



# 进程及进程间通讯

---

杨劲松 [yjs@oldhand.org](mailto:yjs@oldhand.org)

2011.02.23

北京亚嵌教育研究中心

©2011 AKAE

# 主要内容

---

- 进程概述
- 进程环境
- 进程控制
- 进程关系
- 守护进程
- 信号
- 进程间通讯

# 1. 进程概述

# 1.1 进程介绍

---

## ■ 进程的定义

- 进程是一个独立的可调度的活动实体
- 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
- 进程是可以并行执行的计算部分

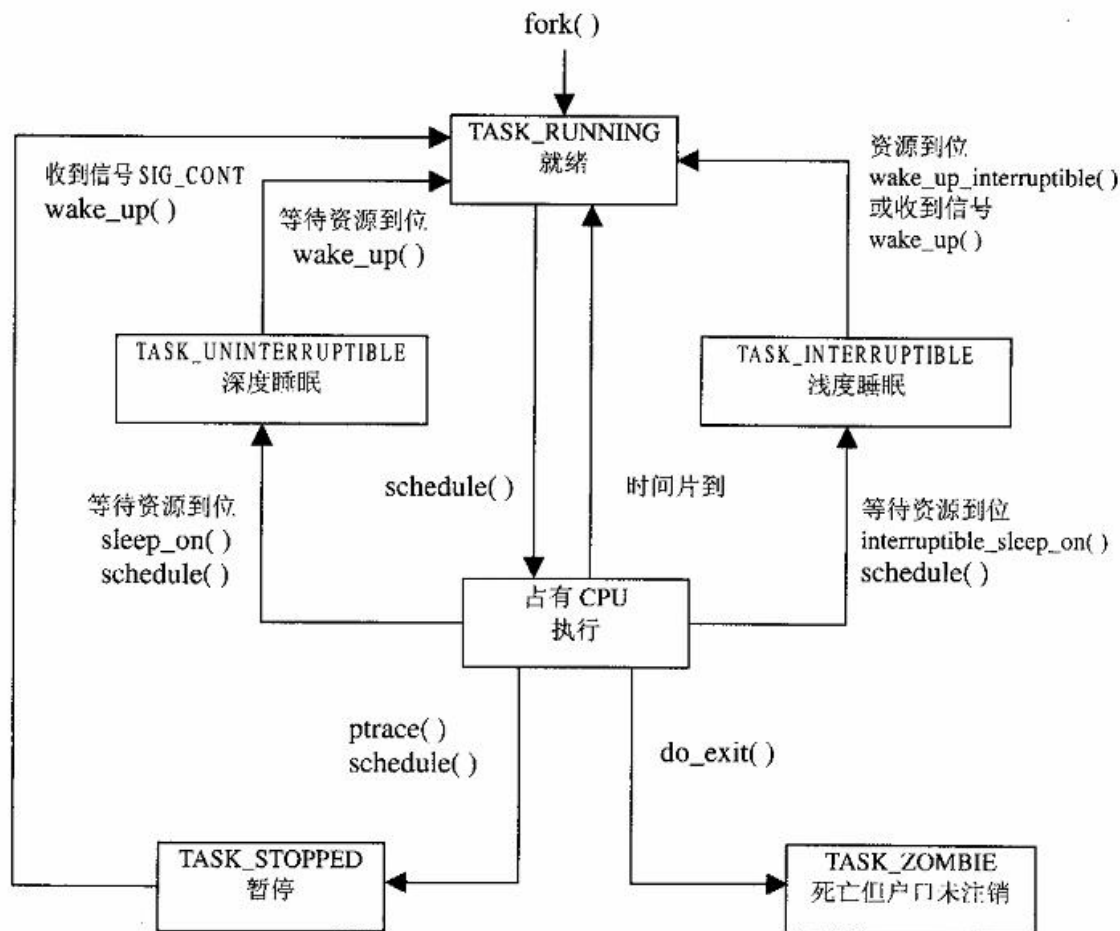
## ■ 进程是一个程序的一次执行的过程

## ■ 进程和程序的区别

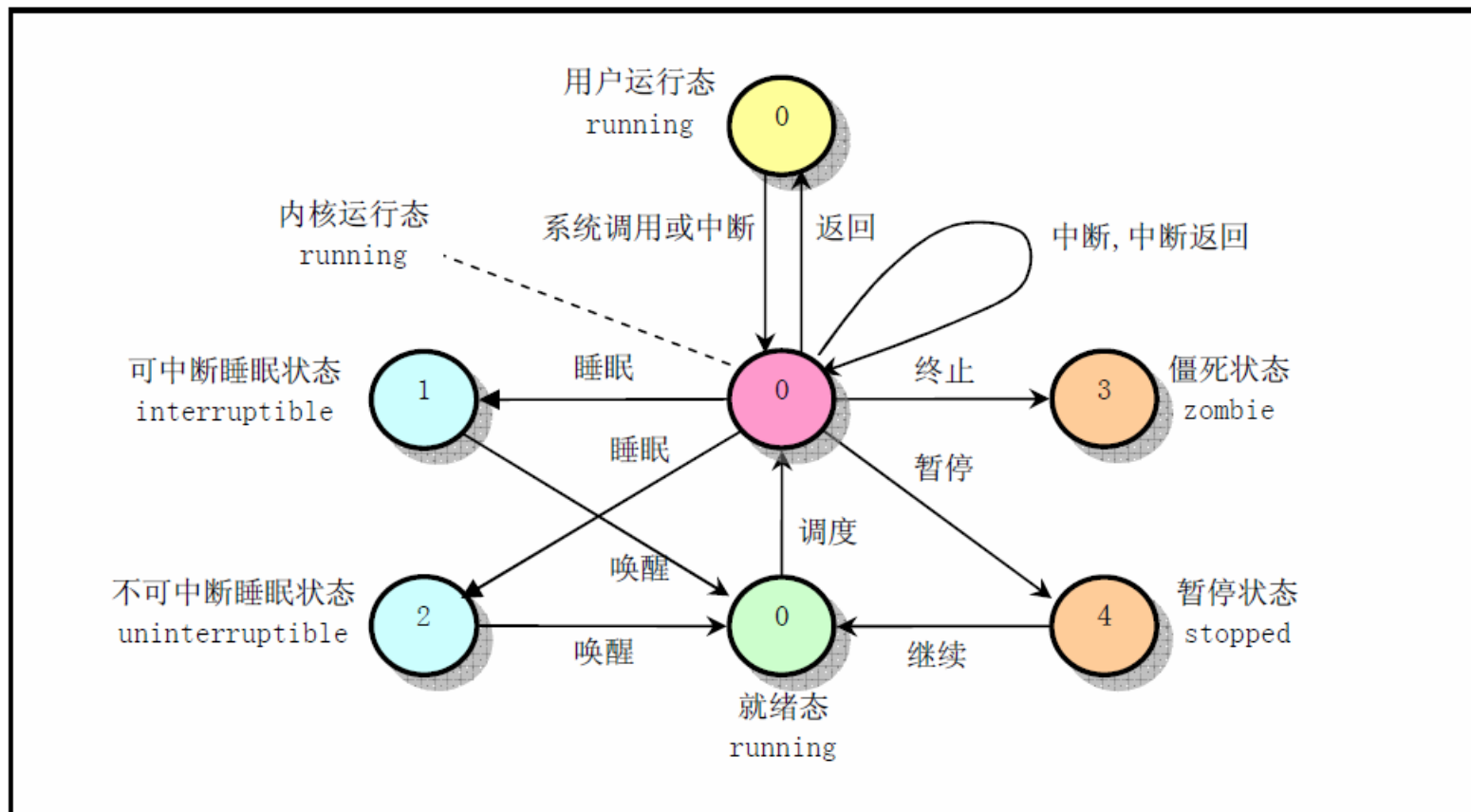
- 程序是静态的，它是一些保存在磁盘上的指令的有序集合，没有任何执行的概念
- 进程是一个动态的概念，它是程序执行的过程，包括了动态创建、调度和消亡的整个过程

## ■ 进程是程序执行和资源管理的最小单位

## 1.2 Linux进程状态转换



## 1.2 Linux进程状态转换



## 1.3 Linux下的进程结构

---

- 主要的进程标识
  - 进程号(Process Identity Number, PID)
  - 父进程号(parent process ID, PPID)
- PID惟一地标识一个进程
- PID和PPID都是非零正整数
- Linux中的进程包含三个段
  - “数据段”存放的是全局变量、常数以及动态数据分配的数据空间（如malloc函数取得的空間）等。
  - “代码段”存放的是程序代码的数据
  - “堆栈段”存放的是子程序的返回地址、子程序的参数以及程序的局部变量

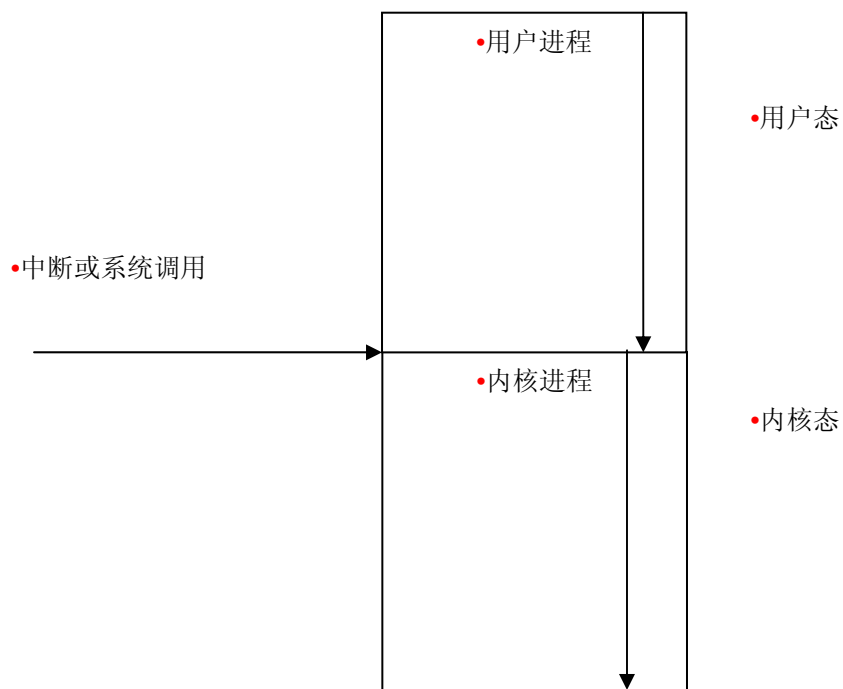
## 1.4 Linux的进程控制块(PCB)

- 每个进程在内核中都有一个进程控制块（PCB）来维护进程相关的信息，Linux内核的进程控制块是task\_struct结构体，包含如下的信息：
  - pid(进程id)
    - 系统中每个进程有唯一的id，在C语言中用pid\_t类型表示，其实就是一个非负整数
  - 进程的状态，有运行、挂起、停止、僵尸等状态。
  - 进程切换时需要保存和恢复的一些 CPU 寄存器。
  - 描述虚拟地址空间的信息。
  - 描述控制终端的信息。
  - 当前工作目录（Current Working Directory）。
  - umask 掩码。
  - 文件描述符表，包含很多指向 file 结构体的指针。
  - 和信号相关的信息。
  - 用户 id 和组id。
  - 控制终端、Session 和进程组。
  - 进程可以使用的资源上限（Resource Limit）。



## 1.5 Linux下的进程的模式和类型

- 进程的执行模式划分为用户模式和内核模式



## 2. 进程环境

## 2.1 main()函数

---

- C程序总是从main()函数开始执行

- main()函数的原型

```
int main(int argc, char **argv);
```

- *argc*是命令行参数的数目,
- *argv*是指向参数的各个指针所构成的数组

- 当内核起动C程序时(使用一个exec()函数), 在调用main()前先调用一个特殊的起动例程。可执行程序文件将此起动例程指定为程序的起始地址--这是由连接编辑程序设置的, 而连接编辑程序则由C编译程序(通常是cc)调用。起动例程从内核取得命令行参数和环境变量值, 然后为调用main()函数作好安排。

## 2.2 进程终止

---

### ■ 有五种方式使进程终止

#### ➤ 正常终止

- 从main()返回
- 调用exit()
- 调用\_exit()

#### ➤ 异常终止

- 调用abort()
- 由一个信号终止

### ■ wait()/waitpid()可以获得进程终止的状态


## 2.3.1 exit()/\_exit()

- 内核使程序执行的唯一方法是调用一个exec()函数
- 进程自愿终止的唯一方法是显式或隐式地调用exit()或调用\_exit()
- exit()和\_exit()函数用于正常终止一个程序
  - \_exit()立即进入内核
  - exit()则先执行一些清除处理，然后进入内核
    - 调用执行各终止处理程序
    - 关闭所有标准I/O流等

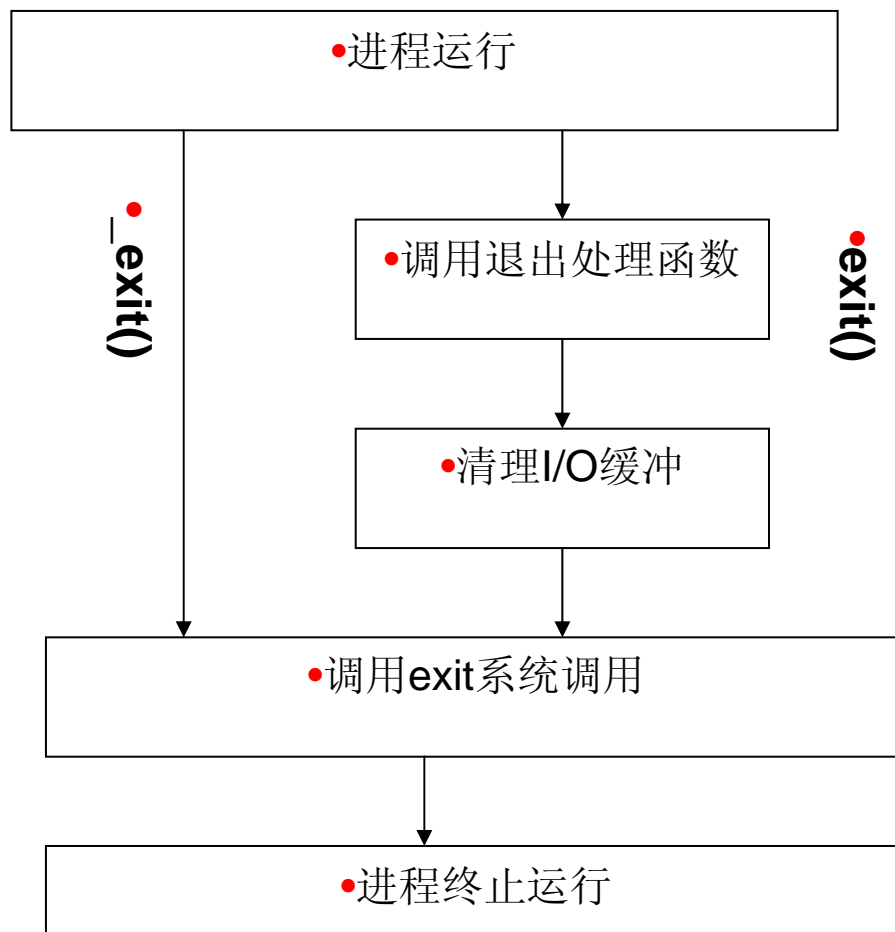
```
#include <stdlib.h>

void exit(int status);
void _Exit(int status);

#include <unistd.h>
void _exit(int status);
```

- 
- exit()和\_exit()都带一个整型参数，称之为终止状态(exit status)。大多数UNIX shell都提供检查一个进程终止状态的方法。如果(a)若调用这些函数时不带终止状态，或(b)main()执行了一个无返回值的return语句，或(c) main()执行隐式返回，则该进程的终止状态是未定义的。

## 2.3.3 exit()/\_exit()



## 2.3.3 exit()/\_exit()

所需头文件	exit: #include <stdlib.h>
	_exit: #include <unistd.h>
函数原型	exit: void exit(int status);
	_exit: void _exit(int status);
函数传入值	<p><b>status</b>是一个整型的参数，可以利用这个参数传递进程结束时的状态。一般来说，0表示没有意外的正常结束；其他的数值表示出现了错误，进程非正常结束。</p> <p>在实际编程时，可以用wait系统调用接收子进程的返回值，从而针对不同的情况进行不同的处理。</p>

## 2.3.4 exit()和\_exit()区别

```
int main(int argc, char **argv)
{
    printf("Using exit...\n");
    printf("This is the content in buffer");
    exit(0);
}
```

```
[root@(none) 1]# ./exit
Using exit...
This is the content in buffer
[root@(none) 1]#
```

```
int main()
{
    printf("Using _exit...\n");
    printf("This is the content in buffer");
    _exit(0);
}
```

```
[root@(none) 1]# ./_exit
Using _exit...
[root@(none) 1]#
```



## 2.4 atexit()

- 按照ANSIC的规定，一个进程可以登记多至32个函数，这些函数将由exit()自动调用。我们称这些函数为终止处理程序(exit handler)，并用atexit()函数来登记这些函数。
- atexit()的参数是一个函数地址，当调用此函数时无需向它传送任何参数，也不期望它返回一个值。exit()以登记这些函数的相反顺序调用它们。同一函数如若登记多次，则也被调用多次。


```
#include <stdlib.h>

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

## 2.5 命令行参数

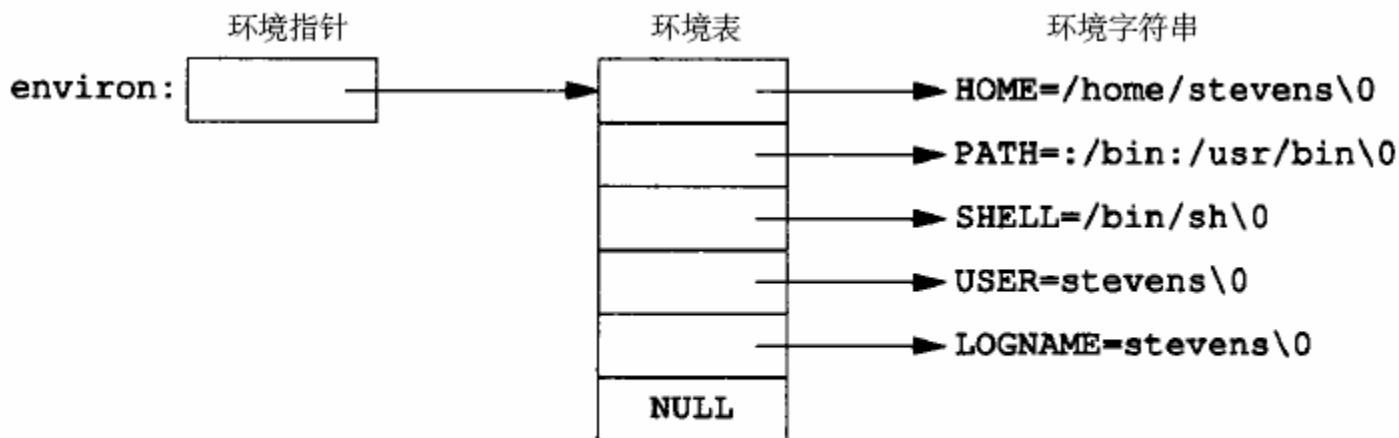
- 当执行一个程序时，调用exec()的进程可将命令行参数传递给该新程序。这是UNIX shell的一部分常规操作。
- ANSI C和POSIX.1都要求argv[argc] 是一个空指针。

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv)
4 {
5     int i;
6
7     for (i = 0; i < argc; i++)
8     {
9         fprintf(stdout, "argv[%d] = %s\n", i, argv[i]);
10    }
11
12    return 0;
13 }
```

- 
- 在UNIX环境下，以“-”指定的参数通常称为短参数(如-h)，以“--”指定的参数通常称为长参数(如--help)
  - 可以使用getopt()和getopt\_long()来解析命令行参数
  - 可以使用GNU getopt()来生成命令行参数解析代码

## 2.6 环境表/环境变量

- 每个程序都接收到一张环境表。与参数表一样，环境表也是一个字符指针数组，其中每个指针包含一个以null结束的字符串的地址。全局变量`environ`则包含了该指针数组的地址。
  - `extern char **environ;`
- 按照惯例，环境由 *name=value* 格式的字符串组成
- `getenv()/putenv()/setenv()` 可以用来取得/设置环境变量



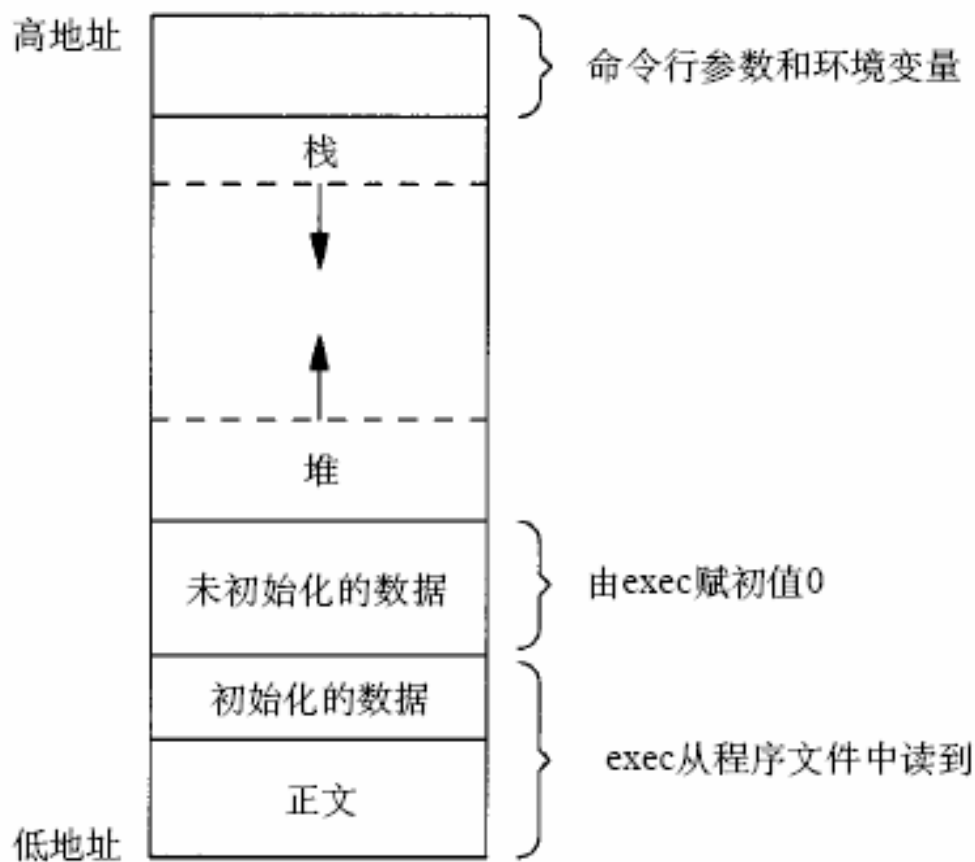
由五个字符串组成的环境

## 2.6.1 一些重要的环境变量

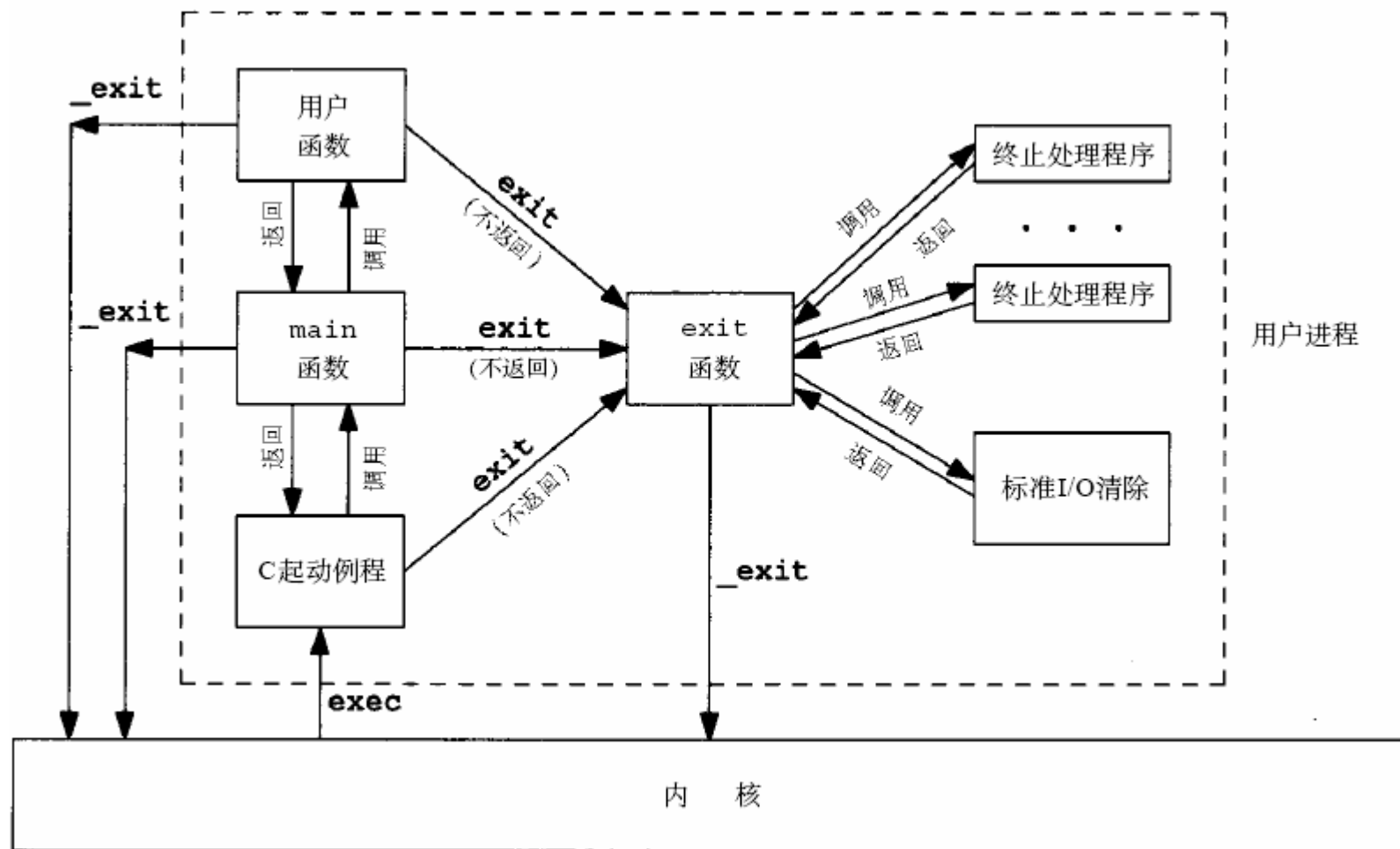
---

- PATH
- SHELL
- TERM
- LANG
- HOME

## 2.7 Linux进程地址空间



## 2.8 C程序的启动和退出



## 2.9 setrlimit()/getrlimit()

- 每个进程都有一组资源限制，可以使用getrlimit()和setrlimit()查询和修改

```
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);

struct rlimit
{
    rlim_t rlim_cur;          /* Soft limit */
    rlim_t rlim_max;         /* Hard limit (ceiling for rlim_cur) */
};
```

