目录[-]

 [计算机组成](http://my.oschina.net/pangyangyang/blog/188508?p=5#OSC_h1_1)

 [硬件层](http://my.oschina.net/pangyangyang/blog/188508?p=5#OSC_h1_2)

 [操作系统&进程](http://my.oschina.net/pangyangyang/blog/188508?p=5#OSC_h1_3)

 [并发性](http://my.oschina.net/pangyangyang/blog/188508?p=5#OSC_h1_4)

 [内存管理](http://my.oschina.net/pangyangyang/blog/188508?p=5#OSC_h1_5)

 [处理器调度](http://my.oschina.net/pangyangyang/blog/188508?p=5#OSC_h1_6)

 [I/O调度](http://my.oschina.net/pangyangyang/blog/188508?p=5#OSC_h1_7)

 [文件管理](http://my.oschina.net/pangyangyang/blog/188508?p=5#OSC_h1_8)

    出来混迟早是要还的，本科时候玩得太开心了，工作了几年一直觉得本科时候的课程是一种缺失，所以干脆裸辞重新拾起那些本该铭记的课程。这多总结一共花了自己差不多二十天的时间，是在深圳大学城图书馆一点点学习整理的，记录下来留给自己回忆和大家分享。毕竟对操作系统的掌握还是很有限，图中难免错误欢迎大家指正、提醒。谢谢各位看官。

    闲话少说，先果断上一张完成的大图，然后再慢慢画起（由于osc有200k限制，没有直接贴大图，在每个图后面加链接看大图）。另外这里只是简单的知识点罗列，同样发布了一篇完整的（2w字，还没来得及校对）总结，[欢迎查看](http://my.oschina.net/pangyangyang/blog/188507)。

[大图](http://zhirenguo.com/images/008.jpg)

# 计算机组成

    先从最简单的开画，这里计算机的组成员工就三层：应用程序、操作系统、硬件。操作系统的作用是管理资源（各种硬件、计算资源、时间等），职称程序的运行（提供了必要的库支持）。

# 硬件层

    先来看硬件层东西：  
    **CPU**：快的不得鸟的东西，它总是被人拖后腿，所以有了跟中缓冲。除此还可以考虑下CPU性能提升的技术：流水化、超标量设计、多核心。  
     **主存**：除去CPU的高速缓冲，最快的就是主存了。当时数据不能永久保存、价格高是硬伤。  
     **总线**：连接了所有设备。又分为多级总线，总线设备之间冲突的解决因此需要总线决策。  
    **I/O设备**：各种输入输出。进化过程：可编程I/O、I/O中断、DMA。  
    **局部性原理**：分为空间局部性和时间局部性，这是缓存有价值的基础。

# 操作系统&进程

[大图](http://zhirenguo.com/images/002.jpg)

    操作系统目标：

* 方便：提供方便的交互和资源管理
* 有效：分配资源，更有效的资源利用（内存管理、处理器调度、I/O调度等）
* 扩展能力：不中断服务增加系统特性的能力

     操作系统发展：   
    串行-->简单批处理-->多道程序批处理-->分时系统

* 串行：这个时候根本就没有操作系统概念。
* 简单批处理：人工组织任务，操作系统负责调度完成一个后继续下一个，省下了每次人工复位输入任务的时间。
* 多道程序设计：最计算资源有了时间片划分，CPU不用因为一个进程I/O阻塞而空转了。
* 分时系统：计算资源昂贵的背景下，每个用户的分时。

    进程：  
    程序执行的最小单位。进程的组成包括：用户程序、数据、栈、进程控制块。  
    进程状态切换：  
    五个状态如上图：新建、就绪、运行、阻塞、退出  
    进程的切换：  
    将进程切换到硬盘的特定区域，目的是追求更多的进程可以同时运行——因为即便引入了多道程序设计CPU仍然很空闲（其它设备太慢了，特别是I/O），所有越多的进程进入进来，CPU的利用率越高。  
进程执行模式

