系统级编程

# 操作系统演变史

计算机内部复杂度最关键的地方就是操作系统。计算机性能发展的本质上是由硬件发展驱动的，尤其是CPU的性能发展。而**将硬件性能充分发挥出来的关键就是操作系统**，所以操作系统本身其实也是跟随硬件的发展而发展的，**操作系统是软件的运行环境**，操作系统的复杂度直接决定了软件系统的复杂度；

操作系统和性能最相关的就是线程和进程。最早的计算机其实是没有操作系统的，**只有输入、计算和输出功能；用户输入一个指令，计算机完成操作；**大部分时候计算机都是在等待用户输入指令，这样处理性能显然是很低的；

为了解决手工操作带来的低效，**批处理操作系统**应运而生。**批处理简单来说就是先把要执行得到指令预先写下来，形成一个指令清单**，这个**指令清单就是我们常说的“任务”**，然后将任务交给计算机去执行;**批处理系统负责读取”任务”中的指令并进行处理,计算机执行过程中无需人工手工操作,这样就等大大提升性能**;**(批处理系统)**

批处理系统的一个明显缺点:计算机一次只能执行一个任务.(I/0操作过程中,CPU其实是空闲的)

为了进一步提升性能,人们发明了”进程”,用进程来对应一个任务,每个任务都有自己独立的内存空间,进程间互不相关,由操作系统来进行调度;**此时的CPU还没有多核多线程的概念,为了达到多进程并行运行的目的,采取了分时方式(即把CPU的时间分成很多片段,每个片段只能执行某个进程中的指令)**;虽然从操作系统和CPU角度来说,还是串行处理;但是从用户角度来看,由于CPU的处理速度很快,感觉是并行处理的;**(进程)**

多进程虽然要求每个任务都有独立的内存空间,进程间互不相关,但从用户角度来看,两个任务之间能够在运行过程中就进行通信,会让任务设计更灵活;**进程间通信的各种方式:**

1. 管道
2. 消息队列
3. 信号量
4. 共享内存(**进程间通信**)

多进程让多任务能够并行处理，但本身还有缺点，单个进程内部只能串行处理，而**实际上很多进程内部的子任务并不要求是严格按照时间顺序来执行的**，也需要并行处理。**由此人们发明了线程**；线程是进程内部的子任务，但这些子任务都**共享同一份进程数据。为了保证数据的准确性，又发明了互斥锁机制。**有了多线程以后，**操作系统调度的最小单位就变成了线程，而进程变成了操作系统分配资源的最小单位；**

多进程多线程虽然让多任务并行处理的性能大大提升，但本质还是分时系统，并不能做到时间上真正的并行；**解决该问题的方式也是显而易见**：让多个CPU能够同时执行计算任务，从而实现真正意义上的多任务并行；

目前多任务并行方案有3种：

SMP(对称多处理器结构，**目前最流行的多核处理器方案**)

NUMA（非一致存储访问结构）

MPP（海量并行处理结构）

# Linux

Linux是一个计算机操作系统（用于管理与维护计算机的资源，是用户与设备之间的“中间商”）,系统为用户提供了大量的API（用户通过API间接操作设备或系统）；

# 文件描述符

[**内核**](https://baike.baidu.com/item/%E5%86%85%E6%A0%B8/108410?fromModule=lemma_inlink)**（kernel）利用文件描述符（file descriptor）来访问文件**。文件描述符是[非负整数](https://baike.baidu.com/item/%E9%9D%9E%E8%B4%9F%E6%95%B4%E6%95%B0/2951833?fromModule=lemma_inlink)。**打开现存文件或新建文件时，内核会返回一个文件描述符**。读写文件也需要使用文件描述符来指定待读写的文件；

标准输入：0

标准输出：1

标准错误：2

文件描述符范围：0 — OPEN\_MAX-1

# 文件类型

1: 普通文件(-：即文本文件、二进制文件、图片等常规文件)

2: 文件夹(d：目录文件)

3： 字符设备（c:通常为外设，如:键盘）

4: 网络设备(s:即套接字文件，用于实现两个进程通信)

5: 块设备（b：通常为外设，如：硬盘）

6: 管道（p:pipe即命名管道文件）

# 文件/文件夹

## 文件操作步骤



### 打开文件

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

**返回值（文件描述符）**：成功 — 返回值为正整数 ， 失败 — 返回值<0

**参数1：**Pathname（文件路径）

**参数2：**flags

操作方式：

1. O\_RDONLY—只读打开 ，
2. O\_WRONLY—只写打开，
3. O\_RDWR—读写打开，
4. O\_TRUNC—截断 ，
5. O\_CREAT—创建，
6. O\_APPEND—追加

**open()打开时，返回值一般>2？**

答：系统觉得每个进程至少需要标准输出(0)，输入（1），错误输出（2）。（**open函数的返回值:占用最小的且未被打开的文件描述符**）

### 操作文件

* 【写操作】

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

**返回值：**返回实际读的长度

**参数1：**fd（文件描述符）

**参数2：**buf（缓冲区）

**参数3：**count（输入（写）内容的长度）

* 【读操作】

**API函数:**size\_t write(int fd , void \*buf , size\_t count);

**返回值：**返回实际写入的长度

**参数1：**fd（文件描述符）

**参数2：**buf（缓冲区）

**参数3：**count（输出（读）内容长度）

* 【偏移操作】

**API函数:**off\_t lseek(int fd, off\_t offset, int whence);

**作用：用于改变读写一个文件时读写指针位置的**。**指针位置可以是绝对的或相对的；**

**返回值：返回文件大小**（即ftell()的作用）

**参数1：**fd（文件描述符）

**参数2：**offset（为文件指针的位置）

**参数3：**whence（以文件头为参考点：SEEK\_SET，以文件指针的当前位置为参考点：SEEK\_CUR , 以文件结尾为参考点：SEEK\_END）

lseek函数两个作用：

1. 计算文件大小；(**返回值返回文件大小**)
2. 移动读写文件内容的起始位置；（**该函数的作用还是移动读写指针的位置**）

**计算文件大小（**lseek(fd,0,SEEK\_END);**）**

而offset参数和whence参数的解释如下：

* 无论offset为多少，**计算文件大小都是从文件开头开始计算**。而offset参数只是表示要计算到哪个位置停止。
* whence参数为参考点，下面以三种情况作为解释。

1. whence设置为SEEK\_SET（以文件头为参考点）时，offset设置为100，那么文件计算就是从头计算到100这个位置的文件大小。注意，whence设置为SEEK\_SET时，offset只能为正数;
2. whence设置为SEEK\_CUR（以文件指针当前位置为参考点），当offset设为正数则为SEEK\_CUR+offset作为终点位置。当offset设为负数则为SEEK\_CUR-offset作为终点位置。一般情况下，SEEK\_CUR=0；
3. whence设置为SEEK\_END（以文件末尾为参考点），当offset设为正数则为SEEK\_END+offset作为终点位置。当offset设为负数则为SEEK\_END-offset作为终点位置。

Ps:无论怎样，都是从文件头开始计算，我们只能决定结束的位置；

**移动读写文件内容的起始位置**

每个文件都有位置偏移量，也就是当前位置处于文件里面的哪个位置。其作用是为了让read函数和write函数从哪个位置开始读。

whence参数为参考点，下面以三种情况作为解释。

1. whence设置为SEEK\_SET（以文件头为参考点）时，offset设置为100，那么就是从100开始读写文件，直到读写到文件最后一个字符。注意，whence设置为SEEK\_SET时，offset只能为正数
2. whence设置为SEEK\_CUR（以文件指针当前位置为参考点），当offset设为正数则为SEEK\_CUR+offset作为起始位置。当offset设为负数则为SEEK\_CUR-offset作为起始位置。一般情况下，SEEK\_CUR=0；
3. whence设置为SEEK\_END（以文件末尾为参考点），当offset设为正数则为SEEK\_END+offset作为起始位置。当offset设为负数则为SEEK\_END-offset作为起始位置。当SEEK\_END+offset作为起始位置时，此时已经超出了文件内容，所以读出的数据会乱码或者无确定的值。

**Ps:计算文件大小和移动读写文件内容的起始位置：就在于一个是确定终点值(计算文件大小)，一个是确定起始值（移动文件内容的起始位置）。**

### 关闭文件

**API函数:**int close(int fd);

**返回值：**成功：0 ，失败：-1

**参数1：**fd（文件描述符）

**Ps：获取文件信息：**stat(const char\*pathname , struct stat&);

## 文件夹操作步骤



### 打开文件夹

**API函数：**DIR\* opendir(const char\* path)；

**返回值：**成功——返回DIR\*形态的目录流（目录地址），失败——返回为Null;

参数1：path（目录路径）

### 读取文件夹

**API函数：**struct dirent \*readdir(DIR \*dirp);

**返回值：**成功——则返回下个目录进入点，返回目录本身所指的文件（以结构体方式返回）若读取目录结束或有错误时，则返回NULL；

参数1：dirp（目录地址）

struct dirent {

ino\_t d\_ino; /\* inode number \*/

off\_t d\_off; /\* offset to the next dirent \*/（目录文件开头到此目录进入点的位移）

unsigned short d\_reclen; /\* length of this record \*/

unsigned char d\_type; /\* type of file; not supported by all file system types \*/

char d\_name[256]; /\* filename \*/

};

### 关闭文件夹

**API函数：**int closedir(DIR \*dirp);

**返回值：**成功—0 ， 失败—1

**参数1：**dirp（目录地址）

# 进程



### 【含义】

**一个执行中的程序实例**；分配资源的最小单位;

### 【进程上下文】

系统中的**每个程序都是运行在某个进程的上下文中的**。

* **上下文：**

操作系统保持并**跟踪进程运行所需的所有状态信息**;（**这些状态信息（PCB中）就称为上下文，状态包括：存放在存储器中的程序的代码和数据，它的栈、通用目的寄存器的内容、程序计数器、环境变量以及打开文件描述符的集合；**）

* **上下文细分：**

上文：已执行的进程指令和数据（指令和数据存放的位置：相关寄存器与堆栈）；

正文：正在执行；

下文：待执行；

* **上下文切换：**

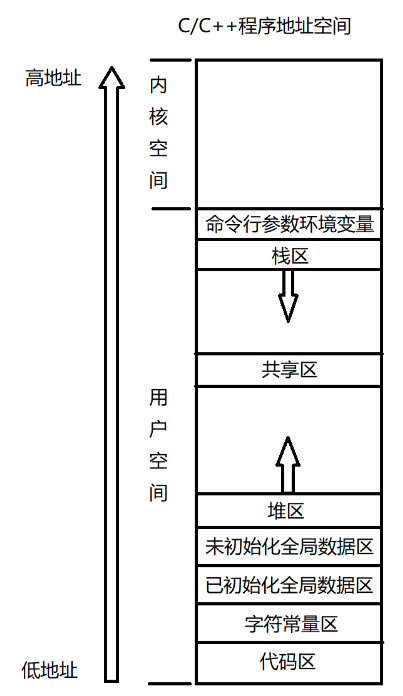
当操作系统决定要把控制权从当前进程转移到新进程时，就会进行**上下文切换（即保存当前进程的上下文、恢复新进程的上下文）**；

### 【进程地址空间】



<https://blog.csdn.net/attemptendeavor/article/details/122539046>

https://blog.csdn.net/Arlingtonroad/article/details/107148527

****

#### 含义

**进程地址空间**本质上就是一种数据结构（mm\_struct），是多个区域的集合；

#### 地址空间区域分类

内核空间：

1. 系统内核中有些代码、数据是所有进程所公用的，所以所有进程的进程地址空间（内核空间）中有一个专门的区域存放公共的内核代码和数据，该区域的内容相同，且该虚拟内存映射到同一物理内存区域；
2. 进程在执行的时候，需要维护进程相关的数据结构，比如页表、task（进程控制块）和mm结构（进程地址空间）、内核栈等，这些数据结构是进程独立的，各个进程之间可能不同。这些数据结构在进程虚拟地址空间（内核空间）中一个专门的区域中；
3. 内核总是驻留在内存中，是操作系统的一部分。内核空间为内核保留，不允许应用程序读写该区域的内容或直接调用内核代码定义的函数；

用户空间：

* **栈区（Stack）:**

栈又称堆栈，由编译器自动分配释放，行为类似数据结构中的栈（先进后出）；

三个用途：

* 1. 为函数内部声明的非静态局部变量(C语言中称“自动变量”)提供存储空间；
  2. 记录函数调用过程相关的维护性信息，称为栈帧(Stack Frame)或过程活动记录(Procedure Activation Record)。它包括函数返回地址、不适合装入寄存器的函数参数及一些寄存器值的保存。除递归调用外，堆栈并非必需。因为编译时可获知局部变量、参数和返回地址所需空间，并将其分配于BSS段（未初始化的全局数据区）；
  3. 临时存储区：用于暂存长算术表达式部分计算结果或alloca()函数分配的栈内内存。

