## 第5-1课 讲稿（进程控制开发-2）

1. **Linux 守护进程**
2. 守护进程概述

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。它是一个**生存期较长**的进程，通常**独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件**。守护进程常常在系统引导载入时启动，在系统关闭时终止。Linux 有很多系统服务，大多数服务都是通过守护进程实现的，如network就是一个守护进程，用来激活已经配置网络接口的脚本程序。

在Linux中，**系统与用户进行交流的界面称为终端**，每一个从该终端开始运行的进程都会依附于这个终端，这个终端就称为这些进程的**控制终端**，当**控制终端被关闭时，相应的进程都会自动关闭**。但是守护进程却能够突破这种限制，它从被执行开始运转，直到整个系统关闭时才会退出。**如果想让某个进程不因为用户、终端或者其他的变化而受到影响，那么就必须把这个进程变成一个守护进程**。

1. 编写守护进程

编写守护进程只需要5个步骤，分别是创建子进程、退出父进程；在子进程中创建新会话；改变当前目录为根目录；重设文件权限掩码和关闭文件描述符。下面分别介绍这5个步骤：

1. 创建子进程，退出父进程

这是编写守护进程的第一步。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终端里造成一种程序已经运行完毕的假象，之后的所有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。

但是，父进程创建了子进程之后退出，此时该子进程不就没有父进程了吗？守护进程中确实会出现这么一个有趣的现象，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（也就是init进程）收养它，这样，原先的子进程就会变成init进程的子进程了。其关键代码如下所示：

|  |
| --- |
| pid = fork();  if (pid> 0)  {  exit(0); /\*父进程退出\*/  } |

1. 在子进程中创建新会话

会话：是一个客户与服务器之间的不中断的请求响应序列。

在这里创建会话使用的是系统函数setsid()，在具体介绍setsid()之前，首先要了解两个概念：进程组和会话期。

* 进程组。

进程组是一个或多个进程的集合，每一个进程都属于一个进程组，进程组由进程组ID 来惟一标识。除了进程号（PID）之外，进程组ID也是一个进程的必备属性。

每个进程组都有一个组长进程，其组长进程的进程号等于进程组ID，且该进程组ID不会因组长进程的退出而受到影响。

* 会话期

会话期是一个或多个进程组的集合。

通常，一个会话开始于用户登录，终止于用户退出，在此期间该用户运行的所有进程都属于这个会话期。

接下来就可以具体介绍setsid()的相关内容。

1. setsid()函数作用。

setsid()函数用于创建一个新的会话，并担任该会话组的组长。调用setsid()有下面的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？因为创建守护进程的第一步调用了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的控制。

1. setsid()函数格式。

表8列出了setsid()函数的语法规范。

|  |  |
| --- | --- |
| 表8 setsid()函数语法 | |
| 所需头文件 | #include <sys/types.h>  #include <unistd.h> |
| 函数原型 | pid\_t setsid(void) |
| 函数返回值 | 成功：该进程组ID  出错：-1 |

1. 改变当前目录为根目录

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（比如“/mnt/usb”等）是不能卸载的，这对以后的使用会造成诸多的麻烦。因此，通常的做法是让“/”作为守护进程的当前工作目录，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()，该函数调用成功返回值为0，调用错误返回值为-1。

1. 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。比如，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)，该函数调用后返回值为修改前的掩码值，不会有错误值返回。

1. 关闭文件描述符

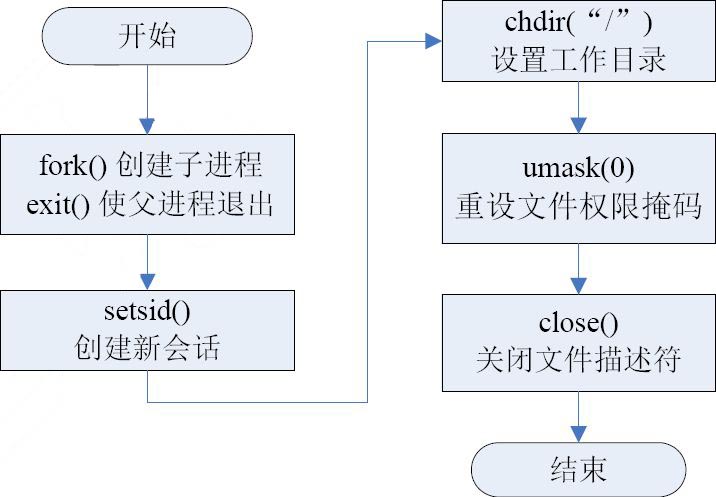
同文件权限掩码一样，用fork()函数新建的子进程会从父进程那里继承一些已经打开了的文件。这些被打开的文件可能永远不会被守护进程读或写，但它们一样消耗系统资源，而且可能导致所在的文件系统无法被卸载。

