Unix环境高级编程笔记

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **I/O** | **文件系统** | **进程** | **线程** | **进程间通信** |
| [一.文件I/O](#ch1)[三.标准I/O库](#ch3)[八.高级I/O](#ch8) | [二.文件和目录](#ch2) | [四.进程环境](#ch4)[五.进程控制](#ch5) | [六.线程](#ch6)[七.线程控制](#ch7) | [九.进程间通信](#ch9)[十.信号](../interview/temp/信号.md) |

目录

[一.文件I/O 1](#_Toc8028874)

[1.文件描述符 1](#_Toc8028875)

[复制文件描述符 1](#_Toc8028876)

[2.相关调用 1](#_Toc8028877)

[2.1 打开文件 1](#_Toc8028878)

[2.2 创建文件 2](#_Toc8028879)

[2.3 关闭文件 3](#_Toc8028880)

[2.4 定位读写位置 3](#_Toc8028881)

[2.5 文件读 3](#_Toc8028882)

[2.6 文件写 4](#_Toc8028883)

[2.7 fcntl函数 4](#_Toc8028884)

[3.进程间文件共享 5](#_Toc8028885)

[4.原子操作 6](#_Toc8028886)

[定位读和定位写 6](#_Toc8028887)

[5.数据同步 6](#_Toc8028888)

[二.文件和目录 7](#_Toc8028889)

[文件系统 7](#_Toc8028890)

[1.获取文件信息 7](#_Toc8028891)

[2.文件类型 8](#_Toc8028892)

[3.用户ID和组ID 8](#_Toc8028893)

[3.1 进程与文件的用户ID及组ID 8](#_Toc8028894)

[3.2 修改文件的所有者和组所有者 9](#_Toc8028895)

[4.文件访问权限 9](#_Toc8028896)

[4.1 进程的文件访问权限 9](#_Toc8028897)

[4.2 新文件的文件访问权限 10](#_Toc8028898)

[4.3 修改文件访问权限 11](#_Toc8028899)

[5.新文件和目录的所有权 11](#_Toc8028900)

[6.粘着位 11](#_Toc8028901)

[7.文件长度 12](#_Toc8028902)

[7.1 文件中的空洞 12](#_Toc8028903)

[7.2 文件截断 12](#_Toc8028904)

[8.硬链接 12](#_Toc8028905)

[8.1 创建硬链接 12](#_Toc8028906)

[8.2 删除硬链接 13](#_Toc8028907)

[9.符号链接 13](#_Toc8028908)

[9.1 创建符号链接 14](#_Toc8028909)

[9.2 读取符号链接 14](#_Toc8028910)

[10.目录 14](#_Toc8028911)

[10.1 创建目录 14](#_Toc8028912)

[10.2 删除目录 14](#_Toc8028913)

[10.3 读目录 14](#_Toc8028914)

[10.4 更改当前目录 15](#_Toc8028915)

[10.5 获取当前目录的绝对路径 15](#_Toc8028916)

[11.重命名 15](#_Toc8028917)

[12.文件的时间 15](#_Toc8028918)

[12.1 更改文件的访问和修改时间 15](#_Toc8028919)

[13.设备特殊文件 16](#_Toc8028920)

[三.标准I/O库 19](#_Toc8028921)

[1.流 19](#_Toc8028922)

[1.1 流的定向 19](#_Toc8028923)

[1.2 3个标准流 20](#_Toc8028924)

[2.FILE对象 20](#_Toc8028925)

[3.缓冲 20](#_Toc8028926)

[3.1 3种缓冲类型 20](#_Toc8028927)

[3.2 缓冲区冲洗 21](#_Toc8028928)

[3.3 标准流与缓冲 21](#_Toc8028929)

[4.相关调用 21](#_Toc8028930)

[4.1 打开流 21](#_Toc8028931)

[4.2 关闭流 22](#_Toc8028932)

[4.3 读写流 22](#_Toc8028933)

[2）读取错误判断 23](#_Toc8028934)

[4.4 定位流 24](#_Toc8028935)

[4.5 格式化I/O 25](#_Toc8028936)

[4.6 获取流相应的文件描述符 26](#_Toc8028937)

[4.7 创建临时文件 26](#_Toc8028938)

[5.内存流 27](#_Toc8028939)

[四.进程环境 28](#_Toc8028940)

[1.进程的启动与终止 28](#_Toc8028941)

[1.1 main函数 28](#_Toc8028942)

[1.2 进程终止的方式 28](#_Toc8028943)

[1.3 终止函数 29](#_Toc8028944)

[1.4 终止状态 29](#_Toc8028945)

[1.5 登记终止处理程序 29](#_Toc8028946)

[2.环境表 29](#_Toc8028947)

[2.1 获取环境变量 30](#_Toc8028948)

[2.2 修改环境变量 30](#_Toc8028949)

[3.C程序的存储空间布局 31](#_Toc8028950)

[4.共享库 33](#_Toc8028951)

[静态链接库与动态链接库 33](#_Toc8028952)

[5.进程堆空间的管理 34](#_Toc8028953)

[1）调整program break来分配与释放 34](#_Toc8028954)

[2）使用C库函数进行分配释放 34](#_Toc8028955)

[3）malloc和free的实现 35](#_Toc8028956)

[6.进程资源限制 36](#_Toc8028957)

[7.Core Dump 37](#_Toc8028958)

[7.1 Core Dump的概念及用途 37](#_Toc8028959)

[7.2 产生Core Dump 37](#_Toc8028960)

[7.3 调试Core Dump 37](#_Toc8028961)

[五.进程控制 38](#_Toc8028962)

[1.进程标识 38](#_Toc8028963)

[1.1 2个特殊进程 38](#_Toc8028964)

[1.2 进程相关的ID获取函数 39](#_Toc8028965)

[2.进程的创建 39](#_Toc8028966)

[2.1 fork 39](#_Toc8028967)

[2）数据共享 39](#_Toc8028968)

[3）fork失败的主要原因 40](#_Toc8028969)

[4）fork的2种用法 40](#_Toc8028970)

[2.2 vfork 40](#_Toc8028971)

[3.进程的终止 41](#_Toc8028972)

[3.1 子进程向父进程传递状态 41](#_Toc8028973)

[3.2 父子进程以不同顺序终止 42](#_Toc8028974)

[4.竞争条件 42](#_Toc8028975)

[5.exec函数 43](#_Toc8028976)

[6.更改用户ID和更改组ID 44](#_Toc8028977)

[UNIX中，特权及访问控制，是基于用户ID和组ID的 44](#_Toc8028978)

[6.1 setuid与setgid 44](#_Toc8028979)

[6.2 setreuid和setregid 44](#_Toc8028980)

[6.3 seteuid和setegid 45](#_Toc8028981)

[7.system函数 45](#_Toc8028982)

[8.用户标识 45](#_Toc8028983)

[9.进程调度 46](#_Toc8028984)

[10.进程时间 46](#_Toc8028985)

[六.线程 47](#_Toc8028986)

[1.相关函数 47](#_Toc8028987)

[1）pthread\_create函数 47](#_Toc8028988)

[2）pthread\_join函数 48](#_Toc8028989)

[3）pthread\_self函数 49](#_Toc8028990)

[4）pthread\_detach函数 49](#_Toc8028991)

[5）pthread\_exit函数 49](#_Toc8028992)

[6）pthread\_equal函数 51](#_Toc8028993)

[7）pthread\_cancel函数 51](#_Toc8028994)

[8）pthread\_cleanup\_push和pthread\_cleanup\_pop函数 51](#_Toc8028995)

[2.线程同步 53](#_Toc8028996)

[2.1 互斥锁 53](#_Toc8028997)

[1）互斥锁的初始化与销毁 53](#_Toc8028998)

[2）互斥锁的加锁与解锁 54](#_Toc8028999)

[3）互斥锁的定时加锁 54](#_Toc8029000)

[Max OS X 10.6.8还没有支持该函数 54](#_Toc8029001)

[2.2 读写锁 54](#_Toc8029002)

[1）读写锁的初始化与销毁 54](#_Toc8029003)

[2）读写锁的加锁与解锁 54](#_Toc8029004)

[3）读写锁的定时加锁 55](#_Toc8029005)

[2.3 条件变量 55](#_Toc8029006)

[1）条件变量的初始化与销毁 55](#_Toc8029007)

[2）等待某个条件变量 55](#_Toc8029008)

[3）通知条件已经满足 55](#_Toc8029009)

[2.4 自旋锁 55](#_Toc8029010)

[1）自旋锁的初始化与销毁 55](#_Toc8029011)

[2）自旋锁的加锁与解锁 55](#_Toc8029012)

[2.5 屏障 56](#_Toc8029013)

[1）屏障的初始化与销毁 56](#_Toc8029014)

[2）增加到达屏障点的线程 56](#_Toc8029015)

[七.线程控制 56](#_Toc8029016)

[1.线程限制 56](#_Toc8029017)

[2.线程属性 56](#_Toc8029018)

[2.1 线程属性 56](#_Toc8029019)

[1）线程属性的初始化和销毁 57](#_Toc8029020)

[2）分离属性的获取与设置 57](#_Toc8029021)

[3）栈属性的获取与设置 58](#_Toc8029022)

[4）栈大小属性的获取与设置 58](#_Toc8029023)

[5）境界缓存大小的获取与设置 58](#_Toc8029024)

[2.2 取消选项 58](#_Toc8029025)

[1）可取消状态属性 58](#_Toc8029026)

[2）可取消类型属性 59](#_Toc8029027)

[3.同步属性 59](#_Toc8029028)

[3.1 互斥锁属性 59](#_Toc8029029)

[1）互斥锁属性的初始化和销毁 59](#_Toc8029030)

[2）进程共享属性的获取和设置 59](#_Toc8029031)

[3）健壮属性的获取和设置 60](#_Toc8029032)

[4）类型属性的获取和设置 60](#_Toc8029033)

[3.2 读写锁属性 61](#_Toc8029034)

[1）读写锁属性的初始化与销毁 61](#_Toc8029035)

[2）进程共享属性的获取与设置 61](#_Toc8029036)

[3.3 条件变量属性 61](#_Toc8029037)

[1）条件变量属性的初始化与销毁 61](#_Toc8029038)

[2）进程共享属性的获取与设置 61](#_Toc8029039)

[3）时钟属性的获取与设置 61](#_Toc8029040)

[3.4 屏障属性 61](#_Toc8029041)

[1）屏障属性的初始化与销毁 61](#_Toc8029042)

[2）进程共享属性的获取与设置 61](#_Toc8029043)

[4.线程特定数据 61](#_Toc8029044)

[1）pthread\_once和pthread\_key\_create函数 62](#_Toc8029045)

[2）pthread\_getspecific和pthread\_setspecific函数 63](#_Toc8029046)

[3）pthread\_key\_delete函数 63](#_Toc8029047)

[5.线程和信号 63](#_Toc8029048)

[5.1 阻止信号发送 63](#_Toc8029049)

[5.2 等待信号 63](#_Toc8029050)

[6.线程和fork 64](#_Toc8029051)

[7.线程和I/O 65](#_Toc8029052)

[八.高级I/O 65](#_Toc8029053)

[2.记录锁 65](#_Toc8029054)

[1）读写锁的关系 66](#_Toc8029055)

[2）锁与文件描述符类型的关系 66](#_Toc8029056)

[3）一般性步骤 67](#_Toc8029057)

[4）相邻区域的组合与分裂 67](#_Toc8029058)

[5）记录锁的隐含继承和释放 67](#_Toc8029059)

[6）文件尾端加锁与解锁 67](#_Toc8029060)

[7）建议性锁和强制性锁 67](#_Toc8029061)

[4.异步I/O 68](#_Toc8029062)

[4.1 AIO控制块 68](#_Toc8029063)

[4.2 异步读与异步写 69](#_Toc8029064)

[4.3 获取异步I/O的状态 69](#_Toc8029065)

[4.4 获取异步I/O返回值 69](#_Toc8029066)

[4.5 阻塞进程到异步I/O完成 70](#_Toc8029067)

[4.6 取消异步I/O 70](#_Toc8029068)

[4.7 批量提交异步I/O请求 70](#_Toc8029069)

[4.8 异步I/O的数量限制 71](#_Toc8029070)

[6.存储映射I/O 71](#_Toc8029071)

[6.1 mmap建立映射 71](#_Toc8029072)

[6.2 mprotect修改映射区权限 73](#_Toc8029073)

[6.3 msync冲洗映射区 73](#_Toc8029074)

[6.4 munmap解除映射 73](#_Toc8029075)

[使用mmap拷贝文件 73](#_Toc8029076)

[九.进程间通信 74](#_Toc8029077)

[1.管道 74](#_Toc8029078)

[1.1 创建管道 75](#_Toc8029079)

[1.2 管道的读写规则 75](#_Toc8029080)

[1.3 标准I/O库管道函数 75](#_Toc8029081)

[2.协同进程 75](#_Toc8029082)

[3.FIFO 76](#_Toc8029083)

[3.1 创建FIFO 76](#_Toc8029084)

[3.2 打开FIFO 76](#_Toc8029085)

[3.3 读写FIFO 76](#_Toc8029086)

[4.XSI IPC 76](#_Toc8029087)

[4.1 标识符和键 77](#_Toc8029088)

[4.2 权限结构 77](#_Toc8029089)

[4.3 结构限制 77](#_Toc8029090)

[4.4 优点和缺点 77](#_Toc8029091)

[5.消息队列 78](#_Toc8029092)

[5.1 与消息队列相关的结构 78](#_Toc8029093)

[5.2 创建或打开消息队列 78](#_Toc8029094)

[5.3 操作消息队列 78](#_Toc8029095)

[5.4 添加消息 79](#_Toc8029096)

[5.5 获取消息 79](#_Toc8029097)

[6.信号量 80](#_Toc8029098)

[6.1 信号量的相关结构 80](#_Toc8029099)

[6.2 获得信号量 80](#_Toc8029100)

[6.3 操作信号量 81](#_Toc8029101)

[7.共享存储 82](#_Toc8029102)

[7.1 共享存储的内核结构 83](#_Toc8029103)

[7.2 创建或获得共享存储 83](#_Toc8029104)

[7.3 操作共享存储 83](#_Toc8029105)

[7.4 与共享存储段连接 84](#_Toc8029106)

[7.5 与共享存储段分离 84](#_Toc8029107)

[进程连接共享存储段的位置 84](#_Toc8029108)

[8.POSIX信号量 85](#_Toc8029109)

[8.1 创建或获取命名信号量 85](#_Toc8029110)

[8.2 关闭释放信号量 86](#_Toc8029111)

[8.3 销毁命名信号量 86](#_Toc8029112)

[8.4 调节信号量的值 86](#_Toc8029113)

[8.5 创建未命名信号量 86](#_Toc8029114)

[8.6 销毁未命名信号量 86](#_Toc8029115)

[8.7 检索未命名信号量的值 86](#_Toc8029116)

# 一.文件I/O

本章描述的函数经常被称为**不带缓冲的I/O**，不带缓冲值的是每个read和write都调用内核中的一个系统调用。这些不带缓冲的I/O函数不是ISO C的组成部分，它们是POSIX.1和Single UNIX Specification的组成部分

## 1.文件描述符

对于内核而言，所有打开的文件都通过文件描述符引用。文件描述符是一个非负数。范围是0~OPEN\_MAX-1。当进程创建时，默认为它打开了3个文件描述符，它们都链接向终端：

* 0： 标准输入
* 1： 标准输出
* 2： 标准错误输出

通常使用STDIN\_FILENO，STDOUT\_FILENO和 STDERR\_FILENO来替代这三个幻数，从而提高可读性。这三个常量位于<unistd.h>中

### 复制文件描述符

* 对于dup函数
  + 返回的新的文件描述符一定是当前可用的文件描述符中最小的数字
* 对于dup2函数：
  + 如果 fd2已经是被打开的文件描述符且不等于fd，则先将其关闭，然后再打开（注意关闭再打开是一个原子操作）
  + 如果 fd2等于fd，则直接返回fd2（也等于fd），而不作任何操作

这些函数返回的新文件描述符与参数fd共享同一个文件表项：

## 2.相关调用

### 2.1 打开文件

* 参数：
  + path:要打开或者创建文件的名字
  + oflag：用于指定函数的操作行为：
    - O\_RDONLY常量：文件只读打开
    - O\_WRONLY常量：文件只写打开
    - O\_RDWR常量：文件读、写打开
    - O\_EXEC常量：只执行打开
    - O\_SEARCH常量：只搜索打开（应用于目录）。本书涉及的操作系统都没有支持该常量
  + **以上五个常量必须且只能指定一个。下面的常量是可选的（进行或运算）**：
    - O\_APPEND：每次写时都追加到文件的尾端
    - O\_CLOEXEC：将FD\_CLOEXEC常量设置为文件描述符标志
    - O\_CREAT：若文件不存在则创建。使用此选项时，需要同时说明参数mode（指定该文件的访问权限）
    - O\_DIRECTORY：若path引用的不是目录，则出错
    - O\_EXCL：若同时指定了O\_CREAT时，且文件已存在则出错。根据此可以测试一个文件是否存在。若不存在则创建此文件。这使得测试和创建两者成为一个原子操作
    - O\_NOCTTY：若path引用的是终端设备，则不将该设备分配作为此进程的控制终端
    - O\_NOFOLLOW：若path引用的是一个符号链接，则出错
    - O\_NONBLOCK：如果path引用的是一个FIFO、一个块特殊文件或者一个字符特殊文件，则文件本次打开操作和后续的 I/O 操作设为非阻塞模式
    - O\_SYNC：每次 write 等待物理 I/O 完成，包括由 write 操作引起的文件属性更新所需的 I/O
    - O\_TRUNC： 如果此文件存在，且为O\_WRONLY或者O\_RDWR成功打开，则将其长度截断为0
    - O\_RSYNC：使每一个read操作等待，直到所有对文件同一部分挂起的写操作都完成
    - O\_DSYNC：每次 write 等待物理 I/O 完成，但不包括由 write 操作引起的文件属性更新所需的 I/O
  + mode：文件访问权限。文件访问权限常量在 <sys/stat.h> 中定义，有下列九个：
    - S\_IRUSR：用户读
    - S\_IWUSR：用户写
    - S\_IXUSR：用户执行
    - S\_IRGRP：组读
    - S\_IWGRP：组写
    - S\_IXGRP：组执行
    - S\_IROTH：其他读
    - S\_IWOTH：其他写
    - S\_IXOTH：其他执行

对于openat函数，被打开的文件名由fd和path共同决定：

* 如果path指定的是绝对路径，此时fd被忽略。openat等价于open
* 如果path指定的是相对路径名，则fd是一个目录的文件描述符。被打开的文件的绝对路径由该fd描述符对应的目录加上path组合而成
* 如果path是一个相对路径名，而fd是常量AT\_FDCWD，则path相对于当前工作目录。被打开文件在当前工作目录中查找。

由 open/openat 返回的文件描述符一定是**最小的未使用**的描述符数字

### 2.2 创建文件

* 参数：
  + path:要创建文件的名字
  + mode：指定该文件的访问权限。文件访问权限常量在 <sys/stat.h> 中定义，有下列九个：
    - S\_IRUSR：用户读
    - S\_IWUSR：用户写
    - S\_IXUSR：用户执行
    - S\_IRGRP：组读
    - S\_IWGRP：组写
    - S\_IXGRP：组执行
    - S\_IROTH：其他读
    - S\_IWOTH：其他写
    - S\_IXOTH：其他执行

该函数等价于open(path,O\_WRONLY|O\_CREAT|O\_TRUNC,mode)

creat的存在一个不足是：**它以只写方式打开创建的文件**。如果要创建一个临时文件，并要先写该文件，然后又读该文件，则必须先调用create、close，然后再调用open，新版本open出来后，可以以下列方式实现：

open(path,O\_RDWR|O\_CREAT|O\_TRUNC,mode)

### 2.3 关闭文件

注意：

* 进程关闭一个文件会释放它加在该文件上的所有记录锁
* 当一个进程终止时，内核会自动关闭它所有的打开的文件

### 2.4 定位读写位置

* 参数：
  + fd：打开的文件的文件描述符
  + whence：必须是 SEEK\_SET、SEEK\_CUR、SEEK\_END三个常量之一
  + offset：
    - 如果 whence是SEEK\_SET，则将该文件的偏移量设置为距离文件开始处offset个字节
    - 如果 whence 是 SEEK\_CUR，则将该文件的偏移量设置为当前值加上offset个字节，offset可正，可负
    - 如果 whence 是 SEEK\_END，则将该文件的偏移量设置为文件长度加上offset个字节，offset可正，可负

每个打开的文件都有一个与其关联的“当前文件偏移量”。它通常是个非负整数，用于度量从文件开始处计算的字节数。通常读、写操作都从当前文件偏移量处开始，并且使偏移量增加所读写的字节数。注意：

* 打开一个文件时，除非指定O\_APPEND选项，否则系统默认将该偏移量设为0
* 如果文件描述符指定的是一个管道、FIFO、或者网络套接字，则无法设定当前文件偏移量，则lseek将返回 -1 ，并且将 errno 设置为 ESPIPE
* 对于普通文件，其当前文件偏移量必须是非负值。但是某些设备运行负的偏移量出现。因此比较lseek的结果时，不能根据它小于0 就认为出错。要根据是否等于 -1 来判断是否出错
* **lseek 并不会引起任何 I/O 操作，lseek仅仅将当前文件的偏移量记录在内核中**
* **当前文件偏移量可以大于文件的当前长度。此时对该文件的下一次写操作将家常该文件，并且在文件中构成一个空洞。空洞中的内容位于文件中但是没有被写过，其字节被读取时都被读为0（文件中的空洞并不要求在磁盘上占据存储区。具体处理方式与操作系统有关）**

### 2.5 文件读

* 参数：
  + fd：打开的文件的文件描述符
  + buf：存放读取内容的缓冲区的地址（由程序员手动分配）
  + nbytes：期望读到的字节数

读操作从文件的当前偏移量开始，在成功返回之前，文件的当前偏移量会增加实际读到的字节数

有多种情况可能导致实际读到的字节数少于期望读到的字节数：

* 读普通文件时，在读到期望字节数之前到达了文件尾端
* 当从终端设备读时，通常一次最多读取一行（终端默认是行缓冲的）
* 当从网络读时，网络中的缓存机制可能造成返回值小于期望读到的字节数
* 当从管道或者FIFO读时，若管道包含的字节少于所需的数量，则 read只返回实际可用的字节数
* 当从某些面向记录的设备（如磁带）中读取时，一次最多返回一条记录
* 当一个信号造成中断，而已读了部分数据时

### 2.6 文件写

* 参数：
  + fd：打开的文件的文件描述符
  + buf：存放待写的数据内容的缓冲区的地址（由程序员手动分配）
  + nbytes：期望写入文件的字节数

write的返回值通常都是与nbytes相同。否则表示出错（出错的一个常见原因是磁盘写满，或者超过了一个给定进行的文件长度限制）

对于普通文件，写操作从文件的当前偏移量处开始。如果打开文件时指定了O\_APPEND选项，则每次写操作之前，都会将文件偏移量设置在文件的当前结尾处。在一次成功写之后，该文件偏移量增加实际写的字节数

### 2.7 fcntl函数

fcntl可以改变已经打开文件的属性

* 参数：
  + fd：已打开文件的描述符
  + cmd：有下列若干种：
    - **复制一个已有的描述符**（还可以使用dup或dup2）
      * F\_DUPFD：复制文件描述符 fd。新文件描述符作为函数值返回。它是尚未打开的文件描述符中大于或等于arg中的最小值。新文件描述符与fd共享同一个文件表项，但是新描述符有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
      * F\_DUPFD\_CLOEXEC：复制文件描述符。新文件描述符作为函数值返回。它是尚未打开的个描述符中大于或等于arg中的最小值。新文件描述符与fd共享同一个文件表项，但是新描述符有自己的一套文件描述符标志，其中FD\_CLOEXEC文件描述符标志被设置
    - **获取/设置文件描述符标志**
      * F\_GETFD：对应于fd的文件描述符标志作为函数值返回。当前只定义了一个文件描述符标志FD\_CLOEXEC
      * F\_SETFD：设置fd的文件描述符标志为arg
    - **获取/设置文件状态标志**
      * F\_GETFL：返回fd的文件状态标志。获得文件状态标志后，必须首先用屏蔽字 O\_ACCMODE 取得访问方式位，然后与O\_RDONLY、O\_WRONLY、O\_RDWR、O\_EXEC、O\_SEARCH比较（这5个值互斥，且并不是各占1位）。剩下的还有：O\_APPEND、O\_NONBLOCK、O\_SYNC 、O\_DSYNC、O\_RSYNC、F\_ASYNC、O\_ASYNC
      * F\_SETFL：设置fd的文件状态标志为 arg。可以更改的标志是： O\_APPEND、O\_NONBLOCK、O\_SYNC、O\_DSYNC、O\_RSYNC、F\_ASYNC、O\_ASYNC
    - **获取/设置异步I/O所有权**
      * F\_GETOWN：获取当前接收 SIGIO和SIGURG信号的进程 ID或者进程组 ID
      * F\_SETOWN：设置当前接收 SIGIO和SIGURG信号的进程 ID或者进程组 ID为arg。若 arg是个正值，则设定进程 ID；若 arg是个负值，则设定进程组ID
    - **获取/设置记录锁**
      * F\_GETLK、F\_SETLK、F\_SETLKW
  + arg：依赖于具体的命令

## 3.进程间文件共享

UNIX系统支持在不同进程间共享打开文件。内核使用3种数据结构描述打开文件。它们之间的关系决定了在文件共享方面一个进程对另一个进程可能产生的影响

* 内核为每个进程分配一个**进程表项**（所有进程表项构成进程表），进程表项中都有一个打开的文件描述符表。每个文件描述符占用一项，其内容为：
  + 文件描述符标志
  + 指向一个文件表项的指针
* 内核为每个打开的文件分配一个**文件表项**（所有的文件表项构成文件表）。每个文件表项的内容包括：
  + 文件状态标志（读、写、添写、同步和阻塞等）
  + 当前文件偏移量
  + 指向该文件 v 结点表项的指针
* 每个打开的文件或者设备都有一个 **v 结点结构**。 v 结点结构的内容包括：
  + 文件类型和对此文件进行各种操作函数的指针
  + 对于大多数文件， v 结点还包含了该文件的 i 结点 **这些信息都是在打开文件时从磁盘读入内存的**。如 i 结点包含了文件的所有者、文件长度、指向文件实际数据在磁盘上所在位置的指针等等。 v 结点结构和 i 结点结构实际上代表了文件的实体（**Linux没有使用v节点，而是使用了通用i节点结构。虽然两种实现有所不同，但在概念上，v节点与i节点是一样的，两者都指向文件系统特有的i节点结构**）

现在假设进程 A 打开文件 file1，返回文件描述符 3；进程 B 也打开文件 file1，返回文件描述符 4：

内核在文件表上新增两个表项：

* 这两个文件表项指向同一个 v 结点表项
* 进程 A 、B 各自的文件描述符表项分别指向这两个文件表项（因此每个进程都有自己的对该文件的当前偏移）

对文件的操作结果：

* 每次 write 之后，在文件表项中的当前文件偏移量即增加所写入的字节数
  + 若这导致当前文件偏移量超过当前文件长度，则修改 i 节点的当前文件长度，设为当前文件偏移量
* 如果用 O\_APPEND 标志打开一个文件，则相应标志也设置到文件表项的文件状态标志中
  + 每次对具有追加写标志的文件执行写操作时，文件表项中的当前文件偏移量首先被置为 i 结点中的文件长度
* 若用 lseek 定位到文件当前的尾端，则文件表项中的当前文件偏移量设置为 i 结点中的文件长度
  + lseek 函数只是修改文件表项中的当前文件偏移量，不进行任何 I/O 操作

## 4.原子操作

多个进程写同一文件时，可能产生预想不到的结果。为了避免这种情况，需要理解原子操作

多个进程同时追加写一个文件时，如果通过 lseek 到末尾然后 write 可能出现问题：因为 “先定位到文件尾端，然后写” 并不是一个原子操作，如果先执行两个进程的 lseek ，然后执行 write ，就会出现问题

UNIX提供了一种原子操作方法：通过 O\_APPEND选项打开文件。这样做使得内核在每次调用 write 执行写操作之前，都将进程的当前偏移量设置到该文件的末尾，于是就不需要执行lseek定位操作

### 定位读和定位写

* 参数：
  + fd：打开的文件描述符
  + buf：读出数据存放的缓冲区/ 写到文件的数据的缓冲区
  + nbytes：预期读出/写入文件的字节数
  + offset：从文件指定偏移量开始执行read/write

