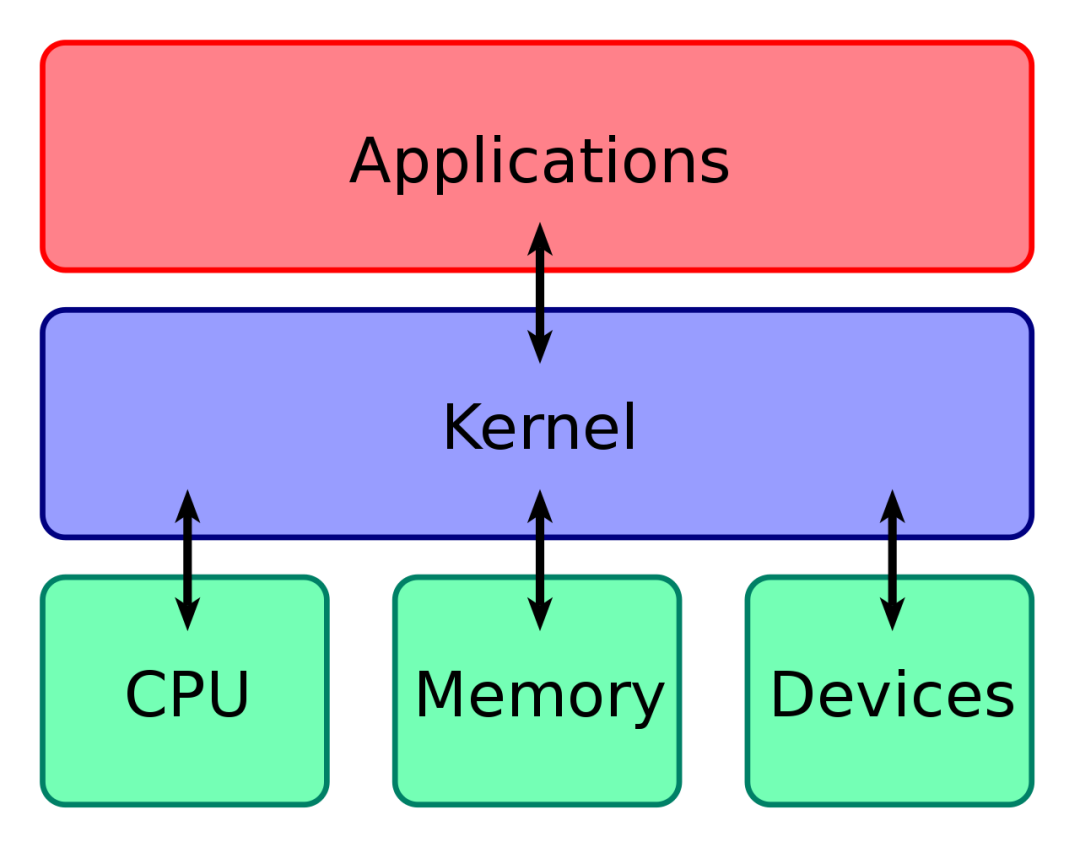
# 内核

## 什么是内核呢？

计算机是由各种外部硬件设备组成的，比如内存、cpu、硬盘等，如果每个应用都要和这些硬件设备对接通信协议，那这样太累了。

所以，这个中间人就由内核来负责，让内核作为应用连接硬件设备的桥梁，应用程序只需关心与内核交互，不用关心硬件的细节。



## 内核有哪些能力呢？

现代操作系统，内核一般会提供 4 个基本能力：

* 管理进程、线程，决定哪个进程、线程使用 CPU，也就是进程调度的能力；
* 管理内存，决定内存的分配和回收，也就是内存管理的能力；
* 管理硬件设备，为进程与硬件设备之间提供通信能力，也就是硬件通信能力；
* 提供系统调用，如果应用程序要运行更高权限运行的服务，那么就需要有系统调用，它是用户程序与操作系统之间的接口。

## 内核是怎么工作的？

内核具有很高的权限，可以控制 cpu、内存、硬盘等硬件，而应用程序具有的权限很小，因此大多数操作系统，把内存分成了两个区域：

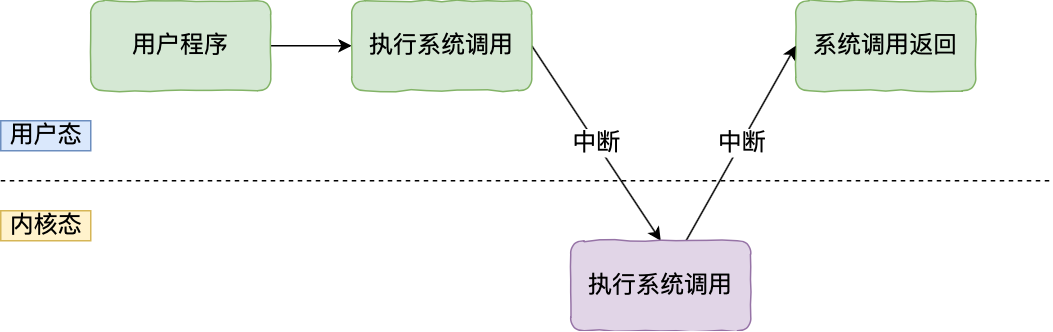
内核空间，这个内存空间只有内核程序可以访问；

用户空间，这个内存空间专门给应用程序使用；

用户空间的代码只能访问一个局部的内存空间，而内核空间的代码可以访问所有内存空间。

因此，当程序使用用户空间时，我们常说该程序在用户态执行，而当程序使内核空间时，程序则在内核态执行。

应用程序如果需要进入内核空间，就需要通过「系统调用」，下面来看看系统调用的过程：



内核程序执行在内核态，用户程序执行在用户态。当应用程序使用系统调用时，会产生一个中断。发生中断后， CPU 会中断当前在执行的用户程序，转而跳转到中断处理程序，也就是开始执行内核程序。内核处理完后，主动触发中断，把 CPU 执行权限交回给用户程序，回到用户态继续工作。

## Linux

### Linux 内核设计的理念

* MutiTask，多任务
* SMP，对称多处理
* ELF，可执行文件链接格式
* Monolithic Kernel，宏内核

### MutiTask多任务

MutiTask 的意思是多任务，代表着 Linux 是一个多任务的操作系统。

多任务意味着可以有多个任务同时执行，这里的「同时」可以是并发或并行：

对于单核 CPU 时，可以让每个任务执行一小段时间，时间到就切换另外一个任务，从宏观角度看，一段时间内执行了多个任务，这被称为并发。

对于多核 CPU 时，多个任务可以同时被不同核心的 CPU 同时执行，这被称为并行。

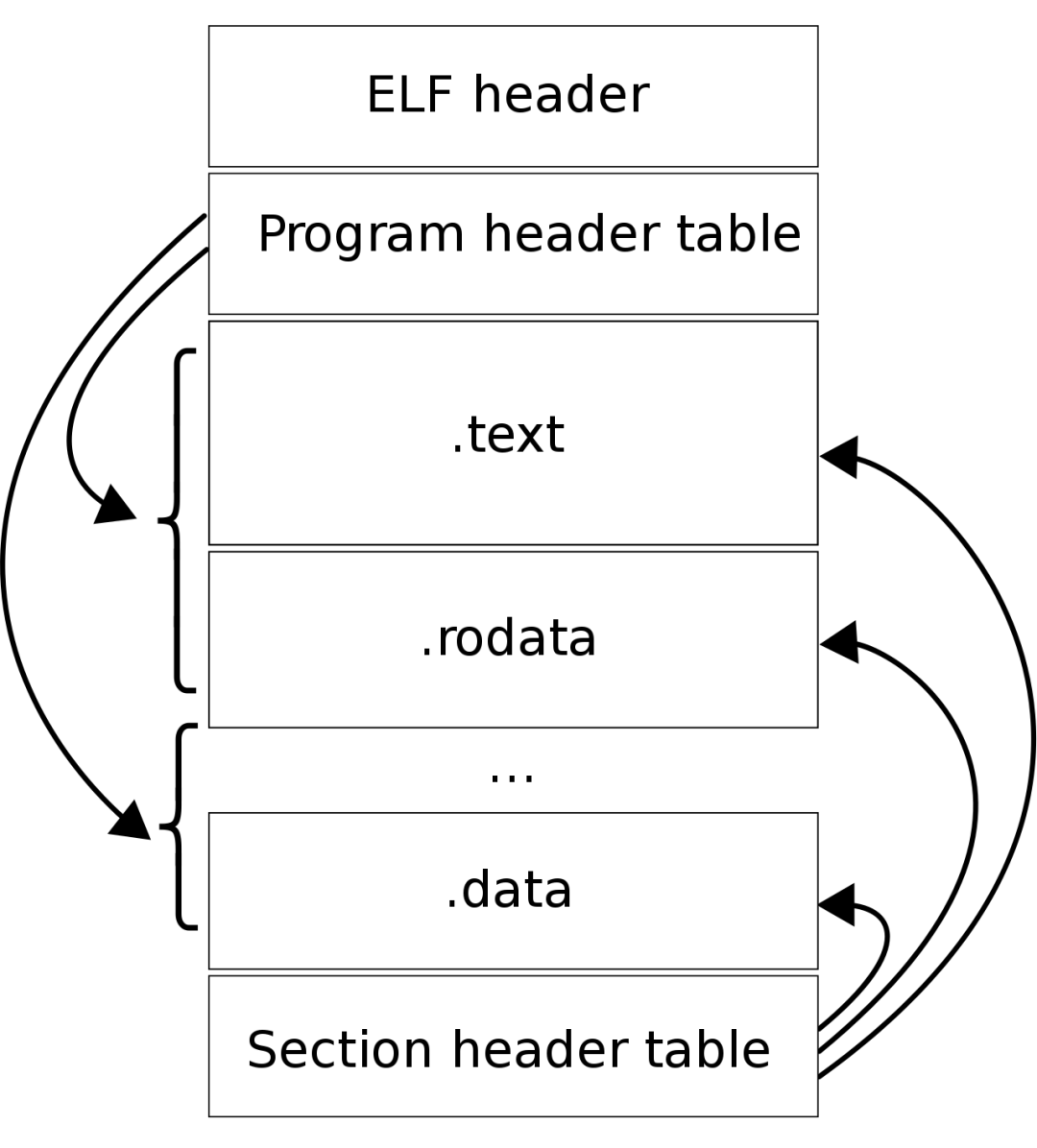
### SMP对称多处理

SMP 的意思是对称多处理，代表着每个 CPU 的地位是相等的，对资源的使用权限也是相同的，多个 CPU 共享同一个内存，每个 CPU 都可以访问完整的内存和硬件资源。

这个特点决定了 Linux 操作系统不会有某个 CPU 单独服务应用程序或内核程序，而是每个程序都可以被分配到任意一个 CPU 上被执行。

### ELF

ELF 的意思是可执行文件链接格式，它是 Linux 操作系统中可执行文件的存储格式，你可以从下图看到它的结构：



ELF 把文件分成了一个个分段，每一个段都有自己的作用，具体每个段的作用这里我就不详细说明了，感兴趣的同学可以去看《程序员的自我修养——链接、装载和库》这本书。

另外，ELF 文件有两种索引，Program header table 中记录了「运行时」所需的段，而 Section header table 记录了二进制文件中各个「段的首地址」。

那 ELF 文件怎么生成的呢？

我们编写的代码，首先通过「编译器」编译成汇编代码，接着通过「汇编器」变成目标代码，也就是目标文件，最后通过「链接器」把多个目标文件以及调用的各种函数库链接起来，形成一个可执行文件，也就是 ELF 文件。

那 ELF 文件是怎么被执行的呢？

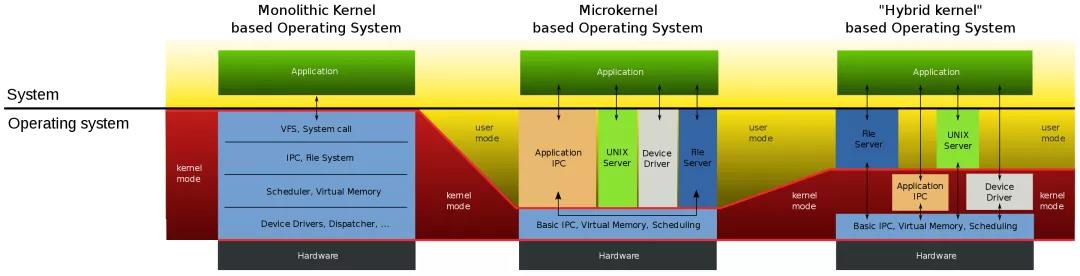
执行 ELF 文件的时候，会通过「装载器」把 ELF 文件装载到内存里，CPU 读取内存中的指令和数据，于是程序就被执行起来了。

### Monolithic Kernel宏内核

Monolithic Kernel 的意思是宏内核，Linux 内核架构就是宏内核，意味着 Linux 的内核是一个完整的可执行程序，且拥有最高的权限。

宏内核的特征是系统内核的所有模块，比如进程调度、内存管理、文件系统、设备驱动等，都运行在内核态。

不过，Linux 也实现了动态加载内核模块的功能，例如大部分设备驱动是以可加载模块的形式存在的，与内核其他模块解藕，让驱动开发和驱动加载更为方便、灵活。



与宏内核相反的是微内核，微内核架构的内核只保留最基本的能力，比如进程调度、虚拟机内存、中断等，把一些应用放到了用户空间，比如驱动程序、文件系统等。这样服务与服务之间是隔离的，单个服务出现故障或者完全攻击，也不会导致整个操作系统挂掉，提高了操作系统的稳定性和可靠性。

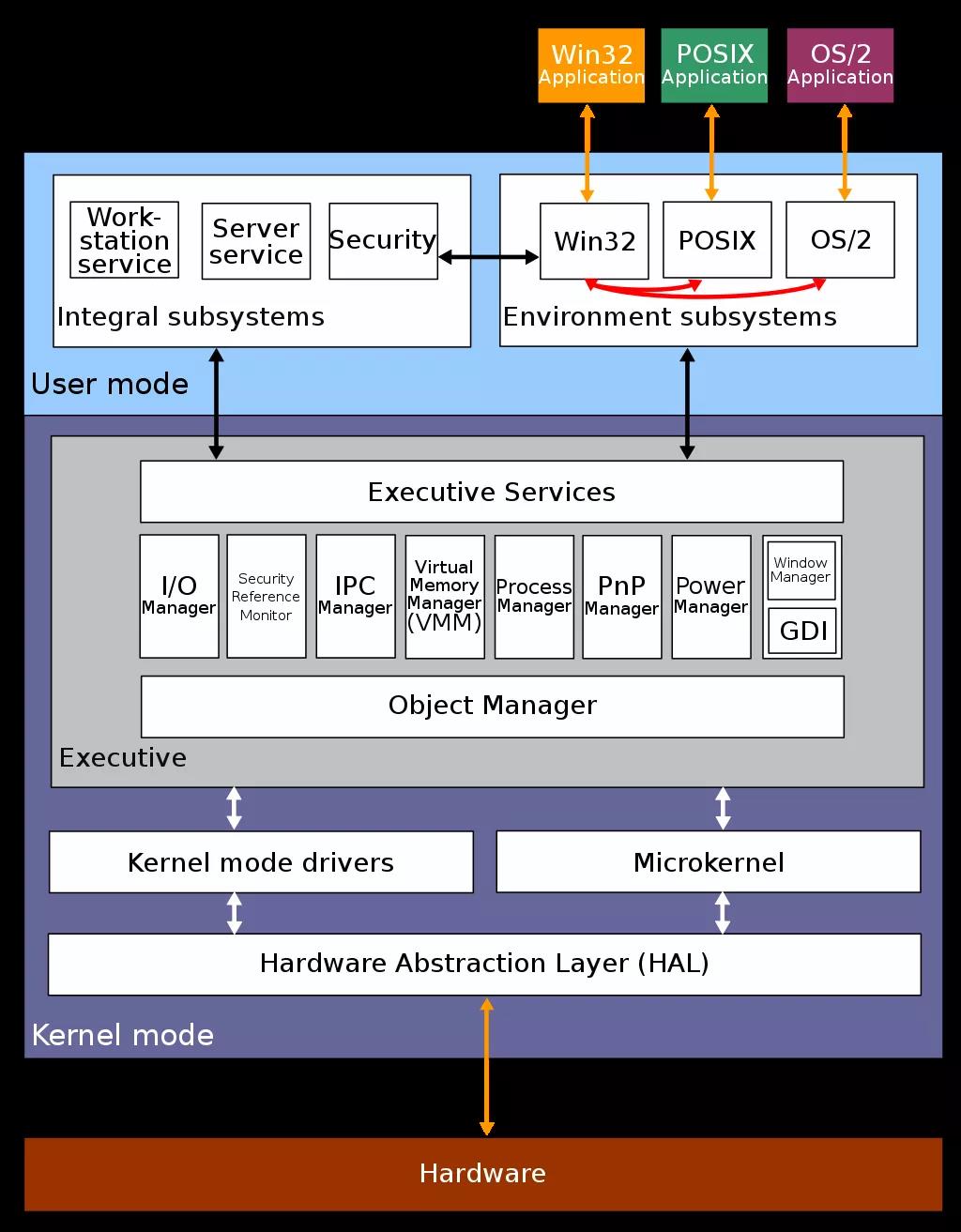
微内核内核功能少，可移植性高，相比宏内核有一点不好的地方在于，由于驱动程序不在内核中，而且驱动程序一般会频繁调用底层能力的，于是驱动和硬件设备交互就需要频繁切换到内核态，这样会带来性能损耗。华为的鸿蒙操作系统的内核架构就是微内核。

还有一种内核叫混合类型内核，它的架构有点像微内核，内核里面会有一个最小版本的内核，然后其他模块会在这个基础上搭建，然后实现的时候会跟宏内核类似，也就是把整个内核做成一个完整的程序，大部分服务都在内核中，这就像是宏内核的方式包裹着一个微内核。

## Windows设计

当今 Windows 7、Windows 10 使用的内核叫 Windows NT，NT 全称叫 New Technology。

下图是 Windows NT 的结构图片：

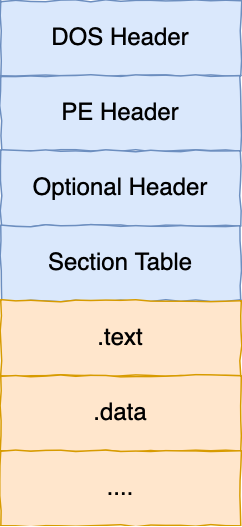


Windows 和 Linux 一样，同样支持 MutiTask 和 SMP，但不同的是，Windows 的内核设计是混合型内核，在上图你可以看到内核中有一个 MicroKernel 模块，这个就是最小版本的内核，而整个内核实现是一个完整的程序，含有非常多模块。

Windows 的可执行文件的格式与 Linux 也不同，所以这两个系统的可执行文件是不可以在对方上运行的。

Windows 的可执行文件格式叫 PE，称为可移植执行文件，扩展名通常是.exe、.dll、.sys等。

PE 的结构你可以从下图中看到，它与 ELF 结构有一点相似。



## 总结

对于内核的架构一般有这三种类型：

### 宏内核

包含多个模块，整个内核像一个完整的程序；

### 微内核

有一个最小版本的内核，一些模块和服务则由用户态管理；

### 混合内核

是宏内核和微内核的结合体，内核中抽象出了微内核的概念，也就是内核中会有一个小型的内核，其他模块就在这个基础上搭建，整个内核是个完整的程序；

Linux 的内核设计是采用了宏内核，Windows 的内核设计则是采用了混合内核。

这两个操作系统的可执行文件格式也不一样， Linux 可执行文件格式叫作 ELF，Windows 可执行文件格式叫作 PE。

# Linux头文件

异步IO aio.h

目录项 dirent.h

文件控制 fcntl.h

文件名匹配类型 fnmatch.h

路径名模式匹配与生成 glob.h

组文件 grp.h

代码集变换实用程序 iconv.h

网络数据库操作 netdb.h

消息类 nl\_types.h

poll函数 poll.h

线程 pthread.h

信号量 semaphore.h

字符串操作 string.h

符号常量 unistd.h

因特网定义 arpa/inet.h

套接字本地接口 net/if.h

因特网地址族 netinet/in.h

传输控制协议定义 betubet/tcp.h

存储管理声明 sys/mman.h

select函数 sys/select.h

套接字接口 sys/socket.h

文件状态 sys/stat.h

文件系统信息 sys/statvfs.h

进程时间 sys/time.h

基本系统数据类型 sys/types.h

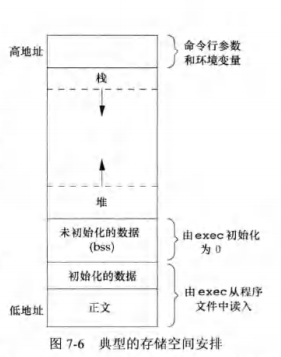
UNIX域套接字定义 sys/un.h

系统名 sys/untname.h

进程控制 sys/wait.h

# 内存管理

## C程序存储空间

 用户区（0-3G）

stack段：用户栈区，由系统自动分配，开口向下

env段：存储全局环境变量

.so段：存储共享库

.heap段：用户堆区，进行动态内存分配，开口向上

.bss段：未初始化的全局变量

.rodata：常量区。

.data段：已经初始化的全局变量，属于静态内存分配。static声明的变量也存储在这里。

text段：存储代码，

### 堆和栈的区别

#### 申请方式不同

栈由系统自动分配；堆是自己申请和释放的。

#### 申请大小限制不同

栈顶和栈底是之前预设好的，栈是向栈底扩展，大小固定，可以通过ulimit -a查看，由ulimit -s修改；堆向高地址扩展，是不连续的内存区域，大小可以灵活调整。

#### 申请效率不同

栈由系统分配，速度快，不会有碎片；堆由程序员分配，速度慢，且会有碎片。

### malloc与free的实现原理？

1、 在标准C库中，提供了malloc/free函数分配释放内存，这两个函数底层是由brk、mmap、，munmap这些系统调用实现的;

2、 brk是将数据段(.data)的最高地址指针\_edata往高地址推,mmap是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）找一块空闲的虚拟内存。这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系；

3、 malloc小于128k的内存，使用brk分配内存，将\_edata往高地址推；malloc大于128k的内存，使用mmap分配内存，在堆和栈之间找一块空闲内存分配；brk分配的内存需要等到高地址内存释放以后才能释放，而mmap分配的内存可以单独释放。当最高地址空间的空闲内存超过128K（可由M\_TRIM\_THRESHOLD选项调节）时，执行内存紧缩操作（trim）。在上一个步骤free的时候，发现最高地址空闲内存超过128K，于是内存紧缩。

4、 malloc是从堆里面申请内存，也就是说函数返回的指针是指向堆里面的一块内存。操作系统中有一个记录空闲内存地址的链表。当操作系统收到程序的申请时，就会遍历该链表，然后就寻找第一个空间大于所申请空间的堆结点，然后就将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。

## 地址空间

地址空间是进程可以用来寻址内存的地址集。每个进程都有它自己的地址空间，独立于其他进程的地址空间。但是某些进程会希望可以共享地址空间（共享内存）。

### 早年直接用物理地址空间

使用动态重定位技术，将每个进程的地址空间映射到物理内存的不同区域。

基址寄存器存储内存的起始位置，变址寄存器存储应用程序的长度。访问的内存地址 = 基址寄存器 + offset，且必须不大于变址寄存器。

### 虚拟地址空间

虚拟内存技术能够允许应用程序部分运行在内存中。许多程序的片段同时保存在内存中，当一个程序等待它的一部分呢读入内存时，可以把CPU交给另一个进程使用。磁盘上有一个最多64k的程序核心映像的完整副本，以保证程序片段在需要时调入内存。

每个进程都有自己的地址空间，这个虚拟地址空间被划分为多个虚拟页面。这些虚拟页面被映射到物理内存块。

当程序引用到一部分在物理内存中的地址空间时，MMU把虚拟地址映射为物理地址。

当程序引用到一部分不在物理内存中的地址空间时，CPU引发缺页中断。OS会选择一个很少使用的页，并把它的内容写入磁盘（交换）。随后把需要访问的页面读到刚才回收的页框中，修改MMU映射关系，然后重新执行失败的指令。

#### 页表项

页框号：将虚拟页号映射为物理块（页框）。

在/不在位：该物理块在不在物理内存。

保护位：表示读写权限，0读写，1只读。

修改位：表示一个页面是否被修改过，用于重新分配页框。1则必须把它写回磁盘，0则在重新分配这个页框时直接丢弃。

访问位：在页面被访问时设置，用于“最近最久未使用”算法。分为访问位R，修改位M。

## 交换

把一个进程完整地调入内存，然后在内存中运行一段时间，再把它放回磁盘。

空闲进程会存储在磁盘中，所以这些进程在没有运行时不会占用太多内存。

### 空闲内存管理

* 位图：0表示空闲，1表示占用
* 链表：维护一个记录已分配内存段和空闲内存段的链表。

## TLB转换检测缓冲区

大多数程序总是对少量页面进行多次访问。因此，只有很少的页面能够被再次访问，而其他的页表项很少被访问。

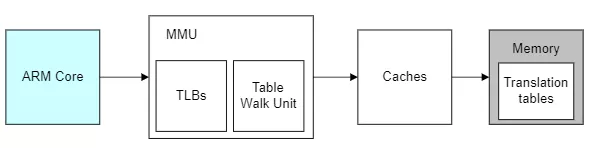
基于这种设想，在MMU内设置了一个小型的硬件设备，能够将虚拟地址直接映射到物理地址，而不必再访问页表。TLB实际上就是一种内存缓存，将虚拟地址到物理地址的转换存储起来，称为地址翻译缓存。

如果虚拟页号在TLB中，则将页框号直接从TLB取出而不必再访问页表。

如果虚拟页号不在TLB中，从页表找，然后从TLB逐出一个页表项，然后把从页表中找到的项放到TLB。

## Linux内存管理

### CPU访问内存的过程



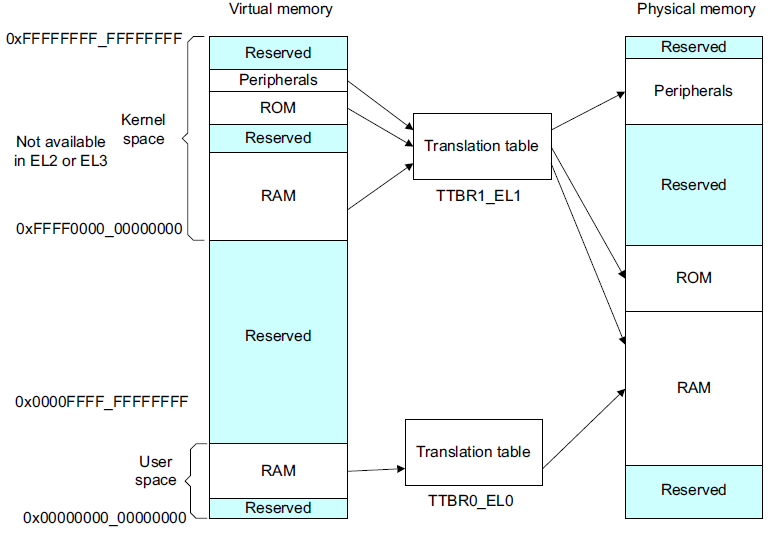
蓝色部分是cpu，灰色部分是内存，白色部分就是cpu访问内存的过程，也是地址转换的过程。

TLB：MMU工作的过程就是查询页表的过程。如果把页表放在内存中查询的时候开销太大，因此为了提高查找效率，专门用一小片访问更快的区域存放地址转换条目。（当页表内容有变化的时候，需要清除TLB，以防止地址映射出错。）

Caches：cpu和内存之间的缓存机制，用于提高访问速率，armv8架构的话上图的caches其实是L2 Cache，这里就不做进一步解释了。

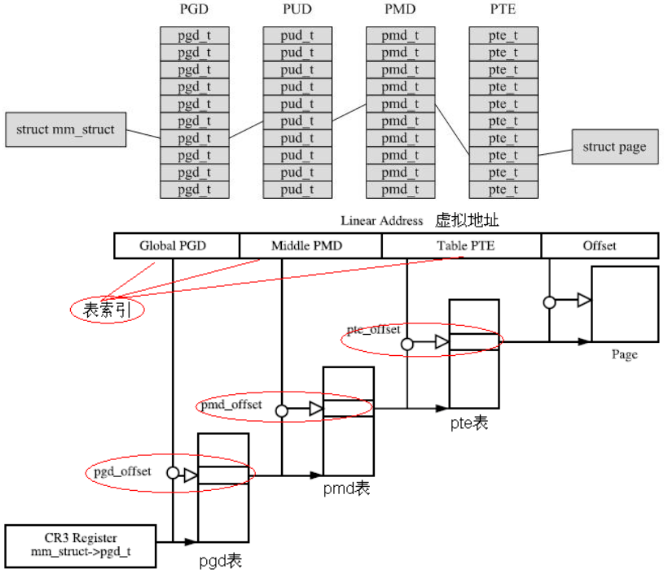
### **虚拟地址转换为物理地址的本质**

我们知道内核中的寻址空间大小是由CONFIG\_ARM64\_VA\_BITS控制的，这里以48位为例，ARMv8中，Kernel Space的页表基地址存放在TTBR1\_EL1寄存器中，User Space页表基地址存放在TTBR0\_EL0寄存器中，其中内核地址空间的高位为全1，(0xFFFF0000\_00000000 ~ 0xFFFFFFFF\_FFFFFFFF)，用户地址空间的高位为全0，(0x00000000\_00000000 ~ 0x0000FFFF\_FFFFFFFF)



有了宏观概念，下面我们以内核态寻址过程为例看下是如何把虚拟地址转换为物理地址的。

我们知道linux采用了分页机制，通常采用四级页表，页全局目录(PGD)，页上级目录(PUD)，页中间目录(PMD)，页表(PTE)。如下：



1. 从CR3寄存器中读取页目录所在物理页面的基址(即所谓的页目录基址)，从线性地址的第一部分获取页目录项的索引，两者相加得到页目录项的物理地址。
2. 第一次读取内存得到pgd\_t结构的目录项，从中取出物理页基址取出，即页上级页目录的物理基地址。
3. 从线性地址的第二部分中取出页上级目录项的索引，与页上级目录基地址相加得到页上级目录项的物理地址。
4. 第二次读取内存得到pud\_t结构的目录项，从中取出页中间目录的物理基地址。
5. 从线性地址的第三部分中取出页中间目录项的索引，与页中间目录基址相加得到页中间目录项的物理地址。
6. 第三次读取内存得到pmd\_t结构的目录项，从中取出页表的物理基地址。
7. 从线性地址的第四部分中取出页表项的索引，与页表基址相加得到页表项的物理地址。
8. 第四次读取内存得到pte\_t结构的目录项，从中取出物理页的基地址。
9. 从线性地址的第五部分中取出物理页内偏移量，与物理页基址相加得到最终的物理地址。
10. 第五次读取内存得到最终要访问的数据。

整个过程是比较机械的，每次转换先获取物理页基地址，再从线性地址中获取索引，合成物理地址后再访问内存。不管是页表还是要访问的数据都是以页为单位存放在主存中的，因此每次访问内存时都要先获得基址，再通过索引(或偏移)在页内访问数据，因此可以将线性地址看作是若干个索引的集合。

### Linux内存初始化

#### 创建启动页表

在汇编代码阶段的head.S文件中，负责创建映射关系的函数是create\_page\_tables。create\_page\_tables函数负责identity mapping和kernel image mapping。

* identity map：是指把idmap\_text区域的物理地址映射到相等的虚拟地址上，这种映射完成后，其虚拟地址等于物理地址。idmap\_text区域都是一些打开MMU相关的代码。
* kernel image map：将kernel运行需要的地址（kernel txt、rodata、data、bss等等）进行映射。

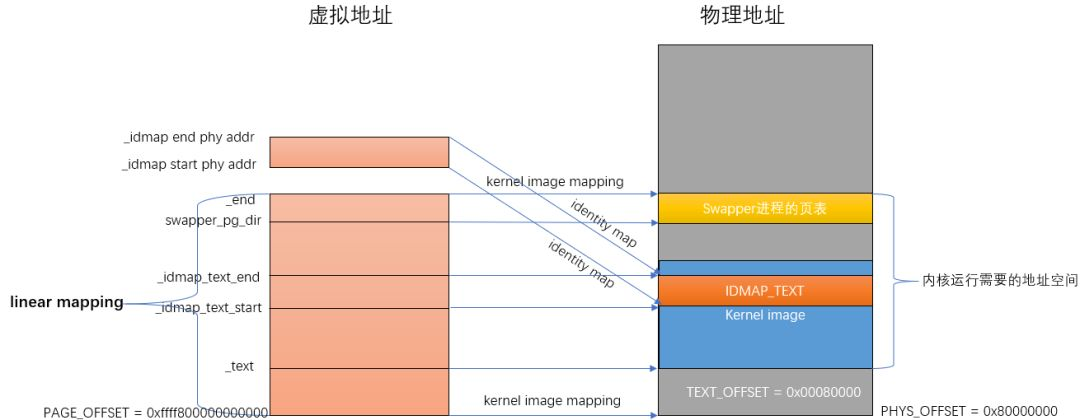
arch/arm64/kernel/head.S:  
ENTRY(stext)  
        bl      preserve\_boot\_args  
        bl      el2\_setup                       // Drop to EL1, w0=cpu\_boot\_mode  
        adrp    x23, \_\_PHYS\_OFFSET  
        and     x23, x23, MIN\_KIMG\_ALIGN - 1    // KASLR offset, defaults to 0  
        bl      set\_cpu\_boot\_mode\_flag  
        bl      \_\_create\_page\_tables  
        /\*  
         \* The following calls CPU setup code, see arch/arm64/mm/proc.S for  
         \* details.  
         \* On return, the CPU will be ready for the MMU to be turned on and  
         \* the TCR will have been set.  
         \*/  
        bl      \_\_cpu\_setup                     // initialise processor  
        b       \_\_primary\_switch  
ENDPROC(stext)

\_\_create\_page\_tables主要执行的就是identity map和kernel image map：

 \_\_create\_page\_tables:  
......  
        create\_pgd\_entry x0, x3, x5, x6  
        mov     x5, x3                          // \_\_pa(\_\_idmap\_text\_start)  
        adr\_l   x6, \_\_idmap\_text\_end            // \_\_pa(\_\_idmap\_text\_end)  
        create\_block\_map x0, x7, x3, x5, x6  
  
        /\*  
         \* Map the kernel image (starting with PHYS\_OFFSET).  
         \*/  
        adrp    x0, swapper\_pg\_dir  
        mov\_q   x5, KIMAGE\_VADDR + TEXT\_OFFSET  // compile time \_\_va(\_text)  
        add     x5, x5, x23                     // add KASLR displacement  
        create\_pgd\_entry x0, x5, x3, x6  
        adrp    x6, \_end                        // runtime \_\_pa(\_end)  
        adrp    x3, \_text                       // runtime \_\_pa(\_text)  
        sub     x6, x6, x3                      // \_end - \_text  
        add     x6, x6, x5                      // runtime \_\_va(\_end)  
        create\_block\_map x0, x7, x3, x5, x6  
 ......

其中调用create\_pgd\_entry进行PGD及所有中间level(PUD, PMD)页表的创建，调用create\_block\_map进行PTE页表的映射。关于四级页表的关系如下图所示。

汇编结束后的内存映射关系如下图所示：



等内存初始化后就可以进入真正的内存管理了，初始化我总结了一下，大体分为四步：

1. 物理内存进系统前
2. 用memblock模块来对内存进行管理
3. 页表映射
4. zone初始化

### Linux是如何组织物理内存的？

node 目前计算机系统有两种体系结构：

1. 非一致性内存访问 NUMA（Non-Uniform Memory Access）意思是内存被划分为各个node，访问一个node花费的时间取决于CPU离这个node的距离。每一个cpu内部有一个本地的node，访问本地node时间比访问其他node的速度快
2. 一致性内存访问 UMA（Uniform Memory Access）也可以称为SMP（Symmetric Multi-Process）对称多处理器。意思是所有的处理器访问内存花费的时间是一样的。也可以理解整个内存只有一个node。

zone

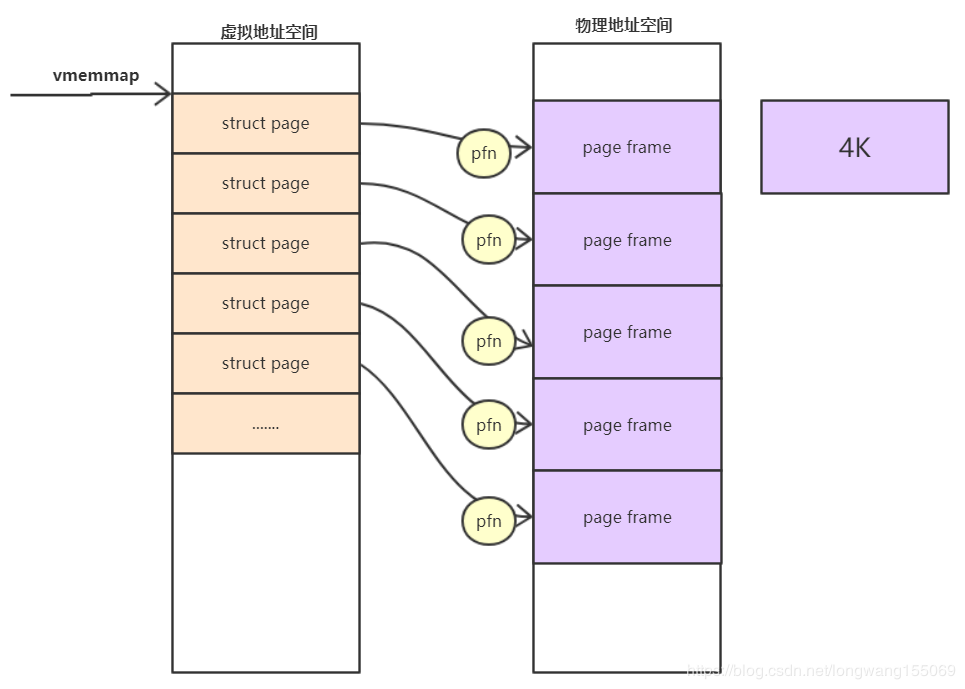
ZONE的意思是把整个物理内存划分为几个区域，每个区域有特殊的含义

page

代表一个物理页，在内核中一个物理页用一个struct page表示。

page frame

为了描述一个物理page，内核使用struct page结构来表示一个物理页。假设一个page的大小是4K的，内核会将整个物理内存分割成一个一个4K大小的物理页，而4K大小物理页的区域我们称为page frame



page frame num(pfn)

pfn是对每个page frame的编号。故物理地址和pfn的关系是：

物理地址>>PAGE\_SHIFT = pfn

pfn和page的关系

内核中支持了好几个内存模型：CONFIG\_FLATMEM（平坦内存模型）CONFIG\_DISCONTIGMEM（不连续内存模型）CONFIG\_SPARSEMEM\_VMEMMAP（稀疏的内存模型）目前ARM64使用的稀疏的类型模式。

系统启动的时候，内核会将整个struct page映射到内核虚拟地址空间vmemmap的区域，所以我们可以简单的认为struct page的基地址是vmemmap，则：

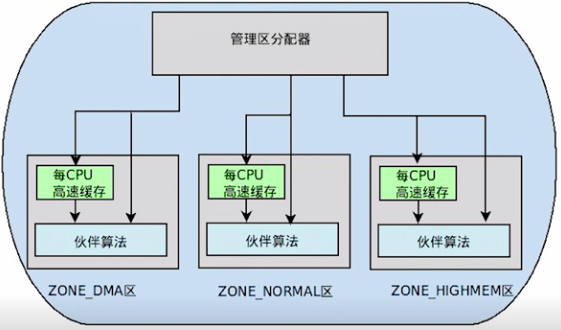
vmemmap+pfn的地址就是此struct page对应的地址。

### Linux分区页框分配器

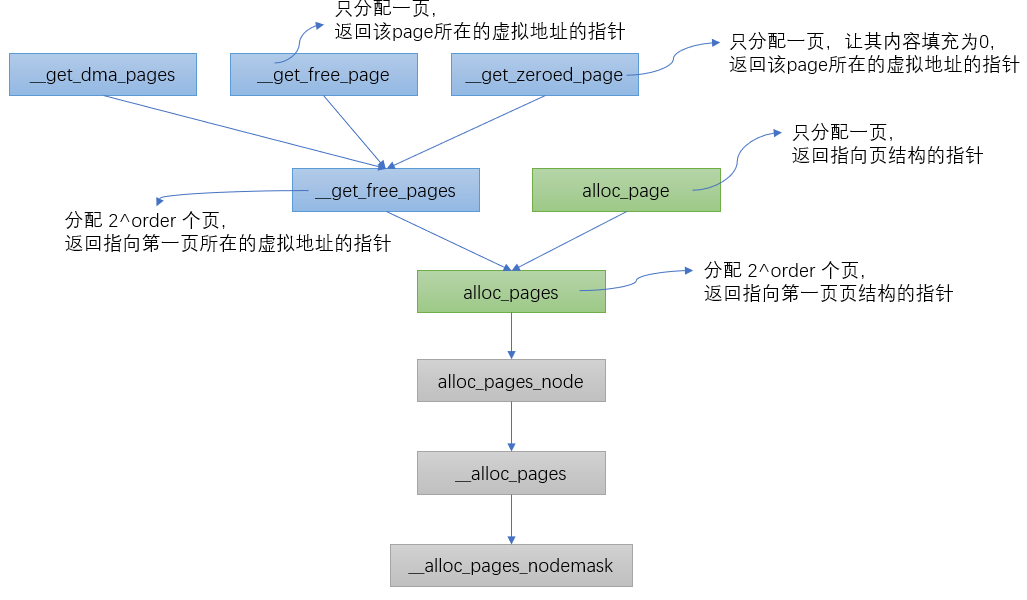
页框分配在内核里的机制我们叫做分区页框分配器(zoned page frame allocator)，在linux系统中，分区页框分配器管理着所有物理内存，无论你是内核还是进程，都需要请求分区页框分配器，这时才会分配给你应该获得的物理内存页框。当你所拥有的页框不再使用时，你必须释放这些页框，让这些页框回到管理区页框分配器当中。

有时候目标管理区不一定有足够的页框去满足分配，这时候系统会从另外两个管理区中获取要求的页框，但这是按照一定规则去执行的，如下：

* 如果要求从DMA区中获取，就只能从ZONE\_DMA区中获取。
* 如果没有规定从哪个区获取，就按照顺序从 ZONE\_NORMAL -> ZONE\_DMA 获取。
* 如果规定从HIGHMEM区获取，就按照顺序从 ZONE\_HIGHMEM -> ZONE\_NORMAL -> ZONE\_DMA 获取。



内核中根据不同的分配需求有6个函数接口来请求页框，最终都会调用到\_\_alloc\_pages\_nodemask。



struct page \*  
\_\_alloc\_pages\_nodemask(gfp\_t gfp\_mask, unsigned int order, int preferred\_nid,  
       nodemask\_t \*nodemask)  
{  
  page = get\_page\_from\_freelist(alloc\_mask, order, alloc\_flags, &ac);//fastpath分配页面：从pcp(per\_cpu\_pages)和伙伴系统中正常的分配内存空间  
  ......  
  page = \_\_alloc\_pages\_slowpath(alloc\_mask, order, &ac);//slowpath分配页面：如果上面没有分配到空间，调用下面函数慢速分配，允许等待和回收  
  ......  
}

在页面分配时，有两种路径可以选择，如果在快速路径中分配成功了，则直接返回分配的页面；快速路径分配失败则选择慢速路径来进行分配。总结如下：

* 正常分配（或叫快速分配）：

1. 如果分配的是单个页面，考虑从per CPU缓存中分配空间，如果缓存中没有页面，从伙伴系统中提取页面做补充。
2. 分配多个页面时，从指定类型中分配，如果指定类型中没有足够的页面，从备用类型链表中分配。最后会试探保留类型链表。

* 慢速（允许等待和页面回收）分配：

1. 当上面两种分配方案都不能满足要求时，考虑页面回收、杀死进程等操作后在试。

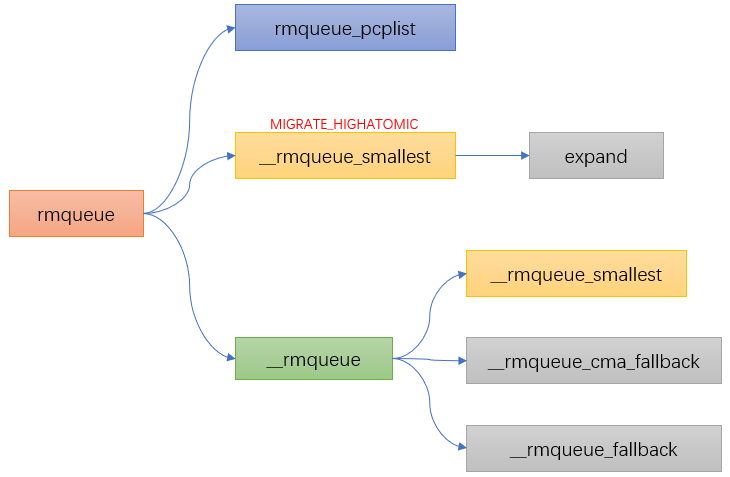
#### Linux页框分配器之伙伴算法

static struct page \*  
get\_page\_from\_freelist(gfp\_t gfp\_mask, unsigned int order, int alloc\_flags,  
      const struct alloc\_context \*ac)  
{  
  for\_next\_zone\_zonelist\_nodemask(zone, z, ac->zonelist, ac->high\_zoneidx, ac->nodemask)  
  {  
    if (!zone\_watermark\_fast(zone, order, mark, ac\_classzone\_idx(ac), alloc\_flags))  
    {  
      ret = node\_reclaim(zone->zone\_pgdat, gfp\_mask, order);   
      switch (ret) {  
      case NODE\_RECLAIM\_NOSCAN:  
        continue;  
      case NODE\_RECLAIM\_FULL:  
        continue;  
      default:  
        if (zone\_watermark\_ok(zone, order, mark, ac\_classzone\_idx(ac), alloc\_flags))  
          goto try\_this\_zone;  
  
        continue;  
      }  
    }  
      
try\_this\_zone: //本zone正常水位  
    page = rmqueue(ac->preferred\_zoneref->zone, zone, order, gfp\_mask, alloc\_flags, ac->migratetype);  
  }  
    
  return NULL;  
}

首先遍历当前zone，按照HIGHMEM->NORMAL的方向进行遍历，判断当前zone是否能够进行内存分配的条件是首先判断free memory是否满足low water mark水位值，如果不满足则进行一次快速的内存回收操作，然后再次检测是否满足low water mark，如果还是不能满足，相同步骤遍历下一个zone，满足的话进入正常的分配情况，即rmqueue函数，这也是伙伴系统的核心。

#### Buddy 分配函数

static inline  
struct page \*rmqueue(struct zone \*preferred\_zone,  
   struct zone \*zone, unsigned int order,  
   gfp\_t gfp\_flags, unsigned int alloc\_flags,  
   int migratetype)  
{  
  if (likely(order == 0)) { //如果order=0则从pcp中分配  
    page = rmqueue\_pcplist(preferred\_zone, zone, order, gfp\_flags, migratetype);  
 }  
  do {  
    page = NULL;  
    if (alloc\_flags & ALLOC\_HARDER) {//如果分配标志中设置了ALLOC\_HARDER，则从free\_list[MIGRATE\_HIGHATOMIC]的链表中进行页面分配  
        page = \_\_rmqueue\_smallest(zone, order, MIGRATE\_HIGHATOMIC);  
    }  
    if (!page) //前两个条件都不满足，则在正常的free\_list[MIGRATE\_\*]中进行分配  
      page = \_\_rmqueue(zone, order, migratetype);  
  } while (page && check\_new\_pages(page, order));  
  ......  
}



### Linux分区页框分配器之水位

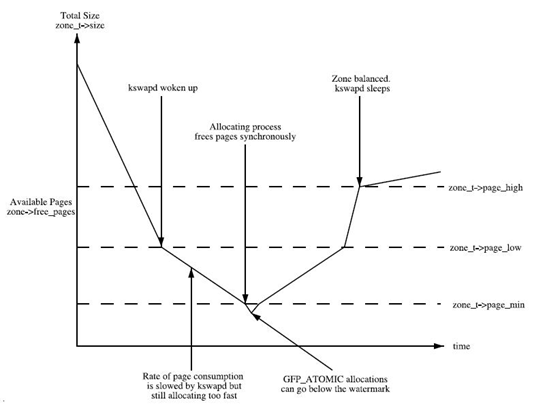
我们讲页框分配器的时候讲到了快速分配和慢速分配，其中伙伴算法是在快速分配里做的。

可以看到在进行伙伴算法分配前有个关于水位的判断，今天我们就看下水位的概念。

简单的说在使用分区页面分配器中会将可以用的free pages与zone里的水位(watermark)进行比较。

#### 水位初始化

* nr\_free\_buffer\_pages 是获取ZONE\_DMA和ZONE\_NORMAL区中高于high水位的总页数nr\_free\_buffer\_pages = managed\_pages - high\_pages
* min\_free\_kbytes 是总的min大小，min\_free\_kbytes = 4 \* sqrt(lowmem\_kbytes)
* setup\_per\_zone\_wmarks 根据总的min值，再加上各个zone在总内存中的占比，然后通过do\_div就计算出他们各自的min值，进而计算出各个zone的水位大小。min,low,high的关系如下：low = min \*125%;
* high = min \* 150%
* min:low:high = 4:5:6
* setup\_per\_zone\_lowmem\_reserve 当从Normal失败后，会尝试从DMA申请分配，通过lowmem\_reserve[DMA]，限制来自Normal的分配请求。其值可以通过/proc/sys/vm/lowmem\_reserve\_ratio来修改。



从这张图可以看出：

* 如果空闲页数目min值，则该zone非常缺页，页面回收压力很大，应用程序写内存操作就会被阻塞，直接在应用程序的进程上下文中进行回收，即direct reclaim。
* 如果空闲页数目小于low值，kswapd线程将被唤醒，并开始释放回收页面。
* 如果空闲页面的值大于high值，则该zone的状态很完美, kswapd线程将重新休眠。

### Linux页框分配器之内存碎片化整理

#### 什么是内存碎片化

Linux物理内存碎片化包括两种：内部碎片化和外部碎片化。

* 内部碎片化：

指分配给用户的内存空间中未被使用的部分。

例如进程需要使用3K bytes物理内存，于是向系统申请了大小等于3Kbytes的内存，但是由于Linux内核伙伴系统算法最小颗粒是4K bytes，所以分配的是4Kbytes内存，那么其中1K bytes未被使用的内存就是内存内碎片。

