目录

1. 操作系统的主要功能、操作系统的主要目的是什么 3

2. 软件访问硬件的几种方式 3

3. 内存管理机制 3

4. 操作系统文件管理 5

5. 进程、线程和协程的区别和联系 6

6. 线程与进程的比较 8

7. 不同的线程什么内存空间共享，什么内存空间不共享？ 8

8. 协程（及使用场景、协程之间切换的具体过程、作用、切换过程cpu和内存具体变化） 8

9. 多线程多进程使用场景（什么情况下使用多进程、什么情况下使用多线程） 8

10. Linux系统下线程是怎么实现的？（什么语言支持协程） 8

11. 多线程有哪些库？你用哪个？为什么？ 8

12. 进程fork后父子进程什么相同什么不同 8

13. 外中断和异常有什么区别？ 9

14. 死锁，及原因，及如何避免死锁(解决方法) 9

15. 银行家算法描述，找到造成死锁的进程怎么处理（杀死、抢占、回滚）？回滚如何实现 10

16. 知道浮点数吗，在计算机中怎么表示的，说下原理 10

17. 线程同步方式 11

18. cache一致性 11

19. 原子操作 11

20. i++是原子的吗？为什么不是原子的？会出现什么情况？ 12

21. 操作系统堆和栈的区别，局部变量在哪个区（经典问题了，但是答的不是很好-。 12

22. TCP和UDP的包头多大记得吗，长什么样 （大概描述了一下里面的字段 13

23. 什么是非阻塞IO 14

24. 布隆过滤器 16

25. 虚拟内存与物理内存 17

26. 怎么通过逻辑地址找到物理地址 19

27. 进程间通信的方式（本质区别） 19

28. 线程间通信的方式 20

29. 内核态用户态的区别 20

26. 线程知道哪些锁（及其运用场景） 21

27. 操作系统实现锁的方法 22

28. 锁机制，如果只显示上锁，怎么样让锁自动释放？ 22

29. 信号量的具体实现 23

30. 多线程存在的问题是什么 23

31. 进程间通信之间的区别 23

32. 如何杀死一个进程， kill实质做了什么事 23

33. 管道通信，管道有哪几类（五种进程间的通信方式） 23

34. 信号量 25

35. 共享内存（为什么说共享内存是进程高级通信方式最快的） 25

36. 消息队列 26

37. 线程间通信方式 26

38. 信号、信号量跟锁 26

39. Linux中的mmap 26

40. 简单说一下什么是IO多路复用 27

41. Malloc申请一块内存会不会立即绑定到物理内存里 27

42. 页面置换算法有哪些 27

43. LRU和LFU的区别 27

44. LRU和LFU如何实现 27

45. 进程调度算法 27

46. 为什么 Linux 系统下的应用程序不能直接在 Windows 下运行 27

47. Epoll中的函数族、ET与LT（使用场景，为什么ET高效） 28

48. Socket函数族，简单使用过程说一下 28

49. 不同链接，请求有多有少，项目怎么实现负载均衡 28

50. 在一个运行多个进程的系统中，如果某一个或者多个进程瞬间把CPU打满，所有的进程都会被卡死，你有什么样子的解决方案呢？ 28

51. 有没有不用加锁的方式，无锁队列 28

52. 服务器如何提高吞吐量、测试项目的时延和传输速率 28

53. 28

54. 一个进程fork出一个子进程，内存占用会不会翻倍 28

55. 多进程和多线程的区别（使用场景） 29

56. Linux 查看磁盘容量、找某个文件、内存占用量命令、查看进程 29

57. 如果让你自己实现一个线程池，你怎么控制这个线程的并发数 29

58. 消息队列了解吗 29

59. 讲一下你项目的设计思想以及那些中间和做这个项目自己的一些考虑 31

60. 项目部署在服务器上，当发现其实一些CPU占用过高的情况下，如何去定位问题 31

61. 手写LRU 32

62. 实现memcpy函数 32

63. 100w个玩家一起创建角色…中途如果有玩家取消创建 32

64. 亿级数据排序 32

65. 在项目面碰到最棘手或者最难的一个问题是什么 32

66. 32

67. 和平精英遇人掉帧问题真的存在吗？还是概率性存在 32

68. 平板玩家普遍比手机玩家操作流畅，有没有想过制衡 32

69. 和平精英的匹配机制很奇怪，我经常赛季没打，后面接着一下子匹配高星段位 32

70. 32

71. 64匹马8赛道怎么决出前4名 32

1. 操作系统的主要功能、操作系统的主要目的是什么、什么是内核、什么是实时系统

一般来说，现代操作系统主要提供下面几种功能

* 进程管理: 进程管理的主要作用就是任务调度，在单核处理器下，操作系统会为每个进程分配一个任务，进程管理的工作十分简单；而在多核处理器下，操作系统除了要为进程分配任务外，还要解决处理器的调度、分配和回收等问题
* 内存管理：内存管理主要是操作系统负责管理内存的分配、回收，在进程需要时分配内存以及在进程完成时回收内存，协调内存资源，通过合理的页面置换算法进行页面的换入换出
* 设备管理：根据确定的设备分配原则对设备进行分配，使设备与主机能够并行工作，为用户提供良好的设备使用界面。
* 文件管理：有效文件的存储空间，合理地组织和管理文件系统，为文件访问和文件保护提供更有效的方法及手段。
* 提供用户接口：操作系统提供了访问应用程序和硬件的接口，使用户能够通过应用程序发起系统调用从而操纵硬件，实现想要的功能。

操作系统是一种软件，它的主要目的有三种

管理计算机资源，这些资源包括 CPU、内存、磁盘驱动器、打印机等。

提供一种图形界面，就像我们前面描述的那样，它提供了用户和计算机之间的桥梁。

为其他软件提供服务，操作系统与软件进行交互，以便为其分配运行所需的任何必要资源。

在计算机中，内核是一个计算机程序，它是操作系统的核心，可以控制操作系统中所有的内容。内核通常是在 boot loader 装载程序之前加载的第一个程序。

实时操作系统对时间做出了严格的要求，实时操作系统分为两种：硬实时和软实时

硬实时操作系统规定某个动作必须在规定的时刻内完成或发生，比如汽车生产车间，焊接机器必须在某一时刻内完成焊接，焊接的太早或者太晚都会对汽车造成永久性伤害。

软实时操作系统虽然不希望偶尔违反最终的时限要求，但是仍然可以接受。并且不会引起任何永久性伤害。比如数字音频、多媒体、手机都是属于软实时操作系统。

你可以简单理解硬实时和软实时的两个指标：是否在时刻内必须完成以及是否造成严重损害。

1. 软件访问硬件的几种方式

软件访问硬件其实就是一种 IO 操作，软件访问硬件的方式，也就是 I/O 操作的方式有哪些。

硬件在 I/O 上大致分为并行和串行，同时也对应串行接口和并行接口。

随着计算机技术的发展，I/O 控制方式也在不断发展。选择和衡量 I/O 控制方式有如下三条原则

（1） 数据传送速度足够快，能满足用户的需求但又不丢失数据；

（2） 系统开销小，所需的处理控制程序少；

（3） 能充分发挥硬件资源的能力，使 I/O 设备尽可能忙，而 CPU 等待时间尽可能少。

根据以上控制原则，I/O 操作可以分为四类

直接访问：直接访问由用户进程直接控制主存或 CPU 和外围设备之间的信息传送。直接程序控制方式又称为忙/等待方式。

中断驱动：为了减少程序直接控制方式下 CPU 的等待时间以及提高系统的并行程度，系统引入了中断机制。中断机制引入后，外围设备仅当操作正常结束或异常结束时才向 CPU 发出中断请求。在 I/O 设备输入每个数据的过程中，由于无需 CPU 的干预，一定程度上实现了 CPU 与 I/O 设备的并行工作。

上述两种方法的特点都是以 CPU 为中心，数据传送通过一段程序来实现，软件的传送手段限制了数据传送的速度。接下来介绍的这两种 I/O 控制方式采用硬件的方法来显示 I/O 的控制

DMA 直接内存访问：为了进一步减少 CPU 对 I/O 操作的干预，防止因并行操作设备过多使 CPU 来不及处理或因速度不匹配而造成的数据丢失现象，引入了 DMA 控制方式。

通道控制方式：通道，独立于 CPU 的专门负责输入输出控制的处理机，它控制设备与内存直接进行数据交换。有自己的通道指令，这些指令由 CPU 启动，并在操作结束时向 CPU 发出中断信号。

1. 内存管理机制

* 操作系统的内存管理主要负责内存的分配与回收（malloc 函数：申请内存，free 函数：释放内存），另外地址转换也就是将逻辑地址转换成相应的物理地址等功能也是操作系统内存管理做的事情

内存管理机制有两种：  
第一种

连续分配管理方式：连续分配管理方式是指一个用户程序分配一个连续的内存空间，常见的如**块式管理**

* 块式管理：将内存分为几个固定大小的块，每个块中只包含一个进程。如果程序运行需要内存的话，操作系统就分配给它一块，如果程序运行只需要很小的空间的话，分配的这块内存很大一部分几乎被浪费了。这些在每个块中未被利用的空间，我们称之为碎片

第二种

非连续分配管理方式：非连续分配管理方式允许一个程序使用的内存分布在离散或者说不相邻的内存中

* 页式管理：把主存分为大小相等且固定的一页一页的形式，页较小，相对相比于块式管理的划分力度更大，提高了内存利用率，减少了碎片。页式管理通过页表对应逻辑地址和物理地址。
* 段式管理： 页式管理虽然提高了内存利用率，但是页式管理其中的页实际并无任何实际意义。 段式管理把主存分为一段段的，每一段的空间又要比一页的空间小很多 。但是，最重要的是段是有实际意义的，每个段定义了一组逻辑信息，例如,有主程序段 MAIN、子程序段 X、数据段 D 及栈段 S 等。 段式管理通过段表对应**逻辑地址和物理地址**
* 段页式管理机制：段页式管理机制结合了段式管理和页式管理的优点。简单来说段页式管理机制就是把主存先分成若干段，每个段又分成若干页，也就是说段页式管理机制中段与段之间以及段的内部的都是离散的。

分页和分段共同点和区别：

共同点：

1. 分页机制和分段机制都是为了提高内存利用率，减少内存碎片
2. 页和段都是离散存储的，所以两者都是离散分配内存的方式。但是每个页和段中的内存是连续的

区别：

1. 目的不同：分页是由于系统管理的需要而不是用户的需要，它是信息的物理单位；分段的目的是为了更好地满足用户地需要，它是信息的逻辑单位，它含有一组意义相对完整地信息
2. 大小不同：页的大小固定且由系统决定，而段的大小却不固定，由其所完成的功能决定
3. 地址空间不同：段向用户提供二维空间地址；页向用户提供是一维空间地址

基本分页储存管理方式

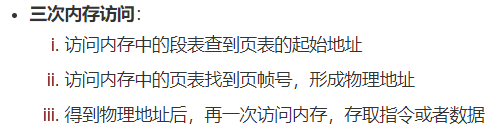
* 在分页内存管理中，很重要的两点是：1. 虚拟地址到物理地址的转换要快【快表】；2. 解决虚拟地址空间大，页表也会很大的问题【多级分页】
* 因为程序数据存储在不同的页面中，而页面又离散的分布在内存中，因此需要一个页表来记录逻辑地址和实际存储地址之间的映射关系，以实现从页号到物理块号的映射
* 由于页表也是存储在内存中的，因此访问分页系统中内存数据需要两次的内存访问【一次是从内存中访问页表，从中找到指定的物理块号，加上页内偏移得到实际物理地址；第二次就是根据第一次得到的物理地址访问内存取出数据】

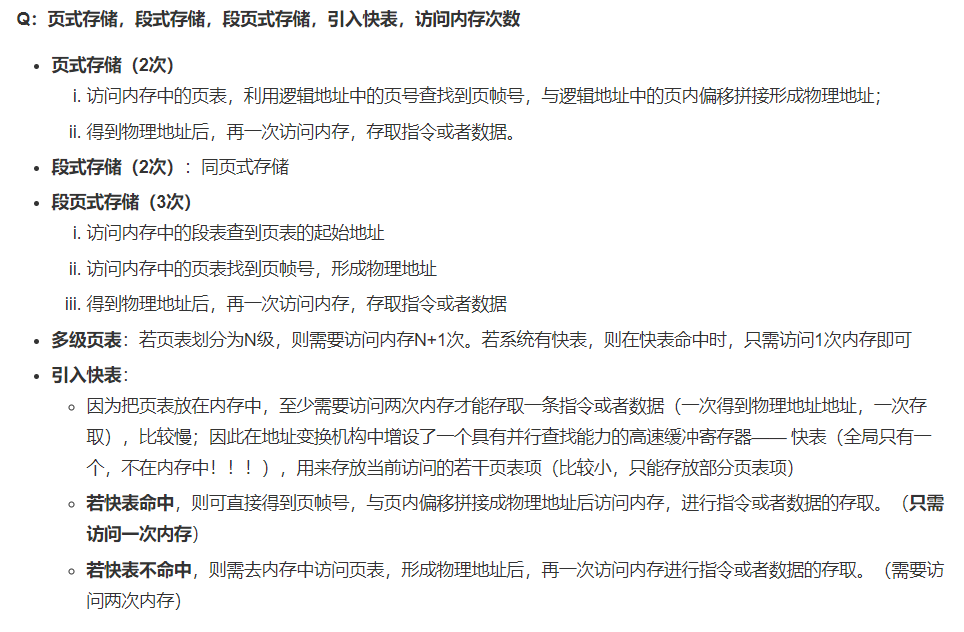
基本分段存储管理方式

* 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。段表中的每一个表项记录了该段在内存中的起始地址和该段的长度。段表可以放在内存中也可以放在寄存器中。
* 访问内存的时候根据段号和段表项的长度计算当前访问段在段表中的位置，然后访问段表，得到该段的物理地址，根据该物理地址以及段内偏移量就可以得到需要访问的内存。由于也是两次内存访问，所以分段管理中同样引入了[联想](https://www.nowcoder.com/jump/super-jump/word?word=联想)寄存器。

段页式内存管理

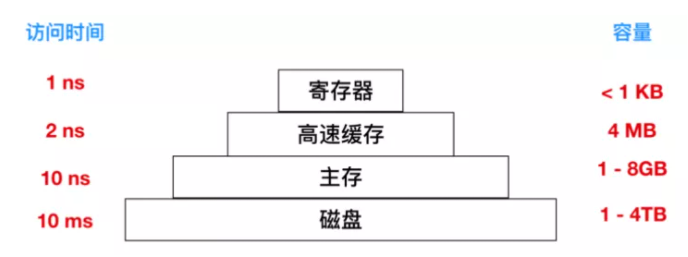
* 页式存储管理能有效地提高内存利用率，而分段存储管理能反映程序的逻辑结构并有利于段的共享
* 段页式管理就是将程序分为多个逻辑段，在每个段里面又进行分页，即将分段和分页组合起来使用。
* 为了实现地址变换，系统为每个进程建立一张段表，而每个分段有一张页表（在一个进程中，段表只有一个，而页表可能有多个）
* 在进行地址变换时，首先通过段表查到页表起始地址，然后通过页表找到页帧号，最后形成物理地址。如图所示，进行一次访问实际需要至少三次访问主存，这里同样可以使用快表以加快查找速度，其关键字由段号、页号组成，值是对应的页帧号和保护码。





