目录[-]

 [计算机组成](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h1_1)

 [硬件层](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h1_2)

 [CPU：](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_3)

 [总线：](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_4)

 [速度：](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_5)

 [局部性：](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_6)

 [I/O设备进化：](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_7)

 [操作系统](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h1_8)

 [操作系统](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_9)

 [操作系统目的：](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_10)

 [操作系统发展](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_11)

 [操作系统的几张表](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_12)

 [进程和线程](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_13)

 [进程](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_14)

 [进程的组成：](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_15)

 [进程状态切换](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_16)

 [进程切换（Swapping）：](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_17)

 [进程执行模式：](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_18)

 [多线程定义](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_19)

 [与进程的关系](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_20)

 [创建方式](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_21)

 [线程优势](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_22)

 [线程类型](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_23)

 [线程的四个应用场景](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_24)

 [并发性](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h1_25)

 [竞争条件](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_26)

 [临界区](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_27)

 [互斥（multual exclusion）](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_28)

 [进程交互](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_29)



 [并发性的硬件支持](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_31)

 [Interrupt disable](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_32)

 [Compare & swap instruction](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_33)

 [Exchange Instruction](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_34)

 [硬件支持存在的弊端：](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_35)

 [并发性的软件支持](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_36)

 [信号量（semaphore）](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_37)

 [生产者与消费者](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h4_38)

 [监听者（管程）](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_39)

 [消息传递](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_40)

 [读者和写者问题](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_41)

 [死锁和饥饿](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_42)

 [联合进程图（Join Process Diagram）](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_43)

 [资源类型](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_44)

 [死锁形成的条件](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_45)

 [死锁预防（protection）](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_46)

 [死锁避免（avoidance）](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_47)

 [死锁的检测和恢复](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_48)

 [哲学家就餐问题](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_49)

 [内存管理](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h1_50)

 [内存分区](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_51)

 [固定分区](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_52)

 [动态分区](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_53)

 [伙伴系统](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_54)

 [重定位](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_55)

 [分页](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_56)

 [页表](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h4_57)

 [根据页表寻址](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h5_58)

 [反向页表（Inverter Page Table）](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h4_59)

 [转移后备缓冲器（TLB，Translation Lookside Buffer）](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h4_60)

 [分段](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_61)

 [段页结合](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_62)

 [虚拟内存](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_63)

 [系统抖动](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_64)

 [页缓冲](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_65)

 [驻留集的替换策略](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_66)

 [工作集策略（Working Set）](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h4_67)

 [页错误频率（Page Fault Frequency，PFF）](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h4_68)

 [采样间隔可变工作集（Variable-interval Simple Working Set，VSWS）](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h4_69)

 [处理器调度](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h1_70)

 [单处理器调度](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_71)

 [调度类型](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_72)

 [调度准则](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_73)

 [面向用户](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h4_74)

 [面向系统](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h4_75)

 [调度算法](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_76)

 [多处理器调度](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_77)

 [多处理器调度关注的类型](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_78)

 [粒度](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_79)

 [进程调度](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_80)

 [线程调度](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_81)

 [I/O设备调度](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h1_82)

 [I/O功能的逻辑结构](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_83)

 [I/O缓冲](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_84)

 [磁盘调度](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_85)

 [磁盘性能参数](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_86)

 [磁盘调度策略](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_87)

 [RAID](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_88)

 [磁盘高速缓冲](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_89)

 [文件管理](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h1_90)

 [文件系统软件结构](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_91)

 [文件组织和访问](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_92)

 [记录组块](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_93)

 [二级存储管理](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h2_94)

 [文件分配](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_95)

 [分配方法：](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h4_96)

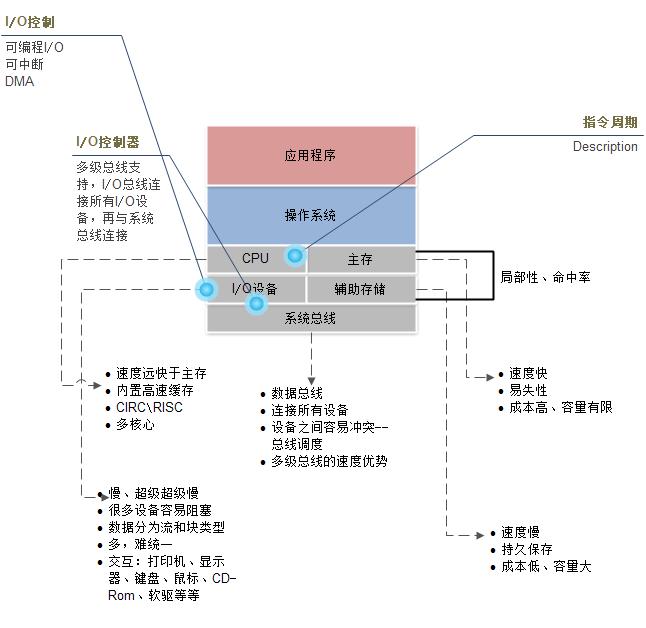
 [空闲空间管理](http://my.oschina.net/pangyangyang/blog/188507?fromerr=a2lWjStQ#OSC_h3_97)

**计算机组成**

[](http://static.oschina.net/uploads/space/2013/1228/105507_eSCm_870846.jpg)

计算机的运行简单理解为这三层：硬件即组成计算机的所有摸得见看得着的东西是计算机运行的基础；应用程序即完成特定功能、目的的用户程序是计算机的价值体现；中间就是操作系统，连接了硬件和应用程序负责硬件调度、资源管理和分配（内存、文件、CPU等等）、安全等一系列功能。

**硬件层**

[](http://static.oschina.net/uploads/space/2013/1228/110149_M5il_870846.jpg)

主要硬件包括CPU（算术、逻辑单元）、主存、辅助存储、系统总线、I/O设备（即输入输出）。

**CPU：**

CPU本身（Processor）可以是单核、多核、多CPU架构，主要目的就是满足日益增长的运算需求。单核比较简单，多核（包括一CPU多核心和多CPU）立即涉及到核心之间高速缓存的共享，此处是CPU内置高速缓存非主存，进程之间寄存器数据的共享和进程处理器调度问题。

**总线：**

总线连接了所有的设备提供通讯的能力，注意设备之间同一时间的通信是会冲突的，这就要涉及到总线的决策，负责决策的可能是CPU或专有芯片。所有设备需要通信时要提交通信请求有CPU决定下一个通信的设备。

另外一个问题显然连接到总线上的设备越多，冲突的几率就越大效率越差。所以总线可以有多层总线设计，比如设置I/O总线所有的I/O设备都通过I/O总线和I/O控制器连接，I/O控制器则连接在系统总线上代理I/O的通信请求。

**速度：**

CPU飞快，其中CPU内置的一级高速缓存保持了和CPU同样的速度但是容量极小，接下来可能存在二级、三级高速缓存；主存（内存）其次和CPU存在数量级上的差距；辅寸（多硬盘）的速度就不是一般的慢了，因为有机械运动，但是容量大很多；I/O设备多数慢的要死，但不是没有快的（比如图形、千兆以太网的速度是超过硬盘的）。总而言之就是快的太贵，贱的太慢，访问快的断电数据即失效，不失效的访问慢，所以就有了多级存储的设计。程序的运行必然是加载到主存的（内存）！

**局部性：**

多级存储的设计得益于局限性能够大幅提升性能。局限性本身分为时间局限性和空间局限性。时间局限性是说现在用到的指令很可能短时间内还会多次调用，空间局限性是现在调用的数据在辅寸上邻接的块很可能即将被用到。就是因为局限性的存在预读取才有了市场（所以有了磁盘整理），当然局限性是必然的——程序肯定趋向于统一存取、循环、分支逻辑肯定是指令的循环调用。——好的命中率决定了计算机的性能。

指令周期（取指周期）：

计算机中CPU始终是取指-》执行的动作循环运行的，计算机从取指到完成指令的周期称为取指周期。因此针对CPU的性能优化有流水线和超标量体系结构，前者目的在于合理分割指令使取指和执行能够并行进行（预取指），后者则通过相同的运算结构重复设置实现多条指令同时执行（得益于指令的乱序执行）。

