# 操作系统特征

* 并发性，操作系统中的程序从宏观上看是同步发生的，操作系统具有同时处理多个程序的能力。
* 共享性，操作系统中的资源可以由若干个并发进程使用。
* 异步性，各个进程执行走走停停，进程什么时候以什么样的速度推进时不可预知的。
* 虚拟性，操作系统的虚拟技术包括时分复用技术和空分复用技术，时分复用入时分系统，多个程序并发使用系统资源，空分复用如虚拟磁盘，将一个物理磁盘分成多个逻辑磁盘，更加安全方便；虚拟内存，逻辑上扩大程序的存储容量。

# 进程、线程和协程

* 进程是对运行时程序的封装，是系统进行资源调度和分配的的基本单位，实现了操作系统的并发；
* 线程是进程的子任务，是CPU调度和分派的基本单位，用于保证程序的实时性，实现进程内部的并发；一个程序至少有一个进程，一个进程至少有一个线程，线程依赖于进程而存在；进程在执行过程中拥有独立的内存单元，而多个线程共享进程的内存。
* 协程：相较于线程更加轻量级，不被操作系统内核管理，完全是由程序所控制。协程在程序内部是由用户中断的，转而执行别的协程，在适当的时候再返回来接着执行。

## 协程好处

* 性能提升，不会像线程切换那样消耗资源。
* 不需要多线程锁的机制，因为只有一个线程，不存在同时写变量冲突。因此在协程中控制共享资源不加锁，只需判断状态，因此执行效率比线程高很多。

## 进程与线程的区别

* 进程有自己的地址空间和资源，同一进程下的线程共享进程的地址空间和资源。
* 进程之间一般互不影响，但是同一进程中一个线程崩溃整个进程崩溃，所以多进程比多线程健壮；
* 进程切换，消耗的资源大。所以涉及到频繁的切换，使用线程要好于进程；
* 进程可以独立运行，但线程一般只能依靠进程运行。

## 什么时候用多进程？什么时候用多线程？

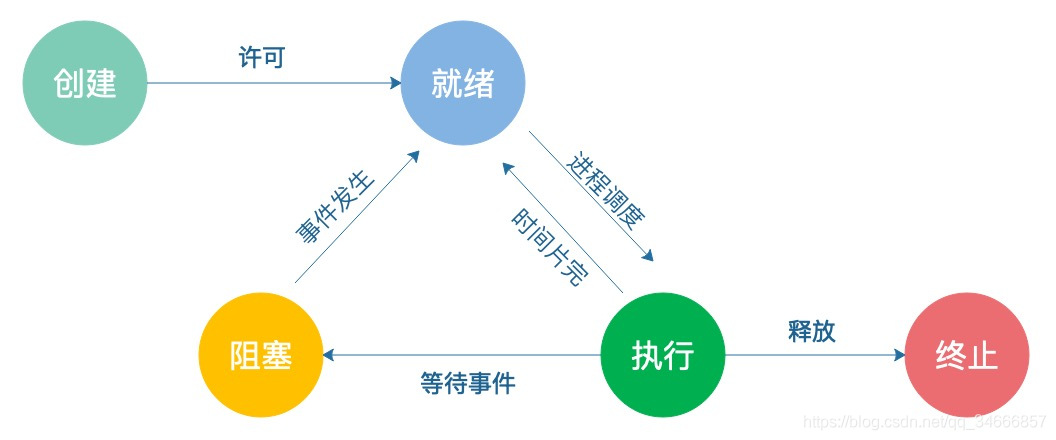
* 需要频繁创建销毁的优先用多线程；
* CPU密集型使用多进程，IO密集型使用多线程；
* 隔离性较强用多进程（如linux的init进程，几乎所有其他进程都是由它创建，这些进程之间联系较弱，隔离性较强），隔离性较弱用多线程；
* 可能要扩展到多机分布的用多进程，多核分布的用多线程；

# 进程的状态

就绪状态：进程已获得除处理机以外的所需资源，等待分配CPU资源；

运行状态：占用CPU资源运行，处于此状态的进程数小于等于CPU数；

阻塞状态：进程等待某种条件（资源），在条件满足之前无法执行；



运行态→阻塞态：往往是由于等待外设，等待主存等资源分配或等待人工干预（I/O）而引起的。

阻塞态→就绪态：则是等待的条件已满足，只需分配到处理器后就能运行。

运行态→就绪态：不是由于自身原因，而是由外界原因使运行状态的进程让出处理器，这时候就变成就绪态。例如时间片用完，或有更高优先级的进程来抢占处理器等。

就绪态→运行态：系统按某种调度策略从就绪队列中选择的一个进程占用处理器，此时就变成了运行态。

# 进程的创建fork

pid\_t fork(void);