1. 操作系统文件管理

访问磁盘的效率要比内存慢很多，是时候又祭出这张图了



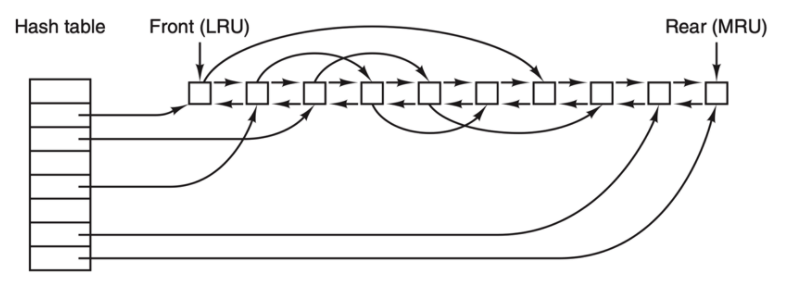
所以磁盘优化是很有必要的，下面我们会讨论几种优化方式

#### 高速缓存

最常用的减少磁盘访问次数的技术是使用 块高速缓存(block cache) 或者 缓冲区高速缓存(buffer cache)。高速缓存指的是一系列的块，它们在逻辑上属于磁盘，但实际上基于性能的考虑被保存在内存中。

管理高速缓存有不同的算法，常用的算法是：**检查全部的读请求，查看在高速缓存中是否有所需要的块**。**如果存在，可执行读操作而无须访问磁盘**。**如果检查块不再高速缓存中，那么首先把它读入高速缓存，再复制到所需的地方。之后，对同一个块的请求都通过高速缓存来完成**。

高速缓存的操作如下图所示



由于在高速缓存中有许多块，所以需要某种方法快速确定所需的块是否存在。常用方法是将设备和磁盘地址进行散列操作。然后在散列表中查找结果。具有相同散列值的块在一个链表中连接在一起（这个数据结构是不是很像 HashMap?），这样就可以沿着冲突链查找其他块。

如果高速缓存已满，此时需要调入新的块，则要把原来的某一块调出高速缓存，如果要调出的块在上次调入后已经被修改过，则需要把它写回磁盘。这种情况与分页非常相似。

#### 块提前读（适用于顺序读取）

第二个明显提高文件系统的性能是**在需要用到块之前试图提前将其写入高速缓存从而提高命中率**。许多文件都是顺序读取。如果请求文件系统在某个文件中生成块 k，文件系统执行相关操作并且在完成之后，会检查高速缓存，以便确定块 k + 1 是否已经在高速缓存。如果不在，文件系统会为 k + 1 安排一个预读取，因为文件希望在用到该块的时候能够直接从高速缓存中读取。

当然，块提前读取策略只适用于实际顺序读取的文件。对随机访问的文件，提前读丝毫不起作用。甚至还会造成阻碍。

减少磁盘臂运动

高速缓存和块提前读并不是提高文件系统性能的唯一方法。另一种重要的技术是**把有可能顺序访问的块放在一起，当然最好是在同一个柱面上，从而减少磁盘臂的移动次数**。当写一个输出文件时，文件系统就必须按照要求一次一次地分配磁盘块。如果用位图来记录空闲块，并且整个位图在内存中，那么选择与前一块最近的空闲块是很容易的。如果用空闲表，并且链表的一部分存在磁盘上，要分配紧邻的空闲块就会困难很多。

1. 什么是设备驱动程序、什么是DMA、直接内存访问的特点

在计算机中，设备驱动程序是一种计算机程序，它能够控制或者操作连接到计算机的特定设备。驱动程序提供了与硬件进行交互的软件接口，使操作系统和其他计算机程序能够访问特定设备，不用需要了解其硬件的具体构造。

DMA 的中文名称是直接内存访问，它意味着 CPU 授予 I/O 模块权限在不涉及 CPU 的情况下读取或写入内存。也就是 DMA 可以不需要 CPU 的参与。这个过程由称为 DMA 控制器（DMAC）的芯片管理。由于 DMA 设备可以直接在内存之间传输数据，而不是使用 CPU 作为中介，因此可以缓解总线上的拥塞。DMA 通过允许 CPU 执行任务，同时 DMA 系统通过系统和内存总线传输数据来提高系统并发性。

DMA 方式有如下特点：

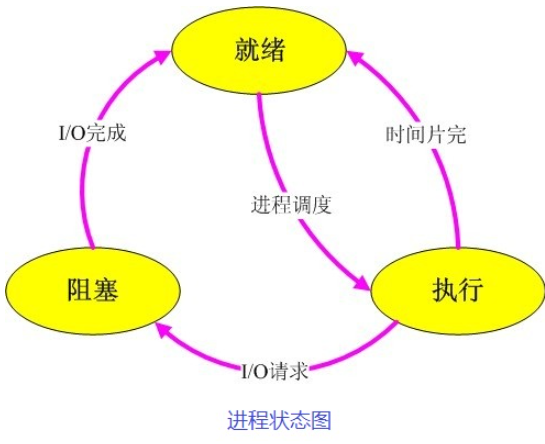
* 数据传送以数据块为基本单位
* 所传送的数据从设备直接送入主存，或者从主存直接输出到设备上
* 仅在传送一个或多个数据块的开始和结束时才需 CPU 的干预，而整块数据的传送则是在控制器的控制下完成。

DMA 方式和中断驱动控制方式相比，减少了 CPU 对 I/O 操作的干预，进一步提高了 CPU 与 I/O 设备的并行操作程度。

DMA 方式的线路简单、价格低廉，适合高速设备与主存之间的成批数据传送，小型、微型机中的快速设备均采用这种方式，但其功能较差，不能满足复杂的 I/O 要求。

1. 进程、线程和协程的区别和联系
2. 进程

进程，直观点说，保存在硬盘上的程序运行以后，会在内存空间里形成一个独立的内存体，这个内存体有自己独立的地址空间，有自己的堆，上级挂靠单位是操作系统。操作系统会以进程为单位，分配系统资源(CPU时间片、内存等资源)，进程是**资源分配**的最小单位。

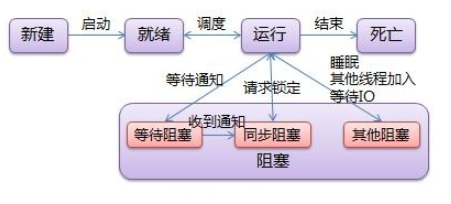
 

进程间通信(IPC):

管道（Pipe）、命名管道（FIFO）、消息队列（Message Queue）、信号量（Semaphore）、共享内存（Shared Memory）；套接字（Socket）.

1. 线程

线程，有时被称为轻量级进程（Lightweight Process,LWP），是操作系统**调度**（CPU调度）执行的最小单位。



1. 进程和线程的区别与联系

区别：

* 调度：线程作为调度和分配的基本单位，进程作为拥有资源的基本单位
* 并发性：不仅进程之间可以并发执行，同一个进程的多个线程之间也可并发执行
* 拥有资源：进程是拥有资源的一个独立单位，线程不拥有系统资源，但可以访问隶属于线程的资源。进程所维护的是程序所包含的资源（静态资源），如：地址空间、打开的文件句柄集、文件系统状态、信号处理handler等；线程锁维护的运行相关的资源（动态资源），如：运行栈、调度相关的控制信息，待处理的信号集等。
* 系统开销：在创建或撤销进程时，由于系统都要为之分配和回收资源，导致系统的开销明显大于创建或撤销线程时的开销。但是进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有独立的地址空间，一个进程死掉就等于所有的线程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。

联系：

* 一个线程只属于一个进程，而一个进程可以有多个线程，但至少有一个线程
* 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
* 处理机分给线程，即真正在处理机上运行的是线程
* 线程在执行过程中，需要协作同步。不同进程的线程要利用消息通信的方法实现同步

1. 一个形象的例子解释进程和线程的区别



这幅图是一个双向多车道的道路图，假如我们把整条道路看成一是一个“进程”的话，那么图中由白色虚线分隔开来的各个车道就是进程中的各个“线程”了。

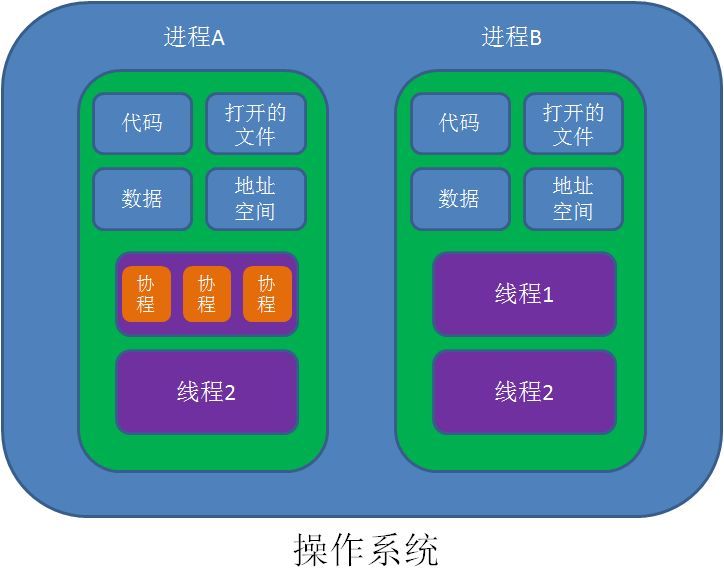
* 这些线程（车道）共享了进程（道路）的公共资源（土地资源）。
* 这些线程（车道）必须依赖于进程（道路），也就是说，线程不能脱离于进程而存在（就像离开了道路，车道也就没有意义了）。
* 这些线程（车道）之间可以并发执行（各个车道你走你的，我走我的），也可以互相同步（某些车道在交通灯亮时禁止继续前行或转弯，必须等待其它车道的车辆通行完毕）。
* 这些线程（车道）之间依靠代码逻辑（交通灯）来控制运行，一旦代码逻辑控制有误（死锁，多个线程同时竞争一个资源），那么线程将陷入混乱，无序之中。
* 这些线程（车道）之间谁先运行是未知的，只有在线程刚好被分配到CPU时间片（交通灯变化）的那一刻才知道

1. 进程/线程之间的亲缘性

亲缘性的意思是进程/线程只在某个CPU上执行（多核系统）

使用CPU亲缘性的好处：设置CPU亲缘性是为了防止进程/线程在CPU的核上频繁切换，从而避免因切换带来的CPU的L1/L2 cache失效，cache失效会降低程序的性能

1. 协程



协程，是一种比线程更加轻量级的存在，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。

子程序，或者称为函数，在所有语言中都是层级调用，比如A调用B，B在执行过程中又调用了C，C执行完毕返回，B执行完毕返回，最后是A执行完毕。所以子程序调用是通过栈实现的，一个线程就是执行一个子程序。子程序调用总是一个入口，一次返回，调用顺序是明确的。而协程的调用和子程序不同。

协程在子程序内部是可中断的，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。



假设由协程执行，在执行A的过程中，可以随时中断，去执行B，B也可能在执行过程中中断再去执行A，结果可能是：1 2 x y 3 z。

协程的特点在于是一个线程执行，那和多线程比，协程有何优势？

* 极高的执行效率：因为子程序的切换不是线程切换，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；
* 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. 线程特有，线程间共享了哪些资源

查理芒格经常说这样一句话：“反过来想，总是反过来想”，如果你对线程之间共享了哪些进程资源这个问题想不清楚的话那么也可以反过来思考，那就是**有哪些资源是线程私有的**。

线程运行的本质其实就是函数的执行，函数的执行总会有一个源头，这个源头就是所谓的入口函数，CPU从入口函数开始执行从而形成一个执行流，只不过我们人为的给执行流起一个名字，这个名字就叫线程。

既然线程运行的本质就是函数的执行，那么函数执行都有哪些信息呢？

在《[函数运行时在内存中是什么样子](http://mp.weixin.qq.com/s?__biz=MzU2NTYyOTQ4OQ%3D%3D&chksm=fcb9817dcbce086b10cb44cad7c9777b0088fb8d9d6baf71ae36a9b03e1f8ef5bec62b79d6f7&idx=1&mid=2247484963&scene=21&sn=542d3bec57c6a9dfc17c83005fd2c030" \l "wechat_redirect)》这篇文章中我们说过，函数运行时的信息保存在栈帧中，栈帧中保存了函数的返回值、调用其它函数的参数、该函数使用的局部变量以及该函数使用的寄存器信息，如图所示，假设函数A调用函数B：

此外，CPU执行指令的信息保存在一个叫做程序计数器的寄存器中，通过这个寄存器我们就知道接下来要执行哪一条指令。由于操作系统随时可以暂停线程的运行，因此我们保存以及恢复程序计数器中的值就能知道线程是从哪里暂停的以及该从哪里继续运行了。

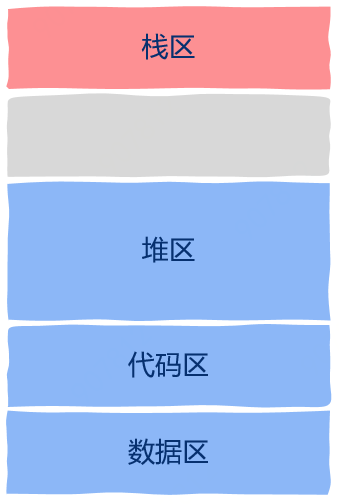
由于线程运行的本质就是函数运行，函数运行时信息是保存在栈帧中的，因此每个线程都有自己独立的、私有的栈区。

同时函数运行时需要额外的寄存器来保存一些信息，像部分局部变量之类，这些寄存器也是线程私有的，一个线程不可能访问到另一个线程的这类寄存器信息。

从上面的讨论中我们知道，到目前为止，所属线程的栈区、程序计数器、栈指针以及函数运行使用的寄存器是线程私有的。

以上这些信息有一个统一的名字，就是线程上下文，thread context。

我们也说过操作系统调度线程需要随时中断线程的运行并且需要线程被暂停后可以继续运行，操作系统之所以能实现这一点，依靠的就是线程上下文信息。



这其实就是进程地址空间的样子，也就是说线程共享进程地址空间中除线程上下文信息中的所有内容，意思就是说线程可以直接读取这些内容。

接下来我们分别来看一下这些区域。

代码区

进程地址空间中的代码区，这里保存的是什么呢？从名字中有的同学可能已经猜到了，没错，这里保存的就是我们写的代码，更准确的是编译后的可执行机器指令。

那么这些机器指令又是从哪里来的呢？答案是从可执行文件中加载到内存的，可执行程序中的代码区就是用来初始化进程地址空间中的代码区的。

 数据区

进程地址空间中的数据区，这里存放的就是所谓的全局变量。

什么是全局变量？所谓全局变量就是那些你定义在函数之外的变量，在C语言中就像这样：

堆区

堆区是程序员比较熟悉的，我们在C/C++中用malloc或者new出来的数据就存放在这个区域，很显然，只要知道变量的地址，也就是指针，任何一个线程都可以访问指针指向的数据，因此堆区也是线程共享的属于进程的资源。

栈区

唉，等等！刚不是说栈区是线程私有资源吗，怎么这会儿又说起栈区了？

确实，从线程这个抽象的概念上来说，栈区是线程私有的，然而从实际的实现上看，栈区属于线程私有这一规则并没有严格遵守，这句话是什么意思？

通常来说，注意这里的用词是通常，通常来说栈区是线程私有，既然有通常就有不通常的时候。

不通常是因为不像进程地址空间之间的严格隔离，线程的栈区没有严格的隔离机制来保护，因此如果一个线程能拿到来自另一个线程栈帧上的指针，那么该线程就可以改变另一个线程的栈区，也就是说这些线程可以任意修改本属于另一个线程栈区中的变量。

1. 深入线程、最多能开多少个线程，Linux 如何看 stack 大小

对32位系统来说，理论上的寻址范围是4G(2^32)，但实际系统还要预留系统空间用于自身管理，一般来说真正的最大寻址范围都要小于4G。对于Linux来说，一个进程拥有的用户空间为0~0xBFFFFFFF，共3G，另外1G是内核空间；Windows的到0x7FFFFFFF，只有2G。（32位系统的寻址范围还可以扩大，应用某些扩展标准，例如PAE，最多能支持到16G到64G不等。但这个情况不在我的讨论范围）。

