链滴

linux 进程管理

作者：jared

原文链接：https://ld246.com/article/1531210463046

来源网站：链滴

linux进程管理

- 什么是程序，什么是进程
- 进程的创建
  - 让程序在后台运行
  - 查看进程ID
  - 查看进程的内存映象
- 查看进程的属性及状态
  - ps 命令查看进程属性
  - 通过pstree查看进程亲缘关系
  - 用top命令动态查看进程信息
  - 确保特定程序只有一个副本
- 调整进程的优先级
  - 获取进程的优先级
  - 调整进程的优先级
- 结束进程
  - 结束进程
  - 暂停某个进程
  - 查看进程的退出状态
- 进程通信
  - 无名管道
  - 有名管道
  - 信号
- 作业和作业控制
  - 创建后台进程，获取进程的作业号和进程号
  - 把作业调到前台并暂停
  - 查看当前作业情况
  - 启动停止的进程并运行在后台

什么是程序，什么是进程

程序是指令的集合，而进程则是程序执行的基本单元。为了让程序完成它的工作，必须让程序运行起成为进程，进而利用处理器资源、内存资源，进行各种 I/O 操作，从而完成某项特定工作。

从这个意思上说，程序是静态的，而进程则是动态的。

进程有区别于程序的地方还有：进程除了包含程序文件中的指令数据以外，还需要在内核中有一个数

结构用以存放特定进程的相关属性，以便内核更好地管理和调度进程，从而完成多进程协作的任务。此，从这个意义上可以说“高于”程序，超出了程序指令本身。

如果进行过多进程程序的开发，又会发现，一个程序可能创建多个进程，通过多个进程的交互完成任在 Linux 下，多进程的创建通常是通过 fork 系统调用来实现。从这个意义上来说程序则”包含”了程。

另外一个需要明确的是，程序可以由多种不同程序语言描述，包括 C 语言程序、汇编语言程序和最后译产生的机器指令等。

下面简单讨论 Linux 下面如何通过 Shell 进行进程的相关操作。

进程的创建

通常在命令行键入某个程序文件名以后，一个进程就被创建了。例如，

让程序在后台运行

```
$ sleep 100 &
[1] 9298
```

查看进程ID

用 pidof 可以查看指定程序名的进程ID：

```
$ pidof sleep
9298
```

查看进程的内存映像

```
$ cat /proc/9298/maps
08048000-0804b000 r-xp 00000000 08:01 977399 /bin/sleep
0804b000-0804c000 rw-p 00003000 08:01 977399 /bin/sleep
0804c000-0806d000 rw-p 0804c000 00:00 0 [heap]
b7c8b000-b7cca000 r--p 00000000 08:01 443354
...
bfbd8000-bfbed000 rw-p bfbd8000 00:00 0 [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0 [vdso]
```

程序被执行后，就被加载到内存中，成为了一个进程。上面显示了该进程的内存映像（虚拟内存），括程序指令、数据，以及一些用于存放程序命令行参数、环境变量的栈空间，用于动态内存申请的堆间都被分配好。

实际上，创建一个进程，也就是说让程序运行，还有其他的办法，比如，通过一些配置让系统启动时动启动程序（具体参考 man init），或者是通过配置 crond （或者 at） 让它定时启动程序。除此外，还有一个方式，那就是编写 Shell 脚本，把程序写入一个脚本文件，当执行脚本文件时，文件中程序将被执行而成为进程。这些方式的细节就不介绍，下面了解如何查看进程的属性。

需要补充一点的是：在命令行下执行程序，可以通过 ulimit 内置命令来设置进程可以利用的资源，比如进程可以打开的最大文件描述符个数，最大的栈空间，虚拟内存空间等。具体用法见 help ulimit 。

查看进程的属性和状态

可以通过 ps 命令查看进程相关属性和状态，这些信息包括进程所属用户，进程对应的程序，进程对 cu 和内存的使用情况等信息。熟悉如何查看它们有助于进行相关的统计分析等操作

通过 ps 命令查看进程属性

查看系统当前所有进程的属性：

```
$ ps -ef / ps aux
```

查看命令中包含某字符的程序对应的进程，进程 ID 是 1 。 TTY 为？表示和终端没有关联：

```
$ ps -C init
PID TTY TIME CMD
1 ? 00:00:01 init
```

