操作系统

- 操作系统特性，主要功能
  - 计算机的组成：**应用程序、操作系统、硬件。**
  - 硬件：CPU、储存器、输入/输出设备、总线等
  - 操作系统有四个特征
    - 并发
      - 两个或者多个事件在同一时间段内间隔发生，即程序在分时地交替执行。
    - 共享
      - 共享是指系统中的资源可供内存中多个并发执行的进程共同使用。
      - 1）互斥共享方式，如打印机，在一段时间内只允许一个进程访问该资源。
      - 2）同时共享方式，如磁盘
    - **并发和共享是操作系统的最基本特征**，互为依存。并发执行的要求引出了资源的共享；而资源共享的管理又直接影响到程序的并发执行。
    - 异步
      - 在多道程序环境下，允许多个程序并发执行。但由于资源有限，进程的执行不是一贯到底，而是走走停停，已不可预知的速度向前推进，这就是进程的异步性。
      - 异步性使得操作系统运行在一种随机的环境下，可能导致进程产生与时间有关的错误。
      - 但只要运行环境相同，操作系统必须保证多次运行进程，都获得相同的结果。
    - 虚拟
      - 虚拟是指把一个物理上的实体变为若干个逻辑上的对应物。（如通过分时系统）
  - 主要有以下管理功能
    - （1）作业管理：包括任务、界面管理、人机交互、图形界面、语音控制和虚拟现实等；
    - （2）文件管理：又称为信息管理；
    - （3）存储管理：实质是对存储“空间”的管理，主要指对主存的管理；
    - （4）设备管理：实质是对硬件设备的管理，其中包括对输入输出设备的分配、启动、完成和回收；
    - （5）进程管理：实质上是对处理机执行“时间”的管理，即如何将CPU真正合理地分配给每个任务。
- 并发并行
  - 并行是指两个或多个事件在**同一时刻**发生
  - 并发：同一时间段内多个事件**间隔**发生

- 阻塞，同步
  - IO操作分两个阶段
    - 1、等待数据准备好(读到内核缓存)
    - 2、将数据从内核读到用户空间(进程空间) 一般来说1花费的时间远远大于2。
    - 1上阻塞2上也阻塞的是同步阻塞IO
    - 1上非阻塞2阻塞的是同步非阻塞IO
    - 1上非阻塞2上非阻塞是异步非阻塞IO
  - 阻塞：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（内核IO操作彻底完成后，才返回到用户空间，执行用户的操作）
  - 非阻塞：指在不能立刻得到结果之前，该调用不会阻塞当前线程（用户程序不需要等待内核IO操作完成后）
  - 同步：当一个同步调用发出后，调用者要一直等待返回结果。通知后，才能进行后续的执行。
  - 异步：当一个异步过程调用发出后，调用者不能立刻得到返回结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。
- 进程，线程、协程
  - 进程：是系统进行资源分配和调度的一个基本单位，资源分配的最小单位
  - 线程：进程的一个实体（执行路径），也是 CPU 调度和分派的基本单位
  - 不同进程地址空间相互独立，同一进程内的线程共享同一地址空间。一个进程的线程在另一个进程内是不可见的；
  - 操作系统在分配资源时是把资源分配给进程的，但是CPU资源比较特殊，它是被分配到线程的，因为真正要占用CPU运行的是线程，所以也说线程是CPU分配的基本单位。
  - 在Java中，当我们启动main函数时其实就启动了一个JVM的进程，而main函数所在的线程就是这个进程中的一个线程，也称主线程。
  - 一个进程里包含什么?
    - 一个进程由三部分组成：程序、数据及进程控制块(PCB)。
    - 进程控制块是记录进程有关信息的一块主存，是进程存在的程序唯一标识。
  - 什么时候适合用线程/进程
    - 1、需要频繁创建销毁的优先使用线程；因为创建和销毁一个进程代价是很大的。
    - 2、线程的切换速度快，所以在需要大量计算，切换频繁时用线程，还有耗时的操作使用线程可提高应用程序的响应；
    - 3、因为对CPU系统的效率使用上线程更占优，所以可能要发展到多机分布的用进程，多核分布用线程；
    - 4、需要更稳定安全时，适合选择进程；需要速度时，选择线程更好。
    - 对比图

对比维度	多进程	多线程	总结
数据共享、同步	数据共享复杂，需要用IPC；数据是分开的，同步简单	因为共享进程数据，数据共享简单，但也是因为这个原因导致同步复杂	各有优势
内存、CPU	占用内存多，切换复杂，CPU利用率低	占用内存少，切换简单，CPU利用率高	线程占优
创建销毁、切换	创建销毁、切换复杂，速度慢	创建销毁、切换简单，速度很快	线程占优
编程、调试	编程简单，调试简单	编程复杂，调试复杂	进程占优
可靠性	进程间不会互相影响	一个线程挂掉将导致整个进程挂掉	进程占优
分布式	适应于多核、多机分布式；如果一台机器不够，扩展到多台机器比较简单	适应于多核分布式	进程占优

- 协程
  - 协程，又称微线程，纤程。英文名Coroutine。
  - 协程可以理解为用户级线程，协程和线程的区别是：线程是抢占式的调度，而协程是协同式的调度，协程避免了无意义的调度，由此可以提高性能，但也因此，程序员必须自己承担调度的责任，同时，协程也失去了标准线程使用多CPU的能力。
- 描述符、句柄
  - 句柄:是windows下概念,在linux/unix下没有句柄这一说法,在linux/unix下都是"描述符",是整形的
  - 句柄是操作系统在生成对象时分配给对象的唯一标识。 通过句柄可以获取操作系统提供的服务。最终目的都是用来定位打开的文件在内存中的位置.
  - 文件句柄定位到的是文件对象，而非文件。而文件对象是对这个文件的一些状态、属性的封装，例如读取到的文件位置等。
  - 文件描述符是一个简单的整数，用以标明每一个被进程所打开的文件和socket。第一个打开的文件是0，第二个是1，依此类推。linux 操作系统通常对每个进程l能打开的文件数量有一个限制。
  - linux系统默认的最大文件描述符限制是1024
- 进程间同步/互斥/通信的关系
  - 进程互斥、同步的概念
    - 进程互斥、同步的概念是并发进程下存在的概念，有了并发进程，就产生了资源的竞争与协作，从而就要通过进程的互斥、同步、通信来解决资源的竞争与协作问题。
  - 在多道程序设计系统中，同一时刻可能有许多进程，这些进程之间存在两种基本关系：竞争关系和协作关系。
  - 进程的互斥、同步、通信都是基于这两种基本关系而存在的。
    - 为了解决进程间竞争关系（间接制约关系）而引入进程互斥；
    - 为了解决进程间松散的协作关系(直接制约关系)而引入进程同步；

    - 为了解决进程间紧密的协作关系而引入进程通信。
- 第一种是竞争关系
    - 系统中的多个进程之间彼此无关，它们并不知道其他进程的存在，并且也不受其他进程执行的影响。例如，批处理系统中建立的多个用户进程， 分时系统中建立的多个终端进程。由于这些进程共用了一套计算机系统资源，因而， 必然要出现多个进程竞争资源的问题。当多个进程竞争共享硬设备、存储器、处理器 和文件等资源时，操作系统必须协调好进程对资源的争用。
- 资源竞争出现了两个控制问题：
    - 一个是死锁 （deadlock ）问题，一组进程如果都获得了部分资源，还想要得到其他进程所占有的资源，最终所有的进程将陷入死锁。
    - 另一个是饥饿（starvation ）问题，这是指这样一种情况：一个进程由于其他进程总是优先于它而被无限期拖延。
- 操作系统需要保证诸进程能互斥地访问临界资源，既要解决饥饿问题，又要解决死锁问题。
    - 进程的互斥（mutual exclusion ）是解决进程间竞争关系(间接制约关系) 的手段。 进程互斥指若干个进程要使用同一共享资源时，任何时刻最多允许一个进程去使用，其他要使用该资源的进程必须等待，直到占有资源的进程释放该资源。
- 第二种是协作关系
    - 某些进程为完成同一任务需要分工协作，由于合作的每一个进程都是独立地以不可预知的速度推进，这就需要相互协作的进程在某些协调点上协 调各自的工作。当合作进程中的一个到达协调点后，在尚未得到其伙伴进程发来的消息或信号之前应阻塞自己，直到其他合作进程发来协调信号或消息后方被唤醒并继续执行。这种协作进程之间相互等待对方消息或信号的协调关系称为进程同步。
    - 进程间的协作可以是双方不知道对方名字的间接协作，例如，通过共享访问一个缓冲区进行松散式协作；也可以是双方知道对方名字，直接通过通信机制进行紧密协作。允许进程协同工作有利于共享信息、有利于加快计算速度、有利于实现模块化程序设计。
- 进程的同步（Synchronization）是解决进程间协作关系(直接制约关系) 的手段。
    - 进程同步指两个以上进程基于某个条件来协调它们的活动。一个进程的执行依赖于另一个协作进程的消息或信号，当一个进程没有得到来自于另一个进程的消息或信号时则需等待，直到消息或信号到达才被唤醒。
    - 不难看出，进程互斥关系是一种特殊的进程同步关系，即逐次使用互斥共享资源，也是对进程使用资源次序上的一种协调。
- 进程通信的概念
    - 并发进程之间的交互必须满足两个基本要求：同步和通信。
    - 进程竞争资源时要实施互斥，互斥是一种特殊的同步，实质上需要解决好进程同步问题，进程同步是一种进程通信，通过修改信号量，进程之间可建立起联系，相互协调运行和协同工作。但是信号量与PV操作只能传递信号，没有传递数据的能力。有些情况下进程之间交换的信息量虽很少，例如，仅仅交换某个状态信息，但很多情况下进

程之间需要交换大批数据，例如，传送一批信息或整个文件，这可以通过一种新的通信机制来完成，进程之间互相交换信息的工作称之为进程通信IPC （InterProcess Communication）（主要是指大量数据的交换）。