**I/O设备进化：**

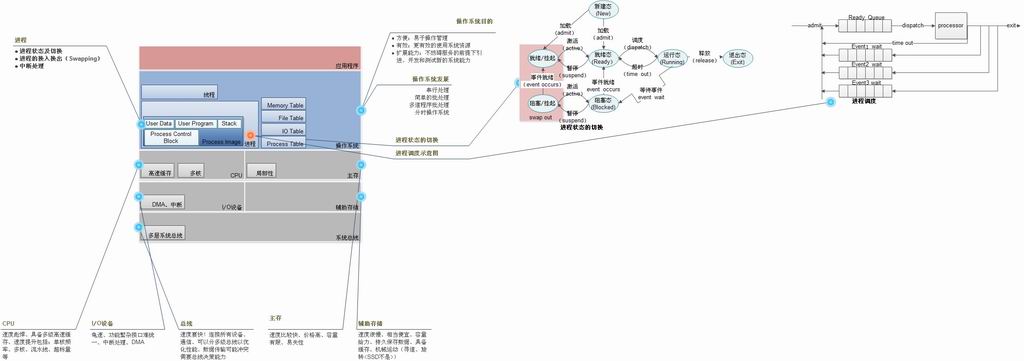
I/O设备一般较慢，极端点的比如键盘这货简直慢的没法说。那么当程序需要读取I/O数据时（如读取一块数据、监听键盘输入）就会被I/O设备阻塞，这是最差的情况整个系统空转直到I/O设备读取数据完成。

于是有了可编程I/O设备——你可以定期问我准备好数据了没，好了你就取没好你就干别的去，也就是轮询这法子还是非常慢因为CPU要定期过来问而且多数情况都是没准备好空耗系统资源（进程切换）。

再后来有了中断，前提是指令的运行是可被中断和恢复的（现在当然都可以，把寄存器数据保存到缓存，完了再恢复嘛），需要读取的时候CPU发给I/O指令然后不理它了，需要读取的进程（或线程）进入阻塞状态换下一个进程来执行，当I/O准备好数据后发送一个中断信号给CPU，这时现在执行的进程被中断CPU会执行一段中断处理程序（通常很短）把之前阻塞的进程标记为Ready（可执行）状态，处理完中断后恢复之前中断的进程（或线程）继续执行。在当前进程执行完或者超时中断后（分时多道程序处理，超时也是一种中断），之前从阻塞中恢复的进程可能会被执行（取决于进程调度，不一定是下一个时间片里也可能是下几个时间片后或者干脆饿死了）。

再后来又有了DMA（直接存储器访问），主要原因就是CPU很忙，你一个拷贝/传输整块数据的动作就不要每块数据都让我来处理了，系统中多了一个专门的辅助芯片干这个事情。CPU下达指令后辅助芯片负责设备之间的数据直接传输。DMA模块可以是总线中的一个模块也可以是I/O模块，但是仍然要占用总线（传输数据）所以并不是不会对系统性能产生影响，至少DMA冲突时CPU要等待一个总线周期。

**操作系统**

[](http://static.oschina.net/uploads/space/2013/1228/114504_az8Q_870846.jpg)

看完硬件层（简单看看，后面可能还要回头），再来看看操作系统层。最基本的元素肯定要包括进程、线程等等用于程序的执行。

**操作系统**

**操作系统目的：**

方便：简化了计算机的使用，无论是用户还是开发者角度都极大的简化了对计算机的使用。用户角度提供了交互的能力，开发者角度提供了底层设备的接口、公共库等等。

有效：提高计算机的利用效率。对于操作系统CPU运算、内存、辅存、I/O等等都是资源，如何能最大的利用资源是计算机要考虑的事情。（没有操作系统的年代显然效率非常低，同一时间只运行一个程序计算资源多数时间都在等待I/O设备、人等）。

扩展的能力：在不阻碍服务前提下开发、测试、加入新的计算机能力。比如安装个程序、加个设备。

**操作系统发展**

串行处理：这是个久远的年代（上世纪50年代也不算太远哈）,计算机一次只能运行一个程序，要通过输入设备读入程序（读卡器吧）运行结束后再将结果输出到打印机。这个年代是没有操作系统的。

简单批处理：这个算是操作系统的鼻祖吧？就是常驻内存的一个监控程序，要运行的程序被管理员组织成一批，监控程序从存储器（卡片或磁带）读取要执行的Job将处理器控制权转交给程序运行结束后（成功或失败）控制权返回监控程序继续读入下一个任务。

简单批处理节约了计算机调度和准备的时间——任务不再是一个一个的处理了，变成一批了。

多道程序批处理：现代操作系统进程切换之父？哈哈。由于内存的加大除了容纳操作系统、一个程序以外还有足够的空间容纳第二、第三个程序，所以就有了同时运行多个程序的能力。在第一个程序被阻塞后（I/O等），可以转交控制权个第二、第三个程序。

多道程序批处理节约了CPU等待I/O等慢速设备的时间，这个效率的提升非常客观。

分时操作系统：注意关注在人的交互上。人肯定是比I/O还慢的设备了，由于早年计算机资源的稀缺当然要达到多人共用一台机器的目的。分时操作系统把计算机资源做时间切片，用户通过终端连接到计算机，每个中断都获取到时间切片内的计算资源，由于人是反应很钝的，所以就像没人都有一台计算机服务一样。

**操作系统的几张表**

Memory table：记录内存的使用情况，回收和分配内存时这里都会被更新。

I/O table：记录I/O设备和通道的情况。

File table：文件系统的占用情况，文件是否存在，文件状态。

Process table：管理进程。

**进程和线程**

**进程**

进程包括程序代码和一组（set）数据，是程序运行的实体（应该能算作最小单元吧，线程能执行的是一段逻辑，一个程序的启动至少是一个或多个进程）。

**进程的组成：**

进程所有属性的集合被称为进程映像（Process Image），这之中一共包括四部分：用户程序（User Program）、数据（User Data）、栈（Stack）和进程控制块（Process Control Block）。

用户程序：要执行的程序。

数据：用户空间中可以被用户修改的部分。比如进程数据、可修改程序。

Stack：每个进程都有一个或者多个栈用于保存参数、过程调用地址、系统调用地址。

进程控制块：操作系统控制进程需要的数据。包含了进程的属性，最基本的每个进程总要有个Id吧，还有进程状态、当前寄存器的值、程序计数器、Stack的指针等等。

**进程状态切换**

最起码的两个状态：Ready、Running，进程自创建后进入Ready状态也就是可以执行的，这时的进程进入等待队列知道进程调度轮到自己执行时才能够被分配资源进入执行状体Running。

进程的基本状态一共五个，包括上面两个以外还会有被阻塞的情况（I/O、等待生产者生产、信号量等等）所以存在Blocked状态，进程创建过程中存在New状态、进程运行终结后处于Exit状态等待操作系统做进一步处理并销毁进程。

进程状态之间的切换可以参考图中进程状态部分，大体过程：进程被允许创建进入New状态，这个过程中要分配进程Id、划分内存区域、初始化PCB（Process control block）、连接和创建或扩展其它的数据；上述过程完成以后进程可以运行了所以进入Ready状态等待系统调度；终于等到自己运行了，进程为Running状态这时候可能出现几种情况。1，进程运行结束进入Exit状态等待销毁。2，进程运行超时重新进入Ready状态等待下一次调度。3，进程被阻塞了进入Block状态等待所需要的数据或信号准备好，重新唤醒进入Ready状态。除此以外还有两个挂起状态，见下面的切换。

调度的示意图参考进程调度示意子图，其中一个改进是操作系统很可能为不同的中断事件设立不同的阻塞队列，以便提高效率。

**进程切换（Swapping）：**

说白了还是CPU计算资源宝贵和内存有限（内存在涨吃内存的程序也在涨）。为了能让更多的进程可以同时执行（实际上不是同时是调度），程序运行中有很大一部分会被I/O阻塞——原因是I/O太慢了，所以即便你要的数据不多那这个取数的过程CPU也够跑很多其它程序的了。所以导致的问题就是在内存中的进程都阻塞了，内存没地方了，CPU依然闲的蛋疼，怎么办？把硬盘上画出一块地儿（个人理解就是windows下的虚拟内存、Linux中的swap），塞得比较久的那些进程你们先出去待会，换些后面排着的进来。

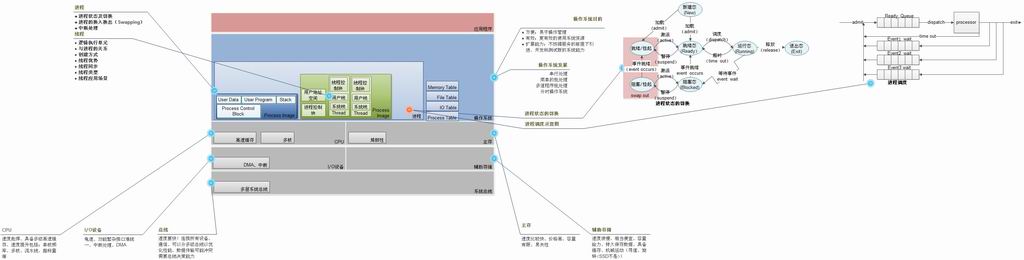
看到进程切换的子图中除了五个状态以外还有两个挂起态（就绪/挂起、阻塞/挂起）就是这个情况。虽然把进程从硬盘换入换出这个开销非常高，但是硬盘比起I/O设备还是快了很多，所以这一步是有价值的。