* 外部碎片化：

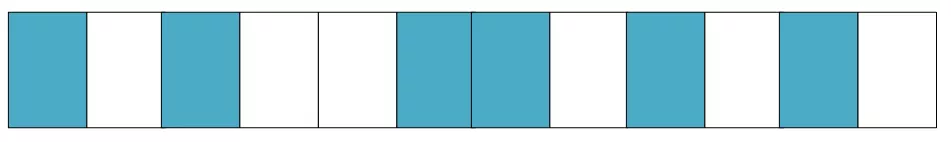
指系统中无法利用的小内存块。

例如系统剩余内存为16K bytes，但是这16K bytes内存是由4个4K bytes的页面组成，即16K内存物理页帧号#1不连续。在系统剩余16K bytes内存的情况下，系统却无法成功分配大于4K的连续物理内存，该情况就是内存外碎片导致。

#### 碎片化整理算法

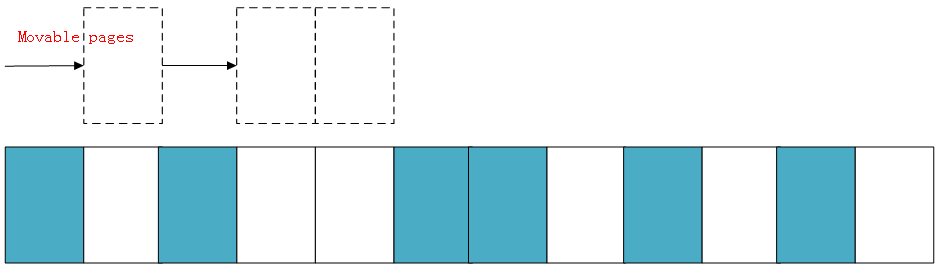
Linux内存对碎片化的整理算法主要应用了内核的页面迁移机制，是一种将可移动页面进行迁移后腾出连续物理内存的方法。

假设存在一个非常小的内存域如下：

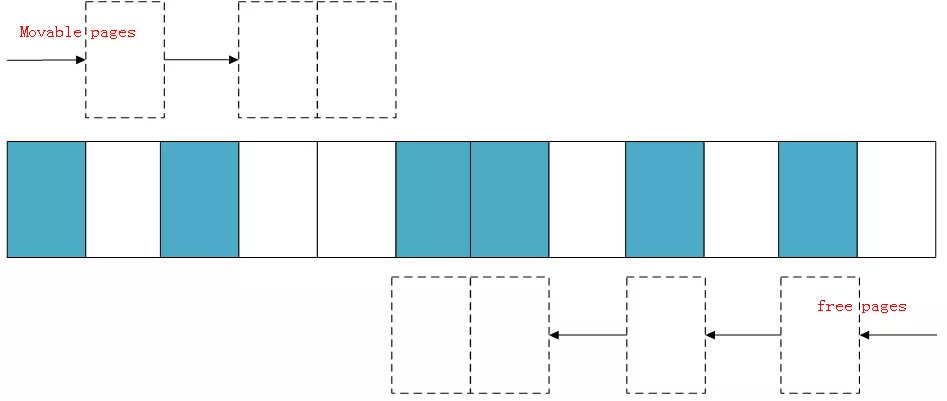


蓝色表示空闲的页面，白色表示已经被分配的页面，可以看到如上内存域的空闲页面（蓝色）非常零散，无法分配大于两页的连续物理内存。

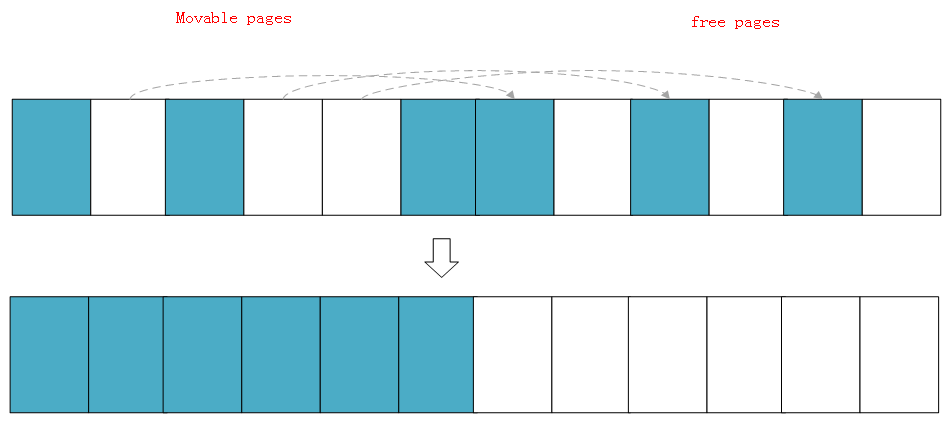
下面演示一下内存规整的简化工作原理，内核会运行两个独立的扫描动作：第一个扫描从内存域的底部开始，一边扫描一边将已分配的可移动（MOVABLE）页面记录到一个列表中：



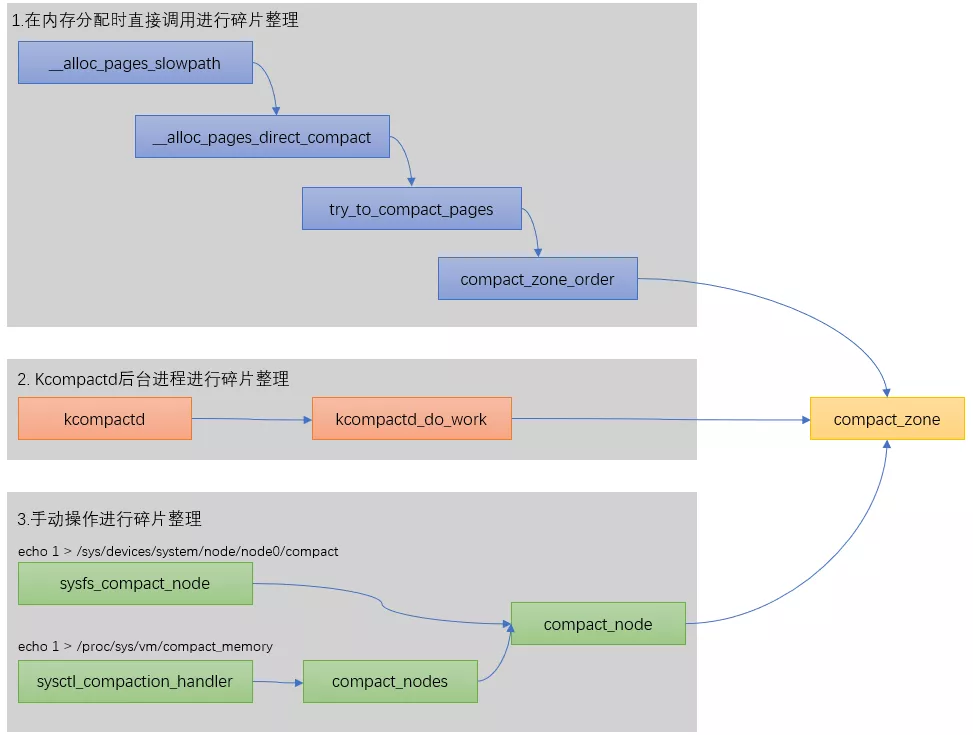
另外第二扫描是从内存域的顶部开始，扫描可以作为页面迁移目标的空闲页面位置，然后也记录到一个列表里面：



等两个扫描在域中间相遇，意味着扫描结束，然后将左边扫描得到的已分配的页面迁移到右边空闲的页面中，左边就形成了一段连续的物理内存，完成页面规整。



static struct page \*  
\_\_alloc\_pages\_direct\_compact(gfp\_t gfp\_mask, unsigned int order,  
  unsigned int alloc\_flags, const struct alloc\_context \*ac,  
  enum compact\_priority prio, enum compact\_result \*compact\_result)  
{  
 struct page \*page;  
 unsigned int noreclaim\_flag;  
  
 if (!order)  
  return NULL;  
  
 noreclaim\_flag = memalloc\_noreclaim\_save();  
 \*compact\_result = try\_to\_compact\_pages(gfp\_mask, order, alloc\_flags, ac,  
         prio);  
 memalloc\_noreclaim\_restore(noreclaim\_flag);  
  
 if (\*compact\_result <= COMPACT\_INACTIVE)  
  return NULL;  
  
 count\_vm\_event(COMPACTSTALL);  
  
 page = get\_page\_from\_freelist(gfp\_mask, order, alloc\_flags, ac);  
  
 if (page) {  
  struct zone \*zone = page\_zone(page);  
  
  zone->compact\_blockskip\_flush = false;  
  compaction\_defer\_reset(zone, order, true);  
  count\_vm\_event(COMPACTSUCCESS);  
  return page;  
 }  
  
 count\_vm\_event(COMPACTFAIL);  
  
 cond\_resched();  
  
 return NULL;  
}

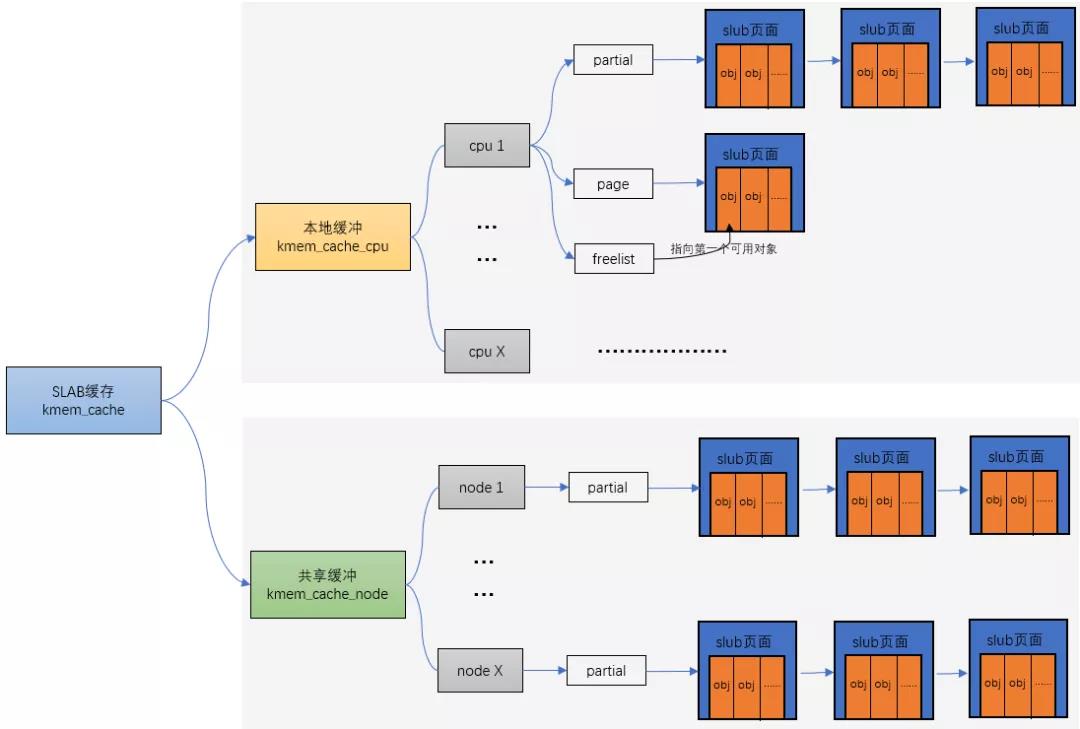


### Linux slab分配器

在Linux中，伙伴系统是以页为单位分配内存。但是现实中很多时候却以字节为单位，不然申请10Bytes内存还要给1页的话就太浪费了。

slab分配器就是为小内存分配而生的。slab分配器分配内存以Byte为单位。但是slab分配器并没有脱离伙伴系统，而是基于伙伴系统分配的大内存进一步细分成小内存分配。

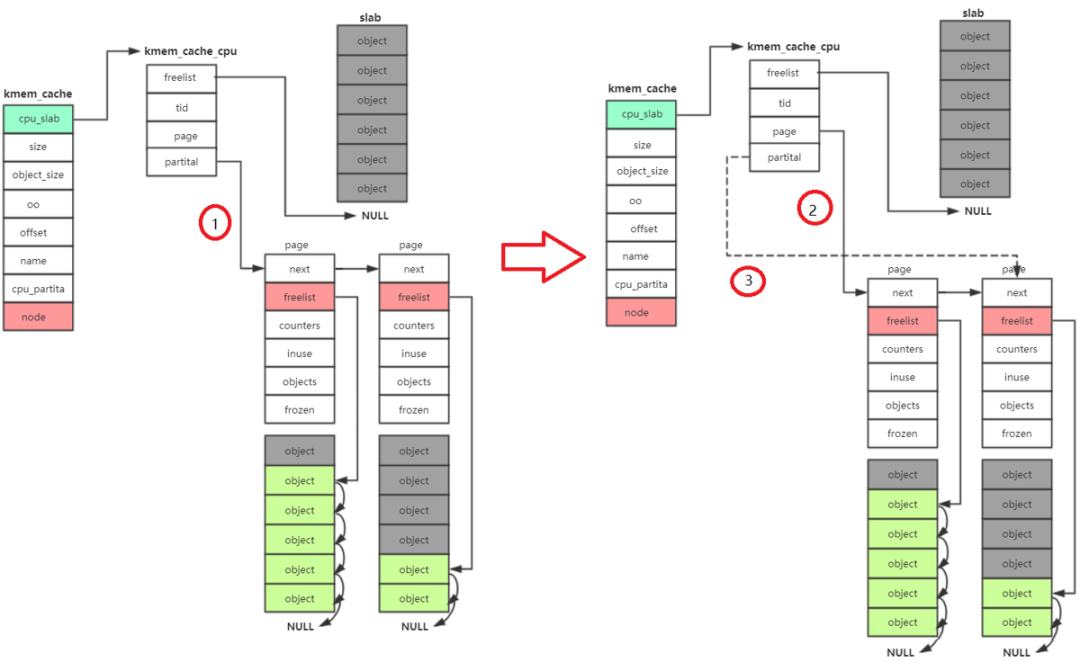
他们之间的关系可以用一张图来描述：



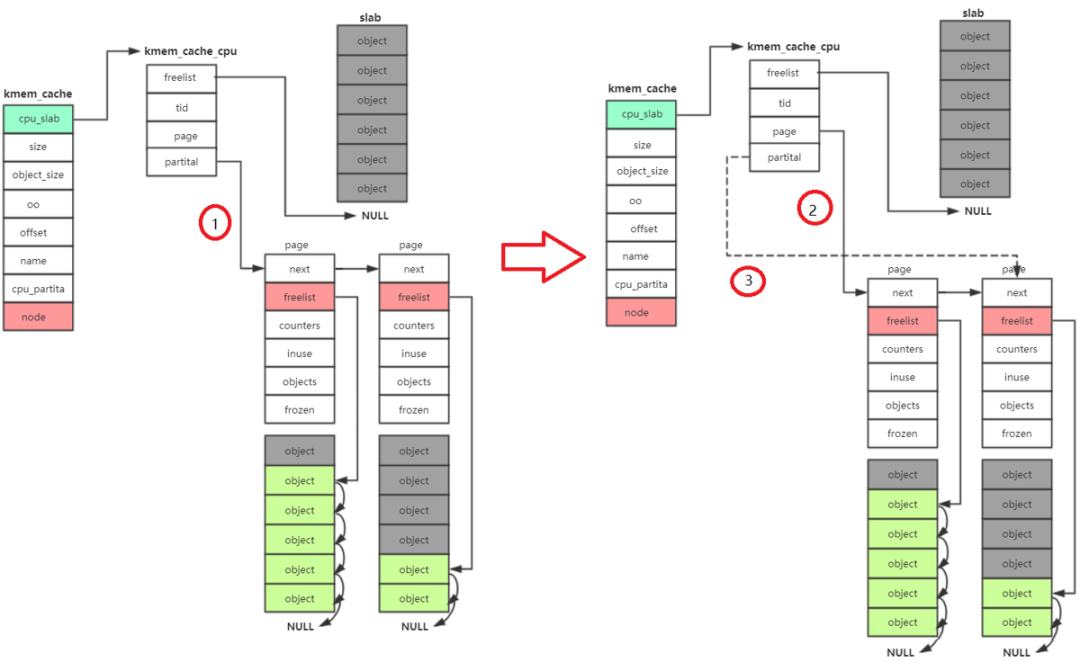
流程分析

kmem\_cache\_alloc 主要四步：

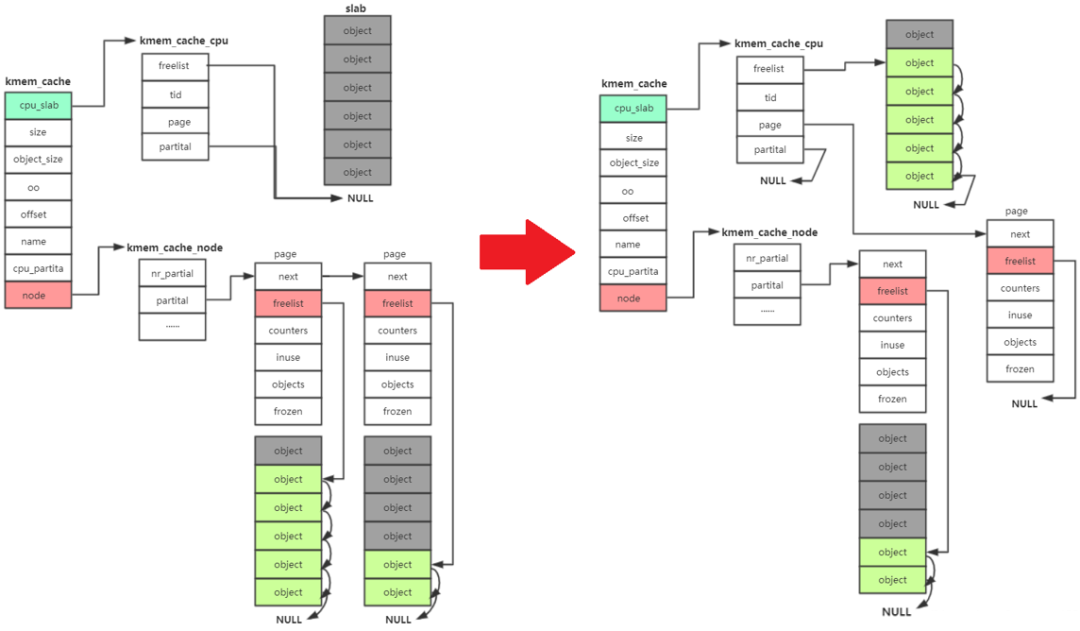
1. 先从 kmem\_cache\_cpu->freelist中分配，如果freelist为null



1. 接着去 kmem\_cache\_cpu->partital链表中分配，如果此链表为null



1. 接着去 kmem\_cache\_node->partital链表分配，如果此链表为null

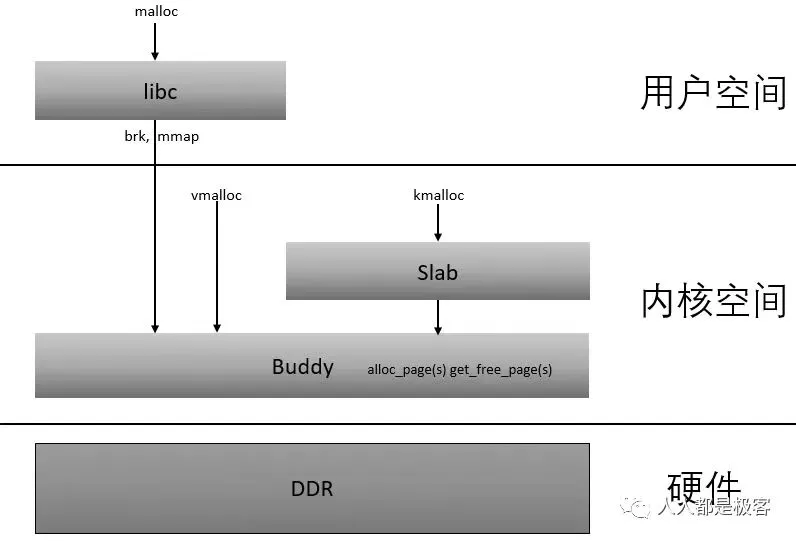


1. 重新分配一个slab。

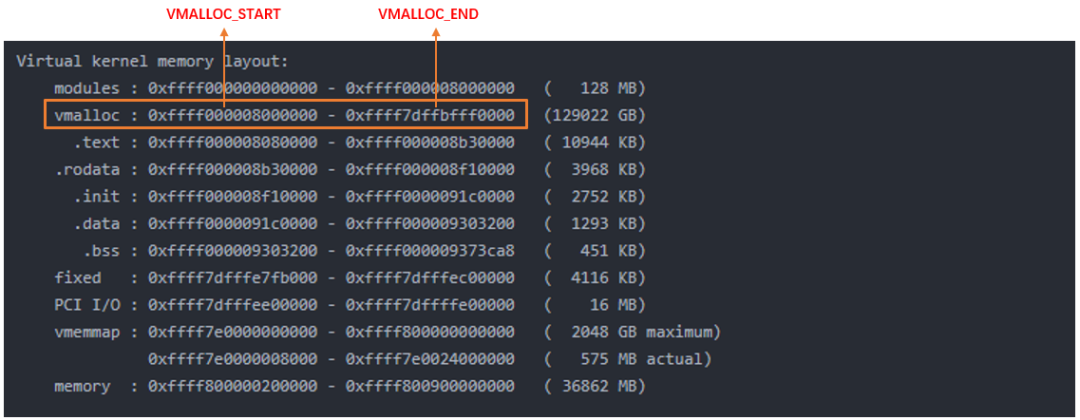
### Linux 内存管理之vmalloc

根据前面的系列文章，我们知道了buddy system是基于页框分配器，kmalloc是基于slab分配器，而且这些分配的地址都是物理内存连续的。

但是随着碎片化的积累，连续物理内存的分配就会变得困难，对于那些非DMA访问，不一定非要连续物理内存的话完全可以像malloc那样，将不连续的物理内存页框映射到连续的虚拟地址空间中，这就是vmap的来源）（提供把离散的page映射到连续的虚拟地址空间），vmalloc的分配就是基于这个机制来实现的。



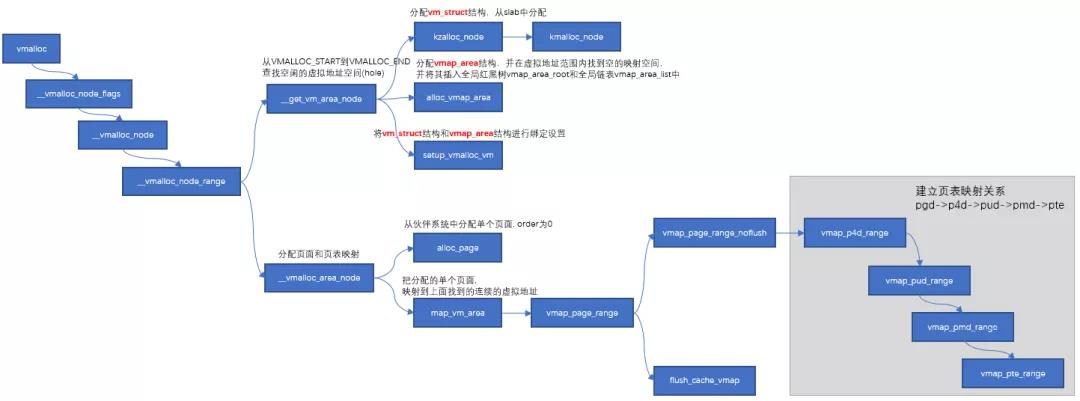
vmalloc最小分配一个page，并且分配到的页面不保证是连续的，因为vmalloc内部调用alloc\_page多次分配单个页面。



#### vmalloc流程

主要分以下三步：

1. 从VMALLOC\_START到VMALLOC\_END查找空闲的虚拟地址空间(hole)
2. 根据分配的size,调用alloc\_page依次分配单个页面.
3. 把分配的单个页面，映射到第一步中找到的连续的虚拟地址。把分配的单个页面，映射到第一步中找到的连续的虚拟地址。



### Linux进程的内存管理之缺页异常

当进程访问这些还没建立映射关系的虚拟地址时，处理器会自动触发缺页异常。

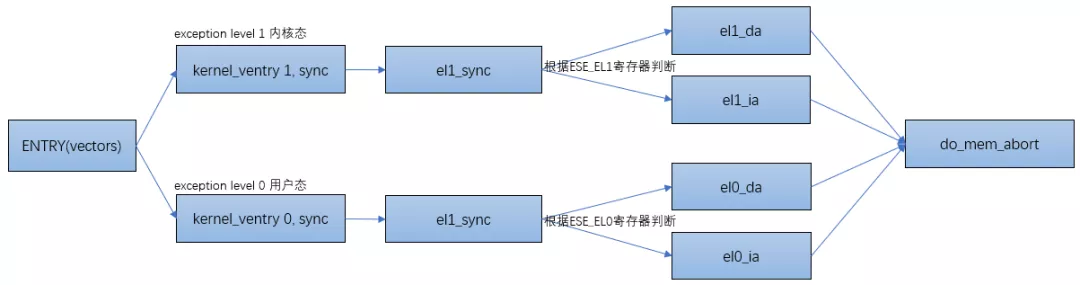
ARM64把异常分为同步异常和异步异常，通常异步异常指的是中断（可看《上帝视角看中断》），同步异常指的是异常。关于ARM异常处理的文章可参考《ARMv8异常处理简介》。

当处理器有异常发生时，处理器会先跳转到ARM64的异常向量表中：

ENTRY(vectors)  
 kernel\_ventry 1, sync\_invalid   // Synchronous EL1t  
 kernel\_ventry 1, irq\_invalid   // IRQ EL1t  
 kernel\_ventry 1, fiq\_invalid   // FIQ EL1t  
 kernel\_ventry 1, error\_invalid  // Error EL1t  
  
 kernel\_ventry 1, sync    // Synchronous EL1h  
 kernel\_ventry 1, irq    // IRQ EL1h  
 kernel\_ventry 1, fiq\_invalid   // FIQ EL1h  
 kernel\_ventry 1, error\_invalid  // Error EL1h  
  
 kernel\_ventry 0, sync    // Synchronous 64-bit EL0  
 kernel\_ventry 0, irq    // IRQ 64-bit EL0  
 kernel\_ventry 0, fiq\_invalid   // FIQ 64-bit EL0  
 kernel\_ventry 0, error\_invalid  // Error 64-bit EL0  
  
#ifdef CONFIG\_COMPAT  
 kernel\_ventry 0, sync\_compat, 32  // Synchronous 32-bit EL0  
 kernel\_ventry 0, irq\_compat, 32  // IRQ 32-bit EL0  
 kernel\_ventry 0, fiq\_invalid\_compat, 32 // FIQ 32-bit EL0  
 kernel\_ventry 0, error\_invalid\_compat, 32 // Error 32-bit EL0  
#else  
 kernel\_ventry 0, sync\_invalid, 32  // Synchronous 32-bit EL0  
 kernel\_ventry 0, irq\_invalid, 32  // IRQ 32-bit EL0  
 kernel\_ventry 0, fiq\_invalid, 32  // FIQ 32-bit EL0  
 kernel\_ventry 0, error\_invalid, 32  // Error 32-bit EL0  
#endif  
END(vectors)

以el1下的异常为例，当跳转到el1\_sync函数时，读取ESR的值以判断异常类型。根据类型跳转到不同的处理函数里，如果是data abort的话跳转到el1\_da函数里，instruction abort的话跳转到el1\_ia函数里：

el1\_sync:  
 kernel\_entry 1  
 mrs x1, esr\_el1   // read the syndrome register  
 lsr x24, x1, #ESR\_ELx\_EC\_SHIFT // exception class  
 cmp x24, #ESR\_ELx\_EC\_DABT\_CUR // data abort in EL1  
 b.eq el1\_da  
 cmp x24, #ESR\_ELx\_EC\_IABT\_CUR // instruction abort in EL1  
 b.eq el1\_ia  
 cmp x24, #ESR\_ELx\_EC\_SYS64  // configurable trap  
 b.eq el1\_undef  
 cmp x24, #ESR\_ELx\_EC\_SP\_ALIGN // stack alignment exception  
 b.eq el1\_sp\_pc  
 cmp x24, #ESR\_ELx\_EC\_PC\_ALIGN // pc alignment exception  
 b.eq el1\_sp\_pc  
 cmp x24, #ESR\_ELx\_EC\_UNKNOWN // unknown exception in EL1  
 b.eq el1\_undef  
 cmp x24, #ESR\_ELx\_EC\_BREAKPT\_CUR // debug exception in EL1  
 b.ge el1\_dbg  
 b el1\_inv



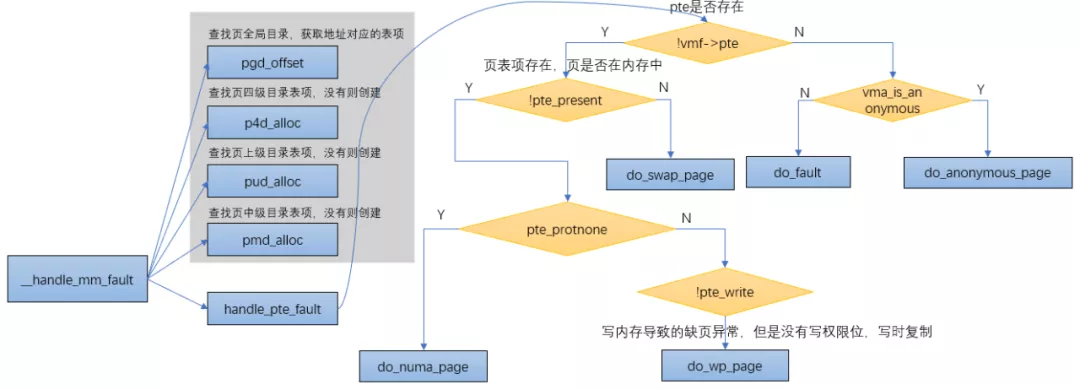
#### do\_page\_fault

static int \_\_do\_page\_fault(struct mm\_struct \*mm, unsigned long addr,  
      unsigned int mm\_flags, unsigned long vm\_flags,  
      struct task\_struct \*tsk)  
{  
 struct vm\_area\_struct \*vma;  
 int fault;  
  
 vma = find\_vma(mm, addr);  
 fault = VM\_FAULT\_BADMAP; //没有找到vma区域，说明addr还没有在进程的地址空间中  
 if (unlikely(!vma))  
  goto out;  
 if (unlikely(vma->vm\_start > addr))  
  goto check\_stack;  
  
 /\*  
  \* Ok, we have a good vm\_area for this memory access, so we can handle  
  \* it.  
  \*/  
good\_area://一个好的vma  
 /\*  
  \* Check that the permissions on the VMA allow for the fault which  
  \* occurred.  
  \*/  
 if (!(vma->vm\_flags & vm\_flags)) {//权限检查  
  fault = VM\_FAULT\_BADACCESS;   
  goto out;  
 }  
  
 //重新建立物理页面到VMA的映射关系  
 return handle\_mm\_fault(vma, addr & PAGE\_MASK, mm\_flags);  
  
check\_stack:  
 if (vma->vm\_flags & VM\_GROWSDOWN && !expand\_stack(vma, addr))  
  goto good\_area;  
out:  
 return fault;  
}

从\_\_do\_page\_fault函数能看出来，当触发异常的虚拟地址属于某个vma，并且拥有触发页错误异常的权限时，会调用到handle\_mm\_fault函数来建立vma和物理地址的映射，而handle\_mm\_fault函数的主要逻辑是通过\_\_handle\_mm\_fault来实现的。

#### \_\_handle\_mm\_fault

static int \_\_handle\_mm\_fault(struct vm\_area\_struct \*vma, unsigned long address,  
  unsigned int flags)  
{  
  ......  
 //查找页全局目录，获取地址对应的表项  
 pgd = pgd\_offset(mm, address);  
 //查找页四级目录表项，没有则创建  
 p4d = p4d\_alloc(mm, pgd, address);  
 if (!p4d)  
  return VM\_FAULT\_OOM;  
  
 //查找页上级目录表项，没有则创建  
 vmf.pud = pud\_alloc(mm, p4d, address);  
 ......  
 //查找页中级目录表项，没有则创建  
 vmf.pmd = pmd\_alloc(mm, vmf.pud, address);  
  ......  
 //处理pte页表  
 return handle\_pte\_fault(&vmf);  
}



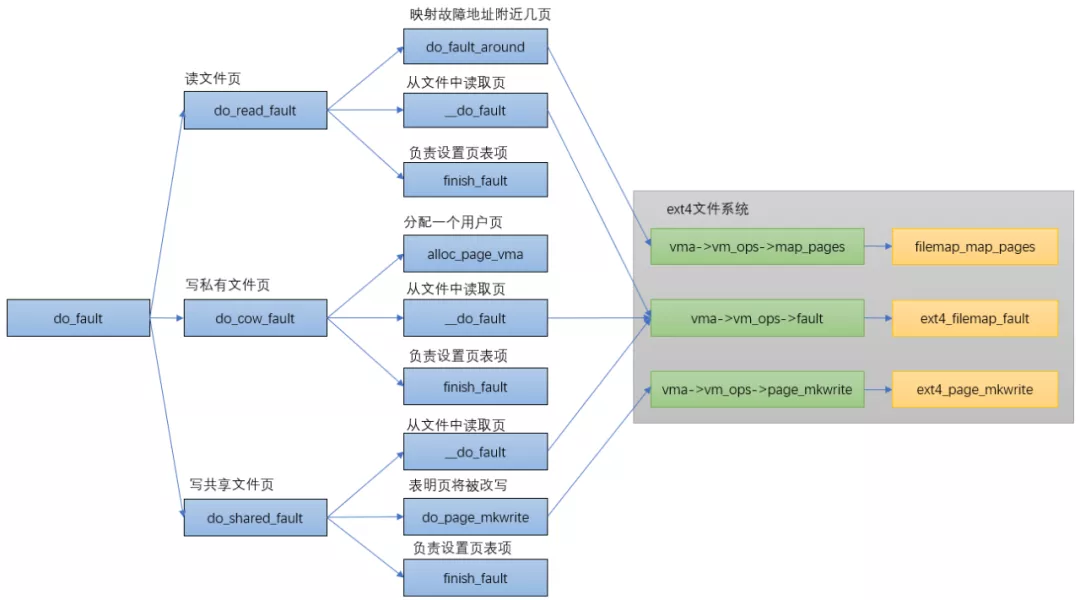
#### do\_anonymous\_page

匿名页缺页异常，对于匿名映射，映射完成之后，只是获得了一块虚拟内存，并没有分配物理内存，当第一次访问的时候：

1. 如果是读访问，会将虚拟页映射到０页，以减少不必要的内存分配
2. 如果是写访问，用alloc\_zeroed\_user\_highpage\_movable分配新的物理页，并用０填充，然后映射到虚拟页上去
3. 如果是先读后写访问，则会发生两次缺页异常：第一次是匿名页缺页异常的读的处理（虚拟页到0页的映射），第二次是写时复制缺页异常处理。

从上面的总结我们知道，第一次访问匿名页时有三种情况，其中第一种和第三种情况都会涉及到0页。

#### do\_fault



#### do\_swap\_page

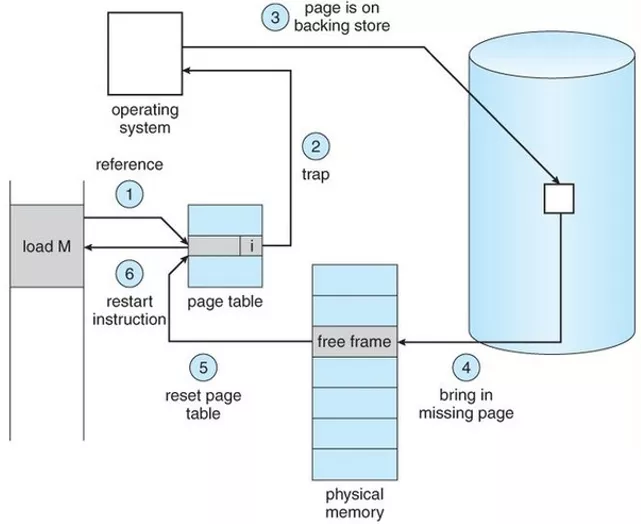
上面已经讲过，pte对应的内容不为0(页表项存在)，但是pte所对应的page不在内存中时，表示此时pte的内容所对应的页面在swap空间中，缺页异常时会通过do\_swap\_page()函数来分配页面。

do\_swap\_page发生在swap in的时候，即查找磁盘上的slot，并将数据读回。

换入的过程如下：

1. 查找swap cache中是否存在所查找的页面，如果存在，则根据swap cache引用的内存页，重新映射并更新页表；如果不存在，则分配新的内存页，并添加到swap cache的引用中，更新内存页内容完成后，更新页表。
2. 换入操作结束后，对应swap area的页引用减1，当减少到0时，代表没有任何进程引用了该页，可以进行回收。

int do\_swap\_page(struct vm\_fault \*vmf)  
{  
  ......  
 //根据pte找到swap entry, swap entry和pte有一个对应关系  
 entry = pte\_to\_swp\_entry(vmf->orig\_pte);  
  ......  
 if (!page)  
  //根据entry从swap缓存中查找页, 在swapcache里面寻找entry对应的page  
  //Lookup a swap entry in the swap cache  
  page = lookup\_swap\_cache(entry, vma\_readahead ? vma : NULL,  
      vmf->address);  
 //没有找到页  
 if (!page) {  
  if (vma\_readahead)  
   page = do\_swap\_page\_readahead(entry,  
    GFP\_HIGHUSER\_MOVABLE, vmf, &swap\_ra);  
  else  
   //如果swapcache里面找不到就在swap area里面找，分配新的内存页并从swap area中读入  
   page = swapin\_readahead(entry,  
    GFP\_HIGHUSER\_MOVABLE, vma, vmf->address);  
  ......  
 //获取一个pte的entry，重新建立映射  
 vmf->pte = pte\_offset\_map\_lock(vma->vm\_mm, vmf->pmd, vmf->address,  
   &vmf->ptl);  
  ......  
 //anonpage数加1，匿名页从swap空间交换出来，所以加1  
 //swap page个数减1，由page和VMA属性创建一个新的pte  
 inc\_mm\_counter\_fast(vma->vm\_mm, MM\_ANONPAGES);  
 dec\_mm\_counter\_fast(vma->vm\_mm, MM\_SWAPENTS);  
 pte = mk\_pte(page, vma->vm\_page\_prot);  
  ......  
 flush\_icache\_page(vma, page);  
 if (pte\_swp\_soft\_dirty(vmf->orig\_pte))  
  pte = pte\_mksoft\_dirty(pte);  
 //将新生成的PTE entry添加到硬件页表中  
 set\_pte\_at(vma->vm\_mm, vmf->address, vmf->pte, pte);  
 vmf->orig\_pte = pte;  
 //根据page是否为swapcache  
 if (page == swapcache) {  
  //如果是，将swap缓存页用作anon页，添加反向映射rmap中  
  do\_page\_add\_anon\_rmap(page, vma, vmf->address, exclusive);  
  mem\_cgroup\_commit\_charge(page, memcg, true, false);  
  //并添加到active链表中  
  activate\_page(page);  
 //如果不是  
 } else { /\* ksm created a completely new copy \*/  
  //使用新页面并复制swap缓存页，添加反向映射rmap中  
  page\_add\_new\_anon\_rmap(page, vma, vmf->address, false);  
  mem\_cgroup\_commit\_charge(page, memcg, false, false);  
  //并添加到lru链表中  
  lru\_cache\_add\_active\_or\_unevictable(page, vma);  
 }  
  
 //释放swap entry  
 swap\_free(entry);  
  ......  
 if (vmf->flags & FAULT\_FLAG\_WRITE) {  
  //有写请求则写时复制  
  ret |= do\_wp\_page(vmf);  
  if (ret & VM\_FAULT\_ERROR)  
   ret &= VM\_FAULT\_ERROR;  
  goto out;  
 }  
  ......  
  return ret;  
}



#### do\_wp\_page

走到这里说明页面在内存中，只是PTE只有读权限，而又要写内存的时候就会触发do\_wp\_page。

do\_wp\_page函数用于处理写时复制（copy on write），其流程比较简单，主要是分配新的物理页，拷贝原来页的内容到新页，然后修改页表项内容指向新页并修改为可写(vma具备可写属性)。

static int do\_wp\_page(struct vm\_fault \*vmf)  
 \_\_releases(vmf->ptl)  
{  
 struct vm\_area\_struct \*vma = vmf->vma;  
  
 //从页表项中得到页帧号，再得到页描述符，发生异常时地址所在的page结构  
 vmf->page = vm\_normal\_page(vma, vmf->address, vmf->orig\_pte);  
 if (!vmf->page) {  
  //没有page结构是使用页帧号的特殊映射  
  /\*  
   \* VM\_MIXEDMAP !pfn\_valid() case, or VM\_SOFTDIRTY clear on a  
   \* VM\_PFNMAP VMA.  
   \*  
   \* We should not cow pages in a shared writeable mapping.  
   \* Just mark the pages writable and/or call ops->pfn\_mkwrite.  
   \*/  
  if ((vma->vm\_flags & (VM\_WRITE|VM\_SHARED)) ==  
         (VM\_WRITE|VM\_SHARED))  
   //处理共享可写映射  
   return wp\_pfn\_shared(vmf);  
  
  pte\_unmap\_unlock(vmf->pte, vmf->ptl);  
  //处理私有可写映射  
  return wp\_page\_copy(vmf);  
 }  
  
 /\*  
  \* Take out anonymous pages first, anonymous shared vmas are  
  \* not dirty accountable.  
  \*/  
 if (PageAnon(vmf->page) && !PageKsm(vmf->page)) {  
  int total\_map\_swapcount;  
  if (!trylock\_page(vmf->page)) {  
   //添加原来页的引用计数，方式被释放  
   get\_page(vmf->page);  
   //释放页表锁  
   pte\_unmap\_unlock(vmf->pte, vmf->ptl);  
   lock\_page(vmf->page);  
   vmf->pte = pte\_offset\_map\_lock(vma->vm\_mm, vmf->pmd,  
     vmf->address, &vmf->ptl);  
   if (!pte\_same(\*vmf->pte, vmf->orig\_pte)) {  
    unlock\_page(vmf->page);  
    pte\_unmap\_unlock(vmf->pte, vmf->ptl);  
    put\_page(vmf->page);  
    return 0;  
   }  
   put\_page(vmf->page);  
  }  
  //单身匿名页面的处理  
  if (reuse\_swap\_page(vmf->page, &total\_map\_swapcount)) {  
   if (total\_map\_swapcount == 1) {  
    /\*  
     \* The page is all ours. Move it to  
     \* our anon\_vma so the rmap code will  
     \* not search our parent or siblings.  
     \* Protected against the rmap code by  
     \* the page lock.  
     \*/  
    page\_move\_anon\_rmap(vmf->page, vma);  
   }  
   unlock\_page(vmf->page);  
   wp\_page\_reuse(vmf);  
   return VM\_FAULT\_WRITE;  
  }  
  unlock\_page(vmf->page);  
 } else if (unlikely((vma->vm\_flags & (VM\_WRITE|VM\_SHARED)) ==  
     (VM\_WRITE|VM\_SHARED))) {  
  //共享可写，不需要复制物理页，设置页表权限即可  
  return wp\_page\_shared(vmf);  
 }  
  
 /\*  
  \* Ok, we need to copy. Oh, well..  
  \*/  
 get\_page(vmf->page);  
  
 pte\_unmap\_unlock(vmf->pte, vmf->ptl);  
 //私有可写，复制物理页，将虚拟页映射到物理页  
 return wp\_page\_copy(vmf);  
}

### Linux 内存管理之CMA

CMA是reserved的一块内存，用于分配连续的大块内存。当设备驱动不用时，内存管理系统将该区域用于分配和管理可移动类型页面；当设备驱动使用时，此时已经分配的页面需要进行迁移，又用于连续内存分配；其用法与DMA子系统结合在一起充当DMA的后端，具体可参考《没有IOMMU的DMA操作》

#### CMA区域 cma\_areas 的创建

CMA区域的创建有两种方法，一种是通过dts的reserved memory，另外一种是通过command line参数和内核配置参数。

* dts方式：

reserved-memory {  
        /\* global autoconfigured region for contiguous allocations \*/  
        linux,cma {  
                compatible = "shared-dma-pool";  
                reusable;  
                size = <0 0x28000000>;  
                alloc-ranges = <0 0xa0000000 0 0x40000000>;  
                linux,cma-default;  
        };  
};

device tree中可以包含reserved-memory node，系统启动的时候会打开rmem\_cma\_setup

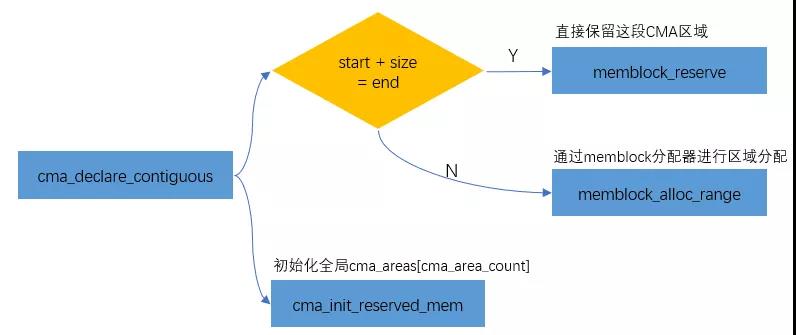
RESERVEDMEM\_OF\_DECLARE(cma, "shared-dma-pool", rmem\_cma\_setup);

* command line方式：cma=nn[MG]@[start[MG][-end[MG]]]

static int \_\_init early\_cma(char \*p)  
{  
 pr\_debug("%s(%s)\n", \_\_func\_\_, p);  
 size\_cmdline = memparse(p, &p);  
 if (\*p != '@') {  
  /\*  
  if base and limit are not assigned,  
  set limit to high memory bondary to use low memory.  
  \*/  
  limit\_cmdline = \_\_pa(high\_memory);  
  return 0;  
 }  
 base\_cmdline = memparse(p + 1, &p);  
 if (\*p != '-') {  
  limit\_cmdline = base\_cmdline + size\_cmdline;  
  return 0;  
 }  
 limit\_cmdline = memparse(p + 1, &p);  
  
 return 0;  
}  
early\_param("cma", early\_cma);

系统在启动的过程中会把cmdline里的nn, start, end传给函数dma\_contiguous\_reserve，流程如下：

setup\_arch--->arm64\_memblock\_init--->dma\_contiguous\_reserve->dma\_contiguous\_reserve\_area->cma\_declare\_contiguous



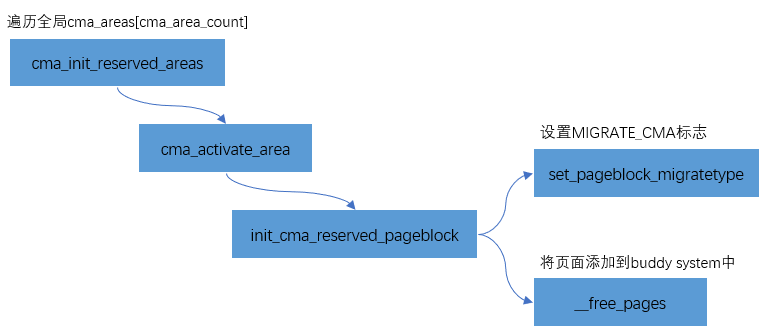
#### 将CMA区域添加到Buddy System

为了避免这块reserved的内存在不用时候的浪费，内存管理模块会将CMA区域添加到Buddy System中，用于可移动页面的分配和管理。CMA区域是通过cma\_init\_reserved\_areas接口来添加到Buddy System中的。

static int \_\_init cma\_init\_reserved\_areas(void)  
{  
 int i;  
  
 for (i = 0; i < cma\_area\_count; i++) {  
  int ret = cma\_activate\_area(&cma\_areas[i]);  
  
  if (ret)  
   return ret;  
 }  
  
 return 0;  
}  
core\_initcall(cma\_init\_reserved\_areas);

其实现比较简单，主要分为两步：

1. 把该页面设置为MIGRATE\_CMA标志
2. 通过\_\_free\_pages将页面添加到buddy system中



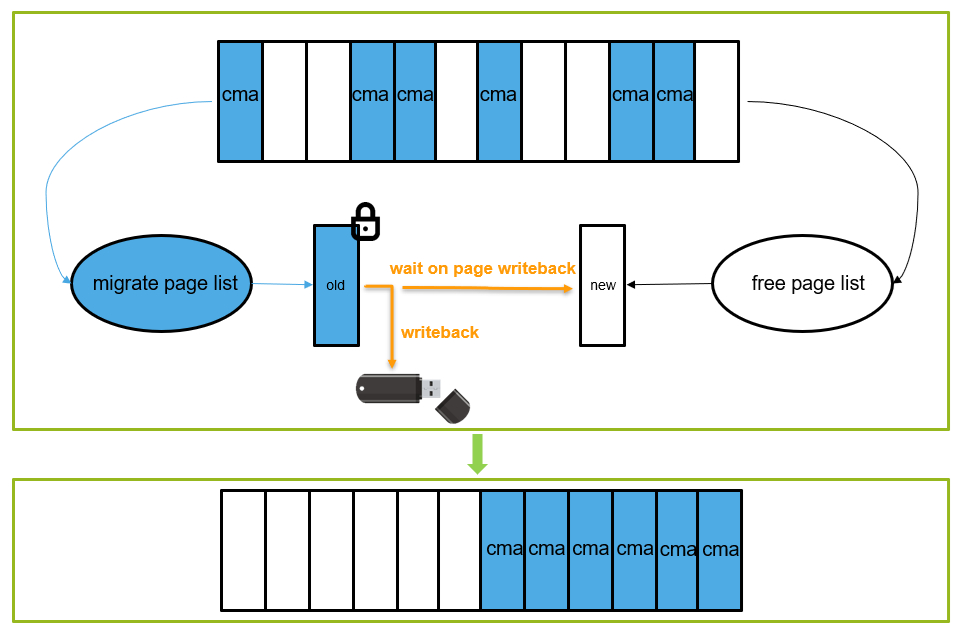
#### CMA分配

《没有IOMMU的DMA操作》里讲过，CMA是通过cma\_alloc分配的。cma\_alloc->alloc\_contig\_range(..., MIGRATE\_CMA,...)，向刚才释放给buddy system的MIGRATE\_CMA类型页面，重新“收集”过来。



用CMA的时候有一点需要注意：

也就是上图中黄色部分的判断。CMA内存在分配过程是一个比较“重”的操作，可能涉及页面迁移、页面回收等操作，因此不适合用于atomic context。比如之前遇到过一个问题，当内存不足的情况下，向U盘写数据的同时操作界面会出现卡顿的现象，这是因为CMA在迁移的过程中需要等待当前页面中的数据回写到U盘之后，才会进一步的规整为连续内存供gpu/display使用，从而出现卡顿的现象。



# 文件系统

## 打开、创建

### open

函数原型:

int open(const char \*pathname, int flags);

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

参数:

flags

必选项 O\_RDONLY, O\_WRONLY, O\_RDWR

可选项

创建文件: O\_CREAT

创建文件时检测文件是否存在: O\_EXCL

如果文件存在, 返回-1

必须与O\_CREAT一起使用

追加文件: O\_APPEND

文件截断: O\_TRUNC

设置非阻塞: O\_NONBLOCK

mode -- 指定0777 八进制数

最终权限: mode & ~umaks

创建新文件 creat

### fopen

打开文件 FILE\* fopen(const char\* path,const char\* type)

## 读

### read

函数原型: ssize\_t read(int fd, void \*buf, size\_t count);

参数:

fd -- open的返回值

buf - 缓冲区, 存放读取的数据

count -- 缓冲区的最大容量 sizeof(buf)

返回值:

-1: 失败

>0: 读出的字节数

=0: 文件读完了

### 库函数

读一个 getc(File\* file)、fgetc(FILE\* file)

读一行 fgets(buffer,len.pf)

读一块 fread(buffer,len,count,fp)

格式化读 fscnaf

### readv



readv函数将数据从fd读到分散的内存块中。即分散读。

## 写

### write

函数原型:

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

参数:

fd: 文件描述符, open 返回值

buf: 要往文件中写的数据

count: 有效数据的长度

返回值:

1: 失败

>0: 写入的字节数

### 库函数

写一个字节 putc、fputc

写一行 fputs(buffer,size,fp)

写一块 fwrite(buffer,size,count,fp)

格式化写 fprintf/sprintf

### writev



集中写

## 偏移

### lseek

函数原型: off\_t lseek(int fd, off\_t offset, int whence);

SEEK\_SET

SEEK\_CUR

SEEK\_END

使用:

文件指针移动到头部

lseek(fd, 0, SEEK\_SET);

获取文件指针当前的位置:

int len = lseek(fd, 0, SEEK\_CUR);

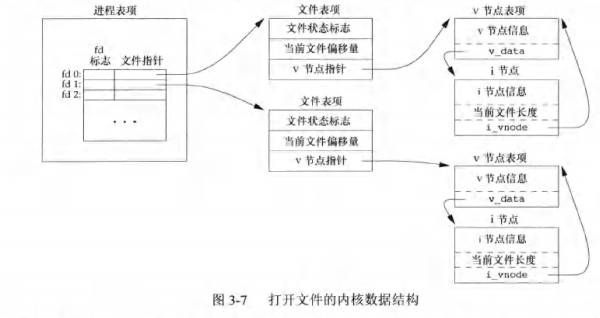
获取文件长度:

int len = lseek(fd, 0, SEEK\_END);

文件拓展

文件原大小100k， 拓展为1100k lseek(fd, 1000, SEE\_END);最后做一次写操作。

## 文件共享



每个进程PCB内有自身的文件描述符表。其表项包含file\*，指向文件表项。

文件表项包含i节点指针，指向该文件的元数据。

多个文件指针可以指向同一个文件表项（父子进程共享打开的文件描述符）

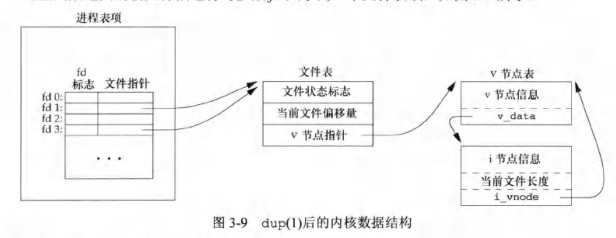
多个进程可以打开同一个文件（表现为多个不同的文件表项，指向同一个i节点）



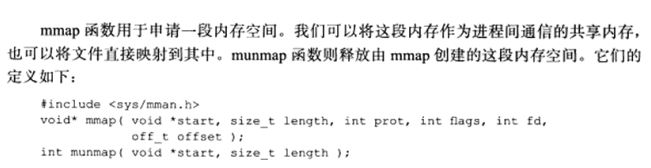
### 复制文件指针dup

dup拷贝fd指定的文件指针，返回指向该指针的fd。

dup2拷贝fd1到fd2，如果fd2已打开则关闭它。



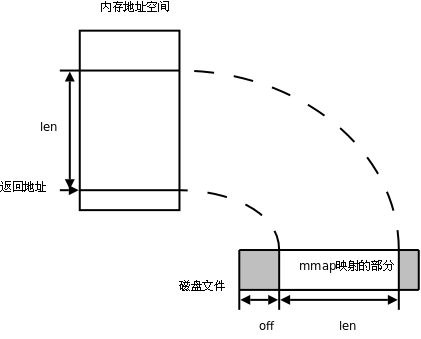
## 共享存储映射mmap



### 存储映射I/O

存储映射I/O (Memory-mapped I/O) 使一个磁盘文件与存储空间中的一个缓冲区相映射。于是当从缓冲区中取数据，就相当于读文件中的相应字节。于此类似，将数据存入缓冲区，则相应的字节就自动写入文件。这样，就可在不适用read和write函数的情况下，使用地址（指针）完成I/O操作。

使用这种方法，首先应通知内核，将一个指定文件映射到存储区域中。这个映射工作可以通过mmap函数来实现。



#### mmap函数

void \*mmap(void \*adrr, size\_t length, int prot, int flags, int fd, off\_t offset);

返回：成功：返回创建的映射区首地址；**失败：MAP\_FAILED宏**

参数：

addr: 建立映射区的首地址，由Linux内核指定。使用时，直接传递NULL

length： 欲创建映射区的大小

prot： 映射区权限PROT\_READ、PROT\_WRITE、PROT\_READ|PROT\_WRITE

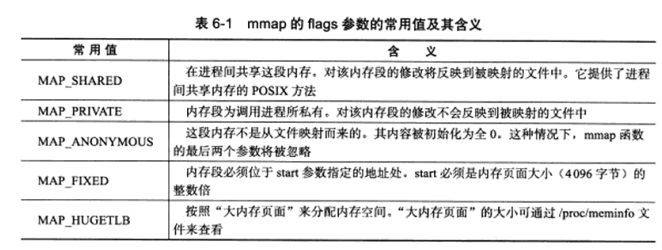
flags： 标志位参数(常用于设定更新物理区域、设置共享、创建匿名映射区)

MAP\_SHARED: 会将映射区所做的操作反映到物理设备（磁盘）上。

MAP\_PRIVATE: 映射区所做的修改不会反映到物理设备。

fd： 用来建立映射区的文件描述符

offset： 映射文件的偏移(4k的整数倍)



#### munmap函数

同malloc函数申请内存空间类似的，mmap建立的映射区在使用结束后也应调用类似free的函数来释放。

int munmap(void \*addr, size\_t length); 成功：0； 失败：-1

借鉴malloc和free函数原型，尝试装自定义函数smalloc，sfree来完成映射区的建立和释放。思考函数接口该如何设计？

【smalloc.c】

#### mmap注意事项

【mmap.c】

思考：

1. 可以open的时候O\_CREAT一个新文件来创建映射区吗?

