阶段、有的正在处理请求等等。

而线程的引入，则希望减小上面这种应用情形下资源开销——这些 N 个进程既然是一样的，那么共享就好了。线程概念的引入后，将“调度执行”与“资源分配”这两个属性做了分割，上述的 N 个执行流共用一套资源（进程实体），但是拥有各自不同的执行流。

最主要的支撑技术就是如何在同一套资源（特别是进程代码空间）上，区分各自不同的执行流——当它们被调度[3]中断时用于保留现场，以便下次被调度执行时能无缝地从上次端点处继续运行。不同的执行流首先需要具有不同的程序指针（IP），其次在函数调用上要记录各自不同的函数调用栈。各个执行流不能光靠 IP 来区分，否则两个执行流以不同调用进入到同一个函数 f()代码处，在 f()返回时必须知道是从那个上级函数调用进来的——这些信息是用堆栈来记录。

因此，在相同的进程映像等基本资源之上，各个线程保留少量资源：各自的执行流的 IP（当然包括其他 CPU 现场）、各自的用户态堆栈（记录者该执行流的函数调用层次关系等）。这些 IP 等 CPU 现场和用户态堆栈等信息可以简单笼统地认为记录在进程控制块内，对于 Linux 而言就是用 struct task_struct 结构体来统一管理本线程的相关资源。

[2] 函数 getpid()用于获取自己的 PID 编号。

[3] 由于还未讨论调度，如果读者感到阅读有困难，请先学习后面调度知识再回过头来再次阅读。

2.3.1. 进程与线程

系统同时运行多个进程，这些进程可能基于相同的可执行文件而创建，也可能使用不同的可执行文件来创建，另外每个进程有可能会创建多个线程。下面我们就来绘制这些多进程、多线程并发执行时的情形。

图 2-7 表示了多进程、多线程并发的示意图，注意其中黄色的内核空间是相同、共用的（虽然每个进程都绘制了一个内核空间）。

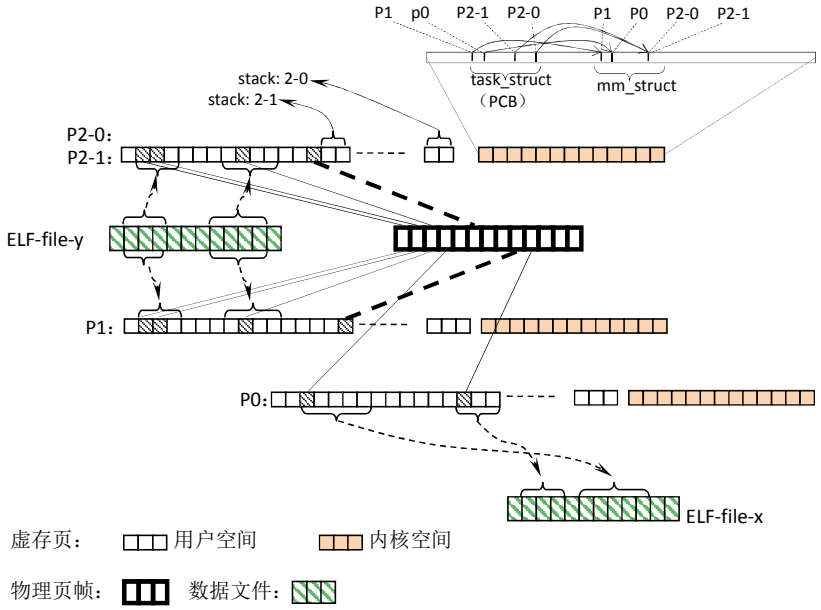

图 2-7 多进程、多线程并发的地址空间

图 2-7 中展示了由 ELF-file-x 可执行程序创建的进程 P0，以及 ELF-file-y 可执行程序创建 P1 和 P2，其中 P2 共有两个线程。可以看出 P0 的虚存空间是完全独立的，而 P1 和 P2 进程的虚存空间虽然独立，但是由于映射了相同的 ELF-file-y 代码段和部分数据段（细实线），因此它们的一些虚存页面映射到了相同的物理页帧上，只有可写数据各自有不同的物理映射（粗虚线）。而 P2 进程的两个线程 P2-0 和 P2-1 则共享一个空间、共用一套页表，但是各自有不同的进程控制块和线程堆栈（除了主线程外，其他线程的栈建立在进程的堆空间上）。

