目录

[1. 进程、线程与协程 3](#_Toc66210627)

[2. 零拷贝技术 5](#_Toc66210628)

[3. 伙伴系统 7](#_Toc66210629)

[4. 操作系统的四大特性：并发，共享，异步，虚拟。 7](#_Toc66210630)

[5. 虚拟内存提供了三个重要的能力： 7](#_Toc66210631)

[6. 说一说Linux虚拟地址空间 7](#_Toc66210632)

[7. 虚拟内存页表 8](#_Toc66210633)

[8. 缓冲（cache）和缓存(buffer)的区别 9](#_Toc66210634)

[9. 操作系统中的程序的内存结构 10](#_Toc66210635)

[10. 为什么要划分初始化的全局变量和未初始化的全局变量 11](#_Toc66210636)

[11. 缺页中断 11](#_Toc66210637)

[12. 进程同步（四种） 12](#_Toc66210638)

[13. fork和vfork的区别 13](#_Toc66210639)

[14. 有了进程，为什么还要有线程？ 15](#_Toc66210640)

[15. 单核机器上写多线程程序，是否需要考虑加锁，为什么？ 15](#_Toc66210641)

[16. 请问线程需要保存哪些上下文，SP、PC、EAX这些寄存器是干嘛用的 15](#_Toc66210642)

[17. 线程间的同步方式，最好说出具体的系统调用 16](#_Toc66210643)

[18. 死锁发生的条件以及如何解决死锁 17](#_Toc66210644)

[19. 死锁避免 17](#_Toc66210645)

[20. 操作系统中的结构体对齐，字节对齐 17](#_Toc66210646)

[21. 虚拟内存置换的方式 18](#_Toc66210647)

[22. 互斥锁（mutex）机制，以及互斥锁和读写锁的区别 19](#_Toc66210648)

[23. 进程状态转换图，动态就绪，静态就绪，动态阻塞，静态阻塞 20](#_Toc66210649)

[24. A\* a = new A; a->i = 10;在内核中的内存分配上发生了什么？ 20](#_Toc66210650)

[25. 给你一个类，里面有static，virtual，之类的，来说一说这个类的内存分布 21](#_Toc66210651)

[26. 软链接和硬链接 22](#_Toc66210652)

[27. 什么是大端小端以及如何判断大端小端 23](#_Toc66210653)

[28. core dump（核心转储） 23](#_Toc66210654)

[29. 提高磁盘IO性能的方法： 23](#_Toc66210655)

[30. 静态变量什么时候初始化 23](#_Toc66210656)

[31. 什么是原子操作 24](#_Toc66210657)

[32. 如何设计server，使得能够接收多个客户端的请求：多线程，线程池，io复用。 24](#_Toc66210658)

[33. 缓存IO、直接IO、内存映射 24](#_Toc66210659)

[34. 内存溢出和内存泄漏 26](#_Toc66210660)

[35. 常用线程模型 26](#_Toc66210661)

[36. 协程 27](#_Toc66210662)

[37. 作业调度算法 27](#_Toc66210663)

[38. linux 进程地址空间分布 28](#_Toc66210664)

[39. 说一下源码到可执行文件的过程 28](#_Toc66210665)

[40. 微内核与宏内核 30](#_Toc66210666)

[41. 孤儿进程、僵尸进程和守护进程 30](#_Toc66210667)

[42. 介绍一下5种IO模型 32](#_Toc66210668)

[43. 为什么要有page cache，操作系统怎么设计的page cache 32](#_Toc66210669)

[44. 怎么实现线程池 32](#_Toc66210670)

[45. 线程的哪些资源共享，哪些资源不共享 32](#_Toc66210671)

[46. 怎么理解操作系统里的内存碎片，有什么解决办法？ 33](#_Toc66210672)

[47. 系统如何提高并发性 33](#_Toc66210673)

[48. 进程 33](#_Toc66210674)

[49. Linux 内核的同步方式 35](#_Toc66210675)

[50. 文件系统 36](#_Toc66210676)

[51. 页面置换算法 36](#_Toc66210677)

[52. 内核态(内核空间)和用户态(用户空间)的区别和联系 36](#_Toc66210678)

[53. IO多路复用 38](#_Toc66210679)

[54. 共享内存实现 42](#_Toc66210680)

[55. Reactor和Proactor 42](#_Toc66210681)

[56. malloc函数的实现 43](#_Toc66210682)

# 进程、线程与协程

**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；线程是**操作系统可识别的最小执行和调度单位**。每个线程都独自占用一个虚拟处理器：独自的寄存器组，指令计数器和处理器状态。每个线程完成不同的任务，但是共享同一地址空间（也就是同样的动态内存，映射文件，目标代码等等），打开的文件队列和其他内核资源。

**协程，又称微线程，也称用户态线程**，它与线程最本质的不同是**协程是用户态的，线程是操作系统内核态的**。协程看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。

1. 进程和线程的区别

1）一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。线程依赖于进程而存在。

2）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。（资源分配给进程，同一进程的所有线程共享该进程的所有资源。同一进程中的多个线程共享代码段（代码和常量），数据段（全局变量和静态变量），扩展段（堆存储）。但是**每个线程拥有自己的栈段**，**栈段又叫运行时段，用来存放所有局部变量和临时变量**。）

3）进程是资源分配的最小单位，线程是CPU调度的最小单位。

4）系统开销：由于在创建或撤消进程时，系统都要为之分配或回收资源，如内存空间、I/O设备等。因此，操作系统所付出的开销将显著地大于在创建或撤消线程时的开销。类似地，在进行进程切换时，涉及到整个当前进程CPU环境的保存以及新被调度运行的进程的CPU环境的设置。而线程切换只须保存和设置少量寄存器的内容，并不涉及存储器管理方面的操作。可见，**进程切换的开销也远大于线程切换的开销**。

5）通信：由于同一进程中的多个线程具有相同的地址空间，致使它们之间的同步和通信的实现，也变得比较容易。线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。在有的系统中，线程的切换、同步和通信都无须操作系统内核的干预。

6）进程编程调试简单可靠性高，但是创建销毁开销大；线程正相反，开销小，切换速度快，但是编程调试相对复杂。

7）进程间不会相互影响；但是一个线程崩溃整个进程都死掉。所以多进程要比多线程健壮。

8）进程适应于多核、多机分布；线程适用于多核。

1. 进程间六种通信方式总结

1）匿名管道：单向，**速度慢**，**容量有限**，只有父子进程能通讯。

2）有名管道FIFO：任何进程间都能通讯，但速度慢。

3）消息队列：**通信不及时**，**容量受到系统限制**，需要在用户态和内核态之间进行数据拷贝。且要注意第一次读的时候，要考虑上一次没有读完数据的问题。

4）共享内存区：能够很容易控制容量，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全，当然，共享内存区同样可以用作线程间通讯，不过没这个必要，线程间本来就已经共享了同一进程内的一块内存。

5）信号量：不能传递复杂消息，只能用来同步。

6）信号：信号是进程间通信机制中**唯一的异步通信机制**，对于**异常情况**下的工作模式，需要用信号的方式来通知进程。

7）Socket通信不仅可以跨网络与不同主机的进程通信，还可以在同主机上进程间通信。

1. 线程之间的通信方式

同个进程下的线程之间都是共享进程的资源，只要是共享变量都可以做到线程间通信，比如全局变量，所以对于线程间关注的不是通信方式，而是关注多线程竞争共享资源的问题，信号量也同样可以在线程间实现互斥与同步：

* 互斥的方式，可保证任意时刻只有一个线程访问共享资源。
* 同步的方式，可保证线程A应在线程B之前执行。

1）**锁机制**：包括互斥锁/量（mutex）、读写锁（reader-writer lock）、自旋锁（spin lock）、条件变量（condition）

* 互斥锁/量（mutex）：提供了以排他方式防止数据结构被并发修改的方法。
* 读写锁（reader-writer lock）：允许多个线程同时读共享数据，而对写操作是互斥的。
* 自旋锁（spin lock）与互斥锁类似，都是为了保护共享资源。互斥锁是当资源被占用，申请者进入睡眠状态；而自旋锁则循环检测保持者是否已经释放锁。
* 条件变量（condition）：可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。

2）**信号量机制(Semaphore)**

信号量是一个特殊类型的变量，可对其进行增加或者减少操作，对其的操作是原子操作。sem\_t：1.无名线程信号量。2.命名线程信号量。

3）**条件变量pthread\_cond\_t**

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。条件变量分为两部分: 条件和变量。条件本身是由互斥量保护的。线程在改变条件状态前先要锁住互斥量。条件变量使我们可以睡眠等待某种条件出现。条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待"条件变量的条件成立"而挂起；另一个线程使"条件成立"（给出条件成立信号）。条件的检测是在互斥锁的保护下进行的。如果一个条件为假，一个线程自动阻塞，并释放等待状态改变的互斥锁。如果另一个线程改变了条件，它发信号给关联的条件变量，唤醒一个或多个等待它的线程，重新获得互斥锁，重新评价条件。如果两进程共享可读写的内存，条件变量可以被用来实现这两进程间的线程同步。

**4）信号机制(Signal)**：类似进程间的信号处理。

5）**屏障（barrier）**：屏障允许每个线程等待，直到所有的合作线程都达到某一点，然后从该点继续执行。

**线程间的通信目的主要是用于线程同步，所以线程没有像进程通信中的用于数据交换的通信机制。**

1. 进程之间私有和共享的资源

* 私有：地址空间、堆、全局变量、栈、寄存器。
* 共享：代码段，公共数据，进程目录，进程ID。

1. 线程之间私有和共享的资源

* 私有：线程栈，寄存器，程序计数器
* 共享：堆，地址空间，全局变量，静态变量

1. 多进程与多线程间的对比、优劣与选择
   1. 对比

| **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 |

* 1. 优劣

| **优劣** | **多进程** | **多线程** |
| --- | --- | --- |
| 优点 | 编程、调试简单，可靠性较高 | 创建、销毁、切换速度快，内存、资源占用小 |
| 缺点 | 创建、销毁、切换速度慢，内存、资源占用大 | 编程、调试复杂，可靠性较差 |

* 1. 进程和线程的选择
* 需要频繁创建销毁的优先用线程。
* 需要进行大量计算的优先使用线程。
* 强相关的处理用线程，弱相关的处理用进程。
* 可能要扩展到多机分布的用进程，多核分布的用线程。

# 零拷贝技术

零拷贝可以在某种程度上减少甚至完全避免不必要 CPU 数据拷贝操作。这样就可以让系统资源的利用更加有效。从技术实现上来讲，零拷贝不是真的一次拷贝都没有，而是取消了用户缓冲区的拷贝。但是内核态还是需要将磁盘文件内容拷贝到pageCache，然后再将内容从pageCache拷贝到socket缓冲区。零拷贝技术有可能会产生很多负面的影响，甚至会导致零拷贝技术自身的优点完全丧失。

（1）零拷贝技术的目标可以概括如下：

1）避免数据拷贝

* 避免操作系统内核缓冲区之间进行数据拷贝操作。
* 避免操作系统内核和用户应用程序地址空间这两者之间进行数据拷贝操作。
* 用户应用程序可以避开操作系统直接访问硬件存储。
* 数据传输尽量让DMA来做。

2）将多种操作结合在一起

* 避免不必要的系统调用和上下文切换。
* 需要拷贝的数据可以先被缓存起来。
* 对数据进行处理尽量让硬件来做。

（2）零拷贝技术分类

* **直接I/O**：对于这种数据传输方式来说，应用程序可以直接访问硬件存储，操作系统内核只是辅助数据传输：这类零拷贝技术针对的是操作系统内核并不需要对数据进行直接处理的情况，数据可以在应用程序地址空间的缓冲区和磁盘之间直接进行传输，完全不需要Linux操作系统内核提供的页缓存的支持。
* **在数据传输的过程中，避免数据在操作系统内核地址空间的缓冲区和用户应用程序地址空间的缓冲区之间进行拷贝**。有的时候，应用程序在数据进行传输的过程中不需要对数据进行访问，那么，将数据从Linux的页缓存拷贝到用户进程的缓冲区中就可以完全避免，传输的数据在页缓存中就可以得到处理。在某些特殊的情况下，这种零拷贝技术可以获得较好的性能。Linux中提供类似的系统调用主要有**mmap()，sendfile()以及splice()**。
* **对在页缓存和用户进程的缓冲区之间的传输过程进行优化**。该零拷贝技术侧重于灵活地处理在用户进程的缓冲区和操作系统的页缓存之间的拷贝操作。这种方法延续了传统的通信方式，但是更加灵活。在Linux中，该方法主要利用了**写时复制技术**。

**前两类方法的目的主要是为了避免应用程序地址空间和操作系统内核地址空间缓冲区之间的拷贝操作。**这两类零拷贝技术通常适用在某些特殊的情况下，比如要传送的数据不需要经过操作系统内核的处理或者不需要经过应用程序的处理。**第三类**方法则继承了传统的应用程序地址空间和操作系统内核地址空间之间数据传输的概念，进而**针对数据传输本身进行优化。**我们知道，硬件和软件之间的数据传输可以通过使用DMA来进行，DMA进行数据传输的过程中几乎不需要CPU参与，这样就可以把CPU解放出来去做更多其他的事情，但是当数据需要在用户地址空间的缓冲区和操作系统内核的页缓存之间进行传输的时候，并没有类似DMA这种工具可以使用，CPU需要全程参与到这种数据拷贝操作中，所以这第三类方法的目的是可以有效地改善数据在用户地址空间和操作系统内核地址空间之间传递的效率。

**大文件交给异步IO和直接IO处理，小文件交给零拷贝处理：**但是在高并发场景处理大文件时，应当使用异步IO和直接IO来替换零拷贝技术。因为**PageChache不适应传输大文件的场景，大文件容易把PageCache占满**，而且由于文件太大，文件中某一个部分被再次访问的概率低。这样会导致大文件在PageCache中没有享受到缓存的优势，同时也因为PageCache被大文件占据，影响其他热点小文件的缓存。异步IO可以把读操作分为两部分，前半部分向内核发起读请求，但不用等待数据就位就返回，然后可以继续处理其他任务。当内核把磁盘中的数据拷贝到进程缓冲区后，会通知进程去处理数据。异步IO是不会阻塞用户进程的。对于磁盘，异步IO只支持直接IO。直接IO是应用程序绕过PageCache，即不经过内核缓冲区，直接访问磁盘中的数据，从而减少了内核缓存与用户程序之间的数据拷贝。

