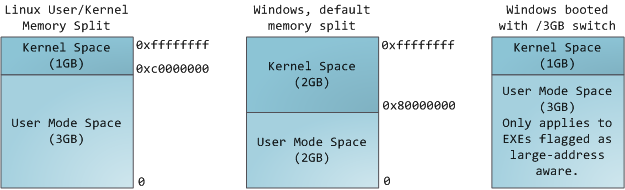
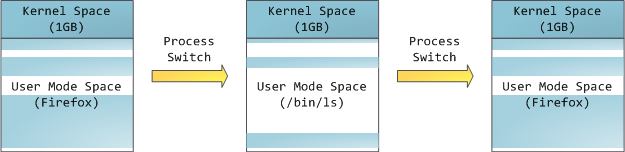
**Linux程序内存分配方式**

**1. Linux程序的内存分布**

在一个多任务OS中，每个进程都运行在它自己的内存沙箱中。这个沙箱就是虚拟地址空间，在32位下就是一块容量为4GB的内存地址。内核将这些虚拟地址按页表（page table）映射为物理内存，并交由CPU访问。每个进程有自己的页表集，但有一点要注意。虚拟地址一旦被启用，就会应用到机器上所有运行的程序上，也包括内核自己。因此虚拟地址空间必须为内核预留一部分（kernel space）（否则就没办法和内核交互了）：



这并不意味着内核使用了那么多的物理内存，仅表示它可支配这么大的地址空间，可根据内核需要，将其映射到物理内存。内核空间在页表中拥有较高的特权级（ring 2或以下），因此只要用户态的程序试图访问这些页，就会导致一个页错误（page fault）。在Linux中，内核空间是持续存在的，并且在所有进程中都映射到同样的物理内存。内核代码和数据总是可寻址的，随时准备处理中断和系统调用。与此相反，用户模式地址空间的映射随进程切换的发生而不断变化：



蓝色区域表示映射到物理内存的虚拟地址，而白色区域表示未映射的部分。在上面的例子中，Firefox使用了相当多的虚拟地址空间，因为它是传说中的吃内存大户。地址空间中的各个条带对应于不同的内存段（memory segment），如：堆、栈之类的。所谓内存段只是简单的内存地址范围，与Intel处理器的段没有关系。不管怎样，下面是一个Linux进程的标准的内存段布局：

当计算机开心、安全、可爱、正常的运转时，那么上图显示的内存段起始虚拟地址在几乎每个进程中都是一样的。这导致了远程利用安全漏洞变得非常容易。一次漏洞探测通常需要引用内存的绝对地址：一个栈上地址，一个库函数的地址，等等。远程攻击者只能盲目的选择这样的地址，指望地址空间都是一样的。如果这种情况真发生了，用户就悲剧了。因此地址空间随机化变的很流行。Linux会给栈、mmap段、和堆的起始地址一个随机偏离。不幸的是，32位地址空间实在是太紧凑了，只给随机化很小的空间，妨碍了它的效果。

进程地址空间中最顶部的段是栈，大多数编程语言将之用于存储局部变量和函数参数。调用一个方法或函数会将一个新的栈桢（stack frame）压入栈中。栈桢在函数返回时被清理。也许是因为严格的遵从LIFO的顺序，这个简单的设计意味着不必使用复杂的数据结构来追踪栈的内容，只需要一个简单的指针指向栈的顶端即可。因此压栈（pushing）和退栈（popping）过程非常迅速、准确。另外，持续的重用栈空间有助于使活跃的栈内存保持在CPU缓存中，从而加速访问。进程中的每一个线程都有属于自己的栈。

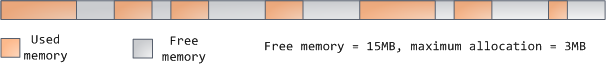
通过不断向栈中压入的数据，超出其容量就会耗尽栈所对应的内存区域。这将触发一个页故障（page fault），并被 Linux 的expand\_stack()处理，这个函数会调用acct\_stack\_growth()来检查是否还有合适的地方用于栈的增长。如果栈的大小低于RLIMIT\_STACK（通常是8MB），那么一般情况下栈会被加长，程序继续愉快的运行，感觉不到发生了什么事情。这是一种将栈扩展至所需大小的常规机制。然而，如果达到了最大的栈空间大小，就会栈溢出（stack overflow），程序收到一个段错误（Segmentation Fault）。当映射了的栈区域扩展到所需的大小后，它就不会再收缩回去，即使栈不那么满了。这就好比联邦预算，它总是在增长的。

动态栈增长是唯一一种访问未映射内存区域（图中白色区域）而被允许的情形。其它任何对未映射内存区域的访问都会触发页故障，从而导致段错误。一些被映射的区域是只读的，因此企图写这些区域也会导致段错误。

在栈的下方，是我们的内存映射段。此处，内核将文件的内容直接映射到内存。任何应用程序都可以通过 Linux 的 mmap() 系统调用（实现）或 Windows 的 CreateFileMapping()/MapViewOfFile()请求这种映射。内存映射是一种方便高效的文件 I/O 方式，所以它被用于加载动态库。创建一个不对应于任何文件的匿名内存映射也是可能的，此方法用于存放程序的数据。在 Linux 中，如果你通过 malloc()请求一大块内存，C 运行库将会创建这样一个匿名映射而不是使用堆内存。‘大块’意味着比MMAP\_THRESHOLD 还大，缺省是 128KB ，可以通过mallopt()调整。

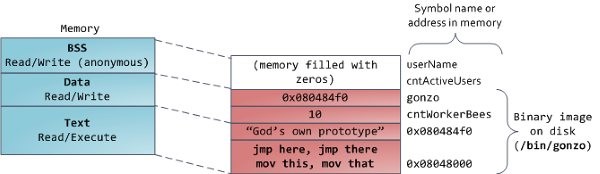
说到堆，它是接下来的一块地址空间。与栈一样，堆用于运行时内存分配；但不同点是，堆用于存储那些生存期与函数调用无关的数据。大部分语言都提供了堆管理功能。因此，满足内存请求就成了语言运行时库及内核共同的任务。在 C 语言中，堆分配的接口是malloc()系列函数，而在具有垃圾收集功能的语言（如 C# ）中，此接口是 new 关键字。

如果堆中有足够的空间来满足内存请求，它就可以被语言运行时库处理而不需要内核参与。否则，堆会被扩大，通过brk()系统调用（实现）来分配请求所需的内存块。堆管理是很复杂的，需要精细的算法，应付我们程序中杂乱的分配模式，优化速度和内存使用效率。处理一个堆请求所需的时间会大幅度的变动。实时系统通过特殊目的分配器来解决这个问题。堆也可能会变得零零碎碎，如下图所示：

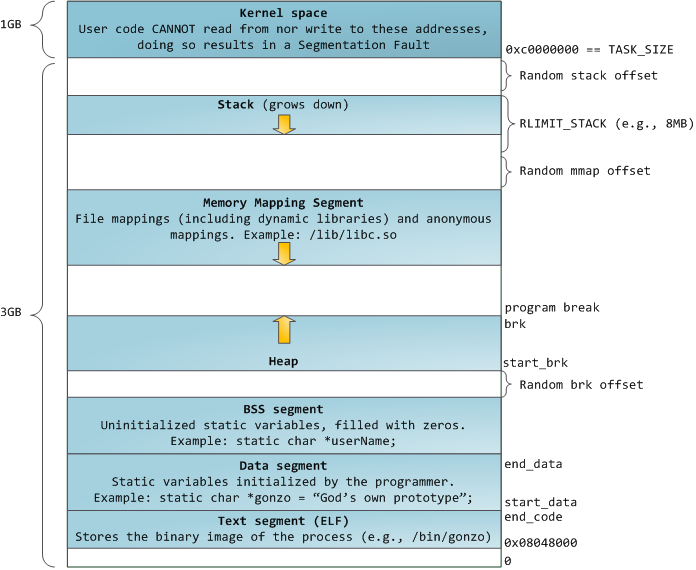


最后，我们来看看最底部的内存段：BSS，数据段，代码段。在C语言中，BSS和数据段保存的都是静态（全局）变量的内容。区别在于BSS保存的是未被初始化的静态变量内容，它们的值不是直接在程序的源代码中设定的。BSS内存区域是匿名的：它不映射到任何文件。如果你写static int cntActiveUsers，则cntActiveUsers的内容就会保存在BSS中。另一方面，data段保存在源代码中已经初始化了的静态变量内容。这个内存区域不是匿名的。它映射了一部分的程序二进制镜像，也就是源代码中指定了初始值的静态变量。所以，如果你写static int cntWorkerBees = 10，则cntWorkerBees的内容就保存在数据段中了，而且初始值为10。尽管data段映射了一个文件，但它是一个私有内存映射，这意味着更改此处的内存不会影响到被映射的文件。也必须如此，否则给全局变量赋值将会改动你硬盘上的二进制镜像，这是不可想象的。

下图中data段的例子更加复杂，因为它用了一个指针。在此情况下，指针gonzo（4字节内存地址）本身的值保存在data段中。而它所指向的实际字符串则不在这里。这个字符串保存在代码段中，代码段是只读的，保存了你全部的代码外加零零碎碎的东西，比如字符串字面值。代码段将你的二进制文件也映射到了内存中，但对此区域的写操作都会使你的程序收到段错误。这有助于防范指针错误，虽然不像在C语言编程时就注意防范来得那么有效。下图展示了这些段以及我们例子中的变量：



总结如下：



1）内核空间（kernel space）：包括命令行参数和环境变量，其大小恒定，缺省为1G

2）栈区（stack）：由编译器自动分配释放 ，存放函数的参数值，局部变量的值等，每个线程独自拥有。

3）内存映射区（mmap）：将文件内容映射为内存，高效IO。

4）堆区（heap）：一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。注意它与数据结构中的堆是两回事。

5）全局/静态区：全局变量和静态变量的存储是放在一块的，在程序编译时分配。又分为未初始化区（bss）和初始化（data）。bss存放未初始化的全局/静态变量，data存放有初始值的全局/静态变量。

6) 代码和只读数据区：对于所有进程来说，代码都是从同一固定地址开始，如Linux 32系统通常从0x08048000开始代码段（如前所述，从地址0到代码段起始地址的部分通常为操作系统保留区）。代码及只读数据区是直接按照可执行目标文件的内容初始化的，与目标文件中的代码段(.text)、初始化段（.init）及只读数据段（.rodata）相对应。

7)保留区：它并不是一个单一的内存区域，而是对地址空间中受到操作系统保护而禁止用户进程访问的地址区域的总称。大多数操作系统中，极小的地址通常都是不允许访问的，如NULL。C语言将无效指针赋值为0也是出于这种考虑，因为0地址上正常情况下不会存放有效的可访问数据

**2. 栈与堆的比较**

1）申请方式

stack: 由系统自动分配。例如，声明在函数中一个局部变量int b; 系统自动在栈中为b开辟空间

heap: 需要程序员自己申请，并指明大小，在c中malloc函数，如在C++中用new运算符

2）系统响应

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。另外，对于大多数系统，首地址处会记录这块内存空间中本次分配的大小，这样，代码中的delete语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

3）大小限制

栈：在Windows下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在WINDOWS下，栈的大小是2M（也有的说是1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。

堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。。由此可见，堆获得的空间比堆的大小受限于计算机系统中有效的虚拟内存较灵活，也比较大。

4）申请效率

栈由系统自动分配，速度较快。但程序员无法控制。

堆是由new分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便。

5）存储内容

栈在函数调用时，第一个进栈的是主函数的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的C编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。

堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容由程序员安排。

**3.栈与堆的区别**

1) 管理方式：对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生memory leak

2) 空间大小：一般来讲在32位系统下，堆内存可以达到4G的空间，从这个角度来看堆内存几乎是没有什么限制的。但是对于栈来讲，一般都是有一定的空间大小的，例如，在VC6下面，默认的栈空间大小是1M

3) 分配方式：堆都是动态分配的，没有静态分配的堆。内存有2种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由malloc,calloc函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。

4) 生长方向：对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长。

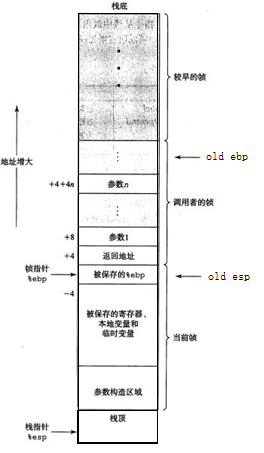
5） 碎片问题：对于堆来讲，频繁的new/ delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，因为栈是先进后出的，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出，在他弹出之前，在它上面的后进的栈内容已经被弹出

6) 生长方向：对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长

**4.函数调用实际过程**

### 4.1. 基础知识

#### 4.1.1.栈



栈，相信大家都十分熟悉，push/pop，只允许在一端进行操作，后进先出(LIFO)，凡是学过编程的人都能列出一二三点。但就是这个最简单的数据结构，构成了计算机中程序执行的基础，用于内核中程序执行的栈具有以下特点：

（1）每一个进程在用户态对应一个调用栈结构(call stack)，一个调用栈包含若干个栈帧，每个堆栈帧对应一个函数。

（2）程序中每一个未完成运行的函数对应一个栈帧(stack frame)，包含：

函数参数，默认调用惯例情况下从右向左的顺序依次把参数压入栈中。由主调函数执行。

函数返回地址，即调用方调用此函数（如call func1）的下一条指令的地址。函数调用方（call指令）执行。

保存调用方函数的EBP寄存器，即将调用方函数的EBP压入堆栈，并令EBP指向此栈中的地址：pushl %ebp; movl %esp, %ebp。由被调函数执行。

上下文：保存在函数调用过程中需要保持不变的寄存器（函数调用方的），如ebx，esi，edi等。由被调函数执行。

临时变量，如非静态局部变量。

（3）栈底对应高地址，栈顶对应低地址，栈由内存高地址向低地址生长

#### 4.1.2.寄存器

寄存器位于CPU内部，用于存放程序执行中用到的数据和指令，CPU从寄存器中取数据，相比从内存中取快得多。寄存器又分通用寄存器和特殊寄存器。

通用寄存器有ax/bx/cx/dx/di/si，尽管这些寄存器在大多数指令中可以任意选用，但也有一些规定某些指令只能用某个特定“通用”寄存器，例如函数返回时需将返回值mov到ax寄存器中；特殊寄存器有bp/sp/ip等，特殊寄存器均有特定用途，例如sp寄存器用于存放以上提到的栈帧的栈顶地址，除此之外，不用于存放局部变量，或其他用途。对于有特定用途的几个寄存器，简要介绍如下：

ax(accumulator): 可用于存放函数返回值

bp(base pointer): 用于存放执行中的函数对应的栈帧的栈底地址

sp(stack poinger): 用于存放执行中的函数对应的栈帧的栈顶地址

ip(instruction pointer): 指向当前执行指令的下一条指令

不同架构的CPU，寄存器名称被添以不同前缀以指示寄存器的大小。例如对于x86架构，字母“e”用作名称前缀，指示各寄存器大小为32位；对于x86\_64寄存器，字母“r”用作名称前缀，指示各寄存器大小为64位。

#### 4.1.3函数调用栈实现

1）将被调用函数的参数压栈（注：在x86\_64平台中，CPU拥有16个通用64-bit寄存器，故调用函数时，前6个参数通常由寄存器来传递，剩下的才通过栈传递）

2）将eip的值压入栈中，即函数调用指令的下一条指令地址作为返回地址压入栈中

3）修改指令指针寄存器eip的值，使其指向被调函数(foo)的执行位置。将执行权限交给被调函数 。

其中，第2、3步由call指令来实现。跳转到函数体后，该函数体的开始指令通常是这样的：

1）push %ebp ：将调用者的帧指针%ebp压栈，即保存旧栈帧的帧指针以便函数返回时恢复旧栈帧

2）mov %esp, %ebp：将当前栈顶地址传给%ebp，此时，%ebp既是旧栈帧的结束地址即之前的%esp，又是被调用者的新栈帧的起始地址

3）sub xxx, %esp：将栈顶下移，即为被调用函数开辟栈空间，xxx为立即数且通常为16的整数倍（这会浪费一些空间，但gcc采用该规则来保证数据的严格对齐）

4）push xxx：该命令为可选项，如有必要，由被调用者负责保存/恢复某些寄存器

不难推导出函数返回时通常由如下指令序列构成：

1）pop xxx：可选项，与进入函数时是否push xxx保存寄存器相对应

2）mov %ebp, %esp：恢复调用者的栈顶指针，即调用者栈帧的结束边界

3）pop %ebp：调用者的帧指针弹栈并修改%ebp的值，即恢复调用者栈帧的起始边界

4）ret：从栈中得到返回地址并修改%eip的值，并跳转到该位置处继续执行。

注意：函数退出前指令序列的第2、3步也可由指令leave来实现，具体用哪种方式，由编译器决定。上面给出的是函数调用时，典型的进入/退出指令序列，某些情况下，编译器生成的指令序列并不按照上面的方式进行。例如若C函数被声明为static（只在本编译单元内可见）且函数在编译单元内被直接调用，没有被显示或隐式取地址（即没有任何函数指针指向该函数），在这种情况下，编译器确信该函数不会被其它编译单元调用，因此可以随意修改其进/出指令序列以达到优化的目的。

# LINUX 输入输出与文件

## 1 基于文件指针的文件操作(缓冲)

linux 的文件和文件描述符

linux 中对目录和设备的操作都是文件操作，文件分为普通文件，目录文件，链接文件和设备文件。

普通文件，也称磁盘文件，并且能够进行随机的数据存储(能够自由 seek 定位到某一个位置)；

管道：是一个从一端发送数据，另一端接收数据的数据通道；

目录：也称为目录文件，它包含了保存在目录中文件列表的简单文件。

设备：该类型的文件提供了大多数物理设备的接口。它又分为两种类型： 字符型设备和块设备。字符型设备一次只能读出和写入一个字节的数据， 包括调制解调器、终端、打印机、声卡以及鼠标；块设备必须以一定大小 的块来读出或者写入数据，块设备包括 CD-ROM、RAM 驱动器和磁盘驱动 器等，一般而言，字符设备用于传输数据，块设备用于存储数据。

链接：类似于 Windows 的快捷方式和 Linux 里的别名，指包含到达另一个 文件路径的文件。

套接字：在 Linux 中,套接字也可以当作文件来进行处理。

基于文件指针的文件操作函数是 ANSI 标准函数库的一部分。

d ：目录

l ：符号链接 相当于快捷图标

b ：块设备文件 设备数据以区域为单位

c ：字符设备文件 设备数据额以单个字符为单位

p ：命名管道

s ：socket 文件。

### 1.1 文件的创建，打开与关闭

原型为：

#include <stdio.h> //头文件包含

FILE \*fopen(const char \*path,const char \*mode); //文件名 模式

FILE \*fdopen(int fd, const char \*mode);

FILE \*freopen(const char \*path, const char \*mode, FILE \*stream);

int fclose(FILE \*stream);

fopen 以 mode 的方式打开或创建文件流，

fdopen 以mode的方式把文件描述符关联到文件流，

freopen以mode的方式重新将一个文件关联到一个文件流

如果成功，将返回一个文件指针，失败则返回 NULL. 设置了标准错误描述，可使用perror函数查看。

fopen 创建的文件的访问权限将以 0666 与当前的 umask 结合来确定。

mode 的可选模式列表

模式 读 写 位置 截断原内容 创建

rb Y N 文件头 N N

rb+ Y Y 文件头 N N

wb N Y 文件头 Y Y

wb+ Y Y 文件头 Y Y

ab N Y 文件尾 N Y

ab+ Y Y 文件尾 N Y

在 Linux 系统中,mode 里面的’b’(二进制)可以去掉，但是为了保持与其他系统的兼容性，建议不要去掉。ab 和 ab+ 为追加模式，在此两种模式下，无 论文件读写点定位到何处，在写数据时都将是在文件末尾添加，所以比较适合于多进程写同一个文件的情况下保证数据的完整性。

### 1.2 读写文件

基于文件指针的数据读写函数较多，可分为如下几组：

**数据块读写**：

#include <stdio.h>

size\_t fread(void \*buf, size\_t size, size\_t nmemb, FILE \*stream);

size\_t fwrite(void \*buf, size\_t size, size\_t nmemb, FILE \*stream);

fread 从文件流 stream 中读取 nmemb 个元素，写到 buf指向的内存中，每个元素的大小为 size 个 字节。

fwrite 从 buf 指向的内存中读取 nmemb 个元素，写到文件流 stream 中，每个元素 size 个字节。 所有的文件读写函数都从文件的当前读写点开始读写，读写完以后，当前读写点自动往后移动 size\*nmemb 个字节。 整块 copy，速度较快，但是是二进制操作

成功后fread（）和fwrite（）返回读取或写入的项目数。如果发生错误或达到文件结尾，则返回值为短项目计数（或零）。  fread（）不区分文件结尾和错误，调用者必须使用feof和ferror来确定发生的情况。

**格式化写**：

#include <stdio.h>

int printf(const char \*format, ...); // fprintf(stdout,format,…)与dprintf(STDOUT\_FILENO,format,…)

int fprintf(FILE \*stream, const char \*format, ...);

int sprintf(char \*str, const char \*format, ...);

int dprintf(int fd, const char \*format, ...);

int snprintf(char \*str, size\_t size, const char \*format, ...);

printf：写入标准输出

fprintf：写入指定的流

dprintf：写入指定的文件描述符

sprintf：存入指定的数组buf内，会自动在结尾追加’\0’。sprintf可能会溢出，用户要确保buf尺寸

snprintf：相对于sprintf明确指定了尺寸，防止溢出问题。

printf系列成功返回输出的字节数，不包括末尾的空字符。

**格式化读**

int scanf(const char \*format, ...);

int fscanf(FILE \*stream, const char \*format, ...);

int sscanf(const char \*str, const char \*format, ...);

scanf从标准流中读取

fscanf从文件流中读取

sscanf从buf中读取

sscanf系列函数返回成功匹配并分配的输入项数;可能少于提供的，甚至为零。如果在第一次成功转换或发生匹配故障之前达到输入结束，则返回值EOF。 如果发生读取错误，也会返回EOF，在这种情况下，流的错误指示符（见ferror）被设置，并且errno被设置为指示错误。

以8个上函数均由下列函数实现：

#include <stdarg.h>

int vprintf(const char \*format, va\_list ap);

int vfprintf(FILE \*stream, const char \*format, va\_list ap);

int vdprintf(int fd, const char \*format, va\_list ap);

int vsprintf(char \*str, const char \*format, va\_list ap);

int vsnprintf(char \*str, size\_t size, const char \*format, va\_list ap);

int vscanf(const char \*format, va\_list ap);

int vsscanf(const char \*str, const char \*format, va\_list ap);

int vfscanf(FILE \*stream, const char \*format, va\_list ap);

**单个字符读：**

使用下列函数可以一次读写一个字符包括\n，\0，\r等控制字符。

#include <stdio.h>

int getc(FILE \*stream);  //等同于 fgetc(FILE\* stream)

int fgetc(FILE \*stream);

int getchar(void);  //等同于 fgetc(stdin);

int ungetc(int c, FILE \*stream);

getchar从标准输入流中读数据，其他函数从文件流stream 中读数据。

前三个函数成功返回读取的字符，出错或到达末尾返回EOF。

ugetc成功返回放回的字符，出错返回EOF。

**单个字符写：**

int putc(int c, FILE\*stream);  //等同于 fputc(int c, FILE\* stream)

int fputc(int c,FILE \*stream);

int putchar(int c);  //等同于 fputc(int c,stdout);

putchar向标准输入输出流中写数据，其他函数向文件流stream 中写数据。

函数成功返回写入的字符，出错返回EOF。

**字符串读**：

char \*gets(char \*s); // 等同于 fgets(const char\*s, int size, stdin);linux中弃用

char \*fgets(char \*s,int size,FILE \*stream);

成功返回写入的字符串，出错或到达末尾返回NULL.

**字符串写**：

int puts(const char \*s); // 等同于 fputs(const char\*s,stdout);

int fputs(const char \*s,FILE \*stream);

成功返回写入的字节数，出错返回EOF.

fgets 和 fputs 从文件流 stream 中读写一行数据;

puts 和 gets 从标准输入输出流中读写一行数据。 fgets 可以指定目标缓冲区的大小size，所以相对于 gets 安全，但是 fgets调用 时，如果文件中当前行的字符个数大于 size，则下一次 fgets 调用时，将继续读取该行剩下的字符,fgets 读取一行字符时，保留行尾的换行符，而gets不会。 fputs 不会在行尾自动添加换行符，但是 puts 会在标准输出流中自动添加一换行符。因此只能成套使用，不然就有可会造成换行符丢失。

**文件定位：**

文件定位指读取或设置文件当前读写点，所有的通过文件指针读写数据的函数，都是从文件的当前读写点读写数据的。

常用的函数有：

int feof(FILE \*stream); //通常的用法为 while(!feof(fp))

int fseek(FILE \*stream, long offset, int whence);//将当前读写点从whence处偏移offset个长度

long ftell(FILE \*stream); //用来获得文件流当前的读写位置

void rewind(FILE \*stream); //把文件流的读写位置移至文件开头=fseek(fp, 0, SEEK\_SET);

int fgetpos(FILE \*stream, fpos\_t \*pos);

int fsetpos(FILE \*stream, const fpos\_t \*pos);

feof 判断是否到达文件末尾的下一个（注意到达文件末尾之后还会做一次），是返回1，否返回0.

fseek 设置当前读写点到偏移 whence 长度为 offset 处，成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看，whence 可以是：

SEEK\_SET (文件开头 0)

SEEK\_CUR (文件当前位置 1)

SEEK\_END (文件末尾 2)

ftell 获取当前的读写点 。返回当前文件读写标记位置，出错返回-1，设置了标准错误描述，可使用perror函数查看。

rewind 将文件当前读写点移动到文件头。

fgetpos/fsetpos获取或者设置当前文件读写标记位的位置，成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看

**文件错误**

#include <stdio.h>

void clearerr(FILE \*stream);

int ferror(FILE \*stream);

int fileno(FILE \*stream);

clearerr clearerr函数重置错误标记和给出的流的EOF指针. 当发生错误时,你可以使用perror()判断实际上发生了何种错误.

ferror()函数检查stream(流)中的错误, 如果没发生错误返回0,否则返回非零. 如果发生错误, 使用perror()检测发生什么错误.

fileno获取流对应的描述符

### 1.3 mode\_t详解与改变目录或文件的访问权限

#include<sys/stat.h>

int chmod(const char\* path, mode\_t mode);//mode 形如：0777,

int fchmod(int fd, mode\_t mode);

int fchmodat(int fd, const char \*path, mode\_t mode, int flag);

参数path指定的文件访问权限被修改为mode参数给出的访问权限。可以是绝对或者相对路径。当path是相对路径时，fchmodat是相对fd而言，而chmod是相对当前工作目录而言；当path时绝对路径时fd会被忽略。

参数fd文件描述符，若fd设为AT\_FDCWD则fchmodat和chmod一样。

参数mode 读写权限。

参数flags 定义在fcntl.h头文件中。一般为0.

都是成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看。

**mode\_t详解**

mode\_t 是unsiged int，一共32位取其中16位使用，高4位文件类型（4是目录8是普通文件），中3位，特殊权限（s和S以及T和t权限），后9位读写执行权限。

**普通读写执行权限:**

S\_IRWXU 00700 权限，代表该文件所有者具有可读、可写及可执行的权限。

S\_IRUSR 或 S\_IREAD，00400 权限，代表该文件所有者具有可读取的权限。

S\_IWUSR 或 S\_IWRITE，00200 权限，代表该文件所有者具有可写入的权限。

S\_IXUSR 或 S\_IEXEC，00100 权限，代表该文件所有者具有可执行的权限。

S\_IRWXG 00070 权限，代表该文件用户组具有可读、可写及可执行的权限。

S\_IRGRP 00040 权限，代表该文件用户组具有可读的权限。

S\_IWGRP 00020 权限，代表该文件用户组具有可写入的权限。

S\_IXGRP 00010 权限，代表该文件用户组具有可执行的权限。

S\_IRWXO 00007 权限，代表其他用户具有可读、可写及可执行的权限。

S\_IROTH 00004 权限，代表其他用户具有可读的权限

S\_IWOTH 00002 权限，代表其他用户具有可写入的权限。

S\_IXOTH 00001 权限，代表其他用户具有可执行的权限。

**文件特殊权限:**

S\_ISUID 04000权限：任何用户执行具有SUID权限和x权限的文件时拥有此文件所有者的权限，此文件执行时其euid会变成文件所有者的UID。如果对目录设置了SUID权限，则所有用户在该目录下创建的文件都属于该目录所有者，而不是创建者。

S\_ISGID权限02000：任何用户组执行具有SGID权限和x权限的文件时拥有此文件所在组的权限，此文件执行时其egid会变成文件所在组的GID。如果对目录设置了SGID权限，则所有用户在该目录下创建的文件都属于该目录所在组，而不是创建者所在组。

S\_ISVTX权限01000：只对目录有效，使任何用户都可以在该目录下创建文件，但只有文件所有者和root才能删除，即是其他人对目录有写权限。

chmod u+s 文件名：增加SUID权限，（4）原来有x权限显示为s，原来没有x权限显示为S

chmod u-s 文件名：删除SUID权限

chmod g+s 文件名：增加SGID权限，（2）原来有x权限显示为s，原来没有x权限显示为S

chmod g-s 文件名：删除SGID权限，

chmod o+t 文件名：增加SBIT权限，（1）原来有x权限显示为t，原来没有x权限显示为T

chmod o-t 文件名：删除SBIT权限

**权限操作**

**获取对应位权限状态或者文件类型**

通过文件模式字st\_mode与S\_ISGID或者S\_ISUID进行“按位与”运算可以判断对应位的状态，即：

st\_mode & S\_ISGID：取得S\_ISGID位的状态。

st\_mode & S\_ISUID：取得S\_ISUID位的状态。

如：某文件权限drwxrwxr-x 十进制为0004 000 421 421 401

0100 000 111 111 101

0000 100 000 000 000 按位与

——————————————

0000 000 000 000 000

说明此文件不具有S\_ISGID权限

**增加对应权限**

通过文件模式字st\_mode与S\_ISGID或者S\_ISUID进行“按位或”运算可以置位对应位，即：

st\_mode | S\_ISGID：置位S\_ISGID位。

st\_mode | S\_ISUID：置位S\_ISUID位。

如：某文件权限drwxrwxr-x 十进制为0004 000 421 421 401

0100 000 111 111 101

0000 100 000 000 000 按位或

——————————————

0100 100 111 111 101

说明给此文件增加了S\_ISGID权限

**删除对应位权限**

通过文件模式字st\_mode与S\_ISGID或者S\_ISUID反码进行“按位与”(相当于减法)运算可以清除对应位，即：

st\_mode & (~S\_ISGID)：清除S\_ISGID位。

st\_mode & (~S\_ISUID)：清除S\_ISUID位。

如：某文件权限drwsrwxr-x 十进制为0004 400 421 421 401

0100 100 111 111 101

0000 100 000 000 000 按位取反

1111 011 111 111 111 再按位与

——————————————

0100 000 111 111 101

说明给此文件删除了S\_ISGID权限

但是通常采用直接赋数值的形式,如：

int fd **=** open**(**“1**.**txt”**,**O\_WRONLY**|**O\_CREAT**,0**0755**);** //表示给 755 的权限 ，基于文件描述符

**if(-**1 **==** fd**)**

**{**

perror**(**"open failed!\n"**);**

exit**(-**1**);**

**}**

### 1.4 目录操作（类似文件操作）

**获取、改变当前目录：**

#include <unistd.h> //头文件

char \*getcwd(char \*buf, size\_t size); //获取当前目录，相当于 pwd 命令

char \*getwd(char \*buf);

char \*get\_current\_dir\_name(void);

int chdir(const char \*path); //修改当前目录，即切换目录，相当于 cd 命令

getcwd()函数：将当前的工作目录绝对路径复制到参数 buf 所指的内存空间，返回值为指向指向该路径的字符串指针。size 为 buf 的空间大小. 在调用此函数时，buf 所指的内存空间要足够大，若工作目录绝对路径的字符串长度超过参数 size 大小，则返回NULL，errno 的值则为 ERANGE。倘若参数 buf 为 NULL，getcwd()会依参数size 的大小自动配置内存(使用 malloc())，如果参数 size 也为 0，则 getcwd()会依工作目录绝对路径的字符串长度来决定所配置的内存大小，进程可以在使用完此字符串后自动利用 free()来释放此空间。所以常用的形式：getcwd(NULL, 0);

绝对路径： /home/用户名

相对路径： 从当前目录

getwd函数与getcwd的区别在于不会自动分配内存，在调用此函数时，buf 所指的内存空间要足够大，若工作目录绝对路径的字符串长度超过buf大小，则返回NULL，errno 的值则为 NAMETOOLONG，现在已经废弃。

get\_current\_dir\_name与getcwd的区别在自动分配内存，无需指定buf。

这三个函数成功都返回字符串指针，出错返回NULL，设置了标准错误描述，可使用perror函数查看

chdir()函数：用来将当前的工作目录改变成以参数 path 所指的目录 。成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看。

举例：

#include<unistd.h>

main**()**

**{**

chdir**(**“**/**tmp”**);**

printf**(**“current working directory**:** **%**s\n”**,**getcwd**(NULL,**0**));**

**}**

**创建和删除目录：**

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

int mkdir(const char \*pathname, mode\_t mode); //mode是目录权限，可以是相对或绝对路径

int mkdirat(int fd, const char \*path, mode\_t mode);

int rmdir(const char \*pathname); //删除空目录

int remove(const char \*pathname)//删除文件或目录

mkdir与mkdirat都是创建一个空文件夹，可以是相对或绝对路径，如果path是一个链接则mkdir将失败并返回EEXIST。

mkdirat与mkdir的区别在于当path是相对路径时，mkdir是相对与当前工作目录，而mkdirat是相对于fd，如果fd没有设置O\_SEARCH，则会检查权限，如果设置了则不会检查。如果fd设为AT\_FDCWD，则mkdirat与mkdir一样。

rmdir删除一个目录。

remove删除一个文件或目录。调用unlink删除文件，调用rmdir来删除目录。如果文件未被打开则立即删除，否则保留到文件关闭后才删除。如果是一个符号链接、socket、FIFO、设备则直接删除。

都是成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看。

**获取目录信息：**

#include <sys/types.h>

#include <dirent.h>

DIR \*opendir(const char \*name); //打开一个目录，DIR目录流，相当于FILE文件流

DIR \*fdopendir(int fd);

opendir打开一个目录。

fdopendir打开fd所指的目录。

成功返回目录流指针，出错返回NULL，设置了标准错误描述，可使用perror函数查看。

struct dirent \*readdir(DIR \*dir); //读取目录的一项信息，并返回该项信息的结构体指针

int readdir\_r(DIR \*dirp, struct dirent \*entry, struct dirent \*\*result);

readdir读取一个目录信息。成功返回指向dirp指向的目录流中的下一个目录条目的dirent结构体指针。 在到达目录流的末尾或者发生错误时返回NULL。设置了标准错误描述，可使用perror函数查看。

readdir\_r() 函数是 readdir() 函数可重入版本。它从目录流dirp 里读取下一个目录项，并且通过调用者分配的缓存区 entry返回。返回条目的指针被放置于 \*result 里；如果目录流到达结尾，那么把\*result 设置为 NULL。成功返回0，出错返回错误码，到达末尾时，返回0，置result为NULL。

void rewinddir(DIR \*dir); //重新定位到目录文件的头部

rewinddir重新将目录流dir的读写标记位置为开始

void seekdir(DIR \*dir,off\_t offset);//用来设置目录流目前的读取位置 .offset距离开始文件的距离

seekdir设置目录流dir的读写标志位在距离开始位置的offset处

off\_t telldir(DIR \*dir); //返回目录流当前的读取位置

telldir获取目录流dir的读写标记位位置。成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看。

int closedir(DIR \*dir); //关闭目录文件

closedir关闭目录流dir，成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看。

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

int stat(const char \*pathname, struct stat \*buf); //获取文件状态

int fstat(int fd, struct stat \*buf);

int lstat(const char \*pathname, struct stat \*buf);

int fstatat(int dirfd, const char \*pathname, struct stat \*buf, int flags);

stat,fstat,lstat,fstatat 都是获取一个文件或目录的详细信息存放在buf中。

stat与fstat的区别在要获取信息的文件由文件描述符fd指定，而不是文件名。

stat与lstat的区别在于当pathname是一个链接时，获取其本身信息而不时其链接的文件的信息。

stat与fstatat的区别在于当pathname是相对路径时，stat是相对当前工作目录而言，而fstatat是相对于文件描述符dirfd而言，如果dirfd设为AT\_FDCWD，则与stat一样；当pathname是绝对路径时，dirfd被忽略。

都是成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看。

**读取目录信息的步骤为： **

①用 opendir 函数打开目录；

②用 readdir 函数迭代读取目录的内容，如果已经读取到目录末尾，又想重新开始读，则可以 使用 rewinddir 函数将文件目录指针重新定位到目录文件的起始位置； 

③用 closedir 函数关闭目录

opendir()用来打开参数 name 指定的目录，并返回 DIR\*形态的目录流，和文件操作函数 open()类 似，接下来对目录的读取和搜索都要使用此返回值。函数失败则返回 NULL；读完一个目录后自动偏移。不需手动偏移。

DIR结构体定义如下：

struct \_\_dirstream

{

void \*\_\_fd;

char \*\_\_data;

int \_\_entry\_data;

char \*\_\_ptr;

int \_\_entry\_ptr;

size\_t \_\_allocation;

size\_t \_\_size;

\_\_libc\_lock\_define (, \_\_lock)

};

typedef struct \_\_dirstream DIR;

DIR结构体类似于FILE，是一个内部结构。了解即可。

readdir()函数用来读取目录的信息，并返回一个结构体指针，该指针保存了目录的相关信息。有错误发生或者读取到目录文件尾则返回 NULL；dirent 结构体如下：

struct dirent

{

ino\_t d\_ino; /\* inode number（此目录进入点的索引节点号） \*/

off\_t d\_off; /\* offset to the next dirent（目录开头到进入点的位移 \*/

unsigned short d\_reclen; /\* length of this record（目录名的长度） \*/

unsigned char d\_type; /\* type of file（所指的文件类型）4是目录8是普通文件 \*/

char d\_name[256]; /\* filename（文件名或文件目录名,不含路径） \*/

};

dirent相当于stat结构体的简化版本。dirent结构体存储的关于文件的信息很少，所以dirent同样也是起着一个索引的作用，如果想获得类似ls -l那种效果的文件信息，必须要靠stat函数了。通过readdir函数读取到的文件名存储在结构体dirent的d\_name成员中，而函数stat的作用就是获取文件名为d\_name的文件的详细信息，存储在stat结构体中。

索引节点号inode：文件或者文件目录在磁盘上都是按512B，分块存储，多个文件名可以对应一个inode，每一个inode对应磁盘的一个区域（若干块），找到inode则可以进去对应文件或文件目录。

seekdir()函数用来设置目录流目前的读取位置，再调用 readdir()函数时，便可以从此新位置开始 读取。参数 offset 代表距离目录文件开头的偏移量。

telldir()函数用来返回目录流当前的读取位置。

结构体 stat 的定义为：

struct stat

{

dev\_t st\_dev; /\*如果是设备，返回设备表述符，否则为 0\*/

ino\_t st\_ino; /\* i 节点号 \*/

mode\_t st\_mode; /\* 文件类型 \*/

nlink\_t st\_nlink; /\* 链接数 \*/

uid\_t st\_uid; /\* 属主 ID \*/

gid\_t st\_gid; /\* 组 ID \*/

dev\_t st\_rdev; /\* 设备类型\*/

off\_t st\_size; /\* 文件大小，字节表示 \*/

blksize\_t st\_blksize; /\* 块大小\*/

blkcnt\_t st\_blocks; /\* 块数 \*/

time\_t st\_atime; /\* 最后访问时间，相对于1970 1月1日0时0分0秒的秒数\*/

time\_t st\_mtime; /\* 最后内容修改时间\*/

time\_t st\_ctime; /\* 最后权限修改时间 \*/

};

man 2 stat看 time\_t,可以看到time\_t为int整数。相对于1970年1月1日0时0分0秒的秒数。time\_t 的结构，在头文件 /usr/include/linux/time.h

st\_mtime

struct timespec

{

\_\_kernel\_time\_t tv\_sec; /\* seconds \*/

long tv\_nsec; /\* nanoseconds \*/

};

\_\_kernel\_time\_t 是 long 类型的

ctime()将time\_t时间转为字符串时间。用sprintf和sscanf格式化读取流。符串格式为”Wed Jun 30 21 :49 :08 1993\n”

时间转换函数原型：

#include<time.h>

char \*asctime(const struct tm \*tm);

char \*asctime\_r(const struct tm \*tm, char \*buf);

char \*ctime(const time\_t \*timep);

char \*ctime\_r(const time\_t \*timep, char \*buf);

struct tm \*gmtime(const time\_t \*timep);

struct tm \*gmtime\_r(const time\_t \*timep, struct tm \*result);

struct tm \*localtime(const time\_t \*timep);

struct tm \*localtime\_r(const time\_t \*timep, struct tm \*result);

time\_t mktime(struct tm \*tm);

time\_t time(time\_t \*tloc);

size\_t strftime(char \*s, size\_t max, const char \*format, const struct tm \*tm);

ctime将time\_t时间（当前时间自1970年1月1日0时0分0秒的时间的秒数）转为字符串时间（星期 月 日 时 分 秒 年）；

gmtime将time\_t 时间转为自1900的tm类型时间。

asctime 将 tm 类型的时间转为asc字符串时间

localtime将time\_t时间转为当地time时间。

以上函数成功返回对应指针，出错返回NULL。

mktime 将tm时间转为time\_t时间

time取得当前距离1970年开始的time\_t时间。

以上函数成功返回对应时间，出错返回-1。

strftime按照参数fmt所设定格式将tm类型的格式化为字符串时间，然后存储在字符串str中（包括末尾空字符至多maxsize 个字符,如果超过返回0）。函数返回值为处理结果字符串str中字符的个数，如果发生错误返回零。但是返回0不一定就是发生错误，用于设定时间不同类型的代码为：

|  |  |  |  |
| --- | --- | --- | --- |
| 代码 | 含义 | 代码 | 含义 |
| %a | 星期的缩略形式 | %p | 本地时间的上午或下午（AM or PM） |
| %A | 星期的完整形式 | %S | 秒钟(0-59) |
| %b | 月份的缩略形式 | %U | 年中的第几周，星期天是一周的第一天 |
| %B | 月份的完整形式 | %w | 星期几的数字表示(0-6, 星期天=0) |
| %c | 月份的缩略形式 | %W | 一年中的第几周，星期天是一周的第一天 |
| %d | 月中的第几天(1-31) | %x | 标准日期字符串 |
| %H | 小时, 24小时格式 (0-23) | %X | 标准时间字符串 |
| %I | 小时, 12小时格式 (1-12) | %y | 年(0-99) |
| %j | 年中的第几天(1-366) | %Y | 用CCYY表示的年（如：2004） |
| %m | 月份 (1-12)，在某些版本的vc中为0-11 | %Z | 时区名 |
| %M | 分钟(0-59) | %% | 百分号 |

struct tm

{

int tm\_sec; //取值[0,59)，非正常情况下可到达61

int tm\_min; //取值同上

int tm\_hour; //取值[0,23)

int tm\_mday; //取值(1,31]

int tm\_mon; //取值[0,11)

int tm\_year; //1900年起距今的年数

int tm\_wday; //取值[0,6]

int tm\_yday; //取值[0，366)

int tm\_isdst; //夏令时标志

};

#include <sys/types.h>

#include <pwd.h>

struct passwd \*getpwnam(const char \*name);

struct passwd \*getpwuid(uid\_t uid);

int getpwnam\_r(const char \*name, struct passwd \*pwd, char \*buf, size\_t buflen, struct passwd \*\*result);

int getpwuid\_r(uid\_t uid, struct passwd \*pwd, char \*buf, size\_t buflen, struct passwd \*\*result);

struct passwd \*getpwent(void);

void setpwent(void);

void endpwent(void);

getpwnam用来逐一搜索参数name指定的账号名称, 找到时便将该用户的数据以passwd 结构返回。

返回值：返回 passwd 结构数据, 如果返回NULL 则表示已无数据, 或有错误发生. 设置了标准错误描述，可使用perror函数查看。

getpwuid与getpwnam不同之处在于通过uid查找。

getpwnam\_r与getpwuisd\_r分别时getpwnam和getpwuid的可重入版本。查询结果放在用户分配大小为buflen的缓冲区pwd中，该缓冲区的地址放在result中。成功时返回0，并把result指向pwd，没有找到返回0，置result为NULL，出错时返回错误码，置result为NULL。

getpwent()用来从密码文件（/etc/passwd）中读取一项用户数据，该用户的数据以passwd 结构返回。第一次调用时会取得第一位用户数据，之后每调用一次就会返回下一项数据，直到已无任何数据.成功返回回pssswd结构体指针，出错或到达末尾返回NULL。设置了标准错误描述，可使用perror函数查看。

setpwent将getpwent打开的密码文件的读写指针置为开始位置。

endpwent将getpwent打开的密码文件关闭。

struct passwd

{

char \*pw\_name; /\* username \*/

char \*pw\_passwd; /\* user password \*/

uid\_t pw\_uid; /\* user ID \*/

gid\_t pw\_gid; /\* group ID \*/

char \*pw\_gecos; /\* user information \*/

char \*pw\_dir; /\* home directory \*/

char \*pw\_shell; /\* shell program \*/

};

#include <sys/types.h>

#include <grp.h>

struct group \*getgrnam(const char \*name);

struct group \*getgrgid(gid\_t gid);

int getgrnam\_r(const char \*name, struct group \*grp, char \*buf, size\_t buflen, struct group \*\*result);

int getgrgid\_r(gid\_t gid, struct group \*grp, char \*buf, size\_t buflen, struct group \*\*result);

getgrnam用来逐一搜索参数name指定的组名称, 找到时便将该组的数据以group 结构返回。

返回值：返回group 结构体指针, 如果返回NULL 则表示已无数据, 或有错误发生. 设置了标准错误描述，可使用perror函数查看。

getgrgid与getgrnam不同之处在于通过gid查找。

getgrnam\_r与getgrgid\_r分别时getgrnam和getgrgid的可重入版本。查询结果放在用户分配大小为buflen的缓冲区grp中，该缓冲区的地址放在result中。成功时返回0，并把result指向grp，没有找到返回0，置result为NULL，出错时返回错误码，置result为NULL。

struct group

{

char \*gr\_name; /\* group name \*/

char \*gr\_passwd; \* group password \*/

gid\_t gr\_gid; /\* group ID \*/

char \*\*gr\_mem; /\* NULL-terminated array of pointers to names of group members \*/

};

对于结构体的成员 st\_mode，有一组宏可以进行文件类型的判断（原理就是“按位与”） ：

宏 描述

S\_ISLNK(st\_mode) 判断是否是符号链接

S\_ISREG(st\_mode) 判断是否是普通文件

S\_ISDIR(st\_mode) 判断是否是目录

S\_ISCHR(st\_mode) 判断是否是字符型设备

S\_ISBLK(st\_mode) 判断是否是块设备

S\_ISFIFO(st\_mode) 判断是否是命名管道

S\_ISSOCK(st\_mode) 判断是否是套接字

示例：

#include <stdio.h>

#include <sys/types.h>

#include <dirent.h>

int main**(**int argc**,**char **\***argv**[])**

**{**

struct dirent **\***pDirInfo**;**

DIR **\***pDir；

**if(**argc **<** 2**)**

pDir **=** opendir**(**"."**);**

**else**

pDir **=** opendir**(**argv**[**1**]);**

**if(NULL** **==** pDir**)**

**{**

perror**(**"open dir fail!"**);**

**return** **-**1**;**

**}**

**while(** **(**pDirInfo **=** readdir**(**pDir**))** **!=** **NULL** **)**

printf**(**"%s\n"**,**pDirInfo**->**d\_name**);**

closedir**(**pDir**);**

**return** 0**;**

**}**

举例：以树形结构的形式输出指定目录下面的所有文件

#include <unistd.h>

#include <stdio.h>

#include <dirent.h>

#include <string.h>

#include <sys/stat.h>

#include <stdlib.h>

void printdir**(**char **\***dir**,** int depth**)**

**{**

DIR **\***dp **=** opendir**(**dir**);**

**if(NULL** **==** dp**)**

**{**

fprintf**(**stderr**,**"cannot open directory: %s\n"**,** dir**);**

**return;**

**}**

chdir**(**dir**);**

struct dirent **\***entry**;**

struct stat statbuf**;**

**while((**entry **=** readdir**(**dp**))** **!=** **NULL)**

**{**

stat**(**entry**->**d\_name**,&**statbuf**);**

**if(**S\_ISDIR**(**statbuf**.**st\_mode**))**

**{**

**if(**strcmp**(**"."**,**entry**->**d\_name**)** **==** 0 **||** strcmp**(**".."**,**entry**->**d\_name**)** **==** 0**)**

**continue;**

printf**(**"%\*s%s/\n"**,**depth**,**""**,**entry**->**d\_name**);**

printdir**(**entry**->**d\_name**,**depth**+**4**);**

**}**

**else**

printf**(**"%\*s%s\n"**,**depth**,**""**,**entry**->**d\_name**);**

//printf(“%\*s”,4,”\*”); 该函数表示输出“\_\_\_\*”,前面输出 3 个空格。

//如果是 printf(“%\*s”,4,“\*\*”);则表示输出“\_\_\*\*”，前面输出 2 个空格。

**}**

chdir**(**".."**);**

closedir**(**dp**);**

**}**

int main**(**int argc**,** char**\*** argv**[])**

**{**

char **\***topdir**,** pwd**[**2**]=**"."**;**

**if** **(**argc **<** 2**)**

topdir**=**pwd**;**

**else**

topdir**=**argv**[**1**];**

printf**(**"Directory scan of %s\n"**,**topdir**);**

printdir**(**topdir**,**0**);**

printf**(**"done.\n"**);**

exit**(**0**);**

**}**

### 1.5 标准输入/输出流

在进程一开始运行，就自动打开了三个对应设备的文件，它们是标准输入、标准输出、标准出错输出流，分别用全局文件指针 stdin、stdout、stderr 表示，stdin 具有可读属性，缺省情况下是指从键盘的读取输入，stdout 和 stderr 具有可写属性，缺省情况下是指向屏幕输出数据。

举例：

#include <stdio.h>

#include <unistd.h>

int main**()**

**{**

char szBuf**[**32**];**

printf**(**"Input string:"**);** //向屏幕输出一字符串

fgets**(**szBuf**,sizeof(**szBuf**),**stdin**);**//从键盘读入一行字符串

fprintf**(**stdout**,**"The string is:%s"**,**szBuf**);**//向屏幕输出一行字符串

**return** 0**;**

**}**

## 2 基于文件描述符的文件操作(非缓冲)

### 2.1 文件描述符

内核为每个进程维护一个已打开文件的记录表，文件描述符是一个较小的正整数（0—1023），它代表记录表的一项，通过文件描述符和一组基于文件描述符的文件操作函数，就可以实现对文件的 读、写、创建、删除等操作。常用基于文件描述符的函数有 open（打开）、creat（创建）、close （关闭）、read（读取）、write（写入）、ftruncate（改变文件大小）、lseek（定位）、fsync （同步）、fstat（获取文件状态）、fchmod（权限）、flock（加锁）、fcntl（控制文件属性）、 dup（复制）、dup2、select 和 ioctl。基于文件描述符的文件操作并非 ANSI C 的函数。 如果不清楚某个函数的具体实现形式，可以通过下面的方式查询：

函数名 查看该函数的帮助。

### 2.2 打开、创建和关闭文件

open 和 creat 都能打开和创建函数，

#include <sys/types.h> //头文件

#include <sys/stat.h>

#include <fcntl.h>

int open(const char \*path, int flags, mode\_t mode);//文件名 打开方式 权限，常用

int openat(int fd, const char \*path, int oflag, mode\_t mode);

int creat(const char \*pathname, mode\_t mode); //文件名权限 //现在已经不常用了

open与openat都是创建一个文件描述符。

openat 与open的区别在于当path为相对路径时，open相对当前工作目录而言，openat相对fd而言，如果fd为AT\_FDCWD，则和open一样，当path为绝对路时，fd被忽略，如果fd不以O\_SEARCH打开则会检查权限，反之则不会检查。

creat函数调用open(pathname,O\_CREAT|O\_TRUNC|O\_WRONLY,mode);

都是成功返回描述符，出错返回-1，设置了标准错误描述，可使用perror函数查看。

相关参数如下：

flags 和 mode 都是一组掩码的合成值，flags 表示打开或创建的方式，mode表示文件的访问权限。

**参数flags 的可选项有** ：

O\_RDONLY 以只读的方式打开

O\_WRONLY 以只写的方式打开 ，原内容不会消失

O\_RDWR 以读写的方式打开

前三个为主要标志,其余的为辅助标志位。主要标志位是互斥的，只能与辅助标志位进行按位或运算。

O\_APPEND，以追加的方式打开文件，每次调用 write 时，文件指针自动先移到文件尾，用于多进程写同一个文件的情况。 就像使用lseek一样。 如果多个进程将数据一次附加到文件，则O\_APPEND可能会导致NFS文件系统上的文件损坏。 这是因为NFS不支持附加到文件，所以客户端内核必须模拟它。

O\_ASYNC，启用信号驱动的I / O，当该文件描述符可以输入或输出时，产生一个信号（默认为SIGIO，但是可以通过fcntl进行更改）。 此功能仅适用于最终用户，假终端，套接字和管道和FIFO。

O\_CLOEXEC 为新的文件描述符启用close-on-exec标志位，在子进程中使用额exec函数族时自动关闭文件描述符。注意，在某些多线程程序中使用此标志是必不可少的，因为单独使用fcntl的F\_SETFD操作来设置FD\_CLOEXEC标志并不足以避免一个线程打开文件描述符并设置其close-on-exec标志位，而另一个线程同时fork子进程并执行exec函数族，可能导致open（）返回的文件描述符被无意泄漏到由fork创建的子进程执行的程序的情况。

O\_CREAT，如果该文件不存在，它将被创建。 文件的所有者（用户ID）被设置为进程的有效用户ID。 组所有权（组ID）设置为进程的有效组ID或父目录的组ID（取决于文件系统类型和装载选项以及父目录的模式）。

O\_DIRECT，尽量减少来自该文件的I / O缓存的影响。 一般来说，这会降低性能，但是在特殊情况下，例如应用程序执行自己的缓存时很有用。文件I / O直接从用户空间缓冲区完成。 O\_DIRECT标志是努力同步传输数据，但并没有给O\_SYNC标志来保证传输数据和必要的元数据。 为了保证同步I / O，除了O\_DIRECT之外，还必须使用O\_SYNC。

O\_DIRECTORY，打开一个目录，如果参数pathname不是目录，则打开失败。

O\_DSYNC，根据同步I / O数据完整性的要求，完成对文件的写操作。在write操作返回时，数据就会同步到底层硬件，检索该数据所需的任何文件元数据（就好像每个write操作后调用fdatasync）。

O\_EXCL，与 O\_CREAT 连用，如果文件已存在，则open失败。 当指定这两个标志时，如果路径名是符号链接，则无论符号链接指向何处，open（）将失败。一般来说， O\_EXCL不与O\_CREAT连用时的行为是未定义的，但有一个例外：在Linux 2.6及更高版本上，如果路径名称为块设备，则O\_EXCL可以不使用O\_CREAT来使用。但是如果系统正在使用块设备，则open（）将失败，出现错误EBUSY。

O\_LARGEFILE，打开超大文件。允许打开不能在off\_t(32位中off\_t是int)中显示的文件（但可以在off64\_t中显示）。 必须定义\_LARGEFILE64\_SOURCE宏（在包含任何头文件之前）才能获得此定义。在64位系统中无实际意义。

O\_NOATIME，当文件被读取时，不要更新文件最后访问时间（inode中的st\_atime）。 该标志旨在用于索引或备份程序，其使用可以明显减少磁盘活动量。 该标志可能对所有文件系统无效， 如NFS，其中服务器维护访问时间。

O\_NOCTTY，如果路径名是指终端设备，即使进程没有进程，也不会成为进程的控制终端。

O\_NOFOLLOW，如果路径名是符号链接，则打开失败。 这是一个FreeBSD扩展。

O\_NONBLOCK或O\_NDELAY，如果可能，文件以非阻塞模式打开。对所返回的文件描述符的open（）和任何后续操作都不会导致调用进程等待。请注意，此标志对常规文件和块设备无效; 也就是说，无论是否设置了O\_NONBLOCK，请求设备时，I / O操作将（短暂）阻塞。 由于O\_NONBLOCK语义最终可能会被实现，所以当为常规文件和块设备指定此标志时，程序不应取决于阻塞行为。对于FIFO（命名管道）的处理，请查询fifo。有关O\_NONBLOCK与强制性文件锁和文件租约结合的讨论，请参阅fcntl。

O\_PATH，获取可用于两个目的的文件描述符：表示文件系统树中的位置和执行纯文本描述级别的操作。文件本身未打开，其他文件操作（如，read（2），write（2），fchmod（2），fchown（2），fgetxattr（2），mmap（2））会失败并显示错误EBADF。

可以为生成的文件描述符执行以下操作：

\*close; fchdir; fstat。

\*复制文件描述符（dup（2），fcntl（2）F\_DUPFD等）。

\*获取并设置文件描述符标志（fcntl（2）F\_GETFD和F\_SETFD）。

\*使用fcntl（2）F\_GETFL操作检索打开的文件状态标志：返回的标志将包含位O\_PATH。

\*作为openir和“\* at（）”系统调用的参数，如linkat。

\*通过UNIX域套接字将文件描述符传递给另一个进程。

当标志中指定O\_PATH时，将忽略除O\_CLOEXEC，O\_DIRECTORY和O\_NOFOLLOW之外的其他标志。如果路径名称是符号链接，并且还指定了O\_NOFOLLOW标志，则该调用返回引用符号链接的文件描述符。该文件描述符可用于fchownat，fstatat，linkat和readlinkat中的dirfd参数，并使用空的pathname使调用在符号链接上运行。

O\_SYNC，根据同步I / O文件完整性完成的要求，完成写操作（与O\_DSYNC提供的同步I / O数据完整性完成相反），在write返回时，输出数据和相关的文件元数据已被传输到底层硬件（就像每个write后跟一个fsync）。

O\_TMPFILE，创建一个未命名的临时文件。 pathname参数指定一个目录;将在该目录的文件系统中创建一个未命名的inode。当最后一个文件描述符关闭时，写入生成的文件的任何东西都将丢失，除非该文件被赋予一个名称。O\_TMPFILE 必须与O\_RDWR或O\_WRONLY之一连用，并且可以指定O\_EXCL。如果未指定O\_EXCL，则可以使用linkat将临时文件链接到文件系统中，使其成为永久文件，比如：

char path[PATH\_MAX];

fd = open("/path/to/dir", O\_TMPFILE | O\_RDWR, S\_IRUSR | S\_IWUSR);

