### 1.内部碎片:

内部碎片的产生：因为所有的内存分配必须起始于可被 4、8 或 16 整除（视处理器体系结构而定）的地址或者因为MMU的分页机制的限制，决定内存分配算法仅能把预定大小的内存块分配给客户。假设当某个客户请求一个 43 字节的内存块时，因为没有适合大小的内存，所以它可能会获得 44字节、48字节等稍大一点的字节，因此由所需大小四舍五入而产生的多余空间就叫内部碎片。  
　　外部碎片的产生： 频繁的分配与回收物理页面会导致大量的、连续且小的页面块夹杂在已分配的页面中间，就会产生外部碎片。假设有一块一共有100个单位的连续空闲内存空间，范围是0~99。如果你从中申请一块内存，如10个单位，那么申请出来的内存块就为0~9区间。这时候你继续申请一块内存，比如说5个单位大，第二块得到的内存块就应该为10~14区间。如果你把第一块内存块释放，然后再申请一块大于10个单位的内存块，比如说20个单位。因为刚被释放的内存块不能满足新的请求，所以只能从15开始分配出20个单位的内存块。现在整个内存空间的状态是0~9空闲，10~14被占用，15~34被占用，25~99空闲。其中0~9就是一个内存碎片了。如果10~14一直被占用，而以后申请的空间都大于10个单位，那么0~9就永远用不上了，变成外部碎片。

### 虚拟内存：

内存是直接与CPU交互的，一般都需要将进程的所有信息都加载到内存中。当每个进程创建的时候，内核会为进程分配4G的虚拟内存，当进程还没有开始运行时，这只是一个内存布局。实际上并不立即就把虚拟内存对应位置的程序数据和代码（比如.text .data段）拷贝到物理内存中，只是建立好虚拟内存和磁盘文件之间的映射就好（就是能通过虚拟地址找到实际磁盘文件）。这个时候数据和代码还是在磁盘上的。当CPU运行到对应的程序时利用虚拟地址去寻找页表（每个进程都有一个页表MMU来建立页表）（以虚拟地址为索引，当被查找页在内存时返回内存物理地址，否则去磁盘调用近物理内存），发现页表中地址没有存放在物理内存上，而是在磁盘上，于是发生缺页异常，于是将磁盘上的数据拷贝到物理内存中。

优点：每当程序使用时才调入进内存，提高了效率

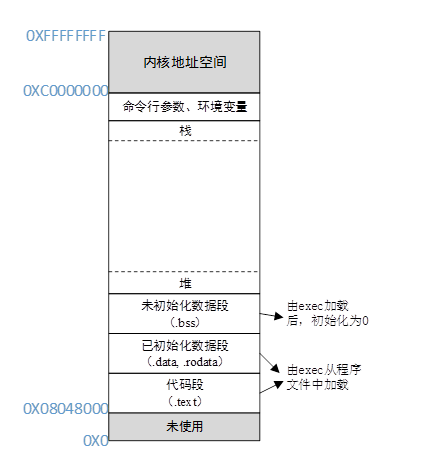
使系统能够运行比物理内存大的程序

能够运行多个进程，不用关心内存的分配。

### 计算机编码：

**补码：**负数的补码是在原码的基础上除符号位外其余位取反后+1。

### 进程空间分布及作用



**每个进程自认为的地址空间**

**内核空间：**

**范围：**Linux 操作系统而言，最高的 1G 字节(从虚拟地址 0xC0000000 到 0xFFFFFFFF)

**Linux 操作系统和驱动程序运行在内核空间**

内容：

本质：存放的是操作系统核心部分。

所有的系统资源管理都是在内核空间中完成的。比如读写磁盘文件，分配回收内存，从网络接口读写数据等等，内核是一个操作系统的核心，是操作系统最基本的部分，一个内核**不是一套完整的操作系统**所以应该存放的是**操作系统核心部分。**