# 3. 进程控制



## 3.1 启动进程

---

### ■ 手工启动

- 前台启动: ls
- 后台启动: ls &

### ■ 调度启动

- crontab
  - at
- 
- Windows下有一个VirtualCron
  - Windows的计划任务与at功能类似

## 3.2 进程相关的命令/工具

---

- ps查看系统中的进程
- top动态显示系统中的进程
- nice按用户指定的优先级运行
- renice改变正在运行进程的优先级。
- kill终止进程（包括后台进程）
- crontab用于安装、删除或者列出用于驱动cron后台进程的任务表
- Job control(依赖于kernel和shell)
  - ^z(Ctrl-Z)将当前运行的进程挂起
  - bg将挂起的进程放到后台执行
  - fg将挂起的进程唤到前台

## 3.3 进程标识

---

- 每个进程都有一个非负整型的唯一进程ID，称为进程ID，即pid，可以通过getpid()获得
- 除了进程ID(pid)，每个进程还有一些其他标识符
  - ppid，父进程ID，可以通过getppid()获得
  - uid，所有者ID，可以通过getuid()获得
  - gid，组ID，可以通过getgid()获得(添加组ID)
  - euid，有效用户ID，可以通过geteuid()获得
  - egid，有效组ID，可以通过getegid()获得
- 每个进程还有saved uid和saved gid

## 3.3 进程标识(cont.)

### ■ 获取进程相关属性信息的函数

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

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

```
pid_t getpid(void);
```

返回：调用进程的进程 ID

```
pid_t getppid(void);
```

返回：调用进程的父进程 ID

```
uid_t getuid(void);
```

返回：调用进程的实际用户 ID

```
uid_t geteuid(void);
```

返回：调用进程的有效用户 ID

```
gid_t getgid(void);
```

返回：调用进程的实际组 ID

```
gid_t getegid(void);
```

返回：调用进程的有效组 ID



## 3.4.1 进程创建: fork()

- 通常情况下, 一个现存进程调用fork()函数是UNIX内核创建一个新进程的唯一方法
  - init进程是例外
- 由fork()创建的新进程被称为子进程(child process)
- fork()函数被调用一次, 但返回两次, 两次返回的区别
  - 子进程的返回值是0
    - fork()使子进程得到返回值0的理由是: 一个进程只会有一个父进程, 所以子进程总是可以调用getppid()以获得其父进程的进程ID
  - 父进程的返回值则是新子进程的进程ID
    - 将子进程ID返回给父进程的理由是: 因为一个进程的子进程可以多于一个, 所以没有一个函数使一个进程可以获得其所有子进程的进程ID
- 子进程和父进程继续执行fork()之后的指令
- 子进程是父进程的复制品, 子进程获得父进程数据空间、堆和栈的复制品(这是子进程所拥有的拷贝, 父、子进程并不共享这些存储空间部分), 如果代码段是只读的, 则父、子进程共享代码段
- 一般来说, 在fork()之后是父进程先执行还是子进程先执行是不确定的, 这取决于内核所使用的调度算法
- 如果要求父、子进程之间相互同步, 则要求某种形式的进程间通信
- Under Linux, fork() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

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

pid_t fork(void);
```

## 3.4.2 进程创建：fork()

所需头文件	<pre>#include &lt;sys/types.h&gt; // 提供类型pid_t的定义 #include &lt;unistd.h&gt;</pre>
函数原型	<pre>pid_t fork(void);</pre>
函数返回值	0：子进程
	子进程ID（大于0的整数）：父进程
	-1：出错

## 3.4.3 fork()实例

```
int main(int argc, char **argv)
{
    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\nIn 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\nIn father process!!\nMy PID is %d\n",result,getpid());
    }
}
```

## 3.4.4 fork()实例

### Parent

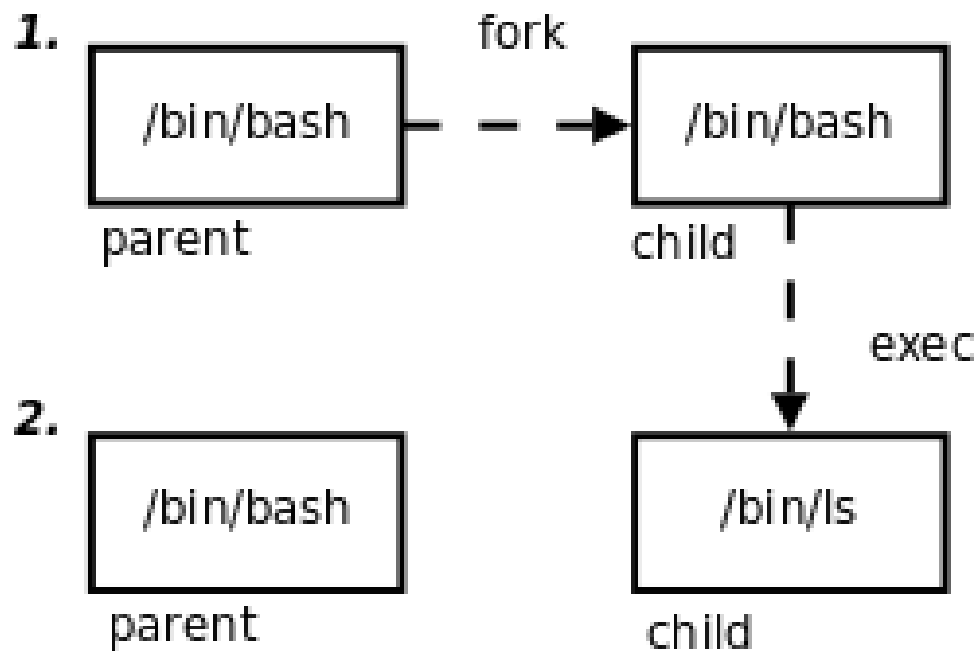
```
int main()
{
    pid_t pid;
    char *message;
    int n;
    ↓ pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    if (pid == 0) {
        message = "This is the child\n";
        n = 6;
    } else {
        message = "This is the parent\n";
        n = 3;
    }
    for(; n > 0; n--) {
        printf(message);
        sleep(1);
    }
    return 0;
}
```

### Child

```
int main()
{
    pid_t pid;
    char *message;
    int n;
    pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    if (pid == 0) {
        message = "This is the child\n";
        n = 6;
    } else {
        message = "This is the parent\n";
        n = 3;
    }
    for(; n > 0; n--) {
        printf(message);
        sleep(1);
    }
    return 0;
}
```



## 3.4.5 fork/exec



## 3.5 vfork()

- 传统上，实现vfork()用于降低fork()的开销
  - 由于fork()完整地拷贝了父进程的整个地址空间，因此执行速度是比较慢的，为了加快fork()的执行速度，有些UNIX系统设计者创建了vfork()
  - vfork()也创建新进程，但它不产生父进程的副本，它通过允许父子进程可访问相同物理内存从而伪装了对进程地址空间的真实拷贝，当子进程需要改变内存中数据时才拷贝父进程。这就是所谓的“写操作时拷贝”(copy-on-write)技术
  - 现在，很多嵌入式Linux系统的fork()函数调用都采用vfork()函数的实现方式
    - 例如uClinux所有的多进程管理都通过vfork()来实现
- 在Linux环境下
  - vfork() is like fork() with CLONE\_VM & CLONE\_VFORK flags set.
  - With vfork() child & parent share address space; parent is blocked until child exits or executes a new program.
- 建议程序员不要刻意的去区分fork()和vfork()，在有些实现下，他们可能是相同的定义
- vfork()通常用于创建新的子进程之后立即调用exec函数族的情况

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

pid_t vfork(void);
```

## 3.6.1 子进程从父进程继承的信息

---

- 打开的文件
- 实际用户ID、实际组ID、有效用户ID、有效组ID
- 添加组ID
- 进程组ID
- 对话期ID
- 控制终端
- 设置-用户-ID标志和设置-组-ID标志
- 当前工作目录
- 根目录(chroot)
- 文件方式创建屏蔽字(umask)
- 信号屏蔽和排列
- 对任一打开文件描述符的在执行时关闭标志
- 环境
- 连接的共享存储段
- 资源限制

## 3.6.2 子进程与父进程的差异

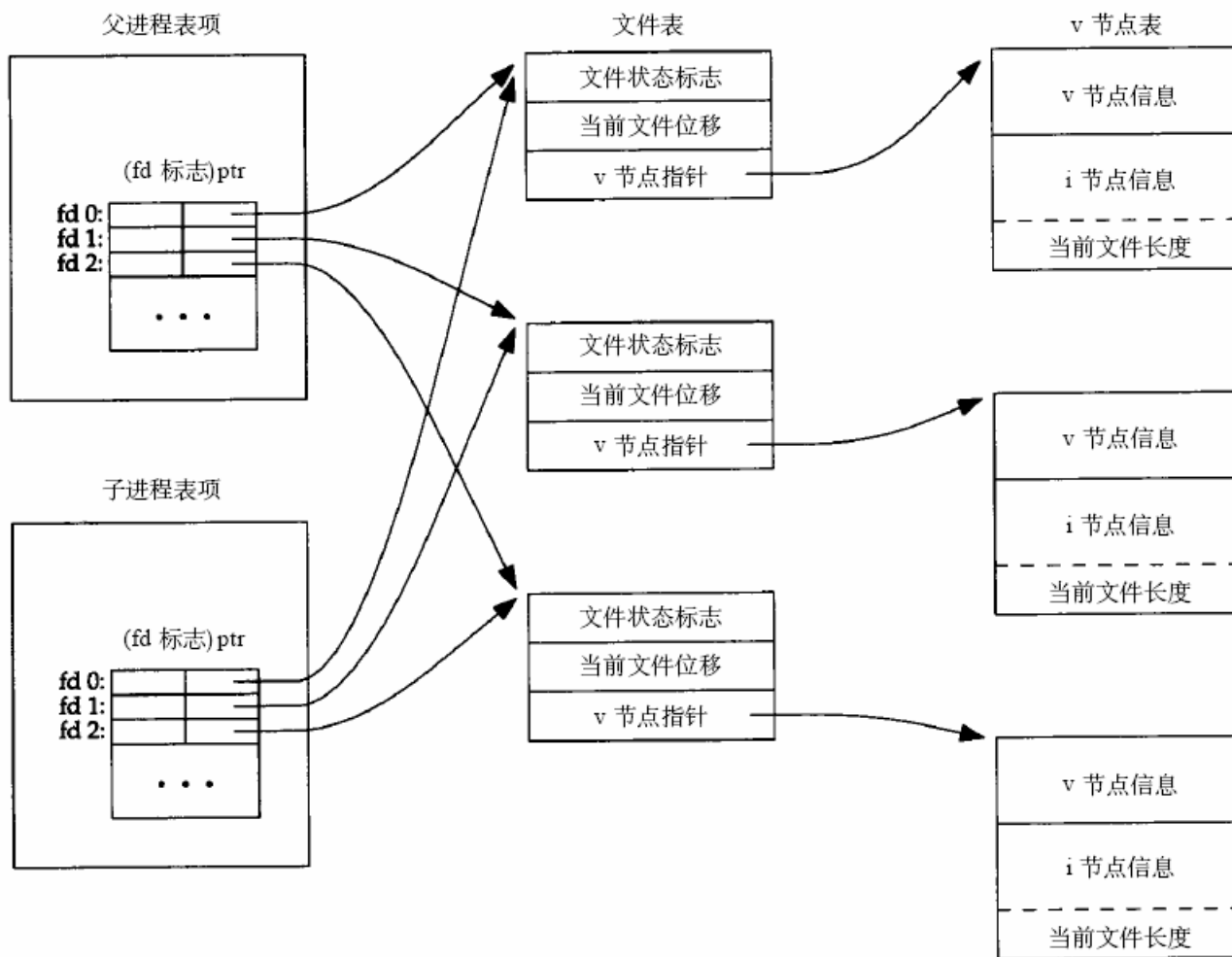
---

- fork的返回值
- 进程ID
- 不同的父进程ID
- 子进程的tms\_utime,tms\_stime,tms\_cutime以及tms\_ustime设置为0
- 父进程设置的锁，子进程不继承
- 子进程的未决告警被清除
- 子进程的未决信号集设置为空集

### 3.7.1 fork()之后父子进程对打开文件的共享

- 如果父、子进程写到同一描述符文件，但又没有任何形式的同步(例如使父进程等待子进程)，那么它们的输出就会相互混合(假定所用的描述符是在fork()之前打开的，如标准输出和标准错误输出)
- 在fork()之后处理文件描述符有两种常见的情况：
  - 父进程等待子进程完成。在这种情况下，父进程无需对其描述符做任何处理。当子进程终止后，它曾进行过读、写操作的任一共享描述符的文件位移量已做了相应更新。
  - 父、子进程各自执行不同的程序段。在这种情况下，在fork()之后，父、子进程各自关闭它们不需使用的文件描述符，并且不干扰对方使用的文件描述符。这种方法是网络服务进程中、管道通讯中经常使用的。

## 3.7.2 fork()之后父子进程对打开文件的共享



## 3.8 fork() 失败

---

### ■ 有两种情况可能会造成fork()失败

- 系统中已经有了太多的进程(通常意味着某个方面出了问题, 或者设计问题)
- 该实际用户ID的进程总数超过了系统限制。进程的限制CHILD\_MAX参数规定了每个实际用户ID在任一时刻可具有的最大进程数。
  - 可以使用sysconf()取得

## 3.9 fork()的用途

### ■ fork()有两种用法

- 一个父进程希望复制自己，使父、子进程同时执行不同的代码段，这在网络服务进程中是常见的--父进程等待委托者的服务请求，当这种请求到达时，父进程调用fork()，使子进程处理此请求，父进程则继续等待下一个服务请求，在这种情况下，父进程充当的是调度者的角色
- 一个进程要执行一个不同的程序，这对shell是常见的情况。在这种情况下，子进程在从fork()返回后立即调用exec，从而实现运行其他程序的目的。
  - 这种情况下使用vfork()更为合适
  - 某些操作系统将fork()-exec()组合成一个，并称其为spawn()



## 3.10.1 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是该进程接受到了一个指定的信号(SIGCHLD)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，还能支持作业控制。
- 实际上，wait()函数只是waitpid()函数的一个特例，Linux内部在实现wait()函数时直接调用的就是waitpid()函数

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

## 3.10.2 wait()和waitpid()

所需头文件	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/wait.h&gt;</pre>
函数原型	<pre>pid_t wait(int *status)</pre>
函数传入值	<p>这里的status是一个整型指针，是该子进程退出时的状态。</p> <ul style="list-style-type: none"><li>• status若为空，则代表任意状态结束的子进程</li><li>• status若不为空，则代表指定状态结束的子进程</li></ul> <p>另外，子进程的结束状态可由Linux中一些特定的宏来测定。</p>
函数返回值	<p>成功：子进程的进程号 失败：-1</p>

## 3.10.3 wait和waitpid

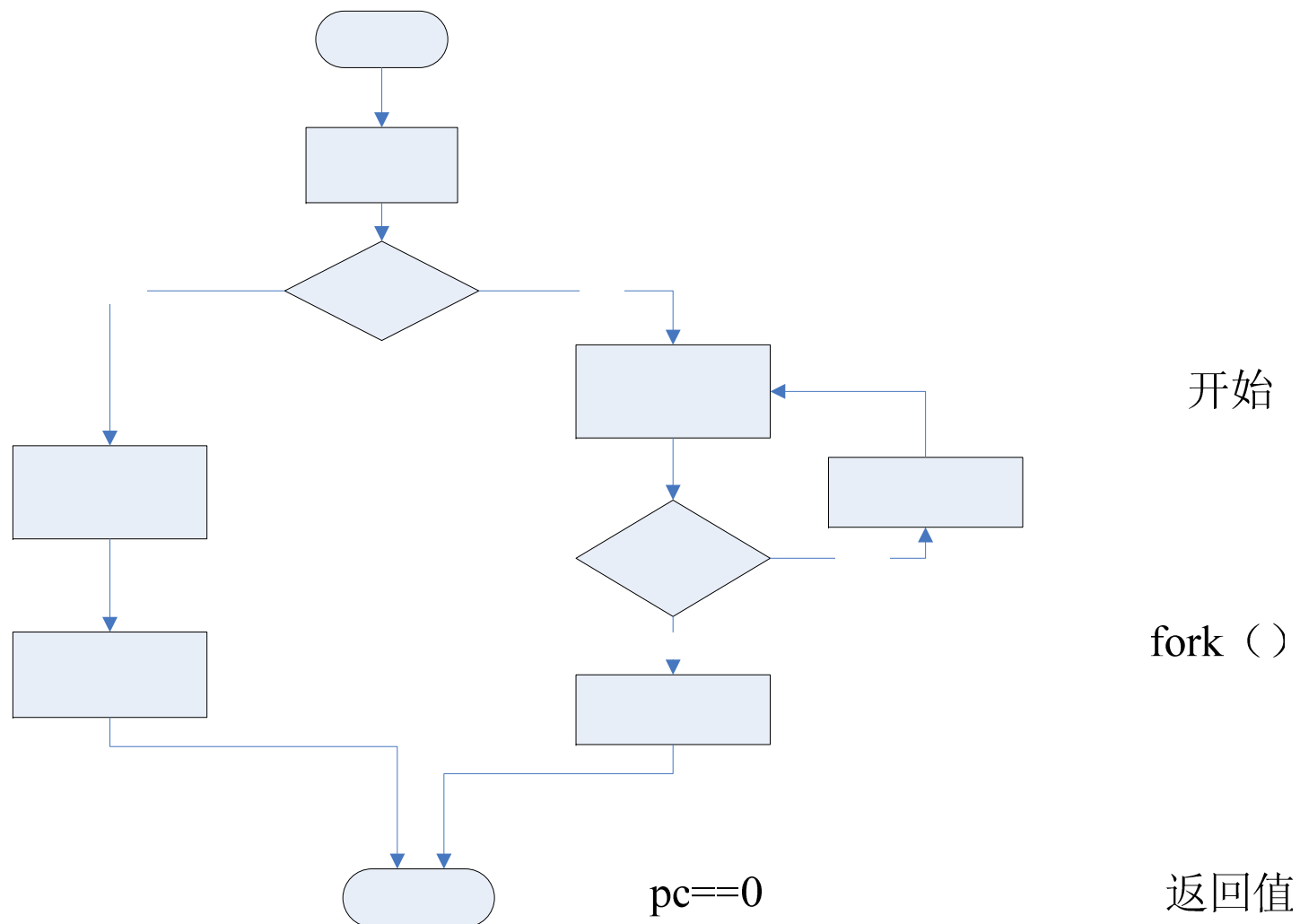
所需头文件	#include <sys/types.h> #include <sys/wait.h>	
函数原型	pid_t waitpid(pid_t pid, int *status, int options)	
函数传入值	pid	pid>0: 只等待进程ID等于pid的子进程, 不管已经有其他子进程运行结束退出了, 只要指定的子进程还没有结束, waitpid就会一直等下去。
		pid=-1: 等待任何一个子进程退出, 此时和wait作用一样。
		pid=0: 等待其组ID等于调用进程的组ID的任一子进程。
		pid<-1: 等待其组ID等于pid的绝对值的任一子进程。
	status	同wait
	options	WNOHANG: 若由pid指定的子进程并不立即可用, 则waitpid不阻塞, 此时返回值为0
		WUNTRACED: 若某实现支持作业控制, 则由pid指定的任一子进程状态已暂停, 且其状态自暂停以来还未报告过, 则返回其状态。
		0: 同wait, 阻塞父进程, 等待子进程退出。
函数返回值	正常: 子进程的进程号	
	使用选项WNOHANG且没有子进程退出: 0	
	调用出错: -1	

## 3.10.4 检查status

宏	说 明
<code>WIFEXITED(status)</code>	<p>若为正常终止子进程返回的状态，则为真。对于这种情况可执行  <code>WEXITSTATUS(status)</code>            取子进程传送给 <code>exit</code> 或 <code>_exit</code> 参数的低8位</p>
<code>WIFSIGNALED(status)</code>	<p>若为异常终止子进程返回的状态，则为真（接到一个不捕捉的信号）。对于这种情况，可执行  <code>WTERMSIG(status)</code>            取使子进程终止的信号编号。            另外，SVR4和4.3+BSD（但是，非POSIX.1）定义宏：  <code>WCOREDUMP(status)</code>            若已产生终止进程的 <code>core</code> 文件，则它返回真</p>
<code>WIFSTOPPED(status)</code>	<p>若为当前暂停子进程的返回的状态，则为真。对于这种情况，可执行  <code>WSTOPSIG(status)</code>            取使子进程暂停的信号编号</p>