进程中的每个线程都有属于自己的栈；

堆栈既可向下增长(向内存低地址)也可向上增长, 这依赖于具体的实现。本文所述堆栈向下增长；

 堆栈的大小在运行时由内核动态调整。

* **共享区（mmap）:**

共享区也称为内存映射区；内核将硬盘文件的内容直接映射到内存；任何应用程序都可通过Linux的mmap()系统调用或Windows的CreateFileMapping()/MapViewOfFile()请求这种映射；

内存映射是一种方便高效的文件I/O方式，因而被用于装载动态共享库；用户也可创建匿名内存映射，该映射没有对应的文件,可用于存放程序数据。

该区域用于映射可执行文件用到的动态链接库。在Linux 2.4版本中，若可执行文件依赖共享库，则系统会为这些动态库在从0x40000000开始的地址分配相应空间，并在程序装载时将其载入到该空间。在Linux 2.6内核中，共享库的起始地址被往上移动至更靠近栈区的位置。

* **堆区（heap）:**

堆用于存放进程运行时动态分配的内存段，可动态扩张或缩减。堆中内容是匿名的，不能按名字直接访问，只能通过指针间接访问。当进程调用malloc(C)/new(C++)等函数分配内存时，新分配的内存动态添加到堆上(扩张)；当调用free(C)/delete(C++)等函数释放内存时，被释放的内存从堆中剔除(缩减) 。

* **未初始化全局变量区（BSS段）:**

存放以下三种数据：a.未初始化的全局变量和静态局部变量；b.初始值为0的全局变量和静态局部变量(依赖于编译器实现)；c.未定义且初值不为0的符号(该初值即common block的大小)

* **已初始化全局变量区（Data段）:**

数据段通常用于存放程序中已初始化且初值不为0的全局变量和静态局部变量。数据段属于静态内存分配(静态存储区)，可读可写。数据段保存在目标文件中(在嵌入式系统里一般固化在镜像文件中)，其内容由程序初始化。例如，对于全局变量int gVar = 10，必须在目标文件数据段中保存10这个数据，然后在程序加载时复制到相应的内存。

* **字符常量区：**

存放字符常量（数字字符和英文字符）的（如：const char\* p = “hello bit”; // p是指针变量（栈区），p指向的字符常量hello bit(字符常量区)）；程序结束后由系统释放；

* **代码区（text）：**

代码段也称正文段或文本段，通常用于存放程序执行代码(即CPU执行的机器指令)。

代码段指令根据程序设计流程依次执行，对于顺序指令，只会执行一次(每个进程)；若有反复，则需使用跳转指令；若进行递归，则需要借助栈来实现。

代码段指令中包括操作码和操作对象(或对象地址引用)。若操作对象是立即数(具体数值)，将直接包含在代码区中；若是局部数据，将在栈区分配空间，然后引用该数据地址；若位于BSS段和数据段，同样引用该数据地址。

#### 地址空间注意事项

* 1. 进程地址空间不是内存；
  2. 进程地址空间，会在进程的整个生命周期内一直存在，直到进程退出（这也就是全局变量，静态变量，堆区为何会一直存在）；

#### 虚拟地址和物理地址（内存）

任何我们学过的语言，里面的地址（&取地址符）绝对不会是物理地址（内存），而是虚拟地址；虚拟地址是由操作系统给我们提供的；既然是虚拟地址，那一定有某种途径将虚拟地址转换为物理地址，因为数据和代码一定在物理内存上；

虚拟地址转换为物理地址是由操作系统转换完成的；

#### 地址空间：操作系统在给每个进程画饼

**举例：**比如有一个富豪，他拥有10个亿的身家，这个富豪有10个私生子，这10个私生子互相并不知道自己的存在，富豪对自己的每一个私生子都说孩子你好好学习，老爸现在有10个亿的家产，以后就全是你的，请问在这十个私生子的视觉来看，他们认为他们有多少的家产？当然是10亿，当每个私生子向这个富豪要钱时，只要能接受，富豪肯定都会给，不能接受，富豪可以直接拒绝，在这个例子中富豪给每个私生子脑海里建立了虚拟的10个亿，此时每个私生子都认为自己有10个亿，每个人要的钱都是不一样的；

**对比言之：**操作系统默认会给每个进程构成一个地址空间的概念(32位下，地址空间是从000000…0000到FFFFFF…FFF)4GB的空间，每个进程都认为自己有4GB的空间，每个进程都可以向内存申请空间，只要能接受都会给你，不能接受操作系统会直接拒绝，但是并没有什么影响，进程依旧认为自己有4GB的空间。

总结：

* 1. 进程地址空间本质：是进程看待内存的方式，抽象出来的一个概念，内核：struct mm\_struct，这样的每个进程，都认为自己独占系统内存资源(每个私生子都认为自己独占10亿家产)；
  2. 地址空间区域划分本质：将线性地址空间划分成为一个一个的area，[start,end]。
  3. 虚拟地址本质：在[start,end]之间的各个地址叫做虚拟地址；

#### 地址空间和物理内存之间的关系

可执行程序运行起来，需要将代码和数据加载到物理内存当中；

加载方法：页表（进程将自己的代码和数据首先放在虚拟地址空间的对应的区域，在这其中会有一种表结构，叫做页表，页表的核心工作就是完成虚拟地址到物理地址之间的映射，**最终可执行程序的代码和数据可以加载到物理内存的任意位置；然后只需要建立代码和数据与物理内存之间的映射关系，就可以通过虚拟地址找到物理内存的对应地址**）

**所以不同的进程的虚拟地址可以完全一样？**因为每个进程都有各自的页表，每个进程都是独立的进行通过各自页表中虚拟地址到物理地址的映射，去找代码和数据的；

总结：**虚拟地址和物理空间之间是通过页表完成的映射关系；**

### 【进程控制块PCB】

****

进程控制块是存放进程的管理信息和控制信息的**数据结构（就是一个结构体task\_struct），进程的灵魂，维护进程运行是所需要的信息**；在创建进程时，建立**PCB（存放在系统空间中，只有操作系统才能存取）**，伴随进程运行的全过程，直到进程撤销而撤销；

**不同的操作系统中对进程的管理和控制机制不同**，**PCB**中的信息存在差异，通常**PCB**包含如下信息：

* 1. **进程标识符：**唯一标识符（PID，进程标识符范围：**0～32767**）
  2. **进程当前状态：**说明进程当前所处的状态（new , ready , running , waiting 或 blocked），为了方便管理，**系统设计时会将相同状态的进程组成一个队列(见下文“进程队列”)**，如就绪进程队列，等待进程则要根据等待的事件组成多个等待队列，如等待打印机队列、等待磁盘I/O完成队列等等；
  3. **进程相应的程序和数据地址（进程地址空间）：**将PCB与其程序和数据联系起来；
  4. **进程资源清单：**列出除CPU以外的资源记录，如拥有I/O设备，打开的文件列表；
  5. **进程优先级：**进程的优先级反映进程的紧迫程度，通常由用户指定和系统设置。UNIX系统采用用户设置和系统计算相结合的方式确定进程的优先级。
  6. **CPU现场保护区：**当进程因某种原因不能继续占用CPU时（等待打印机），释放CPU，这时就要将CPU的各种状态信息保护起来，为将来再次得到处理机恢复CPU的各种状态，继续进行。
  7. **进程同步与通信机制：用于实现进程间的互斥、同步和通信所需的信号量等**。
  8. **进程所在队列PCB的链接字：**根据进程所处的现行状态，进程相应的PCB参加到不同队列中。PCB链接字指出该进程所在队列中下一个进程PCB的首地址。
  9. **与进程相关的其他信息：**如进程记账信息，进程占用CPU的时间等。

### 【进程组成】

提到进程需要知道这三个东西：task\_struct,mm\_struct,页表;由进程常见的数据结构task\_struct(进程控制块)和程序组成；

**程序是进程的“躯体”，包括代码和数据两部分；**

**进程控制块中有**struct mm\_struct(地址空间))；

### 【进程队列PCB链】

虽然叫队列，但是底层都是有链表组成的；

系统中的进程队列分为以下3类：

* **就绪队列：一般整个系统中有一个就绪队列**。所有处于就绪态的进程按照某种组织方式排在这一队列中，进程入队列和出队列的次序与处理器调度算法有关。
* **等待队列：每个等待事件有一个等待队列**。当进程等待某一事件发生时，进入与该事件相关的等待队列中；当某事件发生时，与该事件有关的一个或多个进程离开响应的等待队列，进入就绪队列。
* **运行队列：在单处理器中只有一个运行队列**，在多处理器中每个CPU各有一个运行队列，**每个队列中只有一个进程**。

### 【状态及状态切换】

四种状态：

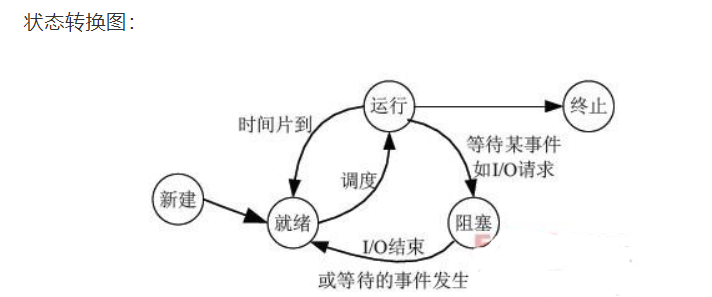
* + 1. **运行态：**进程占有处理器资源，正在运行。在单处理器系统中任一时刻只能有一个进程处于此种状态。**（万事俱备，也获得东风）**
    2. **就绪态:**进程本身**具备运行条件**，但是由于处理器的数量**少于**可运行进程的数量，暂未投入运行，即相当于**等待处理器资源；（万事具备，只欠东风（CPU））**

（就绪队列）----获取CPU----》运行态

* + 1. **等待态:**也称挂起态（Suspend）、阻塞态（block）、睡眠态（sleep）。**进程本身不具备运行条件，**即使分给其处理器也不能运行。进程正在等待某一时间发生，如等待某一资源被释放，等待与该进程相关的数据传输的完成信号等；**（万事不具备）**
    2. **结束态：**进程退出的状态，进程退出

状态切换：

状态切换也称上下文切换；



    Ps: **（1）没有直接从wait->run：为了减少系统开销**

**（2）**没有ready->wait**：没有资源**

发生进程上下文切换有哪些场景？

* 为了保证所有进程可以得到公平调度，CPU 时间被划分为一段段的时间片，这些时间片再被轮流分配给各个进程。这样，当某个进程的时间片耗尽了，进程就从运行状态变为就绪状态，系统从就绪队列选择另外一个进程运行；
* 进程在系统资源不足（比如内存不足）时，要等到资源满足后才可以运行，这个时候进程也会被挂起，并由系统调度其他进程运行；
* 当进程通过睡眠函数 sleep 这样的方法将自己主动挂起时，自然也会重新调度；
* 当有优先级更高的进程运行时，为了保证高优先级进程的运行，当前进程会被挂起，由高优先级进程来运行；
* 发生硬件中断时，CPU 上的进程会被中断挂起，转而执行内核中的中断服务程序；

### 【进程类型和特性】

1. **系统进程：**

这类进程属于操作系统的一部分，它们运行操作系统程序，完成操作系统的某些功能，也被成为守护进程；

1. **用户进程：**

这类进程运行用户程序，直接为用户服务（程序是进程的一部分）；

1. **无论系统进程还是用户进程，都有如下特性：**
2. 并发性：可以与其他进程一到在宏观上同时向前推进；
3. 动态性：进程是执行中的程序；
4. 独立性：进程是调度的基本单位，它可以获得处理器并参与并发执行；
5. 交互性：进程在运行过程中可能会与其他进程发生直接或间接的相互作用；
6. 异步性：每个进程都以其相对独立、不可预知的速度向前推进；
7. 结构性：每个进程都有一个进程控制块；

### 【进程号】

每个进程都由一个进程号来标识，其类型为**pid\_t**，进程号的范围：**0～32767**；

**进程号特点：**

进程号由系统随机给当前进程分配，不能自己控制；

进程号总是唯一的，但进程终止后，进程号可以再次使用；

1. 获取当前进程的进程号：getpid();
2. 获取当前进程的父进程号：getppid();
3. 获取当前进程所在进程组的ID：getpgid();

