内存和编译

C和C++内存模型

- 内存分配方式有三种
  静态存储区域分配，栈上创建，堆上分配
- 由C编译的程序占用的内存分为以下几个部分
  堆，栈，静态全局变量区，常量区
- 由C++编译的程序**占用的内存**分为（堆栈全常代）
  1、堆区（heap）
  由new分配的内存块，其释放编译器不去管，由我们程序自己控制（一个new对应一个delete）。如果程序员没有释放掉，在程序结束时OS会自动回收。涉及的问题：“缓冲区溢出”、“内存泄露”
  2、栈区（stack）
  是那些编译器在需要时分配，在不需要时自动清除的存储区。存放局部变量、函数参数。存放在栈中的数据只在当前函数及下一层函数中有效，一旦函数返回了，这些数据也就自动释放了。
  3、全局区/静态区（.bss段和.data段）
  全局和静态变量被分配到同一块内存中。在C语言中，未初始化的放在.bss段中，初始化的放在.data段中；在C++里则不区分了。
  4、文字常量区（.rodata段）
  存放常量，不允许修改（通过非正当手段也可以修改），程序结束后由系统释放。
  5、程序代码区（.text段）

存放函数体的二进制代码，不允许修改（类似常量存储区），但可以执行（不同于常量存储区）。

- 根据c/c++对象生命周期不同，c/c++的**内存模型**有三种不同的内存区域，即
  自由存储区，动态区、静态区。
  - 自由存储区：局部非静态变量的存储区域，即平常所说的栈
  - 动态区： 用operator new ，malloc分配的内存，即平常所说的堆
  - 静态区：全局变量 静态变量 字符串常量存在位置

> 而代码虽然占内存，但不属于c/c++内存模型的一部分

```
int a = 0; //全局初始化区
char *p1; //全局未初始化区
main() {
int b; // 栈
char s[] = "abc"; //栈     字符串abc保存在栈区，可以通过s去修改
char *p2; //栈
const char * arr = "123"; //字符串123保存在常量区（不能被改变），const是修饰arr指向的值不能通过arr去修改，由于字符串“123”在常量区，加不加const效果都一样
char *p3 = "123456"; //p3在栈上，"123456/0"在常量区，不能通过p3去修改"123456/0"的值
const char crr[] = "123"; //这里123本来是在栈上的，但是编译器可能会做某些优化，将其放到常量区
static int c =0;  //全局（静态）初始化区
p1 = (char *)malloc(10);
p2 = (char *)malloc(20); //分配得来得10和20字节的区域就在堆区。
strcpy(p1, "123456"); //123456/0放在常量区，编译器可能会将它与p3所指向的"123456"优化成一个地方。
}
```

- 学会迁移，可以说到malloc，从malloc说到操作系统的内存管理，说道内核态和用户态
  比如C函数库中的内存分配函数malloc（ ），它具体是使用sbrk（ ）系统调用来分配内存，当malloc调用sbrk（ ）的时候就涉及一次从用户态到内核态的切换，类似的函数还有printf（ ），调用的是wirte（ ）系统调用来输出字符串，等等。

  > 什么高端内存，slab层，伙伴算法，VMA，接着可以迁移到fork()。（选学装逼技能，这里先跳过）

- 堆与栈表示的两种内存管理方式
  1、栈由操作系统自动分配释放 ，用于存放函数的参数值、局部变量等，其操作方式类似于数据结构中的栈。
  其中函数中定义的局部变量按照先后定义的顺序依次压入栈中，也就是说相邻变量的地址之间不会存在其它变量。栈的内存地址生长方向与堆相反，由高到底，所以后定义的变量地址低于先定义的变量，比如上面代码中变量s的地址小于变量b的地址，p2地址小于s的地址。栈中存储的数据的生命周期随着函数的执行完成而结束。
  2、堆由程序员分配释放， 若程序员不释放，程序结束时由OS回收，分配方式倒是类似于链

表。

```
int main(){
 char* p1 = (char *)malloc(10); //C中使用malloc函数申请
 cout<<(int*)p1<<endl;  //输出：00000000003BA0C0
 free(p1); //使用free()释放

 char p2 = new char[10]; //C++中用new运算符申请
 cout<<(int*)p2<<endl;  //输出：00000000003BA0C0 之前申请的空间被释放，所以申请同样大小空间时地址一样的
 delete[] p2; //使用delete运算符释放
}
```

其中p1所指的10字节的内存空间与p2所指的10字节内存空间都是存在于堆的。堆的内存地址生长方向与栈相反，由低到高，但需要注意的是，后申请的内存空间并不一定在先申请的内存空间的后面，即p2指向的地址并不一定大于p1所指向的内存地址

> 原因是先申请的内存空间一旦被释放，后申请的内存空间则会利用先前被释放的内存，从而导致先后分配的内存空间在地址上不存在先后关系。堆中存储的数据的若未释放，则其生命周期等同于程序的生命周期。

关于堆上内存空间的分配过程，首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆节点，然后将该节点从空闲节点链表中删除，并将该节点的空间分配给程序（系统会自动的将多余的那部分重新放入空闲链表）。

> 对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的delete语句才能正确地释放本内存空间。

- 堆和栈的区别
  1)分配和管理方式不同 ：
  堆是动态分配的，其空间的分配和释放都由程序员控制。
  栈由编译器自动管理。栈有两种分配方式：静态分配和动态分配。静态分配由编译器完成，比如局部变量的分配。动态分配由alloca()函数进行分配，但是栈的动态分配和堆是不同的，它的动态分配是由编译器进行释放，无须手工控制。
  2)产生碎片不同
  对堆来说，频繁的new/delete或者malloc/free势必会造成内存空间的不连续，造成大量的碎片，使程序效率降低。
  对栈而言，则不存在碎片问题，因为栈是先进后出的队列，永远不可能有一个内存块从栈中间弹出。
  3)生长方向不同
  堆是向着内存地址增加的方向增长的，从内存的低地址向高地址方向增长。
  栈是向着内存地址减小的方向增长，由内存的高地址向低地址方向增长。
  4)分配效率不同
  堆则通过库函数或运算符来申请和管理，实现机制复杂，容易产生内存碎片。

栈由操作系统自动分配，会在硬件层级栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，栈的效率比较高。
5)存放内容不同
栈存放的内容，函数返回地址、相关参数、局部变量和寄存器内容等。
堆，一般情况堆顶使用一个字节的空间来存放堆的大小，而堆中具体存放内容是由程序员来填充的。

- 主函数调用另外一个函数的时候，要对当前函数执行断点进行保存，需要使用栈来实现
  首先入栈的是主函数下一条语句的地址，即扩展指针寄存器的内容（EIP），然后是当前栈帧的底部地址，即扩展基址指针寄存器内容（EBP），再然后是被调函数的实参等，一般情况下是按照从右向左的顺序入栈，之后是被调函数的局部变量，注意静态变量是存放在数据段或者BSS段，是不入栈的。出栈的顺序正好相反，最终栈顶指向主函数下一条语句的地址，主程序又从该地址开始执行。
- 栈溢出的可能性
  1、程序中出先死循环调用
  2、栈不够使用，比如在栈中一个1M的局部变量，则会导致这个问题。

关于静态内存分配和动态内存分配的区别及过程

最大的区别：
直到程序运行的时候，执行动态分配；而在编译的时候，已经决定好了分配多少text+data+bss+stack
通过malloc()动态分配的内存，需要程序员手工调用free()释放内存，否则容易导致内存泄露；而静态分配的内存则在进程执行结束后系统释放(Text, Data), Stack段中的数据很短暂，函数退出立即被销毁。

静态内存	动态内存
编译时完成的，不占用CPU资源	运行时完成，分配与释放需要占用CPU资源
按计划分配，在编译前确定内存块的大小	运行时按需分配
内存的控制权交给了编译器	内存的控制权交给了程序员
运行效率要比动态分配内存的效率要高（动态内存分配与释放需要额外的开销）	动态内存管理水平严重依赖于程序员的水平，处理不当容易造成内存泄漏

伙伴系统算法

在实际应用中，经常需要分配一组连续的页框，而频繁地申请和释放不同大小的连续页框，必然导致在已分配页框的内存块中分散了许多小块的空闲页框。这样，即使这些页框是空闲的，其他需要分配连续页框的应用也很难得到满足。
为了避免出现这种情况，Linux内核中引入了伙伴系统算法(buddy system)。把所有的空闲页框分组为

11个块链表，每个块链表分别包含大小为1，2，4，8，16，32，64，128，256，512和1024个连续页框的页框块。最大可以申请1024个连续页框，对应4MB大小的连续内存。每个页框块的第一个页框的物理地址是该块大小的整数倍。
假设要申请一个256个页框的块，先从256个页框的链表中查找空闲块，如果没有，就去512个页框的链表中找，找到了则将页框块分为2个256个页框的块，一个分配给应用，另外一个移到256个页框的链表中。如果512个页框的链表中仍没有空闲块，继续向1024个页框的链表查找，如果仍然没有，则返回错误。
页框块在释放时，会主动将两个连续的页框块合并为一个较大的页框块。