调用pread相当于先调用lseek再调用read.但是调用pread时，无法中断其定位和读操作，并且不更新当前文件偏移量

调用pwrite相当于先调用lseek再调用write.但是调用pwrite时，无法中断其定位和写操作，并且不更新当前文件偏移量

## 5.数据同步

UNIX操作系统在内核中设有**缓冲区高速缓存**或**页高速缓存**，大多数磁盘 I/O 都通过缓冲区进行。当我们想文件写入数据时，内核通常都先将数据复制到缓冲区中，然后排入队列，晚些时候再写入磁盘。这种方式称为**延迟写**

以下2种情况会将缓冲区中的数据写回到磁盘：

* 当内核需要重用缓冲区来存放其他数据时，它会把所有延迟写的数据库写入磁盘
* 可以调用sync、fsync或fdatasync来显式的将所有延迟写的数据块写回磁盘
* 参数（前两个函数）：
  + fd：指定写回的文件

3个函数的区别：

* sync（update 守护进程会周期性(一般每隔30s)的调用sync函数。命令sync也会调用sync函数）：
  + 将所有修改过的块缓冲区排入写队列，然后返回
  + 它并不等待实际写磁盘操作结束
* fsync：
  + 只对由fd指定的单个文件起作用
  + 等待写磁盘操作结束才返回
* fdatasync：
  + 只对由fd指定的单个文件起作用，但是它只影响文件的数据部分（fsync会同时更新文件的属性）
  + 等待写磁盘操作结束才返回

# 二.文件和目录

## 文件系统

可以把一个磁盘分成一个或多个分区。每个分区可用包含一个文件系统

inode是固定长度的记录项，包含有关文件的大部分信息

可以进一步分析inode和数据块部分：

* 每个inode中有一个链接数，其值为指向该inode的目录项数（上图中有两个目录项指向同一个inode）。只有当链接数减为0时，才删除该文件。链接数包含在stat结构的st\_nlink成员中（POSIX.1常量LINK\_MAX指定了一个文件链接数的最大值）。**这种链接类型称为硬链接**
* 另外一种链接类型称为**符号链接**。符号链接文件的实际内容（在数据块中）包含了该符号链接所指向的文件的名字
* inode包含了文件有关的所有信息：文件类型、文件访问权限位、文件长度和指向文件数据块的指针等。stat结构中的大多数信息都取自inode。只有2项重要数据存放在目录项中：文件名和inode号
* 因为目录项中的inode编号指向同一文件系统中的相应inode，一个目录项不能指向另一个文件系统的inode
* 在不更换文件系统的情况下为一个文件重命名时，该文件的实际内容并未移动，只需构造一个指向现有inode的新目录项，并删除老的目录项。链接数不会改变

下图为在一个目录下创建一个目录testdir，注意testdir所在目录，以及新建目录testdir的inode链接数：

## 1.获取文件信息

* **stat**：获得与pathname文件有关的信息结构，存在buf中
* **fstat**：获得已在描述符fd上打开文件的有关信息，存在buf中
* **lstat**：类似于**stat**。但是当pathname为一个符号链接时，返回符号链接（而不是由该符号链接引用的文件）的有关信息，存在buf中
* **fstatat**：获取相对于打开目录fd的文件pathname的统计信息
  + 如果fd的值是AT\_FDCWD，则会获取相对于当前目录的文件pathname的信息
  + 如果pathname是一个绝对路径，参数fd就会被忽略
  + flag
    - 设置了AT\_SYMLINK\_NOFOLLOW标志：如果文件是符号链接则直接获取符号链接的信息
    - 否则：如果文件是符号链接，会顺着链接获取链接文件的信息

文件信息结构用一个结构体stat表示，实际定义可能随具体实现有所不同，但基本形式是：

struct stat{  
 mode\_t st\_mode; /\*文件模式字，包含有文件类型、ID和读写权限位信息\*/  
 ino\_t st\_ino; /\* inode号 \*/  
 dev\_t st\_dev; /\* 设备号（文件系统） \*/  
 dev\_t st\_rdev; /\* 特殊文件的设备号 \*/  
 nlink\_t st\_nlink; /\* 链接数 \*/  
 uid\_t st\_uid; /\* 所有者的用户ID \*/  
 gid\_t st\_gid; /\* 组所有者的ID \*/  
 off\_t st\_size; /\* 字节大小，用于一般文件 \*/  
 struct timespec st\_atime; /\* 最后一次访问时间 \*/  
 struct timespec st\_mtime; /\* 最后一次修改时间 \*/  
 struct timespec st\_ctime; /\* 最后一个文件状态改变的时间 \*/  
 blksize\_t st\_blksize; /\* 磁盘块(block)大小 \*/  
 blkcnt\_t st\_blocks; /\* 分配的磁盘块(block)数量 \*/  
};

POSIX.1未要求st\_rdev、st\_blksize和st\_blocks字段。Single UNIX Specification XSI扩展定义了这些字段

timespec结构类型按照**秒**和**纳秒**定义了时间，至少包括下面2个字段：

time\_t tv\_sec;  
long tv\_nsec;

使用stat函数最多的地方可能就是ls -l命令获得有关一个文件的所有信息

## 2.文件类型

* **普通(一般)文件**
* **目录文件**
* **块特殊文件**：这种类型的文件提供对设备（如磁盘）带缓冲的访问，每次访问以固定长度为单位进行
* **字符特殊文件**：这种类型的文件提供对设备不带缓冲的访问，每次访问长度可变（系统中的所有设备要么是字符特殊文件，要么是块特殊文件）
* **FIFO**：这种类型的文件用于进程间通信，有时也称为命名管道
* **套接字**：这种类型的文件用于进程间的网络通信
* **符号链接**：这种类型的文件指向另一个文件

文件类型信息包含在stat结构的st\_mode成员中。可以使用下图的宏判断文件类型（传入st\_mode）：

POSIX.1允许实现将进程间通信(IPC)对象说明为文件。通过下图的宏可以确定IPC对象的类型（传入stat结构的指针）

## 3.用户ID和组ID

### 3.1 进程与文件的用户ID及组ID

与一个**进程**相关联的ID有6个或更多，下图是用户ID和组ID

* **实际用户ID和组ID**：标识我们究竟是谁。这两个字段在登录时取自口令文件中的登录项。通常在一个登录会话期间这些值并不改变，但是超级用户进程有办法改变它们
* **有效用户ID、有效组ID以及附属组ID**：决定了我们的**文件访问权限**
  + 通常情况下**有效用户ID**等于**实际用户ID**、**有效组ID**等于**实际组ID**
  + 每个文件有一个**所有者**（stat结构中的st\_uid）和**组所有者**（stat结构中的st\_gid），可以设置程序文件的**文件模式字**（stat结构中的st\_mode）来改变执行该文件的进程的**有效用户ID**和**有效组ID**
    - 如果设置了文件模式字中的**设置用户ID位**，当执行此文件时，会将**进程的有效用户ID**设置为**文件所有者的用户ID**
    - 如果设置了文件模式字中的**设置组ID位**，”当执行此文件时，会将**进程的有效组ID**设置为**文件组所有者的ID**
    - 可以通过将文件stat结构的st\_mode传入常量S\_ISUID和S\_ISGID来测试是否设置了**设置用户ID位**和**设置组ID位**（**因此，如果文件所有者是超级用户，而且设置了该文件的设置用户ID位，那么当该程序文件由一个进程执行时，该进程具有超级用户权限。不管执行此文件的进程的实际用户ID是什么，都会是这样**）
* **保存的设置用户ID和保存的设置组ID**：在执行一个程序时包含了有效用户ID和有效组ID的副本

### 3.2 修改文件的所有者和组所有者

如果owner或group中的任意一个是-1，则对应的ID不变：

* **chown函数**
* **fchown函数**：更改fd参数指向的打开文件的用户ID和组ID（既然在一个已打开的文件上操作，就不能用于改变符号链接的所有者）
* **fchownat函数**：更改相对于已打开目录fd的文件pathname的用户ID和组ID
  + 函数在下列2种情况下，与**chown**或**lchown**相同
    - pathname为绝对路径
    - fd参数为AT\_FDCWD而pathname为相对路径
  + flag
    - 设置了AT\_SYMLINK\_NOFOLLOW标志：不会跟随符号链接，与**lchown**行为相同
    - 没有设置AT\_SYMLINK\_NOFOLLOW标志：与**chown**相同
* **lchown函数**：直接对符号链接进行更改

基于BSD的系统一直规定只有超级用户才能更改一个文件的所有者；System V则允许任一用户更改他们所拥有的文件的所有者

能否更改还与\_POSIX\_CHOWN\_RESTRICTED常量是否对文件生效有关

## 4.文件访问权限

下图列出了文件访问权限的所有权限位，以及它们对普通文件和目录文件的作用：

最后9个常量还可以分成如下3组：

* S\_IRWXU = S\_IRUSR | S\_IWUSR | S\_IXUSR
* S\_IRWXG = S\_IRGRP | S\_IWGRP | S\_IXGRP
* S\_IRWXO = S\_IROTH | S\_IWOTH | S\_IXOTH

### 4.1 进程的文件访问权限

文件stat结构的st\_mode成员除了包含**设置用户ID位**和**设置组ID位**用以修改进程的**有效用户ID**和**有效组ID**，还包含9个文件访问权限的位，**这些权限位指明了所有者、组和其他对于该文件的访问权限**，可分为3类：

任何文件类型都有访问权限

* **目录**
  + **执行权限**：决定了我们能否通过该目录（即如果某个文件的路径包含该目录，那么要打开这个文件，就需要对这个目录具有执行权限）
  + **读权限**：允许读目录，获得在目录中所有文件名的列表
* **文件**
  + **读权限**：决定了我们能否打开现有文件进行读操作
  + **写权限**：决定了我们能否打开现有文件进程写操作

**创建和删除文件**：必须对该文件所在的目录具有**写权限**和**执行权限**

**进程每次打开、创建或删除一个文件时，内核就进行文件访问权限测试**，内核进行的测试如下：

1. **若进程的有效用户ID是0**（超级用户），则允许访问。这给予了超级用户对整个文件系统进行处理的最充分自由
2. **若进程的有效用户ID等于文件的所有者ID**（也就是进程拥有此文件），那么如果所有者适当的访问权限位被设置，则允许访问，否则拒绝访问
3. **若进程的有效组ID或进程的附属组ID之一等于文件的组ID**，那么如果组适当的访问权限位被设置，则允许访问，否则拒绝访问
4. 若其他用户适当的访问权限位被设置，则允许访问，否则拒绝访问

**按顺序**执行这4步

* 若进程拥有此文件（第2步），则按用户访问权限批准或解决该进程对文件的访问——不查看组访问权限
* 若进程不拥有该文件。但进程属于某个适当的组，则按组访问权限批准或拒绝该进程对文件的访问——不查看其他用户的访问权限

如果进程希望按照其**实际用户ID**和**实际组ID**来测试其访问能力，那么可以使用下列函数（进程使用**实际用户ID**和**实际组ID**来测试其访问能力也分成4步，与上面相同）：

* mode：欲测试的权限位标志（可以是下列值按位或）
  + F\_OK：测试文件是否已经存在
  + R\_OK：测试读权限
  + W\_OK：测试写权限
  + X\_OK：测试执行权限
* flag参数可以用于改变faccessat的行为，如果flag设置为AT\_EACCESS，访问检查用的是调用进程的**有效用户ID和有效组ID**，而不是**实际用户ID和实际组ID**

这两个函数在下面2种情况下是相同的：

* pathname参数为绝对路径
* fd参数取值为AT\_FDCWD，而pathname参数为相对路径

### 4.2 新文件的文件访问权限

可以使用umask函数为进程设置**文件模式创建屏蔽字**，并返回之前的值：

* cmask：同[open函数的mode参数](#X4f892631062a259c02243e0c580974547dd1932)，是9个常量的若干个按位”或“的值

**在进程创建一个新文件或新目录时，一定会使用文件模式创建屏蔽字。在文件模式创建屏蔽字为1的位，在文件mode中的相应位一定被关闭。因此，使用creat创建文件时，指定的mode，屏蔽掉文件模式创建屏蔽字中为1的位，就是新创建文件的权限**

**子进程的文件模式创建屏蔽字不会影响父进程。因此，如果子进程修改了文件模式创建屏蔽字，父进程的文件模式创建屏蔽字不会改变（如shell在调用一个会修改文件创建屏蔽字的程序前后使用umask打印文件创建屏蔽字，不会发生改变）**

下图为八进制的屏蔽位和含义：

### 4.3 修改文件访问权限

* **chmod函数**：对文件pathname进行修改
* **fchmod函数**：对已打开的文件(fd指定)进行修改
* **fchmodat函数**：对已打开目录fd下的pathname文件进行修改
  + 下列情况下等同于**chmod函数**
    - pathname为绝对路径
    - fd参数为AT\_FDCWD而pathname为相对路径
  + flag设置了AT\_SYMLINK\_NOFOLLOW标志时，不会跟随符号链接
* mode参数是下图常量（定义在<sys/stat.h>中）的按位或（图中进位的9个常量是open函数mode参数使用的9个常量）

**修改文件的权限位的前提是：进程的有效用户ID必须等于文件的所有者ID，或者进程必须具有超级用户权限**

ls -l列出的是文件内容最后修改的时间，这里介绍的函数修改文件权限是在修改inode，所以ls -l列出的时间不会发生变化

## 5.新文件和目录的所有权

* 新文件或新目录的**用户ID**设置为**进程的有效用户ID**
* 关于**组ID**，POSIX.1允许实现选择下列之一作为新文件或新目录的组ID
  + 新文件或新目录的组ID可以是**进程的有效组ID**
  + 新文件或新目录的组ID可以是它**所在目录的组ID**（Linux的/var/mail目录中就使用了这种方法）

## 6.粘着位

即S\_ISVTX

* 在UNIX尚未使用请求分页式技术的早期版本中，S\_ISVTX位被称为**粘着位**
* 后来的UNIX版本称它为**保存正文位**

用途

* 以前，如果一个可执行文件设置了该位，当程序第一次被执行，在其终止时，程序正文部分的一个副本仍被保存在交换区。这使得下次执行该程序时能较快地将其装载入内存（原因是：通常的UNIX文件系统中，文件的各数据块很可能是随机存放的，相比较而言，交换区是被作为一个连续文件来处理的）
* 现在，系统**扩展了**粘着位的**使用范围**，Single UNIX Specification**允许针对目录设置粘着位**。**如果对一个目录设置了该位，只有满足下列2个情况，才能删除或重命名该目录下的文件**：
  + **对该目录具有写权限**
  + **满足下列条件之一**
    - 拥有此文件
    - 拥有此目录
    - 是超级用户

目录/tmp和/var/tmp就是设置粘着位的典型候选者

## 7.文件长度

stat结构的st\_size成员表示以字节为单位的文件的长度。该字段只对**普通文件**、**目录文件**和**符号链接**有意义

* 对于**普通文件**，其文件长度可以是0，在读这种文件时，将得到文件结束指示
* 对于**目录**，文件长度通常是一个数（如16或512）的整倍数
* 对于**符号链接**，文件长度是文件名中的实际字节数（因为符号链接文件长度总是由st\_size指示，所以它并不包含通常C语言用作名字结尾的null字节）

### 7.1 文件中的空洞

普通文件可以包含空洞。**空洞是由所设置的偏移量超过文件尾端，并写入了某些数据后造成的**

空洞并不占用实际磁盘空间，但是会参与文件大小的计算，在读文件时，读到空洞部分的内容是字节0

### 7.2 文件截断

可以在打开文件时使用O\_TRUNC标志将文件截断为0

下列函数可以截断文件：

* length
  + <文件长度：文件超过length以外的数据就不再能访问
  + >文件长度：文件长度将增加，在以前的文件尾端和新的文件尾端之间的数据将读作0（也就是可能在文件中创建了一个空洞）

## 8.硬链接

### 8.1 创建硬链接

* **link函数**
  + existingpath：现有文件
  + newpath：创建的新目录项（如果已经存在，则返回出错）。只创建最后一个分量，路径中的其他部分应当已经存在
* **linkat函数**
  + 现有文件
    - efd：现有文件所在目录的文件描述符
    - existingpath：现有文件名
  + 新的路径名
    - nfd：新目录项所在目录的文件描述符
    - newpath：新目录项的文件名
  + flag
    - 设置了AT\_SYMLINK\_NOFOLLOW标志：创建指向符号链接目标的硬链接
    - 没设置AT\_SYMLINK\_NOFOLLOW标志：创建指向符号链接本身的硬链接

对于**linkat函数**：

* 如果两个路径名中的任一个是相对路径，那么需要通过相对于对应的文件描述符进行计算
  + 如果两个文件描述符中的任一个设置为AT\_FDCWD，那么相应的路径名就通过当前目录进行计算
* 如果两个路径名中的任一个是绝对路径，那么相应的文件描述符参数就会被忽略

如果实现支持创建指向一个目录的硬链接，那么也仅限于超级用户才可以这样做

### 8.2 删除硬链接

要删除硬链接，必须对包含该目录项(硬链接)的目录具有**写和执行权限**

* 如果flag参数设置了AT\_REMOVEDIR标志位，那么**unlinkat**函数可以类似于**rmdir**一样删除目录；否则，函数**unlinkat**与**unlink**执行同样的操作
* **如果pathname是符号链接，那么unlink删除该符号链接，而不是删除由该链接所引用的文件**（即，不跟随符号链接）（给出符号链接名的情况下，没有一个函数能删除由该链接所引用的文件）
* 如果文件系统支持的话，超级用户可以调用**unlink**，其参数pathname指定一个目录，但是通常应当使用**rmdir**函数

这两个函数会删除目录项，并将由pathname所引用文件的链接计数减1

* 如果对该文件还有其它链接，则仍可通过其它链接访问该文件的数据
* 如果出错，则不对该文件做任何更改

**何时删除实际文件？**

* 当链接计数到达0时
* 并且没有进程打开该文件（如果有进程打开了该文件，那么会删除目录项，但是进程退出前不会删除实际文件）

关闭一个文件时，内核首先检查打开该文件的进程个数。如果这个数达到0，内核再去检查其链接数；如果链接数，如果也是0，那么就删除该文件的内容

**unlink的这种特性经常被程序用来确保即使是在程序崩溃时，它所创建的临时文件也不会遗留下来。进程用open或creat创建一个文件，然后立即调用unlink，因为该文件仍旧是打开的，所以不会将其内容删除。只有当进程关闭该文件或终止时，该文件的内容才被删除**

也可以用remove函数（c库函数）解除对一个文件或目录的链接：

* 对于文件，remove的功能与unlink相同
* 对于目录，remove的功能与rmdir相同

## 9.符号链接

符号链接是对一个文件的间接指针。与硬链接直接指向文件的inode不同，符号链接是为了避开硬链接的一些限制：

* 硬链接通常要求链接和文件位于同一文件系统中
* 只有超级用户才能创建指向目录的硬链接（在底层文件系统支持的情况下）

**对于符号链接以及它指向何种对象并无任何文件系统限制**（符号链接一般用于将一个文件或整个目录结构移到系统中另一个位置）

下图指明当传入的文件名表示一个符号链接时，函数能否跟随符号链接处理其链接的目标文件：

上图没有mkdir、mkinfo、mknod和rmdir，因为当这些函数的路径名是符号链接时，函数出错返回

用open打开文件时，如果传递给open函数的路径名指定了一个符号链接，那么open跟随此符号链接到达所指定的文件。若此符号链接所指向的文件并不存在，则open返回出错，表示它不能打开该文件

### 9.1 创建符号链接

* **symlink函数**：创建一个指向actualpath的新目录项sympath。在创建此符号链接时，并不要求actualpath已经存在。并且，actualpath和sympath并不需要位于同一文件系统中
* **symlinkat函数**：类似。但sympath参数根据相对于打开文件描述符fd引用的目录进行计算。如果sympath参数指定的是绝对路径或者fd参数设置了AT\_FDCWD值，那么该函数就等同于**symlink**

### 9.2 读取符号链接

因为open函数跟随符号链接，所以需要一种方法打开该链接本身，并读该链接中的名字。下列函数提供了这种功能：

* 当pathname参数指定的是绝对路径名或者fd参数的值为AT\_FDCWD，**readlinkat**函数的行为与**readlink**相同。但是，如果fd参数是一个打开目录的有效文件描述符并且pathname参数是相对路径，则**readlinkat**计算相对于由fd代表的打开目录的路径名

两个函数组合了**open、read和close**的所有操作。如果函数成功执行，则返回读入buf的字节数。**在buf中返回的符号链接的内容不以null字节终止**

## 10.目录

### 10.1 创建目录

* **mkdirat函数**与**mkdir函数**类似，当fd参数具有特殊值AT\_FDCWD或者pathname参数指定了绝对路径名时，两个函数完全一样。否则，fd参数是一个打开的目录，相对路径名根据此打开目录进行计算

对于目录通常至少要设置一个执行权限位，以允许访问该目录中的文件名

### 10.2 删除目录

* 如果调用此函数使目录的链接数成为0，并且也没有其它进程打开此目录，则释放由此目录占用的空间
* 如果在链接数到达0时，有1个或多个进程打开此目录，则在此函数返回前删除最后一个链接及.和..项。另外，在此目录中不能再创建新文件。但是在最后一个进程关闭它之前并不释放此目录

### 10.3 读目录

对某个目录具有访问权限的任一用户都可以读该目录，但是，为了防止文件系统产生混乱，只有内核才能写目录

* **fdopendir函数**：打开fd表示的目录，得到相应的DIR结构
* **telldir函数**和**seekdir函数**不是基本POSIX.1标准的组成部分。它们是Single UNIX Specification中的XSI扩展。所以可以期望所有符合UNIX系统的实现都会提供这2个函数

结构dirent定义在<dirent.h>中，具体实现相关，至少包含下列2个成员：

struct dirent{  
 ino\_t d\_ino; /\* inode号 \*/  
 char d\_name[]; /\* null-terminated文件名 \*/  
 ...  
};

* d\_name的大小并没有指定，但是必须保证至少包含NAME\_MAX个字节（不包含终止null字节）

DIR结构是一个内部结构，用以保存当前正在被读的目录的有关信息。作用类似于FILE结构

### 10.4 更改当前目录

每个进程都有一个当前工作目录，此目录是搜索所有相对路径名的起点（不以斜线开始的路径名为相对路径名）

用户登录到UNIX系统时，其当前工作目录通常是口令文件（/etc/passwd）中该用户登录项的第6个字段——用户的起始目录

* 当前工作目录是进程的一个属性（所以如果调用chdir修改当前目录，只影响调用函数的进程本身）
* 起始目录则是登录名的一个属性

下列2个函数可以更改当前工作目录：

这两个函数分别用pathname或打开文件描述符fd来指定新的当前工作目录

### 10.5 获取当前目录的绝对路径

* buf：存放绝对路径的缓冲区地址
* size：缓冲区的长度（缓冲区必须有足够的长度以容纳绝对路径名再加上一个终止null字节）

## 11.重命名

* 如果oldname指的是一个文件(而不是目录)，并且newname已经存在，且不是一个目录
  + 先将该目录项删除，然后将oldname重命名为newname。对包含oldname的目录以及包含newname的目录，调用进程必须具有写权限
* 如果oldname指的是一个目录，并且newname已经存在，且是一个目录（而且应当是空目录）
  + 先将其删除，然后将oldname重命名为newname。另外，当为一个目录重命名时，newname不能包含oldname作为其路径前缀
* 如果oldname或newname是符号链接
  + 则处理的是符号链接本身，而不是它所引用的文件
* 不能对.和..重命名（更确切的说，.和..都不能出现在oldname和newname的最后部分）
* 如果oldname和newname引用同一文件，则函数不做任何更改而成功返回

**权限**：如果newname已经存在，则调用进程对它需要由写权限。另外，调用进程将删除oldname目录项，并可能要创建newname目录项，所以它需要对包含oldname及包含newname的目录具有写和执行权限

## 12.文件的时间

对每个文件维护3个时间字段，他们的意义如下图：

### 12.1 更改文件的访问和修改时间

下列2个函数包含在POSIX.1中，可以指定**纳秒级精度**的时间戳：

* times：日历时间，自特定时间（1970年1月1日00:00:00）以来所经过的时间
  + times[0]：包含访问时间
  + times[1]：包含修改时间
* **utimensat函数**的path参数是相对于fd参数进行计算的，fd要么是打开目录的文件描述符，要么设置为特殊值AT\_FDCWD（强制通过相对于调用进程的当前目录计算path）。如果path指定了绝对路径，那么fd参数被忽略。如果flag参数设置了AT\_SYMLINK\_NOFOLLOW标志，那么修改的是符号链接本身。默认的行为是会跟随符号链接

时间戳可以按下列4种方式之一进行指定：

1. 如果times参数是一个空指针，则访问时间和修改时间两者都设置为当前时间
2. 如果times数组任一元素的tv\_nsec字段的值为UTIME\_NOW，相应的时间戳就设置为当前时间，忽略相应的tv\_sec字段
3. 如果times数组任一元素的tv\_nsec字段的值为UTIME\_OMIT，相应的时间戳就保持不变，忽略相应的tv\_sec字段
4. 如果times数组任一元素的tv\_nsec字段的值既不是UTIME\_NOW也不是UTIME\_OMIT，相应的时间戳设置为相应的tv\_sec和tv\_nsec字段的值

执行这些函数所要求的权限取决于times参数的值：

* 如果times是一个空指针，或者任一tv\_nsec字段设为UTIME\_NOW，则进程的有效用户ID必须等于该文件的所有者ID。进程对该文件必须具有写权限，或者进程是一个超级用户
* 如果times是非空指针
  + 并且任一tv\_nsec字段的值既不是UTIME\_NOW也不是UTIME\_OMIT，则进程的有效用户ID必须等于该文件的所有者ID，或者进程必须是一个超级用户进程。对文件只有写权限是不够的
  + 并且2个tv\_nsec字段的值都为UTIME\_OMIT，就不执行任何的权限检查

下列函数包含在Single UNIX Specification的XSI扩展选项中，时间戳是用**秒和微妙**表示的：

* times
  + times[0]：包含访问时间
  + times[1]：包含修改时间

## 13.设备特殊文件

st\_dev和st\_rdev两个字段的有关规则：

* 每个文件系统所在的存储设备都由其主、次设备号表示。设备号所用的数据类型是基本系统数据类型dev\_t。主设备号标识设备驱动程序，有时编码为与其通信的外设板；次设备号标识特定的子设备。一个磁盘驱动器经常包含若干个文件系统。在同一磁盘驱动器上的各文件系统通常具有相同的主设备号，但是次设备号却不同
* 通常使用两个宏：**major**和**minor**来访问主、次设备号，大多数实现都定义这两个宏
* 系统中与每个文件名关联的st\_dev值是文件系统的设备号，该文件系统包含了这一文件名以及与其对应的inode
* 只有字符特殊文件和块特殊文件才有st\_rdev值。此值包含实际设备的设备号

可以使用cat /proc/devices查看系统上的字符设备和块设备号：

chenximing@chenximing-MS-7823:~$ cat /proc/devices  
Character devices:  
 1 mem  
 4 /dev/vc/0  
 4 tty  
 4 ttyS  
 5 /dev/tty  
 5 /dev/console  
 5 /dev/ptmx  
 5 ttyprintk  
 6 lp  
 7 vcs  
 10 misc  
 13 input  
 21 sg  
 29 fb  
 89 i2c  
 99 ppdev  
108 ppp  
116 alsa  
128 ptm  
136 pts  
180 usb  
189 usb\_device  
216 rfcomm  
226 drm  
247 mei  
248 hidraw  
249 bsg  
250 watchdog  
251 rtc  
252 dimmctl  
253 ndctl  
254 tpm  
  
Block devices:  
 1 ramdisk  
259 blkext  
 7 loop  
 8 sd  
 9 md  
 11 sr  
 65 sd  
 66 sd  
 67 sd  
 68 sd  
 69 sd  
 70 sd  
 71 sd  
128 sd  
129 sd  
130 sd  
131 sd  
132 sd  
133 sd  
134 sd  
135 sd  
252 device-mapper  
253 virtblk  
254 mdp

我的环境中有一块磁盘划分了3个分区：

chenximing@chenximing-MS-7823:~$ df -h | grep 'sda'  
/dev/sda2 46G 32G 12G 74% /  
/dev/sda1 453M 433M 0 100% /boot  
/dev/sda3 275G 30G 232G 12% /home  
  
chenximing@chenximing-MS-7823:~$ ls -l /dev/sda1 /dev/sda2 /dev/sda3  
brw-rw---- 1 root disk 8, 1 6月 19 10:11 /dev/sda1  
brw-rw---- 1 root disk 8, 2 6月 19 10:11 /dev/sda2  
brw-rw---- 1 root disk 8, 3 6月 19 10:11 /dev/sda3