/\* File I/O on 'fd'... \*/

snprintf(path, PATH\_MAX, "/proc/self/fd/%d", fd);

linkat(AT\_FDCWD, path, AT\_FDCWD, "/path/for/file",AT\_SYMLINK\_FOLLOW);

在这种情况下，open的mode参数确定文件权限模式，与O\_CREAT一样。与O\_TMPFILE一起指定O\_EXCL可以防止以上述方式将临时文件链接到文件系统中。（请注意，在这种情况下，O\_EXCL的含义与其他O\_EXCL的含义不同。）

两个主要的使用用例：

（1）改进的tmpfile功能：无竞争创建临时文件

1）在关闭时自动删除;

2）永远不能通过任何路径名到达;

3）不受符号链接攻击;和

4）不要求调用者设计唯一的名称。

 （2）创建最初不可见的文件，然后使用数据进行填充并调整为具有适当的文件系统属性（chown，chmod，fsetxattr等），然后将其原子化链接到文件系统中完全形成状态（如上所述使用linkat）。

O\_TMPFILE需要底层文件系统的支持;只有一部分Linux文件系统才能提供这种支持。在初始实现中，ext2，ext3，ext4，UDF，Minix和shmem文件系统提供了支持。在Linux 3.15中添加了XFS支持。

 O\_TRUNC，如果文件已经存在并且是常规文件，并且访问模式允许写入（即是O\_RDWR或O\_WRONLY），它将被截断为长度0.如果文件是FIFO或终端设备文件，则O\_TRUNC标志将被忽略。否则，O\_TRUNC的效果是未指定的。

**参数mode**  指定在创建新文件的情况下使用的模式。 必须在flags为O\_CREAT或O\_TMPFILE时提供此参数;当flags为其他值时则忽略mode参数。 有效模式以通常的方式被进程的umask修改：在没有默认ACL的情况下，创建的文件的模式是（mode＆〜umask）。 请注意，此模式仅适用于新创建的文件的将来访问; 创建只读文件的open（）调用可能会返回一个读/写文件描述符。

**可选项有：**

S\_IRWXU 00700 权限，代表该文件所有者具有可读、可写及可执行的权限。

S\_IRUSR 或 S\_IREAD，00400 权限，代表该文件所有者具有可读取的权限。

S\_IWUSR 或 S\_IWRITE，00200 权限，代表该文件所有者具有可写入的权限。

S\_IXUSR 或 S\_IEXEC，00100 权限，代表该文件所有者具有可执行的权限。

S\_IRWXG 00070 权限，代表该文件用户组具有可读、可写及可执行的权限。

S\_IRGRP 00040 权限，代表该文件用户组具有可读的权限。

S\_IWGRP 00020 权限，代表该文件用户组具有可写入的权限。

S\_IXGRP 00010 权限，代表该文件用户组具有可执行的权限。

S\_IRWXO 00007 权限，代表其他用户具有可读、可写及可执行的权限。

S\_IROTH 00004 权限，代表其他用户具有可读的权限

S\_IWOTH 00002 权限，代表其他用户具有可写入的权限。

S\_IXOTH 00001 权限，代表其他用户具有可执行的权限。

S\_ISUID 04000权限，任何用户执行具有SUID权限和x权限的文件时拥有此文件所有者的权限，此文件执行时其euid会变成文件所有者的UID。如果对目录设置了SUID权限，则所有用户在该目录下创建的文件都属于该目录所有者，而不是创建者。

S\_ISGID 02000权限，任何用户组执行具有SGID权限和x权限的文件时拥有此文件所在组的权限，此文件执行时其egid会变成文件所在组的GID。如果对目录设置了SGID权限，则所有用户在该目录下创建的文件都属于该目录所在组，而不是创建者所在组。

S\_ISVTX 01000权限，只对目录有效，使任何用户都可以在该目录下创建文件，但只有文件所有者和root才能删除，即是其他人对目录有写权限。

但是通常采用直接赋数值的形式,如：

int fd = open(“1.txt”,O\_WRONLY | O\_CREAT,0755); //表示给 755 的权限

if(-1 == fd)

{

perror("open failed!\n");

exit(-1);

}

注意：LINUX 中基于文件描述符的 open 函数，对于一个不存在的文件，不能通过 O\_WRONLY 的方式 打开，必须加上 O\_CREAT 选项。

close 用于文件的关闭：

int close(int fd);//fd 表示文件描述词,是先前由 open 或 creat 创建文件时的返回值。

文件使用完毕后，应该调用 close 关闭它，一旦调用 close，则该进程对文件所加的锁全都被释放， 并且使文件的打开引用计数减 1，只有文件的打开引用计数变为 0 以后，文件才会被真正的关闭。成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看。

### 2.3 读写文件

读写文件的函数原型为：

#include <unistd.h>

ssize\_t read(int fd, void \*buf, size\_t nbyte);//文件描述词 缓冲区 长度

ssize\_t pread(int fd, void \*buf, size\_t nbyte, off\_t offset);

ssize\_t write(int fd, const void \*buf, size\_t nbyte);

ssize\_t pwrite(int fd, const void \*buf, size\_t nbyte, off\_t offset);

read 、write与pread、pwrite都是从fd中读取nbyte个字节的数据，区别在于pread与pwrite需要指明读写标记位置。相当于先调用lseek再调用read或者write。

成功返回读取或写入的字节数，，到达文件尾返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看。

举例：将 aaa.txt 中的内容复制到 bbb.txt 中，其中 bbb.txt 起初不存在。

#include <stdio.h>

#include <stdlib.h> //包含 exit

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <errno.h> //用 perror 输出错误

#include <unistd.h>

#define FILENAME1 "./aaa.txt" //用宏定义文件的路径，可以实现一改都改

#define FILENAME2 "./bbb.txt"

main**()**

**{**

char buf**[**512**]** **=** **{**0**};**

int fo1 **=** open**(**FILENAME1**,** O\_RDONLY**);** //fo1,fo2 都是文件描述词

int fo2 **=** creat**(**FILENAME2**,** 0755**);** //创建文件=int fo2 = open(FILENAME2, O\_WRONLY | O\_CREAT);

**if(** **(-**1 **==** fo1**)** **||** **(-**1 **==** fo2**)** **)**

**{**

perror**(**"open failed!\n"**);** //用于输出错误信息.类似于：fputs(”open failed\n”,stderr);

exit**(-**1**);**

**}**

int fr **=** 0**;**

**while(** **(**fr **=** read**(**fo1**,** buf**,** **sizeof(**buf**)))** **>** 0 **)**

//如果 read 读取成功，返回的是长度，否则，返回-1

**{**

int fw **=** write**(**fo2**,** buf**,** fr**);**

**if(** **-**1 **==** fw **)**

**{**

perror**(**"write failed!"**);**

exit**(-**1**);**

**}**

**}**

close**(**fo1**);**

close**(**fo2**);**

**}**

### 2.4 改变文件大小

函数原型：

#include <unistd.h>

int ftruncate(int fd, off\_t length);

函数 ftruncate 会将参数 fd 指定的文件大小改为参数 length 指定的大小。

参数 fd 为已打开的文件描述词，而且必须是以写入模式打开的文件。

如果fd不是一个有效的写入文件描述符，ffluncate函数将失败。

如果fd是普通文件，当length小于文件时则文件截断至length，当length大于文件时增加到length则用‘\0’填充。

成功完成后，如果fd是普通文件，则ftruncate将更新文件的最后一次数据修改和状态更改时间戳，文件模式的S\_ISUID和S\_ISGID位可能会被清除。如果ftruncate函数不成功，则该文件不受影响。

如果请求会导致文件大小超过该进程的软文件大小限制，则请求将失败，并且实现将为该线程生成SIGXFSZ信号。

如果fd是一个目录，则ftruncate将失败。

如果fd是除了普通文件和共享内存之外的其他类型文件，结果是未指定的。

如果fd是共享内存，ftruncate将把共享内存对象的大小设置为length。如果ftruncate的效果是减小内存映射文件或共享内存对象的大小，并且超出新端的整个页面先前被映射，那么超出新端的整个页面将被丢弃。对丢弃的页面的引用将导致产生SIGBUS信号。 如果ftruncate（）的效果是增加内存对象的大小，那么在旧的文件结尾和新的内容之间的任何映射页面的内容是否被刷新到底层对象是不明确的。

返回值：成功则返回 0，出错返回-1。 设置了标准错误描述，可使用perror函数查看。

实例：

int main**()**

**{**

int fd **=** open**(**"a.txt"**,** O\_WRONLY**);**

ftruncate**(**fd**,** 1000**);**

close**(**fd**);**

**return** 0**;**

**}**

### 2.5 文件定位

函数 lseek 将文件指针设定到相对于 whence，偏移值为 offset 的位置

#include <sys/types.h>

#include <unistd.h>

off\_t lseek(int fd, off\_t offset, int whence);//fd 文件描述词

lseek 将文件读写标志位从whence偏移至offset位置处。lseek（）函数将允许将文件偏移设置为超出文件数据的当前结尾。如果在之后写入数据，则中间形成文件空洞。对文件空洞连续读取都是0.

lseek（）函数本身不应扩展文件的大小。

如果fildes引用共享内存对象，则lseek函数的结果是未指定的。

如果fildes是指一个类型化的内存对象，则lseek函数的结果是未指定的。

whence 可以是下面三个常量的一个

SEEK\_SET 从文件头开始计算

SEEK\_CUR 从当前指针开始计算

SEEK\_END 从文件尾开始计算

成功将返回从文件开始以字节计的结果偏移量。 否则返回-1，设置了标准错误描述，可使用perror函数查看。

利用该函数可以实现文件空洞（对一个新建的空文件，可以定位到偏移文件开头 1024 个字节的地 方，在写入一个字符，则相当于给该文件分配了 1025 个字节的空间，形成文件空洞）通常用于多进程间通信的时候的共享内存。lseek(fd, 0,SEEK\_CUR)返回当前文件位置等效于ftell函数。

举例：

int main**()**

**{**

int fd **=** open**(**"c.txt"**,** O\_WRONLY **|** O\_CREAT**);**

lseek**(**fd**,** 1024**,** SEEK\_SET**);**

write**(**fd**,** "a"**,** 1**);**

close**(**fd**);**

**return** 0**;**

**}**

### 2.6 获取文件信息

可以通过 fstat 和 stat 函数获取文件信息，调用完毕后，文件信息被填充到结构体 struct stat 变量中，函数原型为：

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

int stat(const char \*pathname, struct stat \*buf); //获取文件状态

int fstat(int fd, struct stat \*buf);

int lstat(const char \*pathname, struct stat \*buf);

int fstatat(int dirfd, const char \*pathname, struct stat \*buf, int flags);

stat,fstat,lstat,fstatat 都是获取一个文件或目录的详细信息存放在buf中。

stat与fstat的区别在要获取信息的文件由文件描述符fd指定，而不是文件名。

stat与lstat的区别在于当pathname是一个链接时，获取其本身信息而不时其链接的文件的信息。

stat与fstatat的区别在于当pathname是相对路径时，stat是相对当前工作目录而言，而fstatat是相对于文件描述符dirfd而言，如果dirfd设为AT\_FDCWD，则与stat一样；当pathname是绝对路径时，dirfd被忽略。

都是成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看。

结构体 stat 的定义为：

struct stat

{

dev\_t st\_dev; /\*如果是设备，返回设备表述符，否则为 0\*/

ino\_t st\_ino; /\* i 节点号 \*/

mode\_t st\_mode; /\* 文件类型 \*/ 无符号短整型

nlink\_t st\_nlink; /\* 链接数 \*/

uid\_t st\_uid; /\* 属主 ID \*/

gid\_t st\_gid; /\* 组 ID \*/

dev\_t st\_rdev; /\* 设备类型\*/

off\_t st\_size; /\* 文件大小，字节表示 \*/

blksize\_t st\_blksize; /\* 块大小\*/

blkcnt\_t st\_blocks; /\* 块数 \*/

time\_t st\_atime; /\* 最后访问时间\*/

time\_t st\_mtime; /\* 最后修改时间\*/

time\_t st\_ctime; /\* 最后权限修改时间 \*/

};

对于结构体的成员 st\_mode，有一组宏可以进行文件类型的判断 ：

宏 描述

S\_ISLNK(st\_mode) 判断是否是符号链接

S\_ISREG(st\_mode) 判断是否是普通文件

S\_ISDIR(st\_mode) 判断是否是目录

S\_ISCHR(st\_mode) 判断是否是字符型设备

S\_ISBLK(st\_mode) 判断是否是块设备

S\_ISFIFO(st\_mode) 判断是否是命名管道

S\_ISSOCK(st\_mode) 判断是否是套接字

通常用于判断：if(S\_ISDIR(st.st\_mode)){}

举例:获得文件的大小

#include<sys/stat.h>

#include<unistd.h>

main**()**

**{**

struct stat buf**;**

stat **(**“**/**etc**/**passwd”**,&**buf**);**

printf**(**“**/**etc**/**passwd file size **=** **%**d \n”**,**buf**.**st\_size**);**//st\_size 可以得到文件大 小

**}**

如果用 fstat 函数实现，如下：

int fd **=** open **(**“**/**etc**/**passwd”**,**O\_RDONLY**);** //先获得文件描述词

fstat**(**fd**,** **&**buf**);**

实例：

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

#include <time.h>

int main**()**

**{**

int fd **=** open**(**"/home/wangxiao/0926/a.txt"**,** O\_RDONLY**);**

**if(**fd **==** **-**1**)**

**{**

perror**(**"open error"**);**

exit**(-**1**);**

**}**

struct stat buf**;**

int iRet **=** fstat**(**fd**,** **&**buf**);**

**if(**iRet **==** **-**1**)**

**{**

perror**(**"fstat error"**);**

exit**(-**1**);**

**}**

**if(**S\_ISREG**(**buf**.**st\_mode**))**

**{**

printf**(**"regular file!\n"**);**

**}**

**if(**S\_ISDIR**(**buf**.**st\_mode**))**

**{**

printf**(**"directory!\n"**);**

**}**

**if(**S\_ISLNK**(**buf**.**st\_mode**))**

**{**

printf**(**"link file!\n"**);**

**}**

printf**(**"the size of file is : %d\n"**,** buf**.**st\_size**);**

time\_t tt **=** buf**.**st\_atime**;**

struct tm **\***pT **=** gmtime**(&**tt**);** // 格林尼治时间

printf**(**"%4d-%02d-%02d %02d:%02d:%02d\n"**,** **(**1900**+**pT**->**tm\_year**),** **(**1**+**pT**->**tm\_mon**),** pT**->**tm\_mday**,** **(**8**+**pT**->**tm\_hour**),** pT**->**tm\_min**,** pT**->**tm\_sec**);** // printf("the last access time is : %d\n", buf.st\_atime);

close**(**fd**);**

**return** 0**;**

**}**

### 2.7 文件描述符的复制

原型为：

#include <unistd.h> //头文件包含

int dup(int oldfd);

int dup2(int oldfd, int newfd);

系统调用函数dup 和 dup2可以实现文件描述符的复制，经常用来重定向进程的stdin或者0, stdout或者1，stderr或者2。

dup 返回新的文件描述符（没有使用的文件描述符的最小编号）。这个新的描述符是旧文件描述符的拷贝。这意味着两个描述符共享同一个数据结构。等效于fcntl(fildes, F\_DUPFD, 0);

dup2 允许调用者用一个有效描述符(oldfd)和目标描述符(newfd)，函数成功返回时，目标描述符将变成旧描述符的复制品，此时两个文件描述符现在都指向同一个文件，并且是函数第一个参数（也就是 oldfd）指向的文件。除非oldfd等于newfd，否则如果newfd应该先关闭，如果不能关闭则返回-1，如果oldfd大于OPEN\_MAX或者不是一个有效的文件描述符，返回-1.

成功返回描述符，出错返回-1，设置了标准错误描述，可使用perror函数查看。

文件描述符的复制是指用另外一个文件描述符指向同一个打开的文件，它完全不同于直接给文件描述符变量赋值，例如：

**描述符变量的直接赋值**：

char szBuf**[**32**];**

int fd**=**open**(**“**./**a**.**txt”**,**O\_RDONLY**);**

int fd2**=**fd**;** //类似于 C 语言的指针赋值，当释放掉一个得时候，另一个已经不能操作了 close(fd); //导致文件立即关闭

printf**(**“read**:%**d\n”**,**read**(**fd2**,**szBuf**,sizeof(**szBuf**)-**1**));** //读取失败

close**(**fd2**);** //无意义

在此情况下，两个文件描述符变量的值相同，指向同一个打开的文件，但是内核的文件打开引用计数还是为 1，所以 close(fd)或者 close(fd2)都会导致文件立即关闭掉。

**描述符的复制**：

char szBuf**[**32**];**

int fd**=**open**(**“**./**a**.**txt”**,**O\_RDONLY**);**

int fd2**=**dup**(**fd**);** //内核的文件打开引用计算+1，变成 2 了

close**(**fd**);** //当前还不会导致文件被关闭，此时通过 fd2 照样可以访问文件

printf**(**“read**:%**d\n”**,**read**(**fd2**,**szBuf**,sizeof(**szBuf**)-**1**));**

close**(**fd2**);** //内核的引用计数变为 0，文件正式关闭

举例：

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int main**(**int argc**,**char **\***argv**[])**

**{**

char szBuf**[**32**]={**0**};**

int fda**=**open**(**"./a.txt"**,**O\_RDWR**);** //假设 a.txt 的内容为：hello world

int fdaa**=**dup**(**fda**);**

read**(**fda**,**szBuf**,**4**);**

puts**(**szBuf**);** //关闭之前先输出原来的内容

close**(**fda**);**

//lseek(fdaa, 0, SEEK\_SET);

read**(**fdaa**,**szBuf**,sizeof(**szBuf**));**

puts**(**szBuf**);** //输出现在的内容

close**(**fdaa**);**

**}**

解析：假设 a.txt 中的内容为：hello world.上面的例子会发现第一次输出的结果是 hell。关闭 close(fda)的时候，文件实际上还没有真正的关闭，此时文件指针已经向后移动了。执行第二次 read命令将 o world 读出来，最后关闭 fdaa。只有当文件的引用计数为0时才会真正的关闭。

dup 有时会用在一些特定的场合，如下：

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <stdlib.h>

int main**()**

**{**

int fd **=** open**(**"a.txt"**,** O\_WRONLY **|** O\_CREAT**);**

**if(**fd **==** **-**1**)**

**{**

perror**(**"open error"**);**

exit**(-**1**);**

**}**

printf**(**"\n"**);** /\* 必不可少 \*/

close**(**1**);**

int fd2 **=** dup**(**fd**);**

close**(**fd**);**

printf**(**"hello world\n"**);**

close**(**fd2**);**

**return** 0**;**

**}**

该程序首先打开了一个文件，返回一个文件描述符，因为默认的就打开了 0,1,2 表示标准输入，标准输出，标准错误输出。而用 close(1);则表示关闭标准输出，此时这个文件描述符就空着了。后 面又用 dup，此时 dup(fd);则会复制一个文件描述符到当前未打开的最小描述符，此时这个描述符 为 1.后面关闭 fd 自身，然后在用标准输出的时候，发现标准输出重定向到你指定的文件了。那么 printf 所输出的内容也就直接输出到文件了。

dup2(int fdold,int fdnew)也是进行描述符的复制，只不过采用此种复制，新的描述符由用户用 参数 fdnew 显示指定，而不是象 dup 一样由内核帮你选定（内核选定的是随机的）。对于 dup2，如 果 fdnew 已经指向一个已经打开的文件，内核会首先关闭掉 fdnew 所指向的原来的文件。此时再针 对于 fdnew 文件描述符操作的文件，则采用的是 fdold 的文件描述符。如果成功 dup2 的返回值于 fdnew 相同,否则为-1.

思考下面程序的结果：

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int main**(**int argc**,**char **\***argv**[])**

**{**

char szBuf**[**32**]={**0**};**

int fda**=**open**(**"./a.txt"**,**O\_RDONLY**);**

int fdb**=**open**(**"./b.txt"**,**O\_RDONLY**);**

int fdbb**=**dup**(**fdb**);**

int fda2**=**dup2**(**fda**,**fdb**);** //可以设定为：int fda2 = dup2(fda,5);即自己设为 5

printf**(**"fda:%d fdb:%d fdbb:%d fda2:%d"**,**fda**,**fdb**,**fdbb**,**fda2**);**

read**(**fdb**,**szBuf**,sizeof(**szBuf**)-**1**);** //此时 fdb 已经不再定位 b.txt 而是 a.txt

printf**(**"result:%s\n"**,**szBuf**);**

close**(**fda**);**

close**(**fdb**);**

close**(**fdbb**);**

close**(**fda2**);**

**}**

### 2.8 标准输入输出文件描述符

与标准的输入输出流对应，在更底层的实现是用标准输入、标准输出、标准错误文件描述符表示的。 它们分别用 STDIN\_FILENO、STDOUT\_FILENO 和 STDERR\_FILENO 三个宏表示，值分别是 0、1、2 三个 整型数字。

标准输入文件描述符  STDIN\_FILENO  0

标准输出文件描述符  STDOUT\_FILENO  1

标准错误输出文件描述符  STDERR\_FILENO  2

#include <unistd.h>

int fsync(int fd);

fsync同步缓冲区到磁盘, 成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看。

示例：

#include <stdio.h>

#include <unistd.h>

#include <string.h>

int main**()**

**{**

char szBuf**[**32**],**szBuf2**[**50**];**

printf**(**"Input string:"**);**

fflush**(**stdout**);** //要刷新标准输出流，才可以立即在屏幕上显示”Input string” //fflush 用于 linux 中的时候，只对 fflush(stdout)有效。

int iRet**=**read**(**STDIN\_FILENO**,**szBuf**,sizeof(**szBuf**));**

szBuf**[**iRet**]=**0**;** //read是以无类型指针方式读的数据，不会自动在缓冲区后加0结束标记

sprintf**(**szBuf2**,**"The string is:%s"**,**szBuf**);**

write**(**STDOUT\_FILENO**,**szBuf2**,**strlen**(**szBuf2**));**

**return** 0**;**

**}**

### 2.9 I/O 多路复用模型select

命名管道（FIFO）与匿名管道(pipe)

进程间通信必须通过内核提供的通道，而且必须有一种办法在进程中标识内核提供的某个通道，前面讲过的匿名管道是用打开的文件描述符来标识的。匿名管道是不可见。如果要互相通信的几个进程没有从公共祖先那里继承文件描述符，它们怎么通信呢？内核提供一条通道不成问题，问题是如何标识这条通道才能使各进程都可以访问它？文件系统中的路径名是全局的，各进程都可以访问，因此可以用文件系统中的路径名来标识一个IPC通道。

FIFO和UNIX Domain Socket这两种IPC机制都是利用文件系统中的特殊文件来标识的。

FIFO文件在磁盘上没有数据块，仅用来标识内核中的一条通道，如

prw-rw-r-- 1 simba simba 0 May 21 10:13 p2，

文件类型标识为p表示FIFO，文件大小为0。各进程可以打开这个文件进行read/write，实际上是在读写内核通道（根本原因在于这个file结构体所指向的read、write函数和常规文件不一样），这样就实现了进程间通信。UNIX Domain Socket和FIFO的原理类似，也需要一个特殊的socket文件来标识内核中的通道，例如/run目录下有很多系统服务的socket文件：

srw-rw-rw- 1 root root 0 May 21 09:59 acpid.socket

文件类型s表示socket，这些文件在磁盘上也没有数据块。

匿名管道应用的一个限制就是只能在具有共同祖先（具有亲缘关系）的进程间通信。如果我们想在不相关的进程之间交换数据，可以使用FIFO文件来做这项工作，它经常被称为命名管道。匿名管道由pipe函数创建并打开。

命名管道可以从命令行上创建：

$ mkfifo filename

或者由mkfifo函数创建。

#include<sys/types.h>

#include<sys/stat.h>

int mkfifo(const char\*pathname,mode\_t mode);

参数 pathname 为要创建的FIFO 文件的全路径名；

参数 mode 为文件访问权限

函数成功返回0，失败时返回-1. 设置了标准错误描述，可使用perror函数查看。

创建一个可见的管道文件，pathname为文件路径，mode为该FIFO文件的读写权限，之后就可以打开这个文件来进行读写操作。对于命名管道，默认情况下，除非写入方主动打开管道的读取端，否则读取端是无法打开命名管道的。open调用执行后，读取方将被锁住，直到写入方出现为止。（即读端阻塞，写端阻塞）

使用open打开命名管道时只能以只读O\_RDONLY或只写O\_WRONLY方式打开，不能以读写方式O\_RDWR方式打开，因为命名管道是单向（半双工，同一时刻只能向一个方向通信）。如果以读写方式打开，结果将不可预期。要进行双向通信只能采取：

a）创建两个命名管道，分别以不同方向打开；：

b）创建一个命名管道，先使用一个方向进行open，传递数据后，关闭，再从反方向打开，进行反方向数据传输。

FIFO（命名管道）与pipe（匿名管道）之间唯一的区别在它们创建与打开的方式不同，这些工作完成之后，它们具有相同的语义。

//写进程

#include <sys/types.h>

#include <sys/stat.h>

#include <errno.h>

#include <fcntl.h>

#include <string.h>

#include <stdio.h>

#define FIFO\_NAME "/tmp/myfifo"

main**()**

**{**

int fd**;**

char w\_buf**[**50**];**

int w\_num**;**

// 若fifo已存在，则直接使用，否则创建它

**if((**mkfifo**(**FIFO\_NAME**,**0777**)<**0**)&&(**errno**!=**EEXIST**))**

**{**

printf**(**"cannot create fifo...\n"**);**

exit**(**1**);**

**}**

//以阻塞型只写方式打开fifo

fd**=**open**(**FIFO\_NAME**,**O\_WRONLY**);**

w\_num**=**write**(**fd**,**"abcdg\0"**,**6**);**

printf**(**"%d\n"**,**w\_num**);**

**}**

//读进程

#include <sys/types.h>

#include <sys/stat.h>

#include <errno.h>

#include <fcntl.h>

#define FIFO\_NAME "/tmp/myfifo"

main**()**

**{**

char r\_buf**[**50**];**

int fd**;**

int r\_num**;**

// 若fifo已存在，则直接使用，否则创建它

**if((**mkfifo**(**FIFO\_NAME**,**0777**)<**0**)&&(**errno**!=**EEXIST**))**

**{**

printf**(**"cannot create fifo...\n"**);**

exit**(**1**);**

**}**

//以阻塞型只读方式打开fifo

fd**=**open**(**FIFO\_NAME**,**O\_RDONLY**);**

**if(**fd**==-**1**)**

**{**

printf**(**"open %s for read error\n"**);**

exit**(**1**);**

**}**

// 通过键盘输入字符串，再将其写入fifo，直到输入"exit"为止

r\_num**=**read**(**fd**,**r\_buf**,**6**);**

printf**(**" %d bytes read:%s\n"**,**r\_num**,**r\_buf**);**

unlink**(**FIFO\_NAME**);**//删除fifo

**}**

1）写进程阻塞，读进程阻塞。（默认情况）

先运行写进程（被阻塞），再运行读进程，一切正常。

先运行读进程（被阻塞），再运行写进程，一切正常。

2）写进程阻塞，读进程非阻塞。

就改一句代码 fd=open(FIFO\_NAME,O\_RDONLY | O\_NONBLOCK)，下面类似。

先运行写进程（被阻塞），再运行读进程，一切正常。

先运行读进程，直接崩掉，想想也挺自然的，没东西你还要读，而且不愿等。

3）写进程非阻塞，读进程阻塞。

先运行写进程，open调用将返回-1，打开失败。

先运行读进程（被阻塞），再运行写进程，一切正常。

4）写进程非阻塞，读进程非阻塞。

先运行读进程，想想也挺自然的，没东西你还要读，而且不愿等。

先运行写进程，open调用将返回-1，打开失败。

我们在/tmp目录下通过ls -la命令可以看到管道文件myfifo的大小总是0，这是因为虽然FIFO文件存在于文件系统中，但FIFO中的内容都存放在内存中，所以文件大小始终为0。

在这种模型下，如果请求的 I/O 操作阻塞，且它不是真正阻塞 I/O， 而是让其中的一个函数等待，在这期间，I/O 还能进行其他操作。如本节要介绍的 select()函数，就是属于这种模型。

**I/O多路复用Select 函数：**

#include<sys/select.h>

#include<sys/time.h>

int pselect(int maxfds, fd\_set \*readset, fd\_set \*writeset, fd\_set \*exceptionset,const struct timespec \*restrict timeout, const sigset\_t \*restrict sigmask);

int select(int maxfd, fd\_set\*readset,fd\_set \*writeset, fd\_set\*exceptionset, const struct timeval\*timeout);

返回:已处于就绪状态的描述符的数目（或者叫做状态改变了的描述字的数目），0—超时，-1—出错 。设置了标准错误描述，可使用perror函数查看。

Select可以完成非阻塞（所谓非阻塞方式non-block，就是进程或线程执行此函数时不必非要等待事件的发生，一旦执行肯定返回，以返回值的不同来反映函数的执行情况，如果事件发生则与阻塞方式相同，若事件没有发生则返回一个代码来告知事件未发生，而进程或线程继续执行，所以效率较高）方式工作的程序，它能够监视我们需要监视的文件描述符的变化情况——读写或是异常。

参数解释：

maxfd：最大的文件描述符（其值应该为最大的文件描述符字 +1）

readset：内核读操作的描述符字集合

writeset：内核写操作的描述符字集合

exceptionset：内核异常操作的描述符字集合

timeout：等待描述符就绪需要多少时间。NULL 代表永远等下去，一个固定值代表等待固定时间，0代表根本不等待，检查描述字之后立即返回。

其中 readset、writeset、exceptionset 都是 fd\_set 集合。该集合的相关操作 如下：

void FD\_ZERO(fd\_set\*fdset); /\* 将所有 fd 清零 \*/

void FD\_SET(int fd, fd\_set\*fdset); /\* 增加一个 fd\*/

void FD\_CLR(int fd,fd\_set \*fdset); /\* 删除一个 fd\*/

int FD\_ISSET(int fd, fd\_set\*fdset); /\* 判断一个 fd 是否有设置 \*/

一般来说，在使用 select 函数之前，首先要使用FD\_ZERO和FD\_SET来初始化文件描述符集，在使 用select函数时，可循环使用 FD\_ISSET 测试描述符集，在执行完对相关文件描述符操作后之后，使用 FD\_CLR 来清除描述符集。/每次循环都要清空集合,并重新注册事件，否则不能检测描述符变化

另外，select 函数中的 timeout 是一个 struct timeval 类型的指针，该结构体如下：

struct timeval

{

long tv\_sec; /\* second \*/ //秒 l

long tv\_usec; /\* microsecond \*/ //微秒

};

struct timespec

{

\_\_time\_t tv\_sec; /\*seconds 秒\*/

long int tv\_nsec; /\*nanoseconds 纳秒\*/

}

pselect与select功能一致，区别在于

（1）select函数用的timeout参数，是一个timeval的结构体（包含秒和微秒），然而pselect用的是一个timespec结构体（包含秒和纳秒）

（2）select函数可能会为了指示还剩多长时间而更新timeout参数，然而pselect不会改变timeout参数

（3）select函数没有sigmask参数，当pselect的sigmask参数为null时，两者行为时一致的。有sigmask的时候，pselect相当于如下的select()函数，在进入select()函数之前手动将信号的掩码改变，并保存之前的掩码值；select()函数执行之后，再恢复为之前的信号掩码值。

sigset\_t origmask;

sigprocmask(SIG\_SETMASK, &sigmask, &origmask);

select(nfds, &readfds, &writefds, &exceptfds, timeout);

sigprocmask(SIG\_SETMASK, &origmask, NULL);

举例:多路转接模型 select

#include <sys/select.h>

#include <sys/time.h>

#include <sys/types.h>

#include <unistd.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#define FILENAME1 "a.txt"

#define FILENAME2 "dir.c"

int main**()**

**{**

char buf**[**10**]** **=** **{**0**};**

int fd1 **=** open**(**FILENAME1**,**O\_RDWR**);**

int fd2 **=** open**(**FILENAME2**,**O\_RDWR**);**

int fd3 **=** open**(**FILENAME1**,**O\_RDWR**);**

int fd4 **=** open**(**FILENAME2**,**O\_RDWR**);**

**if(** **(-**1 **==** fd1**)** **||** **(-**1 **==** fd2**)** **||** **(-**1 **==** fd3**)** **||** **(-**1 **==** fd4**)** **)**

**{**

perror**(**"open"**);**

exit**(-**1**);**

**}**

fd\_set fdrd**,**fdwr**;** //绑定读写集合

FD\_ZERO**(&**fdrd**);** //清除以前读的绑定

FD\_ZERO**(&**fdwr**);** //清除以前写的绑定

FD\_SET**(**fd1**,&**fdrd**);** //将 fd1 与读绑定

FD\_SET**(**fd2**,&**fdrd**);**

FD\_SET**(**fd3**,&**fdwr**);** //将 fd3 与写绑定

FD\_SET**(**fd4**,&**fdwr**);**

int max1 **=** fd1 **>** fd2 **?** fd1 **:** fd2**;** //获取读绑定中的文件描述词最大值

int max2 **=** fd3 **>** fd4 **?** fd3 **:** fd4**;** //获取写绑定中的文件描述词最大值

int max **=** max1 **>** max2 **?** max1 **:** max2**;** //获得读写文件描述词最大值

struct timeval tv**;** //用于记录时间，表示过这么长时间不响应就退出

tv**.**tv\_sec **=** 2**;** //秒

tv**.**tv\_usec **=** 0**;** //微妙

**while(**1**)**

**{**

**if(** select**(**max**+**1**,** **&**fdrd**,** **&**fdwr**,** **NULL,** **&**tv**)** **==** **-**1 **)** //从 1—max+1 查找

**{**

perror**(**"select"**);**

**break;**

**}**

**if(** FD\_ISSET**(**fd1**,&**fdrd**)** **)** //如果 fd1 设置的是读绑定

**{**

read**(**fd1**,**buf**,sizeof(**buf**)-**1**);**

puts**(**buf**);**

sleep**(**1**);**

**}**

**if(** FD\_ISSET**(**fd2**,&**fdrd**)** **)**

**{**

read**(**fd2**,**buf**,sizeof(**buf**)-**1**);**

puts**(**buf**);**

sleep**(**1**);**

**}**

**if(** FD\_ISSET**(**fd3**,&**fdwr**)** **)** //如果 fd3 设置的是写绑定

**{**

write**(**fd3**,**buf**,sizeof(**buf**));**

sleep**(**2**);**

**}**

**if(** FD\_ISSET**(**fd4**,&**fdwr**)** **)**

**{**

write**(**fd4**,**buf**,sizeof(**buf**));**

sleep**(**2**);**

**}**

**}**

close**(**fd1**);**

close**(**fd2**);**

close**(**fd3**);**

close**(**fd4**);**

**}**

样例存在问题

### **2.10 日志**

许多应用程序需要记录它们的活动，系统程序经常需要向控制台或日志文件写消息。这些消息 可能指示错误、警告或者与系统状态有关的一般信息。通常是在/var/log 目录下的 messages 中包含了系统信息。通过 syslog 可以向系统的日志发送日志信息。

函数原型如下：

#include <syslog.h>

void syslog(int priority, const char \*format, ...);

syslog 向系统的日志发送日志信息。

参数 priority 用来设定事件级别，有如下几个常见的：

LOG\_EMERG 紧急情况

LOG\_ALERT 高优先级故障（如：数据库崩溃）

LOG\_CRIT 严重错误（如：硬件错误）

LOG\_ERR 错误

LOG\_WARNING 警告

LOG\_NOTICE 需要注意的特殊情况

LOG\_INFO 一般信息

LOG\_DEBUG 调试信息(写不到 messages 里面)

#include <syslog.h>

main**()**

**{**

openlog**(**"log"**,**LOG\_PID**|**LOG\_CONS**|**LOG\_NOWAIT**,**LOG\_USER**);**

syslog**(**LOG\_ALERT**,**"this is alert\n"**);**

syslog**(**LOG\_INFO**,**"this is info\n"**);**

syslog**(**LOG\_DEBUG**,**"this is debug%d\t %s"**,**10**,**"aaaa"**);**

syslog**(**LOG\_ERR**,**"err"**);**

syslog**(**LOG\_CRIT**,**"crit"**);**

closelog**();**

**}**

利用 tail -10 /var/log/messages 可以查看。

还可以通过函数 openlog 函数来改变日志信息的表达方式。openlog 的原型如下：

#include <syslog.h>

void openlog(const char\* ident, int logopt, int facility);

void closelog(void);

调用openlog是可选择的。如果不调用openlog，则在第一次调用syslog时，自动调用openlog。调用closelog也是可选择的，它只是关闭被用于与syslog守护进程通信的描述符。

参数ident，该参数常用来表示信息来源。ident指向的字符串会被固定地添加在每行日志的前面。

参数logopt用于指定openlog函数和接下来调用的syslog函数的控制标志。可以单独取其中的某值，也可以通过或运算来获得多种特性。

LOG\_PID 在日志信息中包含进程标识符，这是系统分配给每个进程的一个唯一值

LOG\_CONS 直接写入系统控制台，如果有一个错误，同时发送到系统日志记录。

LOG\_NDELAY 立即打开连接（通常，打开连接时记录的第一条消息）。

LOG\_NOWAIT 不要等待子进程，因为其有可能在记录消息的时候就被创建了（GNU C库不创建子进程，所以该选项在Linux上没有影响。）

LOG\_ODELAY 延迟连接的打开直到syslog函数调用。（这是默认情况下，需要没被指定的情况下。）

LOG\_PERROR （不在SUSv3情况下）同时输出到stderr（标准错误文件）。

参数facility，这个要与syslogd守护进程的配置文件对应，日志信息会写入syslog.conf文件指定的位置，facility参数是用来指定记录消息程序的类型。它根据指定的配置文件，以不同的方式来处理来自不同方式的消息。它的值可能为 LOG\_KERN、LOG\_USER、LOG\_MAIL、LOG\_DAEMON、LOG\_AUTH、LOG\_SYSLOG、LOG\_LPR、LOG\_NEWS、LOG\_UUCP、LOG\_CRON 或 LOG\_AUTHPRIV

### 2.11 MMAP 文件映射

#include<sys/mman.h.>

void \*mmap(void \*startaddr,size\_t length,int protect,int flags,int fd,off\_t offset)

int munmap(void\* startaddr,size\_t length);

mmap将一个文件或者其它对象映射进内存。文件被映射到多个页上，如果文件的大小不是所有页的大小之和，最后一个页不被使用的空间将会置零。mmap在用户空间映射调用系统中作用很大。

munmap执行相反的操作，删除特定地址区域的对象映射.msync同步内存和磁盘的文件变化

成功执行时，mmap()返回被映射区的指针，munmap()返回0。失败时，mmap()返回MAP\_FAILED[其值为(void \*)-1]，munmap返回-1. 设置了标准错误描述，可使用perror函数查看。

startaddr：映射区的开始地址，设置为0时表示由系统决定映射区的起始地址。munmap中startaddr必须是一开始指定的地址而不能是偏移后的地址。这和free()一样。

length：映射区的长度。//长度单位是 以字节为单位，不足一内存页按一内存页处理，mmap()必须以PAGE\_SIZE为单位进行映射，而内存也只能以页为单位进行映射，若要映射非PAGE\_SIZE整数倍的地址范围，要先进行内存对齐，强行以PAGE\_SIZE的倍数大小进行映射。

protect：期望的内存保护标志，不能与文件的打开模式冲突。是以下的某个值，可以通过or运算合理地组合在一起：

PROT\_EXEC //页内容可以被执行

PROT\_READ //页内容可以被读取

PROT\_WRITE //页可以被写入

PROT\_NONE //页不可访问

flags：指定映射对象的类型，映射选项和映射页是否可以共享。它的值可以是一个或者多个以下位的组合体：

MAP\_FIXED //使用指定的映射起始地址，如果由start和len参数指定的内存区重叠于现存的映射空间，重叠部分将会被丢弃。如果指定的起始地址不可用，操作将会失败。并且起始地址必须落在页的边界上。

MAP\_SHARED //与其它所有映射这个对象的进程共享映射空间。对共享区的写入，相当于输出到文件。在msync()或者munmap()被调用之前，文件实际上不会被更新。

MAP\_PRIVATE //建立一个写入时拷贝的私有映射,对其它所有映射这个对象的进程不可见。内存区域的写入不会影响到原文件。这个标志和以上标志是互斥的，只能使用其中一个。

MAP\_DENYWRITE //这个标志被忽略。

MAP\_EXECUTABLE //同上

MAP\_NORESERVE //不要为这个映射保留交换空间。当交换空间被保留，对映射区修改的可能会得到保证。当交换空间不被保留，同时内存不足，对映射区的修改会引起段违例信号。

MAP\_LOCKED //锁定映射区的页面，从而防止页面被交换出内存。

MAP\_GROWSDOWN //用于堆栈，告诉内核VM系统，映射区可以向下扩展。

MAP\_ANONYMOUS //匿名映射，映射区不与任何文件关联。

MAP\_ANON //MAP\_ANONYMOUS的别称，不再被使用。

MAP\_FILE //兼容标志，被忽略。

MAP\_32BIT //将映射区放在进程地址空间的低2GB，MAP\_FIXED指定时会被忽略。当前这个标志只在x86-64平台上得到支持。

MAP\_POPULATE //为文件映射通过预读的方式准备好页表。随后对映射区的访问不会被页违例阻塞。

MAP\_NONBLOCK //仅和MAP\_POPULATE一起使用时才有意义。不执行预读，只为已存在于内存中的页面建立页表入口。

fd：有效的文件描述词。一般是由open()函数返回，其值也可以设置为-1，此时需要指定flags参数中的MAP\_ANON,表明进行的是匿名映射。

off\_t offset：被映射对象内容的起点。一般为0

向文件写内容，必须是文件有大小的。

**刷新变化函数msync()**

进程在映射空间的对共享内容的改变并不直接写回到磁盘文件中，往往在调用[munmap](http://baike.baidu.com/item/munmap)()后才执行该操作。可以通过调用msync()函数来实现磁盘文件内容与共享内存区中的内容一致,即同步操作.

#include<sys/mman.h>

int msync ( void \* addr, size\_t len, int flags)

addr：文件映射到进程空间的地址；

len：映射空间的大小；

flags：刷新的参数设置，可以取值MS\_ASYNC/ MS\_SYNC/ MS\_INVALIDATE

其中：

取值为MS\_ASYNC（异步）时，调用会立即返回，不等到更新的完成；

取值为MS\_SYNC（同步）时，调用会等到更新完成之后返回；

取MS\_INVALIDATE（通知使用该共享区域的进程，数据已经改变）时，在共享内容更改之后，使得文件的其他映射失效，从而使得共享该文件的其他进程去重新获取最新值；

成功则返回0；失败则返回-1；

### 2.12 内存操作

#include <strings.h>

void bzero(void \*s, size\_t n);

bzero将buf的n个字节置为’\0’。

#include <string.h>

void \*memset(void \*s, int c, size\_t n);

memset将s的n个字节置为c字符，返回指向s的指针。

void \*memchr(const void \*s, int c, size\_t n);

void \*memrchr(const void \*s, int c, size\_t n);

void \*rawmemchr(const void \*s, int c);

memchr在内存s的n个字节中顺序查找字符c第一出线的位置。

memrchr在内存s的n个字节中逆序查找字符c第一次出现的位置

rawmemchr如果知道字符c一定会在s中出现，则采用优化搜索查找s出现的位置。

成功返回位置指针，memchr与memrchr没找到返回NULL，rawmemchr没找到行为未定义。

int memcmp(const void \*s1, const void \*s2, size\_t n);

memcmp比较内存s1与s2的前n个字节。

s1大于s2 返回正数，小于返回负数，等于返回0.当n为0时也返回0.

void \*memcpy(void \*dest, const void \*src, size\_t n);

memcpy将内存src的n个字节拷贝到dest中。返回指向目标内存的指针dest。

void \*memmove(void \*dest, const void \*src, size\_t n);

memmove将内存str的n个字节移动到dest。返回指向目标内存的指针dest。

#include <string.h>

char \*strcat(char \*dest, const char \*src);

char \*strncat(char \*dest, const char \*src, size\_t n);

strcat将原字符串src链接到目标字符串dest上，忽略目标字符串的结束字符‘\0’。

strncat将原字符串src的前n字符链接到目标字符串dest上，忽略目标字符串的结束字符‘\0’。

成功返回指向dest的指针。

char \*strchr(const char \*s, int c);

char \*strrchr(const char \*s, int c);

char \*strchrnul(const char \*s, int c);

在字符串s中查找字符c首次出现的位置，strchr是顺序查找，strrchr是逆序查找，strchrnul与strchr的区别在于没有找到c时，strchr返回NULL，而strchrnul返回指向结束字符’\0’的指针。

成功返回字符c出现的位置，未找到时strchr，strrchr返回NULL，strchrnul返回指向结束字符’\0’的指针。

char \*strcpy(char \*dest, const char \*src);

char \*strncpy(char \*dest, const char \*src, size\_t n);

strcpy将字符串src拷贝到dest中。返回指向目标内存的指针dest。

strcpy将字符串src的n个字节拷贝到dest中。返回指向目标内存的指针dest。

int strcmp(const char \*s1, const char \*s2);

int strncmp(const char \*s1, const char \*s2, size\_t n);

int strcasecmp(const char \*s1, const char \*s2);

int strncasecmp(const char \*s1, const char \*s2, size\_t n);

strcmp比较字符串s1与s2 ，strcasecmp忽略大小写,

strncmp比较字符串s1与s2的前n个字节。strncasecmp忽略大小写,

s1大于s2 返回正数，小于返回负数，等于返回0.当n为0时也返回0.

size\_t strlen(const char \*s);

strlen返回字符串s的长度，不包括终止字符’\0’。

char \*strpbrk(const char \*str1, const char \*str2);

函数返回一个指针，它指向字符串str2中任意字符在字符串str1 首次出现的位置，如果不存在返回NULL。

size\_t strspn( const char \*str1, const char \*str2 );

函数返回字符串str1中第一个在不在字符串str2的出现的字符的下标。也就是str1连续有多少个字符属于str2

char \*strstr( const char \*str1, const char \*str2 );

char \*strcasestr(const char \*haystack, const char \*needle);

strstr函数返回一个指针，它指向字符串str2 首次出现于字符串str1中的位置，如果没有找到，返回NULL。strcasestr与strstr一样只是忽略大小写。

char \*strtok(char \*str, const char \*delim);

char \*strtok\_r(char \*str, const char \*delim, char \*\*saveptr);

strtok将字符串str以分隔符delim分成片段，并在每次调用后返回一个每个子串的首地址。

函数功能：切割字符串，将str切分成一个个子串

参数str：在第一次被调用的时间str是传入需要被切割字符串的首地址；在后面调用的时间传入NULL。

参数delimiters：表示切割字符串（字符串中每个字符都会 当作分割符）。

函数返回值：当str中的字符查找到末尾时，返回NULL;如果查不到delimiter所标示的字符，则返回当前strtok的字符串的指针。

char \*strsep(char \*\*stringp, const char \*delim);

函数功能：切割字符串，将\*stringp切分成一个个子串

参数\*stringp为要分解的字符串，delim为分隔符字符串。

返回值：返回每个子串的起始地址，当没有分割的子串时返回NULL。当连续的分隔符时返回指针所指内容为空

相同点：strtok与strsep两者都会改变源字符串，想要避免，可以使用strdupa（由allocate函数实现）或strdup（由malloc函数实现）。

不同点：strtok第一次调用就会替换所有的分隔符，而strsep每次调用会替换一个分隔符。如果有连续的分隔符则第二次调用返回指针的内容为空。

char \*strndup(const char \*s, size\_t n);

char \*strdupa(const char \*s);

char \*strndupa(const char \*s, size\_t n);

**字符串与数的转换：**

double strtod(const char \*nptr, char \*\*endptr);

float strtof(const char \*nptr, char \*\*endptr);

long double strtold(const char \*nptr, char \*\*endptr);

long int strtol(const char \*nptr, char \*\*endptr, int base);

long long int strtoll(const char \*nptr, char \*\*endptr, int base);

unsigned long int strtoul(const char \*nptr, char \*\*endptr, int base);

unsigned long long int strtoull(const char \*nptr, char \*\*endptr, int base);

int atoi(const char \*nptr);

long atol(const char \*nptr);

long long atoll(const char \*nptr);

double atof(const char \*nptr);

# LINUX 进程控制

## 1 Linux 进程概述

进程是一个程序一次执行的过程，它和程序有本质区别。程序是静态的，它是一些保存在磁盘上的指令的有序集合；而进程是一个动态的概念，它是一个运行着的程序，包含了进程的动态创建、调度和 消亡的过程，是 Linux 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？在这里，是通过进程控制块（PCB）来描述的。进程控制块包含了进程的描述信息、控制信息以及资源信息，它是进程的一个静态描述。 内核使用进程来控制对CPU 和其他系统资源的访问，并且使用进程来决定在 CPU 上运行哪个程序， 运行多久以及采用什么特性运行它。内核的调度器负责在所有的进程间分配 CPU 执行时间，称为时间片 (timeslice)，它轮流在每个进程分得的时间片用完后从进程那里抢回控制权。ps –elf查看系统所有进程。

### 1.1 进程标识

OS 会为每个进程分配一个唯一的整型 ID，做为进程的标识号(pid)。进程除了自身的 ID 外，还有父进程 ID(ppid),所有进程的祖先进程是同一个进程，它叫做 init 进程，ID 为 1，init 进程是内核自举后的一个启动的进程。init 进程负责引导系统、启动守护（后台）进程并且运行必要的程序。进程的 pid 和 ppid 可以分别通过函数 getpid()和 getppid()获得。

#include <sys/types.h>

#include <unistd.h>

pid\_t getpid(void);

pid\_t getppid(void);

成功返回进程或父进程id，这两个函数始终成功

示例：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main**()**

**{**

printf**(**"pid:%d ppid:%d\n"**,**getpid**(),**getppid**());**

return0**;**

**}**

### 1.2 进程的用户 ID与组ID

进程在运行过程中，必须具有一类似于用户的身份，以便进行进程的权限控制，缺省情况下，哪个 登录用户运行程序，该程序进程就具有该用户的身份。例如，假设当前登录用户为 gotter,他运行了 ls 程序，则 ls 在运行过程中就具有 gotter 的身份，该 ls 进程的用户 ID 和组 ID 分别为 gotter 和 gotter 所属的组。这类型的 ID 叫做进程的真实用户 ID 和真实组 ID。真实用户 ID 和真实组 ID 可以通过函数 getuid()和 getgid()获得。

与真实 ID 对应，进程还具有有效用户 ID 和有效组 ID 的属性，内核对进程的访问权限检查时，它 检查的是进程的有效用户 ID 和有效组 ID，而不是真实用户 ID 和真实组 ID。缺省情况下，用户的（有 效用户 ID 和有效组 ID）与(真实用户 ID 和真实组 ID)是相同的。有效用户 id 和有效组 id 通过函数 geteuid()和 getegid()获得。

#include <unistd.h>

#include <sys/types.h>

uid\_t getuid(void);

uid\_t geteuid(void)

gid\_t getgid(void);

gid\_t getegid(void);

这些函数始终成功，不会出错。

示例：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main**()**

**{**

printf**(**"uid:%dgid:%deuid:%degid:%d\n"**,**getuid**(),**getgid**(),**geteuid**(),**getegid**());**

return0**;**

**}**

输入id可以看到：

uid=500(ghaha) gid=500(ghaha) groups=500(ghaha)

编译生成可执行文件 main.exe，程序文件的属性可能为：

-rwxrwxr-x 1 ghaha ghaha 12132 Oct 7 09:26 a.out

执行结果可能为：

$./main.exe

uid:500 gid:500 euid:500 egid:500

现在将 main.exe 的所有者可执行属性改为 s

chmod u+s main.exe

$l -al

-rwsrwxr-x 1 ghaha ghaha 12132 Oct 7 09:26 a.out

此时改另外一个用户 gotter 登录并运行程序main.exe

$id

uid=502(gotter) gid=502(gotter) groups=502(gotter)

$./main.exe

uid:502 gid:502 euid:500 egid:500

可以看到，gotter运行进程的有效用户身份由原来的gotter变为了 ghaha，这是因为文件 a.out 的访问权限 的所有者可执行权限设置了s属性，设置了该属性以后，用户运行 main.exe 时main.exe 进程的有效用户身份将不再是运行main.exe 的用户，而是变成了main.exe文件的所有者。

s权限的作用就是让任何用户运行某程序时，该程序都能具有该程序所有者的有效用户ID，从而让此程序可以对仅允许该程序所有者操作的文件进行操作。该用户就拥有了文件所有者的权限

s权限最常见的例子是:

/usr/bin/passwd 改密码程序，它的权限位为

l -l /usr/bin/passwd

-r-s--x--x 1 root root 16336 Feb 13 2003 /usr/bin/passwd

我们知道，用户的用户名和密码是保存在/etc/passwd（后来专门将密码保存在/etc/shadow，它是 根据/etc/passwd 文件来生成/etc/shadow 的，它把所有口令从/etc/passwd 中移到了/etc/shadow 中。这里用到的是影子口令，它将口令文件分成两部分：/etc/passwd 和/etc/shadow，此时 /etc/shadow 就是影子口令文件，它保存的是加密的口令，而/etc/passwd 中的密码全部变成 x）下 的。通过 ls –l 查看/etc/passwd 这个文件，你会发现，这个文件普通用户都没有可写的权限，那 我们执行 passwd 的时候确实能够修改密码，那么这是怎么回事呢？也就是说，任何一个用户运行该程序时，该程序的有效身份euid都将变为 文件所有者root（用普通身份去执行这个操作的时候，它会暂时得到文件拥有者 root 的权限）,而这样 passwd 程序才有权限读取/etc/passwd 文件的信息。

我们也来实现以下 passwd 的功能，实现步骤如下：

1)用 touch 创建一个 a.txt 输入内容“hello” 类似于/etc/passwd

2) 写一个程序main.c 如下：

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <stdlib.h>

#include <errno.h>

int main**()**

**{**

printf**(**"uid : %d gid : %d\n"**,** getuid**(),** getgid**());**

printf**(**"eudi: %d egid: %d\n"**,** geteuid**(),** getegid**());**

FILE**\*** fp **=** fopen**(**"a.txt"**,** "a"**);**//注意这里以追加形式打开，说明a.txt要具有可写权限

**if(**fp **==** **NULL)**

**{**

perror**(**"fopen error"**);**

exit**(-**1**);**

**}**

fputs**(**"world"**,** fp**);**

fclose**(**fp**);**

**return** 0**;**

**}**

3) 编译 gcc –o 1 main.c 生成可执行程序main此时main 类似于/usr/bin/passwd 文件

4) 用 l -l 查看 a.txt 发现权限是-rw-r—r—的权限（跟/etc/passwd 权限一样），说明普通用户没有可写权限，如果直接切换到另一个用户执行./main 会报错。

5) 用 root 身份将main的权限修改为-rwsr-xr-x (操作命令：chmod u+s main，此时跟/usr/bin/passwd 的权限一样)，此时再切换到普通用户 wangxiao，执行./main 发现可以执行成功，因为此时./main 进程的 有效用户 id 变成 root 了，也就是说普通用户是借助 root 身份来实现的。

**文件特殊权限**

SUID权限：任何用户执行具有SUID权限和x权限的文件时拥有此文件所有者的权限，此文件执行时其euid会变成文件所有者的UID。如果对目录设置了SUID权限，则所有用户在该目录下创建的文件都属于该目录所有者，而不是创建者。

SGID权限：任何用户组执行具有SGID权限和x权限的文件时拥有此文件所在组的权限，此文件执行时其egid会变成文件所在组的GID。如果对目录设置了SGID权限，则所有用户在该目录下创建的文件都属于该目录所在组，而不是创建者所在组。

SBIT权限：只对目录有效，使具有w任何用户都可以在该目录下创建文件，但只有文件所有者和root才能删除，即是其他人对目录有写权限。

chmod u+s 文件名：增加SUID权限，原来有x权限显示为s，原来没有x权限显示为S

chmod u-s 文件名：删除SUID权限

chmod g+s 文件名：增加SGID权限，原来有x权限显示为s，原来五x权限显示为S

chmod g-s 文件名：删除SGID权限，

chmod o+t 文件名：增加SBIT权限，原来有x权限显示为t，原来五x权限显示为T

chmod o-t 文件名：删除SBIT权限

### 1.3 进程的状态

进程是程序的执行过程，根据它的生命周期可以划分成 3 种状态。 

执行态：该进程正在运行，即进程正在占用 CPU。 

就绪态：进程已经具备执行的一切条件，正在等待分配 CPU 的处理时间片。 

等待态：进程不能使用 CPU，若等待事件发生（等待的资源分配到）则可将其唤醒。



Linux调度策略分为:

（1）实时调度策略FIFO（先进先出）和RR（时间片轮转）,要求对进程运行的时间掌握很精确，FIFO同优先级只允许先玩一个在运行下一个，RR则要求同优先级进程平均分配时间。FIFO和RR不同优先级先执行优先级高的。

（2）非实时OTHER：在一个调度周期内不论优先级，都会得到时间片

### 1.4 Linux 下的进程结构

Linux 系统是一个多进程的系统，它的进程之间具有并行性、互不干扰等特点。也就是说，进程之间是分离的任务，拥有各自的权利和责任。其中，每个进程都运行在各自独立的虚拟地址空间，因此， 即使一个进程发生了异常，它也不会影响到系统的其他进程。

Linux 中的进程包含3 个段，分别为“代码段”“数据段”、和“堆栈段”。 ·

“数据段”放全局变量、常数以及动态数据分配的数据空间。数据段分成普通数据段（包括可 读可写/只读数据段，存放静态初始化的全局变量或常量）、BSS 数据段（存放未初始化的全局变量）以及堆（存放动态分配的数据）。

“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等。



### 1.5 Linux 下的进程管理

启动进程：手工启动 调度启动

|  |  |
| --- | --- |
| 命令 | 含义 |
| ps | 查看系统进程 |
| top | 动态显示系统进程 |
| 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 | 向进程发送信号（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 |
| bg | 将挂起的进程放到后台执行 |
| fg | 把后台任务带到前台 |

备注： 进程 process：是 os 的最小单元 os 会为每个进程分配大小为 4g 的虚拟内存空间，其中 1g 给内核 空间 3g 给用户空间｛代码区 数据区 堆栈区｝