2. 如果open时O\_RDONLY, mmap时PROT参数指定PROT\_READ|PROT\_WRITE会怎样？

3. 文件描述符先关闭，对mmap映射有没有影响？

4. 如果文件偏移量为1000会怎样？

5. 对mem越界操作会怎样？

6. 如果mem++，munmap可否成功？

7. mmap什么情况下会调用失败？

8. 如果不检测mmap的返回值，会怎样？

**总结：使用mmap时务必注意以下事项：**

1. 创建映射区的过程中，隐含着一次对映射文件的读操作。
2. 当MAP\_SHARED时，要求：映射区的权限应 <=文件打开的权限(出于对映射区的保护)。而MAP\_PRIVATE则无所谓，因为mmap中的权限是对内存的限制。
3. 映射区的释放与文件关闭无关。只要映射建立成功，文件可以立即关闭。
4. 特别注意，当映射文件大小为0时，不能创建映射区。所以：用于映射的文件必须要有实际大小！！ mmap使用时常常会出现总线错误，通常是由于共享文件存储空间大小引起的。
5. munmap传入的地址一定是mmap的返回地址。坚决杜绝指针++操作。
6. 如果文件偏移量必须为4K的整数倍
7. mmap创建映射区出错概率非常高，一定要检查返回值，确保映射区建立成功再进行后续操作。

### mmap父子进程通信

父子等有血缘关系的进程之间也可以通过mmap建立的映射区来完成数据通信。但相应的要在创建映射区的时候指定对应的标志位参数flags：

MAP\_PRIVATE: (私有映射) 父子进程各自独占映射区；

MAP\_SHARED: (共享映射) 父子进程共享映射区；

练习：父进程创建映射区，然后fork子进程，子进程修改映射区内容，而后，父进程读取映射区内容，查验是否共享。 【fork\_mmap.c】

结论：**父子进程共享：1. 打开的文件 2. mmap建立的映射区(但必须要使用MAP\_SHARED)**

### 匿名映射

通过使用我们发现，使用映射区来完成文件读写操作十分方便，父子进程间通信也较容易。但缺陷是，每次创建映射区一定要依赖一个文件才能实现。通常为了建立映射区要open一个temp文件，创建好了再unlink、close掉，比较麻烦。 可以直接使用匿名映射来代替。其实Linux系统给我们提供了创建匿名映射区的方法，无需依赖一个文件即可创建映射区。同样需要借助标志位参数flags来指定。

使用MAP\_ANONYMOUS (或MAP\_ANON)， 如:

int \*p = mmap(NULL, 4, PROT\_READ|PROT\_WRITE, MAP\_SHARED|MAP\_ANONYMOUS, -1, 0);

"4"随意举例，该位置表大小，可依实际需要填写。

【fork\_map\_anon\_linux.c】

需注意的是，MAP\_ANONYMOUS和MAP\_ANON这两个宏是Linux操作系统特有的宏。在类Unix系统中如无该宏定义，可使用如下两步来完成匿名映射区的建立。

① fd = open("/dev/zero", O\_RDWR);

② p = mmap(NULL, size, PROT\_READ|PROT\_WRITE, MMAP\_SHARED, fd, 0);

【fork\_map\_anon.c】

### mmap无血缘关系进程间通信

实质上mmap是内核借助文件帮我们创建了一个映射区，多个进程之间利用该映射区完成数据传递。由于内核空间多进程共享，因此无血缘关系的进程间也可以使用mmap来完成通信。只要设置相应的标志位参数flags即可。若想实现共享，当然应该使用MAP\_SHARED了。

## 高级IO

### fcntl文件控制

功能描述：根据文件描述词来操作文件的特性。

文件控制函数 fcntl -- file control

头文件：

#include <unistd.h>

#include <fcntl.h>

函数原型：

int fcntl(int fd, int cmd);

int fcntl(int fd, int cmd, long arg);

int fcntl(int fd, int cmd, struct flock \*lock);

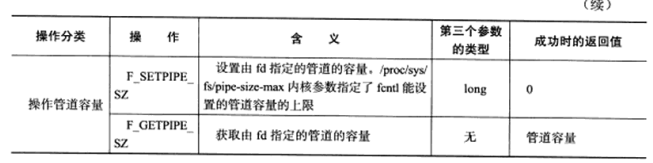
描述：

fcntl()针对(文件)描述符提供控制.参数fd是被参数cmd操作(如下面的描述)的描述符.

针对cmd的值,fcntl能够接受第三个参数（arg）

fcntl函数有5种功能：

1. 复制一个现有的描述符（cmd=F\_DUPFD）.
2. 获得／设置文件描述符标记(cmd=F\_GETFD或F\_SETFD).
3. 获得／设置文件状态标记(cmd=F\_GETFL或F\_SETFL).
4. 获得／设置异步I/O所有权(cmd=F\_GETOWN或F\_SETOWN).
5. 获得／设置记录锁(cmd=F\_GETLK,F\_SETLK或F\_SETLKW).



cmd 选项：

F\_DUPFD 返回一个如下描述的(文件)描述符:

（1）最小的大于或等于arg的一个可用的描述符

　　 （2）与原始操作符一样的某对象的引用

（3）如果对象是文件(file)的话,返回一个新的描述符,这个描述符与arg共享相同的偏移量(offset)

　　　　（4）相同的访问模式(读,写或读/写)

　　　　（5）相同的文件状态标志(如:两个文件描述符共享相同的状态标志)

1. 与新的文件描述符结合在一起的close-on-exec标志被设置成交叉式访问execve(2)的系统调用

F\_GETFD 取得与文件描述符fd联合close-on-exec标志,类似FD\_CLOEXEC.如果返回值和FD\_CLOEXEC进行与运算结果是0的话,文件保持交叉式访问exec(),否则如果通过exec运行的话,文件将被关闭(arg被忽略)

F\_SETFD 设置close-on-exec旗标。该旗标以参数arg的FD\_CLOEXEC位决定。

F\_GETFL 取得fd的文件状态标志,如同下面的描述一样(arg被忽略)

F\_SETFL 设置给arg描述符状态标志,可以更改的几个标志是：O\_APPEND， O\_NONBLOCK，O\_SYNC和O\_ASYNC。

F\_GETOWN 取得当前正在接收SIGIO或者SIGURG信号的进程id或进程组id,进程组id返回成负值(arg被忽略)

F\_SETOWN 设置将接收SIGIO和SIGURG信号的进程id或进程组id,进程组id通过提供负值的arg来说明,否则,arg将被认为是进程id

命令字(cmd)F\_GETFL和F\_SETFL的标志如下面的描述:

O\_NONBLOCK 非阻塞I/O;如果read(2)调用没有可读取的数据,或者如果write(2)操作将阻塞,read或write调用返回-1和EAGAIN错误

　　　　 O\_APPEND 强制每次写(write)操作都添加在文件大的末尾,相当于open(2)的O\_APPEND标志

O\_DIRECT 最小化或去掉reading和writing的缓存影响.系统将企图避免缓存你的读或写的数据. 如果不能够避免缓存,那么它将最小化已经被缓存了的数 据造成的影响.如果这个标志用的不够好,将大大的降低性能

O\_ASYNC 当I/O可用的时候,允许SIGIO信号发送到进程组,例如:当有数据可以读的时候

注意：在修改文件描述符标志或文件状态标志时必须谨慎，先要取得现在的标志值，然后按照希望修改它，最后设置新标志值。不能只是执行F\_SETFD或F\_SETFL命令，这样会关闭以前设置的标志位。

fcntl的返回值： 与命令有关。如果出错，所有命令都返回－1，如果成功则返回某个其他值。下列三个命令有特定返回值：F\_DUPFD,F\_GETFD,F\_GETFL以及F\_GETOWN。第一个返回新的文件描述符，第二个返回相应标志，最后一个返回一个正的进程ID或负的进程组ID。

一：第一种类似于dup操作，在这里不做举例。（fcnlt(oldfd, F\_DUPFD, 0) <==>dup2(oldfd, newfd)）

二：设置close-on-exec旗标

三：用命令F\_GETFL和F\_SETFL设置文件标志，比如阻塞与非阻塞

四：设置异步IO

五：设置获取记录锁

### stat文件属性

stat/lstat函数

获取文件属性

struct stat {

dev\_t st\_dev; //文件的设备编号

ino\_t st\_ino; //节点

mode\_t st\_mode; //文件的类型和存取的权限

nlink\_t st\_nlink; //连到该文件的硬连接数目，刚建立的文件值为1

uid\_t st\_uid; //用户ID

gid\_t st\_gid; //组ID

dev\_t st\_rdev; //(设备类型)若此文件为设备文件，则为其设备编号

off\_t st\_size; //文件字节数(文件大小)

blksize\_t st\_blksize; //块大小(文件系统的I/O 缓冲区大小)

blkcnt\_t st\_blocks; //块数

time\_t st\_atime; //最后一次访问时间

time\_t st\_mtime; //最后一次修改时间

time\_t st\_ctime; //最后一次改变时间(指属性)

};

st\_mode -- 16位整数

0-2 bit -- 其他人权限

S\_IROTH 00004 读权限

S\_IWOTH 00002 写权限

S\_IXOTH 00001 执行权限

S\_IRWXO 00007 掩码, 过滤 st\_mode中除其他人权限以外的信息

3-5 bit -- 所属组权限

S\_IRGRP 00040 读权限

S\_IWGRP 00020 写权限

S\_IXGRP 00010 执行权限

S\_IRWXG 00070 掩码, 过滤 st\_mode中除所属组权限以外的信息

6-8 bit -- 文件所有者权限

S\_IRUSR 00400 读权限

S\_IWUSR 00200 写权限

S\_IXUSR 00100 执行权限

S\_IRWXU 00700 掩码, 过滤 st\_mode中除文件所有者权限以外的信息

12-15 bit -- 文件类型

S\_IFSOCK 0140000 套接字

S\_IFLNK 0120000 符号链接（软链接）

S\_IFREG 0100000 普通文件

S\_IFBLK 0060000 块设备

S\_IFDIR 0040000 目录

S\_IFCHR 0020000 字符设备

S\_IFIFO 0010000 管道

S\_IFMT 0170000 掩码,过滤 st\_mode中除文件类型以外的信息(st\_mode & S\_IFMT) == S\_IFREG

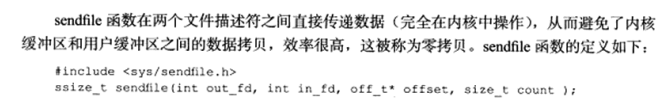
int stat(const char \*path, struct stat \*buf);

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

lstat读取的链接文件本身的属性

stat读取的链接文件指向的文件本身的属性

### sendfile传递数据



参数：

in\_fd是待读出内容的fd。必须是支持mmap的fd，即必须指向真实文件。

outfd是待写入内容的fd。必须是一个socket。

offset指定从读入文件流的哪个位置开始读，为空则从起始位置读。

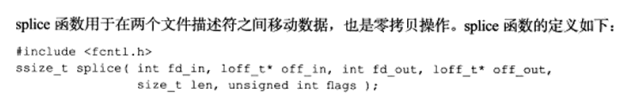
count指定传输的字节数。

返回值：

成功返回传输的字节数

失败返回-1设置errno

### splice移动数据



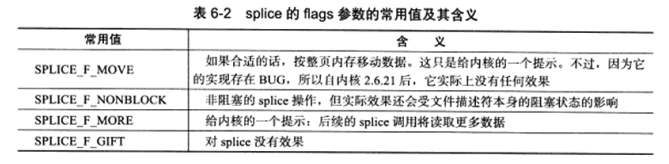
参数：

in\_fd是待读出内容的fd。如果fd\_in是管道，那么off\_in必须为NULL；如果不是管道，那么off\_in表示从哪里开始读，若为NULL则表示从当前位置开始读。

out\_fd是待写入内容的fd。如果fd\_out是管道，那么off\_out必须为NULL；如果不是管道，那么off\_out表示从哪里开始读，若为NULL则表示从当前位置开始读。

len表示移动数据的长度。

flag控制数据如何移动。



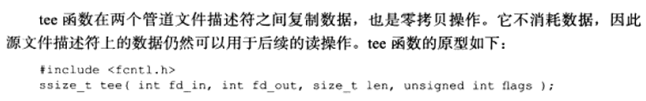
fd\_in和fd\_out至少有一个是管道文件描述符。

返回值：

成功返回传输的字节数

失败返回-1设置errno

### tee复制数据



参数：

in\_fd是待读出内容的fd。必须是支持mmap的fd，即必须指向真实文件。

outfd是待写入内容的fd。必须是一个socket。

offset指定从读入文件流的哪个位置开始读，为空则从起始位置读。

count指定传输的字节数。

返回值：

成功返回传输的字节数

失败返回-1设置errno

## 其他

### 符号链接

创建 symlink

读取 readlink

### 临时文件

创建临时目录 mkdtemp

创建临时文件 mkstemp

### 内存流

可以将缓冲区像文件一样操作。

FILE\* fmemopen(void\* buffer,size\_t len,const cyar\* type)

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IO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。

## 目录

### 文件重命名

int rename(const char \*oldpath, const char

\*newpath);

### 修改当前进程(应用程序)的路径 cd

int chdir(const char \*path);

参数: 切换的路径

### 获取当前进程的工作目录 pwd

char \*getcwd(char \*buf, size\_t size);

返回值:

成功: 当前的工作目录

失败: NULL

参数:

buf: 缓冲区, 存储当前的工作目录

size: 缓冲区大小

### 创建目录 mkdir

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

参数:

pathname: 创建的目录名

mode: 目录权限, 八进制的数, 实际权限:

mode & ~umask

### 删除一个空目录

int rmdir(const char \*pathname);

参数: 空目录的名字

### 打开一个目录

DIR \*opendir(const char \*name);

参数: 目录名

返回值: 指向目录的指针

FILE\* fp = fopen（）

fread（buf， len， len，fp）；

### 读目录

struct dirent

{

ino\_t d\_ino; // 此目录进入点的inode

ff\_t d\_off; // 目录文件开头至此目录进入点的位移signed short int d\_reclen; // d\_name 的长度, 不包含NULL 字符unsigned char d\_type; // d\_name 所指的文件类型

har d\_name[256]; // 文件名

};

d\_type

DT\_BLK - 块设备

DT\_CHR - 字符设备

DT\_DIR - 目录

DT\_LNK - 软连接

DT\_FIFO - 管道

DT\_REG - 普通文件

DT\_SOCK - 套接字

DT\_UNKNOWN - 未知

struct dirent \*readdir(DIR \*dirp);

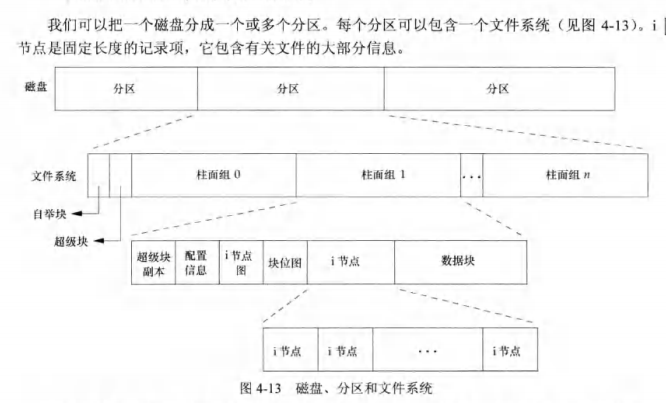
参数: opendir的返回值

返回值: 目录项结构体

### 关闭目录

int closedir(DIR \*dirp);

## 内核



Linux提供一个标准层，使得能在内核和每种fs之间交换数据。这个接口层成为虚拟文件系统。

上层关于内核表格的管理和数据结构，下层是与各具体文件系统相关的函数集合。

虚拟文件系统由层次较高的操作组成，实际在做事的是每种文件系统的函数。

### 文件系统数组

struct file\_system\_type file\_systems[]

现已转化为链表，以允许加载新的文件系统类型。

职责

* 管理缓冲区寄存器
* fcntl,iocntl
* pipe,fifo
* 管理文件和inode表
* 文件锁
* 将文件名（目录项）映射到inode
* select函数
* 提供信息——stat函数
* 加载和卸载文件系统
* exec函数加载二进制程序

struct file\_system\_type {

struct super\_block \*(\*read\_super) (struct super\_block \*, void \*, int);

char \*name;

int requires\_dev;

};

### 超级块super\_block

一个超级块对应一个文件系统

struct super\_block {

dev\_t s\_dev;

unsigned long s\_blocksize;

unsigned char s\_blocksize\_bits;

unsigned char s\_lock;

unsigned char s\_rd\_only;

unsigned char s\_dirt;

struct super\_operations \*s\_op;

unsigned long s\_flags;

unsigned long s\_magic;

unsigned long s\_time;

struct inode \* s\_covered;

struct inode \* s\_mounted;

struct wait\_queue \* s\_wait;

union {

struct minix\_sb\_info minix\_sb;

struct ext\_sb\_info ext\_sb;

struct ext2\_sb\_info ext2\_sb;

struct hpfs\_sb\_info hpfs\_sb;

struct msdos\_sb\_info msdos\_sb;

struct isofs\_sb\_info isofs\_sb;

struct nfs\_sb\_info nfs\_sb;

struct xiafs\_sb\_info xiafs\_sb;

struct sysv\_sb\_info sysv\_sb;

} u;

};

* 超级块代表了整个文件系统，超级块是文件系统的控制块，有整个文件系统信息，一个文件系统所有的inode都要连接到超级块上，可以说，一个超级块就代表了一个文件系统。
* s\_magic区别于其他文件系统的标识
* s\_covered应该表示其覆盖的i节点，s\_mounted应该表示挂载的i节点。
* s\_op表示对其进行操作的函数集

struct super\_operations {

void (\*read\_inode) (struct inode \*);

int (\*notify\_change) (int flags, struct inode \*);

void (\*write\_inode) (struct inode \*);

void (\*put\_inode) (struct inode \*);

void (\*put\_super) (struct super\_block \*);

void (\*write\_super) (struct super\_block \*);

void (\*statfs) (struct super\_block \*, struct statfs \*);

int (\*remount\_fs) (struct super\_block \*, int \*, char \*);

};

包括读i节点集，写inode,写super\_block，监听inode变化等。

struct super\_operations {

void (\*read\_inode) (struct inode \*);

int (\*notify\_change) (int flags, struct inode \*);

void (\*write\_inode) (struct inode \*);

void (\*put\_inode) (struct inode \*);

void (\*put\_super) (struct super\_block \*);

void (\*write\_super) (struct super\_block \*);

void (\*statfs) (struct super\_block \*, struct statfs \*);

int (\*remount\_fs) (struct super\_block \*, int \*, char \*);

};

包含超级块操作函数的指针，指向具体文件系统的函数

### 文件对象file

进程打开的文件对象

struct file {

mode\_t f\_mode;

dev\_t f\_rdev; /\* needed for /dev/tty \*/

off\_t f\_pos;

unsigned short f\_flags;

unsigned short f\_count;

unsigned short f\_reada;

struct file \*f\_next, \*f\_prev;

struct inode \* f\_inode;

struct file\_operations \* f\_op;

};

* 与进程关联。进程通过PCB的文件描述符表的表值file\*，找到该结构体。
* 这个结构体与进程对应，表示该进程打开的文件。
* f\_rdev表示其设备号，f\_mode表示文件打开标志，f\_pos表示当前文件偏移量。
* F\_next和f\_prev应该是其数据组织方式：链表。f\_inode是其指向的i节点指针。
* f\_op是对file结构体及其inode节点操作函数的函数指针集合。

struct file\_operations {

int (\*lseek) (struct inode \*, struct file \*, off\_t, int);

int (\*read) (struct inode \*, struct file \*, char \*, int);

int (\*write) (struct inode \*, struct file \*, char \*, int);

int (\*readdir) (struct inode \*, struct file \*, struct dirent \*, int);

int (\*select) (struct inode \*, struct file \*, int, select\_table \*);

int (\*ioctl) (struct inode \*, struct file \*, unsigned int, unsigned long);

int (\*mmap) (struct inode \*, struct file \*, unsigned long, size\_t, int, unsigned long);

int (\*open) (struct inode \*, struct file \*);

void (\*release) (struct inode \*, struct file \*);

int (\*fsync) (struct inode \*, struct file \*);

};

包括读写文件，读目录，打开和释放文件，mmap，ioctl等。

struct file\_operations {

int (\*lseek) (struct inode \*, struct file \*, off\_t, int);

int (\*read) (struct inode \*, struct file \*, char \*, int);

int (\*write) (struct inode \*, struct file \*, char \*, int);

int (\*readdir) (struct inode \*, struct file \*, struct dirent \*, int);

int (\*select) (struct inode \*, struct file \*, int, select\_table \*);

int (\*ioctl) (struct inode \*, struct file \*, unsigned int, unsigned long);

int (\*mmap) (struct inode \*, struct file \*, unsigned long, size\_t, int, unsigned long);

int (\*open) (struct inode \*, struct file \*);

void (\*release) (struct inode \*, struct file \*);

int (\*fsync) (struct inode \*, struct file \*);

};

包含文件操作函数的指针，指向具体文件系统的函数。

### 索引节点inode

保存元数据，映射一块磁盘区块。

struct inode {

dev\_t i\_dev;

unsigned long i\_ino;

umode\_t i\_mode;

nlink\_t i\_nlink;

uid\_t i\_uid;

gid\_t i\_gid;

dev\_t i\_rdev;

off\_t i\_size;

time\_t i\_atime;

time\_t i\_mtime;

time\_t i\_ctime;

unsigned long i\_blksize;

unsigned long i\_blocks;

struct semaphore i\_sem;

struct inode\_operations \* i\_op;

struct super\_block \* i\_sb;

struct wait\_queue \* i\_wait;

struct file\_lock \* i\_flock;

struct vm\_area\_struct \* i\_mmap;

struct inode \* i\_next, \* i\_prev;

struct inode \* i\_hash\_next, \* i\_hash\_prev;

struct inode \* i\_bound\_to, \* i\_bound\_by;

struct inode \* i\_mount;

struct socket \* i\_socket;

unsigned short i\_count;

unsigned short i\_flags;

unsigned char i\_lock;

unsigned char i\_dirt;

unsigned char i\_pipe;

unsigned char i\_seek;

unsigned char i\_update;

union {

struct pipe\_inode\_info pipe\_i;

struct minix\_inode\_info minix\_i;

struct ext\_inode\_info ext\_i;

struct ext2\_inode\_info ext2\_i;

struct hpfs\_inode\_info hpfs\_i;

struct msdos\_inode\_info msdos\_i;

struct iso\_inode\_info isofs\_i;

struct nfs\_inode\_info nfs\_i;

struct xiafs\_inode\_info xiafs\_i;

struct sysv\_inode\_info sysv\_i;

} u;

};

* 保存的其实是实际的数据的一些信息，这些信息称为“元数据”(也就是对文件属性的描述)。例如：文件大小，设备标识符，用户标识符，用户组标识符，文件模式，扩展属性，文件读取或修改的时间戳，链接数量，指向存储该内容的磁盘区块的指针，文件分类等等。( 注意数据分成：元数据+数据本身 )
* 同时注意：inode有两种，一种是虚拟文件系统的inode，一种是具体文件系统的inode。前者在内存中，后者在磁盘中。所以每次其实是将磁盘中的inode调进填充内存中的inode，这样才是算使用了磁盘文件inode。
* 注意inode怎样生成的：每个inode节点的大小，一般是128字节或256字节。inode节点的总数，在格式化时就给定(现代OS可以动态变化)，一般每2KB就设置一个inode。一般文件系统中很少有文件小于2KB的，所以预定按照2KB分，一般inode是用不完的。所以inode在文件系统安装的时候会有一个默认数量，后期会根据实际的需要发生变化。
* 注意inode号：inode号是唯一的，表示不同的文件。其实在Linux内部的时候，访问文件都是通过inode号来进行的，所谓文件名仅仅是给用户容易使用的。当我们打开一个文件的时候，首先，系统找到这个文件名对应的inode号；然后，通过inode号，得到inode信息，最后，由inode找到文件数据所在的block，现在可以处理文件数据了。
* inode和文件的关系：当创建一个文件的时候，就给文件分配了一个inode。一个inode只对应一个实际文件，一个文件也会只有一个inode。inodes最大数量就是文件的最大数量。
* 保存文件的管理信息（文件属性），在内核通过inoe\*访问。
* 进程通过fd找到struct file\*，再通过struct file\*找到struct iode\*
* 包含i\_dev文件的设备号，i\_ino i节点号，i\_rdev（若设备文件）的设备编号
* 包含ls -l能查到的：i\_mode权限和文件类型，i\_nlink硬连接计数，i\_gid所属组ID，i\_size文件字节大小，i\_atime访问时间，i\_mtime修改时间，i\_ctime属性修改时间，i\_blksize块大小，i\_blocks块数。
* 包含i\_sem信号量，i\_flock文件锁（应该用于互斥读写该文件），i\_mmap共享内存。
* i\_count引用计数
* 一系列inode\*实现数据结构（或许是链表？）
* i\_op是对inode节点操作函数的函数指针集合。

struct inode\_operations {

struct file\_operations \* default\_file\_ops;

int (\*create) (struct inode \*,const char \*,int,int,struct inode \*\*);

int (\*lookup) (struct inode \*,const char \*,int,struct inode \*\*);

int (\*link) (struct inode \*,struct inode \*,const char \*,int);

int (\*unlink) (struct inode \*,const char \*,int);

int (\*symlink) (struct inode \*,const char \*,int,const char \*);

int (\*mkdir) (struct inode \*,const char \*,int,int);

int (\*rmdir) (struct inode \*,const char \*,int);

int (\*mknod) (struct inode \*,const char \*,int,int,int);

int (\*rename) (struct inode \*,const char \*,int,struct inode \*,const char \*,int);

int (\*readlink) (struct inode \*,char \*,int);

int (\*follow\_link) (struct inode \*,struct inode \*,int,int,struct inode \*\*);

int (\*bmap) (struct inode \*,int);

void (\*truncate) (struct inode \*);

int (\*permission) (struct inode \*, int);

};

包含inode操作函数的指针，指向具体文件系统的函数。

* 包括创建、创建硬链接、断开硬链接、创建符号链接、指定文件大小、访问权限检查、读符号链接等。
* 还有些感觉像目录和目录项的函数：mkdir,rmdir,rename等。

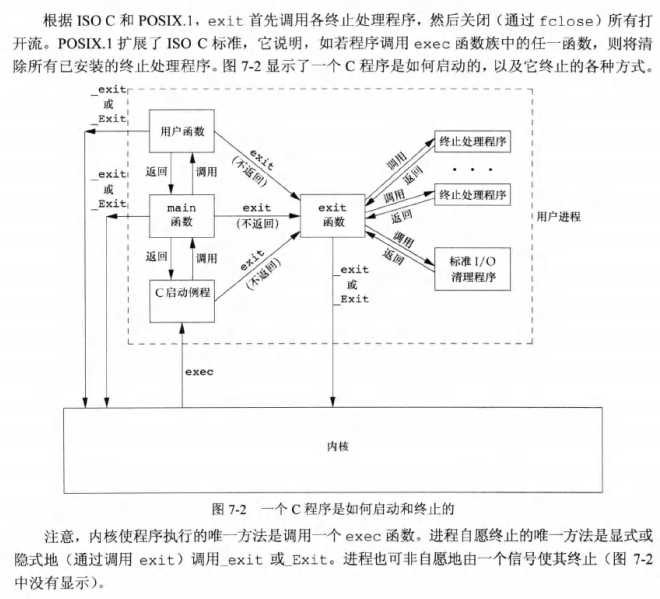
### 挂载卸载

Linux无论几个分区，他只有一个根目录。他的每个分区都是用来组成整个文件系统的一部分，他采用载入的方法将分区和文件系统关联。

一个文件系统中包含一整套目录和文件，将一个目录如/mnt/Space和一个分区联系起来。

# 进程调度

## 进程运行



可以通过 int atexit(void (\*func)(void)) 注册终止处理程序，这些函数由exit自动调用。

## 进程相关概念

### 程序和进程

程序，是指编译好的二进制文件，在磁盘上，不占用系统资源(cpu、内存、打开的文件、设备、锁....)

进程，是一个抽象的概念，与操作系统原理联系紧密。进程是活跃的程序，占用系统资源。在内存中执行。(程序运行起来，产生一个进程)

程序 → 剧本(纸) 进程 → 戏(舞台、演员、灯光、道具...)

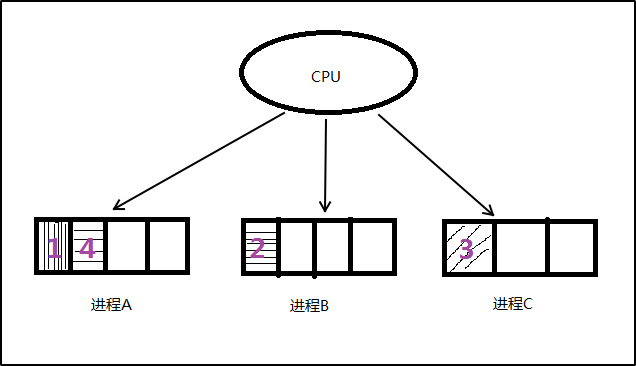
同一个剧本可以在多个舞台同时上演。同样，同一个程序也可以加载为不同的进程(彼此之间互不影响)

如：同时开两个终端。各自都有一个bash但彼此ID不同。

### 并发

并发，在[操作系统](http://baike.baidu.com/view/880.htm" \t "_blank)中，一个时间段中有多个进程都处于已启动运行到运行完毕之间的状态。但，任一个时刻点上仍只有一个进程在运行。

例如，当下，我们使用计算机时可以边听音乐边聊天边上网。 若笼统的将他们均看做一个进程的话，为什么可以同时运行呢，因为并发。



分时复用cpu

### 单道程序设计

所有进程一个一个排对执行。若A阻塞，B只能等待，即使CPU处于空闲状态。而在人机交互时阻塞的出现时必然的。所有这种模型在系统资源利用上及其不合理，在计算机发展历史上存在不久，大部分便被淘汰了。

### 多道程序设计

在计算机内存中同时存放几道相互独立的程序，它们在管理程序控制之下，相互穿插的运行。多道程序设计必须有[硬件基础](http://baike.baidu.com/view/4419390.htm" \t "_blank)作为保证。

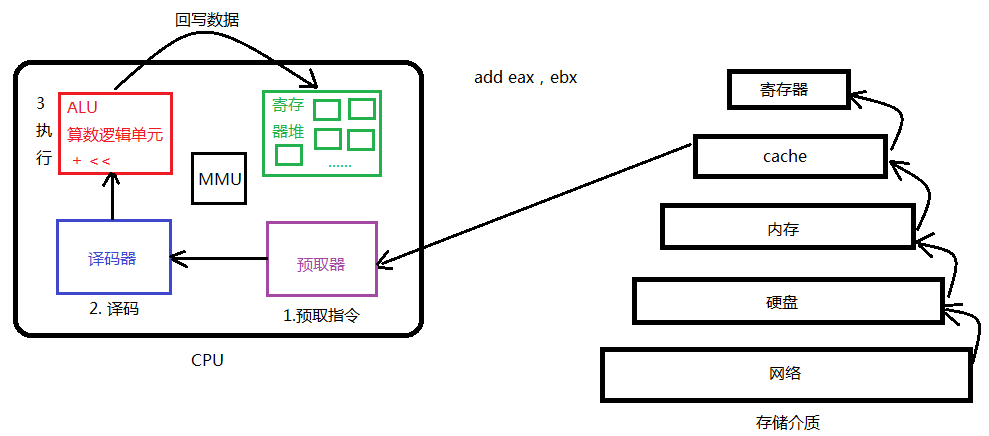
**时钟中断**即为多道程序设计模型的理论基础。 并发时，任意进程在执行期间都不希望放弃cpu。因此系统需要一种强制让进程让出cpu资源的手段。时钟中断有硬件基础作为保障，对进程而言不可抗拒。 操作系统中的中断处理函数，来负责调度程序执行。

在多道程序设计模型中，多个进程轮流使用CPU (分时复用CPU资源)。而当下常见CPU为纳秒级，1秒可以执行大约10亿条指令。由于人眼的反应速度是毫秒级，所以看似同时在运行。

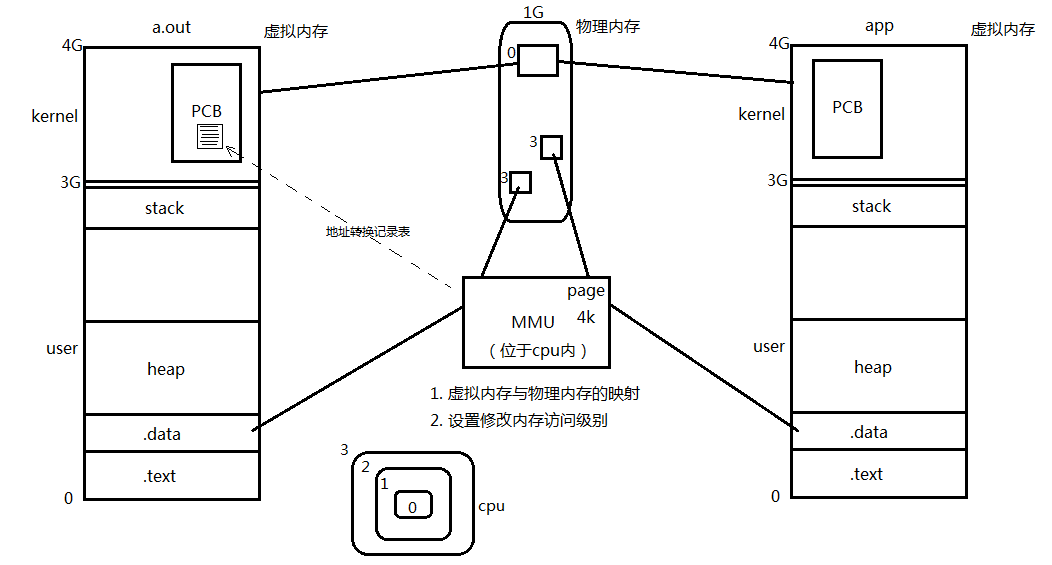
1s = 1000ms, 1ms = 1000us, 1us = 1000ns 1000000000

实质上，并发是宏观并行，微观串行！ -----推动了计算机蓬勃发展，将人类引入了多媒体时代。

### CPU和MMU



中央处理器(CPU)



内存管理单元MMU

### 进程控制块PCB

我们知道，每个进程在内核中都有一个进程控制块（PCB）来维护进程相关的信息，Linux内核的进程控制块是task\_struct结构体。

/usr/src/linux-headers-3.16.0-30/include/linux/sched.h文件中可以查看struct task\_struct 结构体定义。其内部成员有很多，我们重点掌握以下部分即可：

\* 进程id。系统中每个进程有唯一的id，在C语言中用pid\_t类型表示，其实就是一个非负整数。

\* 进程的状态，有就绪、运行、挂起、停止等状态。

\* 进程切换时需要保存和恢复的一些CPU寄存器。

\* 描述虚拟地址空间的信息。

\* 描述控制终端的信息。

\* 当前工作目录（Current Working Directory）。

\* umask掩码。

\* 文件描述符表，包含很多指向file结构体的指针。

\* 和信号相关的信息。

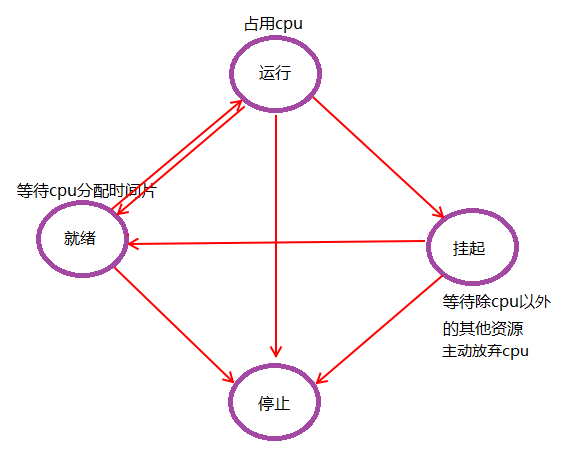
\* 用户id和组id。

\* 会话（Session）和进程组。

\* 进程可以使用的资源上限（Resource Limit）。

### 进程状态

进程基本的状态有5种。分别为初始态，就绪态，[运行态](http://baike.baidu.com/subview/1730379/1730379.htm" \t "_blank)，挂起态与终止态。其中初始态为进程准备阶段，常与就绪态结合来看。



# 环境变量：

环境变量，是指在[操作系统](http://baike.baidu.com/view/880.htm" \t "_blank)中用来指定操作系统运行环境的一些参数。通常具备以下特征：

① 字符串(本质) ② 有统一的格式：名=值[:值] ③ 值用来描述进程环境信息。

存储形式：与命令行参数类似。char \*[]数组，数组名environ，内部存储字符串，NULL作为哨兵结尾。

使用形式：与命令行参数类似。

加载位置：与命令行参数类似。位于用户区，高于stack的起始位置。

引入环境变量表：须声明环境变量。extern char \*\* environ;

练习：打印当前进程的所有环境变量。 【environ.c】

### 常见环境变量

按照惯例，环境变量字符串都是name=value这样的形式，大多数name由大写字母加下划线组成，一般把name的部分叫做环境变量，value的部分则是环境变量的值。环境变量定义了进程的运行环境，一些比较重要的环境变量的含义如下：

#### PATH

可执行文件的搜索路径。ls命令也是一个程序，执行它不需要提供完整的路径名/bin/ls，然而通常我们执行当前目录下的程序a.out却需要提供完整的路径名./a.out，这是因为PATH环境变量的值里面包含了ls命令所在的目录/bin，却不包含a.out所在的目录。PATH环境变量的值可以包含多个目录，用:号隔开。在Shell中用echo命令可以查看这个环境变量的值：

$ echo $PATH

#### SHELL

当前Shell，它的值通常是/bin/bash。

#### TERM

当前终端类型，在图形界面终端下它的值通常是xterm，终端类型决定了一些程序的输出显示方式，比如图形界面终端可以显示汉字，而字符终端一般不行。

#### LANG

语言和locale，决定了字符编码以及时间、货币等信息的显示格式。

#### HOME

当前用户主目录的路径，很多程序需要在主目录下保存配置文件，使得每个用户在运行该程序时都有自己的一套配置。

### getenv函数

获取环境变量值

char \*getenv(const char \*name); 成功：返回环境变量的值；失败：NULL (name不存在)

练习：编程实现getenv函数。 【getenv.c】

### setenv函数

设置环境变量的值

int setenv(const char \*name, const char \*value, int overwrite); 成功：0；失败：-1

参数overwrite取值： 1：覆盖原环境变量

0：不覆盖。(该参数常用于设置新环境变量，如：ABC = haha-day-night)

### unsetenv函数

删除环境变量name的定义

int unsetenv(const char \*name); 成功：0；失败：-1

注意事项：name不存在仍返回0(成功)，当name命名为"ABC="时则会出错。

# 进程控制

### fork函数

创建一个子进程。

pid\_t fork(void); 失败返回-1；成功返回：① 父进程返回子进程的ID(非负) ②子进程返回 0

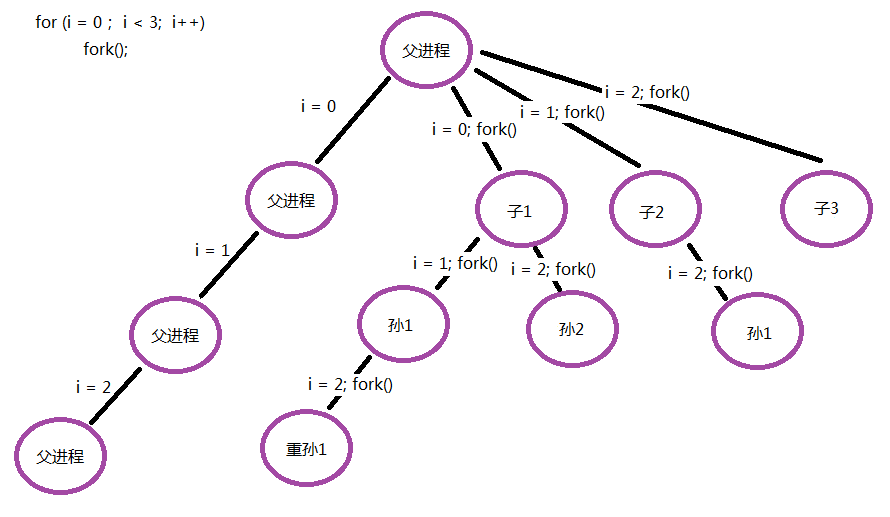
pid\_t类型表示进程ID，但为了表示-1，它是有符号整型。(0不是有效进程ID，init最小，为1)

注意返回值，不是fork函数能返回两个值，而是fork后，fork函数变为两个，父子需【各自】返回一个。

#### 循环创建n个子进程

一次fork函数调用可以创建一个子进程。那么创建N个子进程应该怎样实现呢？

简单想，for(i = 0; i < n; i++) { fork() } 即可。但这样创建的是N个子进程吗？



循环创建N个子进程

从上图我们可以很清晰的看到，当n为3时候，循环创建了(2^n)-1个子进程，而不是N的子进程。需要在循环的过程，保证子进程不再执行fork ，因此当(fork() == 0)时，子进程应该立即break;才正确。

练习：通过命令行参数指定创建进程的个数，每个进程休眠1S打印自己是第几个被创建的进程。如：第1个子进程休眠0秒打印：“我是第1个子进程”；第2个进程休眠1秒打印：“我是第2个子进程”；第3个进程休眠2秒打印：“我是第3个子进程”。 【fork1.c】

通过该练习**掌握框架**：循环创建n个子进程，使用循环因子i对创建的子进程加以区分。

### getpid函数

获取当前进程ID

pid\_t getpid(void);

### getppid函数

获取当前进程的父进程ID

pid\_t getppid(void);

区分一个函数是“系统函数”还是“库函数”依据：

1. 是否访问内核数据结构

② 是否访问外部硬件资源 二者有任一 → 系统函数；二者均无 → 库函数

### getuid函数

获取当前进程实际用户ID

uid\_t getuid(void);

获取当前进程有效用户ID

uid\_t geteuid(void);

### getgid函数

获取当前进程使用用户组ID

gid\_t getgid(void);

获取当前进程有效用户组ID

gid\_t getegid(void);

### 进程共享

父子进程之间在fork后。有哪些相同，那些相异之处呢？

刚fork之后：

父子相同处: 全局变量、.data、.text、栈、堆、环境变量、用户ID、宿主目录、进程工作目录、信号处理方式...