选择某个特定用户启动的进程：

```
$ ps -U falcon
```

按照指定格式输出指定内容，下面输出命令名和 cpu 使用率：

```
$ ps -e -o "%C %c"
```

打印 cpu 使用率最高的前 4 个程序：

```
$ ps -e -o "%C %c" | sort -u -k1 -r | head -5
7.5 firefox-bin
1.1 Xorg
0.8 scim-panel-gtk
0.2 scim-bridge
```

获取使用虚拟内存最大的 5 个进程：

```
$ ps -e -o "%z %c" | sort -n -k1 -r | head -5
349588 firefox-bin
96612 xfce4-terminal
88840 xfdesktop
76332 gedit
58920 scim-panel-gtk
```

通过 pstree 查看进程亲缘关系

系统所有进程之间都有“亲缘”关系，可以通过 pstree 查看这种关系：

```
$ pstree
```

用top动态查看进程信息

```
$ top
```

该命令最大特点是可以动态地查看进程信息，当然，它还提供了一些其他的参数，比如 -S可以按照累

执行时间的大小排序查看，也可以通过 -u 查看指定用户启动的进程等。补充： top 命令支持交互式比如它支持 u 命令显示用户的所有进程，支持通过 k 命令杀掉某个进程；如果使用 -n 1 选项可以启批处理模式，具体用法为：

```
$ top -n 1 -b
```

确保特定程序只有一个副本在运行

下面来讨论一个有趣的问题：如何让一个程序在同一时间只有一个在运行。

这意味着当一个程序正在被执行时，它将不能再被启动。那该怎么做呢？

假如一份相同的程序被复制成了很多份，并且具有不同的文件名被放在不同的位置，这个将比较糟糕所以考虑最简单的情况，那就是这份程序在整个系统上是唯一的，而且名字也是唯一的。这样的话，哪些办法来回答上面的问题呢？

总的机理是：在程序开头检查自己有没有执行，如果执行了则停止否则继续执行后续代码。

策略则是多样的，由于前面的假设已经保证程序文件名和代码的唯一性，所以通过 ps 命令找出当前有进程对应的程序名，逐个与自己的程序名比较，如果已经有，那么说明自己已经运行了。

```
ps -e -o "%c" | tr -d " " | grep -q ^init$ #查看当前程序是否执行
[ $? -eq 0 ] && exit #如果在，那么退出, $?表示上一条指令是否执行成功
```

每次运行时先在指定位置检查是否存在一个保存自己进程 ID 的文件，如果不存在，那么继续执行，果存在，那么查看该进程 ID 是否正在运行，如果在，那么退出，否则往该文件重新写入新的进程 ID 并继续。

```
pidfile=/tmp/$0".pid"
if [ -f $pidfile ]; then
OLDPID=$(cat $pidfile)
ps -e -o "%p" | tr -d " " | grep -q "^$OLDPID$"
[ $? -eq 0 ] && exit
fi
echo $$ > $pidfile
#... 代码主体
#设置信号0的动作，当程序退出时触发该信号从而删除掉临时文件
trap "rm $pidfile" 0
```

调整进程的优先级

在保证每个进程都能够顺利执行外，为了让某些任务优先完成，那么系统在进行进程调度时就会采用定的调度办法，比如常见的有按照优先级的时间片轮转的调度算法。这种情况下，可以通过 renice 整正在运行的程序的优先级，例如：

获取进程优先级

```
$ ps -e -o "%p %c %n" | grep xfs
5089 xfs
```

调整进程的优先级

```
$ renice 1 -p 5089
renice: 5089: setpriority: Operation not permitted
$ sudo renice 1 -p 5089 #需要权限才行
[sudo] password for falcon:
5089: old priority 0, new priority 1
$ ps -e -o "%p %c %n" | grep xfs #再看看，优先级已经被调整过来了
5089 xfs
```

结束进程

既然可以通过命令行执行程序，创建进程，那么也有办法结束它。可以通过kill命令给用户自己启动的程发送某个信号让进程终止，当然“万能”的root几乎可以 kill 所有进程（除了 init 之外） 。例如，