内存碎片和外部碎片

- 外部碎片
  是由于大量信息由于先后写入、置换、删除而形成的空间碎片。
  为了便于理解，我们将信息比作货物，将存储空间比作仓库来举例子。假设，我们有编号为1、2、3、4、5、6的6间仓库库房，前天送来了一大宗货，依次装入了1、2、3、4、5号仓库，昨天又因故将4号库房的货物运走了，那么数值上说我们还有两间空仓库的空间，但是如果这时候送来两间仓库容量的货物但要求必须连续存放的话，我们实际上是装不下的。这时的4、6号仓库，就成为一种空间的碎片。由于这样的原因形成的空间碎片，我们称之为**外部碎片**。
  从上面的例子我们可以理解，外部碎片是可以通过一些措施来改善或者解决的。对于在硬盘上的外部碎片，我们通常用磁盘碎片整理来解决，对应上面的例子，就是将5号仓库的货物及时移动到新腾出的4号仓库，这样，1-4号仓库都是满的，而5、6号仓库则形成了有效的、连续的空间，能够适应新的应用要求了；对于内存中的外部碎片，我们内存管理中常用的页面管理形式，就是为了解决这个问题的。这里就不详述了。
- 内部碎片
  是由于存量信息容量与最小存储空间单位不完全相符而造成的空间碎片。还是沿用上面的例子，这次我们的6间仓库目前都是空置的，但是假设我们管理仓库的最小空间单位是间，今天运来了容量为2.5间仓库的货物，那也要占用我们1-3号3间仓库，尽管3号仓库还闲置着一半的空间，但是这半间仓库已经不能再利用了（因为是以间为最小单位么）；这时，我们的仓库中就形成了半间仓库的空间碎片，仓库的有效容量只剩下3间仓库了。

静态连接与动态链接的区别

- 静态链接
  所谓静态链接就是在编译链接时直接将需要的执行代码拷贝到调用处，优点就是在程序发布的时候就不需要依赖库，也就是不再需要带着库一块发布，程序可以独立执行，但是体积可能会相对大一些。
- 动态链接
  所谓动态链接就是在编译的时候不直接拷贝可执行代码，而是通过记录一系列符号和参数，在程序运行或加载时将这些信息传递给操作系统，操作系统负责将需要的动态库加载到内存中，然后程序在运行到指定的代码时，去共享执行内存中已经加载的动态库可执行代码，最终达到运行时连接的目的。优点是多个程序可以共享同一段代码，而不需要在磁盘上存储多个拷贝，缺点是由于是运行时加载，可能会影响程序的前期执行性能。

静态链接库和动态链接库的实现

- 静态链接为a库

```
几个嗲用的函数源文件，如test1.c，test2.c
gcc -c test1.c test2.c //开始编译，将源文件编译成.o文件,生成test1.o和test2.o
ar rcs libtest.a test1.o test2.o//生成a库，链接库前缀必须以lib开头，生成libtest.a
得到libtest.a库文件后，然后将.a库连接到主程序 中，写主程序对库函数进行声明调用。
gcc main.c -L. -ltest -o test //加载a库，生成可执行文件并执行
```

- 动态链接so库，编译代码如下：

```
gcc test1.c test2.c -fPIc -shared -o libtest.so //so库的前缀必须为lib，然后将.so库链接到主程序main.c中
gcc main.c -L. -ltest -o main  //so库复制到默认目录下
```

> 因为动态库的特性，编译器会到指定的目录去寻找动态库，目录的地址在/etc/ld.so.conf.d/ 目录里的libc.conf文件里，你可以在里面加一行地址表示你so库的位置，更改完conf文件里的内容，记得输入命令行：ldconfig。
> 或者将so库复制到默认的目录下，生成可执行文件并运行

- 加载动态链接库的时候，有可能会遇到加载不到的错误，原因在于系统默认加载的动态链接库路径里没有找到你的动态库，有三种解决方法：
  a. 在执行gcc main.c -L. -ltest -o main 前，执行 export LD_LIBRARY_PATH=$(pwd)
  b. 将你so所在的目录写到/etc/ld.so.conf文件里，然后执行ldconfig。
  c. 将你的so放在/etc/ld.so.conf里的路径位置里。

动态链接库的两种使用方法及特点

1)载入时动态链接，模块非常明确调用某个导出函数，使得他们就像本地函数一样。这需要链接时链接那些函数所在DLL的导入库，导入库向系统提供了载入DLL时所需的信息及DLL函数定位。
2)运行时动态链接。

硬链接和软链接

- 硬链接
  一般情况下，文件名和inode号码是"一一对应"关系，每个inode号码对应一个文件名。但是，Unix/Linux系统允许，多个文件名指向同一个inode号码。
  意味着：
  - 可以用**不同的文件名访问同样的内容**；
  - 对文件内容进行修改，会影响到所有文件名；
  - 但是，删除一个文件名，不影响另一个文件名的访问。

这种情况就被称为"硬链接"（hard link）。

```
ln 源文件 目标文件
```

- 软链接
  文件A和文件B的inode号码虽然不一样，但是文件A的内容是文件B的路径。读取文件A时，系统会自动将访问者导向文件B。因此，无论打开哪一个文件，最终读取的都是文件B。这时，文件A就称为文件B的"软链接"（soft link）或者"符号链接（symbolic link）。
  意味着：
  - 文件A依赖于文件B而存在，如果删除了文件B，打开文件A就会报错："No such file or directory"。
    这是软链接与硬链接最大的不同：文件A指向文件B的文件名，而不是文件B的inode号码，文件B的inode"链接数"不会因此发生变化。

```
ln -s 源文文件或目录 目标文件或目录
```

ELF文件

ELF是一种用于二进制文件、可执行文件、目标代码、共享库和核心转储格式文件。可以减少重新编程重新编译的代码。ELF文件格式提供了两种视图，分别是链接视图和执行视图。链接视图是以节（section）为单位，执行视图是以段（segment）为单位。

> bss段代表的是未初始化的全局变量，在ELF文件中是不占大小的。data段是占字节的。

1. 可重定位的对象文件(Relocatable file)(没有segments)
   这是由汇编器汇编生成的 .o 文件。后面的链接器(link editor)拿一个或一些 Relocatable object files 作为输入，经链接处理后，生成一个可执行的对象文件 (Executable file) 或者一个可被共享的对象文件(Shared object file)。我们可以使用 ar 工具将众多的 .o Relocatable object files 归档(archive)成 .a 静态库文件。
2. 可执行的对象文件(Executable file)
3. 可被共享的对象文件(Shared object file)，就是所谓的动态库文件，也即 .so 文件。
   > section 是被链接器使用的，但是 segments 是被加载器所使用的。加载器会将所需要的 segment 加载到内存空间中运行。

在ELF文件里面，每一个 sections 内都装载了性质属性都一样的内容，比方：
1) .text section 里装载了可执行代码；
2) .data section 里面装载了被初始化的数据；
3) .bss section 里面装载了未被初始化的数据；
4) 以 .rec 打头的 sections 里面装载了重定位条目；
5) .symtab 或者 .dynsym section 里面装载了符号信息；
6) .strtab 或者 .dynstr section 里面装载了字符串信息；
7) 其他还有为满足不同目的所设置的section，比方满足调试的目的、满足动态链接与加载的目的等

等。

把带有相同属性(比方都是只读并可加载的)的 section 都合并成所谓 segments(段)。最重要的是三个 segment：代码段，数据段和堆栈段。

当ELF文件被加载到内存中后，系统会将多个具有相同权限（flg值）section合并一个segment。
操作系统往往以页为基本单位来管理内存分配，一般页的大小为4096B，即4KB的大小。同时，内存的权限管理的粒度也是以页为单位。ELF文件在被映射时，是以系统的页长度为单位的，那么每个section在映射时的长度都是系统页长度的整数倍，如果section的长度不是其整数倍，则导致多余部分也将占用一个页。而我们从上面的例子中知道，一个ELF文件具有很多的section，那么会导致内存浪费严重。
这样可以减少页面内部的碎片，节省了空间，显著提高内存利用率。

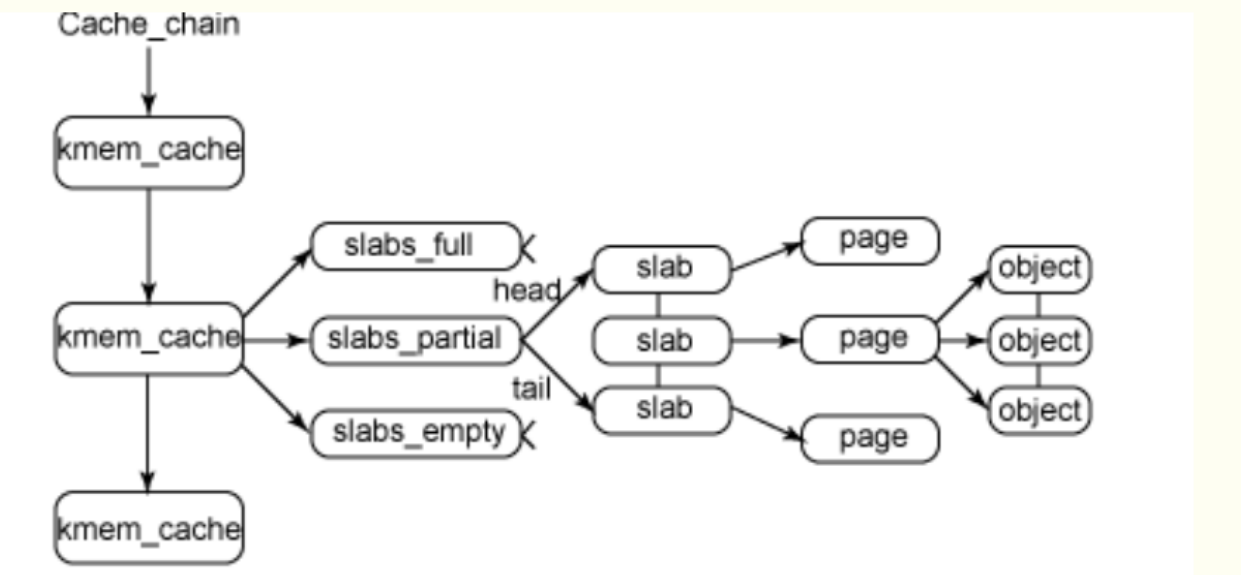
slab

slab是Linux操作系统的一种内存分配机制，slab分配算法采用cache 存储内核对象。slab 缓存、从缓存中分配和释放对象然后销毁缓存的过程必须要定义一个 kmem_cache 对象，然后对其进行初始化这个特定的缓存包含 32 字节的对象