    内核模式、用户模式，模式区分的目的是对进程功能的限制，高风险的CPU指令应该只有操作系统可以执行。

[大图](http://zhirenguo.com/images/003.jpg)  
    聊完进程聊线程，一个进程可以是多线程的。有了进程之后还要有线程的原因是进程的换入换出、数据交换开销太大，线程远远小于进程的开销。首先，进程之间通信需要获取共享资源锁，而同一进程内的线程完全享有进程获得的资源，也因此线程的编写要特别注意同步问题。第二，进程的创建成本很高，涉及空间分配、初始化栈、初始化数据等，而线程的创建成本低很多可以包括操作系统创建和进程调用库自己创建。于是有了两种不同类型的线程——用户级别和内核级别。第三，多道程序设计中线程的换入换出比进程来得快多了，因为不需要保存进程控制块等一大堆数据嘛（当然要保存线程控制块，但是成本很低）。最后，销毁成本低。  
    说到线程的两种类型——用户、内核。用户级别的线程是进程调用线程库创建的，对操作系统是透明的，是进程中通过代码实现切换的因此因此一旦一个线程引起了I/O阻塞从操作系统的视角就是进程阻塞，所以要阻塞整个进程；而内核级别的线程是操作系统创建的，所以该阻塞线程的不会阻塞整个进程，当然效率也就没有用户级别的高。  
    多线程的应用场景：

* 用户界面和后台数据运行：用户界面渲染和后台数据运算使用不同的线程，没有喜欢后台一运算界面就卡死的程序吧。
* 异步计算：不如实时备份，一个线程定时触发就可以了，没必要进程定时的检查是不是该备份了。
* 速度敏感的计算：可以并发计算的、多核处理的，在计算和组织上线程更具优势。
* 模块化：模块化的程序有时候很适合用多线程来设计。

# 并发性

[大图](http://zhirenguo.com/images/004.jpg)

    多道程序设计带来了效率的提升也带来复杂性的提升。进程之间、线程之间的竞争可能会坏菜，有几种方式导致坏菜——竞争条件、死锁、饥饿。竞争条件——多个进程同时访问一个资源，最后状态与各个进程的竞争顺序敏感；死锁：多个进程各持有一个资源请求另外的资源，大家互不相让；饥饿：竞争不过的进程一直得不到所请求的资源，结果耗着不动了。  
    有三个经典的并发问题：

* 生产者/消费者：消费者不能消耗还没来得及生产的资源，生产者不能在容器满了以后继续往里面塞。
* 读者/写者问题：资源可以被多个进程读，但是同一时间只能有一个写者并且写者与读者互斥。
* 哲学家就餐问题：这个是一个死锁的问题，五个哲学家没用要用两个叉子吃饭，他们先拿起左手叉子，再拿起右手叉子，然后进餐，然后释放叉子。桌上一共五个作为五把叉子，如果哲学家同时坐下就会发生死锁。

    互斥：进程间尝试访问不可共享的资源时应该是互斥的，这个资源成为临界资源（critical resource），程序中访问临界资源的代码段成为临界区。  
    既然有竞争和并发，就需要提供必要的支持。一种是硬件提供的如CPU特定的指令Compare&Swap、Exchange等，这种方案有一个要命的问题是效率差需要忙等待，另外还有死锁和饥饿的问题，软件级别提供的支持是更好的途径，包括信号量、监视者、消息传递。  
    关于并发性的支持很好的解决了竞争条件的问题，下一个问题是死锁。操作系统对死锁的解决方案包括三个区域：死锁的预防、死锁的阻止、死锁解决。  
    死锁的预防目的在于阻止死锁形成条件中的一个的形成，这里有四个条件：进程互斥、存在持有和等待（资源）、不存在抢占、形成环路。总的来说死锁的预防存在不可能实现（进程互斥）、条件苛刻、效率地等问题。  
    死锁的阻止这是更加实际的途径，有两个切入点：1，进程预先声明需要的资源，只有资源满足情况下才允许进程启动——效率低。2，只有不会导致死锁情况下才分配资源给进程。  
    死锁的解决，这个听起来有点粗暴。前提条件是死锁可以回滚，一旦发生了死锁就取消所有进程，回滚到之前的状态重新竞争，并不断重复类似的过程——由于竞争的不确定性，最终一定会解决死锁。

# 内存管理

[大图](http://zhirenguo.com/images/005.jpg)