# 伙伴系统

在Linux系统中，内存的分配与回收速率直接影响系统的存取效率。当内核频繁请求和释放不同大小的一组连续页框时，会导致许多**外部空闲碎片**，造成空间的浪费。使用伙伴算法可以有效地缓解该问题。伙伴关系机制是操作系统中的一种动态存储管理算法。在进行内存分配时，该算法通过不断平分较大的空闲内存块来获得较小的空闲内存块，直到获得所需要的内存块；在进行内存回收时，该算法尽可能地合并空闲块。

伙伴系统算法：内核为分配一组连续的页框而建立的一种健壮、高效的分配策略，这种策略缓解了内存碎片的发生。算法的核心思想：是把所有的空闲页框分组为11个块链表，每个块链表分别包含1、2、4、8、16、...、512、1024个连续页框。举个简单的例子，说明算法的工作过程。

假设需要256个页框的连续内存，算法先在256个页框的链表中，检查是否还有空闲块，如果有就分配出去。如果没有，算法会找到下一个更大的512页框的链表，如果存在空闲块，内核会把512页框分割成两部分，一半用来分配，另一半插入到256页框的链表中。

# 操作系统的四大特性：并发，共享，异步，虚拟。

**并发：**是指两个或多个事件在同一时间间隔内发生。

**共享：**是指系统中的资源（硬件资源和信息资源）可以被多个并发执行的程序共同使用，而不是被其中一个独占。资源共享有两种方式：互斥访问和同时访问。

**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，以不可预知的速度向前推进，这就是进程的异步性。

**虚拟：**虚拟性是一种管理技术，把物理上的一个实体变成逻辑上的多个对应物，或把物理上的多个实体变成逻辑上的一个对应物的技术。采用虚拟技术的目的是为用户提供易于使用、方便高效的操作环境。

# 虚拟内存提供了三个重要的能力：

* + - **将主存看成是一个存储在磁盘上的地址空间的高速缓存**，在主存中只保存活动区域，并根据需要在磁盘和主存之间来回传递数据，通过这种方式，它高效地使用了主存。
    - 为**每个进程提供了一致的地址空间，从而简化了内存管理。**
    - **保护了每个进程的地址空间不被其他进程破坏**。

# 说一说Linux虚拟地址空间

为了防止不同进程同一时刻在物理内存中运行而对物理内存的争夺和践踏，采用了虚拟内存。所有进程共享同一物理内存，每个进程只把自己目前需要的虚拟内存空间映射并存储到物理内存上。事实上，**在每个进程创建加载时，内核只是为进程“创建”了虚拟内存的布局，**具体就是初始化进程控制表中内存相关的链表，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和.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），等到运行到对应的程序时，才会通过缺页异常，来拷贝数据。还有进程运行过程中，要动态分配内存，比如**malloc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。**

请求分页系统、请求分段系统和请求段页式系统都是针对虚拟内存的，通过请求实现内存与外存的信息置换。

* 1. 虚拟内存的好处

1.**扩大地址空间。**

2.**内存保护**：每个进程运行在各自的虚拟内存地址空间，互相不能干扰对方。虚存还对特定的内存地址提供写保护，可以防止代码或数据被恶意篡改。

3.**公平内存分配**。**采用了虚存之后，每个进程都相当于有同样大小的虚存空间。**

4.当进程通信时，可采用**虚存共享**的方式实现。

5.**节省内存：**当不同的进程使用同样的代码时，比如库文件中的代码，物理内存中可以只存储一份这样的代码，不同的进程只需要把自己的虚拟内存映射过去就可以了。

6.虚拟内存很适合在多道程序设计系统中使用，许多程序的片段同时保存在内存中。当一个程序等待它的一部分读入内存时，可以把CPU交给另一个进程使用。在内存中可以保留多个进程，**系统并发度提高**。

7.**可以利用碎片：**在程序需要分配连续的内存空间的时候，只需要在虚拟内存空间分配连续空间，而不需要实际物理内存的连续空间。

* 1. 虚拟内存的代价

1.虚存的管理需要建立很多数据结构，这些数据结构要占用额外的内存。

2.虚拟地址到物理地址的转换，增加了指令的执行时间。

3.页面的换入换出需要磁盘I/O，这是很耗时的。

4.如果一页中只有一部分数据，会浪费内存（内部碎片）。

# 虚拟内存页表

**一级页表：**

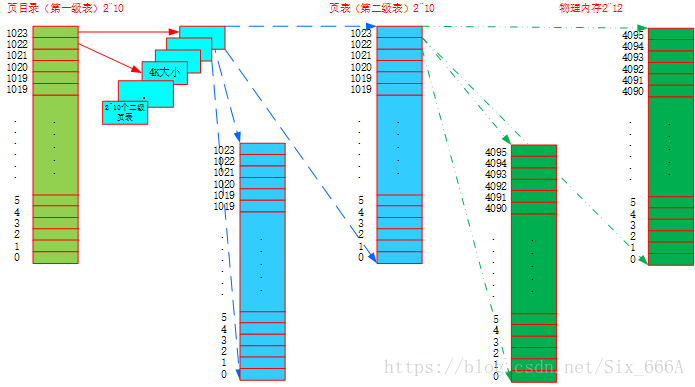


Fig.1 二级页表

（以32位操作系统为例）比如对于4G的空间，4K页, 那需要2^20 == 1M个页标项

1M ＊ 4 ＝＝ 4M，相当于每个进程都要4M的连续内存。

**二级页表：**

每个进程最多需要一个页目录（4K） ＋ 4K\*2^10个页表=4M+4K

每个进程最少需要一个页目录（4K） ＋ 4K\*2^0个页表=4K+4K=8K

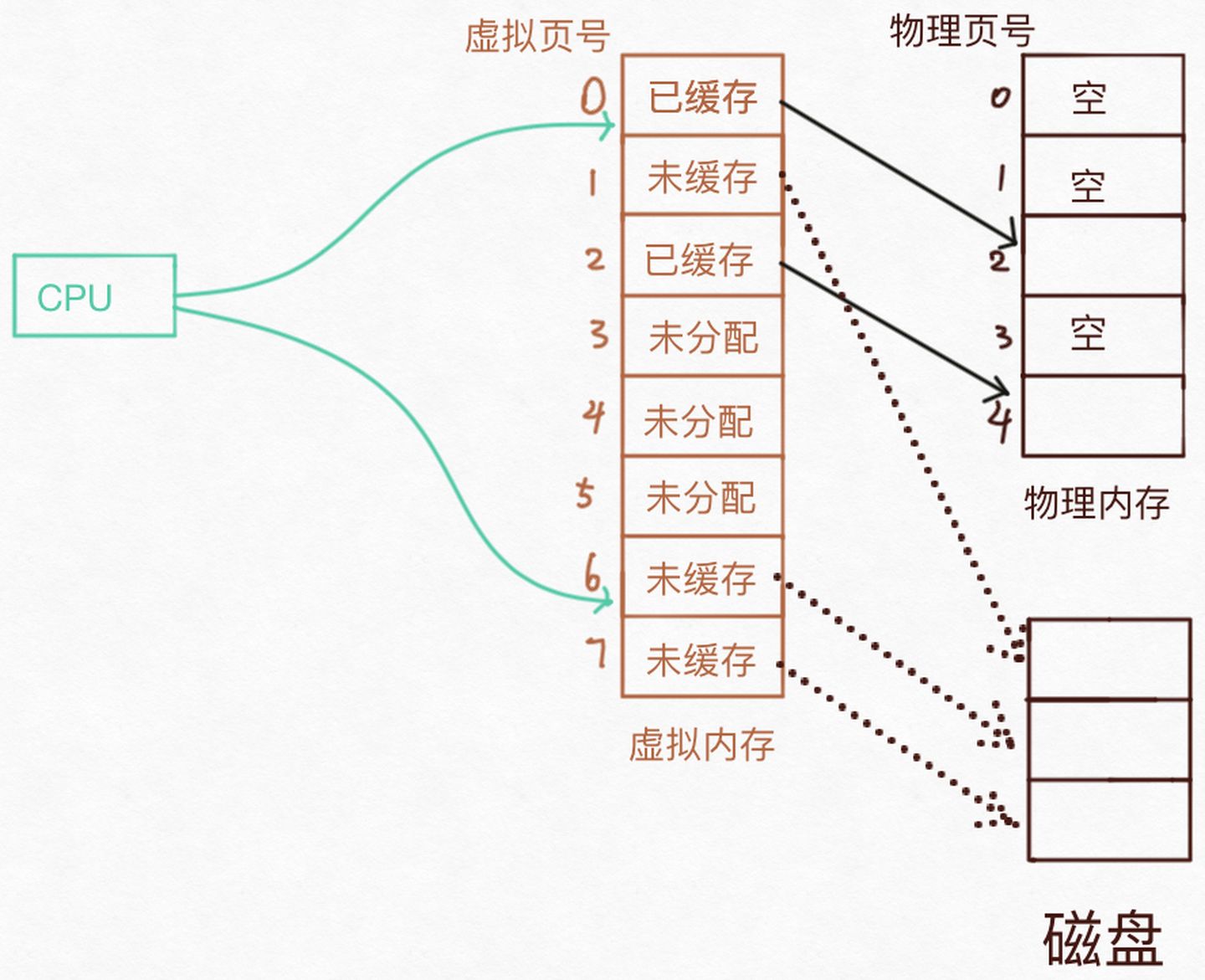


Fig.2 虚拟内存映射

**一级页表的两个问题：**

1.页表在内存分配是必须连续，否则无法查表。

2.每个进程都必须实实在在占用4M，必须全部分配，进程才可以使用。

**二级页表的优势：**

1.允许页表被分散在内存的各个页面中，不需要连续的4M内存块；

2.并不需要为不存在的或线性地址空间未使用部分分配二级页表；

3.可以在虚拟内存中存放二级页表。

**虚拟地址映射**

分页就是把整个虚拟内存和物理内存分割成大小固定的块，以一个页作为映射的最小单位。运行时，CPU使用虚拟地址向内存寻址，通过专用的内存管理单元（MMU）硬件把虚拟地址转换为真实的物理地址（地址翻译），操作系统负责把虚拟地址和物理地址的映射关系维护在页表之中。

当CPU寻址的时候，这个映射会有三种可能。

* 未分配：虚拟地址所在的那一页并未被分配，代表没有数据和他们关联，这部分也不会占用内存。
* 未缓存：虚拟地址所在的那一页被分配了，但并不在内存中。
* 已缓存：虚拟地址所在的那一页就在内存中。

当访问一个未缓存的区域时，系统将产生缺页中断，然后进程被阻塞，等待操作系统将缺失的那一页从磁盘复制到内存。当复制完成后，CPU继续执行导致缺页中断的那条指令，此时就会正常执行了。这种仅在需要的时候将页面拷贝到内存的策略叫做按需调度页面。 可以想象当程序被装入内存的时候，开始时仅有有很小的一部分内容被放入内存。程序在运行中不断缺页，不断的把需要的部分拷贝进内存。

从进程的视角来看，我的数据和代码被存放在一个连续的空间之中，每个区域分别有着不同的功能。典型的如存放代码的区域和存放数据的区域。但是，代码和数据中的地址都是一个虚拟地址，还需要经过地址翻译才能得到真正的物理地址。

Malloc()和mmap（）等内存分配函数，在分配时只是建立了进程虚拟地址空间，并没有分配虚拟内存对应的物理内存。当进程访问这些没有建立映射关系的虚拟内存时，处理器自动触发一个缺页中断，会导致内核实际去分配物理内存，并更新页表映射关系。

# 缓冲（cache）和缓存(buffer)的区别

用户进程通过系统调用访问系统资源的时候，需要切换到**内核态**，而这对应一些特殊的堆栈和内存环境，必须在系统调用前建立好。而在系统调用结束后，CPU会从核心模式切回到用户模式，而堆栈又必须恢复成用户进程的上下文。而这种切换就会有大量的耗时。

**用户进程缓冲区(buffer)：**

你看一些程序在读取文件时，会先申请一块内存数组，称为buffer，然后每次调用read，读取设定字节长度的数据，写入buffer。（用较小的次数填满buffer）。之后的程序都是从buffer中获取数据，当buffer使用完后，在进行下一次调用，填充buffer。

所以说：**用户缓冲区的目的是为了减少系统调用次数，从而降低操作系统在用户态与核心态切换所耗费的时间。除了在进程中设计缓冲区，内核也有自己的缓冲区（buffer）**。

**内核缓冲区（buffer）：**

当一个用户进程要从磁盘读取数据时，内核一般不直接读磁盘，而是将内核缓冲区中的数据复制到进程缓冲区中。但若是内核缓冲区中没有数据，内核会把对数据块的请求，加入到请求队列，然后把进程挂起，为其它进程提供服务。等到数据已经读取到内核缓冲区时，把内核缓冲区中的数据读取到用户进程中，才会通知进程，当然不同的io模型，在调度和使用内核缓冲区的方式上有所不同。

你可以认为，read是把数据从内核缓冲区复制到进程缓冲区。write是把进程缓冲区复制到内核缓冲区。当然，write并不一定导致内核的写动作，比如os可能会把内核缓冲区的数据积累到一定量后，再一次写入。这也就是为什么断电有时会导致数据丢失。所以说内核缓冲区，是为了在OS级别，提高磁盘IO效率，优化磁盘写操作。

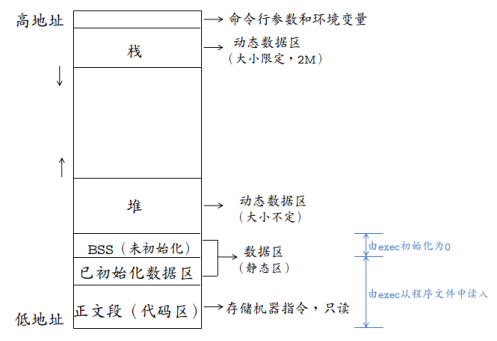
**内核缓存区（cache）：**cache叫做高速缓冲存储器，是介于中央处理器和主存储器之间的高速小容量存储器，他的实现主要利用了程序的时间局部性和空间局部性。

**时间局部性：**如果执行了程序中的某条指令，那么不久后这条指令很有可能再次执行；如果某个数据被访问过，不久之后该数据很可能再次被访问。（程序中存在大量的循环）

**空间局部性：**一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也很有可能被访问（因为横夺数据在内存中都是连续存放的，并且程序的指令也是顺序的在内存中存放的）

注意**内核缓存区**（cache）和**内核缓冲区**（buffer）的区别！！！：内核缓存区（cache）是位于CPU与内存之间的临时存储器，因为CPU的计算速度要比内存的读写速度块很多，而把这些可能会被重复访问到的数据存储与CPU缓存中，就会提高读取速度。而内核缓冲区任然是存在内存上的。