- 进程间通信的方式很多，包括：
  - 1 mmap（文件映射）
  - 2 信号
  - 3 管道
  - 4 共享内存
  - 5 消息队列（重要）
  - 6 信号量集（与signal无关）
  - 7 网络（套接字）
- 进程同步的方法
  - 前面提到，进程互斥关系是一种特殊的进程同步关系，下面给出常见的进程同步的方法，实际上也可用于进程的互斥（个人理解）。
  - Linux 下常见的进程同步方法有：
  - 1、信号量
  - 2、管程
  - 3、 互斥量（基于共享内存的快速用户态 ）
  - 4、文件锁（通过 fcntl 设定，针对文件）
  - 针对线程（pthread）的还有 pthread_mutex 和 pthread_cond（条件变量）。
- 线程的同步方法：
  - 1、信号量
  - 2、互斥量
  - 3、临界区
  - 4、事件
- 同步机制：
  - 四种进程或线程同步互斥的控制方法
  - 1、临界区:通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。
  - 2、互斥量:为协调共同对一个共享资源的单独访问而设计的。
  - 3、信号量:为控制一个具有有限数量用户资源而设计。
  - 4、事 件:用来通知线程有一些事件已发生，从而启动后继任务的开始。
- 科普：
- 1.临界资源
  - 临界资源是一次仅允许一个进程使用的共享资源。各进程采取互斥的方式，实现共享的资源称作临界资源。属于临界资源的硬件有，打印机，磁带机等；软件有消息队

列，变量，数组，缓冲区等。诸进程间采取互斥方式，实现对这种资源的共享。

- 2.临界区：
  - 每个进程中访问临界资源的那段代码称为临界区（criticalsection），每次只允许一个进程进入临界区，进入后，不允许其他进程进入。不论是硬件临界资源还是软件临界资源，多个进程必须互斥的对它进行访问。多个进程涉及到同一个临界资源的的临界区称为相关临界区。使用临界区时，一般不允许其运行时间过长，只要运行在临界区的线程还没有离开，其他所有进入此临界区的线程都会被挂起而进入等待状态，并在一定程度上影响程序的运行性能。
  - 临界区是一种轻量级的同步机制，与互斥和事件这些内核同步对象相比，临界区是用户态下的对象，即只能在同一进程中实现线程互斥。因无需在用户态和核心态之间切换，所以工作效率比较互斥来说要高很多。虽然临界区同步速度很快，但却只能用来同步本 进程内的线程，而不可用来同步多个进程中的线程。
- 临界区（Critical Section）
  - 保证在某一时刻只有一个线程能访问数据的简便办法。在任意时刻只允许一个线程对共享资源进行访问。如果有多个线程试图同时访问临界区，那么 在有一个线程进入后其他所有试图访问此临界区的线程将被挂起，并一直持续到进入临界区的线程离开。临界区在被释放后，其他线程可以继续抢占，并以此达到用原子方式操 作共享资源的目的。
  - 临界区包含两个操作原语：
  - EnterCriticalSection（） 进入临界区
  - LeaveCriticalSection（） 离开临界区
  - EnterCriticalSection（） 语句执行后代码将进入临界区以后无论发生什么，必须确保与之匹配的 LeaveCriticalSection（）都能够被执行到。否则临界区保护的共享资源将永远不会被释放。虽然临界区同步速度很快，但却只能用来同步本 进程内的线程，而不可用来同步多个进程中的线程。
  - MFC提供了很多功能完备的类，我用MFC实现了临界区。MFC为临界区提供有一个CCriticalSection类，使用该类进行线程同步处理是 非常简单的。只需在线程函数中用CCriticalSection类成员函数Lock（）和UnLock（）标定出被保护代码片段即可。Lock（）后代 码用到的资源自动被视为临界区内的资源被保护。UnLock后别的线程才能访问这些资源。
- 互斥量（Mutex）
  - 互斥量跟临界区很相似，只有拥有互斥对象的线程才具有访问资源的权限，由于互斥对象只有一个，因此就决定了任何情况下此共享资源都不会同时被多个线程所访问。当前占据资源的线程在任务处理完后应将拥有的互斥对象交出，以便其他线程在获得后得以访问资源。互斥量比临界区复杂。因为使用互斥不仅仅能够在同一应用程序不同线程中实现资源的安全共享，而且可以在不同应用程序的线程之间实现对资源的安全共享。
  - 互斥量包含的几个操作原语：
  - CreateMutex（） 创建一个互斥量
  - OpenMutex（） 打开一个互斥量

- ReleaseMutex（） 释放互斥量
- WaitForMultipleObjects（） 等待互斥量对象

- 信号量（Semaphores）
  - 信号量对象对线程的同步方式与前面几种方法不同，信号允许多个线程同时使用共享资源，这与操作系统中的PV操作相同。它指出了同时访问共享 资源的线程 最大数目。它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。
  - 在用CreateSemaphore（）创建信号量 时即要同时指出允许的最大资源计数和当前可用资源计数。一般是将当前可用资源计数设置为最大资源计数，每增加一个线程对共享资源的访问，当前可用资源计数 就会减1，只要当前可用资源计数是大于0的，就可以发出信号量信号。但是当前可用计数减小到0时则说明当前占用资源的线程数已经达到了所允许的最大数目， 不能在允许其他线程的进入，此时的信号量信号将无法发出。线程在处理完共享资源后，应在离开的同时通过ReleaseSemaphore（）函数将当前可 用资源计数加1。在任何时候当前可用资源计数决不可能大于最大资源计数。
  - PV操作及信号量的概念都是由荷兰科学家E.W.Dijkstra提出的。信号量S是一个整数，S大于等于零时代表可供并发进程使用的资源实体数，但S小于零时则表示正在等待使用共享资源的进程数。
  - P操作 申请资源：
  - （1）S减1；
  - （2）若S减1后仍大于等于零，则进程继续执行；
  - （3）若S减1后小于零，则该进程被阻塞后进入与该信号相对应的队列中，然后转入进程调度。
  - V操作 释放资源：
  - （1）S加1；
  - （2）若相加结果大于零，则进程继续执行；
  - （3）若相加结果小于等于零，则从该信号的等待队列中唤醒一个等待进程，然后再返回原进程继续执行或转入进程调度。
  - 信号量包含的几个操作原语：
  - CreateSemaphore（） 创建一个信号量
  - OpenSemaphore（） 打开一个信号量
  - ReleaseSemaphore（） 释放信号量
  - WaitForSingleObject（） 等待信号量
- 事件（Event）
  - 事件对象也可以通过通知操作的方式来保持线程的同步。并且可以实现不同进程中的线程同步操作。
  - 信号量包含的几个操作原语：
  - CreateEvent（） 创建一个事件

- OpenEvent（） 打开一个事件
- SetEvent（） 回置事件
- WaitForSingleObject（） 等待一个事件
- WaitForMultipleObjects（） 等待多个事件
- WaitForMultipleObjects 函数原型：
- WaitForMultipleObjects（
- IN DWORD nCount, // 等待句柄数
- IN CONST HANDLE *lpHandles, //指向句柄数组
- IN BOOL bWaitAll, //是否完全等待标志
- IN DWORD dwMilliseconds //等待时间
-)
- 参 数nCount指定了要等待的内核对象的数目，存放这些内核对象的数组由lpHandles来指向。fWaitAll对指定的这nCount个内核对象的两种等待方式进行了指定，为TRUE时当所有对象都被通知时函数才会返回，为FALSE则只要其中任何一个得到通知就可以返回。 dwMilliseconds在这里的作用与在WaitForSingleObject（）中的作用是完全一致的。如果等待超时，函数将返回 WAIT_TIMEOUT。

- 总结：
  - 1． 互斥量与临界区的作用非常相似，但互斥量是可以命名的，也就是说它可以跨越进程使用。所以创建互斥量需要的资源更多，所以如果只为了在进程内部是用的话使用临界区会带来速度上的优势并能够减少资源占用量 。因为互斥量是跨进程的互斥量一旦被创建，就可以通过名字打开它。
  - 2． 互斥量（Mutex），信号灯（Semaphore），事件（Event）都可以被跨越进程使用来进行同步数据操作，而其他的对象与数据同步操作无关，但对于进程和线程来讲，如果进程和线程在运行状态则为无信号状态，在退出后为有信号状态。所以可以使用WaitForSingleObject来等待进程和 线程退出。
  - 3． 通过互斥量可以指定资源被独占的方式使用，但如果有下面一种情况通过互斥量就无法处理，比如现在一位用户购买了一份三个并发访问许可的数据库系统，可以根据用户购买的访问许可数量来决定有多少个线程/进程能同时进行数据库操作，这时候如果利用互斥量就没有办法完成这个要求，信号灯对象可以说是一种资源计数器。

- 进程间通信
  - 进程同步、进程互斥、进程通信的关系
    - 在多道程序设计系统中，同一时刻可能有许多进程，这些进程之间存在两种基本关系：竞争关系和协作关系。
    - 进程的互斥、同步、通信都是基于这两种基本关系而存在的。
    - 为了解决进程间竞争关系（间接制约关系）而引入进程互斥；
    - 为了解决进程间松散的协作关系(直接制约关系)而引入进程同步；
    - 为了解决进程间紧密的协作关系而引入进程通信。
  - 进程通信：

    - 进程用户空间是相互独立的，一般而言是不能相互访问的。但很多情况下进程间需要互相通信，来完成系统的某项功能。进程通过与内核及其它进程之间的互相通信来协调它们的行为。
- 目的：
    - 数据传输：一个进程需要将它的数据发送给另一个进程，发送的数据量在一个字节到几兆字节之间。
    - 共享数据：多个进程想要操作共享数据，一个进程对共享数据的修改，别的进程应该立刻看到。通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。
    - 资源共享：多个进程之间共享同样的资源。为了作到这一点，需要内核提供锁和同步机制。
    - 进程控制：有些进程希望完全控制另一个进程的执行（如Debug进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。
- 进程通信的方式
- 1、管道(pipe)
    - 普通管道（匿名管道）：
        - 在内核中申请一块固定大小的缓冲区，程序拥有写入和读取的权利
        - 管道是一种半双工的通信方式，数据只能**单向流动，而且只能在父子进程间使用**
    - 命名管道
        - 也是半双工的通信方式，但是它允许无亲缘关系进程间的通信
- 2、信号量(semophore)
    - 信号量是一个计数器，可以用来控制**多个进程对共享资源的访问。**
    - 在内核中创建一个信号量集合（本质是个数组），数组的元素（信号量）都是1，使用P操作进行-1，使用V操作+1
- 3、信号(sinal)
    - 信号是一种比较复杂的通信方式，用于通知接收进程某些事件已经发生，要注意信号处理中调用的函数是否为信号安全。
- 4、消息队列(message queue)
    - 消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列提供了一个从一个进程向另外一个进程发送一块数据的方法。
    - 消息队列允许一个或多个进程写入或者读取消息。
    - 双向通信
    - 消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。
- 5、套接字
    - 套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。

