

# 嵌入式Linux应用程序开发

主讲人：方攀

Email: [fpmystar@gmail.com](mailto:fpmystar@gmail.com)

Blog: [fpmystar.cublog.cn](http://fpmystar.cublog.cn)

# 嵌入式Linux应用程序开发

- 嵌入式Linux应用程序开发及交叉编译：Hello World !
- 嵌入式Linux内核模块开发：Hello Module !
- Linux环境下多进程及多线程编程

# 1.1 Linux应用程序介绍

- 在为Linux开发应用程序时,绝大多数情况下使用的都是C语言,因此几乎每一位Linux程序员面临的首要问题都是灵活运用C编译器.目前Linux下最常用的C语言编译器是GCC(GNU Compiler Collection),它是GNU项目中符合ANSI C标准的编译系统,能够编译用C、C++和Object C等语言编写的程序.GCC不仅功能非常强大,结构也异常灵活.最值得称道的一点就是它可以通过不同的前端模块来支持各种语言,如Java、Fortran、Pascal、Modula-3和Ada等.

开放自由和灵活是Linux的魅力所在,而这一点在GCC上的体现就是程序员通过它能够更好地控制整个编译过程.在使用GCC编译程序时,编译过程可以被细分为四个阶段:

◆ 预处理(Pre-Processing)

◆ 编译(Compiling)

◆ 汇编(Assembling)

◆ 链接(Linking)

Linux程序员可以根据自己的需要让GCC在编译的任何阶段结束,以便检查或使用编译器在该阶段的输出信息,或者对最后生成的二进制文件进行控制,以便通过加入不同数量和种类的调试代码来为今后的调试做好准备.和其它常用的编译器一样,GCC也提供了灵活而强大的代码优化功能,利用它可以生成执行效率更高的代码.GCC提供了30多条警告信息和三个警告级别,使用它们有助于增强程序的稳定性和可移植性.此外,GCC还对标准的C和C++语言进行了大量的扩展,提高程序的执行效率,有助于编译器进行代码优化,能够减轻编程的工作量.

博芯电子

**Prochip**  
Your Embedded Partner

# 1.1 应用程序起步—编写源文件

在/root/下建一个自己的目录

```
#mkdir project
```

```
#cd project
```

```
#vim helloworld.c
```

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

```
int main(void)
```

```
{
```

```
    printf("We love arm,we love sep4020!!\n");
```

```
    return 1;
```

```
}
```

# 代码注意问题

- ◆main函数的返回值应该是int类型；
- ◆main函数在终止前没有调用“return 1;”语句来结尾。

## 1.2 编译helloworld.c

- （1）在pc机上编译能在pc机上运行的应用程序的方法

# gcc -o helloworld\_pc helloworld.c（这个是用i386的gcc编译器编译的，所以只能在pc机的linux环境中运行）

- （2）在pc机上编译能在sep4020板子上运行的应用程序的方法

# arm-linux-gcc -o helloworld\_arm helloworld.c（这个是用arm的交叉gcc编译器编译的，所以可以下载到板子上运行）

## 1.3运行helloworld程序

- （1）在pc机上运行i386的程序

```
[root@localhost code]# ./helloworld_pc
```

```
We love arm,we love sep4020!!
```

- （2）在开发板上运行应用程序

将应用程序helloworld\_arm拷贝到/nfs/demo文件夹下

```
[root@localhost code]# cp helloworld_arm /nfs/demo
```

```
/ # cd demo
```

```
/demo # ./helloworld_arm
```

```
We love arm,we love sep4020!!
```

## 2.1 编写Hello Module源代码

- 前面我们介绍了一个简单的Linux程序Hello World，它是运行于用户态的应用程序，现在我们再介绍一个运行于内核态的 Hello Module程序，它其实是一个最简单的驱动程序模块。

我们将Hello Module的源代码放置于/root/project/module目录,名称为hellomodule.c，内容如下：



- #include <linux/kernel.h>
- #include <linux/module.h>
- 
- MODULE\_LICENSE("GPL");
- 
- static int \_\_init sep4020\_hello\_module\_init(void)
- {
- printk("Hello, sep4020 module is installed !\n");
- return 0;
- }
- 
- static void \_\_exit sep4020\_hello\_module\_cleanup(void)
- {
- printk("Good-bye, sep4020 module was removed!\n");
- }
- 
- module\_init(sep4020\_hello\_module\_init);
- module\_exit(sep4020\_hello\_module\_cleanup);

## 2.2 编译Hello Module源代码

由于这个模块是加到嵌入式linux的内核中的，所以它肯定会用到许多嵌入式linux源码的头文件的，我们的嵌入式linux的内核源码位置在/linux-3.2/下面，这中间的链接过程非常复杂，为了不让我们手动输入编译指令，一般编译2.6版本的驱动模块需要把驱动代码加入内核代码树，并做相应的配置，如下步骤

- Step1: 把刚才的源码文件拷贝到/linux-3.2/drivers/char/sep4020\_char目录下
- Step 2: 编辑配置 /linux-3.2/drivers/char/sep4020\_char 的Kconfig文件，加入驱动选项（注意tristate 前面有tab键）

config SEP4020\_HELLOMODULE

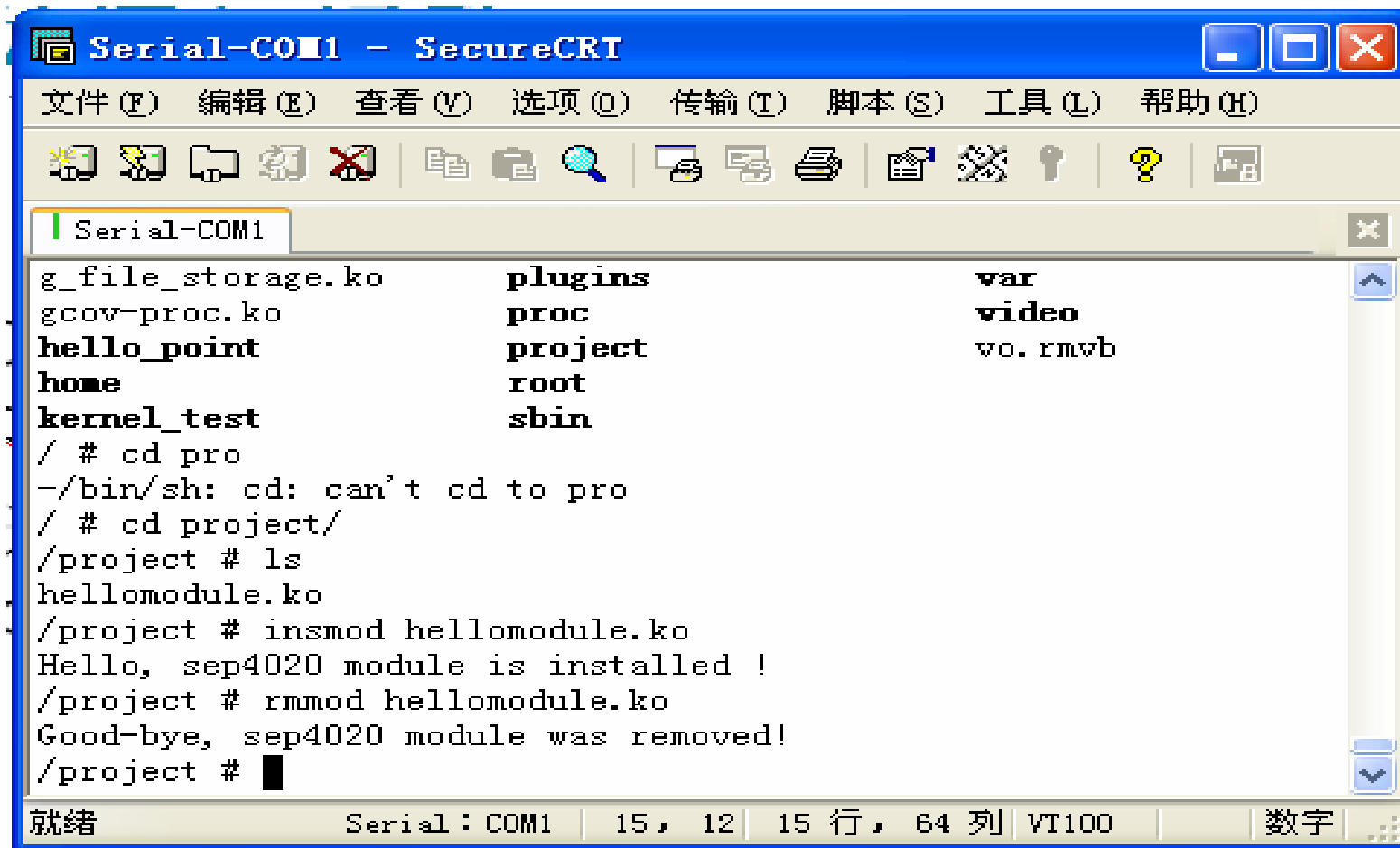
tristate "sep4020 hello module driver"

使之在 make menuconfig的时候出现；



- Step3: 修改Makefile文件，在其中加入：
- `obj-$(CONFIG_SEP4020_HELLOMODULE) += hellomodule.o`
- Step4: 这时回到 /linux-3.2 源代码根目录位置，执行 `make modules`

## 2.3 把 HelloModule 下载到开发板并安装使用



```
Serial-COM1 - SecureCRT
文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)
Serial-COM1
g_file_storage.ko      plugins      var
gcov-proc.ko          proc         video
hello_point          project      wo.rmwb
home                 root
kernel_test          sbin
/ # cd pro
-/bin/sh: cd: can't cd to pro
/ # cd project/
/project # ls
hellomodule.ko
/project # insmod hellomodule.ko
Hello, sep4020 module is installed !
/project # rmmod hellomodule.ko
Good-bye, sep4020 module was removed!
/project # █
就绪      Serial: COM1 | 15, 12 | 15 行, 64 列 | VT100 | 数字
```

# 3 Linux进程编程

- 3.1 Linux进程的概念
- 3.2 Linux下的进程启动
- 3.3 进程控制编程
- 3.4 守护进程编程
- 3.5 进程间通信

## 3.1 Linux进程的概念

- （1）进程是一个独立的可调度的活动；
- （2）进程是一个抽象实体，当它执行某个任务时，将要分配和释放各种资源（P. Denning）；
- （3）进程是可以并行执行的计算部分。
- 以上进程的概念都不相同，但其本质是一样的。它指出了进程是一个程序的一次执行的过程。它和程序是有本质区别的，程序是静态的，它是一些保存在磁盘上的指令的有序集合，没有任何执行的概念；而进程是一个动态的概念，它是程序执行的过程，包括了动态创建、调度和消亡的整个过程。它是程序执行和资源管理的最小单位。

# 进程控制块（1）

进程是 Linux 系统的基本调度单位，那么从系统的角度看如何描述并表示它的变化呢？

- 进程标识符：当一个进程产生时，系统都会为它分配一个标识符；
- 进程所占的内存区域：每个进程执行时都需要占用一定的内存区域，此区域用于保存该进程所运行的程序代码和使用的程序变量。每一个进程所占用的内存是相互独立的，因此改变一个进程所占内存中数据的任何改动，都只对该进程产生影响，不会影响到其它进程的顺利执行；
- 文件描述符：当一个进程在执行时，它需要使用一些相关的文件描述符。文件描述符描述了被打开文件的信息，不同的进程打开同一个文件时，所使用的文件描述符是不同的。一个进程文件描述符的改变并不会对其它的进程打开同一个文件的描述符产生任何影响；



## 进程控制块（2）

- 安全信息：一个进程的安全信息包括用户识别号和组织识别号；
- 进程环境：一个进程的运行环境包括环境变量和启动该进程的程序调用的命令行；
- 信号处理：一个进程有时需要用信号同其它进程进行通信。进程可以发送和接收信号，并对其作出相应处理；
- 资源安排：进程是调度系统资源的基本单位。当多个进程同时运行时，linux系统内核安排不同进程轮流使用系统的各种资源；

## 进程控制块（3）

- 同步处理：多个程序之间同步运行的实现，也是通过进程来完成的。这将会使用到诸如共享内存、文件锁定等方法。
- 进程状态：在一个进程存在期间，每一时刻进程都处在一定的状态，包括运行、等待被调度或睡眠状态。

# 进程结构体描述

```
struct task_struct {
    long state                //任务的运行状态 (-1 不可运行, 0 可运行(就绪), >0 已停止)。
    long counter              // 任务运行时间计数(递减)(滴答数), 运行时间片。
    long priority              // 运行优先数。任务开始运行时 counter=priority, 越大运行越长。
    long signal                // 信号。是位图, 每个比特位代表一种信号, 信号值=位偏移值+1。
    struct sigaction sigaction[32] // 信号执行属性结构, 对应信号将要执行的操作和标志信息。
    long blocked               // 进程信号屏蔽码(对应信号位图)。
    int exit_code               // 任务执行停止的退出码, 其父进程会取。
    unsigned long start_code    // 代码段地址。
    unsigned long end_code      // 代码长度(字节数)。
    unsigned long end_data      // 代码长度 + 数据长度(字节数)。
    unsigned long brk           // 总长度(字节数)。
    unsigned long start_stack   // 堆栈段地址。
    long pid                   // 进程标识号(进程号)。
    long father                 // 父进程号。
    long pgrp                   // 父进程组号。
    long session                // 会话号。
    long leader                  // 会话首领。
    unsigned short uid           // 用户标识号(用户 id)。
    unsigned short euid          // 有效用户 id。
    unsigned short suid          // 保存的用户 id。
    unsigned short gid           // 组标识号(组 id)。
    unsigned short egid          // 有效组 id。
    unsigned short sgid          // 保存的组 id。
    long alarm                   // 报警定时值(滴答数)。
    long utime                   // 用户态运行时间(滴答数)。
    long stime                   // 系统态运行时间(滴答数)。
    long cutime                  // 子进程用户态运行时间。
    long cstime                  // 子进程系统态运行时间。
    long start_time              // 进程开始运行时刻。
    unsigned short used_math     // 标志: 是否使用了协处理器。
}
```