常用的ps命令：

ps 查看活动进程

ps -aux 查看所有的进程所占硬件资源

ps -elf查看系统所有进程

ps -ux| grep 进程名 查找指定（aa）进程

ps -f 可以显示父子进程关系

top 显示前 20 条进程，动态的改变

vim a.c &(&表示后台运行)，一个死循环，按 ctrl+z 可以把进程暂停，再执行[bg 作业ID]可以将

该进程带入后台。利用 jobs 可以查看后台任务，[fg 作业ID]把后台任务带到前台

kill 信号编号 进程id表示向某个进程发送 某号信号，从而杀掉某个进程

pkill 进程名 可以杀死进程名为 a 的进程

ps –elf输出项解释

F 代表这个程序的旗标 (flag)， 4 代表使用者为 superuser；

S 代表这个程序的状态 (STAT)；

UID 代表执行者身份

PID 进程的ID号！

PPID 则父进程的ID；

C CPU 使用的资源百分比

PRI指进程的执行优先权(Priority的简写)，其值越小越早被执行；

NI 这个进程的nice值，其表示进程可被执行的优先级的修正数值。

ADDR 这个是内核函数，指出该程序在内存的那个部分。如果是个执行的程序，一般就是 -

SZ 使用掉的内存大小；

STIME 开始执行时间

WCHAN 目前这个程序是否正在运作当中，若为 - 表示正在运作；

TTY 登入者的终端机位置啰；

TIME 使用掉的 CPU 时间。

CMD 所下达的指令名称

ps -aux 输出项解释：

USER 进程的属主；

PID 进程的ID；

PPID 父进程；

%CPU 进程占用的CPU百分比；

%MEM 占用内存的百分比；kB

NI 进程的NICE值，数值大，表示较少占用CPU时间；

VSZ 进程虚拟大小；

RSS 驻留中页的数量；

TTY 该进程是在那个终端机上面运作，若与终端机无关，则显示 ?，另外， tty1-tty6 是本机上面的登入者程序，若为 pts/0 等等的，则表示为由网络连接进主机的程序。

STAT 进程状态（有以下几种）

D 无法中断的休眠状态（通常 IO 的进程）；

R 正在运行可中在队列中可过行的；

S 处于休眠状态；

T 停止或被追踪；

W 进入内存交换（从内核2.6开始无效）；

X 死掉的进程（从来没见过）；

Z 僵尸进程；

< 优先级高的进程

N 优先级较低的进程

L 有些页被锁进内存；

s 进程的领导者（在它之下有子进程）；

l 多进程的（使用 CLONE\_THREAD, 类似 NPTL pthreads）；

+ 位于后台的进程组；

WCHAN 正在等待的进程资源；

START 启动进程的时间；

TIME 进程消耗CPU的时间；

COMMAND 命令的名称和参数

### 1.6 top 命令详解

top - 22:53:11 up 3 days, 5:12, 4 users, load average: 0.03, 0.02, 0.00

Tasks: 258 total, 1 running, 257 sleeping, 0 stopped, 0 zombie

%Cpu(s): 0.2 us, 0.3 sy, 0.0 ni, 99.5 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

KiB Mem : 1446520 total, 771296 free, 326220 used, 349004 buff/cache

KiB Swap: 2094076 total, 1566932 free, 527144 used. 925884 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

7071 liao 20 0 1129548 38916 13124 S 1.3 2.7 36:10.36 compiz

42164 liao 20 0 43660 4152 3420 R 0.7 0.3 0:00.37 top

第一行：

10:01:23 — 当前系统时间

126 days, 14:29 — 系统已经运行了126天14小时29分钟（在这期间没有重启过）

2 users — 当前有2个用户登录系统

load average: 1.15, 1.42, 1.44 — load average后面的三个数分别是1分钟、5分钟、15分钟的负载情况。

第二行：

Tasks — 任务（进程），系统现在共有183个进程，其中处于运行中的有1个，182个在休眠（sleep），stoped状态的有0个，zombie状态（僵尸）的有0个。

第三行：cpu状态

6.7% us — 用户空间占用CPU的百分比。

0.4% sy — 内核空间占用CPU的百分比。

0.0% ni — 改变过优先级的进程占用CPU的百分比

92.9% id — 空闲CPU百分比

0.0% wa — IO等待占用CPU的百分比

0.0% hi — 硬中断（Hardware IRQ）占用CPU的百分比

0.0% si — 软中断（Software Interrupts）占用CPU的百分比

0.0%st — steal time

第四行：内存状态

8306544k total — 物理内存总量（8GB）

7775876k used — 使用中的内存总量（7.7GB）

530668k free — 空闲内存总量（530M）

79236k buffers — 缓存的内存量 （79M）

第五行：swap交换分区

2031608k total — 交换区总量（2GB）

2556k used — 使用的交换区总量（2.5M）

2029052k free — 空闲交换区总量（2GB）

4231276k cached — 缓冲的交换区总量（4GB）

第六行是空行

第七行以下：各进程（任务）的状态监控

|  |  |  |
| --- | --- | --- |
| 序号 | 列名 | 含义 |
| 1 | PID | 进程id |
| 2 | PPID | 父进程id |
| 3 | RUSER | Real user name |
| 4 | UID | 进程所有者的用户id |
| 5 | USER | 进程所有者的用户名 |
| 6 | GROUP | 进程所有者的组名 |
| 7 | TTY | 启动进程的终端名.不是从终端启动的进程则显示为 ? |
| 8 | PR | 优先级 |
| 9 | NI | nice值.负值表示高优先级，正值表示低优先级 |
| 10 | P | 最后使用的CPU,仅在多CPU环境下有意义 |
| 11 | %CPU | 上次更新到现在的CPU时间占用百分比 |
| 12 | TIME | 进程使用的CPU时间总计,单位秒 |
| 13 | TIME+ | 进程使用的CPU时间总计,单位1/100秒 |
| 14 | %MEM | 进程使用的物理内存百分比 |
| 15 | VIRT | 进程使用的虚拟内存总量,单位kb,VIRT=SWAP+RES |
| 16 | SWAP | 进程使用的虚拟内存中,被换出的大小,单位kb. |
| 17 | RES | 进程使用的、未被换出的物理内存大小,单位kb,RES=CODE+DATA |
| 18 | CODE | 可执行代码占用的物理内存大小,单位kb |
| 19 | DATA | 可执行代码以外的部分(数据段+栈)占用的物理内存大小,单位kb |
| 20 | SHR | 共享内存大小,单位kb |
| 21 | nFLT | 页面错误次数 |
| 22 | nDRT | 最后一次写入到现在,被修改过的页面数. |
| 23 | S | 进程状态:  **D**=不可中断的睡眠状态 ； **R**=运行； **S**=睡眠；**T**=跟踪/停止； **Z**=僵尸进程 |
| 24 | COMMAND | 命令名/命令行 |
| 25 | WCHAN | 若该进程在睡眠,则显示睡眠中的系统函数名 |
| 26 | Flags | 任务标志,参考 sched.h |

## 2 进程的创建

Linux 下有四类创建子进程的函数：system(),fork(),exec\*(),popen()

### 2.1 system 函数

原型：

#include<stdlib.h>

int system(const char\*string);

system 函数通过调用 shell 程序/bin/sh –c 来执行 string 所指定的命令，该函数在内部是通过调用 execve(“/bin/sh”,..)函数来实现的。通过 system 创建子进程后，原进程和子进程各自运行，相互间关联较少。system（）的返回值是以下之一：

如果命令为NULL，则如果shell可用，则为非零值，如果没有可用的shell则为0。

如果无法创建子进程，或者无法检索其状态，返回值为-1。

如果在子进程中无法执行shell，那么返回值就好像通过调用具有状态127的\_exit（2）来终止子shell。

如果所有系统调用成功，则返回值是用于执行命令的子shell的终止状态。 （shell的终止状态是其执行的最后一个命令的终止状态。）

在最后两种情况下，返回值是一个“等待状态”，可以使用宏中描述的宏进行检查waitpid函数。 （即WIFEXITED（）WEXITSTATUS（）等）。

system（）不影响任何其他孩子的等待状态。

示例：

#include <stdio.h>

#include <stdlib.h>

int main**()**

**{**

system**(**"ls -l "**);** //system(“clear”);表示清屏

return0**;**

**}**

此外，system函数后面的参数还可以是一个可执行程序，例如： system(“/home/wangxiao/1”);如果想要执行system后面进程的时候，不至于对当前进程进行阻塞，可以利用&将/home/wangxiao/1调到后台运行。让程序在后台执行，只需在命令行的最后加上“&”符号。

### 2.2 fork 函数

原型：

#include <unistd.h>

pid\_t fork(void);

在 linux 中 fork 函数是非常重要的函数，它从已存在进程中创建一个新进程。新进程为子进程， 而原进程为父进程。它和其他函数的区别在于：它执行一次返回两个值。其中父进程的返回值是子进程 的进程号，而子进程的返回值为 0.若出错则父进程返回-1. 设置了标准错误描述，可使用perror函数查看。因此可以通过返回值来判断是父进程还是子进程。使用了fork函数后必须进行判断。

fork函数创建子进程的过程为：使用fork 函数得到的子进程是父进程的一个复制品，它从父进程继承了进程的地址空间，包括进程上下文、进程堆栈、内存信息、打开的文件描述符、信号控制设定、进程优先级、进程组号、当前工作目录、根目录、资源限制、控制终端，而子进程所独有的只有它的进程号、资源使用和计时器等。通过这种复制方式创建出子进程后，原有进程和子进程都从函数 fork返回， 各自继续往下运行，但是原进程的 fork 返回值与子进程的fork 返回值不同，在原进程中，fork 返回子进程的 pid,而在子进程中，fork返回 0,如果fork 返回负值，表示创建子进程失败。（vfork 函数）

示例：

#include <stdlib.h>

#include <unistd.h>

int main**()**

**{**

printf**(**"Parent processid:%d\n"**,**getpid**());**

pid\_t iRet **=**fork**();**

**if(**iRet**<** 0**)**

**{**//出错

printf**(**"Create child process fail!\n"**);**

**}**

**else** **if(**iRet**==** 0**)**

**{**//表示子进程

printf**(**"child process id:%d ppid:%d\n"**,**getpid**(),**getppid**());**

sleep（10）

**}**

**else**

**{**//表示父进程

printf**(**"parentprocess success,childid:%d\n"**,**iRet**);**

sleep**(**10**)**

**}**

return0**;**

**}**

有人可能会有疑问：这里怎么 if 和 else 里面的语句都得到执行了，和我们以前的 if…else 结 构相矛盾啊？此时相当于有两份 main 函数代码的拷贝，其中一份做的操作是 if(iRet == 0)的情况； 另外一份做的操作是 else(父)的情况。所以可以输出 2 句话。

### 2.3 exec 函数族(进程替换)

exec\*由一组函数组成

#include<unistd.h>

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

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

int execle(const char \*path, const char \*arg, ... /\*, (char \*) NULL, 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[]);

int execvpe(const char \*file, char \*const argv[],char \*const envp[]);

功能：exec 函数族的工作过程与 fork 完全不同，fork 是复制一份原进程，而exec 函数是用exec 的第一个参数指定的程序覆盖现有进程空间（也就是说执行 exec 族函数之后，它后面的所有代码不在执行）。

要记住这六个函数之间的区别，就要搞清楚 “l” 、“v”、“p”、“e”代表的含义

“l”希望接收以逗号分隔的参数列表，列表以NULL指针作为结束标志

“v”希望接收到一个以NULL结尾的字符串数组

“p”表示在PATH中搜索执行文件,可以只给出文件名

“e”表示允许改变进程环境参数。

execlp、execvp这两个函数第一个形参是名为file表示可以只列出文件名就可以了，不用指出它的路径。

注意的地方有：

（1）参数以NULL结尾；

（2）exec函数族执行成功后不会返回，执行失败会返回 -1，设置了标准错误描述，可使用perror函数查看。

（3）可能的几种错误有，找不到文件或路径，errno为ENOENT；数组argv或envp没有以NULL结尾，errno为EFAULT；没有对执行文件的执行权限，errno为EACCESS；

参数file可以是某一个shell脚本，或可执行文件名

参数envp要改变的进程运行环境变量

参数path 是包括执行文件名的全路径名

参数arg是可执行文件的命令行参数，第一位执行文件名，相当于main函数中的argv[0];多个命令行参数用逗号分割注意最后一个参数必须为 NULL。

参数argv[]，字符串数组，用以存放参数，最后一个元素为NULL。

例如，有个加法程序，从命令行接受两个数，输出其和 。 代码如下：

//add.c

#inclue<stdio.h>

#include<string.h>

Int main**(**int argc**,**char**\*** argv**[])**

**{**

Int a**=**atoi**(**argv**[**1**]);**//字符串转整数函数atoi

Int b**=**atoi**(**argv**[**2**]);**

printf**(**“**%**d**+%**d**=%**d”**,**a**,**b**,**a**+**b**);**

Return0**;**

**}**

编译连接得到add.exe.

Gcc–oadd.exe add.c

然后在 main.exe 中调用 add.exe 程序 ,计算 3 和 4 的和。

Main.c 的源程序为 ，

//main.c

#include<stdio.h> #include<string.h>

Int main**()**

**{**

Execl**(**“**./**add**.**exe”**,**”add**.**exe”**,**”3”**,**“4”**,NULL):**

Return 0**;**

**}**

编译连接得，

Gcc–omain.exe main.c

然后运行 。

./main.exe。

在运行 main.exe 的过程中会通过 execl 启动之前的 add.exe 程序。

### 2.4 popen 函数

popen 函数类似于 system 函数，与 system 的不同之处在于它使用管道工作。popen() 函数通过创建一个管道，调用 fork 产生一个子进程。原型为：

#include <stdio.h>

FILE \*popen(const char \*command, const char\*type);

int pclose(FILE \*stream);

command 为可执行文件的全路径和执行参数；如果fork或pipe调用失败，或内存分配失败，返回NULL。

type 可选参数为”r”或”w”, 如果 type 是“r”，（command 命令执行的输出（返回文件流）作为当前进程的输入）被调用程序的输出就可以被调用程序使用，调用程序利用 popen 函数返回的 FILE\*文件流指 针，就可以通过常用的 stdio 库函数（如 fread）来读取被调用程序的输出；如果 tpye 是“w”， （即当前进程的输出作为 command 命令的输入结。调用程序就可以用 fwrite 向被 调用程序发送数据，而被调用程序可以在自己的标准输入上读取这些数据。

popen 的返回值是个标准 I/O 流，必须由 pclose 来终止。出错返回-1，设置了标准错误描述，可使用perror函数查看。

pclose 函数等待新进程的结束，而不是杀新进程。

示例：

#include<unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

int main**()**

**{**

FILE**\***read\_fp**;**

char buffer**[**BUFSIZ**+**1**];**

int chars\_read**;**

memset**(**buffer**,** '\0'**,sizeof(**buffer**));**

read\_fp **=**popen**(**"ps-ax"**,**"r"**);**

**if(**read\_fp**!=NULL)**

**{**

chars\_read**=**fread**(**buffer**,sizeof(**char**),**BUFSIZ**,**read\_fp**);**

**while(**chars\_read**>**0**)**

**{**

buffer**[**chars\_read**-**1**]=**'\0'**;**

printf**(**"Reading:-\n%s\n"**,**buffer**);**

chars\_read**=**fread**(**buffer**,sizeof(**char**),**BUFSIZ**,** read\_fp**);**

**}**

pclose**(**read\_fp**);**

exit**(**EXIT\_SUCCESS**);**

**}**

exit**(**EXIT\_FAILURE**);**

**}**

## 3 进程控制与终止

### 3.1 进程的控制

用 fork 函数启动一个子进程时，子进程就有了它自己的生命并将独立运行。 如果父进程先于子进程退出，则子进程成为孤儿进程，此时将自动被 PID 为 1 的进程 （即 init）接管。孤儿进程退出后，它的清理工作由祖先进程 init 自动处理。但在 init 进程 清理子进程之前，它一直消耗系统的资源，所以要尽量避免。

例1：写一个孤儿进程：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

void main**()**

**{**

pid\_t pid **=**fork**();**

**if(** pid**==**0**)**

**{**

**while(**1**)** **;**

**}**

**else**

**{**

exit**(**10**);**

**}**

**}**

通过 ps–ef 就可以看到此时子进程一直在运行，并且父进程是 1 号进程。 如果子进程先退出，系统不会自动清理掉子进程的环境，而必须由父进程调用 wait 或 waitpid 函数来完成清理工作，如果父进程不做清理工作，则已经退出的子进程将成为僵尸进程(defunct),在系统中如果存在的僵尸（zombie）进程过多，将会影响系统的性能，所以必须对僵尸进程进行处理。

函数原型：

#include <sys/types.h>

#include <sys/wait.h>

pid\_t wait(int \*status);

pid\_t waitpid(pid\_t pid, int \*status,int options);

wait 和 waitpid 都将暂停父进程，等待一个已经退出的子进程，并进行清理工作；

wait 函数随机地等待一个已经退出的子进程做清理工作，并返回该子进程的 pid；NULL 表示等待所有进程

waitpid 等待指定 pid 的子进程做清理工作；如果为-1 表示等待所有子进程。

status参数是传出参数，含有两信息,存放子进程的退出状态；通常用下面的两个宏来获取状态信息： WIFEXITED(status) 如果子进程正常结束，它就取一个非0值。若异常结束则返回0。status传入的时整型值，非地址。 WEXITSTATUS(status) 如果 WIFEXITED 非零，它返回子进程的退出码 ，否则无无意义。

options 用于改变 waitpid 的行为，其中最常用的是 WNOHANG，它表示无论子进程是否退出都将立即返回，不会将调用者的执行挂起。

如果返回时子进程正常退出则返回子进程pid，如果返回时由于信号造成则返回-1，如果wapid输了WNOHANG选项，返回0，否则返回-1. 设置了标准错误描述，可使用perror函数查看。

例1：写一个僵尸进程：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

void main**()**

**{**

pid\_t pid **=**fork**();**

**if(** pid**==**0**)**

**{**

exit**(**10**);**

**}**

**else**

**{**

sleep**(**10**);**

**}**

**}**

通过用 ps–aux 快速查看发现 Z 的僵尸进程。

例2：避免僵尸进程：(wait()函数)

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

void main**()**

**{**

pid\_t pid **=**fork**();**

**if(** pid**==**0**)**

**{**

exit**(**10**);**

**}**

**else**

**{**

wait**(NULL);** //NULL 表示等待所有进程

sleep**(**10**);** //通常要将 sleep 放在 wait 的后面，要不然也会出现僵尸进程

**}**

**}**

例3：waitpid实现

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <signal.h>

void SignChildPsExit**(**int iSignNo**)**

**{**

int iExitCode**;**

pid\_t pid **=**waitpid**(-**1**,NULL,**0**);**//表示等待任何进程，并阻塞。

//如换成 waitpid(-1,NULL,WNOHANG);则跟没有写waitpid效果类似，此时父进程没有阻塞

printf**(**"SignNo:%d child %dexit\n"**,**iSignNo**,**pid**);**

**if(**WIFEXITED**(**iExitCode**))**

**{**

printf**(**"Child exitedwithcode%d\n"**,** WEXITSTATUS**(**iExitCode**));**

**}**

sleep**(**10**);**

**}**

int main**()**

**{**

signal**(**SIGCHLD**,** SignChildPsExit**);**

printf**(**"Parent processid:%d\n"**,** getpid**());**

pid\_t iRet **=**fork**();**

**if(**iRet **==**0**)**

exit**(**3**);**

**}**

### 3.2 进程的终止

进程的终止有 5 种方式： 

main 函数的自然返回； 

调用 exit 函数 

调用\_exit 函数 

调用 abort 函数 

接收到能导致进程终止的信号 ctrl+c SIGINT ctrl+\ SIGQUIT

前 3 种方式为正常的终止，后 2 种为非正常终止。但是无论哪种方式，进程终止时都将 执行相同的关闭打开的文件，释放占用的内存等资源。只是后两种终止会导致程序有些代 码不会正常的执行比如对象的析构、atexit 函数的执行等。

exit 和\_exit 函数都是用来终止进程的。当程序执行到 exit 和\_exit 时，进程会无条件的停止剩下的所有操作，清除包括 PCB 在内的各种数据结构，并终止本程序的运行。但是它 们是有区别的，exit 和\_exit 的区别如图所示：

exit 函数和\_exit 函数的最大区别在于 exit 函数在退出之前会检查文件的打开情况，把文件缓冲区中的内容写回文件，就是图中的“清理 I/O 缓冲”。

由于 linux 的标准函数库中，有一种被称作“缓冲 I/O”操作，其特征就是对应每一个 打开的文件，在内存中都有一片缓冲区。每次读文件时，会连续读出若干条记录，这样在 下次读文件时就可以直接从内存的缓冲区中读取；同样，每次写文件的时候，也仅仅是写 入内存中的缓冲区，等满足一定的条件（如达到一定数量或遇到特定字符等），再将缓冲 区中的内容一次性写入文件。这种技术大大增加了文件读写的速度，但也为编程带来了麻 烦。比如有一些数据，认为已经写入文件，实际上因为没有满足特定的条件，它们还只是 保存在缓冲区内，这时用\_exit 函数直接将进程关闭，缓冲区中的数据就会丢失。因此，如 想保证数据的完整性，建议使用 exit 函数。

exit用于结束正在运行的整个程序，它将参数返回给OS，把控制权交给操作系统；而return 是退出当前函数，返回函数值，把控制权交给调用函数。

exit是系统调用级别，它表示一个进程的结束；而return 是语言级别的，它表示调用堆栈的返回。

在main函数结束时，会隐式地调用exit函数，所以一般程序执行到main()结尾时，则结束主进程。exit将删除进程使用的内存空间，同时把错误信息返回给父进程。

在C++中类的成员函数中调用exit函数，将不会执行析构函数。

exit 和\_exit 函数的原型：

#include<stdlib.h> //exit 的头文件

#include<unistd.h> //\_exit 的头文件

void exit(int status);

void \_exit(int status);

status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 示正常结束；其他的数值表示出现了错误，进程非正常结束。

Example1：exit 的举例如下：

#include <stdio.h>

#include <stdlib.h>

int main**()**

**{**

printf**(**"Using exit...\n"**);**

printf**(**"This is the content in buffer"**);**

exit**(**0**);**

**}**

可以发现，调用 exit 函数，缓冲区中的记录也能正常输出。

Example2：\_exit 的举例如下：

#include <stdio.h>

#include <unistd.h>

int main**()**

**{**

printf**(**"Using \_exit...\n"**);**

printf**(**"This is the content in buffer"**);**

\_exit**(**0**);**

**}**

可以发现，最后的输出结果没有 This is the content inbuffer，说明\_exit 函数无法输出缓 冲区中的记录。

## 4 进程间打开文件的继承

### 4.1 用 fork 继承打开的文件

fork 以后的子进程自动继承了父进程的打开的文件，继承以后，父进程关闭打开的文件不会对子进 程造成影响。

示例：

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

int main**()** //这是一个孤儿进程

**{**

char szBuf**[**32**]** **={**'\0'**};**

int iFile **=**open**(**"./a.txt"**,**O\_RDONLY**);**

**if(**fork**()>**0**)**

**{**//parent process

close**(**iFile**);**

return0**;**

**}** //child process

sleep**(**3**);** //wait forparent processclosing fd

**if(**read**(**iFile**,** szBuf**,** **sizeof(**szBuf**)-**1**)<**1**)**

**{**

perror**(**"readfail"**);**

**}**

**else**

**{**

printf**(**"string:%s\n"**,**szBuf**);**

**}**

close**(**iFile**);**

return0**;**

**}**

### 4.2 守护进程

Daemon 运行在后台也称作“后台服务进程”。 它是没有控制终端与之相连的进程。它独 立与控制终端、会话周期的执行某种任务。那么为什么守护进程要脱离终端后台运行呢？ 守护进程脱离终端是为了避免进程在执行过程中的信息在任何终端上显示并且进程也不会被任何终端所产生的任何终端信息所打断。那么为什么要引入守护进程呢？由于在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赖这个终端，这个终端就称为这些进程的控制终端。当控制终端被关闭时，相应的进程都会自动关闭。但是守护进程却能突破这种限制，它被执行开始运转，直到整个系统关闭时才退 出。几乎所有的服务器程序如 Apache 和 wu-FTP，都用 daemon 进程的形式实现。很多 Linux 下常见的命令如 inetd 和 ftpd，末尾的字母d通常就是指 daemon。

守护进程的特性：

1> 守护进程最重要的特性是后台运行。

2> 其次，守护进程必须与其运行前的环境隔离开来。这些环境包括未关闭的文件描述符、控制终端、会话和进程组、工作目录、已经文件创建掩码等。这些环境通常是守护进程从父进程那里继承下来的。

3> 守护进程的启动方式

daemon 进程的编程规则

①创建子进程，父进程退出

调用 fork 产生一个子进程，同时父进程退出。我们所有后续工作都在子进程中完成。 这样做我们可以交出控制台的控制权,并为子进程作为进程组长作准备;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，变成一个孤儿进程（orphan）。每当系统发现一个孤儿进程，就会自动由 1 号进程收养它，这样，原先的子进程就会变成 1 号进程的子进 程。代码如下：

pid=fork();

if(pid>0)

exit(0); 

②在子进程中创建新会话：

使用系统函数 setsid()。由于创建守护进程的第一步调用了 fork 函数来创建子进程，再 将父进程退出。子进程就成了孤儿进程，有init代管。由于在调用 fork 函数的时候，子进程全盘拷贝了父进程的会话期、进程组、 控制终端等，虽然父进程退出了，但会话期、进程组、控制终端并没有改变，因此，还不 是真正意义上的独立开来。而调用 setsid 函数会创建一个新的会话并自任该会话的组长，调 用 setsid 函数有下面 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；

进程组：是一个或多个进程的集合。进程组由进程组 ID 来唯一标识。除了进程号(PID) 之外，进程组 ID（PGID）也是一个进程的必备属性。每个进程都有一个组长进程，其组长进程的进程号等于进程组 ID。且该进程组 ID 不会因为组长进程的退出而受影响。pgid可由getpgid和setpgid来得到和设置。

#include <unistd.h>

pid\_t getpgid(pid\_t pid);

int setpgid(pid\_t pid, pid\_t pgid);

成功返回进程组id，或者返回0.出错返回-1，设置了标准错误描述，可使用perror函数查看。

会话周期（会话组）：会话期是一个或多个进程组的集合。通常，一个会话开始于用户登录，终止于用户退出，在此期间该用户运行的所有进程都属于这个会话期。

#include <unistd.h>

pid\_t getsid(pid\_t pid);

pid\_t setsid(void);

成功返回原会话组或者新会话组id，出错返回-1，设置了标准错误描述，可使用perror函数查看。

控制终端：由于在 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赖这个控制终端。 

③改变当前目录为根目录：

使用 fork 函数创建的子进程继承了父进程的当前工作目录。由于在进程运行中，当前 目录所在的文件是不能卸载的，这对以后的使用会造成很多的不便。利用 chdir("/");把当前 工作目录切换到根目录。 

④重设文件权限掩码：

umask(0);将文件权限掩码设为 0,Deamon 创建文件不会有太大麻烦； 

⑤关闭所有不需要的文件描述符：

新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 守护进程读写，而它们一直消耗系统资源。另外守护进程已经与所属的终端失去联系，那 么从终端输入的字符不可能到达守护进程，守护进程中常规方法（如 printf）输出的字符也 不可能在终端上显示。所以通常关闭从 0 到 MAXFILE 的所有文件描述符。

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

close(i);

（注：有时还要处理 SIGCHLD 信号 signal(SIGCHLD,SIG\_IGN);防止僵尸进程（zombie）） 下面就可以添加任何你要 daemon 做的事情

示例：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/stat.h>

void Daemon**()**

**{**

const int MAXFD**=**64**;**

int i**=**0**;**

**if(**fork**()!=**0**)**//父进程退出

exit**(**0**);**

setsid**();** //成为新进程组组长和新会话领导，脱离控制终端

chdir**(**"/"**);** //设置工作目录为根目录

umask**(**0**);** //重设文件访问权限掩码

**for(;**i**<**MAXFD**;**i**++)** //尽可能关闭所有从父进程继承来的文件

close**(**i**);**

**}**

int main**()**

**{**

Daemon**();** //成为守护进程

**while(**1**)**

**{**

sleep**(**1**);**

**}**

**return** 0**;**

**}**

Example：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <syslog.h>

#include <sys/stat.h>

#include <time.h>

int main**()**

**{**

int i **=** 0**;**

**if(**fork**()** **>** 0**)**

exit**(**0**);**

setsid**();**

chdir**(**"/"**);**

umask**(**0**);**

**for(;** i **<** 64**;** i**++)**

**{**

close**(**i**);**

**}**

i **=** 0**;**

**while(**i **<** 10**)**

**{**

printf**(**"%d\n"**,**i**);**

time\_t ttime**;** time**(&**ttime**);**

struct tm **\***pTm **=** gmtime**(&**ttime**);**

syslog**(**LOG\_INFO**,**"%d %04d:%02d:%02d"**,** i**,** **(**1900 **+** pTm**->**tm\_year**),** **(**1 **+** pTm**->**tm\_mon**),** **(**pTm**->**tm\_mday**));**

i**++;**

sleep**(**2**);**

**}**

**}**

通过查看vim /var/log/messages

# Linux IPC 之管道

## 1 标准流管道

像文件操作有标准 io 流一样，管道也支持文件流模式。用来创建连接到另一进程的管道，是通过函数 popen 和 pclose。

函数原型：

#include<stdio.h>

FILE \*popen(const char \*command,const char \*open\_mode);

int pclose(FILE\* fp);

函数 popen()：允许一个程序将另一个程序作为新进程来启动，并可以传递数据给它或者通过它接收数据。如果fork或pipe调用失败，或内存分配失败，返回NULL。

参数command 字符串是要运行的程序名。

参数open\_mode 必须是“r”或“w”。 如果open\_mode 是“r”，被调用程序的输出就可以被调用程序(popen)使用，调用程序利用 popen 函数返回的FILE\*文件流指针，就可以通过常用的stdio 库函数（如 fread）来读取被调用程序的输出；如果 open\_mode 是“w”，调用程序(popen)就可以用 fwrite 向被调用程序发送数据，而被调用程序可以在自己的标准输入上读取这些数据。

函数 pclose()：用popen 启动的进程结束时，我们可以用pclose 函数关闭与之关联的文件流。出错返回-1，设置了标准错误描述，可使用perror函数查看。

Example1：从标准管道流中读取 打印/etc/profile 的内容

#include<stdio.h>

int main**()**

**{**

FILE **\***fp**=**popen**(**"cat /etc/profile"**,** "r"**);**

char buf**[**512**]={**0**};**

**while(**fgets**(**buf**,sizeof(**buf**),**fp**))**

**{**

puts**(**buf**);**

**}**

pclose**(**fp**);**

return0**;**

**}**

Example2：写到标准管道流 统计 buf 单词数(被调用程序必须阻塞等待标准输入)

#include<stdio.h>

int main**()**

**{**

char buf**[]={**"aaa bbb ccc ddd eee fff ggg hhh"**};**

FILE**\***fp**=**popen**(**"wc -w"**,**"w"**);**

fwrite**(**buf**,sizeof(**buf**),**1**,**fp**);**

pclose**(**fp**);**

return0**;**

**}**

## 2 无名管道(PIPE)

管道是 linux 进程间通信的一种方式，如命令 ps -ef|grep ntp

无名管道的特点：

1）只能在亲缘关系进程间通信（父子或兄弟）

2）半双工（固定的读端和固定的写端）

3）他是特殊的文件，可以用 read、write 等，只能在内存中

管道函数原型：

#include<unistd.h>

int pipe(int fds[2]);

管道在程序中用一对文件描述符表示，其中一个文件描述符有可读属性，一个有可写的属性。fds[0]是读，fds[1]是写。

函数pipe用于创建一个无名管道，如果成功，fds[0]存放可读的文件描述符，fds[1]存放可写文件描述符,并且函数成功返回0，失败时返回-1. 设置了标准错误描述，可使用perror函数查看。

通过调用 pipe 获取这对打开的文件描述符后，一个进程就可以从 fds[0]中读数据，而另一个进程就可以往 fds[1]中写数据。当然两进程间必须有继承关系，才能继承这对打开的文 件描述符。管道不象真正的物理文件，不是持久的，即两进程终止后，管道也自动消失了。



示例：创建父子进程，创建无名管道，父写子读

#include<stdio.h>

#include<string.h>

#include<unistd.h>

int main**()**

**{**

int fds**[**2**]={**0**};**

pipe**(**fds**);**

char szBuf**[**32**]={**'\0'**};**

**if(**fork**()==**0**)**

**{**

//表示子进程

close**(**fds**[**1**]);** //子进程关闭写

sleep**(**2**);** //确保父进程有时间关闭读，并且往管道中写内容

**if(**read**(**fds**[**0**],**szBuf**,sizeof(**szBuf**))>**0**)**

puts**(**sizebuf**);**

close**(**fds**[**0**]);** //子关闭读

exit**(**0**);**

**}**

**else**

**{**

//表示父进程

close**(**fds**[**0**]);** //父关闭读

write**(**fds**[**1**],** "hello"**,**6**);**

waitpid**(-**1**,NULL,** 0**);** //等子关闭读

//write(fds[1],"world",6); //此时将会出现“断开的管道”因为子的读已经关闭了

close**(**fds**[**1**]);** //父关闭写

exit**(**0**);**

**}**

return0**;**

**}**

管道两端的关闭是有先后顺序的，如果先关闭写端则从另一端读数据时， read函数将返回0，表示管道已经关闭；但是如果先关闭读端，则从另一端写数据时，将会使写数据的进程接收到SIGPIPE 信号，如果写进程不对该信号进行处理，将导致写进程终止，如果写进程处理了该信号，则写数据的 write 函数返回一个负值，表示管道已经关闭。

示例：

#include<signal.h>

#include<stdio.h>

#include<string.h>

#include<unistd.h>

int main**()**

**{**

int fds**[**2**];**

pipe**(**fds**);** //注释掉这部分将导致写进程被信号SIGPIPE终止

sigset\_t setSig**;**

sigemptyset**(&**setSig**);**

sigaddset**(&**setSig**,**SIGPIPE**);**

sigprocmask**(**SIG\_BLOCK**,&**setSig**,NULL);**

charszBuf**[**10**]={**0**};**

**if(**fork**()==**0**)**

**{**

close**(**fds**[**1**]);**//子关闭写

sleep**(**2**);**//确保父关闭读

**if(**read**(**fds**[**0**],**szBuf**,sizeof(**szBuf**))>**0**)**

puts**(**szBuf**);**

close**(**fds**[**0**]);**//子关闭读

**}**

**else**

**{**

close**(**fds**[**0**]);**//父关闭读

write**(**fds**[**1**],**"hello"**,**6**);**

wait**(NULL);**

write**(**fds**[**1**],**"world"**,** 6**);**//子的读关闭，父还在写

close**(**fds**[**1**]);**//父关闭写

**}**

return0**;**

**}**

## 3 命名管道(FIFO)

无名管道只能在亲缘关系的进程间通信大大限制了管道的使用，有名管道突 破了这个限制，通过指定路径名的范式实现不相关进程间的通信

**创建、删除 FIFO 文件**

创建 FIFO 文件与创建普通文件很类似，只是创建后的文件用于 FIFO。

1）用函数创建和删除 FIFO 文件

创建 FIFO 文件的函数原型：

#include<sys/types.h>

#include<sys/stat.h>

int mkfifo(const char\*pathname, mode\_t mode);

参数 pathname 为要创建的FIFO 文件的全路径名；

参数 mode 为文件访问权限

函数成功返回0，失败时返回-1. 设置了标准错误描述，可使用perror函数查看。

示例：

#include<sys/types.h>

#include<sys/stat.h>

#include<unistd.h>

#include<stdio.h>

int main**(**intargc**,**char**\***argv**[])**//演示通过命令行传递参数

**{**

**if(**argc**!=**2**)**

**{**

puts**(**"Usage: MkFifo.exe {filename}"**);**

**return-**1**;**

**}**

**if(**mkfifo**(**argv**[**1**],**0666**)==-**1**)**

**{**

perror**(**"mkfifofail"**);**

**return** **-**2**;**

**}**

return0**;**

**}**

删除文件（包括FIFO）的函数原型为：

#include<unistd.h>

int unlink(constchar\*pathname);

函数成功返回0，失败时返回-1. 设置了标准错误描述，可使用perror函数查看。

补充函数:

#include<unistd.h>

int link(const char \*path1, const char \*path2);

int unlinkat(int fd, const char \*path, int flag);

int linkat(int fd1, const char \*path1, int fd2, const char \*path2, int flag);

函数成功返回0，失败时返回-1. 设置了标准错误描述，可使用perror函数查看。

示例：

#include<unistd.h>

int main**()**

**{**

unlink**(**"pp"**);**

**}**

2）用命令创建和删除FIFO文件

用命令mkfifo创建 //不能重复创建 ·

用命令unlink删除

创建完毕之后，就可以访问FIFO文件了：

一个终端：cat < myfifo

另一个终端：echo “hello” > myfifo

**打开、关闭 FIFO 文件**

对 FIFO 类型的文件的打开/关闭跟普通文件一样，都是使用 open 和 close 函数。如果打开时使用 O\_WRONLY 选项，则打开 FIFO 的写入端，如果使用 O\_RDONLY 选项，则打开 FIFO 的读取端，写入端和读取端都可以被几个进程同时打开。 如果以读取方式打开 FIFO，并且还没有其它进程以写入方式打开 FIFO，open 函数将被阻塞；同样，如果以写入方式打开 FIFO，并且还没其它进程以读取方式打开 FIFO，open 函数也将被阻塞。 与 PIPE 相同，关闭 FIFO 时，如果先关读取端，将导致继续往 FIFO 中写数据的进程接收 SIGPIPE 的信号,如果不对信号处理则会崩溃。如果先关闭写端，read函数将返回0，表示管道已经关闭

**读写 FIFO**

可以采用与普通文件相同的读写方式读写 FIFO。

Example：先执行#mkfifo MyFifo.pip 命令

然后 vim write.c 如下：

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<stdio.h>

int main**()**

**{**

int fdFifo**=**open**(**"MyFifo.pip"**,**O\_WRONLY**);** //1. 打开（判断是否成功打开略）

write**(**fdFifo**,** "hello"**,** 6**);** //2. 写

close**(**fdFifo**);** //3. 关闭

return0**;**

**}**

然后 vi read**.**c 如下：

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<stdio.h>

int main**()**

**{**

charszBuf**[**128**];** int fdFifo**=**open**(**"MyFifo.pip"**,**O\_RDONLY**);** //1. 打开

**if(**read**(**fdFifo**,**szBuf**,sizeof(**szBuf**))>**0**)** //2. 读

puts**(**szBuf**);**

close**(**fdFifo**);** //3. 关闭

**return** 0**;**

**}**

然后：

gcc–o write write.c

gcc–oreadread.c

./write //发现阻塞，要等待执行./read

./read 在屏幕上输出 hello

管道示例：基于管道的客户端服务器程序。

程序说明：

1） 服务器端： 维护服务器管道，接受来自客户端的请求并处理（本程序为接受客户端发来的字符串， 将小写字母转换为大写字母。）然后通过每个客户端维护的管道发给客户端。

2） 客户端 向服务端管道发送数据，然后从自己的客户端管道中接受服务器返回的数据。

示例代码见下：

服务器端 server.c

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<unistd.h>

#include<fcntl.h>

#include<ctype.h>

**typedef** struct tagmag

**{**

int client\_pid**;**

char my\_data**[**512**];**

**}**MSG**;**

int main**()**

**{**

int server\_fifo\_fd **,**client\_fifo\_fd **;**

char client\_fifo**[**256**];**

MSG my\_msg **;**

char **\***pstr**;**

memset**(&**my\_msg **,**0**,sizeof(**MSG**));**

mkfifo**(**"SERVER\_FIFO\_NAME"**,**0777**);**

server\_fifo\_fd **=**open**(**"./SERVER\_FIFO\_NAME"**,**O\_RDONLY**);**

**if(**server\_fifo\_fd**==-**1**)**

**{**

perror**(**"server\_fifo\_fd"**);**

exit**(-**1**);**

**}**

int iret**;**

**while(** **(**iret**=**read**(**server\_fifo\_fd**,&**my\_msg **,sizeof(**MSG**))>**0**))**

**{**

//iret =read(server\_fifo\_fd,&my\_msg ,sizeof(MSG));

pstr**=**my\_msg**.**my\_data **;**

printf**(**"%s\n"**,**my\_msg**.**my\_data**);**

**while(\***pstr**!=** '\0'**)**

**{**

**\***pstr **=**toupper**(\***pstr**);**

pstr**++;**

**}**

memset**(**client\_fifo **,**0**,**256**);**

sprintf**(**client\_fifo**,**"CLIENT\_FIFO\_%d"**,**my\_msg**.**client\_pid**);**

client\_fifo\_fd **=**open**(**client\_fifo**,**O\_WRONLY**);**

**if(**client\_fifo\_fd **==-**1**)**

**{**

perror**(**"client\_fifo\_fd"**);**

exit**(-**1**);**

**}**

write**(**client\_fifo\_fd**,&**my\_msg**,** **sizeof(**MSG**));**

printf**(**"%s\n"**,**my\_msg**.**my\_data**);**

printf**(**"OVER!\n"**);**

close**(**client\_fifo\_fd**);**

**}**

return0**;**

**}**

客户端代码：client**.**c

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<unistd.h>

#include<fcntl.h>

**typedef** struct tagmag

**{**

int client\_pid**;**

charmy\_data**[**512**];**

**}**MSG**;**

int main**()**

**{**

int server\_fifo\_fd**,**client\_fifo\_fd**;**

char client\_fifo**[**256**]={**0**};**

sprintf**(**client\_fifo**,**"CLIENT\_FIFO\_%d"**,**getpid**());**

MSG my\_msg **;**

memset**(&**my\_msg **,**0**,sizeof(**MSG**));**

my\_msg**.**client\_pid **=**getpid**();**

server\_fifo\_fd **=**open**(**"./SERVER\_FIFO\_NAME"**,**O\_WRONLY**);**

mkfifo**(**client\_fifo **,**0777**);**

**while(**1**)**

**{**

int n**=**read**(**STDIN\_FILENO**,**my\_msg**.**my\_data **,**512**);**

my\_msg**.**my\_data**[**n**]=**'\0'**;**

write**(**server\_fifo\_fd **,&**my\_msg **,sizeof(**MSG**));**

client\_fifo\_fd**=**open**(**client\_fifo **,**O\_RDONLY**);**

//memset(&my\_msg ,0 ,sizeof(MSG));

n**=**read**(**client\_fifo\_fd**,&**my\_msg**,sizeof(**MSG**));**

my\_msg**.**my\_data**[**n**]=**0**;**

write**(**STDOUT\_FILENO**,**my\_msg**.**my\_data**,**strlen**(**my\_msg**.**my\_data**));**

close**(**client\_fifo\_fd**);**

**}**

unlink**(**client\_fifo**);**

**}**

# LinuxIPC 之共享内存

**SystemV 共享内存机制： shmget shmat shmdt shmctl**

原理及实现：system V IPC 机制下的共享内存本质是一段特殊的内存区域，进程间需要共享的数据被放在该共享内存区域中，所有需要访问该共享区域的进程都要把该共享区域映射到本进程的地址空间中去。这样一个使用共享内存的进程可以将信息写入该空间，而另一个使用共享内存的进程又可以通过简单的内存读操作获取刚才写入的信息，使得两个不同进程之间进行了一次信息交换，从而实现进程间的通信。共享内存允许一个或多个进程通过同时出现在它们的虚拟地址空间的内存进行通信，而这块虚拟内存的页面被每个共享进程的页表条目所引用，同时并不需要在所有进程的虚拟内存都有相同的地址。进程对象对于共享内 存的访问通过key（键）来控制，同时通过 key 进行访问权限的检查。

函数定义如下：

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/shm.h>

key\_t ftok(const char \*pathname, int proj\_id);

int shmget(key\_t key,int size,int shmflg);

void \*shmat(int shmid, const void\*shmaddr, int shmflg);

int shmdt(const void \*shmaddr);

int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);

函数 ftok 用于创建一个关键字，可以用该关键字关联一个共享内存段。如果调用成功，返回一关键字，否则返回-1。设置了标准错误描述，可使用perror函数查看。

参数pathname 为一个全路径文件名，并且该文件必须可访问。

参数proj\_id 通常传入一非0 字符 通过pathname 和 proj\_id 组合可以创建唯一的 key

函数 shmget 用于创建或打开一和已存在的共享内存段，该内存段由函数的第一个参数唯一创建。函数成功，则返回一个唯一的共享内存标识号（相当于进程号，唯一的标识着共享内存）， 出错返回-1。 设置了标准错误描述，可使用perror函数查看。

参数key 是一个与共享内存段相关联关键字，如果事先已经存在一个与指定关键字关联的共享内存段，则直接返回该内存段的标识，表示打开，如果不存在，则创建一个新的共享内存段。key 的值既可以用 ftok 函数产生，也可以是 IPC\_PRIVATE（用于创建一个只属于创建进程的共享内存，主要用于父子通信）,表示总是创建新的共享内存段；

参数size 指定共享内存段的大小，以字节为单位；

参数shmflg 是一掩码合成值，可以是访问权限值与(IPC\_CREAT 或 IPC\_EXCL)的 合成。IPC\_CREAT 表示如果不存在该内存段，则创建它。IPC\_EXCL 表示如果该内存段存在，则函数返回失败结果(-1)。

函数 shmat 将共享内存段映射到进程空间的某一地址。如果调用成功，返回映射后该进程中的的共享内存空间的首地址，否则返回(void\*)-1。 设置了标准错误描述，可使用perror函数查看。

参数shmid 是共享内存段的标识 通常应该是shmget 的成功返回值；

参数shmaddr 指定的是共享内存连接到当前进程中的地址位置。通常是 NULL，表示让系统来选择共享内存出现的地址。

参数shmflg 是一组位标识，通常为0 即可。

函数 shmdt 用于将共享内存段与进程空间分离。 函数成功返回0，失败时返回-1. 设置了标准错误描述，可使用perror函数查看。

参数shmaddr 通常为 shmat 的成功返回值。

注意，将共享内存分离并没删除它，只是使得该共享内存对当前进程不在可用。

函数 shmctl 是共享内存的控制函数，可以用来删除共享内存段。有进程连接，执行返回0，删除失败 。出错时返回-1，设置了标准错误描述，可使用perror函数查看。成功时返回值取决于cmd参数： IPC\_INFO 、SHM\_INFO返回内核内部数组记录的关于所有共享内存的最高索引，（该信息可与重复的SHM\_STAT操作一起使用来获取关于系统上所有共享内存的信息。），SHM\_STAT操作返回在shmid中给出的共享内存标识。其余命令返回0.

参数shmid 是共享内存段标识，通常应该是 shmget 的成功返回值 ；

参数cmd 是对共享内存段的操作方式，可选为 ：IPC\_STAT得到共享内存的状态把共享内存的shmid\_ds结构复制到buf中；IPC\_SET改变共享内存的状态，把buf所指的shmid\_ds结构中的uid、gid、mode复制到共享内存的shmid\_ds结构内,IPC\_RMID。通常为 IPC\_RMID，表示删除共享内存段。

参数buf 是表示共享内存段的信息结构体数据，通常为 NULL。例如：shmctl (kshareMem, IPC\_RMID, NULL )表示删除调共享内存段 kHareMem

shmid\_ds 结构体:

struct shmid\_ds

{

struct ipc\_perm shm\_perm; /\* 所有则者与操作权限\*/

size\_t shm\_segsz; /\*段的大小（以字节为单位） \*/

time\_t shm\_atime; /\*最后一个进程连接该shm的时间 \*/

time\_t shm\_dtime; /\*最后一个进程分离该shm的时间 \*/

time\_t shm\_ctime; /\*最后一个进程修改该shm的时间\*/

pid\_t shm\_cpid; /\*创建该shm的进程的pid \*/

pid\_t shm\_lpid; /\*最后链接或分离该shm的进程的pid \*/

shmatt\_t shm\_nattch; /\*当前附加到该shm的进程的个数\*/ ...

/\*以下为私有成员\*/

unsigned short shm\_unused; /\* compatibility \*/

void \*shm\_unused2; /\* ditto - used by DIPC \*/

void \*shm\_unused3; /\* unused \*/

};

struct ipc\_perm

{

key\_t \_\_key; /\*shmget中的参数key\*/

uid\_t uid; /\*所有者的有效用户ID \*/

gid\_t gid; /\*所有者的有效组ID \*/

uid\_t cuid; /\*创造者的有效用户ID \*/

gid\_t cgid; /\*创造者的有效组ID \*/

unsigned short mode; /\*Permissions+SHM\_DEST and SHM\_LOCKED flags\*/

unsigned short \_\_seq;/\*Sequence number\*/

};

ipc\_perm 的 mode 详解表

操作者 读 写（更改 更新）

用户 0400 0200

组 0040 0020

其他 0004 0002

查看系统共享内存 ipcs

删除共享内存 ipcrm-mshmid

示例：有亲缘关系

#include<stdio.h>

#include<string.h>

#include<errno.h>

#include<unistd.h>

#include<sys/stat.h>

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/shm.h>

#define PERM S\_IRUSR |S\_IWUSR //表示用户可读可写 即 0600

int main**(**int argc**,**char**\*\***argv**)**

**{**

/\*在两个有亲属关系进程间通信,KEY 采用 IPC\_PRIVATE 由系统自选\*/

int shmid **=**shmget**(**IPC\_PRIVATE**,**1024**,**PERM**);**

//只有IPC\_PRIVATE 情况可以不设置 IPC\_CREAT

**if(**shmid **==-**1**)**/\* 创建 byte 的共享内存\*/

**{**

fprintf**(**stderr**,**"Create Share Memory Error:%s\n\a"**,**strerror**(**errno**));**

exit**(**1**);**

**}**

**if(**fork**()>**0**)** /\* 父进程代码\*/

**{**

char **\***p\_addr **=(**char**\*)**shmat**(**shmid**,NULL,**0**);** //获得该段共享内存的首地址

memset**(**p\_addr**,**'\0'**,**1024**);** //初始化为 0

strncpy**(**p\_addr**,**"sharememory"**,** 1024**);** //存入（写入）内容

printf**(**"parent %d Write buffer:%s\n"**,**getpid**(),**p\_addr**);**

sleep**(**2**);**

wait**(NULL);** //防止僵尸进程

shmctl**(**shmid**,**IPC\_RMID**,** 0**);**/\*删除共享内存,用ipcs -m看共享内存\*/

exit**(**0**);**

**}**

**else**/\* 子进程代码\*/

**{**

sleep**(**5**);** //让父有足够的时间写

char **\***c\_addr**=(**char**\*)**shmat**(**shmid**,NULL,**0**);** //取出（读出）内容

printf**(**"Client pid=%d,shmid =%d Readbuffer:%s\n"**,**getpid**(),**shmid**,**c\_addr**);**

exit**(**0**);**

**}**

**}**

示例:非亲进程间通信的实现步骤如下：

写内存端

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<errno.h>

#include<sys/ipc.h>

#include<sys/shm.h> //头文件包含

#include<sys/types.h>

int main**()**

**{**

key\_t key**=**ftok**(**"b.dat"**,**1**);** //写入端先用 ftok 函数获得 key

**if(**key**==-**1**)**

**{**

perror**(**"ftok"**);**

exit**(-**1**);**

**}**

int shmid **=**shmget**(**key**,**4096**,**IPC\_CREAT**);** //写入端用shmget函数创建一共享内存段

**if(**shmid**==-**1**)**

**{**

perror**(**"shmget"**);**

exit**(-**1**);**

**}**

char **\***pMap**=(**char**\*)**shmat**(**shmid**,** **NULL,**0**);** //获得共享内存段的首地址

memset**(**pMap**,**0**,**4096**);**

memcpy**(**pMap**,**"helloworld"**,**4096**);** //往共享内存段中写入内容

**if(**shmdt**(**pMap**)==-**1**)** //分离共享内存段

**{**

perror**(**"shmdt"**);**

exit**(-**1**);**

**}**

**}**

读内存端：

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<errno.h>

#include<sys/ipc.h>

#include<sys/shm.h>

#include<sys/types.h>

int main**()**

**{**

key\_tkey**=**ftok**(**"b.dat"**,**1**);** //读入端用 ftok 函数获得 key

**if(**key**==-**1**)**

**{**

perror**(**"ftok"**);**

exit**(-**1**);**

**}**

int shmid **=**shmget**(**key**,**4096**,**0600**|**IPC\_CREAT**);** //读入端用shmget函数打开共享内存段

**if(**shmid**==-**1**)**

**{**

perror**(**"shmget"**);**

exit**(-**1**);**

**}**

char **\***pMap**=(**char**\*)**shmat**(**shmid**,** **NULL,**0**);** //获得共享内存段的首地址

printf**(**"receive the data:%s\n"**,**pMap**);** //读取共享内存段中的内容

**if(**shmctl**(**shmid**,** IPC\_RMID**,**0**)==-**1**)** // 删除共享内存段

**{**

perror**(**"shmctl"**);**

exit**(-**1**);**

**}**

**}**

示例，通过共享内存实现两个程序间的对话。

程序1，shmwr**.**c

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<unistd.h>

#include<sys/ipc.h>

#include<sys/shm.h>

struct text

**{**

int useful**;**

char buf**[**1024**];**

**};**

int main**()**

**{**

int shmid **=**shmget**((**key\_t**)**5080**,sizeof(**struct text**),**0600**|**IPC\_CREAT**);**

printf**(**"%d\n" **,**shmid**);**

struct text **\***ptext **=(**struct text **\*)**shmat**(**shmid**,NULL,**0**);**

ptext**->**useful **=**0**;**

**while(**1**)**

**{**

**if(**ptext **->**useful **==**0**)**

**{**

int iret**=**read**(**STDIN\_FILENO**,**ptext**->**buf**,**1024**);**

ptext**->**useful**=**1**;**

**if(**strncmp**(**"end"**,**ptext**->**buf**,**3**)==**0**)**

**{**

**break;**

**}** //ptext ->useful =0;

**}**

sleep**(**1**);**

**}**

shmdt**((**void**\*)**ptext**);**

**return** 0**;**

**}**

程序2**:**shmrd**.**c

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<unistd.h>

#include<sys/ipc.h>

#include<sys/shm.h>

struct text

**{**

intuseful**;**

charbuf**[**1024**];**

**};**

int main**()**

**{**

int shmid **=**shmget**((**key\_t**)**5080**,sizeof(**structtext**),**0600**|**IPC\_CREAT**);**

struct text **\***ptext **=(**struct text **\*)**shmat**(**shmid**,NULL,**0**);**

ptext**->**useful**=**0**;**

**while(**1**)**

**{**

**if(**ptext **->**useful **==**1**)**

**{**

write**(**STDOUT\_FILENO**,**ptext **->**buf**,**strlen**(**ptext**->**buf**));**

ptext**->**useful**=**0**;**

**if(**strncmp**(**"end"**,**ptext**->**buf**,**3**)==**0**)**

**{**

**break;**

**}**

**}**

sleep**(**1**);**

**}**

shmdt**((**void**\*)**ptext**);**

shmctl**(**shmid**,**IPC\_RMID**,**0**);**

return0**;**

**}**

# LinuxIPC 之信号量

信号量（也叫信号灯）是一种用于提供不同进程间或一个给定进程的不同线程间同步手段的原语。信号量是进程/线程同步的一种方式，有时候我们需要保护一段代码，使它每次只能被一个执行进程/线程运行，这种工作就需要一个二进制开关；有时候需要限制一段代码可以被多少个进程/线程执行，这就需要用到计数信号量。信号量开关是二进制信号量的一种逻辑扩展，两者实际调用的函数都是一样。

信号量分为以下三种：

1）System V 信号量，在内核中维护，可用于进程或线程间的同步，常用于进程的同步。

2）Posix 有名信号量，一种来源于 POSIX 技术规范的实时扩展方案（POSIX Realtime Extension）,可用于进程或线程间的同步，常用于线程。

3）Posix 基于内存的信号量，存放在共享内存区中，可用于进程或线程间的同步。

为了获得共享资源进程需要执行下列操作：

（1）测试控制该资源的信号量。

（2）若信号量的值为正，则进程可以使用该资源。进程信号量值减1，表示它使用了一个资源单位。此进程使用完共享资源后对应的信号量会加 1。以便其他进程使用。

（3）若信号量的值为 0，则进程进入休息状态，直至信号量值大于 0。进程被唤醒，返回第 （1）步。

为了正确地实现信号量，信号量值的测试值的测试及减 1 操作应当是原子操作（原子操作是不可分割的，在执行完毕前不会被任何其它任务或事件中 断）。为此信号量通常是在内核中实现的。

SystemVIPC 机制：信号量。

函数原型：

#include<sys/sem.h>

#include<sys/ipc.h>

#include<sys/types.h>

key\_t ftok(const char \*pathname, int proj\_id);

int semget(key\_t key,int nsems,int flag);

int semop(int semid,struct sembuf \*sops,size\_t num\_sops);

int semtimedop(int semid, struct sembuf \*sops, size\_t nsops,const struct timespec \*timeout);

int semctl(int semid,int semnum, int cmd, …);

函数 semget 创建一个信号量集或访问一个已存在的信号量集。返回值：成功时，返回一个称为信号量标识符的整数，semop 和 semctl 会使用它；出错时，返回-1，设置了标准错误描述，可使用perror函数查看。

参数key 是唯一标识一个信号量的关键字，如果为 IPC\_PRIVATE(值为 0，创建一个只有创建者进程才可以访问的信号量，通常用于父子进程之间；非 0 值的key(可以通 过 ftok 函数获得)表示创建一个可以被多个进程共享的信号量；

参数nsems 指定需要使用的信号量数目。如果是创建新集合，则必须制定 nsems,一般取1。 如果引用一个现存的集合，则将 nsems 指定为 0.

参数flag 是一组标志，其作用与 open 函数的各种标志很相似。它低端的九个位是该信号量的权限，其作用相当于文件的访问权限。此外，它们还可以与键值 IPC\_CREAT 按位或操作，以创建一个新的信号量。即使在设置了IPC\_CREAT 标志后给出的是一个现有的信号量的键字，也并不是一个错误。我们也可以通过 IPC\_CREAT 和 IPC\_EXCL 标志的联合使用确保自己将创建出一个新的独一无二的信号量来，如果该信号量已经存在，就会返回一个错误。

函数 semop 与semtimedop用于改变信号量对象中各个信号量的状态。返回值：成功时，返回 0；出错时返回-1，设置了标准错误描述，可使用perror函数查看。

参数semid 是由semget函数返回的信号量标识符。

参数 sops 是指向一个结构体数组的指针。每个数组元素至少包含以下几个成员：

struct sembuf

{

short sem\_num; //要操作信号量在信号量集合中的编号，第一个信号量的编号是0。

short sem\_op; //sem\_op 成员的值是信号量在一次操作中需要改变的数值。通常只会用到两个

//值，一个是-1，也就是 p 操作，它等待信号量变为可用；一个是+1，也就是 v操作，它发

//送信号通知信号量现在可用。

short sem\_flg; //通常设为：SEM\_UNDO。IPC\_NOWAIT对信号的操作不能满足时，semop()

//不会阻塞，并立即返回，同时设定错误信息。SEM\_UNDO程序结束时(不论正常或不正

//常)，保证信号值会被重设为semop()调用前的值。这样做的目的在于避免程序在异常情况

//下结束时未将锁定的资源解锁，造成该资源永远锁定。

};

参数num\_sops信号操作结构的数量，恒大于或等于1。

函数semctl用来直接控制信号量信息。 返回值：出错时返回-1，设置了标准错误描述，可使用perror函数查看。成功时返回值取决于cmd参数： GETNCNT、GETPID 、GETVAL、 GETZCNT 返回对应的值， IPC\_INFO 、SEM\_INFO返回内核内部数组记录的关于所有信号量集合的最高索引，（该信息可与重复的SEM\_STAT操作一起使用来获取关于系统上所有信号量集合的信息。），SEM\_STAT操作返回在semid中给出其信号量集合标识。其余命令返回0.