    这块大概是最精彩的了。先说内存分区的集中方式——固定大小，等大小的划分限制了进程大小并且存在明显浪费（每块尾部都有剩余），稍微改进一点变成不等大小的固定划分一定程度上能加入更大的程序了，但是限制仍然存在效率也非常低——很多内部碎片。动态划分，来一个进程划分一块的方式会产生外部碎片，需要不断的进行压缩才能保证有足够的连续空间给新进程。分页，把内存划分成固定大小的很多小块需要页表的支持，但是内存的利用提高了很多，会产生很少的内部碎片。分段，很分页的思路一样都是进程可以划分到不连续的内存区域中，需要段表支持。段页结合，每个段内划分成页，这样进程更利于数据共享（分段的便利），同时消除了外部碎片（段释放后无论如何都是若干页），存在少量的内部碎片。  
    还有一处漂亮的地方是虚拟内存，由于局部性原理。进程中常用的数据往往只是一部分，其它部分只有在需要时在换入就可以了。在进程运行过程中常驻内存的部分称为**常驻集**。虚拟内存作用就是在硬盘上划分了一块区域用于进程的换入换出，虽然硬盘也很慢但是比它更慢的是I/O，有了虚拟内存就能把阻塞的进程及时换出，换入就绪进程的部分或者全部，让更多的进程同时运行提升CPU的利用率。  
    有了驻留集的感念就该涉及到驻留集的管理了，首先驻留集可以有几种分配方式：

* 固定分配，局部范围：每个进程的驻留集是固定不变的，在进程启动时决定，块的换入换出总发生在进程内部。
* 动态分配，局部范围：进程的驻留集是动态调整的，内存块的交换限制在进程内部（即一个进程的数据被请求时，只能替换自己的块出去）
* 动态分配，全局范围：同上，但是在替换策略上可以换成任意进程中的内存块。

    在驻留集固定分配情况下，页的替换策略包括：先入先出、最近最少使用、始终。  
    当驻留集动态分配时，页的替换策略变得更精彩：工作集（很理想，开销太大）、页错误频率（定期查看频率，决定扩充或清理）、采样间隔可变的工作集（考虑页错误频率在局部性发生偏移时或膨胀的改进方案，间隔时间是可变的）。  
    除此以外还有加载侧路的问题——多道程序设计导致加入的进程太多时，同样会导致每个进程的驻留集太小导致页错误率居高的情况。

# 处理器调度

[大图](http://zhirenguo.com/images/006.jpg)  
    处理器调度在单处理器和多处理器的系统中呈现出完全不同的侧重，单处理器的情况下系统CPU利用率、响应时间称为非常重要的关注点，多处理器下由于计算资源较多响应时间能够很好的保证，CPU的利用率变得不太重要，如何能有效的减少进程交换、线程依赖导致的阻塞等成为了更关注的点。在单处理器下进程的调度策略包括：

* 先来先服务：最简单、最懒的办法，效果一般，偏向于长进程。
* 轮转：平均分配每个进程的时间，增加了进程的切换次数，对I/O密集型的进程不公平。
* 最短进程优先：需要估计进程的运行时间，偏向于短进程。
* 最少剩余时间优先：上面策略的抢占版本，有助于提高吞吐率，但是如果短进程不断到来，长进程可能会饥饿。
* 最高响应比优先：(w+s)/s，w为等待时间，s是服务时间，偏向于短进程，但是当长进程等待时间不断增加时它的响应比会增大，最后超越短进程。
* 反馈：不需要估计进程运行时间的策略，分多级优先级递减的队列，进程运行中会被超时中断，加入到次一级的队列中。也就是运行时间越长优先级会越低，同样可能导致长进程饥饿，因此在等待时间超过一定长度时尝试把进程重新拿到高优先级队列来。

    多处理器的调度则不太相同，主要关注多线程的调度策略包括：

* 负载分配：任务池和处理器池，始终保持处理器忙碌。
* 组调度：考虑到线程间可能存在依赖的情形，一组相关的线程分配一对一的处理器同时运行。
* 专用处理器分配：组调度的升级版，对进程分配特定的处理器，进程内的所有线程在专用处理器上运行直到进程运行结束才回收处理器。
* 动态分配：进程的线程数可以动态改变，由操作系统和程序共同决定分配和负载。