* 在父进程中，返回子进程的pid。
* 在子进程中，返回0，可以用getppid获取父进程的pid。
* 出错，返回-1。

Ps：fork后子进程继承了父进程整个地址空间，包括进程上下文，堆栈、静态变量等，采用读时共享，写时复制的方式。

# 进程调度算法

* 先来先服务调度算法，进程按顺序排队调度。
* 短作业优先调度算法，优先调度完成最快的进程。
* 优先级调度算法，进程按优先级调度，包括抢占式和非抢占式，抢占式就是运行在低优先级进程运行时切换成刚来的高优先级进程。
* 高响应比优先调度算法，响应比=作业周转时间/作业处理时间=1+（作业等待时间/作业处理时间），即考虑了作业处理时间也考虑了作业等待时间，是先来先服务和短作业优先的折中。
* 时间片轮转法调度算法，当时间片用完时进程切出，换就绪队列中别的进程执行，并将这个进程放入队尾。

# 线程的状态（java）

1. 新建(NEW)：新创建了一个线程对象。
2. 可运行(RUNNABLE)：线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取cpu 的使用权 。
3. 运行(RUNNING)：可运行状态(runnable)的线程获得了cpu 时间片（timeslice） ，执行程序代码。
4. 阻塞(BLOCKED)：阻塞状态是指线程因为某种原因放弃了cpu 使用权，也即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得cpu timeslice 转到运行(running)状态。阻塞的情况分三种：

* 等待阻塞：运行(running)的线程执行o.wait()方法，JVM会把该线程放入等待队列(waitting queue)中。
* 同步阻塞：运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。
* 其他阻塞：运行(running)的线程执行Thread.sleep(long ms)或t.join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入可运行(runnable)状态。

1. 死亡(DEAD)：线程run()、main() 方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。

# 进程间通信

进程是一个独立的资源分配单元，不同进程（这里所说的进程通常指的是用户进程）之间的资源是独立的，没有关联，不能在一个进程中直接访问另一个进程的资源。

但是，进程不是孤立的，不同的进程需要进行信息的交互和状态的传递等，因此需要进程间通信( IPC：Inter Processes Communication )。

## 进程间通信目的

数据传输：一个进程需要将它的数据发送给另一个进程。

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。

资源共享：多个进程之间共享同样的资源。为了做到这一点，需要内核提供互斥和同步机制。

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。

## 进程间通信方式

* 管道pipe：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。
* 命名管道FIFO：有名管道也是半双工的通信方式，FIFO内容保存在内存中，但是它允许无亲缘关系进程间的通信。
* 消息队列MessageQueue：消息队列是由消息组成的链表，存放在内核中。消息队列基于消息，管道基于字节流，消息队列可以传递更多信息，消息队列可以按任意顺序或者优先级读取消息，管道是先进先出的。
* 共享存储SharedMemory(mmap)：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。共享内存是最快的进程间通信方式，它是针对其他进程间通信方式运行效率低而专门设计的。
* 信号量Semaphore：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。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的互斥同步手段。
* 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个事件已经发生。（如子进程终止时可以用信号通知父进程回收子进程，如果子进程结束，父进程并没有回收子进程，子进程就会成为僵尸进程）
* 套接字Socket：它是唯一可用于不同主机间的进程通信。

# 孤儿进程

父进程运行结束，但子进程还在运行（未运行结束）的子进程就称为孤儿进程（Orphan Process）。

每当出现一个孤儿进程的时候，内核就把孤儿进程的父进程设置为 init ，而 init 进程会循环地 wait() 它的已经退出的子进程。这样，当一个孤儿进程凄凉地结束了其生命周期的时候，init 进程就会代表党和政府出面处理它的一切善后工作。因此孤儿进程并不会有什么危害。