父子不同处: 1.进程ID 2.fork返回值 3.父进程ID 4.进程运行时间 5.闹钟(定时器) 6.未决信号集

似乎，子进程复制了父进程0-3G用户空间内容，以及父进程的PCB，但pid不同。真的每fork一个子进程都要将父进程的0-3G地址空间完全拷贝一份，然后在映射至物理内存吗？

当然不是!父子进程间遵循**读时共享写时复制**的原则。这样设计，无论子进程执行父进程的逻辑还是执行自己的逻辑都能节省内存开销。

练习：编写程序测试，父子进程是否共享全局变。 【fork\_shared.c】

重点注意！躲避父子进程共享全局变量的知识误区！

【重点】：父子进程共享：1. 文件描述符(打开文件的结构体) 2. mmap建立的映射区 (进程间通信详解)

特别的，fork之后父进程先执行还是子进程先执行不确定。取决于内核所使用的调度算法。

### gdb调试

使用gdb调试的时候，gdb只能跟踪一个进程。可以在fork函数调用之前，通过指令设置gdb调试工具跟踪父进程或者是跟踪子进程。默认跟踪父进程。

set follow-fork-mode child 命令设置gdb在fork之后跟踪子进程。

set follow-fork-mode parent 设置跟踪父进程。

注意，一定要在fork函数调用之前设置才有效。 【follow\_fork.c】

## exec函数族

fork创建子进程后执行的是和父进程相同的程序（但有可能执行不同的代码分支），子进程往往要调用一种exec函数以执行另一个程序。当进程调用一种exec函数时，该进程的用户空间代码和数据完全被新程序替换，从新程序的启动例程开始执行。调用exec并不创建新进程，所以调用exec前后该进程的id并未改变。

将当前进程的.text、.data替换为所要加载的程序的.text、.data，然后让进程从新的.text第一条指令开始执行，但进程ID不变，换核不换壳。

其实有六种以exec开头的函数，统称exec函数：

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

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

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

int execv(const char \*path, char \*const argv[]);

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

int execve(const char \*path, char \*const argv[], char \*const envp[]);

### execlp函数

加载一个进程，借助PATH环境变量

int execlp(const char \*file, const char \*arg, ...); 成功：无返回；失败：-1

参数1：要加载的程序的名字。该函数需要配合PATH环境变量来使用，当PATH中所有目录搜索后没有参数1则出错返回。

该函数通常用来调用系统程序。如：ls、date、cp、cat等命令。

### execl函数

加载一个进程， 通过 路径+程序名 来加载。

int execl(const char \*path, const char \*arg, ...); 成功：无返回；失败：-1

对比execlp，如加载"ls"命令带有-l，-F参数

execlp("ls", "ls", "-l", "-F", NULL); 使用程序名在PATH中搜索。

execl("/bin/ls", "ls", "-l", "-F", NULL); 使用参数1给出的绝对路径搜索。

### execvp函数

加载一个进程，使用自定义环境变量env

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

变参形式： ①... ② argv[] (main函数也是变参函数，形式上等同于 int main(int argc, char \*argv0, ...))

变参终止条件：① NULL结尾 ② 固参指定

execvp与execlp参数形式不同，原理一致。

练习：将当前系统中的进程信息，打印到文件中。 【exec\_ps.c】

### exec函数族一般规律

exec函数一旦调用成功即执行新的程序，不返回。**只有失败才返回，错误值-1**。所以通常我们直接在exec函数调用后直接调用perror()和exit()，无需if判断。

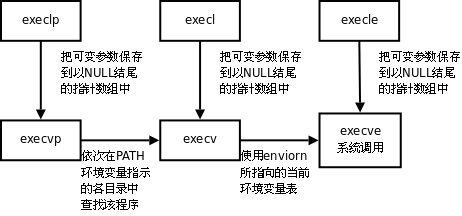
l (list) 命令行参数列表

p (path) 搜素file时使用path变量

v (vector) 使用命令行参数数组

e (environment) 使用环境变量数组,不使用进程原有的环境变量，设置新加载程序运行的环境变量

事实上，只有execve是真正的系统调用，其它五个函数最终都调用execve，所以execve在man手册第2节，其它函数在man手册第3节。这些函数之间的关系如下图所示。



exec函数族

## 回收子进程

### 孤儿进程

孤儿进程: 父进程先于子进程结束，则子进程成为孤儿进程，子进程的父进程成为init进程，称为init进程领养孤儿进程。

【orphan.c】

### 僵尸进程

僵尸进程: 进程终止，父进程尚未回收，子进程残留资源（PCB）存放于内核中，变成僵尸（Zombie）进程。

【zoom .c】

特别注意，僵尸进程是不能使用kill命令清除掉的。因为kill命令只是用来终止进程的，而僵尸进程已经终止。思考！用什么办法可清除掉僵尸进程呢？

### wait函数

一个进程在终止时会关闭所有文件描述符，释放在用户空间分配的内存，但它的PCB还保留着，内核在其中保存了一些信息：如果是正常终止则保存着退出状态，如果是异常终止则保存着导致该进程终止的信号是哪个。这个进程的父进程可以调用wait或waitpid获取这些信息，然后彻底清除掉这个进程。我们知道一个进程的退出状态可以在Shell中用特殊变量$?查看，因为Shell是它的父进程，当它终止时Shell调用wait或waitpid得到它的退出状态同时彻底清除掉这个进程。

父进程调用wait函数可以回收子进程终止信息。该函数有三个功能：

① 阻塞等待子进程退出

② 回收子进程残留资源

③ 获取子进程结束状态(退出原因)。

**pid\_t wait(int \*status);** 成功：清理掉的子进程ID；失败：-1 (没有子进程)

当进程终止时，操作系统的隐式回收机制会：1.关闭所有文件描述符 2. 释放用户空间分配的内存。内核的PCB仍存在。其中保存该进程的退出状态。(正常终止→退出值；异常终止→终止信号)

可使用wait函数传出参数status来保存进程的退出状态。借助宏函数来进一步判断进程终止的具体原因。宏函数可分为如下三组：

1. WIFEXITED(status) 为非0 → 进程正常结束

WEXITSTATUS(status) 如上宏为真，使用此宏 → 获取进程退出状态 (exit的参数)

2. WIFSIGNALED(status) 为非0 → 进程异常终止

WTERMSIG(status) 如上宏为真，使用此宏 → 取得使进程终止的那个信号的编号。

\*3. WIFSTOPPED(status) 为非0 → 进程处于暂停状态

WSTOPSIG(status) 如上宏为真，使用此宏 → 取得使进程暂停的那个信号的编号。

WIFCONTINUED(status) 为真 → 进程暂停后已经继续运行

【wait1.c、wait2.c】

### waitpid函数

作用同wait，但可指定pid进程清理，可以不阻塞。

**pid\_t waitpid(pid\_t pid, int \*status, in options);** 成功：返回清理掉的子进程ID；失败：-1(无子进程)

特殊参数和返回情况：

参数pid：

**> 0 回收指定ID的子进程**

**-1 回收任意子进程（相当于wait）**

0 回收和当前调用waitpid一个组的所有子进程

< -1 回收指定进程组内的任意子进程

返回0：参3为WNOHANG，且子进程正在运行。

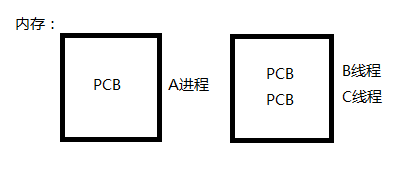
注意：**一次wait或waitpid调用只能清理一个子进程**，清理多个子进程应使用循环。 【waitpid.c】

作业：父进程fork 3 个子进程，三个子进程一个调用ps命令， 一个调用自定义程序1(正常)，一个调用自定义程序2(会出段错误)。父进程使用waitpid对其子进程进行回收。

# 线程概念

### 什么是线程

LWP：light weight process 轻量级的进程，本质仍是进程(在Linux环境下)

 进程：独立地址空间，拥有PCB

线程：也有PCB，但没有独立的地址空间(共享)

区别：在于是否共享地址空间。 独居(进程)；合租(线程)。

Linux下： 线程：最小的执行单位

进程：最小分配资源单位，可看成是只有一个线程的进程。

### Linux内核线程实现原理

类Unix系统中，早期是没有“线程”概念的，80年代才引入，借助进程机制实现出了线程的概念。因此在这类系统中，进程和线程关系密切。

1. 轻量级进程(light-weight process)，也有PCB，创建线程使用的底层函数和进程一样，都是clone

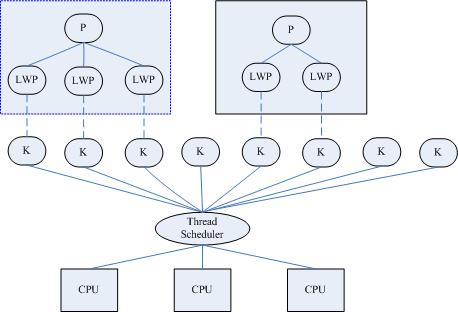
2. 从内核里看进程和线程是一样的，都有各自不同的PCB，但是PCB中指向内存资源的三级页表是相同的

3. 进程可以蜕变成线程

4. 线程可看做寄存器和栈的集合

5. 在linux下，线程最是小的执行单位；进程是最小的分配资源单位

察看LWP号：ps –Lf pid 查看指定线程的lwp号。



三级映射：进程PCB --> 页目录(可看成数组，首地址位于PCB中) --> 页表 --> 物理页面 --> 内存单元

参考：《Linux内核源代码情景分析》 ----毛德操

对于进程来说，相同的地址(同一个虚拟地址)在不同的进程中，反复使用而不冲突。原因是他们虽虚拟址一样，但，页目录、页表、物理页面各不相同。相同的虚拟址，映射到不同的物理页面内存单元，最终访问不同的物理页面。

但！线程不同！两个线程具有各自独立的PCB，但共享同一个页目录，也就共享同一个页表和物理页面。所以两个PCB共享一个地址空间。

实际上，无论是创建进程的fork，还是创建线程的pthread\_create，底层实现都是调用同一个内核函数clone。

如果复制对方的地址空间，那么就产出一个“进程”；如果共享对方的地址空间，就产生一个“线程”。

因此：Linux内核是不区分进程和线程的。只在用户层面上进行区分。所以，线程所有操作函数 pthread\_\* 是库函数，而非系统调用。

### 线程共享资源

1.文件描述符表

2.每种信号的处理方式

3.当前工作目录

4.用户ID和组ID

5.内存地址空间 (.text/.data/.bss/heap/共享库)

### 线程非共享资源

1.线程id

2.处理器现场和栈指针(内核栈)

3.独立的栈空间(用户空间栈)

4.errno变量

5.信号屏蔽字

6.调度优先级

### 线程优、缺点

优点： 1. 提高程序并发性 2. 开销小 3. 数据通信、共享数据方便

缺点： 1. 库函数，不稳定 2. 调试、编写困难、gdb不支持 3. 对信号支持不好

优点相对突出，缺点均不是硬伤。Linux下由于实现方法导致进程、线程差别不是很大。

## 线程控制原语

### pthread\_self函数

获取线程ID。其作用对应进程中 getpid() 函数。

pthread\_t pthread\_self(void); 返回值：成功：0； 失败：无！

线程ID：pthread\_t类型，本质：在Linux下为无符号整数(%lu)，其他系统中可能是结构体实现

线程ID是进程内部，识别标志。(两个进程间，线程ID允许相同)

注意：不应使用全局变量 pthread\_t tid，在子线程中通过pthread\_create传出参数来获取线程ID，而应使用pthread\_self。

### pthread\_create函数

创建一个新线程。 其作用，对应进程中fork() 函数。

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine) (void \*), void \*arg);

返回值：成功：0； 失败：错误号 -----Linux环境下，所有线程特点，失败均直接返回错误号。

参数：

pthread\_t：当前Linux中可理解为：typedef unsigned long int pthread\_t;

参数1：传出参数，保存系统为我们分配好的线程ID

参数2：通常传NULL，表示使用线程默认属性。若想使用具体属性也可以修改该参数。

参数3：函数指针，指向线程主函数(线程体)，该函数运行结束，则线程结束。

参数4：线程主函数执行期间所使用的参数。

在一个线程中调用pthread\_create()创建新的线程后，当前线程从pthread\_create()返回继续往下执行，而新的线程所执行的代码由我们传给pthread\_create的函数指针start\_routine决定。start\_routine函数接收一个参数，是通过pthread\_create的arg参数传递给它的，该参数的类型为void \*，这个指针按什么类型解释由调用者自己定义。start\_routine的返回值类型也是void \*，这个指针的含义同样由调用者自己定义。start\_routine返回时，这个线程就退出了，其它线程可以调用pthread\_join得到start\_routine的返回值，类似于父进程调用wait(2)得到子进程的退出状态，稍后详细介绍pthread\_join。

pthread\_create成功返回后，新创建的线程的id被填写到thread参数所指向的内存单元。我们知道进程id的类型是pid\_t，每个进程的id在整个系统中是唯一的，调用getpid(2)可以获得当前进程的id，是一个正整数值。线程id的类型是thread\_t，它只在当前进程中保证是唯一的，在不同的系统中thread\_t这个类型有不同的实现，它可能是一个整数值，也可能是一个结构体，也可能是一个地址，所以不能简单地当成整数用printf打印，调用pthread\_self(3)可以获得当前线程的id。

attr参数表示线程属性，本节不深入讨论线程属性，所有代码例子都传NULL给attr参数，表示线程属性取缺省值，感兴趣的读者可以参考APUE。

【练习】：创建一个新线程，打印线程ID。注意：链接线程库 -lpthread 【pthrd\_crt.c】

由于pthread\_create的错误码不保存在errno中，因此不能直接用perror(3)打印错误信息，可以先用strerror(3)把错误码转换成错误信息再打印。如果任意一个线程调用了exit或\_exit，则整个进程的所有线程都终止，由于从main函数return也相当于调用exit，为了防止新创建的线程还没有得到执行就终止，我们在main函数return之前延时1秒，这只是一种权宜之计，即使主线程等待1秒，内核也不一定会调度新创建的线程执行，下一节我们会看到更好的办法。

【练习】：循环创建多个线程，每个线程打印自己是第几个被创建的线程。(类似于进程循环创建子进程) 【more\_pthrd.c】

拓展思考：将pthread\_create函数参4修改为(void \*)&i, 将线程主函数内改为 i=\*((int \*)arg) 是否可以？

#### 线程与共享

**线程间共享全局变量！**

【**牢记**】：线程默认共享数据段、代码段等地址空间，常用的是全局变量。而进程不共享全局变量，只能借助mmap。

【练习】：设计程序，验证线程之间共享全局数据。 【glb\_var\_pthrd.c】

### pthread\_exit函数

将单个线程退出

void pthread\_exit(void \*retval); 参数：retval表示线程退出状态，通常传NULL

思考：使用exit将指定线程退出，可以吗？ 【pthrd\_exit.c】

结论：线程中，禁止使用exit函数，会导致进程内所有线程全部退出。

在不添加sleep控制输出顺序的情况下。pthread\_create在循环中，几乎瞬间创建5个线程，但只有第1个线程有机会输出（或者第2个也有，也可能没有，取决于内核调度）如果第3个线程执行了exit，将整个进程退出了，所以全部线程退出了。

所以，多线程环境中，应尽量少用，或者不使用exit函数，取而代之使用pthread\_exit函数，将单个线程退出。任何线程里exit导致进程退出，其他线程未工作结束，主控线程退出时不能return或exit。

另注意，pthread\_exit或者return返回的指针所指向的内存单元必须是全局的或者是用malloc分配的，不能在线程函数的栈上分配，因为当其它线程得到这个返回指针时线程函数已经退出了。

【练习】：编写多线程程序，总结exit、return、pthread\_exit各自退出效果。

return：返回到调用者那里去。

pthread\_exit()：将调用该函数的线程

exit: 将进程退出。

### pthread\_join函数

阻塞等待线程退出，获取线程退出状态 其作用，对应进程中 waitpid() 函数。

int pthread\_join(pthread\_t thread, void \*\*retval); 成功：0；失败：错误号

参数：thread：线程ID （【注意】：不是指针）；retval：存储线程结束状态。

对比记忆：

进程中：main返回值、exit参数-->int；等待子进程结束 wait 函数参数-->int \*

线程中：线程主函数返回值、pthread\_exit-->void \*；等待线程结束 pthread\_join 函数参数-->void \*\*

【练习】：参数 retval 非空用法。 【pthrd\_exit\_join.c】

调用该函数的线程将挂起等待，直到id为thread的线程终止。thread线程以不同的方法终止，通过pthread\_join得到的终止状态是不同的，总结如下：

1. 如果thread线程通过return返回，retval所指向的单元里存放的是thread线程函数的返回值。
2. 如果thread线程被别的线程调用pthread\_cancel异常终止掉，retval所指向的单元里存放的是常数PTHREAD\_CANCELED。
3. 如果thread线程是自己调用pthread\_exit终止的，retval所指向的单元存放的是传给pthread\_exit的参数。
4. 如果对thread线程的终止状态不感兴趣，可以传NULL给retval参数。

【练习】：使用pthread\_join函数将循环创建的多个子线程回收。 【pthrd\_loop\_join.c】

### pthread\_detach函数

实现线程分离

int pthread\_detach(pthread\_t thread); 成功：0；失败：错误号

线程分离状态：指定该状态，线程主动与主控线程断开关系。线程结束后，其退出状态不由其他线程获取，而直接自己自动释放。网络、多线程服务器常用。

进程若有该机制，将不会产生僵尸进程。僵尸进程的产生主要由于进程死后，大部分资源被释放，一点残留资源仍存于系统中，导致内核认为该进程仍存在。

也可使用 pthread\_create函数参2(线程属性)来设置线程分离。

【练习】：使用pthread\_detach函数实现线程分离 【pthrd\_detach.c】

一般情况下，线程终止后，其终止状态一直保留到其它线程调用pthread\_join获取它的状态为止。但是线程也可以被置为detach状态，**这样的线程一旦终止就立刻回收它占用的所有资源，而不保留终止状态。**不能对一个已经处于detach状态的线程调用pthread\_join，这样的调用将返回EINVAL错误。也就是说，如果已经对一个线程调用了pthread\_detach就不能再调用pthread\_join了。

### pthread\_cancel函数

杀死(取消)线程 其作用，对应进程中 kill() 函数。

int pthread\_cancel(pthread\_t thread); 成功：0；失败：错误号

【注意】：线程的取消并不是实时的，而有一定的延时。需要等待线程到达某个取消点(检查点)。

类似于玩游戏存档，必须到达指定的场所(存档点，如：客栈、仓库、城里等)才能存储进度。杀死线程也不是立刻就能完成，必须要到达取消点。

取消点：是线程检查是否被取消，并按请求进行动作的一个位置。通常是一些系统调用creat，open，pause，close，read，write..... 执行命令man 7 pthreads可以查看具备这些取消点的系统调用列表。也可参阅 APUE.12.7 取消选项小节。

可粗略认为一个系统调用(进入内核)即为一个取消点。如线程中没有取消点，可以通过调用pthreestcancel函数自行设置一个取消点。

被取消的线程， 退出值定义在Linux的pthread库中。常数PTHREAD\_CANCELED的值是-1。可在头文件pthread.h中找到它的定义：**#define PTHREAD\_CANCELED ((void \*) -1)。**因此当我们对一个已经被取消的线程使用pthread\_join回收时，得到的返回值为-1。

【练习】：终止线程的三种方法。注意“取消点”的概念。 【pthrd\_endof3.c】

#### 终止线程方式

总结：终止某个线程而不终止整个进程，有三种方法：

1. 从线程主函数return。这种方法对主控线程不适用，从main函数return相当于调用exit。
2. 一个线程可以调用pthread\_cancel终止同一进程中的另一个线程。
3. 线程可以调用pthread\_exit终止自己。

### pthread\_equal函数

比较两个线程ID是否相等。

int pthread\_equal(pthread\_t t1, pthread\_t t2);

有可能Linux在未来线程ID pthread\_t 类型被修改为结构体实现。

### 控制原语对比

进程 线程

fork pthread\_create

exit pthread\_exit

wait pthread\_join

kill pthread\_cancel

getpid pthread\_self 命名空间

## 线程属性

本节作为指引性介绍，linux下线程的属性是可以根据实际项目需要，进行设置，之前我们讨论的线程都是采用线程的默认属性，默认属性已经可以解决绝大多数开发时遇到的问题。如我们对程序的性能提出更高的要求那么需要设置线程属性，比如可以通过设置线程栈的大小来降低内存的使用，增加最大线程个数。

typedef struct

{

int etachstate; //线程的分离状态

int schedpolicy; //线程调度策略

struct sched\_param schedparam; //线程的调度参数

int inheritsched; //线程的继承性

int scope; //线程的作用域

size\_t guardsize; //线程栈末尾的警戒缓冲区大小

int stackaddr\_set; //线程的栈设置

void\* stackaddr; //线程栈的位置

size\_t stacksize; //线程栈的大小

} pthread\_attr\_t;

主要结构体成员：

1. 线程分离状态

2. 线程栈大小（默认平均分配）

3. 线程栈警戒缓冲区大小（位于栈末尾） 参 APUE.12.3 线程属性

属性值不能直接设置，须使用相关函数进行操作，初始化的函数为pthread\_attr\_init，这个函数必须在pthread\_create函数之前调用。之后须用pthread\_attr\_destroy函数来释放资源。

线程属性主要包括如下属性：作用域（scope）、栈尺寸（stack size）、栈地址（stack address）、优先级（priority）、分离的状态（detached state）、调度策略和参数（scheduling policy and parameters）。默认的属性为非绑定、非分离、缺省的堆栈、与父进程同样级别的优先级。

### 线程属性初始化

注意：应先初始化线程属性，再pthread\_create创建线程

初始化线程属性

int pthread\_attr\_init(pthread\_attr\_t \*attr); 成功：0；失败：错误号

销毁线程属性所占用的资源

int pthread\_attr\_destroy(pthread\_attr\_t \*attr); 成功：0；失败：错误号

### 线程的分离状态

线程的分离状态决定一个线程以什么样的方式来终止自己。

非分离状态：线程的默认属性是非分离状态，这种情况下，原有的线程等待创建的线程结束。只有当pthread\_join()函数返回时，创建的线程才算终止，才能释放自己占用的系统资源。

分离状态：分离线程没有被其他的线程所等待，自己运行结束了，线程也就终止了，马上释放系统资源。应该根据自己的需要，选择适当的分离状态。

线程分离状态的函数：

设置线程属性，分离or非分离

int pthread\_attr\_setdetachstate(pthread\_attr\_t \*attr, int detachstate);

获取程属性，分离or非分离

int pthread\_attr\_getdetachstate(pthread\_attr\_t \*attr, int \*detachstate);

参数： attr：已初始化的线程属性

detachstate： PTHREAD\_CREATE\_DETACHED（分离线程）

PTHREAD \_CREATE\_JOINABLE（非分离线程）

这里要注意的一点是，如果设置一个线程为分离线程，而这个线程运行又非常快，它很可能在pthread\_create函数返回之前就终止了，它终止以后就可能将线程号和系统资源移交给其他的线程使用，这样调用pthread\_create的线程就得到了错误的线程号。要避免这种情况可以采取一定的同步措施，最简单的方法之一是可以在被创建的线程里调用pthread\_cond\_timedwait函数，让这个线程等待一会儿，留出足够的时间让函数pthread\_create返回。设置一段等待时间，是在多线程编程里常用的方法。但是注意不要使用诸如wait()之类的函数，它们是使整个进程睡眠，并不能解决线程同步的问题。

### 线程的栈地址

POSIX.1定义了两个常量\_POSIX\_THREAD\_ATTR\_STACKADDR 和\_POSIX\_THREAD\_ATTR\_STACKSIZE检测系统是否支持栈属性。也可以给sysconf函数传递\_SC\_THREAD\_ATTR\_STACKADDR或 \_SC\_THREAD\_ATTR\_STACKSIZE来进行检测。

当进程栈地址空间不够用时，指定新建线程使用由malloc分配的空间作为自己的栈空间。通过pthread\_attr\_setstack和pthread\_attr\_getstack两个函数分别设置和获取线程的栈地址。

int pthread\_attr\_setstack(pthread\_attr\_t \*attr, void \*stackaddr, size\_t stacksize); 成功：0；失败：错误号

int pthread\_attr\_getstack(pthread\_attr\_t \*attr, void \*\*stackaddr, size\_t \*stacksize); 成功：0；失败：错误号

参数： attr：指向一个线程属性的指针

stackaddr：返回获取的栈地址

stacksize：返回获取的栈大小

### 线程的栈大小

当系统中有很多线程时，可能需要减小每个线程栈的默认大小，防止进程的地址空间不够用，当线程调用的函数会分配很大的局部变量或者函数调用层次很深时，可能需要增大线程栈的默认大小。

函数pthread\_attr\_getstacksize和 pthread\_attr\_setstacksize提供设置。

int pthread\_attr\_setstacksize(pthread\_attr\_t \*attr, size\_t stacksize); 成功：0；失败：错误号

int pthread\_attr\_getstacksize(pthread\_attr\_t \*attr, size\_t \*stacksize); 成功：0；失败：错误号

参数： attr：指向一个线程属性的指针

stacksize：返回线程的堆栈大小

### 线程属性控制示例

#include <pthread.h>

#define SIZE 0x100000

void \*th\_fun(void \*arg)

{

while (1)

sleep(1);

}

int main(void)

{

pthread\_t tid;

int err, detachstate, i = 1;

pthread\_attr\_t attr;

size\_t stacksize;

void \*stackaddr;

pthread\_attr\_init(&attr);

pthread\_attr\_getstack(&attr, &stackaddr, &stacksize);

pthread\_attr\_getdetachstate(&attr, &detachstate);

if (detachstate == PTHREAD\_CREATE\_DETACHED)

printf("thread detached\n");

else if (detachstate == PTHREAD\_CREATE\_JOINABLE)

printf("thread join\n");

else

printf("thread unknown\n");

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_DETACHED);

while (1) {

stackaddr = malloc(SIZE);

if (stackaddr == NULL) {

perror("malloc");

exit(1);

}

stacksize = SIZE;

pthread\_attr\_setstack(&attr, stackaddr, stacksize);

err = pthread\_create(&tid, &attr, th\_fun, NULL);

if (err != 0) {

printf("%s\n", strerror(err));

exit(1);

}

printf("%d\n", i++);

}

pthread\_attr\_destroy(&attr);

return 0;

} 【pthrd\_attr\_change.c】

## NPTL

1.察看当前pthread库版本getconf GNU\_LIBPTHREAD\_VERSION

2.NPTL实现机制(POSIX)，Native POSIX Thread Library

3.使用线程库时gcc指定 –lpthread

## 线程使用注意事项

1. 主线程退出其他线程不退出，主线程应调用pthread\_exit
2. 避免僵尸线程

pthread\_join

pthread\_detach

pthread\_create指定分离属性

被join线程可能在join函数返回前就释放完自己的所有内存资源，所以不应当返回被回收线程栈中的值;

1. malloc和mmap申请的内存可以被其他线程释放
2. 应避免在多线程模型中调用fork除非，马上exec，子进程中只有调用fork的线程存在，其他线程在子进程中均pthread\_exit
3. 信号的复杂语义很难和多线程共存，应避免在多线程引入信号机制

## 同步概念

所谓同步，即同时起步，协调一致。不同的对象，对“同步”的理解方式略有不同。如，设备同步，是指在两个设备之间规定一个共同的时间参考；数据库同步，是指让两个或多个数据库内容保持一致，或者按需要部分保持一致；文件同步，是指让两个或多个文件夹里的文件保持一致。等等

而，编程中、通信中所说的同步与生活中大家印象中的同步概念略有差异。“同”字应是指协同、协助、互相配合。主旨在协同步调，按预定的先后次序运行。

### 线程同步

同步即协同步调，按预定的先后次序运行。

线程同步，指一个线程发出某一功能调用时，在没有得到结果之前，该调用不返回。同时其它线程为保证数据一致性，不能调用该功能。

举例1： 银行存款 5000。柜台，折：取3000；提款机，卡：取 3000。剩余：2000

举例2： 内存中100字节，线程T1欲填入全1， 线程T2欲填入全0。但如果T1执行了50个字节失去cpu，T2执行，会将T1写过的内容覆盖。当T1再次获得cpu继续 从失去cpu的位置向后写入1，当执行结束，内存中的100字节，既不是全1，也不是全0。

产生的现象叫做“与时间有关的错误”(time related)。为了避免这种数据混乱，线程需要同步。

“同步”的目的，是为了避免数据混乱，解决与时间有关的错误。实际上，不仅线程间需要同步，进程间、信号间等等都需要同步机制。

因此，**所有“多个控制流，共同操作一个共享资源”的情况，都需要同步。**

### 数据混乱原因：

1. 资源共享（独享资源则不会）

2. 调度随机（意味着数据访问会出现竞争）

3. 线程间缺乏必要的同步机制。

以上3点中，前两点不能改变，欲提高效率，传递数据，资源必须共享。只要共享资源，就一定会出现竞争。只要存在竞争关系，数据就很容易出现混乱。

所以只能从第三点着手解决。使多个线程在访问共享资源的时候，出现互斥。

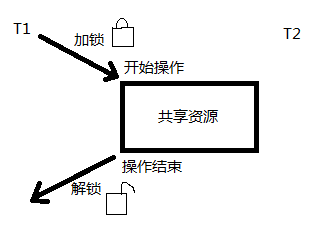
# 互斥量mutex

Linux中提供一把互斥锁mutex（也称之为互斥量）。

每个线程在对资源操作前都尝试先加锁，成功加锁才能操作，操作结束解锁。

资源还是共享的，线程间也还是竞争的，

但通过“锁”就将资源的访问变成互斥操作，而后与时间有关的错误也不会再产生了。



但，应注意：同一时刻，只能有一个线程持有该锁。

当A线程对某个全局变量加锁访问，B在访问前尝试加锁，拿不到锁，B阻塞。C线程不去加锁，而直接访问该全局变量，依然能够访问，但会出现数据混乱。

所以，互斥锁实质上是操作系统提供的一把“建议锁”（又称“协同锁”），建议程序中有多线程访问共享资源的时候使用该机制。但，并没有强制限定。

因此，即使有了mutex，如果有线程不按规则来访问数据，依然会造成数据混乱。

### 主要应用函数：

pthread\_mutex\_init函数

pthread\_mutex\_destroy函数

pthread\_mutex\_lock函数

pthread\_mutex\_trylock函数

pthread\_mutex\_unlock函数

以上5个函数的返回值都是：成功返回0， 失败返回错误号。

pthread\_mutex\_t 类型，其本质是一个结构体。为简化理解，应用时可忽略其实现细节，简单当成整数看待。

pthread\_mutex\_t mutex; 变量mutex只有两种取值1、0。

#### pthread\_mutex\_init函数

初始化一个互斥锁(互斥量) ---> 初值可看作1

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, const pthread\_mutexattr\_t \*restrict attr);

参1：传出参数，调用时应传 &mutex

restrict关键字：只用于限制指针，告诉编译器，所有修改该指针指向内存中内容的操作，只能通过本指针完成。不能通过除本指针以外的其他变量或指针修改

参2：互斥量属性。是一个传入参数，通常传NULL，选用默认属性(线程间共享)。 参APUE.12.4同步属性

1. 静态初始化：如果互斥锁 mutex 是静态分配的（定义在全局，或加了static关键字修饰），可以直接使用宏进行初始化。e.g. pthead\_mutex\_t muetx = PTHREAD\_MUTEX\_INITIALIZER;
2. 动态初始化：局部变量应采用动态初始化。e.g. pthread\_mutex\_init(&mutex, NULL)

#### pthread\_mutex\_destroy函数

销毁一个互斥锁

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

#### pthread\_mutex\_lock函数

加锁。可理解为将mutex--（或-1）

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

#### pthread\_mutex\_unlock函数

解锁。可理解为将mutex ++（或+1）

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

#### pthread\_mutex\_trylock函数

尝试加锁

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

### 加锁与解锁

#### lock与unlock：

lock尝试加锁，如果加锁不成功，线程阻塞，阻塞到持有该互斥量的其他线程解锁为止。

unlock主动解锁函数，**同时将阻塞在该锁上的所有线程全部唤醒**，至于哪个线程先被唤醒，取决于优先级、调度。默认：先阻塞、先唤醒。

例如：T1 T2 T3 T4 使用一把mutex锁。T1加锁成功，其他线程均阻塞，直至T1解锁。T1解锁后，T2 T3 T4均被唤醒，并自动再次尝试加锁。

可假想mutex锁 init成功初值为1。 lock 功能是将mutex--。 unlock将mutex++

#### lock与trylock：

lock加锁失败会阻塞，等待锁释放。

trylock加锁失败直接返回错误号（如：EBUSY），不阻塞。

### 加锁步骤测试：

看如下程序：该程序是非常典型的，由于共享、竞争而没有加任何同步机制，导致产生于时间有关的错误，造成数据混乱：

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

void \*tfn(void \*arg)

{

srand(time(NULL));

while (1) {

printf("hello ");

sleep(rand() % 3); /\*模拟长时间操作共享资源，导致cpu易主，产生与时间有关的错误\*/

printf("world\n");

sleep(rand() % 3);

}

return NULL;

}

int main(void)

{

pthread\_t tid;

srand(time(NULL));

pthread\_create(&tid, NULL, tfn, NULL);

while (1) {

printf("HELLO ");

sleep(rand() % 3);

printf("WORLD\n");

sleep(rand() % 3);

}

pthread\_join(tid, NULL);

return 0;

} 【mutex.c】

【练习】：修改该程序，使用mutex互斥锁进行同步。

1. 定义全局互斥量，初始化init(&m, NULL)互斥量，添加对应的destry
2. 两个线程while中，两次printf前后，分别加lock和unlock
3. 将unlock挪至第二个sleep后，发现交替现象很难出现。

线程在操作完共享资源后本应该立即解锁，但修改后，线程抱着锁睡眠。睡醒解锁后又立即加锁，这两个库函数本身不会阻塞。

所以在这两行代码之间失去cpu的概率很小。因此，另外一个线程很难得到加锁的机会。

1. main 中加flag = 5 将flg在while中-- 这时，主线程输出5次后试图销毁锁，但子线程未将锁释放，无法完成。
2. main 中加pthread\_cancel()将子线程取消。 【pthrd\_mutex.c】

**结论：**

**在访问共享资源前加锁，访问结束后立即解锁。锁的“粒度”应越小越好。**

## 死锁

1. 线程试图对同一个互斥量A加锁两次。

2. 线程1拥有A锁，请求获得B锁；线程2拥有B锁，请求获得A锁

【作业】：编写程序，实现上述两种死锁现象。

# 读写锁

与互斥量类似，但读写锁允许更高的并行性。其特性为：写独占，读共享。

### 读写锁状态：

一把读写锁具备三种状态：

1. 读模式下加锁状态 (读锁)

2. 写模式下加锁状态 (写锁)

3. 不加锁状态

### 读写锁特性：

1. 读写锁是“写模式加锁”时， 解锁前，所有对该锁加锁的线程都会被阻塞。
2. 读写锁是“读模式加锁”时， 如果线程以读模式对其加锁会成功；如果线程以写模式加锁会阻塞。
3. 读写锁是“读模式加锁”时， 既有试图以写模式加锁的线程，也有试图以读模式加锁的线程。那么读写锁会阻塞随后的读模式锁请求。优先满足写模式锁。**读锁、写锁并行阻塞，写锁优先级高**

读写锁也叫共享-独占锁。当读写锁以读模式锁住时，它是以共享模式锁住的；当它以写模式锁住时，它是以独占模式锁住的。**写独占、读共享。**

读写锁非常适合于对数据结构读的次数远大于写的情况。

### 主要应用函数：

pthread\_rwlock\_init函数

pthread\_rwlock\_destroy函数

pthread\_rwlock\_rdlock函数

pthread\_rwlock\_wrlock函数

pthread\_rwlock\_tryrdlock函数

pthread\_rwlock\_trywrlock函数

pthread\_rwlock\_unlock函数

以上7 个函数的返回值都是：成功返回0， 失败直接返回错误号。

pthread\_rwlock\_t类型 用于定义一个读写锁变量。

pthread\_rwlock\_t rwlock;

#### pthread\_rwlock\_init函数

初始化一把读写锁

int pthread\_rwlock\_init(pthread\_rwlock\_t \*restrict rwlock, const pthread\_rwlockattr\_t \*restrict attr);

参2：attr表读写锁属性，通常使用默认属性，传NULL即可。

#### pthread\_rwlock\_destroy函数

销毁一把读写锁

int pthread\_rwlock\_destroy(pthread\_rwlock\_t \*rwlock);

#### pthread\_rwlock\_rdlock函数

以读方式请求读写锁。（常简称为：请求读锁）

int pthread\_rwlock\_rdlock(pthread\_rwlock\_t \*rwlock);

#### pthread\_rwlock\_wrlock函数

以写方式请求读写锁。（常简称为：请求写锁）

int pthread\_rwlock\_wrlock(pthread\_rwlock\_t \*rwlock);

#### pthread\_rwlock\_unlock函数

解锁

int pthread\_rwlock\_unlock(pthread\_rwlock\_t \*rwlock);

#### pthread\_rwlock\_tryrdlock函数

非阻塞以读方式请求读写锁（非阻塞请求读锁）

int pthread\_rwlock\_tryrdlock(pthread\_rwlock\_t \*rwlock);

#### pthread\_rwlock\_trywrlock函数

非阻塞以写方式请求读写锁（非阻塞请求写锁）

int pthread\_rwlock\_trywrlock(pthread\_rwlock\_t \*rwlock);

### 读写锁示例

看如下示例，同时有多个线程对同一全局数据读、写操作。

#include <stdio.h>

#include <unistd.h>

#include <pthread.h>

int counter;

pthread\_rwlock\_t rwlock;

/\* 3个线程不定时写同一全局资源，5个线程不定时读同一全局资源 \*/

void \*th\_write(void \*arg)

{

int t, i = (int)arg;

while (1) {

pthread\_rwlock\_wrlock(&rwlock);

t = counter;

usleep(1000);

printf("=======write %d: %lu: counter=%d ++counter=%d\n", i, pthread\_self(), t, ++counter);

pthread\_rwlock\_unlock(&rwlock);

usleep(10000);

}

return NULL;

}

void \*th\_read(void \*arg)

{

int i = (int)arg;

while (1) {

pthread\_rwlock\_rdlock(&rwlock);

printf("----------------------------read %d: %lu: %d\n", i, pthread\_self(), counter);

pthread\_rwlock\_unlock(&rwlock);

usleep(2000);

}

return NULL;

}

int main(void)

{

int i;

pthread\_t tid[8];

pthread\_rwlock\_init(&rwlock, NULL);

for (i = 0; i < 3; i++)

pthread\_create(&tid[i], NULL, th\_write, (void \*)i);

for (i = 0; i < 5; i++)

pthread\_create(&tid[i+3], NULL, th\_read, (void \*)i);

for (i = 0; i < 8; i++)

pthread\_join(tid[i], NULL);

pthread\_rwlock\_destroy(&rwlock);

return 0;

} 【rwlock.c】

# 条件变量：

条件变量本身不是锁！但它也可以造成线程阻塞。通常与互斥锁配合使用。给多线程提供一个会合的场所。

### 主要应用函数：

pthread\_cond\_init函数

pthread\_cond\_destroy函数

pthread\_cond\_wait函数

pthread\_cond\_timedwait函数

pthread\_cond\_signal函数

pthread\_cond\_broadcast函数

以上6 个函数的返回值都是：成功返回0， 失败直接返回错误号。

pthread\_cond\_t类型 用于定义条件变量

pthread\_cond\_t cond;

#### pthread\_cond\_init函数

初始化一个条件变量

int pthread\_cond\_init(pthread\_cond\_t \*restrict cond, const pthread\_condattr\_t \*restrict attr);

参2：attr表条件变量属性，通常为默认值，传NULL即可

也可以使用静态初始化的方法，初始化条件变量：

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

#### pthread\_cond\_destroy函数

销毁一个条件变量

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

#### pthread\_cond\_wait函数

阻塞等待一个条件变量

int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond, pthread\_mutex\_t \*restrict mutex);

函数作用：

1. 阻塞等待条件变量cond（参1）满足
2. 释放已掌握的互斥锁（解锁互斥量）相当于pthread\_mutex\_unlock(&mutex);

**1.2.两步为一个原子操作。**

1. 当被唤醒，pthread\_cond\_wait函数返回时，解除阻塞并重新申请获取互斥锁pthread\_mutex\_lock(&mutex);

#### pthread\_cond\_timedwait函数

限时等待一个条件变量

int pthread\_cond\_timedwait(pthread\_cond\_t \*restrict cond, pthread\_mutex\_t \*restrict mutex, const struct timespec \*restrict abstime);

参3： 参看man sem\_timedwait函数，查看struct timespec结构体。

struct timespec {

time\_t tv\_sec; /\* seconds \*/ 秒

long tv\_nsec; /\* nanosecondes\*/ 纳秒

}

形参abstime：绝对时间。

如：time(NULL)返回的就是绝对时间。而alarm(1)是相对时间，相对当前时间定时1秒钟。

struct timespec t = {1, 0};

pthread\_cond\_timedwait (&cond, &mutex, &t); 只能定时到 1970年1月1日 00:00:01秒(早已经过去)

正确用法：

time\_t cur = time(NULL); 获取当前时间。

struct timespec t; 定义timespec 结构体变量t

t.tv\_sec = cur+1; 定时1秒

pthread\_cond\_timedwait (&cond, &mutex, &t); 传参 参APUE.11.6线程同步条件变量小节

在讲解setitimer函数时我们还提到另外一种时间类型：

struct timeval {

time\_t tv\_sec; /\* seconds \*/ 秒

suseconds\_t tv\_usec; /\* microseconds \*/ 微秒

};

#### pthread\_cond\_signal函数

唤醒至少一个阻塞在条件变量上的线程

int pthread\_cond\_signal(pthread\_cond\_t \*cond);

#### pthread\_cond\_broadcast函数

唤醒全部阻塞在条件变量上的线程

int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);

### 生产者消费者条件变量模型

线程同步典型的案例即为生产者消费者模型，而借助条件变量来实现这一模型，是比较常见的一种方法。假定有两个线程，一个模拟生产者行为，一个模拟消费者行为。两个线程同时操作一个共享资源（一般称之为汇聚），生产向其中添加产品，消费者从中消费掉产品。

看如下示例，使用条件变量模拟生产者、消费者问题：

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

struct msg {

struct msg \*next;

int num;

};

struct msg \*head;

pthread\_cond\_t has\_product = PTHREAD\_COND\_INITIALIZER;

pthread\_mutex\_t lock = PTHREAD\_MUTEX\_INITIALIZER;

void \*consumer(void \*p)

{

struct msg \*mp;

for (;;) {

pthread\_mutex\_lock(&lock);

while (head == NULL) { //头指针为空,说明没有节点 可以为if吗

pthread\_cond\_wait(&has\_product, &lock);

}

mp = head;

head = mp->next; //模拟消费掉一个产品

pthread\_mutex\_unlock(&lock);

printf("-Consume ---%d\n", mp->num);

free(mp);

sleep(rand() % 5);

}

}

void \*producer(void \*p)

{

struct msg \*mp;

while (1) {

mp = malloc(sizeof(struct msg));

mp->num = rand() % 1000 + 1; //模拟生产一个产品

printf("-Produce ---%d\n", mp->num);

pthread\_mutex\_lock(&lock);

mp->next = head;

head = mp;

pthread\_mutex\_unlock(&lock);

pthread\_cond\_signal(&has\_product); //将等待在该条件变量上的一个线程唤醒

sleep(rand() % 5);

}

}

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

{

pthread\_t pid, cid;

srand(time(NULL));

pthread\_create(&pid, NULL, producer, NULL);

pthread\_create(&cid, NULL, consumer, NULL);

pthread\_join(pid, NULL);

pthread\_join(cid, NULL);

return 0;

} 【conditionVar\_product\_consumer.c】

### 条件变量的优点：

相较于mutex而言，条件变量可以减少竞争。

如直接使用mutex，除了生产者、消费者之间要竞争互斥量以外，消费者之间也需要竞争互斥量，但如果汇聚（链表）中没有数据，消费者之间竞争互斥锁是无意义的。有了条件变量机制以后，只有生产者完成生产，才会引起消费者之间的竞争。提高了程序效率。

# 信号量

进化版的互斥锁（1 --> N）

由于互斥锁的粒度比较大，如果我们希望在多个线程间对某一对象的部分数据进行共享，使用互斥锁是没有办法实现的，只能将整个数据对象锁住。这样虽然达到了多线程操作共享数据时保证数据正确性的目的，却无形中导致线程的并发性下降。线程从并行执行，变成了串行执行。与直接使用单进程无异。

信号量，是相对折中的一种处理方式，既能保证同步，数据不混乱，又能提高线程并发。

### 主要应用函数：

sem\_init函数

sem\_destroy函数

sem\_wait函数

sem\_trywait函数

sem\_timedwait函数

sem\_post函数

以上6 个函数的返回值都是：成功返回0， 失败返回-1，同时设置errno。(注意，它们没有pthread前缀)

sem\_t类型，本质仍是结构体。但应用期间可简单看作为整数，忽略实现细节（类似于使用文件描述符）。

sem\_t sem; 规定信号量sem不能 < 0。头文件 <semaphore.h>

#### 信号量基本操作：

sem\_wait: 1. 信号量大于0，则信号量-- （类比pthread\_mutex\_lock）

| 2. 信号量等于0，造成线程阻塞

对应

|

sem\_post： 将信号量++，同时唤醒阻塞在信号量上的线程 （类比pthread\_mutex\_unlock）

但，由于sem\_t的实现对用户隐藏，所以所谓的++、--操作只能通过函数来实现，而不能直接++、--符号。

**信号量的初值，决定了占用信号量的线程的个数。**

#### sem\_init函数

初始化一个信号量

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

参1：sem信号量

参2：pshared取0用于线程间；取非0（一般为1）用于进程间

参3：value指定信号量初值

#### sem\_destroy函数

销毁一个信号量

int sem\_destroy(sem\_t \*sem);

#### sem\_wait函数

给信号量加锁 --

int sem\_wait(sem\_t \*sem);

#### sem\_post函数

给信号量解锁 ++

int sem\_post(sem\_t \*sem);

#### sem\_trywait函数

尝试对信号量加锁 -- (与sem\_wait的区别类比lock和trylock)

int sem\_trywait(sem\_t \*sem);

#### sem\_timedwait函数

限时尝试对信号量加锁 --

int sem\_timedwait(sem\_t \*sem, const struct timespec \*abs\_timeout);

参2：abs\_timeout采用的是绝对时间。

定时1秒：

time\_t cur = time(NULL); 获取当前时间。

struct timespec t; 定义timespec 结构体变量t

t.tv\_sec = cur+1; 定时1秒

t.tv\_nsec = t.tv\_sec +100;

sem\_timedwait(&sem, &t); 传参

### 生产者消费者信号量模型

【练习】：使用信号量完成线程间同步，模拟生产者，消费者问题。 【sem\_product\_consumer.c】

分析：

规定： 如果□中有数据，生产者不能生产，只能阻塞。

如果□中没有数据，消费者不能消费，只能等待数据。

定义两个信号量：S满 = 0， S空 = 1 （S满代表满格的信号量，S空表示空格的信号量，程序起始，格子一定为空）

所以有： T生产者主函数 { T消费者主函数 {

sem\_wait(S空); sem\_wait(S满);

生产.... 消费....

sem\_post(S满); sem\_post(S空);

} }

假设： 线程到达的顺序是:T生、T生、T消。

那么： T生1 到达，将S空-1，生产，将S满+1

T生2 到达，S空已经为0， 阻塞

T消 到达，将S满-1，消费，将S空+1

三个线程到达的顺序是：T生1、T生2、T消。而执行的顺序是T生1、T消、T生2

这里，S空 表示空格子的总数，代表可占用信号量的线程总数-->1。其实这样的话，信号量就等同于互斥锁。

但，如果S空=2、3、4……就不一样了，该信号量同时可以由多个线程占用，不再是互斥的形式。因此我们说信号量是互斥锁的加强版。

【推演练习】： 理解上述模型，推演，如果是两个消费者，一个生产者，是怎么样的情况。

【作业】：结合生产者消费者信号量模型，揣摩sem\_timedwait函数作用。编程实现，一个线程读用户输入， 另一个线程打印“hello world”。如果用户无输入，则每隔5秒向屏幕打印一个“hello world”；如果用户有输入，立刻打印“hello world”到屏幕。

## 进程间同步

### 互斥量mutex

进程间也可以使用互斥锁，来达到同步的目的。但应在pthread\_mutex\_init初始化之前，修改其属性为进程间共享。mutex的属性修改函数主要有以下几个。

#### 主要应用函数：

pthread\_mutexattr\_t mattr 类型： 用于定义mutex锁的【属性】

**pthread\_mutexattr\_init函数**： 初始化一个mutex属性对象

int pthread\_mutexattr\_init(pthread\_mutexattr\_t \*attr);

**pthread\_mutexattr\_destroy函数**： 销毁mutex属性对象 (而非销毁锁)

int pthread\_mutexattr\_destroy(pthread\_mutexattr\_t \*attr);

**pthread\_mutexattr\_setpshared函数**： 修改mutex属性。

int pthread\_mutexattr\_setpshared(pthread\_mutexattr\_t \*attr, int pshared);

参2：pshared取值：

线程锁：PTHREAD\_PROCESS\_PRIVATE (mutex的默认属性即为线程锁，进程间私有)

进程锁：PTHREAD\_PROCESS\_SHARED

#### 进程间mutex示例

进程间使用mutex来实现同步：

#include <fcntl.h>

#include <pthread.h>

#include <sys/mman.h>

#include <sys/wait.h>

struct mt {

int num;

pthread\_mutex\_t mutex;

pthread\_mutexattr\_t mutexattr;

};

int main(void)

{

int fd, i;

struct mt \*mm;

pid\_t pid;

fd = open("mt\_test", O\_CREAT | O\_RDWR, 0777);

ftruncate(fd, sizeof(\*mm));

mm = mmap(NULL, sizeof(\*mm), PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd, 0);

close(fd);

unlink("mt\_test");

//mm = mmap(NULL, sizeof(\*mm), PROT\_READ|PROT\_WRITE, MAP\_SHARED|MAP\_ANON, -1, 0);

memset(mm, 0, sizeof(\*mm));

pthread\_mutexattr\_init(&mm->mutexattr); //初始化mutex属性对象

pthread\_mutexattr\_setpshared(&mm->mutexattr, PTHREAD\_PROCESS\_SHARED); //修改属性为进程间共享

pthread\_mutex\_init(&mm->mutex, &mm->mutexattr); //初始化一把mutex琐

pid = fork();

if (pid == 0) {

for (i = 0; i < 10; i++) {

pthread\_mutex\_lock(&mm->mutex);

(mm->num)++;

printf("-child----num++ %d\n", mm->num);

pthread\_mutex\_unlock(&mm->mutex);

sleep(1);

}

} else if (pid > 0) {

for ( i = 0; i < 10; i++) {

sleep(1);

pthread\_mutex\_lock(&mm->mutex);

mm->num += 2;

printf("-parent---num+=2 %d\n", mm->num);

pthread\_mutex\_unlock(&mm->mutex);

}

wait(NULL);

}

pthread\_mutexattr\_destroy(&mm->mutexattr); //销毁mutex属性对象

pthread\_mutex\_destroy(&mm->mutex); //销毁mutex

munmap(mm,sizeof(\*mm)); //释放映射区

return 0;

} 【process\_mutex.c】

### 文件锁

借助 fcntl函数来实现锁机制。 操作文件的进程没有获得锁时，可以打开，但无法执行read、write操作。

fcntl函数： 获取、设置文件访问控制属性。

int fcntl(int fd, int cmd, ... /\* arg \*/ );

参2：

F\_SETLK (struct flock \*) 设置文件锁（trylock）

F\_SETLKW (struct flock \*) 设置文件锁（lock）W --> wait

F\_GETLK (struct flock \*) 获取文件锁

参3：

struct flock {

...

short l\_type; 锁的类型：F\_RDLCK 、F\_WRLCK 、F\_UNLCK

short l\_whence; 偏移位置：SEEK\_SET、SEEK\_CUR、SEEK\_END

off\_t l\_start; 起始偏移：1000

off\_t l\_len; 长度：0表示整个文件加锁

pid\_t l\_pid; 持有该锁的进程ID：(F\_GETLK only)

...