分配器

slab是Linux操作系统的一种内存分配机制。其工作是针对一些**经常分配并释放的对象**，如进程描述符等，这些对象的大小一般比较小，如果直接采用伙伴系统来进行分配和释放，不仅会造成大量的内存碎片，而且处理速度也太慢。

- slab分配器是基于对象进行管理的
  相同类型的对象归为一类(如进程描述符就是一类)，每当要申请这样一个对象，slab分配器就从一个slab列表中分配一个这样大小的单元出去，而当要释放时，将其重新保存在该列表中，而不是直接返回给伙伴系统，从而避免这些内碎片。slab分配器并不丢弃已分配的对象，而是释放并把它们保存在内存中。当以后又要请求新的对象时，就可以从内存直接获取而不用重复初始化。
- 对象高速缓存的组织如右下图所示，高速缓存的内存区被划分为多个slab，每个slab由一个或多个连续的页框组成（通常只有一页），这些页框中既包含已分配的对象，也包含空闲的对象。



简要分析下这个图：kmem_cache是一个cache_chain的链表，描述了一个高速缓存，每个高速

缓存包含了一个slabs的列表，这通常是一段连续的内存块。
存在3种slab：slabs_full(完全分配的slab),slabs_partial(部分分配的slab),slabs_empty(空slab,或者没有对象被分配)。
slab是slab分配器的最小单位，在实现上一个slab有一个货多个连续的物理页组成（通常只有一页）。
单个slab可以在slab链表之间移动，例如如果一个半满slab被分配了对象后变满了，就要从slabs_partial中被删除，同时插入到slabs_full中去。

分配算法编辑

slab分配算法采用cache 存储内核对象。

- 当创建cache 时，起初包括若干标记为空闲的对象。
  对象的数量与slab的大小有关。开始，所有对象都标记为空闲。当需要内核数据结构的对象时，可以直接从cache 上直接获取，并将对象初始化为使用。
- 下面考虑内核如何将slab分配给表示进程描述符的对象。
  在Linux系统中，进程描述符的类型是struct task_struct ，其大小约为1.7KB。当Linux 内核创建新任务时，它会从cache 中获得struct task_struct 对象所需要的内存。Cache 上会有已分配好的并标记为空闲的struct task_struct 对象来满足请求。
- Linux 的slab 可有三种状态：
  满的：slab 中的所有对象被标记为使用。
  空的：slab 中的所有对象被标记为空闲。
  部分：slab 中的对象有的被标记为使用，有的被标记为空闲。
- slab 分配器首先从部分空闲的slab 进行分配。如有，则从空的slab 进行分配。如没有，则从物理连续页上分配新的slab，并把它赋给一个cache ，然后再从新slab 分配空间

优点

与传统的内存管理模式相比， slab 缓存分配器提供了很多优点。
1、内核通常依赖于对小对象的分配，它们会在系统生命周期内进行无数次分配。
2、slab 缓存分配器通过对类似大小的对象进行缓存而提供这种功能，从而避免了常见的碎片问题。
3、slab 分配器还支持通用对象的初始化，从而避免了为同一目的而对一个对象重复进行初始化。
4、slab 分配器还可以支持硬件缓存对齐和着色，这允许不同缓存中的对象占用相同的缓存行，从而提高缓存的利用率并获得更好的性能。

栈大小

编译期限制栈大小，和系统限制栈深度根本是两回事。

- 系统限制栈深是限制进程主线程的栈深，限制的是整个函数调用链的最大栈深，这个栈深是函数调用链上各个函数栈帧大小之和。
- 编译期限制栈大小是限制单个函数栈帧的大小。

linux系统下默认栈大小是10M,windows系统下默认栈大小是1M.

多线程和多进程

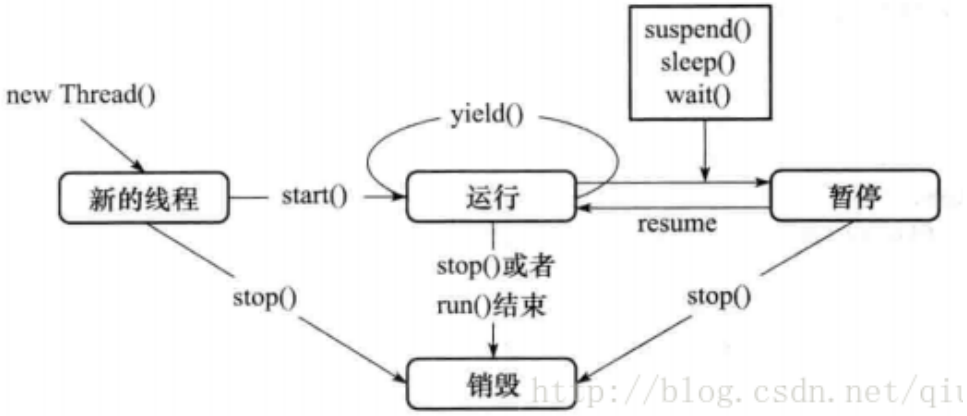
线程和进程

- 线程常用的函数

函数	说明
pthread_create()	创建线程开始运行相关线程函数，运行结束则线程退出
pthread_eixt()	因为exit()是用来结束进程的，所以则需要使用特定结束线程的函数
pthread_join()	挂起当前线程，用于阻塞式地等待线程结束，如果线程已结束则立即返回，0=成功
pthread_cancel()	发送终止信号给thread线程，成功返回0，但是成功并不意味着thread会终止
pthread_attr_init()	初始化配置一个线程对象的属性,需要用pthread_attr_destroy函数去除已有属性
pthread_attr_setscope()	设置线程属性
pthread_attr_setschedparam()	设置线程优先级
pthread_attr_getschedparam()	获取线程优先级

> 还有锁的函数，以及信号量的函数，这里就不展开了

- 线程的几种状态：
  1）就绪：参与调度，等待被执行，一旦被调度选中，立即开始执行
  2）运行：占用CPU，正在运行中
  3）休眠：暂不参与调度，等待特定事件发生
  4）中止：已经运行完毕，等待回收线程资源



- 主线程等待子线程退出：

当thread::join()函数被调用后，调用它的线程会被block，join的作用是让主线程等待直到线程的执行被完成，是用来知道一个线程已结束的机制。当thread::join()返回时，OS的执行的线程已经完成，C++线程对象可以被销毁。

- 线程的锁：
    - mutex是用来保证线程同步的，防止不同的线程同时操作同一个共享数据。
    - 使用lock_guard则相对安全，它是基于作用域的，能够自解锁，当该对象创建时，它会像m.lock()一样获得互斥锁，当生命周期结束时，它会自动析构(unlock)，不会因为某个线程异常退出而影响其他线程。

- 线程退出的方法
  1、线程函数返回（推荐）
  2、.调用 _endthreadex()函数 或 ExitThread()函数(最好不要，不会调用线程函数作用域内申请的类对象的析构函数，容易造成内存泄漏):
  3、用同一个进程中的另一个线程调用 TerminateThread()函数(必须避免);
  4、终止该线程所在的进程(绝对避免);

线程的上下文切换

CPU通过给每个线程分配CPU时间片实现多线程，时间片是CPU分配给各个线程的时间，因为时间片非常短，所以CPU通过不停地切换线程执行，让我们感觉多个线程时同时执行的，时间片一般是几十毫秒（ms）。
CPU通过**时间片分配算法**来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。但是，在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再次加载这个任务的状态，从任务保存到再加载的过程就是一次**上下文切换。**

> 上下文切换也会影响多线程的执行速度（上下文切换的开销），所以线程数尽量和核数相等。

- 引起线程上下文切换的原因
  对于我们经常使用的抢占式操作系统而言，引起线程上下文切换的原因大概有以下几种：
    - 当前执行任务的时间片用完之后，系统CPU正常调度下一个任务
    - 当前执行任务碰到IO阻塞，调度器将此任务挂起，继续下一任务
    - 多个任务抢占锁资源，当前任务没有抢到锁资源，被调度器挂起，继续下一任务
    - 用户代码挂起当前任务，让出CPU时间
    - 硬件中断

- 查看上下文切换次数
  使用vmstat命令（Lmbench3可以查看上下文的切换时长）
- 如何减少上下文切换
  无锁并发编程
  使用最少线程

多线程和多进程的区别

（重点 必须从cpu调度，上下文切换，数据共享，多核cup利用率，资源占用，等等各方面回答，然后有一个问题必须会被问到：哪些东西是一个线程私有的？答案中必须包含寄存器，否则悲催）！

多进程	多线程
进程数据是分开的，共享复杂，需要用IPC，同步简单	共享进程数据：共享简单，同步复杂
创建销毁慢、切换复杂，速度慢	创建销毁快、切换简单，速度快
占用内存多， CPU利用率低	占用内存少， CPU利用率高
编程简单，调试简单	编程复杂，调试复杂
进程间不会相互影响	一个线程挂掉将导致整个进程挂掉
适用于多机分布	适应于多核

- 进程与线程的选择取决以下几点
  1、需要**频繁创建销毁**的优先使用**线程**；因为对进程来说创建和销毁一个进程代价是很大的。
  2、线程的切换速度快，所以在需要**大量计算，切换频繁**时用**线程**，还有耗时的操作使用线程可提高应用程序的响应
  3、因为对**CPU系统的效率**使用上**线程**更占优，所以可能要发展到多机分布的用进程，多核分布用线程；
  4、**并行操作**时使用**线程**，如C/S架构的服务器端并发线程响应用户的请求；
  5、需要更**稳定安全**时，适合选择**进程**；需要速度时，选择线程更好。
- 线程独立资源包括：
  线程id、寄存器的值、线程栈、线程的优先级和调度策略、线程的私有数据、信号屏蔽字、errno变量