# 操作系统中的程序的内存结构



一个程序本质上都是由BSS段、data段、text段三个组成的。可以看到一个可执行程序在存储（没有调入内存）时分为代码段、数据区和未初始化数据区三部分。

**BSS段（未初始化数据区）**：通常用来存放程序中未初始化的全局变量和静态变量的一块内存区域。**BSS段属于静态分配**，程序结束后静态变量资源由系统自动释放。

**数据段（**data**）**：存放程序中已初始化的全局变量的一块内存区域。**数据段也属于静态内存分配。**

**代码段（**text**）**：存放程序代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域属于**只读**。在代码段中，也有可能包含一些只读的常数变量。

text段和data段在编译时已经分配了空间，而BSS段并不占用可执行文件的大小，它是由链接器来获取内存的。

**BSS段**的内容并不存放在磁盘上的程序文件中。其原因是内核在程序开始运行前将它们设置为0。需要存放在程序文件中的只有正文段和初始化数据段。

data段（已经初始化的数据）则为数据分配空间，数据保存到目标文件中。

数据段包含经过初始化的全局变量以及它们的值。BSS段的大小从可执行文件中得到，然后链接器得到这个大小的内存块，紧跟在数据段的后面。当这个内存进入程序的地址空间后全部清零。包含数据段和BSS段的整个区段此时通常称为数据区。

可执行程序在运行时又多出两个区域：栈区和堆区。

**栈区**：由编译器自动释放，存放函数的**参数值、局部变量**等。每当一个函数被调用时，该函数的返回类型和一些调用的信息被存放到栈中。然后这个被调用的函数再为他的自动变量和临时变量在栈上分配空间。每调用一个函数一个新的栈就会被使用。栈区是从高地址位向低地址位增长的，是一块连续的内存区域，最大容量是由系统预先定义好的，申请的栈空间超过这个界限时会提示溢出，用户能从栈中获取的空间较小。

**堆区**：**用于动态分配内存**，位于BSS和栈中间的地址区域。由程序员申请分配和释放。堆是从低地址位向高地址位增长，采用链式存储结构。频繁的malloc/free造成内存空间的不连续，产生碎片。当申请堆空间时库函数是按照一定的算法搜索可用的足够大的空间。因此堆的效率比栈要低的多。

# 为什么要划分初始化的全局变量和未初始化的全局变量

C规定，未初始化变量的初值为0，这个清0的操作是由**启动代码**完成的，还有已初始化变量的初值的设置，也是由启动代码完成的。

**为了启动代码的简单化**，编译链接器会把已初始化的变量放在同一个段：.data，这个段的映像（包含了各个变量的初值）保存在“只读数据段”，这样启动代码就可以简单地复制这个映像到.data段，所有的已初始化变量就都初始化了。

而未初始化变量也放在同一个段：.bss启动代码简单地调用memset就可以把所有未初始化变量都清0。

# 缺页中断

**malloc()和mmap()等内存分配函数，在分配时只是建立了进程虚拟地址空间，并没有分配虚拟内存对应的物理内存**。当进程访问这些没有建立映射关系的虚拟内存时，处理器自动触发一个缺页异常。

**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是，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。

但是缺页中断是由于所要访问的页面不存在于内存时，由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）缺页中断返回是执行产生中断的那条指令，而一般的中断返回是，执行下一条指令。

# 进程同步（四种）

* + 1. **临界区（Critical Section）**

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。

优点：保证在某一时刻只有一个线程能访问数据的简便办法。

缺点：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。

* + 1. **互斥量（Mutex）:为协调共同对一个共享资源的单独访问而设计的。**

互斥量跟临界区很相似，比临界区复杂，互斥对象只有一个，只有拥有互斥对象的线程才具有访问资源的权限。

优点：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。

缺点：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。

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，信号量对象可以说是一种资源计数器。

* + 1. **信号量（Semaphore）:为控制一个具有有限数量用户资源而设计。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。互斥量是信号量的一种特殊情况，当信号量的最大资源数=1就是互斥量了。**

优点：适用于对Socket（套接字）程序中线程的同步。（例如，网络上的HTTP 服务器要对同一时间内访问同一页面的用户数加以限制，只有不大于设定的最大用户数目的线程能够进行访问，而其他的访问企图则被挂起，只有在有用户退出对此页面的访问后才有可能进入。）

缺点：信号量机制必须有公共内存，不能用于分布式操作系统，这是它最大的弱点；信号量机制功能强大，但使用时对信号量的操作分散，而且难以控制，读写和维护都很困难，加重了程序员的编码负担；核心操作P-V分散在各用户程序的代码中，不易控制和管理，一旦错误，后果严重，且不易发现和纠正。

* + 1. **事件（Event）: 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。**

优点：事件对象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。

缺点：临界区不是内核对象，只能用于进程内部的线程同步，是用户方式的同步。互斥、信号量是内核对象可以用于不同进程之间的线程同步（跨进程同步）。

互斥其实是信号量的一种特殊形式。互斥可以保证在某一时刻只有一个线程可以拥有临界资源。信号量可以保证在某一时刻有指定数目的线程可以拥有临界资源。

# fork和vfork的区别

* + 1. fork的基础知识

fork:创建一个和当前进程映像一样的进程可以通过fork( )系统调用：

pid\_t fork(void);

成功调用fork会创建一个新的进程，它几乎与调用fork的进程一模一样，这两个进程都会继续运行。在子进程中，成功的fork调用会返回0。在父进程中fork返回子进程的pid。如果出现错误，fork返回一个负值。

最常见的fork用法是创建一个新的进程，然后使用exec载入二进制映像，替换当前进程的映像。这种情况下，派生了新的进程，而这个子进程会执行一个新的二进制可执行文件的映像。这种“派生加执行”的方式是很常见的。

**在fork多线程的进程时，创建的子进程只包含一个线程，该线程是调用fork函数的那个线程的副本。在多线程进程中，为了多线程的同步及互斥，会有锁，在fork时，这些锁会一同fork到子进程中，这会导致一些问题，最好不要fork多线程的进程，除非你有能力解决这个问题。**

在早期的Unix系统中，创建进程比较原始。当调用fork时，内核会把所有的内部数据结构复制一份，复制进程的页表项，然后把父进程的地址空间中的内容逐页的复制到子进程的地址空间中。但从内核角度来说，逐页的复制方式是十分耗时的。现代的Unix系统采取了更多的优化，例如Linux，**采用了写时复制的方法，而不是对父进程空间进程整体复制。**

* + 1. vfork的基础知识

在实现写时复制之前，Unix的设计者们就一直很关注在fork后立刻执行exec所造成的地址空间的浪费。BSD的开发者们在3.0的BSD系统中引入了vfork( )系统调用。

pid\_t vfork(void);

除了子进程必须要立刻执行一次对exec的系统调用，或者调用exit退出，对vfork的成功调用所产生的结果和fork是一样的。vfork会挂起父进程直到子进程终止或者运行了一个新的可执行文件的映像。通过这样的方式，vfork避免了地址空间的按页复制。在这个过程中，父进程和子进程共享相同的地址空间和页表项。实际上vfork只完成了一件事：复制内部的内核数据结构。因此，子进程也就不能修改地址空间中的任何内存。

vfork是一个历史遗留产物，Linux本不应该实现它。需要注意的是，即使增加了写时复制，vfork也要比fork快，因为它没有进行页表项的复制。然而，写时复制的出现减少了对于替换fork争论。实际上，直到2.2.0内核，vfork只是一个封装过的fork。因为对vfork的需求要小于fork，所以vfork的这种实现方式是可行的。

* + 1. fork和vfork的区别：

1. fork的子进程拷贝父进程的数据段和代码段；vfork的子进程与父进程共享数据段。

2. fork的父子进程的执行次序不确定；**vfork保证子进程先运行**，在调用exec或exit之前与父进程数据是共享的，在它调用exec或exit之后父进程才可能被调度运行。

3. **vfork保证子进程先运行，在它调用exec或exit之后父进程才可能被调度运行。如果在调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致死锁。**

4.当需要改变共享数据段中变量的值，则拷贝父进程。

（4）fork实例

#include<unistd.h>

#include<stdio.h>

int main(void){

pid\_t pid;

signal(SIGCHLD, SIG\_IGN);

printf("before fork pid:%d\n", getpid());

int count = 0;

pid = fork();

if (pid == -1) { //错误返回

printf("failed to folk");

return -1;

}

if (pid > 0) { //父进程空间

count++;

printf("parent:pid:%d \n", getpid());

printf("count:%d \n", count);

sleep(20);

}

else if (pid == 0) { //子进程空间

count++;

printf("child:%d,parent: %d\n", getpid(), getppid());

printf("count:%d", count);

}

printf("fork after...\n");

return 0;

}

* + 1. **写时复制**

Linux采用了写时复制的方法，以减少fork时对父进程空间进程整体复制带来的开销。

写时复制是一种采取了惰性优化方法来避免复制时的系统开销。它的前提很简单：如果有多个进程要读取它们自己的那部分资源的副本，那么复制是不必要的。每个进程只要保存一个指向这个资源的指针就可以了。只要没有进程要去修改自己的“副本”，就存在着这样的幻觉：每个进程好像独占那个资源。从而就避免了复制带来的负担。如果一个进程要修改自己的那份资源“副本”，那么就会复制那份资源，并把复制的那份提供给进程。不过其中的复制对进程来说是透明的。这个进程就可以修改复制后的资源了，同时其他的进程仍然共享那份没有修改过的资源。所以这就是名称的由来：在写入时进行复制。

**写时复制的主要好处在于：如果进程从来就不需要修改资源，则不需要进行复制。惰性算法的好处就在于它们尽量推迟代价高昂的操作，直到必要的时刻才会去执行。**

在使用虚拟内存的情况下，写时复制（Copy-On-Write）是以页为基础进行的。所以，只要进程不修改它全部的地址空间，那么就不必复制整个地址空间。在fork调用结束后，父进程和子进程都相信它们有一个自己的地址空间，但实际上它们共享父进程的原始页，接下来这些页又可以被其他的父进程或子进程共享。

写时复制在内核中的实现非常简单。与内核页相关的数据结构可以被标记为只读和写时复制。如果有进程试图修改一个页，就会产生一个缺页中断。内核处理缺页中断的方式就是对该页进行一次透明复制。这时会清除页面的COW属性，表示着它不再被共享。

现代的计算机系统结构中都在内存管理单元（MMU）提供了硬件级别的写时复制支持，所以实现是很容易的。

在调用fork( )时，写时复制是有很大优势的。因为大量的fork之后都会跟着执行exec，那么复制整个父进程地址空间中的内容到子进程的地址空间完全是在浪费时间：如果子进程立刻执行一个新的二进制可执行文件的映像，它先前的地址空间就会被交换出去。写时复制可以对这种情况进行优化。

# 有了进程，为什么还要有线程？

线程产生的原因：进程可以使多个程序能并发执行，以提高资源的利用率和系统的吞吐量；但是其具有一些缺点：

* 进程在同一时间只能干一件事。
* 进程在执行的过程中如果阻塞，整个进程就会挂起，即使进程中有些工作不依赖于等待的资源，仍然不会执行。

因此，操作系统引入了比进程粒度更小的线程，作为并发执行的基本单位，从而减少程序在并发执行时所付出的时空开销，提高并发性。和进程相比，线程的优势如下：

**从资源上来讲**，线程是一种非常"节俭"的多任务操作方式。在linux系统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种"昂贵"的多任务工作方式。

**从切换效率上来讲**，运行于一个进程中的多个线程，它们之间使用相同的地址空间，而且线程间彼此切换所需时间也远远小于进程间切换所需要的时间。据统计，一个进程的开销大约是一个线程开销的30倍左右。

**从通信机制上来讲**，线程间方便的通信机制。对不同进程来说，它们具有独立的数据空间，要进行数据的传递只能通过进程间通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然，由于同一进城下的线程之间共享数据空间，所以一个线程的数据可以直接为其他线程所用，这不仅快捷，而且方便。

除以上优点外，多线程程序作为一种多任务、并发的工作方式，还有如下优点：

1、使多CPU系统更加有效。操作系统会保证当线程数不大于CPU数目时，不同的线程运行于不同的CPU上。

2、改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序才会利于理解和修改。

# 单核机器上写多线程程序，是否需要考虑加锁，为什么？

在单核机器上写多线程程序，仍然需要线程锁。因为线程锁通常用来实现线程的同步和通信。在单核机器上的多线程程序，仍然存在线程同步的问题。因为在抢占式操作系统中，通常为每个线程分配一个时间片，当某个线程时间片耗尽时，操作系统会将其挂起，然后运行另一个线程。如果这两个线程共享某些数据，不使用线程锁的前提下，可能会导致共享数据修改引起冲突。

# 请问线程需要保存哪些上下文，SP、PC、EAX这些寄存器是干嘛用的

线程在切换的过程中需要保存当前**线程Id、线程状态、堆栈、寄存器状态等信息**。其中寄存器主要包括SP、PC、EAX等寄存器，其主要功能如下：

**SP:堆栈指针**，指向当前栈的栈顶地址。

**PC:程序计数器**，存储下一条将要执行的指令。

**EAX:累加寄存器**，用于加法乘法的缺省寄存器。

# 线程间的同步方式，最好说出具体的系统调用

（1）**信号量**

信号量是一种特殊的变量，可用于线程同步，也可用于进程同步。它只取自然数值，并且只支持两种操作：

P(SV)：如果信号量SV大于0，将它减一；如果SV值为0，则挂起该线程。

V(SV)：如果有其他进程因为等待SV而挂起，则唤醒，然后将SV+1；否则直接将SV+1。

**其系统调用为：**

**sem\_wait（sem\_t \*sem）**：以原子操作的方式将信号量减1，如果信号量值为0，则sem\_wait将被阻塞，直到这个信号量具有非0值。

**sem\_post（sem\_t \*sem)**：以原子操作将信号量值+1。当信号量大于0时，其他正在调用sem\_wait等待信号量的线程将被唤醒。

C++中实现信号量

#include <semaphore.h>

sem\_t sem;

// 初始化

// sem\_init() 成功返回0，失败返回1

// pshared = 0 线程间共享 pshared = 1进程间共享 value 信号量的初始值

int sem\_init(sem\_t \*sem,int pshared,unsigned int value);

// P操作，信号量sem减1，若sem=0，该线程被挂起到阻塞列表

sem\_wait(&sem)

// V操作，信号量sem加1，并唤醒阻塞列表中的线程

sem\_post(&sem)

（2）**互斥量**

互斥量又称互斥锁，主要用于线程互斥，不能保证按序访问，可以和条件锁一起实现同步。当进入临界区时，需要获得互斥锁并且加锁；当离开临界区时，需要对互斥锁解锁，以唤醒其他等待该互斥锁的线程。其主要的系统调用如下：