- 6、共享内存
  - 共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。
  - 是最快的 IPC （进程间通信）方式，它是针对其他进程间通信方式运行效率低而专门设计的。
  - 它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。
- **最快的方式是共享内存**：采用共享内存通信的一个显而易见的好处是效率高，因为进程**可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝**

- 进程同步，线程同步
  - 同步就是协同步调，按预定的先后次序进行运行
  - 线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步
  - 线程同步的方式有哪些？（内核）
    - 互斥量：采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以可以保证公共资源不会被多个线程同时访问。
    - 信号量：它允许同一时刻多个线程访问同一资源，但是需要控制同一时刻访问此资源的最大线程数量。
    - 事件：通过通知操作的方式来保持多线程同步，还可以方便的实现多线程优先级的比较操作。
  - 线程同步
    - 临界区（Critical Section）、互斥对象（Mutex）：主要用于互斥控制；都具有拥有权的控制方法，只有拥有该对象的线程才能执行任务，所以拥有，执行完任务后一定要释放该对象。
    - 信号量（Semaphore）、事件对象（Event）：事件对象是以通知的方式进行控制，主要用于同步控制！
    - 1）临界区
      - 通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。在任意时刻只允许一个线程对共享资源进行访问，在有一个线程进入后，其他试图访问公共资源的线程将被挂起，并一直等到进入临界区的线程离开，临界区在被释放后，其他线程才可以抢占。它并**不是核心对象**，不是属于操作系统维护的，而是属于进程维护的。
    - 2）互斥对象
      - 互斥对象和临界区很像，采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以能保证公共资源不会同时被多个线程同时访问。当前拥有互斥对象的线程处理完任务后必须将线程交出，以便其他线程访问该资源。

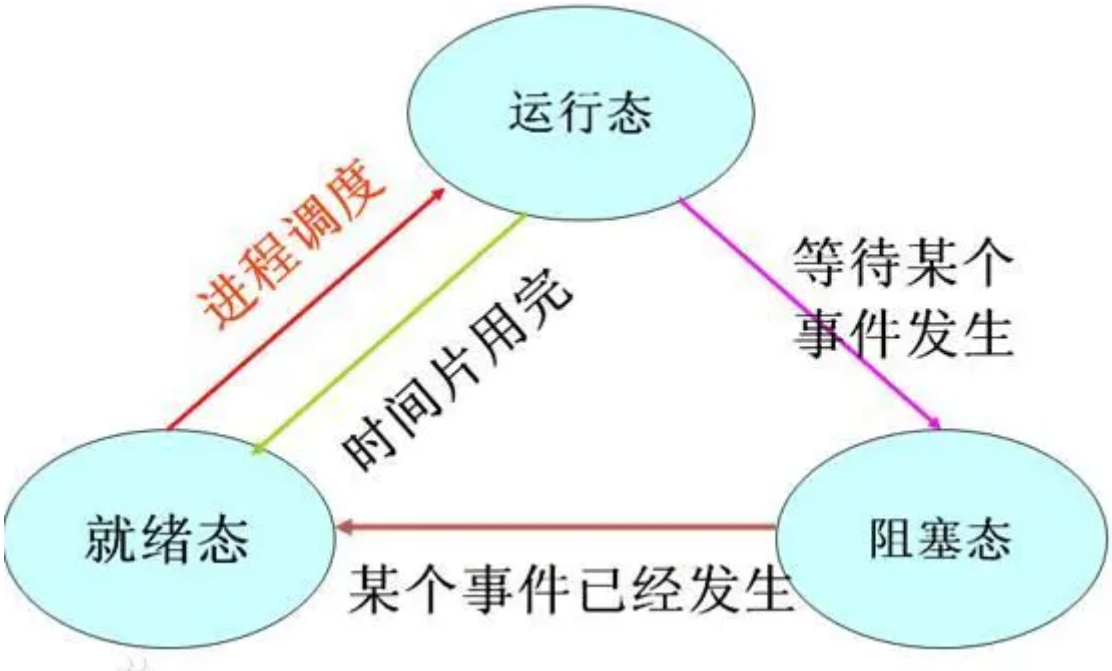
- 3）信号量
  - 信号量也是内核对象。它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目
- 4）事件对象
  - 内核对象，通过通知操作的方式来保持线程的同步，还可以方便实现对多个线程的优先级比较的操作
  - 事件可以解决线程间同步问题，因此也能解决互斥问题。

- 进程同步
  - 在OS中引入进程后，一方面使系统的吞吐量和资源的利用率得到提升，另一方面也使得系统变得复杂，如果没有合理的方式对进程进行妥善的管理，必然会引起进程对系统资源的无序竞争，使系统变得混乱；为了实现对并发进程的有效管理，在多道程序系统中引入了同步机制，常见的同步机制有：硬件同步机制、信号量机制、管程机制等，利用它们确保程序执行的可再现性；
  - 为避免竞争条件，操作系统需要利用同步机制在并发执行时，保证对临界区的互斥访问。进程同步的解决方案主要有：信号量和管程。
  - 信号量
    - 信号量机制（semaphore）是一种协调共享资源访问的方法。信号量由一个变量 semaphore 和两个原子操作组成，信号量只能通过 P 和 V 操作来完成，而且 P 和 V 操作都是原子操作。
    - 整型信号量S表示资源数目，除初始化外，仅能通过两个标准的原子操作进行修改：wait(S)和signal(S)；这两个操作长期以来也别称为P、V操作；
    - p（write），V（signal）
  - 管程
    - 管程采用面向对象思想，将表示共享资源的数据结构及相关的操作，包括同步机制，都集中并封装到一起。所有进程都只能通过管程间接访问临界资源，而管程只允许一个进程进入并执行操作，从而实现进程互斥。
    - 操作系统的作用之一就是实现对计算机系统资源的抽象，管程机制使用少量的信息和对该资源所执行的操作来表征该资源，所以共享系统资源就变为了共享数据结构，并将对这些共享数据结构的操作定义为一组过程。进程对共享资源的申请、释放和其他操作必须通过这组过程。代表共享资源的数据结构以及由对该共享数据结构实施操作的一组过程所组成的资源管理程序共同构成了一个操作系统的资源管理模块，我们称之为管程；
    - 管程由四部分组成：名称、局部于管程的共享数据结构说明、对该数据结构进行操作的一组过程、对局部于管程的共享数据结构设置初始值的语句；
    - 所有进程访问临界资源时，都只能通过管程间接访问，而管程每次只准许一个进程进入管程，从而实现互斥。管程体现了面向对象程序设计的思想；具有：模块化，即管程是一个独立的基本单位，可单独编译；抽象数据类型，不仅有数据还有对数据的操作；信息隐蔽，管程中的数据结构只能被管程中的过程访问，这些过程也是在管程内部定义的，而管程之外的进程只需调用而无需了解其内部的具

体实现细节。（这样，原来遍布系统的共享资源的访问代码，就集中到管程中啦）；

- 管程和进程的对比（两个截然不同的概念，有什么好对比的呢？大概是名字相似吧）
  - 两者都定义了各自的数据结构，但是管程定义的数据结构是对公用资源的抽象，进程定义的是私有数据结构PCB；
  - 两者都有对各自数据结构的操作，但是管程的操作是为了实现同步和初始化，进程是由顺序程序执行有关操作；
  - 进程的目的是在于实现系统的并发，而管程的目的是解决共享资源的互斥访问；
  - 进程是主动工作的，管程需要被其他程序使用，属于被动工作的；
  - 进程有动态性，管程是操作系统中的一个资源管理模块；

- 进程状态转换
  - 状态转换图



  - 运行状态：进程正在处理机上运行。在单处理机环境下，每一时刻最多只有一个进程处于运行状态。
  - 就绪状态：进程已处于准备运行的状态，即进程获得了除处理机之外的一切所需资源，一旦得到处理机即可运行。
  - 阻塞状态，又称等待状态：进程正在等待某一事件而暂停运行，如等待某资源为可用（不包括处理机）或等待输入/输出完成。即使处理机空闲，该进程也不能运行。
  - 就绪状态 -> 运行状态：处于就绪状态的进程被调度后，获得处理机资源（分派处理机时间片），于是进程由就绪状态转换为运行状态。
  - 运行状态 -> 就绪状态：处于运行状态的进程在时间片用完后，不得不让出处理机，从而进程由运行状态转换为就绪状态。此外，在可剥夺的操作系统中，当有更高优先级的进程就 、 绪时，调度程度将正执行的进程转换为就绪状态，让更高优先级的进程执行。
  - 运行状态 -> 阻塞状态：当进程请求某一资源（如外设）的使用和分配或等待某一事件的发生（如I/O操作的完成）时，它就从运行状态转换为阻塞状态。进程以系统调用的形式

