更多嵌入式 Linux 学习资料，请关注：一口 Linux 回复关键字:1024

守护进程

概念：

守护进程，也就是通常所说的 Daemon 进程，是 Linux 中的后台服务进程。周期性的执行某种任务或等待处理某些发生的事件。

Linux 系统有很多守护进程，大多数服务都是用守护进程实现的。比如：像我们的 tftp，samba，nfs 等相关服务。

UNIX 的守护进程一般都命名为*d 的形式，如 httpd，telnetd 等等。

生命周期:

守护进程会长时间运行，常常在系统启动时就开始运行，直到系统关闭时才终止。

守护进程不依赖于终端

从终端开始运行的进程都会依附于这个终端，这个终端称为这些进程的控制终端。当控制终端被关闭时，相应的进程都会被自动关闭。咱们平常写进程时，一个死循环程序，咱们不知道有 ctrl+c 的时候，怎么关闭它呀，是不是关闭终端呀。也就是说关闭终端的同时也关闭了我们的程序，但是对于守护进程来说，其生命周期守护需要突破这种限制，它从开始运行，直到整个系统关闭才会退出，所以守护进程不能依赖于终端。

查看守护进程

ps axj

a: 显示所有

x：显示没有控制终端的进程

j：显示与作业有关的信息（显示的列）：会话期 ID（SID），进程组 ID（PGID），控制终端（TT），终端进程组 ID（TRGID）

```
root@zh:/home/test# ps axj
  PPID    PID   PGID    SID TTY      TPGID STAT   UID   TIME COMMAND
     0      1      1      1 ?           -1 Ss       0   0:12 /sbin/init splash
     0      2      0      0 ?           -1 S        0   0:00 [kthreadd]
     2      3      0      0 ?           -1 I<       0   0:00 [rcu_gp]
     2      4      0      0 ?           -1 I<       0   0:00 [rcu_par_gp]
     2      6      0      0 ?           -1 I<       0   0:00 [kworker/0:0H-kb]
     2      9      0      0 ?           -1 I<       0   0:00 [mm_percpu_wq]
     2     10      0      0 ?           -1 S        0   0:01 [ksoftirqd/0]
```

- 所有的守护进程都是以超级用户启动的(UID 为 0)；
- 没有控制终端(TTY 为？)；
- 终端进程组 ID 为-1(TPGID 表示终端进程组 ID，该值表示与控制终端相关的前台进程组，如果未和任何终端相关，其值为-1；
- 所有的守护进程的父进程：

历史上，Linux 的启动一直采用 init 进程；下面的命令用来启动服务。

这种方法有两个缺点：

1. 启动时间长。init 进程是串行启动，只有前一个进程启动完，才会启动下一个进程。
2. 启动脚本复杂。init 进程只是执行启动脚本，不管其他事情。脚本需要自己处理各种情况，这往往使得脚本变得很长。

Systemd

就是为了解决这些问题而诞生的。它的设计目标是，为系统的启动和管理提供一套完整的解决方案。

根据 Linux 惯例，字母 d 是守护进程（daemon）的缩写。Systemd 这个名字的含义，就是它要守护整个系统。

进程组、会话、控制终端

进程组

shell 里的每个进程都属于一个进程组，创建进程组的目的是用于简化向组内所有进程发送信号的操作，即如果一个信号是发给一个进程组，则这个组内的所有进程都会受到该信号【方便管理】。

• PGID 进程组 ID

进程组内的所有进程都有相同的 PGID，等于该组组长的 PID。（进程组组长：进程组中有一个进程担当组长。进程组 ID（PGID）等于进程组组长的进程 ID。已知一个进程，要得到该进程所属的进程组 ID 可以调用 getpgrp。一个进程可以通过另一个系统调用 setpgrp 来加入一个已经存在的进程组或者创建一个新的进程组。

如果内核支持_POSIX_JOB_CONTROL（该宏被定义）则内核会为 Shell 上的每一条命令行（可能由多个命令通过管道等连接）创建一个进程组。从这点上看，进程组不是进程的概念，而是 shell 上才有，所以在 task_struct 里并没有存储进程组 id 之类的变量。

进程组的生命周期到组中最后一个进程终止或其加入其他进程组（离开本进程组）为止。

会话

一般一个用户登录后新建一个会话，每个会话也有一个 ID 来标识（SID）。登录后的第一个进程叫做会话领头进程（session leader），通常是一个 shell/bash。对于会话领头进程，其 PID=SID。

控制终端

一个会话一般会拥有一个控制终端用于执行 IO 操作。会话的领头进程打开一个终端之后，该终端就成为该会话的控制终端。与控制终端建立连接的会话领头进程也称为控制进程（controlling process）。一个会话只能有一个控制终端。