在子进程中创建新的会话之后，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。

通常按如下方式关闭文件描述符：

|  |
| --- |
| for(i = 0; i < MAXFILE; i++)  {  close(i);  } |

这样，一个简单的守护进程就建立起来了，创建守护进程的流程图如下图所示：



下面是实现守护进程的一个完整实例，该实例首先按照以上的创建流程建立了一个守护进程，然后让该守护进程每隔10s 向日志文件/tmp/daemon.log 写入一句话。

|  |
| --- |
| /\* daemon.c创建守护进程实例\*/  #include<stdio.h>  #include<stdlib.h>  #include<string.h>  #include<fcntl.h>  #include<sys/types.h>  #include<unistd.h>  int main()  {  pid\_t pid;  int i, fd;  char \*buf = "This is a Daemon\n";  pid = **fork()**; /\* 第一步\*/  if (pid< 0)  {  printf("Error fork\n");  exit(1);  }  else if (pid> 0)  {  exit(0); /\* 父进程退出\*/  }  **setsid();** /\*第二步\*/  **chdir("/");** /\*第三步\*/  **umask(0);** /\*第四步\*/  for(i = 0; i <**getdtablesize()**; i++) /\*第五步\*/  {  **close(i);**  }  /\*这时创建完守护进程，以下开始正式进入守护进程工作\*/  while(1)  {  if ((**fd = open("/tmp/daemon.log",O\_CREAT|O\_WRONLY|O\_APPEND, 0600)**) < 0)  {  printf("Open file error\n");  exit(1);  }  **write(fd, buf, strlen(buf));**  close(fd);  sleep(10);  }  exit(0);  } |

将该程序下载到开发板上，可以看到该程序每隔10s就会在对应的文件中输入相关内容。并且使用ps -e可以看到该进程在后台运行。如下所示：

|  |
| --- |
| # **tail -f /tmp/daemon.log**  This is a Daemon  This is a Daemon  This is a Daemon  This is a Daemon  …  # ps -e  76 root 1272 S ./daemon  85 root 1520 S grep daemon |

1. 守护进程的出错处理

读者在前面编写守护进程的具体调试过程中会发现，由于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将错误信息输出到控制终端来通知程序员，即使使用gdb也无法正常调试。那么，守护进程的进程要如何调试呢？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（例如“/var/log/messages”），从而可以直观地看到程序的问题所在。

|  |
| --- |
| 【小知识】  系统日志文件“/var/log/message”只能由拥有root 权限的超级用户查看。在不同Linux发行版本中，系统日志文件路径全名可能有所不同，例如可能是“/var/log/syslog” |

syslog是Linux中的系统日志管理服务，通过守护进程rsyslogd来维护。该守护进程在启动时会读配置文件“/etc/rsyslog.conf”（Linux从2.6内核开始是“/etc/rsyslog.conf”文件，之前的内核是“/etc/syslog.conf”）。**该文件决定了不同种类的消息会发送向何处，例如紧急消息可被送向系统管理员并在控制台上显示，而警告消息则可被记录到一个文件中**。

该机制提供了3个syslog相关函数，分别为openlog()、syslog()和closelog()。下面就分别介绍这3个函数。

1. syslog相关函数说明

通常，openlog()函数用于打开系统日志服务的一个连接；syslog()函数是用于向日志文件中写入消息，在这里可以规定消息的优先级、消息输出格式等；closelog()函数是用于关闭系统日志服务的连接。