1. 线程ID
   每个线程都有自己的线程ID，这个ID在本进程中是唯一的。进程用此来标识线程。
2. 寄存器组的值
   由于线程间是并发运行的，每个线程有自己不同的运行线索，当从一个线程切换到另一个线程上时，必须将原有的线程的寄存器集合的状态保存，以便将来该线程在被重新切换到时能得以恢复。
3. 线程的堆栈
   堆栈是保证线程独立运行所必须的。线程函数可以调用函数，而被调用函数中又是可以层层嵌套的，所以线程必须拥有自己的函数堆栈， 使得函数调用可以正常执行，不受其他线程的影响。
4. 错误返回码
   由于同一个进程中有很多个线程在同时运行，可能某个线程进行系统调用后设置了errno值，而在该 线程还没有处理这个错误，另外一个线程就在此时被调度器投入运行，这样错误值就有可能被修改。所以，不同的线程应该拥有自己的错误返回码变量。
5. 线程的信号屏蔽码
   由于每个线程所感兴趣的信号不同，所以线程的信号屏蔽码应该由线程自己管理。但所有的线程都 共享同样的信号处理器。

6. 线程的优先级
   由于线程需要像进程那样能够被调度，那么就必须要有可供调度使用的参数，这个参数就是线程的优先级。

- 智能指针是线程安全的吗
  不是。引用计数是atomic，shardptr只读数据的话是安全的。写的时候，如果会减少老sharedptr的引用计数，则需要加锁。

进程和内核的通信方式

主要是procfs（系统模拟出来的文件系统），netlink（类似sock的方法），syscall（通过注册字符设备使用mmap和ioctl来操作）

- 用户态通过系统暴露出来的系统调用来进行操作，如mmap，ioctl，open，close，read，write
- 内核态通过建立共享内存remap_pfn_range或者copy_to_user, copy_from_user来进行操作。

> 选择哪种方式需要考虑是用户态单进程与内核态通信还是多进程的通信，还要考虑通信的数据量。根据不同的需求来使用不同的方法。

- 内核和用户空间进行通信
  - 采用内存映射的方式，将内核地址映射到用户态。
    这种方式最直接，可以适用大量的数据传输机制。这种方式的缺点是很难进行“业务控制”，没有一种可靠的机制保障内核和用户态的调动同步，比如信号量等都不能跨内核、用户层使用。因此内存映射机制一般需要配合一种“消息机制”来控制数据的读取，比如采用“消息”类型的短数据通道来完成一个可靠的数据读取功能。（syscall）
  - 内核态主动发起消息的通知方式，而用户态的程序最好可以采用一种“阻塞调用”的方式等待消息。这样的模型可以最大限度的节省CPU的调度，同时可以满足及时处理的要求，使用netlink完成通信的过程（类似socket的一个方法）。

多线程同步

多线程同步的方式：锁、信号量、CAS

锁

线程的最大特点是**资源的共享性**，但资源共享中的**同步问题**是多线程编程的难点。

- linux下提供了多种方式来处理线程同步，最常用的是**互斥锁、条件变量和信号量**。
  - 互斥锁（mutex；lock_guard；unique_lock）+条件变量

信号量

```
#include <semaphore.h>
sem_t sem; //声明一个有名信号量
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

```
//sem：指我们所需要初始化的这个信号量。pshared：一般都是用0，表示这个信号量在一个进程里面的线程之间共享。value：就是我们要初始化信号量的值是多少。返回值：成功返回0，失败返回-1；
//初始化好了之后，就可以直接调用对应的函数进行P/V操作
int sem_wait(sem_t *sem); //这个就是P操作，这个是阻塞型的，会一直等待有可用的资源。
int sem_trywait(sem_t *sem);// 这个也是P操作，但是这个不阻塞，如果没有可用的资源就立刻返回。
int sem_post(sem_t *sem); //这个是V操作。返回值： 成功返回0，失败返回-1；
int sem_destroy(sem_t *sem); //用完之后，销毁它。返回值：成功返回0，失败返回-1；
```

CAS

CAS （ compareAndSwap ），中文叫比较交换，一种无锁原子算法。（ 首先保存寄存器的值，感觉就是比较内存和寄存器的值 ）

- 过程是这样：
  它包含 3 个参数 CAS（ V，E，N ），V表示要更新变量的值，E表示预期值，N表示新值。
  - 仅当 V值等于E值时，才会将V的值设为N
  - 如果V值和E值不同，则说明已经有其他线程做两个更新，则当前线程则什么都不做。
  - 最后，CAS 返回当前V的真实值。CAS 操作时抱着乐观的态度进行的，它总是认为自己可以成功完成操作。

当多个线程同时使用CAS 操作一个变量时，只有一个会胜出，并成功更新，其余均会失败。失败的线程不会挂起，仅是被告知失败，并且允许再次尝试，当然也允许实现的线程放弃操作。基于这样的原理，CAS 操作即使没有锁，也可以发现其他线程对当前线程的干扰。
与锁相比，使用CAS会使程序看起来更加复杂一些，但由于其非阻塞的，它对死锁问题天生免疫，并且，线程间的相互影响也非常小。更为重要的是，使用无锁的方式完全没有锁竞争带来的系统开销，也没有线程间频繁调度带来的开销，因此，他要比基于锁的方式拥有更优越的性能。

