进程和线程的区别和联系

概念上

最开始就是一个进程跑完再跑另一个进程，后来有多道程序分派之后，我们可以利用进程调度算法来在一个电脑上跑多个程序，在我们用户看来这些程序是一起运行的。随着技术的发展，多核处理器的实现使得线程越来越普及，因此我们计算机就可以适配多核出现了线程。

进程：进程是系统中正在运行的程序。是计算机分配资源的单位，每一个进程都会有属于自己独立的内存空间，磁盘空间，I\O设备等等。比如说windows中的任务管理器，每一行都是一个进程。在同一进程中还是在不同进程中时系统功能划分的重要决策点。

> 可以把进程比做一个人。每个人都有自己的记忆（内存），人与人通过谈话（消息传递）来交流，谈话既可以面谈（同一个电脑），也可以远程谈（不同服务器，网络通信）。面谈和远程的区别在于，面谈的话可以立刻知道对方死没死（SIGCHILD信号），而远程只能通过周期性的信条来判断对方是否活着。

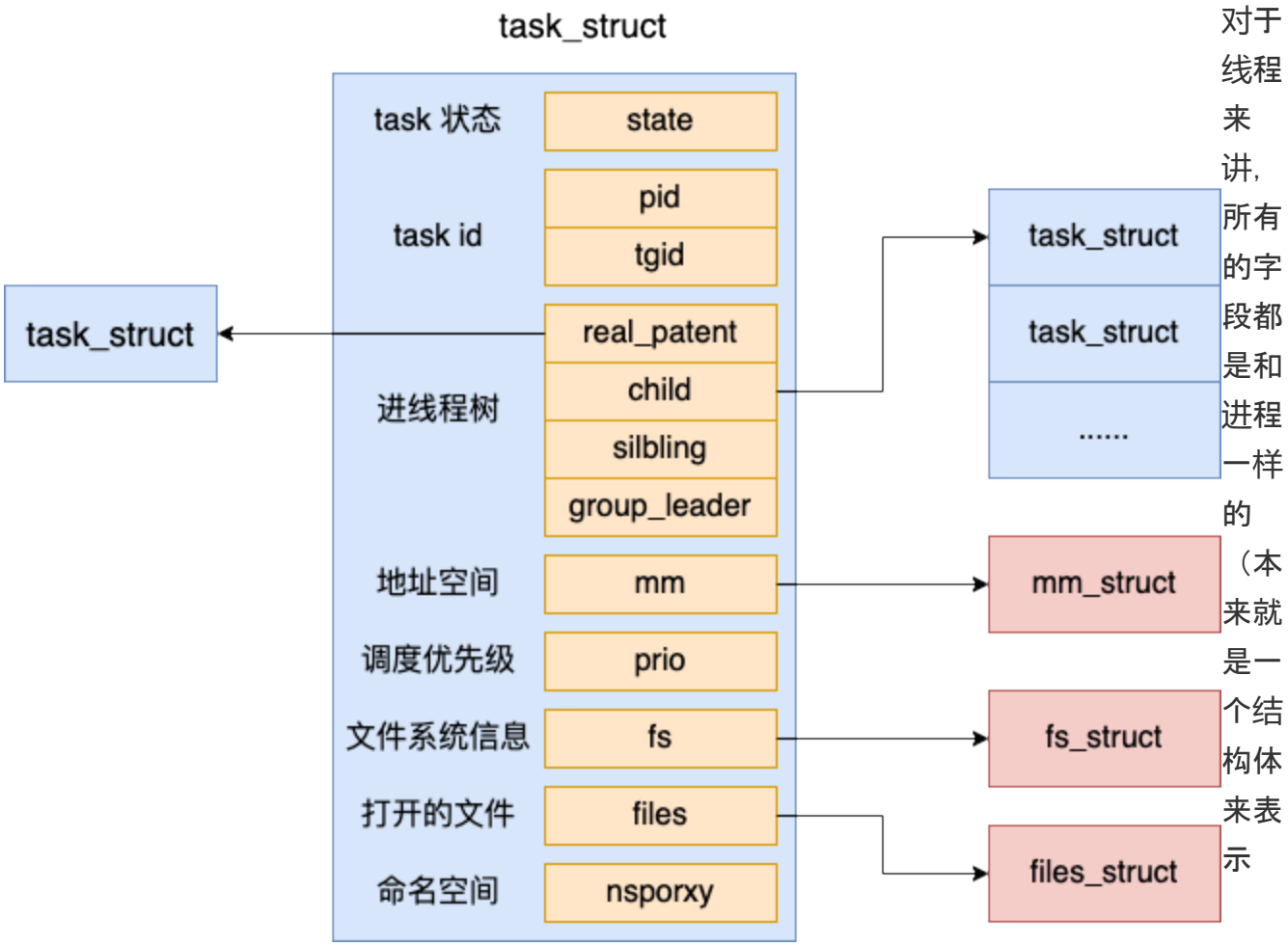
线程：线程是任务调度的基本单元。（现在来看这局话其实就是正在运行的一个程序有很多任务，而线程是将任务呈现出细粒度，更精确了）其实我觉得线程主要扮演的角色就是如何利用cpu处理代码，完成当前进程中的各个子任务。各个子线程之间共享父进程的代码空间和全局变量，但是每个进程都有自己独立的堆栈，即局部变量对于线程来说是私有的。因此创建多进程代价有点大，在一个进程中创建多线程代价要小很多。

线程大概93年出现的，有SUN Solaris操作系统使用的线程叫做**UNIX International线程**，现在一直用的POSIX线程（POSIX threads），Pthreads线程的头文件是`<pthread.h>`，Win32线程是Windows API的一部分。

线程的特点就是共享地址空间，从而可以高效的共享数据。多线程的价值在于更好的发挥了多核处理器的效能，在单核时代多线程没啥用，因为就一个核心，一个执行单元，按状态机的思路去写程序是最高效的。

内核实现上

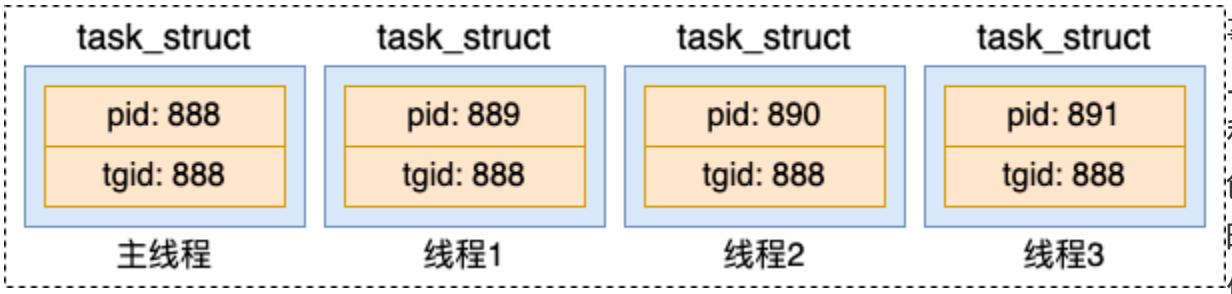
进程和线程的相同点要远远大于不同点。主要依据就是在 Linux 中，无论进程还是线程，都是抽象成了 task 任务，在源码里都是用 task_struct 结构来实现的。