对于下列程序：

#include <sys/stat.h>  
#include <stdio.h>  
#include <stdlib.h>  
  
int main()  
{  
 struct stat buf1,buf2,buf3;  
 if(stat("/dev/sda1",&buf1) < 0)  
 exit(1);  
 if(stat("/dev/sda2",&buf2) < 0)  
 exit(1);  
 if(stat("/dev/sda3",&buf3) < 0)  
 exit(1);  
  
 printf("dev = %d/%d",major(buf1.st\_dev),minor(buf1.st\_dev));  
 if(S\_ISCHR(buf1.st\_mode) || S\_ISBLK(buf1.st\_mode))  
 printf(" (%s) rdev = %d/%d",  
 (S\_ISCHR(buf1.st\_mode)) ? "character" : "block",  
 major(buf1.st\_rdev),minor(buf1.st\_rdev));  
 printf("\n");  
 printf("dev = %d/%d",major(buf2.st\_dev),minor(buf2.st\_dev));  
 if(S\_ISCHR(buf2.st\_mode) || S\_ISBLK(buf2.st\_mode))  
 printf(" (%s) rdev = %d/%d",  
 (S\_ISCHR(buf2.st\_mode)) ? "character" : "block",  
 major(buf2.st\_rdev),minor(buf2.st\_rdev));  
 printf("\n");  
 printf("dev = %d/%d",major(buf3.st\_dev),minor(buf3.st\_dev));  
 if(S\_ISCHR(buf3.st\_mode) || S\_ISBLK(buf3.st\_mode))  
 printf(" (%s) rdev = %d/%d",  
 (S\_ISCHR(buf3.st\_mode)) ? "character" : "block",  
 major(buf3.st\_rdev),minor(buf3.st\_rdev));  
 printf("\n");  
  
 return 0;  
}

输出结果为：

chenximing@chenximing-MS-7823:~$ ./a.out  
dev = 0/6 (block) rdev = 8/1  
dev = 0/6 (block) rdev = 8/2  
dev = 0/6 (block) rdev = 8/3

将“/dev/sda1”，“/dev/sda2”、“/dev/sda3”分别替换成“/boot”、“/”和“/home”，输出结果为:

chenximing@chenximing-MS-7823:~$ ./a.out  
dev = 8/1  
dev = 8/2  
dev = 8/3

# 三.标准I/O库

标准I/O库处理很多细节，如缓冲区分片、以优化的块长度执行I/O等。这些处理使用户不必担心如何选择使用正确的块长度。这使得它便于用户使用，但是如果不深入了解I/O库函数的操作，也会带来一些问题

## 1.流

标准I/O库与文件I/O区别：

* 标准I/O库处理很多细节，如缓冲区分片、以优化的块长度执行I/O等
* 文件I/O函数都是**围绕文件描述符**进行。首先打开一个文件，返回一个文件描述符；后续的文件I/O操作都使用该文件描述符
* 标准I/O库是**围绕流**进行的

**当用标准I/O库打开或者创建一个文件时，就有一个内建的流与之相关联**

### 1.1 流的定向

流的定向决定了所处理的字符是单字节还是多字节的

对于ASCII字符集，一个字符用一个字节表示；对于国际字符集，一个字符可以用多个字节表示。标准I/O文件流可用于单字节或者多字节字符集。**流的定向**决定了所处理的字符是单字节还是多字节的

* **当一个流最初被创建时，它并没有定向**
  + 若在未定向的流上使用一个多字节I/O函数，则将该流的定向设置为宽定向的（即处理多字节）
  + 若在未定向的流上使用一个单字节I/O函数，则将该流的定向设置为字节定向的（即处理单字节）
* 只有两个函数可以改变流的定向
  + [freopen](#X39d34f4213aecf47c4c98253c8d8b05b17b728e)：清除一个流的定向
  + fwide：设置流的定向（并不改变已定向流的定向）
* 参数：
  + fp：FILE文件对象的指针
  + mode：流的定向模式。
    - 如果mode是**负数**，则函数试图使指定的流为**字节定向**（不保证成功，因为fwide不改变已定向流的定向）
    - 如果mode是**正数**，则函数试图使指定的流为**宽定向**的（不保证成功，因为fwide不改变已定向流的定向）
    - 如果mode为0，则函数不试图设置流的定向，而直接返回该流定向的值（可以用以获取流的定向）

注意：如果fp是无效流，由于fwide从返回值无法得知函数执行成功还是失败。那么我们必须采用这个方法：首先在调用fwide之前清除errno。然后在fwide之后检查errno的值。通过errno来检测fwide执行成功还是失败

### 1.2 3个标准流

操作系统对每个进程定义了3个流，并且这3个流可以自动地被进程使用，他们都是定义在<stdio.h>中：

* **标准输入**：预定义的文件指针(FILE对象)为stdin，它内部的文件描述符就是STDIN\_FILENO
* **标准输出**：预定义的文件指针(FILE对象)为stdout，它内部的文件描述符就是STDOUT\_FILENO
* **标准错误**：预定义的文件指针(FILE对象)为stderr，它内部的文件描述符就是STDERR\_FILENO

## 2.FILE对象

当使用fopen函数打开一个流时，它返回一个指向FILE对象的指针（书中称作**文件指针**）。该对象通常是一个结构，包含了标准I/O库为管理该流所需要的所有信息，包括：

* 用于实际I/O的文件描述符
* 指向用于该流缓冲区的指针
* 该流缓冲区的长度
* 当前在缓冲区中的字符数
* 出错标志

应用程序没必要检验FILE对象，只需要将FILE指针作为参数传递给每个标准I/O函数

FILE对象包含有文件描述符，标准I/O函数可以看作是对fd直接操作的系统调用的封装，优点是带有I/O缓冲，从而减少系统调用的次数：

## 3.缓冲

标准IO库提供缓冲的目的是：尽量减少使用read和write调用的次数。标准I/O库对每个I/O流自动地进行缓冲管理，从而避免了程序员需要手动管理这一点带来的麻烦

### 3.1 3种缓冲类型

标准I/O库提供了三种类型的缓冲：

* **全缓冲**：此时在标准I/O缓冲区被填满后，标准I/O库才进行实际的I/O操作
* **行缓冲**：此时当输入和输出中遇到换行符时，标准I/O库执行实际的I/O操作。但是注意：
  + 只要填满了缓冲区，即使还没有写一个换行符，也立即进行I/O操作
  + 任何时候只要通过标准I/O库，从一个不带缓冲的流或者一个行缓冲的流得到输入数据，则会冲洗所有行缓冲输出流。(即要缓冲输入，先冲洗输出缓冲)
* **不带缓冲**：标准I/O库不进行缓冲。此时任何I/O都立即执行实际的I/O操作

**在一个流上执行第一次I/O操作时，相关标准的I/O函数通常调用 malloc 获取使用的缓冲区**

下列函数可以**设置流的缓冲类型**：

* 参数：
  + fp：被打开的文件对象的指针
  + buf：指向缓冲区的指针（如果是setbuf，长度就是BUFSIZ，定义在<stdio.h>中）
  + mode：指定缓冲类型。可以为：
    - \_IOFBF：**全缓冲**
    - \_IOLBF：**行缓冲**
    - \_IONBF：**不带缓冲**。此时忽略buf和size参数
  + size：缓冲的长度

注意：

* **如果在一个函数内分配一个自动变量类型的标准I/O缓冲区，为了防止内存泄露，从该函数返回之前，必须**[**关闭流**](#X9c8ffc60a84de2dea56ab075cd4c82b2391edb1)**。因为自动变量是栈上分配，函数返回之后自动变量被销毁**
* 某些操作系统将缓冲区的一部分存放它自己的管理操作信息，因此可以存放在缓冲区中的实际数据字节数将少于size
* 通常推荐利用操作系统自动选择缓冲区长度并自动分配缓冲区。在**这种情况下若关闭此流，则标准I/O库会自动释放缓冲区**

### 3.2 缓冲区冲洗

* 缓冲区可以由标准的I/O操作自动地冲洗（如，当填满一个缓冲区时）
* 也可以手动调用fflush函数冲洗一个流
* 参数：
  + fp：被打开的文件对象的指针（fp为NULL时，此函数将导致所有输出流被冲洗）

**冲洗是双向**的：1）输出流 —> 内核 —> 磁盘或者终端； 2）输入流—> 用户缓冲区

冲洗**并不是立即写到磁盘文件中。冲洗只是负责数据传到内核**

### 3.3 标准流与缓冲

* **标准输入与标准输出**
  + 默认情况下，1）若是指向终端设备的流，则是**行缓冲**的；2）否则是**全缓冲**的
* **标准错误**
  + 默认情况下，是**不带缓冲**的
  + 不会是全缓冲

## 4.相关调用

### 4.1 打开流

* 参数：
  + type：指定对该I/O流的读写方式（具体见下图）
    - 其中b用于区分二进制文件和文本文件。但是由于UNIX内核并不区分这两种文件，所以在UNIX环境中指定b并没有什么用
    - 对于fdopen，type意义稍微有点区别。因为该描述符已经被打开，所以fdopen为写而打开并不截断该文件。另外该文件既然被打开并返回一个文件描述符，则它一定存在。因此标准I/O追加写方式也不能创建文件
  + pathname：待打开文件的路径名
  + fp：指定的流（若fp已经打开，则先关闭该流；若fp已经定向，则清除该定向）
  + fd：指定的文件描述符。获得的标准I/O流将与该描述符结合

这**几个函数的常见用途**：

* fopen常用于打开一个指定的文件，返回一个文件指针
* freopen常用于在一个指定的流（标准输入、标准输出或者标准错误）上，打开一个指定的文件
* fdopen常用于将文件描述符包装成一个标准I/O流。因为**某些特殊类型的文件（如管道、socket文件）不能用fopen打开，必须先获取文件描述符，然后对文件描述符调用fdopen**

注意：**当以读和写类型打开一个文件时(type中带+号的类型)，有下列限制**：

* **如果写操作后面没有fflush,fseek,fsetpos,rewind操作之一，则写操作后面不能紧跟读操作**
* **如果读操作后面没有fseek,fsetpos,rewind操作之一，也没有到达文件末尾，则在读操作之后不能紧跟写操作**

**默认情况下，流被打开时是全缓冲的。但是如果流引用的是终端设备，则流被打开时是行缓冲的**

### 4.2 关闭流

* 参数：
  + fp：待关闭的文件指针
* 在该文件被关闭之前：
  + fclose会自动冲洗缓冲中的输出数据，缓冲区中的任何输入数据被丢弃
  + 若该缓冲区是标准I/O库自动分配的，则释放此缓冲区
* 当一个进程正常终止时（直接调用exit函数，或者从main函数返回）：
  + 所有带未写缓存数据的标准I/O流都被冲洗
  + 所有打开的标准I/O流都被关闭

### 4.3 读写流

一旦打开了流，可以在3种不同类型的**非格式化I/O**中选择，对流进行读、写操作：

* **每次一个字符的I/O**。一次读、写一个字符。若流是带缓冲的，则标准I/O函数处理所有缓冲
  + [1）每次读一个字符](#X89f1146e37b3e222e722ea40716ad36cfdbeaef)：getc、fgetc、getchar
  + [2）读取错误判断](#Xb2dbd954181ad2be9893d2f6ec6177a6dd4a571)：ferror、feof、clearerr
  + [3）将读取字符压回流中](#X2f2eb1e0b860b1a209f6d057c575bebc72900a9)：ungetc
  + [4）每次写一个字符](#X151d923ebff93d5afc629e1676db05550c22915)：putc、fputc、putchar
* **每次一行的I/O**。一次读、写一行。每一行都以一个换行符终止
  + [5）每次读一行](#X7dedffdb306f95726ab33a3a54146305571e5c1)
  + [6）每次写一行](#Xb8d5ab9a62786e5ad427d1a2f265792f369c615)
* **直接I/O(二进制I/O、一次一个对象I/O)**。每次I/O读、写某种数量的对象，每个对象具有指定的长度。常用于从二进制文件中每次读或写一个结构
  + [7）二进制I/O](#X7b1ab7b9014e4068245e9c85c3f007c109f27b6)

格式化I/O由printf和scanf族函数完成

#### 1）每次读一个字符

* 参数：
  + fp：文件对象指针

注意：

* getchar()等价于getc(stdin)。它从标准输入中读取一个字符
  + getc和fgetc的区别在于：getc可能通过宏定义来实现，而fgetc不能实现为宏。因此：
    - getc的参数不应该是具有副作用的表达式，因为它可能被计算多次
    - fgetc一定是个函数，因此可以得到其地址，这就允许将fgetc的地址作为参数传递。而getc不行
    - 调用fgetc所需的时间可能比调用getc长，因为函数调用所需时间通常比调用宏长
  + 这三个函数在返回下一个字符时，将unsigned char类型表示的字符转换成了int类型。（因为需要通过返回EOF来标记到达末尾或者出错。而EOF通常是常量 -1 。所以需要返回 int）

#### 2）读取错误判断

上述3个函数不管是出错还是到达文件尾端，都返回相同值，下面的函数可以进行区分

* 参数：
  + fp：文件对象指针

大多数实现中，为每个流在FILE对象中维护了两个标志：**文件出错标志和文件结束标志**。clearerr函数用来清除这两个标志

#### 3）将读取字符压回流中

* 参数：
  + c：待压入字符转换成的整数值
  + fp：打开的文件对象指针

注意：

* 若根据某个序列向流中压入一串字符，则再从该流中读取的字符序列是逆序的。即最后压入的字符最先读出
* 可以执行任意次数的压入单个字符，但是不支持一次压入多个字符
* 不能压入 EOF。但是当已经读到文件尾端时，支持压入一个字符。下次读将返回该字符，再读则返回EOF

ungetc通常用于这样的情形：正在读取一个输入流，然后需要根据某个字符串（标记字符串）来对输入进行切分。那么我们就需要先看一看下一个字符，来决定如何处理当前字符。此时需要方便的将刚查看的字符回送

ungetc只是将字符压入流缓冲区中，并没有压入底层的磁盘文件或者操作系统内核中

#### 4）每次写一个字符

* 参数：
  + c：待写字符转换成的整数值
  + fp：文件对象指针

注意：

* putchar(c)等价于putc(c,stdout)。它向标准输出中写一个字符
* putc和fputc的区别在于：putc可能通过宏定义来实现，而fputc不能实现为宏

#### 5）每次读一行

* 参数：
  + buf：存放读取到的字符的缓冲区地址
  + n：缓冲区长度
  + fp：文件对象指针

注意：

* 对于fgets函数，必须指定缓冲区的长度n。该函数一直读到下一个换行符为止，但是不超过n-1个字符
  + 无论读到多少个字符，缓冲区一定以null字节结尾
  + 若某一行包括换行符超过 n-1个字节，则fgets只返回一个不完整的行；下次调用fgets会继续读该行
* 对于gets函数，从标准输入总读取字符。由于无法指定缓冲区的长度，因此很可能造成缓冲区溢出漏洞，故该函数不推荐使用。gets与fgets的另一个区别是：gets并不将换行符存入缓冲区中
* 对于发生错误和读到末尾，都是返回NULL

#### 6）每次写一行

* 参数：
  + str：待写的字符串
  + fp：文件对象指针

注意：

* fputs和puts都是将一个以null字节终止的字符串写到流中，末尾的null字符不写出！字符串不要求以换行符结尾！
* puts将字符串写到标准输出，末尾的null字符不写出！但是puts随后又将一个换行符写到标准输出中！而fputs不会自动添加换行符（虽然puts不像gets那样不安全，但是我们也是要避免使用它，以免要记住它在最后是否添加了一个换行符）

#### 7）二进制I/O

* 参数：
  + ptr:存放二进制数据对象的缓冲区地址
  + size：单个二进制数据对象的字节数（比如一个struct的大小）
  + nobj：二进制数据对象的数量
  + fp：打开的文件对象指针
* 返回值：
  + 成功或失败： 读/写的对象数
    - 对于读：如果出错或者到达文件尾端，则此数字可以少于nobj。此时应调用ferror或者feof来判断究竟是那种情况
    - 对于写：如果返回值少于nobj，则出错

使用二进制I/O的基本问题是：它只能用在读取同一个操作系统上已写的数据。如果跨操作系统读写，则很可能工作异常。因为：

* 同一个struct，可能在不同操作系统或者不同编译系统中，成员的偏移量不同
* 存储多字节整数和浮点数的二进制格式在不同的操作系统中可能不同

### 4.4 定位流

有3种方法定位标准I/O流：

1. ftell、fseek：它们都假定文件的位置可以存放在一个长整形中
2. ftello、fseeko：使文件偏移量可以不必一定使用长整型，使用off\_t数据类型代替了长整型
3. fgetpos、fsetpos：由ISO C引入，使用一个抽象数据类型fpos\_t记录文件位置。这种数据类型可以根据需要定义一个足够大的数，用以记录文件位置

需要移植到非UNIX系统上运行的应用程序应该使用fgetpos和fsetpos

#### 1）ftell与fseek

* 对于二进制文件
  + 其文件位置指示器是从文件起始位置开始度量，并以字节为度量单位
  + ftell用于二进制文件时，返回这种字节位置
  + fseek可以定位位置，whence与[lseek中的whence相同](#X700fefd172d373086f3bc5da02517cfabd56a42)
* 对于文本文件
  + 它们的文件位置可能不以简单的字节偏移量来度量。因为在非Unix系统中，它们可能以不同的格式存放文本文件
  + 为了定位文本文件，whence一定要是SEEK\_SET，而且offset只能是0或是对该文件的ftell所返回的值

rewind函数也可以将一个流设置到文件的起始位置

#### 2）ftello与fseeko

除了偏移量的类型是off\_t而非long，ftello函数与ftell相同，fseeko函数与fseek相同

#### 3）fgetpos与fsetpos

fgetpos和fsetpos都是ISO C标准引入的

fgetpos将文件位置指示器的当前值存入由pos指向的对象中，在以后调用fsetpos时，可以使用此值将流重新定位至该位置

### 4.5 格式化I/O

#### 1）格式化输出

* printf 将格式化数据写到标准输出
* fprintf 写至指定的流
* dprintf 写至指定的文件描述符。使用该函数不需要调用fdopen将文件描述符转换为文件指针（fprintf需要）
* sprintf 将格式化的字符送入数组buf中，并自动在该数组尾端加一个null字节，但该字符不包括在返回值中
* snprintf sprintf可能会造成buf指向的缓冲区溢出，snprintf的参数n指明了缓冲区长度，超过缓冲区尾端写的所有字符都被丢弃，因此可以解决缓冲区溢出问题

**格式说明的格式**：%[flags][fldwidth][precision][lenmodifier]convtype

* 标志(flags)
* 最小字段宽(fldwidth)
  + 说明最小字段宽度
  + 转换后参数字符如果小于宽度，则多余字符位置用空格填充
  + 字段宽度是一个非负十进制数，或者是一个星号 \*
* 精度(precision)
  + 说明整型转换后最少输出数字位数
  + 说明浮点数转换后小数点后的最少位数
  + 说明字符串转换后最大字节数
  + 精度是一个点.后跟随一个可选的非负十进制数或者一个星号\*
* 参数长度(lenmodifier)
* 转换类型(convtype)

#### 2）格式化输入

scanf族用于分析输入字符串，将字符序列转换成指定类型的变量。在格式之后的各参数中包含了变量的地址，用转换结果对这些变量赋值

除了转换说明和空白字符以外，格式字符串中的其他字符必须与输入匹配。如有一个字符不匹配，则停止后续处理，不再读输入的其余部分

**转换说明的格式**：%[\*][fldwidth][m][lenmodifier]convtype

* \*：用于抑制转换。按照转换说明的其余部分对输入进行转换，但是转换结果不存放在参数中
* fldwidth：说明最大宽度，即最大字符数
* lenmodifier：说明要转换结果赋值的参数大小。见前述说明
* convtype：类似于printf族的转换类型字段。但是稍有区别：输入中的带符号的数值可以赋给无符号类型的变量
* m：赋值分配符。用于强迫内存分配。当%c,%s时，如果指定了m，则会自动分配内存来容纳转换的字符串。同时该内存的地址会赋给指针类型的变量（即要求对应的参数必须是指针的地址）。同时要求程序员负责释放该缓冲区（通过free函数）

### 4.6 获取流相应的文件描述符

如果要调用dup或fcntl等函数，则需要此函数

fileno不是ISO C标准部分，而是POSIX.1支持的扩展

### 4.7 创建临时文件

ISO C标准I/O库提供了2个函数以帮助创建临时文件

* tmpnam 产生一个与现有文件名不同的一个有效路径名字符串，每次调用都产生一个不同路径名。最多调用TMP\_MAX次，定义在<stdio.h>中
  + ptr：指向存放临时文件名的缓冲区的指针
    - 若为NULL，则产生的路径名存放在一个静态区中，指向该静态区的指针作为函数值返回（下次再调用tmpnam时，会重写该静态区）
    - 如果为非NULL，则认为它指向长度至少为L\_tmpnam个字符的数组，产生的路径名存放在该数组中，返回ptr。L\_tmpnam常量定义在<stdio.h>头文件中
* tmpfile 创建一个临时二进制文件(类型wb+)，在关闭该文件或程序结束时将自动删除这种文件（注意，UNIX对二进制文件不进行区分）

tmpfile 经常使用的标准UNIX技术是先调用tmpnam产生一个唯一的路径名，然后，用该路径名创建一个文件，并立即unlink它。对一个文件解除链接并不删除其内容，关闭该文件时才删除其内容。而关闭文件可以是显式的，也可以在程序终止时自动进行

Single UNIX Specification为处理临时文件定义了另外两个函数：mkdtemp和mkstemp，它们是XSI的扩展部分：

* mkdtemp 创建一个目录，该目录有一个唯一的名字
  + mkdtemp函数创建的目录具有权限位集： S\_IRUSR|S\_IWUSR|S\_IXUSR。调用进程的文件模式创建屏蔽字可以进一步限制这些权限
  + mkstemp函数返回的文件描述符以读写方式打开。它创建的文件用访问权限位：S\_IRUSR|S\_IWUSR
* mkstemp 创建一个文件，该文件有一个唯一的名字
  + 与tmpfile不同，mkstemp创建的临时文件并不会自动删除。如果希望从文件系统命名空间中删除该文件，必须自己对它解除链接

名字是通过template字符串进行选择的。这个字符串是后6位设置为XXXXXX的路径名。函数将这些占位符替换成不同的字符来构建一个唯一的路径名。如果成功的话，这两个函数将修改template字符串反映临时文件名

## 5.内存流

**内存流**：一种标准IO流，虽然它通过 FILE指针来访问，但是并没有底层的文件 。所有的IO都是通过在缓冲区和主存之间来回传送字节来完成。虽然它看起来像是文件流，但是更适用于字符串操作

* 参数：
  + buf：内存流缓冲区的起始地址
  + size：内存流缓冲区的大小（字节数）
    - 若buf为NULL时，则函数负责分配size字节的缓冲区，并在流关闭时自动释放分配的缓冲区
  + type:控制如何使用流（即打开内存流的方式）

注意：

* 无论何时以追a方式打开内存流时，当前文件位置设为缓冲区中第一个null字节处。
  + 若缓冲区中不存在null字节，则当前位置设为缓冲结尾的后一个字节
* 当内存流不是a方式打开时，当前位置设置为缓冲区的开始位置
* 如果buf是null，则打开流进行读或者写都没有任何意义。因为此时缓冲区是通过fmemopen分配的，没办法找到缓冲区的地址。
* 任何时候需要增加流缓冲区中数据流以及调用fclose、fflush、fseek、fseeko、fsetpos时都会在当前位置写入一个null字节

创建内存流的其他两个函数：

* 参数：
  + bufp：指向缓冲区地址的指针（用于返回缓冲区地址）
  + sizep:指向缓冲区大小的指针（用于返回缓冲区大小）

open\_memstream 创建的流是面向字节的，open\_wmemstream 创建的流是面向宽字节的

这两个函数与fmemopen的不同在于：

* 创建的流只能写打开
* 缓冲区由函数自动创建，不能指定自己的缓冲区
* 关闭流后需要程序员释放缓冲区
* 对流添加字节会增加缓冲区大小

在缓冲区地址和大小使用上要遵守规则：

* 缓冲区地址和长度只有在调用fclose或者fflush后才有效
* 这些值只有在下一次写入或者调用fclose之前才有效。因为缓冲区可能增长，也可能需要重新分配

因为避免了缓冲区溢出，内存流非常适用于创建字符串。因为内存流只访问主存，不访问磁盘上的文件，所以对于把标准I/O流作为参数用于临时文件的函数来说，会有很大的性能提升

# 四.进程环境

## 1.进程的启动与终止

### 1.1 main函数

C程序总是从main函数开始执行的。main函数的原型是：

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

* 参数：
  + argc：命令行参数的数目（ISO C和POSIX.1都要求argv[argc]是一个空指针）
  + argv：由指向各命令行参数的指针所组成的数组。ISOC和POSIX都要求argv[argc]是一个空指针

#include <stdio.h>  
#include <stdlib.h>  
  
int main(int argc,char\* argv[])  
{  
 printf("%d parameters: \n",argc);  
 for(int i = 0;i < argc;i++){ //argv[argc]为NULL  
 printf(" %s\n",argv[i]);  
 }  
 return 0;  
}

上面的程序可以打印调用程序时的参数：

chenximing@chenximingdeMacBook-Pro  ~  ./a.out hello ' ' world  
4 parameters:  
 ./a.out  
 hello  
  
 world

当内核通过exec函数执行 C 程序、在调用main之前先调用一个特殊的启动例程

* 可执行程序文件将此启动例程指定为程序的起始地址（这是由链接器设置的，而链接器由C编译器调用）
* 启动例程从内核取得命令行参数和环境变量值，为调用main函数做好安排

启动例程使得从main返回后立即调用exit函数。如果将启动例程以C代码形式表示（实际上该例程常常以汇编语言编写），则它调用main函数的形式可能是：

exit(main(argc,argv));

### 1.2 进程终止的方式

有 8 种方式使得进程终止，其中 5 种为正常终止，3 种异常终止：

* **正常终止方式**：
  + 从main函数返回
  + 调用exit函数
  + 调用\_exit函数或者\_Exit函数
  + 多线程的程序中，最后一个线程从其启动例程返回
  + 多线程的程序中，从最后一个线程调用pthread\_exit函数
* **异常终止方式**：
  + 调用abort函数
  + 接收到一个信号
  + 多线程的程序中，最后一个线程对取消请求作出响应

### 1.3 终止函数

下列3个函数正常终止一个程序

* 参数：
  + status：终止状态

三个终止函数的区别：

* \_exit和\_Exit函数：立即进入内核
* exit函数：先执行一些清理处理，然后进入内核

exit和\_Exit是由 ISO C 说明的， \_exit是由 POSIX 说明的 ，因此头文件不同

### 1.4 终止状态

exit、exit、\_Exit 都带有一个整型参数，称为终止状态（或称作退出状态）。大多数 UNIX 系统 shell 都提供检查进程终止状态的方法：

* 下列情况，进程的**终止状态是未定义**的：
  + 若调用exit、exit、\_Exit时不带终止状态
  + 若main执行了一个无返回值的return语句
  + 若main没有声明返回类型为整型
* 下列情况，进程的**终止状态是0**：
  + 若main声明返回类型为整型，并且main执行到最后一条语句时返回（隐式返回）

main函数返回一个整型值与用该值调用exit是等价的：

main 函数中: exit(100);   
  
等价于  
  
return 100;

在LINUX中，退出状态码最高是255，一般自定义的代码值为0~255，如果超出255，则返回取模后的值

### 1.5 登记终止处理程序

* 参数：
  + func：函数指针。它指向的函数的原型是：返回值为void，参数为void

按照ISO C的规定，一个进程可以登记最多32个函数（通常操作系统会提供多于32个的限制。可以用sysconf函数查询这个限制值），这些函数将由exit函数自动调用。这些函数称作**终止处理程序**

* exit调用这些终止处理程序的顺序与它们登记的时候顺序相反
* 如果同一个终止处理程序被登记多次，则它也会被调用多次

## 2.环境表

每个程序都会接收一张**环境表**

* 与参数表一样，环境表也是一个字符指针数组
  + 其中数组中的每个指针指向一个以null结束的 C 字符串，这些字符串称之为环境字符串
  + 数组的最后一项是null
* 全局变量environ包含了该指针数组的地址：extern char \*\*environ。我们称environ为环境指针，它位于头文件unistd.h中
* 按照惯例，环境字符串由name=value这种格式的字符串组成