检查 `wait()` 和 `waitpid()` 所返回的终止状态的宏

## 3.10.4 wait和waitpid使用实例



## 3.11.1 exec函数族

- `fork()`函数是用于创建一个子进程，该子进程几乎拷贝了父进程的全部内容，那么，这个新创建的进程如何执行呢？
- `exec`函数族提供了一个在进程中启动另一个程序的方法
  - `exec`函数族调用时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。
  - 调用`exec`执行的可执行文件既可以是二进制文件，也可以是任何Linux下可执行的脚本文件
- 其他方式
  - `system()`
  - `popen()`

## 3.11.2 exec函数族

### ■ 何时使用

- 当有进程认为自己不能再为系统和用户做出任何贡献了，就可以调用任何exec函数族，让自己以新的面貌重生
  - 用户登录的过程：getty-login-bash
- 如果一个进程想执行另一个程序，如shell，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一样。（这种情况非常普遍）
- popen()

```
#include <unistd.h>

extern char **environ;

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[]);
```

## 3.11.3 exec函数族语法

所需头文件	#include <unistd.h>
函数原型	int execl(const char *path, const char *arg, ...);
	int execv(const char *path, char *const argv[]);
	int execlp(const char *path, const char *arg, ..., char *const envp[]);
	int execve(const char *path, char *const argv[], char *const envp[]);
	int execlp(const char *file, const char *arg, ...);
	int execvp(const char *file, char *const argv[]);
函数返回值	-1: 出错



## 3.11.4 exec函数族使用区别

### ■ 可执行文件查找方式

- 表中的前四个函数的查找方式都是完整的文件目录路径，而最后两个函数（也就是以p结尾的两个函数）可以只给出文件名，系统就会自动从环境变量“\$PATH”所指出的路径中进行查找。

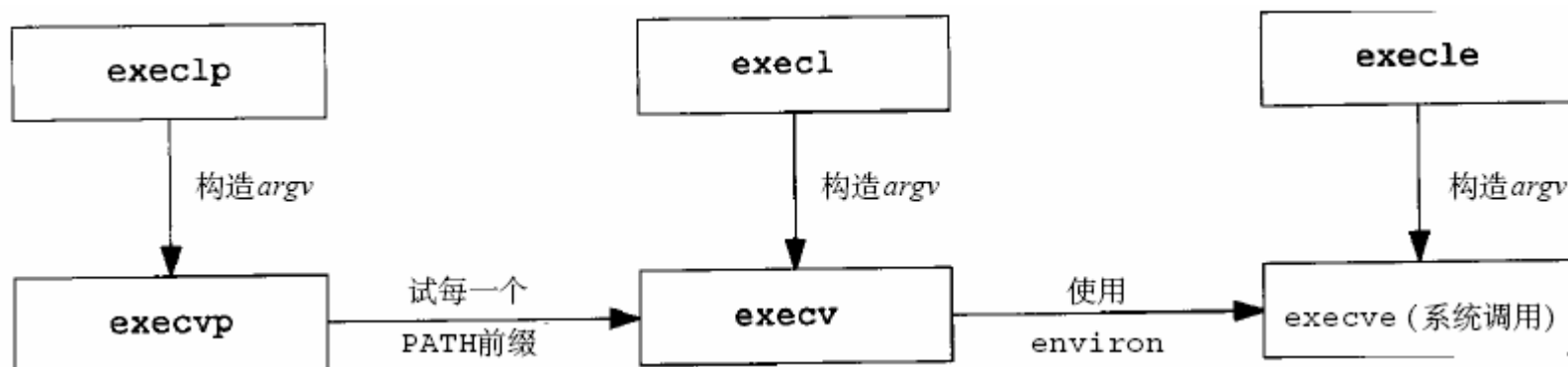
### ■ 参数表传递方式

- 两种方式：逐个列举、将所有参数整体构造指针数组传递
- 以函数名的第五位字母来区分的，字母为“l”（list）的表示逐个列举的方式，其语法为char \*arg；字母为“v”（vector）的表示将所有参数整体构造指针数组传递，其语法为\*const argv[]

### ■ 环境变量的使用

- exec函数族可以默认系统的环境变量，也可以传入指定的环境变量。这里，以“e”（Enviromen）结尾的两个函数execle、execve就可以在envp[]中指定当前进程所使用的环境变量

### 3.11.4 exec函数族使用区别(cont.)



六个exec()函数之间的关系

## 3.11.5 exec函数族语法

前四位	统一为：exec	
第五位	l: 参数传递为逐个列举方式	execl、execle、execlp
	v: 参数传递为构造指针数组方式	execv、execve、execvp
第六位	e: 可传递新进程环境变量	execle、execve
	p: 可执行文件查找方式为文件名	execlp、execvp

## 3.11.6 exec使用实例

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <string.h>
4 #include <errno.h>
5
6 int main(int argc, char **argv)
7 {
8     int return_code;
9
10    //return_code = execl("/bin/ls", "/bin/ls", "-l", (char *) 0);
11    //char *v[] = { "ls", NULL };
12    //char *v[] = { "./test13", NULL };
13    char *v[] = { "test13", NULL };
14    //return_code = execlp("ls1", "ls1", (char *) 0);
15    return_code = execv("./test13", v);
16
17    if (return_code < 0)
18    {
19        fprintf(stderr, "execv() failed: %s\n", strerror(errno));
20    }
21    else
22    {
23        fprintf(stderr, "execv() succeeded.\n");
24    }
25
26    return 0;
27 }
```

## 3.11.7 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
- 最常见的错误原因
  - 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

## 3.11.8 exec后的继承关系

- 新程序的进程还保持了原进程的下列特征
  - 进程ID和父进程ID。
  - 实际用户ID和实际组ID。
  - 添加组ID。
  - 进程组ID。
  - 对话期ID。
  - 控制终端。
  - 闹钟尚余留的时间。
  - 当前工作目录。
  - 根目录。
  - 文件方式创建屏蔽字。
  - 文件锁。
  - 进程信号屏蔽。
  - 未决信号。
  - 资源限制。
  - tms\_utime,tms\_stime,tms\_cutime以及tms\_ustime值。

## 3.12.1 更改用户ID和组ID

- 可以用setuid()函数设置实际用户ID(uid)和有效用户ID(euid)
- 可以用setgid()函数设置实际组ID(gid)和有效组ID(egid)

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

int setuid(uid_t uid);
```

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

int setgid(gid_t gid);
```



## 3.12.2 几个ID的解释

- 内核会给每个进程关联两个和进程ID无关的用户ID，一个是真实用户ID(real uid, uid)，还有一个是有效用户ID(effective uid, euid)。
  - 真实用户ID(uid)用于标识由谁为正在运行的进程负责
  - 有效用户ID(euid)用于为新创建的文件分配所有权、检查文件访问许可，还用于通过kill系统调用向其它进程发送信号时的许可检查
  - 内核允许一个进程以调用exec一个setuid程序或者显式执行setuid系统调用的方式改变它的有效用户ID。
- 所谓setuid程序是指一个设置了许可模式字段中的setuid bit的可执行文件(-r-s--x--x)。当一个进程exec一个setuid程序的时候，内核会把进程表以及u区中的有效用户ID(euid)设置成该文件所有者的ID。为了区分这两个字段，我们把进程表中的那个字段称作保存用户ID(Saved UID)。
- Saved UID和Saved GID，分别是在调用exec之前的有效UID(euid)和GID(egid)的数值。setuid()和setgid()系统调用还可以把有效ID恢复为保存的数值。



## 3.12.3 更改用户ID和组ID-规则

---

### ■ 更改用户ID规则

- 若进程具有超级用户特权，则setuid函数将实际用户ID、有效用户ID，以及保存的设置-用户-ID设置为uid(saved uid)。
- 若进程没有超级用户特权，但是uid等于实际用户ID或保存的设置-用户-ID(saved uid)，则setuid只将有效用户ID设置为uid。不改变实际用户ID和保存的设置-用户-ID(saved uid)。
- 如果上面两个条件都不满足，则errno设置为EPERM，并返回出错。

### ■ 更改组ID的规则与更改用户ID的规则一致

## 3.12.4 更改用户ID和组ID-进一步的解释

- 关于内核所维护的三个用户ID，还要注意下列几点：
  - 只有超级用户进程可以更改实际用户ID。通常，实际用户ID是在用户登录时，由login(1)程序设置的，而且决不会改变它。因为login是一个超级用户进程，当它调用setuid时，设置所有三个用户ID。
  - 仅当对程序文件设置了设置-用户-ID位时，exec函数设置有效用户ID。如果设置-用户-ID位没有设置，则exec函数不会改变有效用户ID，而将其维持为原先值。任何时候都可以调用setuid，将有效用户ID设置为实际用户ID或保存的设置-用户-ID。自然，不能将有效用户ID设置为任一随机值。
  - 保存的设置-用户-ID是由exec从有效用户ID复制的。在exec按文件用户ID设置了有效用户ID后，即进行这种复制，并将此副本保存起来。

### 3.12.5 改变三个用户ID的不同方法

ID	exec		setuid(uid)	
	设置-用户-ID位关闭	设置-用户-ID位打开	超级用户	非特权用户
实际用户 ID	不变	不变	设为uid	不变
有效用户 ID	不变	设置为程序文件的用户 ID	设为uid	设为uid
保存的设置-用户-ID	从有效用户 ID复制	从有效用户 ID复制	设为uid	不变

## 3.12.6 setreuid()/setregid()

- setreuid()用于修改当前调用进程的实际用户ID(uid)和有效用户ID(euid)，其参数通过ruid和euid指定
  - 如果ruid或者euid设为-1，则当前调用进程相对应的实际用户ID(uid)和有效用户ID(euid)保持不变
  - 特权进程(拥有root权限的进程)可以设置ruid和euid为任意值
  - 非特权进程只能设置有效用户ID(euid)，其取值为实际用户ID(uid)、有效用户ID(euid)或者保存的UID(Saved UID)
  - 对于非特权进程来说，是否允许将实际用户ID设置为当前进程的实际用户ID(uid)、有效用户ID(euid)或者保存的设置用户ID(Saved UID)的情况未定义，这依赖于操作系统的实现，经测试，在Linux下是允许的。

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

int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

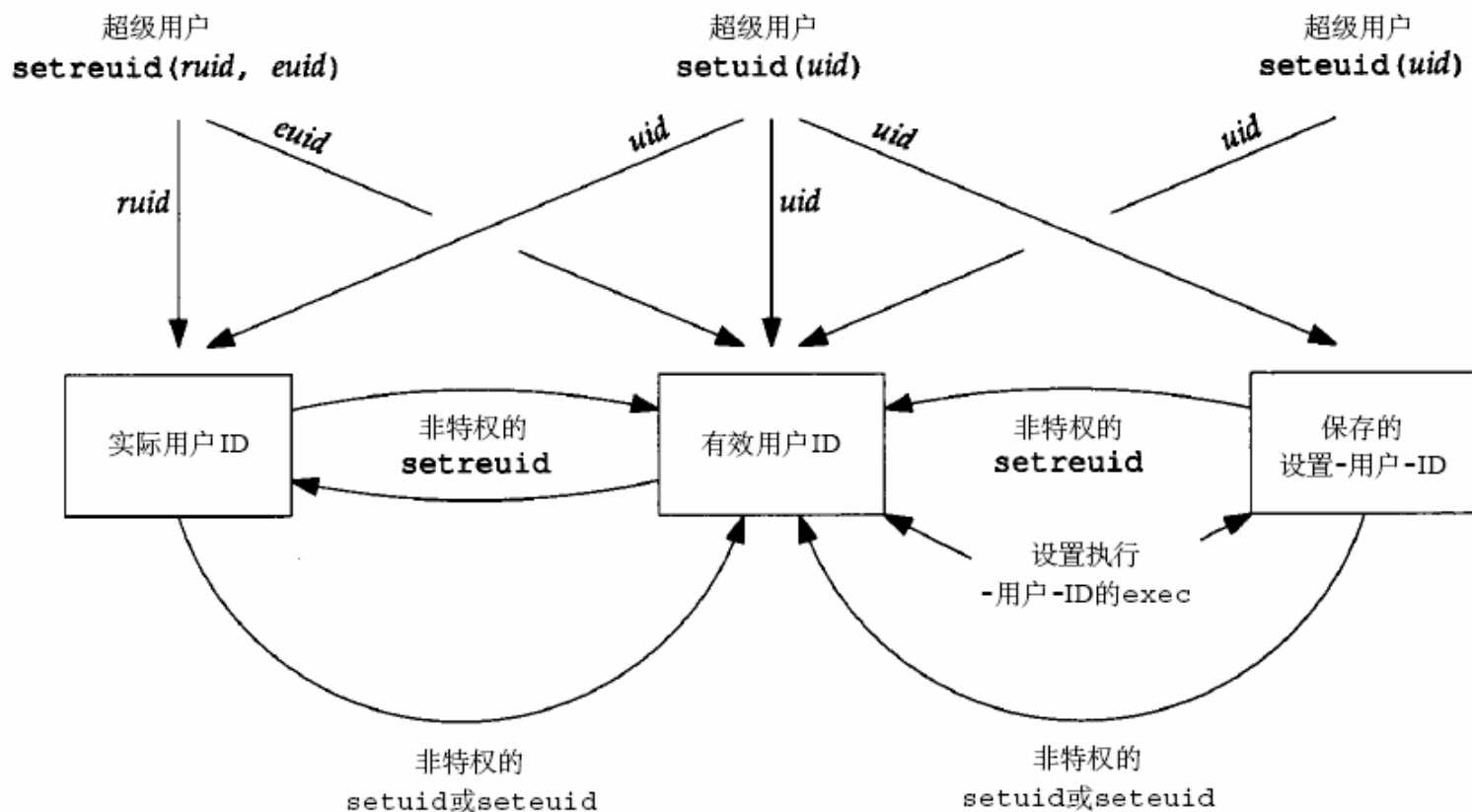
## 3.12.7 seteuid()/setegid()

- seteuid()和setegid()。它们只更改有效用户ID和有效组ID
- 非特权用户可将其有效用户ID设置为其实际用户ID或其保存的设置-用户-ID。
- 特权用户则可将有效用户ID设置为uid。(这区别于setuid()函数，它更改三个用户ID。)这一建议更改也要求支持保存的设置-用户-ID。
- Posix.1建议实现

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

int seteuid(uid_t euid);
int setegid(gid_t egid);
```

## 3.12.8 改变uid/euid的几种方式对比



## 3.13 system()

- ANSI C定义了system()函数，但是其操作对系统的依赖性很强。
- system()在其实现中调用了fork()、exec()和waitpid()，可能的返回值：
  - 如果fork()失败或者waitpid()返回除EINTR之外的出错，则system()返回-1，而且errno中设置了错误类型。
  - 如果exec失败(表示不能执行shell)，则其返回值如同shell执行了exit(127)一样。
  - 否则所有三个函数(fork() ,exec()和waitpid())都成功，并且system()的返回值是shell的终止状态
- 如果在一个设置-用户-ID程序中调用system()，那么发生什么呢？
  - 这是一个安全性方面的漏洞，决不当这样做。

```
#include <stdlib.h>

int system(const char *string);
```

## 4. 进程关系



## 4.1 进程组

- 每个进程除了有一进程ID之外，还属于一个进程组
- 进程组是一个或多个进程的集合
- 每个进程组有一个唯一的进程组ID，进程组ID类似于进程ID -- 它是一个正整数，并可存放在pid\_t数据类型中
- 函数getpgrp()返回调用进程的进程组ID
- 每个进程组有一个组长进程，进程组ID等于组长进程ID
- 进程组组长可以创建一个进程组，创建该组中的进程，然后终止。
- 只要在某个进程组中有一个进程存在，则该进程组就存在，这与其组长进程是否终止无关。
- 从进程组创建开始到其中最后一个进程离开为止的时间区间称为进程组的生命期。
- 某个进程组中的最后一个进程可以终止，也可以参加另一个进程组。

## 4.2 getpgid()/setpgid()

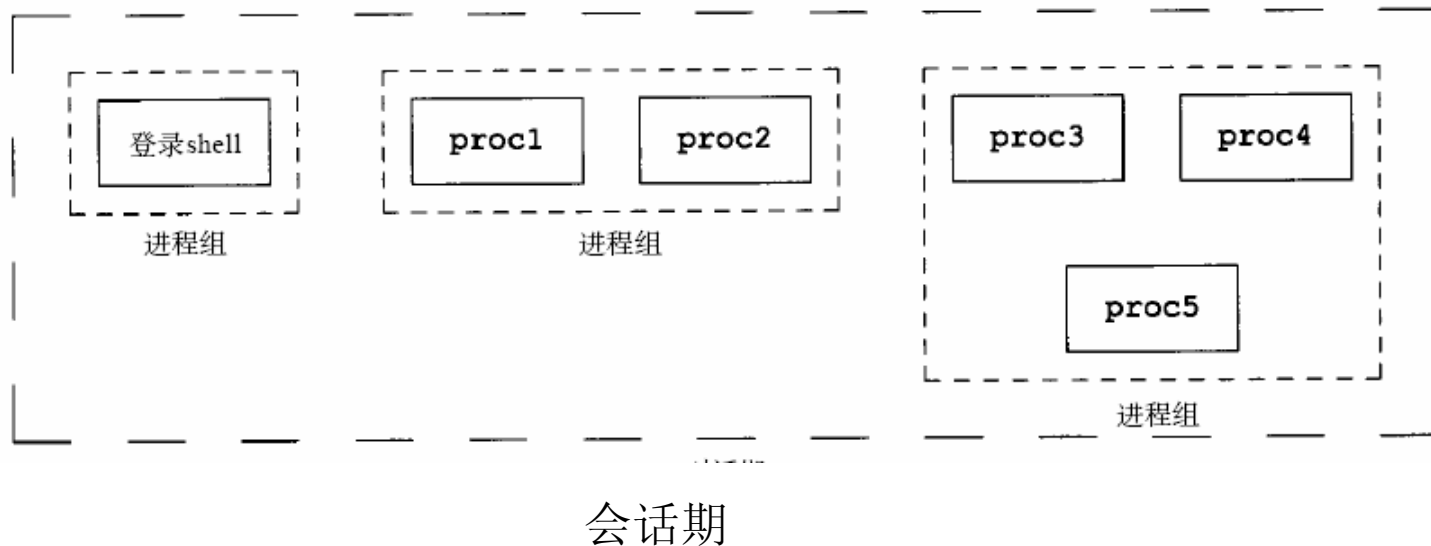
- 进程调用setpgid()可以参加一个现存的组或者创建一个新进程组(setsid()也可以创建一个新的进程组)
- setpgid()将pid进程的进程组ID设置为pgid
  - 如果这两个参数相等,则由pid指定的进程变成进程组组长。
  - 一个进程只能为它自己或它的子进程设置进程组ID。
    - 在它的子进程调用了exec后,它就不再能改变该子进程的进程组ID
  - 如果pid是0,则使用调用者的进程ID。
  - 如果pgid是0,则由pid指定的进程ID被用作为进程组ID。

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
pid_t getpgid(pid_t pid);
int setpgrp(void);
pid_t getpgrp(void);
```

## 4.3 会话期(session)

- 会话期(session)是一个或多个进程组的集合
- 在下图中，在一个对话期中有三个进程组，通常是由shell的管道线将几个进程编成一组的
  - `proc1 | proc2 &`
  - `proc3 | proc4 | proc5`



## 4.4 setsid()

### ■ 进程调用setsid()函数就可建立一个新对话期

```
#include <unistd.h>

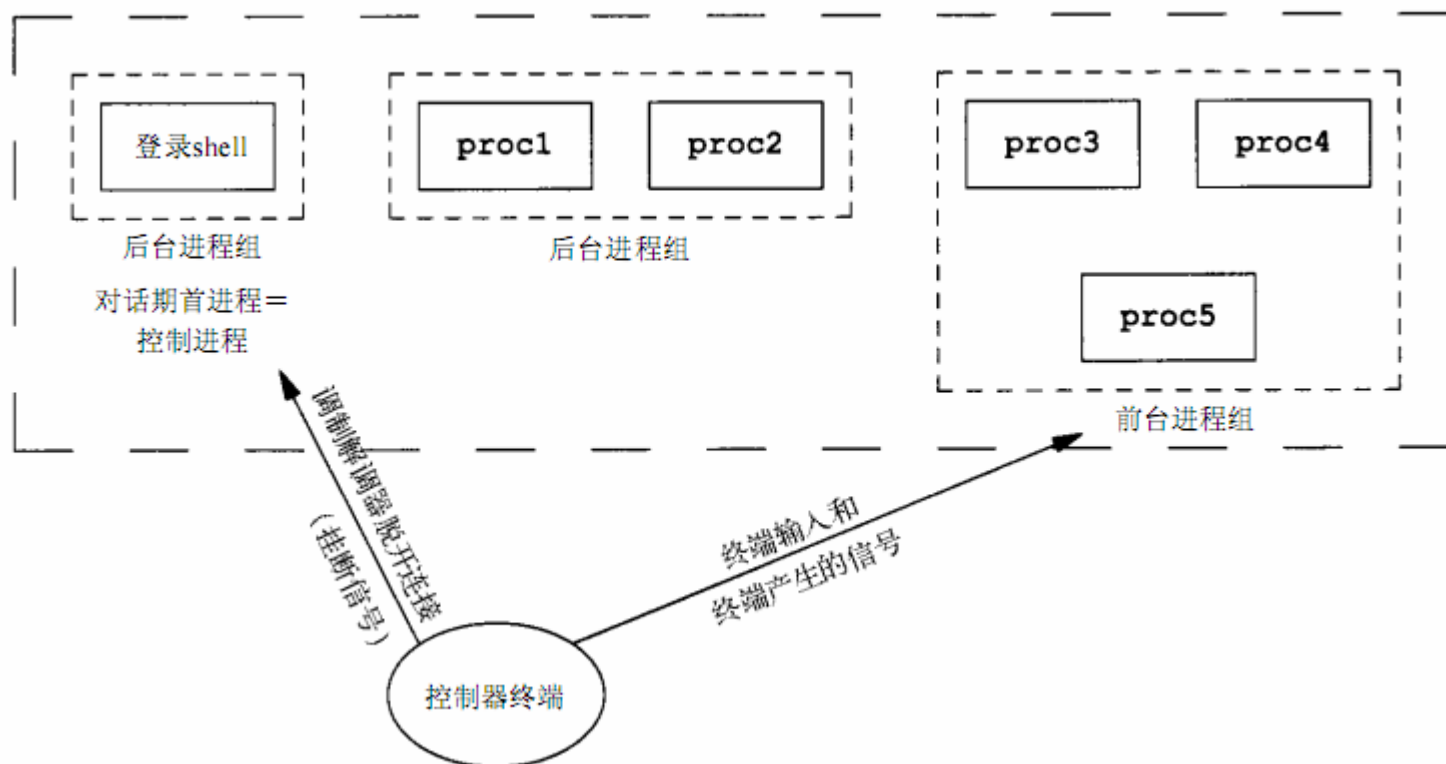
pid_t setsid(void);
```

- 如果调用此函数的进程不是一个进程组的组长，则此函数创建一个新会话期，结果为：
  - 此进程变成该新会话期的会话期首进程(session leader，会话期首进程是创建该会话期的进程)。此进程是该新会话期中的唯一进程。
  - 此进程成为一个新进程组的组长进程。新进程组ID是此调用进程的进程ID。
  - 如果在调用setsid()之前此进程有一个控制终端，创建新的session后，那么这种联系也被解除，也就意味着调用进程脱离了控制终端。
- 如果此调用进程已经是一个进程组的组长，则此函数返回出错。
  - 为了保证不处于这种情况，通常先调用fork()，然后使其父进程终止，而子进程则继续，子进程继承了父进程的进程组ID，而其进程ID则是新分配的，两者不可能相等，所以这就保证了子进程不是一个进程组的组长

## 4.5 控制终端

- 一个会话期可以有一个单独的控制终端(controlling terminal)。这通常是我们在其上登端设备(终端登录情况)或伪终端设备(网络登录情况)。
- 建立与控制终端连接的对话期首进程，被称之为控制进程(controlling process)。
- 一个会话期中的几个进程组可被分成一个前台进程组(foreground process group)以及一个或多个后台进程组(background process group)。
- 如果一个会话期有一个控制终端，则它有一个前台进程组，其他进程组则为后台进程组。
- 无论何时键入中断键(常常是Delete或Ctrl-C)或退出键(常常是Ctrl-\)，就会造成将号或退出信号送至前台进程组的所有进程。
- 如果终端界面检测到调制解调器已经脱开连接，则将挂断信号送至控制进程(会话期首)

## 4.5 控制终端



## 4.5.1 终端的基本概念

- 一般意义上的终端是指人机交互的设备，也就是可以接受用户输入并输出信息给用户的设备
  - 在计算机刚诞生的年代，终端是电传打字机（Teletype）和打印机
  - 现在的终端通常是键盘和显示器
- 在UNIX系统中，用户通过终端登录系统后得到一个Shell 进程，这个终端成为Shell 进程的控制终端（Controlling Terminal）
  - 控制终端是保存在PCB中的信息，而我们知道`fork(2)`会复制PCB中的信息，因此由Shell 进程启动的其它进程的控制终端也是这个终端。
  - 默认情况下（没有重定向），每个进程的标准输入、标准输出和标准错误输出都指向控制终端，进程从标准输入读也就是读用户的键盘输入，进程往标准输出或标准错误输出写也就是输出到显示器上
  - 在控制终端输入一些特殊的控制键可以给前台进程发信号，例如Ctrl-C表示SIGINT，Ctrl-\表示SIGQUIT
- 每个进程都可以通过一个特殊的设备文件`/dev/tty`访问它的控制终端
  - 事实上每个终端设备都对应一个不同的设备文件，`/dev/tty`提供了一个通用的接口，一个进程要访问它的控制终端既可以通过`/dev/tty`也可以通过该终端设备所对应的设备文件来访问。
  - `ttyname()`函数可以由文件描述符查出对应的文件名，该文件描述符必须指向一个终端设备而不能是任意文件

## 4.5.1.1 *ttyname()*

- *ttyname()*可以取得文件描述符所对应的终端名

### NAME

`ttyname`, `ttyname_r` - return name of a terminal

### SYNOPSIS

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