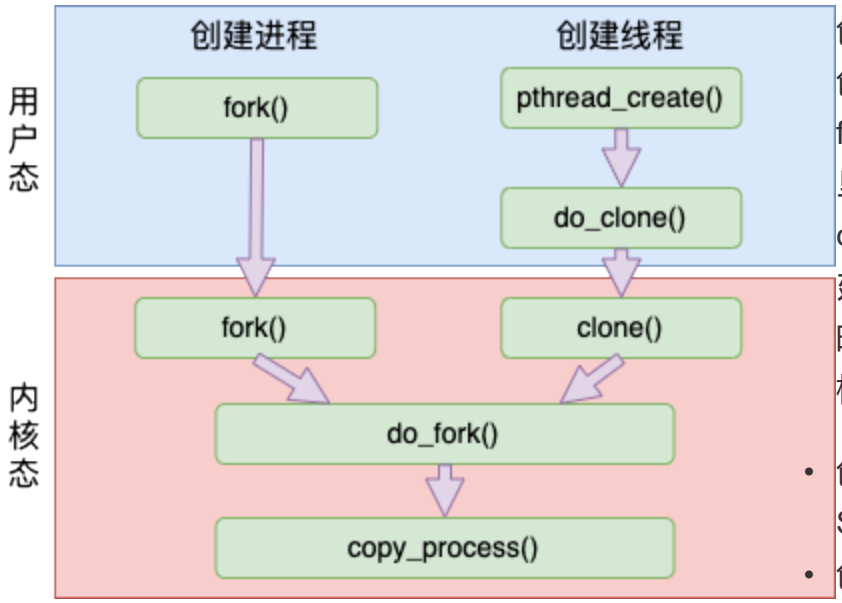
对于线程来讲，所有的字段都是和进程一样的（本来就是一个结构体来表示的）。包括状态、pid、task 树关系、地址空间、文件系统信息、打开的文件信息等等字段，线程也都有。在 Linux 下的线程还有另外一个名字，叫轻量级进程。

进程中pid 就是我们平时常说的进程 pid，在 Linux 中，每一个 task_struct 都需要被唯一的标识，它的 pid 就是唯一标识号。

对于线程来说，我们假如一个进程下创建了多个线程出来。那么每个线程的 pid 都是不同的。但是我们一般又需要记录线程是属于哪个进程的。这时候，tgid 就派上用场了，通过 tgid 字段来表示自己所归属的进程 ID。



事实上，进程线程创建的时候，使用的函数看起来不一样。但实际在底层实现上，最终都是使用同一个函数来实现的。



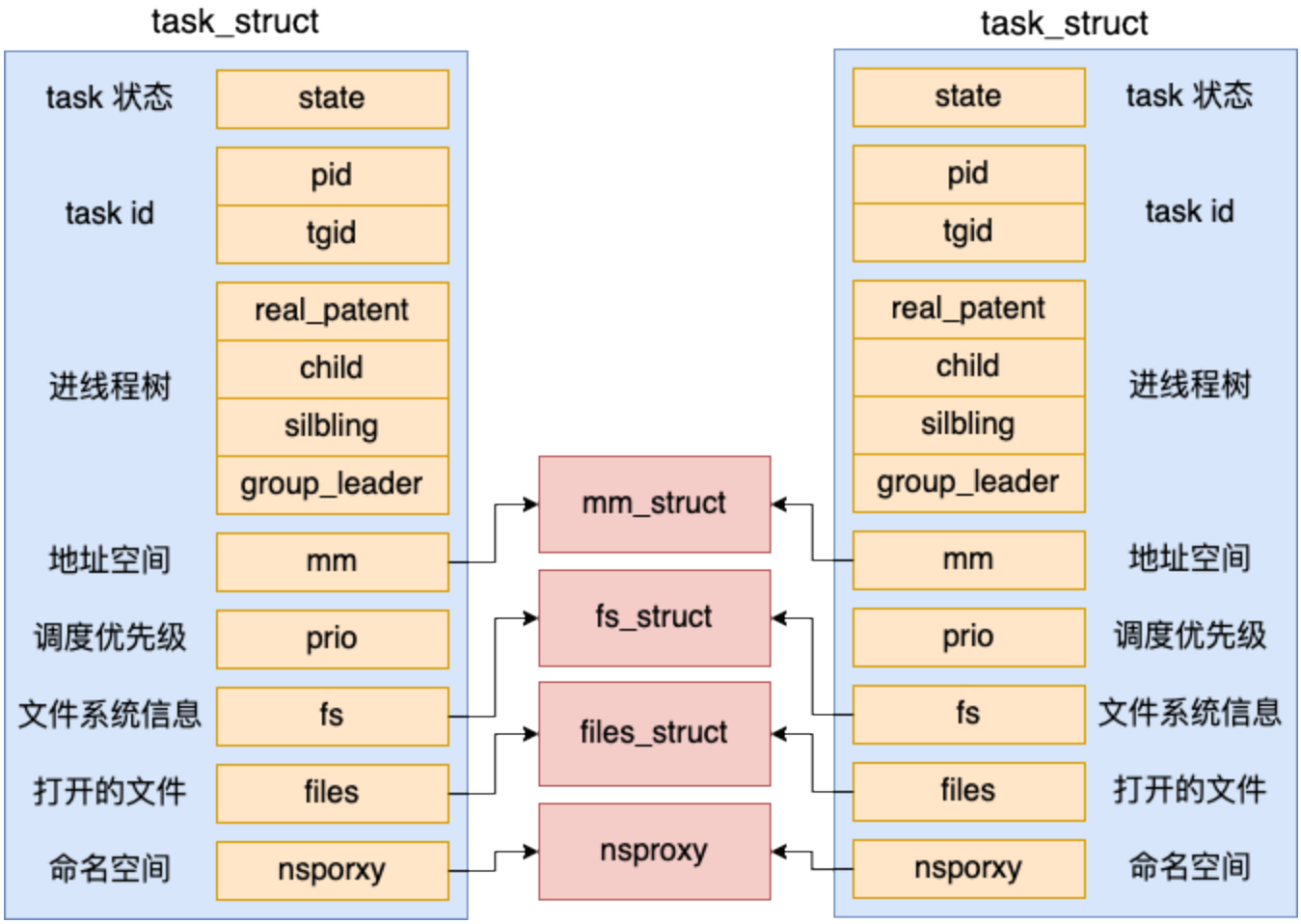
创建进程时使用的 fork 系统调用，创建线程的 clone 系统调用几乎和 fork 差不多，也一样使用的是内核里的 do_fork 函数，最后走到 copy_process 来完整创建。不过创建过程的区别是二者在调用 do_fork 时传入的 clone_flags 里的标记不一样！。

- 创建进程时的 flag：仅有一个 SIGCHLD
- 创建线程时的 flag：包括 CLONE_VM、CLONE_FS、CLONE_FILES、CLONE_SIGNAL、CLONE_SETTLS、CLONE_PARENT_SETTID、CLONE_CHILD_CLEARTID、CLONE_SYSVSEM。