结束进程

```
$ sleep 50 & #启动一个进程
[1] 11347
$ kill 11347
```

kill 命令默认会发送终止信号（ SIGTERM ） 给程序，让程序退出，但是 kill 还可以发送其他信号，些信号的定义可以通过 man 7 signal 查看到，也可以通过 kill -l 列出来。

```
$ man 7 signal
$ kill -l
1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL
5) SIGTRAP 6) SIGABRT 7) SIGBUS 8) SIGFPE
9) SIGKILL 10) SIGUSR1 11) SIGSEGV 12) SIGUSR2
13) SIGPIPE 14) SIGALRM 15) SIGTERM 16) SIGSTKFLT
17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU
25) SIGXFSZ 26) SIGVTALRM 27) SIGPROF 28) SIGWINCH
29) SIGIO 30) SIGPWR 31) SIGSYS 34) SIGRTMIN
35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3 38) SIGRTMIN+4
39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7 58) SIGRTMAX-6
59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

暂停某个进程

例如，用kill命令发送 SIGSTOP 信号给某个程序，让它暂停，然后发送 SIGCONT 信号让它继续运行。

```
$ sleep 50 &
[1] 11441
$ jobs
[1]+ Running sleep 50 &
$ kill -s SIGSTOP 11441 #这个等同于我们对一个前台进程执行CTRL+Z操作
$ jobs
[1]+ Stopped sleep 50
```

```
$ kill -s SIGCONT 11441 #这个等同于之前我们使用bg %1操作让一个后台进程运行起来
$ jobs
[1]+ Running sleep 50 &
$ kill %1 #在当前会话(session)下，也可以通过作业号控制进程
$ jobs
[1]+ Terminated sleep 50
```

可见 kill 命令提供了非常好的功能，不过它只能根据进程的 ID 或者作业来控制进程，而pkill 和 killall 提供了更多选择，它们扩展了通过程序名甚至是进程的用户名来控制进程的方法。更多用法请参考它的手册

查看进程退出状态

当程序退出后，如何判断这个程序是正常退出还是异常退出呢？还记得 Linux 下，那个经典hello word 程序吗？在代码的最后总是有条 return 0 语句。这个return 0实际上是让程序员来检查进程是否正退出的。如果进程返回了一个其他的数值，那么可以肯定地说这个进程异常退出了，因为它都没有执到 return 0 这条语句就退出了。

那怎么检查进程退出的状态，即那个返回的数值呢？

在 Shell 中，可以检查这个特殊的变量 $? ，它存放了上一条命令执行后的退出状态。

```
$ test1
bash: test1: command not found
$ echo $?
127
$ cat ./test.c | grep hello
$ echo $?
1 $
cat ./test.c | grep hi
printf("hi, myself!\n");
$ echo $?
0
```

貌似返回 0 成为了一个潜规则，虽然没有标准明确规定，不过当程序正常返回时，总是可以从 $? 中测到 0，但是异常时，总是检测到一个非 0 值。这就告诉我们在程序的最后最好是跟上一个exit 0以任何人都可以通过检测 $? 确定程序是否正常结束。如果有一天，有人偶尔用到你的程序，试图检查的退出状态，而你却在程序的末尾莫名地返回了一个-1 或者 1，那么他将会很苦恼，会怀疑他自己编的程序到底哪个地方出了问题，检查半天却不知所措，因为他太信任你了，竟然从头至尾都没有怀疑的编程习惯可能会与众不同！

进程通信

为便于设计和实现，通常一个大型的任务都被划分成较小的模块。不同模块之间启动后成为进程，它之间如何通信以便交互数据，协同工作呢？在《UNIX 环境高级编程》 一书中提到很多方法，诸如管（无名管道和有名管道） 、信号（signal ） 、报文（Message ） 队列（消息队列） 、共享内存（map/munmap ） 、信号量（semaphore ，主要是同步用，进程之间，进程的不同线程之间） 、接口（Socket ，支持不同机器之间的进程通信） 等，而在Shell 中，通常直接用到的就有管道和信等。下面主要介绍管道和信号机制在 Shell 编程时的一些用法。

无名管道（pipe）

在 Linux 下，可以通过 | 连接两个程序，这样就可以用它来连接后一个程序的输入和前一个程序的输

，因此被形象地叫做个管道。在 C 语言中，创建无名管道非常简单方便，用pipe 函数，传入一个具两个元素的 int 型的数组就可以。这个数组实际上保存的是两个进程操作文件描述符，父进程往第一文件描述符里头写入东西后，子进程可以从第一个文件描述符中读出来。

如果用多了命令行，这个管子 | 应该会经常用。比如上面有个演示把ps命令的输出作为grep 命令的入：