```
// linux  #include <pthread>
bool __sync_bool_compare_and_swap (type *ptr, type oldval type newval, ...)
type  __sync_val_compare_and_swap (type *ptr, type oldval type newval, ...)
// windows 以及 C++11的 CAS
```

线程间通信方式

- 共享数据
  线程之间的通信比较方便。统一进程下的线程共享数据有**全局变量，静态变量，堆，子进程**，通过这些数据来通信不仅快捷而且方便，处理好**访问的同步与互斥**是编写多线程的难点。

  > 一个线程中打开的文件各个线程均可用。

信号和信号量

1. 信号：（ signal ）

是一种处理异步事件的方式。信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于**进程**外，还可以发送信号给进程本身。

2. 信号量：（Semaphore）
    - 信号量分为：
      POSIX信号量和System V信号量（POSIX信号量更加有优势）。
    - 其中POSIX信号量分为：**有名信号量和无名信号量**，
      无名信号量也被称作基于内存的信号量，如果不放在进程间的共享内存区中，是不能用来进行进程间同步的，只能用来进行线程同步。
      有名信号量通过IPC名字进行进程间的同步，
    - 进程间通信处理同步互斥的机制。是在多线程环境下使用的一种设施, 它**负责协调各个线程**, 以保证它们能够正确、合理的使用公共资源。

多线程锁的种类有哪些

互斥锁（mutex）、条件锁、自旋锁、读写锁、递归锁

- 互斥锁
  互斥锁用于控制多个线程对他们之间共享资源互斥访问的一个信号量。也就是说是为了避免多个线程在某一时刻同时操作一个共享资源。

  > 例如线程池中的有多个空闲线程和一个任务队列。任何是一个线程都要使用互斥锁互斥访问任务队列，以避免多个线程同时访问任务队列以发生错乱。
  > 在某一时刻，只有一个线程可以获取互斥锁，在释放互斥锁之前其他线程都不能获取该互斥锁。如果其他线程想要获取这个互斥锁，那么这个线程只能以阻塞方式进行等待。

- 条件锁
  条件锁就是所谓的条件变量，某一个线程因为某个条件为满足时可以使用条件变量使改程序处于阻塞状态。一旦条件满足以“信号量”的方式唤醒一个因为该条件而被阻塞的线程。

  > 最为常见就是在线程池中，起初没有任务时任务队列为空，此时线程池中的线程因为“任务队列为空”这个条件处于阻塞状态。一旦有任务进来，就会以信号量的方式唤醒一个线程来处理这个任务。这个过程中就使用到了条件变量pthread_cond_t。
  > 实际应用中，为什么条件变量与pthread_mutex 经常一起使用呢？
  > 1.cond_wait函数解锁并等待是一个原子操作，不可以被打断；
  > 2.等待函数返回之前，重新锁定，如果不锁定，其他线程可能会对这个线程进行修改。

- 自旋锁
  如果一个线程想要获取一个被使用的自旋锁，那么它会一致占用CPU请求这个自旋锁使得CPU不能去做其他的事情，直到获取这个锁为止，这就是“自旋”的含义。当发生阻塞时，互斥锁可以让CPU去处理其他的任务；而自旋锁让CPU一直不断循环请求获取这个锁。

  > 应用在实时性要求较高的场合（缺点：CPU浪费较大）

- 读写锁
  写者：写者使用写锁，如果当前没有读者，也没有其他写者，写者立即获得写锁；否则写者将等

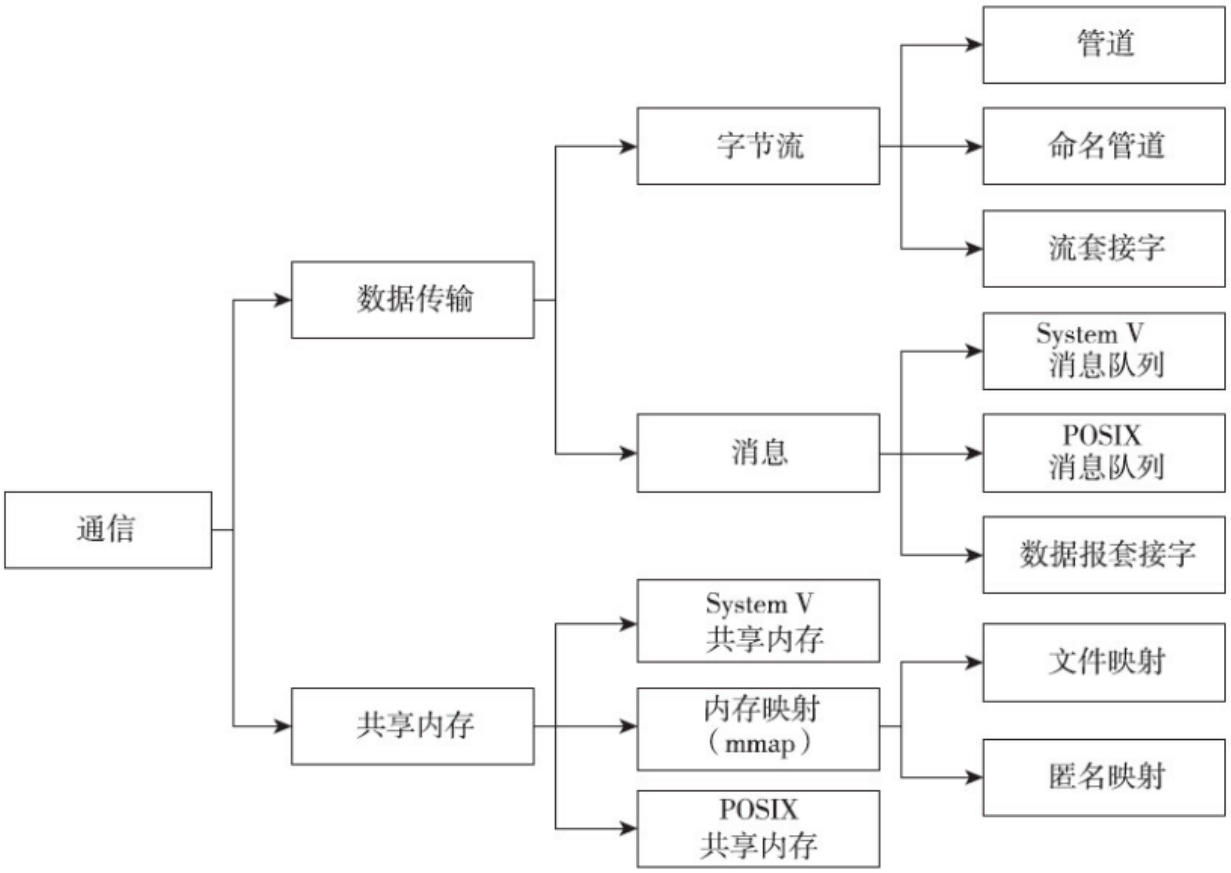
待，直到没有读者和写者。
读者：读者使用读锁，如果当前没有写者，读者立即获得读锁；否则读者等待，直到没有写者。
读读共享， 读写互斥，写优先级高（同时到达）

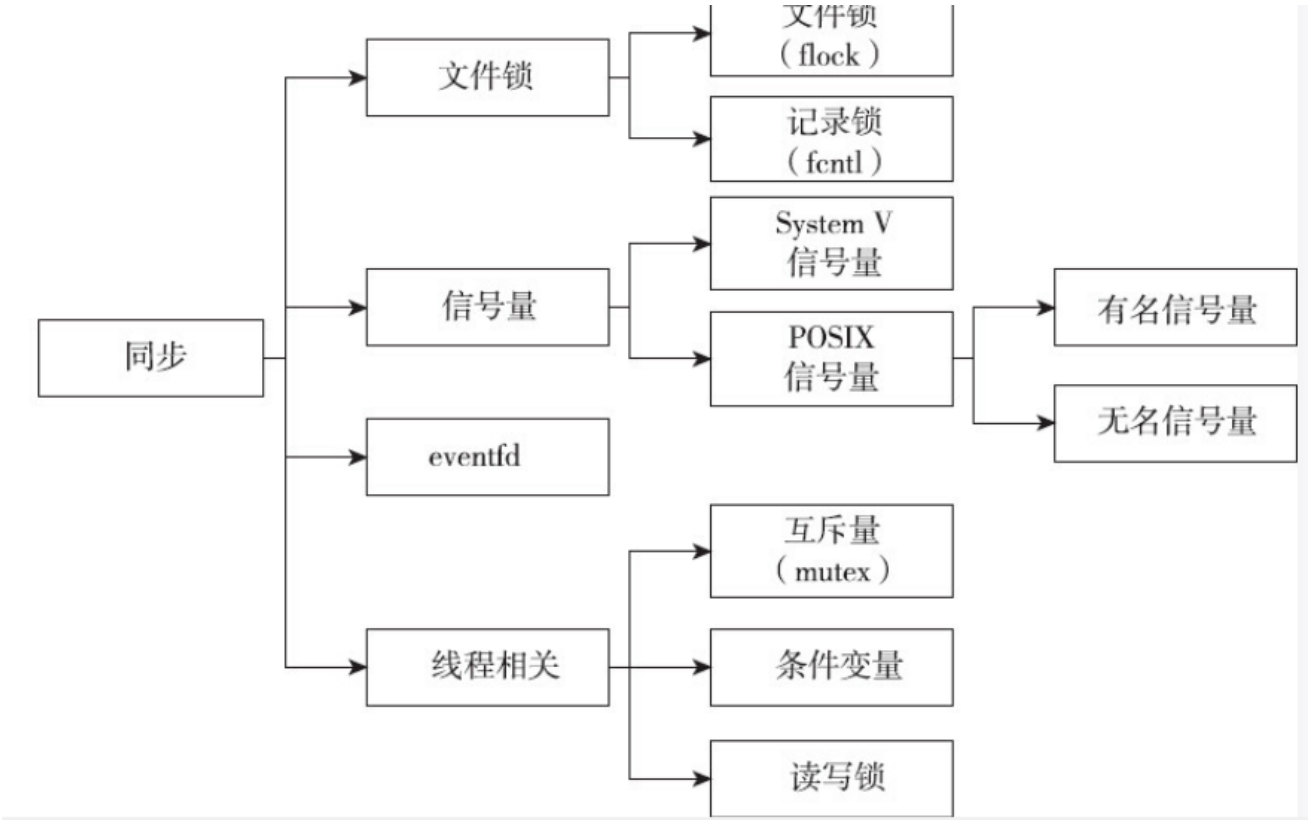
> 应用场景---大量的读操作 较少的写操作

- 递归所（重入锁）
  - 广义上的可重入锁指的是可重复可递归调用的锁，在外层使用锁之后，在内层仍然可以使用，并且不发生死锁（前提得是同一个对象或者class），这样的锁就叫做可重入锁。
  - 不可重入锁，与可重入锁相反，不可递归调用，递归调用就发生死锁。
    二者唯一的区别是，同一个线程可以多次获取同一个递归锁，不会产生死锁。而如果一个线程多次获取同一个非递归锁，则会产生死锁。
- 多线程程序架构中，线程数量应该如何设置？
  应尽量和CPU核数相等或者为CPU核数+1的个数

进程间通信（IPC）五种通信方式

管道、消息队列、共享内存、信号量、套接字





参考链接：https://blog.csdn.net/wh_sjc/article/details/70283843

管道

- 特点
  管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道；
  单独构成一种独立的文件系统：管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在与内存中。
  它可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。通过内核缓冲区实现数据传输。
- 无名管道
  它是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端。
  它只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间）。

```
#include <unistd.h>
int pipe(int fd[2]); // 返回值：若成功返回0，失败返回-1
```

```
    if((r_num=read(pipe_fd[0],buf_r,100))>0)
printf("子进程从管道读取%d个字符，读取的字符串是：%s\n",r_num,buf_r);
    if(write(pipe_fd[1],buf_w,strlen(buf_w))!=-1)
printf("父进程向管道写入：%s\n",buf_w);
```

通常调用 pipe 的进程接着调用 fork，这样就创建了父进程与子进程之间的 IPC 通道

- 命名管道（FIFO）

FIFO可以在无关的进程之间交换数据，与无名管道不同。FIFO有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中（命名管道的名字对应于一个磁盘索引节点，有了这个文件名，任何进程有相应的权限都可以对它进行访问）。

```
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);// 返回值：成功返回0，出错返回-1
```

```
struct timeval net_timer;
mkfifo("fifo1",S_IWUSR|S_IRUSR|S_IRGRP|S_IROTH); /* mkfifo 函数创建命名管道 */
mkfifo("fifo2",S_IWUSR|S_IRUSR|S_IRGRP|S_IROTH); /* mkfifo 函数创建命名管道 */
wfd = open("fifo1",O_WRONLY); /* 以写方式打开管道文件 */
rfd = open("fifo2",O_RDONLY); /* 以只读方式打开管道文件 */
read(rfd,str,sizeof(str)); /* 读取管道,将管道内容存入 str 变量 */
printf("----------\n");
printf("李四：%s\n",str); /* 打印输出 str 变量内容 */

printf("----------\n");
fgets(str,sizeof(str),stdin);
len = write(wfd,str,strlen(str)); /* 写入管道 */
```

FIFO的通信方式类似于在进程中使用文件来传输数据，只不过FIFO类型文件同时具有管道的特性。在数据读出时，FIFO管道中同时清除数据，并且**“先进先出”**。

消息队列

消息队列，是消息的链接表，存放在**内核中**。一个消息队列由一个标识符（即队列ID）来标识。用户进程可以向消息队列添加消息，也可以向消息队列读取消息。

- 特点
  消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级。
  消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除。
  消息队列可以实现消息的**随机查询**,消息**不一定要以先进先出的次序读取**,也可以按消息的类型读取。

  > 消息队列克服了 之前信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限 等缺点

- 例子
  用函数msgget创建消息队列，调用msgsnd函数，把输入的字符串添加到消息队列中，
  然后调用msgrcv函数，读取消息队列中的消息并打印输出，
  最后再调用msgctl函数，删除系统内核中的消息队列。