环境表和环境字符串通常存放在进程存储空间的顶部（栈的上方）

### 2.1 获取环境变量

getenv由ISO C定义，可以用其获取环境变量值

下图为一些环境变量和在不同系统上的支持情况：

### 2.2 修改环境变量

并不是所有系统都支持修改环境变量：

中间3个函数的原型如下：

* putenv ：添加环境变量，将其放入环境表中。str为name=value形式的环境变量。如果已经存在，则先删除原有的定义
* setenv ：修改环境变量。如果环境变量已经存在，有下列2种可能：
  + 若rewrite非0，则先删除其现有定义，然后添加新定义
  + 若rewrite为0，则不删除其现有定义，并且不将name修改为新值value，而且也不出错
* unsetenv ：删除环境变量，即删除name的定义。即使不存在这种定义也不算出错

这些函数内部操作环境表非常复杂，下面是原理：

* **修改**：
  + 如果新的value长度少于或等于现有value的长度，则只需要将新字符串复制到原字符串所用的空间即可
  + 如果新的value长度大于现有value的长度，则必须调用malloc为新字符串分配空间，然后将新字符串复制到该空间，接着使环境表中对name的指针指向新分配区并释放旧分配区
* **增加**：
  + 如果这是第一次增加一个新的name：
    - 则必须调用malloc为新指针表分配空间
    - 然后将原来的环境表复制到新分配区
    - 并将新的name=value字符串的指针存放到该指针表的表尾，
    - 然后将一个空指针存放在其后
    - 然后使environ指向新指针表
    - 最后释放旧的指针表
  + 如果这不是第一次增加一个新的name，则可知以前已经调用了malloc：
    - 则只需要调用realloc，以分配比原空间多存放一个指针的空间
    - 并将新的name=value字符串的指针存放到该指针表的表尾，
    - 然后将一个空指针存放在其后
* **删除**：
  + 则只需要先在环境表中找到该指针，然后将所有的后续指针都向环境表的首部依次顺序移动一个位置即可

修改和增加复杂的原因在于：环境表和环境字符串通常占用的是进程地址空间的顶部，所以它不能再向高地址方向扩展；同时也不能移动在它之下的各栈帧，所以也不能向低地址方向扩展。两者组合使得该空间的长度不能再增加

## 3.C程序的存储空间布局

* **正文段**（.text段）：这是由CPU执行的机器指令部分
  + 通常正文段是可以共享的。一个程序可以同时执行N次，但是该程序的正文段在内存中只需要有一份而不是N份
  + 通常正文段是只读的，以防止程序由于意外而修改其指令
* **初始化数据段**（.data段）：通常将它称作数据段（存放在磁盘可执行文件中，故而占磁盘空间）
  + 它包含了程序中明确地赋了初值的变量：包括**函数外的赋初值的全局变量**、**函数内的赋初值的静态变量**
* **未初始化数据段**（.bss段）：通常将它称作**bss段**。在程序开始执行之前，内核将此段中的数据初始化为0或者空指针
  + 它包含了程序中未赋初值的变量：包括**函数外的未赋初值的全局变量**、**函数内的未赋初值的静态变量**
* **栈**：临时变量以及每次函数调用时所需要保存的信息都存放在此段中
  + 每次函数调用时，函数返回地址以及调用者的环境信息（如某些CPU 寄存器的值）都存放在栈中
  + 最新的正被执行的函数，在栈上为其临时变量分配存储空间（通过这种方式使用栈，C 递归函数可以工作。递归函数每次调用自身时，就创建一个新的栈帧，因此某一次函数调用中的变量不影响下一次调用中的变量）
* **堆**：通常在堆中进行动态存储分配
  + 由于历史习惯，堆位于未初始化数据段和栈段之间

注意：

1. 栈从高地址向低地址增长。堆顶和栈顶之间未使用的虚拟地址空间很大
2. 未初始化数据段的内容并不存放在磁盘程序文件中。需要**存放在磁盘文件中**的段只有**正文段**和**初始化数据段**（size a.out令可以查看程序的正文段、数据段 和bss段长度）

下列程序打印不同段数据的位置：

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
  
extern char\*\* environ;  
  
int global1;  
int global2 = 1;  
  
int main()  
{  
 //1.环境变量  
 char \*\*env = environ;  
 while(\*env){  
 printf("(%p) (%p) %s\n",env,\*env,\*env);  
 env++;  
 }  
 printf("\n");  
 //2.全局变量  
 printf("%p\n",&global1);  
 printf("%p\n\n",&global2);  
 //3.栈  
 int local;  
 char \*localc = (char\*)malloc(10);  
 printf("%p\n",&local);  
 printf("%p\n\n",&localc);  
 //4.堆  
 printf("%p\n\n",localc);  
 return 0;  
}

我的环境下，输出如下：

chenximing@chenximing-MS-7823:~$ ./a.out  
(0x7ffe4aa6f4c8) (0x7ffe4aa7078a) XDG\_SESSION\_ID=1  
(0x7ffe4aa6f4d0) (0x7ffe4aa7079b) TERM=xterm-256color  
(0x7ffe4aa6f4d8) (0x7ffe4aa707af) SHELL=/bin/bash  
(0x7ffe4aa6f4e0) (0x7ffe4aa707bf) XDG\_SESSION\_COOKIE=526d72c400c09a742df8809a58e50d0e-1529375799.783462-1767108743  
(0x7ffe4aa6f4e8) (0x7ffe4aa70810) SSH\_CLIENT=192.168.2.1 53068 22  
(0x7ffe4aa6f4f0) (0x7ffe4aa70830) SSH\_TTY=/dev/pts/25  
(0x7ffe4aa6f4f8) (0x7ffe4aa70844) USER=chenximing  
...  
(0x7ffe4aa6f508) (0x7ffe4aa70d75) MAIL=/var/mail/chenximing  
(0x7ffe4aa6f510) (0x7ffe4aa70d8f) PATH=/home/chenximing/bin:/usr/local/go/bin:/usr/lib/jvm/jdk1.8.0\_144/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/home/chenximing/bin  
(0x7ffe4aa6f518) (0x7ffe4aa70e47) QT\_QPA\_PLATFORMTHEME=appmenu-qt5  
(0x7ffe4aa6f520) (0x7ffe4aa70e68) PWD=/home/chenximing  
(0x7ffe4aa6f528) (0x7ffe4aa70e7d) JAVA\_HOME=/usr/lib/jvm/jdk1.8.0\_144  
(0x7ffe4aa6f530) (0x7ffe4aa70ea1) LANG=zh\_CN.UTF-8  
(0x7ffe4aa6f538) (0x7ffe4aa70eb2) SHLVL=1  
(0x7ffe4aa6f540) (0x7ffe4aa70eba) HOME=/home/chenximing  
(0x7ffe4aa6f548) (0x7ffe4aa70ed0) GOROOT=/usr/local/go  
(0x7ffe4aa6f550) (0x7ffe4aa70ee5) LANGUAGE=zh\_CN:zh  
(0x7ffe4aa6f558) (0x7ffe4aa70ef7) LOGNAME=chenximing  
(0x7ffe4aa6f560) (0x7ffe4aa70f0a) CLASSPATH=.:/usr/lib/jvm/jdk1.8.0\_144/lib  
(0x7ffe4aa6f568) (0x7ffe4aa70f34) SSH\_CONNECTION=192.168.2.1 53068 192.168.2.4 22  
(0x7ffe4aa6f570) (0x7ffe4aa70f64) LC\_CTYPE=zh\_CN.UTF-8  
(0x7ffe4aa6f578) (0x7ffe4aa70f79) GOPATH=~/go  
(0x7ffe4aa6f580) (0x7ffe4aa70f85) LESSOPEN=| /usr/bin/lesspipe %s  
(0x7ffe4aa6f588) (0x7ffe4aa70fa5) XDG\_RUNTIME\_DIR=/run/user/1000  
(0x7ffe4aa6f590) (0x7ffe4aa70fc4) LESSCLOSE=/usr/bin/lesspipe %s %s  
(0x7ffe4aa6f598) (0x7ffe4aa70fe6) \_=./a.out  
  
0x60106c  
0x601050  
  
0x7ffe4aa6f3bc  
0x7ffe4aa6f3c0  
  
0x1725010

画图表示：

## 4.共享库

大多数UNIX系统支持共享库

* 共享库的优点
  + 使得可执行文件中不再需要包含公用的库函数，而只需在所有进程都可引用的存储区中保存这种库例程的一个副本。程序第一次执行或者第一次调用某个库函数时，用动态链接方法将程序与共享库函数相链接，减少了每个可执行文件的长度
  + 可以用库函数的新版本代替老版本而无需对使用该库的程序重新链接编辑（假定参数的数目和类型都没有改变）
* 共享库的缺点
  + 动态链接增加了一些运行时开销。这种时间开销发生在该程序第一次被执行时，或者每个共享库函数第一次被调用时

不同系统中，程序可能使用不同的方法说明是否要使用共享库。比较典型的有cc(1)：

//以下阻止使用共享库  
gcc -static hello1.c  
//以下，gcc默认使用共享库  
gcc hello1.c

可以使用size a.out观察上面两种情况下可执行文件正文和数据段的长度变化

### 静态链接库与动态链接库

链接库按是否被编译到程序内部而分成动态与静态链接库

* **静态链接库**
  + 扩展名为.a，通常命名为libxxx.a
  + 整合到可执行文件中，因此会导致可执行文件体积增大
  + 可独立执行，而不需要再向外部要求读取链接库的内容
  + 链接库升级时，需要重新编译生成可执行文件
* **动态链接库**
  + 扩展名为.so，通常命名为libxxx.so
  + 动态链接库的内容并没有整合到可执行文件中，需要使用时才去读取链接库，因此可执行文件体积较小
  + 不能独立执行，链接库必须存在
  + 链接库升级时，通常不需要重新编译

为了加速动态链接库的访问效率，一般需要将动态链接库载入内存中，从而避免读取磁盘。加载到高速缓存中的动态链接库所在的目录在文件/etc/ld.so.conf中指定，可以使用命令ldconfig将该文件中指定目录下的动态库读入缓存中

如果在编译时指定了动态链接库及库的路径，但是/etc/ld.so.conf文件中没有包含该路径，即这个动态链接库没有载入内存，那么运行生成的可执行文件会报错

ldd命令可以查看某个可执行的二进制文件含有什么动态链接库，对于前面的问题，如果某个动态链接库没有载入内存，即动态链接库的路径没有包含在文件/etc/ld.so.conf中。使用ldd查看时，这个库会显示“not found”

[生成/加载静态链接库与动态链接库](../Linux/Linux常用命令.md#2编译器gcc和g)

[图片来源](https://blog.csdn.net/freestyle4568world/article/details/49817799)

## 5.进程堆空间的管理

堆上空间分配和释放核心都是通过调整**program break**来实现的，malloc和free维护空闲空间链表，以减少调整**program break**的系统调用的调用次数

### 1）调整program break来分配与释放

可以通过**brk**函数和**sbrk**函数改变进程的**program break**位置，从而分配或释放内存。**program break**与[进程的内存布局结构](#Xe54dfa24506d8899c39bb57e597a1848c8c8ef7)中&end位置相同

**program break的位置抬升后，程序可以访问新分配区域内的任何内存地址，而此时物理内存页尚未分配。内核会在进程首次试图访问这些虚拟内存地址时自动分配新的物理内存页**

* **brk**：将**program break**设置为参数指定的位置
  + 虚拟内存以页为单位进行分配，所以end\_data\_segment实际会四舍五入到下一个内存页的边界处
  + 当试图将end\_data\_segment设置为一个**低于**其初始值（&end）的位置时，可能导致无法预知的行为
* **sbrk**：将**program break**在原有地址上增加参数increment大小（intptr\_t为整形类型），该函数成功时返回前一**program break**的地址

**sbrk(0)**将返回**program break**的当前位置，可以用于跟踪堆的大小，或是监视内存分配函数的行为

### 2）使用C库函数进行分配释放

malloc相比brk和sbrk，具有以下优点：

* 属于C标准的一部分
* 更易于在多线程程序中使用
* 接口简单，允许分配小块内存
* 允许随意释放内存块，它们被维护于一张空闲内存列表内，在后续分配内存时循环使用
* malloc ：分配指定字节数的存储区。存储区中的初始值不确定
  + malloc返回内存所采用的字节对齐方式，总是适宜于高效访问任何类型的C语言数据结构。在大多数架构上，意味着malloc是基于8字节或16字节边界来分配内存的
  + malloc(0)要么返回NULL，要么是一块可（并且应该）被free释放的内存
* calloc ：为指定数量指定长度的对象分配存储空间。空间中的每一位都初始化为0
* realloc ：增加或减少以前分配区的长度
  + 当增加长度时，可能需将以前分配区的内容移到另外一个足够大的区域，以便在尾端提供增加的存储区，而新增区域内的初始值则不确定
  + 函数调用前后可能是不同的存储区，所以调用前不应该有指针指向这段存储区，不然修改后，可能该指针会非法访问
  + 应该使用另一个指针保存realloc的返回值，因为如果使用传入的实参保存返回值，那么一旦realloc失败，则会传回NULL，原来的动态内存区再也无法访问，从而发生内存泄露
* free ：上述3个函数都需通过free释放。一般情况下，free并不降低**program break**的位置，而是将这块内存增加到空闲内存列表中，供后续的malloc循环使用，原因是：
  + 被释放的内存块通常会位于堆的中间，而非堆的顶部，因为降低**program break**是不可能的
  + 它最大限度地减少了程序必须执行的sbrk调用次数
  + 在大多数情况下，降低**program break**不会对那些分配大量内存的程序有多少帮助，因为它们通常倾向于持有已分配内存或是反复释放和重新分配内存，而非释放所有内存后再持续运行一段时间

**这3个分配函数所返回的指针一定是适当对齐的，使其可用于任何数据对象**

**这些分配例程通常用sbrk系统调用实现**

* sbrk可以扩充或缩小进程的存储空间。但是大多数malloc和free的实现都不减小进程的存储空间：释放的空间可供以后再分配，将它们保持在malloc池中而不返回给内核
* 大多数实现所分配的存储空间要比所要求的稍大一些，额外的空间用来记录管理信息，比如分配块的长度、指向下一个分配块的指针等等（如果在一个动态分配区的尾部之后或者在起始位置之前进行写操作，会修改管理记录信息。这种类型的错误是灾难性的，但是由于这种错误不会立即暴露出来，因此很难被发现）
* malloc和free相关的致命性错误
  + 调用了malloc函数但是没有调用free函数：会发生内存泄漏，该进程占用的存储空间就会连续增加，直到不再有空闲空间。此时过度的换页开销会导致性能下降
  + free一个已经释放了的块
  + 调用free时所用的指针不是3个alloc函数的返回值

### 3）malloc和free的实现

**malloc的实现**

1. 首先会扫描（不同实现扫描策略可能不同）之前由free释放的空闲内存块列表，试图找到尺寸大小**大于或等于**要求的一块空闲内存
   * 如果一内存块的尺寸恰好符合要求，那么直接返回给调用者
   * 如果内存块较大，那么进行分割，将一块大小相当的内存返回给调用者，剩余内存块保留在空闲列表
2. 如果空闲内存列表中根本找不到足够大的空闲内存块（**如首次调用malloc时**），那么malloc会调用**sbrk**以分配更多的内存（为了减少对**sbrk**的调用次数，malloc并不是严格按所需字节数来分配内存，而是以更大幅度来增加**program break**，并将超出部分置于空闲内存列表）

当malloc分配内存块时，会额外分配几个字节来存放**记录这块内存大小的整数值**。该整数位于内存块的起始处，而实际返回给调用者的内存地址恰好位于 这一长度记录字节之后：

**free的实现**

1. free会使用内存块本身的空间来存放链表指针，将自身添加到列表中
2. 随着对内存不断地释放和重新分配，空闲列表中的空闲内存会和已分配的在用内存混杂在一起

C语言允许程序创建指向堆中任意位置的指针，并修改其指向的数据，包括由free和malloc函数维护的内存长度、指向前一空闲块和后一空闲块的指针。如果发生这种缓冲区溢出的情况，会带来严重的后果

## 6.进程资源限制

每个进程都有一组资源限制，其中一些可以用getrlimit和setrlimit函数查询和更改：

进程的资源限制通常是在系统初始化时由0进程建立的，然后由后续进程继承。每种实现都可以用自己的方法对资源限制做出调整

* 参数
  + resource：相应的资源。可以设置为下列值（不同系统支持不同）：
    - RLIMIT\_AS：进程总的可用存储空间的最大长度(字节)（这会影响到sbrk函数和mmap函数）
    - RLIMIT\_CORE：core文件的最大字节数。如果为0，则阻止创建core文件
    - RLIMIT\_CPU：CPU时间的最大量值（秒），如果超过此软限制时，向该进程发送SIGXCPU信号
    - RLIMIT\_DATA：数据段的最大字节长度（包括初始化数据、非初始以及堆的总和）
    - RLIMIT\_FSIZE：可以创建的文件的最大字节长度。当超过此软限制时，向该进程发送SIGXFSX信号
    - RLIMIT\_MEMLOCK：一个进程使用mlock能够锁定在存储空间中的最大字节长度
    - RLIMIT\_MSGQUEUE：进程为POSIX消息队列可分配的最大存储字节数
    - RLIMIT\_NICE：为了影响进程的调度优先级，nice值可设置的最大限制
    - RLIMIT\_NOFILE：每个进程能打开的最多文件数。更改此限制将影响到sysconf函数在参数\_SC\_OPEN\_MAX中返回的值
    - RLIMIT\_NPROC：每个实际用户ID可以拥有的最大子进程数。更改此限制将影响到sysconf函数在参数\_SC\_CHILD\_MAX中返回的值
    - RLIMIT\_RSS：最大驻内存集字节长度
    - RLIMIT\_SIGPENDING：一个进程可排队的信号的最大数量。这个限制是sigqueue函数实施的
    - RLIMIT\_STACK：栈的最大字节长度
  + rlptr：包含该资源软限制和硬限制的结构体
    - getrlimit调用后，这个指针指向的结构中包含了欲获取的相应资源的限制信息
    - setrlimit调用时，这个指针指向的结构中包含了欲设置的相应资源的现在信息
      * 任何进程都可将一个软限制值更改为小于或者等于其硬限制值
      * 任何进程都可以降低其硬限制值，但是它必须大于或者等于其软限制值（这种降低，对普通用户而言不可逆，因为普通用户不可提高其硬限制值）
      * 只有超级用户进程才能够提高其硬限制值

struct rlimit{  
 rlim\_t rlim\_cur; /\* soft limit: current limit \*/  
 rlim\_t rlim\_max; /\* hard limit: maximum value for rlim\_cur \*/  
};

**资源限制影响到调用进程并由其子进程继承**

## 7.Core Dump

[内容参考](http://www.cnblogs.com/hazir/p/linxu_core_dump.html)

### 7.1 Core Dump的概念及用途

Core Dump是一个文件，它记录了程序运行过程中异常终止或崩溃时的内存映像

**概念**：当程序运行的过程中**异常终止**或**崩溃**，操作系统会将程序当时的内存状态记录下来，保存在一个文件中，这种行为就叫做Core Dump（中文有的翻译成“核心转储”)。可以认为 core dump 是“内存快照”，但实际上，除了内存信息之外，还有些关键的程序运行状态也会同时 dump 下来，例如寄存器信息（包括程序指针、栈指针等）、内存管理信息、其他处理器和操作系统状态和信息

**Dump**指的是拷贝一种存储介质中的部分内容到另一个存储介质，或者将内容打印、显示或者其它输出设备。dump 出来的内容是格式化的，可以使用一些工具来解析它

**用途**：core dump 对于编程人员诊断和调试程序是非常有帮助的，因为对于有些程序错误是很难重现的，例如指针异常，而 core dump 文件可以再现程序出错时的情景

### 7.2 产生Core Dump

ulimit -c可以查看shell进程产生的core dump文件大小，默认情况下为0，即不产生core dump文件，若要产生，使用ulimit -c unlimited开启core dump功能（也可以改成为某一值，注意这种修改方式只会对当前终端环境生效）

常见的几种情况：调用abort()函数、访存错误、非法指令等

以下列出几种会产生Core Dump的信号（不仅限于这几种）：

|  |  |  |
| --- | --- | --- |
| 信号 | 动作 | 说明 |
| SIGQUIT | 终止+core | 终端退出符(ctrl+\) |
| SIGILL | 终止+core | 非法硬件指令 |
| SIGABRT | 终止+core | 异常终止(abort) |
| SIGSEGV | 终止+core | 无效内存引用 |
| SIGTRAP | 终止+core | 硬件故障 |

* **修改core dump文件名**：通过修改 /proc/sys/kernel/core\_uses\_pid 文件可以让生成 core 文件名自动加上 pid 号。例如 echo 1 > /proc/sys/kernel/core\_uses\_pid，生成的 core 文件名将会变成 core.pid，其中 pid 表示该进程的 PID
* **修改core dump文件的保存位置及文件名格式**：例如可以用 echo "/tmp/core-%e-%p-%t" > /proc/sys/kernel/core\_pattern 设置生成的 core 文件保存在 “/tmp/corefile” 目录下，文件名格式为 “core-命令名-pid-时间戳”

### 7.3 调试Core Dump

1. 首先，使用 gcc 编译源文件，加上 -g 以增加调试信息
2. 打开 core dump 以使程序异常终止时能生成 core 文件
3. 运行程序，当core dump 之后，使用命令 gdb program core 来查看 core 文件（program为可执行程序名，core为生成的 core 文件名）

#### 1）无效内存引用

#include <stdio.h>  
  
int main()  
{  
 int \*p = NULL;  
 int x = \*p;  
 return 0;  
}

生成core dump：

root@068ca8da6d06:~# gcc -g test.c  
root@068ca8da6d06:~# ls  
a.out test.c  
root@068ca8da6d06:~# ./a.out  
Segmentation fault (core dumped)  
root@068ca8da6d06:~# ls  
a.out core test.c

使用gdb调试：

root@068ca8da6d06:~# gdb a.out core  
...  
...  
Core was generated by `./a.out'.  
Program terminated with signal SIGSEGV, Segmentation fault.  
#0 0x0000556146c2160a in main () at test.c:6  
6 int x = \*p;  
(gdb) where  
#0 0x0000556146c2160a in main () at test.c:6  
(gdb) info frame  
Stack level 0, frame at 0x7ffe5f70daa0:  
 rip = 0x556146c2160a in main (test.c:6); saved rip = 0x7f65fcff1b97  
 source language c.  
 Arglist at 0x7ffe5f70da90, args:  
 Locals at 0x7ffe5f70da90, Previous frame's sp is 0x7ffe5f70daa0  
 Saved registers:  
 rbp at 0x7ffe5f70da90, rip at 0x7ffe5f70da98

从上面可以看出,我们可以还原程序执行时的场景,并使用 where 可以查看当前程序调用函数栈帧, 还可以使用 gdb 中的命令查看寄存器,变量等信息

# 五.进程控制

## 1.进程标识

进程通过进程ID进行标识

进程ID可以复用。大多数UNIX系统实现延迟复用算法，使得赋予新建进程的ID不同于最近终止进程所使用的ID，从而防止新进程被认为是使用同一ID的某个已终止进程

### 1.1 2个特殊进程

* **0号进程：调度进程(也被称为系统进程或交换进程)**
  + **性质**：内核的一部分，并不执行任何磁盘上的程序
* **1号进程：init进程**
  + **性质**：普通进程，但以超级用户特权运行。程序文件在早期版本中是/etc/init，新版本中是/sbin/init
  + **启动时机**：在自举（自举是什么鬼？）过程结束时由内核调用
  + **职责**：
    - 读取与系统有关的初始化文件，并将系统引导到一个状态（如多用户）
    - 成为所有孤儿进程的父进程
    - …

### 1.2 进程相关的ID获取函数

这些函数都没有出错返回

## 2.进程的创建

下列函数可以用来创建进程：

* [fork](#Xa7a67b98d39226fe765d04446de60f8bfe99a97)
* [vfork](#X278285a5b0cab46b1cd8ab7cd897610796e8e75)（fork 4种变体的一种，另外3种支持的平台不多）

### 2.1 fork

* 返回值：
  + **父进程返回子进程ID**：原因是，一个进程的子进程可以有多个，并且没有一个函数使一个进程可以获得其所有子进程的进程ID
  + **子进程返回0**：原因是，子进程总是可以调用getppid获得父进程的ID（0号进程总是由内核交换进程使用，所以一个子进程的进程ID不可能为0）

**子进程和父进程继续执行fork调用之后的指令**

#### 1）执行顺序

一般来说，fork后父子进程谁先执行是不确定的，取决于内核所用的调度算法

#### 2）数据共享

因为fork之后经常跟随着exec，所以很多实现中，子进程的**数据段**、**堆**和**栈**指向相应的父进程版本，因此父子进程共享使用，如果子进程有修改，会执行**写时复制**(**COW**)（通常将修改所在的一”页“进行复制）。从而不会影响父进程

* 注意父子进程之间**I/O缓冲区**的复制：
  + 文件I/O中write不带缓冲
  + 标准I/O库带缓冲。因此要特别注意父子进程间缓冲区的复制
* 注意父子进程的**文件描述符**：
  + **在重定向父进程的标准输出时，子进程的标准输出也会被重定向**
  + fork的一个特性是父进程的所有打开文件描述符都被”复制“（就好像执行了dup函数）到子进程中，**父子进程每个相同的打开描述符共享一个文件表项**（故**父子进程共享同一个文件偏移量** ）

一个进程fork了一个子进程，然后等待子进程终止。假定，作为普通处理的一部分，父进程和子进程都向标准输出进行写操作。如果父进程的标准输出已重定向，那么子进程写到该标准输出时，它将更新与父进程共享的该文件的偏移量

除了打开的文件描述符之外，**子进程还继承了父进程的下列属性**：

* 实际用户ID
* 实际组ID
* 有效用户ID
* 有效组ID
* 附属组ID
* 进程组ID
* 会话ID
* 控制终端
* 设置用户ID标志和设置组ID标志
* 当前工作目录
* 根目录
* 文件模式创建屏蔽字
* 信号屏蔽和信号处理
* 对任一打开文件描述符的执行时关闭标志
* 环境
* 连接的共享存储段
* 存储映像
* 资源限制

**父子进程的区别**为：

* fork返回值不同
* 进程ID不同
* 进程父进程ID不同
* 子进程的tms\_utime,tms\_stime,tms\_cutime,tms\_ustime的值设置为0
* 子进程不继承父进程设置的文件锁
* 子进程的未处理闹钟被清除
* 子进程的未处理信号集设置为空集

#### 3）fork失败的主要原因

* 系统已经有了太多的进程
* 实际用户的进程总数超过了系统的限制（CHILD\_MAX规定了每个实际用户在任一时刻拥有的最大进程数）

#### 4）fork的2种用法

1. 父进程希望复制自己，使父进程和子进程同时执行不同的代码段（在网络服务中很常见：父进程等待请求，然后调用fork并使子进程处理请求）
2. 父进程要执行一个不同的程序（在shell是很常见。此时子进程从fork返回之后立即调用exec）

### 2.2 vfork

vfork用于创建一个新进程，该新进程的目的是exec一个新程序，因此子进程并不将父进程的地址空间完全复制到子进程中

在子进程调用exec或exit之前，它在父进程的空间中运行。这种优化在某些UNIX系统的实现中提高了效率，但如果子进程修改数据、进程函数调用、或者没有调用exec或exit就返回都可能会带来未知的结果

vfork和fork之间的另一个区别是：**vfork保证子进程先运行**，在它调用exec或exit之后父进程才可能被调度运行（如果在调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致死锁）

int globval =6;  
  
int main(int argc,char \*\* argv) {  
 int val;  
 pid\_t pid;  
 val = 88;  
 printf("before vfork\n");  
 if ((pid = vfork()) < 0) {  
 exit(1);  
 } else if (pid == 0) {  
 globval++;  
 val++;  
 \_exit(0); //\_exit并不执行标准I/O缓冲区的冲洗操作，如果调用的是exit，输出是不确定的  
 }  
 printf("pid=%ld,glob=%d,val=%d\n", (long) getpid(), globval, val);  
 exit(0);  
}

上述程序子进程会修改父进程地址空间中的数据，子进程修改变量后即退出，以下为父进程的输出：

before vfork  
pid=30595,glob=7,val=89

## 3.进程的终止

[8种终止方式](#X20027587313809af14b0fbeafd820e43a4b88cd)

不管进程如何终止，最后都会执行内核中的同一段代码。这段代码为相应进程关闭所有打开描述符，释放它所使用的存储器等

### 3.1 子进程向父进程传递状态

* **子进程正常终止**：子进程将其退出状态作为参数传给exit、\_exit、\_Exit
* **子进程异常终止**：内核会产生一个指示子进程异常终止原因的终止状态