每个进程或线程在内核空间中都会有自己的进程控制块 task_struct，因此在图 2-7 的右上角我们绘制了内核空间里的各个进程自己的 task_struct 示意图。但是只有进程才会有独立的内存空间，即用 mm_struct 描述的进程内存布局和相应的页表，因此 P2-0 和 P2-1 共享同一个 mm_struct。它们虽然有独立的 task_struct，但是双方的 task_struct 的 mm 成员指向同一个内存描述符。由于各个进程的内核空间是一样的，因此没必要在每个进程的内核空间范围内反复标注相同的上述内容。需要注意各个进程的内核空间部分是相同的，因此任何一个进程进入它

的内核空间，都可以访问到其他进程的进程控制块 task_strcut 和内存描述符 mm_struct，图 2-7 只在一个进程的内核空间作了描绘。由于这些数据结构比起页的尺寸来说很小，因此图中只用一根竖线表示所占用的空间。

从上述示意图可以看出，创建进程和线程的开销并不相同。无论是进程还是线程，都必须有进程控制块 PCB（struct task_struct），但是线程间共享进程空间因此不需要独立的内存描述符（struct mm_struct）以及所辖的内存区域描述符等，也不需要独立的文件描述符或其他资源。

2.3.2. 创建方法

下面我们尝试使用 pthread 库来创建线程，我们编写代码 2-6 的 pthread-demo.c 代码，然后用“gcc –lpthread pthread-demo.c –o pthread-demo”完成编译并输出 pthread-demo 可执行文件，注意其中的参数-lpthread 表示编译链接时需要用到 pthread 库。其中函数 pthread_create()用于创建线程，第三个参数是一个函数指针——用于指定新创建线程将执行哪个函数，第一个参数用于记录新创建进程的标识 ID。主线程使用 pthread_join(id,NULL)等待指定线程（以 id 为标识的线程，也就是前面刚创建的线程）结束。

代码 2-6 pthread-demo.c 线程创建示例代码

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void thread(void)
{
        printf("This is a pthread.\n");
        sleep(10);
}

int main(void)
{
        pthread_t id;
        int i,ret;
        ret=pthread_create(&id,NULL,(void *) thread,NULL);
        if(ret!=0){
                printf ("Create pthread error!\n");
                exit (1);
        }

        printf("This is the main process.\n");
        pthread_join(id,NULL);
        return (0);
}
```

运行 pthread-demo 将看到两个线程都在执行，各自输出一行信息，等待 10 秒后两个线程都将自动结束，如屏显 2-11 所示。

屏显 2-11 pthread-demo 的输出

```
[root@localhost lqm]# pthread-demo
This is the main process.
This is a pthread.
[root@localhost lqm]#
```

在 pthread-demo 为结束前，可在另一个终端窗口执行 ps -aLf 察看信息（其中参数-L 表示察看 LWP[4]），可以获得屏显 2-12 所示的输出（NLWP 列表示 Number of LWP，即统一进程的线程数）。

屏显 2-12 pthread-demo 运行时执行 ps –aLf 的输出

```
[lqm@localhost ~]$ ps -aLf
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
lqm       5497  4296  5497  0    2 16:24 pts/1    00:00:00 pthread-demo
lqm       5497  4296  5498  0    2 16:24 pts/1    00:00:00 pthread-demo
lqm       5499  3814  5499  0    1 16:24 pts/0    00:00:00 ps -aLf
[lqm@localhost ~]$
```

可以看出 pthread-demo 进程号为 5497，NLWP 列指出他又两个线程，这两个线程各自的线程号由 LWP 列指出——分别为 5497 和 5498。

2.4. 进程/线程资源开销

为了对比进程和线程的资源开销，让读者形成相应的直观认识。我们来编写两个程序 fork-100-demo.c 和 pthread-100-demo.c，分别创建 100 个进程和 100 个线程。然后用 Linux 的 /proc/slabinfo 中给出的信息来简单比较各自在进程控制块和内存描述符上的资源开销。

2.4.1. PCB 开销

代码 2-7 用 100 次循环创建了 100 个子进程，然后各个子进程睡眠 10 秒，最后全部正常退出。

代码 2-7 fork-100-demo.c 代码

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char ** argv )
{
        int i;
        for(i=0;i<100;i++){
                int pid = fork();
                if(pid == -1 ) {
                        printf("error!\n");
                } else if( pid ==0 ) {
                        printf("This is the child process!\n");
                        sleep(10);
                        return 0;
                } else {
                        printf("This is the parent process! child process id = %d\n",
pid);
                }
        }
        sleep(10);
        return 0;
}
```