```
$ ps -ef | grep init
```

也许会觉得这个“管子”好有魔法，竟然真地能够链接两个程序的输入和输出，它们到底是怎么实现呢？实际上当输入这样一组命令时，当前Shell 会进行适当的解析，把前面一个进程的输出关联到管的输出文件描述符，把后面一个进程的输入关联到管道的输入文件描述符，这个关联过程通过输入输重定向函数 dup （或者 fcntl ） 来实现。

有名管道（named pipe)

有名管道实际上是一个文件（无名管道也像一个文件，虽然关系到两个文件描述符，不过只能一边读外一边写），不过这个文件比较特别，操作时要满足先进先出，而且，如果试图读一个没有内容的有管道，那么就会被阻塞，同样地，如果试图往一个有名管道里写东西，而当前没有程序试图读它，也被阻塞。下面看看效果。

```
$ mkfifo fifo_test #通过mkfifo命令创建一个有名管道
$ echo "fewfefe" > fifo_test
# 试图往fifo_test文件中写入内容，但是被阻塞，要另开一个终端继续下面的操作
$ cat fifo_test #另开一个终端，记得，另开一个。试图读出fifo_test的内容
fewfefe
```

这里的 echo 和 cat 是两个不同的程序，在这种情况下，通过 echo 和 cat 启动的两个进程之间并没父子关系。不过它们依然可以通过有名管道通信。

这样一种通信方式非常适合某些特定情况：例如有这样一个架构，这个架构由两个应用程序构成，其一个通过循环不断读取 fifo_test 中的内容，以便判断，它下一步要做什么。如果这个管道没有内容那么它就会被阻塞在那里，而不会因死循环而耗费资源，另外一个则作为一个控制程序不断地往 fifo_test 中写入一些控制信息，以便告诉之前的那个程序该做什么。下面写一个非常简单的例子。可以设计些控制码，然后控制程序不断地往fifo_test 里头写入，然后应用程序根据这些控制码完成不同的动作当然，也可以往fifo_test 传入除控制码外的其他数据。

- 应用程序的代码

```
$ cat app.sh
#!/bin/bash
FIFO=fifo_test
while :;
do
CI=`cat $FIFO` #CI --> Control Info
case $CI in
0) echo "The CONTROL number is ZERO, do something ..."
;;
1) echo "The CONTROL number is ONE, do something ..."
;;
*) echo "The CONTROL number not recognized, do something else..."
;;
esac
done
```

- 控制程序的代码

```
$ cat control.sh
#!/bin/bash
FIFO=fifo_test
CI=$1
[ -z "$CI" ] && echo "the control info should not be empty" && exit
echo $CI > $FIFO
```

- 一个程序通过管道控制另外一个程序的工作

```
$ chmod +x app.sh control.sh #修改这两个程序的可执行权限，以便用户可以执行它们
$ ./app.sh #在一个终端启动这个应用程序，在通过./control.sh发送控制码以后查看输出
The CONTROL number is ONE, do something ... #发送1以后
The CONTROL number is ZERO, do something ... #发送0以后
The CONTROL number not recognized, do something else... #发送一个未知的控制码以后
$ ./control.sh 1 #在另外一个终端，发送控制信息，控制应用程序的工作
$ ./control.sh 0
$ ./control.sh 4343
```

这样一种应用架构非常适合本地的多程序任务设计，如果结合 web cgi ，那么也将适合远程控制的要。引入 web cgi 的唯一改变是，要把控制程序 ./control.sh 放到 web 的 cgi目录下，并对它作一些改，以使它符合 CGI 的规范，这些规范包括文档输出格式的表示（在文件开头需要输出 content-tpye: text/html 以及一个空白行） 和输入参数的获取 (web输入参数都存放在 QUERY_STRING 环境变量头） 。因此一个非常简单的 CGI 控制程序可以

写成这样：

```
#!/bin/bash
FIFO=./fifo_test
CI=$QUERY_STRING
[ -z "$CI" ] && echo "the control info should not be empty" && exit
echo -e "content-type: text/html\n\n"
echo $CI > $FIFO
```

在实际使用时，请确保 control.sh 能够访问到 fifo_test 管道，并且有写权限，以便通过浏览器控制 app.sh ：