**内核的作用**

* ****进程管理****：内核负责创建和销毁进程, 并处理它们与外部世界的联系(输入和输出)，不同进程间通讯(通过信号，管道，或者进程间通讯原语)对整个系统功能来说是基本的，也由内核处理。 另外， 调度器， 控制进程如何共享CPU，是进程管理的一部分。更通常地，内核的进程管理活动实现了多个进程在一个单个或者几个CPU 之上的抽象。
* ****内存管理****：计算机的内存是主要的资源， 处理它所用的策略对系统性能是至关重要的。内核为所有进程的每一个都在有限的可用资源上建立了一个虚拟地址空间。内核的不同部分与内存管理子系统通过一套函数调用交互，从简单的malloc/free对到更多更复杂的功能。
* ****文件管理****：Linux 在很大程度上基于文件系统的概念;几乎Linux中的任何东西都可看作一个文件。内核在非结构化的硬件之上建立了一个结构化的文件系统，结果是文件的抽象非常多地在整个系统中应用。另外，Linux 支持多个文件系统类型，就是说，物理介质上不同的数据组织方式。例如，磁盘可被格式化成标准Linux的ext3文件系统，普遍使用的FAT文件系统，或者其他几个文件系统。
* ****驱动管理****：几乎每个系统操作终都映射到一个物理设备上，除了处理器，内存和非常少的别的实体之外，全部中的任何设备控制操作都由特定于要寻址的设备相关的代码来进行。这些代码称为设备驱动。内核中必须嵌入系统中出现的每个外设的驱动，从硬盘驱动到键盘和磁带驱动器。
* ****网络管理****：网络必须由操作系统来管理，因为大部分网络操作不是特定于某一个进程： 进入系统的报文是异步事件。报文在某一个进程接手之前必须被收集，识别，分发，系统负责在程序和网络接口之间递送数据报文，它必须根据程序的网络活动来控制程序的执行。另外，所有的路由和地址解析问题都在内核中实现。

### 5.**死锁概念及产生原理**

****概念：**** 多个并发进程因争夺系统资源而产生相互等待的现象。

****原理：**** 当一组进程中的每个进程都在等待某个事件发生，而只有这组进程中的其他进程才能触发该事件，这就称这组进程发生了死锁。

****本质原因：****

        1）、系统资源有限。

        2）、进程推进顺序不合理。

****死锁产生的4个必要条件****

****1、互斥：**** 某种资源一次只允许一个进程访问，即该资源一旦分配给某个进程，其他进程就不能再访问，直到该进程访问结束。

****2、占有且等待：**** 一个进程本身占有资源（一种或多种），同时还有资源未得到满足，正在等待其他进程释放该资源。

****3、不可抢占：**** 别人已经占有了某项资源，你不能因为自己也需要该资源，就去把别人的资源抢过来。

****4、循环等待：**** 存在一个进程链，使得每个进程都占有下一个进程所需的至少一种资源。

        当以上四个条件均满足，必然会造成死锁，发生死锁的进程无法进行下去，它们所持有的资源也无法释放。这样会导致CPU的吞吐量下降。所以死锁情况是会浪费系统资源和影响计算机的使用性能的。那么，解决死锁问题就是相当有必要的了。

****避免死锁的方法****

****1、死锁预防 ----- 确保系统永远不会进入死锁状态****

     产生死锁需要四个条件，那么，只要这四个条件中至少有一个条件得不到满足，就不可能发生死锁了。由于互斥条件是非共享资源所必须的，不仅不能改变，还应加以保证，所以，主要是破坏产生死锁的其他三个条件。

****a、****破坏“占有且等待”条件

     方法1：所有的进程在开始运行之前，必须一次性地申请其在整个运行过程中所需要的全部资源。

         优点：简单易实施且安全。

         缺点：因为某项资源不满足，进程无法启动，而其他已经满足了的资源也不会得到利用，严重降低了资源的利用率，造成资源浪费。

                  使进程经常发生饥饿现象。

     方法2：该方法是对第一种方法的改进，允许进程只获得运行初期需要的资源，便开始运行，在运行过程中逐步释放掉分配到的已经使用完毕的资源，然后再去请求新的资源。这样的话，资源的利用率会得到提高，也会减少进程的饥饿问题。

****b、****破坏“不可抢占”条件

      当一个已经持有了一些资源的进程在提出新的资源请求没有得到满足时，它必须释放已经保持的所有资源，待以后需要使用的时候再重新申请。这就意味着进程已占有的资源会被短暂地释放或者说是被抢占了。

      该种方法实现起来比较复杂，且代价也比较大。释放已经保持的资源很有可能会导致进程之前的工作实效等，反复的申请和释放资源会导致进程的执行被无限的推迟，这不仅会延长进程的周转周期，还会影响系统的吞吐量。

****c、****破坏“循环等待”条件

     可以通过定义资源类型的线性顺序来预防，可将每个资源编号，当一个进程占有编号为i的资源时，那么它下一次申请资源只能申请编号大于i的资源。如图所示：

### 6.进程间通信方式

**原因**：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**

#### 1.管道

**本质：通过**内核缓冲区**来进行通信**

**特点**：

它是**半双工**的（即数据只能在一个方向上流动），具有固定的读端和写端。

它只能用于具有**亲缘关系**的进程之间的通信（也是父子进程或者兄弟进程之间）。

它可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write 等函数。它不是普通的文件，并不属于其他任何文件系统，并且**只存在于内存**中。