博芯电子

**Prochip**  
Your Embedded Partner

# 进程的标识

- 进程的标识：进程号、父进程号、父进程组号、会话号、会话首领号都是用于从不同的侧面来描述进程的身份
- 测试进程号的代码：

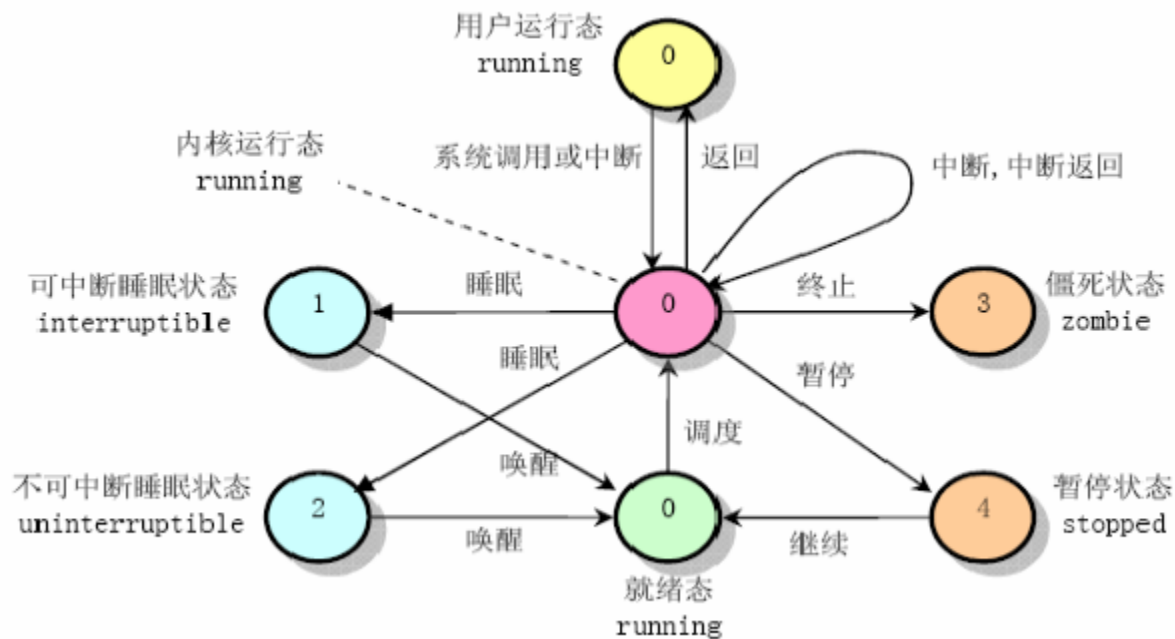
```
/*process.c*/  
#include<stdio.h>  
#include<unistd.h>  
#include <stdlib.h>  
int main()  
{  
/*获得当前进程的进程ID和其父进程ID*/  
    printf("The PID of this process is %d\n",getpid());  
    printf("The PPID of this process is %d\n",getppid());  
    return 1;  
}
```

# 进程状态

一个进程在其生存期内，可处于一组不同的状态下，称为进程状态

- 运行状态（**TASK\_RUNNING**）—当进程正在被CPU 执行，或已经准备就绪随时可由调度程序执行，则称该进程为处于运行状态（**running**）。进程可以在内核态运行，也可以在用户态运行；
- 可中断睡眠状态（**TASK\_INTERRUPTIBLE**）—当进程处于可中断等待状态时，系统不会调度该进程执行。当系统产生一个中断或者释放了进程正在等待的资源，或者进程收到一个信号，都可以唤醒进程转换到就绪状态（运行状态）；

- 不可中断睡眠状态（**TASK\_UNINTERRUPTIBLE**）—与可中断睡眠状态类似。但处于该状态的进程只有被使用**wake\_up()**函数明确唤醒时才能转换到可运行的就绪状态。
- 暂停状态（**TASK\_STOPPED**）—当进程收到信号**SIGSTOP**、**SIGTSTP**、**SIGTTIN** 或**SIGTTOU** 时就会进入暂停状态。可向其发送**SIGCONT** 信号让进程转换到可运行状态。
- 僵死状态（**TASK\_ZOMBIE**）—当进程已停止运行，但其父进程还没有询问其状态时，则称该进程处于僵死状态。



# Linux下进程的模式和类型

- 在 Linux 系统中，进程的执行模式划分为用户模式和内核模式。如果当前运行的是用户程序、应用程序或者内核之外的系统程序，那么对应进程就在用户模式下运行；如果在用户程序执行过程中出现系统调用或者发生中断事件，那么就要运行操作系统（即核心）程序，进程模式就变成内核模式。在内核模式下运行的进程可以执行机器的特权指令，而且此时该进程的运行不受用户的干扰，即使是 root 用户也不能干扰内核模式下进程的运行。
- 用户进程既可以在用户模式下运行，也可以在内核模式下运行，

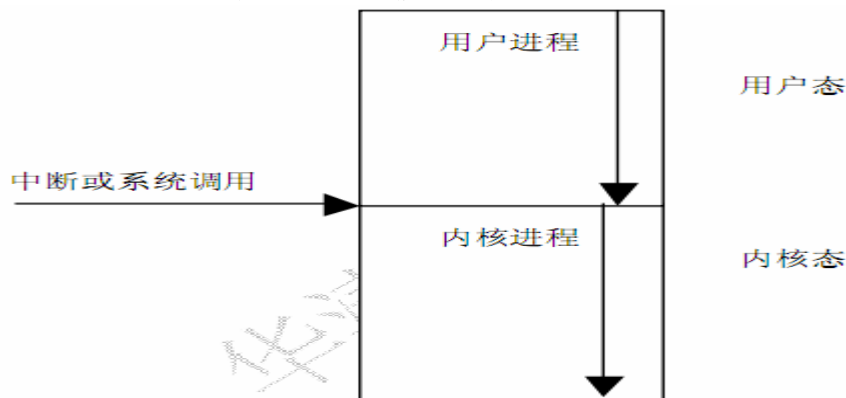


图 7.3 用户进程的两运行模式



## 3.2 Linux下的进程启动

Linux 下启动一个进程有两种主要途径：手工启动和调度启动。手动启动是由用户输入命令直接启动进程，而调度启动是指系统根据用户的设置自行启动进程。

(1) 手工启动 ,又可分为前台启动和后台启动。

- 前台启动,如“ls -l”;
- 后台启动往往是在该进程非常耗时，且用户也不急着需要结果的时候启动的,如“tftp -gr tcpip.rar 192.168.0.2 &”

(2) 调度启动 ，系统需要进行一些比较费时而且占用资源的维护工作，如at和cron

## 3.3 进程控制编程

(1)fork ( )

(2)exec函数族

(3)exit和\_exit

(4)wait和waitpid

# (1) fork ()

- **fork** 函数用于从已存在进程中创建一个新进程。新进程称为子进程，而原进程称为父进程。这两个分别带回它们各自的返回值，其中父进程的返回值是子进程的进程号，而子进程则返回0。因此，可以通过返回值来判定该进程是父进程还是子进程。

- **fork ()** 函数

头文件: `#include<unistd.h>`

格式: `pid_t fork();`

返回值: 0: 子进程

子进程ID(大于0的整数): 父进程

-1: 出错

- 子进程是父进程的拷贝，它具有和父进程相同的代码段，但是它具有自己的数据段和堆栈段。
- 子进程将从父进程获得绝大部分属性，但也会更改部分属性的值，要更改的属性为：
  - 进程ID
  - 进程组ID(更改为父进程ID)
  - SESSION ID(为子进程的运行时记录)
  - 所打开文件及文件的偏移量（父进程对文件的锁定）

```

/*fork.c*/
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    pid_t result;
    /*调用fork函数，其返回值为result*/
    result = fork();
    /*通过result的值来判断fork函数的返回情况，首先进行出错处理*/
    if(result == -1)
    {
        perror("fork");
        exit;
    }
    /*返回值为0代表子进程*/
    else if(result == 0)
    {
        printf("The return value is %d\n\n child
process!!\nMy PID is%d\n",result,getpid());
    }
    /*返回值大于0代表父进程*/
    else
    {
        printf("The return value is %d\n\n father
process!!\nMy PID is%d\n",result,getpid());
    }
    return 1;
}

```

# 编译源文件fork.c并在板子上运行

- 先在上位机编译源文件：

进入/nfs/project目录下

```
[root@localhost project]# arm-linux-gcc -o fork fork.c
```

将可执行程序下载到目标板上，运行结果如下所示：

```
Serial-COM1
/project # mount -t yaffs /dev/mtdblock0 /mnt
yaffs: dev is 32505856 name is "mtdblock0"
yaffs: Attempting MTD mount on 31.0, "mtdblock0"
block 2 is bad
/project # cd /mnt
/mnt # ls
lost+found
/mnt # cd ..
/ # cd /project/
/project # ls
fork.c          process          process.c~
hellomodule.ko  process.c
/project # ./fork
The return value is 258
In father process!!
My PID is257
The return value is 0
In child process!!
My PID is258
..
```

## (2)exec函数族

- **exec** 函数族就提供了一个在进程中启动另一个程序执行的方法。它可以根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除了进程号外，其他全部被新的进程替换了。另外，这里的可执行文件既可以是二进制文件，也可以是 Linux 下任何可执行的脚本文件。
- 在Linux中使用**exec**函数族主要有两种情况：
  - (1) 当进程认为自己不能再为系统和用户做出任何贡献时，就可以调用任何 **exec** 函数族让自己重生；
  - (2) 如果一个进程想执行另一个程序，那么它就可以调用 **fork** 函数新建一个进程，然后调用任何一个 **exec**，这样看起来就好像通过执行应用程序而产生了一个新进程。（这种情况非常普遍）

## exec函数族有下列6个函数:

- #include<unistd.h>

int execl(const char\* path,const char\* arg,...);

int execlp(const char\* file,const char\* arg,...);

int execlx(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[]);



# Execvp实例

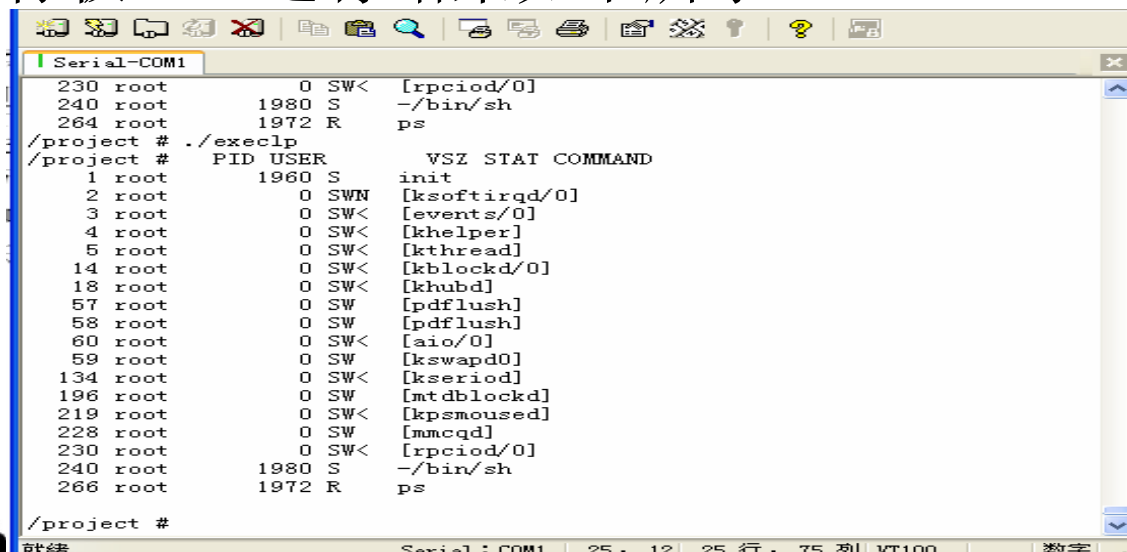
```
/*execvp.c*/
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    if(fork()==0)
    {
        /*调用execvp函数，这里相当于调用了“ps ”命令*/
        if(execvp("ps","ps",NULL)<0)
            perror("execvp error!");
    }
}
```

将源码文件execvp.c保存到/nfs/project目录下并交叉编译：  
[root@localhost project]# arm-linux-gcc -o execvp execvp.c