当然还有一个路子是虚拟内存的时候，这个稍晚的时候再扯进来。

**进程执行模式：**

用户模式和内核模式，这两个模式还有多个别名不记录了。这是出于操作系统安全考虑的，有些重要的指令就只有内核模式下才能被CPU执行（硬件支持），总不能任意来个程序什么事情都给他干吧。

有了执行模式就算有模式的切换，比如进程要调用系统操作时（system call）就有可能从用户模式切换到内核模式，system call执行后也会在切换回去。

[](http://static.oschina.net/uploads/space/2013/1228/115936_xkDK_870846.jpg)

**多线程定义**

多线程是指进程内支持并发路径执行的能力。

**与进程的关系**

一个进程至少包含一个线程，当进程中存在多个线程的时各个线程可以共享进程内的资源。进程内的线程共享进程的用户空间，操作系统仍然通过进程控制块控制进程进而影响到线程。线程本事具备自己的线程控制块、用户栈和系统栈。

**创建方式**

由于线程类型的不同（下面介绍）线程可以是操作系统创建的也可以使通过线程库（Lib）由进程自己创建的，对于进程创建的线程操作系统是不知道的。

**线程优势**

l 创建时间开销远小于进程的创建。因为不需要分配用户空间和那么多初始化动作。

l 销毁线程的成本也远低于进程。

l 线程之间的切换消耗低于进程，特别是同一进程内的线程切换消耗更低。

l 线程间通信的效率比进程间通信要高，因为进城之间安全性问题需要隔离和互斥，同一进程内的线程可以共享进程资源而不需要提前获取锁。

线程同步

进程也涉及到同步，但是线程的同步更需要开发者注意。上面提到了线程共享进程的资源并且不需要获取锁，所以线程之间是没有操作系统来保证隔离的。

**线程类型**

类型分为用户级线程和内核级线程。用户级线程即通过Lib创建的对操作系统是透明的，内核级线程是由操作系统创建的。

|  |  |
| --- | --- |
| **用户级** | **内核级** |
| **通过线程库创建，对操作系统不可见** | 由操作系统来管理 |
| **比内核级的线程更高效，不涉及模式切换** | 在线程切换时需要模式切换（因为是调用系统级的指令） |
| **进程中同时只能有一个线程在运行，还是多段程序的思路** | 线程可以并发运行，特别指多核计算机 |
| **一旦被阻塞整个进程就被阻塞，也就是所有的线程都完蛋了** | 只会阻塞引起阻塞的线程 |

**线程的四个应用场景**

l 前后台运算：即有UI和后台运算的程序，你总不希望后台数据运算时前面的UI界面就对用户没响应了吧？所以这里应该分开线程，后台启动单独的线程运算，界面在不同的线程了所有能时时相应用户的操作（哪怕只是提示计算中）。

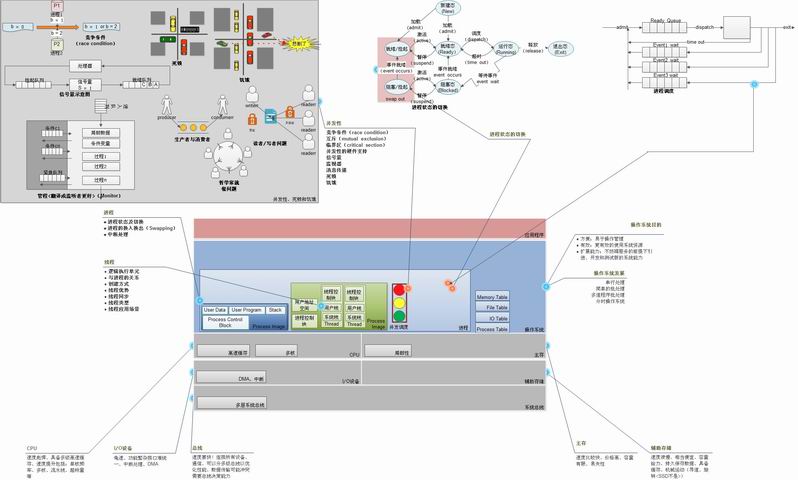
l 异步计算：比如应对电路故障的实时备份功能，通过线程无论是在代码量上还是开销上都要比你编写定时调度的功能要高效。

l 速度敏感的运算：多线程的计算机可以在计算一组数据的同时读入下一组数据（当然弄几个进程干这事儿也可以，但是开销明显更大）。

l 模块化的程序架构：对于包含多个活动或者多个源头和目标的输入输出程序，也许更容易使用多线程来设计和实现（就是每个线程干一摊子事儿，大家数据在一起自己取谁也别干涉谁）。

l 另外的比如Java程序，每个程序就是一个JVM的进程，至于这里面你起多少线程操作系统是不关心的。

**并发性**

[](http://static.oschina.net/uploads/space/2013/1228/121714_KQFH_870846.jpg)

**竞争条件**

竞争条件发生在多进程或者多线程同时修改同一个数据项的情况下，这时数据的最终结果依赖于各个进程（线程）执行的顺序（也就是结果不是唯一确定的了，比如同时修改变量b，P1执行b = 1， P2执行b = 2结果有竞争失败者决定）。

**临界区**

类似于b的这种资源称为临界资源（critical resource），程序中操作临界资源的程序段称为临界区（critical section）。就是说事儿肯定是出在临界区里的。

**互斥（multual exclusion）**

多个进程尝试进入同一个不可共享的资源的时候进程间需要是互斥的。这个不可共享的资源就是上面说的临界资源，进入的程序段即临界区。

互斥的要求：

l 互斥是系统强制的。

l 在临界区外挂起的进程不能够干涉其他进程。

l 请求临界资源的进程不能被无限期的延迟，即不能有死锁。

l 当没有进程处于临界区时，对临界资源的请求必须立即分配。

l 互斥不能建立在任何关于进程相对速度或执行顺序的假设上。

l 一个进程在临界区的时间应该是有限的。

**进程交互**

进程交互分三种情况：

l 进程间互不相认：比如多道程序设计，这些进程间存在共享资源但是彼此不知道，操作系统需要保证进程间的安全。

l 进程间简介知道彼此：进程不知道彼此的ID，但是知道存在共享资源，进程间表现为合作。

l 进程间直接知道彼此：进程知道彼此的ID，存在共享资源，进程间表现为合作。

**并发性的硬件支持**

**Interrupt disable**

进程声明自己的某一段程序是不可被中断的，这招只在单个处理器的情况下有用，因为多个处理器则存在进程并行运行。

**Compare & swap instruction**

由CPU提供的原子指令，用测试值（test value）检查内存单元（\*word），如果相等就用给定的值设置内存单元（newvalue），最终返回替换前的内存单元值。

使用：内存单元的初始值是0，多个进程执行用0去测试并替换为1的指令，只有获取到0的返回值的进程获得了进入临界区的资格，在离开临界区前进程要重置内存单元值为0。

**Exchange Instruction**

有CPU提供的原子指令，用内存区域的值和一个寄存器的值交换。也就是只有换到寄存器初始值（换完以后检查内存区域的值）的那个进程可以进入临界区并在离开时重置寄存器。

**硬件支持存在的弊端：**

1，采用的是忙等待（busy waiting）的方式，CPU一直在进程间切换，效率低。

2，可能发生饥饿：总有一个二活人品差，抢不到而又没有任何办法干预。

3，可能存在死锁：P1获得了临界资源但是同时P1被P2中断了（P2优先级高），P2却无法获得被P1占有的临界资源，P1同时得不到CPU的计算周期。

**并发性的软件支持**

**信号量（semaphore）**

用于进程间传递信号的一个整数值。提供了三种操作初始化、信号加和信号减。只有0和1的信号量称为二元信号量。减操作semwait用于阻塞一个进程（当信号量为0的时候阻塞），加操作semsignal用于激活被阻塞的进程。、

**生产者与消费者**

生产者负责生产资源并加入到指定的缓存中，加入过程如果缓存已满要阻塞生产者；消费者负责消费资源，如果缓存已空要避免消费者消费不存在的资源。

const int bufferSize = n1;

semaphore s = 1, n = 0, e = bufferSize;

producer

{

while(true)

{

produce();

semwait(e);

semwait(s);//s信号量用于控制每次只有一个进入critical section

append();

semsignal(s);

semsignal(n);

}

}

consumer

{

while(true)