```
char *ttyname(int fd);  
int ttyname_r(int fd, char *buf, size_t buflen);
```

### DESCRIPTION

The function `ttyname()` returns a pointer to the NUL-terminated pathname of the terminal device that is open on the file descriptor `fd`, or `NULL` on error (for example, if `fd` is not connected to a terminal). The return value may point to static data, possibly overwritten by the next call. The function `ttyname_r()` stores this pathname in the buffer `buf` of length `buflen`.

### RETURN VALUE

The function `ttyname()` returns a pointer to a pathname on success. On error, `NULL` is returned, and `errno` is set appropriately. The function `ttyname_r()` returns 0 on success, and an error number upon error.



## 4.5.1.2 实验

- 以下程序通过调用 `ttyname()` 取得文件描述符所对应的终端名称

```
1 #include <unistd.h>
2 #include <stdio.h>
3
4 int main(int argc, char **argv)
5 {
6     fprintf(stdout, "fd 0: %s\n", ttyname(0));
7     fprintf(stdout, "fd 1: %s\n", ttyname(1));
8     fprintf(stdout, "fd 2: %s\n", ttyname(2));
9     fprintf(stdout, "fd 4: %s\n", ttyname(4));
10
11     return 0;
12 }
```

- 输出结果

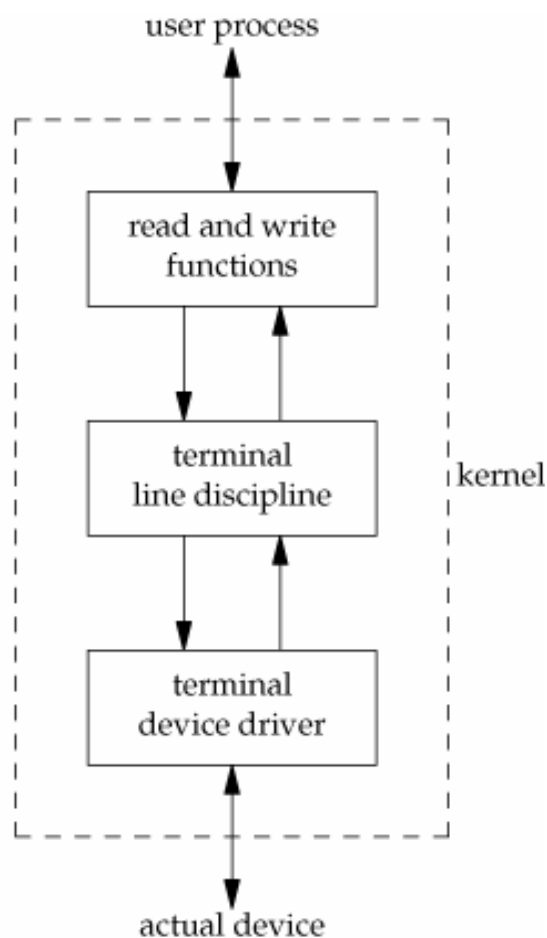
```
[yjs@amigaone ~/work/akae/tmp]$ ./test1
fd 0: /dev/pts/0
fd 1: /dev/pts/0
fd 2: /dev/pts/0
fd 4: (null)
[yjs@amigaone ~/work/akae/tmp]$
```

## 4.5.2 终端登录过程

- 一台PC通常只有一套键盘和显示器，也就是只有一套终端设备，但是可以通过Alt-F1~Alt-F6（或者在X-Window下通过Ctrl-Alt-F1~Ctrl-Alt-F6）可以切换到6个字符终端，相当于有6套虚拟的终端设备，它们共用同一套物理终端设备，对应的设备文件分别是/dev/tty1~/dev/tty6，所以称为**虚拟终端（Virtual Terminal）**
- 设备文件/dev/tty0表示当前虚拟终端，比如切换到Ctrl-Alt-F1的字符终端时/dev/tty0就表示/dev/tty1，切换到Ctrl-Alt-F2的字符终端时/dev/tty0就表示/dev/tty2，就像/dev/tty一样也是一个通用的接口，但它不能表示图形终端窗口所对应的终端
- 在嵌入式系统中，通常使用串口终端，目标板的每个串口对应一个终端设备，比如/dev/ttyS0、/dev/ttyS1等，将主机和目标板用串口线连起来，就可以在主机上通过Linux的minicom或Windows的超级终端工具登录到目标板的系统

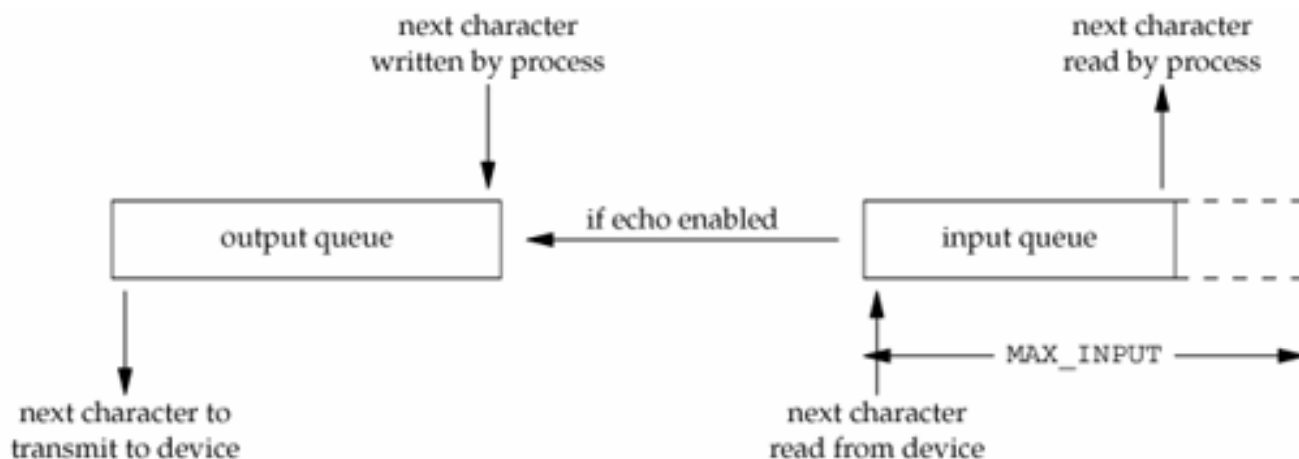
## 4.5.2.1 内核中处理终端设备的模块

- 内核中处理终端设备的模块包括硬件驱动程序和线路规程（Line Discipline）
- 硬件驱动程序负责读写实际的硬件设备，比如从键盘读入字符和把字符输出到显示器
- 线路规程像一个过滤器，对于某些特殊字符并不是让它直接通过，而是做特殊处理
  - 比如在键盘上按下Ctrl-Z，对应的字符并不会被用户程序的read读到，而是被线路规程截获，解释成SIGTSTP信号发给前台进程，通常会使该进程停止
  - 线路规程应该过滤哪些字符和做哪些特殊处理是可以配置的



## 4.5.2.2 终端设备输入和输出队列缓冲区

- 终端设备有输入和输出队列缓冲区
- 以输入队列为例
  - 从键盘输入的字符经线路规程过滤后进入输入队列，用户程序以先进先出的顺序从队列中读取字符
  - 一般情况下，当输入队列满的时候再输入字符会丢失，同时系统会响铃警报
  - 终端可以配置成回显（echo）模式，在这种模式下，输入队列中的每个字符既送给用户程序也送给输出队列，因此我们在命令行键入字符时，该字符不仅可以被程序读取，我们也可以同时同时在屏幕上看到该字符的回显。



## 4.5.3.1 终端登录的过程

- 系统启动时，*init*进程根据配置文件/etc/inittab确定需要打开哪些终端

```
1:2345:respawn:/sbin/getty 38400 tty1
```

- *getty*根据命令行参数打开终端设备作为它的控制终端，把文件描述符0、1、2都指向控制终端，然后提示用户输入帐号。用户输入帐号之后，*getty*的任务就完成了，它再执行*login*程序

```
execl("/bin/login", "login", "-p", username, (char *)0, envp);
```

- *login*程序提示用户输入密码（输入密码期间关闭终端的回显），然后验证帐号密码的正确性。如果密码不正确，*login*进程终止，*init*会重新fork/exec一个*getty*进程。如果密码正确，*login*程序设置一些环境变量，设置当前工作目录为该用户的主目录，然后执行shell

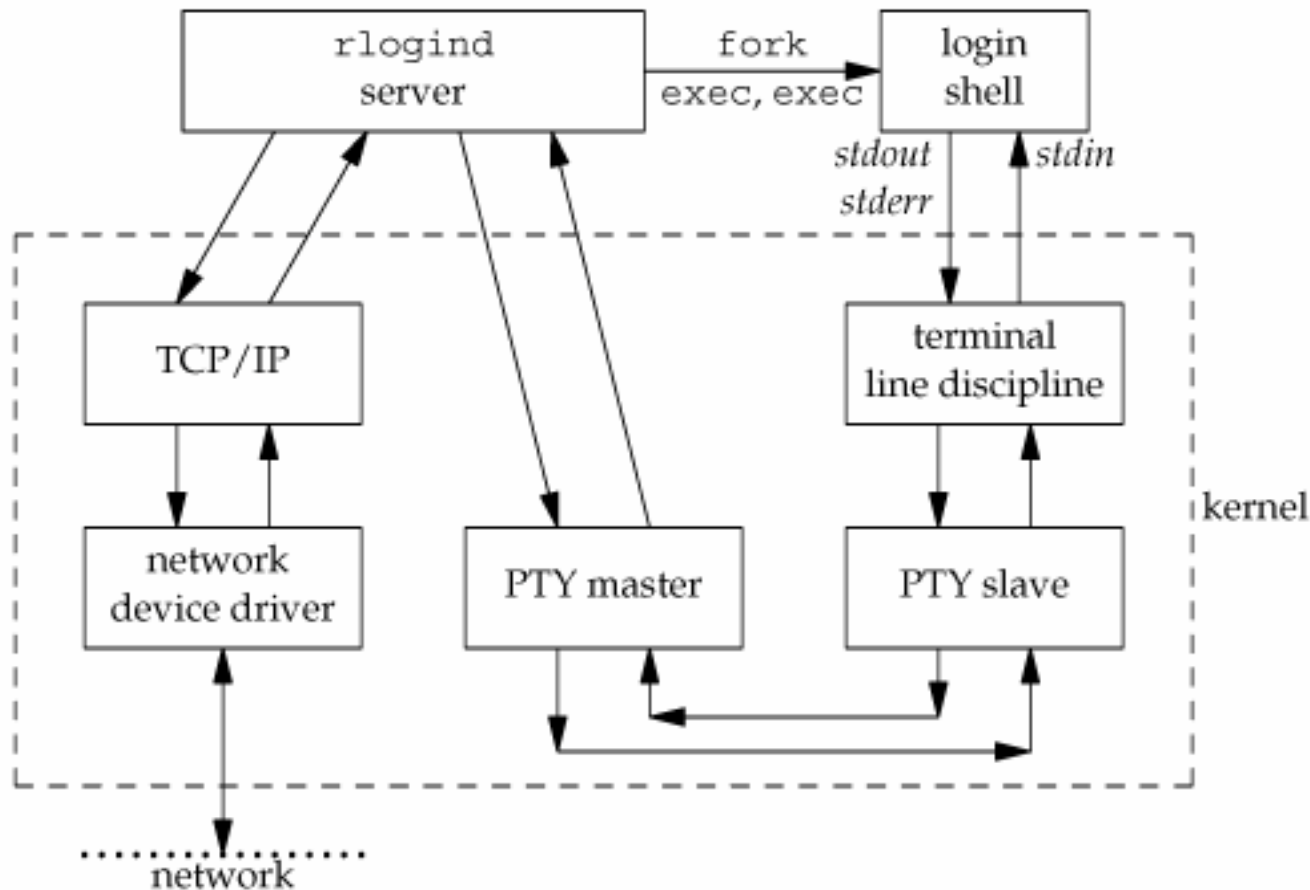
```
execl("/bin/bash", "-bash", (char *)0);
```

没变，文件描述符0、1、2也仍然指向控制终端。由于*fork()*会复制PCB信息，所以由shell启动的其它进程也都是如此

## 4.5.3.2 网络登录过程

- 网络终端或图形终端窗口的数目是不受限制的，这是通过伪终端（Pseudo TTY）实现的
- 一套伪终端由一个主设备（PTY Master）和一个从设备（PTY Slave）组成
  - 主设备在概念上相当于键盘和显示器，只不过它不是真正的硬件而是一个内核模块，操作它的也不是用户而是另外一个进程
  - 从设备与`/dev/tty1`这样的终端设备模块类似，只不过它的底层驱动程序不是访问硬件而是访问主设备
  - 网络终端或图形终端窗口的Shell 进程以及它启动的其它进程都会认为自己的控制终端是伪终端从设备，例如`/dev/pts/0`、`/dev/pts/1`等

## 4.5.3.2.1 伪终端示意图



## 4.5.3.2.2 伪终端登录过程

- 用户通过`telnet`客户端连接服务器。如果服务器配置为独立（Standalone）模式，则在服务器监听连接请求是一个`telnetd`进程，它`fork`出一个`telnetd`子进程来服务客户端，父进程仍监听其它连接请求。
- 另外一种可能是服务器端由系统服务程序`inetd`或`xinetd`监听连接请求，`inetd`称为Internet Super-Server，它监听系统中的多个网络服务端口，如果连接请求的端口号和`telnet`服务端口号一致，则`fork/exec`一个`telnetd`子进程来服务客户端。`xinetd`是`inetd`的升级版，配置更为灵活。
- `telnetd`子进程打开一个伪终端设备，然后再经过`fork`一分为二：父进程操作伪终端主设备，子进程将伪终端从设备作为它的控制终端，并且将文件描述符0、1、2指向控制终端，二者通过伪终端通信，父进程还负责和`telnet`客户端通信，而子进程负责用户的登录过程，提示输入帐号，然后调用`exec`变成`login`进程，提示输入密码，然后调用`exec`变成`Shell`进程。这个`Shell`进程认为自己的控制终端是伪终端从设备，伪终端主设备可以看作键盘显示器等硬件，而操作这个伪终端的“用户”就是父进程`telnetd`。
- 当用户输入命令时，`telnet`客户端将用户输入的字符通过网络发给`telnetd`服务器，由`telnetd`服务器代表用户将这些字符输入伪终端。`Shell`进程并不知道自己连接的是伪终端而不是真正的键盘显示器，也不知道操作终端的“用户”其实是`telnetd`服务器而不是真正的用户。`Shell`仍然解释执行命令，将标准输出和标准错误输出写到终端设备，这些数据最终由`telnetd`服务器发回给`telnet`客户端，然后显示给用户看。



### 4.5.3.3 伪终端分类

- BSD系列的UNIX在/dev目录下创建很多ptyXX和ttyXX设备文件，XX由字母和数字组成，ptyXX是主设备，相对应的ttyXX 是从设备，伪终端的数目取决于内核配置。
- 在SYS V系列的UNIX上，伪终端主设备是/dev/ptmx，“mx”表示Multiplex，意思是多个主设备复用同一个设备文件，每打开一次/dev/ptmx，内核就分配一个主设备，同时在/dev/pts目录下创建一个从设备文件，当终端关闭时就从/dev/pts目录下删除相应的从设备文件。
- Linux同时支持上述两种伪终端，目前的标准倾向于SYS V 的伪终端。

## 4.6 作业控制

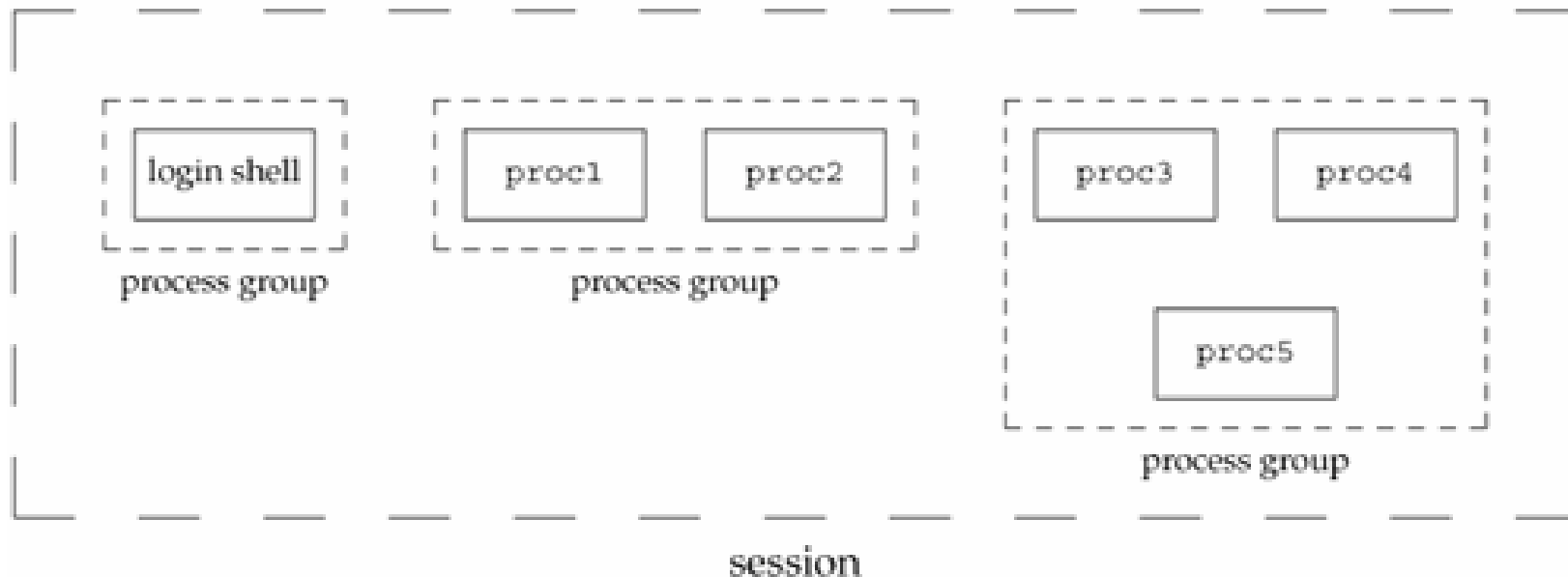
---

- Shell分前后台来控制的不是进程而是作业（Job）或者进程组（Process Group）
  - 一个前台作业可以由多个进程组成，一个后台作业也可以由多个进程组成
  - Shell可以同时运行一个前台作业和任意多个后台作业，这称为作业控制（Job Control）

## 4.6.1 作业控制的实例

### ■ 例如前面提到的5个进程：

- proc1和proc2属于同一个后台进程组
- proc3、proc4、proc5 属于同一个前台进程组
- Shell进程本身属于一个单独的进程组
- 这些进程组的控制终端相同，它们属于同一个Session。当用户在控制终端输入特殊的控制键（例如Ctrl-C）时，内核会发送相应的信号（例如SIGINT）给前台进程组的所有进程。



## 4.6.2 进程组与作业

### ■ 从Session和进程组的角度重新来看登录和执行命令的过程

- `getty`或`telnetd`进程在打开终端设备之前调用`setsid()`函数创建一个新的Session，该进程称为Session Leader，该进程的id也可以看作Session的id，然后该进程打开终端设备作为这个Session中所有进程的控制终端。在创建新Session的同时也创建了一个新的进程组，该进程是这个进程组的Process Group Leader，该进程的id也是进程组的id
- 在登录过程中，`getty`或`telnetd`进程变成`login`，然后变成Shell，但仍然是同一个进程，仍然是Session Leader
- 由Shell进程fork出的子进程本来具有和Shell相同的Session、进程组和控制终端，但是Shell调用`setpgid`函数将作业中的某个子进程指定为一个新进程组的Leader，然后调用`setpgid`将该作业中的其它子进程也转移到这个进程组中
  - 如果这个进程组需要在前台运行，就调用`tcsetpgrp()`函数将它设置为前台进程组，由于一个Session只能有一个前台进程组，所以Shell所在的进程组就自动变成后台进程组

### ■ 在上面的例子中

- `proc3`、`proc4`、`proc5`被Shell放到同一个前台进程组，其中有一个进程是该进程组的leader，Shell调用`wait()`等待它们运行结束
  - 一旦它们全部运行结束，Shell就调用`tcsetpgrp()`函数将自己提到前台继续接受命令
  - 需要注意的是，如果`proc3`、`proc4`、`proc5`中的某个进程又fork出子进程，子进程也属于同一进程组，但是Shell并不知道子进程的存在，也不会调用`wait`等待它结束，也就是说，`proc3 | proc4 | proc5`是Shell的作业，而这个子进程不是，这是作业和进程组在概念上的区别
  - 一旦作业运行结束，Shell 就把自己提到前台，如果原来的前台进程组还存在（如果这个子进程还没终止），则它自动变成后台进程组

## 4.6.3 作业相关的信号

---

### ■ SITTSTP

- 挂起，Ctrl-Z产生，送给前台进程组中的所有进程，后台进程组不受影响

### ■ SIGTTIN

- 后台作业试图读取控制台输入时，会被SIGTTIN阻止，该信号通常会暂停后台作业

### ■ SIGTTOU

- 阻止后台作业向控制台写

### ■ SIGCONT

- 唤醒被挂起的作业

## 5. 守护进程

## 5.1 守护进程(Daemon process)

---

- 守护进程，也就是通常所说的Daemon进程，是UNIX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的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，大多数服务都是用守护进程实现的

## 5.2 Linux守护进程

- 在Linux中，每一个系统与用户进行交流的界面称为终端，每一个从此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，相应的进程都会被自动关闭
  - 但是守护进程却能够突破这种限制，它从被执行开始运转，直到整个系统关闭才会退出
  - 如果想让某个进程不因为用户或终端或其他的变化而受到影响，就必须把这个进程变成一个守护进程
- 编写步骤
  - 创建子进程，父进程退出
  - 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
  - 关闭文件描述符
  - 通常守护进程要向syslog输出log信息，所以需要打开syslog，此时调用openlog()



## 5.2.1 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(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

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

## 5.2.2 在子进程中创建新会话

- **setsid()函数作用**
  - **setsid()函数**用于创建一个新的会话，并自任该会话组的组长
  - 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
- 由于调用**fork()函数**时，子进程全盘拷贝了父进程的进会话期、进程组、控制终端等，虽然之后父进程退出了，但原先的会话期、进程组、控制终端等并没有改变，因此，还没有真正意义上独立开来，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。

## 5.2.3 改变当前目录为根目录

- 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）
- 通常的做法是让“/”作为守护进程的当前工作目录

## 5.2.4 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
- 调用`fork()`新创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
  - 把文件权限掩码设置为0，可以大大增加该守护进程的灵活性
  - 设置文件权限掩码的函数是`umask()`
  - 重设文件创建掩码的目的是让守护进程能够尽可能的不受环境的影响
  - 重设文件创建掩码也是出于安全的考虑
  - 通常的使用方法为`umask(0)`
- `~umask & user_set_permission`
- `~0777 & 0777 = 0000`

## 5.2.5 关闭文件描述符

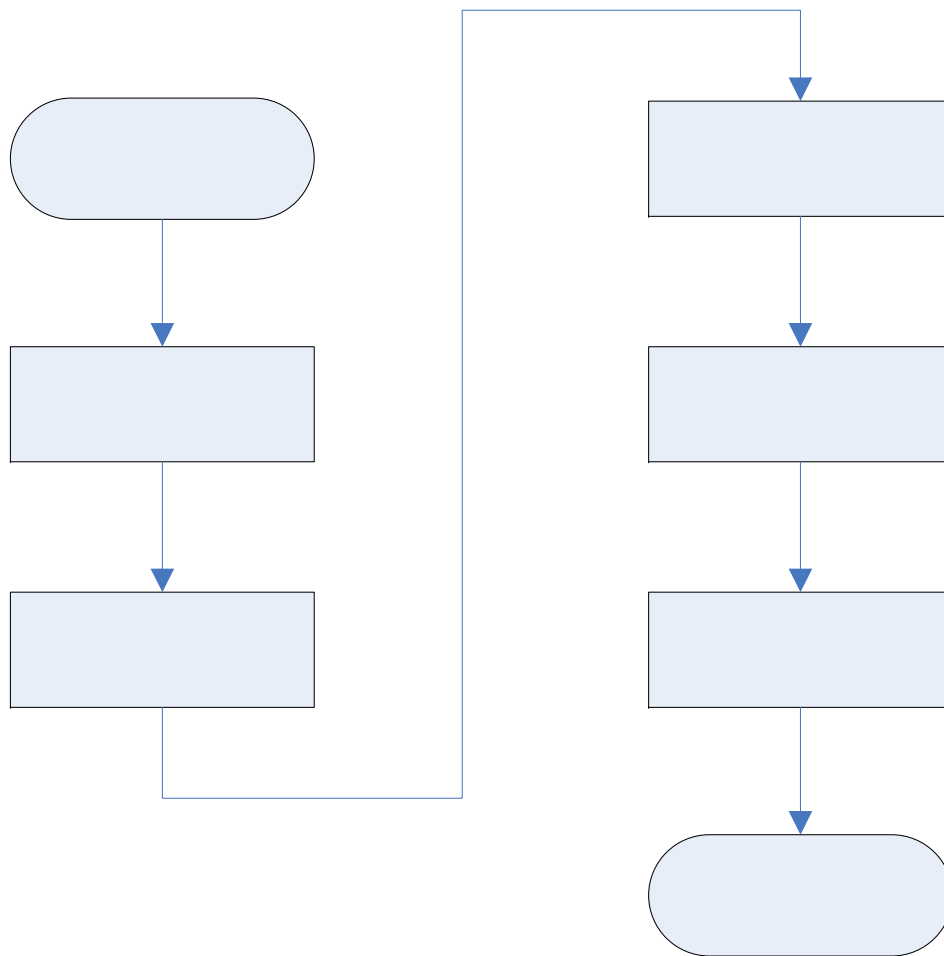
- 同文件权限掩码一样，用fork新建的子进程会从父进程那里继承一些已经打开了的文件。这些被打开的文件可能永远不会被守护进程读或写，但它们一样消耗系统资源，而且可能导致所在的文件系统无法卸下
- 在上面的第二步后，守护进程已经与所属的控制终端失去了联系。因此从终端输入的字符不可能达到守护进程，守护进程中用常规的方法（如printf）输出的字符也不可能在终端上显示出来。所以，文件描述符为0、1和2的三个文件（常说的输入、输出和报错这三个文件）已经失去了存在的价值，也应被关闭

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

## 5.2.6 openlog()

- *openlog()*函数用来打开一个到系统日志记录程序(*syslogd*)的连接，打开之后就可以用*syslog()*或*vsyslog()*函数向系统日志里添加信息了
- *openlog()*的参数
  - 第一个参数*ident*是一个标记，*ident*所表示的字符串将固定地加在每行日志的前面以标识这个日志，通常就写成当前程序的名称以作标记
  - 第二个参数*option*是下列值做与运算的结果：LOG\_CONS， LOG\_NDELAY， LOG\_NOWAIT， LOG\_ODELAY， LOG\_PERROR， LOG\_PID， 各值意义请参考man *openlog*手册
  - 第三个参数指明记录日志的程序类型
- *syslog()*函数
  - *syslog()*函数用于把日志消息发给系统程序*syslogd*去记录
    - *void syslog(int priority, const char \*format, ...);*
  - 第一个参数是消息的紧急级别
  - 第二个参数是消息的格式
  - 后面的其他参数是格式对应的参数，就是*printf()*函数的参数意义相同
- *closelog()*函数用来关闭*openlog()*打开的连接
- *openlog()*不是必须的，如果没有事先调用*openlog()*，那么会在调用*syslog()*时自动调用*openlog()*

## 5.3 守护进程创建流程



## 6. 信号



## 6. 信号-主要内容

---

### ■ 概述

- 认识信号
- 信号的概念
- 信号的产生条件
- 信号的处理方式
- 信号列表
- 信号的默认处理方式
- 信号用途
- 信号本质
- 信号来源
- 信号分类
- 使用信号

### ■ 产生信号

- 通过终端按键产生信号
- 调用系统函数向进程发信号
- 由软件条件产生信号

## 6. 信号-主要内容(cont.)

---

### ■ 阻塞信号

- 信号的生命周期
- 信号在内核中的表示
- 信号集及其操作函数
- *sigprocmask()*
- *sigpending()*

### ■ 捕捉信号

- 内核如何实现信号的捕捉
- *signal()*
- *sigaction()*

### ■ 其他的问题

- 信号与进程
- 中断的系统调用
- 可重入问题
- 竞态条件与*sigsuspend()*函数
- SIGCHLD信号

## 6.1 信号概述

---

- 认识信号
- 信号的概念
- 信号的产生条件
- 信号的处理方式
- 信号列表
- 信号的默认处理方式
- 信号用途
- 信号本质
- 信号来源
- 信号分类
- 使用信号

## 6.1.1 认识信号

### ■ 先从我们最熟悉的场景说起

- 用户输入命令，在 Shell下启动一个前台进程
- 用户按下Ctrl-C，这个键盘输入产生一个硬件中断
  - Ctrl-C产生的信号只能发给前台进程
- 如果CPU当前正在执行这个进程的代码，则该进程的用户空间代码暂停执行，CPU从用户态切换到内核态处理硬件中断
- 终端驱动程序将Ctrl-C解释成一个SIGINT信号，记在该进程的PCB中（也可以说发送了一个SIGINT信号给该进程）。
- 当某个时刻要从内核返回到该进程的用户空间代码继续执行之前，首先处理PCB中记录的信号，发现有一个SIGINT信号待处理，而这个信号的默认处理动作是终止进程，所以直接终止进程而不再返回它的用户空间代码执行
  - 信号相对于进程的控制流程来说是异步（Asynchronous）的

### ■ 认识信号

- 信号是软件中断
- 很多比较重要的应用程序都需处理信号
- 信号提供了一种处理异步事件的方法：终端用户键入中断键，则会通过信号机制停止一个程序
- 产生信号的事件对进程而言是随机出现的
- 如果处理不当，信号会造成程序运行状态的混乱

## 6.1.2 信号的概念

---

- 每个信号都有一个名字, 这些名字都以三个字符SIG开头
  - SIGABRT是夭折信号, 当进程调用`abort()`函数时产生这种信号。
  - SIGALRM是闹钟信号, 当由`alarm()`函数设置的时间已经超过后产生此信号
- 在头文件`signal.h`中, 信号都被定义为正整数(信号编号), 没有一个信号其编号为0

## 6.1.3 信号的产生条件

- 用户在终端按下某些键时，终端驱动程序会发送信号给前台进程
  - Ctrl-C产生SIGINT信号
  - Ctrl-\产生SIGQUIT信号
  - Ctrl-Z产生SIGTSTP信号
    - 可使前台进程停止，这个信号通常用于作业控制
- 硬件异常产生信号，这些条件由硬件检测到并通知内核，然后内核向当前进程发送适当的信号
  - 当前进程执行了除以0的指令，CPU的运算单元会产生异常，内核将这个异常解释为SIGFPE信号发送给进程
  - 当前进程访问了非法内存地址，MMU会产生异常内核将这个异常解释为SIGSEGV信号发送给进程
- 一个进程调用kill(2)可以发送信号给另一个进程
  - 另一个进程如何处理？
- 通过kill(1)发送信号给某个进程
  - kill(1)通过调用kill(2)实现的
  - 如果不明确指定信号则发送SIGTERM信号，该信号的默认处理动作是终止进程。
- 当内核检测到某种软件条件发生时也可以通过信号通知进程，例如闹钟超时产生SIGALRM信号，向读端已关闭的管道写数据时产生SIGPIPE信号

## 6.1.4 信号的处理方式

- 用户程序可调用`signal(2)`或者`sigaction(2)`通知内核如何处理某种信号
- 信号有三种处理方式
  - 忽略信号
    - 大多数信号都可使用这种方式进行处理,
    - 但有两种信号(**SIGKILL**和**SIGSTOP**)却决不能被忽略
    - 其原因是: 它们向超级用户提供一种使进程终止或停止的可靠方法。另外, 如果忽略某些由硬件异常产生的信号(例如非法存储访问或除以0), 则进程的行为是未定义的。
  - 捕捉信号
    - 为了做到这一点要通知内核在某种信号发生时, 调用一个用户函数。在用户函数中, 可执行用户希望对这种事件进行的处理。
    - 若编写一个命令解释器, 当用户用键盘产生中断信号时, 很可能希望返回到程序的主循环, 终止系统正在为该用户执行的命令。
    - 如果捕捉到**SIGCHLD**信号, 则表示子进程已经终止, 所以此信号的捕捉函数可以调用`waitpid()`以取得该子进程的进程ID以及它的终止状态。
    - 如果进程创建了临时文件, 那么可能要为**SIGTERM**信号编写一个信号捕捉函数以清除临时文件(`kill(1)`命令传送的系统默认信号是终止信号)。
  - 执行系统默认动作

## 6.1.5 信号列表

### ■ 使用`kill -l`命令查看信号