### 【进程使用步骤】

第一步：创建子进程

**API函数：**pid\_t fork(void);

**作用：**在已有的进程基础上创建一个子进程；

**返回值：**

**成功（**>0：返回值是子进程的进程号，但以下空间还是表示父进程的代码区； =0：返回值是子进程的进程号，以下空间为子进程的代码区**）**

**失败（**-1：返回给父进程，子进程不会创建**）**

**详解：**

使用fork函数得到的**子进程是父进程的一个复制品，它从父进程处拷贝了整个进程的地址空间（地址空间包括：**进程上下文、进程堆栈、打开的文件描述符、信号控制设定、进程优先级、进程组号等，子进程所独有的只有它的进程号和计时器等**）；**

头脑内始终要有一个概念：在调用fork()函数前是一个进程在执行这段代码，而**调用fork()函数后就变成了两个进程在执行这段代码（就是每个进程中都有这段代码）**。两个进程所执行的代码完全相同，都会执行接下来的if-else判断语句块；

**拓展：**

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

创建子进程；

子进程与父进程共享数据段；保证子进程先运行，在调用exec与exit之前与父进程数据是共享的；在调用exec与exit之后才可能被调度。**如果在调用**exec或exit**之前子进程依赖于父进程的进一步动作，则会导致死锁；**

第二步：等待子进程退出

**API函数：**int wait(int\* status) 阻塞函数（同步函数）

**作用：**等待子进程退出，并善后处理(**解决“孤儿进程”和“僵尸进程”问题**)

**参数1：**status存储子进程的状态（**子进程退出，有时父进程必须知道子进程的退出状态；**）

**详解：**

* + 1. 孤儿进程：父进程先于子进程退出，子进程则成为孤儿进程，那么**孤儿进程的父进程则为1号进程**。
    2. 僵尸(zombie)进程：子进程退出时，父进程不为子进程善后处理。
    3. **获取status中的返回值**：WEXITSTATUS（status）；
    4. **获取是否异常退出状态**：WIFEXITED (status)；非0：是正常退出，0：是异常退出；
    5. **进程退出**：**正常退出（**先清空缓冲区，最后释放空间 --- main中：return 0; exit(0) ;**） ；异常退出：直接退出（**运行异常）；

### 【守护进程创建步骤】



**含义：**

是运行在后台的一种进程。它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；不受用户的控制所影响。守护进程的父进程都是1，即所有精灵进程都是孤儿进程；

**创建精灵进程的步骤：**

1. 不受终端(子进程)，用户影响：

pid\_t setsid(void);//创建新的会话；

2.修改文件掩码:

mode\_t umask(mode\_t mask);

3.修改指定路径:

int chdir(const char \*path); (**修改指定工作目录**(当打开文件时，是绝对路径时,这个没有作用，相对路径时，才有作用);根目录和根目录下的目录没有权限创建文件)

4.关闭文件描述符

sysconf(\_SC\_OPEN\_MAX):获取配置信息

\_SC\_OPEN\_MAX : 一个进程打开文件的最大数(最少不能小于20)

### 【进程空间与父子进程运行】

1. 进程间的空间：是独立的
2. 父子进程运行：是抢占式的

# 进程间通信（Inter-process Communication IPC）

* 1. 匿名管道
  2. 命名管道
  3. 共享内存
  4. 消息队列
  5. 信号

--------------------------以上都是“**同一主机不同进程之间的通信**”-----------------------------

* 1. 网络通信（Socket）：“不同地域”或“同一地域”的不同主机之间的通信；

### 【通信传输的方式】

1. **单工：**单工就是指终端A只能发信号，而终端B只能接收信号，通信是单向的。
2. **半双工：**半双工就是指终端A能发信号给终端B，终端B也能发信号给终端A，但是，这**两个过程不能同时进行；**（一个信道）
3. **全双工：**全双工就是指终端A和终端B都能作为信号的收发端，并且**信号的收发可以同时进行**（两个信道），**两个单工通信方式的结合**。相对来说采用全双工对信道的带宽要求比半双工要高。

### 【匿名管道】



1. **含义**

是**存放在内存中**的一种特殊的文件结构（**文件描述符标识在内存中的位置**）；

1. **特点**
   * + 1. 读写两端；
       2. 管道是半双工的，但是数据传输的方向是单向的；
       3. 只能应用于有关系的进程(父子进程，兄弟进程)；
2. **Linux案例**

cat a.txt （”标准输出”作为 grep 的 “标准输入”） | grep -i LIFEI (“|”：匿名管道) \

1. **匿名管道使用步骤**

第一步：创建匿名管道（作用于父子进程）

**API函数：**int pipe(int pipefd[2]);

**作用：**创建一个匿名管道；

**参数1：**pipefd存储读（fd[0]）/写(fd[1])端的文件描述符

**返回值：**成功>0，失败<0 **；**

1. **管道破裂4种情况**

创建管道成功，写入端进程关闭写入端管道，读取端进程读取没有意义，读取时失败（read返回-1），直接返回不等待；

创建管道成功，写入端进程保持写入端管道打开，读取端进程读取信息时，若管道中没有信息则等待管道信息中有信息被写入，否则一直阻塞。

创建管道成功，读取端进程优先于关闭读取端管道，写入进程写入信息没有意义，写入失败；

创建管道成功，读取端进程不关闭读取端管道，写入端进程也不关闭写入端管道，但是写入端一直写入信息时达到管道为满时，写入端写入被阻塞。

### 【命名管道】



1. **含义**

是一个特殊的文件(存放在外存中)，进程同步；

1. **特点**

每个FIFO文件都有一个路径名与之对应，从而允许不同关系进程可以实现通信；

1. **命名管道使用步骤**

第一步：创建命名管道

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

**作用：**创建一个命名管道文件

**参数1：**管道文件路径（包括名称）

**参数2：**管道文件权限

**返回值：**成功（0） ， 失败（-1，EEXIST:因为创建的文件存在）

第二步：文件的操作步骤：

1、打开

2、操作

3、关闭

1. **命名管道阻塞4种情况**

若写进程打开，但读进程不打开，则写进程阻塞在open,等待读进程打开;

若读进程打开，但写进程不打开，则读进程阻塞在open,等待写进程打开;

若读写都打开时，不写入消息或者文件为空时，阻塞读，直到有信息写入;

若读写都打开时，写入端直接退出时，读取端直接读取信息直到信息读完时 不阻塞等待;

### 【共享内存】



1. **含义**

允许两个及以上的不同的进程（没有关系）共享一个的存储区（逻辑内存：地址空间）；

**共享内存原理**：在物理内存中直接开辟一片空间（共享区），并将共享区空间映射到各个进程的虚拟地址空间的共享区；这时候进程就可以通过虚拟地址来直接对共享内存进行操作。

1. **特点**
2. **当一个进程改变了存储区的内容时，其他进程也会察觉到相应内容的改变。**
3. 数据直接写入到存储区中，不用若干次拷贝；
4. 是所有IPC中最快的通信；
5. **共享内存使用步骤**

**第一步：**生成唯一键值

**API函数：**key\_t ftok(const char\* pathname , int proj\_id);

**作用：**生成唯一键值，以便生成唯一的共享内存；

**参数1：**pathname为一个路径名称；

**参数2：**proj\_id代表项目id,为0—255之间的一个数值；

**返回值：**成功（返回键值），失败（-1）

**第二步：**创建共享内存

**API函数：**int shmget(key\_t key, size\_t size, int shmflg);

**作用：**创建共享内存，返回共享内存标识符；

**参数1：**key生成的唯一键值；

**参数2：**size共享内存的大小（以“字节”为单位）；

**参数3：shmflg**设置共享内存的操作和权限（与“文件的访问权限”一样）；

**返回值：**成功（**共享内存标识符ID**），失败（-1）

**shmflg**设置操作：

**IPC\_CREAT:**不管是否存在该块共享内存，则都返回该共享内存ID，若不存在,则创建共享内存；

**IPC\_EXCL：**不管有没有该块共享内存，shmget都返回-1；

**IPC\_CREAT|IPC\_EXCL（常用）：**如果没有该块共享内存，创建并返回共享内存ID, 若已有该块共享内存，则返回-1；

**shmflg**设置权限（Linux：r - 4 , w - 2 , x - 1）：

IPC\_CREAT|IPC\_EXCL|**0666**

**第三步：**地址空间映射

**API函数：**void \*shmat(int shmid, const void \*shmaddr, int shmflg);

**作用：**将共享内存的逻辑地址映射（连接）到当前进程的虚拟地址空间上，以便当前进程使用共享内存；

**参数1：**shmid共享内存标识符；

**参数2：s**hmaddr指定共享内存映射到一个合适的的地址空间（**为NULL(常用)**:当前进程,**不为NULL**：系统会根据参数及地址边界对齐等分配一个合适的地址）；

**参数3：**shmflg设置连接到进程虚拟地址的权限（读写权限（shmflg:0，常用），IPC\_RDONLY:只读权限，IPC\_REMAP:替换位于shmaddr处的任意既有映射)

**返回值：**成功（返回一个指向共享内存第一个字节的指针），失败（-1）；

**第四步：**给共享内存赋值（共享内存读写）

strcpy()等函数；

**第五步：**解除内存映射（可选）

**API函数：**int shmdt(const void \*shmaddr);

**作用：**将共享内存从当前进程中分离（解除内存映射）

**参数1：**shmaddr共享内存地址（**shmat函数的返回值**）

**返回值：** 成功（0），失败（-1）

**第六步：**删除共享内存（可选）

**API函数：**int shmctl(int shm\_id, int command, struct shmid\_ds \*buf);

**作用：**控制共享内存（**进程结束后会自动删除共享内存**）；

**参数1：**shm\_id共享内存标识符

**参数2：**command指定shmctl控制共享内存的方式（**IPC\_STAT:**将buf结构中的数据设置为共享内存的当前关联值（内核维护了一个shmid\_ds结构）；**IPC\_SET：**如果进程有足够权限，就把共享内存的当前关联值（内核维护了一个shmid\_ds结构）设置为buf结构中的值；**IPC\_RMID**：删除共享内存段）

**参数3（通常为NULL）：**buf是一个结构指针，它指向共享内存模式和访问权限的结构

**返回值：**成功（0），失败（-1）

1. **共享内存注意事项**

多进程共享内存读写，需要注意访问同步；

### 【消息队列】

1. **含义**

存放消息的链表容器，存放在内核中，他只是遵循了队列容器的特性（FIFO）;

一个消息队列由一个标识符（即队列ID）来标识；

1. **特点**
2. 消息队列可以实现消息的随机查询，满足队列的特点但不一定要以先进先出的次序读取，**可以按消息的类型读取**；
3. 消息队列允许一个或多个进程向它写入或者读取消息
4. 与无名管道、[有名管道](https://so.csdn.net/so/search?q=%E6%9C%89%E5%90%8D%E7%AE%A1%E9%81%93&spm=1001.2101.3001.7020)一样，从消息队列中读出消息，消息队列中数据会被删除;
5. 消息队列是面向记录的，其中的消息具有特定的格式以及特定的[优先级](https://so.csdn.net/so/search?q=%E4%BC%98%E5%85%88%E7%BA%A7&spm=1001.2101.3001.7020)；
6. 只有内核重启或人工删除时，该消息才会被删除，若不人工删除消息队列，消息队列会一直存在于[内存](https://so.csdn.net/so/search?q=%E5%86%85%E5%AD%98&spm=1001.2101.3001.7020)中；
7. **消息结构体**

struct msgbuf {

long mtype; /\* message type, must be > 0 \*/

char mtext[1]; /\* message data \*/

};

1. **消息队列使用步骤**

**第一步：**生成唯一键值

**API函数：**key\_t ftok(const char\* pathname , int proj\_id);

**作用：**生成唯一键值，以便生成唯一的消息队列；

**参数1：**pathname为一个路径名称；

**参数2：**proj\_id代表项目id,为0—255之间的一个数值；

**返回值：**成功（返回键值），失败（-1）

**第二步：**创建消息队列

**API函数：**int msgget(key\_t key, int msgflg);

**作用：**创建消息队列；

**参数1：**key生成的唯一键值；

**参数2：**msgflg所需要的操作和权限(同**“创建共享内存”**的shmflg标志位)，可以用来控制创建一个消息队列；

**返回值：**成功（**消息队列标识符ID**），失败（-1）；

**第三步：使用消息队列——**往消息队列中写消息  
 **API函数：**int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg);

**作用：**往消息队列中发送消息；

**参数1：**msqid消息队列标识符ID(msgget的返回值);

**参数2：**msgp消息结构体（mtext调用者自己定义的结构体）；

**参数3：**msgsz消息结构体中的字符数组（mtext存放了需要发送的消息数据）大小;

**参数4：**msgflg标志位（权限位），值可以为**0(常用)**，IPC\_NOWAIT；（**为0时**：当消息队列满时，msgsnd将会阻塞，直到消息能写进消息队列或消息队列被删除时；**为IPC\_NOWAIT时**：消息队列满了，msgsnd函数将不会等待，立即返回错误EAGAIN标志；）