参数semid 是由semget 返回的信号量标识符。

参数semnum 为集合中信号量的编号，当要用到成组的信号量时，从 0 开始。一般取值为 0，表示这是第一个也是唯一的一个信号量。

参数 cmd 为执行的操作。通常为：

·IPC\_STAT读取一个信号量集的状态数据结构semid\_ds，并将其存储在semun中的buf参数中。参数 semnum 被忽略。调用进程对必须在信号量集合有读权限。

·IPC\_SET设置信号量集的状态数据结构semid\_ds中的元素ipc\_perm，其值取自semun中的buf参数。同时更新 sem\_ctime 成员。结构中下列成员被更新：sem\_perm.uid、sem\_perm.gid 以及 sem\_perm.mode (低端 9 位)。调用进程的有效用户ID必须匹配信号量集合的所有者(sem\_perm.uid)或创建者(sem\_perm.cuid)，或者调用者必须有特权。参数 semnum 被忽略。

·IPC\_RMID将信号量集从内存中删除。立即删除信号量集合，唤醒所有因调用 semop（） 阻塞在该信号量集合里的所有进程(相应调用会返回错误且 errno 被设置为 EIDRM)。调用进程的有效用户ID必须匹配信号量集合的创建者或所有者，或者调用者必须有特权。参数 semnum 被忽略。

·IPC\_INFO (Linux特别定义) 通过buf 指向的数据结构返回系统范围内的信号量限制和参数。这个数据结构的类型是 seminfo，如果宏 \_GNU\_SOURCE 特性宏被定义，则该结构定义在头文件 <sys/sem.h> 。

struct seminfo

{

int semmap; // 信号量映射里的条数，内核未使用

int semmni; // 信号量集合的最大个数

int semmns; // 在所有信号量集合里信号量个数上限

int semmnu; // 系统范围内的 undo 结构最大个数，内核未使用

int semmsl; // 一个信号量集合里信号量个数上限

int semopm; // 执行的最大操作个数

int semume; // 每个进程内 undo 结构最大个数，内核未使用

int semusz; // 结构 sem\_undo 的尺寸

int semvmx; // 信号量值的上限

int semaem; // Max. value that can be recorded for semaphore adjustment (SEM\_UNDO)

};

semmsl、semmns、semopm 和 semmni 设置可以通过 /proc/sys/kernel/sem 更改

·GETALL用于读取信号量集中的所有信号量的值。获取所有信号量的值，第二个参数semnum为 0，将所有信号的值存入semun.array 中。

·GETNCNT返回正在等待资源的进程数目。

·GETPID返回最后一个执行semop操作的进程的PID。

·GETVAL返回信号量集中的一个单个的信号量的值。第四个参数被忽略

·GETZCNT返回正在等待完全空闲的资源的进程数目。

·SETALL设置信号量集中的所有的信号量的值。设置所有信号量的值，第二个参数semnum为 0，所有信号的值由semun.array 中设置。

·SETVAL设置信号量集中的一个单独的信号量的值。根据semun.val设定信号的值，从 0 开始，第一个信号量编号为 0。

4）参数…是一个 union semun（需要由程序员自己定义），它至少包含以下几个成员：

union semun

{

int val; /\*Value for SETVAL\*/

struct semid\_ds \*buf; /\*Buffer for IPC\_STAT, IPC\_SET\*/

unsigned short \*array; /\*Array for GETALL,SETALL\*/

};

通常情况仅使用 val，给val 赋值为 1

The semid\_ds data structure is defined in<sys/sem.h> as follows:

struct semid\_ds

{

struct ipc\_perm sem\_perm; /\*Ownership and permissions\*/

time\_t sem\_otime; /\*最后一次pv操作的时间 \*/

time\_t sem\_ctime; /\*最后一次修改sem的时间\*/

unsigned long sem\_nsems; /\*信号量级集合中信号量个数\*/

};

ipc\_perm结构体定义如下(the highlighted fields are settable using IPC\_SET):

struct ipc\_perm//与共享类存一样

{

key\_t \_\_key; /\*Key supplied to semget(2)\*/

uid\_t uid; /\*Effective UID of owner\*/

gid\_t gid; /\*Effective GID of owner\*/

uid\_t cuid; /\*Effective UIDof creator\*/

gid\_t cgid; /\*Effective GIDof creator\*/

unsigned short mode; /\*Permissions\*/

unsignedshort \_\_seq;/\*Sequence number\*/

};

示例：有亲缘关系的信号量进程间通信

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<errno.h>

#include<sys/sem.h>

#include<sys/ipc.h>

#include<sys/types.h>

#include<string.h>

union semun //必须重写这个共用体

**{**

int val**;**

struct semid\_ds **\***buf**;**

unsignedshort**\***array**;**

**};**

int main**()**

**{**

int semid**=**semget**(**IPC\_PRIVATE**,**1**,**0666**|**IPC\_CREAT**);**//创建信号量集

**if(**semid **==-**1**)**

**{**

perror**(**"semgeterror"**);**

exit**(-**1**);**

**}**

**if(**fork**()==**0**)** //表示子进程

**{**

struct sembuf sem**;** //定义信号量结构体

memset**(&**sem**,** 0**,sizeof(**struct sembuf**));**

sem**.**sem\_num **=**0**;** //信号量的编号,第一个为 0

sem**.**sem\_op **=**1**;** //+1 表示执行 V 操作,生产产品

sem**.**sem\_flg **=**0**;** //或写SEM\_UNDO

union semun arg**;**

arg**.**val **=**0**;** //初始化数据

semctl**(**semid**,**0**,**SETALL**,** arg**);** //将数据全部设置到信号量集里面去,相当于公共数据

**while(**1**)**

**{**

semop**(**semid**,&**sem**,** 1**);**//执行指定的V 操作,表示生产产品

printf**(**"product total number:%d\n"**,**semctl**(**semid**,**0**,**GETVAL**));**//获得公共值

sleep**(**1**);**

**}**

**}**

**else**

**{**

sleep**(**2**);** //先让子进程有时间生产

struct sembuf sem**;** //定义信号量结构体

memset**(&**sem**,** 0**,sizeof(**structsembuf**));**

sem**.**sem\_num **=**0**;** //信号量的编号,第一个为 0

sem**.**sem\_op **=-**1**;** //-1 表示执行 P 操作,消费产品

sem**.**sem\_flg **=**0**;** //或写SEM\_UNDO

**while(**1**)**

**{**

semop**(**semid**,&**sem**,** 1**);**//执行指定的 P 操作消费产品

printf**(**"costomer total number:%d\n"**,**semctl**(**semid**,**0**,**GETVAL**));**

//获得公共值

sleep**(**2**);**

**}**

**}**

**}**

示例：没有亲缘关系的生产者消费者问题

生产者源码：

#include<unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/sem.h>

#include<errno.h>

void init**();** //initlization semaphore

void del**();** //delete semaphore

int sem\_id**;**

int main**(**int argc**,**char**\***argv**[])**

**{**

struct sembuf sops**[**2**];** /\*set operatewayforsemaphore\*/

sops**[**0**].**sem\_num **=**0**;** //第一个信号的编号，表示生产了几个产品

sops**[**0**].**sem\_op **=**1**;** //进行V 操作

sops**[**0**].**sem\_flg**=**0**;** //或写为 SEM\_UNDO

ops**[**1**].**sem\_num **=**1**;** //第二个信号编号，表示还可以生产几个产品

sops**[**1**].**sem\_op **=-**1**;** //进行P 操作

sops**[**1**].**sem\_flg**=**0**;** //或写为 SEM\_UNDO

init**();** //初始化操作

printf**(**"this isaproducor\n"**);**

**while(**1**)**

**{**

printf**(**"\n\n before produce\n"**);**

printf**(**"productornumberis%d\n"**,** semctl**(**sem\_id**,** 0**,**GETVAL**));**

printf**(**"space numberis%d\n"**,** semctl**(**sem\_id**,**1**,**GETVAL**));**

semop**(**sem\_id**,(**struct sembuf**\*)&**sops**[**1**],**1**);**

printf**(**"nowproducing .....\n"**);**

semop**(**sem\_id**,(**struct sembuf**\*)&**sops**[**0**],**1**);**

printf**(**"space number is%d\n"**,** semctl**(**sem\_id**,**1**,**GETVAL**));**

printf**(**"productor number is%d\n"**,** semctl**(**sem\_id**,** 0**,**GETVAL**));**

sleep**(**2**);**

**}**

del**();**

**}**

void init**()**

**{**

int ret**;**

unsigned short sem\_array**[**2**];**

union semun

**{**

int val**;**

struct semid\_ds **\***buf**;**

unsignedshort **\***array**;**

**}**arg**;**

sem\_id**=**semget**((**key\_t**)**1234**,**2**,**IPC\_CREAT**|**0644**);**/\*get semaphoreinclude two sem\*/

/\*set sem's vaule\*/

sem\_array**[**0**]=**0**;**

sem\_array**[**1**]=**10**;**

arg**.**array**=**sem\_array**;**

ret**=**semctl**(**sem\_id**,** 0**,**SETALL**,** arg**);** //将semun.array的值设定到信号集中，第二个参数为 0

**if(**ret**==-**1**)**

**{**

printf**(**"SETALL failed(%d)\n"**,**errno**);**

**}**

printf**(**"productorinit is%d\n"**,** semctl**(**sem\_id**,**0**,**GETVAL**));**

printf**(**"space init is%d\n\n"**,** semctl**(**sem\_id**,**1**,**GETVAL**));**

**}**

void del**()**

**{**

semctl**(**sem\_id**,**IPC\_RMID**,**0**);** //删除信号集

**}**

消费者源码：

#include<unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <errno.h>

void init**();**

int sem\_id**;**

int main**(**int argc**,**char**\***argv**[])**

**{**

structsembufsops**[**2**];** /\*set operatewayforsem\*/

sops**[**0**].**sem\_num **=**0**;**

sops**[**0**].**sem\_op **=-**1**;** //P 操作，表示有多少个产品可以消费（相当于生产了多少个产品）

sops**[**0**].**sem\_flg**=**0**;**

sops**[**1**].**sem\_num **=**1**;**

sops**[**1**].**sem\_op **=**1**;** //V 操作，表示还可以生成的产品数

sops**[**1**].**sem\_flg**=**0**;**

init**();**

printf**(**"this isacustomer\n"**);**

**while(**1**)**

**{**

printf**(**"\n\nbeforeconsume\n"**);**

printf**(**"customernumberis%d\n"**,**semctl**(**sem\_id**,**0**,**GETVAL**));**

printf**(**"space numberis%d\n"**,** semctl**(**sem\_id**,**1**,**GETVAL**));**

semop**(**sem\_id**,&**sops**[**0**],**1**);**

printf**(**"nowconsume .....\n"**);**

semop**(**sem\_id**,&**sops**[**1**],**1**);**

printf**(**"space numberis%d\n"**,** semctl**(**sem\_id**,**1**,**GETVAL**));**

printf**(**"customernumberis%d\n"**,**semctl**(**sem\_id**,**0**,**GETVAL**));**

sleep**(**1**);**

**}**

**}**

void init**()**

**{**

sem\_id**=**semget**((**key\_t**)**1234**,**2**,**IPC\_CREAT**|**0644**);**/\*get semaphoreinclude two sem\*/

**}**

在生产者源码里，首先用函数 semctl()初始化信号量集合 sem\_id，它包含两个信号，分 别表示生产的数量和空仓库的数量，那么在消费者的进程中用相同的key 值就会得到该信号 量集合；实现两个进程之间的通信。

在主函数里，设定对两个信号量的 PV 操作，然后在各自的进程中对两个信号进行操作。

（1）如果只运行生产者进程，则生产10 个之后，该进程就会因为在得不到空仓库资源而阻 塞，这个时候运行消费者进程，阻塞就会被解除；

（2）如果先运行生产者进程，生产几个产品之后，关闭该进程，则运行消费者进程，当消 费完生产的产品后，该进程就会因为在得不到产品资源而阻塞，这个时候运行生产者进程， 阻塞就会被解除；

（3）如果同时运行两个进程，由于消费比生产快，因此消费者每次都要等待生产者生产产 品之后才能消费； 在每次运行程序之前，一定要先运行生产者进程先初始化信号量。

# LinuxIPC 之消息队列

## 1 消息队列

消息队列与FIFO 很相似，都是一个队列结构，都可以有多个进程往队列里面写信息，多个进程从队列中读取信息。但FIFO 需要读、写的两端事先都打开，才能够开始信息传递工作。而消息队列可以事先往队列中写信息，需要时再打开读取信息。但是，消息队列先打开读，仍然会阻塞，因为此时没有消息可读。

## 2 System V IPC消息队列。

函数原型：

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgget(key\_t key, int msgflg);

int msgsnd(int msqid, struct msgbuf \*msgp, size\_t msgsz, int msgflg);

ssize\_t msgrcv(int msqid, struct msgbuf \*msgp, size\_t msgsz, long msgtyp, int msgflg);

int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);

函数msgget 创建和访问一个消息队列。该函数成功则返回一个唯一的消息队列标识符（类似于进程ID 一样），失败则返回-1,设置了标准错误描述，可使用perror函数查看。

参数key 是唯一标识一个消息队列的关键字，如果为IPC\_PRIVATE(值为0)，用创建一个只有创建者进程才可以访问的消息队列，可以用于父子间通信；非0 值的key(可以通过ftok 函数获得)表示创建一个可以被多个进程共享的消息队列；

参数msgflg 指明队列的访问权限和创建标志，创建标志的可选值为IPC\_CREAT和IPC\_EXCL 如果单独指定IPC\_CREAT,msgget 要么返回新创建的消息队列id,要么返回具有相同key 值的消息队列id；如果IPC\_EXCL 和IPC\_CREAT 同时指明，则要么创建新的消息队列，要么当队列存在时，调用失败并返回-1。

函数msgsnd 和msgrcv 用来将消息添加到消息队列中和从一个消息队列中获取信息。成功msgrcv返回字节数，msgsnd返回0，失败都返回-1，设置了标准错误描述，可使用perror函数查看。

参数msgid 指明消息队列的ID; 通常是msgget 函数成功的返回值。

参数msgbuf 是消息结构体，它的长度必须小于系统规定的上限，必须以一个长整型成员变量开始，接收函数将用这个成员变量来确定消息的类型。必须重写这个结构体，其中第一个参数不能改，其他自定义。如下：

struct msgbuf

{

long mtype; /\* type of message \*/

char mtext[1]; /\* message text \*/

};

字段mtype 是用户自己指定的消息类型（通常是0—5 中的任意一个数值），该结构体第2 个成员仅仅是一种说明性的结构，实际上用户可以使用任何类型的数据，就是消息内容；

参数msgsz 是消息体的大小，每个消息体最大不要超过4K;

参数msgflg 可以为0（通常为0）或IPC\_NOWAIT，如果设置IPC\_NOWAIT,则msgsnd 和msgrcv 都不会阻塞，此时如果队列满并调用msgsnd 或队列空时调用msgrcv将返回错误；

参数msgtyp 有3 种选项：

msgtyp == 0 接收队列中的第1 个消息（通常为0）

msgtyp > 0 接收对列中的第1 个类型等于msgtyp 的消息

msgtyp < 0 接收其类型小于或等于msgtyp 绝对值的第1 个最低类型消息

函数msgctl 是消息队列的控制函数，常用来删除消息队列。成功后，IPC\_STAT，IPC\_SET和IPC\_RMID返回0.成功的IPC\_INFO或MSG\_INFO操作返回内核内部数组记录有关所有消息队列的的最大值（该信息可与重复的MSG\_STAT操作一起使用， 关于系统上所有队列的信息。）成功的MSG\_STAT操作返回在msqid中给出的队列的标识符。失败都返回-1，设置了标准错误描述，可使用perror函数查看。

参数msqid 是由msgget 返回的消息队列标识符。

参数cmd 通常为IPC\_RMID 表示删除消息队列。

参数buf 通常为NULL 即可。

示例：有亲缘关系的消息队列（IPC\_PRIVATE）：

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <sys/stat.h>

struct MSG

**{**

long mtype**;**

char buf**[**64**];**

**};**

int main**()**

**{**

int msgid **=** msgget**((**key\_t**)**1234**,** 0666 **|** IPC\_CREAT**);**

**if(**msgid **==** **-**1**)**

**{**

perror**(**"msgget error"**);**

exit**(-**1**);**

**}**

struct MSG msg**;**

memset**(&**msg**,**0**,sizeof(**struct MSG**));**

**if(**fork**()** **>** 0**)**

**{**

msg**.**mtype **=** 1**;**

strcpy**(**msg**.**buf**,**"hello"**);**

msgsnd**(**msgid**,&**msg**,sizeof(**struct MSG**),**0**);**

wait**(NULL);**

msgctl**(**msgid**,** IPC\_RMID**,** **NULL);**

exit**(**0**);**

**}**

**else**

**{**

sleep**(**2**);** //让父进程有时间往消息队列里面写

msgrcv**(**msgid**,** **&**msg**,** **sizeof(**struct MSG**),** 1**,** 0**);**

puts**(**msg**.**buf**);**

exit**(**0**);**

**}**

**}**

示例：没有亲缘关系

消息发送

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <sys/stat.h>

#define BUFFER 255

struct msgtype

**{** //重新定义该结构体

long mtype**;** //第一个参数不变

char buffer**[**BUFFER**+**1**];**

**};**

int main**(**int argc**,**char **\*\***argv**)**

**{**

int msgid **=** msgget**((**key\_t**)**1234**,**0666 **|** IPC\_CREAT**);** //获取消息队列

**if(**msgid **==** **-**1**)**

**{**

fprintf**(**stderr**,**"Creat Message Error:%s\a\n"**,**strerror**(**errno**));**

exit**(**1**);**

**}**

struct msgtype msg**;**

memset**(&**msg**,**0**,sizeof(**struct msgtype**));**

msg**.**mtype **=** 1**;** //给结构体的成员赋值

strncpy**(**msg**.**buffer**,**"hello"**,**BUFFER**);**

msgsnd**(**msgid**,&**msg**,sizeof(**struct msgtype**),**0**);** //发送信号

memset**(&**msg**,**0**,sizeof(**struct msgtype**));** //清空结构体

msgrcv**(**msgid**,&**msg**,sizeof(**struct msgtype**),**2**,**0**);** //接收信号

fprintf**(**stdout**,**"Client receive:%s\n"**,**msg**.**buffer**);**

exit**(**0**);**

**}**

消息接收

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <errno.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/stat.h>

#include <sys/msg.h>

#define BUFFER 255

struct msgtype

**{** //重定义消息结构体

long mtype**;**

char buffer**[**BUFFER**+**1**];**

**};**

int main**()**

**{**

int msgid **=** msgget**((**key\_t**)**1234**,** 0666 **|** IPC\_CREAT**);** //获得消息队列

**if(**msgid **==** **-**1**)**

**{**

fprintf**(**stderr**,**"Creat Message Error:%s\a\n"**,**strerror**(**errno**));**

exit**(**1**);**

**}**

struct msgtype msg**;**

memset**(&**msg**,**0**,sizeof(**struct msgtype**));**

**while(**1**)**

**{**

msgrcv**(**msgid**,&**msg**,sizeof(**struct msgtype**),**1**,**0**);** //接收消息

fprintf**(**stdout**,**"Server Receive:%s\n"**,**msg**.**buffer**);**

msg**.**mtype **=** 2**;**

strncpy**(**msg**.**buffer**,**"world"**,**BUFFER**);**

msgsnd**(**msgid**,&**msg**,sizeof(**struct msgtype**),**0**);** //发送消息

**}**

exit**(**0**);**

**}**

# 对LINUX内核各种锁的理解

## 1 自旋锁spin lock

在linux kernel的实现中，经常会遇到这样的场景：共享数据被中断上下文和进程上下文访问，该如何保护呢？如果只有进程上下文的访问，那么可以考虑使用semaphore或者mutex的锁机制，但是现在中断上下文也参和进来，那些可以导致睡眠的lock就不能使用了，这时候，可以考虑使用spin lock。

spin lock的特点

我们可以总结spin lock的特点如下：

（1）spin lock是一种死等的锁机制。当发生访问资源冲突的时候，可以有两个选择：一个是死等，一个是挂起当前进程，调度其他进程执行。spin lock是一种死等的机制，当前的执行thread会不断的重新尝试直到获取锁进入临界区。

（2）只允许一个thread进入。semaphore可以允许多个thread进入，spin lock不行，一次只能有一个thread获取锁并进入临界区，其他的thread都是在门口不断的尝试。

（3）执行时间短。由于spin lock死等这种特性，因此它使用在那些代码不是非常复杂的临界区（当然也不能太简单，否则使用原子操作或者其他适用简单场景的同步机制就OK了），如果临界区执行时间太长，那么不断在临界区门口“死等”的那些thread是多么的浪费CPU啊（当然，现代CPU的设计都会考虑同步原语的实现，例如ARM提供了WFE和SEV这样的类似指令，避免CPU进入busy loop的悲惨境地）

（4）可以在中断上下文执行。由于不睡眠，因此spin lock可以在中断上下文中适用。

自旋锁是内核中最基础的锁机制。自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元持有，调用者就一直循环在那里看是否该自旋锁的持有者已经释放了锁，"自旋"一词就是因此而得名。

自旋锁适用于锁使用者保持锁时间比较短的情况。

使用自旋锁需要注意有可能造成的死锁情况:

static DEFINE\_SPINLOCK(xxx\_lock);

unsigned long flags;

spin\_lock\_irqsave(&xxx\_lock, flags);

... critical section here ..

spin\_unlock\_irqrestore(&xxx\_lock, flags);

代码中spin\_lock\_irqsave会禁止本地cpu中断的抢占。以上代码在任何情况下都是安全的。但问题是关中断的代价太大。

如果把spin\_lock\_irqsave/spin\_unlock\_irqrestore换成spin\_lock/spin\_unlock会有什么问题吗？

答案是，如果中断中调用了spin\_lock，可能会引起死锁！

例如：

spin\_lock(&lock);

...

<- interrupt comes in:

spin\_lock(&lock);

值得注意的是，如果产生中断的cpu和进程中调用spin\_lock的cpu不是同一个，则不会有问题。这也是irq版本的spin\_lock函数实现时只需要禁止本地cpu中断的原因。

结论：要想在进程中用spin\_lock代替spin\_lock\_irqsave，条件是中断中不会使用相应的spin\_lock

何时使用自旋锁？

不允许睡眠的上下文且临界区操作较短时使用自旋锁。

## 2 读写自旋锁rwlock

如果读写锁当前没有读者，也没有写者，那么写者可以立刻获得读写锁，否则它必须自旋在那里，直到没有任何写者或读者。如果读写锁没有写者，那么读者可以立即获得该读写锁，否则读者必须自旋在那里，直到写者释放该读写锁。

读写锁适合于对数据结构的读次数比写次数多得多的情况。

注意：读写锁需要比spin locks更多的访问原子内存操作，如果读临界区不是很大，最好别使用读写锁。

读写锁比较适合链表等数据结构，特别是查找远多于修改的情况。

另外，可以灵活的使用read-write和irq版本的自旋锁。例如，如果中断中只是用了读锁，进程中就可以使用non-irq版本的读锁和irq版本的写锁。

注意：RCU比读写锁更适合遍历list，但需要更关注细节。目前kernel社区正在努力用RCU代替读写锁。

## 3 信号量semaphore

semaphore和spin lock的区别是semaphore会引起睡眠。

查看semaphore的数据结构可以发现，semaphore除了拥有spinlock，还有一个计数器和一个等待队列。当某个进程获取信号量的count值小于等于0时，被添加到wait\_list中。

struct semaphore

{

raw\_spinlock\_t lock;

unsigned int count;

struct list\_head wait\_list;

};

何时使用semaphore？

允许睡眠的上下文、临界区操作较长、计数值大于1时使用semaphore

信号量也有读写信号量，在此略过。

## 4 mutex

mutex可以理解成计数值只有0和1的semaphore

既然有了semaphore，内核为何还需要mutex？

因为内核中对二值信号量的需求很大，单独提供一个mutex更利于代码编写和清晰度。

mutex缺点：

为了实现某些性能上的优化，mutex数据结构比semaphore更大（这已经违背了mutex刚设计时的意愿），这也会消耗更多的CPU cache和memory footprint.

何时使用mutex？

允许睡眠的上下文、临界区操作较长、计数值只为0或1时使用mutex

kernel文档建议，在任何需要加锁且mutex可以满足需求的情况都应该使用mutex而不是其他锁。

## 5 RCU

RCU(Read-Copy Update)即读-拷贝，更新。对于用RCU保护的资源，读者不需要任何等待，而写者访问它时，需要先拷贝一个副本，然后对副本修改，最后在适当的时机把指向原来数据的指针指向新的数据。这个“适当的时机”指的是没有任何读者操作该资源时。

RCU相关API：

rcu\_read\_lock()读者进入临界区

rcu\_read\_unlock()读者退出临界区

synchronize\_rcu()由写者调用，当读者都退出老更新前的临界区后，写者才可以返回该函数。

call\_rcu()由写者调用，但不阻塞。该函数的参数中有一个回调函数，当读者都退出更新前的临界区后，调用该回调函数。

rcu\_assign\_pointer()给临界区资源赋新值

rcu\_dereference()使用临界区资源

RCU 写者的典型流程：

（1）拷贝一份临界区资源，此时有两份临界区资源，这里称为老资源和新资源

（2）用新资源代替老资源，使得之后的读者访问的是新资源

（3）等待读取老临界区的读者全部退出

（4）此时，老资源已没有读者操作，释放该资源

内核提供了对list,hlist等常用数据结构的RCU版本。对于RCU，对共享数据的操作必须保证能够被没有使用同步机制的读者看到，所以内存栅是非常必要的。内存栅只在alpha架构上才使用。

何时使用RCU？

读操作远多于写操作、且写操作不是特别紧急时使用RCU

## 6 顺序锁seqlock

顺序锁为写者赋予更高的优先级，写者永远不会等待读者。缺点是读者有时不得不读多次数据以获取正确的结果。

顺序锁的数据结构中除了有spinlock外，还有一个顺序号。如果成功获得锁，顺序锁的顺序号会加1，以便读者能够检查出是否在读期间有写者访问过。读者在读取数据前后两次读顺序值，如果两次值不相同，则说明读取期间有新的写者操作过数据了，那么本次读取就是无效的。

典型使用：

读端：

do

{

seqnum = read\_seqbegin(&seqlock\_a);

//读操作代码块

...

} while (read\_seqretry(&seqlock\_a, seqnum));

写端：

spin\_lock(&lock);

write\_seqlock(&seqlock\_a)

...

write\_sequnlock(&seqlock\_a)

spin\_unlock(&lock);

写者通过调用write\_seqlock()和write\_sequnlock()获取和释放顺序锁。write\_seqlock()函数获取seqlock\_t数据结构中的自旋锁，然后使顺序计数器sequence加1；write\_sequnlock()函数再次增加顺序计数器sequence，然后释放自旋锁。这样可以保证写者在整个写的过程中，计数器sequence的值是奇数，并且当没有写者在改变数据的时候，计数器的值是偶数。

read\_seqbegin()返回顺序锁的当前顺序号；如果局部变量seq的值是奇数（写者在read\_seqbegin()函数被调用后，正更新数据结构），或seq的值与顺序锁的顺序计数器的当前值不匹配（当读者正执行临界区代码时，写者开始工作），read\_seqretry()就返回1，说明本次读取失败，需要重新读取 。

并不是每一种资源都可以使用顺序锁来保护。一般来说，必须在满足下述条件时才能使用顺序锁：

（1）读者的临界区资源不包括被写者修改和被读者取值的指针，否则，写者有可能使指针失效，读者读取时会产生OPPs。

（2）读者的临界区代码没有副作用。

何时使用顺序锁？

读操作远多于写操作、且写操作很紧急时使用顺序锁。

# LINUX 信号处理

## 1 信号概念

信号是进程在运行过程中，由自身产生或由进程外部发过来的消息（事件）。信号是硬件中断的软件模拟(软中断)。每个信号用一个整型常量宏表示，以 SIG开头，比如SIGCHLD、SIGINT 等，它们在系统头文件<signal.h>中定义,也可以通过在 shell下键入kill–l 查看信号列表，或者键入man 7 signal查看更详细的说明。

信号的生成来自内核，让内核生成信号的请求来自3 个地方： 

①用户：用户能够通过输入 CTRL+c、Ctrl+\，或者是终端驱动程序分配给信号控制字符的其他任何键来请求内核产生信号； 

②内核：当进程执行出错时，内核会给进程发送一个信号，例如非法段存取(内存访问违规)、浮点数溢出等；

③进程：一个进程可以通过系统调用 kill 给另一个进程发送信号，一个进程可以通过信号和另外 一个进程进行通信。

kill原型

#include<sys/types.h>

#include<signal.h>

int kill(pid\_t pid,int sig)

成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看。

由进程的某个操作产生的信号称为同步信号(synchronous signals),例如除 0；由像用户击键这样的进程外部事件产生的信号叫做异步信号(asynchronoussignals)。

进程接收到信号以后，可以有如下 3 种选择进行处理：

①接受默认处理：接收默认处理的进程通常会导致进程本身消亡。例如连接到终端的进程，用户按下 CTRL+c，将导致内核向进程发送一个 SIGINT 的信号，进程如果不对该信号做特殊的处 理，系统将采用默认的方式处理该信号，即终止进程的执行； signal(SIGINT,SIG\_DFL)；通过man 7 signal来查看 .

编号为1 ~ 31的信号为传统UNIX支持的信号，是不可靠信号(非实时的)，编号为32 ~ 63的信号是后来扩充的，称做可靠信号(实时信号)。不可靠信号和可靠信号的区别在于前者不支持排队，可能会造成信号丢失，而后者不会。下面我们对编号小于SIGRTMIN的信号进行讨论。

1) SIGHUP:本信号在用户终端连接(正常或非正常)结束时发出, 通常是在终端的控制进程结束时, 通知同一session内的各个作业, 这时它们与控制终端不再关联。登录Linux时，系统会分配给登录用户一个终端(Session)。在这个终端运行的所有程序，包括前台进程组和后台进程组，一般都属于这个Session。当用户退出Linux登录时，前台进程组和后台有对终端输出的进程将会收到SIGHUP信号。这个信号的默认操作为终止进程，因此前台进程组和后台有终端输出的进程就会中止。不过可以捕获这个信号，比如wget能捕获SIGHUP信号，并忽略它，这样就算退出了Linux登录，wget也能继续下载。此外，对于与终端脱离关系的守护进程，这个信号用于通知它重新读取配置文件。

2) SIGINT:程序终止(interrupt)信号, 在用户键入INTR字符(通常是Ctrl-C)时发出，用于通知前台进程组终止进程。

3) SIGQUIT:和SIGINT类似, 但由QUIT字符(通常是Ctrl-\)来控制. 进程在因收到SIGQUIT退出时会产生core文件, 在这个意义上类似于一个程序错误信号。

4) SIGILL:执行了非法指令. 通常是因为可执行文件本身出现错误, 或者试图执行数据段. 堆栈溢出时也有可能产生这个信号。

5) SIGTRAP:由断点指令或其它trap指令产生. 由debugger使用。

6) SIGABRT:调用abort函数生成的信号。

7) SIGBUS:非法地址, 包括内存地址对齐(alignment)出错。比如访问一个四个字长的整数, 但其地址不是4的倍数。它与SIGSEGV的区别在于后者是由于对合法存储地址的非法访问触发的(如访问不属于自己存储空间或只读存储空间)。

8) SIGFPE：在发生致命的算术运算错误时发出. 不仅包括浮点运算错误, 还包括溢出及除数为0等其它所有的算术的错误。

9) SIGKILL：用来立即结束程序的运行. 本信号不能被阻塞、处理和忽略。如果管理员发现某个进程终止不了，可尝试发送这个信号。

10) SIGUSR1：留给用户使用

11) SIGSEGV：试图访问未分配给自己的内存, 或试图往没有写权限的内存地址写数据.

12) SIGUSR2：留给用户使用

13) SIGPIPE：管道破裂。这个信号通常在进程间通信产生，比如采用FIFO(管道)通信的两个进程，读管道没打开或者意外终止就往管道写，写进程会收到SIGPIPE信号。此外用Socket通信的两个进程，写进程在写Socket的时候，读进程已经终止。

14) SIGALRM：时钟定时信号, 计算的是实际的时间或时钟时间. alarm函数使用该信号.

15) SIGTERM：程序结束(terminate)信号, 与SIGKILL不同的是该信号可以被阻塞和处理。通常用来要求程序自己正常退出，shell命令kill缺省产生这个信号。如果进程终止不了，我们才会尝试SIGKILL。

17) SIGCHLD：子进程结束时, 父进程会收到这个信号。如果父进程没有处理这个信号，也没有等待(wait)子进程，子进程虽然终止，但是还会在内核进程表中占有表项，这时的子进程称为僵尸进程。这种情况我们应该避免(父进程或者忽略SIGCHILD信号，或者捕捉它，或者wait它派生的子进程，或者父进程先终止，这时子进程的终止自动由init进程来接管)。

18) SIGCONT：让一个停止(stopped)的进程继续执行. 本信号不能被阻塞. 可以用一个handler来让程序在由stopped状态变为继续执行时完成特定的工作. 例如, 重新显示提示符

19) SIGSTOP：停止(stopped)进程的执行. 注意它和terminate以及interrupt的区别:该进程还未结束, 只是暂停执行. 本信号不能被阻塞, 处理或忽略.

20) SIGTSTP：停止进程的运行, 但该信号可以被处理和忽略. 用户键入SUSP字符时(通常是Ctrl-Z)发出这个信号

21) SIGTTIN：当后台作业要从用户终端读数据时, 该作业中的所有进程会收到SIGTTIN信号. 缺省时这些进程会停止执行.

22) SIGTTOU：类似于SIGTTIN, 但在写终端(或修改终端模式)时收到.

23) SIGURG：有"紧急"数据或out-of-band数据到达socket时产生.

24) SIGXCPU：超过CPU时间资源限制. 这个限制可以由getrlimit/setrlimit来读取/改变。

25) SIGXFSZ：当进程企图扩大文件以至于超过文件大小资源限制。

26) SIGVTALRM：虚拟时钟信号. 类似于SIGALRM, 但是计算的是该进程占用的CPU时间.

27) SIGPROF：类似于SIGALRM/SIGVTALRM, 但包括该进程用的CPU时间以及系统调用的时间.

28) SIGWINCH：窗口大小改变时发出.

29) SIGIO：文件描述符准备就绪, 可以开始进行输入/输出操作.

30) SIGPWR：Power failure

31) SIGSYS：非法的系统调用。

②忽略信号：进程可以通过代码，显示地忽略某个信号的处理，例如：signal(SIGINT,SIG\_IGN); 但是某些信号是不能被忽略的,例如9 号信号； 

③捕捉信号并处理：进程可以事先注册信号处理函数，当接收到信号时，由信号处理函数自动捕 捉并且处理信号。

有两个信号既不能被忽略也不能被捕捉，它们是SIGKILL 和 SIGSTOP。即进程接收到这两个信号 后，只能接受系统的默认处理，即终止进程。SIGSTOP 是暂停进程。

## 2 signal信号处理机制

可以用函数signal 注册一个信号捕捉函数。原型为：

#include<signal.h>

typedef void (\*sighandler\_t)(int); //函数指针

sighandler\_t signal(int signum, sighandler\_t handler);

signal 的第1个参数signum表示要捕捉的信号，第2个参数是个函数指针，表示要对该信号进行捕捉的函数，该参数也可以是 SIG\_DFL(表示交由系统缺省处理，相当于白注册了)或SIG\_IGN(表示忽略掉 该信号而不做任何处理)。signal如果调用成功，返回以前该信号的处理函数的地址，否则返回SIG\_ERR。设置了标准错误描述，可使用perror函数查看。

sighandler\_t 是信号捕捉函数，由 signal 函数注册，注册以后，在整个进程运行过程中均有效，并且对不同的信号可以注册同一个信号捕捉函数。该函数只有一个整型参数，表示信号值。

示例：1：捕捉终端 CTRL+c 产生的 SIGINT 信号

#include<unistd.h>

#include<stdio.h>

#include<sys/wait.h>

#include<sys/types.h>

#include<signal.h>

void SignHandler**(**int iSignNo**)**

**{**

printf**(**"Capture sign no:%d\n"**,**iSignNo**);**

**}**

int main**()**

**{**

signal**(**SIGINT**,**SignHandler**);**

**while(**1**)**

sleep**(**1**);**

**return** 0**;**

**}**

该程序运行起来以后，通过按CTRL+c 将不再终止程序的运行（或者另开一个终端，然后发送消息：“kill –s 2 进程号”或者“kill -2 进程号”，可以实现Ctrl+c 同样的效果。因为CTRL+c 产生的SIGINT 信号已经由进程中注册的 SignHandler 函数捕捉了。该程序可以通过 Ctrl+\终止，因为组合键 Ctrl+\能够产生 SIGQUIT 信号，而该信号的捕捉函数尚未在程序中注册。

示例2：忽略掉终端CTRL+c 产生的 SIGINT 信号：

#include<unistd.h>

#include<stdio.h>

#include<sys/wait.h>

#include<sys/types.h>

#include<signal.h>

int main**()**

**{**

signal**(**SIGINT**,**SIG\_IGN**);**

**while(**1**)**

sleep**(**1**);**

return0**;**

**}**

该程序运行起来以后，将 CTRL+C 产生的 SIGINT 信号忽略掉了，所以 CTRL＋C 将不再能使 该进程终止，要终止该进程，可以向进程发送 SIGQUIT 信号，即组合键 CTRL+\。或者另外开一个端口，然后执行 ps –aux 查看进程，发现该进程号之后用kill-9 进程号 杀掉该进程。

示例3：接受信号的默认处理,接受默认处理就相当于没有写信号处理程序：

#include<unistd.h>

#include<stdio.h>

#include<sys/wait.h>

#include<sys/types.h>

#include<signal.h>

int main**()**

**{**

signal**(**SIGINT**,**SIG\_DFL**);**

**while(**1**)**

sleep**(**1**);**

return0**;**

**}**

## 3 sigaction 信号处理机制

### 3.1 信号处理情况分析

在signal 处理机制下，还有许多特殊情况需要考虑：

1） 注册一个信号处理函数，并且处理完毕一个信号之后，是否需要重新注册，才能够捕捉下一个 信号；（不需要）

2） 如果信号处理函数正在处理信号，并且还没有处理完毕时，又发生了一个或多个同类型的信号，这时该怎么处理；（继续执行），后续相同信号忽略（但会多执行一次）。

3） 如果信号处理函数正在处理信号，并且还没有处理完毕时，又发生了一个或多个不同类型的信号，这时该怎么处理；（跳转去执行另一个信号，之后再执行剩下的没有处理完的信号）

4） 如果程序阻塞在一个系统调用(如 read(...))时，发生了一个信号，这时是让系统调用返回错误再接着进入信号处理函数，还是先跳转到信号处理函数，等信号处理完毕后，系统调用直接中断返回,不再执行之前的操作。（实际是后者）

相互打断就是一次，相同信号处理函数不重入 ,不存在无限嵌套打断的情况

示例：

#include<stdio.h>

#include<string.h>

#include<unistd.h>

#include<signal.h>

int g\_iSeq **=**0**;**

void SignHandler**(**int iSignNo**)**

**{**

int iSeq **=**g\_iSeq**++;**

printf**(**"%d Enter SignHandler,signo:%d\n"**,**iSeq**,**iSignNo**);**

sleep**(**3**);**

printf**(**"%d Leave SignHandler,signo:%d\n"**,**iSeq**,**iSignNo**);**

**}**

int main**()**

**{**

char szBuf**[**8**];**

int iRet**;**

signal**(**SIGINT**,**SignHandler**);** //不同的信号调用同一个处理函数

signal**(**SIGQUIT**,**SignHandler**);**

**do**

**{**

iRet **=**read**(**STDIN\_FILENO**,**szBuf**,sizeof(**szBuf**)-**1**);**

**if(**iRet**<**0**)**

**{**

perror**(**"readfail."**);**

**break;**

**}**

szBuf**[**iRet**]=**0**;**

printf**(**"Get:%s"**,**szBuf**);**

**}while(**strcmp**(**szBuf**,**"quit\n"**)!=**0**);**

return0**;**

**}**

程序运行时，针对于如下几种输入情况(要输入得快)，看输出结果：

（1）[CTRL+c] [CTRL+c] （一个一个挨着执行）

（2）[CTRL+c][CTRL+\] （先执行 c 的进入，被\打断，转而执行\，最后执行 c 的退出）

（3）hello[CTRL+\][Enter] （先执行中断，没有任何输出）

（4）[CTRL+\]hello [Enter] （先执行中断，输出内容）

（5）hel[CTRL+\]lo[Enter] （先执行中断，只输出lo）

### 3.2 sigaction 信号处理注册

函数原型：

#include<signal.h>

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

sigaction 也用于注册一个信号处理函数 成功返回0，失败返回-1，设置了标准错误描述，可使用perror函数查看。

参数signum 为需要捕捉的信号

参数act 是一个结构体，里面包含信号处理函数地址、处理方式等信息

参数oldact 是一个传出参数，sigaction 函数调用成功后，oldact 里面包含以前对 signum 的处理方式的信息，通常为 NULL

结构体 struct sigaction(注意名称与函数 sigaction 相同)的原型为：

struct sigaction

{

void (\*sa\_handler)(int); //老类型的信号处理函数指针

void (\*sa\_sigaction)(int,siginfo\_t \*, void\*);//新类型的信号处理函数指针

sigset\_t sa\_mask; //将要被阻塞的信号的集合

int sa\_flags; //信号处理方式掩码 (SA\_SIGINFO) ·

void(\*sa\_restorer)(void); //保留，不要使用

};

该结构体的各字段含义及使用方式：

1）字段 sa\_handler 是一个函数指针，用于指向原型为 void handler(int)的信号处理函数地址， 即老类型的信号处理函数（如果用再将 sa\_flags =0,就等同于 signal()函数）

2）字段 sa\_sigaction 也是一个函数指针，用于指向原型为： void handler(int iSignNum, siginfo\_t \*pSignInfo, void\*pReserved)；的信号处理函数，即新类型的信号处理函数 该函数的三个参数含义为：

iSignNum：传入的信号

pSignInfo：与该信号相关的一些信息，它是个结构体

pReserved：保留，现没用，通常为 NULL

3）字段 sa\_handler 和 sa\_sigaction 只应该有一个生效，如果想采用老的信号处理机制，就应该让 sa\_handler 指向正确的信号处理函数，并且让字段 sa\_flags 为 0；否则应该让 sa\_sigaction 指向正确的信号处理函数，并且让字段 sa\_flags 包含SA\_SIGINFO 选项 (或运算)

4）字段 sa\_mask 是一个包含信号集合的结构体，该结构体内的信号表示在进行某个信号处理时，将要被阻塞的信号。针对 sigset\_t 结构体，有一组专门的函数对它进行处理，它们是：

#include<signal.h>

int sigemptyset(sigset\_t \*set); //清空信号集合 set

int sigfillset(sigset\_t \*set); //将所有信号填充进 set 中

int sigaddset(sigset\_t \*set, int signum); //往 set 中添加信号signum

int sigdelset(sigset\_t \*set,int signum); //从 set 中移除信号signum

int sigismember(const sigset\_t\*set, int signum); //判断 signum 是否包含在set中(是:返回 1,否:0）

int sigpending(sigset\_t\*set); //将被阻塞的信号(挂起信号)集合由参数 set 指针返回

其中，对于函数 sigismember 而言，如果 signum 在 set 集中，则返回1；不在，则返回0；出错时返 回-1.其他的函数都是成功返回 0，出错返回-1。设置了标准错误描述，可使用perror函数查看。

对sigset\_t集合中的同一信号多次删除或增加都会成功，原因时因为函数内部只是简单的赋值操作。

信号被阻塞并不是忽略信号，在原信号处理过程结束后，仍然会继续执行当前被阻塞而挂起的信号。

例如，如果打算在处理信号 SIGINT 时，只阻塞对SIGQUIT 信号的处理，可以用如下方法：

struct sigaction act**;**

act**.**sa\_flags**=**SA\_SIGINFO**;**

act**.**sa\_sigaction**=**newHandler**;**

sigemptyset**(&**act**.**sa\_mask**);**

sigaddset**(&**act**.**sa\_mask**,**SIGQUIT**);**

sigaction**(**SIGINT**,&**act**,NULL);**

5）字段sa\_flags 是一组掩码的合成值，指示信号处理时所应该采取的一些行为，各掩码的含义为：

SA\_RESETHAND 处理完毕要捕捉的信号后，将自动撤消信号处理函数的注册， 即必须再重新注册信号处理函数，才能继续处理接下来产生的信号。该选项不符合一般的信号处理流程，现已经被废弃。

SA\_NODEFER 在处理信号时，如果又发生了其它的信号，则立即进入其它信号的处理，等其它信号处理完毕后，再继续处理当前的信号， 即递规地处理。（不常用）不断重入，次数不丢失

SA\_RESTART 如果在发生信号时，程序正阻塞在某个系统调用，例如调用 read()函数，则在处理完毕信号后，接着从阻塞的系统调用继续直到返回。如果不指定该参数，中断处理完毕之后，read函数读取失败。

SA\_SIGINFO 指示结构体的信号处理函数指针是哪个有效，如果 sa\_flags包含该掩码，则 sa\_sigaction 指针(新处理函数)有效，否则是 sa\_handler（旧处理函数）指针有效。（常用）

Example1：用sigaction 实现和 signal（只能传递一个参数）一样的功能。

#include<signal.h>

#include<stdio.h>

void handle**(**int signo**)**

**{**

printf**(**"signo: %d\n"**,**signo**);**

**}**

int main**()**

**{**

struct sigaction st**;**

st**.**sa\_handler**=**handle**;**

st**.**sa\_flags**=**0**;**

sigaction**(**SIGINT**,&**st**,NULL);**

**while(**1**)**

**{**

sleep**(**1**);**

**}**

**}**

Example2：用sigaction 实现调用新的信号处理函数

#include<stdio.h>

#include<string.h>

#include<unistd.h>

#include<signal.h>

int g\_iSeq **=**0**;**

void SignHandlerNew**(**int iSignNo**,**siginfo\_t **\***pInfo**,**void**\***pReserved**)**

**{**

int iSeq **=** g\_iSeq**++;**

printf**(**"%d Enter SignHandlerNew,signo:%d\n"**,**iSeq**,**iSignNo**);**

sleep**(**3**);**

printf**(**"%d Leave SignHandlerNew,signo:%d\n"**,**iSeq**,**iSignNo**);**

**}**

int main**()**

**{**

struct sigaction act**;**

act**.**sa\_sigaction**=**SignHandlerNew**;**

act**.**sa\_flags**=**SA\_SIGINFO**;**

sigaction**(**SIGINT**,&**act**,NULL);**

sigaction**(**SIGQUIT**,&**act**,NULL);**

**while(**1**)**

**{**

sleep**(**1**);**

**}**

**return** 0**;**

**}**

练习与验证： 针对于先前的5 种输入情况，给下面代码再添加一些代码，使之能够进行如下各种形式的响应：

1）[CTRL+c][CTRL+c]时，第 1 个信号处理阻塞第 2个信号处理；

2）[CTRL+c][CTRL+c]时，第 1 个信号处理时，允许递规地第2 个信号处理；

3）[CTRL+c][CTRL+\]时，第 1 个信号阻塞第2 个信号处理；

4）read不要因为信号处理而返回失败结果。

你会发现，当一个信号被阻塞之后，在解除阻塞之前，该信号发生了多次，但是解除阻塞的时候， 内核只会向进程发送一个信号而已，而不管在其阻塞期间有多少个信号产生，因为 linux 并不会对信号 进行排队。另外，这里用到了打断 read 输入的中断处理，必须要加参数 SA\_RESTART，对于signal函数而言，它安装的信号处理函数，系统默认会自动重启被中断的系统调用，而不是让它出错返回。而对于sigaction函数而言，必须要自己指定 SA\_RESTART 实现重启功能，如果不指定则会read 失败，提示 read的时候中断发生。

### 3.3 sigprocmask 信号阻塞

函数sigaction 中设置的被阻塞信号集合只是针对于要处理的信号，例如

struct sigaction act;

sigemptyset(&act.sa\_mask);

sigaddset(&act.sa\_mask,SIGQUIT);

sigaction(SIGINT,&act,NULL);

表示只有在处理信号 SIGINT 时，才阻塞信号 SIGQUIT；(重点区分)

函数sigprocmask 是全程阻塞，在 sigprocmask 中设置了阻塞集合后，被阻塞的信号将不能再被信号处理函数捕捉，直到重新设置阻塞信号的集合。成功返回0出错返回-1，设置了标准错误描述，可使用perror函数查看。 原型为：

#include<signal.h>

int sigprocmask(int how,const sigset\_t \*set, sigset\_t \*oldset);

参数how 的值为如下 3 者之一：

a：SIG\_BLOCK,将参数 2 的信号集合添加到进程原有的阻塞信号集合中

b：SIG\_UNBLOCK,从进程原有的阻塞信号集合移除参数 2 中包含的信号

c：SIG\_SETMASK，重新设置进程的阻塞信号集为参数 2 的信号集

参数set 为阻塞信号集

参数oldset 是传出参数，存放进程原有的信号集，通常为 NULL

perf统计用户程序性能，oprofile统计内核的程序性能。benchark也可测试软硬件性能。



示例：添加全程阻塞

#include<stdio.h>

#include<string.h>

#include<unistd.h>

#include<signal.h>

int g\_iSeq**=**0**;**

void SignHandlerNew**(**int iSignNo**,**siginfo\_t **\***pInfo**,**void**\***pReserved**)**

**{**

int iSeq**=**g\_iSeq**++;**

printf**(**"%d Enter SignHandlerNew,signo:%d\n"**,**iSeq**,**iSignNo**);**

sleep**(**3**);**

printf**(**"%d Leave SignHandlerNew,signo:%d\n"**,**iSeq**,**iSignNo**);**

**}**

int main**()**

**{**

char szBuf**[**8**];**

int iRet**;**

//屏蔽掉 SIGQUIT 信号

sigset\_t sigSet**;**

sigemptyset**(&**sigSet**);**

sigaddset**(&**sigSet**,**SIGQUIT**);**

sigprocmask**(**SIG\_BLOCK**,&**sigSet**,NULL);**

struct sigaction act**;**

act**.**sa\_sigaction**=**SignHandlerNew**;**

act**.**sa\_flags**=**SA\_SIGINFO**;**

sigemptyset**(&**act**.**sa\_mask**);**

sigaction**(**SIGINT**,&**act**,NULL);**

**while(**1**);**

return0**;**

**}**

实例：取消指定信号的全程阻塞。

#include<stdio.h>

#include<string.h>

#include<unistd.h>

#include<signal.h>

int g\_iSeq**=**0**;**

void SignHandlerNew**(**int iSignNo**,**siginfo\_t **\***pInfo**,**void**\***pReserved**)**

**{**

int iSeq**=**g\_iSeq**++;**

printf**(**"%d Enter SignHandlerNew,signo:%d\n"**,**iSeq**,**iSignNo**);**

sleep**(**3**);**

printf**(**"%d Leave SignHandlerNew,signo:%d\n"**,**iSeq**,**iSignNo**);**

**}**

int main**()**

**{**

//屏蔽掉 SIGINT 和 SIGQUIT 信号，SigHandlerNew 将不能再捕捉 SIGINT 和 SIGQUIT

sigset\_t sigSet**;**

sigemptyset**(&**sigSet**);**

sigaddset**(&**sigSet**,**SIGINT**);**

sigaddset**(&**sigSet**,**SIGQUIT**);**

sigprocmask**(**SIG\_BLOCK**,&**sigSet**,NULL);**//将SIGINT、SIGQUIT 屏蔽

struct sigaction act**;**

act**.**sa\_sigaction**=**SignHandlerNew**;**

act**.**sa\_flags**=**SA\_SIGINFO**;**

sigemptyset**(&**act**.**sa\_mask**);**

sigaction**(**SIGINT**,&**act**,NULL);**

sigaction**(**SIGQUIT**,&**act**,NULL);**

int iCount **=**0**;**

**while(**1**)**

**{**

**if(**iCount **>**3**)**

**{**

sigset\_t sigSet2**;**

sigemptyset**(&**sigSet2**);**

sigaddset**(&**sigSet2**,**SIGINT**);**

sigprocmask**(**SIG\_UNBLOCK**,&**sigSet2**,** **NULL);**

**}**

iCount **++;**

sleep**(**2**);**

**}**

return0**;**

**}**

Example3：利用 sigpending 测试信号，并采用上图的模型

#include<signal.h>

#include<unistd.h>

void handler**(**int signum**,** siginfo\_t**\***pInfo**,** void**\***pReversed**)**

**{**

printf**(**"receivesignal %d\n"**,**signum**);**

**}**

int main**()**

**{**

sigset\_t new\_mask**,**old\_mask**,**pending\_mask**;**

sigemptyset**(&**new\_mask**);**

sigaddset**(&**new\_mask**,**SIGINT**);**

**if(**sigprocmask**(**SIG\_BLOCK**,&**new\_mask**,&**old\_mask**))**

printf**(**"block signal SIGINT error\n"**);**

struct sigaction act**;**

sigemptyset**(&**act**.**sa\_mask**);**

act**.**sa\_flags**=**SA\_SIGINFO**;**

act**.**sa\_sigaction**=**handler**;**

**if(**sigaction**(**SIGINT**,** **&**act**,NULL))**

printf**(**"install signal SIGINT error\n"**);**

sleep**(**10**);**

printf**(**"now begin to get pending mask and unblock SIGINT\n"**);**

**if(**sigpending**(&**pending\_mask**)<**0**)**//将阻塞信号全部添加到pending\_mask 信号集中并返回

printf**(**"get pending mask error\n"**);**

**if(**sigismember**(&**pending\_mask**,**SIGINT**))**

printf**(**"signal SIGINT is pending\n"**);**

**if(**sigprocmask**(**SIG\_SETMASK**,&**old\_mask**,NULL)<**0**)**

printf**(**"unblock signalerror\n"**);**

printf**(**"signal unblocked\n"**);**

sleep**(**10**);**

return0**;**

**}**

注意：信号注册应该在子进程创建后，否则所有的进程都会继承信号处理属性

## 4 用程序发送信号

### 4.1 kill 信号发送函数

原型为：

#include<sys/types.h>

#include<signal.h>

int kill(pid\_t pid, int signum);

函数kill将信号发送给参数pid代表的进程，如果成功，返回 0，否则为-1,设置了标准错误描述，可使用perror函数查看。

参数 pid 为将要接受信号的进程的 pid，可以通过 getpid()函数获得来给自身发送信号，还可以发送 信号给指定的进程，此时 pid 有如下描述：

pid>0 将信号发给 ID 为 pid 的进程

pid==0 将信号发送给与发送进程属于同一个进程组的所有进程

pid<0 将信号发送给进程组 ID 等于pid 绝对值的所有进程

pid==-1 将信号发送给该进程有权限发送的系统里的所有进程

参数sig 为要发送的信号

示例，输入结束后，将通过发送信号 SIGQUIT 把自己杀掉：

#include<stdio.h>

#include<signal.h>

#include<unistd.h>

#include<sys/types.h>

int main**()**

**{**

**while(**1**)**

**{**

**if(**getchar**()==**EOF**)** //运行之后输入没有反应，当按下Ctrl+d（EOF），进程关闭

kill**(**getpid**(),**SIGQUIT**);**

**}**

**return** 0**;**

**}**

除此之外，还可以向指定的进程发送信号：

程序1：

#include<stdio.h>

intmain**()**

**{**

**while(**1**);**

**return** 0**;**

**}**

程序2：

#include <stdio.h>

#include<signal.h>

#include<unistd.h>

#include<sys/types.h>

int main**(**int argc**,** char**\***argv**[])**

**{**

int pid **=**atoi**(**argv**[**1**]);** //字符串数字转数值

kill**(**pid**,**SIGQUIT**);**

return0**;**

**}**

首先运行程序1，然后用 ps –aux 查看其进程号，假设位11002。再运行程序2 ./2 11002 即可

## 5 计时器与信号

### 5.1 睡眠函数

Linux 下有两个睡眠函数，原型为：

#include<unistd.h>

unsigned int sleep(unsigned int seconds);

void usleep(unsigned long usec);

函数sleep 让进程睡眠seconds 秒，函数 usleep 让进程睡眠usec 微秒。成功返回0，失败返回-1，设置了标准错误描述，可使用perror函数查看。

sleep 睡眠函数内部是用信号机制进行处理的，用到的函数有：

#include<unistd.h>

unsigned int alarm(unsigned int seconds);

功能：告知自身进程，要进程在 seconds 秒后自动产生一个 SIGALRM 的信号 , 如果先前有一个alarm（）请求剩余时间，alarm（）将返回一个非零值，该值是前一个请求产生SIGALRM信号的秒数。 否则，alarm（）将返回0。

int pause(void);

功能：将自身进程挂起，直到有信号发生时才从 pause 返回. 成功无返回值，失败返回-1，设置了标准错误描述，可使用perror函数查看。

示例：模拟睡眠 3 秒：

#include<signal.h>

#include<stdio.h>

#include<unistd.h>

void SignHandler**(**int iSignNo**)**

**{**

printf**(**"signal:%d\n"**,**iSignNo**);**

**}**

int main**()**

**{**

signal**(**SIGALRM**,**SignHandler**);**

alarm**(**3**);** //等待 3 秒之后自动产生 SIGALRM 信号

printf**(**"Before pause().\n"**);**

pause**();** //将进程挂起，直到有信号发生才退出挂起状态

printf**(**"Afterpause().\n"**);**

**return** 0**;**

**}**

注意：因为 sleep 在内部是用 alarm 实现的，所以在程序中最好不要 sleep 与 alarm 混用，以免造成 混乱。

### 5.2 时钟处理

Linux 为每个进程维护 3 个计时器，分别是真实计时器、虚拟计时器和实用计时器。 

真实计时器计算的是程序运行的实际时间；---直接 

虚拟计时器计算的是程序运行在用户态时所消耗的时间(可认为是实际时间减掉(系统调用和程序睡眠所消耗)的时间)；---需要了解内核 

实用计时器计算的是程序处于用户态和处于内核态所消耗的时间之和。---常用 (不计睡眠时间)