{

semwait(n);

semwait(s);

taken();

semsignal(s);

semsignal(e);

consume();

}

}

**监听者（管程）**

信号量的麻烦在于分布在程序的各个地方，一处错误就可能导致并发同步的错误。监听者提供了更完善的机制用于并发同步。参考监听者子图。

监听者包括局部数据、条件变量和若干过程和等待队列等，局部变量是被监听者机制保存的处于外部进程不能访问或影响局部变量。

进程可以通过调用监听者的某一过程来进入监听者，但是同一时间只有一个进程处于监听者中（其它的在外面排队，也就是进入队列）。

监听者包含若干个条件变量，可以视条件变量为指向某一等待队列的指针。

监听者提供了cwait和csignal方法，调用cwait(c)将在条件变量c上阻塞当前进程并使监听者可用，当前进程加入到条件变量c的等待队列，调用csignal(c)将激活条件变量c的等待队列上的进程并重新尝试获得监听者（一般是激活一个，也有可能是signalAll<对于条件变量不明确的情况激活所有的进程，使正确的进程运行其它的进程则因为未获得条件变量再次阻塞>）。

除此监听者还提供了紧急队列（也可能没有），对于进入到过程中的但最后没有调用csignal的进程（它可能是在过程中阻塞了或者怎么完蛋了）有两种处理方式：1，踢出去重新回到进入队列和大家竞争。2，考虑到这个进程已经执行了过程的部分逻辑有必要把它加入到紧急队列中，监听者会在空闲后重新激活紧急队列中的进程。

注意与信号量不同的是，在调用csignal(c)的时候如果条件变量c的等待队列上没有任何进程，那这个动作将不产生任何效果也不会累计下去。

生产者/消费者问题中Monitor的实现：

monitor

{

int size, nextin, nextout

appand(node)

{

if(nextin == size) cwait(notFull);

buffer[nextin] = node;

nextin = (nextin + 1) % size;

count++;

csignal(notEmpty);

}

take(char x)

{

if(count == 0)cwait(notEmpty);

x = buffer[nextout];

nextout = (next + 1) % size;

count--;

csignal(notFull);

}

}

**消息传递**

消息传递是进程间通信的另一个手段，其一个优点是除了进程间还可以适应分布式、共享的系统。消息传递最典型的两个原语：send(source, message)和receive(destination, message)，其中可以是阻塞send、阻塞receive的也可以是不阻塞send阻塞receive的或者两者都不阻塞。

消息的组成包括消息头和消息体，消息头包括目的地Id、消息格式、源Id、消息长度等信息，消息体则是消息内容。

寻址：消息可以是一对一的也可以是一对多、多对一或者多对多。

消息的排队原则：默认情况下消息的接收应该是先进先出的对了，除此还要考虑紧急程度的优先级设置。

生产者消费者的消息实现：

producer

{

while(true)

{

receive(mayproduce, pmsg);

pmsg = produce();

send(mayconsume, pmsg);

}

}

consumer

{

while(true)

{

receive(mayconsume, pmsg);

pmsg = consume();

send(mayproduce, pmsg);

}

}

main()

{

create\_messagebox(mayconsume);

create\_messagebox(mayproduce);

for(int i = 0; i < N; i++) send(mayproduce, null);

parbegin(producer, consumer);

}

**读者和写者问题**

读者/写者问题不同于生产者消费者，一个资源可以同时有多个读者但是同一时间只能有一个写者。写者和读者不能同时获取资源。

可以存在两种类型的锁

1，读者优先：文件未被占用或存在读者时可以继续加入读者。

2，写者优先：文件被读者占用后一旦出现写者后续不能加入读者。

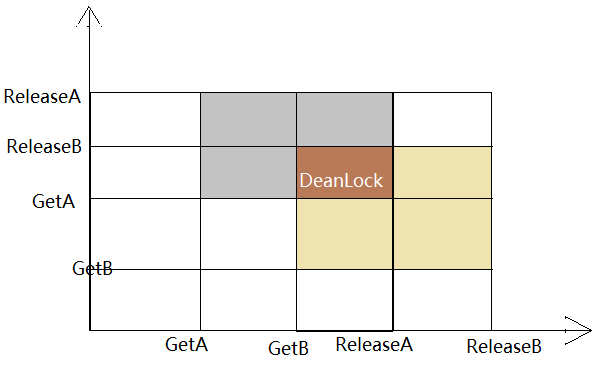
可以基于信号量或者消息发送来解决，个人觉得能通过信号量的一定能通过Monitor解决。

**死锁和饥饿**

死锁：两个或多个进程间都需要几个临界资源，但是各个进程持有其中一个临界资源而尝试获取另外的临界资源。

饥饿：可能是优先级或者调度算法本身的原因导致某个进程始终无法获取到临界资源（竞争激烈），虽然不存在死锁但是这个进程做不了任何事情。

**联合进程图（Join Process Diagram）**

[](http://static.oschina.net/uploads/space/2013/1228/135927_bTNl_870846.png)

**资源类型**

可重用资源：比如I/O设备、文件等等，它们在同一时间只可以被一个进程使用，但是使用之后并不会因此而销毁。

可消耗资源：比如存在I/O中的一段流数据，一旦被使用就不在存在了，但是可以被制造出来。除此以外还有信号、中断、消息等等。

**死锁形成的条件**

1．互斥

2．不存在抢占

3．持有和等待

4．形成了等待的环路

1~3：有可能产生死锁但是并没死锁。只有4发生的时候死锁才是真的产生了。

**死锁预防（protection）**

死锁的预防不同于死锁避免，死锁预防的目的在于干涉死锁形成的条件1~4使得死锁无法形成，往往导致的成本很高并且不很实用。

1．互斥：无法消除，有并发就有互斥。

2．抢占：这个可以做到有几种途径：a，当进程资源请求被拒绝时强制其释放已占有的资源，在后续需要时可以重新申请。b，当进程申请已被其它进程占有的资源时系统允许其抢占。这两种方法都有一个大前提就是资源状态必须是容易保存和恢复的，否则就啥都没了。

3．持有和等待：可以让进程一次申请所有需要的资源，这将不会出现持有并等待的情况。但是这是在进程能够预测它所使用的所有资源的前提下才成立的，并且效率很低。进程会在没有用到资源前先占用资源。

4．形成环路：可以将各个资源类型定义成线性顺序，只有占有了前面资源的进程才能进一步申请后续资源——同样效率是硬伤。

**死锁避免（avoidance）**

死锁的避免是运行时发现进程的启动或者请求会导致死锁，从而采取不启动或阻塞的措施。

1，如果进程的请求导致死锁，则不启动进程。这个比较难做到，因为它要求进程提前预知自己要用到的所有资源。

2，如果进程请求的资源可能导致死锁，则不分配资源给进程（塞你一会儿）。这个是更可行的办法。

方法2的运算如下：

OS中始终维护下列数据：

1，矩阵C(claim)为各个进程声明的需要用到资源。

2，矩阵A(allocation)现在以分配给各个进程的情况。

3，向量R当前系统拥有的资源

当一个进程申请起源是，OS假设分配资源给它并更下上面的数据，之后查看是否会产生死锁：

1，C - A得到矩阵P描述每个进程顺利完成时要得到的剩余资源。

2，向量R - A中各个资源的合计值等到向量V当前可用的资源。

3，如果向量V中的资源不足以使P中任何一个进程得到满足，那么即将发生死锁，这时候拒绝进程的请求。

4，如果存在可完成的进程，把进程的资源加入到向量V中重复3步骤直到所有进程可完成或出现死锁。

|  |  |  |  |
| --- | --- | --- | --- |
|  | R1 | R2 | R3 |
| P1 | 3 | 2 | 2 |
| P2 | 6 | 1 | 3 |
| P3 | 3 | 1 | 4 |
| P4 | 4 | 2 | 2 |

矩阵C

|  |  |  |  |
| --- | --- | --- | --- |
|  | R1 | R2 | R3 |
| P1 | 1 | 0 | 0 |
| P2 | 6 | 1 | 2 |
| P3 | 2 | 1 | 1 |
| P4 | 0 | 0 | 2 |

矩阵A

|  |  |  |  |
| --- | --- | --- | --- |
|  | R1 | R2 | R3 |
| P1 | 2 | 2 | 2 |
| P2 | 0 | 0 | 1 |
| P3 | 1 | 0 | 3 |
| P4 | 4 | 2 | 0 |