# Execvp运行

- 在该程序中，首先使用fork函数新建一个子进程，然后在子进程里使用 **execvp**函数。读者可以看到，这里的参数列表就是在 **shell**中使用的命令名和选项。并且当使用文件名的方式进行查找时，系统会在默认的环境变量 **PATH** 中寻找该可执行文件。读者可将编译后的结果下载到目标板上，运行结果如下所示：



```
Serial-COM1
230 root      0 SW<  [rpciod/0]
240 root     1980 S  -/bin/sh
264 root     1972 R  ps
/project # ./execvp
/project #
  PID USER      VSZ STAT COMMAND
  1 root        1960 S    init
  2 root         0 SWN   [ksoftirqd/0]
  3 root         0 SW<   [events/0]
  4 root         0 SW<   [khelper]
  5 root         0 SW<   [kthread]
 14 root         0 SW<   [kblockd/0]
 18 root         0 SW<   [khubd]
 57 root         0 SW    [pdflush]
 58 root         0 SW    [pdflush]
 60 root         0 SW<   [aio/0]
 59 root         0 SW    [kswapd0]
134 root         0 SW<   [kseriod]
196 root         0 SW    [mtdblockd]
219 root         0 SW<   [kpsmouse]
228 root         0 SW    [mmcsd]
230 root         0 SW<   [rpciod/0]
240 root     1980 S  -/bin/sh
266 root     1972 R  ps
/project #
```

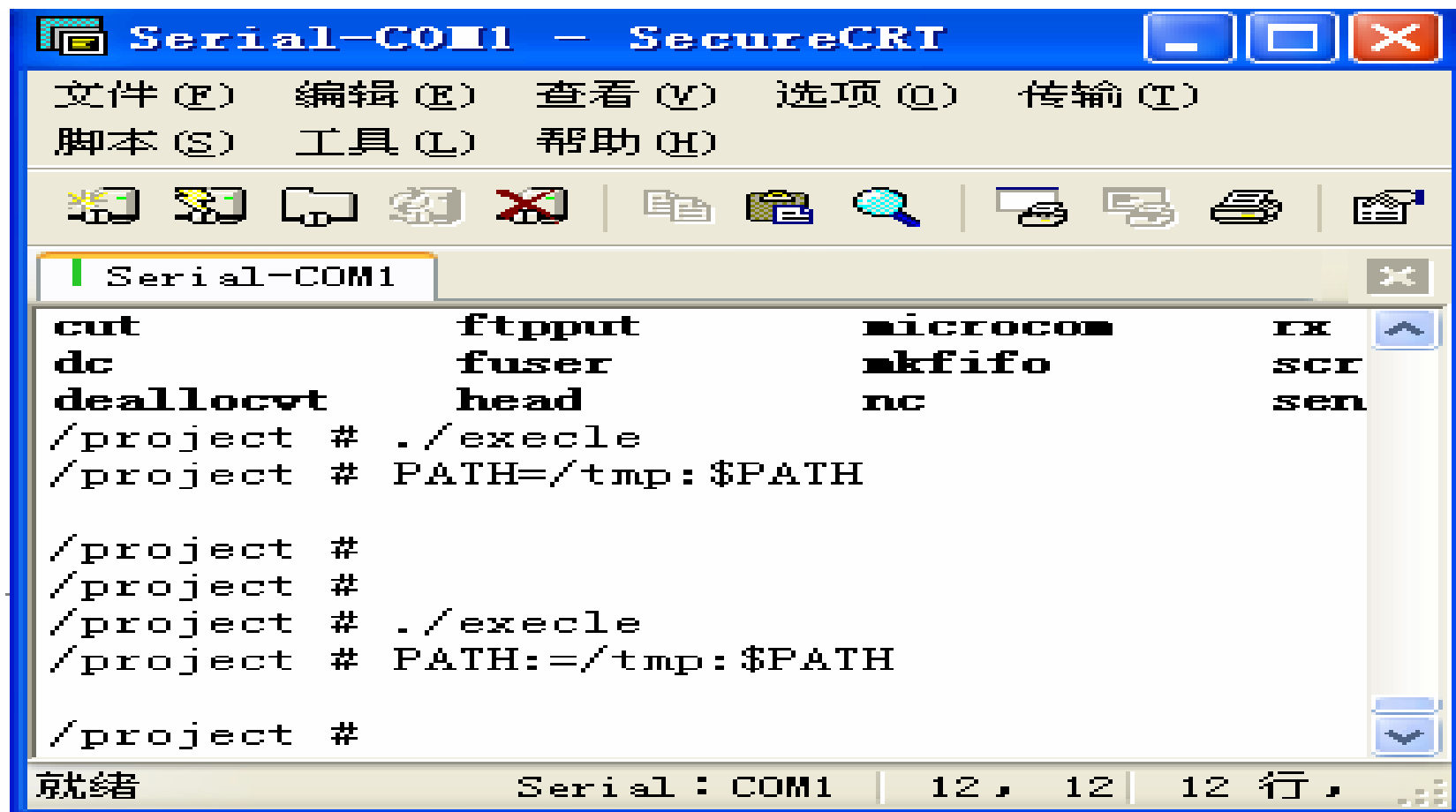
# Execl实例

```
/*execl.c*/
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
/*命令参数列表，必须以NULL结尾*/
    char *envp[]={"PATH:=/tmp:$PATH",NULL};
    if(fork()==0){
/*调用execl函数，注意这里也要指出env的完整路径*/
        if(execl("/usr/bin/env","env",NULL,envp)<0)
            perror("execl error!");
    }
    return 1;
}
```

将源码文件execl.c保存到/nfs/project目录下并交叉编译：  
[root@localhost project]# arm-linux-gcc -o execl execl.c

# Execle运行



```
Serial-COM1 - SecureCRT
文件(F)  编辑(E)  查看(V)  选项(O)  传输(T)
脚本(S)  工具(L)  帮助(H)

Serial-COM1

cut      ftpput      microcom    rx
dc        fuser      mkfifo      scr
deallocvt head        nc          sen

/project # ./execle
/project # PATH=/tmp:$PATH

/project #
/project #
/project # ./execle
/project # PATH:=/tmp:$PATH

/project #

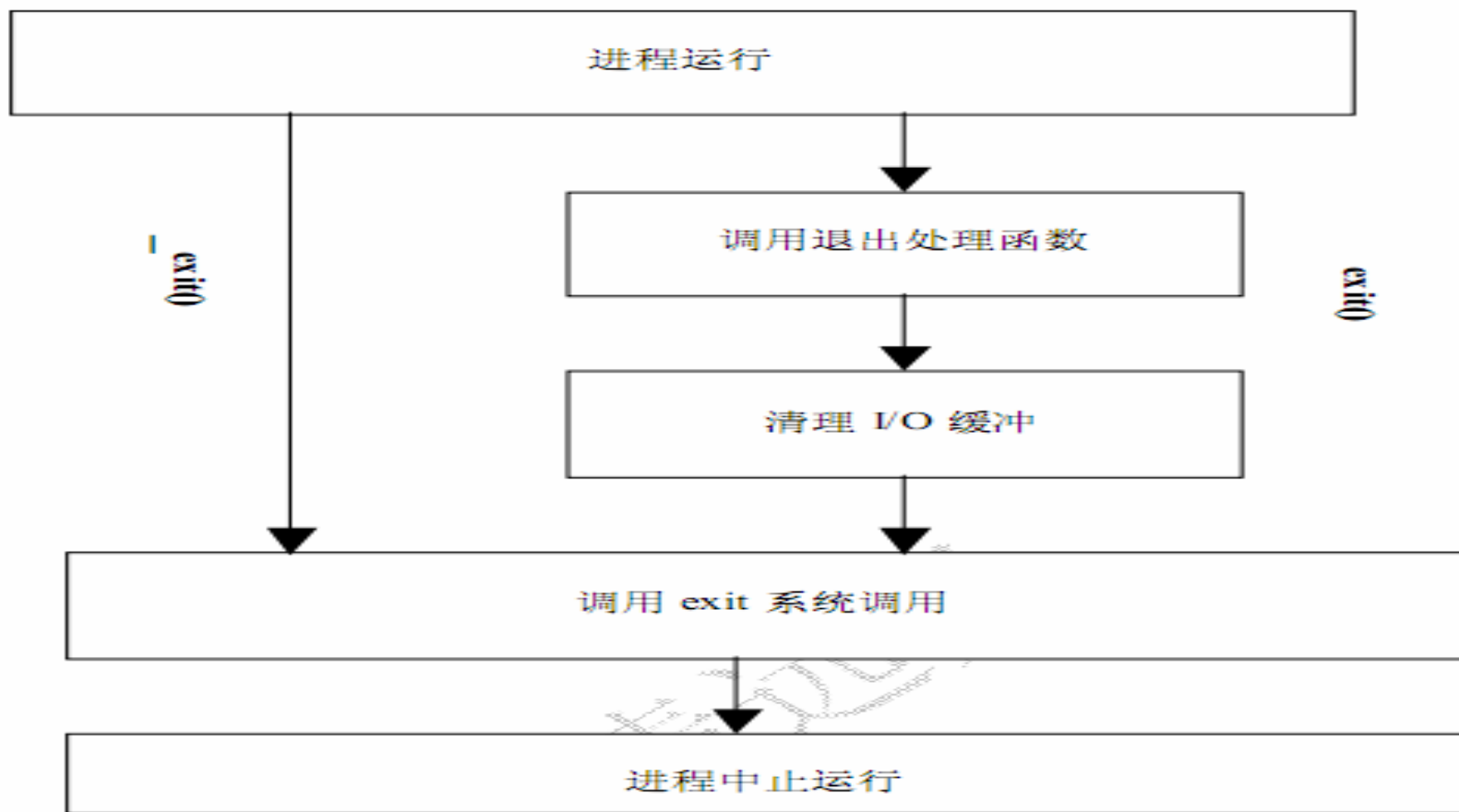
就绪      Serial: COM1 | 12, 12 | 12 行,
```

# exec函数族使用注意点

在使用 **exec** 函数族时，一定要加上错误判断语句。因为 **exec** 很容易执行失败，其中最常见的原因有：

- 找不到文件或路径，此时**errno**被设置为 **ENOENT**；
- 数组**argv**和**envp**忘记用**NULL**结束，此时 **errno**被设置为 **EFAULT**；
- 没有对应可执行文件的运行权限，此时**errno**被设置为 **EACCES**。

### (3)exit和\_exit



# 终止进程函数

- 形式一：

```
#include<stdlib.h>
```

```
void exit(int status);
```

- 说明：**exit()**函数是标准C中提供的函数。它用来终止正在运行的程序，将关闭所有被该文件打开的文件描述符

- 形式二：

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

```
void _exit(intstatus)
```

- 说明：**\_exit()**函数也可用于结束一个进程，与**exit**函数不同的是，调用**\_exit()**是为了关闭一些linux特有的退出句柄。

# 终止进程函数

- 形式三：

```
#include <stdlib.h>
```

```
int atexit(void(*function)(void));
```

- 说明：**atexit**函数在结束一个程序后，调用一个不带参数同时也没有返回值的函数。该函数名由 **function** 指针指向

- 形式四：

```
#include<stdlib.h>
```

```
void abort(void);
```

- 说明：**abort()**函数用来发送一个SIGABRT信号，这个信号将使当前进程终止。

- 形式五：

```
#include<assert.h>
```

```
void assert(intexpression);
```

- 说明：**assert**是一个宏。调用**assert**函数时，首先要计算表达式**expression**的值，如果为0，则调用**abort**（）函数结束进程。



# Exit实例

```
/**exit.c*/
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main(void)
{
    pid_t pid;
    if((pid=fork())<0)
    {
        exit(0);
    }
    else if(pid==0)
    {
        printf("go into child process!\n");
        printf("child process PID:%d",getpid());
        exit(0);
    }
    else
    {
        printf("go into parent process!\n");
        printf("Parent process
PID:%4d",getpid());
        _exit(0);
    }
    return 0;
}
```

# Exit运行

将源码文件exec1p.c保存到/nfs/project目录下并交叉编译：

```
[root@localhost project]# arm-linux-gcc -o execlp execlp.c
```

将板子启动Linux，并挂载网络文件系统，然后进入/project目录下面，运行exit实例：

```
/project #  
/project #  
/project # ./exit  
go into parent process!  
/project # go into child process!  
child process PID:246  
/project #
```

就绪

## (4)wait和waitpid

- **wait**函数是用于使父进程（也就是调用**wait**的进程）阻塞，直到一个子进程结束或者该进程收到了一个指定的信号为止。如果该父进程没有子进程或者他的子进程已经结束，则**wait**就会立即返回。
- **waitpid**的作用和**wait**一样，但它并不一定要等待第一个终止的子进程，它还有若干选项，如可提供一个非阻塞版本的**wait**功能，也能支持作业控制。实际上 **wait**函数只是 **waitpid**函数的一个特例，在Linux内部实现**wait**函数时直接调用的就是 **waitpid**函数。

wait函数调用形式:

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

```
#include <sys/wait.h>
```

```
pid_t wait(int *status)
```

这里的status是一个整型指针，是该子进程退出时的状态

- status若为空，则代表任意状态结束的子进程
- status若不为空，则代表指定状态结束的子进程

另外，子进程的结束状态可由Linux中一些特定的宏来测定  
成功的话返回子进程的进程号，失败则返回-1

- waitpid函数调用形式:

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

```
#include <sys/wait.h>
```

```
pid_t waitpid(pid_t pid, int* stat_loc, int options);
```

作用: waitpid等待指定的子进程直到子进程返回, 如果pid为正值则等待指定的进程(pid); 如果为0则等待任何一个组ID和调用者的组ID相同的进程; 为-1时等同于wait调用; 小于-1时等待任何一个组ID等于pid绝对值的进程。

参数含义: stat\_loc和wait的意义一样;

options可以决定父进程的状态, 可以取下面两个值:

WNOHANG:当没有子进程存在时, 父进程立即返回; WUNTACHED:当子进程结束时waitpid返回, 但是子进程的退出状态不可得到。

```

/*waitpid.c*/
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

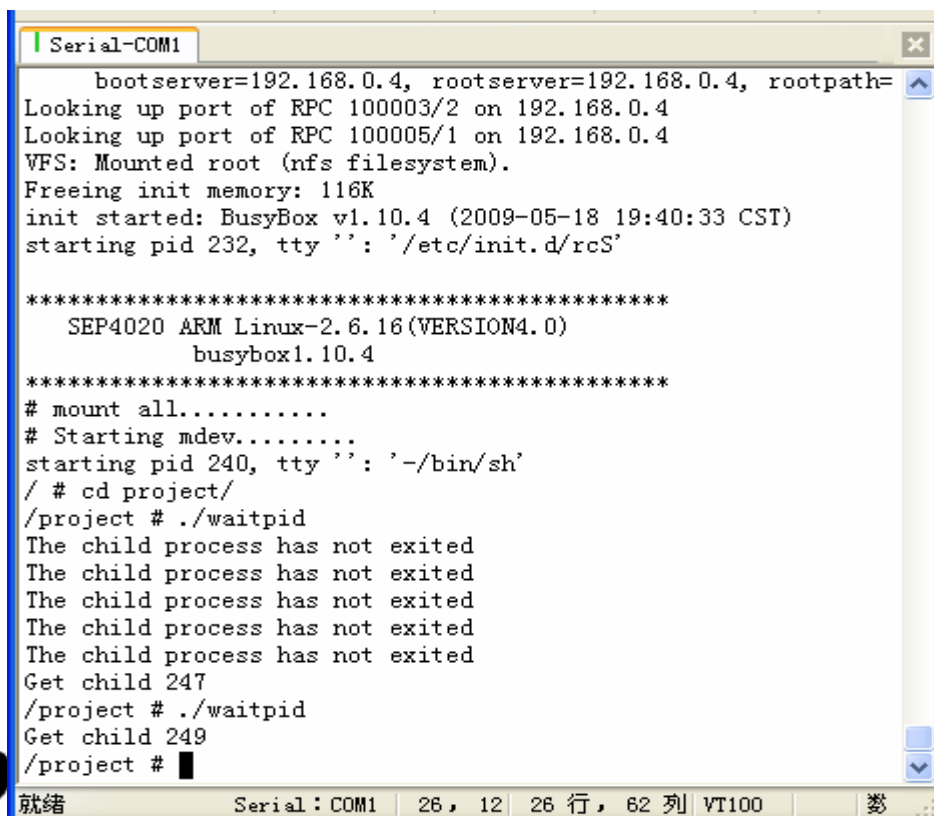
int main()
{
    pid_t pc,pr;
    pc=fork();
    if(pc<0)
        printf("Error fork.\n");
    /*子进程*/
    else if(pc==0){
/*子进程暂停5s*/
        sleep(5);
/*子进程正常退出*/
        exit(0);
    }
    /*父进程*/
    else{
/*循环测试子进程是否退出*/
        do{
/*调用waitpid, 且父进程不阻塞*/
            pr=waitpid(pc,NULL,WNOHANG);
            //pr=wait(NULL);

/*若子进程还未退出, 则父进程暂停1s*/
            if(pr==0){
                printf("The child process has not exited\n");
                sleep(1);
            }
        }while(pr==0);
/*若发现子进程退出, 打印出相应情况*/
        if(pr==pc)
            printf("Get child %d\n",pr);
        else
            printf("some error occured.\n");
    }
}

```

# Waitpid运行

将源码文件exec1p.c保存到/nfs/project目录下并交叉编译：  
[root@localhost project]# arm-linux-gcc -o execlp execlp.c  
将板子启动Linux，并挂载网络文件系统，然后进入/project  
目录下面，运行exit实例：



```
Serial-COM1
bootserver=192.168.0.4, rootserver=192.168.0.4, rootpath=
Looking up port of RPC 100003/2 on 192.168.0.4
Looking up port of RPC 100005/1 on 192.168.0.4
VFS: Mounted root (nfs filesystem).
Freeing init memory: 116K
init started: BusyBox v1.10.4 (2009-05-18 19:40:33 CST)
starting pid 232, tty '': '/etc/init.d/rcS'

*****
SEP4020 ARM Linux-2.6.16 (VERSION4.0)
busybox1.10.4
*****
# mount all.....
# Starting mdev.....
starting pid 240, tty '': '/bin/sh'
/ # cd project/
/project # ./waitpid
The child process has not exited
The child process has not exited
The child process has not exited
The child process has not exited
The child process has not exited
Get child 247
/project # ./waitpid
Get child 249
/project # █

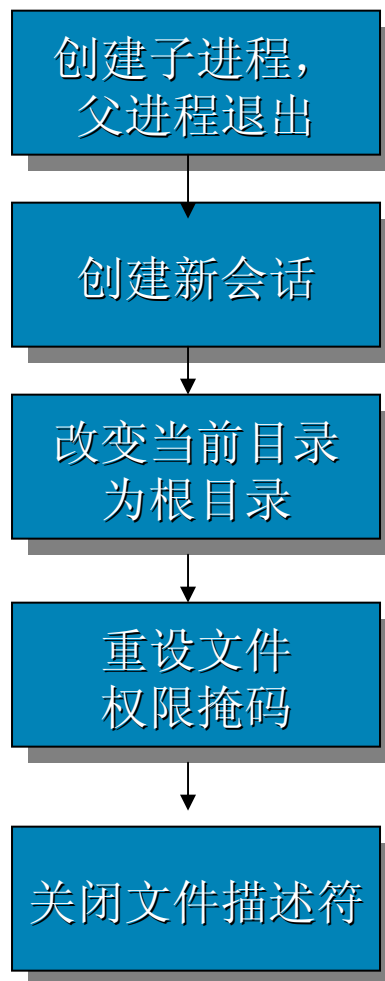
就绪      Serial: COM1      26, 12      26 行, 62 列 VT100      参
```

## 3.4 守护进程编程

- 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装入时启动，在系统关闭时终止。**Linux** 系统有很多守护进程，大多数服务都是通过守护进程实现的。



# 创建守护进程的步骤



# (1) 创建子进程，父进程退出

- 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。
- /\*父进程退出\*/

```
pid=fork();  
    if(pid>0){  
        exit(0);  
    }
```

## （2）在子进程中创建新会话（setsid函数）

- 这个步骤是创建守护进程中最重要的一步
- 进程组：进程组是一个或多个进程的集合。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 ID 不会因组长进程的退出而受到影响。
- 会话期：会话组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。
- 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。

## (3) 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（比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的问题，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。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)**。

## (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卸下。

```
for(i=0;i<MAXFILE;i++)
```

```
    close(i);
```

```

/*dameon.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>
#include<sys/wait.h>

#define MAXFILE 65535
int main()
{
    pid_t pc;
    int i,fd,len;
    char *buf="This is a Dameon\n";
    len =strlen(buf);
    pc=fork(); //第一步
    if(pc<0){
        printf("error fork\n");
        exit(1);
    }else if(pc>0)
        exit(0);
    /*第二步*/
    setsid();
    /*第三步*/
    chdir("/");

```

博芯电子

/\*第三步\*/

**Prochip**  
Your Embedded Partner

```

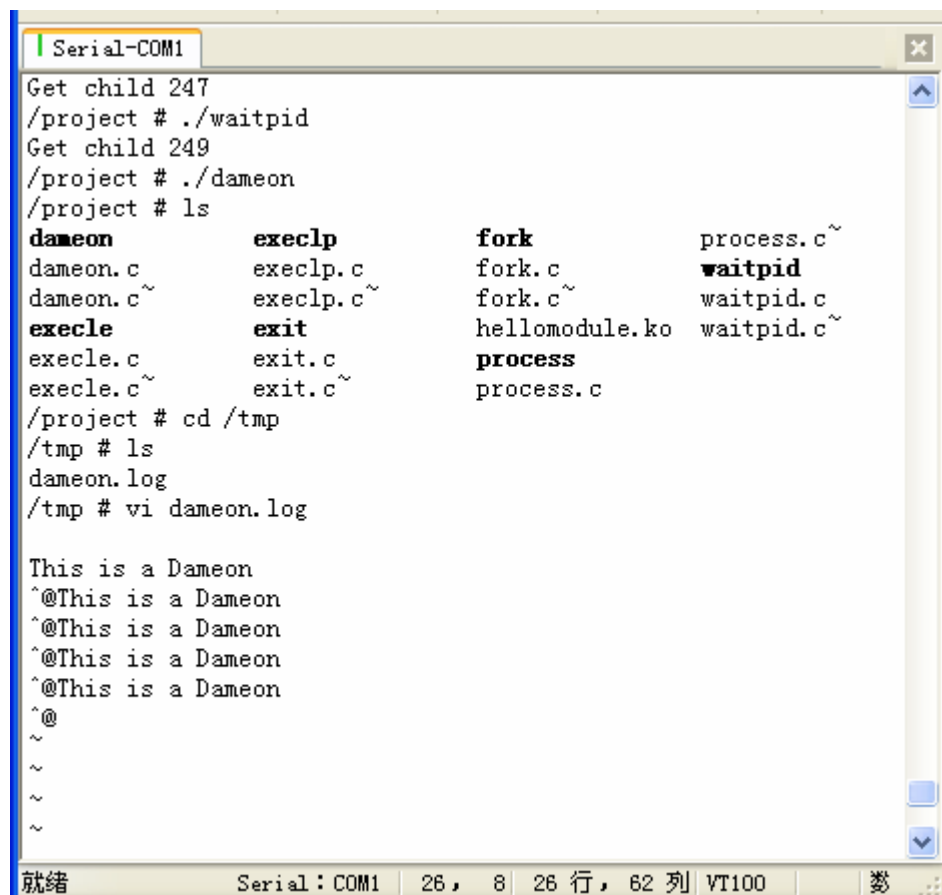
/*第四步*/
    umask(0);
    for(i=0;i<MAXFILE;i++)
/*第五步*/
        close(i);
/*这时创建完守护进程，以下开始正式进入
守护进程工作*/
    while(1){

        if((fd=open("/tmp/dameon.log",O_C
REAT|O_WRONLY|O_APPEND,0600))<0)
        {
            perror("open");
            exit(1);
        }
        write(fd, buf, len+1);
        close(fd);
        sleep(10);
    }
}

```



# 守护进程运行



```
Serial-COM1
Get child 247
/project # ./waitpid
Get child 249
/project # ./dameon
/project # ls
dameon      execlp      fork        process.c~
dameon.c    execlp.c    fork.c      waitpid
dameon.c~   execlp.c~   fork.c~     waitpid.c
execle      exit        hellomodule.ko waitpid.c~
execle.c    exit.c      process
execle.c~   exit.c~     process.c
/project # cd /tmp
/tmp # ls
dameon.log
/tmp # vi dameon.log

This is a Dameon
^@This is a Dameon
^@This is a Dameon
^@This is a Dameon
^@This is a Dameon
^@
~
~
~
~

就绪      Serial : COM1      26,  8      26 行, 62 列 VT100      姜
```

## 3.5 进程间通信

- （1）管道（**Pipe**）及有名管道（**named pipe**）：管道可用于具有亲缘关系进程间的通信，有名管道，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。
- （2）信号（**Signal**）：信号是在软件层次上对中断机制的一种模拟，它是比较复杂的通信方式，用于通知接受进程有某事件发生，一个进程收到一个信号与处理器收到一个中断请求效果上可以说是一样的。
- （3）消息队列：消息队列是消息的链接表，包括 **Posix** 消息队列 **systemV** 消息队列。它克服了前两种通信方式中信息量有限的缺点，具有写权限的进程可以向消息队列中按照一定的规则添加新消息；对消息队列有读权限的进程则可以从消息队列中读取消息。
- （4）共享内存：可以说这是最有用的进程间通信方式。它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。这种通信方式需要依靠某种同步机制，如互斥锁和信号量等。
- （5）信号量：主要作为进程间以及同一进程不同线程之间的同步手段。
- （6）套接字（**Socket**）：这是一种更为一般的进程间通信机制，它可用于不同机器之间的进程间通信，应用非常广泛。

# (1) 管道通信

```
Serial-COM1 - SecureCRT
文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)

Serial-COM1
/tmp #
/tmp # ps
  PID USER      VSZ STAT COMMAND
    1 root        1960 S    init
    2 root          0 SWN   [ksoftirqd/0]
    3 root          0 SW<   [events/0]
    4 root          0 SW<   [khelper]
    5 root          0 SW<   [kthread]
   14 root          0 SW<   [kblockd/0]
   18 root          0 SW<   [khubd]
   57 root          0 SW    [pdflush]
   58 root          0 SW    [pdflush]
   60 root          0 SW<   [aio/0]
   59 root          0 SW    [kswapd0]
  134 root          0 SW<   [kseriod]
  196 root          0 SW    [mtdblockd]
  219 root          0 SW<   [kpsmoused]
  228 root          0 SW    [mncqd]
  230 root          0 SW<   [rpciod/0]
  240 root        1980 S    -/bin/sh
  251 root        1456 S    ./dameon
  256 root        1972 R    ps
/tmp # ps|grep init
    1 root        1960 S    init
  258 root        1964 S    grep init
/tmp #
```

就绪 Serial: COM1 26, 8 26 行, 62 列 VT100

- 管道是 **Linux** 中进程间通信的一种方式。这里所说的管道主要指无名管道，它具有如下特点。
- 它只能用于具有亲缘关系的进程之间的通信（也就是父子进程或者兄弟进程之间）。
- 它是一个半双工的通信模式，具有固定的读端和写端。
- 管道也可以看成是一种特殊的文件，对于它的读写也可以使用普通的 **read**、**write** 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