# I/O调度

[大图](http://zhirenguo.com/images/007.jpg)  
    I/O功能的逻辑结构  
　　　逻辑I/O：逻辑I/O层将I/O设备视为逻辑资源，不关心底层的细节。逻辑I/O代表用户进程管理的一般I/O功能，允许它们根据设备标识符以及诸如打开、关闭、读取等操作与设备打交道。  
　　　设备I/O：请求的操作和数据（缓冲的数据、记录等）被转换成适当的I/O指令序列、通道命令和控制器指令。可以使用缓存技术以提高使用率。  
　　　调度和控制：关于I/O操作的排队、调度和控制发生在这一层，可以在这一次处理中断，收集和报告I/O状态。这一层是与I/O设备真正打交道的软件层。  
    I/O的慢是出了名的，所以更需要缓冲，需要指出的是缓存是一种技术旨在平缓I/O请求峰值的，当进程需求的I/O平均值大于I/O传输速度是再多的缓冲也不能解决问题。

* 单缓冲：系统在发送I/O指令后给I/O设备分配一块位于主存中的缓冲区，传输数据被放在缓冲区中，当传输完成时立即尝试读取下一块。根据局部性原理有理由期望下一块被读取，这种机制称为超前读。
* 双缓冲：单缓冲时I/O设备必须等待读取缓冲区中数据被完全读出才能再次写入，双缓冲设置两块缓存区域以平滑这种趋势。设C为进程处理块的时间，T位读取块的时间，我们可以粗略估计块的执行时间位max（C，T），当C>=T是进程将不需要等待I/O，当C<T是则I/O设备可以全速运行。
* 循环缓冲：当爆发性的执行大量的I/O操作时，则仅有双缓冲就不够了，这种情况下使用多于两个缓冲的方案来缓解，这种缓冲区域自身被当成循环区域使用。

    除此以外，还有一个不是那么慢的重要设备就是磁盘了，磁盘的耗时一共分三部分：寻道时间，磁头臂移动到磁道的时间这个是最大的消耗；旋转延迟，找到磁道以后转到读取数据位置的时间，一般认为是1/2r，r是转速；数据读取时间，读数据的时间是与数据多少相关的。硬盘调度的策略目的在于减少磁头臂移动的次数和距离，主要是利用了局部性：

* 先进先出：先来的请求先服务，由于数据的请求式随机的，会导致较高的寻址时间，效率差。
* 优先级：优先级是高优先级的请求先服务，一般是为了满足操作系统的特定目的，并没有改善磁盘性能的能力。
* 后进先出（LIFO）：令人惊讶的是这种选取最近请求的策略有许多优点。把设备资源提供给最近（使用系统）的用户时会导致磁头臂在一个顺序文件中移动时移动的很少，甚至不移动。利用这种局部性原理可以提高吞吐量减少队列长度，只要一个作业积极的使用磁盘它就能尽快得到处理。
* 最短服务时间优先（SSTF）：总是选择磁头臂移动最少的请求响应，对于移动距离相等的请求可以随机移动向一边。同样如果一个进程大量的请求临近的数据会导致其它请求饥饿。
* SCAN：SCAN要求磁头臂向一个方向移动，移动过程中响应在当前磁道的请求。当磁头臂到达最外（内）层磁道时，再反向扫描。这种算法并没有很好的利用局部性原理（对最近横跨过的区域不公平，不如SSTF和LIFO好），由于两侧的数据被快速的扫描了两次因此它偏向于外围数据（局部性原理）。
* C-SCAN：限定在一个方向扫描，当达到最后一个磁道时，磁头臂返回到相反方向的磁道末端重新开始扫描。
* N-step-SCAN和FSCAN：为了克服进程的粘滞性，在SCAN和C-SCAN中一个或多个进程对一个磁道有较高的访问速度时可能会垄断这个磁道一段时间。N-step-SCAN设置若干个N个请求的队列，每次扫描只响应一个队列里的请求，当开始扫描时新的请求需要加入到下一个队列中。

    除了好的调度策略，另一个减少开销的办法是在主存中建立硬盘的高速缓冲，进行预读。高速缓冲的替换策略包括：

* LRU：最少访问，将缓冲区设置为一个栈，当一个块被访问后加入到栈中，如果再次得当访问则把该块从当前位置移动到栈顶，随着块的加入那些不被访问的将会挤出栈中。
* LFU：最小频率访问，对缓存中的块增加计数特性，每次被访问到时计数加1。当访问辅存时，把计数最小的块移除，加入最近的块。由于局部性的问题，一个块可能短时间内多次访问使得计次很高，但是这之后并不意味着还会再次访问它，所以LFU并不是一个好的算法。
* 基于频率的替换算法：克服LFU的确定，在LRU的栈模型中划分出位于栈顶的若干帧为新区，当块位于新区是重复访问不增加访问次数。