一个LInux进程在用户态最多访问3G内存，每个线程必须占用特定大小的堆栈空间（stack size），这个因素使32位系统创建的线程数不可能超过n=3G/StackSize   
注：Stack Size可以用ulimit -s查看，单位是k，同时这个size是可以设定的。

在64位的系统中，因为64位系统，没有寻址范围的限制，所以理论上，可以创建的线程数应该是几乎无限的，除非其他条件限制。

但是，我在实际测试中，当创建线程数达到7500左右的时候，出现错误。但是原因一直没能找到。



Linux

ulimit -s 查看线程栈空间stack的size

修改stack size

1.ulimit -s stack\_size （临时）

2.运行程序是否用root运行，避免限制资源

3.pthread\_att\_setstacksize 设置线程栈空间

1. 线程与进程的比较

线程与进程的区别：

* 调度：线程作为调度和分配的基本单位，进程作为拥有资源的基本单位
* 并发性：不仅进程之间可以并发执行，同一个进程的多个线程之间也可以并发执行
* 拥有资源：进程是拥有资源的一个独立单位，线程不拥有资源，但可以访问隶属于进程的资源。进程所维护的是程序所包含的资源（静态资源），如：地址空间、打开的文件句柄集、文件系统状态、信号处理handler等；线程所维护的运行相关的资源（动态资源），如：运行栈、调度相关的控制信息，待处理的信号集等。

线程与进程的联系：

* 一个线程只属于一个进程，而一个进程可以有多个线程，但至少有一个进程
* 资源分配给进程，同一个进程的所有线程共享该进程的所有资源
* 处理机分给线程，即真正在处理机上运行的是线程
* 线程在执行过程中，需要协作同步。不同进程的线程间要利用消息通信的方法实现同步。

1. 协程（及使用场景、协程之间切换的具体过程、作用、切换过程cpu和内存具体变化）
2. 多线程多进程使用场景（什么情况下使用多进程、什么情况下使用多线程）
3. C++ 多线程的十三点建议

可更多使用C++11的std::thread，而不是pthread，推荐std::thread没啥别的性能方面的原因，只是因为使用起来很方便。

● std::thread配合lambda表达式创建个线程运行，很方便！

● thread对象直接join或者detach，很方便！

● 使用thread再配合mutex的std::unique\_lock和std::lock\_guard使用，很方便！

● 使用thread再配合条件变量使用，很方便！

● 使用std::this\_thread::sleep\_for(xxx)休眠某段时间，很方便！

使用sleep(xxx)永远解决不了任何时序相关的bug，一定要使用条件变量来保证时序

不要迷信多线程，我们要明确知道，为什么要使用多线程，是为了更高的性能？还是为了不阻塞当前线程？还是有其他考虑？想清楚利弊，最好能综合做出评估后再决定。

最好的同步就是没有同步：尽可能使用更合理的方式设计线程，让所有的线程在使用共享数据时只读不写，或者只写入其他线程不会读取的部分，或者确保数据的所有权是单线程模式，同一时刻只会有一个线程在访问这块数据，那么多线程编码就会简单很多，不会有任何数据竞争，也不会出现死锁等问题。

先考虑原子类型再考虑锁：通过原子类型或原子操作更方便编写没有数据竞争和死锁的代码，因为他们能自动处理同步问题。如果不能使用原子类型或原子操作，那再考虑使用互斥锁来保护临界区。（看到过有大佬不推荐原子操作的，但是没说为啥，这是有什么顾虑吗？大家可以留言聊一聊。）

先确保解决了同步问题，再考虑优化：典型的就是普通互斥锁和读写锁的问题，很多人上来就使用读写锁，追求更高的性能，除非读操作比写操作频繁的多，否则读写锁并不会提高多少性能，我看见过很多使用读写锁导致出现同步问题的案例。所以，开始写代码时还是消停的使用普通锁吧，真正需要优化时再考虑使用其他手段。

使用RAII锁对象：使用lock\_guard、unique\_lock、shared\_lock或scoped\_lock等RAII类来管理锁，这样可以确保一定会释放锁。降低出现死锁的风险，但我们也要了解，如果真的出现了死锁，我们要如何定位？再出个思考题：我们都知道加锁的顺序不一致可能会导致死锁，如果释放锁的顺序不一致会导致死锁吗？

尽快释放锁：当需要通过锁保护共享数据时，务必尽快释放锁，尽可能缩小锁控制的粒度，明确哪些数据需要加锁，哪些根本就不需要，不要无脑加锁。因为当一个线程持有一个锁时，会使得其他线程阻塞等待这个锁，这可能会降低程序的性能。

使用线程池：动态频繁的创建和销毁大量的线程会导致性能下降。这种情况下，最好使用线程池来重用已有的线程，我之前写过如何撸一个线程池的文章，大家可以去看看。

做好日志记录：使用多线程程序很容易出现各种问题，而且问题还不稳定复现，复现的时机多数时候还不一样，一定要做好日志记录，确保出现问题时有据可查，可快速分析出问题所在。

1. 进程fork后父子进程什么相同什么不同

fork时子进程获得父进程数据空间、堆和栈的复制，所以变量的地址（当然是虚拟地址）也是一样的。

首先，子进程是完全复制父进程的，所以num的地址是一样的，可是子进程复制的是虚拟地址空间，而非物理空间。如果，子进程和父进程对变量只读，也就是说变量不会被改变，这时候，变量表现为共享的,此时物理空间只有一份。如果说父进程或者子进程需要改变变量，那么进程将会对物理内存进行复制，这个时候变量是独立的，也就是说，物理内存中存在两份空间。

写时复制。fork()之后，kernel把父进程中所有的内存页的权限都设为read-only，然后子进程的地址空间指向父进程。当父子进程都只读内存时，相安无事。当其中某个进程写内存时，CPU硬件检测到内存页是read-only的，于是触发页异常中断（page-fault），陷入kernel的一个中断例程。中断例程中，kernel就会把触发的异常的页复制一份，于是父子进程各自持有独立的一份。

延伸：子进程和父进程的文件描述符

**系统文件表位于系统空间中，不会被fork()复制，但是系统文件表中的条目会保存指向它的文件描述符表的计数，fork()时需要对这个计数进行维护， 以体现子进程对应的新的文件描述符表也指向它。程序关闭文件时，也是将系统文件表条目内部的计数减一，当计数值减为0时，才将其删除。**

1. 外中断和异常有什么区别？

外中断是指由 CPU 执行指令以外的事件引起，如 I/O 完成中断，表示设备输入/输出处理已经完成，处理器能够发送下一个输入/输出请求。此外还有时钟中断、控制台中断等。

而异常时由 CPU 执行指令的内部事件引起，如非法操作码、地址越界、算术溢出等。

1. 死锁，及原因，及如何避免死锁(解决方法)

根据操作系统中的定义：死锁是指一组进程中的各个进程均有不会释放的资源，但因互相申请被其他进程所占用不会释放的资源而处于的一种永久等待状态。

多线程以及多进程改善了系统资源的利用率并提高了系统的处理能力。然而，并发执行也带来了新的问题——死锁。所谓死锁是指多个线程/进程因竞争资源而形成的一种僵局（互相等待），若无外力作用，这些进程将无法向前推进

死锁产生的原因：

1. **系统资源的竞争**

通常系统中拥有的不可剥夺的资源，其数量不足以满足多个进程运行的需要，使得进程在运行过程中，会因争夺资源而陷入僵局，如磁带机、打印机等。对不可剥夺的资源的竞争 才可能产生死锁，对可剥夺的竞争不会引起死锁的

1. **进程推进顺序非法**

进程在运行过程中，请求和释放资源的顺序不当，也同样会导致死锁。例如，并发进程P1、P2分别保持了资源R1、R2，而进程P1申请了资源R2，进程R2申请了资源R1，两者都会因为所需资源被占用而阻塞。

1. **信号量使用不当造成的死锁**

进程间彼此相互等待对方发来的消息，结果会使得这些进程无法继续向前推进。例如，进程A等待进程B发的消息，进程B又在等待进程A发的消息，可以看出进程A和B不是因为竞争同一资源，而实在等待对方的资源导致死锁。

产生死锁的四个必要条件：

1. 互斥条件：一个资源每次只能被一个进程使用
2. 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放
3. 不剥夺条件：进程已获得的资源，在未使用完之前，不能强行剥夺
4. 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系

如何避免死锁：

1. **加锁顺序**(线程按照一定的顺序加锁)

按照顺序加锁是一种有效的死锁预防机制。但是，这种方式需要你事先知道所有可能会用到的锁(译者注：并对这些锁做适当的排序)，但总有些时候是无法预知的。

1. **加锁时限(线程尝试获取锁的时候加上一定的时限，超过时限则放弃对该锁的请求，并释放自己占有的锁)**

另外一个可以避免死锁的方法是在尝试获取锁的时候加一个超时时间，这也就意味着在尝试获取锁的过程中若超过了这个时限该线程则放弃对该锁请求。若一个线程没有在给定的时限内成功获得所有需要的锁，则会进行回退并释放所有已经获得的锁，然后等待一段随机的时间再重试。这段随机的等待时间让其它线程有机会尝试获取相同的这些锁，并且让该应用在没有获得锁的时候可以继续运行(加锁超时后可以先继续运行干点其它事情，再回头来重复之前加锁的逻辑)。

1. **死锁检测**

死锁检测是一个更好的死锁预防机制，它主要是针对那些不可能实现按序加锁并且锁超时也不可行的场景。

每当一个线程获得了锁，会在线程和锁相关的数据结构中（map、graph等等）将其记下。除此之外，每当有线程请求锁，也需要记录在这个数据结构中。

当一个线程请求锁失败时，这个线程可以遍历锁的关系图看看是否有死锁发生。例如，线程A请求锁7，但是锁7这个时候被线程B持有，这时线程A就可以检查一下线程B是否已经请求了线程A当前所持有的锁。如果线程B确实有这样的请求，那么就是发生了死锁（线程A拥有锁1，请求锁7；线程B拥有锁7，请求锁1

当然，死锁一般要比两个线程互相持有对方的锁这种情况要复杂的多。线程A等待线程B，线程B等待线程C，线程C等待线程D，线程D又在等待线程A。线程A为了检测死锁，它需要递进地检测所有被B请求的锁。从线程B所请求的锁开始，线程A找到了线程C，然后又找到了线程D，发现线程D请求的锁被线程A自己持有着。这是它就知道发生了死锁。

那么当检测出死锁时，这些线程该做些什么呢？

一个可行的做法是释放所有锁，回退，并且等待一段随机的时间后重试。这个和简单的加锁超时类似，不一样的是只有死锁已经发生了才回退，而不会是因为加锁的请求超时了。虽然有回退和等待，但是如果有大量的线程竞争同一批锁，它们还是会重复地死锁（原因同超时类似，不能从根本上减轻竞争）。

一个更好的方案是给这些线程设置优先级，让一个（或几个）线程回退，剩下的线程就像没发生死锁一样继续保持着它们需要的锁。如果赋予这些线程的优先级是固定不变的，同一批线程总是会拥有更高的优先级。为避免这个问题，可以在死锁发生的时候设置随机的优先级。锁的类别

从数据库系统的角度来看：分为独占锁（排它锁）、共享锁和更新锁

1. 并行编程的锁难题(\*)

锁的使用会主要会引发两类难题：一类是诸如死锁、活锁等引起的多线程Bug；另一类是由锁竞争引起的性能瓶颈

## 用锁来防止数据竞跑

## 死锁和活锁

### 锁竞争的常用解决办法

避免使用锁

为了提高程序的并行性，最好的办法自然是不使用锁。从设计角度上来讲，锁的使用无非是为了保护共享资源。如果我们可以避免使用共享资源的话那自然就避免了锁竞争造成的性能损失。幸运的是，在很多情况下我们都可以通过资源复制的方法让每个线程都拥有一份该资源的副本，从而避免资源的共享。如果有需要的话，我们也可以让每个线程先访问自己的资源副本，只在程序的后讲各个线程的资源副本合并成一个共享资源。例如，如果我们需要在多线程程序中使用计数器，那么我们可以让每个线程先维护一个自己的计数器，只在程序的最后将各个计数器两两归并（类比二叉树），从而最大程度提高并行度，减少锁竞争。

使用读写锁

如果对共享资源的访问多数为读操作，少数为写操作，而且写操作的时间非常短，我们就可以考虑使用读写锁来减少锁竞争。读写锁的基本原则是同一时刻多个读线程可以同时拥有读者锁并进行读操作；另一方面，同一时刻只有一个写进程可以拥有写者锁并进行写操作。读者锁和写者锁各自维护一份等待队列。当拥有写者锁的写进程释放写者锁时，所有正处于读者锁等待队列里的读线程全部被唤醒并被授予读者锁以进行读操作；当这些读线程完成读操作并释放读者锁时，写者锁中的第一个写进程被唤醒并被授予写者锁以进行写操作，如此反复。

换句话说，多个读线程和一个写线程将交替拥有读写锁以完成相应操作。这里需要额外补充的一点是锁的公平调度问题。例如，如果在写者锁等待队列中有一个或多个写线程正在等待获得写者锁时，新加入的读线程会被放入读者锁的等待队列。这是因为，尽管这个新加入的读线程能与正在进行读操作的那些读线程并发读取共享资源，但是也不能赋予他们读权限，这样就防止了写线程被新到来的读线程无休止的阻塞。

需要注意的是，并不是所有的场合读写锁都具备更好的性能，大家应该根据Profling的测试结果来判断使用读写锁是否能真的提高性能，特别是要注意写操作虽然很少但很耗时的情况。

**保护数据而不是操作**

在实际程序中，有不少程序员在使用锁时图方便而把一些不必要的操作放在临界区中。例如，如果需要对一个共享数据结构进行删除和销毁操作，我们只需要把删除操作放在临界区中即可，资源销毁操作完全可以在临界区之外单独进行，以此增加并行度。正是因为临界区的执行时间大大影响了并行程序的整体性能，我们必须**尽量少在临界区中做耗时的操作**，例如函数调用，数据查询，I/O操作等。简而言之，我们需要保护的只是那些共享资源，而不是对这些共享资源的操作，尽可能的把对共享资源的操作放到临界区之外执行有助于减少锁竞争带来的性能损失。

**尽量使用轻量级的原子操作**

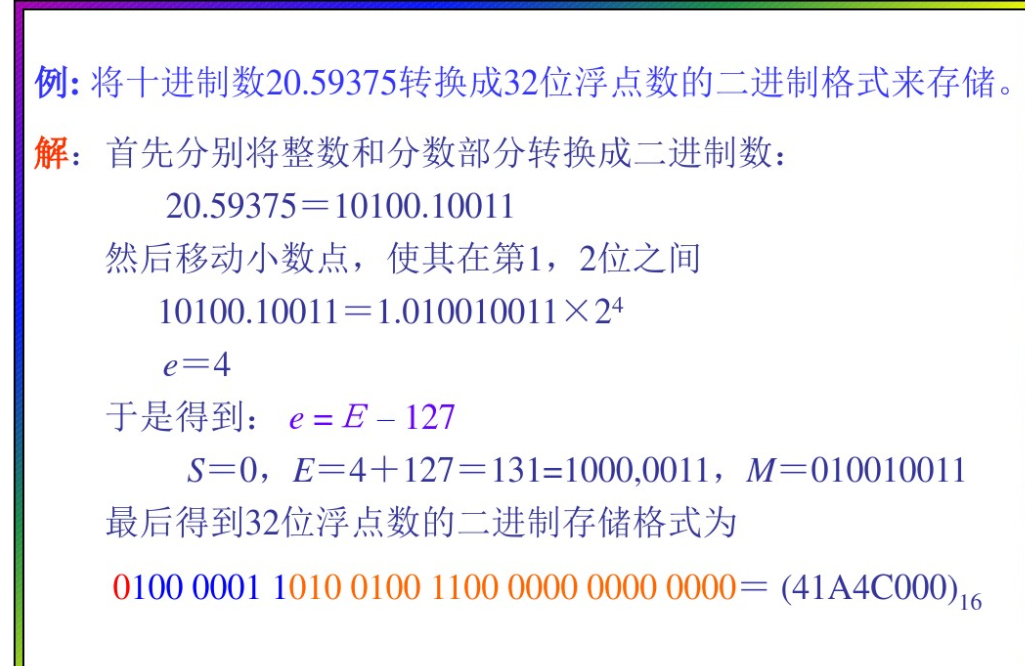
在例3中，我们使用了mutex锁来保护counter++操作。实际上，counter++操作完全可以使用更轻量级的原子操作来实现，根本不需要使用mutex锁这样相对较昂贵的机制来实现。在今年程序员第四期的《volatile与多线程的那些事儿》中我们就有对Java和C/C++中的原子操作做过相应的介绍。