# 僵尸进程

进程终止，父进程尚未回收，子进程残留资源（PCB）存放于内核中，变成僵尸（Zombie）进程。

这样就会导致一个问题，如果进程不调用wait()或 waitpid()的话，那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的，如果大量的产生僵尸进程，将因为没有可用的进程号而导致系统不能产生新的进程，此即为僵尸进程的危害，应当避免。

# 守护进程

守护进程（Daemon Process），也就是通常说的Daemon进程（精灵进程），是Linux中的后台服务进程。它是一个生存期较长的进程，通常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。比如init进程，就负责回收孤儿进程。一般采用以d结尾的名字。

守护进程是个特殊的孤儿进程，这种进程脱离终端。

## 实现

* 创建子进程，父进程exit退出。
* 调用setsid创建新的会话，脱离控制终端。
* chdir(“/”)更改当前工作目录为根目录。
* 设置权限掩码为0，增加守护进程的权限。
* 关闭文件描述符STDIN\_FILENO, STDOUT\_FILENO, STDERR\_FILENO。
* 执行守护进程核心任务。

# 父子进程先后退出的情况

* 父进程永不退出，子进程结束时成为僵尸进程
* 父进程先于子进程退出，子进程在父进程退出后成为孤儿进程，由系统回收。
* 子进程先于父进程退出，子进程结束时变成僵尸进程，父进程结束时，子进程由僵尸进程变为孤儿进程，由系统回收。

# 线程同步方式

* 互斥量 Synchronized/Lock：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 Semphare：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(信号)，Wait/Notify：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。

# 什么是线程上下文切换？

多线程编程中一般线程的个数都大于CPU核心的个数。线程在CPU时间片用完或线程发生阻塞时会进行线程切换，切换到另一个线程前会保存自己的状态，以便下一次再切换回这个线程时，可以再加载这个线程的状态。上下文切换是计算密集型的，耗时。

Linux相比其他操作系统的优点之一是：上下文切换和模式切换的时间消耗很少。

# 如何优化线程上下文切换？

线程对锁的竞争会导致线程阻塞，线程阻塞又会导致上下文切换。

* 无锁并发编程：用一些办法来避免使用锁，如将数据的ID按照Hash取模分段，不同线程处理不同段数据。
* CAS算法：Java的Atomic包使用CAS算法来更新数据，而不需加锁
* 使用最少线程：避免创建不需要的线程
* 协程：在单线程里实现多任务的调度，维持多任务间的切换。

# 线程调度策略

Linux下的线程调度策略：

* SCHED\_OTHER：普通任务调度策略，调度器通过动态优先级来决定调用哪个线程，动态优先级随着等待运行但是未被调度执行的时间总量的增长而增加；这样的机制保证了所有SCHED\_OTHER线程调度的公平性。
* SCHED\_FIFO：实时任务调度策略，先到先服务。一直占用cpu运行，直到有更高优先级任务到达或自己放弃。
* SCHED\_RR：实时任务调度策略，时间片轮转。当任务的时间片用完，系统将重新分配时间片，并置于就绪队列尾。