```
if((qid = msgget(key,IPC_CREAT|0666)) == -1)/* 调用 msgget 函数,创建、打开消息队列 */
{
    perror("创建消息队列出错");
    exit(1);
}
```

```
if((msgsnd(qid,&msg,len,0))<0)  /* 调用 msgsnd 函数,添加消息到消息队列 */
{
   perror("添加消息出错");
   exit(1);
}
if((msgrcv(qid,&msg,512,0,0))<0)/* 调用 msgrcv 函数,从消息队列读取消息 */
{
   perror("读取消息出错");
   exit(1);
}
```

共享内存

共享内存（Shared Memory），指两个或多个进程共享一个给定的存储区。这一段存储区可以被两个或两个以上的进程映射至自身的地址空间中，一个进程写入共享内存的信息，可以被其他使用这个共享内存的进程，通过一个简单的内存读取做读出，从而实现了进程间的通信。

- 特点
  共享内存是**最快的一种 IPC**，因为进程是直接读写内存，而不需要任何数据的拷贝（对于像管道和消息队里等通信方式，则需要再内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次：一次从输入文件到共享内存区，另一次从共享内存到输出文件）。
  进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时在重新建立共享内存区域；而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件，因此，采用共享内存的通信方式效率非常高。

表 7.4　共享内存的常用函数

函数	功能
mmap	建立共享内存映射
munmap	解除共享内存映射
shmget	获取共享内存区域的 ID
shmat	建立映射共享内存
shmdt	解除共享内存映射

因为多个进程可以同时操作，所以需要进行同步。（信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问）

> 共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。

- 例子
  使用了【共享内存+信号量+消息队列】的组合来实现服务器进程与客户进程间的通信。
  - 共享内存用来传递数据；

    - 信号量用来同步；
    - 消息队列用来 在客户端修改了共享内存后 通知服务器读取。

- 共享内存有两种实现方式：1、内存映射 2、共享内存机制

内存映射

内存映射 memory map机制使进程之间通过映射同一个普通文件实现共享内存，通过mmap()系统调用实现。普通文件被映射到进程地址空间后，进程可以像访问普通内存一样对文件进行访问，不必再调用read/write等文件操作函数。

- 例子：
  创建子进程，父子进程通过匿名映射实现共享内存。
- 分析：
  主程序中先调用mmap映射内存，然后再调用fork函数创建进程。那么在调用fork函数之后，子进程继承父进程匿名映射后的地址空间，同样也继承mmap函数的返回地址，这样，父子进程就可以通过映射区域进行通信了。

```
char temp;
        p_map = (people * )mmap(NULL,sizeof(people) * 10,PROT_READ|PROT_WRITE,MAP_SHARED|
MAP_ANONYMOUS,-1,0);            /* 调用 mmap 函数,匿名内存映射 */
for(i = 0;i<5;i ++ )
   printf("子进程读取：第 %d 个人的年龄是：%d\n",i + 1,( * (p_map + i)).age);
( * p_map).age = 110;
munmap(p_map,sizeof(people) * 10); /* 解除内存映射关系 */
exit(0);
```

共享内存机制

IPC的共享内存指的是把所有的共享数据放在共享内存区域（IPC shared memory region），任何想要访问该数据的进程都必须在本进程的地址空间新增一块内存区域，用来映射存放共享数据的物理内存页面。
和前面的mmap系统调用通过映射一个普通文件实现共享内存不同，UNIX system V共享内存是通过**映射特殊文件系统shm**中的文件实现进程间的共享内存通信。

- 例子：
  设计两个程序，通过unix system v共享内存机制，一个程序写入共享区域，另一个程序读取共享区域。
- 分析：
  一个程序调用fotk函数产生标准的key，接着调用shmget函数，获取共享内存区域的id，调用shmat函数，映射内存，循环计算年龄，另一个程序读取共享内存。
  （fotk函数在消息队列部分已经用过了，根据pathname指定的文件（或目录）名称，以及proj参数指定的数字，ftok函数为IPC对象生成一个唯一性的键值。）

```
key = ftok(name,0);                 /* 调用 ftok 函数,产生标准的 key */
shm_id = shmget(key,4096,IPC_CREAT);/* 调用 shmget 函数,获取共享内存区域的 ID */
p_map = (people * )shmat(shm_id,NULL,0);/* 调用 shmat 函数,映射共享内存 */
```

```
for(i = 0;i<10;i ++ )
{
   temp + = 1;
   memcpy(( * (p_map + i)).name,&temp,1);
   ( * (p_map + i)).age = 20 + i;
}
if(shmdt(p_map) == -1)                    / * 调用 shmdt 函数,解除进程对共享内存区域的映
   char * name = "/dev/shm/mysha2" ;
   key = ftok(name,0); / * 调用 ftok 函数,产生标准的 key * /
  shm_id = shmget(key,4096,IPC_CREAT);/ * 调用 shmget 函数,获取共享内存区域的 ID * /
   if(shm_id == - 1)
   {
        perror("获取共享内存区域的 ID 出错");
        return;
   }
  p_map = (people * )shmat(shm_id,NULL,0);
   for(i = 0;i<10;i ++ )
   {
        printf( "姓名:%s\t",( * (p_map + i)).name );
        printf( "年龄:%d\n",( * (p_map + i)).age );
   }
  if(shmdt(p_map) == -1) / * 调用 shmdt 函数,解除进程对共享内存区域的映射 * /
   perror("解除映射出错");
}
```

套接字

信号

信号是Linux系统中用于进程之间通信或操作的一种机制，信号可以在任何时候发送给某一进程，而无须知道该进程的状态。
* 如果该进程并未处于执行状态，则该信号就由内核保存起来，知道该进程恢复执行并传递给他为止。
* 如果一个信号被进程设置为阻塞，则该信号的传递被延迟，直到其阻塞被取消时才被传递给进程。

信号是在软件层次上对中断机制的一种模拟，是一种异步通信方式，信号可以在用户空间进程和内核之间直接交互。内核也可以利用信号来通知用户空间的进程来通知用户空间发生了哪些系统事件。信号事件有两个来源：

- 硬件来源，例如按下了cltr+C，通常产生中断信号sigint
- 软件来源，例如使用系统调用或者命令发出信号。最常用的发送信号的系统函数是kill,raise,setitimer,sigation,sigqueue函数。软件来源还包括一些非法运算等操作。

一旦有信号产生，用户进程对信号产生的相应有三种方式：

1. 执行默认操作，linux对每种信号都规定了默认操作。
2. 捕捉信号，定义信号处理函数，当信号发生时，执行相应的处理函数。
3. 忽略信号，当不希望接收到的信号对进程的执行产生影响，而让进程继续执行时，可以忽略该信号，即不对信号进程作任何处理。

> 有两个信号是应用进程无法捕捉和忽略的，即SIGKILL和SEGSTOP，这是为了使系统管理

> 员能在任何时候中断或结束某一特定的进程。

信号发送函数

kill，raise，alarm，pause，signal等等

- 例子：
  创建子进程，为了使子进程不在父进程发出信号前结束，子进程中使用raise函数发送sigstop信号，使自己暂停；父进程使用信号操作的kill函数，向子进程发送sigkill信号，子进程收到此信号，结束子进程。

```
raise(SIGSTOP);      /* 调用 raise 函数,发送 SIGSTOP 使子进程暂停 */
            {
                printf("子进程的进程号(PID)是：%d\n",result);
                if((waitpid(result,NULL,WNOHANG)) == 0)
                {
                    if(ret = kill(result,SIGKILL) == 0)
                    /* 调用 kill 函数,发送 SIGKILL 信号结束子进程 result */
                    printf("用 kill 函数返回值是：%d,发出的 SIGKILL 信号结束的进程进程号：%
    d\n",ret,result);
                    else{ perror("kill 函数结束子进程失败");}
                }
```

信号处理

当某个信号被发送到一个正在运行的进程时，该进程即对次特定的信号注册相应的信号处理函数，以完成所需处理。设置信号处理方式的是signal函数，在程序正常结束前，在应用signal函数恢复系统对信号的默认处理方式。