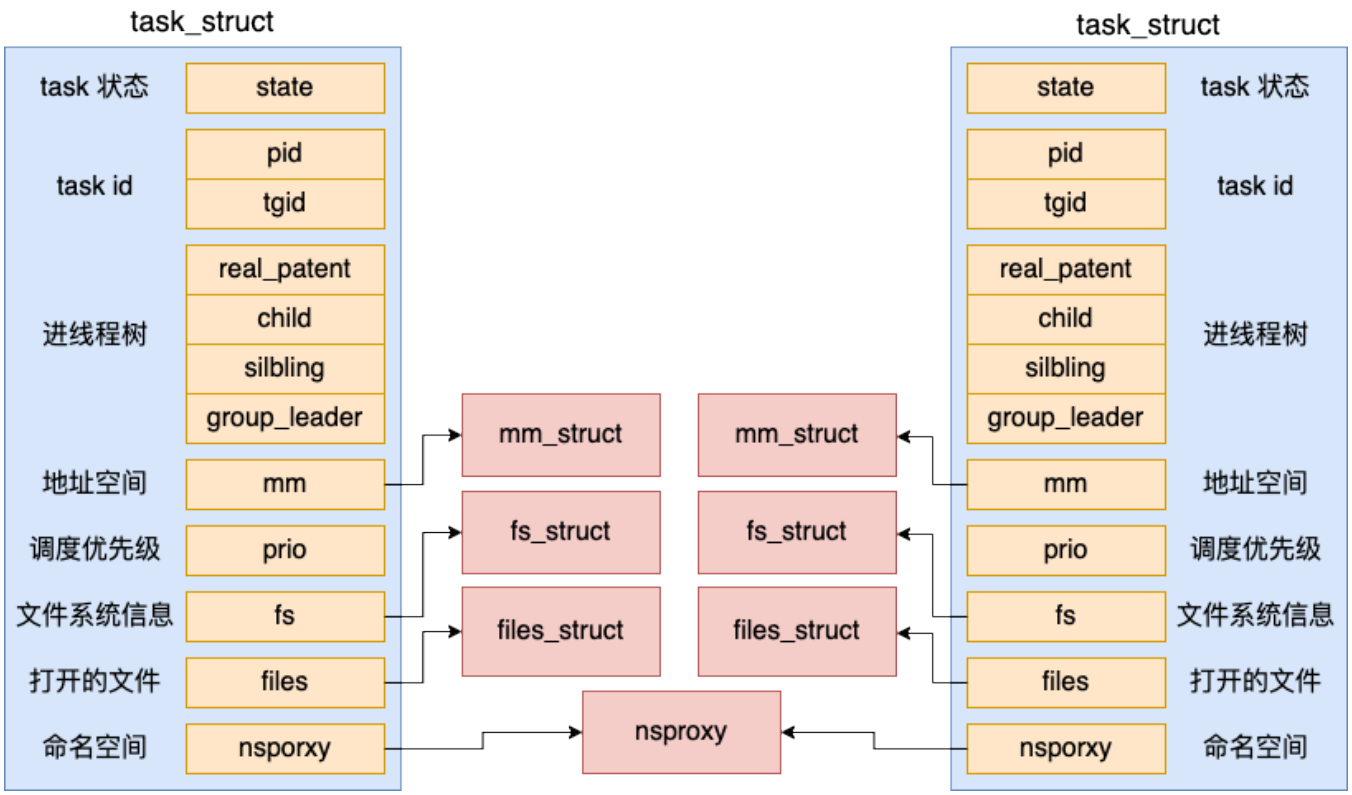
> - CLONE_VM: 新 task 和父进程共享地址空间
> - CLONE_FS：新 task 和父进程共享文件系统信息
> - CLONE_FILES：新 task 和父进程共享文件描述符表

进程和线程创建都是调用内核中的 do_fork 函数来执行的。在 do_fork 的实现中，核心是一个 copy_process 函数，它以拷贝父进程（线程）的方式来生成一个新的 task_struct 出来。copy_process 先是复制了一个新的 task_struct 出来，然后调用 copy_xxx 系列的函数对 task_struct 中的各种核心对象进行拷贝处理，还申请了 pid 。

对于线程来讲，其地址空间 mm_struct、目录信息 fs_struct、打开文件列表 files_struct 都是和创建它的任务共享的。如图：



对于进程来讲，地址空间 mm_struct、挂载点 fs_struct、打开文件列表 files_struct 都要是独立拥有的，都需要去申请内存并初始化它们。如图：



在 Linux 内核中并没有对线程做特殊处理，还是由 task_struct 来管理。从内核的角度看，用户态的线程本质上还是一个进程。只不过和普通进程比，稍微“轻量”了那么一些。那么线程具体能轻量多少呢？我之前曾经做过一个进程和线程的上下文切换开销测试。进程的测试结果是一次上下文切换平均 2.7 - 5.48 us 之间。线程上下文切换是 3.8 us左右。总的来说，进程线程切换还是没差太多。

比喻

做个简单的比喻：进程=火车，线程=车厢

1. 线程在进程下行进（单纯的车厢无法运行）
2. 一个进程可以包含多个线程（一辆火车可以有多个车厢）
3. 不同进程间数据很难共享（一辆火车上的乘客很难换到另外一辆火车，比如站点换乘）
4. 同一进程下不同线程间数据很易共享（A车厢换到B车厢很容易）
5. 进程要比线程消耗更多的计算机资源（采用多列火车相比多个车厢更耗资源）
6. 进程间不会相互影响，一个线程挂掉将导致整个进程挂掉（不一定）（一列火车不会影响到另外一列火车，但是如果一列火车上中间的一节车厢着火了，将影响到所有车厢）
7. 线程使用的内存地址可以上锁，即一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。（比如火车上的洗手间）—"互斥锁"

> 所谓内核中的任务调度，实际上的调度对象是线程；而进程只是给线程提供了虚拟内存、全局变量等资源。所以，对于线程和进程，我们可以这么理解： - 当进程只有一个线程时，可以认为进程就等于线程。 - 当进程拥有多个线程时，这些线程会共享相同的虚拟内存和全局变量等资源。这些资源在上下文切换时是不需要修改的。 - 另外，线程也有自己的私有数据，比如栈和寄存器等，这些在上下文切换时也是需要保存的。

线程调度为什么比进程调度更少开销？

进程与线程的差异

从概念上来讲，线程是进程的一部分，只是任务调度相关的部分，所以我们才说，“线程是调度的最小单位”。进程拥有着资源，这些资源不属于某一个特定线程，因为所有线程共享进程拥有的资源，所以我们才说，“进程是资源分配的最小单位”。

我们fork一个新的进程时，实际上“伴生”了一个线程，而这个唯一的线程，实际上代表了这个进程参与到任务调度。在linux中，不管进程还是线程，都用`struct task_struct`描述。`struct mm_struct *mm`，这是一个指针，指向实际的内存资源。同一个进程内的所有线程，他们都使用相同的资源，只需要把对应的资源指针指向相同的地址。

Linux内核就好像淡化了“线程”的概念，每一个线程描述都是`struct task_struct`，他们都是一个独立的“进程”，都有着自己的进程号，都参与任务调度，只不过指向相同的进程资源。

任务调度的开销

任务调度的主要开销

1. CPU执行任务调度的开销，主要是进程上下文切换的开销
2. 任务调度后，CPU Cache/TLB不命中，导致缺页中断的开销

对于第1点的开销，不管是进程调度还是线程调度都是必须的，所以，两者的差异体现在第2点。既然线程调度的`struct task_struct`都使用相同的资源，是不是就意味着，我即使切换到了其他的线程，CPU Cache/TLB命中的概率会高很多？相反，进程调度使用的是不同的资源，每次换了个进程，就意味着原有的Cache就不适用了，没命中，就触发更多的缺页中断，开销自然就更多。

linux中，线程都是独立的`struct task_struct`，都参与任务调度，那这里说的线程调度和进程调度怎么区分？

线程调度：使用相同资源的`struct task_struct`之间的调度

进程调度：使用不同资源的`struct task_struct`之间的调度

线程和进程之前共享那些资源？

同一进程的所有线程共享以下资源：

1. 堆。堆是在进程空间内开辟的，所以被共享
2. 全局变量。与某一函数无关，与特定线程无关
3. 静态变量。静态变量存放位置和全局变量一样，都存在于堆中开辟的.bss和.data段，是共享的
4. 其他一些共用资源，比如文件。