* pthread\_mutex\_init:初始化互斥锁。
* pthread\_mutex\_destroy：销毁互斥锁。
* pthread\_mutex\_lock：以原子操作的方式给一个互斥锁加锁，如果目标互斥锁已经被上锁，pthread\_mutex\_lock调用将阻塞，直到该互斥锁的占有者将其解锁。
* pthread\_mutex\_unlock:以一个原子操作的方式给一个互斥锁解锁。

（3）**条件变量**

条件变量，又称条件锁，用于在线程之间同步共享数据的值。条件变量提供一种线程间通信机制：**当某个共享数据达到某个值时，唤醒等待这个共享数据的一个/多个线程**。即，当某个共享变量等于某个值时，调用signal/broadcast。此时操作共享变量时需要加锁。其主要的系统调用如下：

* pthread\_cond\_init:初始化条件变量。
* pthread\_cond\_destroy：销毁条件变量。
* pthread\_cond\_signal：唤醒一个等待目标条件变量的线程。哪个线程被唤醒取决于调度策略和优先级。
* pthread\_cond\_wait：等待目标条件变量。需要一个加锁的互斥锁确保操作的原子性。该函数中在进入wait状态前首先进行解锁，然后接收到信号后会再加锁，保证该线程对共享资源正确访问。

# 死锁发生的条件以及如何解决死锁

**原因：**系统资源不足、资源分配不当、进程运行推进顺序不合适。

死锁是指两个或两个以上进程在执行过程中，因争夺资源而造成的相互等待的现象。死锁发生的四个必要条件如下：

* **互斥条件**：进程对所分配到的资源不允许其他进程访问，若其他进程访问该资源，只能等待，直至占有该资源的进程使用完成后释放该资源。
* **请求和保持条件**：进程获得一定的资源后，又对其他资源发出请求，但是该资源可能被其他进程占有，此时请求阻塞，但该进程不会释放自己已经占有的资源。
* **不可剥夺条件**：进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用后自己释放。
* **环路等待条件**：进程发生死锁后，必然存在一个进程-资源之间的环形链。

解决死锁的方法即破坏上述四个条件之一，主要方法如下：

* **资源一次性分配**，从而剥夺请求和保持条件。
* **可剥夺资源**：即当进程新的资源未得到满足时，释放已占有的资源，从而破坏不可剥夺的条件。
* **资源有序分配法**：系统给每类资源赋予一个序号，每个进程按编号递增的请求资源，释放则相反，从而破坏环路等待的条件。

# 死锁避免

死锁避免并不是事先采取某种限制措施破坏死锁的必要条件，而是在资源动态分配的过程中，**防止系统进入不安全状态**，以避免发生死锁。这种方法所施加的限制条件较弱，可以获得较好的系统性能。

* 1. 系统安全状态

允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次资源分配的安全性。若此次分配不会导致系统进入不安全状态，则将资源分配进程；否则让进程等待。

所谓安全状态，是指系统能按某种进程推进顺序为每个进程分配其所需的资源，直到满足每个进程对资源的最大需求，使每个进程都可顺序完成。此时称为安全序列。若系统无法找到一个安全序列，则称系统处于不安全状态。并非所有的不安全状态都是死锁状态，但当系统进入不安全状态后，便可能进入死锁状态；反之，只要系统处于安全状态，系统便可避免进入死锁状态。

* 1. 银行家算法

# 操作系统中的结构体对齐，字节对齐

（1）原因：

1）平台原因（移植原因）：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。

2）性能原因：数据结构（尤其是栈）应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。

3) 例子：比如在32位cpu下，假设一个整型变量的地址为0x00000004，那它就是自然对齐的。假设上面整型变量的地址不是自然对齐，比如为0x00000002，则CPU如果取它的值的话需要访问两次内存，第一次取从0x00000002-0x00000003的一个short，第二次取从0x00000004-0x00000005的一个short然后组合得到所要的数据，如果变量在0x00000003地址上的话则要访问三次内存，第一次为char，第二次为short，第三次为char，然后组合得到整型数据。而如果变量在自然对齐位置上，则只要一次就可以取出数据。

（2）规则

1）数据成员对齐规则：结构(struct)(或联合(union))的数据成员，第一个数据成员放在offset为0的地方，以后每个数据成员的对齐按照#pragma pack指定的数值和这个数据成员自身长度中，比较小的那个进行。

2）结构(或联合)的整体对齐规则：在数据成员完成各自对齐之后，结构(或联合)本身也要进行对齐，对齐将按照#pragma pack指定的数值和结构(或联合)最大数据成员长度中，比较小的那个进行。

3）**结构体作为成员：如果一个结构里有某些结构体成员，则结构体成员要从其内部最大元素大小的整数倍地址开始存储。**

（3）定义结构体对齐

可以通过预编译命令#pragma pack(n)，n=1,2,4,8,16来改变这一系数，其中的n就是指定的“对齐系数”。举例：

#pragma pack(2)

struct AA {

int a; //长度4 > 2 按2对齐；偏移量为0；存放位置区间[0,3]

char b; //长度1 < 2 按1对齐；偏移量为4；存放位置区间[4]

short c; //长度2 =2按2对齐；偏移量要提升到2的倍数6；存放位置区间[6,7]

char d; //长度1 < 2 按1对齐；偏移量为7；存放位置区间[8]；共九个字节

};//结构体自身对齐，共10个字节

#pragma pack()

# 虚拟内存置换的方式

比较常见的内存替换算法有：FIFO，LRU，LFU，LRU-K，2Q。

**（1）FIFO（先进先出淘汰算法）**

思想：最近刚访问的，将来访问的可能性比较大。

实现：使用一个队列，新加入的页面放入队尾，每次淘汰队首的页面，即最先进入的数据，最先被淘汰。

弊端：无法体现页面冷热信息。

**（2）LFU（最不经常访问淘汰算法）**

思想：如果数据过去被访问多次，那么将来被访问的频率也更高。

实现：每个数据块一个引用计数，所有数据块按照引用计数排序，具有相同引用计数的数据块则按照时间排序。每次淘汰队尾数据块。

开销：排序开销。

弊端：缓存颠簸。

**（3）LRU（最近最少使用替换算法）**

思想：如果数据最近被访问过，那么将来被访问的几率也更高。

实现：使用一个栈，新页面或者命中的页面则将该页面移动到栈底，每次替换栈顶的缓存页面。

优点：LRU算法对热点数据命中率是很高的。

缺陷：

1）缓存颠簸，当缓存（1，2，3）满了，之后数据访问（0，3，2，1，0，3，2，1…）。

2）缓存污染，突然大量偶发性的数据访问，会让内存中存放大量冷数据。

（4）LRU-K（LRU-2、LRU-3）

思想：最久未使用K次淘汰算法。

# 互斥锁（mutex）机制，以及互斥锁和读写锁的区别

（1）互斥锁和读写锁区别：

**互斥锁**：mutex，用于保证在任何时刻，都只能有一个线程访问该对象。当获取锁操作失败时，线程会进入睡眠，等待锁释放时被唤醒。

**读写锁**：rwlock，分为读锁和写锁。处于读操作时，可以允许多个线程同时获得读操作。但是同一时刻只能有一个线程可以获得写锁。其它获取写锁失败的线程都会进入睡眠状态，直到写锁释放时被唤醒。 注意：**写锁会阻塞其它读写锁**。当有一个线程获得写锁在写时，读锁也不能被其它线程获取；写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）。适用于读取数据的频率远远大于写数据的频率的场合。

**互斥锁和读写锁的区别**：

1）读写锁区分读者和写者，而互斥锁不区分。

2）互斥锁同一时间只允许一个线程访问该对象，无论读写；读写锁同一时间内只允许一个写者，但是允许多个读者同时读对象。

（2）Linux的4种锁机制：

**互斥锁**：mutex，用于保证在任何时刻，都只能有一个线程访问该对象。当获取锁操作失败时，线程会进入睡眠，等待锁释放时被唤醒。

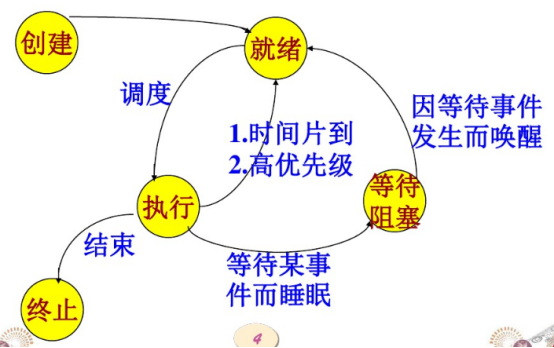
**读写锁**：rwlock，分为读锁和写锁。处于读操作时，可以允许多个线程同时获得读操作。但是同一时刻只能有一个线程可以获得写锁。其它获取写锁失败的线程都会进入睡眠状态，直到写锁释放时被唤醒。写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）。适用于读取数据的频率远远大于写数据的频率的场合。

**自旋锁**：spinlock，在任何时刻同样只能有一个线程访问对象。**但是当获取锁操作失败时，不会进入睡眠，而是会在原地自旋，直到锁被释放**。这样节省了线程从睡眠状态到被唤醒期间的消耗，在加锁时间短暂的环境下会极大的提高效率。但如果加锁时间过长，则会非常浪费CPU资源。

**RCU：即read-copy-update**，在修改数据时，首先需要读取数据，然后生成一个副本，对副本进行修改。修改完成后，再将老数据update成新的数据。使用RCU时，读者几乎不需要同步开销，既不需要获得锁，也不使用原子指令，不会导致锁竞争，因此就不用考虑死锁问题了。而对于写者的同步开销较大，它需要复制被修改的数据，还必须使用锁机制同步并行其它写者的修改操作。在有大量读操作，少量写操作的情况下效率非常高。

# 进程状态转换图，动态就绪，静态就绪，动态阻塞，静态阻塞

* 1. 进程的五种基本状态：



1）创建状态：进程正在被创建。

2）就绪状态：进程被加入到就绪队列中等待CPU调度运行。

3）执行状态：进程正在被运行。

4）等待阻塞状态：进程因为某种原因，比如等待I/O，等待设备，而暂时不能运行。

5）终止状态：进程运行完毕。

* 1. 交换技术

当多个进程竞争内存资源时，会造成内存资源紧张，并且，如果此时没有就绪进程，处理机会空闲，I/0速度比处理机速度慢得多，可能出现全部进程阻塞等待I/O。

针对以上问题，提出了两种解决方法：

1）交换技术：换出一部分进程到外存，腾出内存空间。

2）虚拟存储技术：每个进程只能装入一部分程序和数据。

在交换技术上，将内存暂时不能运行的进程，或者暂时不用的数据和程序，换出到外存，来腾出足够的内存空间，把已经具备运行条件的进程，或进程所需的数据和程序换入到内存。从而出现了进程的挂起状态：进程被交换到外存，进程状态就成为了挂起状态。

* 1. 活动阻塞，静止阻塞，活动就绪，静止就绪

1）活动阻塞：进程在内存，但是由于某种原因被阻塞了。

2）静止阻塞：进程在外存，同时被某种原因阻塞了。

3）活动就绪：进程在内存，处于就绪状态，只要给CPU和调度就可以直接运行。

4）静止就绪：进程在外存，处于就绪状态，只要调度到内存，给CPU和调度就可以运行。

从而出现了：

活动就绪——静止就绪 （内存不够，调到外存）

活动阻塞——静止阻塞 （内存不够，调到外存）

执行 ——静止就绪 （时间片用完）

# A\* a = new A; a->i = 10;在内核中的内存分配上发生了什么？

1）A \*a：a是一个局部变量，类型为指针，故而操作系统在程序**栈区**开辟4/8字节的空间（0x000m），分配给指针a。

2）new A：通过new动态的在**堆区**申请类A大小的空间（0x000n）。

3）a = new A：将指针a的内存区域填入栈中类A申请到的空间地址。即\*（0x000m）=0x000n。

4）a->i：先找到指针a的地址0x000m，通过a的值0x000n和i在类a中偏移offset，得到a->i的地址0x000n + offset，进行\*(0x000n + offset) = 10的赋值操作，即内存0x000n + offset的值是10。

# 给你一个类，里面有static，virtual，之类的，来说一说这个类的内存分布

1. static修饰符

1）static修饰成员变量

对于非静态数据成员，每个类对象都有自己的拷贝。而静态数据成员被当做是类的成员，无论这个类被定义了多少个，静态数据成员都只有一份拷贝，为该类型的所有对象所共享(包括其派生类)。所以，静态数据成员的值对每个对象都是一样的，它的值可以更新。

因为静态数据成员在**全局数据区分配内存**，属于本类的所有对象共享，所以它不属于特定的类对象，在没有产生类对象前就可以使用。

2）static修饰成员函数

与普通的成员函数相比，静态成员函数由于不是与任何的对象相联系，因此它不具有this指针。从这个意义上来说，**它无法访问属于类对象的非静态数据成员，也无法访问非静态成员函数，**只能调用其他的静态成员函数，**仍然在代码区分配内存**。

**C++中的单例模式**：1.将构造函数放在私有区，使该对象无法实例化。2.使用静态成员函数创建一个该对象的静态实例化，并返回。3.所有操作者均可以使用对象名来访问这个静态成员函数从而访问这个静态对象的所有成员。（注意：静态成员函数的声明要加static，但是定义的时候不需要加static）

1. C++继承和虚函数

C++多态分为静态多态和动态多态。**静态多态是通过重载和模板技术实现，在编译的时候确定。动态多态通过虚函数和继承关系来实现，执行动态绑定，在运行的时候确定。**动态多态实现有几个条件：

1) 虚函数；

2) 一个基类的指针或引用指向派生类的对象；

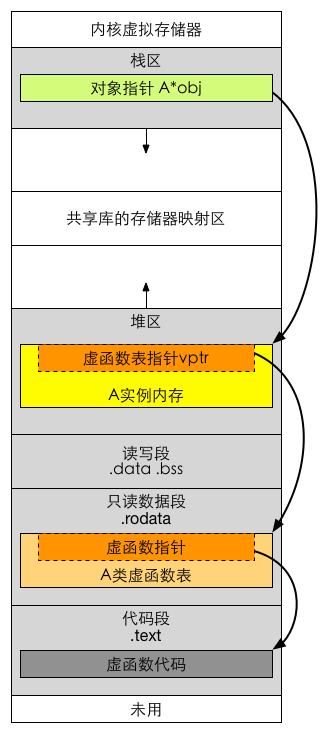
基类指针在调用成员函数(虚函数)时，就会去查找该对象的虚函数表。虚函数表的地址在每个对象的首地址。查找该虚函数表中该函数的指针进行调用。**每个对象中保存的只是一个虚函数表的指针，C++内部为每一个类维持一个虚函数表，该类的对象的都指向这同一个虚函数表**。虚表指针在堆区分配，虚函数表在全局数据区。