[4] Lightweight Process 的缩写，Linux 用 LWP 指代线程

代码 2-8 也是用 100 次循环创建 100 个线程，注意与 ptread-demo.c 在线程 id 存储上区别，这里使用了数组。

代码 2-8 pthread-100-demo.c 代码

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void thread(void)
{
        printf("This is a pthread.\n");
        sleep(10);
}

int main(void)
{
        pthread_t id[100];
        int i,ret;
        for(i=0;i<100;i++){
                ret=pthread_create(&id[i],NULL,(void *) thread,NULL);
                if(ret!=0){
                        printf ("Create pthread error!\n");
                        exit (1);
                }
        }

        printf("This is the main process.\n");
        for(i=0;i<100;i++){
                pthread_join(id[i],NULL);
        }
        return (0);
}
```

我们先比较两者在进程控制块上的开销（包括 struct task_struc 和内核栈）。由于 Linux 中的进程动态的创建和撤销，因此在数量上并不十分准确，但是大体上能看出变化的趋势。/proc/slabinfo 中统计了各种内核数据对象的数量，其中也包括进程控制块的数量。在 fork-100-demo 运行前、运行中和结束后各执运行一次 cat /proc/slabinfo |grep task_struct，得到屏显 2-13 所示的输出。可以看出，在 fork-100-demo 运行时，比运行前后都大约多了 100 个 task_struct 的数据对象（由于有进程动态创建与撤销，因此数据有扰动）。

屏显 2-13 fork-100-demo 运行前后的/proc/slabinfo 中关于 task_strcut 数量

```
[root@localhost lqm]# cat /proc/slabinfo |grep task_struct
task_struct         371    432   4016    8    8 : tunables    0    0    0 : slabdata     54     54      0
[root@localhost lqm]# cat /proc/slabinfo |grep task_struct
task_struct         472    472   4016    8    8 : tunables    0    0    0 : slabdata     59     59      0
[root@localhost lqm]# cat /proc/slabinfo |grep task_struct
task_struct         367    432   4016    8    8 : tunables    0    0    0 : slabdata     54     54      0
[root@localhost lqm]#
```

下面也来看看 pthread-100-demo 运行前、运行中和结束后的 task_struct 数量变化。可以看出，在 pthread-100-demo 运行时，比运行前后都大约多了 100 个 task_struct 的数据对象（资源）。

屏显 2-14 pthread-100-demo 运行前后的/proc/slabinfo 中关于 task_strcut 数量

```
[root@localhost lqm]# cat /proc/slabinfo |grep task_struct
task_struct         367    432   4016    8    8 : tunables    0    0    0 : slabdata     54     54      0
[root@localhost lqm]# cat /proc/slabinfo |grep task_struct
```

```
task_struct          472    472   4016    8    8 : tunables    0    0    0 : slabdata     59     59      0
[root@localhost lqm]# cat /proc/slabinfo |grep task_struct
task_struct          368    432   4016    8    8 : tunables    0    0    0 : slabdata     54     54      0
[root@localhost lqm]#
```

也就是说，线程作为调度执行单位，进程控制块 PCB（task_struct）还是需要的，这是最小资源的一部分。因此这方面资源开销对进程和线程都是一样的。

2.4.2. 内存描述符开销

前面提到了属于同一进程的线程间共享进程的内存空间，因此共用一个内存描述符 mm_struct。因此，创建 100 个进程也将需要创建 100 个 mm_struct，但是在进程的主线程存在的前提下创建 100 个线程则不需要新创建任何 mm_struct。下面我们来验证上述论断，使用 cat /proc/slabinfo |grep mm_struct 命令来检查 fork-100-demo/pthread-100-demo 运行前、运行中和结束后的 mm_struct 数量。对于创建进程的 fork-100-demo，其结果如屏显 2-15 所示。

屏显 2-15 fork-100-demo 运行前后的/proc/slabinfo 中关于 mm_strcut 数量