```
http://ipaddress\_or\_dns/cgi-bin/control.sh?0
```

问号？后面的内容即 QUERY_STRING，类似之前的 $1 。

这样一种应用对于远程控制，特别是嵌入式系统的远程控制很有实际意义。在去年的暑期课程上，我就通过这样一种方式来实现马达的远程控制。首先，实现了一个简单的应用程序以便控制马达的转动包括转速，方向等的控制。为了实现远程控制，我们设计了一些控制码，以便控制马达转动相关的不属性。

在 C 语言中，如果要使用有名管道，和 Shell 类似，只不过在读写数据时用 read ， write调用，在建 fifo 时用 mkfifo 函数调用。

信号（Signal）

信号是软件中断，**Linux** 用户可以通过kill命令给某个进程发送一个特定的信号，也可以通过键盘发送些信号，比如 CTRL+C 可能触发 SGIINT 信号，而CTRL+\可能触发 SGIQUIT信号等，除此之外，内

在某些情况下也会给进程发送信号，比如在访问内存越界时产生SGISEGV 信号，当然，进程本身也以通过 kill ， raise 等函数给自己发送信号。对于Linux 下支持的信号类型，大家可以通过 man 7 signal 或者 kill -l 查看到相关列表和说明。

对于有些信号，进程会有默认的响应动作，而有些信号，进程可能直接会忽略，当然，用户还可以对些信号设定专门的处理函数。在 Shell 中，可以通过 trap 命令（Shell 内置命令） 来设定响应某个信的动作（某个命令或者定义的某个函数） ，而在 C 语言中可以通过signal 调用注册某个信号的处理数。这里仅仅演示trap命令的用法。

```
$ function signal_handler { echo "hello, world."; } #定义signal_handler函数
$ trap signal_handler SIGINT #执行该命令设定：收到SIGINT信号时打印hello, world
$ hello, world #按下CTRL+C，可以看到屏幕上输出了hello, world字符串
```

类似地，如果设定信号 0 的响应动作，那么就可以用trap来模拟 C 语言程序中的 atexit程序终止函数登记，即通过trap signal_handler SIGQUIT设定的 signal_handler 函数将在程序退出时执行。信号 0 是一个特别的信号，在 POSIX.1 中把信号编号 0 定义为空信号，这常被用来确定一个特定进程是否旧存在。当一个程序退出时会触发该信号。

```
$ cat sigexit.sh
#!/bin/bash
function signal_handler {
echo "hello, world"
}
trap signal_handler 0
$ chmod +x sigexit.sh
$ ./sigexit.sh #实际Shell编程会用该方式在程序退出时来做一些清理临时文件的收尾工作
hello, world
```

作业和作业控制

当我们为完成一些复杂的任务而将多个命令通过 |,\>,<, ;, (,) 等组合在一起时，通常这个命令序列会动多个进程，它们间通过管道等进行通信。而有时在执行一个任务的同时，还有其他的任务需要处理那么就经常会在命令序列的最后加上一个&，或者在执行命令后，按下 CTRL+Z 让前一个命令暂停。便做其他的任务。等做完其他一些任务以后，再通过 fg命令把后台任务切换到前台。这样一种控制过通常被成为作业控制，而那些命令序列则被成为作业，这个作业可能涉及一个或者多个程序，一个或多个进程。下面演示一下几个常用的作业控制操作。

创建后台进程，获取进程的作业号和进程号

```
$ sleep 50 &
[1] 11137
```

把作业调到前台并暂停

使用 Shell 内置命令 fg 把作业 1 调到前台运行，然后按下 CTRL+Z 让该进程暂停

```
$ fg %1
sleep 50
^Z
[1]+ Stopped sleep 50
```

查看当前作业情况

```
$ jobs #查看当前作业情况，有一个作业停止
[1]+ Stopped sleep 50
$ sleep 100 & #让另外一个作业在后台运行
[2] 11138
$ jobs #查看当前作业情况，一个正在运行，一个停止
[1]+ Stopped sleep 50
[2]- Running sleep 100 &
```

启动停止的进程并运行在后台

```
$ bg %1
[2]+ sleep 50 &
```

不过，要在命令行下使用作业控制，需要当前 **Shell**，内核终端驱动等对作业控制支持才行。