矩阵C - A

|  |  |  |
| --- | --- | --- |
| R1 | R2 | R3 |
| 9 | 3 | 6 |

系统资源向量R

|  |  |  |
| --- | --- | --- |
| R1 | R2 | R3 |
| 0 | 1 | 1 |

当前可用资源向量V

由于当前可用资源可以满足P2，所以可以加回P2的资源并重复步骤3。

**死锁的检测和恢复**

死锁的检测其实和上面的步骤是一样的需要两个矩阵：Q——进程请求资源（是排除已分配资源后）、A——进程已经分配的资源，一个向量V当前可用资源。

1，标记A中所有资源占用都为0的进程行。

2，初始化临时的向量W是它等于V。

3，查找Q中为标记的i行，其中i行小于向量W。如果找不到i则种植算法。

4，如果找到i行，将i标记并把A中i行数据加回到W中。重复步骤3。

当算法结束时如果存在为标记的行则已经发生死锁。

死锁的恢复有点野蛮：

1，取消所有死锁的进程。这是现代操作系统最常用的办法。

2，回滚进程到之前一个检查点（checkpoint），重新启动。操作系统需要具备回滚和恢复的能力，这样仍然有死锁的风险，但是并发的不确定性能保证死锁最终不再发生。

3，连续取消死锁进程直到最终不再出现死锁。取消进程的顺序依赖某种成本的原则，取消后重新调用死锁检测，如果仍然存在死锁继续取消。

4，连续抢占资源直到死锁不再发生。同样基于某些成本选择的方法，资源抢占后重新调用死锁检测，一个被抢占资源的进程需要回滚到获取该资源前的检查点。

3、4步骤可以考虑一下原则：

l 目前消耗处理器时间最少。

l 目前为止产生的输出最少

l 预计剩下的时间最长。

l 目前位置分配的资源总量最少。

l 优先级最低。

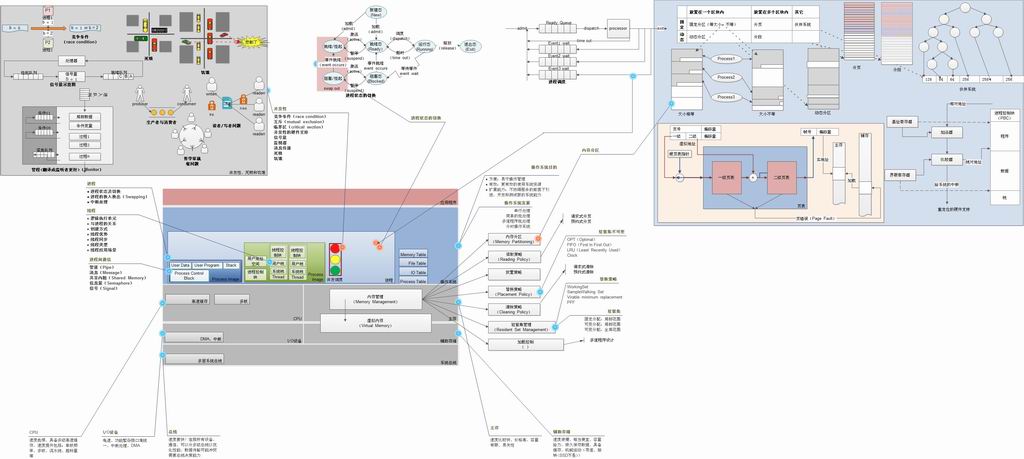
**哲学家就餐问题**

一个屋子里有5个哲学家他们一天中除了睡觉只做思考和吃饭两件事情，屋子里有一个桌子提供了哲学家喜欢的意大利粉，但是由于哲学家每天至思考导致身体退化他们需要用两个叉子才能够进食，他们坐下后会先拿起左手的叉子，然后拿起右手的叉子，之后进食吃饱以后放下两把叉子。桌子周围一共提供了五把椅子在每个椅子间提供了一把叉子，请为哲学家设计吃饭的算法。

如果5个哲学家同时饿了那么每个人都会拿到左手的叉子而死锁。

哲学家吃饭问题可以通过信号量或者Monitor来处理。

**内存管理**

[](http://static.oschina.net/uploads/space/2013/1228/124400_zAnc_870846.jpg)

内存管理的目的：

l 重定位

l 保护

l 共享

l 逻辑组织

l 物理组织

**内存分区**

**固定分区**

分为大小相等和大小不等的固定分区策略。内存预先被划分为固定大小的内存区域，进程可以安装到大于等于自身的内存区域中。

存在内部碎片、大于最大内存区域的进程无法加载、内存利用不充分。

**动态分区**

动态的根据进程大小进行内存区域划分，从而使得每个进程可以装进和自己大小相等的内存区域。

存在外部碎片、需要定时压缩外部碎片否则内存被割裂成很多小区域。

**伙伴系统**

采用二分法把内存等大小的两块区域（2^1），再将其中一块区域继续二分为2^2层，逐次分配下去直到进程所需的大小M在2^(N+1) < M <2^（N)时保存进程。在进程释放后再进行合并为较大的块。

伙伴系统弥补了固定分区和动态分区的缺点，但是分页和分段提供了更高级的分区方式。

**重定位**

因为进程换入换出后不一定仍加载到原来的内存位置，所以在程序中不可能确切的写出实际的内存地址，而是通过偏移量来描述位置。

**分页**

内存被划分成许多等大小的帧，程序被划分成与帧大小相等的若干页。进程加载时需要将所有页加载到内存中不一定连续的帧中。系统维护了页表描述程序占用的页和空闲页表。

分页没有外部碎片，由于每个帧很小只有最后一帧是可能存在内部碎片的，所以只会出现很小的内部碎片。

页的大小将直接影响性能。页太小时：页数多开始时页面错误率很高，一段时间后由于页面都已加入趋于平稳，但是过小的页将使每次读写的区块很小。页增大时：每一页所包含的单元和相关单元越来越远，局部性被削弱错误率增加，不断增大时错误率又减小当页大小等于进程P时不再有也错误，整个进程都在内存中。由此导致的问题是主存中能存入的进程越来越少。

**页表**

操作系统为每个进程维护一个页表描述进程中页与帧的对应，逻辑地址分为了页号和偏移量两部分。一般情况下页表的大小位页的大小，页表中每条记录称为页表实体（PTE，page table entry）。

页表可以是多级页表，受制于页大小的限制页表的大小不能大于一页（也不可能把巨大的页表存放在主存中），因此页表做多级处理，根页表始终在主存中，当次级页表不在主存中时从辅存加载对应的页表进主存。

**根据页表寻址**

l 根据虚拟地址的页号查找根页表对应的帧号。

l 帧号+次级页号合成次级页表的地址找到对应的主存中帧号（如果存在的话）。

l 帧号+偏移量获得实地址。

l 如果页表中页表实体的标志位标志当前页没有加载到主存中，发生页错误中断交给操作系统加载，进程被中断处于Block状态。

**反向页表（Inverter Page Table）**

这是另一个可行的从虚地址获取实地址的方案。

针对虚拟内存中页表大小和虚拟地址范围成比例的缺点（太大了），反向页表大小是固定的取决于主存中实际帧数。反响列表对虚拟地址的页号做Hash运算取得Hash值，每一个Hash值对应一个数据项。数据项中记录了进程ID和主存中帧号，通过帧号和偏移量就可以得到实地址了。由于多个虚拟地址可能映射到同一个Hash值，反向页表需要维持链结构（当前项连接到同值的其它项），所以进程ID和Hash值共同决定了虚拟地址对应的数据项。

反响的含义是指使用帧号而不是页号来索引页表项。

**转移后备缓冲器（TLB，Translation Lookside Buffer）**

这是（虚拟内存地址转换）硬件设备的一部分，相当于高速缓存。因为正常情况下虚拟地址的访问要经过两次主存——一次查找帧地址，一次取数据，转移后备缓冲器缓存了页号和帧地址的对应关系，只有在未命中的情况下采取访问页表查找帧号。

**分段**

简单分段类似于简单分页，是将主存换分成大小不等的若干段，一个进程可以存储在不连续的段中。分段的大小受限于分段最大长度。分段对程序员是可见的，它具有处理不断增长的数据结构的能力以及支持共享和保护的能力。操作系统为每个进程维护一个段表。

分段存在外部碎片。

**段页结合**

段页结构是将程序划分成段，每段保存在主存中对应的段里，在主存中段是由等大小的多个页组成，当段最后不足一个页时占用一个页。

段页结合的方式结合了分段和分页的长处。在分段系统中由于每段有长度属性，避免了程序不经意的越界访问。进程间存在共享时多个进程可能持有同一个段的引用，页结构也可以是实现类似的结构。

**虚拟内存**

当程序太大超过主存时就需要虚拟内存了，另外多道程序设计希望同时有尽可能多的进程加载到内存中，由于局部性的原理进程不需要全部加载进内存而是加载频繁使用的区块（称为驻留集），进程的其它部分保存在专门的辅存区域中。这个机制称为虚拟内存。