**返回值：**成功（0），失败（-1）

**第四步：使用消息队列——**从消息队列中读消息

**API函数：**ssize\_t msgrcv(int msqid, void \*msgp, size\_t msgsz, long msgtyp, int msgflg);

**作用：**接受消息队列中的信息；

**参数1:** msqid消息队列标识符ID(msgget的返回值);

**参数2:** msgp消息结构体（调用者自己定义的结构体）；

**参数3:** msgsz消息结构体中的字符数组（存放了需要接受的消息数据）大小;

**参数4: msgtyp消息结构体中的消息类型**；（**msgtyp为0时**：返回队列中的第一个消息；**msgtyp为>0时**：返回队列中消息类型为msgtype的第一个消息；**msgtyp为<0时**：返回“队列中消息类型值”小于等于“msgtype绝对值”的消息，若该消息有若干个，则**取消息类型值最小的消息**）；

**参数5：**msgflg值可以为0,IPC\_NOWAIT, IPC\_EXCEPT；（**msgflg为0时**：阻塞式接收消息，即消息队列中没有消息时，msgrcv函数一直阻塞等待；**msgflg为IPC\_NOWAIT时**：如果队列中没有请求类型的消息，则立即返回，系统调用失败，错误号设置为ENOMSG回错误；**msgflg为IPC\_EXCEPT时**：与大于0的msgtype配合使用返回队列中第一个消息类型**不为msgtype**的消息）

**第五步：控制消息队列（**msgctl(id,IPC\_RMID,NULL);删除id号的消息队列**）**

**API函数：**int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);

**作用：**删除消息队列，也可以获取和改变消息队列的状态；

**参数1：**msqid消息队列标识符ID

**参数2：**cmd指定msgctl控制消息队列的方式（**IPC\_STAT:**将buf结构中的数据设置为消息队列的当前关联值(内核维护的msqid\_ds结构)；**IPC\_SET：**改变消息队列状态，即把消息队列的当前关联值(内核维护的msqid\_ds结构)设置为buf结构中的值（uid,gid,node等）；**IPC\_RMID**：删除消息队列）

**参数3（通常为NULL）：**buf是一个结构指针，它指向消息队列模式和访问权限的结构

**返回值：**成功（0），失败（-1）

1. **消息队列注意事项**

### 【信号】

<https://blog.csdn.net/caogenwangbaoqiang/article/details/79688530>



#### 含义

signal, 又简称为信号（软中断信号 ，中断的一种模拟）用来**通知进程发生了异步事件；**

（查看“从零开始学习架构（扩展知识）”文档，**I/O操作是由一个用户进程或硬件触发（触发程序：键盘，鼠标，管道破裂等软硬件），交由内核处理完成后，通知另外一个用户进程；**）

#### 原理

信号是进程间通信机制中唯一的异步通信机制（异步通知），一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达。**进程之间可以互相通过系统调用kill发送软中断信号**。内核也可以因为内部事件而给进程发送信号，通知进程发生了某个事件。信号机制除了基本通知功能外，还可以传递附加信息。

#### 信号分类



从两个角度对信号进行分类：

可靠性方面：可靠信号与不可靠信号；

时间关系上：实时信号与非实时信号；

**Linux查看信号**：kill –l 命令；

共62种信号，**前31种：**不可靠信号（非实时信号），**后31种：**可靠信号（实时信号）；

**不可靠信号和可靠信号的区别：**

* 这里的不可靠主要是不支持信号队列，就是当多个信号发生在进程中的时候（收到信号的速度超过进程处理信号的速度的时候），这些没来的及处理的信号就会被丢掉，仅仅留下一个信号。
* 可靠信号是多个信号发送到进程的时候（收到信号的速度超过进程处理信号的速度的时候），这些没来的及处理的信号就会排入进程的队列。等进程有机会来处理的时候，依次再处理，信号不丢失；

#### 信号响应方式

* 系统默认（缺省）处理：SIG\_DFL（本质是：void(\*)(int),函数指针），执行缺省操作；
* 捕捉信号处理，即用户自定义处理函数处理；
* 忽略信号处理：SIG\_IGN（本质是：void(\*)(int),函数指针）cat,**但SIGKILL,SIGSTOP不能被忽略；**

#### 信号状态字

在进程控制块(PCB)中的结构体中，有三个比较重要的变量，分别是：信号屏蔽状态字，信号未决状态字，是否忽略标志

* **信号屏蔽状态字(block)**：64bits，每一位代表该进程对对应号码的信号是否屏蔽：**1是屏蔽，0是不屏蔽；**
* **信号未决状态字(pending)**：64bits，每一位代表该进程对对应号码的信号的状态：**1是未决，0是不抵达；**
* **是否忽略标志**：？？？？

#### 信号生命周期及处理过程分析

**信号生命周期**：信号产生 –> 信号注册 –> 信号在进程中注销 –> 信号处理函数执行完毕**；**

1. **信号产生：**

触发信号的事件发生（键盘，鼠标，管道破裂等软硬件）；

1. **信号注册：**

指的是在目标进程中注册，该目标进程中有未决（未决定：从信号产生到信号处理之间的状态）信号的信息；

**信号注册过程：**

struct sigpending{

struct sigqueue \*head ，\*\*tail; //未决信号链（head:链的头部，tail:链的尾部）

sigset\_t signal;//未决信号集，位向量（其中每一位对应Linux系统中的一个信号）

};

struct sigqueue{

struct sigqueue \*next;

siginfo\_t info; //信号所携带的信息

}

**sigqueue结构组成的链称之为未决信号链；**

信号注册的过程就是将信号值加入到进程的未决信号集sigset\_t中，将信号所携带的信息加入到进程的未决信号链的某一个sigqueue结构体中的siginfo\_t上；

对于可靠信号，可能存在多个未决信号的sigqueue结构，对于每次信号到来都会注册。而对于不可靠信号，只会注册一次，只有一个sigqueue结构；

**只要信号在进程的未决信号集中**，表明进程已经知道信号了，还没来的及处理或者这些信号阻塞；

1. **信号注销（由内核完成）**

在进程的执行过程中，每次从系统调用或中断返回用户空间的时候，都会检查是否有信号没有被处理。如果这些信号没有被阻塞，那么就调用相应的信号处理函数来处理这些信号；（调用信号处理函数之前，进程会把信号在未决信号链中的sigqueue结构卸掉。是否从未决信号集中把信号删除掉，对于实时信号与非实时信号是不相同的。）

* **非实时信号:**由于非实时信号在未决信号链中只有一个sigqueue结构，因此将它删除的同时将信号从未决信号集中删除。
* **实时信号:**由于实时信号在未决信号链中可能有多个sigqueue结构，如果只有一个，也将信号从未决信号集中删除掉。如果有多个那么不从未决信号集中删除信号，注销完毕。

1. **信号执行（有内核完成）**

执行处理函数，本次信号在进程中响应完毕；

#### 信号使用步骤



第一步：信号注册

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

**API函数：**sighandler\_t signal(int signum, sighandler\_t handler);

**作用：**注册信号，当进程接收到该信号时，执行对应的处理函数；

**参数1：**signum 信号分类中的62种信号值；

**参数2：**handler信号响应的方式；

**返回值：**成功（返回当前信号处理程序的上一个信号处理程序），失败（SIG\_ERR）；

第二步：发送信号（**2种**）

**API函数：**int kill(pid\_t pid, int sig);

**作用：**给指定进程发送一个信号；

**参数1：**进程标识符ID（**可以通过某种IPC通信，获取另外进程的进程号**）；

**参数2：**sig 62种信号值；

**返回值：**成功（0），失败（-1）；

**API函数：**int raise(int sig);

**作用：**给本进程发送一个信号；

**参数1：**sig 62种信号值；

**返回值：**成功（0），失败（-1）；

#### 屏蔽信号集来影响当前进程的未决信号集



**第一步：**自定义一个信号集；

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

**第二步：**清空信号集或填满信号集（初始化）；

**API函数：**int sigemptyset(sigset\_t \*set);

**作用：**清空信号集，即将信号集的每一位都置0；

**参数1：**set信号集；

**返回值：**成功（0），失败（-1）

**API函数：**int sigfillset(sigset\_t \*set);

**作用：**填满信号集，即将信号集的每一位都置1；

**参数1：**set信号集；

**返回值：**成功（0），失败（-1）

**第三步：**添加（或删除）指定信号到信号集中；

**API函数：**int sigaddset(sigset\_t \*set, int signum);

**作用：**将某个信号加入信号集；

**参数1：**set信号集；

参数2：signum信号值；

**返回值：**成功（0），失败（-1）

**API函数：**int sigdelset(sigset\_t \*set, int signum);

**作用：**将某个信号从信号集中删除；

**参数1：**set信号集；

参数2：signum信号值；

**返回值：**成功（0），失败（-1）；

**API函数：**int sigismember(const sigset\_t \*set, int signum);

**作用：判断某个信号是否在信号集中**；

**参数1：**set信号集；

参数2：signum信号值；

**返回值：**存在（1），不存在（0），错误（-1）；

**第四步：屏蔽及解除屏蔽 内核信号集（屏蔽状态字：1为屏蔽，0为不屏蔽）**

**API函数：**int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset)；

**作用：**用来屏蔽信号、解除屏蔽；

**参数1：**

假设当前的信号屏蔽字为mask，如下(set表示屏蔽信号集)：(**不需要管mask是什么值，只需要知道屏蔽或解除屏蔽哪个信号，就将其设为1就行了**)

* **SIG\_BLOCK:** 当how设置为此值，set表示需要屏蔽的信号集。相当于 mask = mask | set。添加位1时使用或，因为或时只有0才能保持相同。
* **SIG\_UNBLOCK:** 当how设置为此值，set表示需要解除屏蔽的信号。相当于 mask = mask & ～set。置为位0时使用“取反后位与”，即可将位变成0。但是传参时需要把想要解除屏蔽的信号置为1，其余全部为0。例如原mask为101,想要将第一个信号清除屏蔽，则需要传100，**解除屏蔽的位必须是1**，其余为0，这样取反后为011,101&011结果就变成了001。但是这一步系统会帮我们做，我们只需要传参1和参2即可。
* **SIG\_SETMASK:** 当how设置为此值，set表示用于替代原始屏蔽及的新屏蔽集。相当于 mask = set。不过该标志位很少用，因为你不知道目前的信号状态，很难单独改变某个信号，会造成改变很多信号的屏蔽状态。

若调用sigprocmask解除了对当前若干个信号的阻塞，则在sigprocmask返回前，至少将原本阻塞的其中一个信号递达。

从上面可以看到，我们只需要通过调整how和传入的set即可做到屏蔽信号和解除屏蔽，系统内部会自动将参2和参3进行运算，不需要我们自行运算，参3不需要处理。

**参数2：**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。

**参数3：**传出参数，保存旧的信号屏蔽集。

**返回值：**成功（0），失败（-1）;

**第五步：读取当前进程的未决信号集**

**API函数：**int sigpending(sigset\_t \*set);

**作用：**读取当前进程的未决信号集；

**参数1：**set:自定义信号集；

**返回值：**成功（0），失败（-1）；

### 【网络通信】

https://blog.csdn.net/u010698107/article/details/113528747

#### 含义

“不同地域”或“同一地域”的不同主机之间的通信；

#### 协议组成

**ISO(International(国际) Organization for Standardization)**

双方约定的规则；（**OSI模型:**开放式系统互联通信参考模型）

协议表示：三部分（语法（规定写法）+语义（通俗解释）+规则（做什么））

#### OSI模型

**发送方 接收方**

**7 应用层 -------------------- 应用层 Application**

**6 表示层 -------------------- 表示层 Presentation**

**5 会话层 -------------------- 会话层 Session**

**4 传输层 -------------------- 传输层 Transport**

**3 网络层 -------------------- 网络层 Network**

**2 链路层 -------------------- 链路层 Data Link**

**1 物理层 -------------------- 物理层 Physical**

**物理层：**

**定义：**网络通信的**数据传输介质**（网线、光纤）,由连接不同结点的电缆与设备共同构成；

**主要功能：**利用传输介质为**数据链路层提供物理连接服务**，负责处理数据传输并监控数据出错率，以便数据流的透明传输；

**传输数据：**比特位（电信号、电流）

**数据链路层：（？？？？）**

**主要功能：**在物理层提供的服务基础上，在通信的实体间建立数据链路连接（？？？），并采用差错控制与流量控制方法，使有差错的物理线路变成无差错的数据链路。

**传输数据：**数据帧（格式化数据、根据**字节分组**）

**网络层:（IP协议、ARP协议、路由协议 ？？？）**

**主要功能：**为数据在结点（主机）之间传输创建逻辑链路，通过**路由选择算法**为分组通过通信子网选择最适当的路径，以及实现拥塞控制、**网络互联（不同地域之间的网络中的两个主机之间提供连接）**等功能；

**传输数据：**数据报

**两个问题：**

1.发送方 如何能接受到mac地址？

2.发送方 如何能知道接受者与自己是否属于同一子网？

**传输层：（TCP、UDP）**

**主要功能：**向**用户提供可靠的端到端(End-to-End)服务，处理数据包错误、数据包次序**，以及其他一些关键传输问题。**传输层向高层屏蔽了下层数据通信的细节**，因此，它是计算机通信体系结构中关键的一层；

**传输数据：用户**数据报/报文段

**会话层：**

**主要功能**：**负责维持两个结点（主机）之间的传输链接（即发起会话与接受会话，前提是设备之间互相认识）**，以便确保点到点传输不中断，以及管理数据交换等功能。

**表示层：**

**主要功能：用于处理在两个通信系统中交换信息的表示方式（**确保一个系统发出的信息可以被另一个系统**读取到且读懂）**，主要包括数据格式变换、数据加密与解密、数据压缩与恢复等功能。

如果有必要，表示层会通过使用一种**通用格式**来实现多种数据格式之间的转换；

**应用层：**

**主要功能**：**为应用软件提供了很多服务（操作系统）**，例如文件服务器、数据库服务、电子邮件与其他网络软件服务。

**传输数据：**操作系统提供的服务

**Ps:上一层为下一层提供数据，下一层为上一层提供服务**

#### TCP/IP四层模型

##### 概念

TCP/IP协议成为了当今英特网的**通信协议**（国际标准协议）,也是国际互联网络的基础；（**通信协议：**是对计算机必须遵守的规则描述，只有遵守这些规则，计算机之间才能进行通信；）

