链滴

Linux 中的 session 通俗讲解及使用

作者：ReyRen

原文链接：https://ld246.com/article/1565156667679

来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20180924.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
在我们进入 linux 对 session 的实现之前, 先来看看 Unix 中对线程, 进程, 进程组和会话的描述:<
p>

一个会话包含一些列的进程组, 一个进程组包含一系列的进程, 一个进程包含一些列的
程.
一个 session 可以有一个控制终端, 通俗意义讲我们 ssh 远程连接到终端, 其实就是创
了一个终端. 但是一个 session 中最多只能有一个进程组在前台运行, 也就是连接上终端后对你的终端
入输出进行管控的那个进程组. 但我们在终端敲入一个中断信号的符号时, e.g. Ctrl+c, 这个中断信号
会发送给这个进程组的所有进程成员.
所有这些都是有自己的 ID 的, 线程 ID, 进程 ID, 进程组 ID, 会话 ID

<h3 id="进程">进程</h3>
一个进程的创建通常来说是从 <code>fork()</code> 开始的:
<pre><code class="language-c highlight-chroma"><span clas
="highlight-cl">pid_t p</spa
>;
<span class="high
ight-n">p = fork</s
an>();
<span class="high
ight-k">if (p
== (<span class="hi
hlight-n">pid_t) -</
pan>1)
  <span class=
highlight-cm">/* ERROR */
<span class="high
ight-k">else if (</sp
n>p == <span class
"highlight-mi">0)
  <span class=
highlight-cm">/* CHILD */
<span class="high
ight-k">else
  <span class=
highlight-cm">/* PARENT */
</code></pre>
简单的看就是进程的复制. 子进程通俗地讲和父进程很多方面都是相通的. 但是 <code>fork()</c
de> 给子进程返回的是 0, 给父进程返回的是子进程的 pid. 当然在子进程中也可以 <code>getppid()
/code> 来进行获取它的 pid.
通常终止一个进程调用:
<pre><code class="language-c highlight-chroma"><span clas
="highlight-cl">exit(</span
n);
</code></pre>
或者
<pre><code class="language-c highlight-chroma"><span clas
="highlight-cl">return n</sp
n>;
</code></pre>
它会给父进程返回一个 n 字节的信号. 一个进程如果终止了, 但是没有被 <code>wait</code>,
么就会成为僵尸进程. 当然如果父进程早于子进程终止, 那么 1 号进程 <code>init</code> 会接手

且会成为子进程的父进程.
针对于信号的状态有 Stopping, Continuing 和 Terminating. <code>SIGSTOP</code> 就是一种制让进程停止的信号. <code>SIGCONT</code> 就是继续一个之前停止的进程. 对于终止的也是频可见的, 比如 <code>SIGKILL</code> 直接让挂掉. <code>SIGTERM</code> 大多数情况下, 进收到这个信号会先释放自己的资源然后退出, 但是也有资源在收到这个信号后会会做一些事情, 这些事是可以配置的. 也就是说 SIGTERM 信号多数情况下会阻塞, 忽略. 还有我们经常用的 <code>SIGINT<code> (^C)`. 许多信号都有默认的动作来对目标进程的终止, 但是一般都会对应一个 core 文件. 信号是个大拿, 需要好好的学习研究. 在今后或许会出信号专题. 现在高优先级的文章有点多. :cold_swet:
<h3 id="进程组">进程组</h3>
每一个进程都隶属于独一无二的进程组中. 当子进程创建后, 与父进程就同属于一个进程组中, 进组用进程组 ID 进行识别. 通常意义来说, 进程组 ID 就是这个组形成的时候的第一个进程的 ID, 这个进也就称为了进程组爸爸. 一个进程获取当前所在进程组 ID 的方式是 <code>getpgrp()</code> 或是 <code>getpgid(0)</code>. 查看进程 p 的进程组 ID 是 <code>getpgid(p)</code>.
有时候会使用 <code>ps j</code> 来查看进程的 PPID (parent process ID), PID (process ID), PGD (process group ID) and SID (session ID). 对于一些无法进行任务控制的 shell, 比如"ash". 那么所的子进程都是在同一个 session 下, 也都在同一个进程组中. 但是对于像"bash"这种有任务控制的 shell
<pre><code class="language-bash highlight-chroma">% cat paper | ideal | pic | tbl |</pan> eqn | ditroff > out
</code></pre>
这几个进程都是在一个进程组中的.
当我们想要讲一个进程加入到一个进程组的时候:
<pre><code class="language-c highlight-chroma">setpgid(</spn>pid, <span class=highlight-n">pgid);
</code></pre>
如果 <code>pgid == pid</code> 或者 <code>pgid == 0</code>, 那么这个函数就是创建个 pid 为爸爸的进程组. 否则的话就讲 pid 加入到 gpid 的进程组中. 如果 pid 是 0, 表示的就是当前程. <code>setpgrp()</code> 等价于 <code>setpgid(0,0)</code>
但是 <code>setpgid</code> 也是有一些限制就是 pid 必须是当前进程或者是当前进程的父进程. 果是父进程调用(加入), 那么必须要保证在 <code>execv</code> 之前. 还有一点是加入和被加入的必须属于同一个会话之下. 还有就是如果 pid 是 session 爸爸, 那么调用这个方法就回报错.
通常的用法是:
<pre><code class="language-c highlight-chroma">p =fork();
if (p== (pid_t) -</pan>1) {
        <span class=highlight-cm">/* ERROR */
} else ifn> (p == 0) {   /* CHILD */</spa>