```
(void) signal(SIGINT,fun_ctrl_c);/* 如果按了 Ctrl+C 键,调用 fun_ctrl_c 函数 */
```

信号阻塞

有时候既不希望进程在接收到信号时立刻中断进程的执行，也不希望此信号完全被忽略掉，而是希望延迟一段时间再去调用信号处理函数，这个时候就需要信号阻塞来完成。

信号量（semaphore）

信号量与已经介绍过的 IPC 结构不同，它是一个计数器。信号量用于**实现进程间的互斥与同步**，而不是用于存储进程间通信数据。
通过信号量来解决多个进程对同一资源的访问竞争的问题，使在任一时刻只能有一个执行线程访问代码的临界区域，也可以说它是协调进程间的对同一资源的访问权，也就是用于同步进程的。

> 线程同步也可以用信号量，是一个东西。但是进程同步主要用两个函数，线程同步使用另外四个函数。

- 特点
  信号量用于进程间互斥和同步，若要在进程间传递数据需要结合共享内存。
  信号量基于操作系统的 PV 操作，程序对信号量的操作都是**原子操作**。
  每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

支持信号量组。

> posix信号量一般用在线程上，System V信号量一般用在进程上。

总结

1. 管道：速度慢，容量有限，只有父子进程能通讯
2. 有名管道（FIFO）：任何进程间都能通讯，但速度慢
3. 消息队列：容量受到系统限制，且要注意第一次读的时候，要考虑上一次没有读完数据的问题
4. 信号量：不能传递复杂消息，只能用来同步
5. 共享内存区：能够很容易控制容量，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全，当然，共享内存区同样可以用作线程间通讯，不过没这个必要，线程间本来就已经共享了同一进程内的一块内存

原子操作

原子性，指的是一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程打断。

- 原子操作CAS(compare-and-swap)
  原子操作分三步：读取addr的值，和old进行比较，如果相等，则将new赋值给*addr，他能保证这三步一起执行完成，叫原子操作也就是说它不能再分了，当有一个CPU在访问这块内容addr时，其他CPU就不能访问

孤儿进程与僵尸进程

正常情况下，子进程是通过父进程创建的，子进程在创建新的进程。子进程的结束和父进程的运行是一个异步过程,即父进程永远无法预测子进程 到底什么时候结束。 当一个 进程完成它的工作终止之后，它的父进程需要调用wait()或者waitpid()系统调用取得子进程的终止状态。

- 孤儿进程：
  一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。
- 僵尸进程：
  一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵死进程。
  - 危害
    进程不调用wait / waitpid的话， 那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的，如果大量的产生僵死进程，将因为没有可用的进程号而导致系统不能产生新的进程.。

    > 孤儿进程则落到了init进程身上，内核就把孤儿进程的父进程设置为init，而init进程会循环地wait()它的已经退出的子进程。init进程就好像是一个民政局，专门负责处理孤儿进程的善后工作。因此孤儿进程并不会有什么危害。

wait和waitpid区别

- wait会令调用者阻塞直至某个子进程终止；
- waitpid则可以通过设置一个选项来设置为非阻塞，另外waitpid并不是等待第一个结束的进程而是等待参数中pid指定的进程。
  waitpid中pid的含义依据其具体值而变：
  - pid==-1 等待任何一个子进程，此时waitpid的作用与wait相同
  - pid >0 等待进程ID与pid值相同的子进程
  - pid==0 等待与调用者进程组ID相同的任意子进程
  - pid<-1 等待进程组ID与pid绝对值相等的任意子进程
    waitpid提供了wait所没有的三个特性：
  - 1、waitpid使我们可以等待指定的进程
  - 2、waitpid提供了一个无阻塞的wait
  - 3、waitpid支持工作控制

父进程fork后父子进程共享的内容

fork之后，子进程会拷贝父进程的数据空间、堆和栈空间（实际上是采用写时复制技术），二者共享代码段。共享fd，以及fd对应的文件表项。
在子进程中修改全局变量（局部变量，分配在堆上的内存同样也是）后，父进程的相同的全局变量不会改变。

- vfork函数
  vfork() 子进程与父进程共享数据段.
  vfork()保证子进程先运行，在她调用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。

linux的任务调度机制是什么

Linux 分实时进程和普通进程，实时进程应该先于普通进程而运行。

- 实时进程调度策略：
  - FIFO(先来先服务调度)
    不同的进程根据静态优先级进行排队，然后在同一优先级的队列中，谁先准备好运行就先调度谁，并且正在运行的进程不会被终止直到以下情况发生：
    （1）被有更高优先级的进程所强占CPU；
    （2）自己因为资源请求而阻塞；
    （3）自己主动放弃CPU（调用sched_yield）。
  - RR（时间片轮转调度）
    这种调度策略跟上面的FIFO一模一样，除了它给每个进程分配一个时间片，时间片到了正在执行的进程就放弃执行。

> 时间片的长度可以通过sched_rr_get_interval调用得到。

- 普通进程调度策略：
  - OTHER调度策略（比例共享的调度策略，保证进程调度时的公平性）
    调度器总是选择那个priority+counter值最大的进程来调度执行
- Linux根据policy的值将进程总体上分为实时进程和普通进程，提供了三种调度算法：一种传统的Unix调度程序和两个POSIX.1b操作系统的"实时"调度程序。
  - 非实时进程有两种优先级：
    静态优先级、动态优先级。（两个调度算法算法）
  - 实时进程有三种优先级
    静态优先级、动态优先级、实时优先级。优先级越高，得到CPU时间的机会也就越大。

死锁

- linux系统的各类同步机制
  - 锁机制：自旋锁、读写锁、顺序锁、读-拷贝-更新（RCU）
  - 互斥：原子操作、信号量、读写信号量
  - 等待队列：关闭中断
- 死锁
  是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。

产生死锁的四个必要条件

（1）互斥条件：某种资源一次只允许一个进程访问，即该资源一旦分配给某个进程，其他进程就不能再访问，直到该进程访问结束。
（2）占有且等待：一个进程本身占有资源（一种或多种），同时还有资源未得到满足，正在等待其他进程释放该资源。
（3）不可强占:别人已经占有了某项资源，你不能因为自己也需要该资源，就去把别人的资源抢过来。
（4）循环等待条件:存在一个进程链，使得每个进程都占有下一个进程所需的至少一种资源。
当以上四个条件均满足，必然会造成死锁，发生死锁的进程无法进行下去，它们所持有的资源也无法释放。这样会导致CPU的吞吐量下降。所以死锁情况是会浪费系统资源和影响计算机的使用性能的。

处理死锁的基本方法：

- 死锁预防：确保系统永远不会进入死锁状态
  通过设置某些限制条件，去破坏死锁的四个条件中的一个或几个条件，来预防发生死锁。但由于所施加的限制条件往往太严格，因而导致系统资源利用率和系统吞吐量降低。
- 死锁避免：在使用前进行判断，只允许不会产生死锁的进程申请资源
  允许前三个必要条件，但通过明智的选择，确保永远不会到达死锁点，因此死锁避免比死锁预防

允许更多的并发。

- 死锁检测：在检测到运行系统进入死锁，进行恢复
  不须实现采取任何限制性措施，而是允许系统在运行过程发生死锁，但可通过系统设置的检测机构及时检测出死锁的发生，并精确地确定于死锁相关的进程和资源，然后采取适当的措施，从系统中将已发生的死锁清除掉。
- 死锁解除：
  与死锁检测相配套的一种措施。当检测到系统中已发生死锁，需将进程从死锁状态中解脱出来。常用方法：撤销或挂起一些进程，以便回收一些资源，再将这些资源分配给已处于阻塞状态的进程。死锁检测盒解除有可能使系统获得较好的资源利用率和吞吐量，但在实现上难度也最大。

如何实现守护进程

1）创建子进程，父进程退出
2）在子进程中创建新会话
3）改变当前目录为根目
4）重设文件权限掩码
5) 关闭文件描述符
6) 守护进程退出处理（ending）
当用户需要外部停止守护进程运行时，往往会使用 kill命令停止该守护进程。所以，守护进程中需要编码来实现kill发出的signal信号处理，达到进程的正常退出。

linux的五种IO模式/异步模式.

1）同步阻塞I/O
2）同步非阻塞I/O
3）同步I/O复用模型
4） 同步信号驱动I/O
5） 异步I/O模型

阻塞、非阻塞、同步、异步、Reactor、Proactor

- 阻塞：阻塞是指当前线程被堵住，不能继续往下执行，被操作系统挂起。
  阻塞的对象是当前线程，而不是IO被阻塞了；外部资源(通常是IO)使得当前线程被挂起才叫阻塞。阻塞的核心表现是等待，不能做其他的事。

  > 如果内核缓冲没有数据可读时，read()系统调用会一直等待有数据到来后才从阻塞态中返回，这就是阻塞I/O。

- 非阻塞：调用外部资源，不管结果如何，不会阻挡线程的继续执行。

  > 非阻塞I/O在遇到内核缓冲没有数据可读时会立即返回给用户态进程一个返回值，并设置errno为EAGAIN

- 同步：“调用”然后得到“结果”。可能立即得到结果，可能等待一毫秒，也可能等待一辈子，反正结果必须紧随其后。

同步的概念关键在于**结果**必须**紧随调用者**之后到来，同步跟是否阻塞没有必然关系。

> 同步I/O指处理I/O操作的进程和处理I/O操作的进程是同一个。

- 异步：“调用”，有“结果”再通知我。异步跟同步的区别是，结果返回时间点跟调用发起时间点没有强制关系，调用者是被动得到结果的，不是主动等待结果的。
  > 异步I/O中I/O操作由操作系统完成，并不由产生I/O的用户进程执行。
- 半同步半异步模式：
  上层的任务（如：数据库查询，文件传输）使用同步I/O模型，简化了编写并行程序的难度。
  而底层的任务（如网络控制器的中断处理）使用异步I/O模型，提供了执行效率。
- Reactor：同步阻塞I/O模式。响应模式，被动的，受外部驱动。
  处理I/O操作的依旧是产生I/O的程序
  基于Reactor模式的网络IO:
  首先声明要监听哪些事件;Reactor的核心是被动响应，程序响应IO事件，具体的收发操作还是需要程序自己去完成，收发操作是同步的。
- Proactor：异步I/O模式。主动模式，不受外部驱动。
  产生I/O调用的用户进程不会等待I/O发生，具体I/O操作由操作系统完成。（异步I/O需要操作系统支持，Linux异步I/O为AIO，Windows为IOCP。
  ）
  基于Proactor模式的网络IO：

  > 不用关心IO是否繁忙，发就是了，成功了会通知我的，通过sendSuccessEvent事件
  > 不用关心有没有数据，我想何时接收就接收，如果我不执行接收，就永远不会触发

- 例子说明
  Reactor和Proactor模式的主要区别就是真正的读取和写入操作是有谁来完成的，Reactor中需要应用程序自己读取或者写入数据，Proactor模式中，应用程序不需要进行实际读写过程。
  - Reactor是：
    主线程往epoll内核上注册socket读事件，主线程调用epoll_wait等待socket上有数据可读，当socket上有数据可读的时候，主线程把socket可读事件放入请求队列。睡眠在请求队列上的某个工作线程被唤醒，处理客户请求，然后往epoll内核上注册socket写请求事件。主线程调用epoll_wait等待写请求事件，当有事件可写的时候，主线程把socket可写事件放入请求队列。睡眠在请求队列上的工作线程被唤醒，处理客户请求。
  - Proactor:
    主线程调用aio_read函数向内核注册socket上的读完成事件，并告诉内核用户读缓冲区的位置，以及读完成后如何通知应用程序，主线程继续处理其他逻辑，当socket上的数据被读入用户缓冲区后，通过信号告知应用程序数据已经可以使用。应用程序预先定义好的信号处理函数选择一个工作线程来处理客户请求。工作线程处理完客户请求之后调用aio_write函数向内核注册socket写完成事件，并告诉内核写缓冲区的位置，以及写完成时如何通知应用程序。主线程处理其他逻辑。当用户缓存区的数据被写入socket之后内核向应用程序发送一个信号，以通知应用程序数据已经发送完毕。应用程序预先定义的数据处理函数就会完成工作。

一些小题

- netstat、tcpdump、ipcs、ipcrm命令
  netstat:检查网络状态，tcpdump:截获数据包，ipcs:检查共享内存，ipcrm:解除共享内存
- 共享内存段被映射进进程空间之后，存在于进程空间的什么位置？共享内存段最大限制是多少？
  将一块内存映射到两个或者多个进程地址空间。通过指针访问该共享内存区。一般通过mmap将文件映射到进程地址共享区；
  存在于进程数据段；
  最大限制是0x2000000Byte。
- 进程内存空间分布情况
  程序段(Text)、初始化过的数据(Data)、未初始化过的数据(BSS)、栈 (Stack)、堆 (Heap)。
- 大端、小端
  对于一个由2个字节组成的16位整数，在内存中存储这两个字节有两种方法：
  一种是将低序字节存储在起始地址，这称为小端(little-endian)字节序；
  另一种方法是将高序字节存储在起始地址，这称为大端(big-endian)字节序。