不管正常终止或异常终止，内核都向父进程发送SIGCHLD信号，父进程可以选择忽略该信号(**默认动作**)，或者提供一个信号处理函数

父进程能通过wait、waitpid获取子进程的终止状态

* 参数
  + statloc：如果关心终止状态就传入一个int变量的地址，终止状态将存于其中；不关心终止状态可以设为NULL。该整形状态字是由实现定义的：
    - 某些位表示退出状态（正常返回）
    - 某些位指示信号编号（异常返回）
    - 有一位指示是否产生了core文件 可以使用POSIX.1规定，终止状态用定义在<sys/wait.h>中的宏来查看，下图4个互斥的宏可用来取得进程终止的原因。基于这4个宏中哪一个值为真，就可选用其它宏来取得退出状态，信号编号等：
  + pid：waitpid可以指定等待哪个子进程终止，wait无法指定（wait和waitpid的区别）
    - -1：等待任一子进程，这种情况下与wait等效
    - >0：等待进程ID与pid相等的子进程
    - 0：等待组ID等于调用进程组ID的任一子进程
    - <-1：等待组ID等于pid绝对值的任一子进程
  + options：waitpid可以通过该参数时调用不阻塞（wait和waitpid的区别）。可以为0，也可以是下图按位或运算的结果：

父进程调用wait、waitpid时可能出现几种情况：

* 如果所有子进程都还在运行，则阻塞
* 如果一个子进程已终止，正等待父进程获取其终止状态，则取得子进程的终止状态立即返回
* 如果没有任何子进程，则出错返回
* 对于waitpid，如果指定的进程或进程组不存在，或参数pid指定的进程不是调用进程的子进程，都可能出错

### 3.2 父子进程以不同顺序终止

* **父进程在子进程之前终止**：该进程的所有子进程会变成**孤儿进程**，这些孤儿进程会被init进程收养，作为新的父进程（过程大致是：在一个进程终止时，内核逐个检查所有活动进程，以判断它是否是正要终止进程的子进程，如果是，则该进程的父进程ID就更改为1）
* **子进程在父进程之前终止**：无法确保子进程终止时，父进程已经准备好了检查其终止状态。因此内核为每个终止子进程保存了一定量的信息。当父进程调用wait或waitpid时，可以得到这些信息。这些信息至少包括进程ID、进程的终止状态、进程使用的CPU时间总量。在父进程调用wait或waitpid之前，终止子进程被称为**僵死进程**（init的子进程终止时，init就会调用一个wait函数取得其终止状态）

## 4.竞争条件

**竞争条件是多个进程企图对共享数据进程某种处理，最终结果取决于进程运行的顺序**

如果fork之后的某种逻辑显示或隐式地依赖于在fork之后是父进程先运行还是子进程先运行，那么fork就会是竞争条件活跃的滋生地

防止竞争条件最简单的手段：

* 父进程可以通过wait等待子进程先运行完
* 子进程可以通过下列方式等待父进程先运行结束 while(getppid() != 1) sleep(1); 但是这种轮询方式会大量浪费CPU浪费

好的办法是使用信号进行通信，或者使用IPC

在父进程和子进程的关系中，常常出现下列情况：在fork之后，父进程和子进程都有一些事情要做。例如，，父进程可能要用子进程ID更新日志文件中的一个记录，而子进程则可能要为父进程创建一个文件。在本例中，要求每个进程在执行完它的一套初始化操作后要通知对方，并且在继续运行之前，要等待另一方完成其初始化操作：

TELL\_WAIT(); /\* set things up for TELL\_xxx & WAIT\_xxx \*/  
if ( (pid = fork()) < 0 ){  
 err\_sys("fork error");  
} else if(pid == 0){ /\* child \*/  
 /\* child does whatever is necessary ... \*/  
 TELL\_PARENT(getppid()); /\* tell parent we're done \*/  
 WAIT\_PARENT(); /\* and wait for parent \*/  
 /\* and the child continues on its way ... \*/  
 exit(0);  
}  
/\* parent does whatever is necessary ... \*/  
TELL\_CHILD(pid); /\* tell child we're done \*/  
WAIT\_CHILD(); /\* and wait for child \*/  
/\* and the parent continues on its way ... \*/  
exit(0);

## 5.exec函数

当进程调用一种exec函数时，该进程执行的程序完全替换成新程序，而新程序则从main函数开始执行

* 调用exec前后，进程ID并未改变。因为exec并不创建新进程
* exec只是用磁盘上的一个新程序替换了当前进程的**正文段**、**数据段**、**堆**和**栈**

有7种不同的exec函数可以供使用，它们被统称称作exec函数：

* **新程序指定方式**
  + **前四个**函数取**路径名**作为参数
  + **后两个**函数取**文件名**作为参数（**函数名中带p**）
    - 如果filename中包含/，则将其视为路径名
    - 否则，按PATH环境变量，在它所指定的各目录中搜寻可执行文件
  + **最后一个**取**文件描述符**做参数
* **新程序的命令行参数**
  + **函数名中的l**：表示列表。要求将新程序的每个命令行参数都说明为一个单独的参数，这种参数表以空指针结尾
  + **函数名中的v**：表示矢量。先构造一个指向各参数的指针数组，然后将该指针数组的地址作为参数
* **新程序的环境变量**
  + **函数名中的e**：可以传递一个指向环境字符串指针数组的指针，数组最后一个元素必须是空指针
  + **否则**：使用进程的environ变量为新程序复制现有环境

7个函数之间的关系如下图：

执行exec后，进程ID没有改变。新程序从调用进程继承了下列属性：

* 进程ID和父进程ID
* 实际用户ID和实际组ID
* 附属组ID
* 进程组ID
* 会话ID
* 控制终端
* 闹钟尚余留的时间
* 当前工作目录
* 根目录
* 文件模式创建屏蔽字
* 文件锁
* 进程信号屏蔽
* 未处理信号
* 资源限制
* nice值
* tms\_utime、tms\_stime、tms\_cutime以及tms\_cstime值

注意：

* 进程中每个打开的文件描述符都有一个执行时关闭标志。若设置了此标志，则执行exec时会关闭该文件描述符；否则该文件描述符仍然保持打开。**系统默认行为是不设置执行时关闭标志**
* 进程的 实际用户ID 和 实际组ID 不变，有效用户ID 和 有效组ID 是否改变取决于所执行程序文件的设置用户ID位和设置组ID位是否设置
  + 1）若新程序的设置用户ID位已设置，则有效用户ID变成程序文件所有者的ID；2）否则有效用户ID不变
  + 1）若新程序的设置组ID位已设置，则有效组ID变成程序文件所有组的ID；2）否则有效组ID不变

## 6.更改用户ID和更改组ID

用以修改进程的**特权**及**访问控制**

UNIX中，特权及访问控制，是基于用户ID和组ID的

* 当程序需要增加特权，或需要访问当前并不允许访问的资源时，需要更换自己的用户ID或组ID，使得新ID具有合适的特权或访问权限
* 当程序需要降低特权，或阻止对某些资源的访问时，也需要更换用户ID或组ID，新ID不具有相应特权或访问这些资源的能力

### 6.1 setuid与setgid

* setuid：设置实际用户ID和有效用户ID
* setgid：设置实际组ID和有效组ID

谁能更改ID需要遵循若干规则，以用户ID为例（针对setgid的讨论类似setuid）：

1. 如果**进程具有超级用户特权**，则setuid函数将实际用户ID，有效用户ID以及保存的设置用户ID(saved set-user-ID) 全部设置为uid（此时uid没有限制）
2. 如果**进程没有超级用户特权**，但是uid等于实际用户ID或者保存的设置用户ID，则setuid只会将有效用户ID设置为uid，不改变实际用户ID和保存的设置用户ID
3. 如果上面两个条件都不满足，则errno设置为EPERM并返回 -1

上述讨论中，假设\_POSIX\_SAVED\_IDS为真。如果未提供此功能，则对于保存的设置用户ID部分都无效

操作系统内核为每个进程维护3个用户ID：实际用户ID、有效用户ID、保存的设置用户ID：

* **只有超级用户进程可以更改实际用户ID**。通常，实际用户ID是在用户登录时，由login(1)程序设置的，而且绝不会改变它。因为login是一个超级用户进程，当它调用setuid时，设置所有3个用户ID
* **仅当对程序文件设置了设置用户ID位时，exec函数才设置有效用户ID**。如果设置用户ID位没有设置，则exec函数不会改变有效用户ID，而是维持其现有值
  + 任何时候都可以调用setuid将有效用户ID设置为实际用户ID或者保存的设置用户ID
  + 调用setuid时，有效用户ID不能随意取值，只能从实际用户ID或者保存的设置用户ID中取得
* **保存的设置用户ID是由exec复制有效用户ID而得到**。如果设置了文件的设置用户ID位，则exec根据文件的用户ID设置了进程的有效用户ID之后，这个副本就保存起来

下图总结了更改这3个用户ID的不同方法：

目前可以通过getuid获取进程的当前实际用户ID，可以通过geteuid获取进程的当前有效用户ID，但是没有函数获取进程当前的保存的设置用户ID

### 6.2 setreuid和setregid

历史上，BSD支持setreuid函数，其功能是交换实际用户ID和有效用户ID的值

若其中任一参数为-1，则表示相应的ID应当保持不变

**一个非特权用户总能交换实际用户ID和有效用户ID**（这就允许一个设置用户ID程序交换用户的普通权限，以后又再次交换回设置用户ID权限。POSIX.1引进了保存的设置用户ID特性后，其规则也相应加强，它允许一个非特权用户将其有效用户ID设置为保存的设置用户ID）

### 6.3 seteuid和setegid

POSIX.1提供下列2个函数，类似于setuid和setgid，但是他们只更改有效ID，不更改实际ID：

* 一个**非特权用户**可将其有效用户ID设置为其实际用户ID或其保存的设置用户ID
* 一个**特权用户**则可将有效用户ID设置为uid

## 7.system函数

system用于将一个字符串作为命令来执行

ISO C定义了system函数，但是其操作对系统的依赖性很强。POSIX.1包括了system接口，它扩展了ISO C定义，描述了system在POSIX.1环境中的运行行为

* 参数：
  + cmdstring：命令字符串（在shell中执行），如 "ps -aux"（如果cmdstring为空指针，则如果system返回 0 表示该操作系统不支持system函数；否则支持）
* 返回值：
  + **system等同于同时调用了fork、exec、waitpid**，有3种返回值：
    - fork失败或者waitpid返回除EINTR之外的错误，则system返回 -1，并且设置errno以指示错误类型
    - 如果exec失败(表示不能执行shell)，则其返回值如同shell执行了exit(127)一样
    - 如果三个函数都执行成功，则system返回值是shell的终止状态，其格式在waitpid中说明

system VS fork + exec：

* **优点**：system进行了所需的各种出错处理以及各种信号处理
* **缺点**：一旦调用system的进程具有超级用户权限，则system执行的命令也具有超级用户权限。因为system的实现过程中并没有更改有效用户ID和实际用户ID的操作
  + 因此如果一个进程以特殊的权限运行，而它又想生成另一个进程执行另外一个程序，则它应该直接使用fork + exec并且在fork之后，exec之前改回普通权限
  + 设置用户ID和设置组ID程序绝不应该调用system函数

## 8.用户标识

* 返回值：
  + 成功：返回指向登录名字符串的指针
  + 失败：返回NULL

通常失败的原因是：调用此函数的进程没有连接到用户登录时所用的终端。比如守护进程

## 9.进程调度

进程可以通过调整nice值选择以更低优先级运行，只有特权进程允许提高调度权限

* nice值的范围在0~(2\*NZERO)-1之间，有些实现支持0~2\*NZERO（NZERO是系统默认的nice值）
* nice值越低，优先级越高

进程可以通过nice函数**获取或更改**它的nice值，进程只能影响自己的nice值，不能影响任何其他进程的nice值：

* 参数
  + incr：nice值的增量。如果太小或太大，系统会修改到边界值（为0时，nice值不变，因此可以用以获取当前nice值）
* 返回值
  + -1：出错会返回-1，但是正常返回也可能是-1，所以调用前要清除errno

getpriority函数可以像nice函数那样用于获取进程的nice值，但是它还可以获取一组相关进程的nice值：

* 参数：
  + which：控制who参数是如何解释的。可以取三个值之一：
    - PRIO\_PROCESS：表示进程
    - PRIO\_PGRP:表示进程组
    - PRIO\_USER表示用户ID
  + who：选择感兴趣的一个或者多个进程
    - 如果who为0，which为PRIO\_PROCESS，返回当前进程的nice值
    - 如果who为0，which为PRIO\_PGRP，则返回进程组中最小的nice值
    - 如果who为0，which为PRIO\_USER，则返回调用进程的实际用户ID拥有的那些进程中最小的nice值
  + value：nice的增量

setpriority函数可用于为进程、进程组和属于特定用户ID的所有进程设置优先级

* 参数：
  + which和who与getpriority函数中相同
  + value：nice值的增量

Single UNIX Specification没有对在fork之后子进程是否继承nice值制定规则。而是留给具体实现自行决定。但是遵循XSI的系统要求进程调用exec后保留nice值。在FreeBSD 8.0、Linux 3.2.0、MacOS x 10.6.8以及Solaris 10中，子进程从父进程中继承nice值

## 10.进程时间

任一进程可调用times函数获得它自己以及已终止子进程的**墙上时钟时间**、**用户CPU时间**、**系统CPU时间**

该函数填写由buf指向的tms结构：

struct tms{  
 clock\_t tms\_utime; /\* 用户CPU时间 \*/  
 clock\_t tms\_stime; /\* 系统CPU时间 \*/  
 clock\_t tms\_cutime; /\* 已终止子进程的用户CPU时间 \*/  
 clock\_t tms\_cstime; /\* 已终止子进程的系统CPU时间 \*/  
};

结构中两个针对子进程的字段包含了此进程用wait函数族已等待到的各子进程的值

**墙上时钟时间**作为函数返回值返回。这个值是相对于过去的某一时刻度量的，所以不能用其绝对值而必须使用相对值。例如，调用times，保存其返回值。在以后某个时间再次调用times，从新返回的值中减去以前返回的值，此差值就是墙上时钟时间（一个长期运行的进程可能其墙上时钟时间会溢出，当然这种可能性极小）

所有由此函数返回的clock\_t值都用sysconf(\_SC\_CLK\_TCK)(每秒时钟滴答数)转换成秒数

# 六.线程

当一个进程需要另一个实体来完成某事时，Unix上大多数网络服务器通过fork一个子进程来处理。但是**fork调用存在一些问题**：

* **fork是昂贵的**。fork要把父进程的内存映像复制到子进程，并在子进程中复制所有描述符。尽管现在使用**写时拷贝**技术，避免在子进程切实需要自己的副本之前把父进程的数据空间拷贝到子进程。但是fork仍然是昂贵哦的
* **fork返回之后父子进程之间信息的传递需要进程间通信(IPC)机制**。调用fork之前，父进程向尚未存在的子进程传递信息相当容易，因为子进程将从父进程数据空间及所有描述符的一个副本开始运行。然而从子进程往父进程返回信息却比较费力

**线程有助于解决上述问题，它被称为“轻量级进程”，创建可能比进程的创建快10~100倍。但是，伴随这种简易性而来的是同步问题**

**线程之间的资源共享**：

* **同一进程内的线程共享**
  + 相同的全局内存
  + 进程指令
  + 大多数数据
  + 打开的文件（即描述符）
  + 信号处理函数和信号设置
  + 当前工作目录
  + 用户ID和组ID
* **线程之间不共享**
  + 线程ID
  + 寄存器集合（包括程序计数器和栈指针）
  + 栈（用于存放局部变量和返回地址）
  + errno
  + 信号掩码
  + 优先级

这一章介绍的是POSIX线程，也称为Pthread。POSIX线程作为POSIX.1c标准的一部分在1995年得到标准化，大多数UNIX版本将来会支持这类线程。所有Pthread函数都以pthread\_打头

## 1.相关函数

### 1）pthread\_create函数

该函数用于创建一个POSIX线程。**当一个程序由exec启动执行时，称为“初始线程”或“主线程”的单个线程就创建了。其余线程则由pthread\_create函数创建**

* **tid**：线程ID，数据类型为pthread\_t，往往是unsigned int，如果线程成功创建，其ID就通过tid指针返回
* **attr**：线程属性，包括：优先级、初始栈大小、是否应该成为一个守护线程等。设置为空指针时表示采用默认设置
* **func**：该线程执行的函数
* **arg**：该线程执行函数的参数，参数为一个无类型指针，如果需要向函数传递的参数有一个以上，那么需要把这些参数放到一个结构中，然后把这个结构的地址作为参数传入

**如果发生错误，函数返回指示错误的某个正值，不会设置errno变量**

**创建的线程通过调用指定的函数开始执行，然后显示地（通过调用pthread\_exit）或隐式地（通过让该函数返回）终止**

**线程创建时，并不能保证哪个线程会先运行**

### 2）pthread\_join函数

pthread\_join类似于进程中的waitpid，用于等待一个给定线程的终止

* **tid**：等待终止的线程ID。和进程不同的是，无法等待任意线程，所以不能通过指定ID参数为-1来企图等待任意线程终止
* **status**：如果该指针非空，来自所等待线程的返回值（一个指向某个对象的指针）将存入由status指向的位置

对于一个非脱离状态的线程，如果没有其它线程调用pthread\_join等待线程终止，那么线程终止后的资源无法回收，会造成资源浪费，进而影响同一进程创建的线程数量

#include "apue.h"  
#include <pthread.h>  
  
void \*  
thr\_fn1(void \*arg)  
{  
 printf("thread 1 returning\n");  
 return((void \*)1);  
}  
  
void \*  
thr\_fn2(void \*arg)  
{  
 printf("thread 2 exiting\n");  
 pthread\_exit((void \*)2);  
}  
  
int  
main(void)  
{  
 int err;  
 pthread\_t tid1, tid2;  
 void \*tret;  
  
 err = pthread\_create(&tid1, NULL, thr\_fn1, NULL);  
 if (err != 0)  
 err\_exit(err, "can't create thread 1");  
 err = pthread\_create(&tid2, NULL, thr\_fn2, NULL);  
 if (err != 0)  
 err\_exit(err, "can't create thread 2");  
 err = pthread\_join(tid1, &tret);  
 if (err != 0)  
 err\_exit(err, "can't join with thread 1");  
 printf("thread 1 exit code %ld\n", (long)tret);  
 err = pthread\_join(tid2, &tret);  
 if (err != 0)  
 err\_exit(err, "can't join with thread 2");  
 printf("thread 2 exit code %ld\n", (long)tret);  
 exit(0);  
}

上述程序输出如下：

thread 1 returning  
thread 2 exiting  
thread 1 exit code 1  
thread 2 exit code 2

### 3）pthread\_self函数

线程可以使用pthread\_self获取自身的线程ID，类似于进程中的getpid

新线程不应该根据主线程调用pthread\_create函数时传入的tid参数来获取自身ID，而是应该调用pthread\_self，因为新线程可能在主线程调用pthread\_create返回之前运行，如果读取tid，看到的是未经初始化的内容

### 4）pthread\_detach函数

该函数把指定的线程转变为**脱离状态**，通常由想让自己脱离的线程调用：pthread\_detach(pthread\_self());

一个线程或者是**可汇合**的，或者是**脱离**的：

* **可汇合**：一个可汇合线程终止时，它的线程ID和退出状态将保存到另一个线程对它调用pthread\_join。如果一个线程需要知道另一个线程什么时候终止，那就最好保持第二个线程的可汇合状态
* **脱离**：脱离的线程像守护进程，当它们终止时，所有相关资源都被释放，不能等待它们终止
  + 一般在不关心线程返回状态，希望系统在线程终止时自动清理并退出时调用pthread\_detach
  + 其它线程调用了exit，或是主线程执行return时，即便是分离的线程也会终止。换而言之，**pthread\_detach只是控制线程终止之后所发生的事情，而非何时或如何终止线程**

### 5）pthread\_exit函数

线程终止的**一个方法**：

* **status**：指定了线程的返回值。不能指向一个局部于调用线程的对象，因为线程终止时这样的对象也消失

让一个线程终止的**其它**方法：

1. **线程执行的函数返回**，在pthread\_create参数中，这个函数的返回值是一个void\*指针，它指向相应线程的终止状态
2. **被同一进程的其它线程调用pthread\_cancel取消**（该函数只是发起一个请求，目标线程可以选择忽略取消或控制如何被取消）
3. **任何线程调用return exit、\_Exit、\_exit终止时，整个进程就终止了，其中包括它的任何线程**

如果主线程调用了pthread\_exit，而非exit或return，那么其它线程将继续运行

下列程序status指向一个栈上的结构，这个栈上的对象被后来的线程覆盖：

#include "apue.h"  
#include <pthread.h>  
  
struct foo {  
 int a, b, c, d;  
};  
  
void  
printfoo(const char \*s, const struct foo \*fp)  
{  
 printf("%s", s);  
 printf(" structure at 0x%lx\n", (unsigned long)fp);  
 printf(" foo.a = %d\n", fp->a);  
 printf(" foo.b = %d\n", fp->b);  
 printf(" foo.c = %d\n", fp->c);  
 printf(" foo.d = %d\n", fp->d);  
}  
  
void \*  
thr\_fn1(void \*arg)  
{  
 struct foo foo = {1, 2, 3, 4};  
  
 printfoo("thread 1:\n", &foo);  
 pthread\_exit((void \*)&foo);  
}  
  
void \*  
thr\_fn2(void \*arg)  
{  
 printf("thread 2: ID is %lu\n", (unsigned long)pthread\_self());  
 pthread\_exit((void \*)0);  
}  
  
int  
main(void)  
{  
 int err;  
 pthread\_t tid1, tid2;  
 struct foo \*fp;  
  
 err = pthread\_create(&tid1, NULL, thr\_fn1, NULL);  
 if (err != 0)  
 err\_exit(err, "can't create thread 1");  
 err = pthread\_join(tid1, (void \*)&fp);  
 if (err != 0)  
 err\_exit(err, "can't join with thread 1");  
 sleep(1);  
 printf("parent starting second thread\n");  
 err = pthread\_create(&tid2, NULL, thr\_fn2, NULL);  
 if (err != 0)  
 err\_exit(err, "can't create thread 2");  
 sleep(1);  
 printfoo("parent:\n", fp);  
 exit(0);  
}

mac上输出如下：

thread 1:  
 structure at 0x700000080ed0  
 foo.a = 1  
 foo.b = 2  
 foo.c = 3  
 foo.d = 4  
parent starting second thread  
thread 2: ID is 123145302839296  
parent:  
 structure at 0x700000080ed0  
[1] 34604 segmentation fault ./badexit2

### 6）pthread\_equal函数

线程ID是用pthread\_t数据类型来表示的，实现的时候可以用一个结构来表示该数据类型，所以可移植的操作系统实现不能把它作为整数处理。因此必须使用一个函数来对两个线程ID进程比较

Linux 3.2.0使用无符号长整型表示pthread\_t数据类型。Solaris 10将其表示为无符号整形。FreeBSD 8.0和Mac OS X 10.6.8用一个指向pthread结构的指针来表示pthread\_t数据类型

### 7）pthread\_cancel函数

该函数可以被某一线程调用，用来请求取消同一进程中的其它线程

* 函数只是发起取消请求，目标线程可以忽略取消请求或控制如何被取消（即执行一些清理函数）

线程控制自身对pthread\_cancel函数的响应行为：[取消选项](#X8d197db28065dccf4f8a33d961e1e212b46f5d9)

### 8）pthread\_cleanup\_push和pthread\_cleanup\_pop函数

以下函数被线程调用时，可以添加或清除清理函数：

这2个函数可以被实现为宏，通常pthread\_cleanup\_push会带有一个{，而pthread\_cleanup\_pop会带有1个’}’。因此，在使用时，2个函数应该配对出现

**下列情况会调用清理函数**：

* 线程调用pthread\_exit时
* 线程响应取消请求时
* 用非零execute参数调用pthread\_cleanup\_pop时

**以下情况不会调用清理函数**；

* 线程通过return或exit终止时
* execute参数为0时

不管excute参数是否为0，pthread\_cleanup\_pop函数都会将线程清理函数栈的栈顶函数删除

以下为一个测试程序：

#include <stdio.h>  
#include <pthread.h>  
  
int exceptionFun()  
{  
 pthread\_exit((void \*)0); //会调用clean  
 //return 0; //会不会调用clean取决于exceptionFun外的pthread\_cleanup\_pop的参数  
 //exit(0); //不会调用clean  
 //\_Exit(0); //不会调用clean  
}  
  
void clean(void \*arg)  
{  
 printf("clean function\n");  
}  
  
void\* thr1(void \*arg)  
{  
 printf("thread 1 created\n");  
  
 pthread\_cleanup\_push(clean,NULL);  
  
 //pthread\_exit((void \*)0); //会调用clean，第2个线程会启动  
 //return 0; //不会调用clean，且第2个线程不会启动  
 //exit(0); //不会调用clean，且第2个线程不会启动  
 //\_Exit(0); //不会调用clean，且第2个线程不会启动  
  
 pthread\_cleanup\_pop(1); //pthread\_cleanup\_pop(0)时，如果没有在push和pop之间退出，那么不会执行clean  
 //否则，根据退出时调用的是pthread\_exit，return...决定  
}  
  
void\* thr2(void \*arg)  
{  
 printf("thread 2 created\n");  
  
 pthread\_cleanup\_push(clean,NULL);  
  
 exceptionFun();  
  
 pthread\_cleanup\_pop(1);  
}  
  
int main()  
{  
 pthread\_t tid;  
 pthread\_create(&tid,NULL,thr1,NULL);  
 pthread\_join(tid,NULL);  
 pthread\_create(&tid,NULL,thr2,NULL);  
 pthread\_join(tid,NULL);  
 return 0;  
}

## 2.线程同步

* 1）**互斥锁**
  + [互斥锁的初始化与销毁](#X5f5b0734830deb08f5277363d8c5aae9a528f6f)
  + [互斥锁的加锁与解锁](#Xe56d2728f05bfaffce6c0979fb071ece17ce43c)
  + [互斥锁的定时加锁](#X3513a470b66cf518877fe85f2f3acf16429f581)
* 2）**读写锁**
  + [读写锁的初始化与销毁](#Xb7d7a911fe65579f0c1f292d183bd8a2bf1be25)
  + [读写锁的加锁与解锁](#Xdcba7c47ab461ccde2806314fb2feb20a16b5b2)
  + [读写锁的定时加锁](#X537bfa0c0ce061c533a58aca420a59a95b5c428)
* 3）**条件变量**
  + [条件变量的初始化与销毁](#Xc11d67aa4252b9d7285b0a910e119fdd136808a)
  + [等待某个条件变量](#Xf0d55035b2e1d44cde67033cf4d69aadfa82d8e)
  + [通知条件已经满足](#X927cc41abd8b83ae09baf9bd0f8fbcdea647c7b)
* 4）**自旋锁**
  + [自旋锁的初始化与销毁](#X29127687eb8508938a3738f8c0d444b3cfaf458)
  + [自旋锁的加锁与解锁](#X1b3534c4aa39b02df7fe782c23c22882b411d69)
* 5）**屏障**
  + [屏障的初始化与销毁](#Xfa5f6830de5b4f60abea4e55a6ac7ad14ea20c3)
  + [增加到达屏障点的线程](#X95e72b0945be1baf00209760ef998b0a3b3b5d4)

### 2.1 互斥锁

也称为**互斥量**

多线程编程中，[多个线程可能修改相同的变量，导致错误发生](https://github.com/arkingc/unpv13e/blob/master/threads/example01.c)。互斥锁可以用于保护共享变量：访问共享变量的前提条件是持有该互斥锁，**按照Pthread，互斥锁的类型为pthread\_mutex\_t的变量**

### 1）互斥锁的初始化与销毁

* **如果某个互斥锁变量是静态分配的，必须把它初始化为常值PTHREAD\_MUTEX\_INITIALIZER**
* **如果在共享内存区中分配一个互斥锁，必须通过调用pthread\_mutex\_init函数在运行时初始化，此时在释放内存前需要调用pthread\_mutex\_destroy**
* attr：互斥锁的属性。为NULL时表示使用默认的属性来初始化

如果释放互斥锁时，有一个以上的线程阻塞，那么**所有该锁上的阻塞线程都会变成可运行状态**，第一个变为运行的线程就可以对互斥锁加锁，其它线程就会看到互斥锁依然是锁着的，只能回去再次等待它重新变为可用。这种方式下，每次只有一个线程可以向前执行

### 2）互斥锁的加锁与解锁

* **mptr**
  + **pthread\_mutex\_lock**锁住mptr指向的互斥锁，如果已经上锁，调用线程将阻塞
  + **pthread\_mutex\_trylock**和前者类似，如果已经上锁，不会阻塞，直接返回EBUSY
  + **pthread\_mutex\_unlock**将mptr指向的互斥锁解锁