```
[root@localhost lqm]# cat /proc/slabinfo |grep mm_struct
mm_struct            119    140   1600   10    4 : tunables    0    0    0 : slabdata     14     14      0
[root@localhost lqm]# cat /proc/slabinfo |grep mm_struct
mm_struct            210    210   1600   10    4 : tunables    0    0    0 : slabdata     21     21      0
[root@localhost lqm]# cat /proc/slabinfo |grep mm_struct
mm_struct            120    150   1600   10    4 : tunables    0    0    0 : slabdata     15     15      0
[root@localhost lqm]#
```

对于创建进程的 pthread-100-demo，其结果如所示。

屏显 2-16 pthread-100-demo 运行前后的/proc/slabinfo 中关于 mm_strcut 数量

```
[root@localhost lqm]# cat /proc/slabinfo |grep mm_struct
mm_struct            119    140   1600   10    4 : tunables    0    0    0 : slabdata     14     14      0
[root@localhost lqm]# cat /proc/slabinfo |grep mm_struct
mm_struct            119    140   1600   10    4 : tunables    0    0    0 : slabdata     14     14      0
 [root@localhost lqm]# cat /proc/slabinfo |grep mm_struct
mm_struct            119    140   1600   10    4 : tunables    0    0    0 : slabdata     14     14      0
```

这里可以看出来，pthread-100-demo 虽然创建了 100 个线程，但是用于共享进程空间，并没有增加一个 mm_struct。不过由于每个进程需要一个独立的用户台堆栈，这些堆栈需要一个独立的 vm_are_struct 结构体来描述（一个 vm_area_struct 结构体用于描述进程内部的连续、相同属性的内存区），因此将会增加 100 个 vm_area_struct 用于描述线程用户态栈。但是 100 个进程的话就更多，每个简单的进程至少需要大约 15 个 vm_area_struct 结构体，100 个进程需要增加大约 1500 个 vm_area_struct。用 cat /proc/slabinfo |grep vm_area_struct 观察 fork-100-demo，可以发现新增大约 15xx 左右的使用量，如屏显 2-17 所示。

屏显 2-17 fork-100-demo 运行前后的/proc/slabinfo 中关于 vm_area_strcut 数量

```
vm_area_struct     20666  20934    216   18    1 : tunables    0    0    0 : slabdata   1163   1163      0
[root@localhost lqm]# cat /proc/slabinfo |grep vm_area_struct
vm_area_struct     22176  22176    216   18    1 : tunables    0    0    0 : slabdata   1232   1232      0
[root@localhost lqm]# cat /proc/slabinfo |grep vm_area_struct
vm_area_struct     20558  20970    216   18    1 : tunables    0    0    0 : slabdata   1165   1165      0
```

然后对 thread-100-demo 的运行前、运行时和结束后，用用 cat /proc/slabinfo |grep vm_area_struct 观察，可以发现之需要新增大约 1xx 左右的使用量，如屏显 2-18 所示。

屏显 2-18 pthread-100-demo 运行前后的/proc/slabinfo 中关于 vm_area_strcut 数量

```
[root@localhost lqm]# cat /proc/slabinfo |grep vm_area_struct
vm_area_struct     20667  20934    216   18    1 : tunables    0    0    0 : slabdata   1163   1163      0
[root@localhost lqm]# cat /proc/slabinfo |grep vm_area_struct
vm_area_struct     20778  20934    216   18    1 : tunables    0    0    0 : slabdata   1163   1163      0
[root@localhost lqm]# cat /proc/slabinfo |grep vm_area_struct
vm_area_struct     20583  20934    216   18    1 : tunables    0    0    0 : slabdata   1163   1163      0
[root@localhost lqm]#
```

从我们的实验可以看出，线程的开销确实比进程要小。如果读者仅需观察，例如观察文件描述符的使用量，通过 cat /proc/files_cache 和 cat /proc/signals_cache 也能发现线程的开销比进程小的多。

2.5. 小结

本章让读者了解进程的基本概念和进程的组织关系，并通过编程实验操作来完成进程的创建、撤销等操作。应该对 Linux 进程创建工作分成 fork()和 exec 两部的过程也有清晰认识。其中还对对孤儿进程和僵尸进程的产生原因及编程问题作了分析。最后给大家展示进程和线程在资源开销上的差异，并通过/proc/slabinfo 观察到直观的结果。

练习

尝试自行设计一个 C 语言小程序，完成最基本的 shell 角色；尝试将你的 shell 增加管道功能