虚函数表中为什么就能准确查找相应的函数指针呢？因为在类设计的时候，虚函数表直接从基类也继承过来，如果覆盖了其中的某个虚函数，那么虚函数的指针就会被替换，因此可以根据指针准确找到该调用哪个函数。

1. virtual修饰符

**如果一个类是局部变量则该类数据存储在栈区，如果一个类是通过new/malloc动态申请的，则该类数据存储在堆区**。

**如果该类是virutal继承而来的子类，则该类的虚函数表指针和该类其他成员一起存储。**虚函数表指针指向**只读数据段中的类虚函数表**，虚函数表中存放着一个个函数指针，函数指针指向代码段中的具体函数。如果类中成员是virtual属性，会隐藏父类对应的属性。



# 软链接和硬链接

1. 硬链接总结：
2. 具有相同**inode**（索引节点）号的多个文件互为硬链接文件。删除硬链接文件或者删除源文件任意之一，文件实体并未被删除。只有删除了源文件和所有对应的硬链接文件，文件实体才会被删除。
3. 硬链接文件是文件的另一个入口。
4. 可以通过给文件设置硬链接文件来防止重要文件被误删。
5. 创建硬链接命令 **ln 源文件 硬链接文件**。
6. 硬链接文件是普通文件，可以用rm删除。
7. 对于静态文件（没有进程正在调用），当硬链接数0时文件就被删除。注意：如果有进程正在调用，则无法删除或者即使文件名被删除但空间不会释放。
8. 软连接总结：
9. 软链接类似windows 系统的快捷方式。
10. 软链接里面存放的是源文件的路径，指向源文件。
11. 删除源文件，软链接依然存在，但无法访问源文件内容。
12. 软链接失效时一般是白字红底闪烁。
13. 创建软链接命令 **ln -s 源文件 软链接文件**。
14. **软链接和源文件是不同的文件，文件类型也不同，inode号也不同。**
15. 软链接的文件类型是“l”，可以用rm删除。
16. 硬链接和软链接的区别

原理上，硬链接和源文件的inode节点号相同，两者互为硬链接。软连接和源文件的inode节点号不同，进而指向的block也不同，软连接block中存放了源文件的路径名。

实际上，硬链接和源文件是同一份文件，而软连接是独立的文件，类似于快捷方式，存储着源文件的位置信息便于指向。

使用限制上，**不能对目录创建硬链接**，**不能对不同文件系统创建硬链接**，**不能对不存在的文件创建硬链接**；**可以对目录创建软连接，可以跨文件系统创建软连接，可以对不存在的文件创建软连接。**

# 什么是大端小端以及如何判断大端小端

大端是指低字节存储在高地址；小端存储是指低字节存储在低地址。

1. 使用强制类型转换

int a = 0x1234;

//由于int和char的长度不同，借助int型转换成char型，只会留下低地址的部分

char c = (char)(a);

if (c == 0x12)

cout << "big endian" << endl;

else if(c == 0x34)

cout << "little endian" << endl;

1. **可以根据联合体来判断该系统是大端还是小端**。**因为联合体变量总是从低地址存储**。

//判断系统是大端还是小端：通过联合体，因为联合体的所有成员都从低地址开始存放

int fun() {

union test {

int i;

char c;

};

test t;

t.i = 1;

//如果是大端,则t.c为0x00,则t.c!=1,返回0;若是小端,则t.c为0x01,则t.c==1,返回1

return t.c == 1;

}

# core dump（核心转储）

**当程序运行过程中异常终止或崩溃，操作系统会将程序当时的内存状态记录下来，保存在一个文件中，这种行为就叫做core dump（核心转储）。**但实际上，除了内存信息之外，还有些关键的程序运行状态也会同时dump下来，例如寄存器信息（包括程序指针、栈指针等）、内存管理信息、其他处理器和操作系统状态和信息。**core dump文件可以再现程序出错时的情景。**

# 提高磁盘IO性能的方法：

* + - **文件越多读取越慢：**如果可以的话，将多个小文件合并成一个文件。
    - **读写次数越多读取越慢：**一次多读一些数据到内存。
    - **将读写操作分配到不同的硬盘上。**
    - **磁盘RAID0比RAID5读写速度快很多。**

# 静态变量什么时候初始化

静态变量存储在虚拟地址空间的数据段和bss段，**C语言中其在代码执行之前初始化，属于编译期初始化**。而C++中由于引入对象，对象生成必须调用构造函数，因此**C++规定全局或局部静态对象当且仅当对象首次用到时进行构造**。

# 什么是原子操作

原子操作（atomic operation）指的是由多步操作组成的一个操作。如果该操作不能原子地执行，则要么执行完所有步骤，要么一步也不执行，不可能只执行所有步骤的一个子集。

现代操作系统中，一般都提供了原子操作来实现一些同步操作，所谓原子操作，也就是一个独立而不可分割的操作。在单核环境中，一般的意义下原子操作中线程不会被切换，线程切换要么在原子操作之前，要么在原子操作完成之后。更广泛的意义下原子操作是指一系列必须整体完成的操作步骤，如果任何一步操作没有完成，那么所有完成的步骤都必须回滚，这样就可以保证要么所有操作步骤都未完成，要么所有操作步骤都被完成。

例如在单核系统里，单个的机器指令可以看成是原子操作（如果有编译器优化、乱序执行等情况除外）；在多核系统中，单个的机器指令就不是原子操作，因为多核系统里是多指令流并行运行的，一个核在执行一个指令时，其他核同时执行的指令有可能操作同一块内存区域，从而出现数据竞争现象。多核系统中的原子操作通常使用内存栅障（memory barrier）来实现，即一个CPU核在执行原子操作时，其他CPU核必须停止对内存操作或者不对指定的内存进行操作，这样才能避免数据竞争问题。

# 如何设计server，使得能够接收多个客户端的请求：多线程，线程池，io复用。

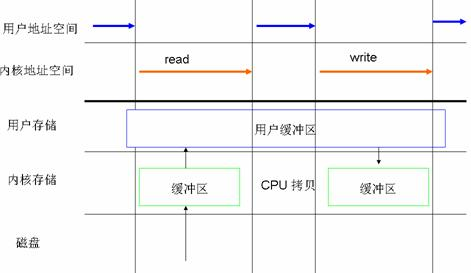
# 缓存IO、直接IO、内存映射

1. 缓存IO (Buffered I/O)

缓存I/O又称为标准I/O，大多数文件系统的默认I/O操作都是缓存I/O。在Linux的缓存I/O机制中，操作系统会将I/O的数据缓存在文件系统的页缓存中，即数据会**先被拷贝到操作系统内核的缓冲区中，然后才会从操作系统内核的缓冲区拷贝到应用程序的地址空间**。

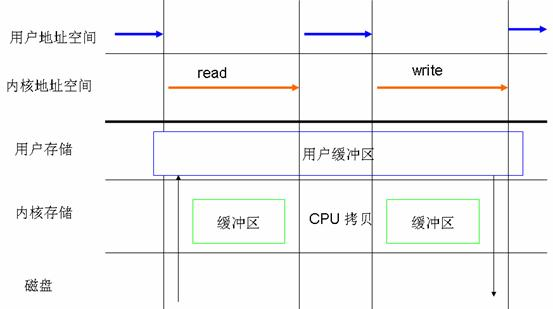
**读操作**：操作系统检查内核的缓冲区有没有需要的数据，如果已经缓存了，那么就直接从缓存中返回；否则从磁盘中读取，然后缓存在操作系统的缓存中。

**写操作**：将数据从用户空间复制到内核空间的缓存中。这时对用户程序来说写操作就已经完成，至于什么时候再写到磁盘中由操作系统决定，除非显示地调用了sync同步命令。



1. **优点：**在一定程度上分离了内核空间和用户空间，保护系统本身的运行安全。可以减少读盘的次数，从而提高性能。
2. **缺点**数据在传输过程中需要在应用程序地址空间和缓存之间进行多次数据拷贝操作，这些数据拷贝操作所带来的CPU以及内存开销是非常大的。
3. 直接IO

省略掉**缓存I/O技术**中操作系统**内核缓冲区**的使用，数据**直接在应用程序地址空间和磁盘之间进行传输**，这样做的目的是减少一次从内核缓冲区到用户程序缓存的数据复制。比如说数据库管理系统这类应用，它们更倾向于选择它们自己的缓存机制，因为数据库管理系统往往比操作系统更了解数据库中存放的数据，数据库管理系统可以提供一种更加有效的缓存机制来提高数据库中数据的存取性能。



1. 优点：

减少操作系统内核缓冲区和应用程序地址空间的数据拷贝的次数，降低了对文件读取和写入时所带来的CPU的使用以及内存带宽的占用。如果传输的数据量很大，使用直接I/O的方式进行数据传输，而不需要操作系统内核地址空间拷贝数据操作的参与，提高性能。

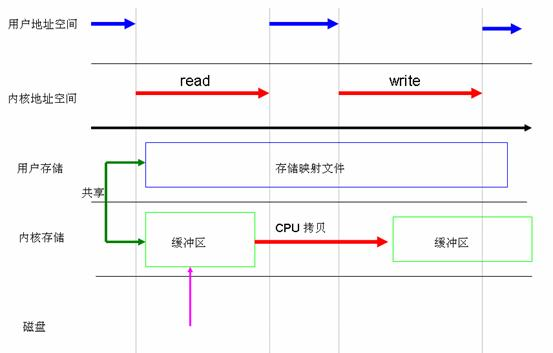
1. 缺点：

如果访问的数据不在应用程序缓存中，那么每次数据都会直接从磁盘加载，这种直接加载会非常缓慢。通常直接IO与异步IO结合使用，会得到比较好的性能。（异步IO：当访问数据的线程发出请求之后，线程会接着去处理其他事，而不是阻塞等待）。

1. 内存映射

内存映射是指将硬盘上文件的位置与进程**逻辑地址空间**中一块大小相同的区域一一对应，**当要访问内存中一段数据时，转换为访问文件的某一段数据**。这种方式的目的同样是**减少数据在用户空间和内核空间之间的拷贝操作**。当大量数据需要传输的时候，采用内存映射方式去访问文件会获得比较好的效率。

使用内存映射文件处理存储于磁盘上的文件时，将不必再对文件执行I/O操作，这意味着在对文件进行处理时将不必再为文件申请并分配缓存，所有的文件缓存操作均由系统直接管理，由于取消了将文件数据加载到内存、数据从内存到文件的回写以及释放内存块等步骤，使得内存映射文件在处理大数据量的文件时能起到相当重要的作用。



# 内存溢出和内存泄漏

（1）内存溢出

指程序申请内存时，**没有足够的内存**供申请者使用。内存溢出就是你要的内存空间超过了系统实际分配给你的空间，此时系统相当于没法满足你的需求，就会报内存溢出的错误

**内存溢出原因：**

1）内存中加载的数据量过于庞大，如一次从数据库取出过多数据。

2）集合类中有对对象的引用，使用完后未清空，使得不能回收。

3）代码中存在死循环或循环产生过多重复的对象实体。

4）使用的第三方软件中的BUG。

5）启动参数内存值设定的过小。

（2）内存泄漏

内存泄漏是指由于疏忽或错误造成了**程序未能释放掉不再使用的内存**的情况。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。

内存泄漏的分类：

1、**堆内存泄漏 （Heap leak）**。对内存指的是程序运行中根据需要分配通过malloc，realloc，new等从堆中分配的一块内存，再是完成后必须通过调用对应的free或者delete删掉。如果程序的设计的错误导致这部分内存没有被释放，那么此后这块内存将不会被使用，就会产生Heap Leak。

**2、系统资源泄露（Resource Leak）**。主要指程序使用系统分配的资源比如 Bitmap，handle ，SOCKET等**没有使用相应的函数释放掉**，导致系统资源的浪费，严重可导致系统效能降低，系统运行不稳定。

3、**没有将基类的析构函数定义为虚函数**。当基类指针指向子类对象时，如果基类的析构函数不是virtual，那么子类的析构函数将不会被调用，子类的资源没有正确是释放，因此造成内存泄露。

# 常用线程模型

（1）fork&join模型

该模型包含递归思想和回溯思想，递归用来拆分任务，回溯用合并结果。可以用来处理一些可以进行拆分的大任务。其主要是把一个大任务逐级拆分为多个子任务，然后分别在子线程中执行，当每个子线程执行结束之后逐级回溯，返回结果进行汇总合并，最终得出想要的结果。

这里模拟一个摘苹果的场景：有100棵苹果树，每棵苹果树有10个苹果，现在要把他们摘下来。为了节约时间，规定每个线程最多只能摘10棵苹果树以便于节约时间。各个线程摘完之后汇总计算总苹果树。

（2）actor模型

actor模型属于一种基于消息传递机制并行任务处理思想，它以消息的形式来进行线程间数据传输，避免了全局变量的使用，进而避免了数据同步错误的隐患。actor在接受到消息之后可以自己进行处理，也可以继续传递（分发）给其它actor进行处理。在使用actor模型的时候需要使用第三方Akka提供的框架。

（3）生产者消费者模型

生产者消费者模型都比较熟悉，其核心是使用一个缓存来保存任务。开启一个/多个线程来生产任务，然后再开启一个/多个来从缓存中取出任务进行处理。这样的好处是任务的生成和处理分隔开，生产者不需要处理任务，只负责向生成任务然后保存到缓存。而消费者只需要从缓存中取出任务进行处理。使用的时候可以根据任务的生成情况和处理情况开启不同的线程来处理。比如，生成的任务速度较快，那么就可以灵活的多开启几个消费者线程进行处理，这样就可以避免任务的处理响应缓慢的问题。

（4）master-worker模型

master-worker模型类似于任务分发策略，开启一个master线程接收任务，然后在master中根据任务的具体情况进行分发给其它worker子线程，然后由子线程处理任务。如需返回结果，则worker处理结束之后把处理结果返回给master。

（5）Future模型

Future是把结果放在将来获取，当前主线程并不急于获取处理结果。允许子线程先进行处理一段时间，处理结束之后就把结果保存下来，当主线程需要使用的时候再向子线程索取。

# 协程

协程，又称微线程。协程看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。

1. 协程和线程的区别

和多线程比，协程最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。第二大优势就是**不需要多线程的锁机制**，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

1. 其他

在协程上利用多核CPU呢——多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。Python对协程的支持还非常有限，用在generator中的yield可以一定程度上实现协程。虽然支持不完全，但已经可以发挥相当大的威力了。