请求操作系统提供服务，这是一种特殊的、由运行用户态程序调用操作系统内核过程的形式。

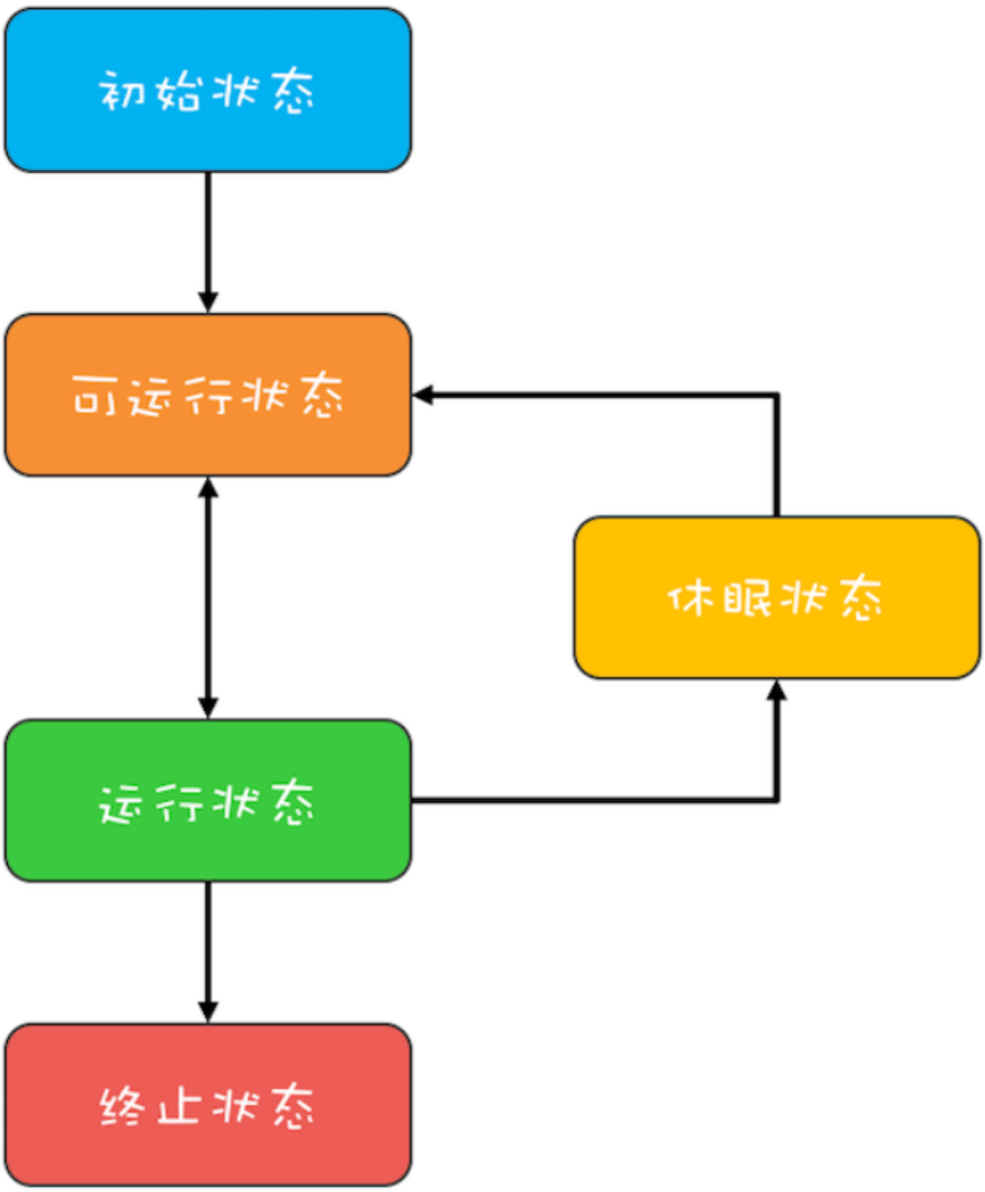
- 阻塞状态 -> 就绪状态：当进程等待的事件到来时，如I/O操作结束或中断结束时，中断处理程序必须把相应进程的状态由阻塞状态转换为就绪状态。

- Linux进程5中状态
  - 运行(正在运行或在运行队列中等待)

对比维度	多进程	多线程	总结
数据共享、同步	数据共享复杂，需要用IPC；数据是分开的，同步简单	因为共享进程数据，数据共享简单，但也是因为这个原因导致同步复杂	各有优势
内存、CPU	占用内存多，切换复杂，CPU利用率低	占用内存少，切换简单，CPU利用率高	线程占优
创建销毁、切换	创建销毁、切换复杂，速度慢	创建销毁、切换简单，速度很快	线程占优
编程、调试	编程简单，调试简单	编程复杂，调试复杂	进程占优
可靠性	进程间不会互相影响	一个线程挂掉将导致整个进程挂掉	进程占优
分布式	适应于多核、多机分布式；如果一台机器不够，扩展到多台机器比较简单	适应于多核分布式	进程占优

  - 中断(休眠中, 受阻, 在等待某个条件的形成或接受到信号)
  - 不可中断(收到信号不唤醒和不可运行, 进程必须等待直到有中断发生)
  - 僵死(进程已终止, 但进程描述符存在, 直到父进程调用wait4()系统调用后释放)
  - 停止(进程收到SIGSTOP, SIGSTP, SIGTIN, SIGTOU信号后停止运行运行)

- 线程状态转换
  - 线程生命周期基本上可以用下图这个“五态模型”来描述。这五态分别是：初始状态、可运行状态、运行状态、休眠状态和终止状态（如图）
    -



- 1.初始状态，指的是线程已经被创建，但是还不允许分配 CPU 执行。这个状态属于编程语言特有的，不过这里所谓的被创建，仅仅是在编程语言层面被创建，而在操作系统层面，真正的线程还没有创建。（还没有调用start()方法）
- 2.运行状态，指的是线程可以分配 CPU 执行。在这种状态下，真正的操作系统线程已经被成功创建了，所以可以分配 CPU 执行。
- 3.当有空闲的 CPU 时，操作系统会将其分配给一个处于可运行状态的线程，被分配到 CPU 的线程的状态就转换成了运行状态。
- 4.运行状态的线程如果调用一个阻塞的 API（例如以阻塞方式读文件）或者等待某个事件（例如条件变量），那么线程的状态就会转换到休眠状态，同时释放 CPU 使用权，休眠状态的线程永远没有机会获得 CPU 使用权。当等待的事件出现了，线程就会从休眠状态转换到可运行状态。
- 5.线程执行完或者出现异常就会进入终止状态，终止状态的线程不会切换到其他任何状态，进入终止状态也就意味着线程的生命周期结束了。

- 死锁
  - 当两个或两个以上进程占有自身资源，并请求对方资源时，若无外力作用，它们都将无法再向前推进。
  - 死锁产生的必要条件
    - 1）互斥条件:进程对所分配的资源进行排他性的使用
    - 2）请求和保持条件：进程被阻塞的时候并不释放锁申请到的资源
    - 3）不可剥夺条件：进程对于已经申请到的资源在使用完成之前不可以被剥夺
    - 4）环路等待条件：发生死锁的时候必然存在一个 进程--资源的环形链
  - 死锁避免：
    - 系统对进程发出每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，如果分配后系统可能发生死锁,则不予分配，否则予以分配。这是一种保证系统不进入死锁状态的动态策略。
  - 死锁预防：
    - 只要打破四个必要条件之一就能有效预防死锁的发生：
    - ● 打破互斥条件： 改造独占性资源为虚拟资源，大部分资源已无法改造。
    - ● 打破不可抢占条件： 当一进程占有一独占性资源后又申请一独占性资源而无法满足，则退出原占有的资源。
    - ● 打破占有且申请条件： 采用资源预先分配策略，即进程运行前申请全部资源，满足则运行，不然就等待，这样就不会占有且申请。
    - ● 打破循环等待条件： 实现资源有序分配策略，对所有设备实现分类编号，所有进程只能采用按序号递增的形式申请资源。
  - 死锁避免和死锁预防的区别：
    - 死锁预防是设法至少破坏产生死锁的四个必要条件之一，严格的防止死锁的出现；而死锁避免则不那么严格的限制产生死锁的必要条件的存在，因为即使死锁的必要条件存在，也不一定发生死锁。死锁避免是在系统运行过程中注意避免死锁的最终发生。
  - 避免死锁的算法
    - 银行家算法等等。
      - 思想: 判断此次请求是否造成死锁若会造成死锁，则拒绝该请求
      - 当一个进程申请使用资源的时候，银行家算法通过先 试探 分配给该进程资源，然后通过安全性算法判断分配后的系统是否处于安全状态，若不安全则试探分配作废，让该进程继续等待。
- 中断
  - 所谓的中断就是在计算机执行程序的过程中，由于出现了某些特殊事情，使得CPU暂停对程序的执行，转而去执行处理这一事件的程序。等这些特殊事情处理完之后再回去执行之前的程序。中断一般分为三类：
  - 1、由计算机硬件异常或故障引起的中断，称为内部异常中断；

- 2、由程序中执行了引起中断的指令而造成的中断，称为软中断（这也是和我们将要说明的系统调用相关的中断）；
- 3、由外部设备请求引起的中断，称为外部中断。简单来说，对中断的理解就是对一些特殊事情的处理。
- 与中断紧密相连的一个概念就是中断处理程序了。当中断发生的时候，系统需要去对中断进行处理，对这些中断的处理是由操作系统内核中的特定函数进行的，这些处理中断的特定的函数就是我们所说的中断处理程序了。
- 另一个与中断紧密相连的概念就是中断的优先级。中断的优先级说明的是当一个中断正在被处理的时候，处理器能接受的中断的级别。中断的优先级也表明了中断需要被处理的紧急程度。每个中断都有一个对应的优先级，当处理器在处理某一中断的时候，只有比这个中断优先级高的中断可以被处理器接受并且被处理。优先级比这个当前正在被处理的中断优先级要低的中断将会被忽略。
- 典型的中断优先级如下所示：
- 机器错误 > 时钟 > 磁盘 > 网络设备 > 终端 > 软件中断
- 当发生软件中断时，其他所有的中断都可能发生并被处理；但当发生磁盘中断时，就只有时钟中断和机器错误中断能被处理了。
- 系统调用
- 在讲系统调用之前，先说下进程的执行在系统上的两个级别：用户级和核心级，也称为用户态和系统态(user mode and kernel mode)。
- 程序的执行一般是在用户态下执行的，但当程序需要使用操作系统提供的服务时，比如说打开某一设备、创建文件、读写文件等，就需要向操作系统发出调用服务的请求，这就是系统调用。
- Linux系统有专门的函数库来提供这些请求操作系统服务的入口，这个函数库中包含了操作系统所提供的对外服务的接口。当进程发出系统调用之后，它所处的运行状态就会由用户态变成核心态。但这个时候，进程本身其实并没有做什么事情，这个时候是由内核在做相应的操作，去完成进程所提出的这些请求。
- 系统调用和中断的关系就在于，当进程发出系统调用申请的时候，会产生一个软件中断。产生这个软件中断以后，系统会去对这个软中断进行处理，这个时候进程就处于核心态了。
- 那么用户态和核心态之间的区别是什么呢？（以下区别摘至《UNIX操作系统设计》）
- 用户态的进程能存取它们自己的指令和数据，但不能存取内核指令和数据（或其他进程的指令和数据）。然而，核心态下的进程能够存取内核和用户地址
- 某些机器指令是特权指令，在用户态下执行特权指令会引起错误
- 对此要理解的一个是，在系统中内核并不是作为一个与用户进程平行的估计的进程的集合，内核是为用户进程运行的。