TCP/IP协议不仅仅指的是TCP和IP两个协议，而是指一个由FTP、SMTP、ICMP(互联网控制报文协议)、TCP、UDP、IP、HTTP（超文本传输协议）、POP3等协议 构成的**协议簇**， 只是因为在TCP/IP协议中TCP协议和IP协议最具代表性，所以被称为TCP/IP协议；

TCP/IP定义了计算机操作系统如何连入互联网，以及数据传输的标准；

TCP/IP是为了解决不同系统的计算机之间的传输通信而提出的一个标准，不同系统之间的计算机采用同一种协议后，就能相互通信，从而建立网络连接，实现资源共享和网络通信（就像两个不同国家的人，可以使用同一种语言交流了）。

##### TCP/IP分层结构

abc 应用层 应用层 abc

[1]abc 传输层（TCP/UDP协议） 传输层 [1]abc

[2][1]abc 网络层 网络层 [2][1]abc

[3][2][1]abc 网络接口层(物理/数据链路层) 网络接口层 [3][2][1]abc

**应用层的主要协议（数据报称为：字节流）：**Telnet、FTP、SMTP、DNS、NNTP、HTTP**等**，是用来接收来自传输层的数据或者按不同应用程序的要求将数据传输至传输层；

* **Telnet协议：**允许一台机器上的用户登录远程机器进行工作；
* **FTP协议：**提供将文件从一台机器上移到另一台机器上的有效方法；
* **SMTP协议：**电子邮件收发；
* **DNS协议：**用于把主机名映射到网络地址；
* **NNTP协议：**用于新闻发布、检索和获取；
* **HTTP协议：**用于在WWW上获取主页；

**传输层的主要协议（数据报称为：段）：**TCP（Transmission Control Protocol）和UDP（User Datagram Protocol）, 是使用者使用平台和计算机信息网内部结合的通道，可以实现数据传输与数据共享；

* **TCP协议：面向连接的协议**，提供**可靠的报文传输**和上层应用的连接服务；除了基本数据传输外，它还有可靠性保证、流量控制、多路复用、优先权和安全性控制等功能；
* **UDP协议：面向无连接的不可靠传输协议**，主要用于不需要TCP排序和流量控制等功能的应用程序；

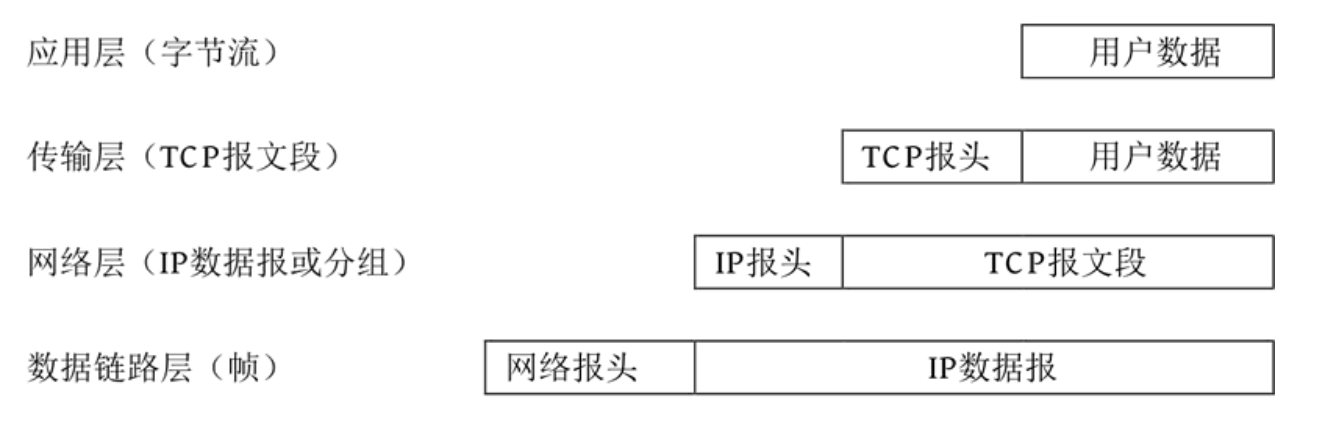
**网络层的主要协议（数据报称为：数据报）：IP、ICMP/IGMP、ARP/RARP**，是使主机可以把数据报（Packet，或称为**分组**）发往任何网络，并使分组独立的传向目标。这些分组经由不同的网络**到达的顺序**和**发送的顺序**可能不同；

**网络接口层的主要协议（数据报称为：帧）：网络接口及硬件层**，负责接收IP层的IP数据报，通过网络向外发送，或接收从网络上来的物理帧，抽出IP数据包，向网络层（IP）发送，该层是主机和网络的实际连接层；

网络接口层下面就是实体线路（比如以太网络、光纤网络等）。网络接口层有**以太网、令牌环网等标准**，负责网卡设备的驱动、**帧同步**（就是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。交换机可以在不同的网络接口层的网络之间（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）转发数据帧，由于不同网络接口层的帧格式不同，交换机要将进来的数据报拆掉报头重新封装之后再转发。

##### 数据发送详解

在主机发送端，从传输层开始会把上一层的数据加上一个报头形成本层的数据，这个过程称为数据封装。在主机接收端，从最下层开始，每一层数据会去掉报头信息，该过程称为数据解封。其过程如下图所示：



发送方数据封装处理：

1. 打开浏览器，输入网址 www.xxx.com，按回车键来访问网页，其实就是访问 Web 服务器上的网页，在应用层采用的协议是 HTTP，浏览器将网址等信息组成 HTTP 数据，并将数据传送给传输层。
2. 传输层在数据前面加上 TCP 报头，并标记端口为 80（Web 服务器的默认端口），将这个数据段给了网络层。
3. 网络层在这个数据段前面加上自己机器的 IP 和目的 IP，这时该段被称为 IP 数据报，然后将这个 IP 数据报给了数据链路层。
4. 数据链路层先在 IP 数据报前面加上自己机器的 MAC 地址以及目的 MAC 地址，加上 MAC 地址的数据称为帧，然后通过物理网卡把这个帧以比特流的方式发送到网络上。

互联网上有路由器，它会读取比特流中的 IP 地址进行路由操作，到达正确的网段后，这个网段的交换机读取比特流中的 MAC 地址，从而找到要接收的对应机器。

接收方数据解封处理：

1. 数据链路层用网卡接收到了比特流，读取比特流中的帧，将帧中的 MAC 地址去掉，就成了 IP 数据报，传递给网络层。
2. 网络层接收下层传来的 IP 数据报，将 IP 从包的前面拿掉，取出带有 TCP 的数据（数据段）交给传输层。
3. 传输层拿到了这个数据段，看到 TCP 标记的端口是 80，说明应用层协议是 HTTP，之后将 TCP 头去掉并将数据交给应用层，告诉应用层对方请求的是 HTTP 数据。
4. 应用层得知发送方请求的是 HTTP 数据，因此调用 Web 服务器程序把 www.xxx.com 的首页文件发送回去。

#### OSI模型和TCP/IP四层模型汇总



#### 抓包工具

Wireshark抓包软件，抓取网卡信息及网络上的通信数据；

#### MAC地址、IP地址、端口号、网络字节序

**MAC地址：**标记本地网卡的唯一的ID；

**IP地址：**标记此主机在网络上的位置；

**IPV4地址组成：**[].[].[].[]，（网络号+主机号），分A/B/C/D/E五类地址；

网络号：此主机所在的网络上的位置；

主机号：所在此网络下的某台主机；

**端口号：**进程**（用户进程、系统进程）**在本地计算机上的标识；128以下的端口号被预留；

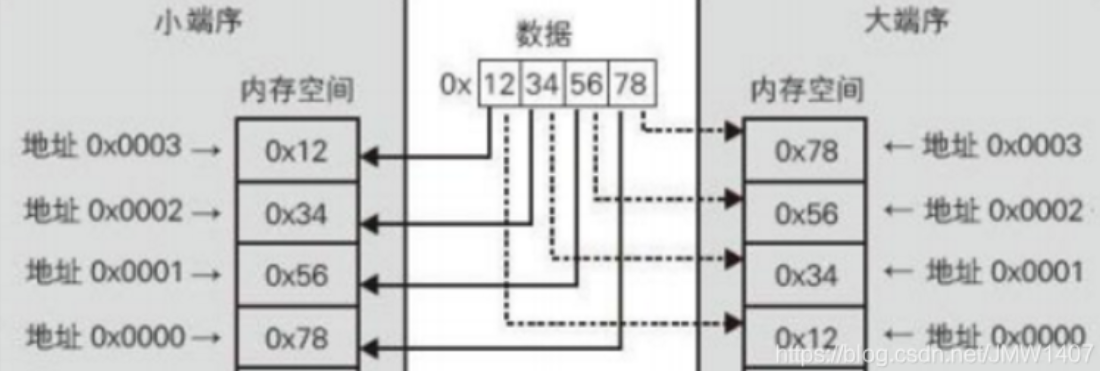
**网络字节序：**网络中统一规定的网络字节序列；

举例：我们看0x12345678数:从右（低）往左（高）看（低位78，高位56，再高位34，再高位12）（注意78、56、34、12各占一个字节，78地址0x0078F20，56地址...21，34地址...22，12地址...23）

**大小端字节序：**

**大端字节序（Big Endian）：**最高有效位存于最低内存地址处，最低有效位存于最高内存处；

**小端字节序（Little Endian）：**最高有效位存于最高内存地址处，最低有效位存于最低内存处。

****

**UDP/TCP/IP协议规定：**

**把接收到的第一个字节当作高位字节来看；这就要求发送端发送的第一个字节是高位字节;而在发送端发送数据时,发送的第一个字节是该数值在内存中的起始地址处对应的那个字节**,也就是说,该数值在内存中的起始地址处对应的那个字节就是要发送的第一个高位字节；所以网络字节序就是大端字节序；有些系统的本机字节序是小端字节序，有些则是大端字节序，为了保证传送顺序的一致性，所以**网际协议使用大端字节序来传送数据**；

**判断自己的本机是哪种字节序：**

/\* 确定你的电脑是大端字节序还是小端字节序 \*/

#include <stdio.h>

int check1()

{

int i = 1; //1在内存中的表示： 0x00000001

char \*pi = (char \*)&i; //将int型的地址强制转换为char型

return \*pi == 0; //如果读取到的第一个字节为1，则为小端法，为0，则为大端法

}

int main()

{

if (check1() == 1)

printf("big\n");

else

printf("little\n");

return 0;

}

**字节序转换函数：**

typedef uint32\_t in\_addr\_t;

struct in\_addr {

in\_addr\_t s\_addr;

};

//将主机字节序转换为网络字节序

unit32\_t htonl (unit32\_t hostlong);

unit16\_t htons (unit16\_t hostshort);

//将网络字节序转换为主机字节序

unit32\_t ntohl (unit32\_t netlong);

unit16\_t ntohs (unit16\_t netshort);

//IP地址转换成网络字节序:

in\_addr\_t inet\_addr(const char \*cp);

网络字节序转换成IP地址

char \*inet\_ntoa(struct in\_addr in);