# 作业调度算法

1. 先来先服务算法（FCFS调度算法）

先进入输入井的作业先被服务，值得注意的是，并不是先来的就一定会被先放入主存区，必须要有合适的而资源才行。

缺点：当作业周转时间长的作业占用主存资源时，主存区资源长时间不能被释放，后面周转时间少的作业只能处于等待状态，这无疑减少了系统的吞吐量。

1. 短作业优先算法（SJF调度算法）

短作业调度算法的核心在于，运行时间越短的作业就先被执行。有两个值得注意的地方：

1)首先第一到达的作业一定先运行。

2)要注意，在后续比较作业时间长短时，要看作业是否到达，就是到达时间，没有到达的作业是不能比较的。

1. 高响应比优先调度算法

优先权=1+（等待时间/要求服务时间），综合了作业的等待时间和作业的运行时间。

1. 优先级调度算法

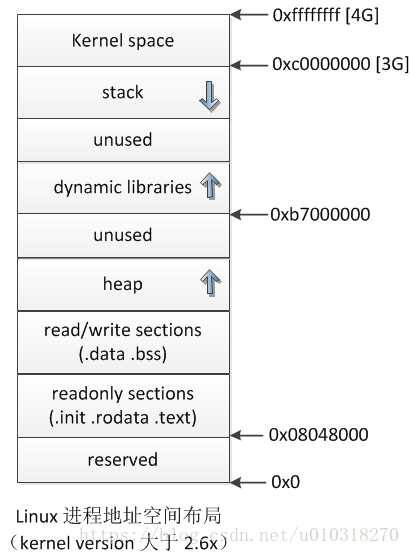
这种算法使用优先级来决定谁先被服务。设定优先级会有两种方法，第一是用户自己设定，第二种是系统设定。

# linux 进程地址空间分布

在32位操作系统中，内存空间拥有4GB的寻址能力。操作系统会把高地址的空间分配给内核，称为内核空间。

1. 内核空间：默认情况下，Windows将高地址的2GB空间分配给内核，Linux将高地址的1GB空间分配给内核。剩下的2GB或3GB的内存空间称为用户空间。在用户空间里，有许多地址区间有特殊的地位，一般来讲，应用程序使用的内存空间里有如下"默认"的区域。
2. 栈：用于维护函数调用的上下文。栈通常在用户空间的最高地址处分配，通常有数兆字节的大小。
3. **动态链接库映射区**：用于映射装载的动态链接库，在linux中，如果可执行文件依赖于其他共享库，系统将在0xbfxxxxxx附近分配地址(kernel > 2.6)，并将共享库载入到该空间。[kernel = 2.4x，从0x40000000开始分配]
4. 堆：用来容纳应用程序动态分配的内存区域，malloc或new分配内存时，得到的内存来自堆里。堆通常存在于栈的下方(低地址方向)，可以有几十或数百兆字节。
5. 可执行文件映像：存储着可执行文件在内存里的映像，由装载器在装载时将可执行文件的内存读取或映射到这里。包括read/write sections和readonly sections。
6. 保留区：对内存中受到保护而禁止访问的内存区域的总称。

栈向低地址增长，堆向高地址增长。当栈或堆现有的大小不够用时，它将按照图中的增长方向扩大自身的尺寸，直到预留的空间被用完为止。



# 说一下源码到可执行文件的过程

（1）预编译

主要处理源代码文件中的以“#”开头的预编译指令。处理规则见下

1）删除所有的#define，展开所有的宏定义。

2）处理所有的条件预编译指令，如“#if”、“#endif”、“#ifdef”、“#elif”和“#else”。

3）处理“#include”预编译指令，将文件内容替换到它的位置，这个过程是递归进行的，文件中包含其他文件。

4）删除所有的注释，“//”和“/\*\*/”。

5）保留所有的#pragma 编译器指令，编译器需要用到他们，如：#pragma once 是为了防止有文件被重复引用。

6）添加行号和文件标识，便于编译时编译器产生调试用的行号信息，和编译时产生编译错误或警告是能够显示行号。

（2）编译

把预编译之后生成的xxx.i或xxx.ii文件，进行一系列词法分析、语法分析、语义分析及优化后，生成相应的汇编代码文件。

1）词法分析：利用类似于“有限状态机”的算法，将源代码程序输入到扫描机中，将其中的字符序列分割成一系列的记号。

2）语法分析：语法分析器对由扫描器产生的记号，进行语法分析，产生语法树。由语法分析器输出的语法树是一种以表达式为节点的树。

3）语义分析：语法分析器只是完成了对表达式语法层面的分析，语义分析器则对表达式是否有意义进行判断，其分析的语义是静态语义——在编译期能分期的语义，相对应的动态语义是在运行期才能确定的语义。

4）优化：源代码级别的一个优化过程。

5）目标代码生成：由代码生成器将中间代码转换成目标机器代码，生成一系列的代码序列——汇编语言表示。

6）目标代码优化：目标代码优化器对上述的目标机器代码进行优化：寻找合适的寻址方式、使用位移来替代乘法运算、删除多余的指令等。

（3）汇编

**将汇编代码转变成机器可以执行的指令(机器码文件)。** 汇编器的汇编过程相对于编译器来说更简单，没有复杂的语法，也没有语义，更不需要做指令优化，只是**根据汇编指令和机器指令的对照表一一翻译过来**，汇编过程有汇编器as完成。经汇编之后，产生目标文件(与可执行文件格式几乎一样)xxx.o(Windows下)、xxx.obj(Linux下)。

（4）链接

将不同的源文件产生的目标文件进行链接，从而形成一个可以执行的程序。链接分为静态链接和动态链接：

1）静态链接：

**函数和数据被编译进一个二进制文件**。在使用静态库的情况下，在编译链接可执行文件时，链接器从库中复制这些函数和数据并把它们和应用程序的其它模块组合起来创建最终的可执行文件。

**空间浪费**：因为每个可执行程序中对所有需要的目标文件都要有一份副本，所以如果多个程序对同一个目标文件都有依赖，会出现同一个目标文件都在内存存在多个副本；

**更新困难**：每当库函数的代码修改了，这个时候就需要重新进行编译链接形成可执行程序。

**运行速度快**：但是静态链接的优点就是，在可执行程序中已经具备了所有执行程序所需要的任何东西，在执行的时候运行速度快。

2）动态链接：

**动态链接的基本思想是把程序按照模块拆分成各个相对独立部分，在程序运行时才将它们链接在一起形成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。**

**共享库**：就是即使需要每个程序都依赖同一个库，但是该库不会像静态链接那样在内存中存在多分，副本，而是这多个程序在执行时共享同一份副本；

**更新方便**：更新时只需要替换原来的目标文件，而无需将所有的程序再重新链接一遍。当程序下一次运行时，新版本的目标文件会被自动加载到内存并且链接起来，程序就完成了升级的目标。

**性能损耗**：因为把链接推迟到了程序运行时，所以每次执行程序都需要进行链接，所以性能会有一定损失。

# 微内核与宏内核

**宏内核**：除了最基本的进程、线程管理、内存管理外，将文件系统，驱动，网络协议等等都集成在内核里面，例如linux内核。

优点：效率高。

缺点：稳定性差，开发过程中的bug经常会导致整个系统挂掉。

**微内核**：内核中只有最基本的调度、内存管理。驱动、文件系统等都是用户态的守护进程去实现的。

优点：稳定，驱动等的错误只会导致相应进程死掉，不会导致整个系统都崩溃

缺点：效率低。

# 孤儿进程、僵尸进程和守护进程

（1）正常进程

正常情况下，子进程是通过父进程创建的，子进程再创建新的进程。子进程的结束和父进程的运行是一个异步过程，即父进程永远无法预测子进程到底什么时候结束。当一个进程完成它的工作终止之后，它的父进程需要调用wait()或者waitpid()系统调用取得子进程的终止状态。

unix提供了一种机制可以保证只要父进程想知道子进程结束时的状态信息， 就可以得到**：在每个进程退出的时候，内核释放该进程所有的资源，包括打开的文件，占用的内存等。 但是仍然为其保留一定的信息，直到父进程通过wait/waitpid来取时才释放。**保存信息包括：

1）进程号the process ID。

2）退出状态the termination status of the process。

3）运行时间the amount of CPU time taken by the process等。

（2）孤儿进程

一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。

（3）僵尸进程

一个进程使用fork创建子进程，如果子进程退出，而父进程并**没有调用wait或waitpid**获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵尸进程。**僵尸进程是一个进程必然会经过的过程：这是每个子进程在结束时都要经过的阶段**。如果子进程在exit()之后，父进程没有来得及处理，这时用ps命令就能看到子进程的状态是“Z”。如果父进程能及时处理，可能用ps命令就来不及看到子进程的僵尸状态，但这并不等于子进程不经过僵尸状态。**如果父进程在子进程结束之前退出，则子进程将由init接管。init将会以父进程的身份对僵尸状态的子进程进行处理。**

**相关接口：**

回收进程1：pid\_wait(int \*status)

一旦调用wait，就会立刻阻塞自己，wait自动分析某个子进程是否退出，如果找到僵尸进程就会负责收集和销毁，如果没有找到就一直阻塞在这里。

status：指向子进程结束状态值。

回收进程2： pid\_t waitpid(pid\_t pid,int \*status,int options)

返回值：返回pid：返回收集的子进程id。返回-1：出错，返回0：没有被收集的子进程。

**危害：**如果进程不调用wait/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，如果大量的产生僵死进程，将因为没有可用的进程号而导致系统不能产生新的进程。**

**外部消灭**：通过kill发送SIGTERM或者SIGKILL信号消灭产生僵尸进程的进程，它产生的僵死进程就变成了孤儿进程，这些孤儿进程会被init进程接管，init进程会wait这些孤儿进程，释放它们占用的系统进程表中的资源

**内部解决：**

1）子进程退出时向父进程发送SIGCHILD信号，父进程处理SIGCHILD信号。在信号处理函数中调用wait进行处理僵尸进程。

2）fork两次，而且使紧跟的子进程直接退出，使得孙子进程成为孤儿进程，其的父进程变为init进程，从而init进程将负责清除这个孤儿进程。

int main(){

pid\_t pid, pid1;

int status;

if ((pid = fork()) < 0){

printf("fork error\n");

return -1;

}else if (pid == 0){

printf("%d,%d\n", getpid(), getppid());

if ((pid1 = fork()) < 0){

printf("fork error\n");

return -1;

}else if (pid1 == 0){

printf("%d,%d\n", getpid(), getppid());

system("ps -ef|grep dlsc30");

return 0;

}

exit(0);

}

wait(&status);

printf("%d,%d\n", getpid(), getppid());

while (1){

printf("i will not break\n");

sleep(5);

}

exit(0);

}

**（4）守护进程**

守护进程是脱离终端并在后台运行的进程，执行过程中信息不会显示在终端上并且也不会被终端发出的信号打断。

**操作步骤：**创建子进程，父进程退出：fork() + if(pid > 0){exit(0);}，使子进程称为孤儿进程被init进程收养。

# 介绍一下5种IO模型

1）**阻塞IO：**调用者调用了某个函数，等待这个函数返回，期间什么也不做，不停的去检查这个函数有没有返回，必须等这个函数返回才能进行下一步动作。

2）**非阻塞IO：**非阻塞等待，每隔一段时间就去检测IO事件是否就绪。没有就绪就可以做其他事。

3）**信号驱动IO**：linux用套接口进行信号驱动IO，安装一个信号处理函数，进程继续运行并不阻塞，当IO时间就绪，进程收到SIGIO信号。然后处理IO事件。

4）**IO复用/多路转接IO**：linux用select/poll函数实现IO复用模型，这两个函数也会使进程阻塞，但是和阻塞IO所不同的是这两个函数可以同时阻塞多个IO操作。而且可以同时对多个读操作、写操作的IO函数进行检测。知道有数据可读或可写时，才真正调用IO操作函数。

5）**异步IO**：linux中可以调用aio\_read函数告诉内核描述字缓冲区指针和缓冲区的大小、文件偏移及通知的方式，然后立即返回，当内核将数据拷贝到缓冲区后，再通知应用程序。

# 为什么要有page cache，操作系统怎么设计的page cache

**加快从磁盘读取文件的速率**。page cache中有一部分磁盘文件的缓存，因为从磁盘中读取文件比较慢，所以读取文件先去page cache中去查找，如果命中，则不需要去磁盘中读取，大大加快读取速度。在Linux内核中，文件的每个数据块最多只能对应一个Page Cache项，它通过两个数据结构来管理这些Cache项，一个是radix tree，另一个是双向链表。Radix tree是一种搜索树，Linux内核利用这个数据结构来通过文件内偏移快速定位Cache项。

# 怎么实现线程池

1）设置一个生产者消费者队列，作为临界资源。

2）初始化n个线程，并让其运行起来，加锁去队列取任务运行。

3）当任务队列为空的时候，所有线程阻塞。

4）当生产者队列来了一个任务后，先对队列加锁，把任务挂在到队列上，然后使用条件变量去通知阻塞中的一个线程。

# 线程的哪些资源共享，哪些资源不共享

（1）共享的资源有

1）**堆**：由于堆是在进程空间中开辟出来的，所以它是理所当然地被共享的；因此new出来的都是共享的（16位平台上分全局堆和局部堆，局部堆是独享的）。

2）**全局变量**：它是与具体某一函数无关的，所以也与特定线程无关；因此也是共享的。

3）**静态变量：**虽然对于局部变量来说，它在代码中是“放”在某一函数中的，但是其存放位置和全局变量一样，存于堆中开辟的.bss和.data段，是共享的。

4）**文件等公用资源**：使用这些公共资源的线程必须同步。Win32提供了几种同步资源的方式，包括信号、临界区、事件和互斥体。

（2）独享的资源有

1）**栈**：栈是独享的。

2）**寄存器**：这个可能会误解，因为电脑的寄存器是物理的，每个线程去取值难道不一样吗？其实线程里存放的是副本，包括程序计数器PC。

# 怎么理解操作系统里的内存碎片，有什么解决办法？

内存碎片分为：**内部碎片和外部碎片**。

内部碎片就是已经被分配出去（能明确指出属于哪个进程）却不能被利用的内存空间；内部碎片是处于区域内部或页面内部的存储块。占有这些区域或页面的进程并不使用这个存储块。而在进程占有这块存储块时，系统无法利用它。直到进程释放它，或进程结束时，系统才有可能利用这个存储块。单道连续分配只有内部碎片。多道固定连续分配既有内部碎片，又有外部碎片。

外部碎片指的是还没有被分配出去（不属于任何进程），但由于太小了无法分配给申请内存空间的新进程的内存空闲区域。外部碎片是出于任何已分配区域或页面外部的空闲存储块。这些存储块的总和可以满足当前申请的长度要求，但是由于它们的地址不连续或其他原因，使得系统无法满足当前申请。使用**伙伴系统算法**。