```
union{
short value;
char a[sizeof(short)]; //两个字节
}test;
test.value= 0x0102;
if((test.a[0] == 1) && (test.a[1] == 2))
    cout << "big"<<endl;
else
    cout << "little" << endl;
```

- 信号怎么应答
  - 显示地忽略信号
  - 执行与信号相关的缺省操作（内核预定义的缺省操作取决于信号的类型）
  - 通过调用相应的信号处理函数捕获信号
- 系统如何将一个信号通知到进程
  内核给进程发送信号，是在进程所在的进程表项的信号域设置对应的信号的位。进程处理信号的时机就是从内核态即将返回用户态的时候。
  - 执行用户自定义的信号处理函数的方法：
    把信号处理函数的地址放在用户栈栈顶，进程从内核返回到用户态时，先弹出信号处理函数地址，于是就执行信号处理函数，运行完会返回一个错误码，指出该次系统调用曾经被打断。

标准库函数和系统调用的区别

- 系统调用：

是操作系统为用户态运行的进程和硬件设备(如CPU、磁盘、打印机等)进行交互提供的一组接口，即就是设置在应用程序和硬件设备之间的一个接口层。
linux内核是单内核，结构紧凑，执行速度快，各个模块之间是直接调用的关系。linux系统上到下依次是用户进程->linux内核->硬件。其中系统调用接口是位于Linux内核中的，整个linux系统从上到下可以是：用户进程->系统调用接口->linux内核子系统->硬件，也就是说Linux内核包括了系统调用接口和内核子系统两部分；或者从下到上可以是：物理硬件->OS内核->OS服务->应用程序，操作系统起到“承上启下”作用，向下管理物理硬件，向上为操作系服务和应用程序提供接口，这里的接口就是系统调用了。

- 库函数：
  把函数放到库里。是把一些常用到的函数编完放到一个lib文件里，供别人用。别人用的时候把它所在的文件名用#include<>加到里面就可以了。一类是c语言标准规定的库函数，一类是编译器特定的库函数。

系统调用是为了方便使用操作系统的接口，而库函数则是为了人们编程的方便。

标准io和文件io的区别

- 定义
  - 标准ＩＯ：具有一定的可移植性。标准IO库处理很多细节。例如缓存分配，以优化长度执行IO等。标准的IO提供了三种类型的缓存。
    （1）全缓存：当填满标准IO缓存后才进行实际的IO操作。
    （2）行缓存：当输入或输出中遇到新行符时，标准IO库执行IO操作。
    （3）不带缓存：stderr就是了。
  - 文件ＩＯ：文件ＩＯ称之为不带缓存的IO（unbuffered I/O)。不带缓存指的是每个read，write都调用内核中的一个系统调用。也就是一般所说的低级I/O——操作系统提供的基本IO服务，与os绑定，特定于Unix平台。
- 区别
  - 文件I/O 又称为低级磁盘I/O，遵循POSIX相关标准。任何兼容POSIX标准的操作系统上都支持文件I/O。
    标准I/O被称为高级磁盘I/O，遵循ANSI C相关标准。只要开发环境中有标准I/O库，标准I/O就可以使用。（Linux 中使用的是GLIBC，它是标准C库的超集。不仅包含ANSI C中定义的函数，还包括POSIX标准中定义的函数。因此，Linux 下既可以使用标准I/O，也可以使用文件I/O）。
  - 通过文件I/O读写文件时，每次操作都会执行相关系统调用。这样处理的好处是直接读写实际文件，坏处是频繁的系统调用会增加系统开销。
    标准I/O可以看成是在文件I/O的基础上封装了缓冲机制。先读写缓冲区，必要时再访问实际文件，从而减少了系统调用的次数。
  - 文件I/O中用文件描述符表现一个打开的文件，可以访问不同类型的文件如普通文件、设备文件和管道文件等。
    而标准I/O中用FILE（流）表示一个打开的文件，通常只用来访问普通文件。

fopen与open的区别

标准I/O 对应 fopen
文件I/O 对应 open

- 缓存文件系统
  缓冲文件系统的特点是：在内存开辟一个“缓冲区”，为程序中的每一个文件使用。内存 “缓冲区”的大小，影响着实际操作外存的次数。
  fopen, fclose, fread, fwrite, fgetc, fgets, fputc, fputs, freopen, fseek, ftell, rewind等
- 非缓冲文件系统
  缓冲文件系统是借助文件结构体指针来对文件进行管理，通过文件指针来对文件进行访问，既可以读写字符、字符串、格式化数据，也可以读写二进制数。
  非缓冲文件系统依赖于操作系统，通过操作系统的功能对文件进行读写，是系统级的输入输出，它不设文件结构体指针，只能读写二进制文件，但效率高、速度快。
  open, close, read, write, getc, getchar, putc, putchar

open 是系统调用 返回的是文件句柄，文件的句柄是文件在文件描述副表里的索引；fopen是C的库函数，返回的是一个指向文件结构的指针。

- 区别

open	fopen
低级IO	高级IO
返回一个文件描述符(用户程序区的)	返回一个文件指针
无缓冲	有缓冲
与 read, write 等配合使用	与 fread, fwrite等配合使用
	在open的基础上扩充而来的，在大多数情况下用fopen

硬中断和软中断

- 硬中断
  硬中断是由硬件产生，比如磁盘，网卡，键盘，时钟。
  处理中断的驱动是需要运行在CPU上的，因此，当中断产生的时候，CPU会中断当前正在运行的任务，来处理中断。
- 软中断
  由当前正在运行的进程所产生，通常是对I/O的请求，会调用内核中可以调度I/O发生的程序。软中断仅与内核相联系，内核负责对进程进行调度。软中断并不会直接中断CPU。
  为了满足实时系统的要求，中断处理应该是越快越好。linux为了实现这个特点，当中断发生的时候，硬中断处理那些短时间就可以完成的工作，而将那些处理事件比较长的工作，放到中断之后

来完成，也就是软中断(softirq)来完成。

- 区别

硬中断	软中断
执行中断指令产生	由外设引发
中断号是由中断控制器提供	由指令直接指出
可屏蔽的	不可屏蔽
确保快速地完成任务	处理硬中断未完成的工作

linux程序启动过程

1、当你在 shell 中敲入一个命令要执行时，内核会帮我们创建一个新的进程，它在往这个新进程的进程空间里面加载进可执行程序的代码段和数据段后，也会加载进动态连接器(在Linux里面通常就是/lib/ld-linux.so 符号链接所指向的那个程序，它本省就是一个动态库)的代码段和数据。
2、在这之后，内核将控制传递给动态链接库里面的代码。动态连接器接下来负责加载该命令应用程序所需要使用的各种动态库。加载完毕，动态连接器才将控制传递给应用程序的main函数。
3、如此，你的应用程序才得以运行。(过程链接表（PLT）, Global Offset Table（GOT）)

Linux开机流程

1. 加载BIOS的硬件信息与进行自我测试，并依据设置取得第一个可启动设备；
2. 读取并执行第一个启动设备内MBR（主引导分区）的Boot Loader（即是gurb等程序）；
3. 依据Boot Loader的设置加载Kernel，Kernel会开始检测硬件与加载驱动程序；
4. 在硬件驱动成功后，Kernel会主动调用init进程（/sbin/init），而init会取得runlevel信息；
5. init执行/etc/rc.d/rc.sysinit文件来准备软件的操作环境（如网络、时区等）；
6. init执行runlevel的各个服务的启动（script方式）；
7. init执行/etc/rc.d/rc.local文件；
8. init执行终端机模拟程序mingetty来启动login程序，最后等待用户登录。