**原理：**

管道就是调用pipe函数时在**内核**中开辟一块**缓冲区**，然后返回两个**文件描述符**来表示对该缓存区的**读端**和**写端**

**过程：**

1. 父进程调用pipe开辟管道，得到**两个**文件描述符指向管道的两端。  
   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（也可以**反过来进行父读子写**）。父进程可以往管道里写，子进程可以从管道里读，管道是用环形队列实现的，数据从写端流入从读端流出，这样就实现了进程间通信。

#### 2.命名管道（FIFO）

**本质：**

命名管道作为一种**特殊的文件**存放在**文件系统**中，命名管道是一个设备文件，因此，即使进程与创建FIFO的进程不存在亲缘关系，只要可以访问该路径，就能够通过FIFO相互通信。

**特点：**

按照**先进先出**的原则工作，第一个被写入的数据将首先从管道中读出。

**有最大的传送数据大小，保证操作的原子性**

#### 3.消息队列

**本质：**

**创建一个有唯一ID号的消息队列，使用相应函数向该队列写入自定义类型数据，接受也指定相应的ID号来进行相应类型数据的接收。**

**特点**：

个数据块都被认为**含有一个类型（由消息的第一个整数字段值来体现）**，接收进程可以独立地接收含有不同类型的数据结构。

**传送数据最大为8192，**也就是2的16次方。

**过程**：

1、msgget函数

该函数用来创建和访问一个消息队列。它的原型为：

int msgget(key\_t, key, int msgflg);

与其他的IPC机制一样，程序必须提供一个键来命名某个特定的消息队列。msgflg是一个权限标志，表示消息队列的访问权限，它与文件的访问权限一样。msgflg可以与IPC\_CREAT做或操作，表示当key所命名的消息队列不存在时创建一个消息队列，如果key所命名的消息队列存在时，IPC\_CREAT标志会被忽略，而只返回一个标识符。

它返回一个以key命名的消息队列的标识符（非零整数），失败时返回-1.

2、msgsnd函数

该函数用来把消息添加到消息队列中。它的原型为：

int msgsend(int msgid, const void \*msg\_ptr, size\_t msg\_sz, int msgflg);

msgid是由msgget函数返回的消息队列标识符。

msg\_ptr是一个指向准备发送消息的指针，但是消息的数据结构却有一定的要求，指针msg\_ptr所指向的消息结构一定要是以一个长整型成员变量开始的结构体，接收函数将用这个成员来确定消息的类型。所以消息结构要定义成这样：

struct my\_message{

long int message\_type;

*/\* The data you wish to transfer\*/*

};

msg\_sz是msg\_ptr指向的消息的长度，注意是消息的长度，而不是整个结构体的长度，也就是说msg\_sz是不包括长整型消息类型成员变量的长度。

msgflg用于控制当前消息队列满或队列消息到达系统范围的限制时将要发生的事情。

如果调用成功，消息数据的一分副本将被放到消息队列中，并返回0，失败时返回-1.

3、msgrcv函数

该函数用来从一个消息队列获取消息，它的原型为

int msgrcv(int msgid, void \*msg\_ptr, size\_t msg\_st, long int msgtype, int msgflg);

msgid, msg\_ptr, msg\_st的作用也函数msgsnd函数的一样。

msgtype可以实现一种简单的接收优先级。如果msgtype为0，就获取队列中的第一个消息。如果它的值大于零，将获取具有相同消息类型的第一个信息。如果它小于零，就获取类型等于或小于msgtype的绝对值的第一个消息。

msgflg用于控制当队列中没有相应类型的消息可以接收时将发生的事情。

调用成功时，该函数返回放到接收缓存区中的字节数，消息被复制到由msg\_ptr指向的用户分配的缓存区中，然后删除消息队列中的对应消息。失败时返回-1.

4、msgctl函数

该函数用来控制消息队列，它与共享内存的shmctl函数相似，它的原型为：

int msgctl(int msgid, int command, struct msgid\_ds \*buf);

command是将要采取的动作，它可以取3个值，

    IPC\_STAT：把msgid\_ds结构中的数据设置为消息队列的当前关联值，即用消息队列的当前关联值覆盖msgid\_ds的值。

    IPC\_SET：如果进程有足够的权限，就把消息列队的当前关联值设置为msgid\_ds结构中给出的值

    IPC\_RMID：删除消息队列

buf是指向msgid\_ds结构的指针，它指向消息队列模式和访问权限的结构。msgid\_ds结构至少包括以下成员：

struct msgid\_ds

{

uid\_t shm\_perm.uid;

uid\_t shm\_perm.gid;

mode\_t shm\_perm.mode;

};

成功时返回0，失败时返回-1.