前台进程组

该进程组中的进程能够向终端设备进行读、写操作的进程组。例如登陆 shell（例如 bash）通过调用 int tcsetpgrp(int fd, pid_t pgrp); 函数设置为某个进程组 pgrp 关联终端设备 fd，该函数执行成功后，该进程组 pgrp 成为前台进程组。

后台进程组

该进程组中的进程只能够向终端设备写。

终端进程组 ID

每个进程还有一个属性，终端进程组 ID(TPGID)，用来标识一个进程是否处于一个和终端相关的进程组中。前台进程组中的进程的 TPGID=PGID，后台进程组的 PGID≠TPGID。若该进程和任何终端无关，其值为-1。通过比较他们来判断一个进程是属于前台进程组，还是后台进程组。

进程组、对话期和控制终端关系

进程组、对话期和控制终端关系

1. 每个会话有且只有一个前台进程组，但会有 0 个或者多个后台进程组。
2. 产生在控制终端上的输入（Input）和信号（Signal）将发送给会话的前台进程组中的所有进程。对于输出（Output）来说，则是在前台和后台共享的，即前台和后台的打印输出都会显示在屏幕上。
3. 终端上的连接断开时（比如网络断开或 Modem 断开），挂起信号将发送到控制进程（controlling process）。
4. 一个用户登录后创建一个会话。一个会话中只存在一个前台进程组，但可以存在多个后台进程组。第一次登陆后第一个创建的进程是 shell，也就是会话的领头进程，该领头进程缺省处于一个前台进程组中并打开一个控制终端可以进行数据的读写。当在 shell 里运行一行命令后（不带&）创建一个新的进程组，命令行中如果有多个命令会创建多个进程，这些进程都处于该新建进程组中，shell 将该新建的进程组设置为前台进程组并将自己暂时设置为后台进程组。

举例

1. 打开第一个终端执行命令:

```
ping 127.0.0.1 -aq | grep icmp &  // 通过管道将两个命令串接起来ping -q 不显示timeout 信息，将其设置到后台并running
```

2. 在第一个终端继续执行命令，在前台再新建一个进程组。【注意没有&】

```
ping 127.0.0.1 -aq | grep icmp //在前台再新建一个进程组，
```

3. 开启第二个终端并运行

```
 ps axj | grep pts/0       即过滤只看 pts/0 里的会话

 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 2109  2111  2111  2111 pts/0     2538 Ss    1000   0:01 bash
```

```
2111  2503  2503  2111 pts/0     2538 S     1000   0:00 ping 127.0.0.1 -aq
2111  2504  2503  2111 pts/0     2538 S     1000   0:00 grep --color=auto icmp
2111  2538  2538  2111 pts/0     2538 S+    1000   0:00 ping 127.0.0.2 -aq
2111  2539  2538  2111 pts/0     2538 S+    1000   0:00 grep --color=auto timeo
```

- SID 都是 2111，说明大家都在一个 Session 里
- 有三个进程组 PGID 2111，2503 和 2538。我们可以看到用|连起来的 ping 和 grep 是在一个进程组里的。
- 2538 这个进程组是一个前台的进程组，因为其 PGID==TGPID， 2503 这个进程组是一个后台进程组

4. 在第一个终端中执行 Ctrl+C
5. 在第二个终端里继续 ps axj | grep pts/0

```
PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
2109  2111  2111  2111 pts/0     2111 Ss+    1000   0:01 bash
2111  2503  2503  2111 pts/0     2111 S      1000   0:00 ping 127.0.0.1 -aq
2111  2504  2503  2111 pts/0     2111 S      1000   0:00 grep --color=auto icmp
```

- 2538 那个前台进程组的所有进程都消失了，说明信号会发给前台进程组的所有进程
- 2111，即 bash 所在的那个进程组成为了前台进程组。

守护进程创建流程

守护进程创建流程如下：

1. 创建子进程，父进程退出
2. 在子进程中创建新会话
3. 改变当前目录为根目录
4. 重设文件权限掩码
5. 关闭文件描述符

1.创建子进程，父进程退出

由于守护进程是脱离控制终端的，因此，完成第一步后就会在 shell 终端里造成一程序已经运行完毕的假象。之后的所有后续工作都在子进程中完成，而用户在 shell 终端里则可以执行其他的命令，从而在形式上做到了与控制终端的脱离。由于父进程已经先于子进程退出，会造成子进程没有父进程，从而变成一个孤儿进程。在 Linux 中，每当系统发现一个孤儿进程，就会自动由 1 号进程收养。原先的子进程就会变成 init 进程的子进程。