[使用互斥锁解决修改相同变量的问题](https://github.com/arkingc/unpv13e/blob/master/threads/example02.c)（本书作者测试这个程序和前面有问题的版本运行的时间，差别是10%，说明互斥锁并不会带来太大的开销）

**使用互斥锁时避免死锁**：

* 如果线程试图对一个互斥锁加锁两次，那么它自身就会陷入死锁状态
* 如果使用多个互斥锁，并且每个线程取得其中一个，阻塞于对另一个的请求上，也会死锁
  + 1）可以通过控制加锁的顺序来防止
  + 2）如果尝试获取另一个锁时失败，那么释放自己占有的锁，过一段时间再试

### 3）互斥锁的定时加锁

该函数试图对一个互斥锁进行加锁，如果互斥锁已经上锁，那么线程会阻塞到参数tsptr指定的时刻（这个时间是一个绝对时间，即某个时刻，而不是一个等待时间段）。如果仍为获得互斥锁，那么返回ETIMEDOUT

Max OS X 10.6.8还没有支持该函数

### 2.2 读写锁

也称作**共享互斥锁**：当读写锁是读模式锁住时，可以说成是共享模式锁住的；当它是写模式锁住时，可以说成是以互斥模式锁住的

**读写锁与互斥锁类似，不过读写锁允许更高的并行性**：

* 互斥锁要么是**锁住状态**，要么是**不加锁状态**，而且一次只有一个线程可以对其加锁
* 读写锁可以有3种状态
  + **读模式下加锁状态**：多个线程可以用时占有读模式的读写锁，此时任何试图以写模式对该读写锁加锁的线程都会阻塞，直到所有的线程释放它们的读锁为止（通常这种情况下会阻塞随后的读模式锁请求，从而避免读模式长期占用，而等待的写模式锁请求一直得不到满足）
  + **写模式下加锁状态**：一次只有一个线程可以占有写模式的读写锁
  + **不加锁状态**

**读写锁非常适合于对数据结构读的次数远大于写的情况**

### 1）读写锁的初始化与销毁

读写锁在使用之前必须初始化，释放它们底层的内存之前必须销毁：

* attr：锁的初始化属性，设为NULL时使用默认属性

Single UNIX Specification在XSI扩张中定义了**PTHREAD\_RWLOCK\_INITIALIZER**常量。如果默认属性就足够的话，可以用它对静态分配的读写锁进行初始化

### 2）读写锁的加锁与解锁

Single UNIX Specification还提供了下列版本：

### 3）读写锁的定时加锁

与互斥锁一样，Single UNIX Specification提供了带有超时的读写锁加锁函数：

该函数试图对一个读写锁进行加锁，如果读写锁已经上锁，那么线程会阻塞到参数tsptr指定的时刻（这个时间是一个绝对时间，即某个时刻，而不是一个等待时间段）。如果仍为获得读写锁，那么返回ETIMEDOUT

### 2.3 条件变量

条件变量可以在某个条件发生之前，将线程投入睡眠

**按照Pthread，条件变量是类型为pthread\_cond\_t的变量**

### 1）条件变量的初始化与销毁

* attr：条件变量的初始化属性，设为NULL时使用默认属性

**如果条件变量是静态分配的，那么可以使用PTHREAD\_COND\_INITIALIZER初始化**

### 2）等待某个条件变量

* **pthread\_cond\_wait**函数等待cond指向的条件变量，投入睡眠之前会释放mutex指向的互斥锁，唤醒后会重新获得mutex指向的互斥锁
* **pthread\_cond\_timewait**在给定的时间内等待条件发生，超时会重新获取mutex指向的互斥锁并返回一个错误码（这个时间仍然是一个绝对时间）

两个函数成功返回时，线程需要重新计算条件，因为另一个线程可能已经在运行并改变了条件

为什么每个条件变量都要关联一个互斥锁呢？因为”条件“（这里不是指条件变量）通常是线程之间共享的某个变量的值。允许不同线程设置和测试该变量要求有一个与该变量关联的互斥锁

### 3）通知条件已经满足

* **pthread\_cond\_signal**：至少能唤醒一个等待该条件的线程
* **pthread\_cond\_broadcast**：能唤醒所有等待该条件的线程

### 2.4 自旋锁

自旋锁与互斥锁类似，区别是：自旋锁在获取锁之前一直处于忙等（自选）阻塞状态。因为忙等会消耗大量CPU，因此适用于锁持有时间不长（即操作可以较快完成）的场景

事实上，有些互斥锁的实现在试图获取互斥锁的时候会自旋一小段时间，只有在自旋计数到达某一阈值的时候才会休眠

### 1）自旋锁的初始化与销毁

* pshared：进程共享属性，表明自旋锁是如何获取的
  + 如果设为PTHREAD\_PROCESS\_SHARED，则自旋锁能被可以访问锁底层内存的线程所获取，即便那些线程属于不同的进程
  + 否则设为PTHREAD\_PROCESS\_PRIVATE，则自旋锁只能被初始化该锁的进程内部的线程所访问

### 2）自旋锁的加锁与解锁

* **pthread\_spin\_lock**：在获取锁之前一直自旋
* **pthread\_spin\_trylock**：如果不能获取锁，就立即返回EBUSY错误
* **pthread\_spin\_unlock**：对自旋锁解锁。如果试图对没有加锁的自旋锁进行解锁，结果是未定义的

**在持有自旋锁时，不要调用可能会进入休眠状态的函数。如果调用了这些函数，会浪费CPU资源，因为其他线程需要获取自旋锁需要等待的时间就延长了**

### 2.5 屏障

**屏障是用户协调多个线程并行工作的同步机制。屏障允许每个线程等待，直到所有的合作线程都到达某一点，然后从该点继续执行**

pthread\_join就是一种屏障，允许一个线程等待，直到另一个线程退出

### 1）屏障的初始化与销毁

* count：在允许所有线程继续运行之前，必须到达屏障的线程数目
  + 假设主线程希望开启n个线程进行排序，每个线程排序数组的一部分，所有n个线程排好序后主线程进行归并，那么count应该为n+1
* attr：屏障对象的属性，设为NULL时使用默认属性初始化屏障

### 2）增加到达屏障点的线程

该函数用以表明，线程已完成工作，准备等待所有其他线程赶上来

* 如果线程不是(满足屏障要求数目的)最后一个线程：会进入休眠
* 如果线程是(满足屏障要求数目的)最后一个线程：唤醒所有因为屏障进入休眠的线程

**一旦到达屏障计数值，而且线程处于非阻塞状态，屏障就可以被重用**

**只有先调用pthread\_barrier\_destroy函数，接着又调用pthread\_barrier\_init并传入一个新的计数值，否则屏障计数无法改变**

# 七.线程控制

## 1.线程限制

Single UNIX Specification定义了与线程操作有关的一些限制，这些限制可以通过sysconf函数进程查询：

下图为apue描述的4种操作系统实现中的限制值：

这些限制的使用是为了增强应用程序在不同的操作系统实现之间的可移植性

我的环境(Max OS X 10.11.6)下：

printf("%ld\n",sysconf(\_SC\_THREAD\_STACK\_MIN)); //8192  
printf("%ld\n",sysconf(\_SC\_THREAD\_THREADS\_MAX)); //-1（正常值，因为没有设置errno）

## 2.线程属性

### 2.1 线程属性

**线程属性用以初始化线程，定义在结构体pthread\_attr\_t中**，一般包括：

* 分离状态属性
* 线程栈末尾的警戒缓冲区大小
* 线程栈的最低地址
* 线程栈的最小长度(字节)

下图为POSIX.1定义的线程属性在apue描述的4种操作系统实现中的支持情况：

POSIX.1还为线程执行调度选项定义了额外的属性，用以支持实时应用，但这里不打算讨论这些属性

在**MAC OS X 10.11.6**下的测试：

pthread\_attr\_t attr;  
pthread\_attr\_init(&attr);  
  
pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_DETACHED);  
  
int state;  
size\_t stack\_size;  
size\_t guardsize;  
  
pthread\_attr\_getdetachstate(&attr,&state);  
pthread\_attr\_getstacksize(&attr,&stack\_size);  
pthread\_attr\_getguardsize(&attr,&guardsize);  
  
printf("state:");  
if(state == PTHREAD\_CREATE\_DETACHED){  
 printf("PTHREAD\_CREATE\_DETACHED\n"); //PTHREAD\_CREATE\_DETACHED  
}  
else if(state == PTHREAD\_CREATE\_JOINABLE){  
 printf("PTHREAD\_CREATE\_JOINABLE\n");  
}  
printf("stack\_size:%u\n",stack\_size); //524288(512\*1024)bytes  
printf("guardsize:%u\n",guardsize); //4096  
  
pthread\_attr\_destroy(&attr);

### 1）线程属性的初始化和销毁

如果pthread\_attr\_init的实现对属性对象的内存空间是动态分配的，那么pthread\_attr\_destroy会释放该内存空间。除此之外，pthread\_attr\_destroy还会用无效的值初始化属性对象，因此，如果该属性对象被误用，将会导致pthread\_create函数返回错误码

要注意pthread\_attr\_destroy的返回值，如果忽略其错误返回可能出现的最坏情况是，如果pthread\_attr\_init已经分配了内存空间，就会有少量的内存泄露。另一方面，如果pthread\_attr\_init成功地对线程属性进行了初始化，但之后的pthread\_attr\_destroy清理工作失败，那么将没有任何补救策略，因为线程属性结构对应用程序来说是不透明的，可以对线程属性结构进行清理的唯一接口是pthread\_attr\_destroy，但它失败了

### 2）分离属性的获取与设置

* 函数pthread\_detach是让已经存在的某个线程变为分离状态
* 如果想让线程从创建开始就处于分离，可以设置分离属性，然后使用该属性调用pthread\_create
  + PTHREAD\_CREATE\_DETACHED：分离状态
  + PTHREAD\_CREATE\_JOINABLE：正常状态

### 3）栈属性的获取与设置

对于进程来说，虚地址空间的大小是固定的。因为进程中只有一个栈，所以它的大小通常不是问题。但对于线程来说，同样大小的虚地址空间必须被所有的线程栈共享。如果应用程序使用了许多线程，以致这些线程栈的累计大小超过了可用的虚地址空间，就需要减小默认的线程栈大小。另一方面，如果线程调用的函数分配了大量的自动变量，或者调用的函数涉及许多很深的栈帧，那么需要的栈大小可能要比默认的大

如果线程栈的虚地址空间都用完了，那么可以使用malloc或mmap来为可替代的栈分配空间，并用下列函数设置或者获取线程栈的最低内存地址：

* stackaddr：线程栈的最低内存地址（不一定是栈的开始地址。对于一个给定的处理器结构来说，如果栈是从高地址向低地址方向增长的，那么stackaddr线程属性将是栈的结尾位置，而不是开始位置）

### 4）栈大小属性的获取与设置

* stacksize：栈大小，不能小于PTHREAD\_STACK\_MIN

如果希望改变默认的栈大小，但又不想自己处理线程栈的分配问题，使用pthread\_attr\_setstacksize非常有用

### 5）境界缓存大小的获取与设置

* guardsize：控制着线程栈末尾之后用于避免栈溢出的扩展内存的大小
  + 默认值：由具体实现决定，常用值是系统页大小
  + 0：表示不允许属性的这种特征值行为发生：在在这种情况下，不会提供警戒缓冲区（如果修改了线程属性stackaddr，系统就会认为我们将自己管理栈，进而使栈警戒缓存区机制无效，等同于把该值设置为0）
* **如果guardsize线程属性被修改了，操作系统可能会把它取为页大小的整数倍**
* **如果线程的栈指针溢出到警戒区域，应用程序就可能通过信号接收到出错信息**

### 2.2 取消选项

有两个线程属性没有包含在pthread\_attr\_t结构中：

1. **可取消状态**属性（控制线程可否被取消）
2. **可取消类型**属性（控制线程何时可取消）

**这两个属性影响着线程在响应pthread\_cancel函数调用时所呈现的行为**

### 1）可取消状态属性

* state：新的可取消状态
* oldstate：旧的可取消状态

新旧状态的设置是一个原子操作

pthread\_cancel调用并不等待线程终止。**默认情况下，线程在取消请求发出后还是继续运行，直到线程到达某个取消点**。取消点是线程检查它是否被取消的一个位置，POSIX.1保证线程调用下列函数时，取消点都会出现：

可取消状态可以是下面的值：

* PTHREAD\_CANCEL\_ENABLE：默认值
* PTHREAD\_CANCEL\_DISABLE：对pthread\_cancel的调用并不会杀死线程。相反，**取消请求对这个线程来说还处于挂起状态**，当取消状态再次变为PTHREAD\_CANCEL\_ENABLE时，线程将在下一个取消点上对所有挂起的取消请求进行处理

如果线程长时间不会调用前面所述函数进入一个取消点，可以通过**下列函数添加取消点**：

下列2个条件满足时，该函数会使线程被取消：

1. 存在处于挂起状态的取消请求
2. 取消状态为PTHREAD\_CANCEL\_ENABLE

否则，该函数没有效果

### 2）可取消类型属性

* type：新的可取消类型
* oldtype：旧的可取消类型
  + PTHREAD\_CANCEL\_DEFERRED：默认值。推迟取消，即遇到取消点才取消
  + PTHREAD\_CANCEL\_ASYNCHRONOUS：线程可在任意时间取消，不是非得遇到取消点才能被取消

## 3.同步属性

就像线程具有属性一样，线程的同步对象也有属性。**在使用某种线程同步机制时，同步属性用以初始化相应的同步机制。比如互斥锁属性用以初始化互斥锁、条件变量属性用以初始化条件变量…**

* [互斥锁](#Xf2336c27c283bee844403d2b46eba0463a85c46)
  + 进程共享属性
  + 健壮属性
  + 类型属性
* [读写锁](#X4740ed423244f4ee4ffb8cc5044d230e3a732c9)
  + 进程共享属性
* [条件变量](#X439af1d4bfdc99c48605ee93cc3488ad3ea44e7)
  + 进程共享属性
  + 时钟属性
* [屏障](#Xd115f8d27d38c67c161146db6e9b5e109dc4a1d)
  + 进程共享属性

**进程共享属性控制着同步对象是否可用于进程与进程之间**

### 3.1 互斥锁属性

**互斥锁属性用pthread\_mutexattr\_t结构表示**

3个值得注意的互斥锁属性：

1. **进程共享属性**
2. **健壮属性**
3. **类型属性**

### 1）互斥锁属性的初始化和销毁

* 获得默认互斥锁属性
  + 调用pthread\_mutex\_init时，传入**空指针**，即用默认互斥锁属性来初始化互斥锁
  + 使用**PTHREAD\_MUTEX\_INITIALIZER**常量
  + 调用pthread\_mutexattr\_init函数

### 2）进程共享属性的获取和设置

这个属性是可选的。可以通过2种方式检查是否支持该属性：

1）检查系统中是否定义了\_POSIX\_THREAD\_PROCESS\_SHARED符号判断这个平台是否支持进程共享这个属性； 2）可以在运行时把\_SC\_THREAD\_PROCESS\_SHARED参数传给sysconf函数进行检查）

* pshared
  + PTHREAD\_PROCESS\_PRIVATE：默认的行为。进程中的多个线程可以访问同一个同步对象。该值允许pthread线程库提供更有效的互斥锁实现。在多个进程共享多个互斥锁的情况下，pthread线程库可以限制开销较大的互斥锁实现
  + PTHREAD\_PROCESS\_SHARED：允许相互独立的多个进程把同一个内存数据块映射到它们各自独立的地址空间中。就像多个线程访问共享数据一样，多个进程访问共享数据通常也需要同步。如果设为该值，那么初始化得到的互斥锁（该锁从多个进程彼此之间共享的内存数据块中分配得到）就能用于这些进程的同步

### 3）健壮属性的获取和设置

互斥锁的健壮属性与在**多个进程间**共享的互斥锁有关。意味着，当持有互斥锁的进程终止时，需要解决互斥锁状态恢复的问题。这种情况发生时，互斥锁处于锁定状态，恢复起来很困难。其它阻塞在这个锁的进程将会一直阻塞下去

* robust
  + PTHREAD\_MUTEX\_STALLED：意味着持有互斥锁的进程终止时不需要采取特别的动作。这种情况下，使用互斥锁后的行为是未定义的，等待该互斥锁解锁的应用程序会被有效地”拖住“
  + PTHREAD\_MUTEX\_ROBUST：设为这个值时，当一个线程调用pthread\_mutex\_lock获取锁，但该锁被另一个进程持有，但它终止时并没有对该锁进行解释，此时线程会阻塞，从pthread\_mutex\_lock返回的值为EOWNERDEAD而不是0。应用程序可以通过这个特殊的返回值获悉这种情况，然后进行恢复

如果应用状态无法恢复，在线程对互斥锁解锁后，该互斥锁将处于永久不可用状态。为了避免这样的问题，线程可以通过调用pthread\_mutex\_consistent函数，指明该互斥锁相关的状态在互斥锁解锁以前是一致的

如果线程没有先调用pthread\_mutex\_consistent就对互斥锁进行了解锁，那么其它试图获取该互斥锁的阻塞线程就会得到错误码ENOTRECOVERABLE。如果发生这种情况，互斥锁将不再可用。线程通过提前调用pthread\_mutex\_consistent，就能让互斥锁正常工作，这样它就可以持续被使用

### 4）类型属性的获取和设置

类型属性控制着互斥锁的**锁定特性**。POSIX.1定义了4种类型：

* PTHREAD\_MUTEX\_NORMAL：标准类型，不做任何特殊的错误检查或死锁检测
* PTHREAD\_MUTEX\_ERRORCHECK：提供错误检查
* PTHREAD\_MUTEX\_RECURSIVE：允许同一线程在互斥锁解锁之前对该互斥锁进程多次加锁。递归互斥锁维护锁的计数，在解锁次数和加锁次数不相同的情况下不会释放锁
* PTHREAD\_MUTEX\_DEFAULT：该类型可以提供**默认特性和行为**。操作系统在实现的时候可以把这种类型自由地映射到其它互斥锁类型中的一种

下图为不同的类型属性和行为：

不占用时解锁：一个线程对被另一个线程加锁的互斥锁进行解锁

### 3.2 读写锁属性

**读写锁属性用pthread\_rwlockattr\_t结构表示**

**进程共享属性**是读写锁的唯一属性

### 1）读写锁属性的初始化与销毁

### 2）进程共享属性的获取与设置

进程共享属性是读写锁的唯一属性，可以通过下列函数获取与设置

虽然POSIX只定义了一个读写锁属性，但不同平台的实现可以自由地定义额外的、非标准的属性

### 3.3 条件变量属性

**条件变量属性用pthread\_condattr\_t结构表示**

Single UNIX Specification目前定义了条件变量的2个属性：

* **进程共享属性**
* **时钟属性**

### 1）条件变量属性的初始化与销毁

### 2）进程共享属性的获取与设置

与其他的同步属性一样，条件变量支持进程共享属性。它控制着条件变量是可以被单进程的多个线程使用，还是可以被多进程的线程使用

### 3）时钟属性的获取与设置

时钟属性控制计算pthread\_cond\_timedwait函数的超时参数tsptr采用的是哪个时钟

* **pthread\_condattr\_getclock**：获取可被用于pthread\_cond\_timewait函数的时钟ID
* **pthread\_condattr\_setclock**：对时钟ID进行修改

### 3.4 屏障属性

**条件变量属性用pthread\_barrierattr\_t结构表示**

**进程共享属性**是屏障的唯一属性

### 1）屏障属性的初始化与销毁

### 2）进程共享属性的获取与设置

目前定义的屏障属性只有进程共享属性，它控制着屏障是可以被多进程的线程使用，还是只能被初始化屏障的进程内的多线程使用

## 4.线程特定数据

线程特定数据也称为线程私有数据，是存储和查询某个线程相关数据的一种机制。它是每个线程私有的数据副本，因此不需要担心与其它线程的同步访问问题

* **无法简单的分配一个每线程数据数组**：因为无法通过线程ID去定位数组中的某个具体数据。因为线程ID并不能保证是小而连续的整数。即使是，我们可能还希望有一些额外保护，防止某个线程的数据与其他线程的数据相混淆（比如线程间的数据可能越界写？）
* **线程特定数据提供了让基于进程的接口适应多线程环境的机制**。比如errno，线程出现以前，errno定义为进程上下文中全局可访问的整数。为了让线程也能使用那些原本基于进程的系统调用和库例程，errno被重新定义为线程私有数据。这样线程之间不会相互影响

一个线程没有办法阻止另一个线程访问它的数据。线程特定数据也不例外。虽然底层的实现部分并不能阻止这种访问能力，但管理线程特定数据的函数可以提高线程间的数据独立性，使得线程不太容易访问到其它线程的线程特定数据

**每个系统支持有限数量的线程特定数据元素，POSIX要求这个限制不小于128(每个进程)**，**系统**为**每个进程**维护一个称之为Key结构的数组，如下图：

* **标志**：指示这个数组元素是否正在使用（所有标志初始化为”不在使用“）

除了**进程范围**的Key结构数组外，**系统**还在**进程内**维护关于每个线程的多条信息，记录在Pthread结构（由系统维护）中：

pKey数组的所有元素都被初始化为空指针。这128个指针是和进程内的128个可能的索引（称为”键“）逐一关联的值

一般步骤如下：

* 定义了一个全局静态的pthread\_key\_t变量，表示键
* 其中一个线程调用pthread\_key\_create从进程的key数组创建一个未使用的键（为了防止被多次调用，可以使用pthread\_once）
* 所有线程可以使用这个新键通过pthread\_getspecific索引自己的pkey数组的相应位置
  + 如果返回一个空指针，说明相应的线程特定数据元素不存在，可以调用malloc分配，然后调用pthread\_setspecific将这个新分配的线程特定数据的指针保存在pkey数组中
  + 如果返回一个非空指针，那么可以使用这个线程特定数据
* 调用pthread\_key\_create函数时指定的析构函数会释放保存在每个线程pkey数组中的线程特定数据

### 1）pthread\_once和pthread\_key\_create函数

**pthread\_key\_create函数**：

* **keyptr**：创建一个新的线程特定数据元素时，系统搜索其Key结构数组找到第一个不在使用的元素，元素的索引（0~127）记录在keyptr成员中，**作为返回值**。随后，线程可以利用记录在keyptr中的索引，在Pthread结构pkey数组的对应索引位置处存储一个指针，这个指针指向malloc动态分配的内存
* **destructor**：指向析构函数的函数指针，当一个线程终止时，系统扫描该线程的pKey数组，为每个非空的pkey指针调用相应的析构函数，释放其指向的动态内存。如果为NULL，表明没有析构函数与该键关联

**下列情况会调用析构函数**：

* 当线程调用pthread\_exit时
* 当线程执行返回，正常退出时
* 线程取消时，只有在最后的清理处理程序返回之后，析构函数才会被调用

**下列情况不会调用析构函数**：

* 线程调用了exit、\_exit、\_Exit或abort时
* 出现其他非正常的退出时

线程退出时，线程特定数据的析构函数将按照操作系统实现中定义的顺序被调用。当所有的析构函数都调用完成之后，系统会检查是否还有非空的线程特定数据值与键关联，如果有的话，再次调用析构函数。这个过程将会一直重复到线程所有的键都为空，或者已经做了PTHREAD\_DESTRUCTOR\_ITERATIONS中定义的最大次数的尝试

**pthread\_once函数**：

* **onceptr**：onceptr参数指向的变量中的值，确保init参数所指的函数在进程范围内只被调用一次
* **init**：进程范围内，对于一个给定的键，pthread\_key\_create只能被调用一次。所以init可以指向一个pthread\_key\_create函数，通过onceptr参数确保只调用一次

### 2）pthread\_getspecific和pthread\_setspecific函数

* pthread\_getspecific函数返回对应指定键的指针
* pthread\_setspecific函数在Pthread结构中把对应指定键的指针设置为指向分配的内存

### 3）pthread\_key\_delete函数

该函数用来取消键与线程特定数据值之间的关联。它并不会激活与键关联的析构函数。要释放任何与键关联的线程特定数据值的内存，需要在应用程序中采取额外的步骤

## 5.线程和信号

每个线程都有自己的信号屏蔽字，但是信号的处理是进程中所有线程共享的。这意味着单个线程可以阻止某些信号，但当线程修改了与某个给定信号相关的处理行为以后，所有的线程都必须共享这个处理行为的改变。这样，如果一个线程选择忽略某个给定信号，那么另一个线程就可以通过以下两种方式撤销上述信号选择：恢复信号的默认处理行为，或者为信号设置一个新的信号处理程序

进程中的信号是递送到单个线程的：

* 如果一个信号与硬件故障相关，那么该信号一般会被发送到引起该事件的线程中去
* 其它的信号则被发送到任意一个线程

### 5.1 阻止信号发送

sigprocmask的行为在多线程的进程中没有定义，线程必须使用pthread\_sigmask：

* how：set与线程信号屏蔽字的作用方式
  + SIG\_BLOCK：把信号集set添加到线程信号屏蔽字中
  + SIG\_SETMASK：用信号集set替换线程的信号屏蔽字
  + SIG\_UNBLOCK：从线程信号屏蔽字中移除信号集set
* set：信号集
* oset：如果不为NULL，保存线程之前的信号屏蔽字

线程可以通过把set设为NULL，把oset参数设为sigset\_t结构的地址，来获取当前的信号屏蔽字。这种情况下，how参数会被忽略

### 5.2 等待信号

线程调用下列函数**等待一个或多个信号出现**

* set：等待的信号集（**如果信号集中的某个信号在函数调用时处于等待挂起状态，那么函数将无阻塞地返回**）
* signop：指向的整数包含发送信号的数量

注意：

* 在返回之前，sigwait将从进程中移除那些处于挂起等待状态的信号。如果具体实现支持排队信号，并且信号的多个实例被挂起，那么sigwait将会移除该信号的一个实例，其他的实例还要继续排队
* 为了避免错误行为发生，线程在调用sigwait前，必须阻塞那些它正在等待的信号。sigwait会原子地取消信号集的阻塞状态，直到有新的信号被递送。在返回之前，sigwait将恢复线程的信号屏蔽字。如果信号在sigwait被调用的时候没有被阻塞，那么在线程完成对sigwait的调用之前会出现一个时间窗，在这个时间窗中，信号就可以被发送给线程
* 使用sigwait的好处在于它可以简化信号处理，允许把异步产生的信号用同步的方式处理。为了防止信号中断线程，可以把信号加到每个线程的信号屏蔽字中。然后可以安排专用线程处理信号。这些专用线程可以进行函数调用，不需要担心在信号处理程序中调用哪些函数是安全的，因为这些函数调用来自正常的线程上下文，而非会中断线程正常执行的传统信号处理程序
* 如果多个线程在sigwait的调用中因等待同一个信号而阻塞，那么在信号递送的时候，就只有一个线程可以从sigwait中返回。如果一个信号被捕获，而且一个线程正在sigwait调用中等待同一信号，那么这时将由操作系统来决定以何种方式递送信号。操作系统可以让sigwait返回，也可以激活信号处理程序，但这两种情况不会同时发生

要**把信号发送给进程**，可以调用kill。要**把信号发送给线程**，可以调用pthread\_kill：

可以传一个0值的signo来检查线程是否存在。如果信号的默认处理动作是终止该进程，那么把信号传递给某个线程仍然会杀死整个进程

闹钟定时器是进程资源，并且所有的线程共享相同的闹钟。所以，进程中的多个线程不可能互不干扰地使用闹钟定时器

## 6.线程和fork

当线程调用fork时，就为子进程创建了整个进程地址空间的副本：

* 子进程通过继承整个地址空间的副本，还从父进程那儿继承了每个互斥量、读写锁和条件变量的状态
* 子进程内部只存在一个线程，它由父进程中调用fork的线程的副本构成。如果父进程中的线程占有锁，子进程将同样占有这些锁。问题是子进程并不包含占有锁的线程的副本，所以子进程没有办法知道它占有了哪些锁、需要释放哪些锁

如果fork后马上调用其中一个exec函数，就能避免这样的问题。这种情况下，旧的地址空间被丢弃，所以锁的状态无关紧要

在多线程的进程中，为了避免不一致状态的问题，POSIX.1声明，在fork返回和子进程调用其中一个exec之间，子进程只能调用**异步信号安全**的函数。这就限制了在调用exec之前子进程能做什么，但不涉及子进程中锁状态的问题