1. syslog相关函数格式

表9列出了openlog()函数的语法规范。

|  |  |  |
| --- | --- | --- |
| 表9 openlog()函数语法 | | |
| 所需头文件 | #include <syslog.h> | |
| 函数原型 | void openlog (char \*ident, int option , int facility) | |
| 函数传入值 | ident | 要向每个消息加入的字符串，通常为程序的名称 |
| option | LOG\_CONS：如果消息无法送到系统日志服务，则直接输出到系统控制终端 |
| LOG\_NDELAY：立即打开系统日志服务的连接。在正常情况下，直接发送到第一条消息时才打开连接 |
| LOG\_PERROR：将消息也同时送到stderr上 |
| LOG\_PID：在每条消息中包含进程的PID |
| facility：指定程序发送的消息类型 | LOG\_AUTHPRIV：安全/授权信息 |
| LOG\_CRON：时间守护进程（cron及at） |
| LOG\_DAEMON：其他系统守护进程 |
| LOG\_KERN：内核信息 |
| LOG\_LOCAL[0~7]：保留 |
| LOG\_LPR：行打印机子系统 |
| LOG\_MAIL：邮件子系统 |
| LOG\_NEWS：新闻子系统 |
| LOG\_SYSLOG：syslogd内部所产生的信息 |
| LOG\_USER：一般使用者等级信息 |
| LOG\_UUCP：UUCP 子系统 |

表10列出了syslog()函数的语法规范。

|  |  |  |
| --- | --- | --- |
| 表10 syslog()函数语法 | | |
| 所需头文件 | #include <syslog.h> | |
| 函数原型 | void syslog(int priority, char \*format, ...) | |
| 函数传入值 | priority：指定消息的重要性 | LOG\_EMERG：紧急情况，需要立即通知技术人员 |
| LOG\_ALERT：应该被立即改正的问题，如系统数据库被破坏，ISP连接丢失 |
| LOG\_CRIT：重要情况，如硬盘错误，备用连接丢失 |
| LOG\_ERR：错误，不是非常紧急，在一定时间内修复即可 |
| LOG\_WARNING：警告信息，不是错误，比如系统磁盘使用了85%等 |
| LOG\_NOTICE：正常情况，不是错误，也不需要立即处理 |
| LOG\_INFO：情报信息，正常的系统消息，比如骚扰报告，带宽数据等，不需要处理 |
| LOG\_DEBUG：包含详细的开发情报的信息，通常只在调试一个程序时使用 |
| format | 以字符串指针的形式表示输出的格式，类似printf |

表11列出了closelog()函数的语法规范

|  |  |
| --- | --- |
| 表11closelog()函数语法 | |
| 所需头文件 | #include <syslog.h> |
| 函数原型 | void closelog(void) |

1. 使用实例。

这里将上一节中的示例程序用syslog 服务进行重写，其中有区别的地方用加粗的字体表示，源代码如下所示：

|  |
| --- |
| /\* syslog\_daemon.c利用syslog 服务的守护进程实例\*/  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <fcntl.h>  #include <sys/types.h>  #include <unistd.h>  #include <sys/wait.h>  #include <syslog.h>  int main()  {  pid\_t pid, sid;  int i, fd;  char \*buf = "This is a Daemon\n";  pid = fork(); /\* 第一步\*/  if (pid< 0)  {  printf("Error fork\n");  exit(1);  }  else if (pid> 0)  {  exit(0); /\* 父进程退出\*/  }  /\* 打开系统日志服务，openlog \*/  **openlog("daemon\_syslog", LOG\_PID, LOG\_DAEMON);**  if ((sid = **setsid()**) < 0) /\*第二步\*/  {  **syslog(LOG\_ERR, "setsid");**  exit(1);  }  if ((sid = chdir("/")) < 0) /\*第三步\*/  {  **syslog(LOG\_ERR, "chdir");**  exit(1);  }  umask(0);/\*第四步\*/  for(i = 0; i <getdtablesize(); i++) /\*第五步\*/  {  close(i);  }  /\*这时创建完守护进程，以下开始正式进入守护进程工作\*/  while(1)  {  if ((fd = open("/tmp/daemon.log",O\_CREAT|O\_WRONLY|O\_APPEND, 0600))<0)  {  **syslog(LOG\_ERR, "open");**  exit**(1)**;  }  write(fd, buf, strlen(buf));  close(fd);  sleep(10);  }  closelog();  exit(0);  } |

使用普通用户的身份执行此程序，由于这里的open()函数必须具有root 权限，因此，syslog 就会将错误信息写入到系统日志文件（例如“/var/log/messages”）中，查看日志文件，可以看到日志文件末尾添加了如下信息：

|  |
| --- |
| Dec 1 12:29:37 localhost daemon\_syslog[18962]: open |