例如：有一程序运行，在用户态运行了5 秒，在内核态运行了6 秒，还睡眠了 7秒，则真实计算器 计算的结果是 18秒，虚拟计时器计算的是 5 秒，实用计时器计算的是 11 秒。

用指定的初始间隔和重复间隔时间为进程设定好一个计时器后，该计时器就会定时地向进程发送时 钟信号。3 个计时器发送的时钟信号分别为：SIGALRM,SIGVTALRM 和 SIGPROF。

用到的函数与数据结构：

#include<sys/time.h>

int getitimer(int which, struct itimerval\*value); //获取计时器的设置

参数 which 指定哪个计时器，可选项为 ITIMER\_REAL(真实计时器)、ITIMER\_VIRTUAL(虚拟 计时器、ITIMER\_PROF(实用计时器))

参数value 为一结构体的传出参数，用于传出该计时器的初始间隔时间和重复间隔时间

返回值：成功返回0，失败返回-1，设置了标准错误描述，可使用perror函数查看。

int setitimer(int which, const struct itimerval\*value, struct itimerval\*ovalue);//设置计时器

参数 which 指定哪个计时器，可选项为 ITIMER\_REAL(真实计时器)、ITIMER\_VIRTUAL(虚拟 计时器、ITIMER\_PROF(实用计时器))

参数value 为一结构体的传入参数，指定该计时器的初始间隔时间和重复间隔时间

参数ovalue 为一结构体传出参数，用于传出以前的计时器时间设置。

返回值：成功返回0，失败返回-1，设置了标准错误描述，可使用perror函数查看。

struct itimerval

{

struct timeval it\_interval; /\*next value\*/ //重复间隔

struct timeval it\_value; /\*current value\*/ //初始间隔

};

struct timeval

{

long tv\_sec; /\*seconds \*/ //时间的秒数部分

long tv\_usec; /\*microseconds\*/ //时间的微秒部分

};

示例：启用真实计时器的进行时钟处理（获得当前系统时间，并一秒更新一次）

#include <signal.h>

#include <time.h>

#include <sys/time.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

void sigHandler**(**int iSigNum**)**

**{**

time\_t tt**;**

time**(&**tt**);**

struct tm**\***pTm **=**gmtime**(&**tt**);**

printf**(**"%04d-%02d-%02d %02d:%02d:%02d\n"**,** **(**1900**+**pTm**->**tm\_year**),** **(**1**+**pTm**->**tm\_mon**),** pTm**->**tm\_mday**,(**8**+**pTm**->**tm\_hour**),**pTm**->**tm\_min**,** pTm**->**tm\_sec**);**

**}**

void InitTime**(**int tv\_sec**,** int tv\_usec**)**

**{**

signal**(**SIGALRM**,**sigHandler**);**

alarm**(**0**);**

struct itimerval tm**;**

tm**.**it\_value**.**tv\_sec **=**tv\_sec**;**

tm**.**it\_value**.**tv\_usec **=**tv\_usec**;**

tm**.**it\_interval**.**tv\_sec**=**tv\_sec**;**

tm**.**it\_interval**.**tv\_usec**=**tv\_usec**;**

**if(**setitimer**(**ITIMER\_REAL**,** **&**tm**,** **NULL)==-**1**)**

**{**

perror**(**"setitimer error"**);**

exit**(-**1**);**

**}**

**}**

int main**()**

**{**

InitTime**(**2**,**0**);**

**while(**1**)** **;**

return0**;**

**}**

# LINUX 多线程

## 1 Linux 多线程概述

### 1.1 概述

进程是系统中程序执行和资源分配的基本单位。每个进程有自己的数据段、代码段和堆栈段。这就 造成进程在进行切换等操作时都需要有比较负责的上下文切换等动作。为了进一步减少处理器的空转时间支持多处理器和减少上下文切换开销，也就出现了线程。

线程通常叫做轻量级进程。线程是在共享内存空间中并发执行的多道执行路径，是一个更加接近于执行体的概念，拥有独立的执行序列，是进程的基本调度单元，每个进程至少都有一个 main 线程。它与同进程中的其他线程共享进程空间｛堆 代码 数据 文件描述符 信号等｝，只拥有自己的栈空间，大大减少了上下文切换的开销。

线程和进程在使用上各有优缺点：线程执行开销小，占用的CPU少，线程之间的切换快，但不利于资源的管理和保护；而进程正相反。从可移植性来讲，多进程的可移植性要好些。

同进程一样，线程也将相关的变量值放在线程控制表内。一个进程可以有多个线程，也就是有多个线程控制表及堆栈寄存器，但却共享一个用户地址空间。要注意的是，由于线程共享了进程的资源和地址空间，因此，任何线程对系统资源的操作都会给其他线程带来影响。



[www.jb51.net/article/102004.htm](http://www.jb51.net/article/102004.htm)

### 1.2 线程分类

按调度者分为用户级线程和核心级线程

用户级线程：主要解决上下文切换问题，调度算法和调度过程全部由用户决定，在运行时不需要特定 的内核支持。缺点是无法发挥多处理器的优势 ·

核心级线程：允许不同进程中的线程按照同一相对优先调度方法调度，发挥多处理器的并发优势

现在大多数系统都采用用户级线程和核心级线程并存的方法。一个用户级线程可以对应一个或多个核心级线程，也就是“一对一”或“一对多”模型。

### 1.3 线程创建的 Linux 实现

Linux 的线程是通过用户级的函数库实现的，一般采用 pthread 线程库实现线程的访问和控制。它用 第 3 方posix 标准的 pthread，具有良好的可移植性。 编译的时候要在后面加上 –lpthread

创建 退出 等待

多进程 fork、system、popen、exel函数簇 exit() wait()

多线程 pthread\_create pthread\_exit() pthread\_join()

## 2 线程的创建和退出

创建线程实际上就是确定调用该线程函数的入口点，线程的创建采用函数 pthread\_create。在线程创建以后，就开始运行相关的线程函数，在该函数运行完之后，线程就退出，这也是线程退出的一种方式，叫做自然退出。 另一种线程退出的方式是使用函数 pthread\_exit()函数，这是线程主动退出行为。这里要注意的是，在使用线程函数时，不能随意使用 exit 退出函数进行出错处理，由于exit 的作用是使调用进程终止，往往一个进程包括了多个线程，所以在线程中通常使用 pthread\_exit函数来代替进程中的退出函数 exit。

由于一个进程中的多个线程是共享数据段的，因此通常在线程退出之后，退出线程所占用的资源并不会随着线程的终止而得到释放。正如进程之间可以通过wait()函数系统调用来同步终止并释放资源一 样，线程之间也有类似的机制，那就是 pthread\_join 函数。pthread\_join 函数可以用于将当前线程挂起， 等待线程的结束然后回收资源。这个函数是一个线程阻塞函数，调用它的函数将一直等待直到被等待的线程结束为止，当函数返回时，被等待线程的资源被回收。

函数原型：

#include<pthread.h>

int pthread\_create (pthread\_t \*threadid,pthread\_attr\_t \*attr,void \*(\*start\_routine) (void\*), void \*arg);

void pthread\_exit(void\*retval);

pthread\_t pthread\_self(void);

通常的形式为：

pthread\_t pthid**;**

pthread\_create**(&**pthid**,NULL,**pthfunc**,NULL);**

pthread\_create**(&**pthid**,NULL,**pthfunc**,(**void**\*)**3**);**

pthread\_exit**(NULL);**或

pthread\_exit**((**void**\*)**3**);**//3 作为返回值被pthread\_join 函数捕获。

函数 pthread\_create 用来创建线程。返回值：成功，则返回0；失败，则返回对应错误码。各参数描述如下：

·参数thread 是传出参数，保存新线程的标识；

·参数attr是一个结构体指针，结构中的元素分别指定新线程的运行属性,attr 可以用 pthread\_attr\_init 等函数设置各成员的值，但通常传入为 NULL 即可；

·参数start\_routine 是一个函数指针，指向新线程的入口点函数，线程入口点函数带有一个void \*的参数由 pthread\_create 的第 4个参数传入；

·参数arg 用于传递给第 3个参数指向的入口点函数的参数，可以为 NULL，表示不传递。

函数pthread\_exit 表示线程的退出。其参数可以被其它线程用 pthread\_join 函数捕获。=

函数pthread\_self获取线程自身ID。这个函数始终成功。

示例：

#include <stdio.h>

#include <pthread.h>

void **\***ThreadFunc**(**void **\***pArg**)** //参数的值为123

**{**

int i **=**0**;**

**for(;**i**<**10**;**i**++)**

**{**

printf**(**"Hi,I'm child thread,arg is:%d\n"**,(**int**)**pArg**);**

sleep**(**1**);**

**}**

pthread\_exit**(NULL);**

**}**

int main**()**

**{**

pthread\_t thdId**;**

pthread\_create**(&**thdId**,** **NULL,**ThreadFunc**,(**void **\*)**123**);**

int i **=**0**;** **for(;**i**<**10**;**i**++)**

**{**

printf**(**"Hi,I'm main thread,child threadid is:%x\n"**,**thdId**);**

sleep**(**1**);**

**}**

**return** 0**;**

**}**

编译时需要带上线程库选项： gcc -o a a.c –l pthread

## 3 等待线程退出

### 3.1 等待线程退出

线程从入口点函数自然返回，或者主动调用 pthread\_exit()函数，都可以让线程正常终止

线程从入口点函数自然返回时，函数返回值可以被其它线程用 pthread\_join 函数获取.

pthread\_join 原型为：

#include<pthread.h>

int pthread\_join(pthread\_t thid, void\*\*thread\_return);

pthread\_join函数是一个阻塞函数，会让当前线程挂起，一直等到参数 thid指定的线程返回；与多进程中的 wait 或 waitpid 类似。 返回值：成功，则返回0；失败，则返回对应错误码

参数thread\_return 是一个传出参数，接收线程函数的返回值。如果线程通过调用 pthread\_exit()终止，则 pthread\_exit()中的参数相当于自然返回值，照样可以被其它线程用pthread\_join获取到。

这里使用二级指针是由于线程函数自然返回或者thread\_exit返回的是一级指针，那么赋值的左边也因该是一个地址，在pthread\_join中要对这个指针进行改变时，相当于在函数中对指针做修改，根据修改变量的值用一级指针，修改指针的值用二级指针。因此这里是二级指针。操作为\*thread\_return=returnval 。

参数thid 传递0 值时，join 返回 ESRCH 错误。

子线程可以回收子线程，但不能回收主线程，主线程可以回收子线程。

该函数还有一个非常重要的作用，由于一个进程中的多个线程共享数据段，因此通常在一个线程退 出后，退出线程所占用的资源并不会随线程结束而释放。如果 th 线程类型并不是自动清理资源类型的， 则 th 线程退出后，线程本身的资源必须通过其它线程调用 pthread\_join 来清除,这相当于多进程程序中 的 waitpid。

Example：返回值的例子

#include <stdio.h>

#include <unistd.h>

#include <pthread.h>

void**\***ThreadFunc**(**void **\***pArg**)**

**{**

int iArg**=(**int**)**pArg**;** //将 void\*转换为

int sleep**(**iArg**);**

**if(**iArg**<**3**)**

**return(**void**\*)(**iArg**\***2**);**

**else**

pthread\_exit**((**void**\*)(**iArg**\***2**));**//和reaturn 达到的效果一样，都可以用于正常返回

**}**

int main**()**

**{**

pthread\_t thdId**;**

int iRet **=**0**;**

pthread\_create**(&**thdId**,** **NULL,**ThreadFunc**,(**void **\*)**2**);**//传递参数值为2

pthread\_join**(**thdId**,(**void**\*\*)&**iRet**);** //接收子线程的返回值

printf**(**"The firstchild thread ret is:%d\n"**,**iRet**);**

pthread\_create**(&**thdId**,** **NULL,**ThreadFunc**,(**void **\*)**4**);**

pthread\_join**(**thdId**,(**void**\*\*)&**iRet**);**

printf**(**"The second child thread ret is:%d\n"**,**iRet**);**

return0**;**

**}**

Example：子线程释放空间

#include <stdio.h>

#include <pthread.h>

#include <malloc.h>

void **\***threadfunc**(**void **\***args**)**

**{**

char**\***p **=(**char**\*)**malloc**(**10**);** //自己分配了内存

int i**=**0**;**

**for(;**i**<**10**;**i**++)**

**{**

printf**(**"hello,my name is wangxiao!\n"**);**

sleep**(**1**);**

**}**

free**(**p**);** //如果父线程中没有调用 pthread\_cancel，此处可以执行

printf**(**"p is freed\n"**);**

pthread\_exit**((**void**\*)**3**);**

**}**

int main**()**

**{**

pthread\_t pthid**;**

pthread\_create**(&**pthid**,** **NULL,**threadfunc**,NULL);**

inti**=**1**;**

**for(;**i**<**5**;**i**++)** //父线程的运行次数比子线程的要少，当父线程结束的时候，

//如果没有 pthread\_join 函数等待子线程执行的话，子线程也会退出。

**{**

printf**(**"hello,nice to meet you!\n"**);**

sleep**(**1**);**

// if(i%3==0)

// pthread\_cancel(pthid);

//表示当 i%3==0 的时候就取消子线程，该函数将导致子线 程直接退出，

//不会执行上面紫色的 free 部分的代码，即释放空间失败。要想释放指针类型的变量 p，

//此时必须要用 pthread\_cleanup\_push 和 pthread\_cleanup\_pop 函数释放间，

**}**

int retvalue**=**0**;**

pthread\_join**(**pthid**,(**void**\*\*)&**retvalue**);**//等待子线程释放空间，并获取子线程的返回值

printf**(**"returnvalueis:%d\n"**,**retvalue**);**

**return** 0**;**

**}**

pthread\_join 不能回收堆内存的，只回收线程的栈内存和内核中的 struct task\_struct 结构占用的内存。堆内存必须在线程函数中单独回收。

### 3.2 线程的取消

线程也可以被其它线程杀掉，在Linux 中的说法是一个线程被另一个线程取消(cancel)。

线程取消的方法是一个线程向目标线程发cancel 信号，但是如何处理cancel 信号则由目标线程自 己决定，目标线程或者忽略、或者立即终止、或者继续运行至 cancelation-point(取消点)后终止。

**取消点：**

根据POSIX 标准，pthread\_join()、pthread\_testcancel()、pthread\_cond\_wait()、 pthread\_cond\_timedwait()、sem\_wait()、sigwait()等函数以及 read()、write()等会引起阻塞的系统调用都是 Cancelation-point，而其他 pthread 函数都不会引起 Cancelation 动作。但是 pthread\_cancel 的手册页声称，由于 Linux 线程库与 C 库结合得不好，因而目前 C 库函数都不是 Cancelation-point；但 CANCEL信号会使线程从阻塞的系统调用中退出，(即终止线程)并置 EINTR 错误码，因此可以在需要作为Cancelation-point 的系统调用前后调用 pthread\_testcancel()，从而达到 POSIX 标准所要求的目标，即如下代码段：

pthread\_testcancel(); //创建一个取消点

retcode=read(fd,buffer,length); //如果是取消点，线程终止，如果不是，执行下面一句

pthread\_testcancel();//如果前面函数不是有取消点，就结束。

但是从 RedHat9.0的实际测试来看，至少有些 C 库函数的阻塞函数是取消点，如 read(),getchar() 等，而 sleep()函数不管线程是否设置了 pthread\_setcancelstate(PTHREAD\_CANCEL\_DISABLE,NULL)， 都起到取消点作用。总之，线程的取消一方面是一个线程强行杀另外一个线程，从程序设计角度看并 不是一种好的风格，另一方面目前 Linux 本身对这方面的支持并不完善，所以在实际应用中应该谨慎使用！！

函数原型：

#include<pthread.h>

int pthread\_cancel(pthread\_t thid);

void pthread\_testcancel(void);

函数pthread\_cancel取消某个线程，必须当目标线程中有取消点时才能取消，否则不可以取消成功返回0，出错返回错误码。

参数thid是要取消线程的id。

增加 man 信息 ，apt-get install manpages-posix-dev

### 3.3 线程终止清理函数

不论是可预见的线程终止还是异常终止，都会存在资源释放的问题，在不考虑因运行出错而退出 的前提下，如何保证线程终止时能顺利的释放掉自己所占用的资源，特别是锁资源和堆空间，就是一个必须考 虑解决的问题。

最经常出现的情形是资源独占锁的使用：线程为了访问临界资源而为其加上锁，但在访问过程中该线程被外界取消，或者发生了中断，则该临界资源将永远处于锁定状态得不到释放。外界取消操作是不可预见的，因此的确需要一个机制来简化用于资源释放的编程。

在POSIX 线程API 中提供了一个pthread\_cleanup\_push()/pthread\_cleanup\_pop()函数对，用于自动释放资源--从pthread\_cleanup\_push()的调用点开始到pthread\_cleanup\_pop()之间的程序段中的终止动作(即pthread\_exit ,pthread\_cancel）都将执行pthread\_cleanup\_push()所指定的清理函数。API 定义如下：

void pthread\_cleanup\_push(void(\*routine)(void\*), void\*arg)

void pthread\_cleanup\_pop(int execute)

pthread\_cleanup\_push()/pthread\_cleanup\_pop()采用先入后出的栈结构管理

参数void routine(void \*arg)函数在pthread\_cleanup\_push()被调用时被压入清理函数栈，多次对 pthread\_cleanup\_push()的调用将在清理函数栈中形成一个函数链，在执行该函数链时按照压栈的相反 顺序弹出。

参数arg为清理函数的传入参数。

参数execute 参数表示执行到 pthread\_cleanup\_pop()时是否在弹出清理函数的同时执行该函数， 为 0 表示不执行，非 0 为执行；这个参数并不影响异常终止时清理函数的执行。

pthread\_cleanup\_pop 的参数 execute 如果为非 0 值,则按栈的顺序注销掉一个原来注册的清理函数， 并执行该函数；当 pthread\_cleanup\_pop()函数的参数为 0 时，仅仅在线程调用 pthread\_exit函数或者其它线程对本线程调用 pthread\_cancel 函数时，才在弹出“清理函数”的同时执行该“清理函数”。

pthread\_cleanup\_push在对资源回收的同时，要注意不要对统一堆内存释放两次

pthread\_cleanup\_push()/pthread\_cleanup\_pop()是以宏方式实现的，这是 pthread.h 中的宏定义：

#define pthread\_cleanup\_push(routine,arg)\

{

struct \_pthread\_cleanup\_buffer \_buffer;\ \_pthread\_cleanup\_push(&\_buffer,(routine),(arg));

#define pthread\_cleanup\_pop(execute) \

\_pthread\_cleanup\_pop(&\_buffer,(execute));

}

可见，pthread\_cleanup\_push()带有一个"{"，而 pthread\_cleanup\_pop()带有一个"}"，因此这两个函数必须成对出现，且必须位于程序的同一级别的代码段中才能通过编译。

示例：

#include <stdio.h>

#include <pthread.h>

void CleanFunc**(**void**\***pArg**)**

**{**

printf**(**"CleanFunc(%d)\n"**,(**int**)**pArg**);**

**}**

void **\***ThreadFunc**(**void **\***pArg**)**

**{**

pthread\_cleanup\_push**(**CleanFunc**,(**void**\*)**1**);**

pthread\_cleanup\_push**(**CleanFunc**,(**void**\*)**2**);**

sleep**(**2**);**

pthread\_cleanup\_pop**(**1**);**

pthread\_cleanup\_pop**(**1**);**

**}**

int main**()**

**{**

pthread\_t thdId**;**

pthread\_create**(&**thdId**,** **NULL,**ThreadFunc**,(**void **\*)**2**);**

pthread\_join**(**thdId**,NULL);**

return0**;**

**}**

运行结果为：

CleanFunc(2)

CleanFunc(1)

如果将里面的两次 pthread\_cleanup\_pop(1);改为pthread\_cleanup\_pop(0);推测一下结果是怎样？ 

没有任何输出（此时 CleanFunc 函数得不到执行）

如果修改为 0 之后，再在sleep(2)之后添加 pthread\_exit(NULL);则此时的结果又是如何： 

跟 pthread\_cleanup\_pop(1);实现的结果一样了。

Example：用pthread\_cleanup\_push 和pthread\_cleanup\_pop来释放子线程分配的内存空间

#include <stdio.h>

#include <pthread.h>

#include <malloc.h>

void freemem**(**void **\***args**)**

**{**

free**(**args**);**

printf**(**"cleanup thememory!\n"**);**

**}**

void **\***threadfunc**(**void **\***args**)**

**{**

char**\***p **=(**char**\*)**malloc**(**10**);** //自己分配了内存

pthread\_cleanup\_push**(**freemem**,**p**);**

int i**=**0**;**

**for(;**i**<**10**;**i**++)**

**{**

printf**(**"hello,my name is wangxiao!\n"**);**

sleep**(**1**);**

**}**

pthread\_exit**((**void**\*)**3**);**

pthread\_cleanup\_pop**(**0**);**

**}**

int main**()**

**{**

pthread\_t pthid**;**

pthread\_create**(&**pthid**,** **NULL,**threadfunc**,NULL);**

inti**=**1**;**

**for(;**i**<**5**;**i**++)** //父线程的运行次数比子线程的要少，当父线程结束的时候，

//如果没有 pthread\_join函数等待子线程执行，子线程也会退出，即子线程只执行了4 次。

**{**

printf**(**"hello,nicetomeet you!\n"**);**

sleep**(**1**);**

**if(**i**%**3**==**0**)**

pthread\_cancel**(**pthid**);** //表示当i%3==0的时候就取消子线程，该函数将导致直接退出

//不会执行上面紫色的 free 部分的代码，即释放空间失败。要想释放指针类型的变量p，

//必须要用 pthread\_cleanup\_push 和 pthread\_cleanup\_pop 函数释放空间

**}**

int retvalue**=**0**;**

pthread\_join**(**pthid**,(**void**\*\*)&**retvalue**);**//等待子线程释放空间，并获取子线程的返回值

printf**(**"returnvalue is:%d\n"**,**retvalue**);**

return0**;**

**}**

## 4 线程的同步与互斥

### 4.1 线程的互斥-mutex

在 Posix Thread 中定义了一套专门用于线程互斥的 mutex 函数。mutex 是一种简单的加锁的方法来控制对共享资源的存取，这个互斥锁只有两种状态（上锁和解锁），可以把互斥锁看作某种意义上的全 局变量。为什么需要加锁，就是因为多个线程除了栈空间外共用进程的资源，要访问的是公共区间时（全局变量），当一个线程访问的时候，需要加上锁以防止另外的线程对它进行访问，实现资源的独占。在一个时刻只能有一个线程掌握某个互斥锁，拥有上锁状态的线程才能够对共享资源进行操作。若其他线程希望上锁一个已经上锁了的互斥锁，则该线程就会挂起，直到上锁的线程释放掉互斥锁为止。

#### 4.1.1 创建和销毁锁

有两种方法创建互斥锁，静态方式和动态方式。

·静态方式：POSIX定义了一个宏PTHREAD\_MUTEX\_INITIALIZER来静态初始化互斥锁，方法如下：

pthread\_mutex\_t mutex=PTHREAD\_MUTEX\_INITIALIZER;

在 LinuxThreads 实现中，pthread\_mutex\_t 是一个结构，而 PTHREAD\_MUTEX\_INITIALIZER 则是一个宏常量。

·动态方式：动态方式是采用 pthread\_mutex\_init()函数来初始化互斥锁，API 定义如下：

#include<pthread.h>

int pthread\_mutex\_init(pthread\_mutex\_t \*mutex,const pthread\_mutexattr\_t\*mutexattr)

这个函数始终返回0.

其中mutexattr 用于指定互斥锁属性（见下），如果为 NULL 则使用缺省属性。通常为 NULL

pthread\_mutex\_destroy()用于注销一个互斥锁，API 定义如下：

int pthread\_mutex\_destroy(pthread\_mutex\_t\*mutex);

返回值：成功，则返回0；失败，则返回对应错误码

销毁一个互斥锁即意味着释放它所占用的资源，且要求锁当前处于开放状态。由于在 Linux 中，互斥锁并不占用任何资源，因此 Linux Threads 中的 pthread\_mutex\_destroy()除了检查锁状态以外（锁定状态则返回EBUSY）没有其他动作。

#### 4.1.2 互斥锁属性

互斥锁属性结构体的定义为：

typedef struct

{

int \_\_mutexkind; //注意这里是两个下划线

}pthread\_mutexattr\_t;

互斥锁的属性在创建锁的时候指定，在 Linux Threads 实现中仅有一个锁类型属性\_\_mutexkind，不同的锁类型在试图对一个已经被锁定的互斥锁加锁时表现不同也就是是否阻塞等待。有三个值可供选择：

·PTHREAD\_MUTEX\_TIMED\_NP，普通锁(或快速锁)，这是缺省值（直接写NULL 就是表示这个缺省值）。当一个线程加锁以后，其余请求锁的线程(包括上锁线程自身)将形成一个阻塞等待队列，并在解锁后按优先级获得锁。这种锁策略保证了资源分配的公平性

示例：初始化一个快速锁。

pthread\_mutex\_t lock;

pthread\_mutex\_init(&lock,NULL);

·PTHREAD\_MUTEX\_RECURSIVE\_NP，嵌套锁，允许同一个线程对同一个锁成功获得多次，并通过多次 unlock 解锁。如果是不同线程请求，则在加锁线程解锁时重新竞争。也就是不会影响本线程对资源的使用，但如果解锁次数小于加锁次数，则其他线程全部挂起。。

示例：初始化一个嵌套锁。

pthread\_mutex\_t lock;

pthread\_mutexattr\_t mutexattr;

pthread\_mutexattr\_settype(&mutexattr, PTHREAD\_MUTEX\_RECURSIVE \_NP); pthread\_mutex\_init(&lock,&mutexattr);

·PTHREAD\_MUTEX\_ERRORCHECK\_NP，检错锁，如果同一个线程请求同一个锁，则返回 EDEADLK，否则与 PTHREAD\_MUTEX\_TIMED\_NP 类型动作相同。这样就保证当不允许多次加锁时不会出现最简单情况下的死锁。如果锁的类型是快速锁，一个线程加锁之后，又加锁，则此时就是死锁。

示例：初始化一个检错锁。

pthread\_mutex\_t lock;

pthread\_mutexattr\_t mutexattr;

pthread\_mutexattr\_settype(&mutexattr, PTHREAD\_MUTEX\_ERRORCHECK\_NP);

pthread\_mutex\_init(&lock,&mutexattr);

补充：linux的锁有四种：互斥锁（mutex），自旋锁（spin lock），条件锁（condition variable）读写锁（read/write lock）

当然锁属性也可以通过函数来操作。

#include <pthread.h>

int pthread\_mutexattr\_init(pthread\_mutexattr\_t \*attr);

int pthread\_mutexattr\_destroy(pthread\_mutexattr\_t \*attr);

int pthread\_mutexattr\_settype(pthread\_mutexattr\_t \*attr, int kind);

int pthread\_mutexattr\_gettype(const pthread\_mutexattr\_t \*attr, int \*kind);

除了pthread\_mutexatr\_settype成功返回0.出错返回错误码。其余始终返回0。

还分为睡眠锁（sleep-waiting）和忙碌锁（busy-waiting）

#### 4.1.3 锁操作

锁操作主要包括

加锁 int pthread\_mutex\_lock(pthread\_mutex\_t\*mutex)

解锁 int pthread\_mutex\_unlock(pthread\_mutex\_t\*mutex)

测试加锁 int pthread\_mutex\_trylock(pthread\_mutex\_t\*mutex)

·pthread\_mutex\_lock：加锁，不论哪种类型的锁，都不可能被两个不同的线程同时得到，而必须等待解锁。成功返回0.出错返回错误码。

对于普通锁类型，解锁者可以是同进程内任何线程；同一线程二次加锁则死锁

对于检错锁则必须由加锁者解锁才有效，否 则返回 EPERM；同一线程二次加锁，锁引用计数加1

对于嵌套锁，文档和实现要求必须由加锁者解锁，但实验结果表明并没有这种限制， 这个不同目前还没有得到解释。同一线程二次加锁返回EDEADLK

在同一进程中的线程，如果加锁后没有解锁，则任何其他线程都无法再获得锁。

·pthread\_mutex\_unlock：根据不同的锁类型，实现不同的行为, 成功返回0.出错返回错误码。

对于快速锁，pthread\_mutex\_unlock 解除锁定；

对于嵌套锁，pthread\_mutex\_unlock 使锁上的引用计数减 1；

对于检错锁，如果锁是当前线程锁定的，则解除锁定，否则什么也不做。

· pthread\_mutex\_trylock：语义与pthread\_mutex\_lock()类似，不同的是在锁已经被占据时返回 EBUSY 而不是挂起等待, 成功返回0.出错返回错误码。

Example：比较 pthread\_mutex\_trylock()与pthread\_mutex\_lock()

#include<stdio.h>

#include<pthread.h>

pthread\_mutex\_t lock**;**

void**\*** pthfunc**(**void **\***args**)**

**{**

pthread\_mutex\_lock**(&**lock**);** //先加一次锁

pthread\_mutex\_lock**(&**lock**);** //再用lock 加锁，会挂起阻塞

//pthread\_mutex\_trylock(&lock); //用 trylock 加锁，则不会挂起阻塞

printf**(**"hello\n"**);**

sleep**(**1**);**

pthread\_exit**(NULL);**

**}**

int main**()**

**{**

pthread\_t pthid **=**0**;**

pthread\_mutex\_init**(&**lock**,NULL);**

pthread\_create**(&**pthid**,NULL,**pthfunc**,NULL);**

pthread\_join**(**pthid**,NULL);**

pthread\_mutex\_destroy**(&**lock**);**

**}**

#### 4.1.4 加锁注意事项

如果线程在加锁后解锁前被取消，锁将永远保持锁定状态，因此如果在关键区段内有取消点存在， 则必须在退出回调函数 pthread\_cleanup\_push/pthread\_cleanup\_pop 中解锁。同时不应该在信号处理函数中使用互斥锁，否则容易造成死锁。

总结：线程互斥 mutex：加锁步骤如下：

（1）定义一个全局的 pthread\_mutex\_t lock;

或者用 pthread\_mutex\_t lock=PTHREAD\_MUTEX\_INITIALIZER; //则 main 函数中不用 init

（2）在 main 中调用 pthread\_mutex\_init 函数进行初始化

（3）在子线程函数中调用 pthread\_mutex\_lock 加锁

（4）在子线程函数中调用 pthread\_mutex\_unlock 解锁

（5）最后在 main 中调用 pthread\_mutex\_destroy 函数进行销毁

#### 4.1.5 互斥锁实例

Example：火车站售票（此处不加锁，则会出现卖出负数票的情况）

#include <stdio.h>

#include <pthread.h>

int ticketcount**=**20**;** //火车票，公共资源（全局）

void**\*** salewinds1**(**void**\***args**)** //售票口 1

**{**

**while(**ticketcount **>**0**)** //如果有票,则卖票

**{**

printf**(**"windows1 start sale ticket!the ticket is:%d\n"**,**ticketcount**);**

sleep**(**3**);** //卖一张票需要 3 秒的操作时间

ticketcount**--;** //出票

printf**(**"sale ticket finish!,the last ticket is:%d\n"**,**ticketcount**);**

**}**

**}**

void**\*** salewinds2**(**void**\***args**)** //售票口 2

**{**

**while(**ticketcount **>**0**)** //如果有票,则卖票

**{**

printf**(**"windows2 start saleticket!the ticket is:%d\n"**,**ticketcount**);**

sleep**(**3**);** //卖一张票需要 3 秒的操作时间

ticketcount**--;** //出票

printf**(**"sale ticket finish!,the last ticket is:%d\n"**,**ticketcount**);**

**}**

**}**

int main**()**

**{**

pthread\_t pthid1**=**0**;**

pthread\_t pthid2**=**0**;**

pthread\_create**(&**pthid1**,NULL,**salewinds1**,NULL);** //线程 1

pthread\_create**(&**pthid2**,NULL,**salewinds2**,NULL);** //线程 2

pthread\_join**(**pthid1**,NULL);**

pthread\_join**(**pthid2**,NULL);**

return0**;**

**}**

Example：加锁之后的火车售票

#include<stdio.h>

#include<pthread.h>

int ticketcount **=**20**;**

pthread\_mutex\_t lock**;**

void**\*** salewinds1**(**void**\***args**)**

**{**

**while(**1**)**

**{**

pthread\_mutex\_lock**(&**lock**);** //因为要访问全局的共享变量，所以就要加锁

**if(**ticketcount **>**0**)** //如果有票

**{**

printf**(**"windows1 start sale ticket!the ticket is:%d\n"**,**ticketcount**);**

sleep**(**3**);** //卖一张票需要3 秒的操作时间

ticketcount**--;** //出票

printf**(**"saleticket finish!,the lastticket is:%d\n"**,**ticketcount**);**

**}**

**else** //如果没有票

**{**

pthread\_mutex\_unlock**(&**lock**);** //解锁

pthread\_exit**(NULL);** //退出线程

**}**

pthread\_mutex\_unlock**(&**lock**);** //解锁

sleep**(**1**);** //要放到锁的外面，让另一个有时间锁

**}**

**}**

void**\*** salewinds2**(**void**\***args**)**

**{**

**while(**1**)**

**{**

pthread\_mutex\_lock**(&**lock**);** //因为要访问全局的共享变量，所以就要加锁

**if(**ticketcount**>**0**)** //如果有票

**{**

printf**(**"windows2 start sale ticket!the ticket is:%d\n"**,**ticketcount**);**

sleep**(**3**);** //卖一张票需要3 秒的操作时间

ticketcount**--;** //出票

printf**(**"sale ticket finish!,the last ticket is:%d\n"**,**ticketcount**);**

**}**

**else** //如果没有票

**{**

pthread\_mutex\_unlock**(&**lock**);** //解锁

pthread\_exit**(NULL);** //退出线程

**}**

pthread\_mutex\_unlock**(&**lock**);** //解锁

sleep**(**1**);** //要放到锁的外面，让另一个有时间锁

**}**

**}**

int main**()**

**{**

pthread\_t pthid1**=**0**;**

pthread\_t pthid2**=**0**;**

pthread\_mutex\_init**(&**lock**,NULL);** //初始化锁

pthread\_create**(&**pthid1**,NULL,**salewinds1**,NULL);** //线程 1

pthread\_create**(&**pthid2**,NULL,**salewinds2**,NULL);** //线程 2

pthread\_join**(**pthid1**,NULL);**

pthread\_join**(**pthid2**,NULL);**

pthread\_mutex\_destroy**(&**lock**);** //销毁锁

**return** 0**;**

**}**

### 4.2 线程的同步-条件变量

条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程因等待条件变量的条件成立而挂起；另一个线程使条件成立（给出条件成立信号）。为了防止竞争，条件变量的使用总是和一个互斥锁结合在一起,以防止多个进程同时激活时发生资源竞争

#### 4.2.1 创建和注销

条件变量和互斥锁一样，都有静态、动态两种创建方式：

静态方式使用PTHREAD\_COND\_INITIALIZER 常量，如下：

pthread\_cond\_t cond=PTHREAD\_COND\_INITIALIZER;

动态方式调用pthread\_cond\_init()函数，API 定义如下：

int pthread\_cond\_init(pthread\_cond\_t \*cond, pthread\_condattr\_t\*cond\_attr);

成功返回0，出错返回错误码

尽管POSIX 标准中为条件变量定义了属性，但在 Linux Threads 中没有实现，因此 cond\_attr 值通 常为 NULL，且被忽略。

注销一个条件变量需要调用 pthread\_cond\_destroy()，同mutex一样只有在没有线程在该条件变量上等待的时候 能注销这个条件变量，否则返回 EBUSY。因为 Linux 实现的条件变量没有分配什么资源，所以注销动作只包括检查是否有等待线程。API 定义如下：

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

成功返回0，出错返回错误码

#### 4.2.2 等待和激发

等待条件有两种方式：

无条件等待 pthread\_cond\_wait()和计时等待 pthread\_cond\_timedwait(): 原型如下：

int pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t\*mutex);

int pthread\_cond\_timedwait(pthread\_cond\_t \*cond, pthread\_mutex\_t\*mutex, const struct timespec\*abstime);

成功返回0，出错返回错误码

函数解开 mutex 指向的锁并被条件变量 cond 阻塞，直到条件成立，转为激发态，再将mutex锁上。其中计时等待方式表示经历 abstime 段时间后，即使条件变量不满足，阻塞也被解除。无论哪种等待方式，都必须和一个互斥锁配合，以防止多个线程同时请求 pthread\_cond\_wait()（或 pthread\_cond\_timedwait()，下同）的竞争条件（RaceCondition）。mutex 互斥锁必须是普通锁（PTHREAD\_MUTEX\_TIMED\_NP），且在调用pthread\_cond\_wait()前必须由本线程加锁（pthread\_mutex\_lock()），而在更新条件等待队列以前，mutex保持锁定状态，并在线程挂起进入等待前解锁。在条件满足从而离开 pthread\_cond\_wait()之前，mutex将被重新加锁，以与进入 pthread\_cond\_wait()前的加锁动作对应。（也就是说在做pthread\_cond\_wait之前，往往要用 pthread\_mutex\_lock 进行加锁，而调用 pthread\_cond\_wait 函数会将锁解开，然后将线程挂起阻塞。直到条件被 pthread\_cond\_signal激发，再将锁状态恢复为锁定状态，最后再用 pthread\_mutex\_unlock 进行解锁）。

激发条件有两种形式，pthread\_cond\_signal()激活一个等待该条件的线程，存在多个等待线程时按入队顺序激活其中一个；而pthread\_cond\_broadcast()则激活所有等待线程

#include <pthread.h>

int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);

int pthread\_cond\_signal(pthread\_cond\_t \*cond);

成功返回0，出错返回错误码

timespec结构体定义如下：

struct timespec

{

\_\_time\_t tv\_sec; /\*seconds 秒\*/

long int tv\_nsec; /\*nanoseconds 纳秒\*/

}

超时是指 当前时间 + 多长时间超时

#### 4.2.3 其他

pthread\_cond\_wait()和 pthread\_cond\_timedwait()都被实现为取消点，也就是说如果 pthread\_cond\_wait()被取消，则退出阻塞，然后将锁状态恢复，则此时 mutex 是保持锁定状态的，而当 前线程已经被取消掉，那么解锁的操作就会得不到执行，此时锁得不到释放，就会造成死锁，因而需要定义退出回调函数来为其解锁。

以下示例集中演示了互斥锁和条件变量的结合使用，以及取消对于条件等待动作的影响。在例子 中，有两个线程被启动，并等待同一个条件变量，如果不使用退出回调函数（见范例中的注释部分）， 则 tid2 将在pthread\_mutex\_lock()处永久等待。如果使用回调函数，则tid2 的条件等待及主线程的条件 激发都能正常工作。

实例：

#include<stdio.h>

#include<pthread.h>

#include<unistd.h>

pthread\_mutex\_t mutex**;**

pthread\_cond\_t cond**;**

void ThreadClean**(**void**\***arg**)**

**{**

pthread\_mutex\_unlock**(&**mutex**);**

**}**

void **\***child1**(**void**\***arg**)**

**{**

//pthread\_cleanup\_push(ThreadClean,NULL); //1

**while(**1**)**

**{**

printf**(**"thread1 get running\n"**);**

printf**(**"thread1 pthread\_mutex\_lock returns%d\n"**,**pthread\_mutex\_lock**(&**mutex**));**

pthread\_cond\_wait**(&**cond**,&**mutex**);**//等待父进程发送信号

printf**(**"thread1 condition applied\n"**);**

pthread\_mutex\_unlock**(&**mutex**);**

sleep**(**5**);**

**}**

//pthread\_cleanup\_pop(0); //2

**return** 0**;**

**}**

void **\***child2**(**void**\***arg**)**

**{**

**while(**1**)**

**{**

sleep**(**3**);** //3

printf**(**"thread2 get running.\n"**);**

printf**(**"thread2 pthread\_mutex\_lock returns%d\n"**,**pthread\_mutex\_lock**(&**mutex**));**

pthread\_cond\_wait**(&**cond**,&**mutex**);**

printf**(**"thread2 condition applied\n"**);**

pthread\_mutex\_unlock**(&**mutex**);**

sleep**(**1**);**

**}**

**}**

int main**(**void**)**

**{**

pthread\_t tid1**,**tid2**;**

printf**(**"hello, condition variable test\n"**);**

pthread\_mutex\_init**(&**mutex**,NULL);**

pthread\_cond\_init**(&**cond**,NULL);**

pthread\_create**(&**tid1**,NULL,**child1**,NULL);**

pthread\_create**(&**tid2**,NULL,**child2**,NULL);**

**while(**1**)**

**{**

//父线程

sleep**(**2**);** //4

pthread\_cancel**(**tid1**);** //5

sleep**(**2**);** //6

pthread\_cond\_signal**(&**cond**);**

**}**

sleep**(**10**);**

**return** 0**;**

**}**

不做注释 1，2 则导致 child1 中的 unlock 得不到执行，锁一直没有关闭，而 child2 中的锁不能执 行 lock，则会一直在pthread\_mutex\_lock()处永久等待。如果不做注释 5 的pthread\_cancel()动作，即使 没有那些 sleep()延时操作，child1 和 child2 都能正常工作。注释 3 和注释 4 的延迟使得child1 有时间 完成取消动作，从而使 child2 能在child1 退出之后进入请求锁操作。如果没有注释1 和注释 2 的回调 函数定义，系统将挂起在child2 请求锁的地方，因为child1 没有释放锁；而如果同时也不做注释3 和 注释 4的延时，child2 能在child1 完成取消动作以前得到控制，从而顺利执行申请锁的操作，但却可 能挂起在 pthread\_cond\_wait()中，因为其中也有申请 mutex 的操作。child1 函数给出的是标准的条件变量的使用方式：回调函数保护，等待条件前锁定，pthread\_cond\_wait()返回后解锁。

条件变量机制和互斥锁一样，不能用于信号处理中，在信号处理函数中调用pthread\_cond\_signal() 或者 pthread\_cond\_broadcast()很可能引起死锁。

Example：火车售票，利用条件变量，当火车票卖完的时候，再重新设置票数为 10；

#include<pthread.h>

#include<stdio.h>

int ticketcount**=**10**;**

pthread\_mutex\_tlock**;** //互斥锁

pthread\_cond\_t cond**;** //条件变量

void**\***salewinds1**(**void**\*** args**)**

**{**

**while(**1**)**

**{**

pthread\_mutex\_lock**(&**lock**);**//因为要访问全局共享变量ticketcount，所以就要加锁

**if(**ticketcount**>**0**)** //如果有票

**{**

printf**(**"windows1 start sale ticket!the ticket is:%d\n"**,**ticketcount**);**

ticketcount**--;**//则卖出一张票

**if(**ticketcount **==**0**)**

pthread\_cond\_signal**(&**cond**);**//通知没有票了

printf**(**"saleticket finish!,the last ticket is:%d\n"**,**ticketcount**);**

**}**

**else** //如果没有票了，就解锁退出

**{**

pthread\_mutex\_unlock**(&**lock**);**

**break;**

**}**

pthread\_mutex\_unlock**(&**lock**);**

sleep**(**1**);** //要放到锁的外面

**}**

**}**

void**\***salewinds2**(**void**\*** args**)**

**{**

**while(**1**)**

**{**

pthread\_mutex\_lock**(&**lock**);**

**if(**ticketcount**>**0**)**

**{**

printf**(**"windows2 start sale ticket!the ticket is:%d\n"**,**ticketcount**);**

ticketcount**--;**

**if(**ticketcount **==**0**)**

pthread\_cond\_signal**(&**cond**);** //发送信号

printf**(**"sale ticket finish!,the last ticket is:%d\n"**,**ticketcount**);**

**}**

**else**

**{**

pthread\_mutex\_unlock**(&**lock**);**

**break;**

**}**

pthread\_mutex\_unlock**(&**lock**);**

sleep**(**1**);**

**}**

**}**

void**\***setticket**(**void**\***args**)** //重新设置票数

**{**

pthread\_mutex\_lock**(&**lock**);** //因为要访问全局变量 ticketcount，所以要加锁

**if(**ticketcount **>**0**)**

pthread\_cond\_wait**(&**cond**,&**lock**);**//如果有票就解锁并阻塞，直到没有票就执行下面的

ticketcount **=**10**;** //重新设置票数为 10

pthread\_mutex\_unlock**(&**lock**);** //解锁

sleep**(**1**);**

pthread\_exit**(NULL);**

**}**

void main**()**

**{**

pthread\_tpthid1**,**pthid2**,**pthid3**;**

pthread\_mutex\_init**(&**lock**,NULL);** //初始化锁

pthread\_cond\_init**(&**cond**,NULL);** //初始化条件变量

pthread\_create**(&**pthid1**,NULL,** salewinds1**,NULL);** //创建线程

pthread\_create**(&**pthid2**,NULL,** salewinds2**,NULL);**

pthread\_create**(&**pthid3**,NULL,** setticket**,NULL);**

pthread\_join**(**pthid1**,NULL);** //等待子线程执行完毕

pthread\_join**(**pthid2**,NULL);**

pthread\_join**(**pthid3**,NULL);**

pthread\_mutex\_destroy**(&**lock**);** //销毁锁

pthread\_cond\_destroy**(&**cond**);** //销毁条件变量

**}**

## 5 信号灯

信号灯与互斥锁和条件变量的主要不同在于"灯"的概念，灯亮则意味着资源可用（即加锁），灯灭则意味着不可用（即解锁）。如果说后两种同步方式侧重于"等待"操作，即资源不可用的话，信号灯机制则侧重于点灯，即告知资源可用；没有等待线程的解锁或激发条件都是没有意义的，而没有等待灯亮的线程的点灯操作则有效，且能保持灯亮状态。当然，这样的操作原语也意味着更多的开销。

信号灯的应用除了灯亮/灯灭这种二元灯以外，也可以采用大于1的灯数，以表示资源数大于 1， 这时可以称之为多元灯。

### 5.1 创建和注销

POSIX 信号灯标准定义了有名信号灯和无名信号灯两种，但Linux Threads的实现仅有无名灯，有名灯除了总是可用于多进程之间以外，在使用上与无名灯并没有很大的区别，因此下面仅就无名灯进行讨论。

#include <semaphore.h>

int sem\_init(sem\_t\*sem, int pshared,unsigned int value);//通常pshared为0.表示线程间

这是创建信号灯的 API.成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看。

参数 value 为信号灯的初值。

参数pshared 表示是否为多进程共享而不仅仅是 用于一个进程之间的多线程共享。 Linux Threads没有实现多进程共享信号灯，因此所有非0值的pshared 输入都将使 sem\_init()返回-1，且置 errno 为 ENOSYS。初始化好的信号灯由sem 变量表征，用于以下 点灯、灭灯操作。

int sem\_destroy(sem\_t \*sem);

成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看。

被注销的信号灯 sem 要求已没有线程在等待该信号灯，否则返回-1，且置errno 为 EBUSY。除此之外，Linux Threads 的信号灯注销函数不做其他动作。

### 5.2 点灯和灭灯

int sem\_post(sem\_t\*sem);

int sem\_wait(sem\_t \*sem);

int sem\_trywait(sem\_t\*sem);

成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看。

点灯操作将信号灯值原子地加1，表示增加一个可访问的资源。只有信号灯值大于 0，才能访问公共资源。主要用来增加信号量的值。当有线程阻塞在这个信号量上时，调用这个函数会使其中的一个线程不在阻塞。

sem\_wait()为灭灯操作（等待灯亮操作），主要被用来阻塞当前线程直到信号量 sem 的值大于 0，解除阻塞后将 sem 的值减一，表明公共资源经使用后减少。等待灯亮（信号灯值大于0），然后将信号灯原子地减1，并返回。

sem\_trywait()为sem\_wait()的非阻塞版，如果信号灯计数大于 0，则原子地减 1 并返回 0，否则立即返回-1，errno 置为EAGAIN。

使用时一般先灭灯，表示占用资源，操作完成后，再点灯，表示归还一个资源。

### 5.3 获取灯值

int sem\_getvalue(sem\_t\*sem, int \*sval);

读取sem 中的灯计数，存于\*sval 中。成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看。

### 5.4 其他

sem\_wait()被实现为取消点，而且在支持原子"比较且交换"指令的体系结构上，sem\_post()是唯一能用于异步信号处理函数的 POSIX 异步信号安全的API。

Example：sem\_post表示点灯（资源可用，V 操作）， sem\_wait 表示灭灯（资源不可用，P 操作）

#include<pthread.h>

#include<stdio.h>

#include <semaphore.h> //头文件包含

int ticketcount **=**10**;**

sem\_t lock**;**

void**\***chk1**(**void**\***args**)**

**{**

**while(**1**)**

**{**

sem\_wait**(&**lock**);** //因为要访问全局的共享变量 ticketcount，所以就要加锁

**if(**ticketcount**>**0**)** //如果有票

**{**

printf**(**"windows1 start sale ticket!the ticket is:%d\n"**,**ticketcount**);**

ticketcount**--;**//则卖出一张票

sleep**(**3**);**

printf**(**"sale ticket finish!,the last ticket is:%d\n"**,**ticketcount**);**

**}**

**else** //如果没有票了，就解锁退出

**{**

sem\_post**(&**lock**);**

**break;**

**}**

sem\_post**(&**lock**);**

sleep**(**1**);** //要放到锁的外面

**}**

pthread\_exit**(NULL);**

**}**

void**\***chk2**(**void**\***args**)**

**{**

**while(**1**)**

**{**

sem\_wait**(&**lock**);** //因为要访问全局的共享变量 ticketcount，所以就要加锁

**if(**ticketcount**>**0**)** //如果有票

**{**

printf**(**"windows2 start sale ticket!the ticket is:%d\n"**,**ticketcount**);**

ticketcount**--;**//则卖出一张票

sleep**(**3**);**

printf**(**"sale ticket finish!,the last ticketis:%d\n"**,**ticketcount**);**

**}**

**else** //如果没有票了，就解锁退出

**{**

sem\_post**(&**lock**);**

**break;**

**}**

sem\_post**(&**lock**);**

sleep**(**1**);** //要放到锁的外面

**}**

pthread\_exit**(NULL);**

**}**

void main**()**

**{**

pthread\_tpthid1**,**pthid2**;**

sem\_init**(&**lock**,**0**,**1**);** //信号灯值初始为 1，表示资源可用

pthread\_create**(&**pthid1**,NULL,**chk1**,NULL);**

pthread\_create**(&**pthid2**,NULL,**chk2**,NULL);**

pthread\_join**(**pthid1**,NULL);**

pthread\_join**(**pthid2**,NULL);**

sem\_destroy**(&**lock**);**

**}**

### 5.5 生产者消费者问题

Example1链表实现如下：

#include <pthread.h>

#include<unistd.h>

#include<stdio.h>

#include<malloc.h>

static pthread\_mutex\_tmtx**=**PTHREAD\_MUTEX\_INITIALIZER**;**

static pthread\_cond\_t cond **=**PTHREAD\_COND\_INITIALIZER**;**

struct node

**{**

int n\_number**;**

struct node**\***n\_next**;**

**}\***head**=NULL;**

static void cleanup\_handler**(**void **\***arg**)**

**{**

printf**(**"Cleanup handler of second thread\n"**);**

free**(**arg**);**

pthread\_mutex\_unlock**(&**mtx**);**

**}**

static void**\***thread\_func**(**void **\***arg**)**//消费者

**{**

struct node**\***p **=NULL;**

pthread\_cleanup\_push**(**cleanup\_handler**,**p**);**

**while(**1**)**

**{**

If**(** pthread\_mutex\_lock**(&**mtx**);**

**while** **(**head**==NULL||** **(**flag**=**0**))**

**{**

pthread\_cond\_wait**(&**cond**,&**mtx**);**

**}**

p**=**head**;**

head**=**head**->**n\_next**;**

printf**(**"Got%dfrom front ofqueue\n"**,**p**->**n\_number**);**

free**(**p**);**

pthread\_mutex\_unlock**(&**mtx**);**

**else**

**}**

pthread\_exit**(NULL);**

pthread\_cleanup\_pop**(**0**);** //必须放在最后一行

**}**

int main**(**void**)**

**{**

pthread\_ttid**;**

int i**;**

struct node**\***p**;**

pthread\_create**(&**tid**,NULL,**thread\_func**,** **NULL);**

**for(**i**=**0**;**i**<**10**;**i**++)**//生产者

**{**

p**=(**structnode**\*)**malloc**(sizeof(**struct node**));**

p**->**n\_number **=**i**;**

pthread\_mutex\_lock**(&**mtx**);**//因为head 是共享的，访问共享数据必须要加锁

p**->**n\_next**=**head**;**

head**=**p**;**

pthread\_cond\_signal**(&**cond**);**

pthread\_mutex\_unlock**(&**mtx**);**

sleep**(**1**);**

**}**

printf**(**"thread1wanna end theline.So cancelthread2.\n"**);**

pthread\_cancel**(**tid**);**

pthread\_join**(**tid**,** **NULL);**

printf**(**"Alldone------exiting\n"**);**

**return** 0**;**

**}**

Example2队列实现如下：

#include<stdio.h>

#include <stdlib.h>

#include <time.h>

#include <pthread.h>

#define BUFFER\_SIZE16 //表示一次最多可以不间断的生产16 个产品

struct prodcons

**{**

int buffer**[**BUFFER\_SIZE**];** /\* 数据 \*/

pthread\_mutex\_t lock**;** /\* 加锁 \*/

int readpos**,**writepos**;** /\* 读 pos 写位置 \*/

pthread\_cond\_t notempty**;** /\* 不空，可以读 \*/

pthread\_cond\_t notfull**;** /\* 不满，可以写 \*/

**};**

/\* 初始化\*/

void init**(**struct prodcons **\***b**)**

**{**

pthread\_mutex\_init**(&**b**->**lock**,** **NULL);** //初始化锁

pthread\_cond\_init**(&**b**->**notempty**,NULL);** //初始化条件变量

pthread\_cond\_init**(&**b**->**notfull**,** **NULL);** //初始化条件变量

b**->**readpos**=**0**;** //初始化读取位置从 0 开始

b**->**writepos **=**0**;** //初始化写入位置从 0 开始

**}**

/\* 销毁操作 \*/

void destroy**(**struct prodcons**\***b**)**

**{**

pthread\_mutex\_destroy**(&**b**->**lock**);**

pthread\_cond\_destroy**(&**b**->**notempty**);**

pthread\_cond\_destroy**(&**b**->**notfull**);**

**}**

void put**(**struct prodcons**\***b**,**int data**)**//生产者

**{**

pthread\_mutex\_lock**(&**b**->**lock**);**

**while((**b**->**writepos **+**1**)%**BUFFER\_SIZE **==**b**->**readpos**)**

**{**

//判断是不是满了

printf**(**"wait fornot full\n"**);**

pthread\_cond\_wait**(&**b**->**notfull**,&**b**->**lock**);** //此时为满，不能生产，等待不满的信号

**}**

//下面表示还没有满，可以进行生产

b**->**buffer**[**b**->**writepos**]=**data**;**

b**->**writepos**++;** //写入点向后移一位

**if(**b**->**writepos**>=**BUFFER\_SIZE**)**

b**->**writepos **=**0**;**//如果到达最后，就再转到开头

pthread\_cond\_signal**(&**b**->**notempty**);** //此时有东西可以消费，发送非空的信号

pthread\_mutex\_unlock**(&**b**->**lock**);**

**}**

int get**(**struct prodcons**\***b**)**//消费者

**{**

pthread\_mutex\_lock**(&**b**->**lock**);**

**while(**b**->**writepos **==**b**->**readpos**)**

**{**

//判断是不是空

printf**(**"wait fornot empty\n"**);**

pthread\_cond\_wait**(&**b**->**notempty**,&**b**->**lock**);**//此时为空，不能消费，等待非空信号

**}**

//下面表示还不为空，可以进行消费

int data**=**b**->**buffer**[**b**->**readpos**];**

b**->**readpos**++;** //读取点向后移一位

**if(**b**->**readpos**>=**BUFFER\_SIZE**)**

b**->**readpos **=**0**;**//如果到达最后，就再转到开头

pthread\_cond\_signal**(&**b**->**notfull**);** //此时可以进行生产，发送不满的信号

pthread\_mutex\_unlock**(&**b**->**lock**);**

**return** data**;**

**}**

/\*--------------------------------------------------------\*/

#define OVER(-1) //定义结束标志

struct prodconsbuffer**;** //定义全局变量

/\*--------------------------------------------------------\*/

void**\***producer**(**void **\***data**)**

**{**

intn**=**0**;**

**for(;**n**<**50**;**n**++)**

**{**

printf**(**"put-->%d\n"**,**n**);**

put**(&**buffer**,**n**);**

**}**

put**(&**buffer**,**OVER**);**

printf**(**"producerstopped!\n"**);**

pthread\_exit**(NULL);**

**}**

/\*--------------------------------------------------------\*/

void**\***consumer**(**void **\***data**)**

**{**

**while(**1**)**

**{**

int d**=**get**(&**buffer**);**

**if(**d**==**OVER**)break;**

printf**(**"%d-->get\n"**,** d**);**

**}**

printf**(**"consumer stopped!\n"**);**

pthread\_exit**(NULL);**

**}**

/\*--------------------------------------------------------\*/

int main**(**void**)**

**{**

pthread\_tth\_a**,**th\_b**;**

init**(&**buffer**);**

pthread\_create**(&**th\_a**,NULL,** producer**,**0**);**

pthread\_create**(&**th\_b**,NULL,**consumer**,**0**);**

pthread\_join**(**th\_a**,NULL);**

pthread\_join**(**th\_b**,** **NULL);**

destroy**(&**buffer**);**

**return** 0**;**

**}**

## **6 线程的属性（不是完全版本，选修**）

pthread\_attr\_t定义如下：

typedef struct

{

int detachstate; 线程的分离状态

int schedpolicy; 线程调度策略

struct sched\_param schedparam; 线程的调度参数

int inheritsched; 线程的继承性

int scope; 线程的作用域

size\_t guardsize; 线程栈末尾的警戒缓冲区大小

int stackaddr\_set;

void \* stackaddr; 线程栈的位置

size\_t stacksize; 线程栈的大小

}pthread\_attr\_t;

pthread\_creat第二个参数attr 是一个pthread\_attr\_t的结构体指针，结构中的元素分别指定新线程的运行属性,各成员属性为：

\_\_detachstate 表示新线程是否与进程中其他线程脱离同步，如果置位则新线程不能用 pthread\_join() 来同步，且在退出时自行释放所占用的资源。缺省为 PTHREAD\_CREATE\_JOINABLE 状态。这个属性 也可以在线程创建并运行以后用 pthread\_detach()来设置，而一旦设置为PTHREAD\_CREATE\_DETACH 状态（不论是创建时设置还是运行时设置）则不能再恢复到 PTHREAD\_CREATE\_JOINABLE 状态。