```
[yjs@amigaone /data]$  
[yjs@amigaone /data]$  
[yjs@amigaone /data]$ 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  
[yjs@amigaone /data]$
```



## 6.1.5.1 信号列表

- Signals described in the original POSIX.1-1990 standard

Signal	Value	Action	Comment
<b>SIGHUP</b>	1	Term	Hangup detected on controlling terminal or death of controlling process
<b>SIGINT</b>	2	Term	Interrupt from keyboard
<b>SIGQUIT</b>	3	Core	Quit from keyboard
<b>SIGILL</b>	4	Core	Illegal Instruction
<b>SIGABRT</b>	6	Core	Abort signal from <b>abort(3)</b>
<b>SIGFPE</b>	8	Core	Floating point exception
<b>SIGKILL</b>	9	Term	Kill signal
<b>SIGSEGV</b>	11	Core	Invalid memory reference
<b>SIGPIPE</b>	13	Term	Broken pipe: write to pipe with no readers
<b>SIGALRM</b>	14	Term	Timer signal from <b>alarm(2)</b>
<b>SIGTERM</b>	15	Term	Termination signal
<b>SIGUSR1</b>	30,10,16	Term	User-defined signal 1
<b>SIGUSR2</b>	31,12,17	Term	User-defined signal 2
<b>SIGCHLD</b>	20,17,18	Ign	Child stopped or terminated
<b>SIGCONT</b>	19,18,25	Cont	Continue if stopped
<b>SIGSTOP</b>	17,19,23	Stop	Stop process
<b>SIGTSTP</b>	18,20,24	Stop	Stop typed at tty
<b>SIGTTIN</b>	21,21,26	Stop	tty input for background process
<b>SIGTTOU</b>	22,22,27	Stop	tty output for background process

## 6.1.5.2 信号列表

- Signals not in the POSIX.1-1990 standard but described in SUSv2 and POSIX.1-2001.
  - SUSv2 - Single UNIX Specification Version 2

Signal	Value	Action	Comment
-----			
<b>SIGBUS</b>	10,7,10	Core	Bus error (bad memory access)
<b>SIGPOLL</b>		Term	Pollable event (Sys V). Synonym for <b>SIGIO</b>
<b>SIGPROF</b>	27,27,29	Term	Profiling timer expired
<b>SIGSYS</b>	12,-,12	Core	Bad argument to routine (SVr4)
<b>SIGTRAP</b>	5	Core	Trace/breakpoint trap
<b>SIGURG</b>	16,23,21	Ign	Urgent condition on socket (4.2BSD)
<b>SIGVTALRM</b>	26,26,28	Term	Virtual alarm clock (4.2BSD)
<b>SIGXCPU</b>	24,24,30	Core	CPU time limit exceeded (4.2BSD)
<b>SIGXFSZ</b>	25,25,31	Core	File size limit exceeded (4.2BSD)

## 6.1.5.3 信号列表

### ■ Other signals

Signal	Value	Action	Comment
SIGIOT	6	Core	IOT trap. A synonym for <b>SIGABRT</b>
SIGEMT	7,-,7	Term	
SIGSTKFLT	-,16,-	Term	Stack fault on coprocessor (unused)
SIGIO	23,29,22	Term	I/O now possible (4.2BSD)
SIGCLD	-, -,18	Ign	A synonym for <b>SIGCHLD</b>
SIGPWR	29,30,19	Term	Power failure (System V)
SIGINFO	29,-,-		A synonym for <b>SIGPWR</b>
SIGLOST	-, -, -	Term	File lock lost
SIGWINCH	28,28,20	Ign	Window resize signal (4.3BSD, Sun)
SIGUNUSED	-,31,-	Term	Unused signal (will be <b>SIGSYS</b> )

## 6.1.6 信号的默认处理方式

名 字	说 明	ANSI C	POSIX.1	SVR4	4.3+BSD	缺省动作
SIGABRT	异常终止 (abort)	•	•	•	•	终止 w/core
SIGALRM	超时 (alarm)		•	•	•	终止
SIGBUS	硬件故障			•	•	终止 w/core
SIGCHLD	子进程状态改变		作业	•	•	忽略
SIGCONT	使暂停进程继续		作业	•	•	继续/忽略
SIGEMT	硬件故障			•	•	终止 w/core
SIGFPE	算术异常	•	•	•	•	终止 w/core
SIGHUP	连接断开		•	•	•	终止
SIGILL	非法硬件指令	•	•	•	•	终止 w/core
SIGINFO	键盘状态请求				•	忽略
SIGINT	终端中断符	•	•	•	•	终止
SIGIO	异步 I/O			•	•	终止/忽略
SIGIOT	硬件故障			•	•	终止 w/core
SIGKILL	终止		•	•	•	终止
SIGPIPE	写至无读进程的管道		•	•	•	终止
SIGPOLL	可轮询事件 (poll)			•		终止
SIGPROF	梗概时间超时 (setitimer)			•	•	终止
SIGPWR	电源失效/再起动作			•		忽略
SIGQUIT	终端退出符		•	•	•	终止 w/core
SIGSEGV	无效存储访问	•	•	•	•	终止 w/core
SIGSTOP	停止		作业	•	•	暂停进程

## 6.1.6 信号的默认处理方式(cont.)

名 字	说 明	ANSI C	POSIX.1	SVR4	4.3+BSD	缺省动作
SIGSYS	无效系统调用			•	•	终止w/core
SIGTERM	终止	•	•	•	•	终止
SIGTRAP	硬件故障			•	•	终止w/core
SIGTSTP	终端挂起符		作业	•	•	停止进程
SIGTTIN	后台从控制tty读		作业	•	•	停止进程
SIGTTOU	后台向控制tty写		作业	•	•	停止进程
SIGURG	紧急情况			•	•	忽略
SIGUSR1	用户定义信号		•	•	•	终止
SIGUSR2	用户定义信号		•	•	•	终止
SIGVTALRM	虚拟时间闹钟 (setitimer)			•	•	终止
SIGWINCH	终端窗口大小改变			•	•	忽略
SIGXCPU	超过CPU限制 (setrlimit)			•	•	终止w/core
SIGXFSZ	超过文件长度限制 (setrlimit)			•	•	终止w/core

“终止w/core”表示在进程当前工作目录的core文件中复制了该进程的存储图像

## 6.1.7 信号用途

- SIGABRT调用abort函数时产生此信号, 进程异常终止
- SIGALRM超过用alarm函数设置的时间时产生此信号
- SIGBUS指示一个实现定义的硬件故障。
- SIGCHLD在一个进程终止或停止时, SIGCHLD信号被送给其父进程
- SIGCONT此作业控制信号送给需要继续运行的处于停止状态的进程。如果接收到此信号的进程处于停止状态, 则系统默认动作是使该进程继续运行, 否则默认动作是忽略此信号。
- SIGEMT指示一个实现定义的硬件故障
- SIGFPE此信号表示一个算术运算异常, 例如除以0, 浮点溢出等。
- SIGHUP如果终端界面检测到一个连接断开, 则将此信号送给与该终端相关的控制进程(对话期首进程)
- SIGILL此信号指示进程已执行一条非法硬件指令
- SIGINFO 当用户按状态键(一般采用Ctrl-T)时, 终端驱动程序产生此信号并送至前台进程组中的每一个进程
- SIGINT 当用户按中断键(一般采用Delete或Ctrl-C)时, 终端驱动程序产生此信号并送至前台进程组中的每一个进程
- SIGIO 此信号指示一个异步I/O事件

## 6.1.7 信号用途(cont.)

- SIGIOT 这指示一个实现定义的硬件故障
- SIGKILL 这是两个不能被捕捉或忽略信号中的一个，它向系统管理员提供了一种可以杀死任一进程的可靠方法
- SIGPIPE 如果在读进程已终止时写管道，则产生此信号
- SIGPOLL 这是一种SVR4信号，当在一个可轮询设备上发生一特定事件时产生此信号
- SIGPROF 当setitimer(2)函数设置的梗概统计间隔时间已经超过时产生此信号
- SIGPWR 这是一种SVR4信号，它依赖于系统。它主要用于具有不间断电源(UPS)的系统上。如果电源失效，则UPS起作用，而且通常软件会接到通知。
- SIGQUIT 当用户在终端上按退出键(一般采用Ctrl-\)时，产生此信号，并送至前台进程组中的所有进程
- SIGSEGV 指示进程进行了一次无效的存储访问
- SIGSTOP 这是一个作业控制信号，它停止一个进程
- SIGSYS 指示一个无效的系统调用
- SIGTERM 这是由kill(1)命令发送的系统默认终止信号

## 6.1.7 信号用途(cont.)

- SIGTRAP 指示一个实现定义的硬件故障
- SIGTSTP 交互停止信号，当用户在终端上按挂起键(一般采用Ctrl-Z)时，终端驱动程序产生此信号
- SIGTTIN 当一个后台进程组进程试图读其控制终端时，终端驱动程序产生此信号
- SIGTTOU 当一个后台进程组进程试图写其控制终端时产生此信号
- SIGURG 此信号通知进程已经发生一个紧急情况
- SIGUSR1 这是一个用户定义的信号，可用于应用程序
- SIGUSR2 这是一个用户定义的信号，可用于应用程序
- SIGVTALRM 当一个由setitimer(2)函数设置的虚拟间隔时间已经超过时产生此信号
- SIGWINCH SVR4和4.3+BSD内核保持与每个终端或伪终端相关联的窗口的大小
- SIGXCPU SVR4和4.3+BSD支持资源限制的概念
- SIGXFSZ 如果进程超过了其软文件长度限制



## 6.1.8 信号本质

---

- 信号是在软件层次上对中断机制的一种模拟
  - 在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的
  - 信号是异步的，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达
- 信号是进程间通信机制中唯一的异步通信机制
  - 可以看作是异步通知，通知接收信号的进程有哪些事情发生了。
  - 信号机制经过Posix实时扩展后，功能更加强大，除了基本通知功能外，还可以传递附加信息

## 6.1.9 信号来源

---

- 信号事件的发生有两个来源
- 硬件来源
  - 键盘操作
  - 其它硬件故障
- 软件来源
  - *kill(1), kill(2)*
  - *raise(), abort()*
  - *alarm(), setitimer(), sigqueue()*
  - 一些非法运算等操作



## 6.1.10 信号分类 – 依据可靠性

### ■ 不可靠信号

- 早期UNIX下的不可靠信号主要指的是进程可能对信号做出错误的反应以及信号可能丢失
- 信号值小于SIGRTMIN的信号都是不可靠信号
- Linux支持不可靠信号，但是对不可靠信号机制做了改进
  - 在调用完信号处理函数后，不必重新调用该信号的安装函数
  - Linux下的不可靠信号问题主要指的是信号可能丢失

### ■ 可靠信号

- 在保留不可靠信号的前提下（由于原来定义的信号已有许多应用，不好再做改动），又新增加了一些信号，并在一开始就把它们定义为可靠信号，这些信号支持排队，不会丢失
- 信号的发送和安装也出现了新版本
  - 信号发送函数sigqueue()
  - 信号安装函数sigaction()
- Posix对可靠信号机制做了标准化
- 信号值位于SIGRTMIN和SIGRTMAX之间的信号都是可靠信号
- Linux在支持新版本的信号安装函数sigaction()以及信号发送函数sigqueue()
  - 仍然支持早期的signal()信号安装函数
  - 支持信号发送函数kill(2)



## 6.1.10 信号分类 — 依据实时性

- 早期UNIX系统只定义了32种信号
  - 前32种信号已经有了预定义值，每个信号有了确定的用途及含义，并且每种信号都有各自的缺省动作
    - 如按键盘的Ctrl-C时，会产生SIGINT信号，对该信号的默认反应就是进程终止
- 后32个信号表示实时信号，等同于可靠信号
  - 保证了发送的多个实时信号都被接收。
  - 实时信号是POSIX标准的一部分，可用于应用进程
- 非实时信号都不支持排队，都是不可靠信号
- 实时信号都支持排队，都是可靠信号

## 6.1.11 使用信号

---

- 系统运行
  - 系统shutdown时发送SIGTERM通知所有的进程
- 系统管理
  - 通过kill(1)来对进程进行管理
- 进程调度
  - 信号驱动I/O
- 进程间通讯
  - 用于同步
  - 用于通知
    - SIGCHLD
    - SIGALRM

## 6.2. 产生信号

---

- 通过终端按键产生信号
- 调用系统函数向进程发信号
- 由软件条件产生信号



## 6.2.1 通过终端按键产生信号

### ■ 验证SIGQUIT

- SIGQUIT的默认处理动作是终止进程并且Core Dump
  - Core dump
    - 当一个进程要异常终止时，可以选择把进程的用户空间内存数据全部保存到磁盘上，文件名通常是core，这叫做Core dump
    - 进程异常终止通常是因为有bug，比如非法内存访问导致段错误，事后可以用调试器检查core 文件以查清错误原因，这叫做Post-mortem Debug(事后调试, 验尸调试)。
    - 一个进程允许产生多大的core 文件取决于进程的Resource Limit（这个信息保存在PCB中）。默认是不允许产生core 文件的，因为core 文件中可能包含用户密码等敏感信息，不安全。在开发调试阶段可以用ulimit 命令改变这个限制，允许产生core 文件。
- SIGQUIT信号可以通过Ctrl-\产生

## 6.2.1 通过终端按键产生信号

### ■ 验证代码