**要清除锁状态**，可以调用pthread\_atfork**建立fork处理程序**：

**该函数可以安装清除锁的函数，最多3个**：

* prepare：
  + 由父进程在fork创建子进程前调用
  + 任务是获取父进程定义的所有锁
* parent：
  + fork创建子进程以后，返回之前在父进程上下文中调用
  + 任务是对prepare fork处理程序获取的所有锁进行解锁
* child：
  + fork返回之前在子进程上下文中调用
  + 与prepare fork处理程序一样，也必须释放prepare fork处理程序获取的所有锁

如果不需要使用其中某个处理程序，可以给特定的处理程序参数传入空指针，它就不会起任何作用了

可以多次调用pthread\_atfork函数从而设置多套fork处理程序。使用多个fork处理程序时，处理程序的调用顺序并不相同

* parent和child fork处理程序是以它们注册时的顺序进行调用的
* 而prepare fork处理程序的调用顺序与它们注册时的顺序相反

这样可以允许多个模块注册它们自己的fork处理程序，而且可以保持锁的层次

## 7.线程和I/O

在多线程进程中，因为线程共享相同的文件描述符，所以应该使用pread和pwrite而不是read和write，使得偏移量的设定和数据的读取成为一个原子操作

# 八.高级I/O

下图为UNIX下5种I/O模型的比较

* **同步I/O**：I/O操作过程中会导致请求进程阻塞，直到I/O操作完成
  + **1）阻塞式I/O**：进程会在整个调用开始到结束一直阻塞
  + **2）非阻塞式I/O**：非阻塞指I/O调用不导致进程睡眠。如果描述符未就绪，返回一个错误，因为需要轮询直到描述符就绪，因此会耗费大量CPU
  + **3）I/O复用**：I/O复用的优势在于可以等待多个描述符就绪
  + **4）信号驱动I/O**：内核在描述符就绪时发送SIGIO信号进行通知。优势在于等待数据报到达期间进程不被阻塞。主循环可以继续执行
* **异步I/O**：不导致请求进程阻塞
  + **5）异步I/O**：这种模型与信号驱动模型的主要区别在：信号驱动I/O是由内核通知我们何时启动一个I/O操作，而异步I/O模型是由内核通知我们I/O操作何时完成

## 2.记录锁

商用UNIX系统提供了记录锁机制

**记录锁的功能是：当第一个进程正在读或修改文件的某个部分时，使用记录锁可以阻止其他进程修改同一文件区**

POSIX.1通过[fcntl函数](#X559855b9a7704054e3fe4a04172145fd290dae4)提供记录锁功能

此时cmd参数是下面3个：

* F\_GETLK：判断由flockptr所描述的锁是否会被另外一把锁锁排斥（阻塞）。如果存在一把锁，它阻止创建由flockptr所描述的锁，则该现有锁的信息将重写flockptr指向的信息。如果不存在这种情况，则除了l\_type设置为F\_UNLCK之外，flockptr所指向结构中的其他信息保存不变
* F\_SETLK：设置由flockptr所描述的锁，设置失败会出错返回。此命令也用来清除由flockptr指定的锁（l\_type为F\_UNLCK）
* F\_SETLKW：该命令是F\_SETLK的阻塞版本，命令中的W表示等待。如果所请求的读锁或写锁因另一个进程当前已经对所请求区域的某部分进行了加锁而不能被授予，那么调用进程会被置为休眠。如果所请求创建的锁已经可用，或者休眠由信号中断，则该进程被唤醒

第三个参数是一个指向flock结构的指针：

struct flock{  
 short l\_type; /\* F\_RDLCK,F\_WRLCK,or F\_UNLCK \*/  
 short l\_whence; /\* SEEK\_SET,SEEK\_CUR,or SEEK\_END \*/  
 off\_t l\_start; /\* offset in bytes,relative to l\_whence \*/  
 off\_t l\_len; /\* length, in bytes; 0 means lock to EOF \*/  
 pid\_t l\_pid; /\* returned with F\_GETLK \*/  
};

* l\_type
  + F\_RDLCK：共享读锁
  + F\_WRLCK：独占性写锁
  + F\_UNLCK：解锁一个区域
* lstart及l\_whence：要加锁或解锁区域的起始字节偏移量
* l\_len：区域的字节长度
* l\_pid：由 F\_GETLK 返回，表示持有锁的进程的 ID

关于加锁或解锁区域的说明还要注意下列几项规则：

* 指定锁的范围起始偏移量的两个元素l\_whence,l\_start与[lseek函数](#X700fefd172d373086f3bc5da02517cfabd56a42)中最后两个参数类似
* 锁可以在当前文件尾端处开始，或者越过尾端处开始，但是不能在文件起始位置之前开始
* 如果l\_len为 0 ，则表示锁的范围可以扩展到最大可能偏移量。这意味着不论向该文件中追加写了多少数据，它们都可以处于锁的范围内
* 为了对整个文件加锁，我们可以设置l\_whence,l\_start指向文件的起始位置，并指定l\_len为 0

### 1）读写锁的关系

有两种类型的锁：**共享性读锁**(l\_type=F\_RDLCK)，**独占性写锁**(l\_type=F\_WRLCK)。互斥规则为：

* **任意多个进程可以在一个给定的字节上有一把共享性读锁。如果在一个给定字节上已经有了一把或者多把读锁，则不能在该字节上添加写锁**（当进程A在文件的某个区间设置了一把读锁时，如果进程B试图在同一个区间设置一把写锁，则进程B会被阻塞。现在如果有另外进程C试图在同一个区间设置一把读锁，则进程C能不能顺利加读锁？POSIX.1并没有说明这一情况。如果进程C允许添加读锁，则很可能后续大量的读锁被添加从而导致进程B的写锁得不到设置从而进程B被饿死）
* **在一个给定字节上只能有一个进程有一把独占性写锁。如果在一个给定字节上已经有了一把写锁，则不能在该字节上添加任何读锁或者写锁**

注意：该规则适用于不同进程提出的锁请求，不适合单个进程提出的多个锁请求。如果某个进程对一个文件区间已经有了一把锁，后来该进程又企图在同一个文件区间再加一把锁，则新锁将会替换旧锁

### 2）锁与文件描述符类型的关系

* 加读锁时，该文件描述符必须是读打开的
* 加写锁时，该文件描述符必须是写打开的

### 3）一般性步骤

通常使用F\_GETLK命令来测试能否建立一把锁，然后用F\_SETLK或者F\_SETLKW试图建立那把锁。但是注意到：这二者不是一个原子操作！因此不能保证在这两次fcntl调用之间，可能另有一个进程插入并建立一把相同的锁从而导致本进程建立锁的过程失败

### 4）相邻区域的组合与分裂

第100~199字节是加锁区：

解锁第150字节，内核将维持2把锁：

如果又对第150字节加锁，那么系统会把3个相邻的加锁区合并成一个：

### 5）记录锁的隐含继承和释放

* 记录锁与进程和文件两者相关联
  + 当一个进程终止时，它所建立的所有记录锁全部被释放
  + 无论一个文件描述符何时关闭，该进程通过这个描述符所引用的文件上的任何一把锁都将被释放（这些锁都是该进程设置的）
* fork产生的子进程并不继承父进程所设置的记录锁。这是因为父进程与子进程是不同的进程，而记录锁是与进程和文件两者相关联（子进程想获得记录锁，可以在继承而来的文件描述符上调用fcntl函数来设置记录锁）
* 在执行exec后，新程序可以继承原程序的记录锁。但注意：如果对一个文件描述符设置了执行时关闭标志，则作为exec的一部分关闭该文件描述符时，将释放相应文件的所有锁

### 6）文件尾端加锁与解锁

* 当执行fcntl(F\_SETLK,flockptr)，而flockptr->l\_type=F\_WRLCK， flockptr->l\_whence=SEEK\_END，flockptr->l\_start=0，flockptr->l\_len=0 时，表示从当前文件尾端开始，包括以后所有可能追加写到该文件的任何字节都将加写锁
* 然后执行write一个字节时，该字节将被加锁
* 当执行 fcntl(F\_SETLK,flockptr)，而flockptr->l\_type=F\_UNLCK， flockptr->l\_whence=SEEK\_END，flockptr->l\_start=0，flockptr->l\_len=0 时，表示从当前文件尾端开始解锁，但是之前刚写入的一个字节仍然保持在加锁状态！！

如果你想解除的锁包括之前刚写入的一个字节，则应该flockptr->l\_len=-1 ， 负的长度值表示在指定偏移量之前的字节数

当对文件的一部分加锁时，内核将指定的偏移量变换成绝对文件偏移量。这是因为当前偏移量和文件尾端可能不断变化，但是这种变化不应该影响已经存在的锁的状态，所以内核必须独立于当前文件偏移量和文件尾端而记住锁的位置

### 7）建议性锁和强制性锁

* **建议性锁**：并不能阻止对文件有读写权限的任何其他进程不使用记录锁而访问文件
* **强制性锁**：内核会检查每一个open,read,write等操作，验证调用进程是否违背了正在访问的文件上的某一把锁

对于一个特定文件，打开其设置组ID位，同时关闭其组执行位，就开启了对该文件的强制性锁机制。

因为当组执行位关闭时，设置组ID位不再有意义（设置组ID位的目的就是为了那些需要特殊组执行权限）

如果一个进程试图read/write一个强制性锁起作用的文件，但是将要读/写的部分又由其他进程上了锁，则结果取决于三方面的因素：操作类型read/write，其他进程在该区域上持有的记录锁的类型（读锁/写锁），read/write文件描述符是阻塞的还是非阻塞的

除了read/write函数外，open函数也会受到另一个进程持有的强制性锁的影响：

* 通常，即使正在打开的文件具有强制性记录锁，open也会成功，随后的read/write遵从上表中的规则
* 但是，如果open调用中的标志指定了O\_TRUNC或者O\_CREAT，且正在打开的文件具有强制性记录锁，则不论是否指定O\_NONBLOCK，open都将出错返回，errno设置为EAGAIN

一个恶意用户可以使用强制性记录锁，对大家都可读的文件加一把读锁，这样就能阻止任何人写该文件。

强制性锁是可以设法避开的：强制性锁机制对unlink函数没有影响，因此你可以先将内容写到一个临时文件中，然后删除 原文件，最后将临时文件改名为原文件名

## 4.异步I/O

这里主要介绍POSIX异步I/O

POSIX异步I/O接口为对不同类型的文件进程异步I/O提供了一套一致的方法。现在所有的平台都要求支持这些接口

### 4.1 AIO控制块

**这些异步I/O接口使用AIO控制块来描述I/O操作，aiocb结构定义了AIO控制块**，该结构至少包括下面这些字段：

struct aiocb{  
 int aio\_fildes; // 文件描述符  
 off\_t aio\_offset; // 文件偏移量  
 volatile void \*aio\_buf; // IO缓冲区  
 size\_t aio\_nbytes; // 传输的字节数  
 int aio\_reqprio; // 优先级  
 struct sigevent aio\_sigevent; // 信号信息  
 int aio\_lio\_opcode; // IO操作列表  
 }

* aio\_fildes：表示被打开用来读或者写的文件描述符
* aio\_offset：读或者写操作从该字段指定的偏移量开始
* aio\_buf：
  + 对于读操作，它指定了从文件中读取内容存放的缓冲区
  + 对于写操作，它指定了将要向文件写入的内容存放的缓冲区
* aio\_nbytes：指定了要读或者写的字节数
* aio\_reqprio：指定了异步IO的优先级，操作系统根据该优先级来安排执行顺序。但是操作系统对于该顺序只有 有限的控制能力，因此不一定能够遵循该提示
* aio\_lio\_opcode：参见lio\_listio函数的说明。它指定了该AIO控制块是用于多操作、写操作、还是空操作
* aio\_sigevent：指定IO事件完成后，如何通知应用程序 c struct sigevent{ int sigev\_notify; // 通知类型 int sigev\_signo; // 信号编号 union sigval sigev\_value; // 通知的参数 void (\*sigev\_notify\_function)(union sigval);// 作为线程执行的函数 pthread\_attr\_t \*sigev\_notify\_attributes; // 线程属性 };
  + sigev\_notify：指定了通知的类型，可以为下列三种之一
    - SIGEV\_NONE：异步IO请求完成后，不通知进程
    - SIGEV\_SIGNAL：异步IO请求完成后，产生由sigev\_signo字段指定的信号。如果应用程序选择捕捉此信号，且在建立信号处理程序时，指定了SA\_SIGINFO标志，则该信号将被入队列（如果实现支持排队信号）。信号处理程序将被传递一个siginfo结构，该结构的si\_value字段将被设置成sigev\_value
    - SIGEV\_THREAD：当异步IO请求完成后，由sigev\_notify\_function字段指定的函数被调用成为一个子线程。sigev\_value字段被传入作为其参数。如果sigev\_notify\_attributes为NULL，则该线程为分离状态；如果sigev\_notify\_attributes不是NULL，则该线程的属性由sigev\_notify\_attributes指定

### 4.2 异步读与异步写

在进行异步I/O前需要先初始化AIO控制块。调用aio\_read函数来进行异步读操作，或调用aio\_write函数来进行异步写操作

当这些函数返回成功时，异步I/O请求就已经被操作系统放入等待处理的队列中了。这些返回值与实际I/O操作的结果没有任何关系。

注意：I/O操作在等待时，必须确保AIO控制块和数据缓冲区保持稳定。它们下面对应的内存必须始终是合法的，除非I/O操作完成，这些内存不应该被复用

如果想强制所有等待中的异步操作不等待而写入持久化的存储（如硬盘）中，可以设立一个AIO控制块并调用aio\_fsync函数：

* 参数：
  + op：指定模式：
    - O\_DSYNC：操作执行起来就像是调用了fdatasync一样
    - O\_SYNC：操作执行起来就像是调用了fsync一样
  + aiocb：指向AIO控制块

AIO控制块中的aio\_fildes字段指定了其异步写操作不等待而写入持久化的存储的那个文件

就像aio\_read/aio\_write函数一样，在安排了同步时，aio\_fsync操作立即返回。在异步的同步操作（指的是将数据修改同步到硬盘上的这个操作是异步执行的）完成之前，数据不会被持久化。AIO控制块控制我们如何被通知

### 4.3 获取异步I/O的状态

* 返回值：
  + 成功：返回 0 。表示异步操作完成，此时需要调用aio\_return函数获取操作返回值
  + 失败： 返回 -1。对aio\_error的调用失败，此时errno会告诉我们发生了什么
  + EINPROGRESS：对异步读、写、同步操作仍在等待
  + 其他情况：其他任何返回值是相关的异步操作失败返回的错误码

### 4.4 获取异步I/O返回值

* 返回值：
  + 失败： 返回 -1。对aio\_return的调用失败，此时errno会告诉我们发生了什么
  + 其他情况：返回异步读、写、同步操作的结果。即会返回read、write或者fsync在被成功调用时可能返回的结果

直到异步操作完成之前，都要小心的不要调用aio\_return函数。操作完成之前，该函数调用的结果是未定义的。

还需要小心地对每个异步操作只调用一次aio\_return函数，一旦调用了该函数，操作系统就可以释放掉包含IO操作返回值的记录

### 4.5 阻塞进程到异步I/O完成

如果在完成了所有事务时，还有异步操作未完成，可以调用调用该函数来阻塞进程，直到操作完成：

* 参数：
  + list：AIO控制块指针的数组。每个元素指向了要等待完成的异步操作
    - 如果数组元素为NULL，则跳过空指针
    - 如果数组元素非NULL，则它必须是已经初始化的异步I/O操作的AIO控制块
  + nent：该数组的长度
  + timeout：指定超时时间。如果想永不超时，则设定它为NULL

aio\_suspend可能返回三种结果：

* 如果被一个信号中断，则返回-1 ，并将errno设置为EINTR
* 如果list指定的异步I/O中，没有任何I/O操作完成的情况下，超时时间到达，则返回-1 ，并将errno设置为EAGAIN
* 如果list指定的异步I/O中，有任何I/O操作完成，则返回 0

如果在调用该函数时，所有的异步I/O操作都已完成，那么函数将在不阻塞的情况下直接返回

### 4.6 取消异步I/O

当我们不想再完成等待中的异步I/O操作时，可以尝试使用aio\_cancel函数来取消它们

* 参数：
  + fd：指定了那个未完成异步IO操作的文件描述符
  + aiocb：指向AIO控制块
* 返回值：
  + 失败： 返回 -1。对aio\_cacel的调用失败，此时errno会告诉我们发生了什么
  + AIO\_ALLDONE：所有异步操作在尝试取消它们之前已经完成
  + AIO\_CANCELED：所有要求的异步操作已经取消
  + AIO\_NOTCANCELED：至少有一个异步操作没有被取消

如果aiocb为NULL，则系统将尝试取消所有在fd文件上未完成的异步I/O操作。其他情况下尝试取消由AIO控制块描述的单个异步I/O操作。之所以说尝试，是因为无法保证系统能够取消正在进程中的任何异步操作

如果异步I/O操作被成功取消，则对应的AIO控制块调用aio\_error函数将会返回错误ECANCELED

如果异步I/O操作不能被取消，那么相应的AIO控制块不会被修改

### 4.7 批量提交异步I/O请求

AIO控制块列表描述了一系列I/O请求，可以由该函数提交

* 参数：
  + mode：决定了IO是否同步的。其值可以为：
    - LIO\_WAIT：lio\_listio函数将在所有的由列表指定的IO操作完成后返回。此时sigev参数被忽略。这是同步操作
    - LIO\_NOWAIT：lio\_listio函数将在所有的由列表指定的IO操作请求入队立即返回（不等到完成）。进程将在所有IO操作完成后，根据sigev参数指定的方式，被异步的通知（如果不想被通知，则将sigev设置为NULL） > 每个AIO控制块本身也可能启动了在各自操作完成时的异步通知。sigev指定的异步通知是额外加的，并且只会在所有的IO操作完成后发送
  + list：AIO控制块指针的数组，该数组指定了要允许的IO操作。如果数组包含空指针，则跳过这些空指针
  + nent：数组的长度
  + sigev：指定了当所有IO操作完成后，发送的异步通知（如果不想被通知，则将sigev设置为NULL）。只有当mode=LIO\_NOWAIT才有意义

### 4.8 异步I/O的数量限制

异步IO的数量有限制，这些限制都是运行时不变量

* 可以通过sysconf函数并把name设置为\_SC\_IO\_LISTIO\_MAX来设定AIO\_LISTIO\_MAX的值
* 可以通过sysconf函数并把name设置为\_SC\_AIO\_MAX来设定AIO\_MAX的值
* 可以通过sysconf函数并把name设置为\_SC\_AIO\_PRIO\_DELTA\_MAX来设定AIO\_PRIO\_DELTA\_MAX的值

这些常量的意义为：

* AIO\_LISTIO\_MAX：单个列表IO调用中的最大IO操作数量
* AIO\_MAX：未完成的异步IO操作的最大数量
* \_SC\_AIO\_PRIO\_DELTA\_MAX：进程可以减少的其异步IO优先级的最大值

## 6.存储映射I/O

**存储映射I/O能将一个磁盘文件映射到存储空间中的一个缓冲区上**，于是：

* **当从缓冲区中取数据时，就相当于读文件中的相应字节**
* **将数据存入缓冲区时，相应字节就自动写入文件**

因此，就可以在不使用read和write的情况下执行I/O

**效率**：与**mmap**和**memcpy**相比，**read**和**write**执行了更多的系统调用，并做了更多的复制。**read**和**write**将数据从内核缓冲区中复制到应用缓冲区(**read**)，然后再把数据从应用缓冲区复制到内核缓冲区(**write**)。而**mmap**和**memcpy**则直接把数据从映射到地址空间的一个内核缓冲区复制到另一个内核缓冲区。当引用尚不存在的内存页时，这样的复制过程就会作为处理页错误的结果而出现（每次错页读发生一次错误，每次错页写发生一次错误）。如果系统调用和额外的复制操作的开销和页错误的开销不同，那么两者方法中就会有一种比另一种表现更好

**mmap的优劣**：

* 劣：
  + mmap不能用在某些设备（如网络设备或终端设备）之间进行复制
  + 在对被复制的文件进行映射后，也要注意该文件的长度是否改变
* 优
  + 某些应用程序能得益于存储映射I/O，因为它处理的是存储空间而不是读、写一个文件，所以常常可以简化算法

### 6.1 mmap建立映射

内核将一个给定的文件映射到一个存储区域中是由mmap实现的：

* 参数：
  + addr：用于指定映射存储区的起始地址。如果为 0，则表示由系统选择该映射区的起始地址
  + len：指定要映射的字节数
  + prot：指定了映射存储区的保护要求，可以为下列之一（也可以为PROT\_READ、PROT\_WRITE、PROT\_EXEC的按位或）：
    - PROT\_READ：映射区可读
    - PROT\_WRITE：映射区可写
    - PROT\_EXEC：映射区可执行
    - PROT\_NONE：映射区不可访问
  + flag：影响映射存储区的多种属性：
    - MAP\_FIXED：返回值必须等于addr。因为这不利于可移植性，所以不鼓励使用此标志（如果未指定此标志，且addr非 0，则内核只是将addr视为在何处设置映射区的一个建议，但是不保证会使用所要求的地址。将addr设为 0，可以获取最大可移植性）
    - MAP\_SHARED：此标志指定存储操作将修改底层的映射文件（即存储操作相当于对底层的映射文件进行write）
    - MAP\_PRIVATE：此标志指定存储操作将不会修改底层的映射文件，而是创建该底层的映射文件的一个私有副本。所有后续的对该映射区的引用都是引用该副本（此标志的一个用途是用于调试程序，它将程序文件的正文部分映射到存储区，但允许用户修改其中的指令。任何修改只影响程序文件的副本，而不影响源文件）
  + fd：指定要被映射文件的描述符。在文件映射到地址空间之前，必须先打开该文件
  + off：要映射字节在文件中的起始偏移量

注意：

* prot不能超过文件open模式访问权限。如：如果该文件是只读打开的，则对于映射存储区就不能指定PROT\_WRITE
* flag的标志中，MAP\_SHARED和MAP\_PRIVATE必须指定一个，而且二者不能同时指定
* flag的标志中，每种实现可能会自定义一些MAP\_XXX标志值，它们是那些实现特有的
* off的值和addr的值（如果指定了MAP\_FIXED）通常被要求是系统虚拟存储页长（虚拟存储页长可以用带参数\_SC\_PAGESIZE或者\_SC\_PAGE\_SIZE的sysconf函数得到 ）的倍数。因为off和addr通常指定为 0 ，所以这种要求一般并不重要

下图是存储映射文件的基本情况。其中“起始地址”是mmap的返回值

假设文件长度为 12 字节，系统页长为 512 字节，则系统通常会提供 512 字节的映射区，其中后 500字节被设置为0 。可以修改后面的这 500 字节，但是任何变动都不会在文件中反映出来。因此，不能用mmap将数据添加到文件中（可以先加长该文件，使得能够将后面500字节的改动反映到文件中去）

与映射区相关的信号有：SIGSEGV和SIGBUS

* SIGSEGV：通常用于指示进程试图访问对它不可用的存储区。如果映射存储区被mmap指定成了只读的，则进程试图写这个映射存储区时，也产生此信号
* SIGBUS：如果映射区的某个部分在访问时不存在，则产生SIGBUS信号 > 如：用文件长度映射了一个文件，但在引用该映射区之前，另一个进程已将该文件截断，此时如果进程试图访问被截断部分对应的映射区，则接收到SIGBUS信号

**子进程能够通过fork继承存储映射区（因为子进程复制了父进程地址空间，而存储映射区是该地址空间的组成部分）。但是一旦子进程调用了exec，则新程序不再拥有存储映射区了**

### 6.2 mprotect修改映射区权限

* 参数：
  + addr：存储映射区的地址，必须是系统页长的整数倍
  + len：存储映射区的长度
  + prot：存储映射区的修改后的权限，与mmap中的prot一样

### 6.3 msync冲洗映射区

如果修改的页是通过MAP\_SHARED标志映射到地址空间的，那么对内存映射区的修改并不会立即写回到底层文件中。何时写回脏页（即被修改的页）由内核的守护进程决定，决定的依据有两个：

* 系统负载
* 用来限制在系统失败事件中的数据损失的配置参数

因此，如果只修改了一页中的一个字节，当修改被写回到文件中时，整个页都被写回

如果共享映射中的页已修改，则可以调用msync函数将该页冲洗到底层文件中。它类似于fsync，但作用于存储映射区：

* 参数：
  + addr：存储映射区的地址
  + len：存储映射区的长度
  + flags：用于控制如何冲洗存储区。下列两个常量二选一
    - MS\_ASYNC：简单的调试要写的页
    - MS\_SYNC：在返回之前等待写操作完成

如果存储映射是私有的，则不修改底层的文件

### 6.4 munmap解除映射

当进程终止时，会自动解除存储映射区的映射。也可以直接调用munmap函数来手动解除映射区

* 参数：
  + addr：存储映射区的地址
  + len：存储映射区的长度

munmap并不影响被映射的对象。即调用munmap并不会使映射区的内容写到磁盘文件上

* 对于MAP\_SHARED存储映射区的磁盘文件的更新，会在我们将数据写到存储映射区后的某个时刻，由内核虚拟存储算法自动运行
* 对于MAP\_PRIVATE存储映射区：对其做出的修改会被丢弃

注意，对于创建存储映射区时使用的文件描述符，如果关闭该描述符，并不会解除映射区

### 使用mmap拷贝文件

#include "apue.h"  
#include <fcntl.h>  
#include <sys/mman.h>  
  
#define COPYINCR (1024\*1024\*1024) /\* 1 GB \*/  
  
int  
main(int argc, char \*argv[])  
{  
 int fdin, fdout;  
 void \*src, \*dst;  
 size\_t copysz;  
 struct stat sbuf;  
 off\_t fsz = 0;  
  
 if (argc != 3)  
 err\_quit("usage: %s <fromfile> <tofile>", argv[0]);  
  
 if ((fdin = open(argv[1], O\_RDONLY)) < 0)  
 err\_sys("can't open %s for reading", argv[1]);  
  
 if ((fdout = open(argv[2], O\_RDWR | O\_CREAT | O\_TRUNC,  
 FILE\_MODE)) < 0)  
 err\_sys("can't creat %s for writing", argv[2]);  
  
 if (fstat(fdin, &sbuf) < 0) /\* need size of input file \*/  
 err\_sys("fstat error");  
  
 if (ftruncate(fdout, sbuf.st\_size) < 0) /\* set output file size \*/  
 err\_sys("ftruncate error");  
  
 while (fsz < sbuf.st\_size) {  
 if ((sbuf.st\_size - fsz) > COPYINCR)  
 copysz = COPYINCR;  
 else  
 copysz = sbuf.st\_size - fsz;  
  
 if ((src = mmap(0, copysz, PROT\_READ, MAP\_SHARED,  
 fdin, fsz)) == MAP\_FAILED)  
 err\_sys("mmap error for input");  
 if ((dst = mmap(0, copysz, PROT\_READ | PROT\_WRITE,  
 MAP\_SHARED, fdout, fsz)) == MAP\_FAILED)  
 err\_sys("mmap error for output");  
  
 memcpy(dst, src, copysz); /\* does the file copy \*/  
 munmap(src, copysz);  
 munmap(dst, copysz);  
 fsz += copysz;  
 }  
 exit(0);  
}

# 九.进程间通信

## 1.管道

**管道是UNIX系统IPC的最古老形式，所有UNIX系统都提供此种通信机制**

管道有2中局限性：

1. 历史上，是**半双工**的。虽然现在某些系统提供全双工管道，但是为了最佳移植性，不应预先假定系统支持全双工管道
2. 管道**只能在具有公共祖先的两个进程间使用**（通常，一个管道由一个进程创建，在进程调用fork之后，这个管道就能在父进程和子进程之间使用了）

尽管有上述局限，半双工管道仍是最常用的IPC形式。每当在管道中键入一个命令序列，让shell执行时，shell都会为每一条命令单独创建一个进程，然后用管道将前一条命令进程的标准输出与后一条命令的标准输入相连接

### 1.1 创建管道

* fd：函数调用返回后记录了管道的2个文件描述符
  + fd[0]：为读而打开
  + fd[1]：为写而打开

POSIX.1允许实现支持全双工管道。对于这些实现，fd[0]和fd[1]以读/写方式打开

一般创建管道后，会fork子进程，然后根据数据流向选择性的关闭父子进程中管道的读端或写端，从而建立父子进程之间通信的IPC：

### 1.2 管道的读写规则

* 当读一个写端已被关闭的管道时，在所有数据都被读取后，read返回0，表示文件结束。（从技术上讲，如果管道的写端还有进程，就不会产生文件的结束。可以复制一个管道的描述符，使得有多个进程对它具有写打开文件描述符。但是，通常一个管道只有一个读进程和一个写进程）
* 如果写一个读端已被关闭的管道，则产生信号SIGPIPE。如果忽略该信号或者捕捉该信号并从其处理程序返回，则write返回-1，errno设置为EPIPE
* 写管道时，常量PIPE\_BUF规定了内核的管道缓冲区大小。如果对管道调用write。而且要求写的字节数小于等于PIPE\_BUG，则此操作不会与其他进程对同一管道的write操作交叉进行。但是，若有多个进程同时写一个管道，而且我们要求写的字节数超过PIPE\_BUF，那么所写的数据可能会与其他进程所写的数据相互交叉。用pathconf或fpathconf函数可以确定PIPE\_BUF