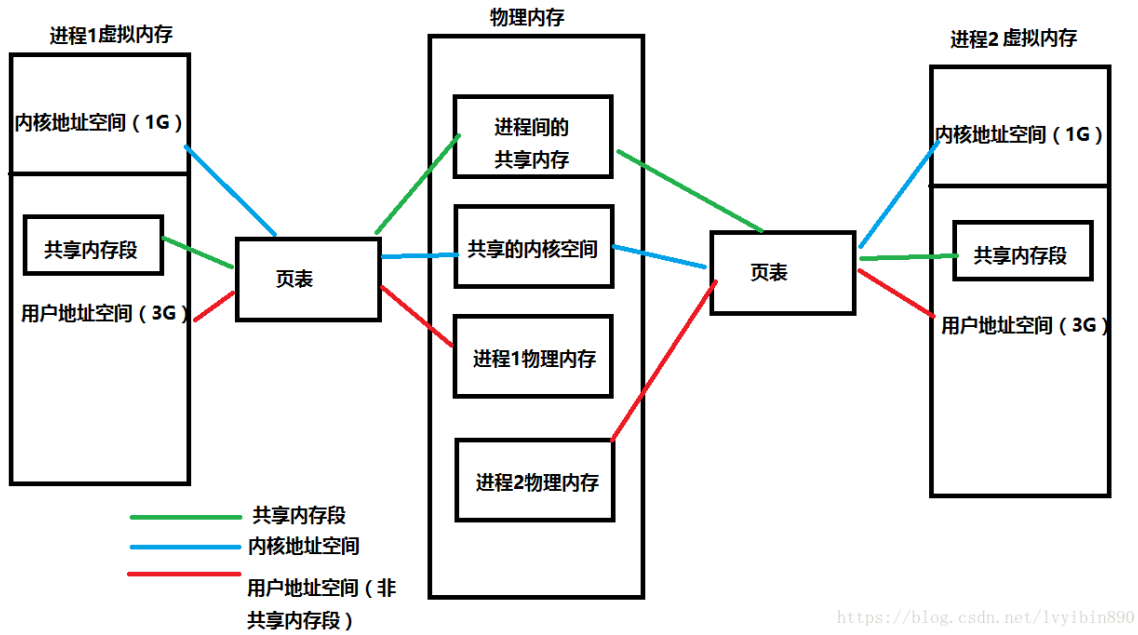
- 缓存区溢出
  - 缓冲区溢出是指当计算机向缓冲区填充数据时超出了缓冲区本身的容量，溢出的数据覆盖在合法数据上。
    - 危害有以下两点：

    - 程序崩溃，导致拒绝服务
    - 跳转并且执行一段恶意代码
  - 造成缓冲区溢出的主要原因是程序中没有仔细检查用户输入。
- 分段，分页
  - 分页和分段有什么区别?
  - 段是信息的逻辑单位，它是根据用户的需要划分的，因此段对用户是可见的；
  - 页是信息的物理单位，是为了管理主存的方便而划分的，对用户是透明的。
  - 段的大小不固定，有它所完成的功能决定；
  - 页大大小固定，由系统决定；
  - 段向用户提供二维地址空间；
  - 页向用户提供的是一维地址空间段是信息的逻辑单位，便于存储保护和信息的共享，页的保护和共享受到限制。
  - 为了利用和管理好子计算机的资源：内存
  - 不使用分段、分页时一个程序需要**连续的内存来装载整个程序**
  - 引入虚拟空间，通过把虚拟空间映射到物理地址空间，用户操作的是虚拟地址
  - 分段、分页主要区别在于粒度
  - 分页
    - 把主存空间划分为**大小相等且固定的块，块相对较小，作为主存的基本单位。**每个进程也以块为单位进行划分，进程在执行时，以块为单位逐个申请主存中的块空间。
    - 程序数据存储在不同的页面中，而页面又离散的分布在内存中，因此需要一个**页表来记录逻辑地址和实际存储地址之间的映射关系**，以实现从页号到物理块号的映射。
    - 由于页表也是存储在内存中的，因此和不适用分页管理的存储方式相比，访问分页系统中内存数据需要**两次的内存访问**(一次是从内存中访问页表，从中找到指定的物理块号，加上页内偏移得到实际物理地址；第二次就是根据第一次得到的物理地址访问内存取出数据)。
  - 分段
    - 分页是为了**提高内存利用率**，而分段是为了满足程序员在编写代码的时候的一些**逻辑需求**(比如数据共享，数据保护，动态链接等)。
    - 分段内存管理当中，地址是二维的，一维是段号，一维是段内地址；其中每个段的长度是不一样的，而且每个段内部都是从0开始编址的。由于分段管理中，每个段内部是连续内存分配，但是段和段之间是离散分配的，因此也存在一个逻辑地址到物理地址的映射关系，相应的就是段表机制。
  - 分页分段的比较
    - 页是信息的**物理单位**，是出于**系统内存利用率**的角度提出的离散分配机制；
    - 段是信息的**逻辑单位**，每个段含有一组意义完整的信息，是出于**用户角度提出的内存管理机制**
    - 页的大小是固定的，由系统决定；段的大小是不确定的，由用户决定

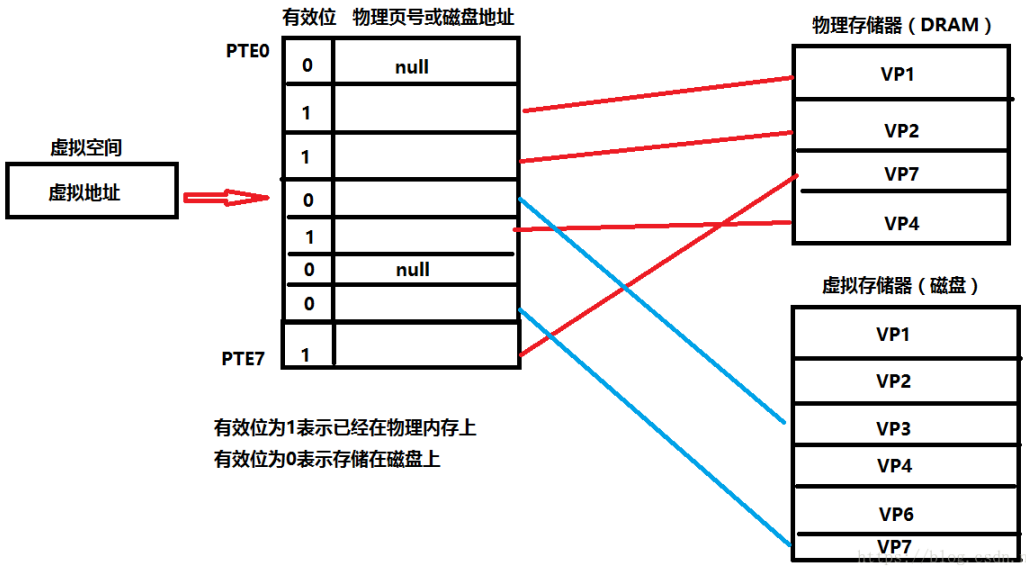
- 物理地址，逻辑地址，虚拟内存
  - 为什么需要虚拟内存
    - 操作系统有虚拟内存与物理内存的概念。在很久以前，还没有虚拟内存概念的时候，程序寻址用的都是物理地址。程序能寻址的范围是有限的，这取决于CPU的地址线条数。比如在32位平台下，寻址的范围是2^32也就是4G。并且这是固定的，如果没有虚拟内存，且每次开启一个进程都给4G的物理内存，就可能会出现很多问题：
    - 因为我的物理内存时有限的，当有多个进程要执行的时候，都要给4G内存，很显然你内存小一点，这很快就分配完了，于是没有得到分配资源的进程就只能等待。当一个进程执行完了以后，再将等待的进程装入内存。这种频繁的装入内存的操作是很没效率的
    - 由于指令都是直接访问物理内存的，那么我这个进程就可以修改其他进程的数据，甚至会修改内核地址空间的数据，这是我们不想看到的
    - 因为内存时随机分配的，所以程序运行的地址也是不正确的。
    - 于是针对上面会出现的各种问题，虚拟内存就出来了。
  - 虚拟内存
    - 是计算机系统内存管理的一种技术。它**使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间）**，而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。
    - 基于局部性原理，在程序装入时，可以将程序的一部分装入内存，而将其余部分留在外存，就可以启动程序执行。在程序执行过程中，当所访问的信息不在内存时，由操作系统将所需要的部分调入内存,然后继续执行程序。另一方面，操作系统将内存中暂时不使用的内容换出到外存上，从而腾出空间存放将要调入内存的信息。这样，系统好像为用户提供了一个比实际内存大得多的存储器，称为虚拟存储器。
    - 实际上，在虚拟内存对应的物理内存上，可能只对应的一点点的物理内存，实际用了多少内存，就会对应多少物理内存。
    - 进程得到的这4G虚拟内存是一个连续的地址空间（这也只是进程认为），而实际上，它通常是被分隔成多个物理内存碎片，还有一部分存储在外部磁盘存储器上，在需要时进行数据交换。
    - 进程开始要访问一个地址，它可能会经历下面的过程
      - 每次我要访问地址空间上的某一个地址，都需要把地址翻译为实际物理内存地址
      - 所有进程共享这整一块物理内存，每个进程只把自己目前需要的虚拟地址空间映射到物理内存上
      - 进程需要知道哪些地址空间上的数据在物理内存上，哪些不在（可能这部分存储在磁盘上），还有在物理内存上的哪里，这就需要通过页表来记录
      - 页表的每一个表项分两部分，第一部分记录此页是否在物理内存上，第二部分记录物理内存页的地址（如果在的话）
      - 当进程访问某个虚拟地址的时候，就会先去看页表，如果发现对应的数据不在物理内存上，就会发生缺页异常

    - 缺页异常的处理过程，操作系统立即阻塞该进程，并将硬盘里对应的页换入内存，然后使该进程就绪，如果内存已经满了，没有空地方了，那就找一个页覆盖，至于具体覆盖的哪个页，就需要看操作系统的页面置换算法是怎么设计的了。