```
[yjs@amigaone ~/work/akae]$ cat test1.c
#include <unistd.h>

int main (void)
{
    while (1);

    return 0;
}
[yjs@amigaone ~/work/akae]$ gcc -o test1 test1.c
[yjs@amigaone ~/work/akae]$ ulimit -c 10240
[yjs@amigaone ~/work/akae]$ ulimit -c
10240
[yjs@amigaone ~/work/akae]$ ./test1
Quit (core dumped)
[yjs@amigaone ~/work/akae]$
```

### ■ 运行test1之后，键盘输入Ctrl-\，屏幕输出Quit (core dumped)



## 6.2.2 调用系统函数向进程发信号

- 通过`kill(1)`发送信号，相当于调用`kill(2)`

- 参数:

- `SIG<name>`
- `<name>`
- `<signo>`

```
[yjs@amigaone ~/work/akae]$ cat test1.c
#include <unistd.h>

int main (void)
{
    while (1);

    return 0;
}

[yjs@amigaone ~/work/akae]$ ./test1 &
[1] 29053
[yjs@amigaone ~/work/akae]$ ps afx | grep "test1"
29053 pts/3    R      0:02      |          \_ ./test1
29069 pts/3    S+     0:00      |          \_ grep test1
[yjs@amigaone ~/work/akae]$ kill -SIGSEGV 29053
[yjs@amigaone ~/work/akae]$
[1]+  Segmentation fault      (core dumped) ./test1
[yjs@amigaone ~/work/akae]$ ./test1 &
[1] 31861
[yjs@amigaone ~/work/akae]$ ps afx | grep "test1"
31861 pts/3    R      0:05      |          \_ ./test1
31920 pts/3    S+     0:00      |          \_ grep test1
[yjs@amigaone ~/work/akae]$ kill -11 29053
bash: kill: (29053) - No such process
[yjs@amigaone ~/work/akae]$ kill -11 31861
[yjs@amigaone ~/work/akae]$
[1]+  Segmentation fault      (core dumped) ./test1
[yjs@amigaone ~/work/akae]$ ./test1 &
[1] 32428
[yjs@amigaone ~/work/akae]$ ps afx | grep "test1"
32428 pts/3    R      0:02      |          \_ ./test1
32440 pts/3    S+     0:00      |          \_ grep test1
[yjs@amigaone ~/work/akae]$ kill -SEGV 32428
[yjs@amigaone ~/work/akae]$
[1]+  Segmentation fault      (core dumped) ./test1
[yjs@amigaone ~/work/akae]$
```

## 6.2.2.1 *kill(2)/raise(3)*

### ■ *kill(2)*函数将信号发送给进程或进程组

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

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

- *pid*>0 将信号发送给进程ID为*pid*的进程。
- *pid*=0 将信号发送给其进程组ID等于发送进程的进程组ID，而且发送进程有许可权向其发送信号的所有进程。
- *pid*<0 将信号发送给其进程组ID等于*pid*绝对值，而且发送进程有许可权向其发送信号的所有进程。
- *pid*=-1 将信号发送给所有的进程

### ■ *raise(3)*函数则允许进程向自身发送信号

```
#include <signal.h>

int raise(int sig);
```

- *raise(sig)*等价于*kill(getpid(), sig)*

## 6.2.2.2 产生信号 – *kill(2)*

- *kill(1)*命令是调用*kill(2)*系统调用实现的
- *kill(2)* 可以给一个指定的进程发送指定的信号

### NAME

`kill` - send signal to a process

### SYNOPSIS

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

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

### DESCRIPTION

The `kill` system call can be used to send any signal to any process group or process.

If `pid` is positive, then signal `sig` is sent to `pid`.

If `pid` equals 0, then `sig` is sent to every process in the process group of the current process.

If `pid` equals -1, then `sig` is sent to every process except for process 1 (init), but see below.

If `pid` is less than -1, then `sig` is sent to every process in the process group `-pid`.

If `sig` is 0, then no signal is sent, but error checking is still performed.

### RETURN VALUE

On success, zero is returned. On error, -1 is returned, and `errno` is set appropriately.

## 6.2.2.3 产生信号 – *raise(3)*

- *raise(3)*函数可以给当前进程发送指定的信号（自己给自己发信号）
- 等价于`kill(getpid(), sig)`

```
NAME
    raise - send a signal to the current process

SYNOPSIS
    #include <signal.h>

    int raise(int sig);

DESCRIPTION
    The raise() function sends a signal to the current process. It is equivalent to

        kill(getpid(), sig);

RETURN VALUE
    0 on success, nonzero for failure.
```

## 6.2.2.4 产生信号 – *abort(3)*

- *abort(3)*函数使当前进程接收到SIGABRT信号而异常终止

### NAME

`abort` - cause abnormal program termination

### SYNOPSIS

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

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

### DESCRIPTION

The `abort()` function causes abnormal program termination unless the signal SIGABRT is caught and the signal handler does not return. If the `abort()` function causes program termination, all open streams are closed and flushed.

If the SIGABRT signal is blocked or ignored, the `abort()` function will still override it.

### RETURN VALUE

The `abort()` function never returns.

## 6.2.3 由软件条件产生信号

- SIGPIPE是一种由软件条件产生的信号
  - 如果所有指向管道读端的文件描述符都关闭了（管道读端的引用计数等于0），这时有进程向管道的写端write，那么该进程会收到信号SIGPIPE，通常会导致进程异常终止。
  - 网络编程中也会有同样的问题
- 调用alarm()函数可以设定一个闹钟，即告诉内核在seconds秒之后给当前进程发SIGALRM信号，该信号的默认处理动作是终止当前进程

```
NAME
    alarm - set an alarm clock for delivery of a signal

SYNOPSIS
    #include <unistd.h>

    unsigned int alarm(unsigned int seconds);

DESCRIPTION
    alarm arranges for a SIGALRM signal to be delivered to the process in seconds seconds.

    If seconds is zero, no new alarm is scheduled.

    In any event any previously set alarm is cancelled.

RETURN VALUE
    alarm returns the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm.
```

## 6.2.3.1 产生信号 - *alarm*(2)

- 使用`alarm()`函数可以设置一个时间值(闹钟时间)，在将来的某个时刻该时间值会被超过。当所设置的时间值被超过后，产生SIGALRM信号。如果不忽略或不捕捉此信号，则其默认动作是终止该进程。

```
#include <unistd.h>

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

- `alarm()`的参数`seconds`的值是秒数，经过了指定的`seconds`秒后会产生信号SIGALRM
- 每个进程只能有一个闹钟时间。如果在调用`alarm()`时，以前已为该进程设置过闹钟时间，而且它还没有超时，则该闹钟时间的余留值作为本次`alarm()`函数调用的值返回。以前登记的闹钟时间则被新值代换。
- 如果有以前登记的尚未超过的闹钟时间，而且`seconds`值是0，则取消以前的闹钟时间，其余留值仍作为函数的返回值。
- 虽然SIGALRM的默认动作是终止进程，但是大多数使用闹钟的进程捕捉此信号

## 6.2.3.2 产生信号 – *alarm(2)*

### ■ 源代码

```
1 #include <unistd.h>
2 #include <stdio.h>
3
4 int main (int argc, char **argv)
5 {
6     int counter;
7
8     alarm(1);
9
10    for (counter = 0; 1; counter++)
11        printf ("counter = %d\n", counter);
12
13    return 0;
14 }
```

### ■ 输出结果

```
[yjs@amigaone ~/work/akae]$ ./alarm
counter = 0
counter = 1
counter = 2
counter = 3
counter = 4
... ..
counter = 18242
counter = 18243
counter = 18244
counter = 1824Alarm clock
[yjs@amigaone ~/work/akae]$
```



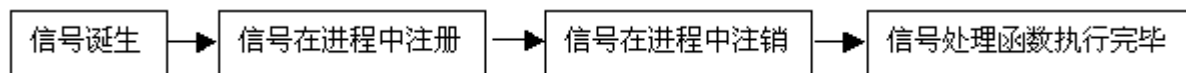
## 6.3. 阻塞信号

---

- 信号的生命周期
- 信号在内核中的表示
- 信号集及其操作函数
- *sigprocmask()*
- *sigpending()*

## 6.3.1 信号的生命周期

- 信号生命周期: 从信号发送到信号处理函数的执行完毕
- 信号生命周期可以分为三个重要的阶段, 这三个阶段由四个重要事件来刻画, 相邻两个事件的时间间隔构成信号生命周期的一个阶段
  - 信号诞生
  - 信号在进程中注册完毕
  - 信号在进程中的注销完毕
  - 信号处理函数执行完毕



- 执行信号的处理动作称为信号递达 (Delivery)
- 信号从产生到递达之间的状态, 称为信号未决 (Pending)
- 进程可以选择阻塞 (Block) 某个信号。被阻塞的信号产生时将保持在未决状态, 直到进程解除对此信号的阻塞, 才执行递达的动作。
- **注意:** 阻塞和忽略是不同的, 只要信号被阻塞就不会递达, 而忽略是在递达之后可选的一种处理动作。

## 6.3.1 信号的生命周期

### ■ 信号的诞生

- 信号的诞生指的是触发信号的事件发生
- 如检测到硬件异常、定时器超时以及调用信号发送函数`kill()`或`sigqueue()`等

### ■ 信号在目标进程中“注册”

- 信号在进程中注册指的就是信号值加入到进程的未决信号集中（`sigpending`结构的第二个成员`sigset_t signal`），并且信号所携带的信息被保留到未决信号信息链的某个`sigqueue`结构中。
- 只要信号在进程的未决信号集中，表明进程已经知道这些信号的存在，但还没来得及处理，或者该信号被进程阻塞。

### ■ 信号在进程中的注销

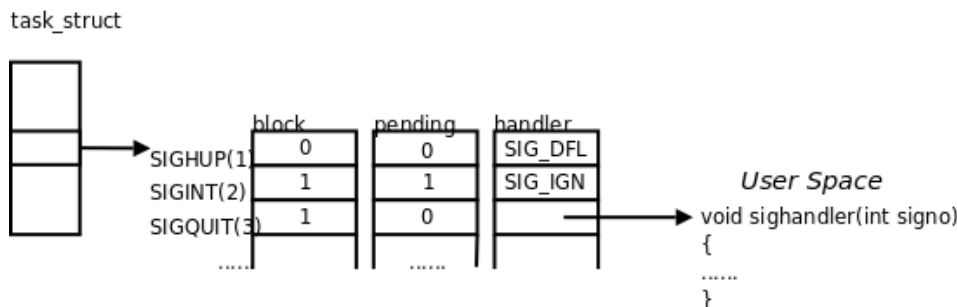
- 在目标进程执行过程中，会检测是否有信号等待处理
  - 每次从系统空间返回到用户空间时都做这样的检查
- 如果存在未决信号等待处理且该信号没有被进程阻塞，则在运行相应的信号处理函数前，进程会把信号在未决信号链中占有的结构卸掉。
  - 对于非实时信号，由于在未决信号信息链中最多只占用一个`sigqueue`结构，因此该结构被释放后，应该把信号在进程未决信号集中删除（信号注销完毕）
  - 对于实时信号，可能在未决信号信息链中占用多个`sigqueue`结构，因此应该针对占用`sigqueue`结构的数目区别对待：如果只占用一个`sigqueue`结构（进程只收到该信号一次），则应该把信号在进程的未决信号集中删除（信号注销完毕）。否则，不应该在进程的未决信号集中删除该信号（信号注销完毕）。
- 进程在执行信号相应处理函数之前，首先要把信号在进程中注销。

### ■ 信号生命终止

- 进程注销信号后，立即执行相应的信号处理函数，执行完毕后，信号的本次发送对进程的影响彻底结束。

## 6.3.2 信号在内核中的表示

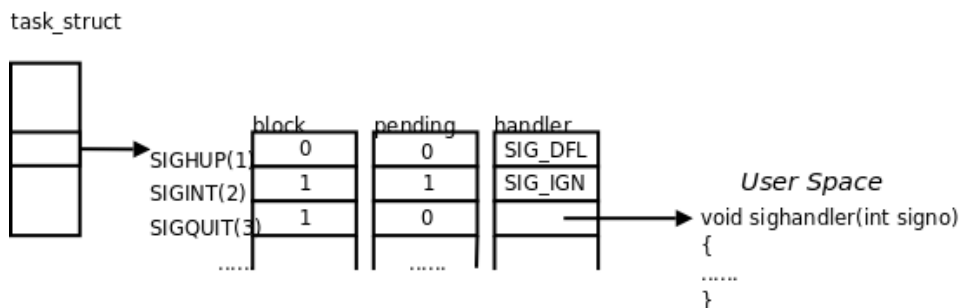
- 每个信号都有两个标志位分别表示阻塞和未决，还有一个函数指针表示处理动作。信号产生时，内核在进程控制块中设置该信号的未决标志，直到信号递达才清除该标志。
- 常规信号(非实时信号)在递达之前产生多次只计一次，而实时信号在递达之前产生多次可以依次放在一个队列里



信号在内核中的表示示意图

- 在上图的例子中，
  - SIGHUP信号未阻塞也未产生过，当它递达时执行默认处理动作
  - SIGINT信号产生过，但正在被阻塞，所以暂时不能递达。虽然它的处理动作是忽略，但在没有解除阻塞之前不能忽略这个信号，因为进程仍有机会改变处理动作之后再解除阻塞
  - SIGQUIT信号未产生过，一旦产生SIGQUIT信号将被阻塞，它的处理动作是用户自定义函数 sighandler

## 6.3.2 信号在内核中的表示



- 每个信号只有一个bit的未决标志，非0即1，不记录该信号产生了多少次
- 阻塞标志也只有一个bit的阻塞标志，非0即1
- 未决和阻塞标志可以用相同的数据类型`sigset_t`来存储，`sigset_t`称为信号集，这个类型可以表示每个信号的“有效”或“无效”状态
  - 在阻塞信号集中“有效”和“无效”的含义是该信号是否被阻塞
  - 在未决信号集中“有效”和“无效”的含义是该信号是否处于未决状态
- 阻塞信号集也叫做当前进程的信号屏蔽字（Signal Mask），这里的“屏蔽”应该理解为阻塞而不是忽略。

## 6.3.3 信号集

- *sigprocmask()*函数中使用信号集(signal set)的数据类型, 定义为*sigset\_t*
- *sigset\_t*类型对于每种信号用一个bit表示“有效”或“无效”状态, 至于这个类型内部如何存储这些bit则依赖于系统实现, 从使用者的角度是不必关心的, 使用者只能调用以下函数来操作*sigset\_t*变量, 而不应该对它的内部数据做任何解释, 比如用*printf()*直接打印*sigset\_t*变量是没有意义的。
  - *select()*用到的*fdset*也是同样的类型
- 信号集只能使用如下的函数进行处理
  - *int sigemptyset(sigset\_t \*set);*
  - *int sigfillset(sigset\_t \*set);*
  - *int sigaddset(sigset\_t \*set, int signo);*
  - *int sigdelset(sigset\_t \*set, int signo);*
  - *int sigismember(const sigset\_t \*set, int signo);*

## 6.3.3 信号集操作函数

- 函数`sigemptyset()`初始化由`set`指向的信号集，使排除其中所有信号
- 函数`sigfillset()`初始化由`set`指向的信号集，使其包括所有信号
- 所有应用程序在使用信号集前，要对该信号集调用`sigemptyset()`或`sigfillset()`一次
- 一旦已经初始化了一个信号集，以后就可在该信号集中增、删特定的信号
- 函数`sigaddset()`将一个信号添加到现存集中
- 函数`sigdelset()`则从信号集中删除一个信号
- 对所有以信号集作为参数的函数，都向其传送信号集地址
- `sigismember()`用于测试某个信号是否在给定的信号集中

```
#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signum);

int sigdelset(sigset_t *set, int signum);

int sigismember(const sigset_t *set, int signum);
```

## 6.3.4 sigprocmask()

- 一个进程的信号屏蔽字规定了当前阻塞而不能递送给该进程的信号集, 调用函数 `sigprocmask()` 可以检测或更改(或两者)进程的信号屏蔽字

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

int sigpending(sigset_t *set);

int sigsuspend(const sigset_t *mask);
```

- `oset` 是非空指针, 进程的当前信号屏蔽字通过 `oset` 返回
- 若 `set` 是一个非空指针, 则参数 `how` 指示如何修改当前信号屏蔽字
  - `SIG_BLOCK` 该进程新的信号屏蔽字是其当前信号屏蔽字和 `set` 指向信号集的并集, `set` 包含了希望阻塞的附加信号
  - `SIG_UNBLOCK` 该进程新的信号屏蔽字是其当前信号屏蔽字和 `set` 所指向信号集的交集, `set` 包含了希望解除阻塞的信号
  - `SIG_SETMASK` 该进程新的信号屏蔽是 `set` 指向的值
- 如果 `set` 是个空指针, 则不改变该进程的信号屏蔽字, `how` 的值也无意义



## 6.3.4 *sigprocmask()*的how参数

The `sigprocmask` call is used to change the list of currently blocked signals. The behaviour of the call is dependent on the value of `how`, as follows.

### `SIG_BLOCK`

The set of blocked signals is the union of the current set and the `set` argument.

### `SIG_UNBLOCK`

The signals in `set` are removed from the current set of blocked signals. It is legal to attempt to unblock a signal which is not blocked.

### `SIG_SETMASK`

The set of blocked signals is set to the argument `set`.

If `oldset` is non-null, the previous value of the signal mask is stored in `oldset`.

- `SIG_BLOCK` - 该进程新的信号屏蔽字是其当前信号屏蔽字和`set`所指向信号集的并集，`set`包含了我们希望阻塞的附加信号
- `SIG_UNBLOCK` - 该进程新的信号屏蔽字是其当前信号屏蔽字和`set`所指向信号集的差集(交集的补集)，`set`包含了我们希望解除阻塞的信号
- `SIG_SET` - 该进程新的信号屏蔽字是`set`所指向的值

## 6.3.5 *sigpending()*

- *sigpending()*读取当前进程的未决信号集，其结果通过*set*参数传出

### NAME

sigaction, sigprocmask, sigpending, sigsuspend - POSIX signal handling functions

### SYNOPSIS

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

```
int sigaction(int signal, const struct sigaction *act, struct sigaction *oldact);
```

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

```
int sigpending(sigset_t *set);
```

```
int sigsuspend(const sigset_t *mask);
```

### DESCRIPTION

The **sigaction** system call is used to change the action taken by a process on receipt of a specific signal.

signal specifies the signal and can be any valid signal except **SIGKILL** and **SIGSTOP**.

If act is non-null, the new action for signal signal is installed from act. If oldact is non-null, the previous action is saved in oldact.

The **sigaction** structure is defined as something like

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

## 6.3.6 实验

- 源代码见右图
- 程序运行时，每秒钟把各信号的未决状态打印一遍，由于我们阻塞了SIGINT 信号，按Ctrl-C将会使SIGINT 信号处于未决状态，按Ctrl-\仍然可以终止程序，因为SIGQUIT 信号没有阻塞。

```
1 #include <signal.h>
2 #include <stdio.h>
3
4 void printsigset(const sigset_t * set)
5 {
6     int i;
7
8     for (i = 1; i < 32; i++)
9         if (sigismember(set, i) == 1)
10             putchar('1');
11         else
12             putchar('0');
13
14     puts("");
15 }
16
17 int main(int argc, char **argv)
18 {
19     sigset_t s, p;
20
21     sigemptyset(&s);
22     sigaddset(&s, SIGINT);
23     sigprocmask(SIG_BLOCK, &s, NULL);
24
25     while (1)
26     {
27         sigpending(&p);
28         printsigset(&p);
29         sleep(1);
30     }
31
32     return 0;
33 }
```

## 6.4. 捕获信号

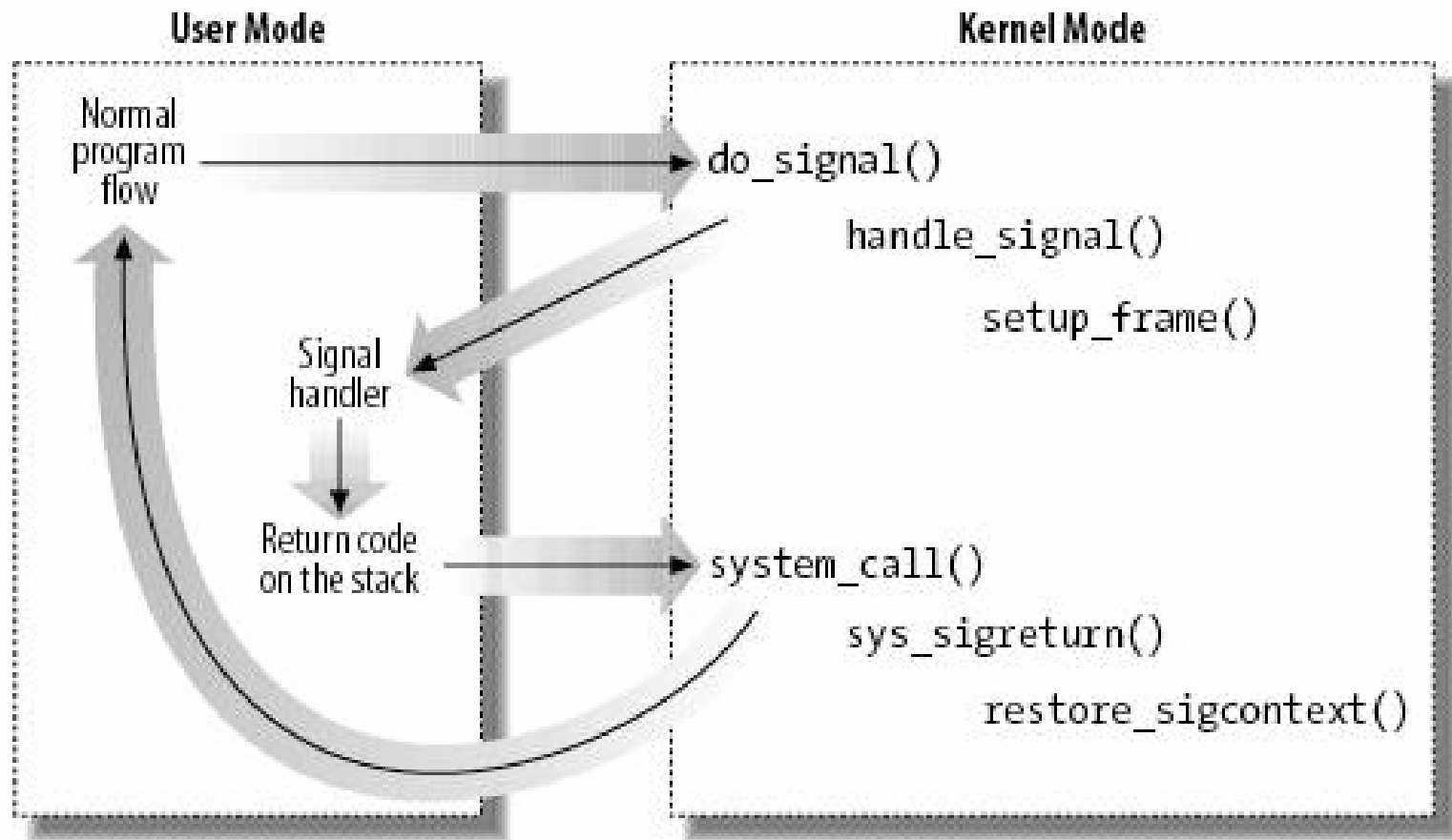
---

- 内核如何实现信号的捕捉
- *signal()*
- *sigaction()*

## 6.4.1 内核如何实现信号的捕捉

- 如果信号的处理动作是用户自定义函数，在信号递达时就调用这个函数，这称为**捕捉信号(catch signal)**
- 信号处理函数的代码是在用户空间的，处理过程比较复杂，过程举例如下：
  - ① 用户程序注册了SIGQUIT信号的处理函数sighandler
  - ② 当前正在执行main()函数，这时发生中断或异常切换到内核态
  - ③ 在中断处理完毕后要返回用户态的main()函数之前检查到有信号SIGQUIT递达
  - ④ 内核决定返回用户态后不是恢复main()函数的上下文继续执行，而是执行sighandler函数，sighandler和main()函数使用不同的堆栈空间，它们之间不存在调用和被调用的关系，是两个独立的控制流程
  - ⑤ sighandler函数返回后自动执行特殊的系统调用sigreturn()再次进入内核态。
  - ⑥ 如果没有新的信号要递达，这次再返回用户态就是恢复main()函数的上下文继续执行了

## 6.4.1 内核如何实现信号的捕捉



## 6.4.2 *signal()*

- *signal()*用于为调用进程设置捕获某个信号时的用户处理函数

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

- 参数

- *sig*是信号名, 其值为SIGXXX
- *func*的取值
  - 常数SIG\_IGN, 忽略此信号
    - 信号SIGKILL和SIGSTOP不能忽略
  - 常数SIG\_DFL, 接到此信号后的动作是系统默认动作
  - 当捕获此信号后要调用的函数(signal handler)的地址

- 成功则返回以前的为信号处理函数地址, 若出错则返回SIG\_ERR

## 6.4.3 *sigaction()*

- *sigaction()*函数可以读取和修改与指定信号相关联的处理动作

### NAME

*sigaction*, *sigprocmask*, *sigpending*, *sigsuspend* - POSIX signal handling functions

### SYNOPSIS

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

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