**粗粒度锁与细粒度锁**

为了减少串行部分的执行时间，我们可以通过把单个锁拆成多个锁的办法来较小临界区的执行时间，从而降低锁竞争的性能损耗，即把“粗粒度锁”转换成“细粒度锁”。但是，细粒度锁并不一定更好。这是因为粗粒度锁编程简单，不易出现死锁等Bug，而细粒度锁编程复杂，容易出错；而且锁的使用是有开销的

**使用无锁算法、数据结构**

1. 知道浮点数吗，在计算机中怎么表示的，说下原理



1. 线程同步方式

线程之间通信的两个基本问题是互斥和同步。

https://www.cnblogs.com/yhlboke-1992/p/9315263.html

1. cache一致性

[高速缓冲存储器](https://baike.baidu.com/item/高速缓冲存储器/9027270)一致性（Cache coherence），也称[缓存一致性](https://baike.baidu.com/item/缓存一致性/15814005" \t "_blank)，高速缓存间一致性。是指在采用层次结构存储系统的计算机系统中，保证高速缓冲存储器中数据与主存储器中数据相同机制。在一个系统中，当许多不同的设备共享一个共同存储器资源，在高速缓存中的数据不一致，就会产生问题。这个问题在有数个[CPU](https://baike.baidu.com/item/CPU/120556)的[多处理机系统](https://baike.baidu.com/item/多处理机系统/10015327" \t "_blank)中特别容易出现。

MESI协议

1. 原子操作（重点）

"原子操作(atomic operation)是不需要synchronized"，这是多线程编程的老生常谈了。所谓原子操作是指不会被[线程调度](https://baike.baidu.com/item/线程调度/10226112" \t "_blank)机制打断的操作；这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch （切 [1]  换到另一个线程）。

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。

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。

在C++11之前，C++标准中并没有对原子操作进行规定。vs和gcc编译器提供了原子操作的api。

## C++11提供的原子操作

C++11中在<atomic>中定义了atomic模板类，atomic的模板参数类型可以为int、long、bool等等，C++中称为trivially copyable type。atomic\_int、atomic\_long为atomic模板实例化后的宏定义。atomic具体的原子操作函数可以参考http://www.cplusplus.com/reference/atomic/atomic/?kw=atomic。

原子操作的效率

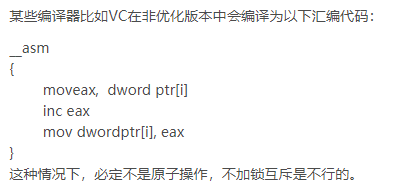
1. 不加锁也不适用原子变量的程序

## 什么时候使用原子操作

在多线程并发的条件下，所有不是原子性的操作需要保证原子性时，都需要进行原子操作处理。



1. i++是原子的吗？为什么不是原子的？会出现什么情况？



"原子操作(atomic operation)是不需要synchronized"，  
答案是否定的，i++和++i都不具有原子性。  
i++：先赋值再自加。  
++i：先自加再赋值。  
i++和++i的线程安全分为两种情况：  
1、如果i是局部变量（在方法里定义的），那么是线程安全的。因为局部变量是线程私有的，别的线程访问不到，其实也可以说没有线程安不安全之说，因为别的线程对他造不成影响。

2、如果i是全局变量，则同一进程的不同线程都可能访问到该变量，因而是线程不安全的，  
会产生脏读。影响后面数据的读取

1. 操作系统堆和栈的区别，局部变量在哪个区（经典问题了，但是答的不是很好-。

堆和栈的区别：

内存分配方面：

堆：一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。注意它与数据结构中的堆是两回事，分配方式是类似于链表。可能用到的关键字：new、malloc、delete、free等

栈：由编译器自动释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构钟的栈

申请方式方面：

堆：需要程序自己申请，并指明大小。在c中malloc函数如p1=(char\*)malloc(10)；在C++中使用new运算符，但是注意p1、p2本身是在栈中。因为他们还是可以认为是局部变量

栈：由系统自动分配。例如：声明在函数中一个局部变量int b；系统自动在栈中为b开辟空间。

系统响应方面：

堆：操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样代码中的delete语句才能正确的释放本内存空间。另外由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

C/C++程序中，占用的内存分为几个部分：

栈区（stack）：由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作类似于数据结构中的栈。

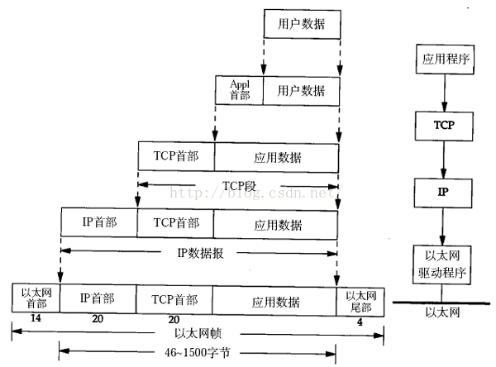
堆区（heap）：一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。

全局区（静态区）（static）：全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域-程序结束后由系统释放

文字常量区：常量字符就是存放在这里的。程序结束后由系统释放

程序代码区：存放函数体的二进制代码

1. TCP和UDP的包头多大记得吗，长什么样 （大概描述了一下里面的字段



应用程序中用到的data的长度最大是多少，直接取决于底层的限制。

1. 链路层：由以太网的物理特性决定了数据帧的长度（46 + 18）-（1500 + 18），其中的18是数据帧的头和尾，也就是说数据帧的内容最大为1500，即MTU（最大传输单元）为1500
2. 网络层：因为**IP包的首部占用要用20字节**，所以这的MTU为1500 – 20 = 1480；
3. 传输层：对于**UDP包的首部要占用8字节**，所以这的MTU为1480 – 8 = 1472；
4. 应用层：你的data最大长度为1472.当我们的UDP包中的数据多于MTU(1472)时，发送方的IP层需要分片fragmentation进行传输，而在接收方IP层组需要进行数据报重组，由于UDP是不可靠的传输协议，如果分片丢失导致重组失败，将导致UDP数据包被丢弃。

不过目前大多数的路由设备的MTU都为1500。

UDP和TCP协议利用端口号实现多项应用同时发送和接收数据。

UDP包的大小就应该是**1500 – IP包(20) - UDP包头(8) = 1472（bytes）**

TCP包的大小就应该是**1500 – IP包(20) – TCP包头(20) = 1460（bytes）**

如果我们定义的TCP和UDP包没有超过范围，那么我们的包在IP层就不用分包了，这样传输过程中就避免了在IP层组包发生的错误；如果超过范围，既IP数据报大于1500字节，发送方IP层就需要将数据包分成若干片，而接收方IP层就需要进行数据报的重组。更严重的是，如果使用UDP协议，当IP层组包发生错误，那么包就会被丢弃。接收方无法重组数据报，将导致丢弃整个IP数据报。UDP不保证可靠传输；但是TCP发生组包错误时，该包会被重传，保证可靠传输。





1. 什么是非阻塞IO
2. 阻塞I/O模型

最传统的一种IO模型，即在读写数据过程中发生阻塞现象。

当用户线程发初I/O请求之后，内核会区查看数据是否就绪，如果没有就绪就会等待数据就绪，而用户线程就会处于阻塞状态，用户线程交出CPU。当数据就绪之后，内核会将数据拷贝到用户线程，并返回结果给用户线程，用户线程才解除block状态。

典型的阻塞IO模型的例子为：



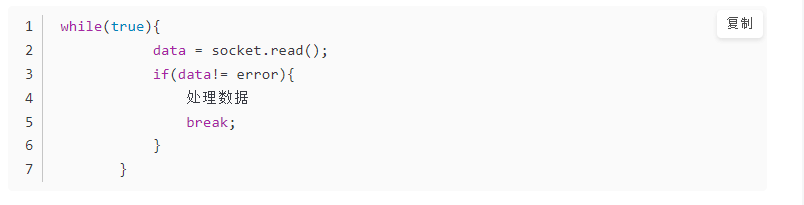
如果数据没有就绪，就会一直阻塞在read方法

1. 非阻塞I/O模型

当用户线程发起一个read操作后，并不需要等待，而是马上就得到了一个结果。如果结果是一个error时，它就知道数据还没有准备好，于是它可以再次发送read操作。一旦内核中的数据准备好了，并且又可以再次收到了用户线程的请求，那么它马上就将数据拷贝到了用户线程，然后返回。

所以事实上，在非阻塞IO模型中，用户线程需要不断地询问内核数据是否就绪，也就说**非阻塞IO不会交出CPU,而会一直占用CPU。**

典型的非阻塞IO模型一般如下：



但是对于非阻塞IO就有一个非常严重的问题，在while循环中需要不断地询问内核数据是否就绪，这样会导致CPU占用率非常高，因此一般情况很少使用while循环方式来读取数据。

1. 多路复用IO模型

在多路复用IO模型中，会有一个线程不断区轮询多个socket状态，只有当socket真正有读写事件时，才真正调用实际的IO读写操作。因为在多路复用IO模型中，只需要使用一个线程就可以管理多个socket，系统不需要简历新的进程或者线程，也不必维护这些线程和进程，并且只有在真正有socket读写时间进行时，才会使用IO资源，所以它大大减少了资源占用。

另外多路复用IO为何比非阻塞IO模型的效率高是因为在非阻塞IO中，不断地询问socket状态是通过用户线程去进行的，而在多路复用IO中，轮询每个socket状态是**内核在进行**的，这个效率要比用户线程要高的多。

　　不过要注意的是，多路复用IO模型是通过轮询的方式来检测是否有事件到达，并且对到达的事件逐一进行响应。因此对于多路复用IO模型来说，一旦事件响应体很大，那么就会导致后续的事件迟迟得不到处理，并且会影响新的事件轮询。

1. 信号驱动IO模型

在信号驱动IO模型中，当用户线程发起一个IO请求操作，会给对应socket注册一个信号函数，然后用户线程会继续执行，当内核数据就绪时会发送一个信号给用户线程，用户线程接收到信号之后，便在信号函数中调用IO读写操作来进行实际IO请求操作

1. 异步IO模型

因为无论是多路复用IO还是信号驱动模型，IO操作的第2个阶段都会引起用户线程阻塞，也就是内核进行数据拷贝的过程都会让用户线程阻塞。

1. 布隆过滤器

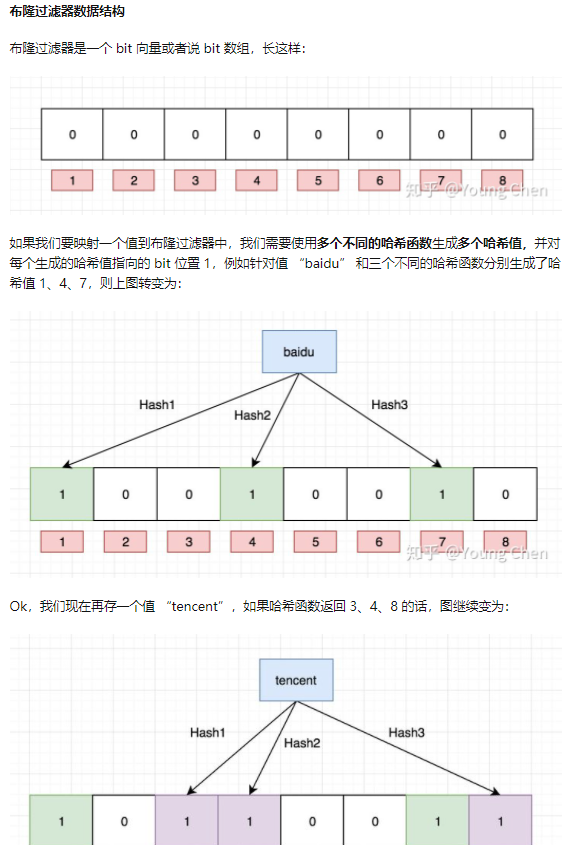
如果想要判断一个元素是不是在一个集合里，一般想到的是将所有元素保存起来，然后通过比较确定。链表，树等等数据结构都是这种思路. 但是随着集合中元素的增加，我们需要的存储空间越来越大，检索速度也越来越慢(O(n),O(logn))。不过世界上还有一种叫作散列表（又叫哈希表，Hash table）的数据结构。它可以通过一个Hash函数将一个元素映射成一个位阵列（Bit array）中的一个点。这样一来，我们只要看看这个点是不是1就可以知道集合中有没有它了。这就是布隆过滤器的基本思想。

Data structures are nothing different. They are like the bookshelves of your application where you can organize your data. Different data structures will give you different facility and benefits. To properly use the power and accessibility of the data structures you need to know the trade-offs of using one.