# 文件管理

[大图](http://zhirenguo.com/images/008.jpg)   
文件结构：

* 域（Field）：基本的数据单元，一个域包含一个值，如雇员名字、日期等。
* 记录（Record）：一组相关域的集合，可以看做应用程序的一个单元。
* 文件（File）：一组相关记录的集合，它被用户或应用程序看做一个实体，并可以通过名字访问。
* 数据库（DB）：一组相关数据的集合，它的本质特征是数据之间存在着明确的关系，可供不同的应用程序使用。

文件组织和访问

* 堆：最简单的文件系统。数据按它们到达的顺序被组织，通过特定的分隔符或特定的域指明。每个域应该是自描述的，数据之间不一定存在联系或相同的格式。当数据在处理器采集并存储时或者当数据难以存储时会用到堆。只能穷举搜索，对大多数应用都不适用。
* 顺序文件：文件具有固定的格式，所有的记录都有相同的格式，具有相同数目、长度固定的域按照顺序组成。每条记录第一个域称为关键域，唯一标识了这条记录。交互的表现较差，需要顺序的搜索。一种情况下顺序文件按照顺序存储在磁盘上，在发生文件修改时需要移动数据，可能的处理方式是把数据存在临时堆上，定期的将数据批量写回顺序文件。另一种情况文件可能采用链式存储，该方法带来一些方便，但是是以额外的处理和开销为代价的。
* 索引顺序文件：弥补了顺序文件检索的问题。检索文件可以是简单的顺序文件，每条记录包括两个值一个关键域和指向文件的指针。简单的检索可以是一级的，也可以由多级检索。查找文件时找到相等的域或者关键域值之前最大的域。
* 索引文件：顺序文件和索引顺序文件只能是顺序检索或对关键域检索，索引文件对感兴趣的域提供了索引，索引文件本身可以是顺序的。索引文件分为完全索引和部分索引，差别在于被索引的域是全部域还仅仅是感兴趣的域。索引文件一般只提供索引访问的方式，不再支持顺序访问，因此记录的组织也不必是顺序的，应用程序只能通过指针访问。
* 直接文件或散列文件：直接文件或散列文件开发直接访问磁盘中任何一个地址一致的块的能力，要求每条记录中都有一个关键域，但这里不存在顺序的概念。

二级存储管理   
文件分配   
　　预分配：文件请求时声明需要的空间，一次性分配。   
　　动态分配：根据文件的增长每次分配一定的空间，或者一块。   
分配方法：

* **连续分配**：始终给文件分配连续的空间。这种分配方式对于顺序文件非常高效，并且可以简单的检索到需要的文件块。但是会产生外部碎片，并且需要实时运行压缩程序以消除外部碎片。
* **链式分配**：文件不需要顺序保存，每块尾部通过指针指向下一块数据，文件分配表中同样只要保存一个表项。链式分配的一个后果是局部性不再被利用，如果要顺序的读取一个文件时需要一连串访问磁盘的不同部分，这对系统的影响很严重。一些系统周期性的的对文件进行合并。
* **索引分配**：每个文件在文件分配表中有一个一级索引，分配给文件的每个分区在索引中都有一个表项，典型的文件索引在物理上并不保存在文件分配表上，而是保存在独立的一个块上，文件分配表中该文件索引指向这个块。可以消除外部碎片，按大小可变的分区分配可以提高局部性，任何情况下（大小可变分区、按块保存）都需要不时的对文件进行合并。使用大小可变的分区情况下合并可以减少文件索引。索引分配支持顺序和直接读取数据，是最普遍的一种文件分配形式。

空闲空间管理

* 位表：每块占用一位，如果被占用了用1表示否则用0，占用的空间很小，并且需要在主存中，需要顺序的检索空闲空间，当剩余空闲空间很小时效率非常低。
* 链式空闲区：几乎不需要保存空间，每个空闲块直接通过指针连成链状，寻找连续空间的效率比较低，无法利用局部性。
* 索引：通过索引指明空闲的区域和长度，在大小可变的分配下很有效。
* 空闲块列表：最常用的方法。通过栈保存了每块空余区域的地址，由于可以部分加载在主存中所以有很高的效率。