# 创建管道

- 调用形式:

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

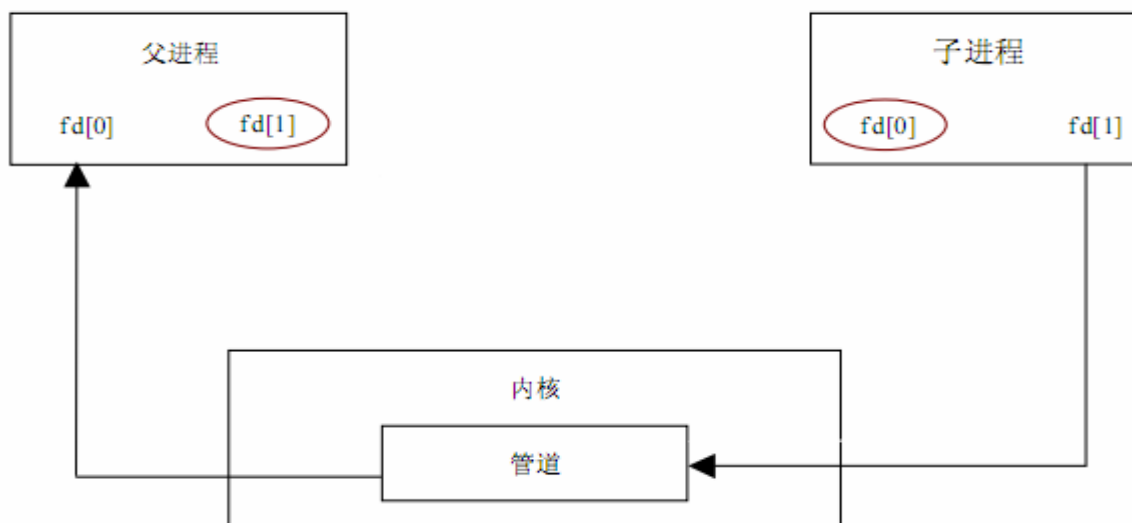
```
int pipe(int fd[2])
```

该函数创建的管道的两端处于一个进程中间，在实际应用中没有太大意义，因此，一个进程在由pipe()创建管道后，一般再fork一个子进程，然后通过管道实现父子进程间的通信

管道的读写规则：— 管道两端可分别用描述字fd[0]以及fd[1]来描述，需要注意的是，管道的两端是固定了任务的。即一端只能用于读，由描述字fd[0]表示，称其为管道读端；另一端则只能用于写，由描述字fd[1]来表示，称其为管道写端。

- 如果试图从管道写端读取数据，或者向管道读端写入数据都将导致错误发生。
- 一般文件的I/O函数都可以用于管道，如close、read、write等等。

# 管道读写说明



# 管道读写实例

```
/*pipe_rw.c*/
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int pipe_fd[2];
    pid_t pid;
    char buf_r[100];
    char* p_wbuf;
    int r_num;
    memset(buf_r,0,sizeof(buf_r));
    /*创建管道*/
    if(pipe(pipe_fd)<0)
    {
        printf("pipe create error\n");
        return -1;
    }
    /*创建一子进程*/
```

博芯电子

**Prochip**  
Your Embedded Partner

```

if((pid=fork())==0)
{
    printf("\n");
    /*关闭子进程写描述符，并通过使父进程暂停2秒确保父进程已关闭相应的读描述符*/
    close(pipe_fd[1]);
    sleep(2);
    /*子进程读取管道内容*/
    if((r_num=read(pipe_fd[0],buf_r,100))>0){
        printf("%d numbers read from the pipe is
%s\n",r_num,buf_r);
    }
    /*关闭子进程读描述符*/
    close(pipe_fd[0]);
    exit(0);
}
else if(pid>0)
{
    /*关闭父进程读描述符，并分两次向管道中写入Hello Pipe*/
    close(pipe_fd[0]);
    if(write(pipe_fd[1],"Hello",5)!= -1)
        printf("parent write1 success!\n");
    if(write(pipe_fd[1]," Pipe",5)!= -1)
        printf("parent write2 success!\n");
    /*关闭父进程写描述符*/
    close(pipe_fd[1]);
    sleep(3);
    /*收集子进程退出信息*/
    waitpid(pid,NULL,0);
    exit(0);
}
}

```



# 管道的局限性

- 管道的主要局限性正体现在它的特点上：
  - 只支持单向数据流；
  - 只能用于具有亲缘关系的进程之间；
  - 没有名字；
  - 管道的缓冲区是有限的（管道制存在于内存中，在管道创建时，为缓冲区分配一个页面大小）；
  - 管道所传送的是无格式字节流，这就要求管道的读出方和写入方必须事先约定好数据的格式，比如多少字节算作一个消息（或命令、或记录）等等

# FIFO

- 管道应用的一个重大限制是它没有名字，因此，只能用于具有亲缘关系的进程间通信，在命名管道（**namedpipe**或**FIFO**）提出后，该限制得到了克服。
- **FIFO**不同于管道之处在于它提供一个路径名与之关联，以**FIFO**的文件形式存在于文件系统中。这样，即使与**FIFO**的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过**FIFO**相互通信（能够访问该路径的进程以及**FIFO**的创建进程之间），因此，通过**FIFO**不相关的进程也能交换数据。值得注意的是，**FIFO**严格遵循先进先出（**firstinfirstout**），对管道及**FIFO**的读总是从开始处返回数据，对它们的写则把数据添加到末尾。它们不支持诸如**lseek()**等文件定位操作。

- 系统调用形式:

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

```
#include<sys/stat.h>
```

```
int mkfifo(const char*pathname,mode_t mode)
```

- 该函数的第一个参数是一个普通的路径名，也就是创建后FIFO的名字。第二个参数与打开普通文件的open()函数中的mode参数相同。
- 一般文件的I/O函数都可以用于FIFO，如close、read、write等等。

O\_RDONLY:读管道  
O\_WRONLY:写管道  
O\_RDWR:读写管道  
O\_NONBLOCK:非阻塞  
O\_CREAT:  
O\_EXCL:

# FIFO读规则

- 约定：如果一个进程为了从FIFO中读取数据而阻塞打开FIFO，那么称该进程内的读操作为设置了阻塞标志的读操作。
- 如果有进程写打开FIFO，且当前FIFO内没有数据，则对于设置了阻塞标志的读操作来说，将一直阻塞。对于没有设置阻塞标志读操作来说则返回-1，当前errno值为EAGAIN，提醒以后再试。
- 对于设置了阻塞标志的读操作说，造成阻塞的原因有两种：
  - A.当前FIFO内有数据，但有其它进程在读这些数据；
  - B.另外就是FIFO内没有数据。解阻塞的原因则是FIFO中有新的数据写入，不论信写入数据量的大小，也不论读操作请求多少数据量。
- 读打开的阻塞标志只对本进程第一个读操作施加作用，如果本进程内有多个读操作序列，则在第一个读操作被唤醒并完成读操作后，其它将要执行的读操作将不再阻塞，即使在执行读操作时，FIFO中没有数据也一样（此时，读操作返回0）。
- 如果没有进程写打开FIFO，则设置了阻塞标志的读操作会阻塞。

# FIFO写规则

- 约定：如果一个进程为了向FIFO中写入数据而阻塞打开FIFO，那么称该进程内的写操作为设置了阻塞标志的写操作。
- 对于设置了阻塞标志的写操作：
  - 当要写入的数据量不大于PIPE\_BUF时，linux将保证写入的原子性。如果此时管道空闲缓冲区不足以容纳要写入的字节数，则进入睡眠，直到当缓冲区中能够容纳要写入的字节数时，才开始进行一次性写操作。
  - 当要写入的数据量大于PIPE\_BUF时，linux将不再保证写入的原子性。FIFO缓冲区一有空闲区域，写进程就会试图向管道写入数据，写操作在写完所有请求写的数据后返回。
- 对于没有设置阻塞标志的写操作：
  - 当要写入的数据量大于PIPE\_BUF时，linux将不再保证写入的原子性。在写满所有FIFO空闲缓冲区后，写操作返回。
  - 当要写入的数据量不大于PIPE\_BUF时，linux将保证写入的原子性。如果当前FIFO空闲缓冲区能够容纳请求写入的字节数，写完后成功返回；如果当前FIFO空闲缓冲区不能够容纳请求写入的字节数，则返回EAGAIN错误，提醒以后再写；

```

/*fifl_read.c*/
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define FIFO "/tmp/myfifo"

main(int argc,char** argv)
{
    char buf_r[100];
    int fd;
    int nread;
    /*创建有名管道，并设置相应的权限*/
    if((mkfifo(FIFO,O_CREAT|O_EXCL)<0)&&(errno!=EEXIST))
        printf("cannot create fifoserver\n");
    printf("Preparing for reading bytes...\n");
    memset(buf_r,0,sizeof(buf_r));
    /*打开有名管道，并设置非阻塞标志*/
    fd=open(FIFO,O_RDONLY|O_NONBLOCK,0);
    if(fd== -1)
    {
        perror("open");
        exit(1);
    }
    while(1)
    {
        memset(buf_r,0,sizeof(buf_r));
        if((nread=read(fd,buf_r,100))== -1){
            if(errno==EAGAIN)
                printf("no data yet\n");
        }
        printf("read %s from FIFO\n",buf_r);
        sleep(1);
    }
    pause();
    unlink(FIFO);
}

```

```

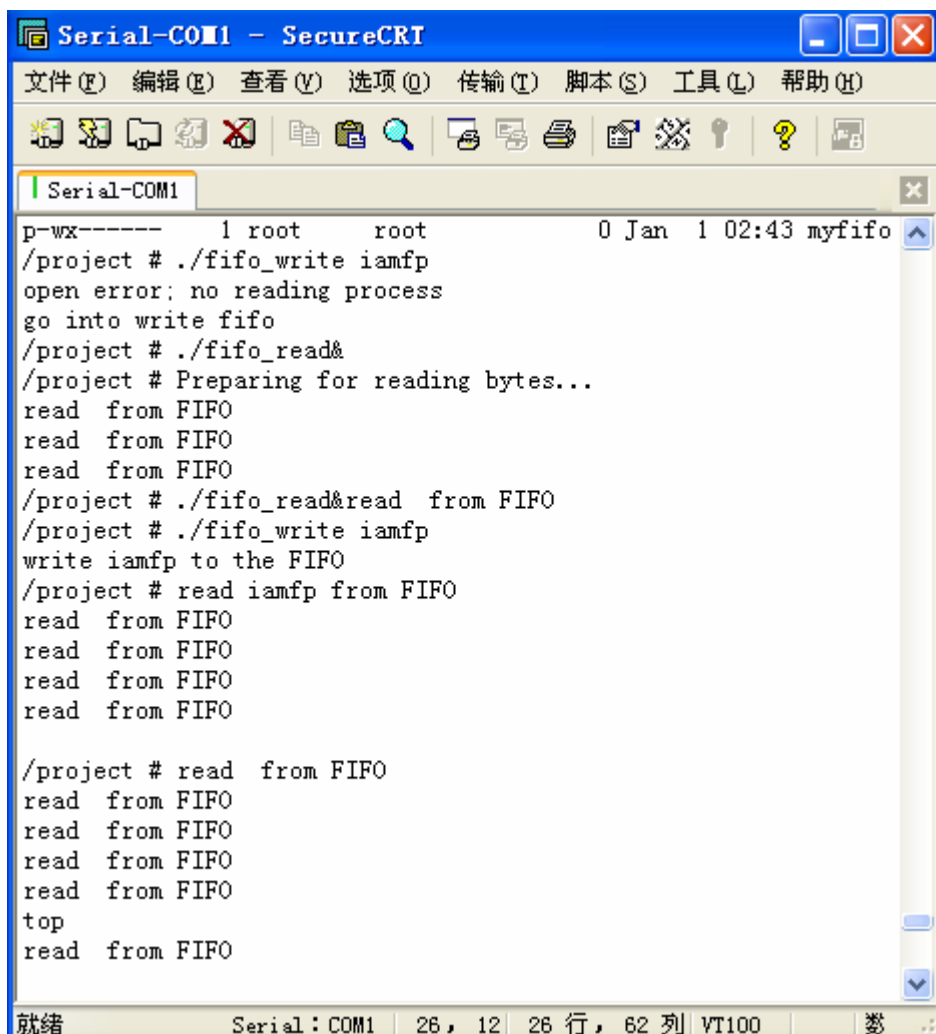
/*fifo_write.c*/
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define FIFO "/tmp/myfifo"

main(int argc,char** argv)
/*参数为即将写入的字节数*/
{
    int fd;
    char w_buf[100];
    int nwrite;

    /*打开FIFO管道，并设置非阻塞标志*/
    fd=open(FIFO,O_WRONLY|O_NONBLOCK,0);
    if(fd== -1)
        printf("open error; no reading process\n");
    if(argc==1)
        printf("Please send something\n");
    strcpy(w_buf,argv[1]);
    /*向管道中写入字符串*/
    if((nwrite=write(fd,w_buf,100))== -1)
    {
        printf("go into write fifo\n");
        if(errno==EAGAIN)
            printf("The FIFO has not been read yet.Please try later\n");
    }
    else
        printf("write %s to the FIFO\n",w_buf);
}