#### UDP详解

**UDP优缺点**：面向无连接，不安全，不可靠的传输协议，但是效率高（**数据是基于数据包发送的，面向消息的**）；

**不可靠原因：**无法确定信息一定能收到，以及发送数据的顺序可能不一致；（**处理方法：**传输层UDP不可靠，但是应用层可以让它可靠；）

UDP使用场景：

1. 能容忍一些数据的丢失，但是不能允许有较大的时延（比如实时视频，直播等）;
2. 常用一次性传输比较少量数据的网络应用(DNS,SNMP);

**UDP报文：**

、

UDP首部有8个字节，由4个字段构成，每个字段都是两个字节；

* 源端口： 源端口号，需要对方回信时选用，不需要时全部置0；
* 目的端口：目的端口号，在终点交付报文的时候需要用到；
* 长度：UDP的数据包的长度（包括首部和数据），其最小值为8（只有首部）；
* 校验和：检测UDP数据报在传输中是否有错，有错则丢弃；
* 数据（payload）

**UDP通信流程：**

****

**第一步：创建套接字**

**API函数：**int socket(int domain, int type, int protocol);

**作用：**创建一个通信端点并返回一个网络标识符ID(描述符)；

**参数1：**domain 通信域，这将选择用于通信的协议族（常用AF地址族：IPv4通信协议[AF\_INET]，IPv6通信协议[AF\_INET6]）

**参数2：**type 通信方式（TCP还是UDP）;（常用:TCP [SOCK\_STREAM] ，UDP[SOCK\_DGRAM]，原始网络访问[SOCK\_RAW ] ）

**参数3:**protocol,置0即可(0：默认，IPPROTO\_TCP：使用tcp协议 ， IPPROTO\_UDP：使用UDP协议)；

**返回值：**成功（非负的文件描述符），失败（-1）

Ps:每个socket被创建后，都会分配两个缓冲区（输入缓冲区和输出缓冲区）；

**第二步：绑定套接字**（服务器端：必须绑定，客户端：可以不绑定）

**API函数：**int bind(int sockfd,const struct sockaddr \*addr,socklen\_t addrlen);

**作用：**将addr指定的IP地址及端口号分配给文件描述符（sockfd）引用的套接字；

**参数1：**sockfd 文件描述符

**参数2：**addr 指定IP地址和端口号

struct sockaddr {

sa\_family\_t sa\_family ; //typedef unsigned short int sa\_family\_t

char sa\_data[14]; }

为了统一，定义自己的结构体，然后强转成通用的结构体struct sockaddr；

IPv4通信定义struct sockaddr\_in 的结构体；

本地通信定义sockaddr\_un的结构体；

struct sockaddr\_in {

unsigned short sin\_family; //协议IPv4,2个字节

unsigned short sin\_port; //端口号 ，2个字节 struct in\_addr sin\_addr;

struct in\_addr{ \_\_be32 s\_addr;//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)];//8个字节

};

**参数3：**addrlen addr的长度

**返回值：**成功（>0），失败（-1）

**第三步：收/发数据**

**接受数据两个API接口：**

**API函数：**ssize\_t recvfrom(int sockfd, void \*buf, size\_t len, int flags, struct sockaddr \*src\_addr, socklen\_t \*addrlen);

**作用：接受数据**

**参数1：**sockfd 网络文件描述符ID；

**参数2**：buf 数据存放的缓冲区；

**参数3：**len 缓冲区大小；

**参数4：**flags 一般置为0；

**参数5：**src\_addr 指向发送数据的客户端地址信息的结构体（包括有发送端的IP/端口号）；

**参数6：**addrlen 为src\_addr的长度；

**返回值：**成功（实际接收到的字符数），失败（-1）

**API函数：**ssize\_t recv (int sockfd, void \*buf, size\_t len, int flags);

**作用：接受数据**

**参数1：**sockfd 网络文件描述符ID；

**参数2**：buf 数据存放的缓冲区；

**参数3：**len 缓冲区大小；

**参数4：**flags 一般置为0；

**返回值：**成功（实际接收到的字符数），失败（-1）

**发送数据两个API接口：**

**API函数：**ssize\_t sendto(int sockfd, const void \*buf, size\_t len, int flags, const struct sockaddr \*dest\_addr, socklen\_t addrlen);

**作用：发送数据**

**参数1：**sockfd 网络文件描述符ID；

**参数2**：buf数据发送的缓冲区；

**参数3：**len 缓冲区的长度；

**参数4：**flags一般置为0；

**参数5：**dest\_addr 目标地址，数据发送到的位置；

**参数6：**addrlen dest\_addr长度；

**返回值：**成功（返回发送的字符数），失败（-1）

**API函数：**ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);

**作用：发送数据**

**参数1：**sockfd 网络文件描述符ID；

**参数2**：buf数据发送的缓冲区；

**参数3：**len 缓冲区的长度；

**参数4：**flags一般置为0；

**返回值：**成功（返回发送的字符数），失败（-1）

**第四步：关闭网络文件描述符**

**API函数：**int close(int fd);

**作用：**关闭网络文件描述符；

**参数1：**fd 网络文件描述符；

**返回值：**成功（0），失败（-1）；

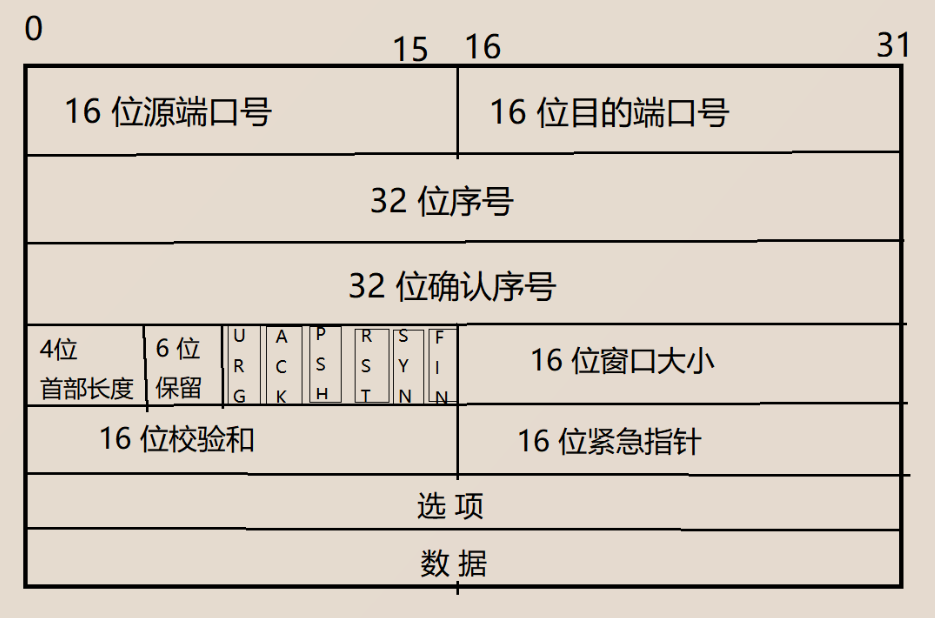
#### TCP详解

https://blog.csdn.net/u012484779/article/details/120302984

**TCP优缺点：**面向连接，安全，可靠的、顺序的传输协议，效率低（数据是基于流传输）；

**可靠原因：**建立连接（三次握手），关闭连接（四次挥手）；

**TCP报文：**



* 源端口号/目的端口号：表示数据从哪个进程来，从哪个进程去；
* 32位序列号；
* 4位首部长度：表示该tcp报头有多少个4字节；
* 6位保留：保留待用，以防万一；
* 6位标志位：
  + 1. URG:标识紧急指针是否有效；
    2. ACK:标识确认序号是否有效；
    3. PSH:用来提示接收端应用程序立刻将数据从tcp缓冲区读走；
    4. RST：要求重新建立连接，我们把含有RST标识的报文称为\*\*复位报文段\*\*；
    5. SYN：请求建立连接。把含有SYN标识的报文称为\*\*同步报文段\*\*；
    6. FIN:通知对端，本端即将关闭。把含有FIN标识的报文称为\*\*结束报文段\*\*；
* 16位窗口大小;
* 16位检验和：由发送端填充，检验形式有CRC校验等。如果接受端校验不通过，则认为数据有问题。此处的校验和不光包含TCP首部，也包含TCP数据部分。
* 16位紧急指针：用来标识哪部分数据是紧急数据；
* 选项和数据:暂时忽略

**TCP通信流程：**



**第一步：创建套接字**

见“UDP通信流程”

**第二步：绑定套接字**

见“UDP通信流程”

**第三步：监听（服务器端，客户端没有）、**

**API函数：**int listen(int sockfd, int backlog);

**作用：**监听客户端连接请求；

**参数1：**sockfd 网络文件描述符；

**参数2：**backlog 指定监听sockfd挂起连接队列的大小；

**返回值：**成功（0），失败（-1）；

**第四步：等待连接/主动连接（服务器端:等待连接，客户端：主动连接）**

**API函数：**int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

**作用：**等待客户端连接；

参数1：sockfd 网络文件描述符ID

参数2：addr 用来保存客户端IP及端口号；（不需要可以设置成NULL）

参数3：addrlen addr的大小；（不需要可以设置成NULL）

返回值：成功（**返回用于收发信息的“新的通信描述符”**），失败（-1）；

**API函数：**int connect(int sockfd, const struct sockaddr \*addr , socklen\_t addrlen);

**作用：**连接服务器端（主动连接）

**参数1：**sockfd 网络文件描述符ID；

**参数2：**addr 设置服务器段IP及端口号等信息；

**参数3：**addrlen addr内存空间大小；

**返回值：**成功（0），失败（-1）

**第五步：收发信息**

见“UDP通信流程”；

**第六步：关闭**

见“UDP通信流程”；

**TCP常见问题：粘包（即多个数据包连接在一起）**

**粘包原因：**

TCP 使用Nagle算法（send() / sendTo()）：将多次间隔较小且数据量小的数据存入缓冲区直到积攒到一定数量，进行封包后一起发送，即TCP面向流的通信是**无消息保护边界**的；（UDP接收端采用链式结构记录每一个UDP包，每一个包就是一个消息，UDP包的消息头就有消息头（数据长度等信息），即UDP是面向消息的有消息保护边界的）；

**解决粘包：**

1.底层：关闭Nagle算法（不可行）；

2.应用层：

协议处理

（

1.每个消息加长度，把消息尺寸与消息一起发送，使用特殊标记来区分消息间隔；

2.或者分两次发送，先发消息长度，在发消息数据；

）

#### setsockopt()/getsockopt()函数

**API函数:** int setsockopt(int sockfd , int level , int optname , const void \*optval , socklen\_t optlen);

**作用：**设置套接字描述符的属性；

**参数1：**sockfd 网络文件描述符；

**参数2：** level 设置网络层次;（ SOL\_SOCKET（通用套接字层次,常用） ， IPPROTO\_TCP （TCP层次）， IPPROTO\_IP（IP层次） ， IPPROTO\_IPV6（IPv6） ）

**参数3：**optname 选项名（一个Level对应多个选项,不同选项对应不同功能）

**参数4：**optval（选项值） 指向存放新选项（optname）的缓冲区；根据选项名称的数据类型进行转换；（可以是一个结构体，也可以是普通变量）；

**参数5：**optlen（选项长度） optval缓冲区的空间大小；

**返回值：**成功（0），失败（-1）；

optname 选项名：

|  |  |  |
| --- | --- | --- |
| SOL\_SOCKET | | |
| 选项名称 | 说明 | 数据类型 |
| SO\_BROADCAST | 允许发送广播数据 | int |
| SO\_DEBUG | 允许调试 | int |
| SO\_DONTROUTE | 不查找**路由** | int |
| SO\_ERROR | 获得套接字错误 | int |
| SO\_KEEPALIVE | 保持连接 | int |
| SO\_LINGER | 延迟关闭连接 | struct linger |
| SO\_OOBINLINE | 带外数据放入正常数据流 | int |
| SO\_RCVBUF | 接收缓冲区大小 | int |
| SO\_SNDBUF | 发送缓冲区大小 | int |
| SO\_RCVLOWAT | 接收缓冲区下限 | int |
| SO\_SNDLOWAT | 发送缓冲区下限 | int |
| SO\_RCVTIMEO | 接收超时 | struct timeval |
| SO\_SNDTIMEO | 发送超时 | struct timeval |
| SO\_REUSERADDR | 允许重用本地地址和端口 | int |
| SO\_TYPE | 获得套接字类型 | int |
| SO\_BSDCOMPAT | 与BSD系统兼容 | int |

|  |  |  |
| --- | --- | --- |
| IPPROTO\_IP | | |
| 选项名称 | 说明 | 数据类型 |
| IP\_HDRINCL | 在数据包中包含IP首部 | int |
| IP\_OPTINOS | IP首部选项 | int |
| IP\_TOS | 服务类型 |  |
| IP\_TTL | 生存时间 | int |

|  |  |  |
| --- | --- | --- |
| IPPRO\_TCP | | |
| 选项名称 | 说明 | 数据类型 |
| TCP\_MAXSEG | TCP最大数据段的大小 | int |
| TCP\_NODELAY | 不使用Nagle算法 | int |