};

#### 进程间文件锁示例

多个进程对加锁文件进行访问：

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

void sys\_err(char \*str)

{

perror(str); exit(1);

}

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

{

int fd;

struct flock f\_lock;

if (argc < 2) {

printf("./a.out filename\n"); exit(1);

}

if ((fd = open(argv[1], O\_RDWR)) < 0)

sys\_err("open");

//f\_lock.l\_type = F\_WRLCK; /\*选用写琐\*/

f\_lock.l\_type = F\_RDLCK; /\*选用读琐\*/

f\_lock.l\_whence = SEEK\_SET;

f\_lock.l\_start = 0;

f\_lock.l\_len = 0; /\* 0表示整个文件加锁 \*/

fcntl(fd, F\_SETLKW, &f\_lock);

printf("get flock\n");

sleep(10);

f\_lock.l\_type = F\_UNLCK;

fcntl(fd, F\_SETLKW, &f\_lock);

printf("un flock\n");

close(fd); return 0;

} 【file\_lock.c】

依然遵循“读共享、写独占”特性。但！如若进程不加锁直接操作文件，依然可访问成功，但数据势必会出现混乱。

【思考】：多线程中，可以使用文件锁吗？

多线程间共享文件描述符，而给文件加锁，是通过修改文件描述符所指向的文件结构体中的成员变量来实现的。因此，多线程中无法使用文件锁。

## 哲学家用餐模型分析

### 多线程版：

选用互斥锁mutex，如创建5个， pthread\_mutex\_t m[5];

模型抽象：

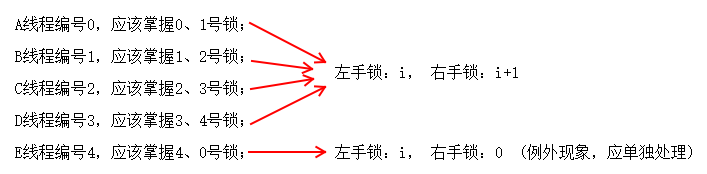
5个哲学家 --> 5个线程； 5支筷子 --> 5把互斥锁 int left(左手)， right(右手)

5个哲学家使用相同的逻辑，可通用一个线程主函数，void \*tfn(void \*arg)，使用参数来表示线程编号：int i = (int)arg;

哲学家线程根据编号知道自己是第几个哲学家，而后选定锁，锁住，吃饭。否则哲学家thinking。

A B C D E

5支筷子，在逻辑上形成环： 0 1 2 3 4 分别对应5个哲学家：



所以有：

if(i == 4)

left = i, right = 0;

else

left = i, right = i+1;

振荡：如果每个人都攥着自己左手的锁，尝试去拿右手锁，拿不到则将锁释放。过会儿五个人又同时再攥着左手锁尝试拿右手锁，依然拿不到。如此往复形成另外一种极端死锁的现象——振荡。

避免振荡现象：只需5个人中，任意一个人，拿锁的方向与其他人相逆即可(如：E，原来：左：4，右：0 现在：左：0， 右：4)。

所以以上if else语句应改为：

if(i == 4)

left = 0, right = i;

else

left = i, right = i+1;

而后， 首先应让哲学家尝试加左手锁：

while {

pthread\_mutex\_lock(&m[left]); 如果加锁成功，函数返回再加右手锁，

如果失败，应立即释放左手锁，等待。

若，左右手都加锁成功 --> 吃 --> 吃完 --> 释放锁（应先释放右手、再释放左手，是加锁顺序的逆序）

}

主线程(main)中，初始化5把锁，销毁5把锁，创建5个线程（并将i传递给线程主函数），回收5个线程。

**避免死锁的方法：**

**1. 当得不到所有所需资源时，放弃已经获得的资源，等待。**

**2. 保证资源的获取顺序，要求每个线程获取资源的顺序一致。**如：A获取顺序1、2、3；B顺序应也是1、2、3。若B为3、2、1则易出现死锁现象。

### 多进程版

相较于多线程需注意问题：

需注意如何共享信号量 (注意：坚决不能使用全局变量 sem\_t s[5])

实现：

main函数中：

循环 sem\_init(&s[i], 0, 1); 将信号量初值设为1，信号量变为互斥锁。

循环 sem\_destroy(&s[i]);

循环 创建 5 个子进程。 if(i < 5) 中完成子进程的代码逻辑。

循环 回收 5 个子进程。

子进程中：

if(i == 4)

left = 0, right == 4;

else

left = i, right = i+1;

while (1) {

使用 sem\_wait(&s[left]) 锁左手，尝试锁右手，若成功 --> 吃； 若不成功 --> 将左手锁释放。

吃完后， 先释放右手锁，再释放左手锁。

}

【重点注意】：

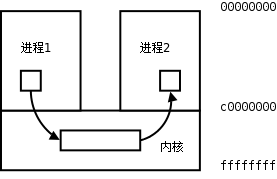
直接将sem\_t s[5]放在全局位置，试图用于子进程间共享是错误的！应将其定义放置与mmap共享映射区中。main中：

sem\_t \*s = mmap(NULL, sizeof(sem\_t) \* 5, PROT\_READ|PROT\_WRITE, MAP\_SHARED|MAP\_ANON， -1， 0);

使用方式：将s当成数组首地址看待，与使用数组s[5]没有差异。

# IPC方法

Linux环境下，进程地址空间相互独立，每个进程各自有不同的用户地址空间。任何一个进程的全局变量在另一个进程中都看不到，所以进程和进程之间不能相互访问，要交换数据必须通过内核，在内核中开辟一块缓冲区，进程1把数据从用户空间拷到内核缓冲区，进程2再从内核缓冲区把数据读走，内核提供的这种机制称为进程间通信（IPC，InterProcess Communication）。



在进程间完成数据传递需要借助操作系统提供特殊的方法，如：文件、管道、信号、共享内存、消息队列、套接字、命名管道等。随着计算机的蓬勃发展，一些方法由于自身设计缺陷被淘汰或者弃用。现今常用的进程间通信方式有：

① 管道 (使用最简单)

② 信号 (开销最小)

③ 共享映射区 (无血缘关系)

④ 本地套接字 (最稳定)

## 管道

### 管道的概念：

管道是一种最基本的IPC机制，作用于有血缘关系的进程之间，完成数据传递。调用pipe系统函数即可创建一个管道。有如下特质：

1. 其本质是一个伪文件(实为内核缓冲区)

2. 由两个文件描述符引用，一个表示读端，一个表示写端。

3. 规定数据从管道的写端流入管道，从读端流出。

管道的原理: 管道实为内核使用环形队列机制，借助内核缓冲区(4k)实现。

管道的局限性：

① 数据自己读不能自己写。

② 数据一旦被读走，便不在管道中存在，不可反复读取。

③ 由于管道采用半双工通信方式。因此，数据只能在一个方向上流动。

④ 只能在有公共祖先的进程间使用管道。

常见的通信方式有，单工通信、半双工通信、全双工通信。

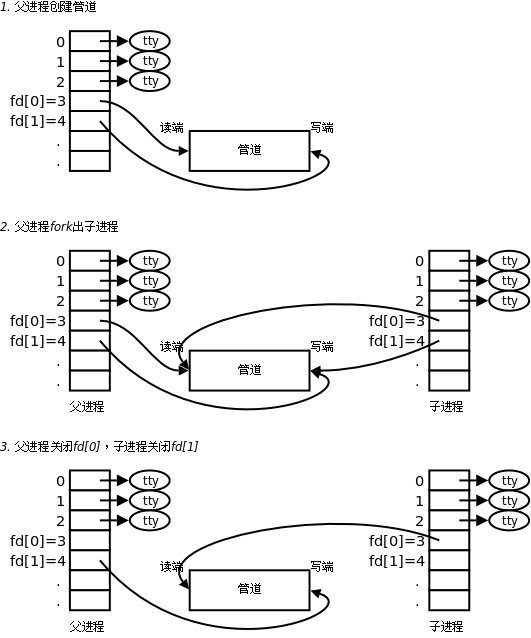
### pipe函数

创建管道

int pipe(int pipefd[2]); 成功：0；失败：-1，设置errno

函数调用成功返回r/w两个文件描述符。无需open，但需手动close。规定：fd[0] → r； fd[1] → w，就像0对应标准输入，1对应标准输出一样。向管道文件读写数据其实是在读写内核缓冲区。

管道创建成功以后，创建该管道的进程（父进程）同时掌握着管道的读端和写端。如何实现父子进程间通信呢？通常可以采用如下步骤：



1. 父进程调用pipe函数创建管道，得到两个文件描述符fd[0]、fd[1]指向管道的读端和写端。

2. 父进程调用fork创建子进程，那么子进程也有两个文件描述符指向同一管道。

3. 父进程关闭管道读端，子进程关闭管道写端。父进程可以向管道中写入数据，子进程将管道中的数据读出。由于管道是利用环形队列实现的，数据从写端流入管道，从读端流出，这样就实现了进程间通信。

练习：父子进程使用管道通信，父写入字符串，子进程读出并，打印到屏幕。 【pipe.c】

思考：为甚么，程序中没有使用sleep函数，但依然能保证子进程运行时一定会读到数据呢？

### 管道的读写行为

使用管道需要注意以下4种特殊情况（假设都是阻塞I/O操作，没有设置O\_NONBLOCK标志）：

1. 如果所有指向管道写端的文件描述符都关闭了（管道写端引用计数为0），而仍然有进程从管道的读端读数据，那么管道中剩余的数据都被读取后，再次read会返回0，就像读到文件末尾一样。

2. 如果有指向管道写端的文件描述符没关闭（管道写端引用计数大于0），而持有管道写端的进程也没有向管道中写数据，这时有进程从管道读端读数据，那么管道中剩余的数据都被读取后，再次read会阻塞，直到管道中有数据可读了才读取数据并返回。

3. 如果所有指向管道读端的文件描述符都关闭了（管道读端引用计数为0），这时有进程向管道的写端write，那么该进程会收到信号SIGPIPE，通常会导致进程异常终止。当然也可以对SIGPIPE信号实施捕捉，不终止进程。具体方法信号章节详细介绍。

4. 如果有指向管道读端的文件描述符没关闭（管道读端引用计数大于0），而持有管道读端的进程也没有从管道中读数据，这时有进程向管道写端写数据，那么在管道被写满时再次write会阻塞，直到管道中有空位置了才写入数据并返回。

总结：

① 读管道： 1. 管道中有数据，read返回实际读到的字节数。

2. 管道中无数据：

(1) 管道写端被全部关闭，read返回0 (好像读到文件结尾)

(2) 写端没有全部被关闭，read阻塞等待(不久的将来可能有数据递达，此时会让出cpu)

② 写管道： 1. 管道读端全部被关闭， 进程异常终止(也可使用捕捉SIGPIPE信号，使进程不终止)

2. 管道读端没有全部关闭：

(1) 管道已满，write阻塞。

(2) 管道未满，write将数据写入，并返回实际写入的字节数。

练习：使用管道实现父子进程间通信，完成：ls | wc –l。假定父进程实现ls，子进程实现wc。

ls命令正常会将结果集写出到stdout，但现在会写入管道的写端；wc –l 正常应该从stdin读取数据，但此时会从管道的读端读。 【pipe1.c】

程序执行，发现程序执行结束，shell还在阻塞等待用户输入。这是因为，shell → fork → ./pipe1， 程序pipe1的子进程将stdin重定向给管道，父进程执行的ls会将结果集通过管道写给子进程。若父进程在子进程打印wc的结果到屏幕之前被shell调用wait回收，shell就会先输出$提示符。

练习：使用管道实现兄弟进程间通信。 兄：ls 弟： wc -l 父：等待回收子进程。

要求，使用“循环创建N个子进程”模型创建兄弟进程，使用循环因子i标示。注意管道读写行为。 【pipe2.c】

测试：是否允许，一个pipe有一个写端，多个读端呢？是否允许有一个读端多个写端呢？ 【pipe3.c】

课后作业: 统计当前系统中进程ID大于10000的进程个数。

### 管道缓冲区大小

可以使用ulimit –a 命令来查看当前系统中创建管道文件所对应的内核缓冲区大小。通常为：

pipe size (512 bytes, -p) 8

也可以使用fpathconf函数，借助参数 选项来查看。使用该宏应引入头文件<unistd.h>

long fpathconf(int fd, int name); 成功：返回管道的大小 失败：-1，设置errno

### 管道的优劣

优点：简单，相比信号，套接字实现进程间通信，简单很多。

缺点：1. 只能单向通信，双向通信需建立两个管道。

2. 只能用于父子、兄弟进程(有共同祖先)间通信。该问题后来使用fifo有名管道解决。

## FIFO

FIFO常被称为命名管道，以区分管道(pipe)。管道(pipe)只能用于“有血缘关系”的进程间。但通过FIFO，不相关的进程也能交换数据。

FIFO是Linux基础文件类型中的一种。但，FIFO文件在磁盘上没有数据块，仅仅用来标识内核中一条通道。各进程可以打开这个文件进行read/write，实际上是在读写内核通道，这样就实现了进程间通信。

创建方式：

1. 命令：mkfifo 管道名

2. 库函数：int mkfifo(const char \*pathname, mode\_t mode); 成功：0； 失败：-1

一旦使用mkfifo创建了一个FIFO，就可以使用open打开它，常见的文件I/O函数都可用于fifo。如：close、read、write、unlink等。

【fifo\_w.c/fifo\_r.c】

## 文件进程间通信

使用文件也可以完成IPC，理论依据是，fork后，父子进程共享文件描述符。也就共享打开的文件。

练习：编程测试，父子进程共享打开的文件。借助文件进行进程间通信。 【fork\_shared\_fd.c】

思考，无血缘关系的进程可以打开同一个文件进行通信吗？为什么？

## mmap

用共享内存进行进程间通信，应注意加锁。

# 信号的概念

信号在我们的生活中随处可见， 如：古代战争中摔杯为号；现代战争中的信号弹；体育比赛中使用的信号枪......他们都有共性：1. 简单 2. 不能携带大量信息 3. 满足某个特设条件才发送。

信号是信息的载体，Linux/UNIX 环境下，古老、经典的通信方式， 现下依然是主要的通信手段。

Unix早期版本就提供了信号机制，但不可靠，信号可能丢失。Berkeley 和 AT&T都对信号模型做了更改，增加了可靠信号机制。但彼此不兼容。POSIX.1对可靠信号例程进行了标准化。

### 信号的机制

A给B发送信号，B收到信号之前执行自己的代码，收到信号后，不管执行到程序的什么位置，都要暂停运行，去处理信号，处理完毕再继续执行。与硬件中断类似——异步模式。但信号是软件层面上实现的中断，早期常被称为“软中断”。

**信号的特质**：由于信号是通过软件方法实现，其实现手段导致信号有很强的延时性。但对于用户来说，这个延迟时间非常短，不易察觉。

**每个进程收到的所有信号，都是由内核负责发送的，内核处理。**

### 与信号相关的事件和状态

**产生信号**:

1. 按键产生，如：Ctrl+c、Ctrl+z、Ctrl+\

2. 系统调用产生，如：kill、raise、abort

3. 软件条件产生，如：定时器alarm

4. 硬件异常产生，如：非法访问内存(段错误)、除0(浮点数例外)、内存对齐出错(总线错误)

5. 命令产生，如：kill命令

**递达**：递送并且到达进程。

**未决**：产生和递达之间的状态。主要由于阻塞(屏蔽)导致该状态。

**信号的处理方式:**

1. 执行默认动作

2. 忽略(丢弃)

3. 捕捉(调用户处理函数)

Linux内核的进程控制块PCB是一个结构体，task\_struct, 除了包含进程id，状态，工作目录，用户id，组id，文件描述符表，还包含了信号相关的信息，主要指阻塞信号集和未决信号集。

**阻塞信号集(信号屏蔽字)**： 将某些信号加入集合，对他们设置屏蔽，当屏蔽x信号后，再收到该信号，该信号的处理将推后(解除屏蔽后)

**未决信号集**:

1. 信号产生，未决信号集中描述该信号的位立刻翻转为1，表信号处于未决状态。当信号被处理对应位翻转回为0。这一时刻往往非常短暂。

2. 信号产生后由于某些原因(主要是阻塞)不能抵达。这类信号的集合称之为未决信号集。在屏蔽解除前，信号一直处于未决状态。

### 信号的编号

可以使用kill –l命令查看当前系统可使用的信号有哪些。

1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP

6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1

11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM

16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP

21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ

26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR

31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3

38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8

43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13

48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12

53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7

58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2

63) SIGRTMAX-1 64) SIGRTMAX

不存在编号为0的信号。其中1-31号信号称之为常规信号（也叫普通信号或标准信号），34-64称之为实时信号，驱动编程与硬件相关。名字上区别不大。而前32个名字各不相同。

### 信号4要素

与变量三要素类似的，每个信号也有其必备4要素，分别是：

1. 编号 2. 名称 3. 事件 4. 默认处理动作

可通过man 7 signal查看帮助文档获取。也可查看/usr/src/linux-headers-3.16.0-30/arch/s390/include/uapi/asm/signal.h

Signal Value Action Comment

────────────────────────────────────────────

SIGHUP 1 Term Hangup detected on controlling terminal or death of controlling process

SIGINT 2 Term Interrupt from keyboard

SIGQUIT 3 Core Quit from keyboard

SIGILL 4 Core Illegal Instruction

SIGFPE 8 Core Floating point exception

SIGKILL 9 Term Kill signal

SIGSEGV 11 Core Invalid memory reference

SIGPIPE 13 Term Broken pipe: write to pipe with no readers

SIGALRM 14 Term Timer signal from alarm(2)

SIGTERM 15 Term Termination signal

SIGUSR1 30,10,16 Term User-defined signal 1

SIGUSR2 31,12,17 Term User-defined signal 2

SIGCHLD 20,17,18 Ign Child stopped or terminated

SIGCONT 19,18,25 Cont Continue if stopped

SIGSTOP 17,19,23 Stop Stop process

SIGTSTP 18,20,24 Stop Stop typed at terminal

SIGTTIN 21,21,26 Stop Terminal input for background process

SIGTTOU 22,22,27 Stop Terminal output for background process

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

在标准信号中，有一些信号是有三个“Value”，第一个值通常对alpha和sparc架构有效，中间值针对x86、arm和其他架构，最后一个应用于mips架构。一个‘-’表示在对应架构上尚未定义该信号。

不同的操作系统定义了不同的系统信号。因此有些信号出现在Unix系统内，也出现在Linux中，而有的信号出现在FreeBSD或Mac OS中却没有出现在Linux下。这里我们只研究Linux系统中的信号。

默认动作：

Term：终止进程

Ign： 忽略信号 (默认即时对该种信号忽略操作)

Core：终止进程，生成Core文件。(查验进程死亡原因， 用于gdb调试)

Stop：停止（暂停）进程

Cont：继续运行进程

注意从man 7 signal帮助文档中可看到 : The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

这里特别强调了**9) SIGKILL 和19) SIGSTOP信号，不允许忽略和捕捉，只能执行默认动作。甚至不能将其设置为阻塞。**

**另外需清楚，只有每个信号所对应的事件发生了，该信号才会被递送(但不一定递达)，不应乱发信号！！**

### Linux常规信号一览表

1) SIGHUP: 当用户退出shell时，由该shell启动的所有进程将收到这个信号，默认动作为终止进程

2) SIGINT：当用户按下了<Ctrl+C>组合键时，用户终端向正在运行中的由该终端启动的程序发出此信号。默认动

作为终止进程。

3) SIGQUIT：当用户按下<ctrl+\>组合键时产生该信号，用户终端向正在运行中的由该终端启动的程序发出些信

号。默认动作为终止进程。

4) SIGILL：CPU检测到某进程执行了非法指令。默认动作为终止进程并产生core文件

5) SIGTRAP：该信号由断点指令或其他 trap指令产生。默认动作为终止里程 并产生core文件。

6) SIGABRT: 调用abort函数时产生该信号。默认动作为终止进程并产生core文件。

7) SIGBUS：非法访问内存地址，包括内存对齐出错，默认动作为终止进程并产生core文件。

8) SIGFPE：在发生致命的运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为0等所有的算法错误。默认动作为终止进程并产生core文件。

9) SIGKILL：无条件终止进程。本信号不能被忽略，处理和阻塞。默认动作为终止进程。它向系统管理员提供了可以杀死任何进程的方法。

10) SIGUSE1：用户定义 的信号。即程序员可以在程序中定义并使用该信号。默认动作为终止进程。

11) SIGSEGV：指示进程进行了无效内存访问。默认动作为终止进程并产生core文件。

12) SIGUSR2：另外一个用户自定义信号，程序员可以在程序中定义并使用该信号。默认动作为终止进程。

13) SIGPIPE：Broken pipe向一个没有读端的管道写数据。默认动作为终止进程。

14) SIGALRM: 定时器超时，超时的时间 由系统调用alarm设置。默认动作为终止进程。

15) SIGTERM：程序结束信号，与SIGKILL不同的是，该信号可以被阻塞和终止。通常用来要示程序正常退出。执行shell命令Kill时，缺省产生这个信号。默认动作为终止进程。

16) SIGSTKFLT：Linux早期版本出现的信号，现仍保留向后兼容。默认动作为终止进程。

17) SIGCHLD：子进程结束时，父进程会收到这个信号。默认动作为忽略这个信号。

18) SIGCONT：如果进程已停止，则使其继续运行。默认动作为继续/忽略。

19) SIGSTOP：停止进程的执行。信号不能被忽略，处理和阻塞。默认动作为暂停进程。

20) SIGTSTP：停止终端交互进程的运行。按下<ctrl+z>组合键时发出这个信号。默认动作为暂停进程。

21) SIGTTIN：后台进程读终端控制台。默认动作为暂停进程。

22) SIGTTOU: 该信号类似于SIGTTIN，在后台进程要向终端输出数据时发生。默认动作为暂停进程。

23) SIGURG：套接字上有紧急数据时，向当前正在运行的进程发出些信号，报告有紧急数据到达。如网络带外数据到达，默认动作为忽略该信号。

24) SIGXCPU：进程执行时间超过了分配给该进程的CPU时间 ，系统产生该信号并发送给该进程。默认动作为终止进程。

25) SIGXFSZ：超过文件的最大长度设置。默认动作为终止进程。

26) SIGVTALRM：虚拟时钟超时时产生该信号。类似于SIGALRM，但是该信号只计算该进程占用CPU的使用时间。默认动作为终止进程。

27) SGIPROF：类似于SIGVTALRM，它不公包括该进程占用CPU时间还包括执行系统调用时间。默认动作为终止进程。

28) SIGWINCH：窗口变化大小时发出。默认动作为忽略该信号。

29) SIGIO：此信号向进程指示发出了一个异步IO事件。默认动作为忽略。

30) SIGPWR：关机。默认动作为终止进程。

31) SIGSYS：无效的系统调用。默认动作为终止进程并产生core文件。

34) SIGRTMIN ～ (64) SIGRTMAX：LINUX的实时信号，它们没有固定的含义（可以由用户自定义）。所有的实时信号的默认动作都为终止进程。

# 信号的产生

### 终端按键产生信号

Ctrl + c → 2) SIGINT（终止/中断） "INT" ----Interrupt

Ctrl + z → 20) SIGTSTP（暂停/停止） "T" ----Terminal 终端。

Ctrl + \ → 3) SIGQUIT（退出）

### 硬件异常产生信号

除0操作 → 8) SIGFPE (浮点数例外) "F" -----float 浮点数。

非法访问内存 → 11) SIGSEGV (段错误)

总线错误 → 7) SIGBUS

### kill函数/命令产生信号

kill命令产生信号：kill -SIGKILL pid

kill函数：给指定进程发送指定信号(不一定杀死)

int kill(pid\_t pid, int sig); 成功：0；失败：-1 (ID非法，信号非法，普通用户杀init进程等权级问题)，设置errno

sig：不推荐直接使用数字，应使用宏名，因为不同操作系统信号编号可能不同，但名称一致。

pid > 0: 发送信号给指定的进程。

pid = 0: 发送信号给 与调用kill函数进程属于同一进程组的所有进程。

pid < 0: 取|pid|发给对应进程组。

pid = -1：发送给进程有权限发送的系统中所有进程。

进程组：每个进程都属于一个进程组，进程组是一个或多个进程集合，他们相互关联，共同完成一个实体任务，每个进程组都有一个进程组长，默认进程组ID与进程组长ID相同。

权限保护：super用户(root)可以发送信号给任意用户，普通用户是不能向系统用户发送信号的。 kill -9 (root用户的pid) 是不可以的。同样，普通用户也不能向其他普通用户发送信号，终止其进程。 只能向自己创建的进程发送信号。普通用户基本规则是：发送者实际或有效用户ID == 接收者实际或有效用户ID

练习：循环创建5个子进程，任一子进程用kill函数终止其父进程。 【kill.c】

### raise和abort函数

raise 函数：给当前进程发送指定信号(自己给自己发) raise(signo) == kill(getpid(), signo);

int raise(int sig); 成功：0，失败非0值

abort 函数：给自己发送异常终止信号 6) SIGABRT 信号，终止并产生core文件

void abort(void); 该函数无返回

### 软件条件产生信号

#### alarm函数

设置定时器(闹钟)。在指定seconds后，内核会给当前进程发送14）SIGALRM信号。进程收到该信号，默认动作终止。

**每个进程都有且只有唯一个定时器。**

unsigned int alarm(unsigned int seconds); 返回0或剩余的秒数，无失败。

常用：取消定时器alarm(0)，返回旧闹钟余下秒数。

例：alarm(5) → 3sec → alarm(4) → 5sec → alarm(5) → alarm(0)

定时，与进程状态无关(自然定时法)！就绪、运行、挂起(阻塞、暂停)、终止、僵尸...无论进程处于何种状态，alarm都计时。

练习：编写程序，测试你使用的计算机1秒钟能数多少个数。 【alarm .c】

使用time命令查看程序执行的时间。 程序运行的瓶颈在于IO，优化程序，首选优化IO。

实际执行时间 = 系统时间 + 用户时间 + 等待时间

#### setitimer函数

设置定时器(闹钟)。 可代替alarm函数。精度微秒us，可以实现周期定时。

int setitimer(int which, const struct itimerval \*new\_value, struct itimerval \*old\_value); 成功：0；失败：-1，设置errno

参数：which：指定定时方式

① 自然定时：ITIMER\_REAL → 14）SIGLARM 计算自然时间

② 虚拟空间计时(用户空间)：ITIMER\_VIRTUAL → 26）SIGVTALRM 只计算进程占用cpu的时间

③ 运行时计时(用户+内核)：ITIMER\_PROF → 27）SIGPROF 计算占用cpu及执行系统调用的时间

练习: 使用setitimer函数实现alarm函数，重复计算机1秒数数程序。 【setitimer.c】

拓展练习，结合man page编写程序，测试it\_interval、it\_value这两个参数的作用。 【setitimer1.c】

提示： it\_interval：用来设定两次定时任务之间间隔的时间。

it\_value：定时的时长

两个参数都设置为0，即清0操作。

## 信号集操作函数

内核通过读取未决信号集来判断信号是否应被处理。信号屏蔽字mask可以影响未决信号集。而我们可以在应用程序中自定义set来改变mask。已达到屏蔽指定信号的目的。

### 信号集设定

sigset\_t set; // typedef unsigned long sigset\_t;

int sigemptyset(sigset\_t \*set); 将某个信号集清0 成功：0；失败：-1

int sigfillset(sigset\_t \*set); 将某个信号集置1 成功：0；失败：-1

int sigaddset(sigset\_t \*set, int signum); 将某个信号加入信号集 成功：0；失败：-1

int sigdelset(sigset\_t \*set, int signum); 将某个信号清出信号集 成功：0；失败：-1

int sigismember(const sigset\_t \*set, int signum);判断某个信号是否在信号集中 返回值：在集合：1；不在：0；出错：-1

sigset\_t类型的本质是位图。但不应该直接使用位操作，而应该使用上述函数，保证跨系统操作有效。

对比认知select 函数。

### sigprocmask函数

用来屏蔽信号、解除屏蔽也使用该函数。其本质，读取或修改进程的信号屏蔽字(PCB中)

**严格注意，屏蔽信号：只是将信号处理延后执行(延至解除屏蔽)；而忽略表示将信号丢处理。**

int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset); 成功：0；失败：-1，设置errno

参数：

set：传入参数，是一个位图，set中哪位置1，就表示当前进程屏蔽哪个信号。

oldset：传出参数，保存旧的信号屏蔽集。

how参数取值： 假设当前的信号屏蔽字为mask

1. SIG\_BLOCK: 当how设置为此值，set表示需要屏蔽的信号。相当于 mask = mask|set
2. SIG\_UNBLOCK: 当how设置为此，set表示需要解除屏蔽的信号。相当于 mask = mask & ~set
3. SIG\_SETMASK: 当how设置为此，set表示用于替代原始屏蔽及的新屏蔽集。相当于 mask = set若，调用sigprocmask解除了对当前若干个信号的阻塞，则在sigprocmask返回前，至少将其中一个信号递达。

### sigpending函数

读取当前进程的**未决**信号集

int sigpending(sigset\_t \*set); set传出参数。 返回值：成功：0；失败：-1，设置errno

练习：编写程序。把所有常规信号的未决状态打印至屏幕。 【sigpending.c】

# 信号捕捉

### signal函数

注册一个信号捕捉函数：

typedef void (\*sighandler\_t)(int);

sighandler\_t signal(int signum, sighandler\_t handler);

该函数由ANSI定义，由于历史原因在不同版本的Unix和不同版本的Linux中可能有不同的行为。因此应该尽量避免使用它，取而代之使用sigaction函数。

void (\*signal(int signum, void (\*sighandler\_t)(int))) (int);

能看出这个函数代表什么意思吗？ 注意多在复杂结构中使用typedef。

### sigaction函数

修改信号处理动作（通常在Linux用其来注册一个信号的捕捉函数）

int sigaction(int signum, const struct sigaction \*act, struct sigaction \*oldact); 成功：0；失败：-1，设置errno

参数：

act：传入参数，新的处理方式。

oldact：传出参数，旧的处理方式。 【signal.c】

#### struct sigaction结构体

struct sigaction {

void (\*sa\_handler)(int);

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);

sigset\_t sa\_mask;

int sa\_flags;

void (\*sa\_restorer)(void);

};

sa\_restorer：该元素是过时的，不应该使用，POSIX.1标准将不指定该元素。(弃用)

sa\_sigaction：当sa\_flags被指定为SA\_SIGINFO标志时，使用该信号处理程序。(很少使用)

重点掌握：

① sa\_handler：指定信号捕捉后的处理函数名(即注册函数)。也可赋值为SIG\_IGN表忽略 或 SIG\_DFL表执行默认动作

② sa\_mask: 调用信号处理函数时，所要屏蔽的信号集合(信号屏蔽字)。注意：仅在处理函数被调用期间屏蔽生效，是临时性设置。

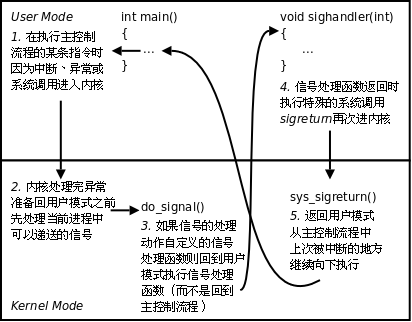
③ sa\_flags：通常设置为0，表使用默认属性。

#### 信号捕捉特性

1. 进程正常运行时，默认PCB中有一个信号屏蔽字，假定为☆，它决定了进程自动屏蔽哪些信号。当注册了某个信号捕捉函数，捕捉到该信号以后，要调用该函数。而该函数有可能执行很长时间，在这期间所屏蔽的信号不由☆来指定。而是用sa\_mask来指定。调用完信号处理函数，再恢复为☆。
2. XXX信号捕捉函数执行期间，XXX信号自动被屏蔽。
3. 阻塞的常规信号不支持排队，产生多次只记录一次。（后32个实时信号支持排队）

练习1：为某个信号设置捕捉函数 【sigaction1.c】练习2： 验证在信号处理函数执行期间，该信号多次递送，那么只在处理函数之行结束后，处理一次。 【sigaction2.c】练习3：验证sa\_mask在捕捉函数执行期间的屏蔽作用。 【sigaction3.c】

### 内核实现信号捕捉过程：



# 竞态条件(时序竞态)：

### pause函数

调用该函数可以造成进程主动挂起，等待信号唤醒。调用该系统调用的进程将处于阻塞状态(主动放弃cpu) 直到有信号递达将其唤醒。

int pause(void); 返回值：-1 并设置errno为EINTR

返回值：

① 如果信号的默认处理动作是终止进程，则进程终止，pause函数么有机会返回。

② 如果信号的默认处理动作是忽略，进程继续处于挂起状态，pause函数不返回。

③ 如果信号的处理动作是捕捉，则【调用完信号处理函数之后，pause返回-1】

errno设置为EINTR，表示“被信号中断”。想想我们还有哪个函数只有出错返回值。

④ pause收到的信号不能被屏蔽，如果被屏蔽，那么pause就不能被唤醒。

练习：使用pause和alarm来实现sleep函数。 【mysleep.c】

注意，unslept = alarm(0)的用法。

例如：睡觉，alarm(10)闹铃。

正常： 10后闹铃将我唤醒，这时额外设置alarm(0)取消闹铃，不会出错。

异常： 5分钟，被其他事物吵醒，alarm(0)取消闹铃防止打扰。

### 时序竞态

#### 前导例

设想如下场景：

欲睡觉，定闹钟10分钟，希望10分钟后闹铃将自己唤醒。

正常：定时，睡觉，10分钟后被闹钟唤醒。

异常：闹钟定好后，被唤走，外出劳动，20分钟后劳动结束。回来继续睡觉计划，但劳动期间闹钟已经响过，不会再将我唤醒。

#### 时序问题分析

回顾，借助pause和alarm实现的mysleep函数。设想如下时序：

1. 注册SIGALRM信号处理函数 （sigaction...)

2. 调用alarm(1) 函数设定闹钟1秒。

3. 函数调用刚结束，开始倒计时1秒。当前进程失去cpu，内核调度优先级高的进程(有多个)取代当前进程。当前进程无法获得cpu，进入就绪态等待cpu。

4. 1秒后，闹钟超时，内核向当前进程发送SIGALRM信号(自然定时法，与进程状态无关)，高优先级进程尚未执行完，当前进程仍处于就绪态，信号无法处理(未决)

5. 优先级高的进程执行完，当前进程获得cpu资源，内核调度回当前进程执行。SIGALRM信号递达，信号设置捕捉，执行处理函数sig\_alarm。

6. 信号处理函数执行结束，返回当前进程主控流程，pause()被调用挂起等待。（欲等待alarm函数发送的SIGALRM信号将自己唤醒）

7. SIGALRM信号已经处理完毕，pause不会等到。

#### 解决时序问题

可以通过设置屏蔽SIGALRM的方法来控制程序执行逻辑，但无论如何设置，程序都有可能在“解除信号屏蔽”与“挂起等待信号”这个两个操作间隙失去cpu资源。除非将这两步骤合并成一个“原子操作”。sigsuspend函数具备这个功能。在对时序要求严格的场合下都应该使用sigsuspend替换pause。

int sigsuspend(const sigset\_t \*mask); 挂起等待信号。

sigsuspend函数调用期间，进程信号屏蔽字由其**参数mask**指定。

可将某个信号（如SIGALRM）从临时信号屏蔽字mask中删除，这样在调用sigsuspend时将解除对该信号的屏蔽，然后挂起等待，当sigsuspend返回时，进程的信号屏蔽字恢复为原来的值。如果原来对该信号是屏蔽态，sigsuspend函数返回后仍然屏蔽该信号。

改进版mysleep 【sigsuspend.c】

#### 总结

竞态条件，跟系统负载有很紧密的关系，体现出信号的不可靠性。系统负载越严重，信号不可靠性越强。

不可靠由其实现原理所致。信号是通过软件方式实现(跟内核调度高度依赖，延时性强)，每次系统调用结束后，或中断处理处理结束后，需通过扫描PCB中的未决信号集，来判断是否应处理某个信号。当系统负载过重时，会出现时序混乱。

这种意外情况只能在编写程序过程中，提早预见，主动规避，而无法通过gdb程序调试等其他手段弥补。且由于该错误不具规律性，后期捕捉和重现十分困难。

### 全局变量异步I/O

分析如下父子进程交替数数程序。当捕捉函数里面的sleep取消，程序即会出现问题。请分析原因。

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

int n = 0, flag = 0;

void sys\_err(char \*str)

{

perror(str);

exit(1);

}

void do\_sig\_child(int num)

{

printf("I am child %d\t%d\n", getpid(), n);

n += 2;

flag = 1;

sleep(1);

}

void do\_sig\_parent(int num)

{

printf("I am parent %d\t%d\n", getpid(), n);

n += 2;

flag = 1;

sleep(1);

}

int main(void)

{

pid\_t pid;

struct sigaction act;

if ((pid = fork()) < 0)

sys\_err("fork");

else if (pid > 0) {

n = 1;

sleep(1);

act.sa\_handler = do\_sig\_parent;

sigemptyset(&act.sa\_mask);

act.sa\_flags = 0;

sigaction(SIGUSR2, &act, NULL); //注册自己的信号捕捉函数 父使用SIGUSR2信号

do\_sig\_parent(0);

while (1) {

/\* wait for signal \*/;

if (flag == 1) { //父进程数数完成

kill(pid, SIGUSR1);

flag = 0; //标志已经给子进程发送完信号

}

}

} else if (pid == 0) {

n = 2;

act.sa\_handler = do\_sig\_child;

sigemptyset(&act.sa\_mask);

act.sa\_flags = 0;

sigaction(SIGUSR1, &act, NULL);

while (1) {

/\* waiting for a signal \*/;

if (flag == 1) {

kill(getppid(), SIGUSR2);

flag = 0;

}

}

}

return 0;

} 【sync\_process.c】

示例中，通过flag变量标记程序实行进度。flag置1表示数数完成。flag置0表示给对方发送信号完成。

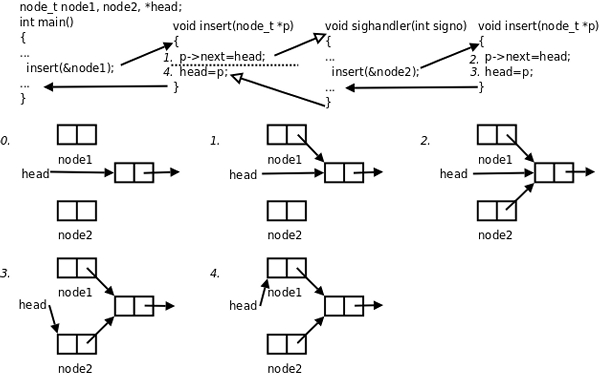
问题出现的位置，在父子进程kill函数之后需要紧接着调用 flag，将其置0，标记信号已经发送。但，在这期间很有可能被kernel调度，失去执行权利，而对方获取了执行时间，通过发送信号回调捕捉函数，从而修改了全局的flag。

如何解决该问题呢？可以使用后续课程讲到的“锁”机制。当操作全局变量的时候，通过加锁、解锁来解决该问题。

现阶段，我们在编程期间如若使用全局变量，应在主观上注意**全局变量的异步IO**可能造成的问题。

### 可/不可重入函数

一个函数在被调用执行期间(尚未调用结束)，由于某种时序又被重复调用，称之为“重入”。根据函数实现的方法可分为“可重入函数”和“不可重入函数”两种。看如下时序。



显然，insert函数是不可重入函数，重入调用，会导致意外结果呈现。究其原因，是该函数内部实现使用了全局变量。

#### 注意事项

1. 定义可重入函数，函数内不能含有全局变量及static变量，不能使用malloc、free
2. 信号捕捉函数应设计为可重入函数
3. 信号处理程序可以调用的可重入函数可参阅man 7 signal
4. 没有包含在上述列表中的函数大多是不可重入的，其原因为：
   1. 使用静态数据结构
   2. 调用了malloc或free
   3. 是标准I/O函数

# SIGCHLD信号

### SIGCHLD的产生条件

子进程终止时

子进程接收到SIGSTOP信号停止时

子进程处在停止态，接受到SIGCONT后唤醒时

### 借助SIGCHLD信号回收子进程

子进程结束运行，其父进程会收到SIGCHLD信号。该信号的默认处理动作是忽略。可以捕捉该信号，在捕捉函数中完成子进程状态的回收。

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#include <signal.h>

void sys\_err(char \*str)

{

perror(str);

exit(1);

}

void do\_sig\_child(int signo)

{

int status; pid\_t pid;

while ((pid = waitpid(0, &status, WNOHANG)) > 0) {

if (WIFEXITED(status))

printf("child %d exit %d\n", pid, WEXITSTATUS(status));

else if (WIFSIGNALED(status))

printf("child %d cancel signal %d\n", pid, WTERMSIG(status));

}

}

int main(void)

{

pid\_t pid; int i;

for (i = 0; i < 10; i++) {

if ((pid = fork()) == 0)

break;

else if (pid < 0)

sys\_err("fork");

}

if (pid == 0) {

int n = 1;

while (n--) {

printf("child ID %d\n", getpid());

sleep(1);

}

return i+1;

} else if (pid > 0) {

struct sigaction act;

act.sa\_handler = do\_sig\_child;

sigemptyset(&act.sa\_mask);

act.sa\_flags = 0;

sigaction(SIGCHLD, &act, NULL);

while (1) {

printf("Parent ID %d\n", getpid());

sleep(1);

}

}

return 0;

}

分析该例子。结合 17)SIGCHLD 信号默认动作，掌握父使用捕捉函数回收子进程的方式。 【sigchild.c】

如果每创建一个子进程后不使用sleep可以吗？可不可以将程序中，捕捉函数内部的while替换为if？为什么？

if ((pid = waitpid(0, &status, WNOHANG)) > 0) { ... }

思考：信号不支持排队，当正在执行SIGCHLD捕捉函数时，再过来一个或多个SIGCHLD信号怎么办？

### 子进程结束status处理方式

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

options

WNOHANG

没有子进程结束，立即返回

WUNTRACED

如果子进程由于被停止产生的SIGCHLD，waitpid则立即返回

WCONTINUED

如果子进程由于被SIGCONT唤醒而产生的SIGCHLD，waitpid则立即返回

获取status

WIFEXITED(status)

子进程正常exit终止，返回真

WEXITSTATUS(status)返回子进程正常退出值

WIFSIGNALED(status)

子进程被信号终止，返回真

WTERMSIG(status)返回终止子进程的信号值

WIFSTOPPED(status)

子进程被停止，返回真

WSTOPSIG(status)返回停止子进程的信号值

WIFCONTINUED(status)

### SIGCHLD信号注意问题

1. 子进程继承了父进程的信号屏蔽字和信号处理动作，但子进程没有继承未决信号集spending。
2. 注意注册信号捕捉函数的位置。
3. 应该在fork之前，阻塞SIGCHLD信号。注册完捕捉函数后解除阻塞。

# 信号传参

### 发送信号传参

sigqueue函数对应kill函数，但可在向指定进程发送信号的同时携带参数

int sigqueue(pid\_t pid, int sig, const union sigval value);成功：0；失败：-1，设置errno

union sigval {

int sival\_int;

void \*sival\_ptr;

};

向指定进程发送指定信号的同时，携带数据。但，如传地址，需注意，不同进程之间虚拟地址空间各自独立，将当前进程地址传递给另一进程没有实际意义。

### 捕捉函数传参

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

struct sigaction {

void (\*sa\_handler)(int);

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);

sigset\_t sa\_mask;

int sa\_flags;

void (\*sa\_restorer)(void);

};

当注册信号捕捉函数，希望获取更多信号相关信息，不应使用sa\_handler而应该使用sa\_sigaction。但此时的**sa\_flags必须指定为SA\_SIGINFO**。siginfo\_t是一个成员十分丰富的结构体类型，可以携带各种与信号相关的数据。

## 中断系统调用

系统调用可分为两类：慢速系统调用和其他系统调用。

1. 慢速系统调用：可能会使进程永远阻塞的一类。如果在阻塞期间收到一个信号，该系统调用就被中断,不再继续执行(早期)；也可以设定系统调用是否重启。如，read、write、pause、wait...
2. 其他系统调用：getpid、getppid、fork...