同一进程的所有线程独享以下资源：

1. 栈。
2. 寄存器。
3. 程序计数器

线程运行的本质就是函数的执行，而函数的执行总会有一个源头，这个源头叫做入口函数，cup从入口函数开始一步一步向下执行，这个过程就叫做线程。由于函数运行时信息是保存在栈中的，比如返回值，参数，局部变量等等，所以栈是私有的。

cpu执行指令的信息会保存在寄存器中，这个寄存器叫做程序计数器。由于操作系统可以随时终止线程的运行，所以保存和恢复程序计数器的值就知道线程从哪里暂停的以及从哪里开始运行。

进程间通信方式

每个进程各自有不同的用户地址空间，任何一个进程的全局变量在另一个进程中都看不到，所以进程之间要交换数据必须通过内核，内核是可以共享的。在内核中开辟一块缓冲区，进程1把数据从用户空间拷到内核缓冲区，进程2再从内核缓冲区把数据读走，内核提供的这种机制称为**进**

程间通信（IPC，InterProcess Communication）

- **管道**

  管道允许进程以先进先出的方式传送数据，是半双工的，意味着数据只能往一个方向流动。因此当双方通信时，必须建立两个管道。

  管道的实质就是在内核中创建一个缓冲区，管道一端的进程进入管道写数据，另一端的进程进入管道读取数据。

  管道分为pipe和FIFO两种

  - pipe：用于相关联的进程，比如父进程和子进程之间的通信。
  - FIFO：命名管道，即任何进程可以根据管道的文件名将其打开和读写。

  缺点：管道本质上是通过内核交换数据的，因此通信效率很低，不适合频繁交换数据的情况。

  匿名管道的周期随着进程的创建而创建，销毁而销毁。

- **消息队列**

  消息队列是保存在内核中的链表，由一个个独立的数据块组成，消息的接收方和发送方要约定具体的消息类型。当进程从消息队列中读取了相关数据块，则内核会将该数据块删除。跟管道相比，消息队列不一定按照先进先出的方式读取，也可以按照消息类型进行兑取。

  消息队列的生命周期与内核相关，如果不显示的删除消息队列，则消息队列会一直存在

  消息队列这种通信方式，跟收发邮件类似。两个进程你来我往的进行沟通

  缺点：不能实现实时通信。数据块是有大小限制的。**消息队列通信过程中，存在用户态与内核态之间的数据拷贝开销**，因为进程写入数据到内核中的消息队列时，会发生从用户态拷贝数据到内核态的过程，同理另一进程读取内核中的消息数据时，会发生从内核态拷贝数据到用户态的过程。

- **共享内存**

  共享内存技术就是要解决用户态和内核态之间频繁发生拷贝过程的。现代操作系统对于内存管理普遍采用的是虚拟内存技术，每个进程都有自己独立的虚拟内存空间，不同进程的虚拟内存空间映射到不同的物理内存中。

  数据不需要在不同进程之间进行复制，这是最快的一种IPC

  共享内存技术的实质就是拿出一块虚拟地址空间，映射到相同的物理内存中。这样的好处是一个进程写入数据后另一个进程可以立刻看到，不用进行拷贝。效率很高。

- **信号**

  上面那几种进程间通信，都是常规状态下的。异常状态下的需要用信号来通知进程。

信号和信号量就像雷锋和雷峰塔的区别。
可以在任何时刻给进程发送信号，信号是进程间通信或操作的一种异步通信机制。
收到信号后进程对信号的处理有三种方式

i. 如果是系统定义的信号函数，执行默认操作。

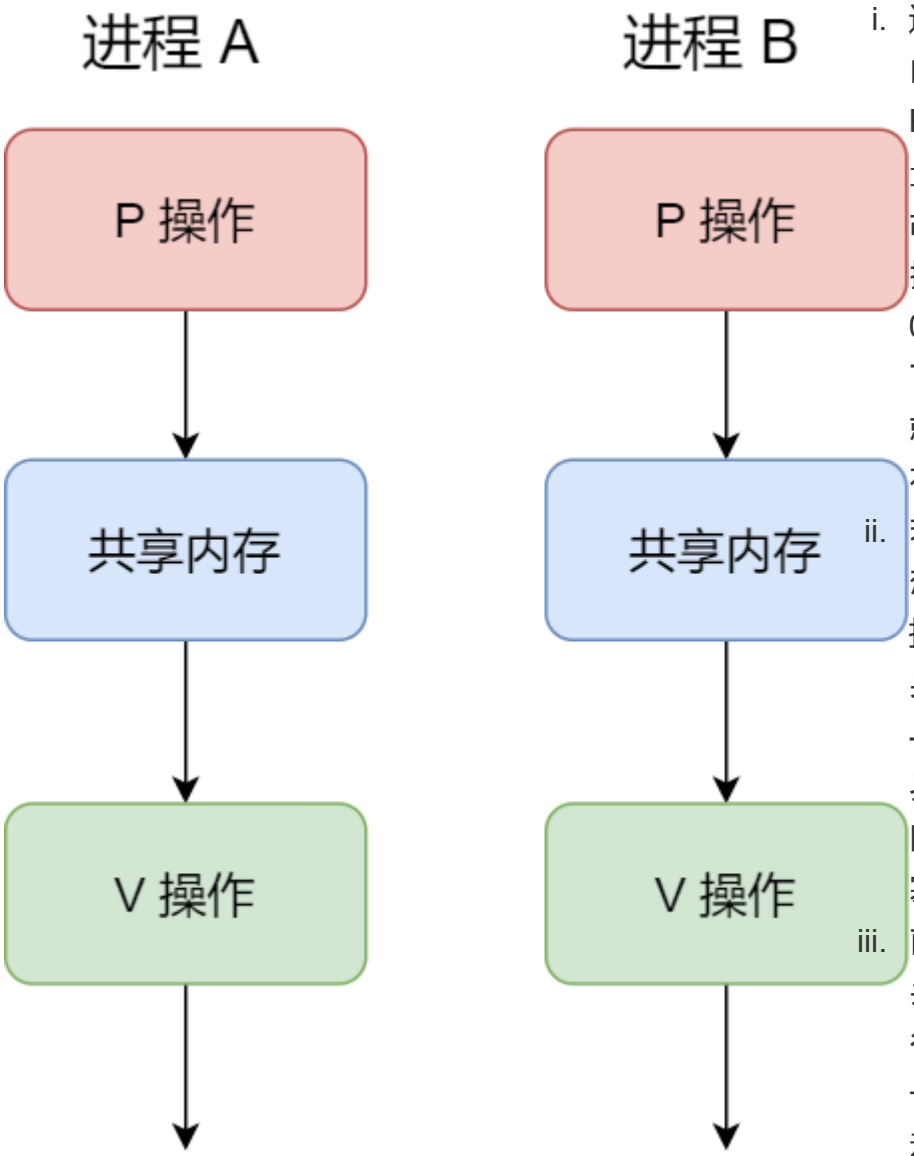
> **SIGINT：**程序终止信号。程序运行过程中，按 `Ctrl+C` 键将产生该信号。
>
> **SIGQUIT：**程序退出信号。程序运行过程中，按 `Ctrl+\\` 键将产生该信号。
>
> **SIGALRM：**定时器信号。
>
> **SIGTERM：**结束进程信号。shell下执行 `kill 进程pid` 发送该信号。

ii. 捕捉信号。用户可以给信号定义信号处理函数，表示收到信号后该进程该怎么做。

iii. 忽略信号。当我们不希望处理某些信号的时候，就可以忽略该信号，不做任何处理。有两个信号是应用进程无法捕捉和忽略的，即 `SIGKILL` 和 `SEGSTOP`，它们用于在任何时候中断或结束某一进程。