```

# FIFO运行结果



```
Serial-COM1 - SecureCRT
文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)
Serial-COM1
p-wx----- 1 root root 0 Jan 1 02:43 myfifo
/project # ./fifo_write iamfp
open error; no reading process
go into write fifo
/project # ./fifo_read&
/project # Preparing for reading bytes...
read from FIFO
read from FIFO
read from FIFO
/project # ./fifo_read&read from FIFO
/project # ./fifo_write iamfp
write iamfp to the FIFO
/project # read iamfp from FIFO
read from FIFO
read from FIFO
read from FIFO
read from FIFO

/project # read from FIFO
read from FIFO
read from FIFO
read from FIFO
read from FIFO
top
read from FIFO
就绪 Serial: COM1 26, 12 26 行, 62 列 VT100
```

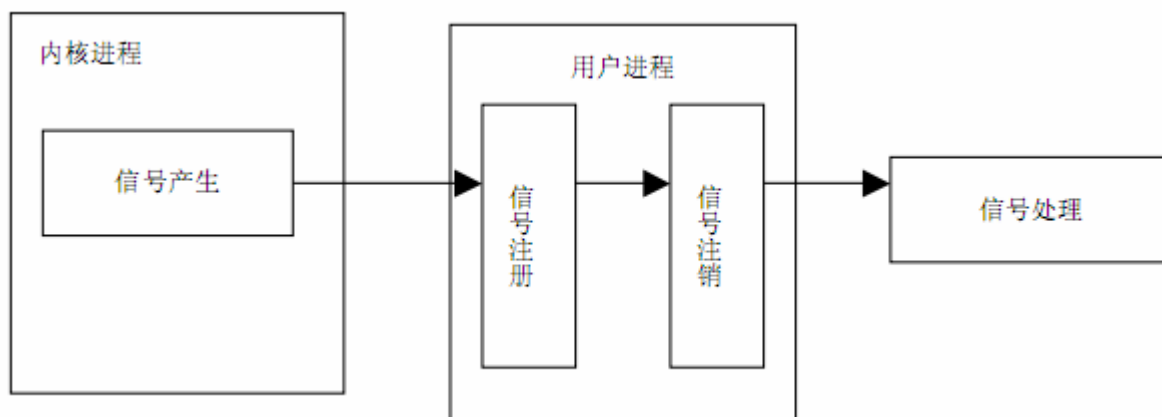


## (2) 信号通信

- 信号(Signal,亦称作软中断)机制是在软件层次上对中断机制的一种模拟。异步进程可以通过彼此发送信号来实现简单通信。系统预先规定若干个不同类型的信号(如x86平台中Linux内核设置了32种信号,而现在的Linux和POSIX.4定义了64种信号),各表示发生了不同的事件,每个信号对应一个编号。运行进程当遇到相应事件或出现特定要求时(如进程终止或运行中出现某些错误—非法指令、地址越界等),就把一个信号写到相应进程task\_struct结构的Signal位图(表示信号的整数)中。接收信号的进程在运行过程中要检测自身是否收到了信号,如果已收到信号,则转去执行预先规定好的信号处理程序。处理之后,再返回原先正在执行的程序。
- 信号可以直接进行用户空间进程和内核进程之间的交互,内核进程也可以利用它来通知用户空间进程发生了哪些系统事件。它可以在任何时候发给某一进程,而无需知道该进程的状态。如果该进程当前并未处于执行态,则该信号就由内核保存起来,直到该进程恢复执行再传递给它为止;如果一个信号被进程设置为阻塞,则该信号的传递被延迟,直到其阻塞被取消时才被传递给进程。

# 信号的三个阶段

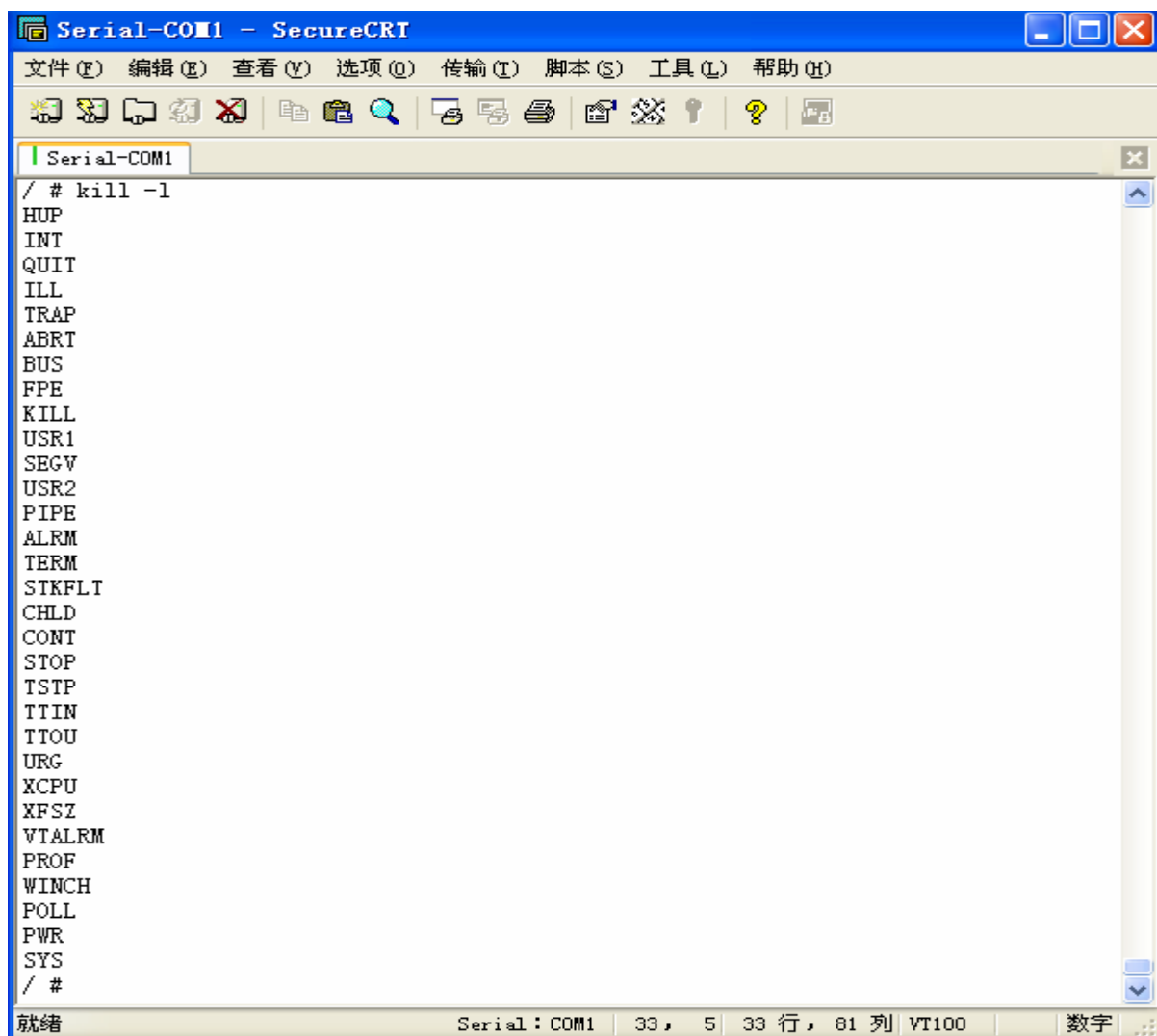
- 一个完整的信号生命周期可以分为 3 个重要阶段，这 3 个阶段由 4 个重要事件来刻画的：信号产生、信号在进程中注册、信号在进程中注销、执行信号处理函数。



# 信号的处理

- 用户进程对信号的响应可以有3种方式。
  - 忽略信号，即对信号不做任何处理，但是有两个信号不能忽略，即 **SIGKILL** 及**SIGSTOP**。
  - 捕捉信号，定义信号处理函数，当信号发生时，执行相应的处理函数。
  - 执行缺省操作，Linux对每种信号都规定了默认操作。

# 信号的分类



The image shows a screenshot of a SecureCRT terminal window titled "Serial-COM1 - SecureCRT". The window has a menu bar with options: 文件(F), 编辑(E), 查看(V), 选项(O), 传输(T), 脚本(S), 工具(L), 帮助(H). Below the menu bar is a toolbar with various icons. The terminal window has a tab labeled "Serial-COM1". The main text area displays a list of Unix signals, starting with a prompt character "/ #". The signals listed are: kill -1, HUP, INT, QUIT, ILL, TRAP, ABRT, BUS, FPE, KILL, USR1, SEGV, USR2, PIPE, ALRM, TERM, STKFLT, CHLD, CONT, STOP, TSTP, TTIN, TTOU, URG, XCPU, XFSZ, VTALRM, PROF, WINCH, POLL, PWR, SYS, and another prompt character "/ #". The status bar at the bottom of the window shows "就绪" (Ready) on the left, and "Serial : COM1 33, 5 33 行, 81 列 VT100 数字" (Serial : COM1 33, 5 33 lines, 81 columns VT100 Numeric) on the right.

```
/ # kill -1
HUP
INT
QUIT
ILL
TRAP
ABRT
BUS
FPE
KILL
USR1
SEGV
USR2
PIPE
ALRM
TERM
STKFLT
CHLD
CONT
STOP
TSTP
TTIN
TTOU
URG
XCPU
XFSZ
VTALRM
PROF
WINCH
POLL
PWR
SYS
/ #
```

就绪 Serial : COM1 33, 5 33 行, 81 列 VT100 数字

# Kill, raise, alarm和pause

函数调用形式:

```
#include<signal.h>
```

```
int kill(pid_t pid,int sig);
```

```
int raise(int sig);
```

```
unsigned int alarm(unsigned int seconds);
```

```
int pause(void);
```

作用: kill系统调用负责向进程发送信号sig;

raise系统调用向自己发送一个sig信号, 可以用kill函数来实现这个功能的; alarm函数和时间有点关系了, 这个函数可以在seconds秒后向自己发送一个SIGALRM信号; pause用于将调用进程挂起直到捕捉到信号为止。

参数:

如果pid是正数, 那么信号sig被发送到进程pid.

如果pid等于0, 那么信号sig被发送到和pid进程在同一个进程组的进程

如果pid等于-1, 那么信号发给所有的进程表中的进程, 除了最大的哪个进程号.

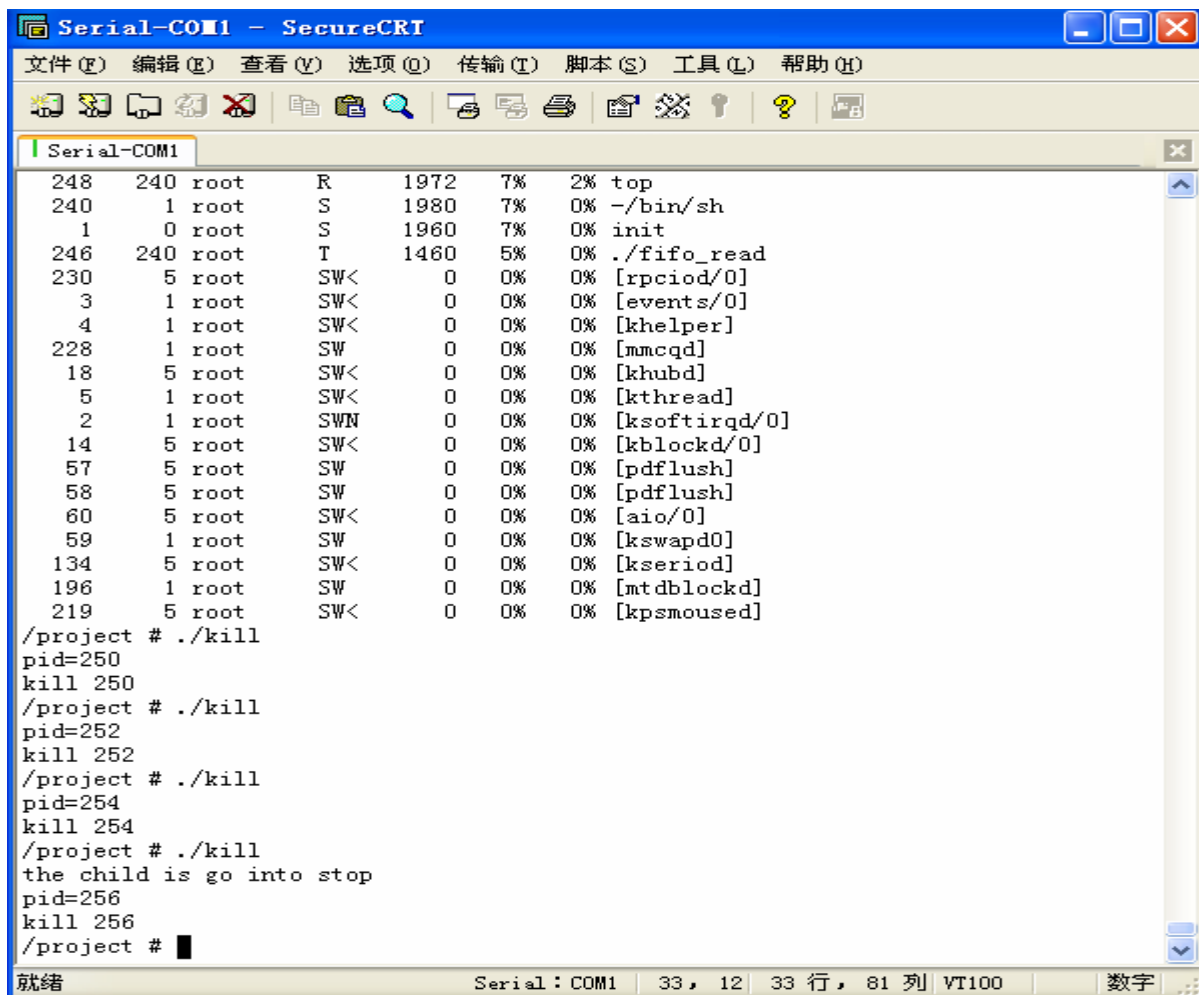
```

/*kill.c*/
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t pid;
    int ret;
    /*创建一子进程*/
    if((pid=fork())<0){
        perror("fork");
        exit(1);
    }
    if(pid == 0){
        /*在子进程中使用raise函数发出SIGSTOP信号*/
        printf("the child is go into stop\n");
        raise(SIGSTOP);
        printf("the child is sleep\n");
        exit(0);
    }
    else{
        /*在父进程中收集子进程发出的信号，并调用kill函数进行相应的操作*/
        sleep(3);
        printf("pid=%d\n",pid);
        if((waitpid(pid,NULL,WNOHANG))==0){
            if((ret=kill(pid,SIGKILL))==0)
                printf("kill %d\n",pid);
            else{
                perror("kill");
            }
        }
    }
}