结合pause，回顾慢速系统调用：

慢速系统调用被中断的相关行为，实际上就是pause的行为： 如，read

① 想中断pause，信号不能被屏蔽。

② 信号的处理方式必须是捕捉 (默认、忽略都不可以)

③ 中断后返回-1， 设置errno为EINTR(表“被信号中断”)

可修改sa\_flags参数来设置被信号中断后系统调用是否重启。SA\_INTERRURT不重启。 SA\_RESTART重启。

扩展了解：

sa\_flags还有很多可选参数，适用于不同情况。如：捕捉到信号后，在执行捕捉函数期间，不希望自动阻塞该信号，可将sa\_flags设置为SA\_NODEFER，除非sa\_mask中包含该信号。

# 终端：

在UNIX系统中，用户通过终端登录系统后得到一个Shell进程，这个终端成为Shell进程的控制终端（Controlling Terminal），进程中，控制终端是保存在PCB中的信息，而fork会复制PCB中的信息，因此由Shell进程启动的其它进程的控制终端也是这个终端。默认情况下（没有重定向），每个进程的标准输入、标准输出和标准错误输出都指向控制终端，进程从标准输入读也就是读用户的键盘输入，进程往标准输出或标准错误输出写也就是输出到显示器上。信号中还讲过，在控制终端输入一些特殊的控制键可以给前台进程发信号，例如Ctrl-C表示SIGINT，Ctrl-\表示SIGQUIT。

Alt + Ctrl + F1、F2、F3、F4、F5、F6 字符终端 pts (pseudo terminal slave) 指伪终端。

Alt + F7 图形终端

SSH、Telnet... 网络终端

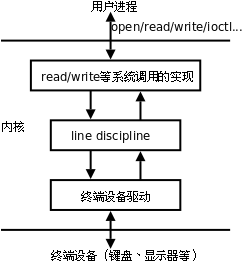
### 终端的启动流程：

文件与I/O中讲过，每个进程都可以通过一个特殊的设备文件/dev/tty访问它的控制终端。事实上每个终端设备都对应一个不同的设备文件，/dev/tty提供了一个通用的接口，一个进程要访问它的控制终端既可以通过/dev/tty也可以通过该终端设备所对应的设备文件来访问。ttyname函数可以由文件描述符查出对应的文件名，该文件描述符必须指向一个终端设备而不能是任意文件。

简单来说，一个Linux系统启动，大致经历如下的步骤：

init --> fork --> exec --> getty --> 用户输入帐号 --> login --> 输入密码 --> exec --> bash

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



终端设备模块

line disciline: 线路规程，用来过滤键盘输入的内容。

### ttyname函数

由文件描述符查出对应的文件名

char \*ttyname(int fd); 成功：终端名；失败：NULL，设置errno

下面我们借助ttyname函数，通过实验看一下各种不同的终端所对应的设备文件名。

#include <unistd.h>

#include <stdio.h>

int main(void)

{

printf("fd 0: %s\n", ttyname(0));

printf("fd 1: %s\n", ttyname(1));

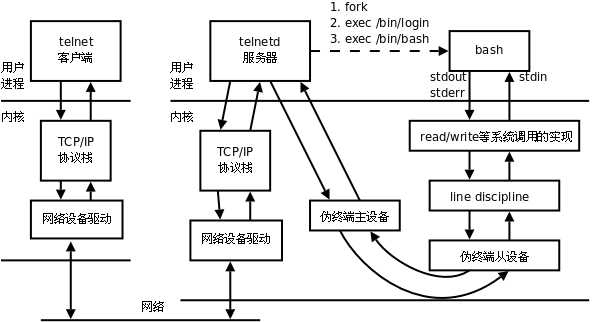
printf("fd 2: %s\n", ttyname(2));

return 0;

} 【ttyname.c】

### 网络终端：

虚拟终端或串口终端的数目是有限的，虚拟终端(字符控制终端)一般就是/dev/tty1∼/dev/tty6六个，串口终端的数目也不超过串口的数目。然而网络终端或图形终端窗口的数目却是不受限制的，这是通过伪终端（Pseudo TTY）实现的。一套伪终端由一个**主设备**（PTY Master）和一个**从设备**（PTY Slave）组成。主设备在概念上相当于键盘和显示器，只不过它不是真正的硬件而是一个内核模块，操作它的也不是用户而是另外一个进程。从设备和上面介绍的/dev/tty1这样的终端设备模块类似，只不过它的底层驱动程序不是访问硬件而是访问主设备。网络终端或图形终端窗口的Shell进程以及它启动的其它进程都会认为自己的控制终端是伪终端从设备，例如/dev/pts/0、/dev/pts/1等。下面以telnet为例说明网络登录和使用伪终端的过程。



网络终端

TCP/IP协议栈：在数据包上添加报头。

如果telnet客户端和服务器之间的网络延迟较大，我们会观察到按下一个键之后要过几秒钟才能回显到屏幕上。这说明我们每按一个键telnet客户端都会立刻把该字符发送给服务器，然后这个字符经过伪终端主设备和从设备之后被Shell进程读取，同时回显到伪终端从设备，回显的字符再经过伪终端主设备、telnetd服务器和网络发回给telnet客户端，显示给用户看。也许你会觉得吃惊，但真的是这样：每按一个键都要在网络上走个来回！

# 进程组

### 概念和特性

进程组，也称之为作业。BSD于1980年前后向Unix中增加的一个新特性。代表一个或多个进程的集合。每个进程都属于一个进程组。在waitpid函数和kill函数的参数中都曾使用到。操作系统设计的进程组的概念，是为了简化对多个进程的管理。

当父进程，创建子进程的时候，默认子进程与父进程属于同一进程组。进程组ID==第一个进程ID(组长进程)。所以，组长进程标识：其进程组ID==其进程ID

可以使用kill -SIGKILL -进程组ID(负的)来将整个进程组内的进程全部杀死。 【kill\_multprocess.c】

组长进程可以创建一个进程组，创建该进程组中的进程，然后终止。只要进程组中有一个进程存在，进程组就存在，与组长进程是否终止无关。

进程组生存期：进程组创建到最后一个进程离开(终止或转移到另一个进程组)。

一个进程可以为自己或子进程设置进程组ID

### 进程组操作函数

#### getpgrp函数

获取当前进程的进程组ID

pid\_t getpgrp(void); 总是返回调用者的进程组ID

#### getpgid函数

获取指定进程的进程组ID

pid\_t getpgid(pid\_t pid); 成功：0；失败：-1，设置errno

如果pid = 0，那么该函数作用和getpgrp一样。

练习：查看进程对应的进程组ID 【getpgid.c】

#### setpgid函数

改变进程默认所属的进程组。通常可用来加入一个现有的进程组或创建一个新进程组。

int setpgid(pid\_t pid, pid\_t pgid); 成功：0；失败：-1，设置errno

将参1对应的进程，加入参2对应的进程组中。

注意：

1. 如改变子进程为新的组，应fork后，exec前。

2. 权级问题。非root进程只能改变自己创建的子进程，或有权限操作的进程

练习：修改子进程的进程组ID 【setpgid.c】

# 会话

### 创建会话

创建一个会话需要注意以下6点注意事项：

1. 调用进程不能是进程组组长，该进程变成新会话首进程(session header)
2. 该进程成为一个新进程组的组长进程。
3. 需有root权限(ubuntu不需要)
4. 新会话丢弃原有的控制终端，该会话没有控制终端
5. 该调用进程是组长进程，则出错返回
6. 建立新会话时，先调用fork, 父进程终止，子进程调用setsid

### getsid函数

获取进程所属的会话ID

pid\_t getsid(pid\_t pid); 成功：返回调用进程的会话ID；失败：-1，设置errno

pid为0表示察看当前进程session ID

ps ajx命令查看系统中的进程。参数a表示不仅列当前用户的进程，也列出所有其他用户的进程，参数x表示不仅列有控制终端的进程，也列出所有无控制终端的进程，参数j表示列出与作业控制相关的信息。

组长进程不能成为新会话首进程，新会话首进程必定会成为组长进程。

### setsid函数

创建一个会话，并以自己的ID设置进程组ID，同时也是新会话的ID。

pid\_t setsid(void); 成功：返回调用进程的会话ID；失败：-1，设置errno

调用了setsid函数的进程，既是新的会长，也是新的组长。

练习：fork一个子进程，并使其创建一个新会话。查看进程组ID、会话ID前后变化 【session.c】

# 守护进程

Daemon(精灵)进程，是Linux中的后台服务进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。一般采用以d结尾的名字。

Linux后台的一些系统服务进程，没有控制终端，不能直接和用户交互。不受用户登录、注销的影响，一直在运行着，他们都是守护进程。如：预读入缓输出机制的实现；ftp服务器；nfs服务器等。

创建守护进程，最关键的一步是调用setsid函数创建一个新的Session，并成为Session Leader。

### 创建守护进程模型

1. 创建子进程，父进程退出

所有工作在子进程中进行形式上脱离了控制终端

1. 在子进程中创建新会话

　　 setsid()函数

　　 使子进程完全独立出来，脱离控制

1. 改变当前目录为根目录

　　 chdir()函数

　　 防止占用可卸载的文件系统

　　 也可以换成其它路径

1. 重设文件权限掩码

　　 umask()函数

　　 防止继承的文件创建屏蔽字拒绝某些权限

　　 增加守护进程灵活性

1. 关闭文件描述符

　　 继承的打开文件不会用到，浪费系统资源，无法卸载

1. 开始执行守护进程核心工作
2. 守护进程退出处理程序模型 【mydaemond.c】

**课后自行安装线程man page，命令：sudo apt-get install manpages-posix-dev**

**安装完成，使用man -k pthread如能看到线程函数列表则表明安装成功。**

# 网络基础

## 协议的概念

### 什么是协议

从应用的角度出发，协议可理解为“规则”，是数据传输和数据的解释的规则。

假设，A、B双方欲传输文件。规定：

第一次，传输文件名，接收方接收到文件名，应答OK给传输方；

第二次，发送文件的尺寸，接收方接收到该数据再次应答一个OK；

第三次，传输文件内容。同样，接收方接收数据完成后应答OK表示文件内容接收成功。

由此，无论A、B之间传递何种文件，都是通过三次数据传输来完成。A、B之间形成了一个最简单的数据传输规则。双方都按此规则发送、接收数据。A、B之间达成的这个相互遵守的规则即为协议。

这种仅在A、B之间被遵守的协议称之为**原始协议**。当此协议被更多的人采用，不断的增加、改进、维护、完善。最终形成一个稳定的、完整的文件传输协议，被广泛应用于各种文件传输过程中。该协议就成为一个**标准协议**。最早的ftp协议就是由此衍生而来。

TCP协议注重数据的传输。http协议着重于数据的解释。

### 典型协议

传输层 常见协议有TCP/UDP协议。

应用层 常见的协议有HTTP协议，FTP协议。

网络层 常见协议有IP协议、ICMP协议、IGMP协议。

网络接口层 常见协议有ARP协议、RARP协议。

TCP[传输控制协议](http://baike.baidu.com/view/544903.htm" \t "_blank)（Transmission Control Protocol）是一种面向连接的、可靠的、基于字节流的[传输层](http://baike.baidu.com/view/239605.htm" \t "_blank)通信协议。

UDP用户数据报协议（User Datagram Protocol）是[OSI](http://baike.baidu.com/view/113948.htm" \t "_blank)参考模型中一种无连接的[传输层](http://baike.baidu.com/view/239605.htm" \t "_blank)协议，提供面向事务的简单不可靠信息传送服务。

HTTP[超文本传输协议](http://baike.baidu.com/view/468465.htm" \t "_blank)（Hyper Text Transfer Protocol）是[互联网](http://baike.baidu.com/view/6825.htm" \t "_blank)上应用最为广泛的一种[网络协议](http://baike.baidu.com/view/16603.htm" \t "_blank)。

FTP文件传输协议（File Transfer Protocol）

IP协议是[因特网](http://baike.baidu.com/view/1706.htm" \t "_blank)互联协议（Internet Protocol）

ICMP协议是Internet控制[报文](http://baike.baidu.com/view/175122.htm" \t "_blank)协议（Internet Control Message Protocol）它是[TCP/IP协议族](http://baike.baidu.com/view/2221037.htm" \t "_blank)的一个子协议，用于在IP[主机](http://baike.baidu.com/view/23880.htm" \t "_blank)、[路由](http://baike.baidu.com/view/18655.htm" \t "_blank)器之间传递控制消息。

IGMP协议是 Internet 组管理协议（Internet Group Management Protocol），是因特网协议家族中的一个组播协议。该协议运行在主机和组播路由器之间。

[ARP](http://baike.baidu.com/view/32698.htm" \t "_blank)协议是正向[地址解析协议](http://baike.baidu.com/view/149421.htm" \t "_blank)（Address Resolution Protocol），通过已知的IP，寻找对应主机的[MAC地址](http://baike.baidu.com/view/69334.htm" \t "_blank)。

[RARP](http://baike.baidu.com/view/32772.htm" \t "_blank)是反向地址转换协议，通过MAC地址确定IP地址。

## 网络应用程序设计模式

### C/S模式

传统的网络应用设计模式，客户机(client)/服务器(server)模式。需要在通讯两端各自部署客户机和服务器来完成数据通信。

### B/S模式

浏览器()/服务器(server)模式。只需在一端部署服务器，而另外一端使用每台PC都默认配置的浏览器即可完成数据的传输。

### 优缺点

对于C/S模式来说，其优点明显。客户端位于目标主机上可以保证性能，将数据缓存至客户端本地，从而**提高数据传输效率**。且，一般来说客户端和服务器程序由一个开发团队创作，所以他们之间**所采用的协议相对灵活**。可以在标准协议的基础上根据需求裁剪及定制。例如，腾讯公司所采用的通信协议，即为ftp协议的修改剪裁版。

因此，传统的网络应用程序及较大型的网络应用程序都首选C/S模式进行开发。如，知名的网络游戏魔兽世界。3D画面，数据量庞大，使用C/S模式可以提前在本地进行大量数据的缓存处理，从而提高观感。

C/S模式的缺点也较突出。由于客户端和服务器都需要有一个开发团队来完成开发。**工作量**将成倍提升，开发周期较长。另外，从用户角度出发，需要将客户端安插至用户主机上，对用户主机的**安全性构成威胁**。这也是很多用户不愿使用C/S模式应用程序的重要原因。

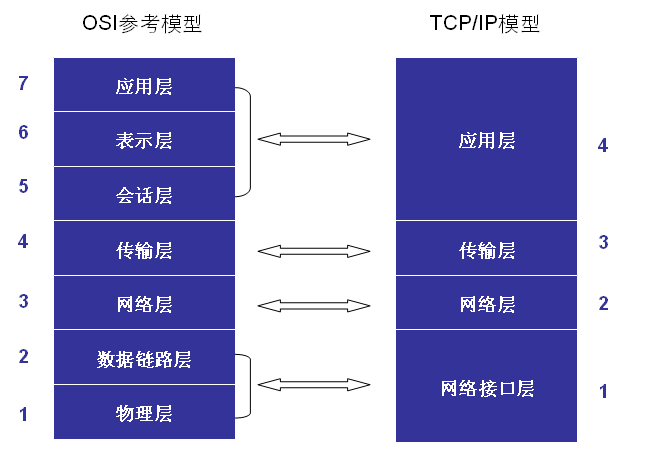
B/S模式相比C/S模式而言，由于它没有独立的客户端，使用标准浏览器作为客户端，其工作**开发量较小**。只需开发服务器端即可。另外由于其采用浏览器显示数据，因此移植性非常好，**不受平台限制**。如早期的偷菜游戏，在各个平台上都可以完美运行。

B/S模式的缺点也较明显。由于使用第三方浏览器，因此**网络应用支持受限**。另外，没有客户端放到对方主机上，**缓存数据不尽如人意**，从而传输数据量受到限制。应用的观感大打折扣。第三，必须与浏览器一样，采用标准http协议进行通信，**协议选择不灵活**。

因此在开发过程中，模式的选择由上述各自的特点决定。根据实际需求选择应用程序设计模式。

## 分层模型

### OSI七层模型

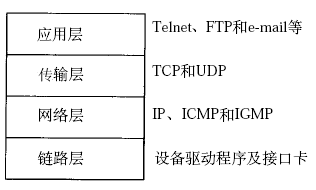


OSI模型

1. **物理层**：主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。它的主要作用是传输比特流（就是由1、0转化为电流强弱来进行传输，到达目的地后再转化为1、0，也就是我们常说的数模转换与模数转换）。这一层的数据叫做比特。
2. **数据链路层**：定义了如何让格式化数据以帧为单位进行传输，以及如何让控制对物理介质的访问。这一层通常还提供错误检测和纠正，以确保数据的可靠传输。如：串口通信中使用到的115200、8、N、1
3. **网络层**：在位于不同地理位置的网络中的两个主机系统之间提供连接和路径选择。Internet的发展使得从世界各站点访问信息的用户数大大增加，而网络层正是管理这种连接的层。
4. **传输层**：定义了一些传输数据的协议和端口号（WWW端口80等），如：TCP（传输控制协议，传输效率低，可靠性强，用于传输可靠性要求高，数据量大的数据），UDP（用户数据报协议，与TCP特性恰恰相反，用于传输可靠性要求不高，数据量小的数据，如QQ聊天数据就是通过这种方式传输的）。 主要是将从下层接收的数据进行分段和传输，到达目的地址后再进行重组。常常把这一层数据叫做段。
5. **会话层**：通过传输层(端口号：传输端口与接收端口)建立数据传输的通路。主要在你的系统之间发起会话或者接受会话请求（设备之间需要互相认识可以是IP也可以是MAC或者是主机名）。
6. **表示层**：可确保一个系统的应用层所发送的信息可以被另一个系统的应用层读取。例如，PC程序与另一台计算机进行通信，其中一台计算机使用扩展二一十进制交换码(EBCDIC)，而另一台则使用美国信息交换标准码（ASCII）来表示相同的字符。如有必要，表示层会通过使用一种通格式来实现多种数据格式之间的转换。
7. **应用层**：是最靠近用户的OSI层。这一层为用户的应用程序（例如电子邮件、文件传输和终端仿真）提供网络服务。

### TCP/IP四层模型

TCP/IP网络协议栈分为应用层（Application）、传输层（Transport）、网络层（Network）和链路层（Link）四层。如下图所示：

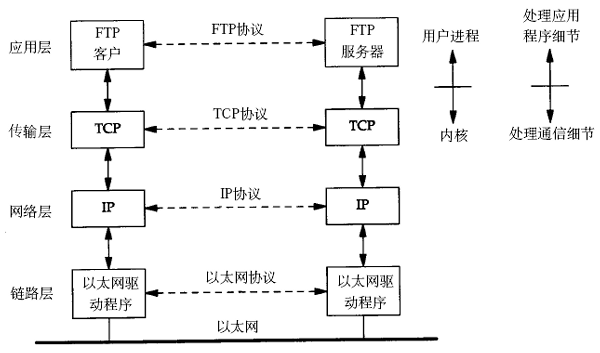


TCP/IP模型

一般在应用开发过程中，讨论最多的是TCP/IP模型。

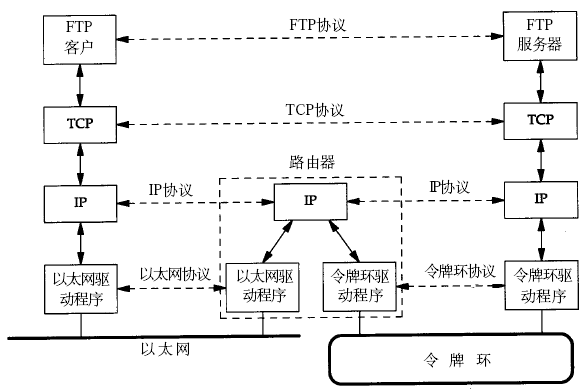
## 通信过程

两台计算机通过TCP/IP协议通讯的过程如下所示：



TCP/IP通信过程

上图对应两台计算机在同一网段中的情况，如果两台计算机在不同的网段中，那么数据从一台计算机到另一台计算机传输过程中要经过一个或多个路由器，如下图所示：



跨路由通信

链路层有以太网、令牌环网等标准，链路层负责网卡设备的驱动、帧同步（即从网线上检测到什么信号算作新帧的开始）、冲突检测（如果检测到冲突就自动重发）、数据差错校验等工作。交换机是工作在链路层的网络设备，可以在不同的链路层网络之间转发数据帧（比如十兆以太网和百兆以太网之间、以太网和令牌环网之间），由于不同链路层的帧格式不同，交换机要将进来的数据包拆掉链路层首部重新封装之后再转发。

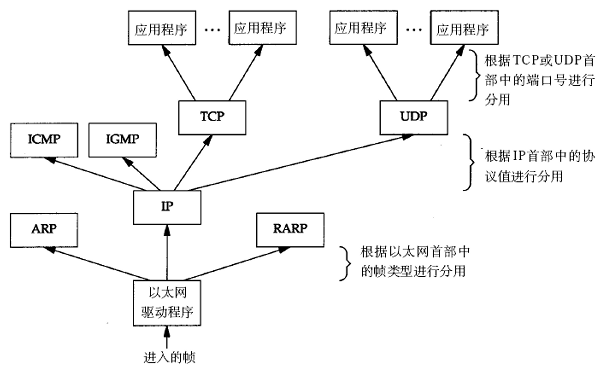
网络层的IP协议是构成Internet的基础。Internet上的主机通过IP地址来标识，Inter-net上有大量路由器负责根据IP地址选择合适的路径转发数据包，数据包从Internet上的源主机到目的主机往往要经过十多个路由器。路由器是工作在第三层的网络设备，同时兼有交换机的功能，可以在不同的链路层接口之间转发数据包，因此路由器需要将进来的数据包拆掉网络层和链路层两层首部并重新封装。IP协议不保证传输的可靠性，数据包在传输过程中可能丢失，可靠性可以在上层协议或应用程序中提供支持。

网络层负责点到点（ptop，point-to-point）的传输（这里的“点”指主机或路由器），而传输层负责端到端（etoe，end-to-end）的传输（这里的“端”指源主机和目的主机）。传输层可选择TCP或UDP协议。

TCP是一种面向连接的、可靠的协议，有点像打电话，双方拿起电话互通身份之后就建立了连接，然后说话就行了，这边说的话那边保证听得到，并且是按说话的顺序听到的，说完话挂机断开连接。也就是说TCP传输的双方需要首先建立连接，之后由TCP协议保证数据收发的可靠性，丢失的数据包自动重发，上层应用程序收到的总是可靠的数据流，通讯之后关闭连接。

UDP是无连接的传输协议，不保证可靠性，有点像寄信，信写好放到邮筒里，既不能保证信件在邮递过程中不会丢失，也不能保证信件寄送顺序。使用UDP协议的应用程序需要自己完成丢包重发、消息排序等工作。

目的主机收到数据包后，如何经过各层协议栈最后到达应用程序呢？其过程如下图所示：



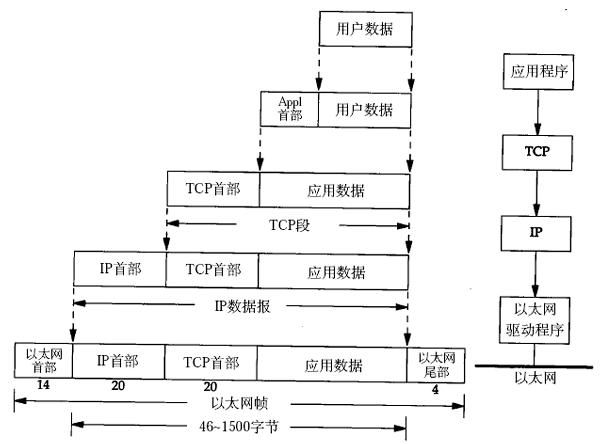
以太网驱动程序首先根据以太网首部中的“上层协议”字段确定该数据帧的有效载荷（payload，指除去协议首部之外实际传输的数据）是IP、ARP还是RARP协议的数据报，然后交给相应的协议处理。假如是IP数据报，IP协议再根据IP首部中的“上层协议”字段确定该数据报的有效载荷是TCP、UDP、ICMP还是IGMP，然后交给相应的协议处理。假如是TCP段或UDP段，TCP或UDP协议再根据TCP首部或UDP首部的“端口号”字段确定应该将应用层数据交给哪个用户进程。IP地址是标识网络中不同主机的地址，而端口号就是同一台主机上标识不同进程的地址，IP地址和端口号合起来标识网络中唯一的进程。

虽然IP、ARP和RARP数据报都需要以太网驱动程序来封装成帧，但是从功能上划分，ARP和RARP属于链路层，IP属于网络层。虽然ICMP、IGMP、TCP、UDP的数据都需要IP协议来封装成数据报，但是从功能上划分，ICMP、IGMP与IP同属于网络层，TCP和UDP属于传输层。

## 协议格式

### 数据包封装

传输层及其以下的机制由内核提供，应用层由用户进程提供（后面将介绍如何使用socket API编写应用程序），应用程序对通讯数据的含义进行解释，而传输层及其以下处理通讯的细节，将数据从一台计算机通过一定的路径发送到另一台计算机。应用层数据通过协议栈发到网络上时，每层协议都要加上一个数据首部（header），称为封装（Encapsulation），如下图所示：

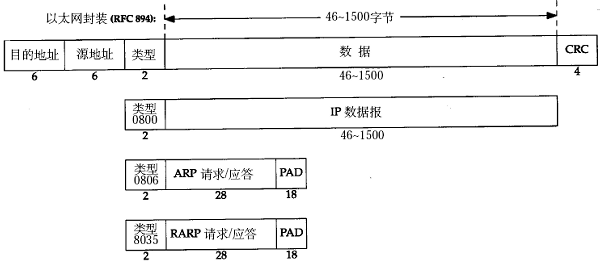


TCP/TP数据包封装

不同的协议层对数据包有不同的称谓，在传输层叫做段（segment），在网络层叫做数据报（datagram），在链路层叫做帧（frame）。数据封装成帧后发到传输介质上，到达目的主机后每层协议再剥掉相应的首部，最后将应用层数据交给应用程序处理。

### 以太网帧格式

以太网的帧格式如下所示：



以太网帧格式

其中的源地址和目的地址是指网卡的硬件地址（也叫MAC地址），长度是48位，是在网卡出厂时固化的。可在shell中使用ifconfig命令查看，“HWaddr 00:15:F2:14:9E:3F”部分就是硬件地址。协议字段有三种值，分别对应IP、ARP、RARP。帧尾是CRC校验码。

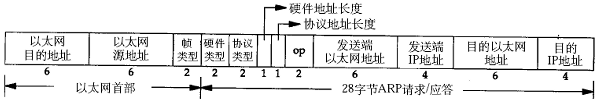
以太网帧中的数据长度规定最小46字节，最大1500字节，ARP和RARP数据包的长度不够46字节，要在后面补填充位。**最大值1500称为以太网的最大传输单元（MTU）**，不同的网络类型有不同的MTU，如果一个数据包从以太网路由到拨号链路上，数据包长度大于拨号链路的MTU，则需要对数据包进行分片（fragmentation）。ifconfig命令输出中也有“MTU:1500”。注意，MTU这个概念指数据帧中有效载荷的最大长度，不包括帧头长度。

### ARP数据报格式

在网络通讯时，源主机的应用程序知道目的主机的IP地址和端口号，却不知道目的主机的硬件地址，而数据包首先是被网卡接收到再去处理上层协议的，如果接收到的数据包的硬件地址与本机不符，则直接丢弃。因此在通讯前必须获得目的主机的硬件地址。ARP协议就起到这个作用。源主机发出ARP请求，询问“IP地址是192.168.0.1的主机的硬件地址是多少”，并将这个请求广播到本地网段（以太网帧首部的硬件地址填FF:FF:FF:FF:FF:FF表示广播），目的主机接收到广播的ARP请求，发现其中的IP地址与本机相符，则发送一个ARP应答数据包给源主机，将自己的硬件地址填写在应答包中。

每台主机都维护一个ARP缓存表，可以用arp -a命令查看。缓存表中的表项有过期时间（一般为20分钟），如果20分钟内没有再次使用某个表项，则该表项失效，下次还要发ARP请求来获得目的主机的硬件地址。想一想，为什么表项要有过期时间而不是一直有效？

ARP数据报的格式如下所示：



ARP数据报格式

源MAC地址、目的MAC地址在以太网首部和ARP请求中各出现一次，对于链路层为以太网的情况是多余的，但如果链路层是其它类型的网络则有可能是必要的。硬件类型指链路层网络类型，1为以太网，协议类型指要转换的地址类型，0x0800为IP地址，后面两个地址长度对于以太网地址和IP地址分别为6和4（字节），op字段为1表示ARP请求，op字段为2表示ARP应答。

看一个具体的例子。

请求帧如下（为了清晰在每行的前面加了字节计数，每行16个字节）：

以太网首部（14字节）

0000: ff ff ff ff ff ff 00 05 5d 61 58 a8 08 06

ARP帧（28字节）

0000: 00 01

0010: 08 00 06 04 00 01 00 05 5d 61 58 a8 c0 a8 00 37

0020: 00 00 00 00 00 00 c0 a8 00 02

填充位（18字节）

0020: 00 77 31 d2 50 10

0030: fd 78 41 d3 00 00 00 00 00 00 00 00

以太网首部：目的主机采用广播地址，源主机的MAC地址是00:05:5d:61:58:a8，上层协议类型0x0806表示ARP。

ARP帧：硬件类型0x0001表示以太网，协议类型0x0800表示IP协议，硬件地址（MAC地址）长度为6，协议地址（IP地址）长度为4，op为0x0001表示请求目的主机的MAC地址，源主机MAC地址为00:05:5d:61:58:a8，源主机IP地址为c0 a8 00 37（192.168.0.55），目的主机MAC地址全0待填写，目的主机IP地址为c0 a8 00 02（192.168.0.2）。

由于以太网规定最小数据长度为46字节，ARP帧长度只有28字节，因此有18字节填充位，填充位的内容没有定义，与具体实现相关。

应答帧如下：

以太网首部

0000: 00 05 5d 61 58 a8 00 05 5d a1 b8 40 08 06

ARP帧

0000: 00 01

0010: 08 00 06 04 00 02 00 05 5d a1 b8 40 c0 a8 00 02

0020: 00 05 5d 61 58 a8 c0 a8 00 37

填充位

0020: 00 77 31 d2 50 10

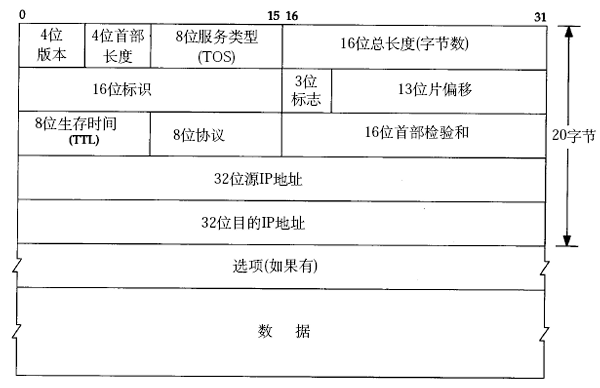
0030: fd 78 41 d3 00 00 00 00 00 00 00 00

以太网首部：目的主机的MAC地址是00:05:5d:61:58:a8，源主机的MAC地址是00:05:5d:a1:b8:40，上层协议类型0x0806表示ARP。

ARP帧：硬件类型0x0001表示以太网，协议类型0x0800表示IP协议，硬件地址（MAC地址）长度为6，协议地址（IP地址）长度为4，op为0x0002表示应答，源主机MAC地址为00:05:5d:a1:b8:40，源主机IP地址为c0 a8 00 02（192.168.0.2），目的主机MAC地址为00:05:5d:61:58:a8，目的主机IP地址为c0 a8 00 37（192.168.0.55）。

思考题：如果源主机和目的主机不在同一网段，ARP请求的广播帧无法穿过路由器，源主机如何与目的主机通信？

### IP段格式

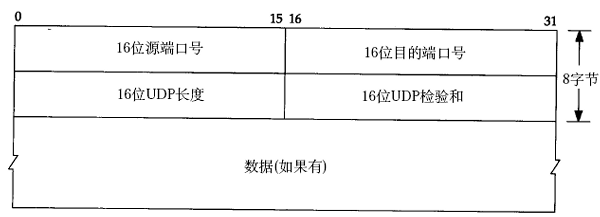


IP数据报格式

IP数据报的首部长度和数据长度都是可变长的，**但总是4字节的整数倍**。对于IPv4，4位版本字段是4。4位首部长度的数值是以4字节为单位的，**最小值为5**，也就是说首部长度最小是4x5=20字节，也就是不带任何选项的IP首部，4位能表示的最大值是15，也就是说**首部长度最大是60字节**。8位TOS字段有3个位用来指定IP数据报的优先级（目前已经废弃不用），还有4个位表示可选的服务类型（最小延迟、最大?吐量、最大可靠性、最小成本），还有一个位总是0。总长度是整个数据报（包括IP首部和IP层payload）的字节数。每传一个IP数据报，16位的标识加1，可用于分片和重新组装数据报。3位标志和13位片偏移用于分片。TTL（Time to live)是这样用的：源主机为数据包设定一个生存时间，比如64，每过一个路由器就把该值减1，如果减到0就表示路由已经太长了仍然找不到目的主机的网络，就丢弃该包，因此这个生存时间的单位不是秒，而是跳（hop）。协议字段指示上层协议是TCP、UDP、ICMP还是IGMP。然后是校验和，只校验IP首部，数据的校验由更高层协议负责。IPv4的IP地址长度为32位。

想一想，前面讲了以太网帧中的最小数据长度为46字节，不足46字节的要用填充字节补上，那么如何界定这46字节里前多少个字节是IP、ARP或RARP数据报而后面是填充字节？

### UDP数据报格式



UDP数据段

下面分析一帧基于UDP的TFTP协议帧。

以太网首部

0000: 00 05 5d 67 d0 b1 00 05 5d 61 58 a8 08 00

IP首部

0000: 45 00

0010: 00 53 93 25 00 00 80 11 25 ec c0 a8 00 37 c0 a8

0020: 00 01

UDP首部

0020： 05 d4 00 45 00 3f ac 40

TFTP协议

0020: 00 01 'c'':''\''q'

0030: 'w''e''r''q''.''q''w''e'00 'n''e''t''a''s''c''i'

0040: 'i'00 'b''l''k''s''i''z''e'00 '5''1''2'00 't''i'

0050: 'm''e''o''u''t'00 '1''0'00 't''s''i''z''e'00 '0'

0060: 00以太网首部：源MAC地址是00:05:5d:61:58:a8，目的MAC地址是00:05:5d:67:d0:b1，上层协议类型0x0800表示IP。

IP首部：每一个字节0x45包含4位版本号和4位首部长度，版本号为4，即IPv4，首部长度为5，说明IP首部不带有选项字段。服务类型为0，没有使用服务。16位总长度字段（包括IP首部和IP层payload的长度）为0x0053，即83字节，加上以太网首部14字节可知整个帧长度是97字节。IP报标识是0x9325，标志字段和片偏移字段设置为0x0000，就是DF=0允许分片，MF=0此数据报没有更多分片，没有分片偏移。TTL是0x80，也就是128。上层协议0x11表示UDP协议。IP首部校验和为0x25ec，源主机IP是c0 a8 00 37（192.168.0.55），目的主机IP是c0 a8 00 01（192.168.0.1）。

UDP首部：源端口号0x05d4（1492）是客户端的端口号，目的端口号0x0045（69）是TFTP服务的well-known端口号。UDP报长度为0x003f，即63字节，包括UDP首部和UDP层pay-load的长度。UDP首部和UDP层payload的校验和为0xac40。

TFTP是基于文本的协议，各字段之间用字节0分隔，开头的00 01表示请求读取一个文件，接下来的各字段是：

c:\qwerq.qwe

netascii

blksize 512

timeout 10

tsize 0

一般的网络通信都是像TFTP协议这样，通信的双方分别是客户端和服务器，客户端主动发起请求（上面的例子就是客户端发起的请求帧），而服务器被动地等待、接收和应答请求。客户端的IP地址和端口号唯一标识了该主机上的TFTP客户端进程，服务器的IP地址和端口号唯一标识了该主机上的TFTP服务进程，由于客户端是主动发起请求的一方，它必须知道服务器的IP地址和TFTP服务进程的端口号，所以，一些常见的网络协议有默认的服务器端口，例如HTTP服务默认TCP协议的80端口，FTP服务默认TCP协议的21端口，TFTP服务默认UDP协议的69端口（如上例所示）。在使用客户端程序时，必须指定服务器的主机名或IP地址，如果不明确指定端口号则采用默认端口，请读者查阅ftp、tftp等程序的man page了解如何指定端口号。/etc/services中列出了所有well-known的服务端口和对应的传输层协议，这是由IANA（Internet Assigned Numbers Authority）规定的，其中有些服务既可以用TCP也可以用UDP，为了清晰，IANA规定这样的服务采用相同的TCP或UDP默认端口号，而另外一些TCP和UDP的相同端口号却对应不同的服务。

很多服务有well-known的端口号，然而客户端程序的端口号却不必是well-known的，往往是每次运行客户端程序时由系统自动分配一个空闲的端口号，用完就释放掉，称为ephemeral的端口号，想想这是为什么？

前面提过，UDP协议不面向连接，也不保证传输的可靠性，例如：

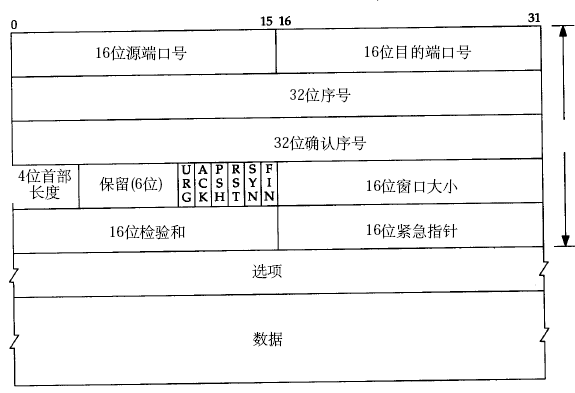
发送端的UDP协议层只管把应用层传来的数据封装成段交给IP协议层就算完成任务了，如果因为网络故障该段无法发到对方，UDP协议层也不会给应用层返回任何错误信息。

接收端的UDP协议层只管把收到的数据根据端口号交给相应的应用程序就算完成任务了，如果发送端发来多个数据包并且在网络上经过不同的路由，到达接收端时顺序已经错乱了，UDP协议层也不保证按发送时的顺序交给应用层。

通常接收端的UDP协议层将收到的数据放在一个固定大小的缓冲区中等待应用程序来提取和处理，如果应用程序提取和处理的速度很慢，而发送端发送的速度很快，就会丢失数据包，UDP协议层并不报告这种错误。

因此，使用UDP协议的应用程序必须考虑到这些可能的问题并实现适当的解决方案，例如等待应答、超时重发、为数据包编号、流量控制等。一般使用UDP协议的应用程序实现都比较简单，只是发送一些对可靠性要求不高的消息，而不发送大量的数据。例如，基于UDP的TFTP协议一般只用于传送小文件（所以才叫trivial的ftp），而基于TCP的FTP协议适用于 各种文件的传输。TCP协议又是如何用面向连接的服务来代替应用程序解决传输的可靠性问题呢。

### TCP数据报格式



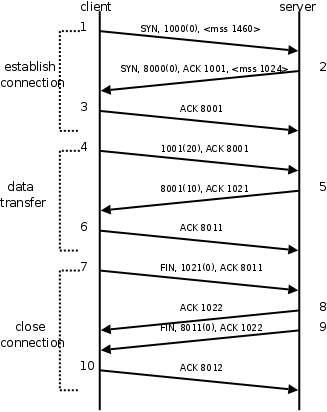
TCP数据段

与UDP协议一样也有源端口号和目的端口号，通讯的双方由IP地址和端口号标识。32位序号、32位确认序号、窗口大小稍后详细解释。4位首部长度和IP协议头类似，表示TCP协议头的长度，以4字节为单位，因此TCP协议头最长可以是4x15=60字节，如果没有选项字段，TCP协议头最短20字节。URG、ACK、PSH、RST、SYN、FIN是六个控制位，本节稍后将解释SYN、ACK、FIN、RST四个位，其它位的解释从略。16位检验和将TCP协议头和数据都计算在内。紧急指针和各种选项的解释从略。

## TCP协议

### TCP通信时序

下图是一次TCP通讯的时序图。TCP连接建立断开。包含大家熟知的**三次握手**和四次握手。



TCP通讯时序

在这个例子中，首先客户端主动发起连接、发送请求，然后服务器端响应请求，然后客户端主动关闭连接。两条竖线表示通讯的两端，从上到下表示时间的先后顺序，注意，数据从一端传到网络的另一端也需要时间，所以图中的箭头都是斜的。双方发送的段按时间顺序编号为1-10，各段中的主要信息在箭头上标出，例如段2的箭头上标着SYN, 8000(0), ACK1001, ，表示该段中的SYN位置1，32位序号是8000，该段不携带有效载荷（数据字节数为0），ACK位置1，32位确认序号是1001，带有一个mss（Maximum Segment Size，最大报文长度）选项值为1024。

建立连接（三次握手）的过程：

1. 客户端发送一个带SYN标志的TCP报文到服务器。这是三次握手过程中的段1。

客户端发出段1，SYN位表示连接请求。序号是1000，这个序号在网络通讯中用作临时的地址，每发一个数据字节，这个序号要加1，这样在接收端可以根据序号排出数据包的正确顺序，也可以发现丢包的情况，另外，规定SYN位和FIN位也要占一个序号，这次虽然没发数据，但是由于发了SYN位，因此下次再发送应该用序号1001。mss表示最大段尺寸，如果一个段太大，封装成帧后超过了链路层的最大帧长度，就必须在IP层分片，为了避免这种情况，客户端声明自己的最大段尺寸，建议服务器端发来的段不要超过这个长度。

1. 服务器端回应客户端，是三次握手中的第2个报文段，同时带ACK标志和SYN标志。它表示对刚才客户端SYN的回应；同时又发送SYN给客户端，询问客户端是否准备好进行数据通讯。

服务器发出段2，也带有SYN位，同时置ACK位表示确认，确认序号是1001，表示“我接收到序号1000及其以前所有的段，请你下次发送序号为1001的段”，也就是应答了客户端的连接请求，同时也给客户端发出一个连接请求，同时声明最大尺寸为1024。

1. 客户必须再次回应服务器端一个ACK报文，这是报文段3。

客户端发出段3，对服务器的连接请求进行应答，确认序号是8001。在这个过程中，客户端和服务器分别给对方发了连接请求，也应答了对方的连接请求，其中服务器的请求和应答在一个段中发出，因此一共有三个段用于建立连接，称为“三方握手（three-way-handshake）”。在建立连接的同时，双方协商了一些信息，例如双方发送序号的初始值、最大段尺寸等。

在TCP通讯中，如果一方收到另一方发来的段，读出其中的目的端口号，发现本机并没有任何进程使用这个端口，就会应答一个包含RST位的段给另一方。例如，服务器并没有任何进程使用8080端口，我们却用telnet客户端去连接它，服务器收到客户端发来的SYN段就会应答一个RST段，客户端的telnet程序收到RST段后报告错误Connection refused：

$ telnet 192.168.0.200 8080

Trying 192.168.0.200...

telnet: Unable to connect to remote host: Connection refused

数据传输的过程：

1. 客户端发出段4，包含从序号1001开始的20个字节数据。
2. 服务器发出段5，确认序号为1021，对序号为1001-1020的数据表示确认收到，同时请求发送序号1021开始的数据，服务器在应答的同时也向客户端发送从序号8001开始的10个字节数据，这称为piggyback。
3. 客户端发出段6，对服务器发来的序号为8001-8010的数据表示确认收到，请求发送序号8011开始的数据。

在数据传输过程中，ACK和确认序号是非常重要的，应用程序交给TCP协议发送的数据会暂存在TCP层的发送缓冲区中，发出数据包给对方之后，只有收到对方应答的ACK段才知道该数据包确实发到了对方，可以从发送缓冲区中释放掉了，如果因为网络故障丢失了数据包或者丢失了对方发回的ACK段，经过等待超时后TCP协议自动将发送缓冲区中的数据包重发。

关闭连接（四次握手）的过程：

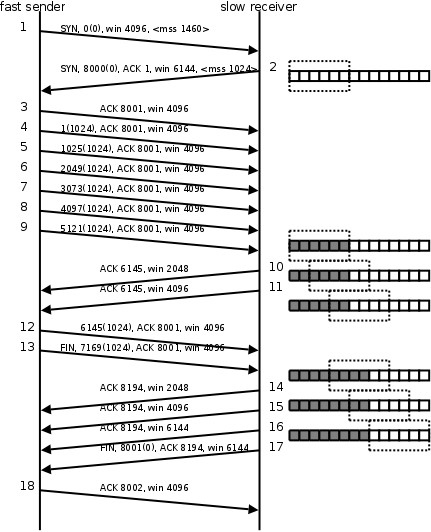
由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

1. 客户端发出段7，FIN位表示关闭连接的请求。
2. 服务器发出段8，应答客户端的关闭连接请求。
3. 服务器发出段9，其中也包含FIN位，向客户端发送关闭连接请求。
4. 客户端发出段10，应答服务器的关闭连接请求。

建立连接的过程是三方握手，而关闭连接通常需要4个段，服务器的应答和关闭连接请求通常不合并在一个段中，因为有连接半关闭的情况，这种情况下客户端关闭连接之后就不能再发送数据给服务器了，但是服务器还可以发送数据给客户端，直到服务器也关闭连接为止。

### 滑动窗口 (TCP流量控制)

介绍UDP时我们描述了这样的问题：如果发送端发送的速度较快，接收端接收到数据后处理的速度较慢，而接收缓冲区的大小是固定的，就会丢失数据。TCP协议通过“滑动窗口（Sliding Window）”机制解决这一问题。看下图的通讯过程：



滑动窗口

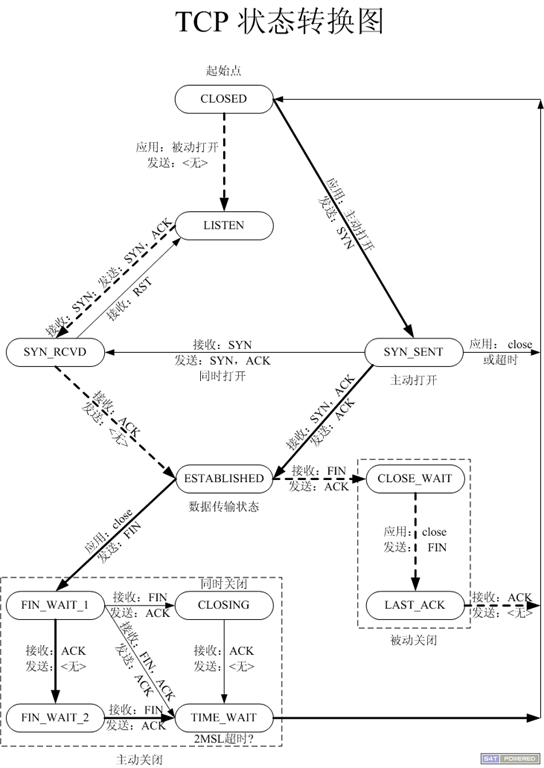
1. 发送端发起连接，声明最大段尺寸是1460，初始序号是0，窗口大小是4K，表示“我的接收缓冲区还有4K字节空闲，你发的数据不要超过4K”。接收端应答连接请求，声明最大段尺寸是1024，初始序号是8000，窗口大小是6K。发送端应答，三方握手结束。
2. 发送端发出段4-9，每个段带1K的数据，发送端根据窗口大小知道接收端的缓冲区满了，因此停止发送数据。
3. 接收端的应用程序提走2K数据，接收缓冲区又有了2K空闲，接收端发出段10，在应答已收到6K数据的同时声明窗口大小为2K。
4. 接收端的应用程序又提走2K数据，接收缓冲区有4K空闲，接收端发出段11，重新声明窗口大小为4K。
5. 发送端发出段12-13，每个段带2K数据，段13同时还包含FIN位。
6. 接收端应答接收到的2K数据（6145-8192），再加上FIN位占一个序号8193，因此应答序号是8194，连接处于半关闭状态，接收端同时声明窗口大小为2K。
7. 接收端的应用程序提走2K数据，接收端重新声明窗口大小为4K。
8. 接收端的应用程序提走剩下的2K数据，接收缓冲区全空，接收端重新声明窗口大小为6K。
9. 接收端的应用程序在提走全部数据后，决定关闭连接，发出段17包含FIN位，发送端应答，连接完全关闭。

上图在接收端用小方块表示1K数据，实心的小方块表示已接收到的数据，虚线框表示接收缓冲区，因此套在虚线框中的空心小方块表示窗口大小，从图中可以看出，随着应用程序提走数据，虚线框是向右滑动的，因此称为滑动窗口。

从这个例子还可以看出，发送端是一K一K地发送数据，而接收端的应用程序可以两K两K地提走数据，当然也有可能一次提走3K或6K数据，或者一次只提走几个字节的数据。也就是说，应用程序所看到的数据是一个整体，或说是一个流（stream），在底层通讯中这些数据可能被拆成很多数据包来发送，但是一个数据包有多少字节对应用程序是不可见的，因此TCP协议是面向流的协议。而UDP是面向消息的协议，每个UDP段都是一条消息，应用程序必须以消息为单位提取数据，不能一次提取任意字节的数据，这一点和TCP是很不同的。

### TCP状态转换

这个图N多人都知道，它排除和定位网络或系统故障时大有帮助，但是怎样牢牢地将这张图刻在脑中呢？那么你就一定要对这张图的每一个状态，及转换的过程有深刻的认识，不能只停留在一知半解之中。下面对这张图的11种状态详细解析一下，以便加强记忆！不过在这之前，先回顾一下TCP建立连接的三次握手过程，以及 关闭连接的四次握手过程。



TCP状态转换图

**CLOSED：**表示初始状态。

**LISTEN：**该状态表示服务器端的某个SOCKET处于监听状态，可以接受连接。

**SYN\_SENT：**这个状态与SYN\_RCVD遥相呼应，当客户端SOCKET执行CONNECT连接时，它首先发送SYN报文，随即进入到了SYN\_SENT状态，并等待服务端的发送三次握手中的第2个报文。SYN\_SENT状态表示客户端已发送SYN报文。

**SYN\_RCVD:** 该状态表示接收到SYN报文，在正常情况下，这个状态是服务器端的SOCKET在建立TCP连接时的三次握手会话过程中的一个中间状态，很短暂。此种状态时，当收到客户端的ACK报文后，会进入到ESTABLISHED状态。

**ESTABLISHED：**表示连接已经建立。

**FIN\_WAIT\_1:** FIN\_WAIT\_1和FIN\_WAIT\_2状态的真正含义都是表示等待对方的FIN报文。区别是：

FIN\_WAIT\_1状态是当socket在ESTABLISHED状态时，想主动关闭连接，向对方发送了FIN报文，此时该socket进入到FIN\_WAIT\_1状态。

FIN\_WAIT\_2状态是当对方回应ACK后，该socket进入到FIN\_WAIT\_2状态，正常情况下，对方应马上回应ACK报文，所以FIN\_WAIT\_1状态一般较难见到，而FIN\_WAIT\_2状态可用netstat看到。

**FIN\_WAIT\_2：主动关闭链接的一方，发出FIN收到ACK以后进入该状态。称之为半连接或半关闭状态。**该状态下的socket只能接收数据，不能发。

**TIME\_WAIT:** 表示收到了对方的FIN报文，并发送出了ACK报文，等2MSL后即可回到CLOSED可用状态。如果FIN\_WAIT\_1状态下，收到对方同时带 FIN标志和ACK标志的报文时，可以直接进入到TIME\_WAIT状态，而无须经过FIN\_WAIT\_2状态。

**CLOSING:** 这种状态较特殊，属于一种较罕见的状态。正常情况下，当你发送FIN报文后，按理来说是应该先收到（或同时收到）对方的 ACK报文，再收到对方的FIN报文。但是CLOSING状态表示你发送FIN报文后，并没有收到对方的ACK报文，反而却也收到了对方的FIN报文。什么情况下会出现此种情况呢？如果双方几乎在同时close一个SOCKET的话，那么就出现了双方同时发送FIN报文的情况，也即会出现CLOSING状态，表示双方都正在关闭SOCKET连接。

**CLOSE\_WAIT:** 此种状态表示在等待关闭。当对方关闭一个SOCKET后发送FIN报文给自己，系统会回应一个ACK报文给对方，此时则进入到CLOSE\_WAIT状态。接下来呢，察看是否还有数据发送给对方，如果没有可以 close这个SOCKET，发送FIN报文给对方，即关闭连接。所以在CLOSE\_WAIT状态下，需要关闭连接。

**LAST\_ACK:** 该状态是被动关闭一方在发送FIN报文后，最后等待对方的ACK报文。当收到ACK报文后，即可以进入到CLOSED可用状态。

### 半关闭

当TCP链接中A发送FIN请求关闭，B端回应ACK后（A端进入FIN\_WAIT\_2状态），B没有立即发送FIN给A时，A方处在半链接状态，此时A可以接收B发送的数据，但是A已不能再向B发送数据。

从程序的角度，可以使用API来控制实现半连接状态。

#include <sys/socket.h>

int shutdown(int sockfd, int how);

sockfd: 需要关闭的socket的描述符

how: 允许为shutdown操作选择以下几种方式:

SHUT\_RD(0)： 关闭sockfd上的读功能，此选项将不允许sockfd进行读操作。

该套接字**不再接受数据**，任何当前在套接字接受缓冲区的数据将被无声的丢弃掉。

SHUT\_WR(1): 关闭sockfd的写功能，此选项将不允许sockfd进行写操作。进程不能在对此套接字发出写操作。

SHUT\_RDWR(2): 关闭sockfd的读写功能。相当于调用shutdown两次：首先是以SHUT\_RD,然后以SHUT\_WR。

使用close中止一个连接，但它只是减少描述符的引用计数，并不直接关闭连接，只有当描述符的引用计数为0时才关闭连接。

**shutdown不考虑描述符的引用计数，直接关闭描述符**。也可选择中止一个方向的连接，只中止读或只中止写。

注意:

1. 如果有多个进程共享一个套接字，close每被调用一次，计数减1，直到计数为0时，也就是所用进程都调用了close，套接字将被释放。
2. 在多进程中如果一个进程调用了shutdown(sfd, SHUT\_RDWR)后，其它的进程将无法进行通信。但，如果一个进程close(sfd)将不会影响到其它进程。

### 2MSL

2MSL (Maximum Segment Lifetime) TIME\_WAIT状态的存在有两个理由：

（1）**让4次握手关闭流程更加可靠**；4次握手的最后一个ACK是是由主动关闭方发送出去的，若这个ACK丢失，被动关闭方会再次发一个FIN过来。若主动关闭方能够保持一个2MSL的TIME\_WAIT状态，则有更大的机会让丢失的ACK被再次发送出去。

（2）防止lost duplicate对后续新建正常链接的传输造成破坏。lost uplicate在实际的网络中非常常见，经常是由于路由器产生故障，路径无法收敛，导致一个packet在路由器A，B，C之间做类似死循环的跳转。IP头部有个TTL，限制了一个包在网络中的最大跳数，因此这个包有两种命运，要么最后TTL变为0，在网络中消失；要么TTL在变为0之前路由器路径收敛，它凭借剩余的TTL跳数终于到达目的地。但非常可惜的是TCP通过超时重传机制在早些时候发送了一个跟它一模一样的包，并先于它达到了目的地，因此它的命运也就注定被TCP协议栈抛弃。

另外一个概念叫做incarnation connection，指跟上次的socket pair一摸一样的新连接，叫做incarnation of previous connection。lost uplicate加上incarnation connection，则会对我们的传输造成致命的错误。

TCP是流式的，所有包到达的顺序是不一致的，依靠序列号由TCP协议栈做顺序的拼接；假设一个incarnation connection这时收到的seq=1000, 来了一个lost duplicate为seq=1000，len=1000, 则TCP认为这个lost duplicate合法，并存放入了receive buffer，导致传输出现错误。通过一个2MSL TIME\_WAIT状态，确保所有的lost duplicate都会消失掉，避免对新连接造成错误。

该状态为什么设计在**主动关闭这一方**：

（1）发最后ACK的是主动关闭一方。

（2）只要有一方保持TIME\_WAIT状态，就能起到避免incarnation connection在2MSL内的重新建立，不需要两方都有。

如何正确对待2MSL TIME\_WAIT?