2. 在子进程中创建新会话

setsid()函数的作用。一个进程调用 setsid()函数后，会发生如下事件：

- 首先内核会创建一个新的会话，并让该进程成为该会话的 leader 进程，
- 同时伴随该 session 的建立，一个新的进程组也会被创建，同时该进程成为该进程组的组长。
- 该进程此时还没有和任何控制终端关联。若需要则要另外调用 tcsetpgrp，前面讲前台进程组时介绍过。

调用 setsid()有以下 3 个作用：

- 让进程摆脱原会话的控制。
- 让进程摆脱原进程组的控制。
- 让进程摆脱原控制终端的控制。

那么，在创建守护进程时为什么要调用 setsid()函数呢？

读者可以回忆一下创建守护进程的第一步，在那里调用了 fork()函数来创建子进程再令父进程退出。由于在调用 fork()函数时，子进程全盘复制了父进程的会话期、进程组和控制终端等，虽然父进程退出了，但原先的会话期、进程组和控制终端等并没有改变，因此，还不是真正意义上的独立。而 setsid()函数能够使进程完全独立出来，从而脱离所有其他进程和终端的控制。

详细见 man 2 setsid。

3.改变当前目录为根目

这一步也是必要的步骤。使用 fork()创建的子进程继承了父进程的当前工作目录。由于在进程运行过程中，当前目录所在的文件系统（如“/mnt/usb”等）是不能卸载的，这对以后的使用会造成诸多的麻烦（如系统由于某种原因要进入单用户模式）。

因此，通常的做法是让“/”作为守护进程的当前工作目录，这样就可以避免上述问题。当然，如有特殊需要，也可以把当前工作目录换成其他的路径，如/tmp。改变工作目录的常见函数是 chdir()。

4. 重设文件权限掩码

文件权限掩码是指屏蔽掉文件权限中的对应位。

例如，有一个文件权限掩码是 050，它就屏蔽了文件组拥有者的可读与可执行权限。由于使用 fork()函数新建的子进程继承了父进程的文件权限掩码，这就给该子进程使用文件带来了诸多的麻烦。

因此，把文件权限掩码设置为 0，可以大大增强该守护进程的灵活性。设置文件权限掩码的函数是 umask()。在这里，通常的使用方法为 umask(0)。即赋予最大的能力。

5. 关闭文件描述符

同文件权限掩码一样，用 fork()函数新建的子进程会从父进程那里继承一些已经打开的文件。这些被打开的文件可能永远不会被守护进程读或写，但它们一样消耗系统资源，而且可能导致所在的文件系统无法被卸载。

在上面的第（2）步之后，守护进程已经与所属的控制终端失去了联系，因此，从终端输入的字符不可能达到守护进程，守护进程中用常规方法（如 printf()）输出的字符也不可能在终端上显示出来。

所以，文件描述符为 0、1 和 2 的 3 个文件（常说的输入、输出和报错这 3 个文件）已经失去了存在的价值，也应被关闭。

代码实现

```c
/*
 关注一口 Linux
*/
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

int main()
{
 pid_t pid;
 int i, fd;
 char *buf = "This is a Daemon\n";

 pid = fork();
 if (pid < 0) {
  printf("Error fork\n");
  exit(1);
 }

 /* 第一步,父进程退出 */
 if (pid > 0) {
  exit(0);
```

```
}
/* 第二步 */
setsid();
/* 第三步 */
chdir("/");
/* 第四步 */
umask(0);
/* 第五步 */
for(i = 0; i < getdtablesize(); i++)
{
 close(i);
}

/* 这时创建完守护进程，以下开始正式进入守护进程实际工作
 * 注意：由于此时守护进程完全脱离了控制终端，因此，不能像其他普通进程
 * 一样通过 printf 或者 perror 将错误信息输出到控制终端，一种通用的办
 * 法是使用 syslog 服务，将程序中的出错信息输入到系统日志文件中。
 * 本程序着重演示创建守护进程的步骤，暂不演示 syslog。
 */
while(1) {
 if ((fd = open("/tmp/daemon.log",
   O_CREAT|O_WRONLY|O_APPEND, 0600)) < 0) {
  exit(1);
 }
 write(fd, buf, strlen(buf) + 1);
 close(fd);
 sleep(10);
}

exit(0);
}
```

执行结果

```
root@zh:/home/test# ps axj
  PPID    PID   PGID    SID TTY        TPGID STAT   UID   TIME COMMAND
     0      1      1      1 ?           -1 Ss       0   0:12 /sbin/init splash
  1516   7520   7520   7520 ?           -1 Ss       0   0:00 ./run
```

由上图可见：

- 守护进程./run 的 UID 为 0；
- 没有控制终端(TTY 为？)；
- 终端进程组 ID 为-1；
- 守护进程的父进程为 1516，即 systemd 。

```
    1   1516   1516   1516 ?           -1 Ss    1001   0:00 /lib/systemd/systemd --user
```