```

# Kill运行结果



```
Serial-COM1 - SecureCRT
文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)
Serial-COM1
248 240 root R 1972 7% 2% top
240 1 root S 1980 7% 0% -/bin/sh
1 0 root S 1960 7% 0% init
246 240 root T 1460 5% 0% ./fifo_read
230 5 root SW< 0 0% 0% [rpciod/0]
3 1 root SW< 0 0% 0% [events/0]
4 1 root SW< 0 0% 0% [khelper]
228 1 root SW 0 0% 0% [mmcqd]
18 5 root SW< 0 0% 0% [khubd]
5 1 root SW< 0 0% 0% [kthread]
2 1 root SWN 0 0% 0% [ksoftirqd/0]
14 5 root SW< 0 0% 0% [kblockd/0]
57 5 root SW 0 0% 0% [pdflush]
58 5 root SW 0 0% 0% [pdflush]
60 5 root SW< 0 0% 0% [aio/0]
59 1 root SW 0 0% 0% [kswapd0]
134 5 root SW< 0 0% 0% [kseriod]
196 1 root SW 0 0% 0% [mtdblockd]
219 5 root SW< 0 0% 0% [kpsmouse]
/project # ./kill
pid=250
kill 250
/project # ./kill
pid=252
kill 252
/project # ./kill
pid=254
kill 254
/project # ./kill
the child is go into stop
pid=256
kill 256
/project #
就绪 Serial: COM1 33, 12 33 行, 81 列 VT100 数字
```

博芯电子

**Prochip**  
Your Embedded Partner

# 信号的处理

- 从前面的信号概述中我们可以看到，特定的信号是与一定的进程相联系的。也就是说，一个进程可以决定在该进程中需要对哪些信号进行什么样的处理。例如，一个进程可以选择忽略某些信号而只处理其他一些信号，另外，一个进程还可以选择如何处理信号。总之，这些都是与特定的进程相联系的。因此，首先就要建立其信号与进程之间的对应关系，这就是信号的处理。
- 信号处理的主要方法有两种，一种是使用简单的 `signal` 函数，另一种是使用信号集函数组。



# signal ( )

- #include <signal.h>
- void (\*signal(int signum, void (\*handler)(int)))(int)

参数： signum： 指定信号

Handler： 处理函数句柄

SIG\_IGN： 忽略该信号

SIG\_DFL： 采用系统默认方式处理信号

自定义的信号处理函数指针

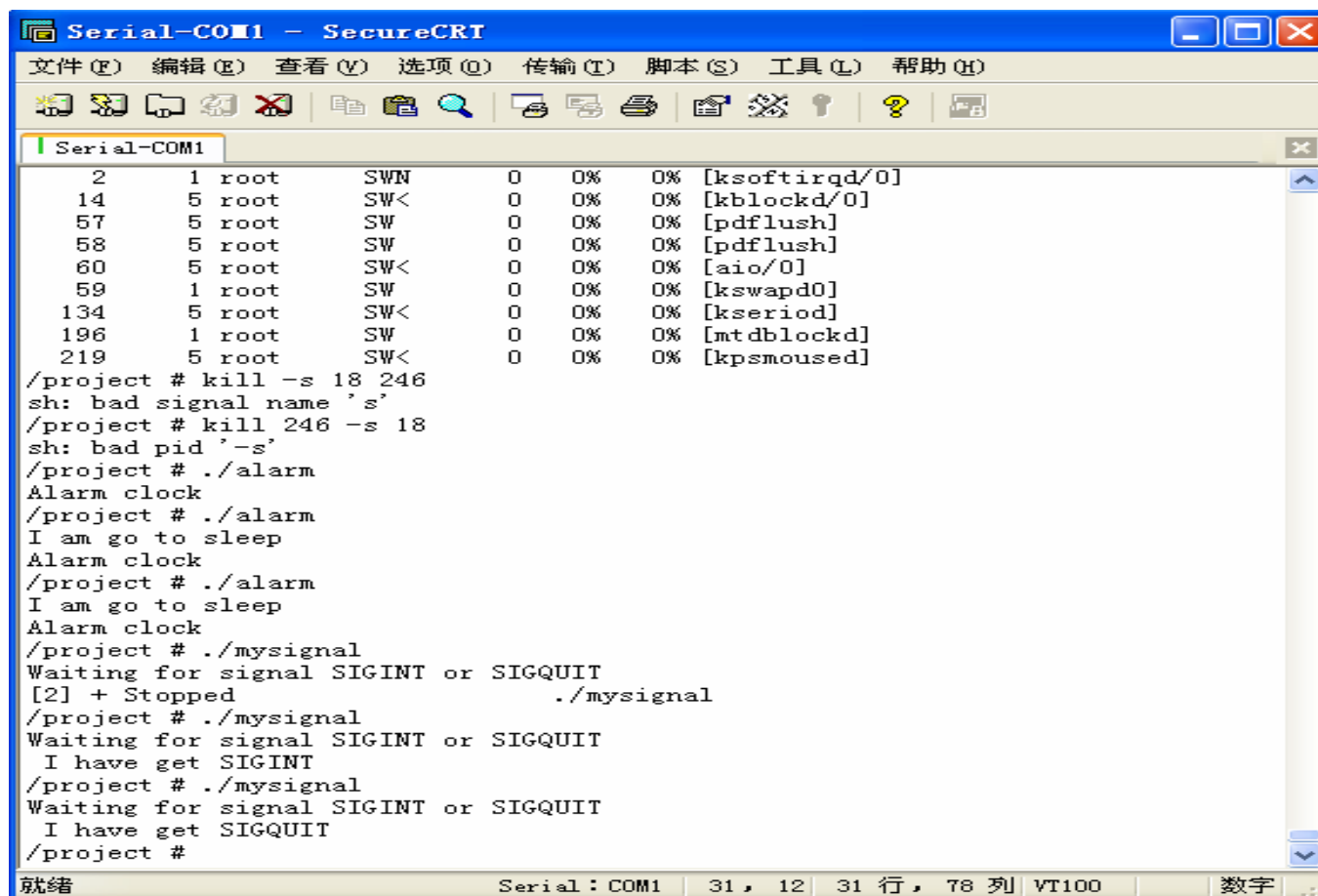
- 函数返回值： 成功时返回以前的信号处理配置， 出错时返回-1。

```

/*mysignal.c*/
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
/*自定义信号处理函数*/
void my_func(int sign_no)
{
    if(sign_no==SIGINT)
        printf("I have get SIGINT\n");
    else if(sign_no==SIGQUIT)
        printf("I have get SIGQUIT\n");
}
int main()
{
    printf("Waiting for signal SIGINT or SIGQUIT \n ");
    /*发出相应的信号，并跳转到信号处理函数处*/
    signal(SIGINT, my_func);
    signal(SIGQUIT, my_func);
    pause();
    exit(0);
}

```

# Signal运行



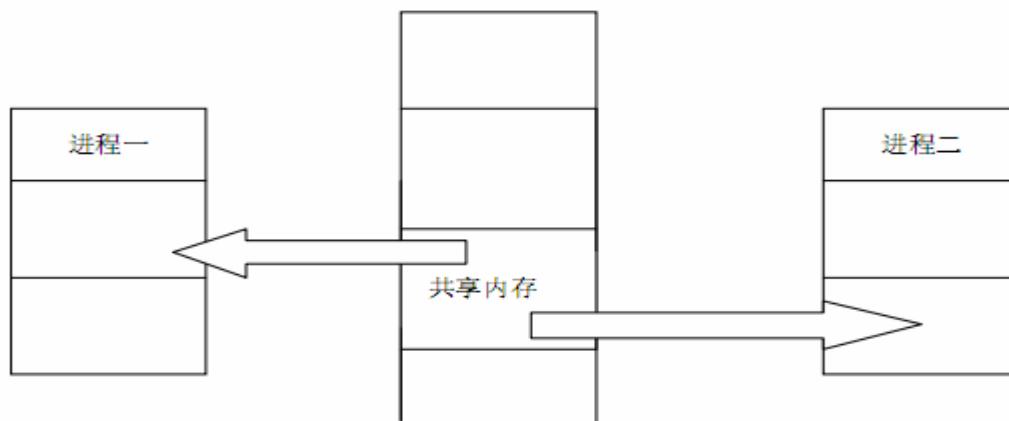
```
Serial-COM1 - SecureCRT
文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)

Serial-COM1
2      1 root    SWN      0      0%      0% [ksoftirqd/0]
14     5 root    SW<      0      0%      0% [kblockd/0]
57     5 root    SW      0      0%      0% [pdflush]
58     5 root    SW      0      0%      0% [pdflush]
60     5 root    SW<      0      0%      0% [aio/0]
59     1 root    SW      0      0%      0% [kswapd0]
134    5 root    SW<      0      0%      0% [kseriod]
196    1 root    SW      0      0%      0% [mtdblockd]
219    5 root    SW<      0      0%      0% [kpsmoused]
/project # kill -s 18 246
sh: bad signal name 's'
/project # kill 246 -s 18
sh: bad pid '-s'
/project # ./alarm
Alarm clock
/project # ./alarm
I am go to sleep
Alarm clock
/project # ./alarm
I am go to sleep
Alarm clock
/project # ./mysignal
Waiting for signal SIGINT or SIGQUIT
[2] + Stopped                  ./mysignal
/project # ./mysignal
Waiting for signal SIGINT or SIGQUIT
I have get SIGINT
/project # ./mysignal
Waiting for signal SIGINT or SIGQUIT
I have get SIGQUIT
/project #

就绪      Serial: COM1      31, 12      31 行, 78 列      VT100      数字
```

### (3) 共享内存

- 共享内存可以说是最有用的进程间通信方式，也是最快的IPC形式。两个不同进程A、B共享内存的意思是，同一块物理内存被映射到进程A、B各自的进程地址空间。进程A可以即时看到进程B对共享内存中数据的更新，反之亦然。
- 由于多个进程共享同一块内存区域，必然需要某种同步机制，互斥锁和信号量都可以。



# 共享内存的好处

- 采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域，而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

# 共享内存的创建与打开

```
int shmget(key_t key,int size,int shmflg);
```

参数含义:

- 系统调用shmget()中的第一个参数是关键字值（它是用系统调用ftok()返回的）。
- 其他的操作都要依据shmflg中的命令进行。
- 当IPC\_CREAT单独使用时，系统调用shmget()要么返回一个新创建的共享内存段的标识符，要么返回一个已经存在的共享内存段的关键字值。
- 如果IPC\_EXCL和IPC\_CREAT一同使用，则要么系统调用新建一个共享的内存段，要么返回一个错误值-1。IPC\_EXCL单独使用没有意义。

返回值:

- 如果成功，返回共享内存段标识符；
- 如果失败，则返回-1

# 映射共享内存

- `int shmat(int shmid, char* shmaddr, int shmflg);`
- 返回值：如果成功，则返回共享内存段连接到进程中的地址。如果失败，则返回-1
- 如果参数`addr`的值为0，那么系统内核则试图找出一个没有映射的内存区域

# 撤销映射

- `int shmdt(char* shmaddr);`
- 当一个进程不在需要共享的内存段时，它将会把内存段从其地址空间中脱离。但这不等于将共享内存段从系统内核中移走。当进程脱离成功后，数据结构`shmid_ds`中元素`shm_nattch`将减1。当此数值减为0以后，系统内核将物理上把内存段从系统内核中移走。



# 共享内存的控制

```
int shmctl(int shmqid,int cmd,struct shmid_ds *buf);
```

cmd参数可以为下列值之一：

- IPC\_STAT读取一个内存段的数据结构shmid\_ds，并将它存储在buf参数指向的地址中。
- IPC\_SET设置内存段的数据结构shmid\_ds中的元素ipc\_perm的值。从参数buf中得到要设置的值。
- IPC\_RMID标志内存段为移走。命令IPC\_RMID并不真正从系统内核中移走共享的内存段，而是把内存段标记为可移除。进程调用系统调用shmdt()脱离一个共享的内存段。

```

/*****write_shm.c*****/
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<unistd.h>
typedef struct{
    char name[4];
    int age;
}people;
int main(int argc,char** argv)
{
    int shm_id,i; key_t key;
    char temp;
    people *p_map;
    char* name="/tmp/fp";
    key=ftok(name,0);
    if(key== -1)
        perror("ftokerror");
    shm_id=shmget(key,4096,IPC_CREAT);
    p_map=(people*)shmat(shm_id,NULL,0);
    temp='a';
    for(i=0;i<10;i++)
    {
        memcpy((*p_map+i).name,&temp,1);
        (*p_map+i).age=20+i;
        temp+=1;
    }
    if(shmdt(p_map)==-1)
        perror("detacherror");
}