Java的线程调度策略：JVM采用抢占式调度模型。

线程调度器：线程调度器是一个操作系统服务，负责为就绪（Runnable）状态的线程分配CPU时间。

# 死锁是什么？必要条件？如何解决？

死锁是指多个进程持有一定资源并循环等待别的进程已经占有的资源而无限期僵持下去的局面。

## 必要条件

### 互斥条件

某个资源只能被一个进程独占，不能同时被2个及以上的进程占有。

### 不剥夺

不剥夺别的进程占有的资源，只能等该进程自己释放。

### 保持并请求

每个进程都占有一定资源，并请求新的资源。

### 循环等待

各个进程循环等待别的进程占有的资源，形成环状。

## 解决方法

### 死锁预防

#### 破坏互斥条件

互斥性是由资源本身的性质决定的，这种方法不现实。

#### 破坏不剥夺条件

当进程占有了一些资源又请求不到资源时，立刻释放所有资源，待以后再申请，这是一种变相的剥夺方式，此方法增加系统开销，降低系统性能。

#### 破坏请求并保持条件

一次性资源分配，此方法资源被严重浪费。

#### 破坏循环等待条件

采用顺序资源分配法，将资源编号，在进程请求资源时，只能按编号从小到大申请，如果进程实际使用资源顺序与系统资源编号不同，会造成资源浪费。

### 死锁避免

银行家算法，尝试对请求资源的进程进行资源分配，检查系统安全状态，如果处于安全状态则正式分配资源，否则推迟分配。

### 死锁检测与解除

#### 死锁检测

通过资源分配图判断是否死锁，如果资源分配图不能完全化简，则死锁。

#### 死锁解除

剥夺某些进程资源，撤销某些进程，回退某些进程。

# 锁

## 互斥锁

用互斥锁实现对共享资源的保护，当线程对共享资源加互斥锁时，如果该资源没有上锁，则加锁并访问资源，如果该资源已上锁则阻塞。

## 读写锁

读写锁使用于读多写少的场景，读写锁是一把锁，有3个状态，加读锁，加写锁，未加锁。当线程对共享资源加读锁后，别的线程也可以对该资源加读锁（共享读），但不能对该资源加写锁。当线程对共享资源加写锁后，别的线程都不能访问该资源。当线程对共享资源加写锁时，后续线程对共享资源加读锁也会被阻塞，以实现读写公平，否则一直有线程加读锁，写锁永远加不上。

## 递归锁

递归锁又叫可重入锁，线程可以加多次的锁称为递归锁，线程不可以加多次的锁称为非递归锁。

## 条件变量

与互斥锁不同，条件变量是用来等待而不是用来上锁的，条件变量本身不是锁！

条件变量用来自动阻塞一个线程，直到某特殊情况发生为止。通常条件变量和互斥锁同时使用。

条件变量的两个动作：

* 条件不满, 阻塞线程，释放互斥锁
* 当条件满足, 通知阻塞的线程开始工作

条件变量的类型: pthread\_cond\_t。

### 条件变量的优缺点

相较于mutex而言，条件变量可以减少竞争。

如直接使用mutex，除了生产者、消费者之间要竞争互斥量以外，消费者之间也需要竞争互斥量，但如果汇聚（链表）中没有数据，消费者之间竞争互斥锁是无意义的。

有了条件变量机制以后，只有生产者完成生产，才会引起消费者之间的竞争。提高了程序效率。

# 用户态和内核态

内核态与用户态是操作系统的两种运行级别，处于用户态执行时，进程所能访问的内存空间和对象受到限制，其所处于占有的处理机是可被抢占的 ；而处于核心态执行中的进程，则能访问所有的内存空间和对象，且所占有的处理机是不允许被抢占的。

## 什么情况用户态到内核态？

中断，中断包括外中断和内中断，外中断一般是用户输入设备的中断，内中断分为自愿中断（如系统调用）和强迫中断（如硬件故障或程序异常）。

# CPU中断

## CPU中断是什么？

* 计算机处于执行期间；
* 系统内发生了非寻常或非预期的急需处理事件；
* CPU暂时中断当前正在执行的程序而转去执行相应的事件处理程序；
* 处理完毕后返回原来被中断处继续执行；

## CPU中断的作用

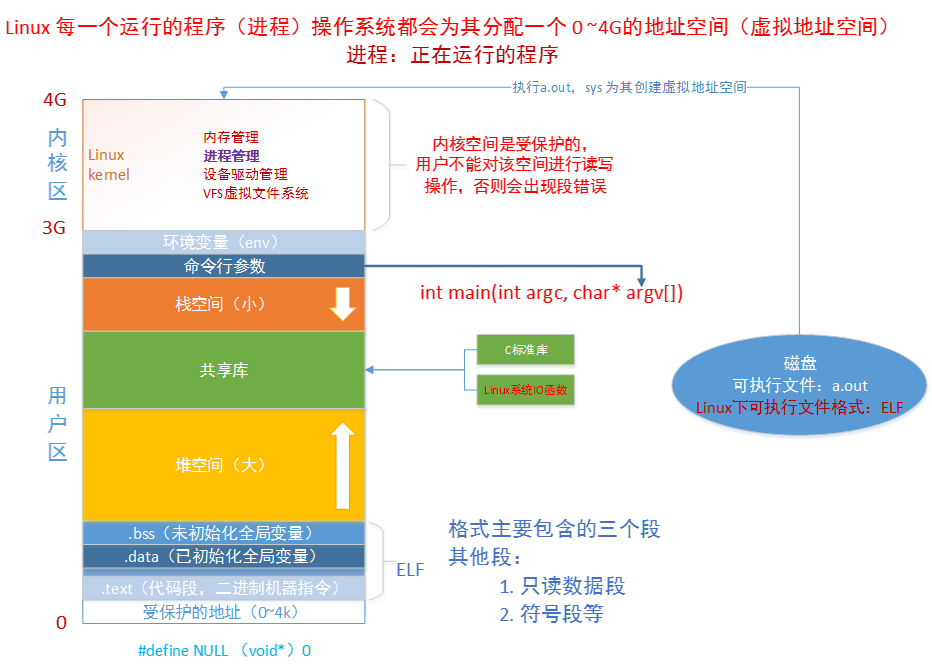
* 可以使CPU和外设同时工作，使系统可以及时地响应外部事件；
* 可以允许多个外设同时工作，大大提高了CPU的利用率；
* 可以使CPU及时处理各种软硬件故障。