大意是不同的数据结构有不同的适用场景和优缺点，你需要仔细权衡自己的需求之后妥善适用它们，布隆过滤器就是践行这句话的代表。

本质上布隆过滤器是一种数据结构，比较巧妙的概率型数据结构（probabilistic data structure），特点是高效地插入和查询，可以用来告诉你 “某样东西一定不存在或者可能存在”。

相比于传统的 List、Set、Map 等数据结构，它更高效、占用空间更少，但是缺点是其返回的结果是概率性的，而不是确切的。

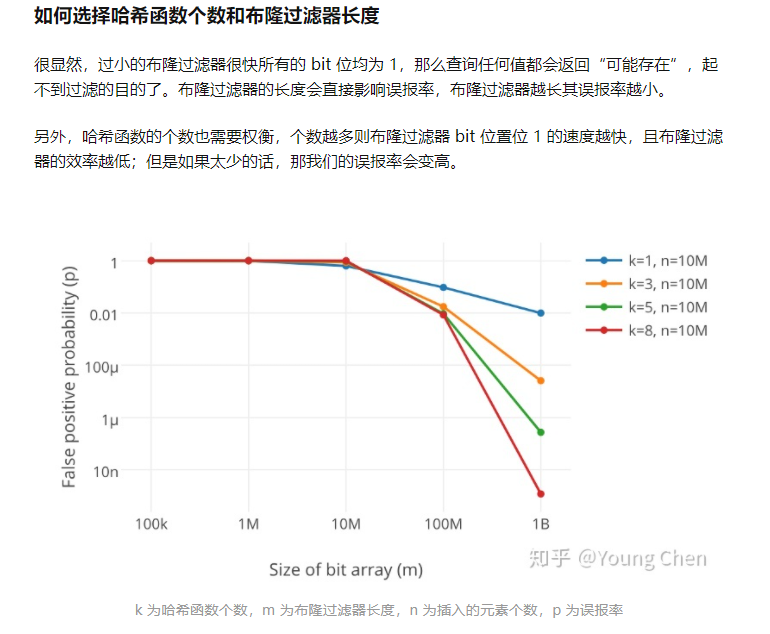


值得注意的是，4 这个 bit 位由于两个值的哈希函数都返回了这个 bit 位，因此它被覆盖了。现在我们如果想查询 “dianping” 这个值是否存在，哈希函数返回了 1、5、8三个值，结果我们发现 5 这个 bit 位上的值为 0，说明没有任何一个值映射到这个 bit 位上，因此我们可以很确定地说 “dianping” 这个值不存在。而当我们需要查询 “baidu” 这个值是否存在的话，那么哈希函数必然会返回 1、4、7，然后我们检查发现这三个 bit 位上的值均为 1，那么我们可以说 “baidu” 存在了么？答案是不可以，只能是 “baidu” 这个值可能存在。

这是为什么呢？答案跟简单，因为随着增加的值越来越多，被置为 1 的 bit 位也会越来越多，这样某个值 “taobao” 即使没有被存储过，但是万一哈希函数返回的三个 bit 位都被其他值置位了 1 ，那么程序还是会判断 “taobao” 这个值存在。

## 支持删除么

感谢评论区提醒，传统的布隆过滤器并不支持删除操作。但是名为 Counting Bloom filter 的变种可以用来测试元素计数个数是否绝对小于某个阈值，它支持元素删除。



## 最佳实践

常见的适用常见有，利用布隆过滤器减少磁盘 IO 或者网络请求，因为一旦一个值必定不存在的话，我们可以不用进行后续昂贵的查询请求。

另外，既然你使用布隆过滤器来加速查找和判断是否存在，那么性能很低的哈希函数不是个好选择，推荐 MurmurHash、Fnv 这些。

1. 什么是按需分页、虚拟内存的实现方式、什么是虚拟内存

按需分页:

在操作系统中，进程是以页为单位加载到内存中的，按需分页是一种虚拟内存的管理方式。在使用请求分页的系统中，只有在尝试访问页面所在的磁盘并且该页面尚未在内存中时，也就发生了缺页异常，操作系统才会将磁盘页面复制到内存中。

虚拟内存中，允许将一个作业分多次调入内存。釆用连续分配方式时，会使相当一部分内存空间都处于暂时或永久的空闲状态，造成内存资源的严重浪费，而且也无法从逻辑上扩大内存容量。因此，虚拟内存的实需要建立在离散分配的内存管理方式的基础上。虚拟内存的实现有以下三种方式：

* 请求分页存储管理。
* 请求分段存储管理。
* 请求段页式存储管理。

不管哪种方式，都需要有一定的硬件支持。一般需要的支持有以下几个方面：

* 一定容量的内存和外存。
* 页表机制（或段表机制），作为主要的数据结构。
* 中断机构，当用户程序要访问的部分尚未调入内存，则产生中断。
* 地址变换机构，逻辑地址到物理地址的变换。

1. 虚拟内存与物理内存

操作系统有虚拟内存与物理内存的概念。在之前，还没虚拟内存概念时候，程序寻址都是物理地址。程序能寻址的范围是有限的，这取决于CPU的地址线条数。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。

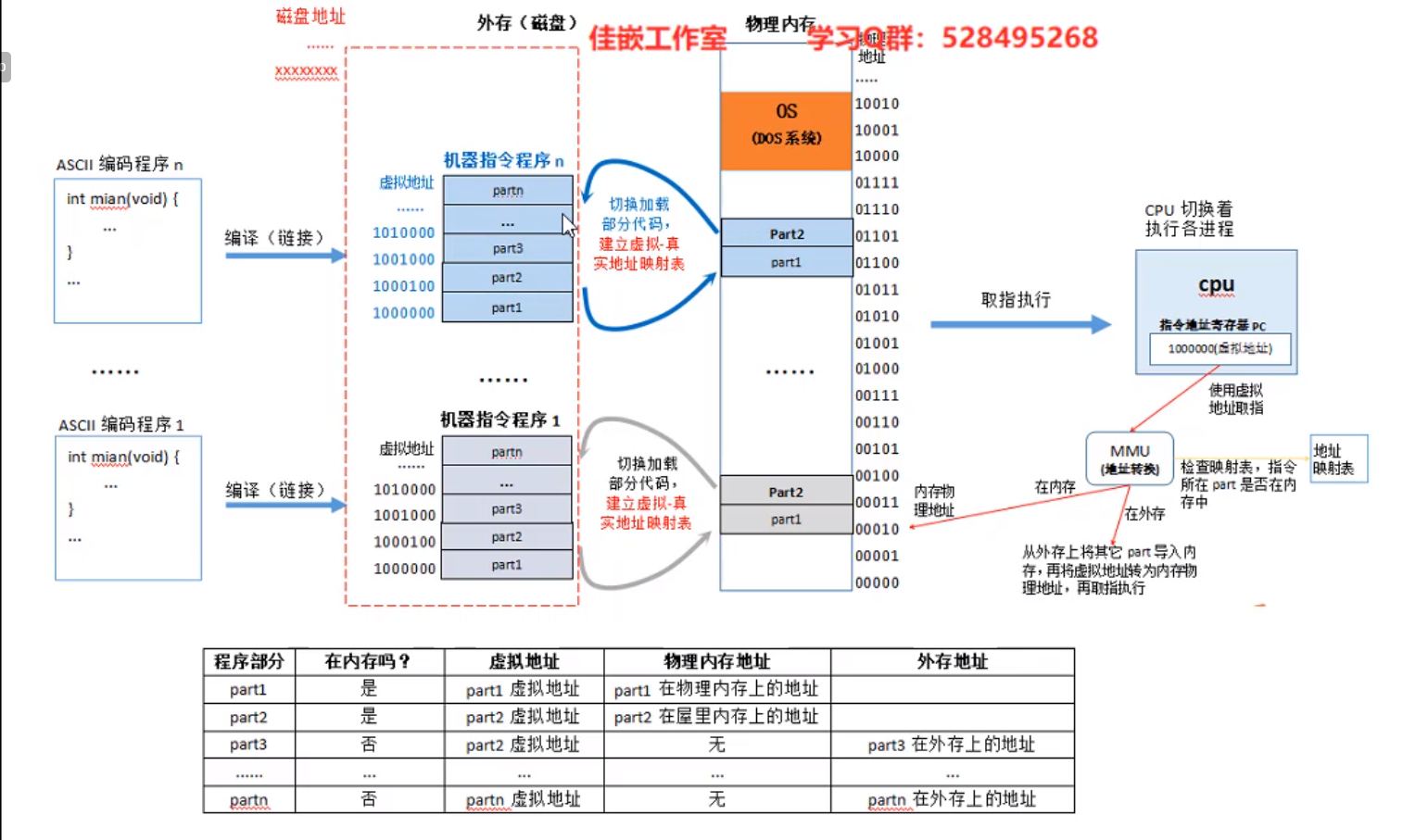
一个进程运行时都会得到4G的虚拟内存。这个虚拟内存你可以认为，每个进程都认为自己拥有4G的空间，这只是每个进程认为的，但是实际上，在虚拟内存对应的物理内存上，可能只对应的一点点的物理内存，实际用了多少内存，就会对应多少物理内存。

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。（虚拟内存->虚拟地址->页面置换算法）
* 每块就是一个机器指令程序

<https://blog.csdn.net/lvyibin890/article/details/82217193>



CPU工作过程：

CPU拿着虚拟地址，查找地址映射表，查看对应虚拟地址是否在内存当中，若在则执行操作1，否则执行操作2

操作1 通过映射表获取虚拟地址所对应的物理地址，再通过其物理地址进行寻址

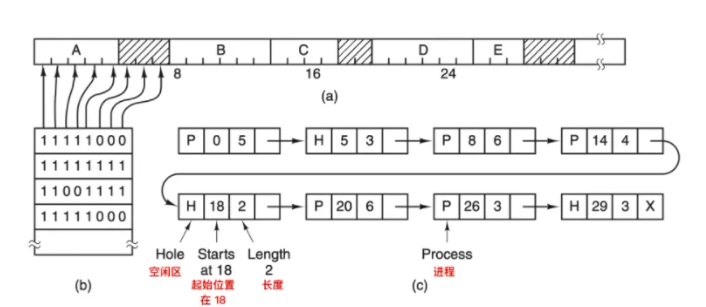
操作2 从外存上将其他part导入内存，再将虚拟地址转换为物理地址，再取址执行

1. 空闲内存管理的方式

操作系统在动态分配内存时（malloc，new），需要对空间内存进行管理。一般采用了两种方式：位图和空闲链表。

#### 使用位图进行管理

使用位图方法时，内存可能被划分为小到几个字或大到几千字节的分配单元。每个分配单元对应于位图中的一位，0 表示空闲， 1 表示占用（或者相反）。一块内存区域和其对应的位图如下

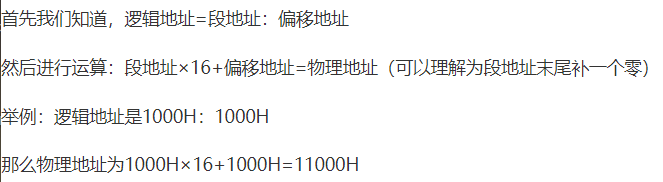


å图 a 表示一段有 5 个进程和 3 个空闲区的内存，刻度为内存分配单元，阴影区表示空闲（在位图中用 0 表示）；图 b 表示对应的位图；图 c 表示用链表表示同样的信息

分配单元的大小是一个重要的设计因素，分配单位越小，位图越大。然而，即使只有 4 字节的分配单元，32 位的内存也仅仅只需要位图中的 1 位。32n 位的内存需要 n 位的位图，所以**1 个位图只占用了 1/32 的内存**。如果选择更大的内存单元，位图应该要更小。如果进程的大小不是分配单元的整数倍，那么在最后一个分配单元中会有大量的内存被浪费。

位图提供了一种简单的方法在固定大小的内存中跟踪内存的使用情况，因为**位图的大小取决于内存和分配单元的大小**。这种方法有一个问题，当决定为把具有 k 个分配单元的进程放入内存时，内容管理器(memory manager) 必须搜索位图，在位图中找出能够运行 k 个连续 0 位的串。在位图中找出制定长度的连续 0 串是一个很耗时的操作，这是位图的缺点。（可以简单理解为在杂乱无章的数组中，找出具有一大长串空闲的数组单元）

1. 怎么通过逻辑地址找到物理地址



段地址、偏移地址来计算。

1. 进程和进程表

操作系统为了跟踪每个进程的活动状态，维护了一个进程表。在进程表的内部，列出了每个进程的状态以及每个进程使用的资源等。

1. 上下文切换

对于单核单线程 CPU 而言，在某一时刻只能执行一条 CPU 指令。上下文切换 (Context Switch) 是一种 **将 CPU 资源从一个进程分配给另一个进程的机制**。从用户角度看，计算机能够并行运行多个进程，这恰恰是操作系统通过快速上下文切换造成的结果。在切换的过程中，操作系统需要先存储当前进程的状态 (包括内存空间的指针，当前执行完的指令等等)，再读入下一个进程的状态，然后执行此进程。

1. 进程间通信的方式（本质区别）

IPC：

共享内存、信号量、消息队列、管道、无名管道

* 管道（PIPE）
  + 优点：简单方便
  + 缺点：
  + 优点：可以实现任意关系的进程间的通信
  + 缺点：
  + 有名管道：一种半双工的通信方式，它允许无亲缘关系进程间的通信
  + 无名管道：一种半双工的通信方式，只能在具有亲缘关系的进程间使用（父子进程）
  + 局限于单向通信
  + 只能创建在它的进程以及其有亲缘关系的进程之间
  + 缓冲区有限
  + 长期存于系统中，使用不当容易出错
  + 缓冲区有限
* 信号量（Semaphore）：一个计数器，可以用来控制多个线程对共享资源的访问
  + 优点：可以同步进程
  + 缺点：信号量有限
* 信号（Signal）：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
* 消息队列（Message Queue）：是消息的链表，存放在内核中并由消息队列标识符标识
  + 优点：可以实现任意进程间的通信，并通过系统调用函数来实现消息发送和接收之间的同步，无需考虑同步问题，方便
  + 缺点：信息的复制需要额外消耗 CPU 的时间，不适宜于信息量大或操作频繁的场合
* 共享内存（Shared Memory）：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
  + 优点：无须复制，快捷，信息量大
  + 缺点：
  + 通信是通过将共享空间缓冲区直接附加到进程的虚拟地址空间中来实现的，因此进程间的读写操作的同步问题
  + 利用内存缓冲区直接交换信息，内存的实体存在于计算机中，只能同一个计算机系统中的诸多进程共享，不方便网络通信
* 套接字（Socket）：可用于不同及其间的进程通信
  + 优点：
  + 缺点：需对传输的数据进行解析，转化成应用级的数据。
  + 传输数据为字节级，传输数据可自定义，数据量小效率高
  + 传输数据时间短，性能高
  + 适合于客户端和服务器端之间信息实时交互
  + 可以加密,数据安全性强

1. 线程间通信的方式

* 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着是否已经释放锁。
  + 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。
* 信号量机制(Semaphore)
  + 无名线程信号量
  + 命名线程信号量
* 信号机制(Signal)：类似进程间的信号处理
* 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。

| **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. 内核态用户态的区别

1．操作系统的两种CPU状态

内核态（kernel mode）：运行操作系统程序，操作硬件

用户态（user mode）：运行客户程序

2. 指令划分

特权指令：只能由操作系统使用、用户程序不能使用的命令。举例：启动I/O内存清零 修改程序状态字

非特权命令：用户程序可以使用的指令。举例：控制转移 算术运算 指数运算

3.特权级别

特权环：R0、R1、R2和R3

R0属于内核态 ，R3属于用户态

不同级别能够运行不同的指令集合

4.cpu状态之间的转换

用户态->内核态：唯一途径就是**通过中断、异常**、陷入机制（访管命令）

内核态->用户态：设置程序状态字PSW

5.内核态和用户态的区别

* 内核态与用户态时操作系统的两种运行级别，当程序运行在3级特权上时，就可以称之为运行在用户态。因为这是最低特权级，是普通的用户进程运行的特权级，大部分用户直接面对的程序都是运行在用户态。
* 当程序运行在0级特权级上时，就可以称之为内核态
* 运行在用户态上的程序不能直接访问操作系统内核数据结构和程序。当我们在系统种执行一个程序时，大部分时运行在用户态下的，在其需要操作系统帮助完成某些它没有权力和能力完成的工作时就会切换到内核态（比如操作硬件）
* 这两种状态主要差别是：
  + 处于用户态执行时，进程能访问的内存空间和对象受到限制，其所处于占有的处理器是可以被抢占的
  + 处于内核态执行时，则能访问所有内存空间和对象，且所占有的处理器时不允许被抢占的

通常，一下三种情况会导致用户态到内核态的切换

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。比如前例中fork()实际上就是执行了一个创建新进程的系统调用**。

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。

一般用户态 -> 内核态的转换我们都称之为 trap 进内核，也被称之为 陷阱指令(trap instruction)。

用户程序通常调用库函数，由库函数再调用系统调用，因此有的库函数会使用户程序进入内核态（只要库函数中某处调用了系统调用），有的则不会。

* **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。

* **外围设备的中断、中断信号**

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。

**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**

那么为什么要有用户态和内核态呢？

这个主要是访问能力的限制的考量，计算机中有一些比较危险的操作，比如设置时钟、内存清理，这些都需要在内核态下完成，如果随意进行这些操作，那你的系统得崩溃多少次。

1. 线程知道哪些锁（及其运用场景）

线程之间的锁有：**互斥锁、条件锁、自旋锁、读写锁、递归锁**

互斥锁

互斥锁用于控制多个线程对他们之间共享资源互斥访问的一个信号量。也就是说是为了避免多个线程在某一时刻同时操作一个共享资源。例如线程池中的有多个空闲线程和一个任务队列。任何是一个线程都要使用互斥锁互斥访问任务队列，以避免多个线程同时访问任务队列以发生错乱。