# 系统如何提高并发性

（1）提高CPU并发计算能力

1）多进程&多线程。

2）减少进程切换，使用线程，考虑进程绑定CPU。

3）减少使用不必要的锁，考虑无锁编程。

4）考虑进程优先级。

5）关注系统负载。

（2）改进I/O模型

1)DMA技术。

2)异步I/O。

3)改进多路I/O就绪通知策略，epoll。

4)Sendfile。

5)内存映射。

6)直接I/O。

# 进程

**（1）进程的概念**

1）通俗的讲程序是一个包含可以执行代码的文件，是一个静态的文件。而进程是一个开始执行但是还没有结束的程序的实例，就是可执行文件的具体实现。 一个程序可能有许多进程，而每一个进程又可以有许多子进程，依次循环下去，而产生子孙进程。

2）当程序被系统调用到内存以后,系统会给程序分配一定的资源(内存,设备等等)然后进行一系列的复杂操作,使程序变成进程以供系统调用。在系统里面只有进程没有程序,为了区分各个不同的进程,系统给每一个进程分配了一个ID(就象我们的身份证)以便识别。

3）为了充分的利用资源，系统还对进程区分了不同的状态。将进程分为新建，运行，阻塞，就绪和完成五个状态。新建表示进程正在被创建。运行是进程正在运行。阻塞是进程正在等待某一个事件发生，就绪是表示系统正在等待CPU来执行命令，而完成表示进程已经结束了系统正在回收资源。

**（2）进程描述符**

为了管理进程，内核需要对每个进程的属性和所需要做的事情，进行清楚的描述，这个就是进程描述符的作用，Linux中的**进程描述符由task\_struct标识**。进程描述符完整的保存了一个进程的属性和生命周期内的数据、状态和行为，由一个复杂的数据结构task\_struct来表示。

**（3）如何创建一个进程**

**写时复制技术**：当新进程（子进程）被创建时，Linux内核并不会立马将父进程的内容复制给子进程，而仅仅当进程空间的内容发生变化时，才执行复制操作。写时复制技术允许父子进程读取相同的物理页，**只要两者有一个试图更改页内容，内核就会把这个页的内容拷贝到新的物理页，并把这块页分给正在写的进程**。

Linux中有**三种系统调用可以创建进程clone()、fork()、vfork()：**

* clone(): 最基础的创建进程的系统调用，可以指明子进程的基础属性（由各种FLAG标识）、堆栈等等。
* fork(): 通过clone()实现，它的堆栈指向的是父进程的堆栈，因此父子进程共享同一个用户态堆栈。**fork的子进程需要完全copy父进程的内存空间**，但是得益于**写时复制技术**，这个过程其实挺快。
* vfork(): 也是基于clone()来实现的，是历史上对fork()的优化，因为fork()需要copy父进程的内存空间，并且fork()后常常执行exec()将另一个程序加载进来，**在写时复制技术之前，这种不必要的copy是代价是比较高昂的**。因此vfork()实现时，会指明flag告诉clone()共享父进程的虚拟内存空间，以加快进程的创建过程。

当计算机开机的时候，内核(kernel)只建立了一个init进程。Linux内核并不提供直接建立新进程的系统调用。剩下的所有进程都是init进程通过fork机制建立的。新的进程要通过老的进程**复制自身**得到，这就是**fork**。fork是一个系统调用。进程存活于内存中。每个进程都在内存中分配有属于自己的一片空间 (address space)。当进程fork的时候，Linux在内存中开辟出一片新的内存空间给新的进程，并将老的进程空间中的内容复制到新的空间中，此后两个进程同时运行。

老进程成为新进程的父进程(parent process)，而相应的，新进程就是老的进程的子进程(child process)。一个进程除了有一个PID之外，还会有一个**PPID**(parent PID)来存储的父进程PID。如果我们循着PPID不断向上追溯的话，总会发现其源头是init进程。所以说，所有的进程也构成一个以init为根的树状结构。**可以用pstree命令来显示整个进程树。也可以使用ps -o pid,ppid,cmd来查询进程信息。**

**fork**通常作为一个函数被调用。**这个函数会有两次返回**，将子进程的PID返回给父进程，0返回给子进程。实际上，子进程总可以查询自己的PPID来知道自己的父进程是谁，这样，一对父进程和子进程就可以随时查询对方。

通常在调用fork函数之后，程序会设计一个if选择结构。当**PID等于0**时，说明该进程为子进程，那么让它执行某些指令,比如说使用**exec**库函数(library function)读取另一个程序文件，并在当前的进程空间执行 (这实际上是我们使用fork的一大目的: 为某一程序创建进程)；而当**PID为一个正整数时**，说明为父进程，则执行另外一些指令。由此，就可以在子进程建立之后，让它执行与父进程不同的功能。

**上下文切换：进程创建好之后，内核必须有能力挂起正在CPU运行的进程，并切换其他进程到CPU上执行。这种过程被称作为进程切换、任务切换或者上下文切换。**

**（4）init进程**

Linux进程是以树形的结构组织的，每一个进程都有唯一的进程标识，简称PID。PID为1的常常是init进程，它相对于普通进程来说，有三个特殊之处：

* **它没有默认的信号处理**，因此如果发信号给init进程的话，会被它忽略掉，除非显示的注册过该信号。
* 如果一个进程退出时，它还有子进程存在，被称为孤儿进程，**那么这些孤儿进程会重新成为init进程的子进程**，转由init进程来管理这些子进程，包括回收退出状态、从进程表中移除等。
* 如果init进程跪了，那么所有用户进程都会被退出。

与孤儿进程类似的是僵尸进程，清理僵尸进程的方法，是杀掉不断产生僵尸进程的父进程，然后这些僵尸进程会称为孤儿进程，由init进程接管、回收。

**（6）子进程的终结(termination)**

当子进程终结时，它会通知父进程，并清空自己所占据的内存，并在内核里留下自己的退出信息(exit code，如果顺利运行，为0；如果有错误或异常状况，为>0的整数)。在这个信息里，会解释该进程为什么退出。父进程在得知子进程终结时，有责任对该子进程使用wait系统调用。这个wait函数能从内核中取出子进程的退出信息，并清空该信息在内核中所占据的空间。但是，如果父进程早于子进程终结，子进程就会成为一个孤儿(orphand)进程。孤儿进程会被过继给init进程，init进程也就成了该进程的父进程。init进程负责该子进程终结时调用wait函数。

当然，一个糟糕的程序也完全可能造成子进程的退出信息滞留在内核中的状况（父进程不对子进程调用wait函数），这样的情况下，子进程成为僵尸（zombie）进程。当大量僵尸进程积累时，内存空间会被挤占

**（7）Linux启动**

1）**内核引导**：当计算机电源通电后，加载**BIOS并进行开机自检**，按照BIOS中设置的启动设备。

2）**导入MBR**：读取**磁盘中的第一个分区，也被称为MBR(Master Boot Record)主引导记录，被读入到一个固定的内存区域并执行**。

3）**调入内核**：操作系统接管硬件以后，首先读入/boot目录下的内核文件。**boot程序被调入内核**，把控制权移交给内核。

4）**init进程**：**内核配置完所有硬件后**，**创建init进程**(进程1)和**守护进程(进程 2)**。init进程是系统所有进程的起点。

5）**运行级别**：**init进程依据/etc/inittab文件夹来设定运行级别**。

6）**系统初始化**：init进程执行rc.sysinit。

7）**建立终端**：rc执行完毕后，返回init。init会打开6个终端，以便用户登录系统。

9）用户登录系统：执行/bin/login程序，启动mingetty，进入登录状态

# Linux 内核的同步方式

**原因：**在现代操作系统里，同一时间可能有多个内核执行流在执行，因此内核其实像多进程多线程编程一样也需要一些同步机制来同步各执行单元对共享数据的访问。尤其是在多处理器系统上，更需要一些同步机制来同步不同处理器上的执行单元对共享的数据的访问。

**同步方式：**

* 原子操作
* 信号量（semaphore）
* 读写信号量（rw\_semaphore）
* 自旋锁（spinlock）
* 大内核锁（BKL，Big Kernel Lock）
* 读写锁（rwlock）
* 大读者锁（brlock-Big Reader Lock）
* 读-拷贝修改(RCU，Read-Copy Update)
* 顺序锁（seqlock）

# 文件系统

* Windows：FCB 表 + FAT + 位图
* Unix：inode + 混合索引 + 成组链接

# 页面置换算法

在地址映射过程中，若在页面中发现所要访问的页面不在内存中，则产生缺页中断。当发生缺页中断时，如果操作系统内存中没有空闲页面，则操作系统必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。而用来选择淘汰哪一页的规则叫做页面置换算法。

**分类**

* 全局置换：在整个内存空间置换。
* 局部置换：在本进程中进行置换。

**算法**

**全局：**

* 工作集算法。
* 缺页率置换算法。

**局部：**

* 最佳置换算法（OPT）。
* 先进先出置换算法（FIFO）。
* 最近最久未使用（LRU）算法。
* 时钟（Clock）置换算法。

# 内核态(内核空间)和用户态(用户空间)的区别和联系

用户空间就是用户进程所在的内存区域，相对的，系统空间就是操作系统占据的内存区域。用户进程和系统进程的所有数据都在内存中。通过**系统调用**将Linux整个体系分为用户态和内核态（或者说内核空间和用户空间）。其实从本质上说内核是一种**特殊的软件程序**，**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。系统调用是操作系统的最小功能单位。**现代操作系统结构是**微内核操作系统。**

（1）微内核操作系统

微内核操作系统往往采用的是C/S模式，它把操作系统分为**微内核和多个服务器**。微内核主要用于：实现与硬件紧密相关的处理、**实现一些较基本的功能**：负责客户和服务器之间的通信。

**内核的功能：**

**1）进程（线程）管理 （进程或者线程的调度）**

**2）低级存储器管理 （用户程序逻辑空间到内存空间的物理地址的变换）**

**3）中断和陷入管理 （中断和陷入）**

由于微内核结构的存在，那么程序就运行在两种不同的地方，**内核态和用户态**，内核态与用户态是操作系统的两种运行级别。

（2）内核态和用户态

**内核态：**当一个任务（进程）执行**系统调用**而陷入内核代码中执行时，我们就称进程处于内核运行态（或简称为内核态）。其他的属于用户态。用户程序运行在用户态，操作系统运行在内核态（操作系统内核运行在内核态，而服务器运行在用户态）。用户态不能干扰内核态，所以CPU指令就有两种：**特权指令和非特权指令**。不同的状态对应不同的指令。特权指令：只能由操作系统内核部分使用，不允许用户直接使用的指令。**如，I/O指令、置终端屏蔽指令、清内存、建存储保护、设置时钟指令（这几种记好，属于内核态）。**非特权指令：所有程序均可直接使用。

所以：

**系统态**（核心态、特态、管态）：执行全部指令。

**用户态**（常态、目态）：执行非特权指令。

（3）用户态和内核态的转换

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。

（4）常见的内核态总结：

内核的功能：

**1）进程（线程）管理 （进程或者线程的调度）**

**2）低级存储器管理 （用户程序逻辑空间到内存空间的物理地址的变换）**

**3）中断和陷入管理 （中断和陷入）**

具体的：

**1） I/O指令、置终端屏蔽指令、清内存、建存储保护、设置时钟指令。**

**2） 中断、异常、陷入，比如缺页中断等**

**3）进程（线程）管理**

**4）系统调用，比如调用了设备驱动程序**

**5）用户内存地址的转换（逻辑—> 物理映射）**

# IO多路复用

IO多路复用是一种同步IO模型，实现一个线程可以监视多个文件句柄；一旦某个文件句柄就绪，就能够通知应用程序进行相应的读写操作；没有文件句柄就绪时会阻塞应用程序，交出CPU。**多路是指网络连接，复用指的是同一个线程**。

1. 为什么用IO多路复用机制？

在单核CPU中，通常用多进程/线程来处理多个事件。但是多进程/线程有如下几方面的成本：

* + - 线程/进程创建成本。
    - CPU切换不同线程/进程成本。
    - 多线程的资源竞争。

有没有一种可以在单线程/进程中处理多个事件流的方法呢？一种答案就是IO多路复用。因此IO多路复用解决的本质问题是在用更少的资源完成更多的事。

1. I/O模型

目前Linux系统中提供了5种IO处理模型：**同步阻塞IO（BIO）、同步非阻塞IO（NIO）、IO多路复用、信号驱动IO、异步非阻塞IO（AIO）**。这五种IO模型中前四个都是同步的IO，只有最后一个是异步IO。信号驱动IO使用的比较少。

1）同步阻塞IO（BIO）

这是最常用的简单的IO模型。阻塞IO意味着当我们发起一次IO操作后一直等待成功或失败之后才返回，在这期间程序不能做其它的事情。阻塞IO操作只能对单个文件描述符进行操作，通过调用read或write来进行操作的。

服务端采用单线程，当accept一个请求后，在recv或send调用阻塞时，将无法accept其他请求（必须等上一个请求处recv或send完），无法处理并发。

服务器端采用多线程，当accept一个请求后，开启线程进行recv，可以完成并发处理，但随着请求数增加需要增加系统线程，大量的线程占用很大的内存空间，并且线程切换会带来很大的开销，10000个线程真正发生读写事件的线程数不会超过20%，每次accept都开一个线程也是一种资源浪费。

2）同步非阻塞（NIO）

我们在发起IO时，通过对文件描述符设置O\_NONBLOCK flag来指定该文件描述符的IO操作为非阻塞。**非阻塞IO通常发生在一个for循环当中**，因为每次进行IO操作时要么IO操作成功，要么当IO操作会阻塞时返回错误EWOULDBLOCK/EAGAIN，然后再根据需要进行下一次的for循环操作，这种类似**轮询**的方式会浪费很多不必要的CPU资源，是一种糟糕的设计。和阻塞IO一样，非阻塞IO也是通过调用read或write来进行操作的，也**只能对单个描述符进行操作**。

服务器端当accept一个请求后，加入fds集合，每次轮询一遍fds集合recv(非阻塞)数据，没有数据则立即返回错误，每次轮询所有fd（包括没有发生读写事件的fd）会很浪费cpu。

3）IO多路复用（现在的做法）

IO多路复用在Linux下包括了三种，select、poll、epoll。抽象来看，他们功能是类似的，但具体细节各有不同：**首先**都会对一组文件描述符进行**相关事件的注册**，**然后阻塞等待某些事件的发生或等待超时**。IO多路复用都可以关注多个文件描述符，但对于这三种机制而言，不同数量级文件描述符对性能的影响是不同的。