# 中断的实现过程

* 关中断，进入不可再次响应中断的状态，由硬件实现。
* 保存断点，为了在[中断处理]结束后能正确返回到中断点。由硬件实现。
* 将[中断服务程序]入口地址送PC，转向[中断服务程序]。可由硬件实现，也可由软件实现。
* 保护现场（保护CPU中某些寄存器的内容）、设置屏蔽字（设置[中断处理]次序）、开中断（允许更高级的中断请求得到响应），实现中断嵌套由软件实现。
* 运行中断服务程序。由软件程序实现。
* 退出中断。

# 虚拟地址空间

每个进程都会分配虚拟地址空间，在32位机器上，该地址空间为4G 。



在进程里平时所说的指针变量，保存的就是虚拟地址。当应用程序使用虚拟地址访问内存时，处理器（CPU）会将其转化成物理地址（MMU）。

MMU(Memory Management Unit)：将虚拟的地址转化为物理地址。

这样做的好处在于：

* 进程隔离，更好的保护系统安全运行。
* 屏蔽物理差异带来的麻烦，方便操作系统和编译器安排进程地址。

.data区包括读写（RW）区域和只读（RO）区域，读写区域存放全局变量和静态变量，只读区域存放字面值常量。

//main.cpp

int a = 0; 全局初始化区

char\* p1; 全局未初始化区

main()

{

int b;// 栈

char s[] = "abc"; //"abc"在常量区，s在栈上。

char\* p2; //栈

char\* p3 = "123456"; //123456\0"在常量区，p3在栈上。

static int c = 0； //全局（静态）初始化区

p1 = (char\*) malloc(10);

p2 = (char\*) malloc(20);

//分配得来得10和20字节的区域就在堆区。

strcpy(p1, "123456");

//123456\0放在常量区，编译器可能会将它与p3所指向的"123456"优化//成一个地方。

}

# gcc/g++工作流程

第一步: 预处理（头文件展开，宏替换，注释去掉）

gcc -E 1hello.c -o 1hello.i

第二步: 编译，生成汇编文件（c文件->汇编文件）

gcc -S 1hello.i -o 1hello.s

第三步: 汇编，生成目标代码（汇编文件->二进制文件）

gcc -c 1hello.s -o 1hello.o

第四步: 链接，生成可以执行文件（将函数库中相应的代码组合到二进制（目标）文件中）

gcc 1hello.o -o 1hello

# 静态链接和动态链接

## 静态链接

由链接器在链接时将库的内容加入到可执行程序中。优点是对运行环境的依赖性较小，具有较好的兼容性，缺点是生成的程序比较大，在装入内存时会消耗更多的时间，库函数有了更新，必须重新编译应用程序。

## 动态链接

链接器在链接时仅仅建立与所需库函数的之间的链接关系，在程序运行时才将所需资源调入可执行程序。优点是在需要的时候才会调入对应的资源函数，有着较小的程序体积，缺点是严重依赖动态库，不能独立运行。

前面我们编写的应用程序大量用到了标准库函数，系统默认采用动态链接的方式进行编译程序，若想采用静态编译，加入-static参数。

# 分页、分段和段页式

## 什么是分页？

把内存空间划分为大小相等且固定的块，作为主存的基本单位。因为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映射关系，以实现从页号到物理块号的映射。

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)。

## 什么是分段？

分段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，二维是段内地址；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。