- 物理地址
    - 它是地址转换的**最终地址**，进程在运行时执行指令和访问数据最后都要通过物理地址从主存中存取，是内存单元真正的地址。
- 虚拟内存和物理内存的联系
    - 如图



    - 页表的工作原理如下图、



    - 我们的cpu想访问虚拟地址所在的虚拟页(VP3)，根据页表，找出页表中第三条的值.判断有效位。 如果有效位为1，DRMA缓存命中，根据物理页号，找到物理页当中的内容，返回。

    - 若有效位为0，参数缺页异常，调用内核缺页异常处理程序。内核通过页面置换算法选择一个页面作为被覆盖的页面，将该页的内容刷新到磁盘空间当中。然后把VP3映射的磁盘文件缓存到该物理页上面。然后页表中第三条，有效位变成1，第二部分存储上了可以对应物理内存页的地址的内容。
    - 缺页异常处理完毕后，返回中断前的指令，重新执行，此时缓存命中，执行1。
    - 将找到的内容映射到告诉缓存当中，CPU从告诉缓存中获取该值，结束。
- 总结一下虚拟内存是怎么工作的
    - 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（叫做存储器映射）。这个时候数据和代码还是在磁盘上的。当运行到对应的程序时，进程去寻找页表，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。
    - 另外在进程运行过程中，要通过malloc来动态分配内存时，也只是分配了虚拟内存，即为这块虚拟内存对应的页表项做相应设置，当进程真正访问到此数据时，才引发缺页异常。
    - 可以认为虚拟空间都被映射到了磁盘空间中（事实上也是按需要映射到磁盘空间上，通过mmap，mmap是用来建立虚拟空间和磁盘空间的映射关系的）
- 利用虚拟内存机制的优点
    - 既然每个进程的内存空间都是一致而且固定的（32位平台下都是4G），所以链接器在链接可执行文件时，可以设定内存地址，而不用去管这些数据最终实际内存地址，这交给内核来完成映射关系
    - 当不同的进程使用同一段代码时，比如库文件的代码，在物理内存中可以只存储一份这样的代码，不同进程只要将自己的虚拟内存映射过去就好了，这样可以节省物理内存
    - 在程序需要分配连续空间的时候，只需要在虚拟内存分配连续空间，而不需要物理内存时连续的，实际上，往往物理内存都是断断续续的内存碎片。这样就可以有效地利用我们的物理内存
- 逻辑地址
    - 是指计算机用户**看到的地址**。例如：当创建一个长度为 100 的整型数组时，操作系统返回一个逻辑上的连续空间：指针指向数组第一个元素的内存地址。由于整型元素的大小为 4 个字节，故第二个元素的地址时起始地址加 4，以此类推。事实上，逻辑地址并不一定是元素存储的真实地址，即数组元素的物理地址（在内存条中所处的位置），并非是连续的，只是**操作系统通过地址映射，将逻辑地址映射成连续的**，这样更符合人们的直观思维。
- 虚拟存储器的特征：
    - 多次性：一个作业可以分多次被调入内存。多次性是虚拟存储特有的属性
    - 对换性：作业运行过程中存在换进换出的过程(换出暂时不用的数据换入需要的数据)

        - 虚拟性：虚拟性体现在其从逻辑上扩充了内存的容量(可以运行实际内存需求比物理内存大的应用程序)。虚拟性是虚拟存储器的最重要特征也是其最终目标。虚拟性建立在多次性和对换性的基础上行，多次性和对换性又建立在离散分配的基础上
- 进程调度
    - 调度种类
        - 高级、中级和低级调度作业从提交开始直到完成，往往要经历下述三级调度：
        - 高级调度：(High-Level Scheduling)又称为作业调度，它决定把后备作业调入内存运行；
        - 低级调度：(Low-Level Scheduling)又称为进程调度，它决定把就绪队列的某进程获得CPU；
        - 中级调度：(Intermediate-Level Scheduling)又称为在虚拟存储器中引入，在内、外存对换区进行进程对换。
    - 非抢占式调度与抢占式调度
        - 非抢占式
            - 分派程序一旦把处理机分配给某进程后便让它一直运行下去，直到进程完成或发生进程调度进程调度某事件而阻塞时，才把处理机分配给另一个进程。
        - 抢占式
            - 操作系统将正在运行的进程强行暂停，由调度程序将CPU分配给其他就绪进程的调度方式。
    - 调度策略的设计
        - 响应时间: 从用户输入到产生反应的时间
        - 周转时间: 从任务开始到任务结束的时间
        - CPU任务可以分为交互式任务和批处理任务，调度最终的目标是合理的使用CPU，使得交互式任务的响应时间尽可能短，用户不至于感到延迟，同时使得批处理任务的周转时间尽可能短，减少用户等待的时间。
    - 调度算法
        - 1、先到先服务First Come, First Served (FCFS)
            - 调度的顺序就是任务到达就绪队列的顺序。
            - 公平、简单(FIFO队列)、非抢占、不适合交互式。未考虑任务特性，平均等待时间可以缩短
        - 2、短作业优先Shortest Job First (SJF)
            - 最短的作业(CPU区间长度最小)最先调度。
            - 可以证明，SJF可以保证最小的平均等待时间。
            - SJF(SRJF): 如何知道下一CPU区间大小？根据历史进行预测: 指数平均法。
        - 3、时间片轮转Round-Robin(RR)
            - 设置一个时间片，按时间片来轮转调度（“轮叫”算法）

        - 优点: 定时有响应，等待时间较短；缺点: 上下文切换次数较多；
        - 如何确定时间片?
        - 时间片太大，响应时间太长；吞吐量变小，周转时间变长；当时间片过长时，退化为FCFS。
    - 4、优先权调度
        - 为每个任务分配优先级，首先执行优先级最高的进程，调度优先权最高的任务。
        - 注意：优先权太低的任务一直就绪，得不到运行，出现“饥饿”现象。
    - 5、多级反馈队列调度算法
        - 短作业优先忽略了长进程，多级反馈队列既能够使得高优先级得到处理，又能使得短作业迅速完成
        - 在多级队列的基础上，任务可以在队列之间移动，更细致的区分任务。
        - 可以根据“享用”CPU时间多少来移动队列，阻止“饥饿”。
        - 最通用的调度算法，多数OS都使用该方法或其变形，如UNIX、Windows等。
- 页面置换算法/缺页中断
    - 缺页中断-概念
        - 进程线性地址空间里的页面不必常驻内存，在执行一条指令时，如果发现他要访问的页没有在内存中（即存在位为0），那么停止该指令的执行，并产生一个页不存在的异常（中断）
        - 对应的故障处理程序可通过从外存加载该页的方法来排除故障，之后，原先引起的异常的指令就可以继续执行，而不再产生异常。
        - 虽然其名为“页缺失”错误，但实际上这并不一定是一种错误。而且这一机制有利于利用虚拟内存来增加程序可用内存空间。
    - 缺页中断
        - 在请求分页系统中，可以通过查询页表中的状态位来确定所要访问的页面是否存在于内存中。每当所要访问的页面不在内存时，会产生一次缺页中断，此时操作系统会根据页表中的外存地址在外存中找到所缺的一页，将其调入内存。
        - 缺页本身是一种中断，与一般的中断一样，需要经过4个处理步骤：
        - 1. 保护CPU现场
        - 2. 分析中断原因
        - 3. 转入缺页中断处理程序进行处理
        - 4. 恢复CPU现场，继续执行
        - 但是缺页中断时由于所要访问的页面不存在与内存时，有硬件所产生的一种特殊的中断，因此，与一般的中断存在区别：
        - 1. 在指令执行期间产生和处理缺页中断信号
        - 2. 一条指令在执行期间，可能产生多次缺页中断
        - 3. 缺页中断返回时，执行产生中断的那一条指令，而一般的中断返回时，执行下一条指令

- 请求调页/按需调页
  - 请求调页，也称按需调页，即对不在内存中的“页”，当进程**执行时要用时才调入**，否则有可能到程序结束时也不会调入。而内存中给页面留的位置是有限的，**在内存中以帧为单位放置页面**。为了防止请求调页的过程出现过多的内存页面错误（即需要的页面当前不在内存中，需要从硬盘中读数据，也即需要做页面的替换）而使得程序执行效率下降，我们需要设计一些页面置换算法，**页面按照这些算法进行相互替换**时，可以尽量达到较低的错误率。
- 1、最佳置换（Optimal， OPT)
  - 置换以后不再被访问，或者在将来最迟才回被访问的页面，缺页中断率最低。置换策略是将当前页面中在未来最长时间内不会被访问的页置换出去。
  - 只具有理论意义的算法，用来评价其他页面置换算法。所以该算法是不能实现的。但该算法仍然有意义，作为很亮其他算法优劣的一个标准。
- 2、先进先出置换算法
  - 简单粗暴的一种置换算法，没有考虑页面访问频率信息。每次淘汰最早调入的页面。
  - 置换最先调入内存的页面，即置换在内存中驻留时间最久的页面。按照进入内存的先后次序排列成队列，从队尾进入，从队首删除。但是该算法会淘汰经常访问的页面，不适应进程实际运行的规律，目前已经很少使用。
- 2、最近最久未使用算法LRU
  - 算法赋予每个页面一个访问字段，用来记录上次页面被访问到现在所经历的时间t，每次置换的时候把t值最大的页面置换出去(实现方面可以采用寄存器或者栈的方式实现)。
  - LRU算法普偏地适用于各种类型的程序，但是系统要时时刻刻对各页的访问历史情况加以记录和更新，开销太大，因此LRU算法必须要有硬件的支持。
- 最少使用算法LFU
  - 设置寄存器记录页面被访问次数，每次置换的时候置换当前访问次数最少的。
- 时钟算法clock(也被称为是最近未使用算法NRU)
  - 页面设置一个访问位，并将页面链接为一个环形队列，页面被访问的时候访问位设置为1。页面置换的时候，如果当前指针所指页面访问为为0，那么置换，否则将其置为0，循环直到遇到一个访问为位0的页面。
- 改进型Clock算法
  - 在Clock算法的基础上添加一个修改位，替换时根究访问位和修改位综合判断。优先替换访问位和修改位都是0的页面，其次是访问位为0修改位为1的页面。