\_\_schedpolicy，表示新线程的调度策略，主要包括 SCHED\_OTHER（正常、非实时）、SCHED\_RR （实时、轮转法）和 SCHED\_FIFO（实时、先入先出）三种，缺省为 SCHED\_OTHER，后两种调度策 略仅对超级用户有效。运行时可以用过 pthread\_setschedparam()来改变。

\_\_schedparam，调度参数，一个 sched\_param结构，目前仅有一个 sched\_priority 整型变量表示线程的运行优先 级。这个参数仅当调度策略为实时（即 SCHED\_RR或 SCHED\_FIFO）时才有效，并可以在运行时通过 pthread\_setschedparam()函数来改变，缺省为 0。

struct sched\_param

{

int sched\_priority;

};

\_\_inheritsched，有两种值可供选择： PTHREAD\_EXPLICIT\_SCHED和PTHREAD\_INHERIT\_SCHED， 前者表示新线程使用显式指定调度策略和调度参数（即 attr 中的值），而后者表示继承调用者线程的值。 缺省为 PTHREAD\_EXPLICIT\_SCHED。

\_\_scope，表示线程间竞争 CPU 的范围，也就是说线程优先级的有效范围。POSIX 的标准中定义了 两个值：PTHREAD\_SCOPE\_SYSTEM 和 PTHREAD\_SCOPE\_PROCESS，前者表示与系统中所有线程一起竞争CPU 时间，后者表示仅与同进程中的线程竞争 CPU。目前 Linux 仅实现了 PTHREAD\_SCOPE\_SYSTEM 一值。

属性设置是由一些函数来完成的，通常调用 pthread\_attr\_init 函数进行初始化。再设置完成属性后，调用 pthread\_creat 函数创建线程。

线程创建时可以设置线程属性：

**·线程属性初始化/销毁：**

int pthread\_attr\_init (pthread\_attr\_t\*attr);

int pthread\_attr\_destroy(pthread\_attr\_t \*attr)

attr：传参数，表示线程属性，后面的线程属性设置函数都会用到。

返回值：成功0，错误非0错误码

**·设置/获取作用域属性：**

int pthread\_attr\_setscope(pthread\_attr\_t\*attr,init scope);

int pthread\_attr\_getscope(pthread\_attr\_t \*attr, int \*scope);

attr：线程属性

scope：PTHREAD\_SCOPE\_SYSTEM(绑定) PTHREAD\_SCOPE\_PRCESS(非绑定)

返回值：成功0，错误非0错误码。

**·设置/获取分离属性:**

int pthread\_attr\_setdetachstate(pthread\_attr\_t \*attr,int detachstate);

int pthread\_attr\_getdetachstate(pthread\_attr\_t \*attr, int \*detachstate);

detachstate ：PTHREAD\_CREAT\_DETACHED(分离) PTHREAD\_CREAT\_JOINABLE(非分离)

返回值：成功0，错误非0错误码。

**·设置/获取线程调度参数：**

int pthread\_attr\_setschedparam(pthread\_attr\_t\*attr,struct sched\_param\*param);

int pthread\_attr\_getschedparam(pthread\_attr\_t\*attr,struct sched\_param\*param);

attr：线程属性

param：线程优先级

返回值：成功0，错误非0错误码。

**·设置/获取线程调度策略**

int pthread\_attr\_setschedpolicy(pthread\_attr\_t \*attr, int policy);

int pthread\_attr\_getschedpolicy(pthread\_attr\_t \*attr, int \*policy);

attr：线程属性

policy :SCHED\_OTHER（正常、非实时）、SCHED\_RR（实时、轮转法）和SCHED\_FIFO（实时、先入先出）

返回值：成功0，错误非0错误码。

**·设置/获取线程继承属性：**

int pthread\_attr\_setinheritsched(pthread\_attr\_t\*attr, int inheritsched);

int pthread\_attr\_getinheritsched(const pthread\_attr\_t \*attr,int \*inheritsched);

attr：线程属性

inheritsched :PTHREAD\_EXPLICIT\_SCHED和PTHREAD\_INHERIT\_SCHED

返回值：成功0，错误非0错误码。

**·设置和获取线程堆栈的大小。**

int pthread\_attr\_getstacksize(const pthread\_attr\_t \*,size\_t \*stacksize);

int pthread\_attr\_setstacksize(pthread\_attr\_t \*attr ,size\_t stacksize);

attr线程属性变量

stacksize堆栈大小

返回值：成功0，错误非0错误码。

**·设置和获取线程堆栈的大小和地址**

int pthread\_attr\_setstack(pthread\_attr\_t \*attr, void \*stackaddr, size\_t stacksize);

int pthread\_attr\_getstack(const pthread\_attr\_t \*attr, void \*\*stackaddr, size\_t \*stacksize);

返回值：成功0，错误非0错误码。

**·设置和获取线程栈末尾的警戒缓冲区大小。**

int pthread\_attr\_getguardsize(const pthread\_attr\_t \*restrict attr,size\_t \*restrictguardsize);

int pthread\_attr\_setguardsize(pthread\_attr\_t \*attr ,size\_t guardsize);

运行时也可设置线程属性：

返回值：成功0，错误非0错误码。

**·设置/获取线程调度策略和优先级：**

int pthread\_setschedparam(pthread\_t thread, int policy, struct sched\_param \*param);

int pthread\_getschedparam(pthread\_t thread, int \*policy, struct sched\_param \*param)

返回值：成功0，错误非0错误码。

**·设置/获取线程分离属性：**

int pthread\_detach(pthread\_t thread);

返回值：成功0，错误非0错误码。

**·获取最大最小优先级**

#include<sched.h>

int sched\_get\_priority\_max(int policy);

int sched\_get\_priority\_min(int policy);

返回值：成功返回最大或最小优先级，出错返回-1，设置了标准错误描述，可使用perror函数查看。

**·获取进程调度参数**

#include<sched.h>

int sched\_setparam(pid\_t pid, const struct sched\_param \*param);

int sched\_getparam(pid\_t pid, struct sched\_param \*param);

返回值：成功返回0,败返回-1，设置了标准错误描述，可使用perror函数查看。

**·获取进程调度策略和调度参数**

#include<sched.h>

int sched\_setscheduler(pid\_t pid, int policy, const struct sched\_param \*param);

int sched\_getscheduler(pid\_t pid);

返回值：成功sched\_setscheduler返回0，sched\_getscheduler返回线程策略，失败都返回-1，设置了标准错误描述，可使用perror函数查看。

实例：

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <pthread.h>

void**\***thread\_function**(**void **\***arg**);**

charmessage**[]=**"HelloWorld"**;**

int thread\_finished**=**0**;**

int main**()**

**{**

intres**=**0**;**

pthread\_t a\_thread**;**

void **\***thread\_result**;**

pthread\_attr\_t thread\_attr**;** //定义属性

struct sched\_param scheduling\_value**;**

res**=**pthread\_attr\_init**(&**thread\_attr**);** //属性初始化

**if(**res**!=**0**)**

**{**

perror**(**"Attribute creationfailed"**);**

exit**(**EXIT\_FAILURE**);** //EXIT\_FAILURE -1

**}**

//设置调度策略

res**=**pthread\_attr\_setschedpolicy**(&**thread\_attr**,**SCHED\_OTHER**);**

**if(**res**!=**0**)**

**{**

perror**(**"Setting schedpolicy failed"**);**

exit**(**EXIT\_FAILURE**);**

**}**

//设置脱离状态

res**=**pthread\_attr\_setdetachstate**(&**thread\_attr**,**PTHREAD\_CREATE\_DETACHED**);**

//创建线程

res**=**pthread\_create**(&**a\_thread**,&**thread\_attr**,**thread\_function**,(**void **\*)**message**);**

**if(**res**!=**0**)**

**{**

perror**(**"Thread creation failed"**);**

exit**(**EXIT\_FAILURE**);**

**}**

//获取最大优先级别

int max\_priority **=**sched\_get\_priority\_max**(**SCHED\_OTHER**);**

//获取最小优先级

int min\_priority**=**sched\_get\_priority\_min**(**SCHED\_OTHER**);**

//重新设置优先级别

scheduling\_value**.**sched\_priority**=**min\_priority**+**5**;**

//设置优先级别

res**=**pthread\_attr\_setschedparam**(&**thread\_attr**,&**scheduling\_value**);**

pthread\_attr\_destroy**(&**thread\_attr**);**

**while(!**thread\_finished**)**

**{**

printf**(**"Waiting for thread to say it's finished...\n"**);**

sleep**(**1**);**

**}**

printf**(**"Other thread finished,bye!\n"**);**

exit**(**EXIT\_SUCCESS**);**

**}**

void**\***thread\_function**(**void **\***arg**)**

**{**

printf**(**"thread\_function is running. Argument was%s\n"**,(**char**\*)**arg**);**

sleep**(**4**);**

printf**(**"Second thread setting finishedflag,andexiting now\n"**);**

thread\_finished**=**1**;**

pthread\_exit**(NULL);**

**}**

# LINUX 网络编程

## 1 TCP/IP模型

### 1.1 TCP/IP 协议概述

协议 protocol：通信双方必须遵循的规矩由iso规定rpc 文档osi参考模型：（应-表-会-传-网-数-物）应用层、表示层、会话层、传输层、网络层、数据链路层、物理层。

tcp/ip 模型 4 层:

应用层{http超文本传输协议；ftp文件传输协议；telnet远程登录；ssh安全外壳协议；stmp简单邮件发送；pop3收邮件}

传输层{tcp 传输控制协议；udp 用户数据包协议}

网络层{ip 网际互联协议； icmp 网络控制消息协议； igmp 网络组管理协议}

网络接口层{arp 地址转换协议；rarp 反向地址转换协议；mpls 多协议标签交换}

TCP 协议：传输控制协议，面向连接的协议，能保证传输安全可靠， 速度慢（有 3 次握手）。

UDP 协议：用户数据包协议，非面向连接，速度快，不可靠。

通常是 ip 地址后面跟上端口号：ip 用来定位主机 port 区别应用（进程）

http 的端口号 80 ssh-->22 telnet-->23 ftp-->21 用户自己定义的通常要大于 1024

### 1.2 OSI 参考模型及 TCP/IP 参考模型



TCP/IP 协议族的每一层的作用：

·网络接口层：负责将二进制流转换为数据帧，并进行数据帧的发送和接收。要注意的是数据帧是 独立的网络信息传输单元。

·网络层：负责将数据帧封装成IP 数据报，并运行必要的路由算法。

·传输层：负责端对端之间的通信会话连接和建立。传输协议的选择根据数据传输方式而定。

·应用层：负责应用程序的网络访问，这里通过端口号来识别各个不同的进程

TCP/IP 协议族的每一层协议的相关注解：

·ARP：（地址转换协议）用于通过ip获得同一物理网络中的硬件主机mac地址。

·MPLS：（多协议标签交换）很有发展前景的下一代网络协议。

·IP：（网际互联协议）负责在主机和网络之间寻址和路由数据包。

·ICMP：（网络控制消息协议）用于发送报告有关数据包的传送错误的协议。

·IGMP：（网络组管理协议）被IP主机用来向本地多路广播路由器报告主机组成员的协议。

·TCP：（传输控制协议）为应用程序提供可靠的通信连接。适合于一次传输大批数据的情况。并适用于要求得到响应的应用程序。

·UDP：（用户数据包协议）提供了无连接通信，且不对传送包进行可靠的保证。适合于一次传输 少量数据。





### 1.3 TCP 协议

#### 1.3.1 概述

TCP是TCP/IP 体系中面向连接的运输层协议，它提供全双工和可靠交付的服务。它采用许多机制来确保端到端结点之间的可靠数据传输，如采用序列号、确认重传、滑动窗口等。

首先，TCP 要为所发送的每一个报文段加上序列号，保证每一个报文段能被接收方接收，并只被正 确的接收一次。

其次，TCP 采用具有重传功能的积极确认技术作为可靠数据流传输服务的基础。这里“确认”是指 接收端在正确收到报文段之-后向发送端回送一个确认（ACK）信息。发送方将每个已发送的报文段备份 在自己的缓冲区里，而且在收到相应的确认之前是不会丢弃所保存的报文段的。“积极”是指发送发在每一个报文段发送完毕的同时启动一个定时器，加入定时器的定时期满而关于报文段的确认信息还没有达到，则发送发认为该报文段已经丢失并主动重发。为了避免由于网络延时引起迟到的确认和重复的确 认，TCP规定在确认信息中捎带一个报文段的序号，使接收方能正确的将报文段与确认联系起来。

最后，采用可变长的滑动窗口协议进行流量控制，以防止由于发送端与接收端之间的不匹配而引起 的数据丢失。这里所采用的滑动窗口协议与数据链路层的滑动窗口协议在工作原理上完全相同，唯一的 区别在于滑动窗口协议用于传输层是为了在端对端节点之间实现流量控制，而用于数据链路层是为了在相邻节点之间实现流量控制，大小是动态改变的。TCP 采用可变长的滑动窗口，使得发送端与接收端可根据自己的 CPU 和 数据缓存资源对数据发送和接收能力来进行动态调整，从而灵活性更强，也更合理。

#### 1.3.2 三次握手

在利用 TCP 实现源主机和目的主机通信时，目的主机必须同意，否则 TCP 连接无法建立。为了确 保 TCP 连接的成功建立，TCP 采用了一种称为三次握手的方式，三次握手方式使得“序号/确认号”系 统能够正常工作，从而使它们的序号达成同步。如果三次握手成功，则连接建立成功，可以开始传送数 据信息。

其三次握手分别为：

1）源主机A 的 TCP 向主机B 发送连接请求报文段，其首部中的 SYN（同步）标志位应置为1，表 示想跟目标主机 B 建立连接，进行通信，并发送一个同步序列号 X（例：SEQ=100）进行同步，表明在 后面传送数据时的第一个数据字节的序号为 X+1（即 101）。

2）目标主机 B 的 TCP 收到连接请求报文段后，如同意，则发回确认。再确认报中应将 ACK 位和 SYN 位置为 1.确认号为 ack为X+1，同时也为自己选择一个序号Y。

3）源主机 A 的 TCP 收到目标主机 B 的确认后要向目标主机 B 给出确认。其 ACK 置为1，确认号 为 Y+1，而自己的序号为 X+1。TCP 的标准规定，SYN 置 1的报文段要消耗掉一个序号。

运行客户进程的源主机 A 的 TCP 通知上层应用进程，连接已经建立。当源主机 A 向目标主机 B 发 送第一个数据报文段时，其序号仍为 X+1，因为前一个确认报文段并不消耗序号。

当运行服务进程的目标主机 B 的 TCP 收到源主机 A 的确认后，也通知其上层应用进程，连接已经 建立。至此建立了一个全双工的连接。

三次握手：为应用程序提供可靠的通信连接。适合于一次传输大批数据的情况。并适用于要求得到

响应的应用程序。



采用三次握手的原因是：因为信道不可靠，了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误。

#### 1.3.3 TCP 数据报头

TCP 头信息



·源端口、目的端口：16位长。标识出远端和本地的端口号。

·序号：32 位长。标识发送的数据报的顺序。

·确认号：32 位长。希望收到的下一个数据报的序列号。

·TCP 头长：4 位长。表明 TCP 头中包含多少个 32 位字。最小5\*4B=20B，最大为16\*4B=60B。

·6 位未用。

·ACK：ACK 位置 1 表明确认号是合法的。如果 ACK 为 0，那么数据报不包含确认信息，确认字段被省略。

·PSH：表示是带有 PUSH 标志的数据。接收方因此请求数据报一到便可送往应用程序而不必等到缓冲 区装满时才发送。

·RST：用于复位由于主机崩溃或其他原因而出现的错误的连接。还可以用于拒绝非法的数据报或拒绝 连接请求。

·SYN：用于建立连接。

·FIN：用于释放连接。

·URG ：紧急位，表示应该优先传输

·窗口大小：16位长。窗口大小字段表示在确认了字节之后还可以发送多少个字节。

·校验和：16 位长。是为了确保高可靠性而设置的。它校验头部、数据和伪 TCP 头部之和。

·可选项：0 个或多个 32位字。包括最大 TCP 载荷，窗口比例、选择重复数据报等选项。

#### 1.3.4 四次挥手

由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

（1） TCP客户端发送一个FIN，用来关闭客户到服务器的[数据传送](http://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E4%BC%A0%E9%80%81)。

（2） 服务器收到这个FIN，它发回一个ACK，确认序号为收到的序号加1。和SYN一样，一个FIN将占用一个序号。

（3） 服务器关闭客户端的连接，发送一个FIN给客户端。

（4） 客户端发回ACK[报文](http://baike.baidu.com/item/%E6%8A%A5%E6%96%87)确认，并将确认序号设置为收到序号加1。

### 1.4 UDP 协议

#### 1.4.1 概述

UDP 即用户数据报协议，它是一种无连接协议，因此不需要像 TCP 那样通过三次握手来建立一个 连接。同时，一个 UDP 应用可同时作为应用的客户或服务器方。由于 UDP 协议并不需要建立一个明确 的连接，因此建立 UDP 应用要比建立TCP 应用简单得多。

它比TCP 协议更为高效，也能更好地解决实时性的问题。如今，包括网络视频会议系统在内的众多 的客户/服务器模式的网络应用都使用UDP 协议。

#### 1.4.2 Udp 数据包头格式



### 1.5 协议的选择

（1）对数据可靠性的要求 对数据要求高可靠性的应用需选择 TCP 协议，如验证、密码字段的传送都是不允许出错的，而对数 据的可靠性要求不那么高的应用可选择 UDP 传送。

（2）应用的实时性 TCP 协议在传送过程中要使用三次握手、重传确认等手段来保证数据传输的可靠性。使用 TCP 协议 会有较大的时延，因此不适合对实时性要求较高的应用，如 VOIP、视频监控等。相反，UDP 协议则在这 些应用中能发挥很好的作用。

（3）网络的可靠性 由于 TCP 协议的提出主要是解决网络的可靠性问题，它通过各种机制来减少错误发生的概率。因此， 在网络状况不是很好的情况下需选用 TCP 协议（如在广域网等情况），但是若在网络状况很好的情况下 （如局域网等）就不需要再采用 TCP 协议，而建议选择 UDP 协议来减少网络负荷。

## 2 网络相关概念

### 2.1 套接口的概念

套接口，也叫“套接字”。是操作系统内核中的一个数据结构，它是网络中的节点进行相互通信的 门户。它是网络进程的 ID。网络通信，归根到底还是进程间的通信（不同计算机上的进程间通信）。在每一个节点（计算机或路由）都有一个网络地址，也就是 IP 地址。两个进程通信时，首先要 确定各自所在的网络节点的网络地址。但是，网络地址只能确定进程所在的计算机，而一台计算机上很 可能同时运行着多个进程，所以仅凭网络地址还不能确定到底是和网络中的哪一个进程进行通信，因此 套接口中还需要包括其他的信息，也就是端口号（PORT）。在一台计算机中，一个端口号一次只能分配 给一个进程，也就是说，在一台计算机中，端口号和进程之间是一一对应关系。所以，使用端口号和网 络地址的组合可以唯一的确定整个网络中的一个网络进程。

例如，如网络中某一台计算机的 IP 为 10.92.20.160，操作系统分配给计算机中某一应用程序进程 的端口号为 1500，则此时 10.92.20.160 1500 就构成了一个套接口。

### 2.2 端口号的概念

在网络技术中，端口大致有两种意思：一是物理意义上的端口，如集线器、交换机、路由器等用于 连接其他网络设备的接口。二是指 TCP/IP 协议中的端口，端口号的范围从 0~65535，一类是由互联网指派名字和号码公司ICANN负责分配给一些常用的应用程序固定使用的“周知的端口”， 其值一般为0~1023. 例如 http 的端口号是 80，ftp 为 21，ssh 为 22，telnet 为 23 等。还有一类是用户自己定义的，通常 是大于 1024 的整型值。

### 2.3 ip地址的表示

通常用户在表达 IP 地址时采用的是点分十进制表示的数值（或者是为冒号分开的十进制 Ipv6 地 址），而在通常使用的 socket 编程中使用的则是二进制值，这就需要将这两个数值进行转换。 ipv4 地址：32bit, 4 字节，通常采用点分十进制记法。

例如对于：10000000 00001011 00000011 00011111

点分十进制表示为：128.11.3.31





0.0.0.0已经不是一个真正意义上的IP地址了。它表示的是这样一个集合：所有不清楚的主机和目的网络。这里的“不清楚”是指在本机的路由表里没有特定条目指明如何到达。对本机来说，它就是一个“收容所”，所有不认识的“三无”人员，一律送进去。如果你在网络设置中设置了缺省网关，那么Windows系统会自动产生一个目的地址为0.0.0.0的缺省路由。

255.255.255.255 指的是本网段的所有节点（可以是主机或路由）。边路由器在接收到到目的地址为255.255.255.255的数据时不会进行转发。又叫做限制广播地址。

### 2.4 socket 概念

Linux 中的网络编程是通过socket 接口来进行的。 socket 是一种特殊的 I/O 接口，既是管道，也是一种文件描述符。它是一种常用的进程之间通信机制，通过它不仅能实现本地机器上的进程之间的通信，而且通过网络能够在不同机器上的进程之间进行通信。 每一个socket 都用一个半相关描述{协议、本地地址、本地端口}来表示；一个完整的套接字则用一个相关描述{协议、本地地址、本地端口、远程地址、远程端口}来表示。socket 也有一个类似于打开文件的函数调用，该函数返回一个整型的 socket 描述符，随后的连接建立、数据传输等操作都是通过 socket 来实现的；

### 2.5 socket 类型

（1）stream socket（SOCK\_STREAM） 也叫流式套接字，用于TCP 通信

流式套接字提供可靠的、面向连接的通信流；它使用 TCP 协议，从而保证了数据传输的正确性和顺 序性。用于可靠性要求高，非实时，网络不好，或者一次传输数据较大。

（2）datagram socket（SOCK\_DGRAM）也叫数据报套接字 ，用于UDP 通信

数据报套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保 证是可靠、无差错的。它使用数据报协议 UDP。 用于可靠性要求不高，实时，网络好，或者一次传输数据较小。

（3）原始socket（SOCK\_RAW） 用于新的网络协议实现的测试等

原始套接字允许对底层协议如 IP 或ICMP 进行直接访问，它功能强大但使用较为不便，主要用于一 些协议的开发。

### 2.6 socket信息数据结构

**ipv4 socket:**

struct sockaddr

{

unsigned short sa\_family; /\*地址族\*/

char sa\_data[14]; /\*14 字节的协议地址，包含该 socket 的 IP 地址和端口号。\*/

};

一共16字节，此结构体使用较为不便，一般使用下面的结构体，但需要强制转换。

struct sockaddr\_in

{

short int sin\_family; /\*地址族\*/

unsigned short int sin\_port; /\*端口号\*/

struct in\_addr sin\_addr; /\*IP 地址\*/

unsigned char sin\_zero[8]; /\*填充0 以保持与 struct sockaddr 同样大小\*/

};

struct in\_addr

{

unsigned int s\_addr; /\*32 位IPv4 地址，网络字节序 \*/

};

头文件<netinet/in.h>

sa\_family: AF\_INET IPv4 协议； AF\_INET6 IPv6 协议。

**ipv6 socket:**

struct sockaddr\_in6

{

\_\_SOCKADDR\_COMMON (sin6\_); /\* 家族协议 \*/

in\_port\_t sin6\_port; /\* Transport layer port # 端口号 \*/

uint32\_t sin6\_flowinfo; /\* IPv6 flow information ipv6中的流标签字段 \*/

struct in6\_addr sin6\_addr; /\* IPv6 address ipv6的地址信息 \*/

uint32\_t sin6\_scope\_id; /\* IPv6 scope-id ipv6的接口范围 \*/

};

struct in6\_addr

{

union

{

uint8\_t \_\_u6\_addr8[16]; // 128 bit

#if defined \_\_USE\_MISC || defined \_\_USE\_GNU

uint16\_t \_\_u6\_addr16[8]; // 64 bit

uint32\_t \_\_u6\_addr32[4]; // 32 bit

#endif

} \_\_in6\_u;

#define s6\_addr \_\_in6\_u.\_\_u6\_addr8

#if defined \_\_USE\_MISC || defined \_\_USE\_GNU

# define s6\_addr16 \_\_in6\_u.\_\_u6\_addr16

# define s6\_addr32 \_\_in6\_u.\_\_u6\_addr32

#endif

};

结构体中的unit8\_t,unit16\_t,unit32\_t这三个数据类型如下：

typedef unsigned char uint8\_t;

typedef unsigned short int uint16\_t;

#ifndef \_\_uint32\_t\_defined

typedef unsigned int uint32\_t;

# define \_\_uint32\_t\_defined

#endif

### 2.4 数据存储优先顺序的转

计算机数据存储有两种字节优先顺序：高位字节优先（称为大端模式）和低位字节优先（称为小端模式）。内存的低地址存储数据的低位，高地址存储数据的高位的方式叫小端模式。内存的高地址 存储数据的低位，低地址存储数据高位的方式称为大端模式。

列入:对于内存中存放的数 0x12345678 来说。

注：十六进制每位表示4个二进制位，两个十六进制表示一个字节

如果是采用大端模式存放的，则其真实的数是：0x12345678 （相反），与平时书写相同

如果是采用小端模式存放的，则其真实的数是：0x78563412 （相同），与平时书写相反

如果称某个系统所采用的字节序为主机字节序，则它可能是小端模式的，也可能是大端模式的。而端口号IP 地址都是以网络字节序存储的，不是主机字节序，网络字节序都是大端模式。要把主机字 节序和网络字节序相互对应起来，需要对这两个字节存储优先顺序进行相互转化。这里用到四个函数： htons(),ntohs(),htonl()和 ntohl().这四个地址分别实现网络字节序和主机字节序的转化，这里的 h 代表 host,n 代表 network，s 代表 short，l 代表 long。通常 16 位的端口号用 s 代表，而32位的 IP 地址用 l 来代表。

#include <arpa/inet.h>

uint32\_t htonl(uint32\_t hostlong);

uint16\_t htons(uint16\_t hostshort);

uint32\_t ntohl(uint32\_t netlong);

uint16\_t ntohs(uint16\_t netshort);

成功返回要转换的字节序列，

### 2.5 地址格式转化

通常用户在表达地址时采用的是点分十进制表示的数值（或者是为冒号分开的十进制 Ipv6 地址）， 而在通常使用的 socket 编程中使用的则是 32 位的网络字节序的二进制值，这就需要将这两个数值进行 转换。这里在 Ipv4 中用到的函数有 inet\_aton()、inet\_addr()和 inet\_ntoa()，而 IPV4 和 Ipv6 兼容的函数有 inet\_pton()和 inet\_ntop()。

IPv4 的函数原型：

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

int inet\_aton(const char \*straddr, struct in\_addr \*addrptr);

char \*inet\_ntoa(struct in\_addr inaddr);

in\_addr\_t inet\_addr(const char \*straddr); //in\_addr\_t就是 unsigned int,代表 s\_addr

函数 inet\_aton()：将点分十进制数的 IP 地址转换成为网络字节序的 32 位二进制数值。返回 值：成功返回1，不成功返回 0。没有设置了标准错误描述，不可使用perror函数查看。

参数 straddr：存放输入的点分十进制数 IP 地址字符串。

参数 addrptr：传出参数，保存网络字节序的 32 位二进制数值。

函数inet\_ntoa()：将网络字节序的32位二进制数值转换为点分十进制的 IP 地址。

成功返回地址，出错返回NULL。

函数 inet\_addr()：功能与 inet\_aton 相同，但是结果传递的方式不同。inet\_addr()若成功则返回 32 位二进制的网络字节序ip地址,与in\_addr.s\_addr相同,出错返回INADDR\_NONE一般为-1。由于计算机中存放的-1的补码与255.255.255.255相同，而后者是一个合法地址。有可能出现bug。

IPv4 和 IPv6 的函数原型：

#include<arpa/inet.h>

int inet\_pton(int family,const char\*src, void\*dst);

const char\*inet\_ntop(int family, const void \*src,char\*dst,socklen\_t len);

函数 inet\_pton 跟 inet\_aton 实现的功能类似，。成功返回1，出错返回-1；字符串地址与协议类型不匹配返回0, 设置了标准错误描述，可使用perror函数查看。

参数family，该参数指定为 AF\_INET，表 示是 IPv4 协议，如果是 AF\_INET6，表示 IPv6 协议其中

参数src指向字符串形式的地址；

参数dst指向sockaddr\_in6的in6\_addr结构体或sockaddr\_in的in\_addr结构体；

函数 inet\_ntop跟inet\_ntoa 类似，成功返回指向字符串的指针，出错返回NULL,设置了标准错误描述，可使用perror函数查看。

参数family，该参数指定为 AF\_INET，表 示是 IPv4 协议，如果是 AF\_INET6，表示 IPv6 协议其中

参数dst指向字符串形式的地址；

参数src指向in6\_addr结构体或in\_addr结构体，成员为网络字节序；

参数len表示表示转换之后的字符串ip的长度（字符串的长度）。

### 2.6 名字地址转化

通常，人们在使用过程中都不愿意记忆冗长的 IP 地址，尤其到 Ipv6 时，地址长度多达 128 位，那 时就更加不可能一次性记忆那么长的 IP 地址了。因此，使用主机名或域名将会是很好的选择。

主机名与域名的区别：主机名通常在局域网里面使用，通过/etc/hosts 文件，主机名可以解析到对应的 ip； 域名通常是再 internet 上使用。

注意：https://mail.163.com和https://[www.163.com](http://www.163.com)中，163.com时域名，www和mail是主机名，https://是协议。

众所周知，百度的域名为：www.baidu.com，而这个域名其实对应了一个百度公司的 IP 地址，那么 百度公司的 IP 地址是多少呢？我们可以利用 ping www.baidu.com 来得到百度公司的 ip 地

在linux中，有一些函数可以实现主机名和ip或者主机名与域名的转化，最常见的有 gethostbyname()、 gethostbyaddr()等，它们都可以实现 IPv4 和 IPv6 的地址和主机名之间的转化。其中 gethostbyname() 是通过域名或主机名获取主机信息，gethostbyaddr()则是通过ip地址获取主机信息，是将 IP 地址转化为主机名。

函数原型：

#include <netdb.h>

struct hostent\* gethostbyname(const char\* hostname);

struct hostent\* gethostbyaddr(const void\* addr, socklen\_t len, int family);

struct hostent

{

char \*h\_name; /\*正式主机名\*/

char \*\*h\_aliases; /\*主机别名\*/

int h\_addrtype; /\*主机 IP 地址类型 IPv4 为 AF\_INET\*/

int h\_length; /\*主机 IP 地址字节长度，对于 IPv4 是 4 字节，即 32 位\*/

char \*\*h\_addr\_list; /\*主机的 IP 地址列表,网络字节序，ipv4为struct in\_addr可用inet\_ntoa或者inet\_ntop,ipv6 为 struct in6\_addr只能用inet\_ntop\*/

}

#define h\_addr h\_addr\_list[0] /\*保存的是 ip 地址\*/

函数 gethostbyname()：用于将域名（www.baidu.com）或主机名转换为 IP 地址。成功返回指针，出错返回NULL。

参数 hostname 指向存放域名或主机名的字符串。

函数 gethostbyaddr()：用于将 IP 地址转换为域名或主机名。成功返回指针，出错返回NULL。

参数 addr 是一个网络字节序的ip，此时这个 ip 地址不是普通的字符串，而是要通过函数inet\_aton()转换。

参数len 为 IP 地址的长度，AF\_INET 为 4。

参数family 可用 AF\_INET：Ipv4 或 AF\_INET6：Ipv6。必须在/etc/hosts 中有配置

Example1：将百度的www.baidu.com 转换为 ip 地址 #include <netdb.h>

#include <sys/socket.h>

#include <stdio.h>

int main**(**int argc**,**char**\*\***argv**)**

**{**

char**\***ptr**,\*\***pptr**;**

struct hostent **\***hptr**;**

char str**[**40**]={**'\0'**};** /\* 取得命令后第一个参数，即要解析的域名或主机名 \*/

ptr**=**argv**[**1**];** //如www.baidu.com /\* 调用gethostbyname()。结果存在hptr 结构中 \*/

**if((**hptr**=**gethostbyname**(**ptr**))==NULL)**

**{**

printf**(**"gethostbyname error for host:%s\n"**,**ptr**);**

**return** 0**;**

**}**

/\* 将主机的规范名打出来 \*/

printf**(**"official host name:%s\n"**,**hptr**->**h\_name**);**

/\* 主机可能有多个别名，将所有别名分别打出来 \*/

**for(**pptr**=**hptr**->**h\_aliases**;\***pptr **!=NULL;** pptr**++)**

printf**(**"alias:%s\n"**,\***pptr**);**

/\* 根据地址类型，将地址打出来 \*/

/\* 将刚才得到的所有地址都打出来。其中调用了inet\_ntop()函数 \*/

**switch(**hptr**->**h\_addrtype**)**

**{**

**case** AF\_INET**:**

**for(**pptr**=**hptr**->**h\_addr\_list**;\***pptr**!=NULL;**pptr**++)**

printf**(**"address:%s\n"**,**inet\_ntop**(**hptr**->**h\_addrtype**,\***pptr**,**str**,**16**));**

**break;**

caseAF\_INET6**:**

**for(**pptr**=**hptr**->**h\_addr\_list**;;\***pptr**!=NULL;**pptr**++)**

printf**(**"address:%s\n"**,**inet\_ntop**(**hptr**->**h\_addrtype**,\***pptr**,**str**,**40**));**

**break;**

**default:**

printf**(**"unknown address type\n"**);**

**break;**

**}**

**return** 0**;**

**}**

注意：这两个函数是不可重入函数，在posix标准中已经被废弃；用下面两个函数替代：

int getnameinfo(const struct sockaddr \*sa, socklen\_t salen, char \*host, socklen\_t hostlen, char \*serv,

socklen\_t servlen, int flags);

int getaddrinfo(const char \*node, const char \*service, const struct addrinfo \*hints, struct addrinfo \*\*res);

这两个函数都是成功返回0，出错返回错误码。

struct addrinfo

{

int ai\_flags;

int ai\_family;

int ai\_socktype;

int ai\_protocol;

socklen\_t ai\_addrlen;

struct sockaddr \*ai\_addr;

char \*ai\_canonname;

struct addrinfo \*ai\_next;

};

### 2.7 获取当前地址和远程地址

int getsockname(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

int getpeername(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

getsockname获取sockfd绑定的本地地址

getpeername获取sockfd绑定的远程地址

成功返回0，出错返回-1，设置了标准错误描述，可使用perror查看。

## 3 socket 编程

### 3.1 使用 TCP 协议的流程图

TCP 通信的基本步骤如下：

服务端：socket---bind---listen---while(1){---accept---recv---send---close---}---close

客户端：socket-------------------------------connect---send---recv--------------close



#### 3.1.1 服务器端

（1）头文件包含：

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<arpa/inet.h>

#include<unistd.h>

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

（2）socket 函数：生成一个套接口描述符。

原型：

int socket(int domain,int type,int protocol);

返回值：成功则返回套接口描述符，出错返回-1。设置了标准错误描述，可使用perror函数查看。

参数：domain{AF\_INET：Ipv4 网络协议 AF\_INET6：IPv6 网络协议}

参数type｛tcp：SOCK\_STREAM udp：SOCK\_DGRAM}

参数protocol指定socket 所使用的传输协议编号。通常为0.

常用实例：

int sfd**=**socket**(**AF\_INET**,**SOCK\_STREAM**,**0**);**

**if(**sfd **==-**1**)**

**{**

perror**(**"socket"**);**

exit**(-**1**);**

**}**

（3）bind 函数：用来绑定一个端口号和IP 地址，使套接口与指定的端口号和 IP 地址相关联。

原型：

int bind(int sockfd,struct sockaddr\*my\_addr, socklen\_t addrlen);

返回值：成功则返回0，出错返回-1, 设置了标准错误描述，可使用perror函数查看。

参数sockfd为前面 socket 的返回值。

参数my\_addr：为结构体指针变量 对于不同的socket domain 定义了一个通用的数据结构

struct sockaddr //此结构体不常用

{

unsigned short int sa\_family; //调用socket（）时的 domain 参数，即 AF\_INET 值。

charsa\_data[14]; //最多使用 14 个字符长度

};

此sockaddr 结构会因使用不同的 socket domain 而有不同结构定义， 例如使用 AF\_INET domain，其 socketaddr 结构定义便为

struct sockaddr\_in //常用的结构体

{

unsigned short int sin\_family; //即为 sa\_family AF\_INET

uint16\_t sin\_port; //为使用的 port 编号

struct in\_addr sin\_addr; //为 IP 地址

unsigned charsin\_zero[8]; //未使用

};

struct in\_addr

{

uint32\_t s\_addr;

};

参数addrlen为sockaddr的结构体长度。通常是计算 sizeof(struct sockaddr);

常用实例：

struct sockaddr\_in my\_addr**;** //定义结构体变量

memset**(&**my\_addr**,**0**,sizeof(**struct sockaddr**));**//将结构体清空

//或bzero(&my\_addr, sizeof(struct sockaddr));

my\_addr**.**sin\_family **=**AF\_INET**;** //表示采用 Ipv4 网络协议

my\_addr**.**sin\_port**=**htons**(**8888**);** //表示端口号为 8888，通常是大于 1024的一个值。

//htons()用来将参数指定的16位 host short 转换成网络字符顺序

my\_addr**.**sin\_addr**.**s\_addr**=**inet\_addr**(**"192.168.0.101"**);**

//inet\_addr()用来将IP地址字符串转

//换成网络所使用的二进制数字，如果为INADDR\_ANY，这表示服务器自动填充本机IP 地址。

**if(**bind**(**sfd**,** **(**struct sockaddr**\*)&**my\_str**,sizeof(**struct sockaddr**))==-**1**)**

**{**

perror**(**"bind"**);**

close**(**sfd**);**

exit**(-**1**);**

**}**

注：通过将 my\_addr.sin\_port 置为 0，函数会自动为你选择一个未占用的端口来使用。同样，通过将 my\_addr.sin\_addr.s\_addr 置为INADDR\_ANY，系统会自动填入本机IP 地址。

（4）listen 函数：使服务器的这个端口和 IP 处于监听状态，等待网络中某一客户机的连接请求。如果客户端有连接请求，端口就会接受这个连接。

原型：

int listen(int sockfd,int backlog);

返回值：成功则返回0，出错返回-1, 设置了标准错误描述，可使用perror函数查看。

参数：sockfd为前面 socket 的返回值.即 sfd

参数backlog指定同时能处理的最大连接要求，通常为10 或者5。最大值可设至 128

常用实例：

if(listen(sfd,10)==-1)

{

perror("listen");

close(sfd);

exit(-1);

}

（5）accept 函数：接受远程计算机的连接请求，建立起与客户机之间的通信连接。服务器处于监听状态时， 如果某时刻获得客户机的连接请求，此时并不是立即处理这个请求，而是将这个请求放在等待队列中， 当系统空闲时再处理客户机的连接请求。当accept 函数接受一个连接时，会返回一个新的 socket 标识符，以后的数据传输和读取就要通过这个新的 socket 编号来处理，原来参数中的socket标识符 也可以继续使用，继续监听其它客户机的连接请求。（也就是说，类似于移动营业厅，如果有客户打电话给 10086，此时服务器就会请求连接，处理一些事务之后，就通知一个话务员接听客户的电话，也就是说，后面的所有操作，此时已经于服务器没有关系，而是话务员跟客户的交流。对应过来，客户请求连接我们的服务器， 我们服务器先做了一些绑定和监听等等操作之后，如果允许连接，则调用 accept 函数产生一个新的套接字，然后用这个新的套接字跟我们的客户进行收发数据。也就是说，服务器跟一个客户端连接成功，会有两个套接字。）

原型：

int accept(int s,struct sockaddr\*addr, socklen\_t \*addrlen);

返回值：成功则返回新的 socket 处理代码new\_fd，出错返回-1, 设置了标准错误描述，可使用perror函数查看。

参数s为前面 socket 的返回值.即sfd

参数addr为结构体指针变量，和 bind 的结构体是同种类型的，系统会把远程主机的信息（远程主机的地址和端口号信息）保存到这个指针所指的结构体中。

参数addrlen表示结构体的长度，为整型指针

常用实例：

struct sockaddr\_in clientaddr**;**

memset**(&**clientaddr**,**0**,sizeof(**struct sockaddr**));**

int addrlen **=sizeof(**struct sockaddr**);**

int new\_fd **=**accept**(**sfd**,** **(**struct sockaddr**\*)&**clientaddr**,&**addrlen**);**

**if(**new\_fd **==-**1**)**

**{**

perror**(**"accept"**);**

close**(**sfd**);**

exit**(-**1**);**

**}**

printf**(**"%s%dsuccess connect\n"**,**inet\_ntoa**(**clientaddr**.**sin\_addr**),**ntohs**(**clientaddr**.**sin\_port**));**

（6）recv 函数：用新的套接字来接收远端主机传来的数据，并把数据存到由参数 buf 指向的内存空间

原型：

int recv(int sockfd,void\*buf,int len,int flags);

返回值：成功则返回实际接收到的字符数，可能会少于你所指定的接收长度。出错返回-1，返回0，对端断开。设置了标准错误描述，可使用perror函数查看。

注意：有些域中的数据报套接字（例如UNIX和Internet域）允许零长度的数据报。当接收到这样的数据报时，返回值为0。如果要从流套接字接收的请求的字节数为0，则也可以返回值0。

errno被设为以下的某个值

EAGAIN或EWOULDBLOCK，套接字被标记为非阻塞，而recv是阻塞，或接收超时。POSIX.1-2001允许为这种情况返回错误，并且不要求这些常量具有相同的值，因此应检查两种可能性。对非阻塞socket而言，EAGAIN不是一种错误

EBADF，参数sockfd是一个无效的描述符。

ECONNREFUSED，远程主机拒绝允许网络连接（通常是因为它没有运行请求的服务）。

EFAULT，接收缓冲区指针指向进程的地址空间之外。

EINTR，接收操作被信号中断。

EINVAL，传递无效的参数。

ENOMEM，无可用内存。

ENOTCONN，套接字是面向连接的但没有连接。

ENOTSOCK，文件描述符sockfd不引用套接字。

参数：sockfd为前面 accept 的返回值.即 new\_fd，也就是新的套接字。

参数buf表示缓冲区

参数len表示缓冲区的长度

参数flags通常为0或者以下组合

MSG\_PEEK 查看数据但并不移除数据。下一个recv或类似函数仍然返回此数据。

MSG\_OOB请求带外数据。带外数据的意义和语义是协议特定的。

MSG\_WAITALL在SOCK\_STREAM 的套接字上，请求功能块，直到可以返回全部数据。如果套接字是基于消息的套接字，当信号被捕获，或连接终止，或指定了MSG\_PEEK，或者套接字出现错误等待，则该功能可能返回较小的数据量。

如果flags为0，则和read,write一样的操作

常用实例：

charbuf**[**512**]={**0**};**

**if(**recv**(**new\_fd**,** buf**,** **sizeof(**buf**),**0**)==-**1**)**

**{**

perror**(**"recv"**);**

close**(**new\_fd**);**

close**(**sfd**);**

exit**(-**1**);**

**}**

puts**(**buf**);**

（7）send 函数：用新的套接字发送数据给指定的远端主机

原型：

int send(int s,const void \*msg,int len,unsigned int flags);

返回值：成功则返回实际传送出去的字符数，可能会少于你所指定的发送长度。返回0，则对端关闭，出错返回-1, 设置了标准错误描述，可使用perror函数查看。

errno被设为以下的某个值：

EACCES，（对于由路径名标识的UNIX域套接字）目标套接字的文件的写入权限拒绝，或者路径前缀中的目录之一的搜索权限拒绝。（对于UDP套接字）尝试发送到网络/广播地址，就像它是单播地址一样。

EAGAIN或EWOULDBLOCK，套接字被标记为非阻塞，所请求的操作是阻塞的。 POSIX.1-2001允许为这种情况返回错误，并且不要求这些常量具有相同的值，因此应检查两种可能性。

EAGAIN（Internet域数据报套接字）参数sockfd引用的套接字未绑定地址，或者尝试将其绑定到临时端口，但临时端口范围内的所有端口号正在使用中

EBADF，sockfd指定了无效的描述符。

ECONNRESET，连接由对端重置。

EDESTADDRREQ，套接字是非连接模式，但没有设置对端地址。

EFAULT，为参数指定了无效的用户空间地址。

EINTR，在发送任何数据之前发生信号，接收操作被信号中断;。

EINVAL，传递无效的参数。

EISCONN，面向连接模式的套接字已经连接了，又指定了收件人

EMSGSIZE，套接字类型要求消息以原子方式发送，并但要发送的消息的大小是不可能的。

ENOBUFS，网络接口的输出队列已满。这通常表示接口已经停止发送，但可能是由于瞬时拥塞造成的。（通常，这不会发生在Linux中。当设备队列溢出时，数据包将被静默地丢弃。）

ENOMEM，无可用内存。

ENOTCONN，套接字是非连接且没有给定目标。

ENOTSOCK，文件描述符sockfd是一个套接字。

EOPNOTSUPP ，flags参数中的某些位不适用于某些类型的套接字。

EPIPE，面向连接的套接字在本地已经已关闭。在这种情况下，除非设置了MSG\_NOSIGNAL，否则该过程也会收到SIGPIPE。

参数：s为前面 accept 的返回值.即new\_fd

参数msg一般为常量字符串

参数len表示长度

参数flags通常为 0

常用实例：

**if(**send**(**new\_fd**,**"hello"**,** 6**,**0**)==-**1**)**

**{**

perror**(**"send"**);**

close**(**new\_fd**);**

close**(**sfd**);**

exit**(-**1**);**

**}**

（8）close 函数：当使用完文件后若已不再需要则可使用close()关闭该文件，并且close()会让数据写回磁盘， 并释放该文件所占用的资源

原型：

int close(int fd);

int shutdown(int fd, int how)

返回值：若文件顺利关闭则返回0，发生错误时返回-1。主动调用close函数的一端将进入timewait时间段。, 设置了标准错误描述，可使用perror函数查看。

参数：fd为前面的 sfd,new\_fd.

参数how 为关闭方式，SHUT\_RD关闭读端，SHUT\_WR关闭写端， SHUT\_RDWR关闭读端和写端。

常用实例：

close(new\_fd);

close(sfd);

#### 3.1.2 客户端

（1）用socket 函数：生成一个套接口描述符。

（2）connect 函数：用来请求连接远程服务器，将参数 sockfd 的 socket 连至参数 serv\_addr 指定的服务器 IP 和端口号上去。

原型：

int connect (int sockfd,struct sockaddr\*serv\_addr,int addrlen);

返回值：成功则返回0，出错返回-1。, 设置了标准错误描述，可使用perror函数查看。

参数：sockfd为前面 socket 的返回值，即 sfd

参数：serv\_addr为结构体指针变量，存储着远程服务器的 IP 与端口号信息。

参数addrlen表示结构体变量的长度

常用实例：

将上面的头文件以及各个函数中的代码全部拷贝就可以形成一个完整的例子，此处省略。

Example：将一些通用的代码全部封装起来，以后要用直接调用函数即可。如下，通用网络封装代码头文件：tcp\_net\_socket.h

#ifndef \_\_TCP\_\_NET\_\_SOCKET\_\_H

#define \_\_TCP\_\_NET\_\_SOCKET\_\_H

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<arpa/inet.h>

#include<unistd.h>

#include<signal.h>

extern int tcp\_init**(**const char**\***ip**,**int port**);**

extern int tcp\_accept**(**int sfd**);**

extern int tcp\_connect**(**constchar**\***ip**,**int port**);**

extern void signalhandler**(**void**);**

#endif

具体的通用函数封装如下： tcp\_net\_socket**.**c

#include"tcp\_net\_socket.h"

int tcp\_init**(**const char**\***ip**,** int port**)** //用于初始化操作

**{**

int sfd**=**socket**(**AF\_INET**,**SOCK\_STREAM**,**0**);**//首先创建一个 socket，向系统申请

**if(**sfd**==-**1**)**

**{**

perror**(**"socket"**);**

exit**(-**1**);**

**}**

struct sockaddr\_in serveraddr**;**

memset**(&**serveraddr**,**0**,sizeof(**struct sockaddr**));**

serveraddr**.**sin\_family **=**AF\_INET**;**

serveraddr**.**sin\_port**=**htons**(**port**);**

serveraddr**.**sin\_addr**.**s\_addr**=**inet\_addr**(**ip**);**//或 INADDR\_ANY ，自动获取

**if(**bind**(**sfd**,** **(**struct sockaddr**\*)&**serveraddr**,sizeof(**struct sockaddr**))==-**1**)**

//将新的 socket 与制定的 ip、port 绑定

**{**

perror**(**"bind"**);**

close**(**sfd**);**

exit**(-**1**);**

**}**

**if(**listen**(**sfd**,** 10**)==-**1**)**//监听它，并设置其允许最大的连接数为10 个

**{**

perror**(**"listen"**);**

close**(**sfd**);**

exit**(-**1**);**

**}**

**return** sfd**;**

**}**

int tcp\_accept**(**int sfd**)** //用于服务端的接收

**{**

struct sockaddr\_in clientaddr**;**

memset**(&**clientaddr**,**0**,sizeof(**struct sockaddr**));**

int addrlen**=sizeof(**struct sockaddr**);**

int new\_fd**=**accept**(**sfd**,(**struct sockaddr**\*)&**clientaddr**,&**addrlen**);**

//sfd 接受客户端连接，并创建新的 socket 为 new\_fd，

//将请求连接的客户端的 ip、port 保存在结构体 clientaddr 中

**if(**new\_fd**==-**1**)**

**{**

perror**(**"accept"**);**

close**(**sfd**);**

exit**(-**1**);**

**}**

printf**(**"%s %d success connect\n"**,**inet\_ntoa**(**clientaddr**.**sin\_addr**),**ntohs**(**clientaddr**.**sin\_port**));**

**return** new\_fd**;**

**}**

int tcp\_connect**(**const char**\***ip**,**int port**)** //用于客户端的连接

**{**

int sfd**=**socket**(**AF\_INET**,**SOCK\_STREAM**,**0**);**//向系统注册申请新的

socket **if(**sfd**==-**1**)**

**{**

perror**(**"socket"**);**

exit**(-**1**);**

**}**

struct sockaddr\_in serveraddr**;**

memset**(&**serveraddr**,**0**,sizeof(**struct sockaddr**));**

serveraddr**.**sin\_family **=**AF\_INET**;**

serveraddr**.**sin\_port**=**htons**(**port**);**

serveraddr**.**sin\_addr**.**s\_addr**=**inet\_addr**(**ip**);**

**if(**connect**(**sfd**,(**struct sockaddr**\*)&**serveraddr**,sizeof(**struct sockaddr**))==-**1**)**

//将 sfd 连接至制定的服务器网络地址serveraddr

**{**

perror**(**"connect"**);**

close**(**sfd**);**

exit**(-**1**);**

**}**

**return** sfd**;**

**}**

void signalhandler**(**void**)** //用于信号处理，让服务端在按下Ctrl+c或Ctrl+\的时候不会退出

**{**

sigset\_t sigSet**;**

sigemptyset**(&**sigSet**);**

sigaddset**(&**sigSet**,**SIGINT**);**

sigaddset**(&**sigSet**,**SIGQUIT**);**

sigprocmask**(**SIG\_BLOCK**,&**sigSet**,NULL);**

**}**

服务器端：tcp\_net\_server**.**c

#include"tcp\_net\_socket.h"

int main**(**intargc**,** char**\***argv**[])**

**{**

**if(**argc**<**3**)**

**{**

printf**(**"usage:./server tcp ip port\n"**);**

exit**(-**1**);**

**}**

signalhandler**();**

int sfd**=**tcp\_init**(**argv**[**1**],**atoi**(**argv**[**2**]));**

//或int sfd=tcp\_init("192.168.0.164",8888);

**while(**1**)** //用 while 循环表示可以与多个客户端接收和发送，但仍是阻塞模式的

**{**

int cfd **=**tcp\_accept**(**sfd**);**

char buf**[**512**]={**0**};**

**if(**recv**(**cfd**,**buf**,sizeof(**buf**),**0**)==-**1**)**//从 cfd 客户端接收数据存于buf 中

**{**

perror**(**"recv"**);**

close**(**cfd**);**

close**(**sfd**);**

exit**(-**1**);**

**}**

puts**(**buf**);**

**if(**send**(**cfd**,** "hello world"**,**12**,**0**)==-**1**)**//从buf 中取向 cfd 客户端发送数据

**{**

perror**(**"send"**);**

close**(**cfd**);**

close**(**sfd**);**

exit**(-**1**);**

**}**

close**(**cfd**);**

**}**

close**(**sfd**);**

**}**

客户端： tcp\_net\_client**.**c

#include"tcp\_net\_socket.h"

int main**(**intargc**,** char**\***argv**[])**

**{**

**if(**argc**<**3**)**

**{**

printf**(**"usage:./clienttcp ip port\n"**);**

exit**(-**1**);**

**}**

int sfd**=**tcp\_connect**(**argv**[**1**],**atoi**(**argv**[**2**]));**

char buf**[**512**]={**0**};**

send**(**sfd**,** "hello"**,**6**,**0**);** //向 sfd 服务端发送数据

recv**(**sfd**,**buf**,sizeof(**buf**),**0**);**//从sfd 服务端接收数据

puts**(**buf**);**

close**(**sfd**);**

**}**

#gcc–o tcp\_net\_server tcp\_net\_server.c tcp\_net\_socket.c

#gcc–o tcp\_net\_client tcp\_net\_client.c tcp\_net\_socket.c

#./tcp\_net\_server 192.168.0.1648888

#./tcp\_net\_client 192.168.0.1648888

/\* 备注 可以通过

gcc –fpic–c tcp\_net\_socket.c –o tcp\_net\_socket.o

gcc –shared tcp\_net\_socket.o–o libtcp\_net\_socket.so

cp lib\*.so/lib //这样以后就可以直接使用该库了

cpt cp\_net\_socket.h /usr/include/ //这样头文件包含可以用include <tcp\_net\_socket.h>了

以后再用到的时候就可以直接用：

gcc–o main main.c–ltcp\_net\_socket //其中 main.c 要包含头文件:include<tcp\_net\_socket.h>

./main

\*/

注：上面的虽然可以实现多个客户端访问，但是仍然是阻塞模式（即一个客户访问的时候会阻塞不让另外的客户访问）。解决办法有：

（1） 多进程（因为开销比较大，所以不常用）

int main**(**intargc**,** char**\***argv**[])**

**{**

**if(**argc**<**3**)**

**{**

printf**(**"usage:./servertcp ip port\n"**);**

exit**(-**1**);**

**}**

int sfd**=**tcp\_init**(**argv**[**1**],**atoi**(**argv**[**2**]));**

charbuf**[**512**]={**0**};**

**while(**1**)**

**{**

int cfd **=**tcp\_accept**(**sfd**);**

**if(**fork**()==**0**)**

**{**

recv**(**cfd**,**buf**,sizeof(**buf**),**0**);**

puts**(**buf**);**

send**(**cfd**,**"hello"**,**6**,**0**);**

close**(**cfd**);**

**}**

**else**

**{**

close**(**cfd**);**

**}**

**}**

close**(**sfd**);**

**}**

（2）多线程 将服务器上文件的内容全部发给客户端

/\*TCP 文件服务器演示代码 \*/

#include<stdio.h>

#include<stdlib.h>

#include<errno.h>

#include<string.h>

#include<sys/types.h>

#include<sys/fcntl.h>

#include<netinet/in.h>

#include<sys/socket.h>

#include<sys/wait.h>

#include<pthread.h>

#define DEFAULT\_SVR\_PORT 2828

#define FILE\_MAX\_LEN 64

char filename**[**FILE\_MAX\_LEN**+**1**];**

static void **\***handle\_client**(**void**\***arg**)**

**{**

int sock**=(**int**)**arg**;**

char buff**[**1024**];**

int len **;**

printf**(**"begin send\n"**);**

FILE **\***file **=**fopen**(**filename**,**"r"**);**

**if(**file **==NULL)**

**{**

close**(**sock**);**

exit**;**

**}** //发文件名

**if(**send**(**sock**,**filename**,**FILE\_MAX\_LEN**,**0**)==-**1**)**

**{**

perror**(**"sendfile name\n"**);**

**goto** EXIT\_THREAD**;**

**}**

printf**(**"begin send file%s....\n"**,**filename**);** //发文件内容

**while(!**feof**(**file**))**

**{**

len**=**fread**(**buff**,**size**(**char**),sizeof(**buff**),**file**);**

printf**(**"server read%s,len%d\n"**,**filename**,**len**);**

**if(**send**(**sock**,**buff**,**len**,**0**)<**0**)**

**{**

perror**(**"send file:"**);**

**goto** EXIT\_THREAD**;**

**}**

**}**

EXIT\_THREAD**:**

**if(**file**)**

fclose**(**file**);**

close**(**sock**);**

**}**

int main**(**int argc**,**char**\***argv**[])**

**{**

int sockfd**,**new\_fd**;** //第 1.定义两个pv4 地址

struct sockaddr\_in my\_addr**;**

struct sockaddr\_in their\_addr**;**

int sin\_size**,**numbytes**;**

pthread\_t cli\_thread**;**

unsigned short port**;**

**if(**argc**<**2**)**

**{**

printf**(**"need a filename without path\n"**);**

exit**;**

**}**

strncpy**(**filename**,**argv**[**1**],**FILE\_MAX\_LEN**);**

port**=**DEFAULT\_SVR\_PORT**;**

**if(**argc**>=**3**)**

**{**

port**=(**unsigned short**)**atoi**(**argv**[**2**]);** **}**

//第一步:建立TCP 套接字 Socket

//AF\_INET-->ip通讯

//SOCK\_STREAM-->TCP

**if((**sockfd **=**socket**(**AF\_INET**,**SOCK\_STREAM**,**0**))==-**1**)**

**{**

perror**(**"socket"**);**

exit**(-**1**);**

**}**

//第二步:设置侦听端口

//初始化结构体，并绑定 2828端口

memset**(&**my\_addr**,**0**,sizeof(**struct sockaddr**));**

//memset(&my\_addr,0,sizeof(my\_addr));

my\_addr**.**sin\_family **=**AF\_INET**;** /\*ipv4 \*/

my\_addr**.**sin\_port**=**htons**(**port**);** /\* 设置侦听端口是 2828, 用 htons 转成网络序\*/

my\_addr**.**sin\_addr**.**s\_addr**=**INADDR\_ANY**;**

/\*INADDR\_ANY 来表示任意IP地址可能其通讯 \*/

//bzero(&(my\_addr.sin\_zero),8);

//第三步:绑定套接口,把socket 队列与端口关联起来.