**系统抖动**

当系统换出一块内存区域后紧接着它有需要调用它，当这种情况频繁发生的时候就导致了系统抖动。

读取策略

分为请求式分页和预约式分页。请求式分页是指当确实需要访问到某页时才把它加载进主存，预约式分页是利用局部性原理将可能访问到的当前请求的后续页面加载到主存中。显然预约式分页更可取，但是会造成一定的浪费，在首次加载页面和发生页错误时适合采取预约式分页。

放置策略

替换策略

替换策略是在加载新的页时主存已满需要替换出页时决定那些页将被替换的算法。

最佳（OPT，Optimal）

最少发生页错误的算法，是优先替换那些距离访问最远的页面，需要预知页的请求顺序，本身是不可能实现的，但是可以作为参考比对其它策略。

最近最少使用

实际上是性能上最接近最佳的，但是仍难以实现。一种实现方式是给每个页定义最近访问的时间标签，并在每次访问后更新标签，即使通过硬件支持成本和开销仍然很高。

先进先出

依赖的理论是一个在很久以前加载的页现在很可能已经不再访问了，但是结果往往并非如此，这种策略很容易实现。

时钟

环形的结构，通过指针指示当前所在位置，每个页有一个使用为在访问后标记其值为1。在需要替换时移动指针找到第一个标记位等于0的页，指针每移动过一个页就将这个页的使用位清零。这样如果没有使用位为0的页，当移动一圈以后最初的位置就会被清理。

一个改进是增加修改位——这也是必须的因为被修改的内存必然要写入辅存。现在有两个指示位（访问u，修改m），算法如下：

1，查找u=0,m=0的页，如果存在替换。这一步不清零访问位。

2，如果1失败，查找u=0，m=1的页，并且这个过程中把访问位清零。

3，如果2失败，重复1，必要时重复2。

这样好处是尽量不去替换发生数据修改的页，少了写回辅存的动作。

**页缓冲**

也是避免系统抖动的一个策略，在主存中划分出一定的缓存，用于保存那些被替换出的页，根据局部性原理他们很可能近期会被访问到。这个缓存是一个先入先出的队列，一般页面被访问则直接从缓存中取回，如果一直没有被访问则最终会被挤出缓存。

驻留集管理（Resident Set）

虚拟内存中并非一个进程的所有部分都加载到主存中，程序运行过程中常驻内存的部分称为驻留集。

驻留集的讨论包括两部分：驻留集分配（大小）和替换范围。其中驻留集不可变的情况下替换算法已经在替换侧路讨论过了，驻留集可变的情况下将有所不同。

固定分配，局部范围：每个进程的驻留集大小固定，每个进程一旦有一个页换进来就要有一个页换出去。需要根据程序的类型、优先级等预先分配固定的大小，一旦过小则会保持较高的也错误率，如果较大则会占用资源，系统运行缓慢。

动态分配，全局范围：根据进程运行的状态调整驻留集的大小，如果页错误率较高就适当加大驻留集，如果进程错误率一直保持较低水平说明程序的驻留集足够大，可以适当削减一些也不会危及程序。全局范围则可在所有进程间选择要被替换出的页，难点在于没办法确定该从哪里换出页（即很可能换了不该换的），解决办法是页缓冲。

动态分配，局部范围：动态分配的效果同上，同时限制在进程自己的范围内换出页。在这个策略中要求对增加或减少驻留集大小进行评估并且要预估将来进程的情况，因此比简单全局替换要复杂得多，但会提供更好的性能。

**驻留集的替换策略**

**工作集策略（Working Set）**

W（t，r）是关于t和r的函数，t是单位时间，r是窗口的宽度定义最近的r个单位时间中用到的页的集合。即始终有W(t, r+1) 包含 W(t, r)。

工作集如下工作：

1，监视每个进程的工作集。

2，周期性的从一个进程的驻留集中移除那些不在工作集中的页，可以使用LRU策略。

3，只有当一个进程的工作集在主存时才可以执行该进程（也就是驻留集包含了工作集）。

首先工作集是有效的，它利用率局部性原理，并未该原理设计了一个可以减少页错误的内存管理策略。遗憾的是工作集存在很多问题：

1，现在不总是代表未来。

2，开销太大，实现监视每个进程不现实。

3，r最优值未知。

但是工作集提供了参考，以下方案都是基于工作集思想的：

**页错误频率（Page Fault Frequency，PFF）**

主存中每个页都有一个使用位（和时钟的一样作用），操作系统记录每个进程上一次页错误的时间戳，如果发生了页错误查看两次页错误的时间间隔如果小于阈值F，则加入新的页（扩大驻留集），否则清除所有使用位为0的页，并把当前驻留集的页使用位清零（缩小驻留集）。一个改进时加入使用两个阈值——一个是驻留集增加的最高阈值，一个是驻留集减小的最低阈值（指频率）。

页面错误频率是工作集的一个折衷，使用页缓冲则可以达到相当好的效率。但是一个缺点是如果要转移到新的局部性，会导致暂时的驻留集猛增。转移到新的局部性是指，进程运行一段时间会切换到不同的逻辑指令部分，这时候局部性发生偏移，之前的驻留集不再需要。

**采样间隔可变工作集（Variable-interval Simple Working Set，VSWS）**

针对PFF在内存高峰是来不及淘汰就得驻留集而导致进程频繁换入换出等开销（频率不够快），采用动态的采样率以期望尽快淘汰不需要的页。

采用三个参数：L采样区间最长宽度、M采样区间最短快读、Q采样区间允许发生的页错误数。VSWS和PFF一样使用了页的使用位，不同之处在于频率的调整：

1，如果采样时间超过了最长时间L，挂起进程并扫描使用位。

2，未达到最长时间，但是页错误数超过了Q：

2.1，如果采样间隔超过了M则挂起进程并扫描使用位。

2.2，如果采样间隔没有超过M，则一直等待采样时间到达M进行扫描。

清除策略

清除策略目的在于确定何时讲一个被修改的页写回辅存，分为请求式清除和预约式清除。

请求式清除采取动作的时间比较慢，影响性能。

预约式清除把需要修改的页在需要用到它们的页帧之前批量的写回辅存。但是预约式清除写回的页在替换策略确定移除它之前仍有驻留在主存中，这期间可能再次发生修改导致清除无意义。

改进办法是使用页缓冲，分为两个表存储替换策略淘汰的页。一个表寸修改的页，一个表存未修改的页，修改表中的页被批量的写回辅存，然后移动到未修改表中，未修改的表准备被挤出或者拿回。

加载控制

加载控制是关注多道程序设计的控制，系统中多道程序的数量称为多道程序设计级。当多道程序数量过少时系统会比较空闲，过多时则会导致较高的页错误率和抖动。

一个方案称为L=S，L是页错误间隔的平均时间，S是页错误的平均处理时间，实验表明当两者接近时处理器使用率达到最大。

另一个方案是50%方案，即始终报纸分页设备的使用率保持在50%左右，此时的处理器使用率也是最大。

另外策略是监视时钟（Clock）替换策略的环形缓存区指针移动速度，当移动速度低于某个阈值时可能是以下两种情况之一：

1，很少发生页错误，指针很少移动。

2，未被使用的驻留页太多，指针不需要移动太多就可以找到可清除页。

这两种情况都表明进程的驻留集分配太大，可以增加多道程序设计级。另外还可以包括一个高阈值，如果指针移动速度超过这个阈值，则表明多道程序设计级太高，导致频繁的页错误，则要挂起进程以减少多道程序设计级。挂起进程的策略可以是：

l 最低优先级

l 正在发生页错误的进程（Faulting process）：原因是该进程很肯能驻留集还没有形成，因此暂停它的代价很低。

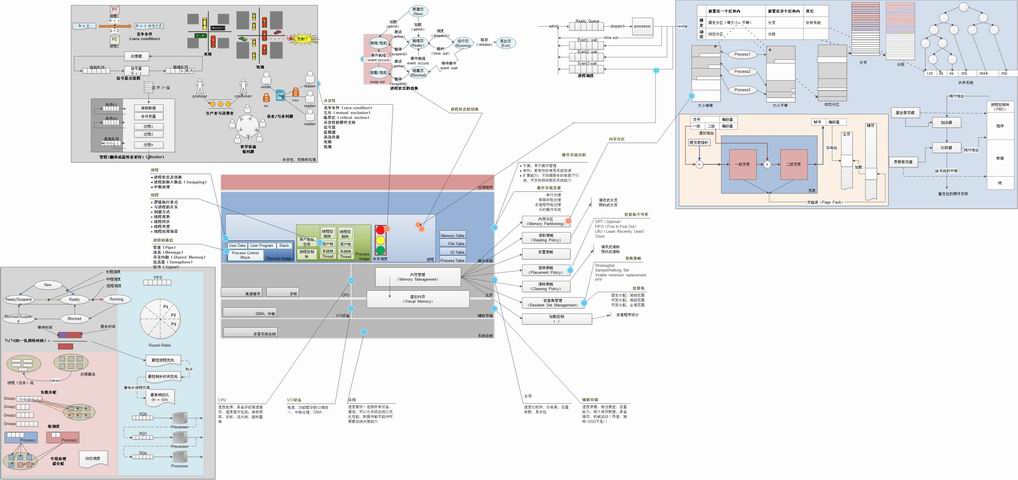
l 最后被激活的进程：同样很可能有最小的驻留集。

l 拥有最小驻留集的进程：重新装入的代价最小，但是不利于驻留集较小的程序。

l 最大的进程：释放较大的空间，不用很快又去取活其它的进程。

l 具有最大剩余执行窗口的进程：类似于最少剩余时间策略，把已运行时间最短的挂起。

**处理器调度**

[](http://static.oschina.net/uploads/space/2013/1228/131140_UYWm_870846.jpg)