- 动态链接，静态链接，动态库加载机制
  - 1. 分别编译与链接
    - 大多数高级语言都支持分别编译（Compiling），程序员可以显式地把程序划分为独立的模块或文件，然后由编译器（Compiler）对每个独立部分分别进行编译。在编译之后，由链接器（Linker）把这些独立编译单元链接（Linking）到一起。链接方式分为两种：

- (1)　静态链接方式：在程序开发中，将各种目标模块（.OBJ）文件、运行时库（.LIB）文件，以及经常是已编译的资源（.RES）文件链接在一起，以便创建Windows的.EXE文件。
- (2)　动态链接方式：在程序运行时，Windows把一个模块中的函数调用链接到库模块中的实际函数上的过程。

- 2.　静态链接库与动态链接库
  - 静态链接库（Static Library，简称LIB）与动态链接库（Dynamic Link Library，简称DLL）都是共享代码的方式。如果使用静态链接库（也称静态库），则无论你愿不愿意，.LIB文件中的指令都会被直接包含到最终生成的.EXE文件中。但是若使用.DLL文件，该.DLL文件中的代码不必被包含在最终的.EXE文件中，.EXE文件执行时可以“动态”地载入和卸载这个与.EXE文件独立的.DLL文件。
- 请阐述动态链接库与静态链接库的区别。
  - 解答：静态链接库是.lib格式的文件，一般在工程的设置界面加入工程中，程序编译时会把lib文件的代码加入你的程序中因此会增加代码大小，你的程序一运行lib代码强制被装入你程序的运行空间，不能手动移除lib代码。
  - 动态链接库是程序运行时动态装入内存的模块，格式*.dll，在程序运行时可以随意加载和移除，节省内存空间。
  - 在大型的软件项目中一般要实现很多功能，如果把所有单独的功能写成一个个lib文件的话，程序运行的时候要占用很大的内存空间，导致运行缓慢；但是如果将功能写成dll文件，就可以在用到该功能的时候调用功能对应的dll文件，不用这个功能时将dll文件移除内存，这样可以节省内存空间。
- 2.1.　　动态链接方式
  - 链接一个DLL有两种方式：
  - 2.1.1　　载入时动态链接（Load-Time Dynamic Linking）
  - 使用载入时动态链接，调用模块可以像调用本模块中的函数一样直接使用导出函数名调用DLL中的函数。这需要在链接时将函数所在DLL的导入库链接到可执行文件中，导入库向系统提供了载入DLL时所需的信息及用于定位DLL函数的地址符号。（相当于注册，当作API函数来使用，其实API函数就存放在系统DLL当中。）
  - 2.1.2　　运行时动态链接（Run-Time Dynamic Linking）
  - 使用运行时动态链接，运行时可以通过LoadLibrary或LoadLibraryEx函数载入DLL。DLL载入后，模块可以通过调用GetProcAddress获取DLL函数的入口地址，然后就可以通过返回的函数指针调用DLL中的函数了。如此即可避免导入库文件了。
- 2.2.　　二者优点及不足
- 2.2.1　　静态链接库的优点
  - (1)　代码装载速度快，执行速度略比动态链接库快；
  - (2)　只需保证在开发者的计算机中有正确的.LIB文件，在以二进制形式发布程序时不需考虑在用户的计算机上.LIB文件是否存在及版本问题，可避免DLL地狱等问题。
- 2.2.2　　动态链接库的优点

- (1)　更加节省内存并减少页面交换；
- (2)　DLL文件与EXE文件独立，只要输出接口不变（即名称、参数、返回值类型和调用约定不变），更换DLL文件不会对EXE文件造成任何影响，因而极大地提高了可维护性和可扩展性；
- (3)　不同编程语言编写的程序只要按照函数调用约定就可以调用同一个DLL函数。

- 2.2.3　　不足之处
  - (1)　使用静态链接生成的可执行文件体积较大，包含相同的公共代码，造成浪费；
  - (2)　使用动态链接库的应用程序不是自完备的，它依赖的DLL模块也要存在，如果使用载入时动态链接，程序启动时发现DLL不存在，系统将终止程序并给出错误信息。而使用运行时动态链接，系统不会终止，但由于DLL中的导出函数不可用，程序会加载失败；
  - (3)　使用动态链接库可能造成DLL地狱。
    - DLL 地狱（DLL Hell）是指因为系统文件被覆盖而让整个系统像是掉进了地狱。
    - 简单地讲，DLL地狱是指当多个应用程序试图共享一个公用组件时，如某个DLL或某个组件对象模型（COM）类，所引发的一系列问题。
    - 最典型的情况是，某个应用程序将要安装一个新版本的共享组件，而该组件与机器上的现有版本不向后兼容。虽然刚安装的应用程序运行正常，但原来依赖前一版本共享组件的应用程序也许已无法再工作。在某些情况下，问题的起因更加难以预料。比如，当用户浏览某些web站点时会同时下载某个Microsoft ActiveX控件。如果下载该控件，它将替换机器上原有的任何版本的控件。如果机器上的某个应用程序恰好使用该控件，则很可能也会停止工作。
    - 在许多情况下，用户需要很长时间才会发现应用程序已停止工作。结果往往很难记起是何时的机器变化影响到了该应用程序。

- 动态链接
  - 动态链接就是在编译的时候不直接拷贝可执行代码，而是通过记录一系列符号和参数，在程序运行或加载时将这些信息传递给操作系统，操作系统负责将需要的动态库加载到内存中，然后程序在运行到指定的代码时，去共享执行内存中已经加载的动态库可执行代码，最终达到运行时连接的目的。优点是多个程序可以共享同一段代码，而不需要在磁盘上存储多个拷贝，缺点是由于是运行时加载，可能会影响程序的前期执行性能。

- 静态链接
  - 在链接阶段，会将汇编生成的目标文件.o与引用到的库一起链接打包到可执行文件中。因此对应的链接方式称为静态链接。静态链接就是在编译链接时直接将需要的执行代码拷贝到调用处，优点就是在程序发布的时候就不需要的依赖库，也就是不再需要带着库一块发布，程序可以独立执行，但是体积可能会相对大一些。

- 用户态，内核态/转换
  - 内核态与用户态是操作系统的两种运行级别
  - 软件中最基础的部分是操作系统，它运行在 内核态 中，内核态也称为 管态 和 核心态，它们都是操作系统的运行状态，只不过是不同的叫法而已。操作系统具有硬件的访问

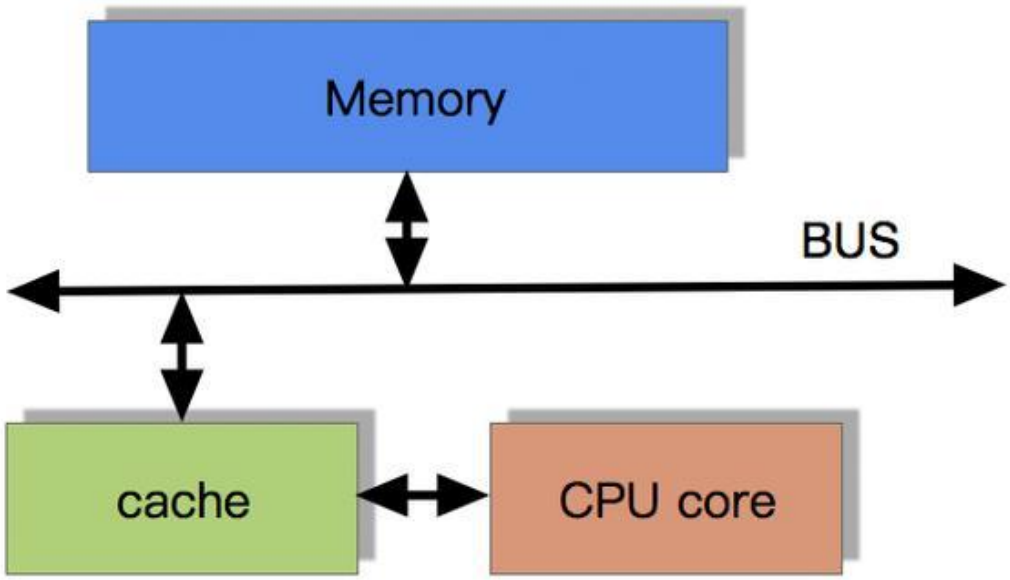
权，可以执行机器能够运行的任何指令。软件的其余部分运行在 用户态 下。

- 内核态
  - 从本质上说就是我们所说的内核，它是一种特殊的软件程序，特殊在哪儿呢？控制计算机的硬件资源，例如协调CPU资源，分配内存资源，并且提供稳定的环境供应用程序运行。
- 用户态
  - 用户态就是提供应用程序运行的空间，为了使应用程序访问到内核管理的资源例如CPU，内存，I/O。内核必须提供一组通用的访问接口，这些接口就叫系统调用。
- 用户态切换到内核态的3种方式
  - 1) 系统调用：这是用户态进程主动要求切换到内核态的一种方式，用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作。而系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现，例如Linux的int 80h中断。
  - 2) 异常：当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。
  - 3) 外围设备的中断：当外围设备完成用户请求的操作后，会向CPU发出相应的中断信号，这时CPU会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序，如果先前执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了由用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等。
- 这3种方式是系统在运行时由用户态转到内核态的最主要方式，其中系统调用可以认为是用户进程主动发起的，异常和外围设备中断则是被动的。