        <span class=
highlight-n">setpgid(<span class="highlight-mi
>0, pgid<s
an class="highlight-p">);
        <span class=
highlight-p">...
<span class="high
ight-p">} else {</sp
n>   /* PARENT */
        <span class=
highlight-n">setpgid(<span class="highlight-n"
p, pgid<sp
n class="highlight-p">);
        <span class=
highlight-p">...
<span class="high
ight-p">}
</code></pre>
这种方式的好处是, 无论是父还是子先调度, 都能达到预期.
那么如何向进程组中的所有进程发送信号呢:
<pre><code class="language-c highlight-chroma"><span clas
="highlight-cl">killpg(</spa
>pgrp, <span class=
highlight-n">sig);
<span class="high
ight-err">如何等待子进程呢, <span class="highli
ht-err">在拥有整个进程组的进程里执行:
<span class="high
ight-err">```c
<span class="high
ight-n">waitpid(0<
span>, &<span
class="highlight-n">status, <span class="highli
ht-p">...);
</code></pre>
回想文章一开头说到一个 session 的前台只有一个进程组. 这个前台绝大多数情况下就是像 tty
样的. 那么 tty 的输入和 tty 收到的信号(^C, ^Z 等产生的)发送到整个前台进程组.
一个进程是可以获取在当前 session 的前端进程组是什么的:
<pre><code class="language-c highlight-chroma"><span clas
="highlight-cl">tcgetpgrp(</
pan>fd);
</code></pre>
这里的 fd 指的是控制终端 tty, 如果没有的话, 这个函数会返回一个大于 1 的任意值, 但是这个值
不是进程组 ID。
当然能获取也能设置:
<pre><code class="language-c highlight-chroma"><span clas
="highlight-cl">tcsetpgrp(</
pan>fd,<span class=
highlight-n">pgrp);
</code></pre>
pgrp 指的就是想要放到 session 前台的进程组. 这里需要注意的是当前调用的进程必须也是在 se
sion 中的.
当然从哪里能获得 tty 的 fd? 从定义上讲 <code>/dev/tty</code> 就是指向 tty 控制终端, 这个是
全独立于标准输入输出的 (有个函数 <code>ctermid()</code> 可以获得控制终端的名字, 在 POSIX