服务器端采用单线程通过select/epoll等系统调用获取fd列表，遍历有事件的fd进行accept/recv/send，使其能支持更多的并发连接请求。

4）信号驱动IO

信号驱动IO是利用信号机制，让内核告知应用程序文件描述符的相关事件。但信号驱动IO在网络编程的时候通常很少用到，因为在网络环境中，和socket相关的读写事件太多了，比如下面的事件都会导致SIGIO信号的产生：

1：TCP连接建立。

2：一方断开TCP连接请求。

3：断开TCP连接请求完成。

4：TCP连接半关闭。

5：数据到达TCP socket。

6：数据已经发送出去(如：写buffer有空余空间)。

上面所有的这些都会产生SIGIO信号，但我们没办法在SIGIO对应的信号处理函数中区分上述不同的事件，SIGIO只应该在IO事件单一情况下使用，比如说用来监听端口的socket，因为只有客户端发起新连接的时候才会产生SIGIO信号。

5）异步非阻塞IO（AIO）

异步IO和信号驱动IO差不多，但它比信号驱动IO可以多做一步：相比信号驱动IO需要在程序中完成数据从用户态到内核态(或反方向)的拷贝，异步IO可以把拷贝这一步也帮我们完成之后才通知应用程序。我们使用aio\_read来读，aio\_write写。

1. 同步与异步的区别

**同步：**发送一个请求，等待返回，再发送下一个请求，同步可以避免出现死锁，脏读的发生。

**异步：**发送一个请求，不等待返回，随时可以再发送下一个请求，可以提高效率，保证并发。

1. 阻塞和非阻塞

**阻塞：**传统的IO流都是阻塞式的。也就是说，当一个线程调用read()或者write()方法时，该线程将被阻塞，直到有一些数据读读取或者被写入，在此期间，该线程不能执行其他任何任务。在完成网络通信进行IO操作时，由于线程会阻塞，所以服务器端必须为每个客户端都提供一个独立的线程进行处理，当服务器端需要处理大量的客户端时，性能急剧下降。

**非阻塞**：当线程从某通道进行读写数据时，若没有数据可用时，该线程会去执行其他任务。线程通常将非阻塞IO的空闲时间用于在其他通道上执行IO操作，所以单独的线程可以管理多个输入和输出通道。因此NIO可以让服务器端使用一个或有限几个线程来同时处理连接到服务器端的所有客户端。

1. 同步IO vs 异步IO

1）同步IO指的是程序会一直阻塞到IO操作如read、write完成。

2）异步IO指的是IO操作不会阻塞当前程序的继续执行。

所以根据这个定义，上面阻塞IO当然算是同步的IO，非阻塞IO也是同步IO，因为当文件操作符可用时我们还是需要阻塞的读或写，同理IO多路复用和信号驱动IO也是同步IO，只有异步IO是完全完成了数据的拷贝之后才通知程序进行处理，没有阻塞的数据读写过程。

1. BIO、NIO、AIO适用场景
   1. BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中。
   2. NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂。
   3. AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂。
2. IO多路复用的三种实现方式（select、poll、epoll）
3. select

select的调用会阻塞到有文件描述符可以进行IO操作或被信号打断或者超时才会返回。select将监听的文件描述符分为三组，每一组监听不同的需要进行的IO操作。readfds是需要进行读操作的文件描述符，writefds是需要进行写操作的文件描述符，exceptfds是需要进行[异常事件](https://link.zhihu.com/?target=http%3A//man7.org/linux/man-pages/man2/poll.2.html" \t "_blank)处理的文件描述符。这三个参数可以用NULL来表示对应的事件不需要监听。

当select返回时，每组文件描述符会被select过滤，只留下可以进行对应IO操作的文件描述符。FD\_xx系列的函数是用来操作文件描述符组和文件描述符的关系。FD\_ZERO用来清空文件描述符组。每次调用select前都需要清空一次。FD\_SET添加一个文件描述符到组中，FD\_CLR对应将一个文件描述符移出组中。FD\_ISSET检测一个文件描述符是否在组中，我们用这个来检测一次select调用之后有哪些文件描述符可以进行IO操作。**select可同时监听的文件描述符数量是通过FS\_SETSIZE来限制的，在Linux系统中，该值为1024**，当然我们可以增大这个值，但随着监听的文件描述符数量增加，select的效率会降低。pselect和select大体上是一样的，但有一些细节上的区别。

1. select缺点

* 单个进程所打开的FD是有限制的，通过FD\_SETSIZE设置，默认1024。
* 每次调用select，都需要把fd集合**从用户态拷贝到内核态**，这个开销在fd很多时会很大。
* 对socket扫描时是线性扫描，采用轮询的方法，效率较低（高并发时）。

1. poll

相关函数定义:

**int** **poll**(**struct** pollfd **\***fds, nfds\_t nfds, **int** timeout);

**和select用三组文件描述符不同的是**，poll只有一个pollfd数组，数组中的每个元素都表示一个需要监听IO操作事件的文件描述符。events参数是我们需要关心的事件，revents是所有内核监测到的事件。

1. poll缺点（poll与select相比，只是没有fd的限制，其它基本一样）

* 每次调用poll，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大。
* 对socket扫描时是线性扫描，采用轮询的方法，效率较低（高并发时）。

1. epoll

相关函数定义如下

**int** **epoll\_create**(**int** size);

**int** **epoll\_create1**(**int** flags);

**int** **epoll\_ctl**(**int** epfd, **int** op, **int** fd, **struct** epoll\_event **\***event);

epoll\_create&epoll\_create1用于创建一个epoll实例，而epoll\_ctl用于往epoll实例中增删改要监测的文件描述符，epoll\_wait则用于阻塞的等待可以执行IO操作的文件描述符直到超时。

1. epoll缺点：**epoll只能工作在linux下。**
2. epoll的level-triggered and edge-triggered

这两种底层的事件通知机制通常被称为**水平触发和边沿触发（状态持续通知和状态变化通知)。**这两个概念来自电路，triggered代表电路激活，也就是有事件通知给程序，**level-triggered表示只要有IO操作可以进行比如某个文件描述符有数据可读，每次调用epoll\_wait都会返回以通知程序可以进行IO操作，edge-triggered表示只有在文件描述符状态发生变化时，调用epoll\_wait才会返回，如果第一次没有全部读完该文件描述符的数据而且没有新数据写入，再次调用epoll\_wait都不会有通知给到程序，因为文件描述符的状态没有变化。**

select和poll都是状态持续通知的机制，且不可改变，只要文件描述符中有IO操作可以进行，那么select和poll都会返回以通知程序。而epoll两种通知机制可选。

1. epoll LT（level-triggered）与ET（edge-triggered）模式的区别

* epoll有EPOLLLT和EPOLLET两种触发模式，LT是默认的模式，ET是“高速”模式。
* LT模式下，只要这个fd还有数据可读，每次epoll\_wait都会返回它的事件，提醒用户程序去操作。
* ET模式下，它只会提示一次，直到下次再有数据流入之前都不会再提示了，无论fd中是否还有数据可读。所以在ET模式下，read一个fd的时候一定要把它的buffer读完，或者遇到EAGAIN错误。

1. 常见软件的IO多路复用方案：

1：redis: Linux下epoll(level-triggered)，没有epoll用select。

2：nginx：Linux下epoll(edge-triggered)，没有epoll用select。

1. select/poll/epoll之间的区别

|  |  |  |  |
| --- | --- | --- | --- |
|  | **select** | **poll** | **epoll** |
| 数据结构 | bitmap | 链表 | 红黑树 |
| 最大连接数 | 1024 | 无上限 | 无上限 |
| fd拷贝 | 每次调用select时拷贝 | 每次调用poll时拷贝 | fd首次调用epoll\_ctl时拷贝，每次调用epoll\_wait不拷贝 |
| 工作效率 | 轮询：O(n) | 轮询：O(n) | 回调：O(1) |

1. 不同IO多路复用方案优缺点
   1. poll vs select

**poll和select基本上是一样的。**

**poll相比select好在如下几点：**

1：**poll传参对用户更友好**。比如不需要和select一样计算很多奇怪的参数比如nfds(值最大的文件描述符+1)，再比如不需要分开三组传入参数。

2：**poll会比select性能稍好些**，因为select是每个bit位都检测，假设有个值为1000的文件描述符，select会从第一位开始检测一直到第1000个bit位。但**poll检测的是一个数组**。

3：select的时间参数在返回的时候各个系统的处理方式不统一，如果希望程序可移植性更好，需要每次调用select都初始化时间参数。

: **而select比poll好在下面几点：**

1：支持select的系统更多，兼容更强大，有一些unix系统不支持poll。

2：select提供精度更高(到microsecond)的超时时间，而poll只提供到毫秒的精度。

* 1. epoll vs poll&select

epoll优于select&poll在下面几点：

1：在需要同时监听的文件描述符数量增加时，select&poll是O(N)的复杂度，epoll是O(1)，在N很小的情况下，差距不会特别大，但如果N很大的前提下，一次O(N)的循环可要比O(1)慢很多，所以高性能的网络服务器都会选择epoll进行IO多路复用。

2：epoll内部用一个文件描述符挂载需要监听的文件描述符，这个epoll的文件描述符可以在多个线程/进程共享，所以epoll的使用场景要比select&poll要多。

# 共享内存实现

* 1. POSIX 共享内存。
  2. XSI共享内存。
  3. mmap 共享内存。

# Reactor和Proactor

服务器通常需要处理三种事件：I/O事件，信号事件，定时事件。同步I/O模型常用于实现Reactor，异步I/O模型常用于Proactor。

* 1. Reactor

要求主线程（I/O处理单元）只负责监听文件描述符上是否有事件发生，有的话就立即将事件通知工作线程（逻辑单元）数据的读写，接受新的连接以及处理客户请求均在工作线程中完成；除此之外，逻辑线程不作任何工作。

1. 主线程往epoll内核事件表中注册socket上的读就绪事件。
2. 主线程调用epoll\_wait等待socket上有数据可读。
3. 当socket上有数据可读时，epoll\_wait通知主线程，主线程则将socket可读事件放入请求队列。
4. 睡眠在请求队列上的某个工作线程被唤醒，它从socket读取数据，并处理客户请求，然后往epoll内核事件表中注册该socket上的写就绪事件。
5. 主线程调用epoll\_wait等待socket可写。
6. 当socket可写时，epoll\_wait通知主线程，主线程将socket可写事件放入请求队列。
7. 睡眠在请求队列上的某个工作线程（工作线程从请求队列读取事件后，根据事件的类型来决定如何处理它，没有必要区分读工作线程和写工作线程）被唤醒，它往socket上写入服务器处理客户请求的结果。
   1. Proactor

Proactor将所有I/O操作都交给主线程和内核来处理，工作线程仅仅负责业务逻辑。使用异步I/O模型（aio\_read和aio\_write）来实现Proactor模式的工作流程是：

1. 主线程调用aio\_read向内核注册socket上的读完成事件，并告诉内核用户缓冲区的位置，以及读操作完成时如何通知应用程序（可以用信号）。
2. 主线程继续处理其他逻辑。
3. 当socket上的读数据被读入用户缓冲区后，内核向应用进程发送一个信号，已通知应用程序数据已经可用。
4. 应用进程预先定义好的信号处理函数选择一个工作线程来处理处理客户请求，工作线程处理完客户请求之后，调用aio\_write向内核注册socket的完成写事件，并告诉内核用户写缓冲区的位置，以及操作完成时如何通知应用程序（可以用信号）。
5. 主线程继续处理其他逻辑。
6. 当用户缓冲区的数据被写入socket之后，内核将向应用程序发送一个信号，已通知应用程序数据已经发送完毕。
7. 应用程序预先定义好的信号处理函数选择一个工作线程来做善后处理，比如决定是否关闭socket。

可使用同步I/O模拟Proactor：主线程执行读数据操作，读完数据之后，主线程向工作线程通知这一“完成事件”，那么从工作线程的角度来看，他们直接获得了读写数据的结果，只需要对读写数据进行处理：

1. 主线程往epoll内核事件表上注册socket上的读就绪。
2. 主线程调用epoll\_wait等待socket上有数据可读。
3. 当socket上有数据可读时，epoll\_wait通知主线程，主线程从socket上循环读取数据，直到没有更多数据可读，然后将读取到的数据封装成一个请求对象并插入到请求队列。
4. 睡眠在请求队列上的某个工作线程被唤醒，他获得请求对象并处理客户请求，然后往epoll内核事件表中注册socket上的写就绪。
5. 主线程调用epoll\_wait等待socket可写。
6. 当socket可写时，epoll\_wait通知主线程，主线程往socket上写入服务器处理客户请求结果。

# malloc函数的实现

malloc函数的实质是它有一个将可用的内存块连接为一个长长的列表的所谓**空闲链表**。 调用malloc函数时，它沿着连接表寻找一个大到足以满足用户请求所需要的内存块。 然后，将该内存块一分为二（一块的大小与用户申请的大小相等，另一块的大小就是剩下来的字节）。 接下来，将分配给用户的那块内存存储区域传给用户，并将剩下的那块（如果有的话）返回到连接表上。 调用free函数时，它将用户释放的内存块连接到空闲链表上。到最后，空闲链会被切成很多的小内存片段，如果这时用户申请一个大的内存片段， 那么空闲链表上可能没有可以满足用户要求的片段了。于是，malloc函数请求延时，并开始在空闲链表上检查各内存片段，对它们进行内存整理，将相邻的小空闲块合并成较大的内存块。

* 1. malloc分配内存前的初始化：

malloc\_init是初始化内存分配程序的函数。它完成以下三个目的：将分配程序标识为已经初始化；找到操作系统中最后一个有效的内存地址；然后建立起指向需要管理的内存的指针。

* 1. 内存块的获取

所要申请的内存是由多个内存块构成的链表。

* + 1. 内存块的大致结构：每个块由meta区和数据区组成，meta区记录数据块的元信息（数据区大小、空闲标志位、指针等等），数据区是真实分配的内存区域，并且数据区的第一个字节地址即为malloc返回的地址。
    2. 寻找合适的block。
    3. 如果现有block都不能满足size的要求，则需要在链表最后开辟一个新的block。
  1. 内存分配

**malloc工作机制/实现原理**：

它有一个将可用的内存块连接为一个长长的列表的所谓**空闲链表**。调用malloc函数时，它沿连接表寻找一个大到足以满足用户请求所需要的内存块。然后将该内存块分割成用户所需要的内存大小，把要分配给用户的内存传给用户，剩下的内存返回到连接表上。

**free的实现原理：**

调用free时，它将用户释放的内存块连接到空闲链上。所以最后空闲链可能被分割成了很多的小内存片段，而这时候用户申请一个很大的内存片段，空闲链上可能就没有这么大的片段了。这时候malloc请求延时，在空闲链上合并相邻的小空间块。