RFC要求socket pair在处于TIME\_WAIT时，不能再起一个incarnation connection。但绝大部分TCP实现，强加了更为严格的限制。在2MSL等待期间，socket中使用的本地端口在默认情况下不能再被使用。

若A 10.234.5.5 : 1234和B 10.55.55.60 : 6666建立了连接，A主动关闭，那么在A端只要port为1234，无论对方的port和ip是什么，都不允许再起服务。这甚至比RFC限制更为严格，RFC仅仅是要求socket pair不一致，而实现当中只要这个port处于TIME\_WAIT，就不允许起连接。这个限制对主动打开方来说是无所谓的，因为一般用的是临时端口；但对于被动打开方，一般是server，就悲剧了，因为server一般是熟知端口。比如http，一般端口是80，不可能允许这个服务在2MSL内不能起来。

解决方案是给服务器的socket设置SO\_REUSEADDR选项，这样的话就算熟知端口处于TIME\_WAIT状态，在这个端口上依旧可以将服务启动。当然，虽然有了SO\_REUSEADDR选项，但sockt pair这个限制依旧存在。比如上面的例子，A通过SO\_REUSEADDR选项依旧在1234端口上起了监听，但这时我们若是从B通过6666端口去连它，TCP协议会告诉我们连接失败，原因为Address already in use.

RFC 793中规定MSL为2分钟，实际应用中常用的是30秒，1分钟和2分钟等。

RFC (Request For Comments)，是一系列以编号排定的文件。收集了有关因特网相关资讯，以及UNIX和因特网社群的[软件](http://baike.baidu.com/view/37.htm" \t "_blank)文件。

#### 程序设计中的问题

做一个测试，首先启动server，然后启动client，用Ctrl-C终止server，马上再运行server，运行结果：

itcast$ ./server

bind error: Address already in use

这是因为，虽然server的应用程序终止了，但TCP协议层的连接并没有完全断开，因此不能再次监听同样的server端口。我们用netstat命令查看一下：

itcast$ netstat -apn |grep 6666

tcp 1 0 192.168.1.11:38103 192.168.1.11:6666 CLOSE\_WAIT 3525/client

tcp 0 0 192.168.1.11:6666 192.168.1.11:38103 FIN\_WAIT2 -

server终止时，socket描述符会自动关闭并发FIN段给client，client收到FIN后处于CLOSE\_WAIT状态，但是client并没有终止，也没有关闭socket描述符，因此不会发FIN给server，因此server的TCP连接处于FIN\_WAIT2状态。

现在用Ctrl-C把client也终止掉，再观察现象：

itcast$ netstat -apn |grep 6666

tcp 0 0 192.168.1.11:6666 192.168.1.11:38104 TIME\_WAIT -

itcast$ ./server

bind error: Address already in use

client终止时自动关闭socket描述符，server的TCP连接收到client发的FIN段后处于TIME\_WAIT状态。TCP协议规定，**主动关闭连接的一方要处于TIME\_WAIT状态**，等待两个MSL（maximum segment lifetime）的时间后才能回到CLOSED状态，因为我们先Ctrl-C终止了server，所以server是主动关闭连接的一方，在TIME\_WAIT期间仍然不能再次监听同样的server端口。

MSL在RFC 1122中规定为两分钟，但是各操作系统的实现不同，在Linux上一般经过半分钟后就可以再次启动server了。至于为什么要规定TIME\_WAIT的时间，可参考UNP 2.7节。

#### 端口复用

在server的TCP连接没有完全断开之前不允许重新监听是不合理的。因为，TCP连接没有完全断开指的是connfd（127.0.0.1:6666）没有完全断开，而我们重新监听的是lis-tenfd（0.0.0.0:6666），虽然是占用同一个端口，但IP地址不同，connfd对应的是与某个客户端通讯的一个具体的IP地址，而listenfd对应的是wildcard address。解决这个问题的方法是使用setsockopt()设置socket描述符的选项SO\_REUSEADDR为1，表示允许创建端口号相同但IP地址不同的多个socket描述符。

在server代码的socket()和bind()调用之间插入如下代码：

int opt = 1;

setsockopt(listenfd, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt));

有关setsockopt可以设置的其它选项请参考UNP第7章。

### TCP异常断开

#### 心跳检测机制

在TCP网络通信中，经常会出现客户端和服务器之间的非正常断开，需要实时检测查询链接状态。常用的解决方法就是在程序中加入心跳机制。

Heart-Beat线程

这个是最常用的简单方法。在接收和发送数据时个人设计一个守护进程(线程)，定时发送Heart-Beat包，客户端/服务器收到该小包后，立刻返回相应的包即可检测对方是否实时在线。

该方法的好处是通用，但缺点就是会改变现有的通讯协议！大家一般都是使用业务层心跳来处理，主要是灵活可控。

UNIX网络编程不推荐使用SO\_KEEPALIVE来做心跳检测，还是在业务层以心跳包做检测比较好，也方便控制。

#### 设置TCP属性

SO\_KEEPALIVE 保持连接检测对方主机是否崩溃，避免（服务器）永远阻塞于TCP连接的输入。设置该选项后，如果2小时内在此套接口的任一方向都没有数据交换，TCP就自动给对方发一个保持存活探测分节(keepalive probe)。这是一个对方必须响应的TCP分节.它会导致以下三种情况：对方接收一切正常：以期望的ACK响应。2小时后，TCP将发出另一个探测分节。对方已崩溃且已重新启动：以RST响应。套接口的待处理错误被置为ECONNRESET，套接 口本身则被关闭。对方无任何响应：源自berkeley的TCP发送另外8个探测分节，相隔75秒一个，试图得到一个响应。在发出第一个探测分节11分钟 15秒后若仍无响应就放弃。套接口的待处理错误被置为ETIMEOUT，套接口本身则被关闭。如ICMP错误是“host unreachable(主机不可达)”，说明对方主机并没有崩溃，但是不可达，这种情况下待处理错误被置为EHOSTUNREACH。

根据上面的介绍我们可以知道对端以一种非优雅的方式断开连接的时候，我们可以设置SO\_KEEPALIVE属性使得我们在2小时以后发现对方的TCP连接是否依然存在。

keepAlive = 1;

setsockopt(listenfd, SOL\_SOCKET, SO\_KEEPALIVE, (void\*)&keepAlive, sizeof(keepAlive));

如果我们不能接受如此之长的等待时间，从TCP-Keepalive-HOWTO上可以知道一共有两种方式可以设置，一种是修改内核关于网络方面的 配置参数，另外一种就是SOL\_TCP字段的TCP\_KEEPIDLE， TCP\_KEEPINTVL， TCP\_KEEPCNT三个选项。

1. The tcp\_keepidle parameter specifies the interval of inactivity that causes TCP to generate a KEEPALIVE transmission for an application that requests them. tcp\_keepidle defaults to 14400 (two hours).

/\*开始首次KeepAlive探测前的TCP空闭时间 \*/

1. The tcp\_keepintvl parameter specifies the interval between the nine retriesthat are attempted if a KEEPALIVE transmission is not acknowledged. tcp\_keep ntvldefaults to 150 (75 seconds).

/\* 两次KeepAlive探测间的时间间隔 \*/

1. The tcp\_keepcnt option specifies the maximum number of keepalive probes tobe sent. The value of TCP\_KEEPCNT is an integer value between 1 and n, where n s the value of the systemwide tcp\_keepcnt parameter.

/\* 判定断开前的KeepAlive探测次数\*/

int keepIdle = 1000;

int keepInterval = 10;

int keepCount = 10;

Setsockopt(listenfd, SOL\_TCP, TCP\_KEEPIDLE, (void \*)&keepIdle, sizeof(keepIdle));

Setsockopt(listenfd, SOL\_TCP,TCP\_KEEPINTVL, (void \*)&keepInterval, sizeof(keepInterval));

Setsockopt(listenfd,SOL\_TCP, TCP\_KEEPCNT, (void \*)&keepCount, sizeof(keepCount));

SO\_KEEPALIVE设置空闲2小时才发送一个“保持存活探测分节”，不能保证实时检测。对于判断网络断开时间太长，对于需要及时响应的程序不太适应。

当然也可以修改时间间隔参数，但是会影响到所有打开此选项的套接口！关联了完成端口的socket可能会忽略掉该套接字选项。

## 网络名词术语解析

### 路由(route)

路由（名词）

数据包从源地址到目的地址所经过的路径，由一系列路由节点组成。

路由（动词）

某个路由节点为数据包选择投递方向的选路过程。

### 路由器工作原理

路由器（Router）是连接因特网中各局域网、广域网的设备，它会根据信道的情况自动选择和设定路由，以最佳路径，按前后顺序发送信号的设备。

传统地，路由器工作于OSI七层协议中的第三层，其主要任务是接收来自一个网络接口的数据包，根据其中所含的目的地址，决定转发到下一个目的地址。因此，路由器首先得在转发路由表中查找它的目的地址，若找到了目的地址，就在数据包的帧格前添加下一个MAC地址，同时IP数据包头的TTL（Time To Live）域也开始减数， 并重新计算校验和。当数据包被送到输出端口时，它需要按顺序等待，以便被传送到输出链路上。

路由器在工作时能够按照某种路由通信协议查找设备中的路由表。如果到某一特定节点有一条以上的路径，则基本预先确定的路由准则是选择最优（或最经济）的传输路径。由于各种网络段和其相互连接情况可能会因环境变化而变化，因此路由情况的信息一般也按所使用的路由信息协议的规定而定时更新。

网络中，每个路由器的基本功能都是按照一定的规则来动态地更新它所保持的路由表，以便保持路由信息的有效性。为了便于在网络间传送报文，路由器总是先按照预定的规则把较大的数据分解成适当大小的数据包，再将这些数据包分别通过相同或不同路径发送出去。当这些数据包按先后秩序到达目的地后，再把分解的数据包按照一定顺序包装成原有的报文形式。路由器的分层寻址功能是路由器的重要功能之一，该功能可以帮助具有很多节点站的网络来存储寻址信息，同时还能在网络间截获发送到远地网段的报文，起转发作用；选择最合理的路由，引导通信也是路由器基本功能；多协议路由器还可以连接使用不同通信协议的网络段，成为不同通信协议网络段之间的通信平台。

路由和交换之间的主要区别就是交换发生在OSI参考模型第二层（数据链路层），而路由发生在第三层，即网络层。这一区别决定了路由和交换在移动信息的过程 中需使用不同的控制信息，所以两者实现各自功能的方式是不同的。

### 路由表(Routing Table)

在计算机网络中，路由表或称路由择域信息库（RIB）是一个存储在路由器或者联网计算机中的电子表格（文件）或类数据库。路由表存储着指向特定网络地址的路径。

### 路由条目

路由表中的一行，每个条目主要由目的网络地址、子网掩码、下一跳地址、发送接口四部分组成，如果要发送的数据包的目的网络地址匹配路由表中的某一行，就按规定的接口发送到下一跳地址。

### 缺省路由条目

路由表中的最后一行，主要由下一跳地址和发送接口两部分组成，当目的地址与路由表中其它行都不匹配时，就按缺省路由条目规定的接口发送到下一跳地址。

### 路由节点

一个具有路由能力的主机或路由器，它维护一张路由表，通过查询路由表来决定向哪个接口发送数据包。

### 以太网交换机工作原理

以太网交换机是基于以太网传输数据的交换机，以太网采用共享总线型传输媒体方式的局域网。以太网交换机的结构是每个端口都直接与主机相连，并且一般都工作在全双工方式。交换机能同时连通许多对端口，使每一对相互通信的主机都能像独占通信媒体那样，进行无冲突地传输数据。

以太网交换机工作于OSI网络参考模型的第二层（即数据链路层），是一种基于MAC（Media Access Control，介质访问控制）地址识别、完成以太网数据帧转发的网络设备。

### hub工作原理

集线器实际上就是中继器的一种，其区别仅在于集线器能够提供更多的端口服务，所以集线器又叫多口中继器。

集线器功能是随机选出某一端口的设备，并让它独占全部带宽，与集线器的上联设备（交换机、路由器或服务器等）进行通信。从Hub的工作方式可以看出，它在网络中只起到信号放大和重发作用，其目的是扩大网络的传输范围，而不具备信号的定向传送能力，是—个标准的共享式设备。其次是Hub只与它的上联设备(如上层Hub、交换机或服务器)进行通信，同层的各端口之间不会直接进行通信，而是通过上联设备再将信息广播到所有端口上。 由此可见，即使是在同一Hub的不同两个端口之间进行通信，都必须要经过两步操作：

第一步是将信息上传到上联设备；

第二步是上联设备再将该信息广播到所有端口上。

### 半双工/全双工

Full-duplex（全双工）全双工是在通道中同时双向数据传输的能力。

Half-duplex（半双工）在通道中同时只能沿着一个方向传输数据。

### DNS服务器

DNS 是域名系统 (Domain Name System) 的缩写，是因特网的一项核心服务，它作为可以将域名和IP地址相互映射的一个分布式数据库，能够使人更方便的访问互联网，而不用去记住能够被机器直接读取的IP地址串。

它是由解析器以及域名服务器组成的。域名服务器是指保存有该网络中所有主机的域名和对应IP地址，并具有将域名转换为IP地址功能的服务器。

### 局域网(LAN)

local area network，一种覆盖一座或几座大楼、一个校园或者一个厂区等地理区域的小范围的计算机网。

1. 覆盖的地理范围较小，只在一个相对独立的局部范围内联，如一座或集中的建筑群内。
2. 使用专门铺设的传输介质进行联网，数据传输速率高（10Mb/s～10Gb/s）
3. 通信延迟时间短，可靠性较高
4. 局域网可以支持多种传输介质

### 广域网(WAN)

wide area network，一种用来实现不同地区的局域网或城域网的互连，可提供不同地区、城市和国家之间的计算机通信的远程计算机网。

覆盖的范围比局域网（LAN）和城域网（MAN）都广。广域网的通信子网主要使用分组交换技术。

广域网的通信子网可以利用公用分组交换网、卫星通信网和无线分组交换网，它将分布在不同地区的局域网或计算机系统互连起来，达到资源共享的目的。如互联网是世界范围内最大的广域网。

1. 适应大容量与突发性通信的要求；
2. 适应综合业务服务的要求；
3. 开放的设备接口与规范化的协议；
4. 完善的通信服务与网络管理。

### 端口

逻辑意义上的端口，一般是指TCP/IP协议中的端口，端口号的范围从0到65535，比如用于浏览网页服务的80端口，用于FTP服务的21端口等等。

1. 端口号小于256的定义为常用端口，服务器一般都是通过常用端口号来识别的。
2. 客户端只需保证该端口号在本机上是惟一的就可以了。客户端口号因存在时间很短暂又称临时端口号；
3. 大多数TCP/IP实现给临时端口号分配1024—5000之间的端口号。大于5000的端口号是为其他服务器预留的。

我们应该在自定义端口时，避免使用well-known的端口。如：80、21等等。

### MTU

MTU:通信术语 最大传输单元（Maximum Transmission Unit，MTU）

是指一种通信协议的某一层上面所能通过的最大数据包大小（以字节为单位）。最大传输单元这个参数通常与通信接口有关（网络接口卡、串口等）。

以下是一些协议的MTU：

FDDI协议：4352字节

**以太网（Ethernet）协议：1500字节**

PPPoE（ADSL）协议：1492字节

X.25协议（Dial Up/Modem）：576字节

Point-to-Point：4470字节

## 常见网络知识面试题

1. TCP如何建立链接
2. TCP如何通信
3. TCP如何关闭链接
4. 什么是滑动窗口
5. 什么是半关闭
6. 局域网内两台机器如何利用TCP/IP通信
7. internet上两台主机如何进行通信
8. 如何在internet上识别唯一一个进程

答：通过“IP地址+端口号”来区分不同的服务

1. 为什么说TCP是可靠的链接，UDP不可靠
2. 路由器和交换机的区别
3. 点到点，端到端

# Socket编程

## 套接字概念

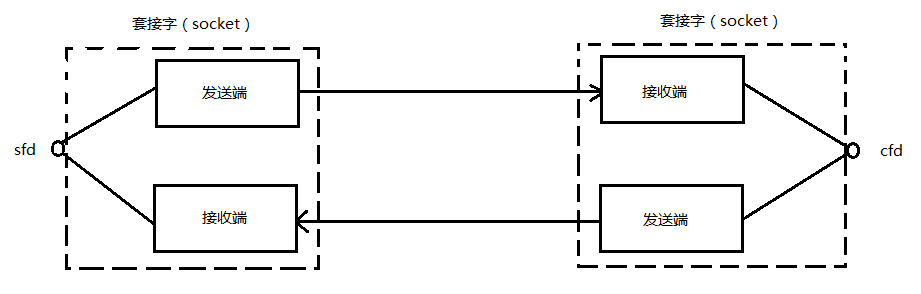
Socket本身有“插座”的意思，在Linux环境下，用于表示进程间网络通信的特殊文件类型。本质为内核借助缓冲区形成的伪文件。

既然是文件，那么理所当然的，我们可以使用文件描述符引用套接字。与管道类似的，Linux系统将其封装成文件的目的是为了统一接口，使得读写套接字和读写文件的操作一致。区别是管道主要应用于本地进程间通信，而套接字多应用于网络进程间数据的传递。

套接字的内核实现较为复杂，不宜在学习初期深入学习。

在TCP/IP协议中，“IP地址+TCP或UDP端口号”唯一标识网络通讯中的一个进程。“IP地址+端口号”就对应一个socket。欲建立连接的两个进程各自有一个socket来标识，那么这两个socket组成的socket pair就唯一标识一个连接。因此可以用Socket来描述网络连接的一对一关系。

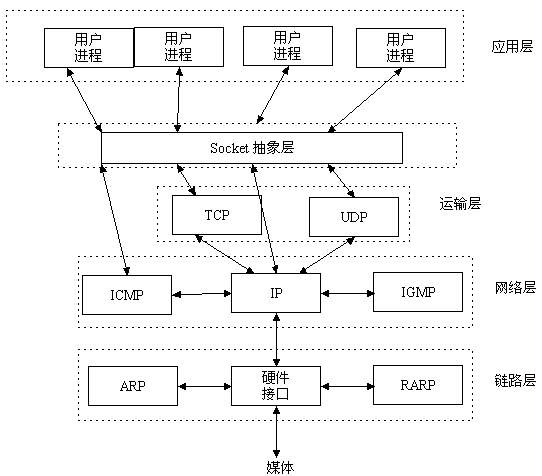
套接字通信原理如下图所示：



套接字通讯原理示意

**在网络通信中，套接字一定是成对出现的。**一端的发送缓冲区对应对端的接收缓冲区。我们使用同一个文件描述符索发送缓冲区和接收缓冲区。

TCP/IP协议最早在BSD UNIX上实现，为TCP/IP协议设计的应用层编程接口称为socket API。本章的主要内容是socket API，主要介绍TCP协议的函数接口，最后介绍UDP协议和UNIX Domain Socket的函数接口。



网络编程接口

## 预备知识

### 网络字节序

我们已经知道，内存中的多字节数据相对于内存地址有大端和小端之分，磁盘文件中的多字节数据相对于文件中的偏移地址也有大端小端之分。网络数据流同样有大端小端之分，那么如何定义网络数据流的地址呢？发送主机通常将发送缓冲区中的数据按内存地址从低到高的顺序发出，接收主机把从网络上接到的字节依次保存在接收缓冲区中，也是按内存地址从低到高的顺序保存，因此，网络数据流的地址应这样规定：先发出的数据是低地址，后发出的数据是高地址。

TCP/IP协议规定，网络数据流应采用大端字节序，即低地址高字节。例如上一节的UDP段格式，地址0-1是16位的源端口号，如果这个端口号是1000（0x3e8），则地址0是0x03，地址1是0xe8，也就是先发0x03，再发0xe8，这16位在发送主机的缓冲区中也应该是低地址存0x03，高地址存0xe8。但是，如果发送主机是小端字节序的，这16位被解释成0xe803，而不是1000。因此，发送主机把1000填到发送缓冲区之前需要做字节序的转换。同样地，接收主机如果是小端字节序的，接到16位的源端口号也要做字节序的转换。如果主机是大端字节序的，发送和接收都不需要做转换。同理，32位的IP地址也要考虑网络字节序和主机字节序的问题。

为使网络程序具有可移植性，使同样的C代码在大端和小端计算机上编译后都能正常运行，可以调用以下库函数做**网络字节序和主机字节序的转换**。

#include <arpa/inet.h>

uint32\_t htonl(uint32\_t hostlong);

uint16\_t htons(uint16\_t hostshort);

uint32\_t ntohl(uint32\_t netlong);

uint16\_t ntohs(uint16\_t netshort);

h表示host，n表示network，l表示32位长整数，s表示16位短整数。

如果主机是小端字节序，这些函数将参数做相应的大小端转换然后返回，如果主机是大端字节序，这些函数不做转换，将参数原封不动地返回。

### IP地址转换函数

早期：

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

int inet\_aton(const char \*cp, struct in\_addr \*inp);

in\_addr\_t inet\_addr(const char \*cp);

char \*inet\_ntoa(struct in\_addr in);

只能处理IPv4的ip地址

不可重入函数

注意参数是struct in\_addr

现在：

#include <arpa/inet.h>

int inet\_pton(int af, const char \*src, void \*dst);

const char \*inet\_ntop(int af, const void \*src, char \*dst, socklen\_t size);

af:IP地址版本，AF\_INET或AF\_INET6。

src:传入的源参数。

dst:传出转换结果。

size:dst的长度。

支持IPv4和IPv6

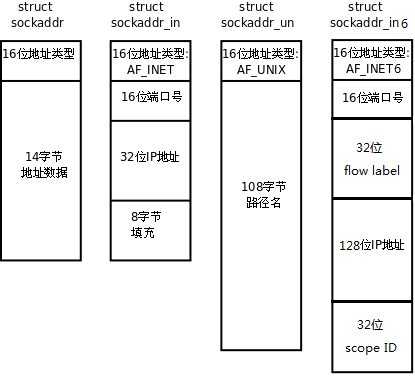
可重入函数

其中inet\_pton和inet\_ntop不仅可以转换IPv4的in\_addr，还可以转换IPv6的in6\_addr。

因此函数接口是void \*addrptr。

### sockaddr数据结构

strcut sockaddr 很多网络编程函数诞生早于IPv4协议，那时候都使用的是sockaddr结构体,为了向前兼容，现在sockaddr退化成了（void \*）的作用，传递一个地址给函数，至于这个函数是sockaddr\_in还是sockaddr\_in6，由地址族确定，然后函数内部再强制类型转化为所需的地址类型。



sockaddr数据结构

#include<arpa/inet.h>

struct sockaddr {

sa\_family\_t sa\_family; /\* address family, AF\_xxx \*/

char sa\_data[14]; /\* 14 bytes of protocol address \*/

};

使用 sudo grep -r "struct sockaddr\_in {" /usr 命令可查看到struct sockaddr\_in结构体的定义。一般其默认的存储位置：/usr/include/linux/in.h 文件中。

struct sockaddr\_in {

\_\_kernel\_sa\_family\_t sin\_family; /\* Address family \*/ 地址结构类型

\_\_be16 sin\_port; /\* Port number \*/ 端口号

struct in\_addr sin\_addr; /\* Internet address \*/ IP地址

/\* Pad to size of `struct sockaddr'. \*/

unsigned char \_\_pad[\_\_SOCK\_SIZE\_\_ - sizeof(short int) -

sizeof(unsigned short int) - sizeof(struct in\_addr)];

};

struct in\_addr { /\* Internet address. \*/

\_\_be32 s\_addr;

};

struct sockaddr\_in6 {

unsigned short int sin6\_family; /\* AF\_INET6 \*/

\_\_be16 sin6\_port; /\* Transport layer port # \*/

\_\_be32 sin6\_flowinfo; /\* IPv6 flow information \*/

struct in6\_addr sin6\_addr; /\* IPv6 address \*/

\_\_u32 sin6\_scope\_id; /\* scope id (new in RFC2553) \*/

};

struct in6\_addr {

union {

\_\_u8 u6\_addr8[16];

\_\_be16 u6\_addr16[8];

\_\_be32 u6\_addr32[4];

} in6\_u;

#define s6\_addr in6\_u.u6\_addr8

#define s6\_addr16 in6\_u.u6\_addr16

#define s6\_addr32 in6\_u.u6\_addr32

};

#define UNIX\_PATH\_MAX 108

struct sockaddr\_un {

\_\_kernel\_sa\_family\_t sun\_family; /\* AF\_UNIX \*/

char sun\_path[UNIX\_PATH\_MAX]; /\* pathname \*/

};

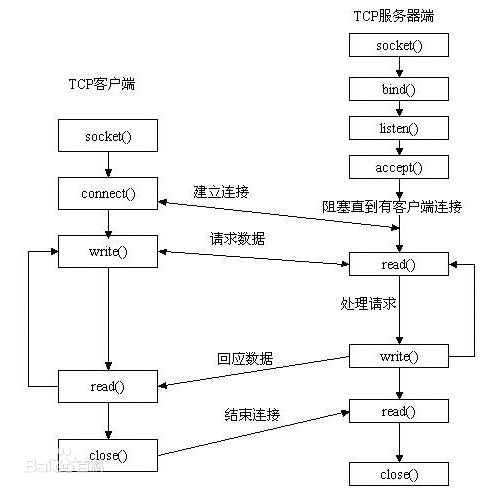
Pv4和IPv6的地址格式定义在netinet/in.h中，IPv4地址用sockaddr\_in结构体表示，包括16位端口号和32位IP地址，IPv6地址用sockaddr\_in6结构体表示，包括16位端口号、128位IP地址和一些控制字段。UNIX Domain Socket的地址格式定义在sys/un.h中，用sock-addr\_un结构体表示。各种socket地址结构体的开头都是相同的，前16位表示整个结构体的长度（并不是所有UNIX的实现都有长度字段，如Linux就没有），后16位表示地址类型。IPv4、IPv6和Unix Domain Socket的地址类型分别定义为常数AF\_INET、AF\_INET6、AF\_UNIX。这样，只要取得某种sockaddr结构体的首地址，不需要知道具体是哪种类型的sockaddr结构体，就可以根据地址类型字段确定结构体中的内容。因此，socket API可以接受各种类型的sockaddr结构体指针做参数，例如bind、accept、connect等函数，这些函数的参数应该设计成void \*类型以便接受各种类型的指针，但是sock API的实现早于ANSI C标准化，那时还没有void \*类型，因此这些函数的参数都用struct sockaddr \*类型表示，在传递参数之前要强制类型转换一下，例如：

struct sockaddr\_in servaddr;

bind(listen\_fd, (struct sockaddr \*)&servaddr, sizeof(servaddr)); /\* initialize servaddr \*/

## 网络套接字函数

### socket模型创建流程图



socket API

### socket函数（申请一个socket）

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

int socket(int domain, int type, int protocol);

domain:

AF\_INET 这是大多数用来产生socket的协议，使用TCP或UDP来传输，用IPv4的地址

AF\_INET6 与上面类似，不过是来用IPv6的地址

AF\_UNIX 本地协议，使用在Unix和Linux系统上，一般都是当客户端和服务器在同一台及其上的时候使用

type:

SOCK\_STREAM 流式协议。这个协议是按照顺序的、可靠的、数据完整的基于字节流的连接。这是一个使用最多的socket类型，这个socket是使用TCP来进行传输。

SOCK\_DGRAM 报式协议。这个协议是无连接的、固定长度的传输调用。该协议是不可靠的，使用UDP来进行它的连接。

SOCK\_SEQPACKET该协议是双线路的、可靠的连接，发送固定长度的数据包进行传输。必须把这个包完整的接受才能进行读取。

SOCK\_RAW socket类型提供单一的网络访问，这个socket类型使用ICMP公共协议。（ping、traceroute使用该协议）

SOCK\_RDM 这个类型是很少使用的，在大部分的操作系统上没有实现，它是提供给数据链路层使用，不保证数据包的顺序

protocol:协议号

传0 表示使用默认协议。

返回值：

成功：返回指向新创建的socket的文件描述符，失败：返回-1，设置errno

socket()打开一个网络通讯端口，如果成功的话，就像open()一样返回一个文件描述符，应用程序可以像读写文件一样用read/write在网络上收发数据，如果socket()调用出错则返回-1。对于IPv4，domain参数指定为AF\_INET。对于TCP协议，type参数指定为SOCK\_STREAM，表示面向流的传输协议。如果是UDP协议，则type参数指定为SOCK\_DGRAM，表示面向数据报的传输协议。protocol参数的介绍从略，指定为0即可。

### bind函数（将socket和addr+port绑定）

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

sockfd：

socket文件描述符

addr:

构造出IP地址加端口号

addrlen:

sizeof(addr)长度

返回值：

成功返回0，失败返回-1, 设置errno

服务器程序所监听的网络地址和端口号通常是固定不变的，客户端程序得知服务器程序的地址和端口号后就可以向服务器发起连接，因此服务器需要调用bind绑定一个固定的网络地址和端口号。

bind()的作用是将参数sockfd和addr绑定在一起，使sockfd这个用于网络通讯的文件描述符监听addr所描述的地址和端口号。前面讲过，struct sockaddr \*是一个通用指针类型，addr参数实际上可以接受多种协议的sockaddr结构体，而它们的长度各不相同，所以需要第三个参数addrlen指定结构体的长度。如：

struct sockaddr\_in servaddr;

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(6666);

首先将整个结构体清零，然后设置地址类型为AF\_INET，**网络地址为INADDR\_ANY，这个宏表示本地的任意IP地址**，因为服务器可能有多个网卡，每个网卡也可能绑定多个IP地址，这样设置可以在所有的IP地址上监听，直到与某个客户端建立了连接时才确定下来到底用哪个IP地址，端口号为6666。

### listen函数（处于半连接状态的最大主机数）

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

int listen(int sockfd, int backlog);

sockfd:

socket文件描述符

backlog:

排队建立3次握手队列和刚刚建立3次握手队列的链接数和

查看系统默认backlog

cat /proc/sys/net/ipv4/tcp\_max\_syn\_backlog

典型的服务器程序可以同时服务于多个客户端，当有客户端发起连接时，服务器调用的accept()返回并接受这个连接，如果有大量的客户端发起连接而服务器来不及处理，尚未accept的客户端就处于连接等待状态，listen()声明sockfd处于监听状态，并且最多允许有backlog个客户端处于连接待状态，如果接收到更多的连接请求就忽略。listen()成功返回0，失败返回-1。

### accept函数（接收连接请求）

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

sockdf:

socket文件描述符

addr:

传出参数，返回链接客户端地址信息，含IP地址和端口号

addrlen:

传入传出参数（值-结果）,传入sizeof(addr)大小，函数返回时返回真正接收到地址结构体的大小

返回值：

成功返回一个新的socket文件描述符，用于和客户端通信，失败返回-1，设置errno

三方握手完成后，服务器调用accept()接受连接，如果服务器调用accept()时还没有客户端的连接请求，就阻塞等待直到有客户端连接上来。addr是一个传出参数，accept()返回时传出客户端的地址和端口号。addrlen参数是一个传入传出参数（value-result argument），传入的是调用者提供的缓冲区addr的长度以避免缓冲区溢出问题，传出的是客户端地址结构体的实际长度（有可能没有占满调用者提供的缓冲区）。如果给addr参数传NULL，表示不关心客户端的地址。

我们的服务器程序结构是这样的：

while (1) {

cliaddr\_len = sizeof(cliaddr);

connfd = accept(listenfd, (struct sockaddr \*)&cliaddr, &cliaddr\_len);

n = read(connfd, buf, MAXLINE);

......

close(connfd);

}

整个是一个while死循环，每次循环处理一个客户端连接。由于cliaddr\_len是传入传出参数，每次调用accept()之前应该重新赋初值。accept()的参数listenfd是先前的监听文件描述符，而accept()的返回值是另外一个文件描述符connfd，之后与客户端之间就通过这个connfd通讯，最后关闭connfd断开连接，而不关闭listenfd，再次回到循环开头listenfd仍然用作accept的参数。accept()成功返回一个文件描述符，出错返回-1。

### connect函数（客户端请求建立连接）

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

sockdf:

socket文件描述符

addr:

传入参数，指定服务器端地址信息，含IP地址和端口号

addrlen:

传入参数,传入sizeof(addr)大小

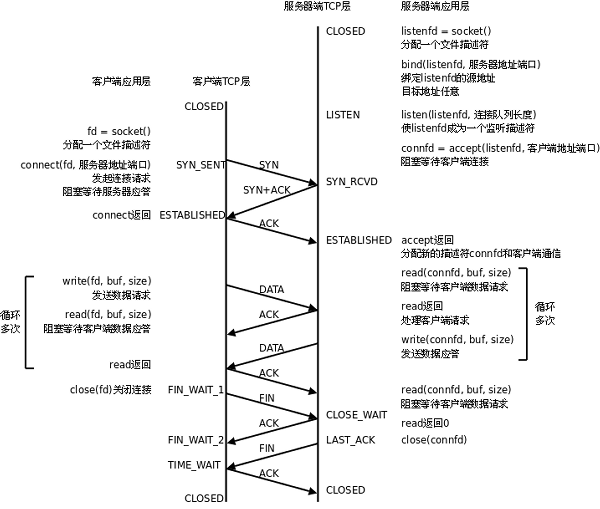
返回值：

成功返回0，失败返回-1，设置errno

客户端需要调用connect()连接服务器，connect和bind的参数形式一致，区别在于bind的参数是自己的地址，而connect的参数是对方的地址。connect()成功返回0，出错返回-1。

## C/S模型-TCP

下图是基于TCP协议的客户端/服务器程序的一般流程：



TCP协议通讯流程

服务器调用socket()、bind()、listen()完成初始化后，调用accept()阻塞等待，处于监听端口的状态，客户端调用socket()初始化后，调用connect()发出SYN段并阻塞等待服务器应答，服务器应答一个SYN-ACK段，客户端收到后从connect()返回，同时应答一个ACK段，服务器收到后从accept()返回。

数据传输的过程：

建立连接后，TCP协议提供全双工的通信服务，但是一般的客户端/服务器程序的流程是由客户端主动发起请求，服务器被动处理请求，一问一答的方式。因此，服务器从accept()返回后立刻调用read()，读socket就像读管道一样，如果没有数据到达就阻塞等待，这时客户端调用write()发送请求给服务器，服务器收到后从read()返回，对客户端的请求进行处理，在此期间客户端调用read()阻塞等待服务器的应答，服务器调用write()将处理结果发回给客户端，再次调用read()阻塞等待下一条请求，客户端收到后从read()返回，发送下一条请求，如此循环下去。

如果客户端没有更多的请求了，就调用close()关闭连接，就像写端关闭的管道一样，服务器的read()返回0，这样服务器就知道客户端关闭了连接，也调用close()关闭连接。注意，任何一方调用close()后，连接的两个传输方向都关闭，不能再发送数据了。如果一方调用shutdown()则连接处于半关闭状态，仍可接收对方发来的数据。

在学习socket API时要注意应用程序和TCP协议层是如何交互的： 应用程序调用某个socket函数时TCP协议层完成什么动作，比如调用connect()会发出SYN段 应用程序如何知道TCP协议层的状态变化，比如从某个阻塞的socket函数返回就表明TCP协议收到了某些段，再比如read()返回0就表明收到了FIN段

### server

下面通过最简单的客户端/服务器程序的实例来学习socket API。

server.c的作用是从客户端读字符，然后将每个字符转换为大写并回送给客户端。

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#define MAXLINE 80

#define SERV\_PORT 6666

int main(void)

{

struct sockaddr\_in servaddr, cliaddr;

socklen\_t cliaddr\_len;

int listenfd, connfd;

char buf[MAXLINE];

char str[INET\_ADDRSTRLEN];

int i, n;

listenfd = socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(SERV\_PORT);

bind(listenfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

listen(listenfd, 20);

printf("Accepting connections ...\n");

while (1) {

cliaddr\_len = sizeof(cliaddr);

connfd = accept(listenfd, (struct sockaddr \*)&cliaddr, &cliaddr\_len);

n = read(connfd, buf, MAXLINE);

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &cliaddr.sin\_addr, str, sizeof(str)),

ntohs(cliaddr.sin\_port));

for (i = 0; i < n; i++)

buf[i] = toupper(buf[i]);

write(connfd, buf, n);

close(connfd);

}

return 0;

}

### client

client.c的作用是从命令行参数中获得一个字符串发给服务器，然后接收服务器返回的字符串并打印。

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#define MAXLINE 80

#define SERV\_PORT 6666

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

{

struct sockaddr\_in servaddr;

char buf[MAXLINE];

int sockfd, n;

char \*str;

if (argc != 2) {

fputs("usage: ./client message\n", stderr);

exit(1);

}

str = argv[1];

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

write(sockfd, str, strlen(str));

n = read(sockfd, buf, MAXLINE);

printf("Response from server:\n");

write(STDOUT\_FILENO, buf, n);

close(sockfd);

return 0;

}

由于客户端不需要固定的端口号，因此不必调用bind()，客户端的端口号由内核自动分配。注意，客户端不是不允许调用bind()，只是没有必要调用bind()固定一个端口号，服务器也不是必须调用bind()，但如果服务器不调用bind()，内核会自动给服务器分配监听端口，每次启动服务器时端口号都不一样，客户端要连接服务器就会遇到麻烦。

客户端和服务器启动后可以使用netstat命令查看链接情况：

netstat -apn|grep 6666

## 出错处理封装函数

上面的例子不仅功能简单，而且简单到几乎没有什么错误处理，我们知道，系统调用不能保证每次都成功，必须进行出错处理，这样一方面可以保证程序逻辑正常，另一方面可以迅速得到故障信息。

为使错误处理的代码不影响主程序的可读性，我们把与socket相关的一些系统函数加上错误处理代码包装成新的函数，做成一个模块wrap.c：

### wrap.c

#include <stdlib.h>

#include <errno.h>

#include <sys/socket.h>

void perr\_exit(const char \*s)

{

perror(s);

exit(1);

}

int Accept(int fd, struct sockaddr \*sa, socklen\_t \*salenptr)

{

int n;

again:

if ( (n = accept(fd, sa, salenptr)) < 0) {

if ((errno == ECONNABORTED) || (errno == EINTR))

goto again;

else

perr\_exit("accept error");

}

return n;

}

int Bind(int fd, const struct sockaddr \*sa, socklen\_t salen)

{

int n;

if ((n = bind(fd, sa, salen)) < 0)

perr\_exit("bind error");

return n;

}

int Connect(int fd, const struct sockaddr \*sa, socklen\_t salen)

{

int n;

if ((n = connect(fd, sa, salen)) < 0)

perr\_exit("connect error");

return n;

}

int Listen(int fd, int backlog)

{

int n;

if ((n = listen(fd, backlog)) < 0)

perr\_exit("listen error");

return n;

}

int Socket(int family, int type, int protocol)

{

int n;

if ( (n = socket(family, type, protocol)) < 0)

perr\_exit("socket error");

return n;

}

ssize\_t Read(int fd, void \*ptr, size\_t nbytes)

{

ssize\_t n;

again:

if ( (n = read(fd, ptr, nbytes)) == -1) {

if (errno == EINTR)

goto again;

else

return -1;

}

return n;

}

ssize\_t Write(int fd, const void \*ptr, size\_t nbytes)

{

ssize\_t n;

again:

if ( (n = write(fd, ptr, nbytes)) == -1) {

if (errno == EINTR)

goto again;

else

return -1;

}

return n;

}

int Close(int fd)

{

int n;

if ((n = close(fd)) == -1)

perr\_exit("close error");

return n;

}

ssize\_t Readn(int fd, void \*vptr, size\_t n)

{

size\_t nleft;

ssize\_t nread;

char \*ptr;

ptr = vptr;

nleft = n;

while (nleft > 0) {

if ( (nread = read(fd, ptr, nleft)) < 0) {

if (errno == EINTR)

nread = 0;

else

return -1;

} else if (nread == 0)

break;

nleft -= nread;

ptr += nread;

}

return n - nleft;

}

ssize\_t Writen(int fd, const void \*vptr, size\_t n)

{

size\_t nleft;

ssize\_t nwritten;

const char \*ptr;

ptr = vptr;

nleft = n;

while (nleft > 0) {

if ( (nwritten = write(fd, ptr, nleft)) <= 0) {

if (nwritten < 0 && errno == EINTR)

nwritten = 0;

else

return -1;

}

nleft -= nwritten;

ptr += nwritten;

}

return n;

}

static ssize\_t my\_read(int fd, char \*ptr)

{

static int read\_cnt;

static char \*read\_ptr;

static char read\_buf[100];

if (read\_cnt <= 0) {

again:

if ((read\_cnt = read(fd, read\_buf, sizeof(read\_buf))) < 0) {

if (errno == EINTR)

goto again;

return -1;

} else if (read\_cnt == 0)

return 0;

read\_ptr = read\_buf;

}

read\_cnt--;

\*ptr = \*read\_ptr++;

return 1;

}

ssize\_t Readline(int fd, void \*vptr, size\_t maxlen)

{

ssize\_t n, rc;

char c, \*ptr;

ptr = vptr;

for (n = 1; n < maxlen; n++) {

if ( (rc = my\_read(fd, &c)) == 1) {

\*ptr++ = c;

if (c == '\n')

break;

} else if (rc == 0) {

\*ptr = 0;

return n - 1;

} else

return -1;

}

\*ptr = 0;

return n;

}

### wrap.h

#ifndef \_\_WRAP\_H\_

#define \_\_WRAP\_H\_

void perr\_exit(const char \*s);

int Accept(int fd, struct sockaddr \*sa, socklen\_t \*salenptr);

int Bind(int fd, const struct sockaddr \*sa, socklen\_t salen);

int Connect(int fd, const struct sockaddr \*sa, socklen\_t salen);

int Listen(int fd, int backlog);

int Socket(int family, int type, int protocol);

ssize\_t Read(int fd, void \*ptr, size\_t nbytes);

ssize\_t Write(int fd, const void \*ptr, size\_t nbytes);

int Close(int fd);

ssize\_t Readn(int fd, void \*vptr, size\_t n);

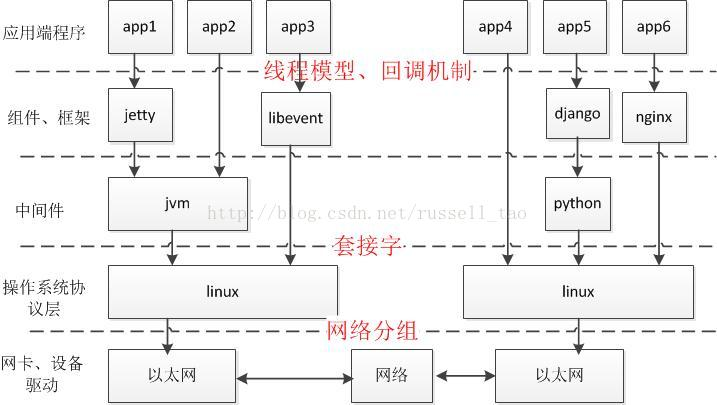
ssize\_t Writen(int fd, const void \*vptr, size\_t n);

ssize\_t my\_read(int fd, char \*ptr);

ssize\_t Readline(int fd, void \*vptr, size\_t maxlen);

#endif

# 高并发服务器



## 多进程并发服务器

使用多进程并发服务器时要考虑以下几点：

1. 父进程最大文件描述个数(父进程中需要close关闭accept返回的新文件描述符)
2. 系统内创建进程个数(与内存大小相关)
3. 进程创建过多是否降低整体服务性能(进程调度)

### server

/\* server.c \*/

#include <stdio.h>

#include <string.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <signal.h>

#include <sys/wait.h>

#include <sys/types.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 800

void do\_sigchild(int num)

{

while (waitpid(0, NULL, WNOHANG) > 0)

;

}

int main(void)

{

struct sockaddr\_in servaddr, cliaddr;

socklen\_t cliaddr\_len;

int listenfd, connfd;

char buf[MAXLINE];

char str[INET\_ADDRSTRLEN];

int i, n;

pid\_t pid;

struct sigaction newact;

newact.sa\_handler = do\_sigchild;

sigemptyset(&newact.sa\_mask);

newact.sa\_flags = 0;

sigaction(SIGCHLD, &newact, NULL);

listenfd = Socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(SERV\_PORT);

Bind(listenfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

Listen(listenfd, 20);

printf("Accepting connections ...\n");

while (1) {

cliaddr\_len = sizeof(cliaddr);

connfd = Accept(listenfd, (struct sockaddr \*)&cliaddr, &cliaddr\_len);

pid = fork();

if (pid == 0) {

Close(listenfd);

while (1) {

n = Read(connfd, buf, MAXLINE);

if (n == 0) {

printf("the other side has been closed.\n");

break;

}

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &cliaddr.sin\_addr, str, sizeof(str)),

ntohs(cliaddr.sin\_port));

for (i = 0; i < n; i++)

buf[i] = toupper(buf[i]);

Write(connfd, buf, n);

}

Close(connfd);

return 0;

} else if (pid > 0) {

Close(connfd);

} else

perr\_exit("fork");

}

Close(listenfd);

return 0;

}

### client

/\* client.c \*/

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

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

{

struct sockaddr\_in servaddr;

char buf[MAXLINE];

int sockfd, n;

sockfd = Socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

Connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

while (fgets(buf, MAXLINE, stdin) != NULL) {

Write(sockfd, buf, strlen(buf));

n = Read(sockfd, buf, MAXLINE);

if (n == 0) {

printf("the other side has been closed.\n");

break;

} else

Write(STDOUT\_FILENO, buf, n);

}

Close(sockfd);

return 0;

}

## 多线程并发服务器

在使用线程模型开发服务器时需考虑以下问题：

1. 调整进程内最大文件描述符上限
2. 线程如有共享数据，考虑线程同步
3. 服务于客户端线程退出时，退出处理。（退出值，分离态）
4. 系统负载，随着链接客户端增加，导致其它线程不能及时得到CPU

### server

/\* server.c \*/

#include <stdio.h>

#include <string.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <pthread.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

struct s\_info {

struct sockaddr\_in cliaddr;

int connfd;

};

void \*do\_work(void \*arg)

{

int n,i;

struct s\_info \*ts = (struct s\_info\*)arg;

char buf[MAXLINE];

char str[INET\_ADDRSTRLEN];

/\* 可以在创建线程前设置线程创建属性,设为分离态,哪种效率高内？ \*/

pthread\_detach(pthread\_self());

while (1) {

n = Read(ts->connfd, buf, MAXLINE);

if (n == 0) {

printf("the other side has been closed.\n");

break;

}

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &(\*ts).cliaddr.sin\_addr, str, sizeof(str)),

ntohs((\*ts).cliaddr.sin\_port));

for (i = 0; i < n; i++)

buf[i] = toupper(buf[i]);

Write(ts->connfd, buf, n);

}

Close(ts->connfd);

}

int main(void)

{

struct sockaddr\_in servaddr, cliaddr;

socklen\_t cliaddr\_len;

int listenfd, connfd;

int i = 0;

pthread\_t tid;

struct s\_info ts[256];

listenfd = Socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(SERV\_PORT);

Bind(listenfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

Listen(listenfd, 20);

printf("Accepting connections ...\n");

while (1) {

cliaddr\_len = sizeof(cliaddr);

connfd = Accept(listenfd, (struct sockaddr \*)&cliaddr, &cliaddr\_len);

ts[i].cliaddr = cliaddr;

ts[i].connfd = connfd;

/\* 达到线程最大数时，pthread\_create出错处理, 增加服务器稳定性 \*/

pthread\_create(&tid, NULL, do\_work, (void\*)&ts[i]);

i++;

}

return 0;

}

### client

/\* client.c \*/

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

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

{

struct sockaddr\_in servaddr;

char buf[MAXLINE];

int sockfd, n;

sockfd = Socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

Connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

while (fgets(buf, MAXLINE, stdin) != NULL) {

Write(sockfd, buf, strlen(buf));

n = Read(sockfd, buf, MAXLINE);

if (n == 0)

printf("the other side has been closed.\n");

else

Write(STDOUT\_FILENO, buf, n);

}

Close(sockfd);

return 0;

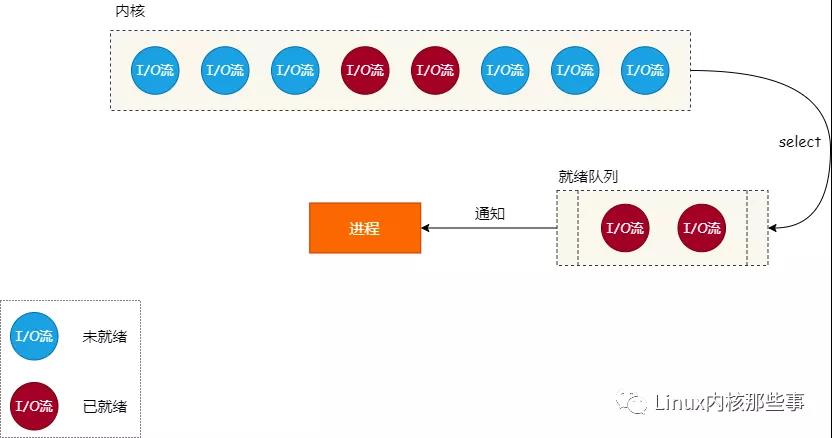
}

## 多路I/O转接服务器

多路IO转接服务器也叫做多任务IO服务器。该类服务器实现的主旨思想是，不再由应用程序自己监视客户端连接，取而代之由内核替应用程序监视文件。

### 多路复用I/O

是指内核负责监听多个 I/O 流，当任何一个 I/O 流处于就绪状态（可读或可写）时都会通知进程，以便可以处理该 I/O 流上的数据。



如 图1 所示，内核负责监听多个 I/O 流，当某些 I/O 流变为就绪状态，内核会把这些 I/O 流添加到就绪队列中，然后通知进程处理就绪队列中的 I/O 流。

与传统的阻塞型 I/O 相比，多路复用 I/O 的优点是可以同时监听多个 I/O 流，并且会把就绪的 I/O 流告知进程。

主要使用的方法有三种

### select

1. select能监听的文件描述符个数受限于FD\_SETSIZE,一般为1024，单纯改变进程打开的文件描述符个数并不能改变select监听文件个数
2. 解决1024以下客户端时使用select是很合适的，但如果链接客户端过多，select采用的是轮询模型，会大大降低服务器响应效率，不应在select上投入更多精力