**if(**bind**(**sockfd**,(**struct sockaddr**\*)&**my\_addr**,sizeof(**struct sockaddr**))==-**1**)**

**{**

perror**(**"bind"**);**

**goto** EXIT\_MAIN**;**

**}**

//第四步:开始在 2828 端口侦听,是否有客户端发来联接

**if(**listen**(**sockfd**,**10**)==-**1**)**

**{**

perror**(**"listen"**);**

**goto** EXIT\_MAIN**;**

**}**

printf**(**"#@listen port%d\n"**,**port**);**

//第五步:循环与客户端通讯

**while(**1**)**

**{**

sin\_size **=sizeof(**struct sockaddr\_in**);**

printf**(**"server waiting...\n"**);**

//如果有客户端建立连接，将产生一个全新的套接字 new\_fd,专门用于跟这个客户端通信

**if((**new\_fd **=**accept**(**sockfd**,(**struct sockaddr**\*)&**their\_addr**,&**sin\_size**))==-**1**)**

**{**

perror**(**"accept:"**);**

**goto** EXIT\_MAIN**;**

**}**

printf**(**"---client(ip=%s:port=%d)request\n"**,**inet\_ntoa**(**their\_addr**.**sin\_addr**),**ntohs**(**their\_addr**.**sin\_port**));**

//生成一个线程来完成和客户端的会话，父进程继续监听

pthread\_create**(&**cli\_thread**,NULL,**handle\_client**,(**void**\*)**new\_fd**);**

**}**

//第六步:关闭 socket

EXIT\_MAIN**:**

close**(**sockfd**);**

**return** 0**;**

**}**

/\*TCP 文件接收客户端 \*/

#include<stdio.h>

#include<stdlib.h>

#include<errno.h>

#include<string.h>

#include<sys/types.h>

#include<netinet/in.h>

#include<sys/socket.h>

#include<sys/wait.h>

#define FILE\_MAX\_LEN 64

#define DEFAULT\_SVR\_PORT 2828

int main**(**int argc**,**char**\***argv**[])**

**{**

int sockfd**,**numbytes**;**

char buf**[**1024**],**

filename**[**FILE\_MAX\_LEN**+**1**];**

char ip\_addr**[**64**];**

struct hostent **\***he**;**

struct sockaddr\_intheir\_addr**;**

int i**=**0**,**len**,**total**;**

unsigned short port**;**

FILE**\***file**=NULL;**

**if(**argc**<**2**)**

**{**

printf**(**"need a server ip\n"**);**

exit**;**

**}**

strncpy**(**ip\_addr**,**argv**[**1**],sizeof(**ip\_addr**));**

port**=**DEFAULT\_SVR\_PORT**;**

**if(**argc**>=**3**)**

**{**

port**=(**unsigned short**)**atoi**(**argv**[**2**]);**

**}**

//做域名解析(DNS)

//he=gethostbyname(argv[1]);

//第一步:建立一个 TCP 套接字

**if((**sockfd **=**socket**(**AF\_INET**,**SOCK\_STREAM**,**0**))==-**1**)**

**{**

perror**(**"socket"**);**

exit**(**1**);**

**}**

//第二步:设置服务器地址和端口2828

memset**(&**their\_addr**,**0**,sizeof(**their\_addr**));**

their\_addr**.**sin\_family **=**AF\_INET**;**

their\_addr**.**sin\_port**=**htons**(**port**);**

their\_addr**.**sin\_addr**.**s\_addr**=**inet\_addr**(**ip\_addr**);**

//their\_addr.sin\_addr=\*((struct in\_addr\*)he->h\_addr);

//bzero(&(their\_addr.sin\_zero),8);

printf**(**"connect server%s:%d\n"**,**ip\_addr**,**port**);**

/\*第三步:用 connect 和服务器建立连接 ,注意,这里没有使用本地端口,将由协议栈自动分配一个端口\*/

**if(**connect**(**sockfd**,(**struct sockaddr**\*)&**their\_addr**,sizeof(**struct sockaddr**))==-**1**)**

**{**

perror**(**"connect"**);**

exit**(**1**);**

**}**

**if(**send**(**sockfd**,**"hello"**,**6**,**0**)<**0**)**

**{**

perror**(**"send"**);**

exit**(**1**);**

**}**

/\* 接收文件名,为编程简单,假设前 64字节固定是文件名,不足用 0 来增充 \*/

total**=**0**;**

**while(**total**<**FILE\_MAX\_LEN**)**

**{**

/\* 注意这里的接收 buffer 长度,始终是未接收文件名剩下的长度,\*/

len**=**recv**(**sockfd**,**filename**+**total**,(**FILE\_MAX\_LEN**-**total**),**0**);**

**if(**len**<=**0**)**

**break;**

total**+=**len**;**

**}**

/\* 接收文件名出错 \*/

**if(**total**!=**FILE\_MAX\_LEN**)**

**{**

perror**(**"failurefilename"**);**

exit**(-**3**);**

**}**

printf**(**"recvfile%s.....\n"**,**filename**);**

file**=**fopen**(**filename**,**"wb"**);**

//file=fopen("/home/hxy/abc.txt","wb");

**if(**file **==NULL)**

**{**

printf**(**"create file %sfailure"**,**filename**);**

perror**(**"create:"**);**

exit**(-**3**);**

**}**

//接收文件数据

printf**(**"recvbegin\n"**);**

total**=**0**;**

**while(**1**)**

**{**

len**=**recv**(**sockfd**,**buf**,sizeof(**buf**),**0**);**

**if(**len**==-**1**)**

**break;**

total**+=**len**;**

//写入本地文件

fwrite**(**buf**,**1**,**len**,**file**);**

**}**

fclose**(**file**);**

printf**(**"recvfile%ssuccess totallenght %d\n"**,**filename**,**total**);**

//第六步:关闭socket

close**(**sockfd**);**

**}**

/\* 备注

读写大容量的文件时，通过下面的方法效率很高

ssize\_treadn(int fd,char\*buf,int size)//读大量内容

{

char\*pbuf=buf;

int total,nread;

for(total=0;total<size;)

{

nread=read(fd,pbuf,size-total);

if(nread==0)

return total;

if(nread==-1)

{

if(errno ==EINTR)

continue;

else

return -1;

}

total+=nread;

pbuf+=nread;

}

return total;

}

ssize\_twriten(int fd,char\*buf,intsize)//写大量内容

{

char\*pbuf=buf;

int total,nwrite;

for(total=0;total<size;)

{

nwrite=write(fd,pbuf,size-total);

if(nwrite<=0)

{

if(nwrite==-1&&errno ==EINTR )

continue;

else

return -1;

}

total+=nwrite;

pbuf+=nwrite;

}

return total;

}

\*/

（3） 调用 fcntl 将sockfd 设置为非阻塞模式。

#include<unistd.h>

#include<fcntl.h>

……

sockfd **=**socket**(**AF\_INET**,**SOCK\_STREAM**,**0**);**

**if** lags**=**fcntl**(**sockfd**,**F\_GETFL**,**0**);**

fcntl**(**sockfd**,**F\_SETFL**,**O\_NONBLOCK**|**iflags**);**

……

（4）多路选择 select

#include<sys/select.h>

#include"tcp\_net\_socket.h"

#define MAXCLIENT 10

main**()**

**{**

int sfd**=**tcp\_init**(**"192.168.0.164"**,**8888**);**

int fd**=**0**;**

charbuf**[**512**]={**0**};**

fd\_setrdset**;**

**while(**1**)**

**{**

FD\_ZERO**(&**rdset**);**

FD\_SET**(**sfd**,&**rdset**);**

**if(**select**(**MAXCLIENT**+**1**,&**rdset**,NULL,NULL,** **NULL)<**0**)**

**continue;**

**for(**fd**=**0**;**fd **<**MAXCLIENT**;** fd**++)**

**{**

**if(**FD\_ISSET**(**fd**,&**rdset**))**

**{**

**if(**fd**==**sfd**)**

**{**

int cfd**=**tcp\_accept**(**sfd**);**

FD\_SET**(**cfd**,&**rdset**);** //……

**}**

**else**

**{**

bzero**(**buf**,** **sizeof(**buf**));**

recv**(**fd**,**buf**,sizeof(**buf**),**0**);**

puts**(**buf**);**

send**(**fd**,** "java"**,**5**,**0**);**

// FD\_CLR(fd, &rdset);

close**(**fd**);**

**}**

**}**

**}**

**}**

close**(**sfd**);**

**}**

具体例子请参考《网络编程之select**.**doc》或《tcp\_select》

### 3.2 使用 UDP 协议的流程图

UDP 通信流程图如下：

服务端：socket---bind---recvfrom---sendto---close

客户端：socket----------sendto---recvfrom---close



·sendto()函数原型：

int sendto(int sockfd,const void\*msg,int len,unsigned int flags,const struct sockaddr\*to, int tolen);

参数：sockfd为前面socket返回值

参数msg一般为常量字符串

参数len表示长度

参数flags通常为 0

参数to 表示目地机的 IP 地址和端口号信息。

参数tolen 常常被赋值为 sizeof(struct sockaddr)。

sendto 函数也返回实际发送的数据字节长度或在出现发送错误时返回-1, 0表示对端关闭，设置了标准错误描述，可使用perror函数查看。

errno被设为以下的某个值：

EACCES，（对于由路径名标识的UNIX域套接字）目标套接字的文件的写入权限拒绝，或者路径前缀中的目录之一的搜索权限拒绝。（对于UDP套接字）尝试发送到网络/广播地址，就像它是单播地址一样。

EAGAIN或EWOULDBLOCK，套接字被标记为非阻塞，所请求的操作是阻塞的。 POSIX.1-2001允许为这种情况返回错误，并且不要求这些常量具有相同的值，因此应检查两种可能性。

EAGAIN（Internet域数据报套接字）参数sockfd引用的套接字未绑定地址，或者尝试将其绑定到临时端口，但临时端口范围内的所有端口号正在使用中

EBADF，sockfd指定了无效的描述符。

ECONNRESET，连接由对端重置。

EDESTADDRREQ，套接字是非连接模式，但没有设置对端地址。

EFAULT，为参数指定了无效的用户空间地址。

EINTR，在发送任何数据之前发生信号，接收操作被信号中断;。

EINVAL，传递无效的参数。

EISCONN，面向连接模式的套接字已经连接了，又指定了收件人

EMSGSIZE，套接字类型要求消息以原子方式发送，并但要发送的消息的大小是不可能的。

ENOBUFS，网络接口的输出队列已满。这通常表示接口已经停止发送，但可能是由于瞬时拥塞造成的。（通常，这不会发生在Linux中。当设备队列溢出时，数据包将被静默地丢弃。）

ENOMEM，无可用内存。

ENOTCONN，套接字是非连接且没有给定目标。

ENOTSOCK，文件描述符sockfd是一个套接字。

EOPNOTSUPP ，flags参数中的某些位不适用于某些类型的套接字。

EPIPE，面向连接的套接字在本地已经已关闭。在这种情况下，除非设置了MSG\_NOSIGNAL，否则该过程也会收到SIGPIPE。

·recvfrom()函数原型：

int recvfrom(int sockfd,void\*buf,int len,unsigned int flags,struct sockaddr\*from,int \*fromlen);

参数from是一个struct sockaddr类型的变量，该变量保存连接机的IP地址及端口号。

参数fromlen常置为sizeof (struct sockaddr)。

当 recvfrom()返回时，fromlen 包含实际存入 from 中的数据字节数。

Recvfrom()函数返回接收到的字节数，0表示对端关闭，当出现错误时返回-1，0表示对端关闭并置相应的 errno, 设置了标准错误描述，可使用perror函数查看。

errno被设为以下的某个值

EAGAIN或EWOULDBLOCK，套接字被标记为非阻塞，而recv是阻塞，或接收超时。POSIX.1-2001允许为这种情况返回错误，并且不要求这些常量具有相同的值，因此应检查两种可能性。对非阻塞socket而言，EAGAIN不是一种错误

EBADF，参数sockfd是一个无效的描述符。

ECONNREFUSED，远程主机拒绝允许网络连接（通常是因为它没有运行请求的服务）。

EFAULT，接收缓冲区指针指向进程的地址空间之外。

EINTR，接收操作被信号中断。

EINVAL，传递无效的参数。

ENOMEM，无可用内存。

ENOTCONN，套接字是面向连接的但没有连接。

ENOTSOCK，文件描述符sockfd不引用套接字。

Example：UDP的基本操作

服务器端：

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

void main**()**

**{**

int sfd **=**socket**(**AF\_INET**,** SOCK\_DGRAM**,**0**);**

**if(**sfd **==-**1**)**

**{**

perror**(**"socket"**);**

exit**(-**1**);**

**}**

struct sockaddr\_in saddr**;**

bzero**(&**saddr**,** **sizeof(**saddr**));**

saddr**.**sin\_family **=**AF\_INET**;** s

addr**.**sin\_port**=**htons**(**8888**);**

saddr**.**sin\_addr**.**s\_addr**=**INADDR\_ANY**;**

**if(**bind**(**sfd**,(**struct sockaddr**\*)&**saddr**,sizeof(**struct sockaddr**))==-**1**)**

**{**

perror**(**"bind"**);**

close**(**sfd**);**

exit**(-**1**);**

**}**

char buf**[**512**]={**0**};**

**while(**1**)**

**{**

struct sockaddr\_in fromaddr**;**

bzero**(&**fromaddr**,sizeof(**fromaddr**));**

int fromaddrlen**=sizeof(**struct sockaddr**);**

**if(**recvfrom**(**sfd**,**buf**,sizeof(**buf**),**0**,(**struct sockaddr**\*)&**fromaddr**,&**fromaddrlen**)==-**1**)**

**{**

perror**(**"recvfrom"**);**

close**(**sfd**);**

exit**(-**1**);**

**}**

printf**(**"receive from %s %d,the message is:%s\n"**,** \

inet\_ntoa**(**fromaddr**.**sin\_addr**),** ntohs**(**fromaddr**.**sin\_port**),**buf**);**

sendto**(**sfd**,**"world"**,**6**,**0**,(**struct sockaddr**\*)&**fromaddr**,sizeof(**struct sockaddr**));**

**}**

close**(**sfd**);**

**}**

客户端：

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

int main**(**int argc**,**char**\***argv**[])**

**{**

int sfd **=**socket**(**AF\_INET**,** SOCK\_DGRAM**,**0**);**

**if(**sfd **==-**1**)**

**{**

perror**(**"socket"**);**

exit**(-**1**);**

**}**

struct sockaddr\_in toaddr**;**

bzero**(&**toaddr**,** **sizeof(**toaddr**));**

toaddr**.**sin\_family **=**AF\_INET**;**

toaddr**.**sin\_port**=**htons**(**atoi**(**argv**[**2**]));**//此处的端口号要跟服务器一样

toaddr**.**sin\_addr**.**s\_addr**=**inet\_addr**(**argv**[**1**]);**//此处为服务器的 ip

sendto**(**sfd**,**"hello"**,**6**,**0**,(**struct sockaddr**\*)&**toaddr**,sizeof(**struct sockaddr**));**

charbuf**[**512**]={**0**};**

structsockaddr\_infromaddr**;**

bzero**(&**fromaddr**,sizeof(**fromaddr**));**

intfromaddrlen**=sizeof(**structsockaddr**);**

**if(**recvfrom**(**sfd**,**buf**,sizeof(**buf**),**0**,(**struct sockaddr**\*)&**fromaddr**,&**fromaddrlen**)==-**1**)**

**{**

perror**(**"recvfrom"**);**

close**(**sfd**);**

exit**(-**1**);**

**}**

printf**(**"receive from %s %d,the message is:%s\n"**,** \

inet\_ntoa**(**fromaddr**.**sin\_addr**),** ntohs**(**fromaddr**.**sin\_port**),**buf**);**

close**(**sfd**);**

**}**

Example：UDP 发送文件先发文件大小再发文件内容

服务器端

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

main**()**

**{**

int sfd **=**socket**(**AF\_INET**,** SOCK\_DGRAM**,**0**);**

**if(**sfd **==-**1**)**

**{**

perror**(**"socket"**);**

exit**(-**1**);**

**}**

struct sockaddr\_in saddr**;**

bzero**(&**saddr**,** **sizeof(**saddr**));**

saddr**.**sin\_family **=**AF\_INET**;** saddr**.**sin\_port**=**htons**(**8888**);**

saddr**.**sin\_addr**.**s\_addr**=**INADDR\_ANY**;**

**if(**bind**(**sfd**,(**struct sockaddr**\*)&**saddr**,sizeof(**struct sockaddr**))==-**1**)**

**{**

1perror**(**"bind"**);**

close**(**sfd**);**

exit**(-**1**);**

**}**

char buf**[**512**]={**0**};**

struct sockaddr\_in fromaddr**;**

bzero**(&**fromaddr**,sizeof(**fromaddr**));**

int fromaddrlen**=sizeof(**structsockaddr**);**

**if(**recvfrom**(**sfd**,**buf**,sizeof(**buf**),**0**,(**struct sockaddr**\*)&**fromaddr**,&**fromaddrlen**)==-**1**)**

**{**

perror**(**"recvfrom"**);**

close**(**sfd**);** exit**(-**1**);**

**}**

printf**(**"receive from %s %d,the message is:%s\n"**,** \

inet\_ntoa**(**fromaddr**.**sin\_addr**),** ntohs**(**fromaddr**.**sin\_port**),**buf**);**

FILE**\***fp **=**fopen**(**"1.txt"**,**"rb"**);**

struct statst**;** //用于获取文件内容的大小

stat**(**"1.txt"**,&**st**);**

int filelen **=**st**.**st\_size**;**

sendto**(**sfd**,(**void**\*)&**filelen**,** **sizeof(**int**),**0**,(**struct sockaddr**\*)&**fromaddr**,sizeof(**struct sockaddr**));**

**while(!**feof**(**fp**))** //表示没有到文件尾

**{**

int len**=**fread**(**buf**,**1**,sizeof(**buf**),**fp**);**

sendto**(**sfd**,**buf**,**len**,**0**,(**struct sockaddr**\*)&**fromaddr**,sizeof(**struct sockaddr**));**

**}**

close**(**sfd**);**

**}**

客户端

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#define BUFSIZE 512

int main**(**int argc**,**char**\***argv**[])**

**{**

intsfd **=**socket**(**AF\_INET**,** SOCK\_DGRAM**,**0**);**

**if(**sfd **==-**1**)**

**{**

perror**(**"socket"**);**

exit**(-**1**);**

**}**

struct sockaddr\_in toaddr**;**

bzero**(&**toaddr**,** **sizeof(**toaddr**));**

toaddr**.**sin\_family **=**AF\_INET**;**

toaddr**.**sin\_port**=**htons**(**atoi**(**argv**[**2**]));**

toaddr**.**sin\_addr**.**s\_addr**=**inet\_addr**(**argv**[**1**]);**

sendto**(**sfd**,**"hello"**,**6**,**0**,(**struct sockaddr**\*)&**toaddr**,sizeof(**struct sockaddr**));**

charbuf**[**BUFSIZE**]={**0**};**

struct sockaddr\_in fromaddr**;**

bzero**(&**fromaddr**,sizeof(**fromaddr**));**

int fromaddrlen**=sizeof(**structsockaddr**);**

int filelen **=**0**;** //用于保存文件长度

FILE**\***fp **=**fopen**(**"2.txt"**,**"w+b"**);** //接收文件的长度

recvfrom**(**sfd**,** **(**void**\*)&**filelen**,sizeof(**int**),**0**,(**struct sockaddr**\*)&**fromaddr**,&**fromaddrlen**);**

printf**(**"the lengthof fileis%d\n"**,**filelen**);**

printf**(**"Create a new file!\n"**);**

printf**(**"begin to reveive file content!\n"**);** //接收文件的内容

**while(**1**)**

**{**

int len**=**recvfrom**(**sfd**,**buf**,sizeof(**buf**),**0**,(**structsockaddr**\*)&**fromaddr**,** **&**fromaddrlen**);**

**if(**len**<**BUFSIZE**)**

//如果接收的长度小于 BUFSIZE，则表示最后一次接收，此时要用 break退出循环

**{**

fwrite**(**buf**,sizeof(**char**),**len**,**fp**);**

**break;**

**}**

fwrite**(**buf**,sizeof(**char**),**len**,**fp**);**

**}**

printf**(**"receive filefinished!\n"**);**

close**(**sfd**);**

**}**

注意：操作系统的 UDP 接收流程如下：收到一个 UDP 包后，验证没有错误后，放入一个包队列中，队列中的每一个元素就是一个完整的 UDP 包。当应用程序通过 recvfrom()读取时， OS 把相应的一个完整 UDP 包取出，然后拷贝到用户提供的内存中，无论物理用户提供的内存大小是多少，OS 都会完整取出一个 UDP 包。如果用户提供的内存小于这个 UDP 包的大小，那么在填充慢内存后，UDP包剩余的部分就会被丢弃，以后再也无法取回。

这与 TCP 接收完全不同，TCP 没有完整包的概念，也没有边界，OS 只会取出用户要求的大小，剩余的仍然保留在 OS 中，下次还可以继续取出。

### 3.3 设置套接口的属性 setsockopt 的用法

函数原型：

#include<sys/types.h>

#include <sys/socket.h>

int getsockopt(int sockfd, int level, int optname, void \*optval, socklen\_t \*optlen);

int setsockopt(int sockfd, int level, int optname,const void\*optval,socklen\_t optlen);

获取或者设置套接字的属性,成功返回0，出错返回-1，设置了标准错误描述，可使用perror函数查看。

参数sockfd：标识一个套接口的描述字

参数level：指定选项代码的类型；支持 SOL\_SOCKET: 基本套接口、IPPROTO\_IP: IPv4套接口、IPPROTO\_IPV6: IPv6套接口、IPPROTO\_TCP: TCP套接口

参数optname：需设置的选项

SO\_DEBUG，bool，打开或关闭调试信息。当option\_value不等于0时，打开调试信息，否则，关闭调试信息。它实际所做的工作是在sock->sk->sk\_flag中置 SOCK\_DBG(第10)位，或清SOCK\_DBG位。

SO\_REUSEADDR，bool，打开或关闭地址复用功能。当option\_value不等于0时，打开，否则，关闭。它实际所做的工作是置sock->sk->sk\_reuse为1或0。

SO\_REUSEPORT，bool，打开或关闭端口复用功能。允许多个套接字 bind()/listen() 同一个TCP/UDP端口,每一个线程拥有自己的服务器套接字,在服务器套接字上没有了锁的竞争,内核层面实现负载均衡。安全层面，监听同一个端口的套接字只能位于同一个用户下面。

SO\_DONTROUTE，bool，打开或关闭路由查找功能。当option\_value不等于0时，打开，否则，关闭。它实际所做的工作是在sock->sk->sk\_flag中置或清 SOCK\_LOCALROUTE位。

SO\_BROADCAST，bool，允许或禁止发送广播数据。当option\_value不等于0时，允许，否则，禁止。它实际所做的工作是在sock->sk->sk\_flag中置或清 SOCK\_BROADCAST位。

SO\_SNDBUF，int，设置发送缓冲区的大小。发送缓冲区的大小是有上下限的，其上限为256 \* (sizeof(struct sk\_buff) + 256)，下限为2048字节。该操作将sock->sk->sk\_sndbuf设置为val \* 2，之所以要乘以2，是防止大数据量的发送，突然导致缓冲区溢出。最后，该操作完成后，因为对发送缓冲的大小 作了改变，要检查sleep队列，如果有进程正在等待写，将它们唤醒。

SO\_RCVBUF，int，设置接收缓冲区的大小。接收缓冲区大小的上下限分别是：256 \* (sizeof(struct sk\_buff) + 256)和256字节。该操作将sock->sk->sk\_rcvbuf设置为val \* 2。

SO\_KEEPALIVE，bool，套接字保活。如果协议是TCP，并且当前的套接字状态不是侦听(listen)或关闭(close)，那么，当option\_value不是零时，启用TCP保活定时 器，否则关闭保活定时器。对于所有协议，该操作都会根据option\_value置或清 sock->sk->sk\_flag中的 SOCK\_KEEPOPEN位。

SO\_OOBINLINE ，bool，当接收到OOB 数据时会马上送至标准输入设备。该操作根据option\_value的值置或清sock->sk->sk\_flag中的SOCK\_URGINLINE位。

SO\_NO\_CHECK，打开或关闭校验和。该操作根据option\_value的值，设置sock->sk->sk\_no\_check。

SO\_PRIORITY，设置在套接字发送的所有包的协议定义优先权。Linux通过这一值来排列网络队列。这个值在0到6之间（包括0和6），由option\_value指定。赋给sock->sk->sk\_priority。

SO\_LINGER，struct linger FAR\*，安全发送或接收。如果选择此选项, close或 shutdown将等到所有套接字里排队的消息成功发送或到达延迟时间后才会返回. 否则, 调用将立即返回。该选项的参数（option\_value)是一个linger结构：

struct linger

{

int l\_onoff;

int l\_linger;

};

如果linger.l\_onoff值为0(关闭），则清 sock->sk->sk\_flag中的SOCK\_LINGER位；否则，置该位，并赋sk->sk\_lingertime值为 linger.l\_linger。

SO\_DONTLINER， bool ，不要因为数据未发送就阻塞关闭操作。设置本选项相当于将SO\_LINGER的l\_onoff元素置为零

SO\_PASSCRED，允许或禁止SCM\_CREDENTIALS 控制消息的接收。该选项根据option\_value的值，清或置sock->sk->sk\_flag中的SOCK\_PASSCRED位。

SO\_TIMESTAMP，打开或关闭数据报中的时间戳接收。该选项根据option\_value的值，清或置sock->sk->sk\_flag中的SOCK\_RCVTSTAMP位，如果打开，则还需 设sock->sk->sk\_flag中的SOCK\_TIMESTAMP位，同时，将全局变量netstamp\_needed加1。

SO\_BINDTODEVICE，将套接字绑定到一个特定的设备上。该选项最终将设备赋给sock->sk->sk\_bound\_dev\_if。

SO\_ATTACH\_FILTER和SO\_DETACH\_FILTER。关于数据包过滤，它们最终会影响sk->sk\_filter。

TCP\_NODELAY， bool 禁止发送合并的Nagle算法

setsockopt ()不支持的BSD选项有：

SO\_ACCEPTCONN ，bool，套接口在监听。

SO\_ERROR， int， 获取错误状态并清除。

SO\_RCVLOWAT ，int，设置接收数据前的缓冲区内的最小字节数。在Linux中，缓冲区内的最小字节数是固定的，为1。即将sock->sk->sk\_rcvlowat固定赋值为1。

SO\_SNDLOWAT ，int，设置发送数据前的缓冲区内的最小字节数。在Linux中，缓冲区内的最小字节数是固定的，为1。即将sock->sk->sk\_sndlowat固定赋值为1。

SO\_RCVTIMEO，int,设置接收超时时间。该选项最终将接收超时时间赋给sock->sk->sk\_rcvtimeo。

SO\_SNDTIMEO，int,设置发送超时时间。该选项最终将发送超时时间赋给sock->sk->sk\_sndtimeo。

SO\_TYPE ，int，套接口类型。

IP\_OPTIONS ，在IP头中设置选项。

参数optval：指针，指向存放选项值的缓冲区，类型：整形，套接口结构， 其他结构类型:linger{}, timeval{ }

参数optlen：optval 的缓冲区长度

全部都必须要放在 bind 之前，通常是用于UDP 的。

（1）如果在已经处于 ESTABLISHED 状态下的 socket(一般由端口号和标志符区分）调用 close socket（一般不会立即关闭而经历 TIME\_WAIT 的过程）后想继续重用该 socket：

int reuse=1;

setsockopt(s,SOL\_SOCKET,SO\_REUSEADDR,(const char\*)&reuse,sizeof(int));

**作用**：

a.允许忽略套接字的timewait状态，立即重用该套接字bind的端口和ip

b.允许同一进程或不同进程bind相同端口不同IP地址

c.允许完全重复的bind，即端口和IP相同，此时需要传输协议的支持，一般只支持UDP。

所有的服务器端都应该启用此选项，以便重启时能够立刻使用这个端口号而不必等待timewait的时间。

**与SO\_REUSEPORT不同点在于**：

a. SO\_REUSEPORT是允许多个socket绑定到同一个ip+port上。SO\_REUSEADDR用于对TCP套接字处于TIME\_WAIT状态下的socket，才可以重复绑定使用。

b.两者使用场景完全不同。SO\_REUSEADDR这个套接字选项通知内核，如果端口忙，但TCP状态位于TIME\_WAIT，可以重用端口。这个一般用于当你的程序停止后想立即重启的时候，如果没有设定这个选项，会报错EADDRINUSE，需要等到TIME\_WAIT结束才能重新绑定到同一个ip+port上。而SO\_REUSEPORT用于多核环境下，允许多个线程或者进程绑定和监听同一个ip+port，而每个线程或进程有自己的套接字，内核只会激活一个accpt，无论UDP、TCP（以及TCP是什么状态），以达到负载均衡。

c.对于多播，两者意义相同。

**使用SO\_REUSEPORT的原因：**

单线程listen/accept，多个工作线程接收任务分发，虽CPU的工作负载不再是问题，但会存在：

单线程listener，在处理高速率海量连接时，一样会成为瓶颈

CPU缓存行丢失套接字结构(socket structure)现象严重

所有工作线程都accept()在同一个服务器套接字上呢，一样存在问题：

多线程访问server socket锁竞争严重

高负载下，线程之间处理不均衡，有时高达3:1不均衡比例

导致CPU缓存行跳跃(cache line bouncing)

在繁忙CPU上存在较大延迟

（2）如果要已经处于连接状态的 soket 在调用 close socket 后强制关闭，不经历 TIME\_WAIT 的过程：

int reuse=0;

setsockopt(s,SOL\_SOCKET,SO\_REUSEADDR,(const char\*)&reuse,sizeof(int));

（3）在 send(),recv()过程中有时由于网络状况等原因，发收不能预期进行,而设置收发时限：

int nNetTimeout=1000;//1 秒

// 发送时限

setsockopt(socket，SOL\_S0CKET,SO\_SNDTIMEO，(char\*)&nNetTimeout,sizeof(int));

// 接收时限

setsockopt(socket，SOL\_S0CKET,SO\_RCVTIMEO，(char\*)&nNetTimeout,sizeof(int));

（4）在 send()的时候，返回的是实际发送出去的字节(同步)或发送到 socket 缓冲区的字节(异步),系统默认的 状态发送和接收一次为 8688字节(约为 8.5K)；在实际的过程中发送数据和接收数据量比较大，可以设置 socket 缓冲区，而避免了 send(),recv()不断的循环收发：

// 接收缓冲区

int nRecvBuf=32\*1024; // 设置为 32K

setsockopt(s,SOL\_SOCKET,SO\_RCVBUF,(const char\*)&nRecvBuf,sizeof(int));

// 发送缓冲区

int nSendBuf=32\*1024; // 设置为 32K

setsockopt(s,SOL\_SOCKET,SO\_SNDBUF,(const char\*)&nSendBuf,sizeof(int));

（5）如果在发送数据时，希望不经历由系统缓冲区到 socket 缓冲区的拷贝而影响程序的性能：

int nZero=0;

setsockopt(socket，SOL\_SOCKET,SO\_SNDBUF，(char\*)&nZero,sizeof(int));

（6）如果在接收数据时，希望不经历由socket 缓冲区到系统缓冲区的拷贝而影响程序的性能：

int nZero=0;

setsockopt(socket，SOL\_SOCKET,SO\_RCVBUF，(char\*)&nZero,sizeof(int));

（7）一般在发送 UDP 数据报的时候，希望该 socket 发送的数据具有广播特性：

int bBroadcast=1;

setsockopt(s,SOL\_SOCKET,SO\_BROADCAST,(const char\*)&bBroadcast,sizeof(int));

### 3.4 单播、广播、组播（多播）

多播广播是用于建立分步式系统：例如网络游戏、ICQ 聊天构建、远程视频会议系统的重要工具。 使用多播广播的程序和 UDP 的单播程序相似。区别在于多播广播程序使用特殊的IP 地址。

对于单播而言，单播用于两个主机之间的端对端通信。

对于广播而言，广播用于一个主机对整个局域网上所有主机上的数据通信。广播只能用于客户机向 服务器广播，因为客户机要指明广播的IP 地址“192.168.0.255”和广播的端口号。服务器端 bind 的时候， 绑定的端口号要跟广播的端口号是同一个。这样才能收到广播消息。实例请参考《udp\_广播》。

对于多播而言，也称为“组播”，将网络中同一业务类型主机进行了逻辑上的分组，进行数据收发的时候其数据仅仅在同一分组中进行，其他的主机没有加入此分组不能收发对应的数据。

单播和广播是两个极端，要么对一个主机进行通信，要么对整个局域网上的主机进行通信。实际情况下，经常需要对一 组特定的主机进行通信，而不是整个局域网上的所有主机，这就是多播的用途。

例如，我们通常所说的讨论组。IPv4 多播地址采用 D 类 IP 地址确定多播的组。在 Internet 中，多播地址范围是从 224.0.0.0 到 234.255.255.255。

多播的程序设计也要使用 setsockopt()函数和getsockopt()函数来实现。其中对于setsockopt 的第二个 参数 level 不再是 SOL\_SOCKET，而是 IPPROTO\_IP；而且第三个参数 optname 常见的选项有：

Optname 含义

IP\_ADD\_MEMBERSHIP 在指定接口上加入组播组

IP\_DROP\_MEMBERSHIP 退出组播组

选项 IP\_ADD\_MEMBERSHIP 和 IP\_DROP\_MEMBERSHIP 加入或者退出一个组播组，通过选项IP\_ADD\_MEMBERSHIP 和 IP\_DROP\_MEMBERSHIP，对一个结构 struct ip\_mreq 类型的变量进行控制。 struct ip\_mreq 原型如下：

structip\_mreq

{

struct in\_addr imr\_multiaddr; /\*加入或者退出的多播组 IP 地址\*/

struct in\_addr imr\_interface; /\*加入或者退出的网络接口 IP 地址，本机 IP\*/

};

选项IP\_ADD\_MEMBERSHIP 用于加入某个多播组，之后就可以向这个多播组发送数据或者从多播组接收数据。此选项的值为mreq结构，成员imr\_multiaddr是需要加入的多播组IP地址，成员imr\_interface 是本机需要加入多播组的网络接口 IP 地址。例如：

struct ip\_mreq mreq**;**

memset**(&**mreq**,**0**,sizeof(**struct ip\_mreq**));**

mreq**.**imr\_interface**.**s\_addr**=**INADDR\_ANY**;**

mreq**.**imr\_multiaddr**.**s\_addr**=**inet\_addr**(**"224.1.1.1"**);**

**if(-**1**==**setsockopt**(**sfd**,**IPPROTO\_IP**,**IP\_ADD\_MEMBERSHIP**,&**mreq**,sizeof(**structip\_mreq**)))**

**{**

perror**(**"setsockopt"**);**

exit**(-**1**);**

**}**

接下来再绑定组播的port 号（如 65000），就可以接收组播消息了。实例请参考《udp\_多播》

选项IP\_ADD\_MEMBERSHIP 每次只能加入一个网络接口的 IP 地址到多播组，但并不是一个多播组仅允许一个主机 IP 地址加入，可以多次调用 IP\_ADD\_MEMBERSHIP 选项来实现多个IP 地址加入同一 个广播组，或者同一个 IP 地址加入多个广播组。

选项IP\_DROP\_MEMBERSHIP 用于从一个多播组中退出。例如：

if(-1==setsockopt(sfd,IPPROTP\_IP,IP\_DROP\_MEMBERSHIP,&mreq,sizeof(structip\_mreq)))

{

perror("setsockopt");

exit(-1);

}

# Linux下的描述符传递

## 1 概述

Linux 系统系下，创建子进程时，子进程会自动继承父进程已打开的描述符，但实际应用中，可能父进程需要向子进程传递“后打开的描述符”，或者子进程需要向父进程传递；或者两个进程可能是无关的，显然这需要一套传递机制。

简单的说，首先需要在这两个进程之间建立一个 Unix 域套接字接口作为消息传递的通道（ Linux 系统上使用 socketpair 函数可以很方面便的建立起传递通道），然后发送进程调用 send\_msg 向通道发送一个特殊的消息，内核将对这个消息做特殊处理，从而将打开的描述符传递到接收进程。然后接收方调用 recv\_msg 从通道接收消息，从而得到打开的描述符。然而实际操作起来并不像看起来那样单纯。

先来看几个注意点：

(1)需要注意的是传递描述符并不是传递一个 int型的描述符编号，而是在接收进程中创建一个新的描述符，并且在内核的文件表中，它与发送进程发送的描述符指向相同的项。

(2)在进程之间可以传递任意类型的描述符，比如可以是 pipe ， open ， mkfifo 或 socket ， accept等函数返回的描述符，而不限于套接字。

(3)一个描述符在传递过程中（从调用 sendmsg 发送到调用 recvmsg 接收），内核会将其标记为“在飞行中”（ in flight ）。在这段时间内，即使发送方试图关闭该描述符，内核仍会为接收进程保持打开状态。发送描述符会使其引用计数加 1 。

(4) 描述符是通过辅助数据发送的（结构体 msghdr 的 msg\_control 成员），在发送和接收描述符时，总是发送至少 1 个字节的数据，即使这个数据没有任何实际意义。否则当接收返回 0 时，接收方将不能区分这意味着“没有数据”（但辅助数据可能有套接字）还是“文件结束符”。

(5)具体实现时， msghdr 的 msg\_control 缓冲区必须与 cmghdr 结构对齐，可以看到后面代码的实现使用了一个 union 结构来保证这一点。

## 2 msghdr和cmsghdr结构体

上面说过，描述符是通过结构体 msghdr 的 msg\_control 成员送的，因此在继续向下进行之前，有必要了解一下 msghdr 和 cmsghdr 结构体，先来看看 msghdr 。

struct msghdr

{

void \*msg\_name;

socklen\_t msg\_namelen;

struct iovec \*msg\_iov;

size\_t msg\_iovcount;

void \*msg\_control;

socklen\_t msg\_controllen;

int msg\_flags;

};

结构成员可以分为下面的四组，这样看起来就清晰多了：

（1）套接口地址成员 msg\_name 与 msg\_namelen ；

只有当通道是UDP数据报套接口时才需要； msg\_name 指向要发送或是接收信息的套接口地址。 msg\_namelen 指明了这个套接口地址的长度。

msg\_name 在调用 recvmsg 时指向接收地址，在调用 sendmsg 时指向目的地址。注意， msg\_name 定义为一个 (void \*) 数据类型，因此并不需要将套接口地址显示转换为 (struct sockaddr \*) 。

（2）I/O 向量引用 msg\_iov 与 msg\_iovcount

它是实际的数据缓冲区，从下面的代码能看到，我们的1个字节就交给了它；这个msg\_iovlen 是 msg\_iov 的个数，不是什么长度。

msg\_iov 成员指向一个 struct iovec 数组， iovec 结构体在 sys/uio.h 头文件定义。

struct iovec

{

void \*iov\_base; /\*Starting address \*/

size\_t iov\_len; /\*Number of bytes to transfer \*/

};

iov\_base指向要发送的验证数据buf；

iov\_len要发送验证数据大小

有了 iovec ，就可以使用 readv 和 writev 函数在一次函数调用中读取或是写入多个缓冲区，显然比多次read ，write 更有效率。 readv 和 writev 的函数原型如下：

#include <sys/uio.h>

int readv(int fd, const struct iovec \*vector, int count);

int writev(int fd, const struct iovec \*vector, int count);

成功返回写入或读取的字节，出错返回-1，, 设置了标准错误描述，可使用perror函数查看。

（3）附属数据缓冲区成员 msg\_control 与 msg\_controllen ，描述符就是通过它发送的，后面将会看到， msg\_control 指向附属数据缓冲区，而 msg\_controllen 指明了附属数据缓冲区大小。

（4）接收信息标记位 msg\_flags ；忽略

轮到 cmsghdr 结构了，附属信息可以包括若干个单独的附属数据对象。在每一个对象之前都有一个 struct cmsghdr 结构。之后是填充字节，然后是对象本身。最后，附属数据对象之后，下一个 cmsghdr 之前也许要有更多的填充字节。

struct cmsghdr

{

socklen\_t cmsg\_len;

int cmsg\_level;

int cmsg\_type;

/\* u\_char cmsg\_data[]; \*/

};

cmsg\_len 附属数据的字节数，这包含结构头的尺寸，这个值是由 CMSG\_LEN() 宏计算的；

cmsg\_level 表明了原始的协议级别 ( 例如， SOL\_SOCKET) ；

cmsg\_type 表明了控制信息类型 ( 例如， SCM\_RIGHTS ，附属数据对象是文件描述符； SCM\_CREDENTIALS ，附属数据对象是一个包含证书信息的结构 ) ；

被注释的 cmsg\_data 用来指明实际的附属数据的位置，帮助理解。

对于 cmsg\_level 和 cmsg\_type ，当下我们只关心 SOL\_SOCKET 和 SCM\_RIGHTS 。

## 3 msghdr和cmsghdr辅助宏

这些结构还是挺复杂的， Linux 系统提供了一系列的宏来简化我们的工作，这些宏可以在不同的 UNIX 平台之间进行移植。这些宏是由 cmsg(3) 的 man 手册页描述的，先来认识一下：

#include <sys/socket.h>

struct cmsghdr \*CMSG\_FIRSTHDR(struct msghdr \*msgh);

struct cmsghdr \*CMSG\_NXTHDR(struct msghdr \*msgh, struct cmsghdr \*cmsg);

size\_t CMSG\_ALIGN(size\_t length);

size\_t CMSG\_SPACE(size\_t length);

size\_t CMSG\_LEN(size\_t length);

void \*CMSG\_DATA(struct cmsghdr \*cmsg);

CMSG\_LEN() 宏

输入参数：附属数据缓冲区中的对象大小；如果是传递描述符，则为sizeof（int）。

功能：计算 cmsghdr 头结构加上附属数据对象大小，包括必要的对齐字段，这个值用来设置 cmsghdr 对象的 cmsg\_len 成员。

CMSG\_SPACE() 宏

输入参数：附属数据缓冲区中的对象大小；

功能：计算 cmsghdr 头结构加上附属数据大小，并包括对齐字段和可能的结尾填充字符，注意 CMSG\_LEN() 值并不包括可能的结尾填充字符。 CMSG\_SPACE() 宏对于确定所需的缓冲区尺寸是十分有用的。

注意：如果在缓冲区中有多个附属数据，一定要同时添加多个 CMSG\_SPACE() 宏调用来得到所需的总空间。

下面的例子反映了二者的区别：

printf("CMSG\_SPACE(sizeof(short))=%d/n", CMSG\_SPACE(sizeof(short))); // 返回16

printf("CMSG\_LEN(sizeof(short))=%d/n", CMSG\_LEN(sizeof(short))); // 返回14

CMSG\_DATA() 宏

输入参数：指向 cmsghdr 结构的指针 ;

功能：返回附属数据缓冲区每个对象中跟随在头部以及填充字节之后的附属数据的第一个字节 ( 如果存在 ) 的地址，等比如传递描述符时，代码接收将是如下的形式：

struct cmsghdr \*pcmsg;

接收端：int fd = \*(int \*)CMSG\_DATA(pcmsg);

发送端：\*(int \*)CMSG\_DATA(pcmsg) = fd;

CMSG\_FIRSTHDR() 宏

输入参数：指向 struct msghdr 结构的指针；

功能：返回指向附属数据缓冲区内的第一个附属对象的 struct cmsghdr 指针。如果不存在附属数据对象则返回的指针值为 NULL 。

CMSG\_NXTHDR() 宏

输入参数：指向当前 struct cmsghdr 的指针；

功能：这个用于返回下一个附属数据对象的 struct cmsghdr 指针，如果没有下一个附属数据对象，这个宏就会返回 NULL 。

通过这两个宏可以很容易遍历所有的附属数据，像下面的形式：

struct msghdr msgh;

struct cmsghdr \*cmsg;

for (cmsg = CMSG\_FIRSTHDR(&msgh); cmsg != NULL; cmsg = CMSG\_NXTHDR(&msgh,cmsg)

{

// 得到了cmmsg，就能通过CMSG\_DATA()宏取得辅助数据了

｝

## 4 sendmsg和recvmsg

函数原型如下：

#include <sys/types.h>

#include <sys/socket.h>

int sendmsg(int s, const struct msghdr \*msg, unsigned int flags);

int recvmsg(int s, struct msghdr \*msg, unsigned int flags);

sendmsg 系统调用，从一个套接字发送消息，recvmsg用于从一个套接字接收消息，函数的返回值为实际发送 / 接收的字节数。0表示对端关闭，否则返回 -1 表明发生了错误, 设置了标准错误描述，可使用perror函数查看。

sendmsg的errno被设为以下的某个值：

EACCES，（对于由路径名标识的UNIX域套接字）目标套接字的文件的写入权限拒绝，或者路径前缀中的目录之一的搜索权限拒绝。（对于UDP套接字）尝试发送到网络/广播地址，就像它是单播地址一样。

EAGAIN或EWOULDBLOCK，套接字被标记为非阻塞，所请求的操作是阻塞的。 POSIX.1-2001允许为这种情况返回错误，并且不要求这些常量具有相同的值，因此应检查两种可能性。

EAGAIN（Internet域数据报套接字）参数sockfd引用的套接字未绑定地址，或者尝试将其绑定到临时端口，但临时端口范围内的所有端口号正在使用中

EBADF，sockfd指定了无效的描述符。

ECONNRESET，连接由对端重置。

EDESTADDRREQ，套接字是非连接模式，但没有设置对端地址。

EFAULT，为参数指定了无效的用户空间地址。

EINTR，在发送任何数据之前发生信号，接收操作被信号中断;。

EINVAL，传递无效的参数。

EISCONN，面向连接模式的套接字已经连接了，又指定了收件人

EMSGSIZE，套接字类型要求消息以原子方式发送，并但要发送的消息的大小是不可能的。

ENOBUFS，网络接口的输出队列已满。这通常表示接口已经停止发送，但可能是由于瞬时拥塞造成的。（通常，这不会发生在Linux中。当设备队列溢出时，数据包将被静默地丢弃。）

ENOMEM，无可用内存。

ENOTCONN，套接字是非连接且没有给定目标。

ENOTSOCK，文件描述符sockfd是一个套接字。

EOPNOTSUPP ，flags参数中的某些位不适用于某些类型的套接字。

EPIPE，面向连接的套接字在本地已经已关闭。在这种情况下，除非设置了MSG\_NOSIGNAL，否则该过程也会收到SIGPIPE。

recvmsg的errno被设为以下的某个值

EAGAIN或EWOULDBLOCK，套接字被标记为非阻塞，而recv是阻塞，或接收超时。POSIX.1-2001允许为这种情况返回错误，并且不要求这些常量具有相同的值，因此应检查两种可能性。对非阻塞socket而言，EAGAIN不是一种错误

EBADF，参数sockfd是一个无效的描述符。

ECONNREFUSED，远程主机拒绝允许网络连接（通常是因为它没有运行请求的服务）。

EFAULT，接收缓冲区指针指向进程的地址空间之外。

EINTR，接收操作被信号中断。

EINVAL，传递无效的参数。

ENOMEM，无可用内存。

ENOTCONN，套接字是面向连接的但没有连接。

ENOTSOCK，文件描述符sockfd不引用套接字。

参数s套接字通道，对于sendmsg 是发送套接字，对于 recvmsg 则对应于接收套接字；

参数msg 信息头msghdr结构指针；

参数flags 可选的标记位， 这与 send /recv或是 sendto/recvfrom 函数调用的标记相同

## 5 Socketpair 函数简介

socketpair创建了一对匿名的套接字描述符（只能在AF\_LOCAL域中使用），描述符存储于一个二元数组,如 s[2] .这对套接字可以进行双工通信，每一个描述符既可以读也可以写。这个在同一个进程中也可以进行通信，类似于管道，只能从一端写入，从另一端读取，如：向s[0]中写入，只能从s[1]中读取，也可以在s[1]中写入，然后从s[0]中读取；但是，若没有在0端写入，而从1端读取，则1端的读取操作会阻塞，即使在1端写入，也不能从1读取，仍然阻塞；反之亦然......

#include <sys/types.h>

#include <sys/socket.h>

int socketpair(int domain, int type, int protocol, int sv[2]);

socketpair在指定的网域中创建一对匿名套接字描述符，创建的套接字对存放在sv [0]和sv [1]中返回。 两个套接字没有区别,都指向同一个sockket文件。成功返回0出错返回-1, 设置了标准错误描述，可使用perror函数查看。

参数：domain这里 AF\_LOCAL

参数type｛tcp：SOCK\_STREAM udp：SOCK\_DGRAM}

参数protocol指定socket 所使用的传输协议编号。通常为0.

参数sv[2],用于存放建立的fd[2]

这对套接字可以进行双工通信，每一个描述符既可以读也可以写。这个在同一个进程中也可以进行通信，向s[0]中写入，就可以从s[1]中读取（只能从s[1]中读取），也可以在s[1]中写入，然后从s[0]中读取；但是，若没有在0端写入，而从1端读取，则1端的读取操作会阻塞，即使在1端写入，也不能从1读取，仍然阻塞；反之亦然......

## 6 writev和readv

#include <sys/uio.h>

ssize\_t writev(int fd,const struct iovec\*iov, int iovcnt)

ssize\_t readv(int fd, const struct iovec\*iov,int iovcnt);

函数writev一次写入多个buf 内容，函数readv一次读取多个buf 内容。成功返回读写字节数，出错返回-1， 设置了标准错误描述，可使用perror函数查看。

参数iovcnt为iovec指向的struct iovec结构体的个数

struct iovec

{

void \*iov\_base; /\*Starting address \*/

size\_t iov\_len; /\*Number of bytes to transfer \*/

};

## 7 进程发送文件描述符

第一步，初始化 socketpair 类型描述符

int fds[2];

socketpair(AF\_LOCAL,SOCK\_STREAM,0,fds);

第二步：sendmsg 发送描述符

（1）用sockfd 即sockpair 初始化的后的描述符 fds[1];

（2）定义结构体 struct msghdrmsg; Sendmsg ，并初始化 msghdr 结构体

struct msghdr

{

void \*msg\_name; /\*optional address \*/ 没用 ，报文包采用

socklen\_t msg\_namelen; /\*sizeofaddress\*/ 没用 ，报文包才用

struct iovec \*msg\_iov; /\*scatter/gatherarray\*/ 没用 ，但是必须写

size\_t msg\_iovcount; /\*#elementsinmsg\_iov \*/ 没用 ，但是必须填

void \*msg\_control; /\* ancillarydata,seebelow\*/ 关键，即下面的cmsghdr结构体地址

size\_t msg\_controllen; /\* ancillary data buffer len \*/ cmsghdr 结构 体的长度

int msg\_flags; /\*flags(unused)\*/ 没用

};

msg\_iov为iovec数组指针，必须赋值 ，msg\_iovcount为数组的元素个数；

struct iovec

{

void \*iov\_base; /\*Starting address \*/ 为msg\_iov所指的数组的个元素（也是数组）的地址

size\_t iov\_len; /\*Number of bytes to transfer \*/ ，为各元素的大小

};

Cmsg 构造结构体 cmsghdr mancmsg

struct cmsghdr

{

socklen\_t cmsg\_len; /\*databytecount,including header\*/

int cmsg\_level; /\*originatingprotocol \*/

int cmsg\_type; /\*protocol-specific type \*/

/\*followed byunsigned charcmsg\_data[];\*/

};

首先定义 struct cmsghdr\*cmsg 指针

cmsg\_len 为，cmsghdr 结构体的长度，通过 CMSG\_LEN 进行计算，我们传递的fd 的大 小为整型四个字节，所以Int len =CMSG\_LEN(sizeof(int));

然后为结构体申请空间： cmsg=(struct cmsghdr\*)calloc(1,len);

最后初始化cmsghar结构体，

Cmsg->cmsg\_len=len;

cmsg->cmsg\_level =SOL\_SOCKET;

Cmsg->cmsg\_type =SCM\_RIGHTS;

\*(int\*) CMSG\_DATA(cmsg)=fd；

第三步：

int sendmsg(int s, const struct msghdr \*msg, unsigned int flags);

## 8 fcntl函数简介

fcntl 函数可以改变已打开的文件描述符性质

函数原型：

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

int fcntl(int fd, int cmd);

int fcntl(int fd, int cmd, long arg);

int fcntl(int fd, int cmd, struct flock \*lock);

fcntl()针对(文件)描述符提供控制. 成功返回文件描述符或相应的flags，出错返回-1

参数fd 是被参数 cmd 操作(如下面的描述)的描述符. 针对cmd 的值,fcntl能够接受第三个参数 arg 或lock。

参数fd 代表欲设置的文件描述符。

参数cmd 代表打算操作的指令。有以下几种情况:

（1）拷贝文件描述符

F\_DUPFD 用来查找大于或等于参数 arg 的最小且仍未使用的文件描述词，并且复制参数 fd 的文件 描述词。执行成功则返回新复制的文件描述词。新描述符与fd 共享同一文件表项，但是新描述符有它自己的一套文件描述符标志，其中 FD\_CLOEXEC 文件描述符标志被清除。这不同于dup2函数，它使用指定的描述符。

F\_DUPFD\_CLOEXEC 和F\_DUPFD功能一样，但是会设置FD\_CLOEXEC标志位，以便在子进程中执行exec族函数时自动关闭文件描述符。

（2）获取或设置文件描述符标志位

F\_GETFD 取得文件描述符的标志位，参数arg被忽略。

F\_SETFD 设置文件描述符的标志位。该旗标以参数arg决定。

注意：在多线程程序中，使用fcntl（）的F\_SETFD设置close-on-exec标志同时另一个线程执行fork和xecve，容易泄漏文件描述符影响程序在子进程中执行。

（3）获取或这是文件状态标志位

F\_GETFL，获取文件访问模式和文件状态标志; arg被忽略。

F\_SETFL，将文件状态标志设置为由arg指定的值。 arg中的文件访问模式（O\_RDONLY，O\_WRONLY，O\_RDWR）和文件创建标志（即O\_CREAT，O\_EXCL，O\_NOCTTY，O\_TRUNC）被忽略。 在Linux上这个   命令只能更改O\_APPEND，O\_ASYNC，O\_DIRECT，O\_NOATIME和O\_NONBLOCK标志。 无法更改O\_DSYNC和O\_SYNC标志; 见下面的BUGS。

出错返回-1，设置了标准错误描述，可使用perror函数查看。成功依照cmd命令不同而有不同情况：

F\_DUPFD 返回新描述符

F\_GETFD 返回描述符标志位

F\_GETFL 返回描述符打开方式

F\_GETLEASE 返回文件描述符持有的租约类型

F\_GETOWN 返回描述符所有者的值

F\_GETSIG 返回可读可写时发出的信号值，对SIGIO 返回0.

F\_GETPIPE\_SZ, F\_SETPIPE\_SZ 返回管道容量

F\_GET\_SEALS A bit mask identifying the seals that have been set for the inode referred to by fd.