消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级

消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除。

消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取。

### 线程间的共享和非共享资源

**共享资源**：

**进程代码段**（保存有相同的代码）

进程的**公有数据全局变量，静态变量，全局堆**(利用这些共享的数据，线程很容易的实现相互之间的通讯)

进程打开的文**件描述符、信号的处理器**

非共享资源：

**1.线程ID** 每个线程都有自己的线程ID，这个ID在本进程中是唯一的。进程用此来标

   识线程。

**2.寄存器组的值**       由于线程间是并发运行的，每个线程有自己不同的运行线索，当从一个线

   程切换到另一个线程上  时，必须将原有的线程的寄存器集合的状态保存，以便

   将来该线程在被重新切换到时能得以恢复。

**3.线程的栈** 栈是保证线程独立运行所必须的。  
      线程函数可以调用函数，而被调用函数中又是可以层层嵌套的，所以线程

必须拥有自己的函数堆栈，  使得函数调用可以正常执行，不受其他线程的影响。

**4.线程私有数据**

### 死锁解决方式

1. 死锁预防：
2. 打破占有且申请条件。可以实行资源预先分配策略。即进程在运行前一次性地向系统申请它所需要的全部资源。如果某个进程所需的全部资源得不到满足，则不分配任何资源，此进程暂不运行。只有当系统能够满足当前进程的全部资源需求时，才一次性地将所申请的资源全部分配给该进程。由于运行的进程已占有了它所需的全部资源，所以不会发生占有资源又申请资源的现象，因此不会发生死锁。但是，这种策略也有如下

缺点：

（1）在许多情况下，一个进程在执行之前不可能知道它所需要的全部资源。这是由于进程在执行时是动态的，不可预测的；

（2）资源利用率低。无论所分资源何时用到，一个进程只有在占有所需的全部资源后才能执行。即使有些资源最后才被该进程用到一次，但该进程在生存期间却一直占有它们，造成长期占着不用的状况。这显然是一种极大的资源浪费；

（3）降低了进程的并发性。因为资源有限，又加上存在浪费，能分配到所需全部资源的进程个数就必然少了。

2.银行家算法

在避免死锁方法中允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次分配资源的安全性，进行安全判断（即尝试把此资源分配出去，然后看后面的进程有没有一条能完全执行的路径）若分配不会导致系统进入不安全状态，则分配，否则等待。为实现银行家算法，系统必须设置若干数据结构。

（1）这个算法要求**客户数保持固定**不变，这在多道程序系统中是难以做到的。

〈2〉这个算法保证所有客户在有限的时间内得到满足，但实时客户要求快速响应，所以要考虑这个因素。

〈3〉由于要寻找一个安全序列，实际上增加了系统的开销。

### 同步与异步 阻塞与非阻塞

**1.同步与异步**  
同步和异步关注的是**消息通信机制** (synchronous communication/ asynchronous communication)  
所谓同步，就是在发出一个\*调用\*时，在没有得到结果之前，该\*调用\*就不返回。但是一旦调用返回，就得到返回值了。  
换句话说，就是由\*调用者\*主动等待这个\*调用\*的结果。

而异步则是相反，**\*调用\*在发出之后，这个调用就直接返回了，所以没有返回结果**。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在\*调用\*发出后，\*被调用者\*通过状态、通知来通知调用者，或通过回调函数处理这个调用。

典型的异步编程模型比如Node.js

举个通俗的例子：  
你打电话问书店老板有没有《分布式系统》这本书，如果是同步通信机制，书店老板会说，你稍等，”我查一下"，然后开始查啊查，等查好了（可能是5秒，也可能是一天）告诉你结果（返回结果）。  
而异步通信机制，书店老板直接告诉你我查一下啊，查好了打电话给你，然后直接挂电话了（不返回结果）。然后查好了，他会主动打电话给你。在这里老板通过“回电”这种方式来回调。

2. 阻塞与非阻塞  
阻塞和非阻塞关注的是**程序在等待调用结果（消息，返回值）时的状态.**

阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。  
非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

还是上面的例子，  
你打电话问书店老板有没有《分布式系统》这本书，你如果是阻塞式调用，你会一直把自己“挂起”，直到得到这本书有没有的结果，如果是非阻塞式调用，你不管老板有没有告诉你，你自己先一边去玩了， 当然你也要偶尔过几分钟check一下老板有没有返回结果。  
在这里阻塞与非阻塞与是否同步异步无关。跟老板通过什么方式回答你结果无关。