#include <sys/select.h>

/\* According to earlier standards \*/

#include <sys/time.h>

#include <sys/types.h>

#include <unistd.h>

int select(int nfds, fd\_set \*readfds, fd\_set \*writefds,

fd\_set \*exceptfds, struct timeval \*timeout);

nfds: 监控的文件描述符集里最大文件描述符加1，因为此参数会告诉内核检测前多少个文件描述符的状态

readfds： 监控有读数据到达文件描述符集合，传入传出参数

writefds： 监控写数据到达文件描述符集合，传入传出参数

exceptfds： 监控异常发生达文件描述符集合,如带外数据到达异常，传入传出参数

timeout： 定时阻塞监控时间，3种情况

1.NULL，永远等下去

2.设置timeval，等待固定时间

3.设置timeval里时间均为0，检查描述字后立即返回，轮询

struct timeval {

long tv\_sec; /\* seconds \*/

long tv\_usec; /\* microseconds \*/

};

void FD\_CLR(int fd, fd\_set \*set); //把文件描述符集合里fd清0

int FD\_ISSET(int fd, fd\_set \*set); //测试文件描述符集合里fd是否置1

void FD\_SET(int fd, fd\_set \*set); //把文件描述符集合里fd位置1

void FD\_ZERO(fd\_set \*set); //把文件描述符集合里所有位清0

#### server

/\* server.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

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

{

int i, maxi, maxfd, listenfd, connfd, sockfd;

int nready, client[FD\_SETSIZE]; /\* FD\_SETSIZE 默认为 1024 \*/

ssize\_t n;

fd\_set rset, allset;

char buf[MAXLINE];

char str[INET\_ADDRSTRLEN]; /\* #define INET\_ADDRSTRLEN 16 \*/

socklen\_t cliaddr\_len;

struct sockaddr\_in cliaddr, servaddr;

listenfd = Socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(SERV\_PORT);

Bind(listenfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

Listen(listenfd, 20); /\* 默认最大128 \*/

maxfd = listenfd; /\* 初始化 \*/

maxi = -1; /\* client[]的下标 \*/

for (i = 0; i < FD\_SETSIZE; i++)

client[i] = -1; /\* 用-1初始化client[] \*/

FD\_ZERO(&allset);

FD\_SET(listenfd, &allset); /\* 构造select监控文件描述符集 \*/

for ( ; ; ) {

rset = allset; /\* 每次循环时都从新设置select监控信号集 \*/

nready = select(maxfd+1, &rset, NULL, NULL, NULL);

if (nready < 0)

perr\_exit("select error");

if (FD\_ISSET(listenfd, &rset)) { /\* new client connection \*/

cliaddr\_len = sizeof(cliaddr);

connfd = Accept(listenfd, (struct sockaddr \*)&cliaddr, &cliaddr\_len);

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &cliaddr.sin\_addr, str, sizeof(str)),

ntohs(cliaddr.sin\_port));

for (i = 0; i < FD\_SETSIZE; i++) {

if (client[i] < 0) {

client[i] = connfd; /\* 保存accept返回的文件描述符到client[]里 \*/

break;

}

}

/\* 达到select能监控的文件个数上限 1024 \*/

if (i == FD\_SETSIZE) {

fputs("too many clients\n", stderr);

exit(1);

}

FD\_SET(connfd, &allset); /\* 添加一个新的文件描述符到监控信号集里 \*/

if (connfd > maxfd)

maxfd = connfd; /\* select第一个参数需要 \*/

if (i > maxi)

maxi = i; /\* 更新client[]最大下标值 \*/

if (--nready == 0)

continue; /\* 如果没有更多的就绪文件描述符继续回到上面select阻塞监听,

负责处理未处理完的就绪文件描述符 \*/

}

for (i = 0; i <= maxi; i++) { /\* 检测哪个clients 有数据就绪 \*/

if ( (sockfd = client[i]) < 0)

continue;

if (FD\_ISSET(sockfd, &rset)) {

if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {

Close(sockfd); /\* 当client关闭链接时，服务器端也关闭对应链接 \*/

FD\_CLR(sockfd, &allset); /\* 解除select监控此文件描述符 \*/

client[i] = -1;

} else {

int j;

for (j = 0; j < n; j++)

buf[j] = toupper(buf[j]);

Write(sockfd, buf, n);

}

if (--nready == 0)

break;

}

}

}

close(listenfd);

return 0;

}

#### client

/\* client.c \*/

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

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

{

struct sockaddr\_in servaddr;

char buf[MAXLINE];

int sockfd, n;

sockfd = Socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

Connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

while (fgets(buf, MAXLINE, stdin) != NULL) {

Write(sockfd, buf, strlen(buf));

n = Read(sockfd, buf, MAXLINE);

if (n == 0)

printf("the other side has been closed.\n");

else

Write(STDOUT\_FILENO, buf, n);

}

Close(sockfd);

return 0;

}

#### pselect

pselect原型如下。此模型应用较少，有需要的同学可参考select模型自行编写C/S

#include <sys/select.h>

int pselect(int nfds, fd\_set \*readfds, fd\_set \*writefds,

fd\_set \*exceptfds, const struct timespec \*timeout,

const sigset\_t \*sigmask);

struct timespec {

long tv\_sec; /\* seconds \*/

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

};

用sigmask替代当前进程的阻塞信号集，调用返回后还原原有阻塞信号集

### poll

#include <poll.h>

int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);

struct pollfd {

int fd; /\* 文件描述符 \*/

short events; /\* 监控的事件 \*/

short revents; /\* 监控事件中满足条件返回的事件 \*/

};

**POLLIN**  普通或带外优先数据可读,即POLLRDNORM | POLLRDBAND

POLLRDNORM 数据可读

POLLRDBAND 优先级带数据可读

POLLPRI 高优先级可读数据

**POLLOUT**  普通或带外数据可写

POLLWRNORM 数据可写

POLLWRBAND 优先级带数据可写

**POLLERR**  发生错误

POLLHUP 发生挂起

POLLNVAL 描述字不是一个打开的文件

nfds 监控数组中有多少文件描述符需要被监控

timeout 毫秒级等待

-1：阻塞等，#define INFTIM -1 Linux中没有定义此宏

0：立即返回，不阻塞进程

>0：等待指定毫秒数，如当前系统时间精度不够毫秒，向上取值

如果不再监控某个文件描述符时，可以把pollfd中，fd设置为-1，poll不再监控此pollfd，下次返回时，把revents设置为0。

#### server

/\* server.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <poll.h>

#include <errno.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

#define OPEN\_MAX 1024

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

{

int i, j, maxi, listenfd, connfd, sockfd;

int nready;

ssize\_t n;

char buf[MAXLINE], str[INET\_ADDRSTRLEN];

socklen\_t clilen;

struct pollfd client[OPEN\_MAX];

struct sockaddr\_in cliaddr, servaddr;

listenfd = Socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(SERV\_PORT);

Bind(listenfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

Listen(listenfd, 20);

client[0].fd = listenfd;

client[0].events = POLLRDNORM; /\* listenfd监听普通读事件 \*/

for (i = 1; i < OPEN\_MAX; i++)

client[i].fd = -1; /\* 用-1初始化client[]里剩下元素 \*/

maxi = 0; /\* client[]数组有效元素中最大元素下标 \*/

for ( ; ; ) {

nready = poll(client, maxi+1, -1); /\* 阻塞 \*/

if (client[0].revents & POLLRDNORM) { /\* 有客户端链接请求 \*/

clilen = sizeof(cliaddr);

connfd = Accept(listenfd, (struct sockaddr \*)&cliaddr, &clilen);

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &cliaddr.sin\_addr, str, sizeof(str)),

ntohs(cliaddr.sin\_port));

for (i = 1; i < OPEN\_MAX; i++) {

if (client[i].fd < 0) {

client[i].fd = connfd; /\* 找到client[]中空闲的位置，存放accept返回的connfd \*/

break;

}

}

if (i == OPEN\_MAX)

perr\_exit("too many clients");

client[i].events = POLLRDNORM; /\* 设置刚刚返回的connfd，监控读事件 \*/

if (i > maxi)

maxi = i; /\* 更新client[]中最大元素下标 \*/

if (--nready <= 0)

continue; /\* 没有更多就绪事件时,继续回到poll阻塞 \*/

}

for (i = 1; i <= maxi; i++) { /\* 检测client[] \*/

if ((sockfd = client[i].fd) < 0)

continue;

if (client[i].revents & (POLLRDNORM | POLLERR)) {

if ((n = Read(sockfd, buf, MAXLINE)) < 0) {

if (errno == ECONNRESET) { /\* 当收到 RST标志时 \*/

/\* connection reset by client \*/

printf("client[%d] aborted connection\n", i);

Close(sockfd);

client[i].fd = -1;

} else {

perr\_exit("read error");

}

} else if (n == 0) {

/\* connection closed by client \*/

printf("client[%d] closed connection\n", i);

Close(sockfd);

client[i].fd = -1;

} else {

for (j = 0; j < n; j++)

buf[j] = toupper(buf[j]);

Writen(sockfd, buf, n);

}

if (--nready <= 0)

break; /\* no more readable descriptors \*/

}

}

}

return 0;

}

#### client

/\* client.c \*/

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

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

{

struct sockaddr\_in servaddr;

char buf[MAXLINE];

int sockfd, n;

sockfd = Socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

Connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

while (fgets(buf, MAXLINE, stdin) != NULL) {

Write(sockfd, buf, strlen(buf));

n = Read(sockfd, buf, MAXLINE);

if (n == 0)

printf("the other side has been closed.\n");

else

Write(STDOUT\_FILENO, buf, n);

}

Close(sockfd);

return 0;

}

#### ppoll

GNU定义了ppoll（非POSIX标准），可以支持设置信号屏蔽字，大家可参考poll模型自行实现C/S。

#define \_GNU\_SOURCE /\* See feature\_test\_macros(7) \*/

#include <poll.h>

int ppoll(struct pollfd \*fds, nfds\_t nfds,

const struct timespec \*timeout\_ts, const sigset\_t \*sigmask);

### epoll

epoll是Linux下多路复用IO接口select/poll的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统CPU利用率，因为它会复用文件描述符集合来传递结果而不用迫使开发者每次等待事件之前都必须重新准备要被侦听的文件描述符集合，另一点原因就是获取事件的时候，它无须遍历整个被侦听的描述符集，只要遍历那些被内核IO事件异步唤醒而加入Ready队列的描述符集合就行了。

目前epell是linux大规模并发网络程序中的热门首选模型。

epoll除了提供select/poll那种IO事件的电平触发（Level Triggered）外，还提供了边沿触发（Edge Triggered），这就使得用户空间程序有可能缓存IO状态，减少epoll\_wait/epoll\_pwait的调用，提高应用程序效率。

可以使用cat命令查看一个进程可以打开的socket描述符上限。

cat /proc/sys/fs/file-max

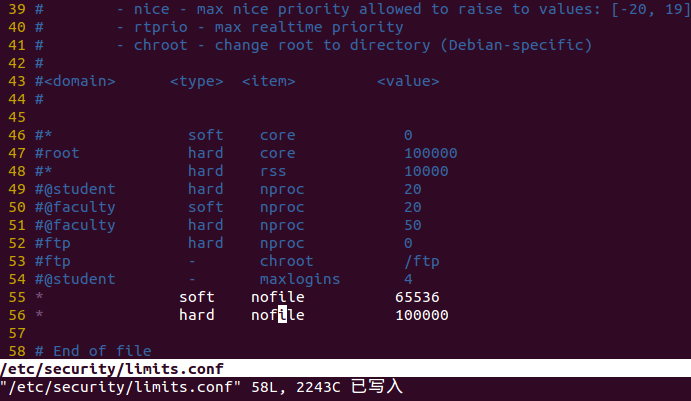
如有需要，可以通过修改配置文件的方式修改该上限值。

sudo vi /etc/security/limits.conf

在文件尾部写入以下配置,soft软限制，hard硬限制。如下图所示。

\* soft nofile 65536

\* hard nofile 100000



#### 基础API

1. 创建一个epoll句柄，参数size用来告诉内核监听的文件描述符的个数，跟内存大小有关。

#include <sys/epoll.h>

int epoll\_create(int size) size：监听数目

1. 控制某个epoll监控的文件描述符上的事件：注册、修改、删除。

#include <sys/epoll.h>

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event)

epfd： 为epoll\_creat的句柄

op： 表示动作，用3个宏来表示：

EPOLL\_CTL\_ADD (注册新的fd到epfd)，

EPOLL\_CTL\_MOD (修改已经注册的fd的监听事件)，

EPOLL\_CTL\_DEL (从epfd删除一个fd)；

event： 告诉内核需要监听的事件

struct epoll\_event {

\_\_uint32\_t events; /\* Epoll events \*/

epoll\_data\_t data; /\* User data variable \*/

};

typedef union epoll\_data {

void \*ptr;

int fd;

uint32\_t u32;

uint64\_t u64;

} epoll\_data\_t;

**EPOLLIN** ： 表示对应的文件描述符可以读（包括对端SOCKET正常关闭）

**EPOLLOUT**： 表示对应的文件描述符可以写

EPOLLPRI： 表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）

**EPOLLERR**： 表示对应的文件描述符发生错误

EPOLLHUP： 表示对应的文件描述符被挂断；

EPOLLET： 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)而言的

EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

1. 等待所监控文件描述符上有事件的产生，类似于select()调用。

#include <sys/epoll.h>

int epoll\_wait(int epfd, struct epoll\_event \*events, int maxevents, int timeout)

events： 用来存内核得到事件的集合，

maxevents： 告之内核这个events有多大，这个maxevents的值不能大于创建epoll\_create()时的size，

timeout： 是超时时间

-1： 阻塞

0： 立即返回，非阻塞

>0： 指定毫秒

返回值： 成功返回有多少文件描述符就绪，时间到时返回0，出错返回-1

#### server

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <sys/epoll.h>

#include <errno.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

#define OPEN\_MAX 1024

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

{

int i, j, maxi, listenfd, connfd, sockfd;

int nready, efd, res;

ssize\_t n;

char buf[MAXLINE], str[INET\_ADDRSTRLEN];

socklen\_t clilen;

int client[OPEN\_MAX];

struct sockaddr\_in cliaddr, servaddr;

struct epoll\_event tep, ep[OPEN\_MAX];

listenfd = Socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(SERV\_PORT);

Bind(listenfd, (struct sockaddr \*) &servaddr, sizeof(servaddr));

Listen(listenfd, 20);

for (i = 0; i < OPEN\_MAX; i++)

client[i] = -1;

maxi = -1;

efd = epoll\_create(OPEN\_MAX);

if (efd == -1)

perr\_exit("epoll\_create");

tep.events = EPOLLIN; tep.data.fd = listenfd;

res = epoll\_ctl(efd, EPOLL\_CTL\_ADD, listenfd, &tep);

if (res == -1)

perr\_exit("epoll\_ctl");

while (1) {

nready = epoll\_wait(efd, ep, OPEN\_MAX, -1); /\* 阻塞监听 \*/

if (nready == -1)

perr\_exit("epoll\_wait");

for (i = 0; i < nready; i++) {

if (!(ep[i].events & EPOLLIN))

continue;

if (ep[i].data.fd == listenfd) {

clilen = sizeof(cliaddr);

connfd = Accept(listenfd, (struct sockaddr \*)&cliaddr, &clilen);

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &cliaddr.sin\_addr, str, sizeof(str)),

ntohs(cliaddr.sin\_port));

for (j = 0; j < OPEN\_MAX; j++) {

if (client[j] < 0) {

client[j] = connfd; /\* save descriptor \*/

break;

}

}

if (j == OPEN\_MAX)

perr\_exit("too many clients");

if (j > maxi)

maxi = j; /\* max index in client[] array \*/

tep.events = EPOLLIN;

tep.data.fd = connfd;

res = epoll\_ctl(efd, EPOLL\_CTL\_ADD, connfd, &tep);

if (res == -1)

perr\_exit("epoll\_ctl");

} else {

sockfd = ep[i].data.fd;

n = Read(sockfd, buf, MAXLINE);

if (n == 0) {

for (j = 0; j <= maxi; j++) {

if (client[j] == sockfd) {

client[j] = -1;

break;

}

}

res = epoll\_ctl(efd, EPOLL\_CTL\_DEL, sockfd, NULL);

if (res == -1)

perr\_exit("epoll\_ctl");

Close(sockfd);

printf("client[%d] closed connection\n", j);

} else {

for (j = 0; j < n; j++)

buf[j] = toupper(buf[j]);

Writen(sockfd, buf, n);

}

}

}

}

close(listenfd);

close(efd);

return 0;

}

#### client

/\* client.c \*/

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#include "wrap.h"

#define MAXLINE 80

#define SERV\_PORT 6666

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

{

struct sockaddr\_in servaddr;

char buf[MAXLINE];

int sockfd, n;

sockfd = Socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

Connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

while (fgets(buf, MAXLINE, stdin) != NULL) {

Write(sockfd, buf, strlen(buf));

n = Read(sockfd, buf, MAXLINE);

if (n == 0)

printf("the other side has been closed.\n");

else

Write(STDOUT\_FILENO, buf, n);

}

Close(sockfd);

return 0;

}

## epoll进阶

### 事件模型

EPOLL事件有两种模型：

Edge Triggered (ET) 边缘触发只有数据到来才触发，不管缓存区中是否还有数据。

Level Triggered (LT) 水平触发只要有数据都会触发。

思考如下步骤：

1. 假定我们已经把一个用来从管道中读取数据的文件描述符(RFD)添加到epoll描述符。
2. 管道的另一端写入了2KB的数据
3. 调用epoll\_wait，并且它会返回RFD，说明它已经准备好读取操作
4. 读取1KB的数据
5. 调用epoll\_wait……

在这个过程中，有两种工作模式：

#### ET模式

ET模式即Edge Triggered工作模式。

如果我们在第1步将RFD添加到epoll描述符的时候使用了EPOLLET标志，那么在第5步调用epoll\_wait之后将有可能会挂起，因为剩余的数据还存在于文件的输入缓冲区内，而且数据发出端还在等待一个针对已经发出数据的反馈信息。只有在监视的文件句柄上发生了某个事件的时候 ET 工作模式才会汇报事件。因此在第5步的时候，调用者可能会放弃等待仍在存在于文件输入缓冲区内的剩余数据。epoll工作在ET模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。最好以下面的方式调用ET模式的epoll接口，在后面会介绍避免可能的缺陷。

1. 基于非阻塞文件句柄
2. 只有当read或者write返回EAGAIN(非阻塞读，暂时无数据)时才需要挂起、等待。但这并不是说每次read时都需要循环读，直到读到产生一个EAGAIN才认为此次事件处理完成，当read返回的读到的数据长度小于请求的数据长度时，就可以确定此时缓冲中已没有数据了，也就可以认为此事读事件已处理完成。

#### LT模式

LT模式即Level Triggered工作模式。

与ET模式不同的是，以LT方式调用epoll接口的时候，它就相当于一个速度比较快的poll，无论后面的数据是否被使用。

LT(level triggered)：LT是缺省的工作方式，并且同时支持block和no-block socket。在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。传统的select/poll都是这种模型的代表。

ET(edge-triggered)：ET是高速工作方式，只支持no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过epoll告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知。请注意，如果一直不对这个fd作IO操作(从而导致它再次变成未就绪)，内核不会发送更多的通知(only once).

### 实例一：

基于管道epoll ET触发模式

#include <stdio.h>

#include <stdlib.h>

#include <sys/epoll.h>

#include <errno.h>

#include <unistd.h>

#define MAXLINE 10

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

{

int efd, i;

int pfd[2];

pid\_t pid;

char buf[MAXLINE], ch = 'a';

pipe(pfd);

pid = fork();

if (pid == 0) {

close(pfd[0]);

while (1) {

for (i = 0; i < MAXLINE/2; i++)

buf[i] = ch;

buf[i-1] = '\n';

ch++;

for (; i < MAXLINE; i++)

buf[i] = ch;

buf[i-1] = '\n';

ch++;

write(pfd[1], buf, sizeof(buf));

sleep(2);

}

close(pfd[1]);

} else if (pid > 0) {

struct epoll\_event event;

struct epoll\_event resevent[10];

int res, len;

close(pfd[1]);

efd = epoll\_create(10);

/\* event.events = EPOLLIN; \*/

event.events = EPOLLIN | EPOLLET; /\* ET 边沿触发 ，默认是水平触发 \*/

event.data.fd = pfd[0];

epoll\_ctl(efd, EPOLL\_CTL\_ADD, pfd[0], &event);

while (1) {

res = epoll\_wait(efd, resevent, 10, -1);

printf("res %d\n", res);

if (resevent[0].data.fd == pfd[0]) {

len = read(pfd[0], buf, MAXLINE/2);

write(STDOUT\_FILENO, buf, len);

}

}

close(pfd[0]);

close(efd);

} else {

perror("fork");

exit(-1);

}

return 0;

}

### 实例二：

基于网络C/S模型的epoll ET触发模式

#### server

/\* server.c \*/

#include <stdio.h>

#include <string.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <signal.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <sys/epoll.h>

#include <unistd.h>

#define MAXLINE 10

#define SERV\_PORT 8080

int main(void)

{

struct sockaddr\_in servaddr, cliaddr;

socklen\_t cliaddr\_len;

int listenfd, connfd;

char buf[MAXLINE];

char str[INET\_ADDRSTRLEN];

int i, efd;

listenfd = socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(SERV\_PORT);

bind(listenfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

listen(listenfd, 20);

struct epoll\_event event;

struct epoll\_event resevent[10];

int res, len;

efd = epoll\_create(10);

event.events = EPOLLIN | EPOLLET; /\* ET 边沿触发 ，默认是水平触发 \*/

printf("Accepting connections ...\n");

cliaddr\_len = sizeof(cliaddr);

connfd = accept(listenfd, (struct sockaddr \*)&cliaddr, &cliaddr\_len);

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &cliaddr.sin\_addr, str, sizeof(str)),

ntohs(cliaddr.sin\_port));

event.data.fd = connfd;

epoll\_ctl(efd, EPOLL\_CTL\_ADD, connfd, &event);

while (1) {

res = epoll\_wait(efd, resevent, 10, -1);

printf("res %d\n", res);

if (resevent[0].data.fd == connfd) {

len = read(connfd, buf, MAXLINE/2);

write(STDOUT\_FILENO, buf, len);

}

}

return 0;

}

#### client

/\* client.c \*/

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#define MAXLINE 10

#define SERV\_PORT 8080

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

{

struct sockaddr\_in servaddr;

char buf[MAXLINE];

int sockfd, i;

char ch = 'a';

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

while (1) {

for (i = 0; i < MAXLINE/2; i++)

buf[i] = ch;

buf[i-1] = '\n';

ch++;

for (; i < MAXLINE; i++)

buf[i] = ch;

buf[i-1] = '\n';

ch++;

write(sockfd, buf, sizeof(buf));

sleep(10);

}

Close(sockfd);

return 0;

}

### 实例三：

基于网络C/S非阻塞模型的epoll ET触发模式

#### server

/\* server.c \*/

#include <stdio.h>

#include <string.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <sys/epoll.h>

#include <unistd.h>

#include <fcntl.h>

#define MAXLINE 10

#define SERV\_PORT 8080

int main(void)

{

struct sockaddr\_in servaddr, cliaddr;

socklen\_t cliaddr\_len;

int listenfd, connfd;

char buf[MAXLINE];

char str[INET\_ADDRSTRLEN];

int i, efd, flag;

listenfd = socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(SERV\_PORT);

bind(listenfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

listen(listenfd, 20);

struct epoll\_event event;

struct epoll\_event resevent[10];

int res, len;

efd = epoll\_create(10);

/\* event.events = EPOLLIN; \*/

event.events = EPOLLIN | EPOLLET; /\* ET 边沿触发 ，默认是水平触发 \*/

printf("Accepting connections ...\n");

cliaddr\_len = sizeof(cliaddr);

connfd = accept(listenfd, (struct sockaddr \*)&cliaddr, &cliaddr\_len);

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &cliaddr.sin\_addr, str, sizeof(str)),

ntohs(cliaddr.sin\_port));

flag = fcntl(connfd, F\_GETFL);

flag |= O\_NONBLOCK;

fcntl(connfd, F\_SETFL, flag);

event.data.fd = connfd;

epoll\_ctl(efd, EPOLL\_CTL\_ADD, connfd, &event);

while (1) {

printf("epoll\_wait begin\n");

res = epoll\_wait(efd, resevent, 10, -1);

printf("epoll\_wait end res %d\n", res);

if (resevent[0].data.fd == connfd) {

while ((len = read(connfd, buf, MAXLINE/2)) > 0)

write(STDOUT\_FILENO, buf, len);

}

}

return 0;

}

#### client

/\* client.c \*/

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#define MAXLINE 10

#define SERV\_PORT 8080

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

{

struct sockaddr\_in servaddr;

char buf[MAXLINE];

int sockfd, i;

char ch = 'a';

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

connect(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

while (1) {

for (i = 0; i < MAXLINE/2; i++)

buf[i] = ch;

buf[i-1] = '\n';

ch++;

for (; i < MAXLINE; i++)

buf[i] = ch;

buf[i-1] = '\n';

ch++;

write(sockfd, buf, sizeof(buf));

sleep(10);

}

Close(sockfd);

return 0;

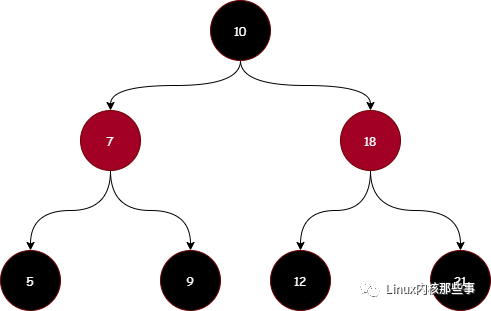
}

## epoll实现原理

在 Linux 系统中，有多种多路复用 I/O 的实现，比如 select 和 poll 等。而 epoll 也是多路复用 I/O 一种实现，与 select 和 poll 相比，epoll 在性能上有较大的提升。

### **红黑树**

epoll 内部使用红黑树来保存所有监听的 socket，红黑树是一种平衡二叉树，添加和查找元素的时间复杂度为 O(log n)，其结构如图所示：

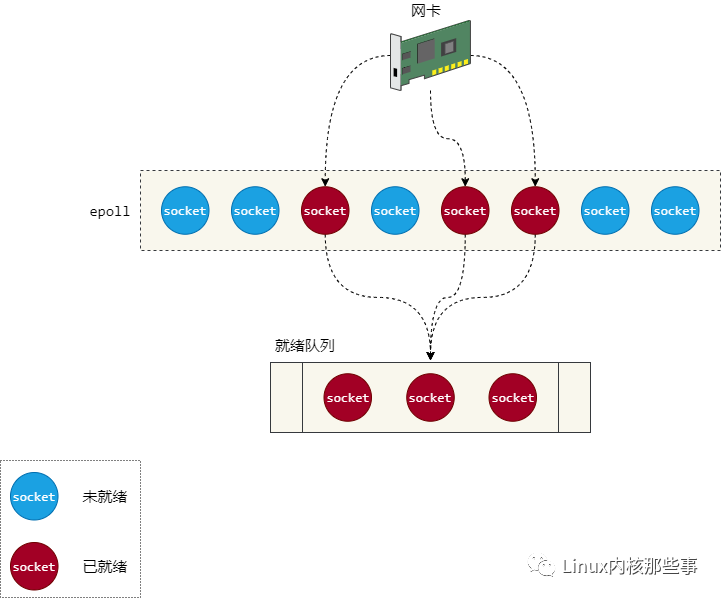


epoll 通过 socket 句柄来作为 key，把 socket 保存在红黑树中。如 图2 所示，每个节点中的数字代表着 socket 句柄。

把监听的 socket 保存在红黑树中的目的是，为了在修改监听 socket 的读写事件时，能够通过 socket 句柄快速找到对应的 socket 对象。

### **就绪队列**

另外，epoll 还维护着一个就绪队列，当 epoll 监听的 socket 状态发生改变（变为可读或可写）时，就会把就绪的 socket 添加到就绪队列中。如图所示：



当 socket 从网络中获取到数据后，会发生通知给 epoll，epoll 会将当前 socket 添加到就绪队列中，并且唤醒等待中的进程（也就是调用 epoll\_wait 的进程）。

当 socket 状态发生变化时，会调用 ep\_poll\_callback 函数来通知 epoll，我们来看看这个函数的处理过程：

static int ep\_poll\_callback(wait\_queue\_t \*wait, unsigned mode,int sync, void \*key)

{

...

struct epitem \*epi = ep\_item\_from\_wait(wait);

struct eventpoll \*ep = epi->ep;

...

*// 1) 把 socket 添加到就绪队列中*

list\_add\_tail(&epi->rdllink, &ep->rdllist);

is\_linked:

*// 2) 唤醒调用 epoll\_wait() 而被阻塞的进程*

if (waitqueue\_active(&ep->wq))

wake\_up\_locked(&ep->wq);

...

return 1;

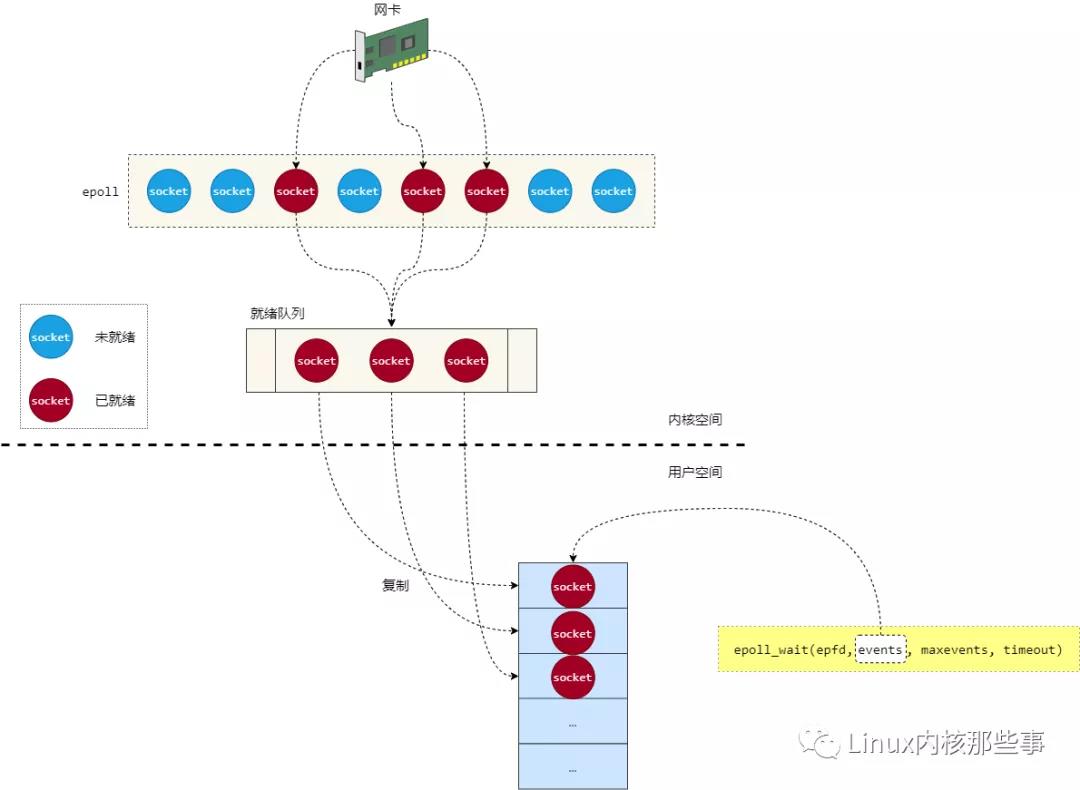
}

ep\_poll\_callback 函数的意图很清晰，主要完成两个工作：

把就绪的 socket 添加到就绪队列中。

唤醒调用 epoll\_wait 函数而被阻塞的进程。

当进程被唤醒后，就会从就绪队列中，把就绪的 socket 复制到用户提供的数组中。如 图4 所示：



如图所示，在调用 epoll\_wait 时需要提供一个 events 数组来存储就绪的 socket。当 epoll\_wait 返回后，用户就可以从events 数组中获取到就绪的 socket，并可对其进行读写操作。

## 线程池并发服务器

1. 预先创建阻塞于accept多线程，使用互斥锁上锁保护accept
2. 预先创建多线程，由主线程调用accept

## UDP服务器

传输层主要应用的协议模型有两种，一种是TCP协议，另外一种则是UDP协议。TCP协议在网络通信中占主导地位，绝大多数的网络通信借助TCP协议完成数据传输。但UDP也是网络通信中不可或缺的重要通信手段。

相较于TCP而言，UDP通信的形式更像是发短信。不需要在数据传输之前建立、维护连接。只专心获取数据就好。省去了三次握手的过程，通信速度可以大大提高，但与之伴随的通信的稳定性和正确率便得不到保证。因此，我们称UDP为“无连接的不可靠报文传递”。

那么与我们熟知的TCP相比，UDP有哪些优点和不足呢？由于无需创建连接，所以UDP开销较小，数据传输速度快，实时性较强。多用于对实时性要求较高的通信场合，如视频会议、电话会议等。但随之也伴随着数据传输不可靠，传输数据的正确率、传输顺序和流量都得不到控制和保证。所以，通常情况下，使用UDP协议进行数据传输，为保证数据的正确性，我们需要在应用层添加辅助校验协议来弥补UDP的不足，以达到数据可靠传输的目的。

与TCP类似的，UDP也有可能出现缓冲区被填满后，再接收数据时丢包的现象。由于它没有TCP滑动窗口的机制，通常采用如下两种方法解决：

1. 服务器应用层设计流量控制，控制发送数据速度。
2. 借助setsockopt函数改变接收缓冲区大小。如：

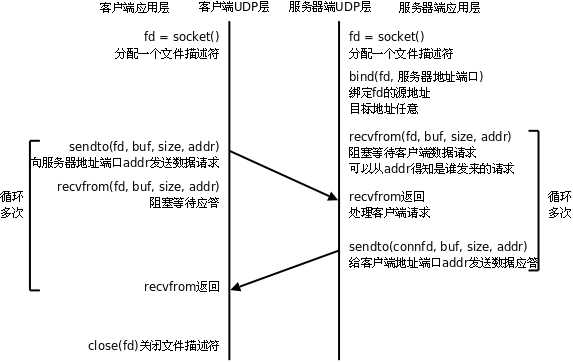
#include <sys/socket.h>

int setsockopt(int sockfd, int level, int optname, const void \*optval, socklen\_t optlen);

int n = 220x1024

setsockopt(sockfd, SOL\_SOCKET, SO\_RCVBUF, &n, sizeof(n));

## C/S模型-UDP



UDP处理模型

由于UDP不需要维护连接，程序逻辑简单了很多，但是UDP协议是不可靠的，保证通讯可靠性的机制需要在应用层实现。

编译运行server，在两个终端里各开一个client与server交互，看看server是否具有并发服务的能力。用Ctrl+C关闭server，然后再运行server，看此时client还能否和server联系上。和前面TCP程序的运行结果相比较，体会无连接的含义。

### server

#include <string.h>

#include <netinet/in.h>

#include <stdio.h>

#include <unistd.h>

#include <strings.h>

#include <arpa/inet.h>

#include <ctype.h>

#define MAXLINE 80

#define SERV\_PORT 6666

int main(void)

{

struct sockaddr\_in servaddr, cliaddr;

socklen\_t cliaddr\_len;

int sockfd;

char buf[MAXLINE];

char str[INET\_ADDRSTRLEN];

int i, n;

sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(SERV\_PORT);

bind(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

printf("Accepting connections ...\n");

while (1) {

cliaddr\_len = sizeof(cliaddr);

n = recvfrom(sockfd, buf, MAXLINE,0, (struct sockaddr \*)&cliaddr, &cliaddr\_len);

if (n == -1)

perror("recvfrom error");

printf("received from %s at PORT %d\n",

inet\_ntop(AF\_INET, &cliaddr.sin\_addr, str, sizeof(str)),

ntohs(cliaddr.sin\_port));

for (i = 0; i < n; i++)

buf[i] = toupper(buf[i]);

n = sendto(sockfd, buf, n, 0, (struct sockaddr \*)&cliaddr, sizeof(cliaddr));

if (n == -1)

perror("sendto error");

}

close(sockfd);

return 0;

}

### client

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <strings.h>

#include <ctype.h>

#define MAXLINE 80

#define SERV\_PORT 6666

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

{

struct sockaddr\_in servaddr;

int sockfd, n;

char buf[MAXLINE];

sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "127.0.0.1", &servaddr.sin\_addr);

servaddr.sin\_port = htons(SERV\_PORT);

while (fgets(buf, MAXLINE, stdin) != NULL) {

n = sendto(sockfd, buf, strlen(buf), 0, (struct sockaddr \*)&servaddr, sizeof(servaddr));

if (n == -1)

perror("sendto error");

n = recvfrom(sockfd, buf, MAXLINE, 0, NULL, 0);

if (n == -1)

perror("recvfrom error");

write(STDOUT\_FILENO, buf, n);

}

close(sockfd);

return 0;

}

## 多播(组播)

组播组可以是永久的也可以是临时的。组播组地址中，有一部分由官方分配的，称为永久组播组。永久组播组保持不变的是它的ip地址，组中的成员构成可以发生变化。永久组播组中成员的数量都可以是任意的，甚至可以为零。那些没有保留下来供永久组播组使用的ip组播地址，可以被临时组播组利用。

224.0.0.0～224.0.0.255 为预留的组播地址（永久组地址），地址224.0.0.0保留不做分配，其它地址供路由协议使用；

224.0.1.0～224.0.1.255 是公用组播地址，可以用于Internet；欲使用需申请。

224.0.2.0～238.255.255.255 为用户可用的组播地址（临时组地址），全网范围内有效；

239.0.0.0～239.255.255.255 为本地管理组播地址，仅在特定的本地范围内有效。

可使用ip ad命令查看网卡编号，如：

itcast$ ip ad

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN group default

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

inet 127.0.0.1/8 scope host lo

valid\_lft forever preferred\_lft forever

inet6 ::1/128 scope host

valid\_lft forever preferred\_lft forever

2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo\_fast state DOWN group default qlen 1000

link/ether 00:0c:29:0a:c4:f4 brd ff:ff:ff:ff:ff:ff

inet6 fe80::20c:29ff:fe0a:c4f4/64 scope link

valid\_lft forever preferred\_lft forever

if\_nametoindex 命令可以根据网卡名，获取网卡序号。

### server

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <net/if.h>

#define SERVER\_PORT 6666

#define CLIENT\_PORT 9000

#define MAXLINE 1500

#define GROUP "239.0.0.2"

int main(void)

{

int sockfd, i ;

struct sockaddr\_in serveraddr, clientaddr;

char buf[MAXLINE] = "itcast\n";

char ipstr[INET\_ADDRSTRLEN]; /\* 16 Bytes \*/

socklen\_t clientlen;

ssize\_t len;

struct ip\_mreqn group;

/\* 构造用于UDP通信的套接字 \*/

sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);

bzero(&serveraddr, sizeof(serveraddr));

serveraddr.sin\_family = AF\_INET; /\* IPv4 \*/

serveraddr.sin\_addr.s\_addr = htonl(INADDR\_ANY); /\* 本地任意IP INADDR\_ANY = 0 \*/

serveraddr.sin\_port = htons(SERVER\_PORT);

bind(sockfd, (struct sockaddr \*)&serveraddr, sizeof(serveraddr));

/\*设置组地址\*/

inet\_pton(AF\_INET, GROUP, &group.imr\_multiaddr);

/\*本地任意IP\*/

inet\_pton(AF\_INET, "0.0.0.0", &group.imr\_address);

/\* eth0 --> 编号 命令：ip ad \*/

group.imr\_ifindex = if\_nametoindex("eth0");

setsockopt(sockfd, IPPROTO\_IP, IP\_MULTICAST\_IF, &group, sizeof(group));

/\*构造 client 地址 IP+端口 \*/

bzero(&clientaddr, sizeof(clientaddr));

clientaddr.sin\_family = AF\_INET; /\* IPv4 \*/

inet\_pton(AF\_INET, GROUP, &clientaddr.sin\_addr.s\_addr);

clientaddr.sin\_port = htons(CLIENT\_PORT);

while (1) {

//fgets(buf, sizeof(buf), stdin);

sendto(sockfd, buf, strlen(buf), 0, (struct sockaddr \*)&clientaddr, sizeof(clientaddr));

sleep(1);

}

close(sockfd);

return 0;

}

### client

#include <netinet/in.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <string.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <unistd.h>

#include <fcntl.h>

#include <net/if.h>

#define SERVER\_PORT 6666

#define MAXLINE 4096

#define CLIENT\_PORT 9000

#define GROUP "239.0.0.2"

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

{

struct sockaddr\_in serveraddr, localaddr;

int confd;

ssize\_t len;

char buf[MAXLINE];

/\* 定义组播结构体 \*/

struct ip\_mreqn group;

confd = socket(AF\_INET, SOCK\_DGRAM, 0);

//初始化本地端地址

bzero(&localaddr, sizeof(localaddr));

localaddr.sin\_family = AF\_INET;

inet\_pton(AF\_INET, "0.0.0.0" , &localaddr.sin\_addr.s\_addr);

localaddr.sin\_port = htons(CLIENT\_PORT);

bind(confd, (struct sockaddr \*)&localaddr, sizeof(localaddr));

/\*设置组地址\*/

inet\_pton(AF\_INET, GROUP, &group.imr\_multiaddr);

/\*本地任意IP\*/

inet\_pton(AF\_INET, "0.0.0.0", &group.imr\_address);

/\* eth0 --> 编号 命令：ip ad \*/

group.imr\_ifindex = if\_nametoindex("eth0");

/\*设置client 加入多播组 \*/

setsockopt(confd, IPPROTO\_IP, IP\_ADD\_MEMBERSHIP, &group, sizeof(group));

while (1) {

len = recvfrom(confd, buf, sizeof(buf), 0, NULL, 0);

write(STDOUT\_FILENO, buf, len);

}

close(confd);

return 0;

}

## socket IPC（本地套接字domain）

socket API原本是为网络通讯设计的，但后来在socket的框架上发展出一种IPC机制，就是UNIX Domain Socket。虽然网络socket也可用于同一台主机的进程间通讯（通过loopback地址127.0.0.1），但是UNIX Domain Socket用于IPC更有效率：不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程。这是因为，IPC机制本质上是可靠的通讯，而网络协议是为不可靠的通讯设计的。UNIX Domain Socket也提供面向流和面向数据包两种API接口，类似于TCP和UDP，但是面向消息的UNIX Domain Socket也是可靠的，消息既不会丢失也不会顺序错乱。

UNIX Domain Socket是全双工的，API接口语义丰富，相比其它IPC机制有明显的优越性，目前已成为使用最广泛的IPC机制，比如X Window服务器和GUI程序之间就是通过UNIXDomain Socket通讯的。

使用UNIX Domain Socket的过程和网络socket十分相似，也要先调用socket()创建一个socket文件描述符，address family指定为AF\_UNIX，type可以选择SOCK\_DGRAM或SOCK\_STREAM，protocol参数仍然指定为0即可。

UNIX Domain Socket与网络socket编程最明显的不同在于地址格式不同，用结构体sockaddr\_un表示，网络编程的socket地址是IP地址加端口号，而UNIX Domain Socket的地址是一个socket类型的文件在文件系统中的路径，这个socket文件由bind()调用创建，如果调用bind()时该文件已存在，则bind()错误返回。

对比网络套接字地址结构和本地套接字地址结构：

struct sockaddr\_in {

\_\_kernel\_sa\_family\_t **sin\_family**; /\* Address family \*/ 地址结构类型

\_\_be16 **sin\_port**; /\* Port number \*/ 端口号

struct in\_addr **sin\_addr**; /\* Internet address \*/ IP地址

};

struct sockaddr\_un {

\_\_kernel\_sa\_family\_t **sun\_family**; /\* AF\_UNIX \*/ 地址结构类型

char **sun\_path**[UNIX\_PATH\_MAX]; /\* pathname \*/ socket文件名(含路径)

};

以下程序将UNIX Domain socket绑定到一个地址。

size = offsetof(struct sockaddr\_un, sun\_path) + strlen(un.sun\_path);

#define offsetof(type, member) ((int)&((type \*)0)->MEMBER)

### server

#include <stdlib.h>

#include <stdio.h>

#include <stddef.h>

#include <sys/socket.h>

#include <sys/un.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#include <errno.h>

#define QLEN 10

/\*

\* Create a server endpoint of a connection.

\* Returns fd if all OK, <0 on error.

\*/

int serv\_listen(const char \*name)

{

int fd, len, err, rval;

struct sockaddr\_un un;

/\* create a UNIX domain stream socket \*/

if ((fd = socket(AF\_UNIX, SOCK\_STREAM, 0)) < 0)

return(-1);

/\* in case it already exists \*/

unlink(name);

/\* fill in socket address structure \*/

memset(&un, 0, sizeof(un));

un.sun\_family = AF\_UNIX;

strcpy(un.sun\_path, name);

len = offsetof(struct sockaddr\_un, sun\_path) + strlen(name);

/\* bind the name to the descriptor \*/

if (bind(fd, (struct sockaddr \*)&un, len) < 0) {

rval = -2;

goto errout;

}

if (listen(fd, QLEN) < 0) { /\* tell kernel we're a server \*/

rval = -3;

goto errout;

}

return(fd);

errout:

err = errno;

close(fd);

errno = err;

return(rval);

}

int serv\_accept(int listenfd, uid\_t \*uidptr)

{

int clifd, len, err, rval;

time\_t staletime;

struct sockaddr\_un un;

struct stat statbuf;

len = sizeof(un);

if ((clifd = accept(listenfd, (struct sockaddr \*)&un, &len)) < 0)

return(-1); /\* often errno=EINTR, if signal caught \*/

/\* obtain the client's uid from its calling address \*/

len -= offsetof(struct sockaddr\_un, sun\_path); /\* len of pathname \*/

un.sun\_path[len] = 0; /\* null terminate \*/

if (stat(un.sun\_path, &statbuf) < 0) {

rval = -2;

goto errout;

}

if (S\_ISSOCK(statbuf.st\_mode) == 0) {

rval = -3; /\* not a socket \*/

goto errout;

}

if (uidptr != NULL)

\*uidptr = statbuf.st\_uid; /\* return uid of caller \*/

/\* we're done with pathname now \*/

unlink(un.sun\_path);

return(clifd);

errout:

err = errno;

close(clifd);

errno = err;

return(rval);

}

int main(void)

{

int lfd, cfd, n, i;

uid\_t cuid;

char buf[1024];

lfd = serv\_listen("foo.socket");

if (lfd < 0) {

switch (lfd) {

case -3:perror("listen"); break;

case -2:perror("bind"); break;

case -1:perror("socket"); break;

}

exit(-1);

}

cfd = serv\_accept(lfd, &cuid);

if (cfd < 0) {

switch (cfd) {

case -3:perror("not a socket"); break;

case -2:perror("a bad filename"); break;

case -1:perror("accept"); break;

}

exit(-1);

}

while (1) {

r\_again:

n = read(cfd, buf, 1024);

if (n == -1) {

if (errno == EINTR)

goto r\_again;

}

else if (n == 0) {

printf("the other side has been closed.\n");

break;

}

for (i = 0; i < n; i++)

buf[i] = toupper(buf[i]);

write(cfd, buf, n);

}

close(cfd);

close(lfd);

return 0;

}

### client

#include <stdio.h>

#include <stdlib.h>

#include <stddef.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/socket.h>

#include <sys/un.h>

#include <errno.h>

#define CLI\_PATH "/var/tmp/" /\* +5 for pid = 14 chars \*/

/\*

\* Create a client endpoint and connect to a server.

\* Returns fd if all OK, <0 on error.

\*/

int cli\_conn(const char \*name)

{

int fd, len, err, rval;

struct sockaddr\_un un;

/\* create a UNIX domain stream socket \*/

if ((fd = socket(AF\_UNIX, SOCK\_STREAM, 0)) < 0)

return(-1);

/\* fill socket address structure with our address \*/

memset(&un, 0, sizeof(un));

un.sun\_family = AF\_UNIX;

sprintf(un.sun\_path, "%s%05d", CLI\_PATH, getpid());

len = offsetof(struct sockaddr\_un, sun\_path) + strlen(un.sun\_path);

/\* in case it already exists \*/

unlink(un.sun\_path);

if (bind(fd, (struct sockaddr \*)&un, len) < 0) {

rval = -2;

goto errout;

}

/\* fill socket address structure with server's address \*/

memset(&un, 0, sizeof(un));

un.sun\_family = AF\_UNIX;

strcpy(un.sun\_path, name);

len = offsetof(struct sockaddr\_un, sun\_path) + strlen(name);

if (connect(fd, (struct sockaddr \*)&un, len) < 0) {

rval = -4;

goto errout;

}

return(fd);

errout:

err = errno;

close(fd);

errno = err;

return(rval);

}

int main(void)

{

int fd, n;

char buf[1024];

fd = cli\_conn("foo.socket");

if (fd < 0) {

switch (fd) {

case -4:perror("connect"); break;

case -3:perror("listen"); break;

case -2:perror("bind"); break;

case -1:perror("socket"); break;

}

exit(-1);

}

while (fgets(buf, sizeof(buf), stdin) != NULL) {

write(fd, buf, strlen(buf));

n = read(fd, buf, sizeof(buf));

write(STDOUT\_FILENO, buf, n);

}

close(fd);

return 0;

}

## 其它常用函数

### 名字与地址转换

gethostbyname根据给定的主机名，获取主机信息。

过时，仅用于IPv4，且线程不安全。

#include <stdio.h>

#include <netdb.h>

#include <arpa/inet.h>

extern int h\_errno;

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

{

struct hostent \*host;

char str[128];

host = gethostbyname(argv[1]);

printf("%s\n", host->h\_name);

while (\*(host->h\_aliases) != NULL)

printf("%s\n", \*host->h\_aliases++);

switch (host->h\_addrtype) {

case AF\_INET:

while (\*(host->h\_addr\_list) != NULL)

printf("%s\n", inet\_ntop(AF\_INET, (\*host->h\_addr\_list++), str, sizeof(str)));

break;

default:

printf("unknown address type\n");

break;

}

return 0;

}

gethostbyaddr函数。

此函数只能获取域名解析服务器的url和/etc/hosts里登记的IP对应的域名。

#include <stdio.h>

#include <netdb.h>

#include <arpa/inet.h>

extern int h\_errno;

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

{

struct hostent \*host;

char str[128];

struct in\_addr addr;

inet\_pton(AF\_INET, argv[1], &addr);

host = gethostbyaddr((char \*)&addr, 4, AF\_INET);

printf("%s\n", host->h\_name);

while (\*(host->h\_aliases) != NULL)

printf("%s\n", \*host->h\_aliases++);

switch (host->h\_addrtype) {

case AF\_INET:

while (\*(host->h\_addr\_list) != NULL)

printf("%s\n", inet\_ntop(AF\_INET, (\*host->h\_addr\_list++), str, sizeof(str)));

break;

default:

printf("unknown address type\n");

break;

}

return 0;

}

getservbyname

getservbyport

根据服务程序名字或端口号获取信息。使用频率不高。

getaddrinfo

getnameinfo

freeaddrinfo

可同时处理IPv4和IPv6，线程安全的。

### 套接口和地址关联

getsockname

根据accpet返回的sockfd，得到临时端口号

getpeername

根据accpet返回的sockfd，得到远端链接的端口号，在exec后可以获取客户端信息。