```

博芯电子

**Prochip**  
Your Embedded Partner

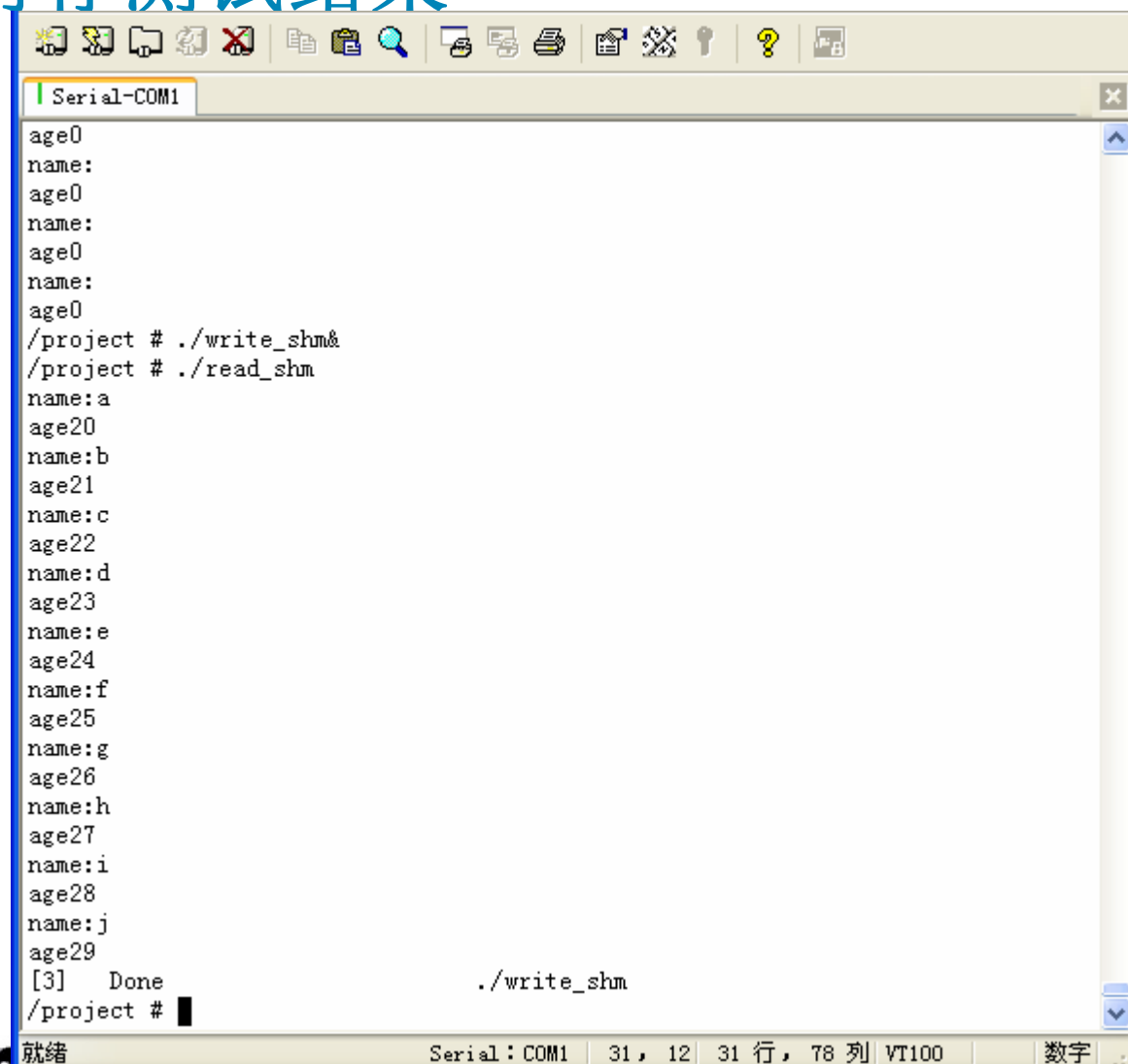
```

/*****read_shm.c*****/
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<unistd.h>
typedef struct{
    char name[4];
    int age;
}people;

int main(int argc,char** argv)
{
    int shm_id,i; key_t key; people* p_map;
    char*name="/tmp/fp";
    key=ftok(name,0);
    if(key==-1)
        perror("ftokerror");
    shm_id=shmget(key,4096,IPC_CREAT);
    p_map=(people*)shmat(shm_id,NULL,0);
    for(i=0;i<10;i++)
    {
        printf("name:%s\n",(*(p_map+i)).name);
        printf("age%d\n",(*(p_map+i)).age);
    }
    if(shmdt(p_map)==-1)
        perror("detacherror");
}

```

# 共享内存测试结果



The screenshot shows a terminal window titled "Serial-COM1" with a standard Windows-style toolbar at the top. The terminal displays the output of a shared memory test. It lists memory addresses from age0 to age29, each followed by a "name:" label. The names for addresses age0 through age20 are all "age0". From age21 onwards, the names are sequential letters from "a" to "j". After the list, there are two prompts: "/project # ./write\_shm&" and "/project # ./read\_shm". The output shows "[3] Done" followed by a space and then "./write\_shm". The status bar at the bottom of the window indicates "就绪" (Ready), "Serial: COM1", "31, 12", "31 行, 78 列" (31 lines, 78 columns), "VT100", and "数字" (Numeric).

```
Serial-COM1
age0
name:
age0
name:
age0
name:
age0
name:
age0
/project # ./write_shm&
/project # ./read_shm
name:a
age20
name:b
age21
name:c
age22
name:d
age23
name:e
age24
name:f
age25
name:g
age26
name:h
age27
name:i
age28
name:j
age29
[3] Done      ./write_shm
/project #
```

就绪 Serial: COM1 31, 12 31 行, 78 列 VT100 数字

## (4) 消息队列

- 消息队列就是一个消息的链表。可以把消息看作一个记录，具有特定的格式以及特定的优先级。对消息队列有写权限的进程可以向中按照一定的规则添加新消息；对消息队列有读权限的进程则可以从消息队列中读走消息。
- 消息是由固定大小的首部和可变长度的正文组成的；可以使用一个整数值（消息类型）标识消息，因此可以允许进程有选择的从消息队列中获取消息。只要进程从消息队列中读出消息，内核就会将该消息删除，因此一个进程只能接收一条给定的消息。
- 消息可以按照非“先进先出”的次序获得

# 消息队列的创建与打开

- 要使用消息队列，首先要创建一个消息队列。创建

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

```
#include<sys/ipc.h>
```

```
#include<sys/msg.h>
```

```
int msgget(key_t key,int msgflg),
```

返回值:

- 如果成功，返回消息队列标识符
- 如果失败，则返回-1

在以下两种情况下，该调用将创建一个新的消息队列:

- 如果没有消息队列与键值key相对应，并且msgflg中包含了IPC\_CREAT标志位;
- key参数为IPC\_PRIVATE;

IPC\_CREAT、  
IPC\_EXCL、  
IPC\_NOWAIT  
或三者或结果

# 向消息队列中发送消息

- 原型:

```
int msgsnd(int msqid, struct msgbuf* msgp, int msgsz, int msgflg);
```

- 参数含义:

- msqid参数是消息队列标识符，它是由系统调用msgget返回的;
- msgp参数是指向消息缓冲区的指针;
- 参数msgsz中包含的是消息的字节大小，但不包括消息类型的长度（4个字节）;
- 参数msgflg可以设置为0（此时为忽略此参数），或者使用IPC\_NOWAIT。

- 返回值:

- 如果成功，返回值为0;
- 如果失败，返回值为-1

# 从消息队列接收消息

- 函数形式:

`int msgrcv(int msqid, struct msgbuf* msgp, int msgsz, long mtype, int msgflg);`

- 参数含义:

- 第一个参数用来指定将要读取消息的队列;
- 第二个参数代表要存储消息的消息缓冲区的地址;
- 第三个参数是消息缓冲区的长度, 不包括mtype的长度, 它可以按照如下的方法计算:

`msgsz = sizeof(struct mymsgbuf) - sizeof(long);`

- 第四个参数是要从消息队列中读取的消息的类型。如果此参数的值为0, 那么队列中最长时间的一条消息将返回, 而不论其类型是什么。

- 返回值:

- 如果成功, 则返回复制到消息缓冲区的字节数。
- 如果失败, 则返回-1。



# 消息队列控制

- 函数形式:

```
int msgctl(int msgqid,int cmd,struct msqid_ds* buf);
```

- 返回值:

- 如果成功, 返回值为0;

- 如果失败, 返回值为-1

- cmd参数的取值:

- IPC\_STAT 读取消息队列的数据结构msqid\_ds, 并将其存储在buf指定的地址中。

- IPC\_SET 设置消息队列的数据结构msqid\_ds中的ipc\_perm元素的值。这个值取自buf参数。

- IPC\_RMID 从系统内核中移走消息队列。

```

/*read_msg.c*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define BUFSZ 512
struct message{
    long msg_type;
    char msg_text[BUFSZ];
};

int main()
{
    int qid;
    key_t key;
    int len;
    struct message msg;
    /*根据不同的路径和键表示产生标准的key*/
    if((key=ftok(".", 'a'))== -1){
        perror("ftok");
        exit(1);
    }
}

```

博芯电子

**Prochip**  
Your Embedded Partner

```

/*创建消息队列*/
    if((qid=msgget(key,IPC_CREAT|0666))== -1){
        perror("msgget");
        exit(1);
    }
    printf("opened queue %d\n",qid);
    while(1)
    {
/*读取消息队列*/
        if(msgrcv(qid,&msg,BUFSZ,0,0)<0){
            perror("msgrcv");
            exit(1);
        }
        printf("message is:%s\n",&msg->msg_text);
    }
/*从系统内核中移走消息队列。*/
    if((msgctl(qid,IPC_RMID,NULL))<0){
        perror("msgctl");
        exit(1);
    }

    exit(0);
}

```

```

/*write_msg.c*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define BUFSZ 512
struct message{
    long msg_type;
    char msg_text[BUFSZ];
};

int main()
{
    int qid;
    key_t key;
    int len;
    struct message msg;
    /*根据不同的路径和关键表示产生标准的key*/
    if((key=ftok(".", 'a'))== -1){
        perror("ftok");
        exit(1);
    }
}

```

```

/*创建消息队列*/
if((qid=msgget(key,IPC_CREAT|0666))== -1){
    perror("msgget");
    exit(1);
}
printf("opened queue %d\n",qid);
while(1)
{
    puts("Please enter the message to queue:");
    if((fgets(&msg->msg_text,BUFSZ,stdin))==NULL){
        puts("no message");
        exit(1);
    }
    msg.msg_type = getpid();
    printf("message is:%s\n",&msg->msg_text);
    len = strlen(msg.msg_text);
    /*添加消息到消息队列*/
    if((msgsnd(qid,&msg,len,0))<0){
        perror("message posted");
        exit(1);
    }
}

/*从系统内核中移走消息队列。*/
if((msgctl(qid,IPC_RMID,NULL))<0){
    perror("msgctl");
    exit(1);
}
exit(0);
}

```

# 消息队列实例运行

```
root@localhost:/nfs/project
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)
root@localhost:/linux-3.2 x root@localhost:~/project x root@localhost:/nfs/proj... x
116 root 18 -5 0 0 0 S 0.0 0.0 0:00.06 khubd
118 root 10 -5 0 0 0 S 0.0 0.0 0:00.00 kseriod
141 root 15 0 0 0 0 S 0.0 0.0 0:02.36 pdflush
142 root 15 0 0 0 0 S 0.0 0.0 0:14.22 pdflush
143 root 10 -5 0 0 0 S 0.0 0.0 0:02.16 kswapd0
144 root 18 -5 0 0 0 S 0.0 0.0 0:00.00 aio/0
283 root 11 -5 0 0 0 S 0.0 0.0 0:00.00 kpsmoused
315 root 20 -5 0 0 0 S 0.0 0.0 0:00.00 scsi_eh_0
319 root 10 -5 0 0 0 S 0.0 0.0 0:29.45 ata/0
320 root 20 -5 0 0 0 S 0.0 0.0 0:00.00 ata_aux
323 root 20 -5 0 0 0 S 0.0 0.0 0:00.00 scsi_eh_1
[root@localhost project]# ./read_msg
opened queue 0
message is:fp
message is:sep4020
message is:message
```

```
root@localhost:/nfs/project
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)
46 root 18 -5 0 0 0 S 0.0 0.0 0:00.75 kblockd/0
47 root 20 -5 0 0 0 S 0.0 0.0 0:00.00 kacpid
112 root 20 -5 0 0 0 S 0.0 0.0 0:00.00 cqueue/0
113 root 10 -5 0 0 0 S 0.0 0.0 0:00.00 ksuspend_usbd
116 root 18 -5 0 0 0 S 0.0 0.0 0:00.06 khubd
118 root 10 -5 0 0 0 S 0.0 0.0 0:00.00 kseriod
141 root 15 0 0 0 0 S 0.0 0.0 0:02.36 pdflush
142 root 15 0 0 0 0 S 0.0 0.0 0:14.22 pdflush
[root@localhost project]# ./write_msg
opened queue 0
Please enter the message to queue:
fp
message is:fp
Please enter the message to queue:
sep4020
message is:sep4020
Please enter the message to queue:
message
message is:message
Please enter the message to queue:
█
```