条件锁

条件锁就是所谓的条件变量，某一个线程因为某个条件为满足时可以使用条件变量使改程序处于阻塞状态。一旦条件满足以“信号量”的方式唤醒一个因为该条件而被阻塞的线程。最为常见就是在线程池中，起初没有任务时任务队列为空，此时线程池中的线程因为“任务队列为空”这个条件处于阻塞状态。一旦有任务进来，就会以信号量的方式唤醒一个线程来处理这个任务。

自旋锁

假设我们有一个两个处理器core1和core2计算机，现在在这台计算机上运行的程序中有两个线程：T1和T2分别在处理器core1和core2上运行，两个线程之间共享着一个资源。

首先我们说明互斥锁的工作原理，互斥锁是是一种sleep-waiting的锁。假设线程T1获取互斥锁并且正在core1上运行时，此时线程T2也想要获取互斥锁（pthread\_mutex\_lock），但是由于T1正在使用互斥锁使得T2被阻塞。当T2处于阻塞状态时，T2被放入到等待队列中去，处理器core2会去处理其他任务而不必一直等待（忙等）。也就是说处理器不会因为线程阻塞而空闲着，它去处理其他事务去了。

而自旋锁就不同了，自旋锁是一种busy-waiting的锁。也就是说，如果T1正在使用自旋锁，而T2也去申请这个自旋锁，此时T2肯定得不到这个自旋锁。与互斥锁相反的是，此时运行T2的处理器core2会一直不断地循环检查锁是否可用（自旋锁请求），直到获取到这个自旋锁为止。

从“自旋锁”的名字也可以看出来，如果一个线程想要获取一个被使用的自旋锁，那么它会一致占用CPU请求这个自旋锁使得CPU不能去做其他的事情，直到获取这个锁为止，这就是“自旋”的含义。

当发生阻塞时，互斥锁可以让CPU去处理其他的任务；而自旋锁让CPU一直不断循环请求获取这个锁。通过两个含义的对比可以我们知道“自旋锁”是比较耗费CPU的。

读写锁

说到读写锁我们可以借助于“读者-写者”问题进行理解。首先我们简单说下“读者-写者”问题。

计算机中某些数据被多个进程共享，对数据库的操作有两种：一种是读操作，就是从数据库中读取数据不会修改数据库中内容；另一种就是写操作，写操作会修改数据库中存放的数据。因此可以得到我们允许在数据库上同时执行多个“读”操作，但是某一时刻只能在数据库上有一个“写”操作来更新数据。这就是一个简单的读者-写者模型。

1. 操作系统实现锁的方法
2. 锁机制，如果只显示上锁，怎么样让锁自动释放？
3. CAS原理分析

是一种无锁原子算法，映射到操作系统就是一条CPU的原子指令，其作用是让CPU先进行比较两个值是否相等，然后原子地更新某个位置的值，其实现方式是基于硬件平台的汇编指令，在intel的CPU中，使用的是cmpxchg指令，就是说CAS是靠硬件实现的，从而在硬件层面提升效率。

**是个原子性操作，对应CPU指令为cmpxchg。CAS的核心：**

**包含3个参数（V、E、N），V表示要更新变量的值，E表示期望的值，N表示新值，仅当V值等于E值得时，才将V值设为N值。如果V值和E值不同，则说明有其它线程完成更新，则当前线程则什么都不做，最后CAS返回当前V的真实值。**

执行过程是这样：它包含 3 个参数 CAS（V，E，N），V表示要更新变量的值，E表示预期值，N表示新值。仅当 V值等于E值时，才会将V的值设为N，如果V值和E值不同，则说明已经有其他线程完成更新，则当前线程则什么都不做，最后CAS 返回当前V的真实值。

**当多个线程同时使用CAS 操作一个变量时，最多只有一个会胜出，并成功更新，其余均会失败。失败的线程不会挂起，仅是被告知失败，并且允许再次尝试（自旋），当然也允许实现的线程放弃操作。基于这样的原理，CAS 操作即使没有锁，也可以避免其他线程对当前线程的干扰。**

与锁相比，使用CAS会使程序看起来更加复杂一些，但是使用无锁的方式完全没有锁竞争带来的线程间频繁调度的开销和阻塞，它对死锁问题天生免疫，因此他要比基于锁的方式拥有更优越的性能。

简单的说，CAS 需要你额外给出一个期望值，也就是你认为这个变量现在应该是什么样子的。如果变量不是你想象的那样，说明它已经被别人修改过了。你就需要重新读取，再次尝试修改就好了

乐观锁的概念中其实也阐述了它的具体实现细节：主要两个步骤：冲突检测和数据更新。其实现方式有一种比较典型的就是CAS。

硬件保证一个从语义上**看起来需要多次操作的行为只通过一条处理器指令**就能完成。这类指令称为原子指令，常用的有：1. 测试并设置（Tetst-and-Set）

2. 获取并增加（Fetch-and-Increment）3. 交换（Swap）

4. 比较并交 换（Compare-and-Swap）

5. 加载链接/条件存储（Load-Linked/Store-Conditional）

CPU实现原子指令的两种方式

**1. 通过总线锁定来保证原子性。**

总线锁定其实就是处理器使用了总线锁，所谓总线锁就是使用处理器**提供的一个 LOCK#** 信号，当一个处理器在总线上输出此信号时，其他处理器的请求将被阻塞住，那么该处理器可以独占共享内存。但是该方法成本太大。因此有了下面的方式。

**2、通过缓存锁定来保证原子性。**

所谓 缓存锁定 是指内存区域如果被缓存在处理器的缓存行中，并且在Lock 操作期间被锁定，那么当他执行锁操作写回到内存时，处理器不在总线上声言 LOCK# 信号，而是修改内部的内存地址，并允许他的缓存一致性机制来保证操作的原子性，因为缓存一致性机制会阻止同时修改两个以上处理器缓存的内存区域数据（这里和 volatile 的可见性原理相同），当其他处理器回写已被锁定的缓存行的数据时，会使缓存行无效。

注意：有两种情况下处理器不会使用缓存锁定：

当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行时，则处理器会调用总线锁定。

有些处理器不支持缓存锁定，对于 Intel 486 和 Pentium 处理器，就是锁定的内存区域在处理器的缓存行也会调用总线锁定

不难发现，cmpxchg这条汇编语言可以直接操作内存进行数据交换，实现CAS最终目的。

然而在多核处理器中，**需要遵循缓存一致性协议通知其他处理器更新自己的缓存**。

Lock在这里的作用：

在cmpxchg执行期间，锁住内存地址[edx]，其他处理器不能访问该内存，保证原子性。

（这个就是保证CAS原子性的关键所在）

写内存屏障，保证每个线程的本地空间与主存一致。

禁止cmpxchg与前后任何指令重排序，防止指令重排序。

CAS的缺点

CAS虽然高效地解决了原子操作，但是还是存在一些缺陷的，主要表现在三个方面：

1.**自旋时间太长**

如果CAS一直不成功呢？这种情况绝对有可能发生，如果自旋CAS长时间地不成功，则会给CPU带来非常大的开销。在JUC中有些地方就限制了CAS自旋的次数，例如BlockingQueue的SynchronousQueue。

2.**只能保证一个共享变量原子操作**

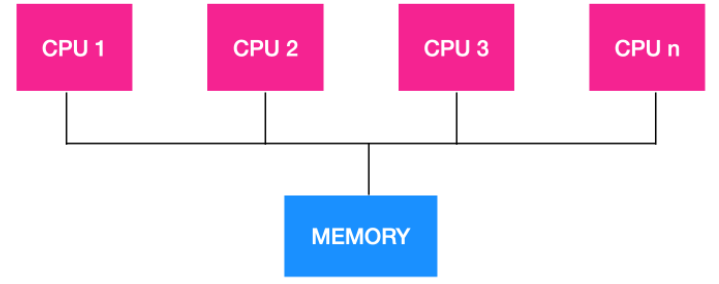
看了CAS的实现就知道这只能针对一个共享变量，如果是多个共享变量就只能使用锁了，当然如果你有办法把多个变量整成一个变量，利用CAS也不错。例如读写锁中state的高低位。（https://www.cnblogs.com/wait-pigblog/p/9350569.html）

3.**ABA问题**

CAS需要检查操作值有没有发生改变，如果没有发生改变则更新。但是存在这样一种情况：如果一个值原来是A，变成了B，然后又变成了A，那么在CAS检查的时候会发现没有改变，但是实质上它已经发生了改变，只是又回到了原来的值而已，这就是所谓的ABA问题。**对于ABA问题其解决方案是加上版本号，即在每个变量都加上一个版本号，每次改变时加1，即A —> B —> A，变成1A —> 2B —> 3A，采用AtomicStampedRdference类可以实现这个方案。**

1. 信号量的具体实现
2. 多处理系统的优势

随着处理器的不断增加，我们的计算机系统由单机系统变为了多处理系统，多处理系统的吞吐量比较高，多处理系统拥有多个并行的处理器，这些处理器共享时钟、内存、总线、外围设备等。



1. 进程间通信之间的区别

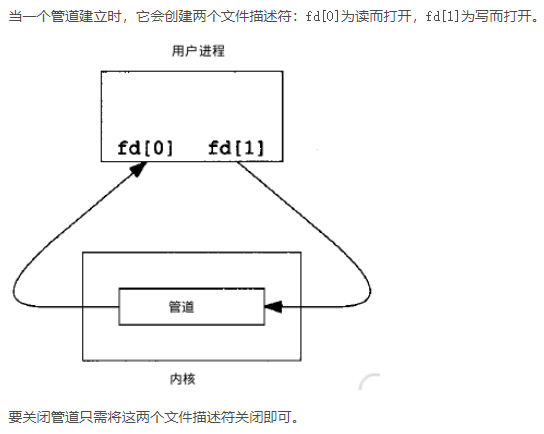
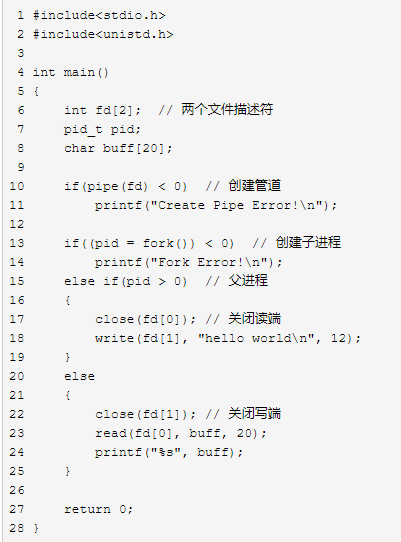
管道（Pipe）、命名管道（FIFO）、消息队列（Message Queue）、信号量（Semaphore）、共享内存（Shared Memory）；套接字（Socket）.

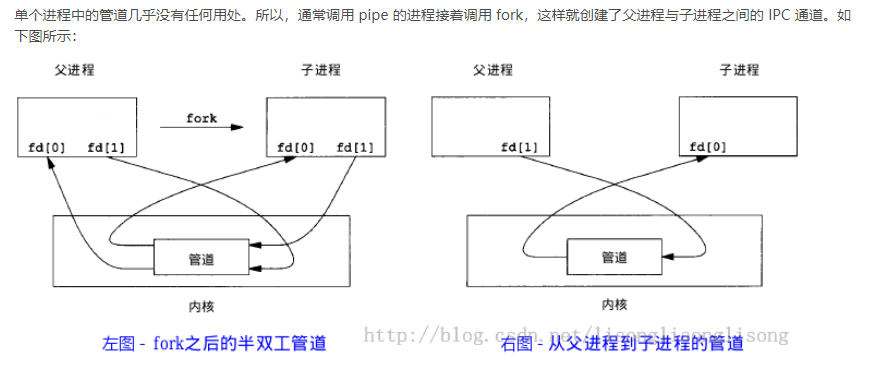
1. 如何杀死一个进程， kill实质做了什么事
2. 管道通信，管道有哪几类（五种进程间的通信方式）

管道通信常指无名管道，数据只能单向流动，而且只有具有亲缘关系的进程间使用

特点：

* 1. 半双工，具有固定的读写端
  2. 只能用于有关系的进程之间通信（也是父子进程或者兄弟进程）
  3. 它可以看成一种特殊的文件，但它不是普通的文件，并不属于系统，并且只存在于内存中

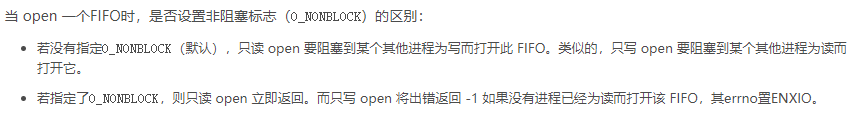


管道有两种：无名管道和命名管道。

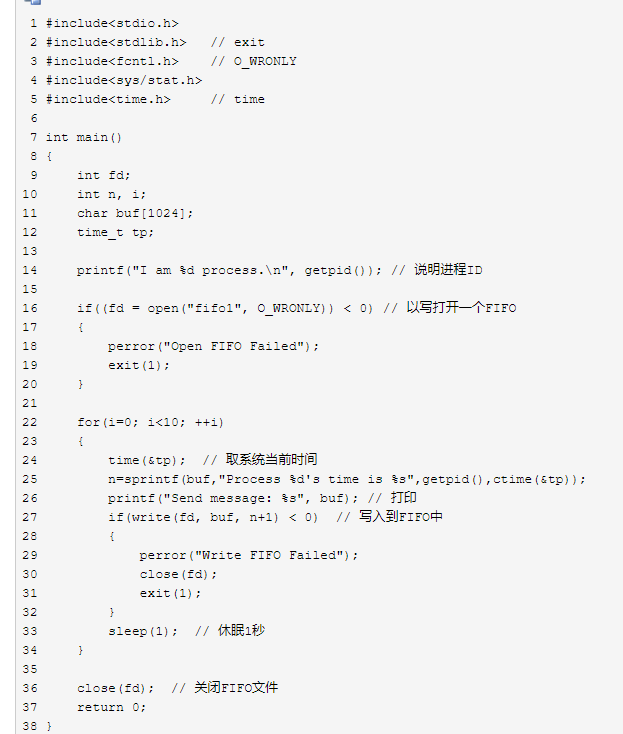
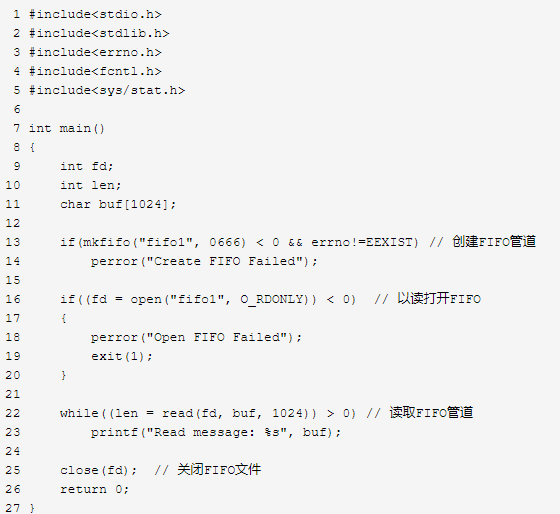
FIFO，也称为命名管道，它是一种文件类型。和无名管道的主要区别在于，命名管道有一个名字，命名管道的名字对应于一个磁盘索引节点，有了这个文件名，任何进程有相应的权限都可以对它进行访问。

特点：（半双工）

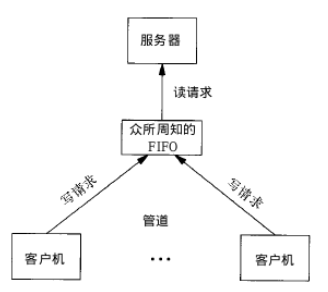
1. FIFO可以在无关进程之间交换数据，与无名管道不同
2. FIFO有路径名与之关联，它以一种特殊设备文件形式存在于文件系统中