## 什么是段页式？

段页式就是先分段再分页。

## 分页和分段有什区别？

* 分页不需要程序员操作，但是分段需要程序员划分。
* 分页的地址空间是一维的（页号和页内偏移，由于页大小固定，可以根据页面大小从一个地址中分割出页号和页内偏移），分段是二维的（段号和段内地址，段大小不固定，所以一个地址是不能分割出段号和段内地址的，所以是二维的）。
* 页的大小不可变，段的大小可以动态改变。
* 分页主要用于实现虚拟内存，从而获得更大的地址空间；分段主要是为了使程序和数据可以被划分为逻辑上独立的地址空间并且有助于共享和保护。
* 现在的操作系统分页是主流。

# 内存碎片怎么理解？

固定大小的分片方式都有内部碎片没有外部碎片。页式和段页式只有内部碎片没有外部碎片，段式只有外部碎片没有内部碎片。

## 内部碎片

内部碎片是由于采用固定大小的内存分区，当一个进程不能完全使用分给它的固定内存区域时就会产生内部碎片。通常内部碎片难以完全避免。

## 外部碎片

外部碎片是由于采用动态划分内存的方法，由于进程与进程之间的空间太小而产生的碎片。

# 虚拟内存

虚拟内存，它会使进程认为自己拥有一块很大且连续的内存，使用外存（硬盘）空间来扩充内存空间，通过一定的换入/换出，使得进程在逻辑上拥有一块很大且连续的远超物理内存的内存。

## 优缺点

* 优点：可以弥补物理内存大小的不足。
* 缺点：占用一定的物理硬盘空间；加大了对硬盘的读写。

## 虚拟地址空间

虚拟地址空间是对于一个单一进程的概念，这个进程看到的将是地址从0000开始的整个内存空间。虚拟存储器是一个抽象概念，它为每一个进程提供了一个假象，好像每一个进程都在独占的使用主存。每个进程看到的存储器都是一致的，称为虚拟地址空间。从最低的地址看起：程序代码和数据，堆，共享库，栈，操作系统内核。32位计算机虚拟地址空间为4GB。

## 硬件支持

* 一定容量的内存和外存。
* 页表机制（或段表机制）。
* 中断机构，如果需要访问的数据不在内存，需要产生中断将数据从外存调入内存。
* 地址变换机构，从逻辑地址转化为物理地址。

## 虚拟内存大小限制

* 小于等于内存+外存容量
* 小于等于计算机地址位数能容纳的最大容量，32位地址虚拟内存<=4GB

# 页面置换算法

## 为什么要页面置换？

因为虚拟内存的使用，应用程序是分多次装入内存的，所以运行到一定的时间，一定会发生缺页。地址映射的过程中，如果页面中发现要访问的页面不在内存中，会产生缺页中断。此时操作系统必须在内存里选择一个页面把他移出内存，为即将调入的页面让出空间。选择淘汰哪一页的规则就是页面置换算法。

## 最佳置换算法（理想）

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。

## 先进先出

淘汰最早调入的页面。

## 最近最久未使用LRU

每个页面有一个t来记录上次页面被访问直到现在，每次置换时置换t值最大的页面（用寄存器或栈实现）。

## 时钟算法clock

页面设置访问为，将页面链接为一个环形列表，每个页有一个访问位0/1, 1表示又一次获救的机会，下次循环指针指向它时可以免除此次置换，但是会把访问位置为0， 代表他下次如果碰到循环指针就该被置换了。页面被访问的时候访问位设为1。页面置换的时候，如果当前指针的访问位为0，置换，否则将这个值置为0，循环直到遇到访问位为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，优先替换访问位和修改位都是0的页面，其次替换访问位为0修改位为1的页面。

最少使用算法LFU：设置寄存器记录页面被访问次数，每次置换当前访问次数最少的。

# mutex底层原理

pthread\_mutex在使用futex之前是在内核维护一个对象，这个对象对所有线程可见，这个对象用来维护管理互斥锁和通知阻塞的线程。如果线程A想要进入临界区，则需要先到内核中访问这个对象，如果有别的线程占用临界区则阻塞等待，如果没有则进入临界区，A出临界区的时候，如果有别的线程阻塞等待进入临界区则唤醒该线程。

## futex的意义

使用futex之前，如果没有多个线程竞争资源，也需要进入内核访问管理互斥锁的对象，系统开销较大。引入futex后，创建mutex时通过mmap申请一片共享内存存放futex对象，当线程A想要进入临界区时，先访问futex对象，如果有别的线程占用了该futex对象，则进入内核阻塞等待唤醒。