- **unix域套接字**
  具体指unix域间套接字。socket API原本是为网络通讯设计的，但后来在socket的框架上发展出一种IPC机制，就是UNIX Domain Socket。虽然网络socket也可用于同一台主机的进程间通讯（通过loopback地址127.0.0.1），但是UNIX Domain Socket用于IPC更有效率：不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程。**UNIX域套接字与TCP套接字相比较，在同一台主机的传输速度前者是后者的两倍。**这是因为，IPC机制本质上是可靠的通讯，而网络协议是为不可靠的通讯设计的。UNIX Domain Socket也提供面向流和面向数据包两种API接口，类似于TCP和UDP，但是面向消息的UNIX Domain Socket也是可靠的，消息既不会丢失也不会顺序错乱。
- **信号量**（同步原语，并不叫IPC）
  本质是一个计数器，用于为多个进程提供对共享对象的访问
  共享内存在效率高的同时也带来了新的问题，即如果多个进程同时对一个共享内存进行操作，会产生冲突造成不可预计的后果。
  为了不冲突，共享内存在一个时间段只能有一个进程访问，就出现了信号量。
  信号量其实是一个计数器，用于实现进程间的互斥和同步。
  信号量表示资源的数量。 P操作 会把信号量-1，-1之后如果信号量的值<0，则表示

资源已经被占用，进程需要阻塞等待。如果信号量-1后>= 0，表明进程可以正常执行。**V操作**跟P操作正好相反。



i. 进程 A 在访问共享内存前，先执行了 P 操作，由于信号量的初始值为 1，故在进程 A 执行 P 操作后信号量变为 0，表示共享资源可用，于是进程 A 就可以访问共享内存。

ii. 若此时，进程 B 也想访问共享内存，执行了 P 操作，结果信号量变为了 -1，这就意味着临界资源已被占用，因此进程 B 被阻塞。

iii. 直到进程 A 访问完共享内存，才会执行 V 操作，使得信号量恢复为 0，接着就会唤醒阻塞中的线程 B，使得进程 B 可以访问共享内存，最后完成共享内存的访问后，执行 V 操作，使信号量恢复到初始值 1。

> **信号量与互斥量之间的区别**：
>
> （1）互斥量用于线程的互斥，信号量用于线程的同步。这是互斥量和信号量的根本区别，也就是互斥和同步之间的区别。
>
> **互斥：**是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。

> **同步：**是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。
>
> （2）互斥量的加锁和解锁必须由同一线程分别对应使用，信号量可以由一个线程释放，另一个线程得到。

进程同步

在多道程序环境下，当程序并发执行时候，由于资源共享和进程间相互协作的关系，同一个系统中的诸进程之间会存在一下两种相互制约的关系：

1. 间接相互制约。主要是资源共享这种情况
2. 直接相互制约。源于进程间的相互合作，例如A进程向B进程提供数据，当A缓存没数据的时候B就阻塞，A缓存满时A就阻塞。

进程同步首先要搞明白临界区

- 临界区
  许多硬件资源如打印机，磁带机等，都属于临界资源，诸进程应该采取互斥方式，实现对这种资源的共享。人们把在每个进程中访问临界资源的那段代码成为临界区，显然，若能保证诸进程互斥地进入自己的临界区，便可实现诸进程对临界资源的互斥访问。
- 同步机制遵循的原则
  i. 空闲让进，当无进程处于临界区时，表明临界资源处于空闲状态，应允许一个请求进入临界区的进程立即进入自己的临界区，以有效的利用临界资源。
  ii. 忙则等待，当已有进程进入临界区时，表明临界资源正在被访问，因而其他视图进入临界区的进程必须等待，以保证对临界资源的互斥访问。
  iii. 有限等待，对要求访问临界资源的进程，应保证在有限时限内能进入自己的临界区，以免陷入死等状态。
  iv. 让权等待，当进程不能进入自己的临界区时，应立即释放处理机，以免进程陷入忙等状态。
- 进程同步实现机制
  i. 提高临界区代码执行中断的优先级
  在传统操作系统中，打断进程对临界区代码的执行只有中断请求、中断

一旦被接受，系统就有可能调用其它进程进入临界区，并修改此全局数据库。所以用提高临界区中断优先级方法就可以屏蔽了其它中断，保证了临界段的执行不被打断，从而实现了互斥。

ii. 自旋锁

内核（处理器也如此）被保持在过渡状态“旋转”，直到它获得锁，“自旋锁”由此而得名。

自旋锁像它们所保护的数据结构一样，储存在共用内存中。为了速度和使用任何在处理器体系下提供的锁定机构，获取和释放自旋锁的代码是用汇编语言写的。

iii. 信号量机制

跟进程同步信号量机制一样

- 经典进程同步问题

  具体参考

  i. 生产者——消费者问题
  ii. 哲学家进餐问题
  iii. 读者——写者问题

经典的进程间通信(同步)问题

> P是减小
>
> V是增加

生产者—消费者

> **问题描述：**两个进程（生产者和消费者）共享一个公共的固定大小的缓冲区。生产者将数据放入缓冲区，消费者从缓冲区中取数据。也可以扩展成m个生产者和n个消费者。当缓冲区空的时候，消费者因为取不到数据就会睡眠，知道缓冲区有数据才会被唤醒。当缓冲区满的时候，生产者无法继续往缓冲区中添加数据，就会睡眠，当缓冲区不满的时候再唤醒。
>
> **出现的问题：**为了时刻监视缓冲区大小，需要有一个变量count来映射。但是这个变量就是映射的共享内存，生产者消费者都可以修改这个变量。由于这里面对count没有加以限制会出现竞争。比如当缓冲区为空时，count=0，消费者读取到count为0，这个时候cpu的

> 执行权限突然转移给了生产者。然后生产者发现count=0，就会马上生产一个数据放到缓冲区中，此时count=1，接着会发送一个wakeup信号给消费者，因为由于之前count=0，生产者以为消费者进入了阻塞状态。但事实上我们知道消费者还没有进入阻塞状态，因此生产者的这个wakeup信号会丢失。接着CPU执行权限有转移到消费者这里，消费者查看自己的进程表项中存储的信息发现count=0然后进入阻塞，永远不去取数据。这个时候生产者迟早会把缓冲区填满，然后生产者也会进入阻塞，然后两个进程都在阻塞下去，出现了问题。

这个模型涉及到了两种关系：

1. 生产者和消费者之间的同步关系。当缓冲区满时，必须等消费者先行动。当缓冲区空时，必须等生产者先行动。
2. 生产者和消费者之间的互斥关系。就是刚才提到的问题，对缓冲区的操作必须与其他进程互斥才行。不然很容易死锁。

解决方案：

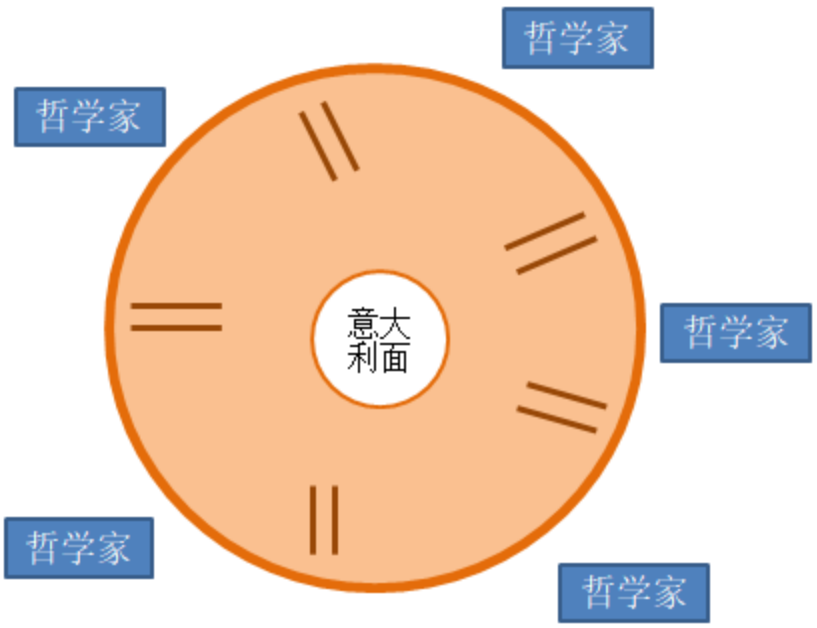
```
pthread_mutex_t mutex;
pthread_cond_t producter;
pthread_cond_t consumer;
count = pool.size()

producer(){
    while(1){
      pthread_mutex_lock(&mutex);
      pool++;
      while(pool == full){
          pthread_cond_wait(&producter, &mutex);
      }
      pthread_cond_signal(&consumer, &mutex);
      pthread_mutex_unlock(&mutex);
    }
}
consumer(){
    while(1){
      pthread_mutex_lock(&mutex);
      pool--;
      while(pool == 0){
          pthread_cond_wait(&consumer, &mutex);
      }
      pthread_cond_signal(&producter, &mutex);
      pthread_mutex_unlock(&mutex);
    }
}
```