例子：FIFO的通信方式类似于在进程中使用文件来传输数据，只不过FIFO类型文件同时具有管道的特性。在数据读出时，FIFO管道中同时清除数据，并且“先进先出”。

上述例子可以扩展成 客户进程—服务器进程 通信的实例，write\_fifo的作用类似于客户端，可以打开多个客户端向一个服务器发送请求信息，read\_fifo类似于服务器，它适时监控着FIFO的读端，当有数据时，读出并进行处理，但是有一个关键的问题是，每一个客户端必须预先知道服务器提供的FIFO接口，下图显示了这种安排



（都是通过内核缓冲区实现的数据传输）

1. 信号量

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。它通常作为一个锁机制，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同步手段。

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，

                      （1） P(sv)：如果sv的值大于零，就给它减1；如果它的值为零，就挂起该进程的执⾏ 。

                      （2） V(sv)：如果有其他进程因等待sv而被挂起，就让它恢复运⾏，如果没有进程因等待sv⽽挂起，就给它加1。

 对临界资源进行保护。

**信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据。**

特点：

1、特点

信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。

信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。

每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

支持信号量组。

2、原型

最简单的信号量是只能取 0 和 1 的变量，这也是信号量最常见的一种形式，叫做二值信号量（Binary Semaphore）。而可以取多个正整数的信号量被称为通用信号量。

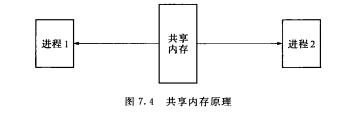
1. 共享内存（为什么说共享内存是进程高级通信方式最快的）

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，但是多个进程都可以访问。共享内存是最快的IPC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，如信号量，配合使用。来实现进程间的同步和通信。

共享内存允许两个或多个进程共享一个给定的存储区，这一段存储区可以被两个或两个以上的进程映射到自身的地址空间中，一个进程写入共享内存的信息，可以被其他使用这个共享内存的进程，通过一个简单的内存读取读出。从而实现进程间的通信。

这一段存储区可以被两个或两个以上进程映射到自己的地址空间中，一个进程写入共享内存的信息，可以被其他使用这个共享内存的进程，通过一个简单的内存读取读出，从而实现了进程间的通信。

采用共享内存进行通信的一个主要好处是**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，而共享内存则只拷贝两次：一次从**输入文件到共享内存区**，另一次从**共享内存到输出文件**。



共享内存由两种实现方式：1、内存映射 2、共享内存机制

特点：

* + - 1. 不用从用户态和内核态的频繁切换和拷贝数据，直接从内存中读取就可以（为什么说是最快的？）
      2. 共享内存是临界资源，所以需要用时必须要保证原子性。使用**信号量或者互斥锁**都可以
      3. 生命周期随内核

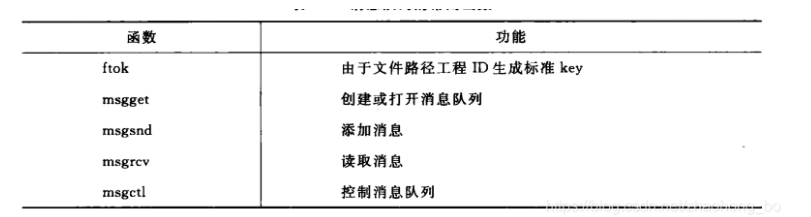
1. 消息队列

消息队列（MessageQueue）:消息队列是有消息的链表，存放在内核中并由消息队列标识符标识。**消息队列，就是一个消息的链表，是一系列保存在内核中消息的列表。用户进程可以向消息队列添加消息，也可以向消息队列读取消息。**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的缺点。

消息队列与管道通信相比，其优势是**对每个消息指定特定的消息类型**，接收的时候**不需要按照队列次序**，而是可以根据**自定义条件接收特定类型**的消息。

可以把消息看做一个记录，具有特定的格式以及特定的优先级。对消息队列有写权限的进程可以向消息队列中按照一定的规则添加新消息，对消息队列有读权限的进程可以从消息队列中读取消息。

消息队列的常用函数如下表：



进程间通过消息队列通信，主要是：创建或打开消息队列，添加消息，读取消息和控制消息队列。

1. 线程间通信方式
2. 信号、信号量跟锁
3. Linux中的mmap
4. 简单说一下什么是IO多路复用
5. Malloc申请一块内存会不会立即绑定到物理内存里

malloc申请的就是虚拟内存

1. 页面置换算法有哪些

页面置换算法：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。

1. **最佳置换算法（OPT）**（理想置换算法）
2. **先进先出置换算法（FIFO）**
3. 最近最久未使用-（LRU）算法（需要特殊的硬件（TLB））没有硬件，就不能使用LRU算法
4. 第二次机会算法：算法是对FIFO的一个修改，它会在删除页面之前检查这个页面是否仍在使用，如果页面正在使用，就会进行保留，这个改进大大提高了性能
5. 时钟(CLOCK)置换算法（LRU的近似算法，因为纯LRU算法需要较多硬件支持，成本较高，LRU的近似，对FIFO的改进 ）

把各个页面组织成环形链表（类似钟表面），把指针指向最老的页面（最先进来）；

当发生一个缺页中断，考察指针所指向的最老的页面，若它的访问为为0，则立即淘汰。若访问为1，则把该位置为0，然后指针往下移动一格。如此下去，直到找到被淘汰的页面，然后把指针移动到它的下一格。

**最好的算法是老化算法和WSClock算法**。他们分别是基于 LRU 和工作集算法。他们都具有良好的性能并且能够被有效的实现。还存在其他一些好的算法，但实际上这两个可能是最重要的。

1. 最近最不常用页面置换算法（LFU）
2. LRU和LFU的区别

LRU是最近最少使用页面置换算法(Least Recently Used),也就是首先淘汰最长时间未被使用的页面!

LFU是最近最不常用页面置换算法(Least Frequently Used),也就是淘汰一定时期内被访问次数最少的页!

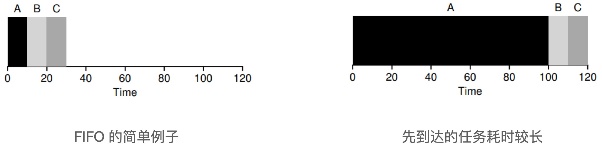
1. LRU和LFU如何实现

LRU缓存机制实现：leetcodeP146，双向链表增删查改O(1)加哈希表查找O(1)

LFU缓存机制实现：leetcodeP460，双向递减链表维护（当存在平局（即两个或更多个键具有相同使用频率）时，应该去除 **最久未使用** 的键），哈希表查找O(1)

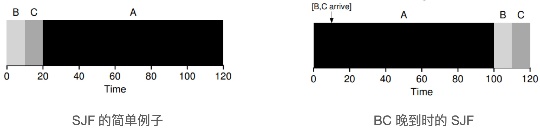
1. 进程调度算法、以及影响调度程序的指标
   * + 1. 先来先服调度算法原理（FIFO）

FIFO 就是先到先出调度 (First In First Out, FIFO)，有时也叫先到先服务调度 (First Come First Served, FCFS)。优点：很简单，易于实现。缺点：当先到达的任务耗时较长时，FIFO 调度算法的平均周转时间会很长。



* + - 1. 短作业优先的调度算法（需要注意的是，在所有的进程都可以运行的情况下，最短作业优先的算法才是最优的。）

最短作业优先的抢占式版本被称作为 **最短剩余时间优先**(Shortest Remaining Time Next) 算法。使用这个算法，调度程序总是选择剩余运行时间最短的那个进程运行。当一个新作业到达时，其整个时间同当前进程的剩余时间做比较。如果新的进程比当前运行进程需要更少的时间，当前进程就被挂起，而运行新的进程。这种方式能够使短期作业获得良好的服务。



* + - 1. 最短完成时间优先 (STCF)

为了解决 SJF 的问题，引入了最短完成时间优先 (Shortest Time-to-Completion First, STFT) 算法，前提是操作系统需要允许任务抢占。STCF 算法在 SJF 的基础上添加抢占，又称抢占式最短作业优先 (Preemptive Shortest Job First, PSJF)。每当新工作进入系统时，就会确定剩余工作和新工作中，谁的剩余时间最少，然后调度该工作。

* + - 1. 优先级调度算法原理

事实情况是不是所有的进程都是优先级相等的。例如，在一所大学中的等级制度，首先是院长，然后是教授、秘书、后勤人员，最后是学生。这种将外部情况考虑在内就实现了优先级调度(priority scheduling)

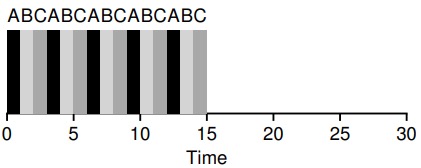


它的基本思想很明确，每个进程都被赋予一个优先级，优先级高的进程优先运行。

但是也不意味着高优先级的进程能够永远一直运行下去，调度程序会在每个时钟中断期间降低当前运行进程的优先级。如果此操作导致其优先级降低到下一个最高进程的优先级以下，则会发生进程切换。或者，可以为每个进程分配允许运行的最大时间间隔。当时间间隔用完后，下一个高优先级的进程会得到运行的机会。

* + - 1. 轮转调度算法原理。

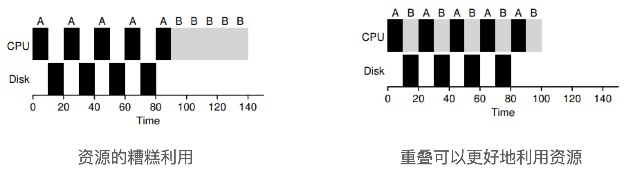
为了解决 STCF 算法响应时间长的问题，引入了轮转调度 (Round-Robin, RR)。轮转调度的思想：RR 在一个时间片 (time slice，有时称为调度因子，scheduling quantum) 内运行一个工作，然后切换到运行队列中的下一个任务，而不是运行一个任务直到结束。RR 有时被称为时间切片，时间片长度必须是时钟中断周期的倍数。



时间片长度对于 RR 是至关重要的。时间片越短，RR 在响应时间上表现越好。然而，时间片太短是有问题的：突然上下文切换的成本将影响整体性能。因此，系统设计者需要权衡时间片的长度，使其足够长，以便摊销上下文切换成本，而又不会使系统不及时响应。注：上下文切换的成本不仅仅来自保存和恢复少量寄存器的操作系统操作。程序运行时，它们在 CPU 高速缓存、TLB、分支预测器和其他片上硬件中建立了大量的状态。切换到另一个工作会导致此状态被刷新，且与当前运行的作业相关的新状态被引入，可能导致显著的性能成本。结合 I/O

当运行的程序在进行 I/O 操作的时候，在 I/O 期间不会使用 CPU，但它被阻塞等待 I/O 完成，这时调度程序应该在 CPU 上安排另一项工作。而在 I/O 完成时，会产生中断，操作系统运行并将发出 I/O 的进程从阻塞状态移回就绪状态。当然，它甚至可以决定在那个时候运行该项工作。操作系统应该如何处理每项工作？假设有两项工作 A 和 B，每项工作需要 50ms 的CPU时间。但是A先运行10ms，然后发出I/O请求（假设 I/O 每个都需要 10ms），而B只是使用CPU 50ms，不执行I/O。调度程序先运行A，然后运行B。

左图所示的调度是非常糟糕的。常见的方法是将 A 的每个 10ms 的子工作视为一项独立的工作。因此，当系统启动时，它的选择是调度 10ms 的 A，还是 50ms 的 B。STCF 会选择较短的 A。然后，A 的工作已完成，只剩下 B，并开始运行。然后提交 A 的一个新子工作，它抢占 B 并运行 10ms。这样做可以实现重叠，一个进程在等待另一个进程的 I/O 完成时使用 CPU，系统因此得到更好的利用。



* + - 1. **多级反馈队列**

操作系统常常不知道工作要运行多久，而这又是 SJF 等算法所必需的；而轮转调度虽然降低了响应时间，周转时间却很差。因此引入了多级反馈队列（Multi-level Feedback Queue，简称MLFQ）。MLFQ 需要解决两方面的问题：首先它要优化周转时间，这可以通过优先执行较短的工作来实现；其次，MLFQ 希望给用户提供较好的交互体验，因此需要降低响应时间。

MLFQ 中有许多独立的队列，每个队列有不同的优先级。任何时刻，一个工作只能存在于一个队列中。MLFQ 总是优先执行较高优先级的工作（即那些在较高级队列中的工作）。每个队列中可能会有多个工作，它们具有同样的优先级。在这种情况下，我们就对这些工作采用轮转调度。

基本规则多级反馈队列由几个基本规则：规则1：如果 A 的优先级大于 B 的优先级，运行 A 不运行 B。规则2：如果 A 的优先级等于 B 的优先级，轮转运行 A 和 B。规则3：工作进入系统时，放在最高优先级（最上层）队列。这一规则使得多级反馈队列算法类似 SJF，保证了良好的响应时间。规则4：一旦工作用完了其在某一层中的时间配额（无论中间主动放弃了多少次 CPU），就降低其优先级（移入低一级队列）。这一规则防止进程主动放弃 CPU，从而造成其他进程饥饿。规则5：每经过一段时间，就将系统中所有工作重新加入最高优先级队列。这一规则解决了两个问题：一是防止长进程饥饿，二是如果一个 CPU 密集型工作变成了交互型，当它优先级提升时，调度程序会正确对待它。比例份额比例份额（proportional-share）调度程序，有时也称为公平份额（fair-share）调度程序。比例份额算法认为，调度程序的最终目标是确保每个工作获得一定比例的 CPU 时间，而不是优化周转时间和响应时间。它的基本思想很简单：每隔一段时间，都会举行一次彩票抽奖，以确定接下来应该运行哪个进程。越是应该频繁运行的进程，越是应该拥有更多地赢得彩票的机会。彩票数表示份额在彩票调度中，彩票数代表了进程占有某个资源的份额。一个进程拥有的彩票数占总彩票数的百分比，就是它占有资源的份额。假设有两个进程A和B，A拥有75张彩票，B拥有25张。因此我们希望A占用75%的CPU时间，而B占用25%。通过不断且定时地抽取彩票，彩票调度从概率上获得这种份额比例。抽取彩票的过程很简单：调度程序知道总共的彩票数（在我们的例子中，有100张）。调度程序抽取中奖彩票，这是从0和99之间的一个数，拥有这个数对应的彩票的进程中奖。假设进程A拥有0到74共75张彩票，进程B拥有75到99的25张，中奖的彩票就决定了运行A或B。调度程序然后加载中奖进程的状态，并运行它。彩票调度利用了随机性，这导致了从概率上满足期望的比例。随着这两个工作运行得时间越长，它们得到的CPU时间比例就会越接近期望。实现彩票调度实现起来非常简单，只需要一个随机数生成器来选择中奖彩票和一个记录系统中所有进程的数据结构，以及所有彩票的总数。假设我们使用列表记录进程，下面的例子中有A、B和C这3个进程，每个进程有一定数量的彩票。步长调度虽然随机方式可以使得调度程序的实现简单，但偶尔并不能产生正确的比例，尤其在工作运行时间很短的情况下。由于这个原因，Waldspurger 提出了步长调度。系统中的每个工作都有自己的步长，这个值与票数值成反比。A、B、C这3个工作的票数分别是100、50和250，我们通过用一个大数分别除以他们的票数来获得每个进程的步长。比如用10000除以这些票数值，得到了3个进程的步长分别为100、200和40。我们称这个值为每个进程的步长（stride）。每次进程运行后，我们会让它的计数器（称为行程值）增加它的步长，记录它的总体进展。基本思路：当需要进行调度时，选择目前拥有最小行程值的进程，并且在运行后将该进程的行程值增加一个步长。伪代码：current = remove\_min(queue); // pick client with minimum passschedule(current); // use resource for quantumcurrent->pass += current->stride; // compute next pass using strideinsert(queue, current); // put back into the queue相比于步长调度，彩票调度的优势是不需要全局状态。假如一个新的进程在步长调度执行过程中加入系统，应该怎么设置它的行程值呢？设置成0吗？这样的话，它就独占CPU了。而彩票调度算法不需要对每个进程记录全局状态，只需要用新进程的票数更新全局的总票数就可以了。因此彩票调度算法能够更合理地处理新加入的进程。