```
int sigpending(sigset_t *set);
```

```
int sigsuspend(const sigset_t *mask);
```

- 参数
  - *signo*是指定信号的编号
  - 若*act*指针非空，则根据*act*修改该信号的处理动作
  - 若*oact*指针非空，则通过*oact*传出该信号原来的处理动作
- 返回值
  - 调用成功则返回0，出错则返回-1



## 6.4.3.1 *struct sigaction*

- *struct sigaction* 定义(/usr/include/asm/signal.h)

```
89 #define SIG_BLOCK      0    /* for blocking signals */
90 #define SIG_UNBLOCK    1    /* for unblocking signals */
91 #define SIG_SETMASK    2    /* for setting the signal mask */
92
93 /* Type of a signal handler. */
94 typedef void (*__sighandler_t)(int);
95
96 #define SIG_DFL ((__sighandler_t)0)    /* default signal handling */
97 #define SIG_IGN ((__sighandler_t)1)    /* ignore signal */
98 #define SIG_ERR ((__sighandler_t)-1)    /* error return from signal */
99
100 /* Here we must cater to libcs that poke about in kernel headers. */
101
102 struct sigaction {
103     union {
104         __sighandler_t _sa_handler;
105         void (*_sa_sigaction)(int, struct siginfo *, void *);
106     } _u;
107     sigset_t sa_mask;
108     unsigned long sa_flags;
109     void (*sa_restorer)(void);
110 };
111
112 #define sa_handler      _u._sa_handler
113 #define sa_sigaction    _u._sa_sigaction
```

## 6.4.3.2 *sigaction()* - 参数

- 将`sa_handler`赋值为常数`SIG_IGN`传给`sigaction()`表示忽略信号，赋值为常数`SIG_DFL`表示执行系统默认动作，赋值为一个函数指针表示用自定义函数捕捉信号，或者说向内核注册了一个信号处理函数，该函数返回值为`void`，可以带一个`int`参数，通过参数可以得知当前信号的编号，这样就可以用同一个函数处理多种信号。显然，这也是一个回调函数，不是被`main()`函数调用，而是被系统所调用。
- 当某个信号的处理函数被调用时，内核自动将当前信号加入进程的信号屏蔽字，当信号处理函数返回时自动恢复原来的信号屏蔽字，这样就保证了在处理某个信号时，如果这种信号再次产生，那么它会被阻塞到当前处理结束为止。
- 如果在调用信号处理函数时，除了当前信号被自动屏蔽之外，还希望自动屏蔽另外一些信号，则用`sa_mask`字段说明这些需要额外屏蔽的信号，当信号处理函数返回时自动恢复原来的信号屏蔽字。
- `sa_flags`字段包含一些选项，通常设为0
- `sa_sigaction`是实时信号的处理函数，一般不使用

## 6.4.3.3 *pause()*

- *pause()*函数使调用进程挂起直至捕捉到一个信号

```
#include <unistd.h>

int pause(void);
```

- 只有执行了一个信号处理程序并从其返回时，*pause()*才返回。在这种情况下，*pause()*返回-1，*errno*设置为EINTR。
- 组合使用*alarm()*和*pause()*，可以实现使进程自己睡眠一段指定的时间。

## 6.5 其他的问题

---

- 信号与进程
- 中断的系统调用
- 可重入问题
- 竞态条件与sigsuspend()函数
- SIGCHLD信号
- SIGPIPE信号

## 6.5.1 信号与进程

- 当启动一个程序时，所有信号的状态都是系统默认或忽略
  - 通常所有信号都被设置为系统默认动作，除非调用`exec`的进程忽略该信号
  - `exec`函数将原先设置为要捕捉的信号都更改为默认动作，其他信号的状态则不变
    - 一个进程原先要捕捉的信号，当其执行一个新程序后，就自然地不能再捕捉了，因为信号捕捉函数的地址很可能在所执行的新程序文件中已无意义
- 当一个进程调用`fork()`时，其子进程继承父进程的信号处理方式
  - 因为子进程在开始时复制了父进程存储图像，所以信号捕捉函数的地址在子进程中是有意义的。

## 6.5.2 中断的系统调用

- 如果在进程执行一个低速系统调用而阻塞期间捕捉到一个信号，则该系统调用就被中断不再继续执行，该系统调用返回出错，其`errno`设置为EINTR
  - 必须区分系统调用和函数，当捕捉到某个信号时，被中断的是内核中执行的系统调用
- 系统调用分成两类：
  - 低速系统调用-可能会使进程永远阻塞的一类系统调用
    - 在读某些类型的文件时，如果数据并不存在则可能会使调用者永远阻塞(管道、终端设备以及网络设备)
    - 在写某些类型的文件时，如果不能立即接受这些数据，则也可能会使调用者永远阻塞(管道、终端设备以及网络设备)
    - 打开文件，在某种条件发生之前也可能会使调用者阻塞
    - `pause()`(按照定义，它使调用进程睡眠直至捕捉到一个信号)和`wait()`
    - 某种`ioctl()`操作
    - 某些进程间通信函数
  - 其他系统调用

## 6.5.2 中断的系统调用-处理方式

- 被中断的系统调用相关的问题是必须用显式方法处理出错返回
- 典型的代码序列(假定进行一个读操作，它被中断，我们希望重新启动它)可能如下列样式：

```
again:
    if ((n = read(fd, buff, BUFSIZE)) < 0)
    {
        if (errno == EINTR)
            goto again;                /* just an interrupted system call */

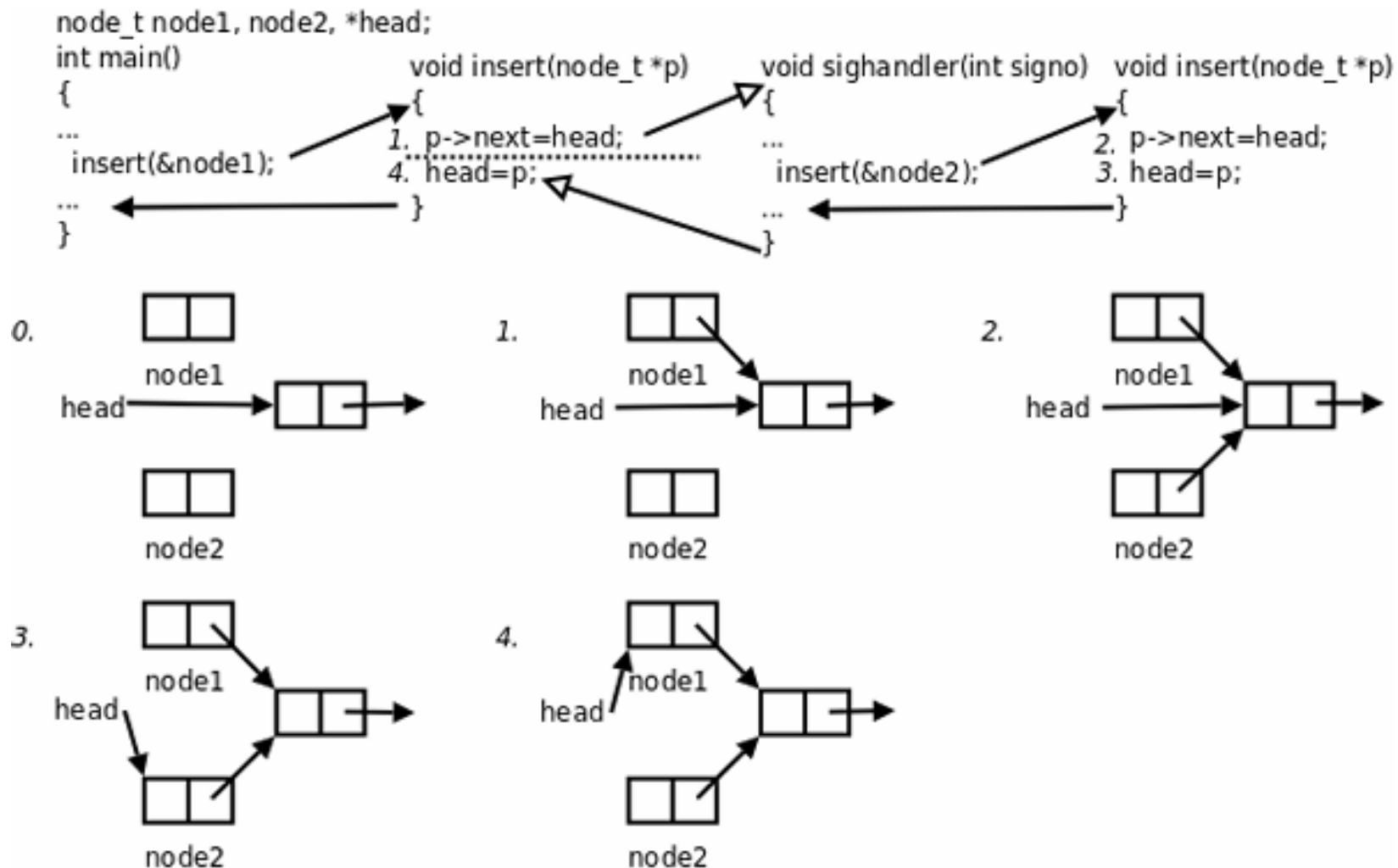
        /* handle other errors */
    }
```

## 6.5.3 可重入问题

- 当捕捉到信号时，不论进程的主控制流程当前执行到哪儿，都会先跳到信号处理函数中执行，从信号处理函数返回后再继续执行主控制流程
- 引入了信号处理函数使得一个进程具有多个控制流程，如果这些控制流程访问相同的全局资源（全局变量、硬件资源等），就有可能出现冲突，导致不可预知的结果
- 信号处理函数是一个单独的控制流程
  - 信号处理函数和主控制流程是异步的
  - 信号处理函数与主控制流程之间不存在调用和被调用的关系
  - 信号处理函数与主控制流程使用不同的堆栈空间
- 可能存在的一些问题
  - 如果进程正在执行`malloc()`，而与此同时信号处理函数中也做`malloc()`，会造成什么情况？
  - 如果进程正在使用某个全局变量，而此时信号处理函数修改了该变量的值，会造成什么情况？
  - 如果信号处理函数中调用了某个函数，而这个函数又重设了`errno`，会造成什么情况？
- 可重入问题可能存在的场景
  - 信号处理
  - 中断处理
  - 多线程



## 6.5.3.1 可重入函数 — 一个例子



## 6.5.3.1 可重入函数 – 分析

- *main()*函数调用*insert()*函数向一个链表*head*中插入节点*node1*，插入操作分为两步，刚做完第一步的时候，因为硬件中断使进程切换到内核，再次回用户态之前检查到有信号待处理，于是切换到*sighandler*函数，*sighandler*也调用*insert()*函数向同一个链表*head*中插入节点*node2*，插入操作的两步都做完之后从*sighandler*返回内核态，再次回到用户态就从*main()*函数调用的*insert()*函数中继续往下执行，先前做第一步之后被打断，现在继续做完第二步。结果是，*main()*函数和*sighandler*先后向链表中插入两个节点，而最后只有一个节点真正插入链表中了。
- 分析：*insert()*函数被不同的控制流程调用，有可能在第一次调用还没返回时就再次进入该函数，这称为**重入(reentrant)**，*insert()*函数访问一个全局链表，有可能因为重入而造成错乱，像这样的函数称为**不可重入函数**，反之，如果一个函数只访问自己的局部变量或参数，则称为**可重入函数**。
  - 问题：为什么两个不同的控制流程调用同一个函数，访问它的同一个局部变量或参数就不会造成错乱？

## 6.5.3.2 Posix和SUS对可重入的要求

- Posix和SUS规定有些系统函数必须以线程安全的方式实现
- 以下是Posix.1要求的可重入函数

---

_exit	fork	pipe	stat
abort*	fstat	read	sysconf
access	getegid	rename	tcdrain
alarm	geteuid	rmdir	tcflow
cfgetispeed	getgid	setgid	tcflush
cfgetospeed	getgroups	setpgid	tcgetattr
cfsetispeed	getpgrp	setsid	tcgetpgrp
cfsetospeed	getpid	setuid	tcsendbreak
chdir	getppid	sigaction	tcsetattr
chmod	getuid	sigaddset	tcsetpgrp
chown	kill	sigdelset	time
close	link	sigemptyset	times
creat	longjmp*	sigfillset	umask
dup	lseek	sigismember	uname
dup2	mkdir	signal*	unlink
execle	mkfifo	sigpending	utime
execve	open	sigprocmask	wait
exit*	pathconf	sigsuspend	waitpid
fcntl	pause	sleep	write

---

## 6.5.3.3 其他函数

- 没有列入该表的大多数函数是不可再入的，其原因为：
  - 已知它们使用静态数据结构
  - 已知它们使用全局变量(如`errno`)
  - 它们调用`malloc()`或`free()`
  - 它们是标准I/O函数, 标准I/O库的很多实现都以不可再入方式使用全局数据结构
- 有些不可重入函数提供可重入的版本，通常其命名为 *function\_r()*
  - `gethostbyname()` – `gethostbyname_r()`

## 6.5.3.4 可重入问题的建议解决办法

- 可重入问题的根本是原子性的问题
- 以下办法可供参考
  - 谨慎使用静态分配的数据以及全局变量
  - 在信号处理函数的开始部分保存`errno`，在信号处理函数的结束部分恢复`errno`
  - 使用`sig_atomic_t`类型与`volatile`限定符，`sig_atomic_t`类型的变量应该总是加上`volatile`限定符
    - C标准定义了一个类型`sig_atomic_t`，在不同平台的C语言库中取不同的类型，例如在32位机上定义`sig_atomic_t`为`int`类型
    - C语言提供了`volatile`限定符，以保证`sig_atomic_t`类型的变量在读写时的原子性(可能的优化问题)
  - 在信号处理函数中，不做具体的操作，仅仅修改一个标记，在程序的主流程(通常为一个循环)中检查标记是否置位，如果置位则进行处理，该标记通常定义为`volatile sig_atomic_t`类型
  - 在多线程的情况下，可以使用互斥机制来解决可重入的问题，即线程间同步
    - Mutex
    - Condition variable
    - Semaphore

## 6.5.4 竞态条件与*sigsuspend()*函数

- 由于异步事件在任何时候都有可能发生（这里的异步事件指出现更高优先级的进程），如果我们写程序时考虑不周密，就可能由于时序问题而导致错误，这叫做**竞态条件(race condition)**或者**竞争**
- 产生静态条件的一种可能情况：
  - 注册 **SIGALRM**信号的处理函数
  - 调用`alarm(nsecs)`设定闹钟
  - 内核调度优先级更高的进程取代当前进程执行，并且优先级更高的进程有很多个，每个都要执行很长时间
  - `nsecs`秒钟之后闹钟超时了，内核发送**SIGALRM**信号给这个进程，处于未决状态
  - 优先级更高的进程执行完了，内核要调度回这个进程执行**SIGALRM** 信号递达，执行处理函数`sig_alm`之后再次进入内核
  - 返回这个进程的主控制流程，`alarm(nsecs)`返回，调用`pause()`挂起等待。
  - 可是 **SIGALRM**信号已经处理完了，还等待什么呢？
- 对上述静态条件问题的分析
  - 出现竞态条件问题的根本原因是系统运行的时序（**Timing**）并不像我们写程序时所设想的那样。虽然`alarm(nsecs)`紧接着的下一行就是`pause()`，但是无法保证`pause()`一定在调用`alarm(nsecs)`之后的`nsecs`秒之内被调用

## 6.5.4 *sigsuspend()*

- *sigsuspend()*函数的功能: 将“解除信号屏蔽”和“挂起等待信号”这两步并成一个原子操作
- *sigsuspend()*包含了*pause()*的挂起等待功能, 同时解决了竞态条件的问题
- 在对时序要求严格的场合下都应该调用*sigsuspend()*而不是*pause()*

### NAME

*sigaction*, *sigprocmask*, *sigpending*, *sigsuspend* - POSIX signal handling functions

### SYNOPSIS

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

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

```
int sigpending(sigset_t *set);
```

```
int sigsuspend(const sigset_t *mask);
```

## 6.5.5 SIGCHLD信号

### ■ `wait()`和`waitpid()`函数清理僵尸进程

#### ➤ 常规的方法

- 父进程阻塞等待子进程结束，父进程阻塞了就不能处理自己的工作
- 父进程非阻塞地查询是否有子进程结束等待清理（也就是轮询的方式），父进程在处理自己的工作的同时还要记得时不时地轮询一下，程序实现复杂

#### ➤ 通过SIGCHLD信号来处理`wait()`

- 子进程在终止时会给父进程发SIGCHLD信号，该信号的默认处理动作是忽略，父进程可以自定义SIGCHLD信号的处理函数，这样父进程只需专心处理自己的工作，不必关心子进程了，子进程终止时会通知父进程，父进程在信号处理函数中调用`wait()`清理子进程即可。

#### ➤ 由于UNIX的历史原因，要想不产生僵尸进程还有另外一种办法：

- 父进程调用`sigaction()`将SIGCHLD的处理动作置为SIG\_IGN，这样`fork()`出来的子进程在终止时会自动清理掉，不会产生僵尸进程，也不会通知父进程。系统默认的忽略动作和用户用`sigaction()`函数自定义的忽略通常是没有区别的，但这是一个特例。
- 此方法对于Linux可用，但不保证在其它UNIX系统上都可用，所以不推荐使用。



## 7. 进程间通讯

---

- IPC概述
- 管道
- 有名管道(FIFO)
- Posix消息队列
- Posix信号灯

## 7.1.1 IPC概述

---

- 进程间通信(IPC – Interprocess communication)
- 传统上(大多数情况下), IPC指的是运行在某个操作系统之上的不同进程间消息传递(Message passing)的方式, 如管道、消息队列、共享内存等。
- 在有些时候, IPC也包括在不同主机进程之间的消息传递, 如socket、Sun RPC等。

## 7.1.2 IPC的类型

---

- 消息传递
  - 管道
  - FIFO
  - 消息队列
- 同步
  - 互斥锁
  - 条件变量
  - 读写锁
  - 文件与记录锁
  - 信号灯
- 共享内存区
  - 匿名共享内存区
  - 有名共享内存区
- 远程过程调用
  - Solaris门
  - SUN RPC



## 7.1.3 IPC对象的持续性

### ■ 随进程持续的IPC

- 一直存在到打开着该对象的最后一个进程关闭该对象为止
  - 管道
  - FIFO

### ■ 随内核持续的IPC

- 一直存在到内核重新自举或显式删除该对象为止
  - System V消息队列
  - 信号灯
  - 共享内存区

### ■ 随文件系统持续的IPC

- 一直存在到显式删除该对象为止，即使内核重新自举了，该对象还保持其值
  - Posix消息队列
  - Posix信号灯
  - Posix共享内存区

## 7.1.4 Posix IPC

---

- Posix消息队列
- Posix信号灯
- Posix共享内存区

## 7.1.5 System V IPC

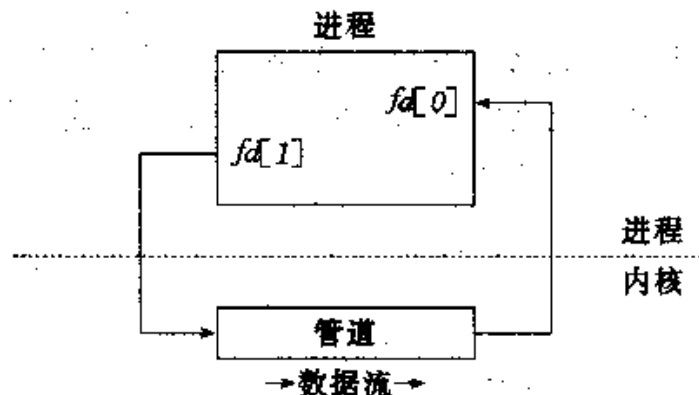
---

- System V消息队列
- System V信号灯
- System V共享内存区

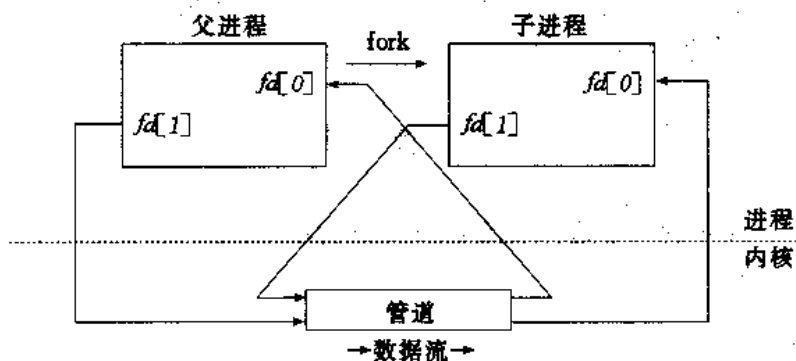
## 7.2 管道

- 管道是Linux支持的最初UNIX IPC形式之一
- 所有的UNIX都支持管道
- 管道是半双工的，数据只能向一个方向传输，需要全双工通信时，需要建立起两个管道，或者使用`socketpair()`创建全双工管道。
- 管道由单个进程创建，但是通常很少在一个进程内部使用，管道通常用于用于具有亲缘关系的进程(父子进程或者兄弟进程之间)间通讯。
- 宏`S_ISFIFO()`可以用来确定一个文件描述符或文件是否为管道或者FIFO
- 数据的读出和写入：一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。
- 管道和FIFO都不支持诸如`lseek()`等文件定位操作。

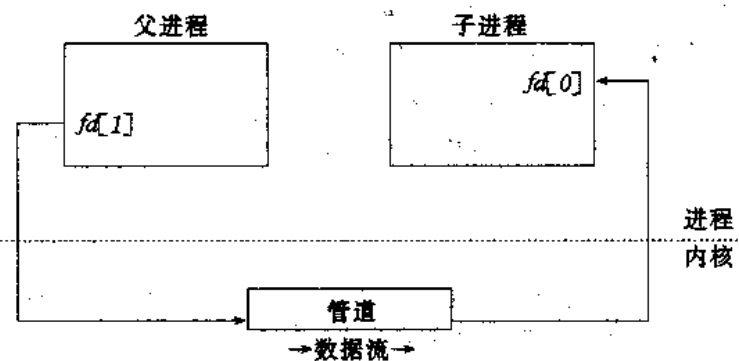
## 7.2.1 管道



单个进程中的管道



单个进程中的管道，fork()后



两个进程间的管道



## 7.2.2 创建管道

### ■ 管道创建方式:

```
#include <unistd.h>

int pipe(int filedes[2]);
```

- 管道两端可分别用描述字fd[0]以及fd[1]来描述，分别作为管道的读端和写端
  - 管道读端，由描述字fd[0]表示
  - 管道写端，由描述字fd[1]表示
- 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。



## 7.2.3 管道的读写规则

### ■ 从管道中读取数据：

- 如果管道的写端不存在，则认为已经读到了数据的末尾，读函数返回的读出字节数为0；
- 当管道的写端存在时
  - 如果请求的字节数目大于PIPE\_BUF，则返回管道中现有的数据字节数
  - 如果请求的字节数目不大于PIPE\_BUF，则返回管道中现有数据字节数（此时，管道中数据量小于请求的数据量）；或者返回请求的字节数（此时，管道中数据量不小于请求的数据量）。

### ■ 向管道中写入数据：

- 向管道中写入数据时，Linux将不保证写入的原子性，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读走管道缓冲区中的数据，那么写操作将一直阻塞。
- 只有在管道的读端存在时，向管道中写入数据才有意义。否则，向管道中写入数据的进程将收到内核传来的SIGPIPE信号，应用程序可以处理该信号，也可以忽略（默认动作则是应用程序终止）

## 7.2.4 管道应用

---

### ■ 管道常用于两个方面：

- 在shell中时常会用到管道（作为输入输出的重定向），在这种应用方式下，管道的创建对于用户来说是透明的；
- 用于具有亲缘关系的进程间通信，用户自己创建管道，并完成读写操作。

## 7.2.5 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（管道只存在于内存中，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(即制定通讯协议)，比如多少字节算作一个消息（或命令、或记录）等等

## 7.3. 有名管道(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
- 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
- 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。
- FIFO严格遵循先进先出（first in first out），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。
- 管道和FIFO都不支持诸如lseek()等文件定位操作。

## 7.3.2 FIFO创建

### ■ FIFO创建方式:

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

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

### ■ 参数说明:

- 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。

### ■ 可能错误:

- 如果mkfifo的第一个参数是一个已经存在的路径名时，会返回EEXIST错误，所以一般典型的调用代码首先会检查是否返回该错误，如果确实返回该错误，那么只要调用打开FIFO的函数就可以了。

### ■ 一般文件I/O函数都可以用于FIFO，如close()、read()、write()等等。

### ■ 使用mkfifo()创建FIFO后，还需要进行open()。

## 7.3.3 FIFO的打开规则

- 如果当前打开操作是为读而打开FIFO时
  - 若已经有相应进程为写而打开该FIFO，则当前打开操作将成功返回；
  - 否则，可能阻塞直到有相应进程为写而打开该FIFO（当前打开操作未设置非阻塞标志）；或者，成功返回（当前打开操作设置非阻塞标志）。
- 如果当前打开操作是为写而打开FIFO时
  - 如果已经有相应进程为读而打开该FIFO，则当前打开操作将成功返回；
  - 否则，可能阻塞直到有相应进程为读而打开该FIFO（当前打开操作未设置非阻塞标志）；或者，返回ENXIO错误（当前打开操作设置了非阻塞标志）。

## 7.3.4 FIFO的读写规则

### ■ 从FIFO中读取数据规则

- 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，则对于未设置非阻塞标志的读操作来说，将一直阻塞。对于设置了非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。
  - 对于未设置非阻塞标志的读操作说，造成阻塞的原因有两种：当前FIFO内有数据，但有其它进程在读这些数据；另外就是FIFO内没有数据。写阻塞的原因则是FIFO中有新的数据写入，不论新写入数据量的大小，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。
- 如果FIFO中有数据，则设置了阻塞标志的读操作不会因为FIFO中的字节数小于请求读的字节数而阻塞，此时，读操作会返回FIFO中现有的数据量。



## 7.3.4 FIFO的读写规则(cont.)

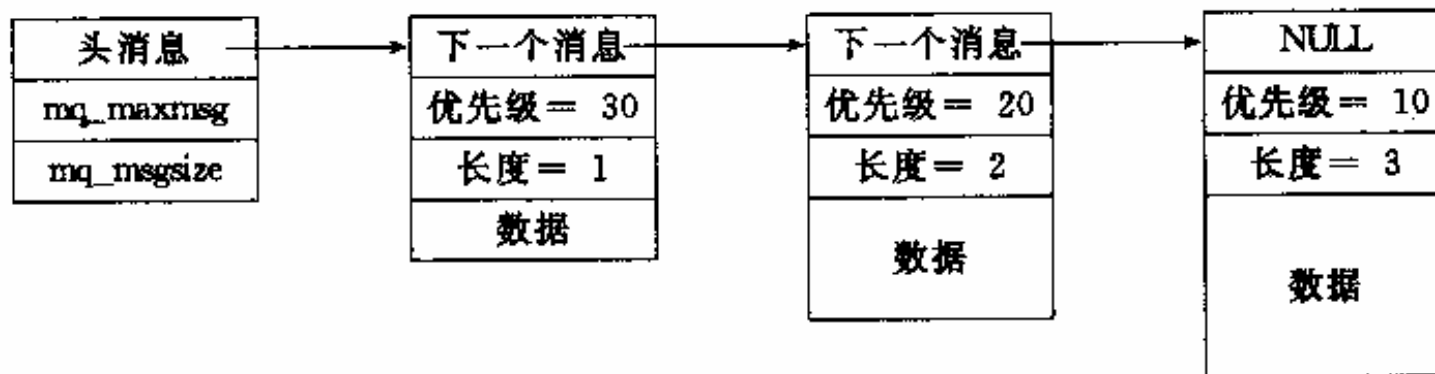
### ■ 向FIFO中写入数据:

- 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;

## 7.4 Posix消息队列

- 消息队列可以认为是一个消息链表，有足够写权限的进程/线程可以向队列中放置(send)消息，有足够读权限的进程/线程可以从队列中取走(receive)消息。
- 每个消息是一个记录，它由发送者赋予一个优先级
- 在某个进程/线程向一个消息队列写入消息之前，并不需要另外某个进程/线程在该消息队列上等待消息的到达
- 消息队列至少具有随内核的持续性，即一个进程可以往某个消息队列写入一些消息后终止，另外一个进程在以后某个时刻可以将消息读出
- Posix消息队列与System V消息队列区别
  - 对Posix消息队列的读总是返回最高优先级的最早消息，对System V消息队列的读则可以返回任意指定优先级的消息
  - 当往一个空队列放置一个消息时，Posix消息队列允许产生一个信号或启动一个线程，System V消息队列则不提供类似机制
- 消息队列中的每个消息具有如下的属性
  - 一个无符号整数优先级(Posix)或一个长整数类型(System V)
  - 消息的数据部分长度(可以为0)
  - 数据本身(如果长度大于0)

## 7.4.1 消息队列的可能布局



含有三个消息的某个Posix消息队列的可能布局

## 7.4.2 mq\_open()

- mq\_open()函数创建一个新的消息队列或者打开一个已经存在的消息队列

```
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag, ...
               /* mode_t mode, struct mq_attr *attr */);
```

返回:成功时为消息队列描述字,出错时为-1

- oflag参数取值
  - O\_RDONLY
  - O\_WRONLY
  - O\_RDWR
- 在创建消息队列时, 需要设定attr
- name是以"/"开头的字符串,并且后边不能再含有"/"(Posix的规定)
- mq\_open()的返回值为消息队列描述字(mqd\_t类型)

## 7.4.3 mq\_close()

- 已打开的消息队列由mq\_close()关闭

```
#include <mqqueue.h>
int mq_close(mqd_t mqdes)
```

返回：成功时为 0，出错时为 -1

- 调用mq\_close()后，调用进程不能再次使用该消息队列，但是消息队列并不从系统中删除
- 一个进程终止时，其所有打开着的消息队列都被自动关闭

## 7.4.4 mq\_unlink()

- 已经创建的消息队列使用mq\_unlink()删除

```
#include <mqueue.h>
int mq_unlink(const char * name);
```

返回：成功时为 0，出错时为 -1

- 每个消息队列都有一个保存其当前打开着的文件描述符的引用计数，只有当引用计数为0时，才真正的删除消息队列
- mq\_unlink()使用的参数为消息队列的名字，而不是消息队列的描述符

## 7.4.5 mq\_getattr()/mq\_setattr()

- 每个消息队列有四个属性，mq\_getattr()取得这些属性，mq\_setattr()设置其中某个属性

```
#include <mqqueue.h>
```

```
int mq_getattr(mqd_t mqdes, struct mq_attr * attr);
```

```
int mq_setattr(mqd_t mqdes, const struct mq_attr * attr, struct mq_attr * oattr);
```

都返回：成功时为 0，出错时为 -1

- struct mq\_attr定义

```
struct mq_attr {  
    long mq_flags;           /* message queue flag: 0, O_NONBLOCK */  
    long mq_maxmsg;         /* max number of messages allowed on queue */  
    long mq_msgsize;        /* max size of a message (in bytes) */  
    long mq_curmsgs;        /* number of messages currently on queue */  
};
```

## 7.4.6 mq\_send()/mq\_receive()

- mq\_send()用于向消息队列中放入一个消息
  - mq\_send()可以为消息设定一个优先级
  - 允许发送0长度的消息
- mq\_receive()用于从消息队列中取出一个消息
  - mq\_receive()总是返回指定队列中最高优先级的最早消息，而且该优先级能随消息的内容及其长度一同返回

```
#include <mqueue.h>
```

```
int mq_send(mqd_t mqdes, const char * ptr, size_t len, unsigned int prio);
```

返回：成功时为 0，出错时为 -1

```
ssize_t mq_receive(mqd_t mqdes, char * ptr, size_t len, unsigned int * priop);
```

返回：成功时为消息中的字节数，出错时为 -1

- Posix消息队列和System V消息队列都不标识消息的发送者



## 7.4.7 消息队列限制

---

- 在创建消息队列时，对创建的消息队列有两个限制
  - mq\_maxmsg
  - mq\_msgsize
- 系统定义了两个限制(在unistd.h中定义，可以通过sysconf获取)
  - MQ\_OPEN\_MAX
    - Posix要求至少为8
    - 在Linux下测试其值为10
  - MQ\_PRIO\_MAX
    - Posix要求该值至少为32

## 7.4.8 mq\_notify()

- Posix消息队列允许异步事件通知(Asynchronous event notification),以通知何时有一个消息放置到了某个空消息队列中,并可以进行相应的处理
  - 产生一个信号
  - 创建一个线程执行一个指定的函数

```
#include <mqueue.h>
```

```
int mq_notify(mqd_t mqdes, const struct sigevent * notification);
```

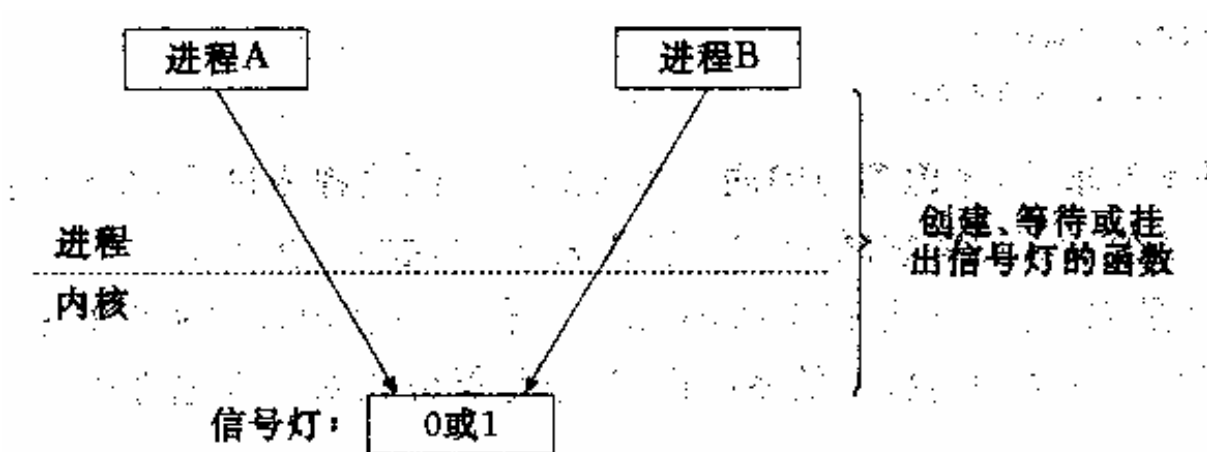
返回:成功时为 0,出错时为 -1

## 7.4.8 mq\_notify() 参数

```
union sigval {  
    int    sival_int;          /* Integer value */  
    void * sival_ptr;         /* pointer value */  
};  
  
struct sigevent {  
    int          sigev_notify; /* SIGEV_ {NONE, SIGNAL, THREAD} */  
    int          sigev_signo;  /* signal number if SIGEV_SIGNAL */  
    /* following two if SIGEV_THREAD */  
    void          (* sigev_notify_function) (union sigval);  
    pthread_attr_t * sigev_notify_attributes;  
};
```

## 7.5 Posix信号灯

- 信号灯(semaphore)是一种用于提供不同进程间或者一个给定进程的不同线程间同步手段的原语
  - Posix有名信号灯，使用Posix IPC名字标识
  - Posix基于内存的信号灯，存放在共享内存中
  - System V信号灯，在内核中维护



## 7.5.1 信号灯的三种操作

---

### ■ 创建(create)

- 需要设定初始值，对于二值信号灯，可以为0或者为1

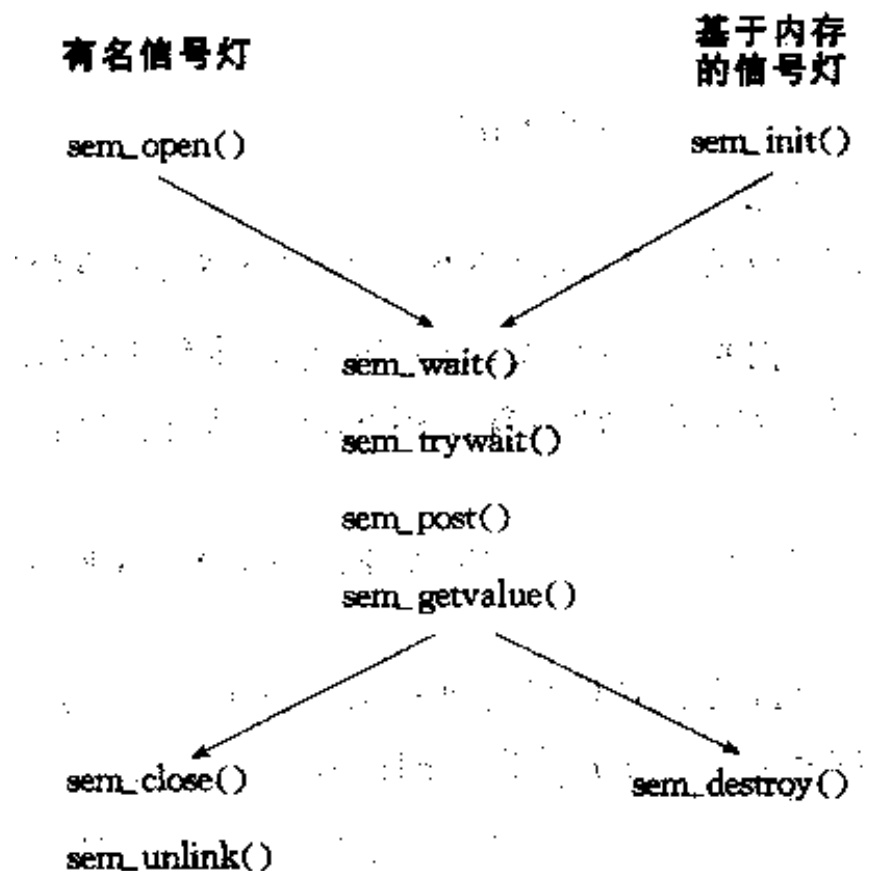
### ■ 等待(wait)

- 测试信号灯的值得
  - 如果信号灯值小于1，则阻塞
  - 如果信号灯值大于0，则将其减1
- 必须为原子操作

### ■ 挂出(post)

- 将信号灯值加1
- 必须为原子操作

## 7.5.2 Posix信号灯的函数调用



## 7.5.3 sem\_open()

- sem\_open()函数创建一个新的有名信号灯或者打开一个已经存在的有名信号灯

```
#include <semaphore.h>
sem_t * sem_open(const char * name, int oflag, ... mode_t mode, unsigned int value * /);
```

返回：成功时为指向信号灯的指针，出错时为 SEM\_FAILED

- oflag参数取值
  - O\_CREAT
  - O\_CREAT | O\_EXCL
- sem\_open()的返回值为sem\_t类型

## 7.5.4 sem\_close()

- 已打开的有名信号灯由sem\_close()关闭

```
#include <semaphore.h>
int sem_close(sem_t * sem);
```

返回:成功时为 0,出错时为-1

- 一个进程终止时,无论自愿终止还是非自愿终止,内核还对其上仍然打开着的所有有名信号灯自动执行sem\_close()操作
- 关闭一个信号灯没有将它从系统中删除,即Posix信号灯至少是随内核持续的



## 7.5.5 sem\_unlink()

- 已经创建的信号灯使用sem\_unlink()删除

```
#include <semaphore.h>
```

```
int sem_unlink(const char * name);
```

返回:成功时为 0,出错时为 -1

- 每个信号灯都有一个保存期当前打开着的文件描述符的引用计数，只有当引用计数为0时，才真正的删除该信号灯
- sem\_unlink()使用的参数为消息队列的名字，而不是消息队列的描述符

## 7.5.6 sem\_post()/sem\_getvalue()

```
#include <semaphore.h>
```

```
int sem_post(sem_t *sem);
```

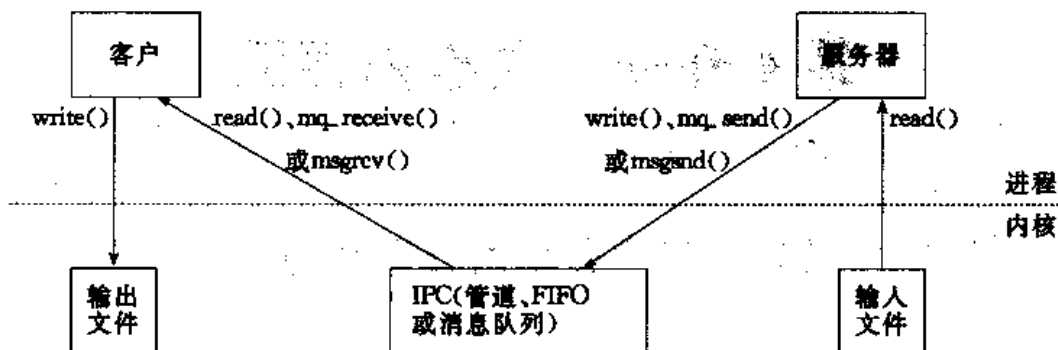
```
int sem_getvalue(sem_t *sem, int *valp);
```

都返回：成功时为 0，出错时为 -1

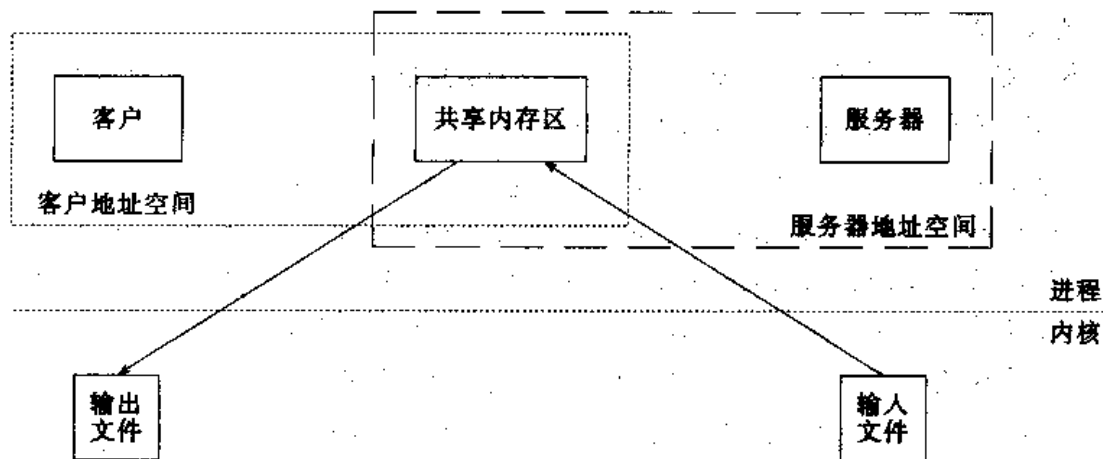
## 7.6 共享内存

- 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
  - 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。
- Linux支持多种共享内存方式
  - mmap()
  - Posix共享内存
  - 系统V共享内存

## 7.6.1 IPC效率比较



通过管道/FIFO/消息队列进行IPC



通过共享内存方式进行IPC

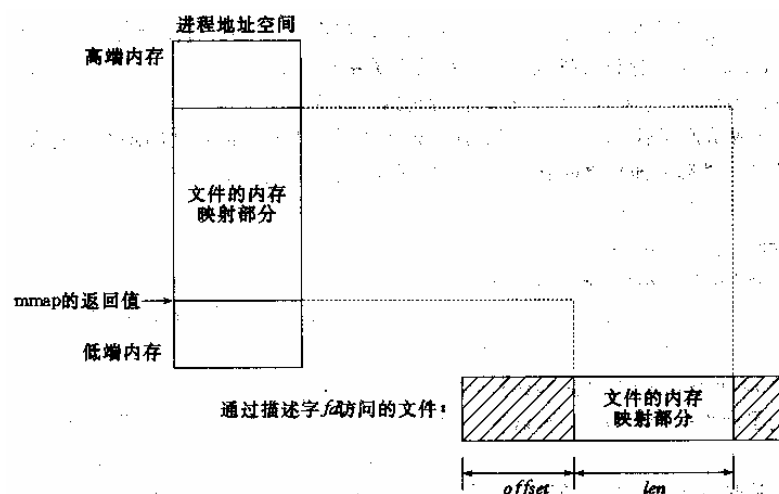
## 7.6.2 mmap()

### ■ mmap()可用于创建共享的文件映射

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

```
void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

返回,成功时为被映射区的起始地址,出错时为 MAP\_FAILED



mmap示意图

<i>prot</i>	说明
PROT_READ	数据可读
PROT_WRITE	数据可写
PROT_EXEC	数据可执行
PROT_NONE	数据不可访问

prot参数

<i>flags</i>	说明
MAP_SHARED	变动是共享的
MAP_PRIVATE	变动是私有的
MAP_FIXED	准确地解释 addr 参数

flags参数

## 7.6.3 munmap()

- munmap()解除mmap()建立的映射

```
#include <sys/mman.h>
int munmap(void *addr, size_t len);
```

返回：成功时为 0，出错时为 -1

## 7.6.4 msync()

### ■ msync()同步文件与映射的内存

```
#include <sys/mman.h>
int msync(void *addr, size_t len, int flags);
```

返回：成功时为 0，出错时为 -1。

Let's DO it!

---

**Thanks for listening!**