### 1.3 标准I/O库管道函数

* popen：先执行fork，然后调用exec执行cmdstring，并且返回一个标准I/O文件指针
  + type
    - r：返回的文件指针是可读的
    - w：返回的文件指针是可写的
* pclose：关闭标准I/O流，等待命令终止，然后返回shell的终止状态

## 2.协同进程

UNIX系统过滤程序从标准输入读取数据，向标准输出写数据。几个过滤程序通常在shell管道中线性连接。**当一个过滤程序既产生某个过滤程序的输入，又读取该过滤程序的输出时，它就变成了协同进程**

popen只提供连接到另一个进程的标准输入或标准输出的一个单向管道，而协同进程则有连接到另一个进程的两个单向管道：一个连接到其标准输入，另一个则来自其标准输出。我们想将数据写到其标准输入，经其处理后，再从标准输出读取数据

协同进程的标准输入和标准输出连接到管道

## 3.FIFO

FIFO有时候被称为**命名管道**，和管道不同的是，它可以在不相关的进程间使用，不必具有继承关系

FIFO是一种文件类型，通过stat结构的st\_mode成员的编码可以知道文件是否是FIFO类型。可以用S\_ISFIFO宏对此进行测试

FIFO有以下2个**用途**：

1. shell命令使用FIFO将数据从一条管道传送到另一条时，无需创建中间临时文件
2. 客户进程-服务器进程应用程序中，FIFO用作汇聚点（多个客户进程向服务器同一个众所周知的FIFO写），在客户进程和服务器进程二者之间传递数据

### 3.1 创建FIFO

创建FIFO类似于创建文件。FIFO的路径名存在于文件系统中

* mkfifo
  + mode：与[open函数](#X4f892631062a259c02243e0c580974547dd1932)中的mode相同
* mkfifoat：和mkfifo函数相似，但是该函数可以被用来在fd文件描述符表示的目录相关的位置创建一个FIFO
  + path
    - 如果指定的是绝对路径名，则fd参数会被忽略掉，并且mkfifoat函数的行为和mkfifo类似
    - 如果指定的是相对路径名，则fd参数是一个打开目录的有效文件描述符，路径名和路径有关
    - 如果指定的是相对路径名，并且fd参数有一个特殊值AT\_FDCWD，则路径名以当前目录开始，mkfifoat和mkfifo类型

应用程序可以用mknod和mknodat函数创建FIFO。因为POSIX.1原先并没有包括mknod函数，所以mkfifo是专门为POSIX.1设计的。mknod和mknodat函数现在已包括在POSIX.1的XSI扩展中

### 3.2 打开FIFO

使用这两个函数创建FIFO时，要用open来打开它：

* 一般情况下，只读open要阻塞到某个其它进程为写而打开这个FIFO为止。类似地，只写open要阻塞到某个其它进程为读而打开它为止
* 如果指定了O\_NONBLOCK，则只读open立即返回。但是，如果没有进程为读而打开一个FIFO，那么只写open将返回-1，并将errno设置成ENXIO

### 3.3 读写FIFO

* 若write一个尚无进程为读而打开的FIFO，则产生信号SIGPIPE
* 若某个FIFO的最后一个写进程关闭了该FIFO，则将为该FIFO的读进程产生一个文件结束标志

一个给定的FIFO有多个写进程是常见的。这意味着，如果不希望多个进程所写的数据交叉，则必须考虑原子写操作。和管道一样，常量PIPE\_BUF说明了可被原子地写到FIFO的最大数据量

## 4.XSI IPC

有3种称作XSI IPC的IPC：**消息队列**、**信号量**、**共享存储**。它们之间有很多相似之处

### 4.1 标识符和键

#### 1）标识符

* 每个内核中的IPC结构（消息队列、信号量或共享内存）都用一个非负整数的**标识符**加以引用
* 与文件描述符不同，**IPC标识符**不是小的整数。当一个IPC结构被创建，然后又被删除时，与这种结构相关的标识符连续加1，直至到达一个整形数的最大正值，然后又回转到0

#### 2）键

**标识符是IPC对象的内部名**。为使多个合作进程能够在同一IPC对象上汇聚，需要提供一个外部命名方案。为此，**每个IPC对象都与一个键相关联，将这个键作为该对象的外部名**（创建IPC结构时，应指定一个键）。**键的类型是基本系统数据类型key\_t**，通常在<sys/types.h>中被定义为长整形。这个键由内核变换成标识符

### 4.2 权限结构

每个IPC结构关联了一个ipc\_perm结构（<sys/ipc.h>），规定了权限和所有者，至少包括以下成员：

struct ipc\_perm{  
 uid\_t uid; /\* 拥有者的有效用户ID \*/  
 gid\_t gid; /\* 拥有者的有效组ID \*/  
 uid\_t cuid; /\* 创建者的有效用户ID \*/  
 gid\_t cgid; /\* 创建者的有效组ID \*/  
 mode\_t mode; /\* 访问模式 \*/  
 ...  
};

* 创建IPC结构时，对所有字段都赋初值
* IPC结构的创建者或超级用户可以调用msgctl、semctl或shmctl修改uid、gid和mode字段。修改这些字段类似于对文件调用chown和chmod

**对于任何IPC结构都不存在执行权限**，消息队列和共享内存使用术语”读“和”写“，信号量则用”读“和”更改“。下图显示了每种IPC的6种权限：

### 4.3 结构限制

所有3种形式的XSI IPC都有内置限制。大多数限制可以通过**重新配置内核**来改变

### 4.4 优点和缺点

* 优点
  + 可靠
  + 流控制
  + 面向记录
  + 可以用非先进先出的顺序处理
* 缺点
  + **XSI IPC结构是在系统范围内起作用的，没有引用计数**
    - 即，不会因为最后一个使用IPC结构的进程终止而删除，它们会一直留在系统中直至一些删除事件发生（作为对比，最后一个引用管道的进程终止时，管道就被完全删除了；对FIFO而言，最后一个引用的进程终止时，虽然FIFO的名字仍保留在系统中，直至被显示删除，但是留在FIFO中的数据已被删除了）
  + **XSI IPC结构在文件系统中没有名字**
    - 因此，不能使用一些已有的系统调用访问或修改其属性，为了加以支持，内核中增加了十几个全新的系统调用。不能使用ls查看IPC对象，不能用rm删除
    - 因为这些形式的IPC不使用文件描述符，所以不支持I/O复用函数

”无连接“指无需先调用某种形式的打开函数就能发送消息的能力

## 5.消息队列

后文把消息队列简称为“队列”，把标识符简称为“队列ID”

消息队列是消息的链接表，存储在内核中，由消息队列标识符标识

### 5.1 与消息队列相关的结构

每个队列都有一个msqid\_ds结构与其关联，这个结构定义了队列的当前状态：

struct msqid\_ds{  
 struct ipc\_perm msg\_perm;  
 msgqnum\_t msg\_qnum; /\* 队列中的消息数 \*/  
 msglen\_t msg\_qbytes; /\* 队列中消息的字节 \*/  
 pid\_t msg\_lspid; /\* 最后调用msgsnd()的进程ID \*/  
 pid\_t msg\_lrpid; /\* 最后调用msgrcv()的进程ID \*/  
 time\_t msg\_stime; /\* 最后调用msgsnd()的时间 \*/   
 time\_t msg\_rtime; /\* 最后调用msgrcv()的时间 \*/  
 time\_t msg\_ctime; /\* 最后一次修改队列的时间 \*/  
 ...  
};

下图为消息队列的系统限制：

“导出的”表示这种限制来源于其它限制

### 5.2 创建或打开消息队列

* msgget：创建一个新队列或打开一个现有队列
  + key\_t：创建IPC结构时需要指定一个键，作为IPC对象的外部名。键由内核转变成标识符
  + 返回值：若成功，返回非负队列ID（标识符），该值可被用于其余几个消息队列函数

创建队列时，需要初始化msqid\_ds结构的下列成员：

* ipc\_perm：按[XSI IPC中的描述](#Xa60cf53155e28124774d8650bdd866967f43aab)初始化
* msg\_qnum、msg\_lspid、msg\_lrpid、msg\_stime和msg\_rtime都设为0
* msg\_ctime设置为当前时间
* msg\_qbytes设置为系统限制值

### 5.3 操作消息队列

* msqid：队列ID（标识符），msgget的返回值
* cmd：
  + IPC\_STAT：取此队列的msgid\_qs结构，并存放在buf指向的结构中
  + IPC\_SET：将字段msg\_perm.uid、msg\_perm.gid、msg\_perm.mode和msg\_qbytes从Buf指向的结构赋值到这个队列的msqid\_ds结构中（此命令只能由下列2种进程执行：1）其有效ID等于msg\_perm.cuid或msg\_perm.uid；2）具有超级用户特权的进程；只有超级用户才能增加msg\_qbytes的值）
  + IPC\_RMID：从系统中删除消息队列以及仍在队列中的所有数据。这种删除立即生效。仍在使用这一消息队列的其它进程在他们下一次试图对此队列进行操作时，将得到EIDRM错误（此命令只能由下列2种进程执行：1）其有效ID等于msg\_perm.cuid或msg\_perm.uid；2）具有超级用户特权的进程）

上面3条命令也可用于**信号量**和**共享存储**

### 5.4 添加消息

msgsnd将新消息添加到队列尾端

每个消息由3部分组成：一个正的长整型类型的字段、一个非负的长度、实际数据字节（对应于长度）

* ptr：指向一个长整型数，它包含了正的整型消息类型，其后紧接着消息数据（若nbytes为0则无消息数据） c struct mymesg{ long mtype; /\* 正的长整型类型字段 \*/ char mtext[512]; /\* \*/ }; 因此，ptr可以是一个指向mymesg结构的指针
* nbytes：消息数据的长度
* flag
  + ICP\_NOWAIT：类似于文件I/O的非阻塞I/O标准
    - 若消息队列已满（数量或字节总数达到限制）
      * 若指定ICP\_NOWAIT，函数立即出错返回EAGAIN
      * 若没指定ICP\_NOWAIT，进程会阻塞到：1）有空间可用；2）从系统中删除了此队列(会返回EIDRM错误)；3）捕捉到一个信号，并从信号处理程序返回(会返回EINTR错误)

当msgsnd返回成功时，消息队列相关的msqid\_ds结构会随之更新

### 5.5 获取消息

* msgrcv：从队列中取消息（并不一定要以先进先出的顺序取消息，也可以按类型字段取消息）
  + ptr：与msgsnd中一样
  + nbytes：指定数据缓冲区的长度
    - 若返回的长度大于nbyte
      * 在flag中设置了MSG\_NOERROR，则消息被截断，但是不会有通知
      * 如果没有设置MSG\_NOERROR，则出错返回E2BIG（消息仍留在队列中）
  + type：欲获取的消息类型
    - 0：返回队列中的第一个消息
    - >0：返回队列中消息类型为type的第一个消息
    - <0：返回队列中消息类型小于等于type绝对值的消息，如果有若干个，则取类型值最小的消息
  + flag
    - IPC\_NOWAIT：可使操作不阻塞
      * 当队列中无消息时
        + 若指定了该标志，函数会返回-1，error设置为ENOMSG
        + 若没有指定该标志，函数会一直阻塞直到：1）有了指定类型的消息可用；2）从系统中删除了此队列（会导致函数返回-1，error设置为EIDRM）；3）捕捉到一个信号并从信号处理程序返回（会导致函数返回-1，error设置为EINTR）

msgrcv成功执行时，内核会更新与该消息队列相关的msgid\_ds结构

## 6.信号量

信号量是一个**计数器**，用于为多个进程提供对**共享数据对象**的访问

为了正确实现信号量，信号量值的测试及减1操作应当是原子操作。为此，信号量通常是在内核中实现的

常用的信号量形式被称为二元信号量，它控制单个资源，其初始值为1。但是一般而言，信号量的初值可以是任意一个正值，该值表明有多少个共享资源单位可供共享应用

下面的特性使得XSI信号量更复杂：

* 信号量并非是单个非负值，而必须定义为含有一个或多个信号量值的集合。当创建信号量时，要指定集合中信号量值的数量
* 信号量的创建是独立于它的初始化的。这是一个致命缺点。因此不能原子地创建一个信号量集合，并且对该集合中的各个信号量赋初值
* 即使没有进程正在使用各种形式的XSI IPC，他们仍然是存在的。有的程序在终止时并没有释放已经分配给它的信号量，我们不得不为这种程序担心

### 6.1 信号量的相关结构

内核为每个**信号量集合**维护着一个semid\_ds结构

struct semid\_ds{  
 struct ipc\_perm sem\_perm;   
 unsigned short sem\_nsems; /\* 集合中的信号量数目 \*/  
 time\_t sem\_otime; /\* 最后一次调用semop()的时间 \*/  
 time\_t sem\_ctime; /\* 最后一次改变的时间 \*/  
 ...  
};

**每个信号量**由一个无名结构表示，它至少包含下列成员：

struct{  
 unsigned short semval; /\* 信号量的值，总是>=0 \*/  
 pid\_t sempid; /\* 最后一个操作信号量的进程ID \*/  
 unsigned short semncnt; /\* 等待 semval>curval 的进程数 \*/  
 unsigned short semzcnt; /\* 等待 semval==0 的进程数 \*/  
 ...  
};

下图是影响信号量集合的系统限制；

### 6.2 获得信号量

* key：创建IPC结构时需要指定一个键，作为IPC对象的外部名。键由内核转变成标识符
* nsems：该信号量集合中的信号量数
  + 如果是创建新集合（一般在服务器进程中），则必须指定nsems
  + 如果是引用现有集合（一个客户进程），则将nsems指定为0
* flag：

创建队列时，需要初始化semid\_ds结构的下列成员：

* ipc\_perm结构按[XSI IPC中的描述](#Xa60cf53155e28124774d8650bdd866967f43aab)初始化。结构中的mode成员被设置为flag中的相应权限位
* sem\_otime设置为0
* sem\_ctime设置为当前时间
* sem\_nsems设置为nsems

### 6.3 操作信号量

* 参数
  + semid：信号量集合
  + semnum：信号量集合中的某一信号量
  + cmd：命令
    - IPC\_STAT：获取信号量集合的semid\_ds结构，存储在arg.buf指向的结构中
    - IPC\_SET：按arg.buf指向的结构中的值设置集合semid\_ds结构中的sem\_perm.uid、sem\_perm.gid和sem\_perm.mode字段（此命令只能由下列2种进程执行：1）其有效ID等于sem\_perm.cuid或sem\_perm.uid；2）具有超级用户特权的进程；）
    - IPC\_RMID：从系统中删除该信号量集合。这种删除是立即发生的。删除时仍在使用这一信号量集合的其它进程在他们下一次试图对此信号量集合进行操作时，将得到EIDRM错误（此命令只能由下列2种进程执行：1）其有效ID等于sem\_perm.cuid或sem\_perm.uid；2）具有超级用户特权的进程；）
    - GETVAL：返回semnum指定信号量的值
    - SETVAL：设置semnum指定信号量的值
    - GETPID：返回semnum指定信号量的sempid（最后一个操作信号量的进程ID）
    - GETNCNT：返回semnum指定信号量的semncnt
    - GETZCNT：返回semnum指定信号量的semzcnt
    - GETALL：取该集合中所有的信号量值。这些值存储在arg.array指向的数组中
    - SETALL：将该集合中所有的信号量值设置成arg.array指向的数组中的值
  + semun：可选参数，是否使用取决于命令cmd，如果使用则类型是联合结构semun c union semun{ int val; /\* for SETVAL \*/ struct semid\_ds \*buf; /\* for ICP\_STAT and IPC\_SET \*/ unsigned short \*array; /\* for GETALL and SETALL \*/ };
* 返回值：对于除GETALL以外的所有GET命令，函数都返回相应值。对于其他命令，若成功则返回值为0，若出错，则设置errno并返回-1

函数semop自动执行信号量集合上的操作数组：

* 指定信号量集合
* semoparray：一个指针，指向一个由sembuf结构表示的信号量操作数组 c struct sembuf{ unsigned short sem\_num; /\* 信号量集合中的某个信号量 \*/ short sem\_op; /\* 操作 \*/ short sem\_flg; /\* IPC\_NOWAIT，SEM\_UNDO \*/ };
  + sem\_op为正值：**这对应于进程释放的占用的资源数。sem\_op值会加到该信号量的值上**
  + sem\_op为负值：**则表示要获取由该信号量控制的资源**
    - 如果**信号量的值**大于等于sem\_op的绝对值，则从信号值中减去sem\_op的绝对值
    - 如果**信号量的值**小于sem\_op的绝对值
      * 若指定了IPC\_NOWAIT，则出错返回EAGAIN
      * 若未指定IPC\_NOWAIT，则该信号量的semncnt增加1，然后调用进程被挂起直到下列事件之一发生
        + 该信号量的值变成大于等于sem\_op的绝对值。此信号量的semncnt值减1，并且从信号量值中减去sem\_op的绝对值
        + 从系统中删除了此信号量。在这种情况下，函数出错返回EIDRM
        + 进程捕捉到一个信号，并从信号处理程序返回，在这种情况下，此信号量的semncnt值减1，并且函数出错返回EINTR
  + sem\_op为0：**则表示调用进程希望等待到信号量的值变为0**
    - 如果**信号量的值**是0，则表示函数立即返回
    - 如果**信号量的值**非0，则：
      * 若指定了IPC\_NOWAIT，则semop出错返回EAGAIN
      * 若未指定IPC\_NOWAIT，则该信号量的semncnt增加1，然后调用进程被挂起直到下列事件之一发生
        + 该信号量值变为0，此信号量的semncnt值减1
        + 从系统中删除了此信号量。在这种情况下，函数出错返回EIDRM
        + 进程捕捉到一个信号，并从信号处理程序返回，在这种情况下，此信号量的semncnt值减1，并且函数出错返回EINTR
* nops：数组的数量，即操作的数量

该函数具有原子性，它或者执行了数组中的所有操作，或者一个也不做

#### exit时的信号量调整

如果在进程终止时，它占用了经由信号量分配的资源，那么就会成为一个问题。无论何时只要为信号量操作指定了SEM\_UNDO标志，然后分配资源(sem\_op值小于0)，那么内核就会记住对于该特定信号量，分配给调用进程多少资源（sem\_op的绝对值）。当该进程终止时，无论自愿或不自愿，内核都将检验该进程是否还有尚未处理的信号量调整值。如果有，则按调整值对相应信号量值进行处理

如果用带SETVAL或SETALL命令的semctl设置一个信号量的值，则在所有进程中，该信号量的调整值都将设置为0

## 7.共享存储

**共享存储允许2个或多个进程共享一个给定的存储区**

**因为数据不需要在客户进程和服务器进程之间复制，所以这是最快的一种IPC**

使用共享存储要注意的是：进程在往共享存储写完成之前，读进程不应该去取数据。通常，信号量用于同步共享存储访问

mmap就是共享存储的一种形式，但是XSI共享存储与其区别在于，XSI共享存储没有相关文件。XSI共享存储段是内存的匿名段

### 7.1 共享存储的内核结构

内核为每个共享存储段维护着一个结构，至少包含以下成员：

struct shmid\_ds{  
 struct ipc\_perm shm\_perm;   
 size\_t shm\_segsz; /\* 共享存储段的字节大小 \*/  
 pid\_t shm\_lpid; /\* 最后调用shmop()的进程ID \*/  
 pid\_t shm\_cpid; /\* 创建该共享存储段的进程ID \*/  
 shmatt\_t shm\_nattch; /\* 当前访问计数 \*/  
 time\_t shm\_atime; /\* 最后一次attach的时间 \*/  
 time\_t shm\_dtime; /\* 最后一次detach的时间 \*/  
 time\_t shm\_ctime; /\* 最后一次change的时间 \*/  
 ...  
};

下图为影响共享存储的系统限制：

### 7.2 创建或获得共享存储

* key：创建IPC结构时需要指定一个键，作为IPC对象的外部名。键由内核转变成标识符
* size：共享存储段的长度，单位是字节。实现通常将其向上取为系统页长的整倍数。但是，如果指定的值不是系统页长的整倍数，那么最后一页的余下部分是不可使用的
  + 如果正在创建一个新段，则必须指定size（段内的内容初始化为0）
  + 如果正在引用一个现存的段，则将size指定为0

创建一个新共享存储段时，初始化shmid\_ds结构的下列成员：

* ipc\_perm结构按[XSI IPC中的描述](#Xa60cf53155e28124774d8650bdd866967f43aab)初始化。结构中的mode成员被设置为flag中的相应权限位
* shm\_lpid、shm\_nattach、shm\_atime和shm\_dtime都设置为0
* shm\_ctime设置为当前时间
* sem\_segsz设置为size

### 7.3 操作共享存储

* shmid：共享存储标识符，由函数shmget得到
* cmd
  + IPC\_STAT：获取段对应的shmid\_ds结构，并将其存储在由buf指向的结构中
  + IPC\_SET：按buf指向的结构中的值设置此共享存储段相关的shmid\_ds结构中的下列3个字段：shm\_perm.uid、shm\_perm.gid和shm\_perm.mode字段（此命令只能由下列2种进程执行：1）其有效ID等于shm\_perm.cuid或shm\_perm.uid；2）具有超级用户特权的进程；）
  + IPC\_RMID：从系统中删除该共享存储段。因为每个共享存储段维护着一个连接计数(shmid\_ds中的shm\_nattach字段)，所以除非使用该段的最后一个进程终止或与该段分离，否则不会实际上删除该存储段。不管此段是否仍在使用，该段标识符都会被立即删除，所以不能再用shmat与该段连接（此命令只能由下列2种进程执行：1）其有效ID等于shm\_perm.cuid或shm\_perm.uid；2）具有超级用户特权的进程；） Linux和Solaris提供了另外两种命令，但它们并非Single UNIX Specification的组成部分
  + SHM\_LOCK：在内存中对共享存储段加锁（该命令只能由超级用户执行）
  + SHM\_UNLOCK：解锁共享存储段（该命令只能由超级用户执行）

### 7.4 与共享存储段连接

可以调用shmat将共享存储段连接到进程的地址空间中

* shmid：共享存储段的标识符
* addr：共享存储段连接到进程的该地址
  + 0：由内核选择（推荐的方式）
  + 非0
    - flag指定了SHM\_RND，则连接到addr所指的地址上
    - flag没指定SHM\_RND，则此段连接到 addr-(addr mod SHMLAB) 所表示的地址上（SHM\_RND意思是”取整“，SHMLBA的意思是”低边界地址倍数“）
* flag
  + SHM\_RDONLY：以只读方式连接此段
  + 否则：以读写方式连接此段

如果函数成功，内核会将与该共享存储段相关的shmid\_ds结构中的shm\_nattch计数器值加1

### 7.5 与共享存储段分离

下列函数可以与共享存储段分离。该调用并不从系统中删除其标识符以及其相关的数据结构。该标识符仍然存在，直到某个进程调用shmctl并使用IPC\_RMID命令特地删除它为止

* addr：进程与共享存储段连接的地址

如果函数成功，共享存储段相关的shmid\_ds结构中的shm\_nattch计数器值减1

### 进程连接共享存储段的位置

内核将以地址0连接共享存储段放在什么位置上与系统密切相关，下列程序可以进行测试：

#include "apue.h"  
#include <sys/shm.h>  
  
#define ARRAY\_SIZE 40000  
#define MALLOC\_SIZE 100000  
#define SHM\_SIZE 100000  
#define SHM\_MODE 0600 /\* user read/write \*/  
  
char array[ARRAY\_SIZE]; /\* uninitialized data = bss \*/  
  
int  
main(void)  
{  
 int shmid;  
 char \*ptr, \*shmptr;  
  
 printf("array[] from %p to %p\n", (void \*)&array[0],  
 (void \*)&array[ARRAY\_SIZE]);  
 printf("stack around %p\n", (void \*)&shmid);  
  
 if ((ptr = malloc(MALLOC\_SIZE)) == NULL)  
 err\_sys("malloc error");  
 printf("malloced from %p to %p\n", (void \*)ptr,  
 (void \*)ptr+MALLOC\_SIZE);  
  
 if ((shmid = shmget(IPC\_PRIVATE, SHM\_SIZE, SHM\_MODE)) < 0)  
 err\_sys("shmget error");  
 if ((shmptr = shmat(shmid, 0, 0)) == (void \*)-1)  
 err\_sys("shmat error");  
 printf("shared memory attached from %p to %p\n", (void \*)shmptr,  
 (void \*)shmptr+SHM\_SIZE);  
  
 if (shmctl(shmid, IPC\_RMID, 0) < 0)  
 err\_sys("shmctl error");  
  
 exit(0);  
}

## 8.POSIX信号量

POSIX信号量接口意在解决[XSI信号量](#X355caed436874f2aba952356f8171bcc11c2ad2)接口的几个缺陷：

* 相比于XSI接口，POSIX信号量接口考虑到了**更高性能**的实现
* POSIX信号量接口**使用更简单**：没有信号量集，在熟悉的文件系统操作后一些接口被模式化了
* POSIX信号量在**删除时表现更完美**
* POSIX信号量有2种形式：命名的和未命名的。差异在创建和销毁的形式。
* **未命名信号量**只存在于内存中，并要求能使用信号量的进程必须可以访问内存。意味着它们只能用于同一进程的线程中，或者不同进程中已经映射相同内存内容到他们的地址空间中的线程
* **命名信号量**可以通过名字访问，因此可被任何已知它们名字的进程中的线程使用

### 8.1 创建或获取命名信号量

* 使用现有的命名信号量时，仅指定2个参数：
  + name：信号量的名字
  + oflag：设为0
* 创建新的命名信号量
  + name：信号量的名字
  + oflag：指定了O\_CREAT标志。当该参数置为O\_CREAT|O\_EXCL并且信号量存在时，函数会失败
  + mode：谁可以访问信号量，值与[open函数](#X4f892631062a259c02243e0c580974547dd1932)的权限位相同
  + value：信号量的初始值（0~SEM\_VALUE\_MAX）

**为了移植性，信号量的命名应该遵循下列规则**：

* 名字的第一个字符应该为斜杠(/)
* 名字不应该包含其他斜杠以此避免实现定义的行为
* 信号量名字的最大长度是实现定义的，不应该鲳鱼\_POSIX\_NAME\_MAX个字符长度。因为这是文件系统的实现能允许的最大名字长度的限制

### 8.2 关闭释放信号量

如果进程没有首先调用sem\_close而退出，那么内核将自动关闭任何打开的信号量

调用该函数，或者内核自动关闭都不会影响信号量值的状态

### 8.3 销毁命名信号量

该函数删除信号量的名字。如果没有打开的信号量引用，则该信号量会被销毁。否则，销毁将延迟到最后一个打开的引用关闭

### 8.4 调节信号量的值

#### 1）减1

不像XSI信号量，我们只能通过一个函数调用来调节POSIX信号量的值

* **sem\_wait函数**：如果信号量计数是0就会发生阻塞。直到成功使信号量减1或者被信号中断时才返回
* **sem\_trywait函数**：可以避免阻塞。当信号量是0时，会返回-1并且将errno置为EAGAIN

函数sem\_timewait可以选择阻塞一段时间

tsptr：绝对时间，超时是基于CLOCK\_REALTIME时钟的

如果信号量可以立即减1，那么超时值就不重要了，此时即使指定的是过去的某个时间，操作依然会成功

如果超时将返回-1，并且将errno置为ETIMEDOUT

#### 2）增1

### 8.5 创建未命名信号量

* pshared：表明是否在多个进程中使用信号量
  + 非0：在多个进程中使用信号量
  + 0：不在多个进程中使用信号量
* sem：未命名信号量，传入其地址，当函数调用返回后，这个未命名信号量会被初始化（如果要在2个进程之间使用信号量，需要确保该参数指向个进程之间共享的内存范围）
* value：初始值

### 8.6 销毁未命名信号量

该函数调用后，不能再使用任何带有sem的信号量函数，除非通过调用sem\_init重新初始化

### 8.7 检索未命名信号量的值

* valp：包含了信号量的值

注意，我们试图要使用刚读出来的值时，信号量的值可能已经改变。除非使用额外的同步机制来避免这种竞争，否则该函数只能用于测试

Mac OS X 10.6.8不支持该函数