**单处理器调度**

**调度类型**

长程调度：决定是否将任务加入到待执行的进程池中。

中程调度：决定进程的部分或全部加入到内存中（从Suspended到Ready）。

短程调度：决定哪一个就绪的进程将被执行。

I/O调度：决定哪一个挂起的I/O请求将被可用的I/O设备执行。

处理器调度关注的是短程调度。

**调度准则**

**面向用户**

响应时间：从任务提交到开始有响应输出的时间，一般的当处理器开始处理任务时即开始有输出。

周转时间：从任务提交到任务完成的时间。

最后期限：当可以指定最后期限时，操作系统降低其它目标，是满足最后期限的作业数目的百分比达到最高。

可预测性：无论系统当前负载情况是繁重还是空闲，一个给定工作完成的总时间和总代价应该是相等的。

**面向系统**

吞吐量：单位时间内完成的进程数。

处理器利用率：处理器处于忙状态的时间百分比。对于昂贵的共享系统来说这是一个重要的准则，对于个人系统则显得不那么重要。

公平性：没有来自用户或其它系统的指导时操作系统应该公平的对待所有进程，没有一个进程会处于饥饿状态。

强制优先级：当指定了优先级后调度策略应优先响应高优先级的进程。

平衡资源：调度策略应是系统的所有资源保持忙碌的状态，较少使用紧缺系统资源的进程应该得到照顾。

**调度算法**

归一化周转时间：它是指进程周转时间与服务时间的比率，最小值是1.0.该值表示进程的相对延迟，典型的进程的执行时间越长可容忍的延迟越长。

先来先服务（FCFS）：没有优先级和抢占，所有进程按照加入的顺序执行。这样的调度偏向于执行时间长的进程。FCFS策略本身对操作系统不是很有吸引力，但是它经常和优先级系统配合使用产生有价值的调度策略。

轮转（Round Robin）：对处理器资源进行时间分片，依次分配相同的时间资源给每个就绪的进程。轮转的时间片最好略大于一次典型交互的时间，当时间片足够大时退化为FCFS策略。

轮转增加了处理时钟中断、进程切换和分配函数的开销，因此时间片不应该过短。该策略对短执行时间的进程有所改善，但是一个问题是进程分为I/O密集型和处理器密集型，由于I/O密集型进程经常被I/O中断，所以轮转策略倾向于处理器密集型。

一个改善的策略是虚拟轮转法（Virtual Round Robin），它增加了一个I/O队列，当一个进程被I/O阻塞后它加入到I/O队列，在就绪后它I/O队列有高于就绪队列的优先级，该进程后续的执行时间与已执行时间的和不会超过时间片。

最短进程优先（SPN，shortest process Next）：具有最短执行时间的进程有更高的优先级，它依赖于系统估计进程的执行时间，在批处理系统中任务加入时程序员给出任务的估计时间，如果任务执行时间远远超过给出的估计时间它将被废弃。在生产环境中有大量的重复任务，系统将监控每次任务执行的时间以估计执行时间，最简单的公式是s=(各次时间和)/n，一个避免每次求和的优化是s=S(n-1) + Tn/n，上述公式每次执行的权值是相同的，典型情况下我们希望最近的执行情况有更大的权值Sn+1 = aTn + (1-a)Sn。

最少剩余时间（SRT，shortest remain time）：是SPN的改进版本增加了抢占机制，在不断有短进程加入的情况下长进程可能处于饥饿状态。

最高相应比优先（HRRN，Highest Response Rapid Next）：使用公式(w+s)/s，其中w是等待时间，s是服务时间。操作系统始终选择具有最高相应比的进程，同样需要估计和记录进程的服务时间。该策略非常具有吸引力，当偏向短进程时，长进程由于得不到处理响应比不断增加。

反馈：不想SPN、STR、HRRN策略那样需要估计时间，反馈策略倾向于短进程，它具备多个队列Q1~Qn，当进程加入时处于Q1队列中，采用FCFS的顺序服务队列中的每个进程，当进程允许超过时间阈值时中断并加入到下一级队列中Q2，依次类推。Q1具有最高的优先级，只有Q1中不存在进程是才执行下一级的队列。这样可能导致的问题是过长的队列可能加入到Qn队列后处于饥饿状态，因此要见识进程的等待时间，如果超过一定长度则重新调入到Q1中。

**多处理器调度**

**多处理器调度关注的类型**

**粒度**

无约束并行性：进程之间没有显示的同步，每个进程都是一个单独的程序。无约束并行可能达到每个用户都像是使用单独的计算机或工作站。

粗粒度和非常粗粒度并行性：进程之间存在这同步，但是在一个非常粗浅的级别上，这种情况可以简单的处理成一组运行在多道程序单处理器的并发进程，在多处理器系统是允许的软件进行很少或者不进行改动就可以支持。

中等粒度并行性：应用程序可以被进程中一组线程有效的实现，这种情况下程序员需要显示的指定潜在的同步性，应用程序之间需要高程度的合作与交互。

细粒度并行性：代表着比线程间更加复杂的同步情况，迄今为止仍是特殊的未被分割的领域。

**进程调度**

集中调度or对等调度：

集中调度即调度中存在主处理器，所有的任务分发由主处理器来完成，这种情况下主处理器可能成为系统的瓶颈。

单处理器使用多道程序设计：

与单处理器性能的不同：

多个处理器系统中由于一个长进程在单个处理器上执行时，其它的进程可以分配到另外的处理器上因此复杂的调度策略不再显得非常重要，调度策略的开销可能成为性能损失。FCFS策略变得可接受。

**线程调度**

负载分配：

提供全局队列，每个处理器只要空闲就从队列中取任务执行。

优点是：负载均匀的分配给处理器，不存在空闲的处理器；不需要集中调度；可以使用单处理的任何一种调度方案进行分配。

缺点是：中心队列占居了必须互斥访问的存储器区域，因此多处理器同时查找时会成为瓶颈，当只有几十个处理器时不是大问题，但是上百个处理器时就会出现瓶颈。被抢占的线程可能在不同的处理器上执行，如果每个处理器都配有高速缓存的话那么命中率将非常低。如果进程的所有线程都视为在公共线程池中那么进程的线程可能不会同时被处理，当线程间存在高度合作时则出现瓶颈。

组调度：

一组相关的线程基于一对一的原则，同时分配到一组处理器上去。

优点：如果紧密相关的进程并行执行那么同步阻塞可能减少，从进程推广到线程组调度把一个进程所有的线程视为相关的；调度开销可能减少，因为进程内线程间可能相关如果一个线程在高速允许，而它以来的线程没有运行就会出现阻塞和调度。

缺点：组调度同一时间调度一个进程中相关线程，某些进程的特性可能至适应单线程允许将会出现其它处理器空闲的情况，解决办法是把若干单线程的进程视为一组允许。

专用处理器分配：

每个程序执行时被分配给一组处理器，处理器个数与进程的线程数相等，当进程执行完后处理器返回到处理器池中等待处理其它任务。这个策略看似是极端浪费的，它会等到进程运行结束才将处理器分配给其它的进程使用，而一旦一个线程被I/O阻塞执行它的处理器将空闲。