**API函数:** int getsockopt(int sockfd , int level , int optname , void \*optval , socklen\_t \*optlen);

**作用：**设置套接字描述符的属性；

**参数1：**sockfd 网络文件描述符；

**参数2：** level 设置网络层次；

**参数3：**optname 选项名

**参数4：**optval（选项值）获取到的选项（optname）的值；

**参数5：**optlen（选项长度）optval的空间大小；

**返回值：**成功（0），失败（-1）；

#### Socket套接字及缓冲区详解

<https://blog.csdn.net/daaikuaichuan/article/details/83061726>

**使用write()/send()发送数据**

【阻塞模式下】：

1. 首先会检查缓冲区，如果缓冲区的可用空间长度小于要发送的数据，那么 write()/send() 会被阻塞（暂停执行），直到缓冲区中的数据被发送到目标机器，腾出足够的空间，才唤醒 write()/send() 函数继续写入数据；
2. 如果TCP协议正在向网络发送数据，那么输出缓冲区会被锁定，不允许写入，write()/send() 也会被阻塞，直到数据发送完毕缓冲区解锁，write()/send() 才会被唤醒。如果TCP协议正在向网络发送数据，那么输出缓冲区会被锁定，不允许写入，write()/send() 也会被阻塞，直到数据发送完毕缓冲区解锁，write()/send() 才会被唤醒；
3. 如果要写入的数据大于缓冲区的最大长度，那么将分批写入。如果要写入的数据大于缓冲区的最大长度，那么将分批写入；直到所有数据被写入缓冲区 write()/send() 才能返回。
4. send()函数默认情况下会使用Nagle算法。Nagle算法通过将未确认的数据存入缓冲区直到积攒到一定数量一起发送的方法，来降低主机发送零碎小数据包的数目。所以假设send()函数发送数据过快的话，该算法会将一些数据打包后统一发出去。通过setsockopt()的TCP\_NODELAY选项来禁用Nagle算法。

【非阻塞模式下】：

send()函数的过程仅仅是将数据拷贝到输出缓冲区而已，如果缓冲区可用空间不够，则尽可能拷贝，然会立即返回成功拷贝的大小；如果缓存区可用空间为0，则返回-1，同时设置errno为EAGAIN。

**使用read()/recv()读取数据**

【阻塞模式下】：

1. 首先会检查缓冲区，如果缓冲区中有数据，那么就读取，否则函数会被阻塞，直到网络上有数据到来；
2. 如果要读取的数据长度小于缓冲区中的数据长度，那么就不能一次性将缓冲区中的所有数据读出，剩余数据将不断积压，直到读取到数据后 read()/recv() 函数才会返回，否则就一直被阻塞。

【非阻塞模式下】：

接收数据时perror时常遇到“Resource temporarilyunavailable”的提示，errno代码为11(EAGAIN)。这表明你在非阻塞模式下调用了阻塞操作，在该操作没有完成就返回这个错误，这个错误不会破坏socket的同步，继续循环接着recv就可以。

**使用accept等待连接**

【阻塞模式下】：

如果没有客户端套接字去请求，它便会在那里一直痴痴地等下去，直到永远

【非阻塞模式下】：

那么accept函数就会立即返回；

# 线程

<https://blog.csdn.net/m0_50534425/article/details/121358172>

https://zhuanlan.zhihu.com/p/158965214

### 【含义】

线程也称为轻量级进程，是进程执行流的最小单元，CPU调度执行的基本单位，**线程共享进程资源**；

### 【主线程和次线程】

主线程：main函数的执行流序列；

次线程：除主线程外的其他执行序列；

PS：线程缺点：当一个线程崩溃时，会导致其所属进程的所有线程崩溃（**用户级线程**）；

### 【线程带来的变化】

* 引入线程后，进程只作为分配程序运行的资源的单位，线程作为CPU调度的基本单位；
* 进程内线程也可并发，进一步提高了系统的并发度；
* 传统进程间并发，需要切换进程的运行环境，系统开销大，线程不需要切换进程环境，减少开销；

### 【线程属性】

* 线程是CPU处理器调度的单位；
* 多CPU计算机中，各个线程可占用不同的CPU；
* 每个线程都有一个线程ID、线程控制块（TCB：进程的私有资源）；
* 线程也有就绪、阻塞、运行三种基本状态，同样也有状态切换一说；
* 线程几乎不拥有系统资源；
* 同一进程的不同线程间共享进程的资源；
* 由于共享进程内存地址空间，同一进程中线程间通信甚至无需系统干预；
* 同一进程的线程切换，不会引起进程切换（切换线程开销小）；
* 不同进程的线程切换，会引起进程切换（切换进程开销大）；

### 【线程私有资源与线程的共享资源】

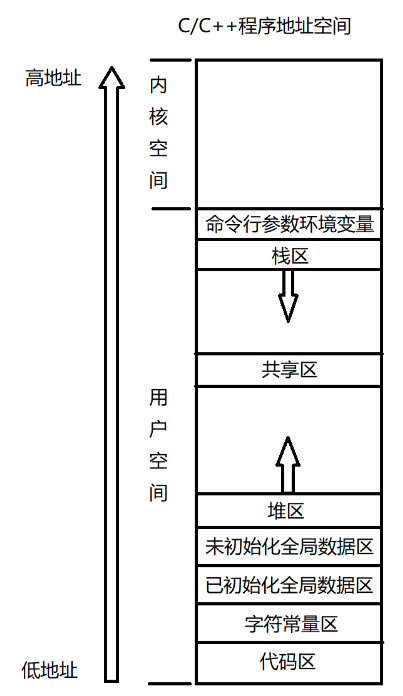
<https://baijiahao.baidu.com/s?id=1687308494061329777&wfr=spider&for=pc、>

**线程私有资源：**

* 线程运行的本质就是函数运行；运行时的信息是保存在栈中；每个线程都有自己的独立的栈区；
* **线程的上下文（**Thread Context**）**就是线程私有资源（1.线程的栈区（并非完全不能被其他线程读取）、2.线程局部存储、3.程序计数器、4.栈指针、5.函数运行时的寄存器等）的一个统称；
* 操作系统可以调度线程，就是依靠线程的上下文信息；

**线程共享资源：**

除了私有资源外，剩下的都是线程间的共享资源（堆区、代码区、数据区、字符常量区等）；

****

这其实就是进程地址空间的样子，也就是说线程共享“进程地址空间中除去线程上下文信息”中的所有内容，意思就是线程可以直接读取这些内容；

### 【线程上下文切换】

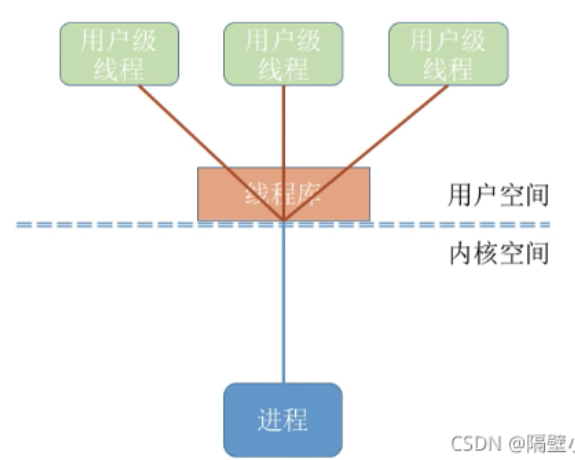
线程上下文切换是什么？

这还得从线程是否属于同一个进程分析：

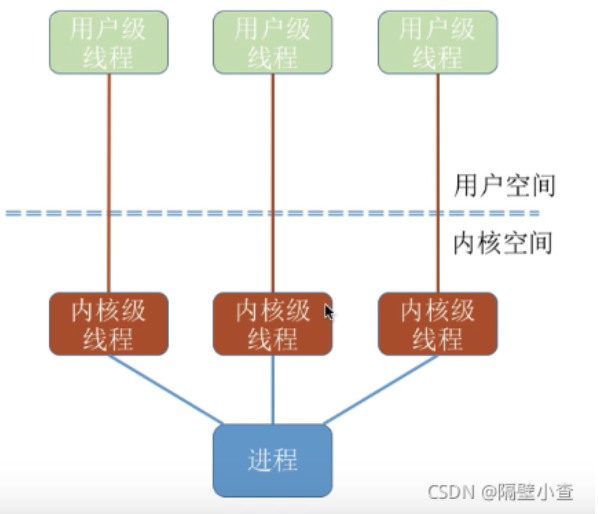
* + - 两个线程不属于同一个进程，则切换过程就跟进程上下文切换一样；
    - 两个线程属于同一个进程，因为虚拟内存地址是共享的，所以切换时，虚拟内存等这些资源就保持不动，只需要切换线程的私有数据、寄存器等不共享的数据；（所以线程的上下文切换要比进程开销小很多；）

### 【线程的实现方式】

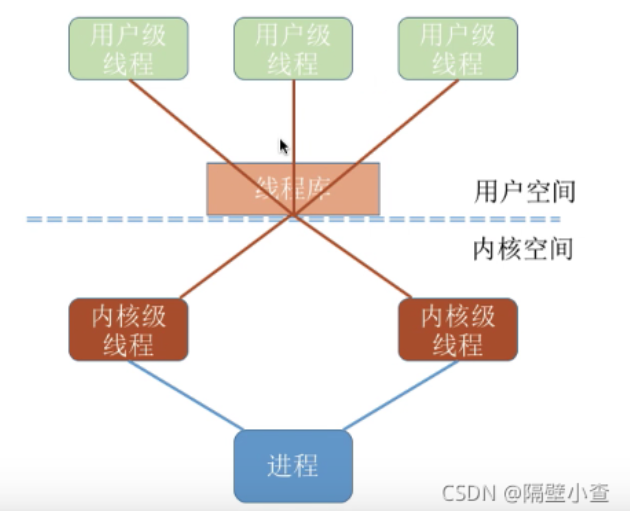
**用户级线程：**由应用程序通过线程库实现，线程切换可以在用户态下完成，**是由用户态的线程库来完成线程管理**；在用户看来是多个线程，但是在操作系统内核看来，意识不到线程的存在（即TCB在线程管理库里实现的，操作系统看不到TCB只能看到PCB）;