注意不能先执行P(mutex)再执行P(empty)，这样生产者阻塞了消费者也阻塞了。

哲学家就餐问题

> **问题描述：**如下图所示



哲学家只有两件事情，吃饭和思考。每个哲学家的两边都有叉子，只有当拥有左面和右面的叉子时候才能吃饭，吃完饭后放下叉子思考。

出现问题：一下两种情况会产生死锁。第一种情况是五位哲学家同时拿起左边的叉子，就没有人能够拿到右面的叉子造成死锁。第二种情况是拿起左边叉子，然后查看右边有没有叉子，没有就放下左边叉子，有就拿起右边叉子吃饭。这样也会造成死锁，五个人还是同时拿左边的叉子，然后放下这样会永远重复。第三种情况就是哲学家拿起左边叉子然后随机等待一段时间，有右边叉子就拿没有就放下左边叉子。但是当对于核电站中的安全系统时候，随机数不可靠。

整个模型中有五个进程，互相之间是互斥的关系。解决方法是：

1. 同时拿到两个筷子
2. 对每个哲学家的动作制定规则，避免饥饿或者死锁。

解决方法是设置一个互斥信号量组用于对进程之间的互斥访问chopstick=[1,1,1,1,1]

```
semaphore chopstick[5] = {1,1,1,1,1}; //定义信号量数组mutex[5],并初始化
Pi(){  //i号哲学家的进程
do{
  P (chopstick[i] ) ; //取左边筷子
  P (chopstick[(i+1) %5] ) ;   //取右边篌子
  eat;  //进餐
  V(chopstick[i]) ; //放回左边筷子
  V(chopstick[(i+l)%5]);  //放回右边筷子
  think;  //思考
} while (1);
}
```

这个算法存在以下问题：**同时就餐拿起左边的筷子会产生死锁**

改进：

对哲学家进程施加一些限制条件，比如至多允许四个哲学家同时进餐;仅当一个哲学家左右两边的筷子都可用时才允许他抓起筷子

```
semaphore chopstick[5] = {1,1,1,1,1}; //初始化信号量
semaphore mutex=l;  //设置取筷子的信号量
Pi(){ //i号哲学家的进程
do{
  P (mutex) ; //在取筷子前获得互斥量
  P (chopstick [i]) ; //取左边筷子
  P (chopstick[ (i+1) %5]) ;  //取右边筷子
  V (mutex) ; //释放取筷子的信号量
  eat;  //进餐
  V(chopstick[i] ) ;  //放回左边筷子
  V(chopstick[ (i+l)%5]) ;  //放回右边筷子
  think;  // 思考
}while(1);
}
```

读写问题

问题描述：有读者和写者两组并发进程，共享一个文件，当两个或以上的读进程同时访问

> 共享数据时不会产生副作用，但是如果有写进程在修改数据库的时候不能有其他读进程或写进程访问数据库。因此要求：①允许多个读者可以同时对文件执行读操作；②只允许一个写者往文件中写信息；③任一写者在完成写操作之前不允许其他读者或写者工作；④写者执行写操作前，应让已有的读者和写者全部退出。

问题中的三个关系：

1. 读者和写者是互斥关系
2. 写者和写者是互斥关系
3. 读者和读者不存在互斥关系

> **分析**：
>
> 1. 写者好分析，他和任意进程互斥，用互斥信号量操作就行
> 2. 读者问题比较复杂。读者必须实现和写者的互斥，而且要实现和其他读者的同步。
>
> 因此用到一个计数器，判断当前有多少读者在读文件。
>
> 当有读者的时候不能写入文件，当没有读者的时候写者才会写入文件。
>
> 同时计数器也是公共内存，对计数器的访问也应该是互斥的

```
int count=0;  //用于记录当前的读者数量
semaphore mutex=1;  //用于保护更新count变量时的互斥
semaphore rw=1;  //用于保证读者和写者互斥地访问文件

//可以看到writer比较简单
writer () {  //写者进程
    while (1){
        P(rw); // 互斥访问共享文件
        Writing;  //写入
        V(rw) ;  //释放共享文件
    }
}

reader () {  // 读者进程
    while(1){
        P (mutex) ;  //互斥访问count变量
        if (count==0)  //当第一个读进程读共享文件时
            P(rw);  //阻止写进程写
        count++;  //读者计数器加1
        V (mutex) ;  //释放互斥变量count

        reading;  //读取

        P (mutex) ;  //互斥访问count变量
        count--; //读者计数器减1
        if (count==0)  //当最后一个读进程读完共享文件
            V(rw) ;  //允许写进程写
        V (mutex) ;  //释放互斥变量 count
    }
}
```

CPU调度算法（进程调度算法）

进程调度算法也称 CPU 调度算法，毕竟进程是由 CPU 调度的。

调度分为两大类：抢占式调度和非抢占式调度。抢占式调度说明程序正在运行时可以被打断，把CPU让给其他进程。非抢占式调度表示一个进程正在运行当进程完成或者阻塞的时候把CPU让