采用这个策略的原因：

1，高度并行的系统中有数十或者数百个处理，每个处理器只占系统总代价的一小部分，处理器利用率不再是衡量有效性或性能的重要因素。

2，一个程序的生命周期里避免进程切换会加快程序运行。

动态调度：

执行期间进程的线程数可以改变。某些应用程序可能提供了语言和系统工具允许动态的改变进程中的线程数目。提供了一种操作系统和应用程序共同进行调度决策的方法。

当一个作业请求一个或多个处理器时：

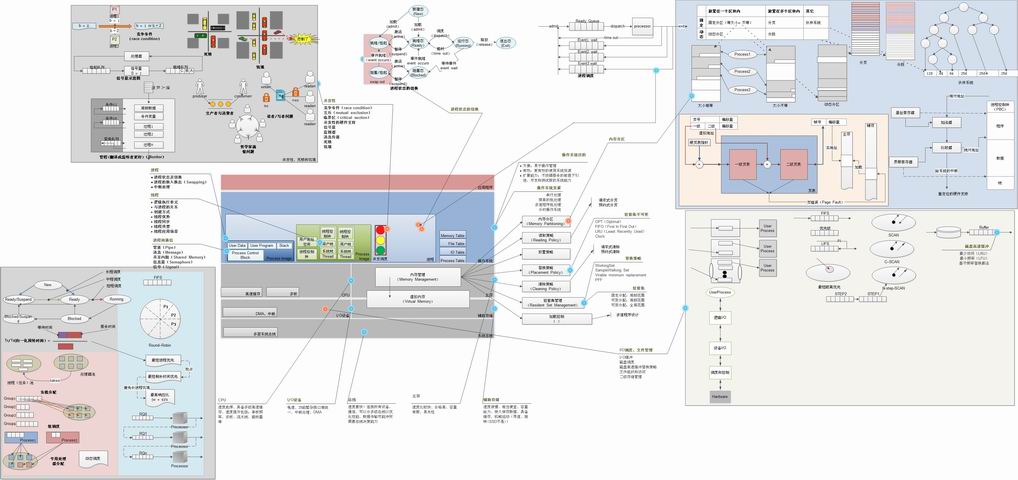
1，如果有空闲处理器分配满足它们需求。

2，否则如果作业新到达，从当前已分配多个处理器的作业中分出一个处理器给它。

3，如果都不满足那么作业处于未完成状态直到有空闲的处理器或者改作业废除它的请求。

4，当释放一个或多个处理器时为这些处理器扫描当前未满足的请求队列，给每个没有分配处理器的作业分配一个处理器，如果还有空闲处理器再次扫描队列，按照FCFS原则分配处理器。

**I/O设备调度**

[](http://static.oschina.net/uploads/space/2013/1228/132821_leSn_870846.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缓冲**

I/O设备划分：

I/O设备分为面向块（block oriented和面向流（stream oriented）的设备，面向块的设备将信息保存在块中，通常是固定大小的，传输过程中一次传送一块。面向流的设备以字节流的方式传输数据。

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。

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。

**磁盘调度**

**磁盘性能参数**

寻道时间：机械臂移动到数据所在轨道的时间，现在典型磁盘的寻道时间Ts=10ms。

旋转延迟：磁盘旋转到要读取的数据位置的延迟，一般取平均时间即1/2r，其中r表示转速。

传送时间：磁头读取数据所花费的时间b/(Nr)，b表示要读取的字节数，N表示磁道上总字节数，r表示转速。

**磁盘调度策略**

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。

当然如果有大量的请求就会导致最先的请求饿死。

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。

**RAID**

RAID是一组磁盘系统把它们看为一个单个的逻辑驱动器。

数据分布在物理驱动器阵列中

使用冗余的磁盘容量保存奇偶检验信息，保障一个磁盘失败时，数据具有可恢复性。

**磁盘高速缓冲**

高速缓存是系统从主存中划分的一块区域，利用了局部性原理保存最近访问的数据块，用于提高更好的磁盘性能。

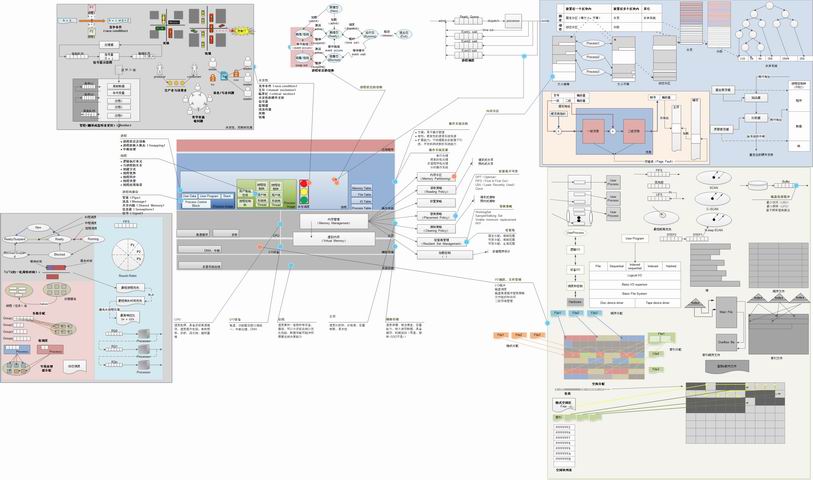
替换算法

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**

[](http://static.oschina.net/uploads/space/2013/1228/133736_B2tj_870846.jpg)

**文件系统软件结构**

基本文件系统：计算机与外部环境的接口，该层处理磁盘、磁带的交互数据。

基本I/O管理程序：负责所有文件I/O的初始和终止。

逻辑I/O：是用户和应用程序能够访问到记录。

访问方法层（堆、顺序文件、索引顺序文件、索引、散列）：距离用户和应用程序最近的层，在应用程序与保存数据的设备之间提供了标准接口。

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。

**记录组块**

磁盘中数据保存在块中，块越大每次传输的数据越多，效率越高。当时大的块要求操作系统提供更大或者复杂的缓存，并且由于局部性的关系大块中的数据可能是应用程序不需要的造成浪费。

固定组块

使用固定长度的记录，并且若干条完整记录保存到一个块中，每个块末尾可能有未使用的空间称为内部碎片。

可变长度跨越式组块

块的长度可变，记录可以跨越块保存，如果一个块的末尾空间不足一条记录时，剩下的数据可以保存在下一个块中，通过后继指针指明。造成了更复杂的处理，并且当读取跨越块的数据时需要读取两次，消除了内部碎片。

可变长度非跨越式组块

和上面相同，但是记录不可以跨越块保存。

**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：使用一个向量，向量的每一位代表一块磁盘，0表示空闲，1表示使用。优点是容易找到一块或一组连续的空间，问题是需要穷举来找到合适大小的区域，当磁盘剩余很少空间时这个问题尤为严重，因此需要设置特殊的数据结构辅助查找。如位表可以在逻辑上划分为不同的区域，建立区域汇总表统计每个区域的使用情况，查找空闲位时可以先找到合适的区域在查找位表中这部分区域的使用情况。

位表需要加载在主存中，一个位表所需要的存储总量为【（磁盘大小）/(8\*文件系统块大小）】（计算的是占用的字节数），因此一个16GB的磁盘，块大小位512字节时位表占用4MB，如果每次去数据都从硬盘加载4MB的位表的话这个速度是难以忍受的。

链式空闲区

空闲表可以使用指向每个空闲区的指针和他们的长度值被连接在一起，因为空闲表只需要一个指向第一个空闲区的指针，因此这种情况下空闲表的大小是可以忽略的。

这种分配法适合所有的文件分配方法，如果按块分配可以移动位于头部的指针，如果是按区域分配则可以顺序的找到合适的区域并更新指针。

存在的问题：1，使用一段时间磁盘会出现很多碎片，许多区变成只有一块大小。2，写时需要先读取该块以发现下一块的指针，如果进程请求大量的单个块这个效率是很差的，同意删除的时候也会导致很慢。

索引

索引是把空闲空间看做文件，使用之前介绍的索引表的方式记录。基于效率的原因，索引适合使用在大小可变的分区分配的情况下。

空闲块列表

每个空闲块都有一个顺序号，把顺序号保存在磁盘的一个保留区中。根据磁盘的大小，存储一个块号需要24位或32位。这是一种令人满意的方法，空闲块列表部分保存在主存里：

1，磁盘空闲块列表占用空间小于磁盘空间的1%。

2，尽管空闲块太大了，不能保存在主存。但是两种有效技术把该表的一小部分保存在主存里：

a，这个表可以是一个下推栈，栈中靠前的数千元素保存在内存中，当分配一个新块时它从栈顶弹出，同样当一个块被接触时把它压入栈中。只有栈中部分满了或者空了时候才需要从磁盘传输数据，通常情况下它是零访问时间的。

b，这个表可以看所FIFO的队列，分配时从头部取走，取消分配时从队尾入队，只有队空了或者满了时才需要与磁盘传输。