**内核级线程：**线程管理工作由操作系统内核来完成。线程的调度、切换由内核负责，线程切换在核心态下完成;



**在同时支持两种线程的系统中，可采用二者结合的方式（组合级线程也称为轻量级进程）：**将n个用户级线程对应到m个内核级线程上（n>=m）。内核级线程才是处理器分配的单位；



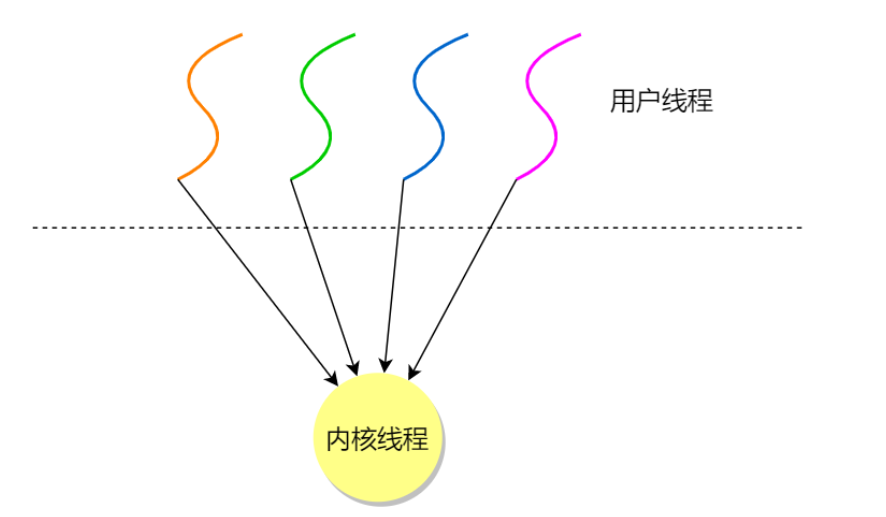
### 【线程模型】

* **多对一模型**

多个用户级线程对应一个内核级线程。每个用户进程只对应一个内核级线程

优点：用户级线程切换在用户空间即可完成，不需要切换到核心态，线程管理的系统开销小，效率高；

缺点：当一个线程被阻塞时，整个进程都会被阻塞；并发能力弱；多核CPU是没办法充分利用CPU的；

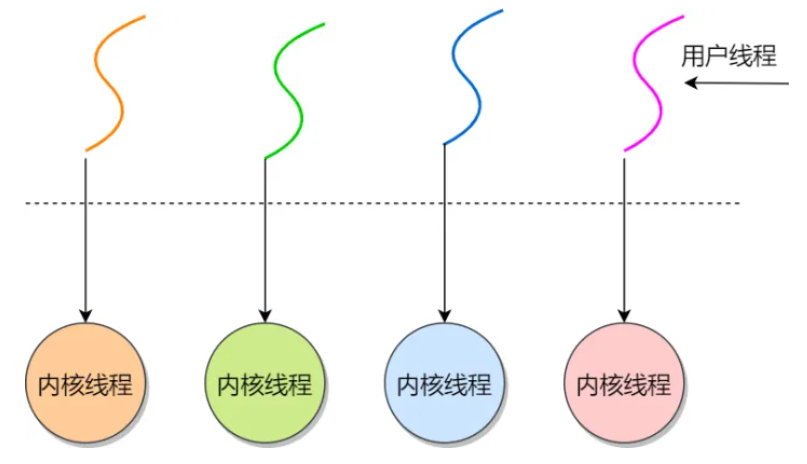


* **一对一模型**

一个用户级线程对应到一个内核级线程。每个用户进程有用户级线程同数量的内核级线程

优点：当一个线程被阻塞后，别的线程还可以继续执行，并发能力强。多线程可以在多核多线程处理器上并发执行。

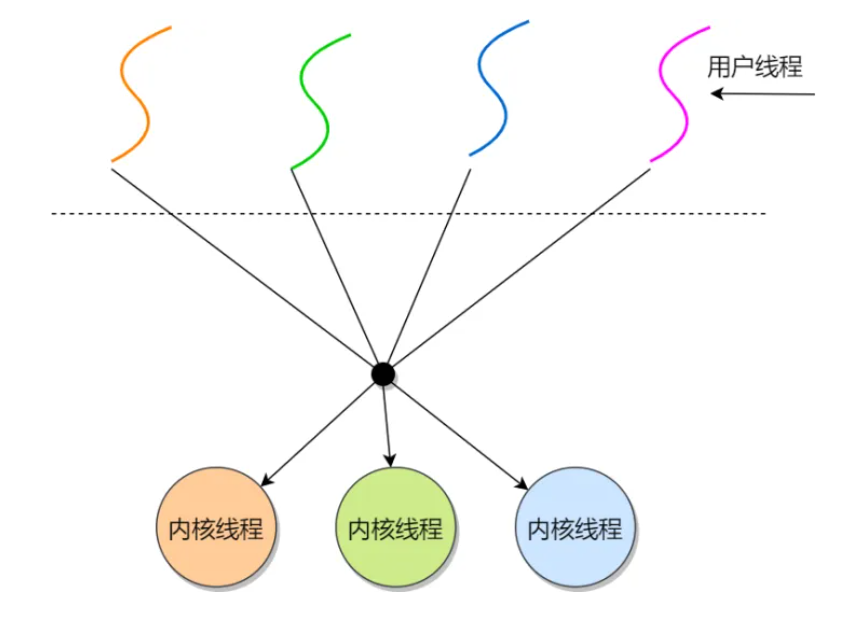
缺点：一个用户进程会占用多个内核级线程，线程切换由操作系统内核完成，需要切换到内核态，因此线程管理的成本高，开销大。



* **多对多模型**

n个用户级线程对应到m个内核级线程上(n>=m)，每个用户进程对应m个内核级线程；

多对多模型克服了多对一模型并发度不高的缺点，又克服了一对一模型一个用户占有太多内核级线程，开销太大的缺点



### 【线程实现方式与线程模型的总结】

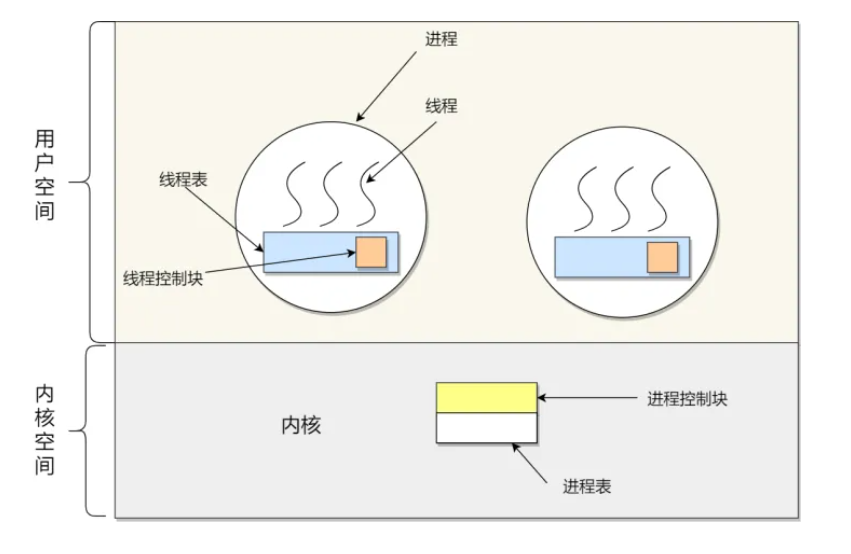
https://www.zhihu.com/question/35128513

**【用户级线程】（**多对一模型**）：**

用户线程是基于用户态的线程管理库来实现的，那么线程控制块（Thread Control Block, TCB） 也是在库里面来实现的，对于操作系统而言是看不到这个 TCB 的，它只能看到整个进程的 PCB。

所以，**用户线程的整个线程管理和调度，操作系统是不直接参与的，而是由用户级线程库函数来完成线程的管理，包括线程的创建、终止、同步和调度等；（用户级线程库管理TCB，操作系统只管理PCB）**

用户级线程的模型，即多个用户线程对应同一个内核线程，如下图所示：



**用户级线程优点：**

* 每个进程都需要有它私有的线程控制块（TCB）列表，用来跟踪记录它各个线程状态信息（PC、栈指针、寄存器），TCB 由用户级线程库函数来维护，可用于不支持线程技术的操作系统；
* 用户线程的切换也是由线程库函数来完成的，无需用户态与内核态的切换，所以速度特别快；

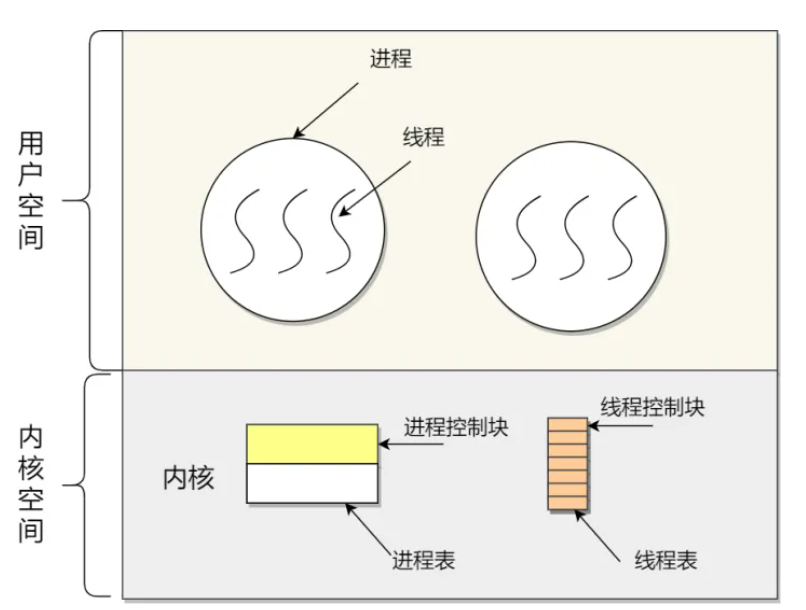
**用户级线程缺点：**

* 由于操作系统不参与线程的调度，如果一个线程发起了系统调用（比如I/O）而阻塞，那进程所包含的用户线程都不能执行了。
* 当一个线程开始运行后，除非它主动地交出 CPU 的使用权，否则它所在的进程当中的其他线程无法运行，因为用户态的线程是没法打断当前运行中的线程，它没有这个特权，只有操作系统才有，但是用户线程不是由操作系统管理的。
* 由于时间片分配给进程，故与其他进程比，在多线程执行时，每个线程得到的时间片较少，执行会比较慢；

**【内核级线程】（**一对一模型**）：**

内核线程是由操作系统管理的，线程对应的 TCB 自然是放在操作系统里的，这样线程的创建、终止和管理都是由操作系统负责。**（操作系统管理TCB、PCB）**

内核线程模型，即一个用户线程对应一个内核线程;



**内核线程的优点：**

* 在一个进程当中，如果某个内核线程发起系统调用（比如I/O）而被阻塞，并不会影响其他内核线程的运行；
* 分配给线程，多线程的进程获得更多的 CPU 运行时间；

**内核线程的缺点：**

* 在支持内核线程的操作系统中，由内核来维护进程和线程的上下文信息，如 PCB 和 TCB；
* 线程的创建、终止和切换都是通过系统调用的方式来进行，因此对于系统来说，系统开销比较大；

【**轻量级进程（Light-weight process , LWP）**】

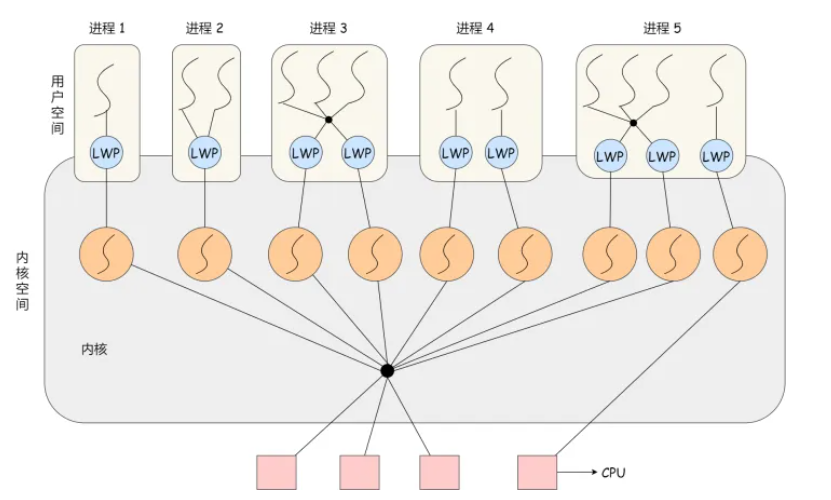
**轻量级进程**是内核中支持的用户线程，一个进程可有一个或多个LWP，每个LWP是跟内核线程一对一映射的；也就是LWP都是由一个内核线程支持；

在大多数系统中，LWP与普通进程的区别：在于它只有一个最小的执行上下文和调度程序所需的统计信息。一般来说，一个进程代表程序的一个实例，而 LWP 代表程序的执行线程，因为一个执行线程不像进程那样需要那么多状态信息，所以 LWP 也不带有这样的信息；

在 LWP 之上也是可以使用用户线程的，那么LWP 与用户线程的对应关系就有三种：

* 1: 1，即一个 LWP 对应 一个用户线程；
* N : 1，即一个 LWP 对应多个用户线程；
* N : N，即多个 LWP 对应多个用户线程；

接下来针对上面这三种对应关系说明它们优缺点。先下图的 LWP 模型：



**1:1 模式（内核级线程的本质）**

一个线程对应到一个 LWP 再对应到一个内核线程，如上图的进程 4，属于此模型。

* 优点：实现并行，当一个 LWP 阻塞，不会影响其他 LWP；
* 缺点：每一个用户线程，就产生一个内核线程，创建线程的开销较大。

**N : 1 模式（用户级线程的本质）**

多个用户线程对应一个 LWP 再对应一个内核线程，如上图的进程 2，线程管理是在用户空间完成的，此模式中用户的线程对操作系统不可见（TCB :操作系统不可见）。

* 优点：用户线程要开几个都没问题，且上下文切换发生用户空间，切换的效率较高；
* 缺点：一个用户线程如果阻塞了，则整个进程都将会阻塞，另外在多核 CPU 中，是没办法充分利用 CPU 的。

**M : N 模式**

根据前面的两个模型混搭一起，就形成 M:N 模型，该模型提供了两级（用户级、内核级）控制，首先多个用户线程对应到多个 LWP，LWP 再一一对应到内核线程，如上图的进程 3。

优点：综合了前两种优点，大部分的线程上下文发生在用户空间，且多个线程又可以充分利用多核 CPU 的资源。

**组合模式**

上图的进程 5，此进程结合 1:1 模型和 M:N 模型。开发人员可以针对不同的应用特点调节内核线程的数目来达到物理并行性和逻辑并行性的最佳方案。

# 线程同步

# IO多路复用

# 重定向