出来。

调度的的话有很多进程都会等待这调度，那么就有一些调度算法：

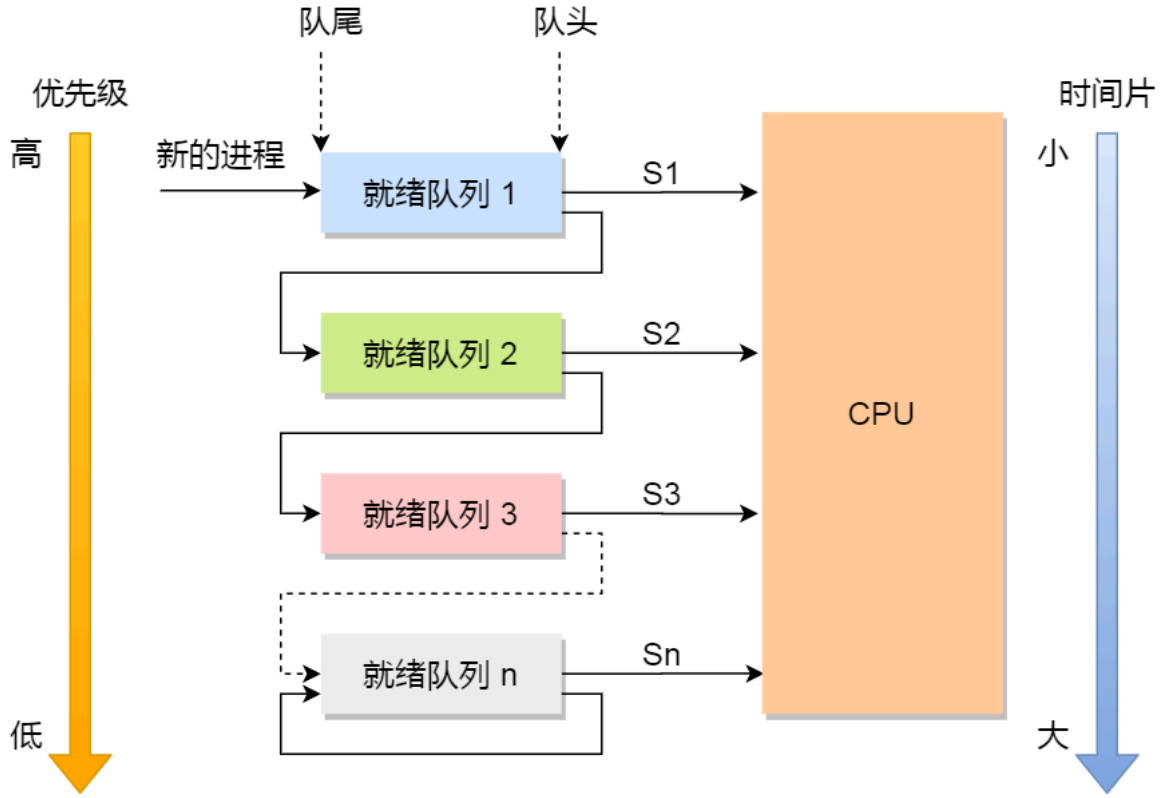
- **先来先服务调度算法**
  每次从就绪队列选择最先进入队列的进程，然后一直运行，直到进程退出或被阻塞，才会继续从队列中选择第一个进程接着运行。
  这似乎很公平，但是当一个长作业先运行了，那么后面的短作业等待的时间就会很长，不利于短作业。
  FCFS 对长作业有利，适用于 CPU 繁忙型作业的系统，而不适用于 I/O 繁忙型作业的系统。
- **最短作业优先调度算法**
  优先选择运行时间最短的进程来运行，这有助于提高系统的吞吐量。
  这显然对长作业不利，很容易造成一种极端现象。比如，一个长作业在就绪队列等待运行，而这个就绪队列有非常多的短作业，那么就会使得长作业不断的往后推，周转时间变长，致使长作业长期不会被运行。
- **高响应比优先调度算法**
  前面的「先来先服务调度算法」和「最短作业优先调度算法」都没有很好的权衡短作业和长作业。
  每次进行进程调度时，先计算「响应比优先级」，然后把「响应比优先级」最高的进程投入运行，「响应比优先级」的计算公式：
  优先权=（等待时间+服务时间）/ 服务时间。
- **时间片轮转调度算法**
  每个进程被分配一个时间段，称为时间片（*Quantum*），即允许该进程在该时间段中运行。
  如果时间片用完，进程还在运行，那么将会把此进程从 CPU 释放出来，并把 CPU 分配另外一个进程；如果该进程在时间片结束前阻塞或结束，则 CPU 立即进行切换；通常时间片设为 `20ms~50ms` 通常是一个比较合理的折中值。
- **最高优先级调度算法**
  对于多用户计算机系统就有不同的看法了，它们希望调度是有优先级的，即希望调度程序能从就绪队列中选择最高优先级的进程进行运行，这称为最高优先级（Highest Priority First，HPF）调度算法。
  非抢占式：当就绪队列中出现优先级高的进程，运行完当前进程，再选择优先级高的进程。
  抢占式：当就绪队列中出现优先级高的进程，当前进程挂起，调度优先级高的进程运行。

- **多级反馈队列调度算法**

  多级反馈队列（Multilevel Feedback Queue）调度算法是「时间片轮转算法」和「最高优先级算法」的综合和发展。

  多级表示有多个队列，每个队列优先级从高到低，同时优先级越高时间片越短。

  反馈表示如果有新的进程加入优先级高的队列时，立刻停止当前正在运行的进程，转而去运行优先级高的队列；



（时间片：S1 < S2 < S3）

其工作流程如下：

i. 设置了多个队列，赋予每个队列不同的优先级，每个队列优先级从高到低，同时优先级越高时间片越短；

ii. 新的进程会被放入到第一级队列的末尾，按先来先服务的原则排队等待被调度，如果在第一级队列规定的时间片没运行完成，则将其转入到第二级队列的末尾，以此类推，直至完成；

iii. 当较高优先级的队列为空，才调度较低优先级的队列中的进程运行。如果进程运行时，有新进程进入较高优先级的队列，则停止当前运行的进程并将其移入到原队列末尾，接着让较高优先级的进程运行；

操作系统是如何做到进程阻塞的？

- 什么是阻塞
  正在运行的进程由于提出系统服务请求（如I/O操作），但因为某种原因未得到操作系统的立即响应，或者需要从其他合作进程获得的数据尚未到达等原因，该进程只能调用阻塞原语把自己阻塞，等待相应的事件出现后才被唤醒。
  进程并不总是可以立即运行的，一方面是 CPU 资源有限，另一方面则是进程时常需要等待外部事件的发生，例如 I/O 事件、定时器事件等。
- 如何做到进程阻塞
  操作系统为了支持多任务，实现了进程调度的功能，会把进程分为“运行”和“等待”等几种状态。操作系统会分时执行各个运行状态的进程，由于速度很快，看上去就像是同时执行多个任务，这里用到了时间片轮转技术，如果在时间片结束时进程还在运行，则CPU使用权将被剥夺并分配给另一个进程。**如果进程在时间片结束前阻塞或结束，则CPU当即进行切换。**
  当CPU从一个进程跑到了别的进程之后，肯定还需要跑回来，因此就有工作队列和等待队列。
  假如现在进程 A 里跑的程序有一个对象执行了某个方法将当前进程阻塞了，内核会立刻将进程A从工作队列中移除，同时创建等待队列，并新建一个引用指向进程A。这时进程A被排在了工作队列之外，不受系统调度了，这就是我们常说的被操作系统“挂起”。这也提现了阻塞和挂起的关系。阻塞是人为安排的，让你程序走到这里阻塞。而阻塞的实现方式是系统将进程挂起。
  当这个对象受到某种“刺激”（某事件触发）之后， 操作系统将该对象等待队列上的进程重新放回到工作队列上就绪，等待时间片轮转到该进程。所以，操作系统不会去尝试运行被阻塞的进程，而是由对象去等待某种“刺激”，喜欢被动。
  （要点：时间片轮转，工作队列和等待队列）
- 进程阻塞不消耗CPU资源，但是会消耗系统资源。因此系统资源不仅包含CPU，还有内存、磁盘I/O等等

线程间通信的方式

线程间很多资源都是共享的，所以线程间没有像进程间那样有许多数据交换机制。**线程间通信主要目的是为了线程同步。**

- **锁机制**

    - 互斥锁。确保同一时间内只有一个线程能访问共享资源。当资源被占用时其他试图加锁的线程会进入阻塞状态。当锁释放后，哪个线程能上锁取决于内核调度。
    - 读写锁。当以写模式加锁的时候，任何其他线程不论以何种方式加锁都会处以阻塞状态。当以读模式加锁时，读状态不阻塞，但是写状态阻塞。“读模式共享，写模式互斥”
    - 自旋锁。上锁受阻时线程不阻塞而是在循环中轮询查看能否获得该锁，没有线程的切换因而没有切换开销，不过对CPU的霸占会导致CPU资源的浪费。
- **posix信号量机制**
  信号量本质上是一个计数器，可以有PV操作，来控制多个进程或者线程对共享资源的访问。
  信号量API有两组，第一组就是System V IPC信号量用于进程间通信的，另外一组就是POSIX信号量，信号量原理都是一样的
- **条件变量**
  条件变量提供了线程间的通知机制：当某个共享数据到达某个值的时候，唤醒等待这个共享数据的线程。
  条件变量要结合互斥锁使用，如下图代码：

```
pthread_mutex_lock(&mutex);
    while(条件为假)
        pthread_cond_wait(&cond,&mutex);
    执行某种操作
pthread_mutex_unlock(&mutex);
```