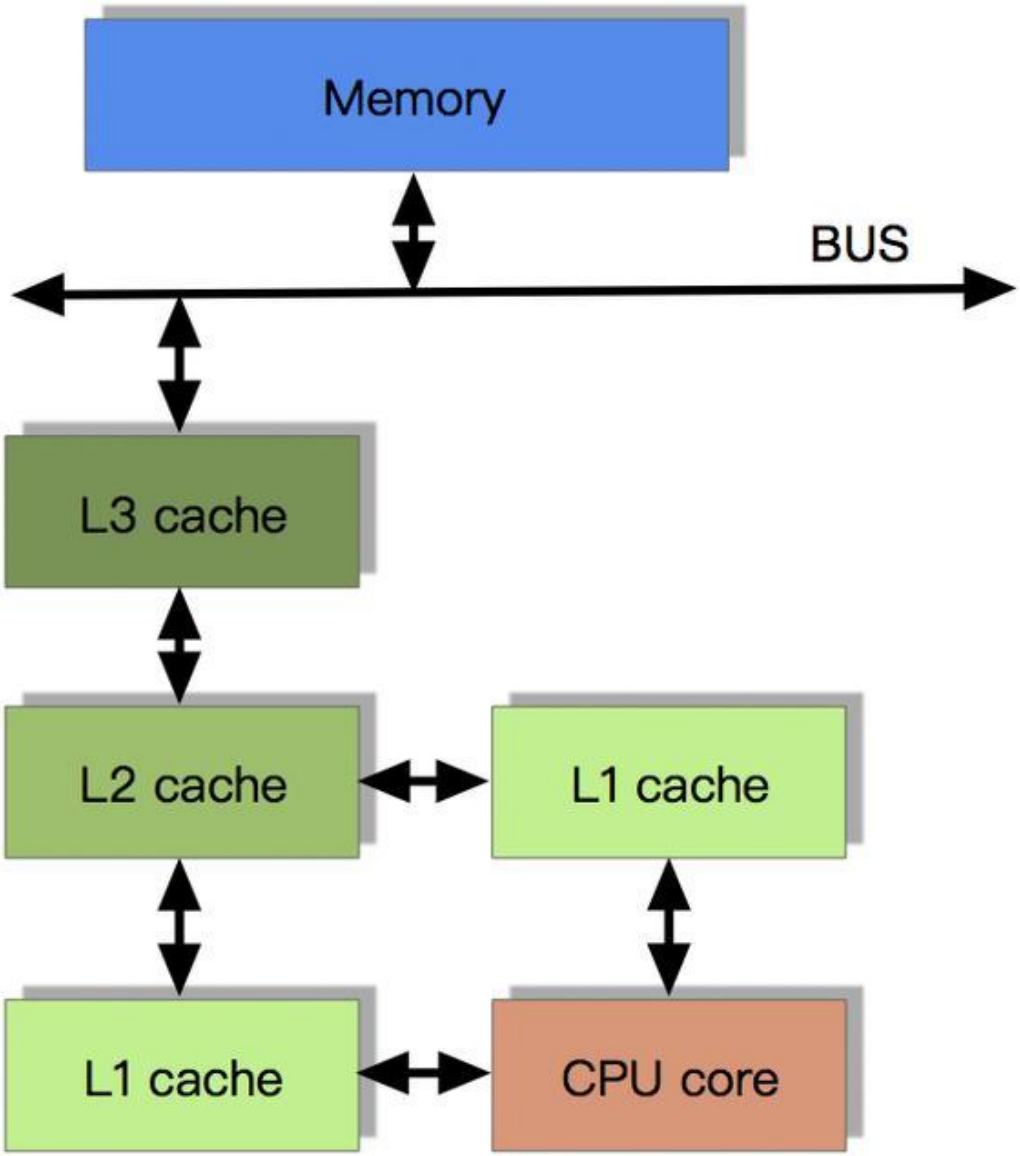
- 临界区/互斥体
  - 每个进程（线程）中，访问临界资源（不可共享的资源）的那段代码称为临界区。每次只允许一个进程（线程）进入临界区，进入后不允许其他进程（线程）进入。
  - 如何解决临界区冲突?
    - 一次只允许一个进程（线程）进入临界区，如果有一个进程（线程）已经进入了自己的临界区，其他试图进入临界区的进程（线程）必须等待
    - 进入临界区的进程（线程）要在有限的时间内退出临界区，以便于其他进程（线程）进入自己的临界区
    - 如果进程（线程）无法进入自己的临界区，则应该让出CPU等资源，避免进程（线程）出现“忙等”现象。
  - 实现临界区的方法有哪些?
    - 软件实现
    - 中断屏蔽（关中断，开中断）
    - 硬件指令方法
  - 互斥体和临界区的区别

    - 1、临界区只能用于对象在同一进程里线程间的互斥访问；互斥体可以用于对象进程间或线程间的互斥访问。
    - 2、临界区是非内核对象，只在用户态进行锁操作，速度快；互斥体是内核对象，在核心态进行锁操作，速度慢。
    - 3、临界区和互斥体在Windows平台都下可用；Linux下只有互斥体可用
- 内存抖动
    - 内存抖动一般是内存分配算法不好，内存太小或者程序的算法不佳引起的页面频繁地从内存调入/调出的行为。
- 僵尸进程
    - 那你知道什么叫僵尸进程？怎么处理僵尸进程?
    - 定义：
        - 完成了生命周期但却依然留在进程表中的进程，我们称之为 “僵尸进程”。
    - 如何产生：
        - 当你运行一个程序时，它会产生一个父进程以及很多子进程。 所有这些子进程都会消耗内核分配给它们的内存和 CPU 资源。
        - 这些子进程完成执行后会发送一个 Exit 信号然后死掉。这个 Exit 信号需要被父进程所读取。父进程需要随后调用 wait 命令来读取子进程的退出状态，并将子进程从进程表中移除。
        - 若父进程正确第读取了子进程的 Exit 信号，则子进程会从进程表中删掉。
        - 但若父进程未能读取到子进程的 Exit 信号，则这个子进程虽然完成执行处于死亡的状态，但也不会从进程表中删掉。
    - 处理：
        - 打开终端并输入下面命令:
        - ps aux | grep Z
        - kill -s SIGCHLD pid
        - 将这里的 pid 替换成父进程的进程 id，这样父进程就会删除所有以及完成并死掉的子进程了。
        - 确保删除子僵尸的唯一方法就是杀掉它们的父进程。
    - ps -aux什么意思
        - ps -aux 显示所有的进程（静态）
        - a 显示所有用户的进程
        - u 显示用户
        - x 显示无控制终端的进程
- CPU高速缓存
    - CPU为何要有高速缓存

    - CPU在摩尔定律的指导下以每18个月翻一番的速度在发展，然而内存和硬盘的发展速度远远不及CPU。这就造成了高性能能的内存和硬盘价格及其昂贵。然而CPU的高度运算需要高速的数据。为了解决这个问题，CPU厂商在CPU中内置了少量的高速缓存以解决I\O速度和CPU运算速度之间的不匹配问题。
    - 在CPU访问存储设备时，无论是存取数据抑或存取指令，都趋于聚集在一片连续的区域中，这就被称为**局部性原理**。
    - 时间局部性（Temporal Locality）：如果一个信息项正在被访问，那么在近期它很可能还会被再次访问。比如循环、递归、方法的反复调用等。
    - 空间局部性（Spatial Locality）：如果一个存储器的位置被引用，那么将来他附近的位置也会被引用。比如顺序执行的代码、连续创建的两个对象、数组等。
- 带有高速缓存的CPU执行计算的流程
    - 程序以及数据被加载到主内存
    - 指令和数据被加载到CPU的高速缓存
    - CPU执行指令，把结果写到高速缓存
    - 高速缓存中的数据写回主内存



- 目前流行的多级缓存结构
    - 由于CPU的运算速度超越了1级缓存的数据I\O能力，CPU厂商又引入了多级的缓存结构。



- 多核CPU多级缓存一致性协议MESI
  - 多核CPU的情况下有多个一级缓存，如何保证缓存内部数据的一致,不让系统数据混乱。这里就引出了一个一致性的协议MESI。
- MESI协议缓存状态
  - MESI 是指4中状态的首字母。每个Cache line有4个状态，可用2个bit表示，它们分别是：

状态	描述	监听任务
M 修改 (Modified)	该Cache line有效，数据被修改了，和内存中的数据不一致，数据只存在于本Cache中。	缓存行必须时刻监听所有试图读该缓存行相对就主存的操作，这种操作必须在缓存将该缓存行写回主存并将状态变成S（共享）状态之前被延迟执行。
E 独享、互斥 (Exclusive)	该Cache line有效，数据和内存中的数据一致，数据只存在于本Cache中。	缓存行也必须监听其它缓存读主存中该缓存行的操作，一旦有这种操作，该缓存行需要变成S（共享）状态。
S 共享 (Shared)	该Cache line有效，数据和内存中的数据一致，数据存在于很多Cache中。	缓存行也必须监听其它缓存使该缓存行无效或者独享该缓存行的请求，并将该缓存行变成无效（Invalid）。
I 无效 (Invalid)	该Cache line无效。	无

    - 对于M和E状态而言总是精确的，他们在和该缓存行的真正状态是一致的，而S状态可能是非一致的。如果一个缓存将处于S状态的缓存行作废了，而另一个缓存实际上可能已经独享了该缓存行，但是该缓存却不会将该缓存行升迁为E状态，这是因为其它缓存不会广播他们作废掉该缓存行的通知，同样由于缓存并没有保存该缓存行的copy的数量，因此（即使有这种通知）也没有办法确定自己是否已经独享了该缓存行。
    - 从上面的意义看来E状态是一种投机性的优化：如果一个CPU想修改一个处于S状态的缓存行，总线事务需要将所有该缓存行的copy变成invalid状态，而修改E状态的缓存不需要使用总线事务。
- 单核读取
    - 那么执行流程是：
    - CPU A发出了一条指令，从主内存中读取x。
    - 从主内存通过bus读取到缓存中（远端读取Remote read）,这是该Cache line修改为E状态（独享）
- 双核读取
    - 那么执行流程是：
    - CPU A发出了一条指令，从主内存中读取x。
    - CPU A从主内存通过bus读取到 cache a中并将该cache line 设置为E状态。
    - CPU B发出了一条指令，从主内存中读取x。
    - CPU B试图从主内存中读取x时，CPU A检测到了地址冲突。这时CPU A对相关数据做出响应。此时x 存储于cache a和cache b中，x在chche a和cache b中都被设置为S状态(共享)。
- 修改数据
    - 那么执行流程是：
    - CPU A 计算完成后发指令需要修改x.
    - CPU A 将x设置为M状态（修改）并通知缓存了x的CPU B, CPU B将本地cache b中的x设置为I状态(无效)
    - CPU A 对x进行赋值。

- 同步数据
  - 那么执行流程是：
  - CPU B 发出了要读取x的指令。
  - CPU B 通知CPU A,CPU A将修改后的数据同步到主内存时cache a 修改为E（独享）
  - CPU A同步CPU B的x,将cache a和同步后cache b中的x设置为S状态（共享）。