其余返回0

## 9 进程接收描述符

第一步：与发送描述第二步一样，唯一的区别在于

int recvmsg(int s, struct msghdr \*msg, unsigned int flags);

fd=\*(int\*) CMSG\_DATA(cmsg)

# IO复用模型

## 1.select模型

**I/O多路复用Select 函数：**

#include<sys/select.h>

#include<sys/time.h>

int pselect(int maxfds, fd\_set \*readset, fd\_set \*writeset, fd\_set \*exceptionset,const struct timespec \*restrict timeout, const sigset\_t \*restrict sigmask);

int select(int maxfd, fd\_set\*readset,fd\_set \*writeset, fd\_set\*exceptionset, const struct timeval\*timeout);

返回:已处于就绪状态的描述符的数目（或者叫做状态改变了的描述字的数目），0—超时或则没有描述符准备好，-1—出错 。设置了标准错误描述，可使用perror函数查看。

Select可以完成非阻塞（所谓非阻塞方式non-block，就是进程或线程执行此函数时不必非要等待事件的发生，一旦执行肯定返回，以返回值的不同来反映函数的执行情况，如果事件发生则与阻塞方式相同，若事件没有发生则返回一个代码来告知事件未发生，而进程或线程继续执行，所以效率较高）方式工作的程序，它能够监视我们需要监视的文件描述符的变化情况——读写或是异常。

参数解释：

maxfd：最大的文件描述符（其值应该为最大的文件描述符字 +1）

readset：内核读操作的描述符字集合

writeset：内核写操作的描述符字集合

exceptionset：内核异常操作的描述符字集合

timeout：等待描述符就绪需要多少时间。NULL 代表永远等下去，一个固定值代表等待固定时间，0代表根本不等待，检查描述字之后立即返回。

其中 readset、writeset、exceptionset 都是 fd\_set 集合。该集合的相关操作 如下：

void FD\_ZERO(fd\_set\*fdset); /\* 将所有 fd 清零 \*/

void FD\_SET(int fd, fd\_set\*fdset); /\* 增加一个 fd\*/

void FD\_CLR(int fd,fd\_set \*fdset); /\* 删除一个 fd\*/

int FD\_ISSET(int fd, fd\_set\*fdset); /\* 判断一个 fd 是否有设置 \*/

一般来说，在使用 select 函数之前，首先要使用FD\_ZERO和FD\_SET来初始化文件描述符集，在使 用select函数时，可循环使用 FD\_ISSET 测试描述符集，在执行完对相关文件描述符操作后之后，使用 FD\_CLR 来清除描述符集。/每次循环都要清空集合,并重新注册事件，否则不能检测描述符变化

另外，select 函数中的 timeout 是一个 struct timeval 类型的指针，该结构体如下：

struct timeval

{

long tv\_sec; /\* second \*/ //秒 l

long tv\_usec; /\* microsecond \*/ //微秒

};

struct timespec

{

\_\_time\_t tv\_sec; /\*seconds 秒\*/

long int tv\_nsec; /\*nanoseconds 纳秒\*/

}

pselect与select功能一致，区别在于

（1）select函数用的timeout参数，是一个timeval的结构体（包含秒和微秒），然而pselect用的是一个timespec结构体（包含秒和纳秒）

（2）select函数可能会为了指示还剩多长时间而更新timeout参数，然而pselect不会改变timeout参数

（3）select函数没有sigmask参数，当pselect的sigmask参数为null时，两者行为时一致的。有sigmask的时候，pselect相当于如下的select()函数，在进入select()函数之前手动将信号的掩码改变，并保存之前的掩码值；select()函数执行之后，再恢复为之前的信号掩码值。

下面的pselect调用

ready = pselect(nfds, &readfds, &writefds, &exceptfds, timeout, &sigmask);

相当于原子的执行以下调用

sigset\_t origmask;

sigprocmask(SIG\_SETMASK, &sigmask, &origmask);

select(nfds, &readfds, &writefds, &exceptfds, timeout);

sigprocmask(SIG\_SETMASK, &origmask, NULL);

举例:多路转接模型 select

#include <sys/select.h>

#include <sys/time.h>

#include <sys/types.h>

#include <unistd.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#define FILENAME1 "a.txt"

#define FILENAME2 "dir.c"

int main**()**

**{**

char buf**[**10**]** **=** **{**0**};**

int fd1 **=** open**(**FILENAME1**,**O\_RDWR**);**

int fd2 **=** open**(**FILENAME2**,**O\_RDWR**);**

int fd3 **=** open**(**FILENAME1**,**O\_RDWR**);**

int fd4 **=** open**(**FILENAME2**,**O\_RDWR**);**

**if(** **(-**1 **==** fd1**)** **||** **(-**1 **==** fd2**)** **||** **(-**1 **==** fd3**)** **||** **(-**1 **==** fd4**)** **)**

**{**

perror**(**"open"**);**

exit**(-**1**);**

**}**

fd\_set fdrd**,**fdwr**;** //绑定读写集合

FD\_ZERO**(&**fdrd**);** //清除以前读的绑定

FD\_ZERO**(&**fdwr**);** //清除以前写的绑定

FD\_SET**(**fd1**,&**fdrd**);** //将 fd1 与读绑定

FD\_SET**(**fd2**,&**fdrd**);**

FD\_SET**(**fd3**,&**fdwr**);** //将 fd3 与写绑定

FD\_SET**(**fd4**,&**fdwr**);**

int max1 **=** fd1 **>** fd2 **?** fd1 **:** fd2**;** //获取读绑定中的文件描述词最大值

int max2 **=** fd3 **>** fd4 **?** fd3 **:** fd4**;** //获取写绑定中的文件描述词最大值

int max **=** max1 **>** max2 **?** max1 **:** max2**;** //获得读写文件描述词最大值

struct timeval tv**;** //用于记录时间，表示过这么长时间不响应就退出

tv**.**tv\_sec **=** 2**;** //秒

tv**.**tv\_usec **=** 0**;** //微妙

**while(**1**)**

**{**

**if(** select**(**max**+**1**,** **&**fdrd**,** **&**fdwr**,** **NULL,** **&**tv**)** **==** **-**1 **)** //从 1—max+1 查找

**{**

perror**(**"select"**);**

**break;**

**}**

**if(** FD\_ISSET**(**fd1**,&**fdrd**)** **)** //如果 fd1 设置的是读绑定

**{**

read**(**fd1**,**buf**,sizeof(**buf**)-**1**);**

puts**(**buf**);**

sleep**(**1**);**

**}**

**if(** FD\_ISSET**(**fd2**,&**fdrd**)** **)**

**{**

read**(**fd2**,**buf**,sizeof(**buf**)-**1**);**

puts**(**buf**);**

sleep**(**1**);**

**}**

**if(** FD\_ISSET**(**fd3**,&**fdwr**)** **)** //如果 fd3 设置的是写绑定

**{**

write**(**fd3**,**buf**,sizeof(**buf**));**

sleep**(**2**);**

**}**

**if(** FD\_ISSET**(**fd4**,&**fdwr**)** **)**

**{**

write**(**fd4**,**buf**,sizeof(**buf**));**

sleep**(**2**);**

**}**

**}**

close**(**fd1**);**

close**(**fd2**);**

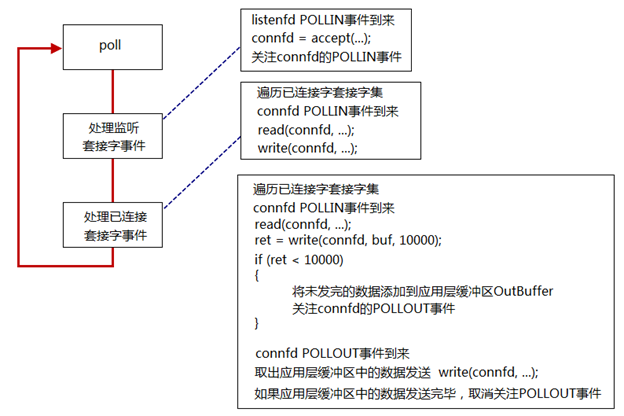
close**(**fd3**);**

close**(**fd4**);**

**}**

样例存在问题

## 2.poll模型



函数原型：

#include <poll.h>

int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);

int ppoll(struct pollfd \*fds, nfds\_t nfds, const struct timespec \*tmo\_p, const sigset\_t \*sigmask);

函数poll（）类似select：它等待一组文件描述符中的一个准备就绪然后执行I / O。

参数 fds为要监视的文件描述符及其事件，是输入也是输出参数。可以是一个元素或者数组。

struct pollfd

{

int fd; /文件描述符 \*/

short events; /\* 输入参数 \*/

short revents; /\* 输出参数\*/

};

成员fd包含打开文件的文件描述符。 如果该字段为负，则忽略相应的事件字段，并且revents字段返回零。 （这提供了在poll（）调用中忽略单个文件描述符的方法：置fd为负数，但请注意，这不能用于忽略文件描述符0.）

成员events是一个输入参数，一个位掩码，指定要对文件描述符fd监听的事件。可以被指定为零，在这种情况下，在revent中可以返回的唯一事件是POLLHUP，POLLERR和POLLNVAL。

成员revents是一个输出参数，由内核填充实际发生的事件。 在revent中返回的位可以包括在事件中指定的任何位，或POLLERR，POLLHUP或POLLNVAL之一。 （这三个位在事件字段中是无意义的，只要相应的条件为真，就会在revents字段中设置。）

events或revents的值可以是下列的组合：

POLLIN ，有数据可读

POLLPRI，有要读取的紧急数据（例如，TCP套接字上的带外数据;分组模式中的伪终端主机已经看到从机的状态改变）。

POLLOUT，可写，即使写入数据是大于套接字或管道中的可用空间，仍将阻塞（除非设置了O\_NONBLOCK）。

POLLRDHUP，流套接字的对端关闭连接，关闭连接的写端。必须定义\_GNU\_SOURCE功能测试宏（包括任何头文件之前）才能获得此定义。

POLLERR，发生错误（仅在eevents中返回;在events中忽略）。

POLLHUP，挂断（仅在eevents中返回;在events中忽略）。请注意，当从诸如管道或流套接字的通道中读取时，该事件仅指示对端封闭其通道的结尾。在通道中的所有未完成数据被消耗之后，从通道的后续读取将返回0（文件结尾）。

POLLNVAL无效请求：fd未打开（仅在revents中返回;在events中忽略）。

当使用定义的\_XOPEN\_SOURCE进行编译时，可以使用以下内容，它们不会超出以上列出的更多信息：

POLLRDNORM相当于POLLIN。

POLLRDBAND可以读取优先级带数据（通常在Linux上未使用）。

POLLWRNORM相当于POLLOUT。

POLLWRBAND可写入优先级数据。

Linux知道，但不使用POLLMSG。

       如果任何文件描述符都没有发生任何事件（并且没有错误），那么poll（）将阻塞，直到发生其中一个事件。

参数 nfds为要监听的文件描述符个数，也就是fds数组的元素个数。

参数timeout timeout参数指定poll等待文件描述符准备好之前应阻塞毫秒数。函数将阻塞直到：

        \*文件描述符就绪;

        \*呼叫被信号处理中断;

        \*超时到期。

 请注意，timeout将四舍五入为系统时钟粒度，并且内核调度延迟意味着阻塞间隔可能会超出少量。 timeout指定负值意味着无限等待。 指定为零会导致poll（）立即返回，即使没有文件描述符准备就绪。

函数poll（）和ppoll（）之间的关系类似于select和pselec之间的关系：就像pselect，ppoll允许应用程序安全地等待，直到文件描述符准备好或直到信号被捕获。

除了超时精度外，下列ppoll调用

ready = ppoll(&fds, nfds, tmo\_p, &sigmask);

相当于原子的执行以下调用

sigset\_t origmask;

int timeout;

timeout = (tmo\_p == NULL) ? -1 : (tmo\_p->tv\_sec \* 1000 + tmo\_p->tv\_nsec / 1000000);

pthread\_sigmask(SIG\_SETMASK, &sigmask, &origmask);

ready = poll(&fds, nfds, timeout);

pthread\_sigmask(SIG\_SETMASK, &origmask, NULL);

如果sigmask参数指定为NULL，则不执行信号掩码操作（因此ppoll（）仅在timeout参数的精度中与poll（）不同）。

参数tmo\_p指定ppoll（）将阻塞的时间的上限。 此参数是指向以下形式的结构的指针：

struct timespec

{

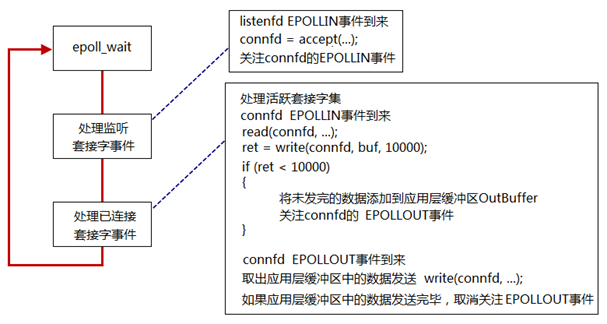
long tv\_sec; /\* seconds \*/

long tv\_nsec; /\* nanoseconds \*/

};

 如果tmo\_p指定为NULL，则ppoll（）可以无限期地阻止。

## 3.epoll模型



在linux的网络编程中，很长的时间都在使用 select来做事件触发。在linux新的内核中， 有了一种替换它的机制，就是 epoll。 相比于 select，epoll 最大的好处在于它不会随着监听 fd 数目的增长而降低效率。因为在内核中的 select 实现中，它是采用轮询来处理的，轮询的 fd 数目越多，自然耗时越多。select模型中，一旦监听对象发生变化，并不能知道究竟时那个对象的状态变化了，需要通过轮询才能直到，这就使得对大多状态为改变的对象的轮询操作变得毫无意义。而epoll则是采用事件响应，当某个对象状态改变，可以直接知道是哪一个发生了改变。

此外 linux/posix\_types.h 头文件有这样的声明：

#define \_\_FD\_SETSIZE 1024

表示 select 最多同时监听 1024 个 fd，当然，可以通过修改头文件再重编译内核来扩大这个数目，但这似乎并不治本。以下文件可以用来限制epoll使用的内核态内存空间大小：

/proc/sys/fs/epoll/max\_user\_watches

max\_user\_watches文件用来设置用户在所有epoll实例中注册的文件描述符数量上限，作用于每个用户ID。单个注册文件描述符在32位内核上消耗90字节，在64位内核上消耗160字节。max\_user\_watches的默认值是可用内核内存空间的1/25（4%）除以单个注册文件描述符消耗的字节数。在1GB内存的机器上大约是10万左 右，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。

epoll优点：

（1）支持一个进程打开大数目的socket描述符(FD)。

（2）IO效率不随FD数目增加而线性下降， epoll只会对"活跃"的socket进行操作---这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。只有活跃的fd才会调用callback

（3）内核微调。

（4）使用mmap加速内核与用户空间的消息传递。select,poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存拷贝就很重要，在这点上，epoll是通过内核于用户空间mmap同一块内存实现的。

（5）与select相比，不复用监听的文件描述集合来传递结果。这样不需要每次等待前对文件描述符集合重新赋值。select每次监听前都需要清空描述符集合，再重新注册。

epoll 的接口非常简单，一共就三个函数：

### 3.1 创建epoll句柄

#include <sys/epoll.h>

int epoll\_create(int size);

int epoll\_create1(int flags);

epoll\_create和epoll\_create1创建一个 epoll 的句柄，成功返回描述符，出错返回-1, 设置了标准错误描述，可使用perror函数查看。

参数size 用来告诉内核这个监听的数目一共有多大。(内核3.0以上这个参数被忽略但是必须大于0)这个参数不同于 select()中的第一个参数是给出最大监听的描述符 fd+1 的值。需要注意的是，当创建好 epoll 句柄后， 它就是会占用一个fd值，在 linux下如果查看/proc/进程 id/fd/，是能够看到这个 fd 的，所以在使用完 epoll 后，必须调用 close()关闭，否则可能导致 fd 被耗尽。

参数flags为文件描述符标志位，取0时和epoll\_create一样，也可以取EPOLL\_CLOEXEC来设置文件描述符的FD\_CLOEXEC标志位。

### 3.2 epoll 事件注册函数

#include <sys/epoll.h>

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

epoll事件注册函数，它不同与 select()是在监听事件时告诉内核要监听什么类型的事件， 而是在这里先注册要监听的事件类型。成功返回0，出错返回-1, 设置了标准错误描述，可使用perror函数查看。

参数epfd是 epoll\_create()的返回值。

参数op表示动作，用三个宏来表示：

EPOLL\_CTL\_ADD：注册新的 fd 到 epfd 中；

EPOLL\_CTL\_MOD：修改已经注册的 fd 的监听事件；

EPOLL\_CTL\_DEL：从 epfd 中删除一个 fd；

参数fd是需要监听的 fd，

参数event是告诉内核需要监听什么事，struct epoll\_event 结构如下：

struct epoll\_event

{

\_\_uint32\_t events; /\* Epoll events \*/

epoll\_data\_t data; /\* User data variable \*/

};

typedef union epoll\_data

{

void \*ptr;

int fd;

uint32\_t u32;

uint64\_t u64;

} epoll\_data\_t;

events 可以是以下几个宏的集合：

EPOLLIN ：表示对应的文件描述符可以读（包括对端 SOCKET 正常关闭）；

EPOLLOUT：表示对应的文件描述符可以写；

EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；

EPOLLRDHUP，Stream socket对端关闭连接，或者关闭写端，这用于编写简单代码特别有用，以便在使用边缘触发监视时检测对等关机。）

EPOLLERR：表示对应的文件描述符发生错误；epoll\_wait将永远等待这个事件;没有必要将其设置为events。

EPOLLHUP，在相关的文件描述符发生挂起。 epoll\_wait将永远等待这个事件;没有必要将其设置为事件。请注意，当从诸如管道或流套接字的通道读取时，该事件仅指示对端关闭其通道的结尾。在通道中的所有未完成数据已被读完之后，从通道的后续读取将返回0（文件结尾）。

EPOLLET：将 EPOLL设为边缘触发(ET)模式，这是相对于水平触发(Level Triggered)来说的。

EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个 socket 的话，需要用epoll\_ctl再次把这个 socket 加入到 EPOLL 队列里。

EPOLLWAKEUP，如果EPOLLONESHOT和EPOLLET显示设置，并且进程具有CAP\_BLOCK\_SUSPEND功能，该标识确保在此事件处于待处理或正在处理的情况下系统不会进入“挂起”或“休眠”。 “处理”是指从这一次epoll\_wait调用返回直到下一次调用epoll\_wait。

data为一个联合体，成员为fd。

### 3.3 等待事件

#include <sys/epoll.h>

int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);

int epoll\_pwait(int epfd, struct epoll\_event \*events, int maxevents, int timeout, const sigset\_t \*sigmask);

函数epoll\_wait等待事件的产生，类似于 select()调用。该函数返回需要处理的事件数目，如返回 0 表示已超时。-1表示错误, 设置了标准错误描述，可使用perror函数查看。

参数 events 用来从内核得到事件的集合，是一个epoll\_event类型的数组或者元素。参数maxevents 告之内核这个 events 有多大，这个 maxevents必须大于0，是参数events的元素个数。否则就就可能装不下那么多事件。

参数 timeout 是超时时间（毫秒，0 会立即返回，-1 将不确定，也有说法说是永久阻塞）。

函数epoll\_wait（）和epoll\_pwait（）之间的关系类似于select和pselect之间的关系：像pselect，epoll\_pwait（）允许应用程序安全地等到文件描述符准备好或直到 一个信号被捕获。

以下epoll\_pwait（）调用：

            ready = epoll\_pwait（epfd，＆events，maxevents，timeout，＆sigmask）;

相当于原子地执行以下调用：

            sigset\_t origmask;

            pthread\_sigmask（SIG\_SETMASK，＆sigmask，＆origmask）;

            ready = epoll\_wait（epfd，＆events，maxevents，timeout）;

            pthread\_sigmask（SIG\_SETMASK，＆origmask，NULL）;

参数sigmask可以指定为NULL，在这种情况下epoll\_pwait（）等效于epoll\_wait（）。

### 3.4 EPOLL 两种事件模型

事件有两种模型：

Edge Triggered (ET) 边缘触发：只有当有数据到来，才触发，不管缓存区中是否还有数据。

Level Triggered (LT) 水平触发 ：只要缓冲区有数据都会触发。

典型场景：

（1）我们已经把管道的读端文件描述符(rfd)注册到 epoll 描述符中。

（2）这个时候从管道的另一端被写入了 2KB 的数据

（3）调用 epoll\_wait(2)，并且它会返回 rfd，说明它已经准备好读取操作

（4）然后我们读取了 1KB 的数据

（5）调用 epoll\_wait(2)......

Edge Triggered 工作模式：如果我们在第 1 步将rfd 注册到 epoll 描述符的时候使用了 EPOLLET 标志，那么在第 5 步调用epoll\_wait之后将有可能会永久阻塞，因为剩余的数据还存在于文件的输入缓冲区内，而数据发出端还在等待一个针对已经发出数据的反馈信息。所以，第5步的调用方可能最终一直在等待数据到来，但数据其实已经在输入缓存区。原因是ET模式下，只有在监视的文件描述符上发生了某个事件的时候epoll才会汇报事件。经过第2步的写操作和第3步的事件处理，rfd上只会产生一次事件。由于第4步的读操作没有读完全部的缓冲区数据，第5步对epoll\_wait的调用可能会永远阻塞。如果调用者对这种情况作出处理，则调用者可能会放弃等待仍在存在于文件输入缓冲区内的剩余数据。

在上面的例子中，会有一个事件产生在 rfd上，因为在第 2 步执行了一个写操作，事件将会在第 3 步汇报后被销毁。因为第 4 步的读取操作没有读完文件输入缓冲区内的数据，因此我们在第 5 步调用 epoll\_wait完成后，是否挂起是不确定的。epoll 工作在 ET 模式的时候，必须使用非阻塞套接口，以避免由于一个文件描述符的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。

最好以下面的方式调用 ET 模式的 epoll 接口：

（1）基于非阻塞文件 (O\_NONBLOCK )

（2）只有当 read或者 write返回 EAGAIN 时才进行下一次epoll\_wait。但并不是说每次 read()时都需要循环读，直到读到产生一个 EAGAIN 才认为此次事件处理完成，只需判断read()/write返回的读到或写入的数据长度小于请求的数据长度时，就可以确定此时缓冲中已没有数据了或缓冲区已满，也就可以认为此事读/写事件已处理完成。

Level Triggered 工作模式相反的，以 LT 方式调用 epoll 接口的时候，它就相当于一个速度比较快的 poll，并且无论后面的数据是否被使用，因此他们具有同样的职能。

LT(level triggered)是缺省的工作方式，并且同时支持 block 和 no-block socket.在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的 fd 进行 IO 操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。 传统的 select/poll 都是这种模型的代表．

ET(edge-triggered)是高速工作方式，只支持 no-block socket。在这种模式下，仅当描述符从未就绪变为就绪时，内核通过 epoll 告诉你。然后它会假设你知道文件描述符已经就绪， 并且不会再为那个文件描述符发送更多的就绪通知，直到你做了某些操作导致那个文件描述符不再为就绪状态了(比如，你在发送，接收或者接收请求，或者发送接收的数据少于一定量时导致了一个 EWOULDBLOCK 错误）。但是请注意，如果一直不对这个 fd 作 IO 操作(从而 导致它再次变成未就绪)，内核不会发送更多的通知(only once),不过在 TCP 协议中，ET模式的加速效用仍需要更多的检测确认（这句话不理解）。

在许多测试中我们会看到如果没有大量的 idle -connection 或者 dead-connection，epoll 的效率并不会比 select/poll 高很多，但是当我们遇到大量的 idle- connection(例如 WAN 环境 中存在大量的慢速连接)，就会发现 epoll 的效率大大高于 select/poll。

当使用 epoll 的 ET 模型来工作时，当产生了一个 EPOLLIN 事件后， 读数据的时候需要考虑的是当 recv()返回的大小如果等于请求的大小，那么很有可能是缓冲区还有数据未读完，也意味着该次事件还没有处理完，所以还需要再次读取：

while(rs)

{

buflen = recv(activeevents[i].data.fd, buf, sizeof(buf), 0);

if(buflen < 0)

{

// 由于是非阻塞的模式,所以当 errno 为 EAGAIN 时,表示当前缓冲区已无数据可读

// 在这里就当作是该次事件已处理处.

if(errno == EAGAIN)

break;

else

return;

}

else if(buflen == 0)

{

// 这里表示对端的 socket 已正常关闭.

}

if(buflen == sizeof(buf)

rs = 1; // 需要再次读取

else

rs = 0;

}

还有，假如发送端流量大于接收端的流量(意思是 epoll 所在的程序读比转发的 socket 要 快),由于是非阻塞的socket,那么send()函数虽然返回,但实际缓冲区的数据并未真正发给接收端,这样不断的读和发，当缓冲区满后会产生 EAGAIN 错误(参考 man send),同时,不理会（忽略）这次请求发送的数据.所以,需要封装socket\_send()的函数用来处理这种情况,该函数会尽量将数据写完再返回，返回-1 表示出错。在 socket\_send()内部,当写缓冲已满(send()返回-1,且 errno 为 EAGAIN),那么会等待后再重试.这种方式并不很完美,在理论上可能会长时间的阻塞在 socket\_send()内部,但暂没有更好的办法。

ssize\_t socket\_send**(**int sockfd**,** const char**\*** buffer**,** size\_t buflen**)**

**{**

ssize\_t tmp**;**

size\_t total **=** buflen**;**

const char **\***p **=** buffer**;**

**while(**1**)**

**{**

tmp **=** send**(**sockfd**,** p**,** total**,** 0**);**

**if(**tmp **<** 0**)**

**{**

// 当 send 收到信号时,可以继续写,但这里返回-1.

**if(**errno **==** EINTR**)**

**return** **-**1**;**

// 当 socket 是非阻塞时,如返回此错误,表示写缓冲队列已满,

// 在这里做延时后再重试.

**if(**errno **==** EAGAIN**)**

**{**

usleep**(**1000**);**

**continue;**

**}**

**return** **-**1**;**

**}**

**if((**size\_t**)**tmp **==** total**)**

**return** buflen**;**

total **-=** tmp**;**

p **+=** tmp**;**

**}**

**return** tmp**;**

**}**

### 3.5 epoll 使用的一些问题

#### 3.5.1 自动休眠问题

如果系统设置了自动休眠模式（通过/sys/power/autosleep），当唤醒设备的事件发生时，设备驱动会保持唤醒状态，直到事件进入排队状态。为了保持设备唤醒直到事件处理完成，必须使用epoll EPOLLWAKEUP 标记。

一旦给structe poll\_event中的events字段设置了EPOLLWAKEUP标记，系统会在事件排队时就保持唤醒，从epoll\_wait调用开始，持续要下一次epoll\_wait调用。

### 3.5.2 监测数量限制

以下文件可以用来限制epoll使用的内核态内存空间大小(Linux 2.6.28 开始)：

/proc/sys/fs/epoll/max\_user\_watches

max\_user\_watches文件用来设置用户在所有epoll实例中注册的文件描述符数量上限，作用于每个用户ID。单个注册文件描述符在32位内核上消耗90字节，在64位内核上消耗160字节。max\_user\_watches的默认值是可用内核内存空间的1/25（4%）除以单个注册文件描述符消耗的字节数。

#### 3.5.3 避免饥饿（边缘触发）

如果I/O数据量很大，可能在读取数据的过程中其他文件得不到处理（因为要循环recv或者send直到EAGAIN才返回），造成饥饿。解决方法是维护一个就绪链表，在关联数据结构中标记文件描述符为就绪状态，由此可以记住哪些文件在等待，并对所有就绪文件作轮转处理。

#### 3.5.4 事件缓存陷阱

如果使用事件缓存，或者存储epoll\_wait返回的所有文件描述符，就需要提供方法动态标记关闭状态（比如，由于其他事件处理造成文件描述符关闭），假设从epoll\_wait收到100个事件，A事件造成B事件关闭，如果移除B事件结构并关闭文件描述符，事件缓存仍然认为有事件在等待文件描述符，从而造成混乱。

解决方法是，在A事件处理过程中，调用epoll\_ctl(EPOLL\_CTL\_DEL)来移除B文件描述符并关闭，然后标记关联的数据结构为已移除，并关联到移除列表。在后续事件处理过程中，当发现B文件描述符的新事件时，可以通过检查标记发现文件描述符已移除，避免产生混乱。

#### 3.5.5 正确的accept

(1) 阻塞模式 accept 存在的问题

考虑这种情况：TCP连接被客户端夭折，即在服务器调用accept之前，客户端主动发送RST终止连接，导致刚刚建立的连接从就绪队列中移出，如果套接口被设置成阻塞模式，服务器就会一直阻塞在accept调用上，直到其他某个客户建立一个新的连接为止。但是在此期间，服务器单纯地阻塞在accept调用上，就绪队列中的其他描述符都得不到处理。

解决办法是把监听套接口设置为非阻塞，当客户在服务器调用accept之前中止某个连接时，accept调用可以立即返回-1，这时源自Berkeley的实现会在内核中处理该事件，并不会将该事件通知给epoll，而其他实现把errno设置为ECONNABORTED或者EPROTO错误，我们应该忽略这两个错误。

(2)ET模式下accept存在的问题

考虑这种情况：多个连接同时到达，服务器的TCP就绪队列瞬间积累多个就绪连接，由于是边缘触发模式，epoll只会通知一次，accept只处理一个连接，导致TCP就绪队列中剩下的连接都得不到处理。

解决办法是用while循环抱住accept调用，处理完TCP就绪队列中的所有连接后再退出循环。如何知道是否处理完就绪队列中的所有连接呢？accept返回-1并且errno设置为EAGAIN就表示所有连接都处理完。

综合以上两种情况，服务器应该使用非阻塞socket来accept。

#### 3.5.6 水平触发模式，当socket可写时会不停的触发socket可写的事件

第一种最普遍的方式：需要向socket写数据的时候才把socket加入epoll，等待可写事件。接受到可写事件后，调用write或者send发送数据。当所有数据都写完后，把socket移出epoll。

缺点是，即使发送很少的数据，也要把socket加入epoll，写完后在移出epoll，有一定操作代价。

第二种方式：

开始不把socket加入epoll，需要向socket写数据的时候，直接调用write或者send发送数据。如果返回EAGAIN，把socket加入epoll，在epoll的驱动下写数据，全部数据发送完毕后，再移出epoll。

优点是：数据不多的时候可以避免epoll的事件处理，提高效率。

#### 3.5.7 ET模式下多个数据块同时到达

因为即使使用ET模式的 epoll，在收到多个数据块的时候仍然可能会产生多个事件。调用者可以设定 EPOLLONESHOT 标志，在 epoll\_wait收到事件后 epoll 会将关联的文件描述符从 epoll 监测集合中禁止掉。当给epoll设置EPOLLONESHOT标记时，调用方必须需要通过epoll\_ctl对文件描述符设置EPOLL\_CTL\_MOD标记。

## 4. select/poll/epoll对比

|  |  |
| --- | --- |
| 原理 | |
| select | select本质上是通过设置或者检查存放fd标志位的数据结构来进行下一步处理。缺点是：  （1）单个进程可监视的fd数量被限制  （2）需要维护一个用来存放大量fd的数据结构，使得用户空间和内核空间在传递该结构时复制开销大  （3）对socket进行扫描时是线性扫描  （4）水平触发 |
| poll | poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态，如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有fd后没有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历fd。这个过程经历了多次无谓的遍历。  它没有最大连接数的限制，原因是它是基于链表来存储的，但是同样有一个缺点：  大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。  poll还有一个特点是“水平触发”，如果报告了fd后，没有被处理，那么下次poll时会再次报告该fd。 |
| epoll | 在前面说到的复制问题上，epoll使用mmap减少复制开销。  还有一个特点是，epoll使用“事件”的就绪通知方式，通过epoll\_ctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epoll\_wait便可以收到通知。 |

|  |  |
| --- | --- |
| **一个进程所能打开的最大连接数** | |
| select | 单个进程所能打开的最大连接数有FD\_SETSIZE宏定义，其大小是32个整数的大小（在32位的机器上，大小就是32\*32，同理64位机器上FD\_SETSIZE为32\*64），当然我们可以对进行修改，然后重新编译内核，但是性能可能会受到影响，这需要进一步的测试。 |
| poll | poll本质上和select没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的 |
| epoll | 虽然连接数有上限，但是很大，1G内存的机器上可以打开10万左右的连接，2G内存的机器可以打开20万左右的连接，(cat /proc/sys/fs/file-max) 通过命令查看 |

注意：epoll效率并不一定高于poll和select。当连接都处于活跃状态时，效率差不多。只有当有大量连接是空闲的或死亡的时，epoll效率远高于poll和select。

# timer/timerfd + eventfd

## 1 timer

int timer\_create(clockid\_t clock\_id, struct sigevent \*sevp, timer\_t \*timerid)

int timer\_delete(timer\_t timerid);

timer\_create（）创建一个进程自己的间隔定时器。新定时器的ID在timerid指向的缓冲区中返回，该缓冲区必须是非空指针。直到定时器被删除，该ID在进程中是唯一的。新计时器创建时并不启动。成功返回0，出错返回-1, 设置了标准错误描述，可使用perror函数查看。进程可以通过调用timer\_create()创建特定的定时器，定时器是每个进程自己的，不是在fork时继承的。

参数clock\_id指定新计时器用来测量时间的时钟。它可以被指定为以下值之一：

CLOCK\_REALTIME，可设置的系统范围的相对时钟。

CLOCK\_MONOTONIC，绝对时钟，一个单调递增的时钟，用于衡量系统启动后的某些未指定点的时间。

CLOCK\_PROCESS\_CPUTIME\_ID，用于测量调用进程（所有线程）消耗的（用户和系统）CPU时间的时钟

 CLOCK\_THREAD\_CPUTIME\_ID，测量调用线程占用（用户和系统）CPU时间的时钟。

 除了上述值之外，clockid可以指定为通过调用clock\_getcpuclockid（）或pthread\_getcpuclockid（）返回的clockid。

int clock\_getcpuclockid(pid\_t pid, clockid\_t \*clock\_id);

int pthread\_getcpuclockid(pthread\_t thread, clockid\_t \*clock\_id);

这两个函数成功返回0，失败返回非0错误码。

参数sevp指向一个sigevent结构，指定当定时器到期时如何通知调用者。如果sevp为NULL，那么定时器到期会产生默认的信号，对CLOCK\_REALTIMER来说，默认信号就是SIGALRM。对那些定时器到期时要产生除默认信号之外的其它信号的定时器来说，程序必须将evp->sigev\_signo设置为期望的信号码。struct sigevent 结构中的成员evp->sigev\_notify说明了定时器到期时应该采取的行动。通常，这个成员的值为SIGEV\_SIGNAL,这个值说明在定时器到期时，会产生一个信号。程序可以将成员evp->sigev\_notify设为SIGEV\_NONE来防止定时器到期时产生信号。

如果几个定时器产生了同一个信号，处理程序可以用evp->sigev\_value来区分是哪个定时器产生了信号。要实现这种功能，程序必须在为信号安装处理程序时，使用struct sigaction的成员sa\_flags中的标志符SA\_SIGINFO。

union sigval

{ /\* 通过通知传递的数据 \*/

int sival\_int; /\* 整数 \*/

void \*sival\_ptr; /\*指针 \*/

};

struct sigevent

{

int sigev\_notify; /\*通知方式 \*/

int sigev\_signo; /\* 通知信号 \*/

union sigval sigev\_value; /\*通过通知传递的数据 \*/

void (\*sigev\_notify\_function) (union sigval); /\* 线程通知函数 (SIGEV\_THREAD) \*/

void \*sigev\_notify\_attributes; /\* 通知线程的属性 (SIGEV\_THREAD) \*/

pid\_t sigev\_notify\_thread\_id; /\* 要发信号的线程号(SIGEV\_THREAD\_ID) \*/

};

sevp.sigev\_notify字段可以具有以下值：

SIGEV\_NONE，定时器到期时不要通知。只提供通过timer\_gettime和timer\_getoverrun查询超时信息。

SIGEV\_SIGNAL，定时器到期后，为进程生成信号sigev\_signo。内核会将sigev\_signo所指定的信号传送给进程。在信号处理程序中，si\_value会被设定会sigev\_value。 siginfo\_t结构的si\_code字段将被设置为SI\_TIMER。在任何时间点，对于给定的定时器，至多一个信号被排队到进程;。通过发送sigev\_signo指定的信号通知进程。如果使用sigaction的SA\_SIGINFO标志注册的信号处理程序捕获信号，则在作为处理程序的第二个参数传递的siginfo\_t结构中设置以下字段：

si\_code此字段设置为取决于传递通知的API的值。

si\_signo此字段设置为信号编号（即与sigev\_signo中相同的值）。

si\_value此字段设置为sigev\_value中指定的值。

根据API，其他字段也可以在siginfo\_t结构中设置。

如果使用sigwaitinfo接受信号，也可以使用相同的信息。

 SIGEV\_THREAD，定时器到期后，当定时器到期，内核会(在此进程内)以sigev\_notification\_attributes（如果不为NULL）为线程属性创建一个线程，并且让它执行sigev\_notify\_function，传入sigev\_value作为一个参数。

 SIGEV\_THREAD\_ID，对于SIGEV\_SIGNAL，但信号是针对sigev\_notify\_thread\_id中给出ID的线程，该线程必须是与调用者相同进程的线程。 sigev\_notify\_thread\_id字段指定内核线程ID，即clone或gettid返回的值。此标志仅供线程库使用。

 指定sevp为NULL等效于指定这样一个sigvent结构的指针，其中sigev\_notify是SIGEV\_SIGNAL，sigev\_signo是SIGALRM，sigev\_value.sival\_int是计时器ID。

错误码

EAGAIN，定时器结构内核分配期间的临时错误。

EINVAL，clock ID，sigev\_notify，sigev\_signo或sigev\_notify\_thread\_id无效。

ENOMEM，无法分配内存。

timer\_delete（）删除在timerid中给出ID的定时器。 如果在该调用是时定时器被启动，则在被删除之前它被停止。 被删除的定时器产生的任何待处理信号的处理是未指定的。

int timer\_settime(timer\_t timerid, int flags, const struct itimerspec \*new\_value, struct itimerspec \*old\_value);

int timer\_gettime(timer\_t timerid, struct itimerspec \*curr\_value);

timer\_settime（）启用或停止由timerid标识的定时器。成功返回0，出错返回-1, 设置了标准错误描述，可使用perror函数查看。

struct timespec

{

time\_t tv\_sec; /\* Seconds \*/

long tv\_nsec; /\* Nanoseconds \*/

};

struct itimerspec

{

struct timespec it\_interval; /\* Timer interval \*/

struct timespec it\_value; /\* Initial expiration \*/

};

itimerspec结构的每个子结构都是一个timespec结构，允许以秒和纳秒指定时间值。这些时间值是根据timer\_create创建定时器时指定的时钟来测量的。

如果new\_value-> it\_value指定非零值（即，任一子字段不为零），则timer\_settime（）将（启动）定时器，将其设置为在给定时间初始到期。 （如果定时器已被启用，则覆盖以前的设置。）

如果new\_value-> it\_value指定零值（即两个子字段为零），则定时器被取消。new\_value-> it\_interval字段指定计时器的周期，以秒和纳秒为单位。如果此字段为非零，则每次启用计时器到期时，定时器将从new\_value-> it\_interval中指定的值重新加载。如果new\_value-> it\_interval指定一个零值，那么在it\_value指定的时间内，定时器会过期一次。

参数flags，指定计时方式是相对还是绝对。0表示相对，TIMER\_ABSTIME表示绝对。默认情况下，new\_value-> it\_value中指定的初始到期时间相对于通话时定时器时钟的当前时间来解释。这可以通过在flags中指定TIMER\_ABSTIME进行修改，在这种情况下，new\_value-> it\_value被解释为在定时器时钟上测得的绝对值;也就是说，当时钟值达到由new\_value-> it\_value指定的值时，定时器将会过期。如果指定的绝对时间已经过去，定时器会立即过期，超程计数（见timer\_getoverrun）将被正确设置。  如果在基于该时钟的绝对定时器被启用时调整CLOCK\_REALTIME时钟的值，则定时器的到期将被适当调整。对CLOCK\_REALTIME时钟的调整对基于该时钟的相对定时器没有影响。

 如果old\_value不为NULL，则它指向用于返回定时器的前一个间隔（在old\_value-> it\_interval）中的缓冲区以及定时器先前已经过期的时间量（在old\_value-> it\_value中）

timer\_gettime（）返回到下一个到期的时间，由timerid指定的定时器在curr\_value指向的缓冲区中的间隔。 在curr\_value-> it\_value中返回到下一个定时器到期的剩余时间; 这永远是一个相对值，无论在启用定时器时是否使用了TIMER\_ABSTIME标志。 如果curr\_value-> it\_value中返回的值为零，则定时器当前被取消。 计时器间隔在curr\_value-> it\_interval中返回。 如果curr\_value-> it\_interval返回的值为零，那么这是一个“一次性”定时器。

int timer\_getoverrun(timer\_t timerid);

timer\_getoverrun（）返回由timerid引用的定时器的“超限计数”。成功时，timer\_getoverrun（）返回指定定时器的超程计数;如果没有超额发生，这个计数可能为0。发生故障时，返回-1，并将errno设置为指示错误

取得一个定时器的超限运行次数，有可能一个定时器到期了，而同一定时器上一次到期时产生的信号还处于挂起状态。在这种情况下，其中的一个信号可能会丢失。这就是定时器超限。程序可以通过调用timer\_getoverrun来确定一个特定的定时器出现这种超限的次数。定时器超限只能发生在同一个定时器产生的信号上。由多个定时器，甚至是那些使用相同的时钟和信号的定时器，所产生的信号都会排队而不会丢失。

执行成功时，timer\_getoverrun()会返回定时器初次到期与通知进程(例如通过信号)定时器已到期之间额外发生的定时器到期次数。举例来说，在我们之前的例子中，一个1ms的定时器运行了10ms，则此调用会返回9。如果超限运行的次数等于或大于DELAYTIMER\_MAX，则此调用会返回DELAYTIMER\_MAX。

## 2 timerfd

timerfd是Linux提供的一个定时器接口。这个接口基于文件描述符，通过文件描述符的可读事件进行超时通知，所以能够被用于select/poll/epoll的应用场景。timerfd是linux内核2.6.25版本中加入的接口。

#include <sys/timerfd.h>

int timerfd\_create(int clockid, int flags);

功能：该函数生成一个定时器对象，返回与之关联的文件描述符。成功返回0，出错返回-1, 设置了标准错误描述，可使用perror函数查看。

参数clockid:可设置为

CLOCK\_REALTIME：相对时间，从1970.1.1到目前的时间。更改系统时间会更改获取的值，它以系统时间为坐标。

CLOCK\_MONOTONIC：绝对时间，获取的时间为系统重启到现在的时间，更改系统时间对齐没有影响。

参数flags: 可设置为

TFD\_NONBLOCK（非阻塞），

TFD\_CLOEXEC（同O\_CLOEXEC）

新版本flags必须置为0.

#include <sys/timerfd.h>

int timerfd\_settime(int fd, int flags, const struct itimerspec \*new\_value, struct itimerspec \*old\_value);

int timerfd\_gettime(int fd, struct itimerspec \*curr\_value);

功能：该函数能够启动和停止定时器。成功返回0，出错返回-1, 设置了标准错误描述，可使用perror函数查看。

参数fd: timerfd对应的文件描述符

参数flags:

0表示是相对定时器

TFD\_TIMER\_ABSTIME表示是绝对定时器

参数new\_value:设置超时时间，如果为0则表示停止定时器。

参数old\_value:一般设为NULL, 不为NULL,则返回定时器这次设置之前的超时时间。

struct timespec

{

time\_t tv\_sec; /\*秒\*/

long tv\_nsec /\*纳秒\*/

}

struct itimerspec

{

struct timespec it\_interval; /\*周期间隔 \*/

struct timespec it\_value; /\*初始间隔\*/

}

操作：

read：读取缓冲区中的数据，其占据的存储空间为sizeof(uint\_64)，表示超时次数。

select/poll/epoll：当定时器超时时，会触发定时器相对应的文件描述符上的读操作，IO复用操作会返回，然后再去对该读事件进行处理。

应用场景：定时执行一些任务。

## 3 eventfd

从Linux 2.6.27版本开始，新增了不少系统调用，其中包括eventfd，它的主要是用于进程或者线程间通信(如通知/等待机制的实现)。

函数原型：

#include <sys/eventfd.h>

int eventfd(unsigned int initval, int flags);

返回值：函数返回一个文件描述符，与打开的其他文件一样，可以进行读写操作。

参数flags支持以下标志位：

EFD\_NONBLOCK类似于使用O\_NONBLOCK标志设置文件描述符。

EFD\_CLOEXEC 类似open以O\_CLOEXEC标志打开，O\_CLOEXEC应该表示执行exec()时，之前通过open()打开的文件描述符会自动关闭。

EFD\_SEMAPHORE,提供类似信号的语义，用于从新的文件描述符读取

如果是2.6.26或之后版本的内核，flags 必须设置为0。

参数initval：初始化计数器值，该值保存在内核。

操作：

read：如果计数器A的值不为0时，读取成功，获得到该值,并将计数器清零。如果A的值为0，非阻塞模式时，会直接返回失败，并把error置为EINVAL;如果为阻塞模式，一直会阻塞到A为非0为止。

write：将缓冲区写入的8字节整形值加到内核计数器上，即会增加8字节的整数在计数器A上，如果其值达到0xfffffffffffffffe时，就会阻塞（在阻塞模式下），直到A的值被read。

# EAGAIN错误和EINTR错误

## 1 EAGIN错误

在linux环境下开发经常会碰到很多错误(设置errno)，其中EAGAIN是其中比较常见的一个错误(比如用在非阻塞操作中)。

从字面上来看，是提示再试一次。这个错误经常出现在当应用程序进行对非阻塞的fd或socket调用了阻塞操作。例如，以O\_NONBLOCK的标志打开文件/socket/FIFO，如果你连续做read操作而没有数据可读。此时程序不会阻塞起来等待数据准备就绪返回，read函数会返回一个错误EAGAIN，提示你的应用程序现在没有数据可读请稍后再试。

又例如，当一个系统调用(比如fork)因为没有足够的资源(比如虚拟内存)而执行失败，返回EAGAIN提示其再调用一次(也许下次就能成功)。

Linux - 非阻塞socket编程处理EAGAIN错误

在linux进行非阻塞的socket接收数据时经常出现Resource temporarily unavailable，errno代码为11(EAGAIN)，这表明你在非阻塞模式下调用了阻塞操作，在该操作没有完成就返回这个错误，这个错误不会破坏socket的同步，不用管它，下次循环接着recv就可以。对非阻塞socket而言，EAGAIN不是一种错误。在VxWorks和Windows上，EAGAIN的名字叫做EWOULDBLOCK。

## **2 EINTR错误**

慢系统调用(slow system call)：此术语适用于那些可能永远阻塞的系统调用。永远阻塞的系统调用是指调用有可能永远无法返回，多数网络支持函数都属于这一类。如：若没有客户连接到服务器上，那么服务器的accept调用就没有返回的保证。

EINTR错误的产生：当阻塞于某个慢系统调用的一个进程捕获某个信号且相应信号处理函数返回时，该系统调用可能返回一个EINTR错误。例如：在socket服务器端，设置了信号捕获机制，有子进程，当在父进程阻塞于慢系统调用时由父进程捕获到了一个有效信号时，内核会致使accept返回一个EINTR错误(被中断的系统调用)。

当碰到EINTR错误的时候，可以采取有一些可以重启的系统调用要进行重启，而对于有一些系统调用是不能够重启的。例如：accept、read、write、select、和open之类的函数来说，是可以进行重启的。不过对于套接字编程中的connect函数我们是不能重启的，若connect函数返回一个EINTR错误的时候，我们不能再次调用它，否则将立即返回一个错误。针对connect不能重启的处理方法是，必须调用select来等待连接完成。

对于socket接口(指connect/send/recv/accept..等等后面不重复，不包括不能设置非阻塞的如select)，在阻塞模式下有可能因为发生信号，返回EINTR错误，由用户做重试或终止。

**但是，在非阻塞模式下，是否出现这种错误呢？**

对此，重温了系统调用、信号、socket相关知识，得出结论是：不会出现。

首先，

1）信号的处理是在用户态下进行的，也就是必须等待一个系统调用执行完了才会执行进程的信号函数，所以就有了信号队列保存未执行的信号

2）用户态下被信号中断时，内核会记录中断地址，信号处理完后，如果进程没有退出则重回这个地址继续执行

socket接口是一个系统调用，也就是即使发生了信号也不会中断，必须等socket接口返回了，进程才能处理信号。

也就是，EINTR错误是socket接口主动抛出来的，不是内核抛的。socket接口也可以选择不返回，自己内部重试之类的..

**那阻塞的时候socket接口是怎么处理发生信号的?**

举例

socket接口，例如recv接口会做2件事情，

1）检查buffer是否有数据，有则复制清除返回

2）没有数据，则进入睡眠模式，当超时、数据到达、发生错误则唤醒进程处理

socket接口的实现都差不了太多，抽象说

1）资源是否立即可用，有则返回

2）没有，就等...

对于

1）这个时候不管有没信号，也不返回EINTR，只管执行自己的就可以了

2）采用睡眠来等待，发生信号的时候进程会被唤醒，socket接口唤醒后检查有无未处理的信号(signal\_pending)会返回EINTR错误。

所以

socket接口并不是被信号中断，只是调用了睡眠，发生信号睡眠会被唤醒通知进程，然后socket接口选择主动退出，这样做可以避免一直阻塞在那里，有退出的机会。非阻塞时不会调用睡眠。

# 进程池流程

第一步： make\_child 初始化子进程 循环创建子进程，并初始化父进程的子进程管理结构体数组 parr，通过socket\_pair 将 socket 描述符一端放入数组

子进程流程

Recv\_fd 等待父进程发送任务 Hand\_request 发送文件数据 Write 向父进程发送完成任务

第二步： 父进程 epoll 监控 fd\_listen 描述符。 父进程 epoll 监控 parr 结构体数组的 socket 描述符

第三步： While1 启动epoll\_wait，等待是否有客户端连接 有客户端连接后，accept 获得描述符，循环找到非忙碌的子进程，并发送给子进程，标记对 应子进程忙碌。 当子进程完成任务后，父进程一旦监控 socket 描述符可读，代表子进程非忙碌，然后标记子 进程非忙碌。

难点： 1.子进程采用变长结构体发送文件（比较新）

不要把内存申请放在锁中

# 命令行参数处理

## 1 函数原型

#include <unistd.h>

int getopt(int argc, char \* const argv[],const char \*optstring);

extern char \*optarg;

extern int optind, opterr, optopt;

#include <getopt.h>

int getopt\_long(int argc, char \* const argv[],const char \*optstring,

const struct option \*longopts, int \*longindex);

int getopt\_long\_only(int argc, char \* const argv[],const char \*optstring,

const struct option \*longopts, int \*longindex);

## 2 函数描述

### 2.1 getopt

功能：getopt（）函数解析命令行参数。

参数argc和argv是程序调用时传递给main（）函数的参数参数。其中argc为传入程序参数的个数，默认为1，代表程序自身，argv为指针数组，存放命令行参数，传入的命令行参数以空白符分界，每个元素存放一个字符串，代表一个参数，argv[0]表示程序自身名字\*。

每个argv数组的元素就是一个命令行参数，以 ‘-’（而不是“ - ”或“ - -”）开头的argv元素就是一个选项。该元素（除了初始’ - ‘之外）是选项字符。如果重复调用了getopt（），它会从每个元素中连续返回每个选项字符。如：argv[1] = “-aabc”或 argv[1] =”-a”,那么此时的选项字符都是a。

参数optstring是选项字符串，比如：”a:b:cd::e”，这就是一个选项字符串。对应到命令行就是-a ,-b ,-c ,-d, -e 。一个冒号就表示这个选项后面必须带有参数（没有带参数会报错哦），但是这个参数可以和选项连在一起写，也可以用空格隔开，比如-a123 和-a 123（中间有空格） 都表示123是-a的参数；两个冒号的就表示这个选项可以有参数，也可以没有参数，但要注意有参数时，参数与选项之间不能有空格（有空格会报错的哦），这一点和一个冒号时是有区别的。

此外 GNU有额外的扩展。如果optstring包含W后跟分号，则-W foo被视为长选项–foo。 （-W选项由POSIX.2保留，用于实现扩展。）此行为是GNU扩展，在glibc 2之前不可用。

全局变量optind是要在argv数组中处理的下一个元素的下标（也叫索引）系统将该值初始化为1.调用者可以将其重置为1，以重新扫描相同的argv，或扫描新的参数向量。

全局变量optarg是用来保存选项的参数的；如果选项无参数则置为NULL。

全局变量oprerr表示是否将错误信息输出到stderr，为0时表示不输出.

全局变量optopt表示不在选项字符串optstring中的选项.

如果getopt（）找到另一个选项字符，则返回该字符，更新外部变量optind和一个静态变量nextchar，以便下次调用getopt（）可以使用选项字符或argv元素恢复扫描。

如果没有更多选项字符，getopt（）返回-1。那么optind是第一个argv元素的索引，它不是一个选项。

默认情况下，getopt（）在扫描时会排列argv的内容，最终所有既不是选项也不是选项参数的元素都会排到数组末尾。

还实现了另外两种模式。如果optstring的第一个字符为“+”或设置了环境变量POSIXLY\_CORRECT，则一旦遇到非选择参数，选项处理就会停止。如果optstring的第一个字符为’ - ‘，那么每个非选项argv元素都将被处理，就像它是一个带有字符代码1的选项的参数一样。（这被编写用于期望选项和其他argv元素以任何顺序使用的程序，并且关心两者的顺序。）。特殊参数“ - -”强制选择扫描的结束，无论扫描模式如何。

如果getopt（）不识别选项字符，则会向stderr打印一条错误消息，将该字符存储在optopt中，并返回’？’。调用程序可以通过将opterr设置为0来阻止错误消息。

如果getopt（）在argv中找到一个未包含在optstring中的选项字符，或者如果检测到错误的选项参数，则返回’？’并将外部变量optopt设置为实际选项字符。如果optstring的第一个字符（上述可选的’+’或’ - ‘）是冒号（’：’），那么getopt（）返回’：’而不是’？’以表示缺少选项参数。如果检测到缺少参数，并且optstring的第一个字符不是冒号，并且外部变量opterr是非零（这是默认值），则getopt（）将输出错误消息。

### 2.2 getopt\_long和getopt\_long\_only

getopt\_long（）函数的作用类似于getopt（），但它可以接收接受长选项，以‘–’开头。 （如果程序仅接受长选项，那么optstring应该被指定为一个空字符串（“”），而不是NULL）。如果缩写是唯一的或者可以精确的匹配一些定义的选项，则可以缩写长选项名称。长选项可能会采用–arg = param或–arg param形式的参数。

参数longopts是指向getopt.h中声明的struct选项数组的第一个元素的指针，通常情况下是一个数组。

struct option

{

const char \*name;

int has\_arg;

int \*flag;

int val;

};

成员name是长选项的名称。

成员has\_arg是：no\_argument（或0）如果该选项没有参数; required\_argument（或1）如果该选项需要参数;或optional\_argument（或2）如果该选项具有可选参数。

成员flag指定如何为长选项返回结果。如果flag为NULL，那么getopt\_long（）返回成员val。 （一般，设flag为NULL，调将val设置为等效的短选项字符），否则，getopt\_long（）返回0，并且flag指向一个变量，如果找到该变量，则将该变量值设置为val，但如果该变量为则不做改变。

成员val是要返回的值，或者加载到由flag指向的变量中。

注意：struct option 数组的最后一个元素必须用{0,0,0,0}填充。

如果longindex不为NULL，则它指向一个变量，该变量设置为long选项相对于longopts的索引。类似optind。

getopt\_long\_only（）类似于getopt\_long（），但’ - ‘以及“ – ”可以表示长选项。如果以“ - ”（而不是“ - ”）开头的选项与长选项不匹配，但与短选项匹配，则会将其解析为短选项。

## 3 返回值

如果成功找到一个选项，那么getopt（）返回选项字符。

如果所有命令行选项都已被解析，那么getopt（）返回-1。

如果getopt（）遇到没有在optstring中的选项字符，那么’？’ 被返回。

如果getopt（）遇到缺少参数的选项，则返回值取决于optstring中的第一个字符：如果为“：”，则返回“：”; 除此以外 ‘？’ 被返回。

当识别短选项时，getopt\_long（）和getopt\_long\_only（）也返回选项字符。 对于长选项，如果flag为NULL，则返回val，否则返回0。 错误和-1返回与getopt（）相同，加上’？’ 用于模糊匹配或无关的参数。

# 数据类型

## 1. 定宽整数类型（C99）

stdint.h是C99引入的头文件，作用在于规定不同编译器中的同一数据类型的长度（或叫宽度），该头文件定义了几种扩展类型(u)intN\_t，(u)int\_leastN\_t，(u)int\_fastN\_t， (u)intptr\_t，(u)intmax\_t,以及对应的最大值和最小值的宏定义\*\_MAX与\*\_MIN.以及其它扩展类型的最大值与最小值的宏定义。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 新类型 | 原始类型 | 新类型 | 原始类型 | 新类型 | 原始类型 |
| size\_t | unsigned long | int8\_t | signed char | uint8\_t | unsigned char |
| ssize\_t | long | int16\_t | signed short | uint16\_t | unsigned short |
| socklen\_t | unsigned int | int32\_t | signed int | uint32\_t | unsigned int |
| off\_t | long | int64\_t(64b) | signed long | uint64\_t(64b) | unsigned long |
|  |  | int64\_t(32b) | signed long long | uint64\_t(32b) | unsigned long long |
|  |  | int\_least8\_t | signed char | uint\_least8\_t | unsigned char |
|  |  | int\_least16\_t | signed short | uint\_least16\_t | unsigned short |
|  |  | int\_least32\_t | signed int | uint\_least32\_t | unsigned int |
|  |  | int\_least64\_t(64b) | signed long | uint\_least64\_t(64b) | unsigned long |
|  |  | int\_least64\_t(32b) | signed long long | uint\_least64\_t(32b) | unsigned long long |
|  |  | int\_fast8\_t | signed char | uint\_fast8\_t | unsigned char |
|  |  | int\_fast16\_t(64b) | signed long | uint\_fast16\_t(64b) | unsigned long |
|  |  | int\_fast32\_t(64b) | signed long | uint\_fast32\_t(64b) | unsigned long |
|  |  | int\_fast64\_t(64b) | signed long | uint\_fast64\_t(64b) | unsigned long |
|  |  | int\_fast16\_t(32b) | signed int | uint\_fast16\_t(32b) | unsigned int |
|  |  | int\_fast32\_t(32b) | signed int | uint\_fast32\_t(32b) | unsigned int |
|  |  | int\_fast64\_t(32b) | signed long long | uint\_fast64\_t(32b) | unsigned long long |
|  |  | intptr\_t(64b) | singed long | uintptr\_t(64b) | unsinged long |
|  |  | intptr\_t(32b) | singed int | uintptr\_t(32b) | unsinged int |
|  |  | intmax\_t(64b) | singed long | uintmax\_t(64b) | unsinged long |
|  |  | intmax\_t(32b) | singed long long | uintmax\_t(32b) | unsinged long long |

## 2格式宏常量

格式宏常量定义于头文件 <inttypes.h>，主要用于printf和scanf输入/输出整数时的格式控制。

### 2.1 fprintf系列函数的格式常量

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| int或unsigned  int的等价者 | 描述 | 数据类型的宏 | | | | |
| [u]intx\_t | [u]int\_leastx\_t | [u]int\_fastx\_t | [u]intmax\_t | [u]intptr\_t |
| x = 8、 16、 32或 64 | | |
| d | 有符号十进制整数值的输出 | PRIdx | PRIdLEASTx | PRIdFASTx | PRIdMAX | PRIdPTR |
| i | PRIix | PRIiLEASTx | PRIiFASTx | PRIiMAX | PRIiPTR |
| u | 无符号十进制整数值的输出 | PRIux | PRIuLEASTx | PRIuFASTx | PRIuMAX | PRIuPTR |
| o | 无符号八进制整数值的输出 | PRIox | PRIoLEASTx | PRIoFASTx | PRIoMAX | PRIoPTR |
| x | 无符号小写十六进制整数值的输出 | PRIxx | PRIxLEASTx | PRIxFASTx | PRIxMAX | PRIxPTR |
| X | 无符号大写十六进制整数值的输出 | PRIXx | PRIXLEASTx | PRIXFASTx | PRIXMAX | PRIXPT |

### 2.2 fscanf系列函数的格式常量

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 与int或unsigned int 的等价者 | 描述 | 数据类型的宏 | | | | |
| [u]intx\_t | [u]int\_leastx\_t | [u]int\_fastx\_t | [u]intmax\_t | [u]intptr\_t |
| x = 8、 16、 32或 64 | | |
| d | 有符号十进制整数值的输入 | SCNdx | SCNdLEASTx | SCNdFASTx | SCNdMAX | SCNdPTR |
| i | 有符号整数值的输入（以首个剖析的字符确定基底） | SCNix | SCNiLEASTx | SCNiFASTx | SCNiMAX | SCNiPTR |
| u | 无符号十进制整数值的输入 | SCNux | SCNuLEASTx | SCNuFASTx | SCNuMAX | SCNuPTR |
| o | 无符号八进制整数值的输入 | SCNox | SCNoLEASTx | SCNoFASTx | SCNoMAX | SCNoPTR |
| x | 无符号十六进制值类型的输入 | SCNxx | SCNxLEASTx | SCNxFASTx | SCNxMAX | SCNxPTR |

## 4 vs\_32位和64位C数据类型大小

|  |  |  |  |
| --- | --- | --- | --- |
| 数据类型 | 大小 | 数据类型 | 大小 |
| int=signed int | 4B | char | 1B |
| unsigned int | 4B | bool | c不支持,c++1B |
| short=short int=signed short=signed short int | 2B | singed char | 1B |
| unsigned short=unsigned short int | 2B | unsigned char | 1B |
| long=long int=signed long=signed long int | 4B | float | 4B |
| unsigned long= unsigned long int | 4B | double | 8B |
| long long =long long int= signed long long =signed long long int | 8B | long double | 8B |
| 指针:32位编译器4B，64位编译器8B |  | sizeof | 32位4B，64位8B |

## 5 GCC\_64位C数据类型及大小

|  |  |  |  |
| --- | --- | --- | --- |
| 数据类型 | 大小 | 数据类型 | 大小 |
| int=signed int | 4B | char | 1B |
| unsigned int | 4B | bool | c不支持,c++1B |
| short=short int=signed short=signed short int | 2B | singed char | 1B |
| unsigned short=unsigned short int | 2B | unsigned char | 1B |
| long=long int=signed long=signed long int | 8B | float | 4B |
| unsigned long= unsigned long int | 8B | double | 8B |
| long long =long long int= signed long long =signed long long int | 8B | long double | 16B |
| 指针:32位编译器4B，64位编译器8B |  | sizeof | 32位4B，64位8B |