下返回的是/dev/tty*). 那么打开这些个文件就能看到 fd 了.
在一个 session 中, 进程组不是在前台就是在后台. 因为用户是通过外设与前台进程进行交互的, 后台程就应该躲的远远的. 当后台进程要从终端读取数据时, 它就会收到一个 SIGINT 的信号, 这个信号会这个后台进程停止(不是终结), 然后任务控制 shell 发现并且告诉了用户, 然后用户就可以选择通过 <code>fg</code> 来让后台进程调度成前台进程进行运行, 并且能够读取终端的数据. 但是如果这个后台程忽略 SIGINT 信号或者阻挡 SIGINT 信号, 再或者这个进程所在的进程组是孤儿(稍后解释), 那么它用 read()去读的时候, 就会返回 EIO 错误, 并且没有信号发送. (这样做的目的就是告诉进程, 现阶段从端获取数据不可以, 如果你看不到我发的信号, 那我就个 ERROR 让你死得瞑目).
当一个后台进程向终端写的时候, 有可能会收到 SIGOUT 信号. 这里说的有可能是看标识位设置没(默是没有设置, 设置了就会收到).
设置标识位:
<pre><code class="language-bash highlight-chroma">% stty tostop
</code></pre>
清除:
<pre><code class="language-bash highlight-chroma">% stty -tostop
</code></pre>
查看:
<pre><code class="language-bash highlight-chroma">% stty -a
</code></pre>
同理, 如果 tostop 标志设置了, 并且对 SIGOUT 信号忽略或者是阻挡,当 <code>write()</code> 的时候, 直接返回 EIO 错误.
最后介绍一下进程组的一种特殊情况, 孤儿.
这个的定义就是当进程组的爸爸进程先于儿子们挂了, 那么这个进程组就没有爸爸了. 那么如果一个进组成为孤儿了, 并且一些成员停止了(不知 terminate). 然后所有的成员都会收到 SIGHUP 然后是 SIGONT 信号. 这样做的思想其实很简单: 进程组爸爸进程有可能是任务控制进程组的爸爸, 也有可能是一个 session 下的不同进程组爸爸. 当他活着的时候能控制组里的儿子进程停止与开始, 但是一旦死掉, 就没有进程能够控制儿子进程的死活了. 因此爸爸进程死的时候附带的一些 stop 的子进程, 或者自己会发送 SIGHUP 信号, 子进程要么也同样 stop, 要么就是捕获信号或者忽略信号. 接着就是 SIGCONT 信号让它们这些没有 stop 的子进程继续工作.
需要注意的是 session 管理者的进程组本身就是孤儿, 所有这个 leader 死掉后并不会发送 sig.
<h3 id="会话">会话</h3>
每个进程组只能属于一个 session 中. (当进程创建后, 它就会成为其父进程所属的 session 的成员) 传统意义来说, session ID 等于这个 session 中第一个进程的 PID. <code>getsid()</code> 可以得改 session leader 的 ID.
每个 session 可能都会有一个控制 tty, 也成为该 session 中各个成员进程的控制 tty. 这个 tty 的文件述符是通过打开"/etc/tty"查看的, 如果打不开说明没有控制 tty. 如果给定终端的文件描述符, <code>tgetsid(fd)</code> 来获取 SID.
一个 session 也经常被一个 login 进程所设定. 我们经常看到的就是登陆终端的进程成为了控制终端会话. 所有 login process 的子进程都成为这个会话的成员.
那么如何创建一个 session 呢:
<pre><code class="language-c highlight-chroma">pid = setsid();
</code></pre>
当且仅当当前进程不是进程组的 leader. 为了保证这样, 我们使用这之前会先:
<pre><code class="language-c highlight-chroma">p = fork();
if (p) exit<span class="h

ghlight-p">(0);</s
an>
<span class="high
ight-n">pid = setsid
/span>();
</code></pre>
是父进程就直接退出, 或者去执行其他的任务, 由子进程 PID 去创建 session 并且 SID 就是这个 P
D. 并且这个进程成为了新进程组的 leader 进程.到目前, session 和进程组都只含有这一个进程, 并且
没有获得控制终端.
那么如何获取一个终端呢:
System V 的思想是第一个打开 tty 的进程成为它的控制 tty.
BSD 的思想是:
<pre><code class="language-c highlight-chroma"><span clas
="highlight-cl">ioctl(</span
fd, <span class="hig
light-n">TIOCSCTTY, <span class="highlight-p"
...);
</code></pre>
Linux 试图兼容两者, 于是规则变得显而易见的复杂(呵呵~), 大体是这样的:
     方法 1: ioctl 中的 “TIOCSCTTY” 给我们一个
制终端, 但是得满足
          (i)当前进程是 session
eader
          (ii)当前进程没有控制
端.
           (iii)控制终端不能控制
其他的 session.
     方法 2: 打开一个 tty 将会的到一个控制终端.
是得满足
           (i)当前进程是 session
eader
           (ii)当前进程没有控制
端.
            (iii)控制终端不能控制
其他的 session.
            (iv)<code>open()</c
de> 不能带有 <code>O_NOCTTY</code> 标志位.
            (v)这个 tty 不能是前

HOLD ON， confused in console and terminal (ps VT), sorry.....!
<h3 id="线程">线程</h3>