* + - 1. **高响应比优先调度算法**

决定调度进程的好坏

**CPU 使用率：**CPU 正在执行任务（即不处于空闲状态）的时间百分比。**等待时间**这是进程轮流执行的时间，也就是进程切换的时间**吞吐量**单位时间内完成进程的数量**响应时间**这是从提交流程到获得有用输出所经过的时间。**周转时间**从提交流程到完成流程所经过的时间。

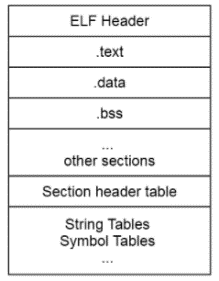
1. 操作系统中的时钟是什么

**时钟(Clocks) 也被称为定时器(timers)**，时钟/定时器对任何程序系统来说都是必不可少的。时钟负责维护时间、防止一个进程长期占用 CPU 时间等其他功能。时钟软件(clock software) 也是一种设备驱动的方式。下面我们就来对时钟进行介绍，一般都是先讨论硬件再介绍软件，采用由下到上的方式，也是告诉你，底层是最重要的。

1. 为什么 Linux 系统下的应用程序不能直接在 Windows 下运行

这是一个老生常谈的问题了，在这里给出具体的回答。

其中一点是因为 Linux 系统和 Windows 系统的格式不同，格式就是协议，就是在固定位置有意义的数据。Linux 下的可执行程序文件格式是 elf，可以使用 readelf 命令查看 **elf 文件头**。



而 Windows 下的可执行程序是 **PE 格式**，它是一种可移植的可执行文件。

还有一点是因为 Linux 系统和 Windows 系统的 API 不同，这个 API 指的就是操作系统的 API，Linux 中的 API 被称为系统调用，是通过 int 0x80 这个软中断实现的。而 Windows 中的 API 是放在动态链接库文件中的，也就是 Windows 开发人员所说的 DLL ，这是一个库，里面包含代码和数据。Linux 中的可执行程序获得系统资源的方法和 Windows 不一样，所以显然是不能在 Windows 中运行的。

1. Epoll中的函数族、ET与LT（使用场景，为什么ET高效）
2. Socket函数族，简单使用过程说一下
3. 不同链接，请求有多有少，项目怎么实现负载均衡
4. 在一个运行多个进程的系统中，如果某一个或者多个进程瞬间把CPU打满，所有的进程都会被卡死，你有什么样子的解决方案呢？
5. 有没有不用加锁的方式，无锁队列
6. 服务器如何提高吞吐量、测试项目的时延和传输速率
7. 软链接与硬链接的区别

<https://www.linuxprobe.com/soft-and-hard-links.html>

创建文件硬链接

ln myfile hard

软链接

ln -s myfile soft



总结

到这里我们其实可以总结一下了：

硬链接： 与普通文件没什么不同，inode 都指向同一个文件在硬盘中的区块  
软链接： 保存了其代表的文件的绝对路径，是另外一种文件，在硬盘上有独立的区块，访问时替换自身路径。

1. 一个进程fork出一个子进程，内存占用会不会翻倍

有可能会。fork处的子进程和父进程的页表都指向同样的物理段，当子进程和父进程只是要去读同样的物理段时不会发送改变，由于父子进程相互独立，而一旦子进程或者父进程要修改数据，采用写时复制的技术，来最大化的提高内存以及内核利用率。通俗点来说，就是子进程需要写时，将父子进程共同指向的数据进行拷贝出一份，而子进程此时页表对应指向该地址，并将该地址的数据再进行写修改。而将所有共有数据进行写时，才会实现内存的翻倍。

1. 多进程和多线程的区别（使用场景）



多线程模型适用于**I/O密集型场景**，因为I/O密集型场景因为I/O阻塞导致频繁切换，线程只占用栈，程序计数器，一组寄存器等少量资源，切换效率高，单机多核分布式；多进程模型适用于**需要频繁的计算场景**，多机分布式。

**多进程CPU密集型 多线程I/O密集型**

1. Linux 查看磁盘容量、找某个文件、内存占用量命令、查看进程

**df -hl：查看磁盘剩余空间**

**df -h**：查看每个根路径的分区大小

**du -sh [目录名]**：返回该目录的大小

**du -sm [文件夹]**：返回该文件夹总M数

**du -h [目录名]**：查看指定文件夹下的所有文件大小（包含子文件夹）

top 是每个系统管理员都知道的工具，用 top 命令，你能够知道到所有当前正在运行的进程有哪些。在命令行里，输入 top 命令能够就看到你正在运行的程序进程

尽管 top 命令很是方便，但也不是得到你所要信息最有效的方法。 你知道你要杀死的 Chrome 进程是那个，并且你也不想看 top 命令所显示的实时信息。 鉴于此，你能够使用 ps 命令然后用 grep 命令来过滤出输出结果。这个 ps 命令能够显示出当前进程列表的快照，然后用 grep 命令输出匹配的样式。我们通过 grep 命令过滤 ps 命令的输出的理由很简单：如果你只输入 ps 命令，你将会得到当前所有进程的列表快照

ps aux | grep chrome（动态实时的看到CPU和内存的占用率）

top -p 2913（[**查看某一进程所占用内存的方法**](https://www.cnblogs.com/xuanbjut/p/11564744.html)）

netstat -napt

1. 无锁队列

在开始说无锁队列之前，我们需要知道一个很重要的技术就是CAS操作——Compare & Set，或是 Compare & Swap，现在几乎所有的CPU指令都支持CAS的原子操作，X86下对应的是 CMPXCHG 汇编指令。有了这个原子操作，我们就可以用其来实现各种无锁（lock free）的数据结构。

这个操作用C语言来描述就是下面这个样子：（代码来自Wikipedia的Compare And Swap词条）意思就是说，看一看内存\*reg里的值是不是oldval，如果是的话，则对其赋值newval。

<https://blog.csdn.net/21cnbao/article/details/108765253>

CAS保证实现

所谓ABA（见维基百科的ABA词条），问题基本是这个样子：

1. 进程P1在共享变量中读到值为A
2. P1被抢占了，进程P2执行
3. P2把共享变量里的值从A改成了B，再改回到A，此时被P1抢占。
4. P1回来看到共享变量里的值没有被改变，于是继续执行。

虽然P1以为变量值没有改变，继续执行了，但是这个会引发一些潜在的问题。ABA问题最容易发生在lock free 的算法中的，CAS首当其冲，因为CAS判断的是指针的值。很明显，值是很容易又变成原样的。

比如上述的DeQueue()函数，因为我们要让head和tail分开，所以我们引入了一个dummy指针给head，当我们做CAS的之前，如果head的那块内存被回收并被重用了，而重用的内存又被EnQueue()进来了，这会有很大的问题。（内存管理中重用内存基本上是一种很常见的行为）

以上基本上就是所有的无锁队列的技术细节，这些技术都可以用在其它的无锁数据结构上。

1）无锁队列主要是通过CAS、FAA这些原子操作，和Retry-Loop实现。

2）对于Retry-Loop，我个人感觉其实和锁什么什么两样。只是这种“锁”的粒度变小了，主要是“锁”HEAD和TAIL这两个关键资源。而不是整个数据结构。

1. 如果让你自己实现一个线程池，你怎么控制这个线程的并发数

<https://blog.csdn.net/manzhizhen/article/details/81413014>

1. 消息队列了解吗

消息队列（Message queue）

我们可以把**消息队列比作是一个存放消息的容器**，当我们**需要使用消息的时候可以取出消息供自己使用**。消息队列是分布式系统中重要的组件，使用消息队列主要是为了**通过异步处理提高系统性能**和**削峰**、**降低系统耦合性**。

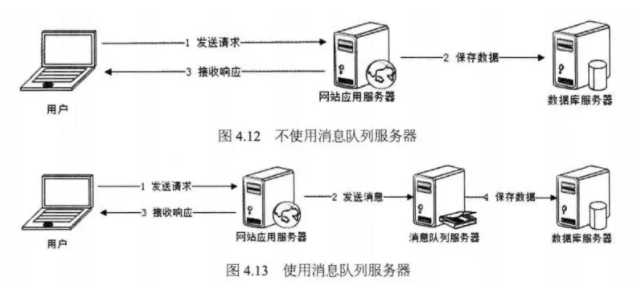
另外，我们知道队列 Queue 是一种先进先出的数据结构，所以消费消息时也是按照顺序来消费的。比如生产者发送消息1,2,3...对于消费者就会按照1,2,3...的顺序来消费。

这三个场景也是消息队列的经典场景，大家基本上要烂熟于心那种，就是一说到消息队列你脑子就要想到**异步、削峰、解耦**，条件反射那种。

好处：

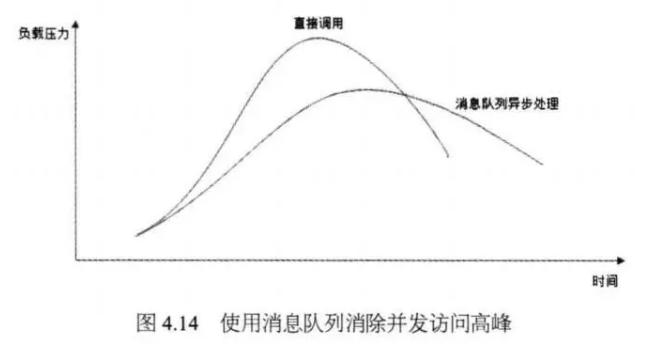
* + - 1. 通过异步处理提高系统性能（削峰、减少响应所需时间）
      2. 降低系统耦合性。

1. **通过异步处理提高系统性能（削峰、减少响应所需时间）**



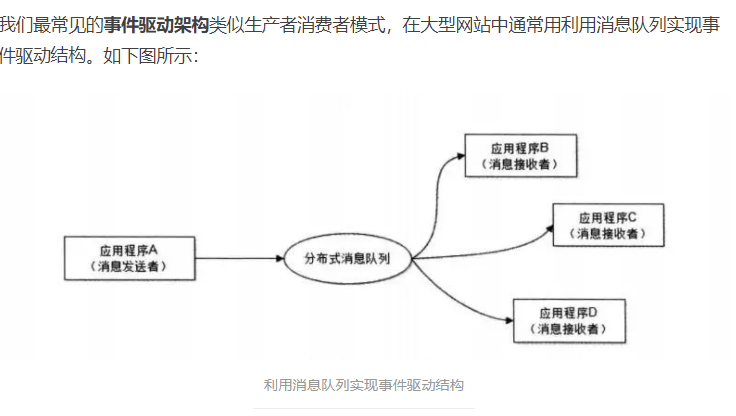
在不使用消息队列服务器的时候，用户的请求数据直接写入数据库，在高并发的情况下数据库压力剧增，使得响应速度变慢。但是在使用消息队列之后，用户的**请求数据发送给消息队列之后立即返回**，再由消息队列的消费者进程从消息队列中获取数据，**异步写入数据库。**由于**消息队列服务器处理速度快于数据库**（消息队列也比数据库有更好的伸缩性），因此响应速度得到大幅改善。

通过以上分析我们可以得出**消息队列具有很好的削峰作用的功能——即通过异步处理，将短时间高并发产生的事务消息存储在消息队列中，从而削平高峰期的并发事务**。 举例：在电子商务一些秒杀、促销活动中，合理使用消息队列可以有效抵御促销活动刚开始大量订单涌入对系统的冲击。如下图所示：



因为**用户请求数据写入消息队列之后就立即返回给用户了，但是请求数据在后续的业务校验、写数据库等操作中可能失败**。因此使用消息队列进行异步处理之后，需要**适当修改业务流程进行配合**，比如**用户在提交订单之后，订单数据写入消息队列，不能立即返回用户订单提交成功，需要在消息队列的订单消费者进程真正处理完该订单之后，甚至出库后，再通过电子邮件或短信通知用户订单成功**，以免交易纠纷。这就类似我们平时手机订火车票和电影票。

1. **降低系统耦合性**



**消息队列使利用发布-订阅模式工作，消息发送者（生产者）发布消息，一个或多个消息接受者（消费者）订阅消息。** 从上图可以看到**消息发送者（生产者）和消息接受者（消费者）之间没有直接耦合**，消息发送者将消息发送至分布式消息队列即结束对消息的处理，消息接受者从分布式消息队列获取该消息后进行后续处理，并不需要知道该消息从何而来。**对新增业务，只要对该类消息感兴趣，即可订阅该消息，对原有系统和业务没有任何影响，从而实现网站业务的可扩展性设计**。

宕机问题：

**另外为了避免消息队列服务器宕机造成消息丢失，会将成功发送到消息队列的消息存储在消息生产者服务器上，等消息真正被消费者服务器处理后才删除消息。在消息队列服务器宕机后，生产者服务器会选择分布式消息队列服务器集群中的其他服务器发布消息。**

消息队列的问题：

**系统复杂性：**本来蛮简单的一个系统，我代码随便写都没事，现在你凭空接入一个中间件在那，我是不是要考虑去维护他，而且使用的过程中是不是要考虑各种问题，比如消息**重复消费、消息丢失、消息的顺序**消费等等，反正用了之后就是贼烦。

**数据一致性：**这个其实是分布式服务本身就存在的一个问题，**不仅仅是消息队列的问题**，但是放在这里说是因为用了消息队列这个问题会暴露得比较严重一点。  
 就像我开头说的，你下单的服务自己保证自己的逻辑成功处理了，你成功发了消息，但是优惠券系统，积分系统等等这么多系统，**他们成功还是失败你就不管了？** 我说了保证自己的业务数据对的就好了，其实还是比较不负责任的一种说法，这样就**像个渣男，没有格局**，**这样呀你的路会越走越窄的**。

**所有的服务都成功才能算这一次下单是成功的**，那怎么才能保证数据一致性呢？

**分布式事务**：把下单，优惠券，积分。。。都放在一个事务里面一样，要成功一起成功，要失败一起失败。

**可用性：**你搞个系统本身没啥问题，你现在突然接入一个中间件在那放着，万一挂了怎么办？我下个单MQ挂了，优惠券不扣了，积分不减了，这不是杀一个程序员能搞定的吧，感觉得杀一片。

1. 讲一下你项目的设计思想以及那些中间和做这个项目自己的一些考虑
2. 项目部署在服务器上，当发现其实一些CPU占用过高的情况下，如何去定位问题

1.使用top命令找到消耗CPU较高的进程，记录下进程ID

再执行top -p 进程ID -H 单独监控对应进程ID下的所有线程信息

Jstack 对上述的线程所有信息做dump记录。

2.使用gdb

使用top查看CPU占用过高的进程，记录下进程ID下的线程号

Attach 2907（对应进程）然后列出info threads 所有线程信息，发现CPU占用高的线程号

thread 对应线程编号 然后**bt 根据打印消息，重点定位响应代码段**。

1. 手写LRU
2. 实现memcpy函数
3. 100w个玩家一起创建角色…中途如果有玩家取消创建
4. 亿级数据排序
5. 在项目面碰到最棘手或者最难的一个问题是什么
7. 和平精英遇人掉帧问题真的存在吗？还是概率性存在
8. 平板玩家普遍比手机玩家操作流畅，有没有想过制衡
9. 和平精英的匹配机制很奇怪，我经常赛季没打，后面接着一下子匹配高星段位
11. 64匹马8赛道怎么决出前4名