  也就是说，一个线程要等到临界区的共享数据达到某种状态时再进行某种操作，而这个状态的成立，则是由另外一个进程/线程来完成后发送信号来通知的。

线程是如何实现的？

这个看《操作系统》总结的也有

实现线程主要有三种方式：（1）使用内核线程实现，（2）使用用户线程实现（3）使用用户线程加轻量级进程混合实现。

- 内核线程实现

内核线程（Kernel-Level Thread,KLT）就是直接由操作系统内核支持的线程，内核通过操纵调度器对线程进行调度，并负责将线程的任务映射到各个处理器上。每个内核线程可以视为内核的一个分身，这种操作系统就有能力同时处理多件事情，支持多线程的内核就叫做多线程内核。

程序一般不会直接去使用内核线程，而是去使用内核线程的一种高级接口：轻量级进程（Light Weight Process，LWP），轻量级进程就是我们通常意义上所讲的线程，由于每个轻量级进程都由一个内核线程支持，因此只有先支持内核线程，才能有轻量级进程。

- 用户线程实现

  从广义上讲，一个线程只要不是内核线程，就可以认为是用户线程（User Thread，UT）。而狭义上的用户线程指的是完全建立在用户空间的线程库上，系统内核不能感知线程存在的实现。用户线程的建立、同步、销毁和调度完全在用户态中完成，不需要内核的帮助。使用用户线程的优势在于不需要系统内核支援，劣势也在于没有系统内核的支援，所有的线程操作都需要用户程序自己处理。线程的创建、切换和调度都是需要考虑的问题，因此比较难。
- 用户线程加轻量级进程混合实现

  线程除了依赖内核线程实现和完全由用户程序自己实现之外，还有一种将内核线程与用户线程一起使用的实现方式。在这种混合实现下，既存在用户线程，也存在轻量级进程。

线程调度

- 协同式调度

  使用协同式调度的多线程系统，线程的执行时间由线程本身来控制，线程把自己的工作执行完了之后，要主动通知系统切换到另一个线程上。协同式多线程的最大好处是实现简单，不会有线程同步问题。缺点是线程执行时间不可控制，甚至如果一个线程编写有问题，一直不告知系统进行线程切换，那么程序就会一直阻塞在那里。
- 抢占式调度

  使用抢占式调度的多线程系统，每个线程将由系统来分配执行时间，线程的切换不由线程本身来决定。在这种实现线程调度的方式下，线程的执行时间是系统可控的，也不会有一个线程导致整个进程阻塞的问题，Java使用的线程调度方式就是抢占式调度。

进程阻塞为什么不占用cpu资源？

因为阻塞了得不到cpu，怎么占用CPU资源

linux内核的同步方式

首先要明确，同步和互斥是计算机系统中，用于控制进程对某些特定资源访问的机制。同步指的是进程按照一定的顺序和规则访问资源，互斥则是控制资源某一时刻只能由一个进程访问。这样看来互斥是同步的一种情况。

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，因此内核其实象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。在主流的Linux内核中包含了几乎所有现代的操作系统具有的同步机制，**这些同步机制包括：原子操作、信号量（semaphore）、读写信号量 （rw_semaphore）、spinlock、BKL(Big Kernel Lock)、rwlock、brlock（只包含在2.4内核中）、RCU（只包含在2.6内核中）和seqlock（只包含在2.6内核中）。**

> linux会因为三种机制而产生并发，需要进行同步。
>
> 1. **中断。**中断就是操作系统随时可以打断正在执行的代码。当进程访问某个临界资源发生中断，会进入中断处理程序。中断处理程序也可能访问该临界资源。
> 2. **内核抢占式调度。**内核中正在执行的任务被另外一个任务抢占。
> 3. **多处理器并发。**每个处理器都可以调度一个进程，多个进程可能会造成并发。

- **禁用中断**
  对于单处理器不可抢占系统来说，系统并发源主要是中断处理。因此在进行临界资源访问时，进行禁用/使能中断即可以达到消除异步并发源的目的。
- **原子操作**
  原子操作保证指令在运行时候不会被任何事物或者事件打断，把读和写的行为包含在一步中执行，避免竞争。
- **内存屏障(memory barrier)**
  程序在运行时内存实际的访问顺序和程序代码编写的访问顺序不一定一致，这就是

内存乱序访问。内存乱序访问行为出现的理由是为了提升程序运行时的性能。内存乱序访问主要发生在两个阶段：

i. 编译时，编译器优化导致内存乱序访问（指令重排）
ii. 运行时， 多 CPU 间交互引起内存乱序访问

- **自旋锁(spin lock)**

  对于复杂的操作原子操作是不能的。比如从一种数据结构中读取数据，写入到另一种数据结构中。自旋锁是linux内核中最常见的一种锁。自旋锁只能被一个可执行线程所拥有，如果有线程要抢占一个已经被占有的自旋锁，则其会一直进行循环来等待锁重新可用。当锁被释放后，请求锁的执行线程便能立刻得到它。其实就是忙等自旋锁只适用于那些在临界区停留很短时间的加锁操作。因为线程在等待锁期间会一直占据处理器，如果长时间等待锁会导致处理器效率降低。而如果线程占用锁只需要短暂的执行一下，那么使用自旋锁更优，因为不需要进行上下文的切换。

- **信号量**

  可以理解为一种“睡眠锁”

  自旋锁是“忙等”机制，在临界资源被锁定的时间很短的情况下很有效。但是在临界资源被持有时间很长或者不确定的情况下，忙等机制则会浪费很多宝贵的处理器时间。

  而信号量机制在进程无法获取到临界资源的情况下，立即释放处理器的使用权，并睡眠在所访问的临界资源上对应的等待队列上；在临界资源被释放时，再唤醒阻塞在该临界资源上的进程。信号量适用于长时间占用锁的情形。

- **读-写自旋锁**

  具体就是在读写场景中了，为读和写提供不同的锁机制。

  当一个或者多个读任务时可以并发的持有读锁，但是写锁只能被一个人物所持有的。即对写者共享对读者排斥

- **读写信号量**

  这个和读写自旋锁的思想是一样的。

- **mutex体制**

  也是一种睡眠锁，是实现互斥的特定睡眠锁。是一种互斥信号

  使用mutex有很多限制，不像信号量那样想用就用

  i. 任何时刻只有一个任务可以持有mutex，引用计数只能是1
  ii. 给mutex上锁必须给其解锁，严格点就是必须在同一上下文中上锁和解锁
  iii. 持有mutex的进程不能退出
  iv. 等等

- **完成变量(completion variable)**

内核中一个任务需要发出信号通知另一个任务发生了某个特定事件，利用完成变量使得两个任务实现同步

- **BLK：大内核锁**
  属于混沌时期的产物，是一个全局的自旋锁
  这个东西是早期linux不支持线程的时候用的，和自旋锁差不多的思想。
  现在一般不鼓励使用这个了
- **顺序锁**
  这种锁用于读写共享数据。
  实现这个机制主要依靠序列计数器，当写数据时，会得到一个锁，序列值会增加。
  在读取数据前后这两个时间内，序列值都会被读取
  如果读取的序列号值相同，表明读操作在进行的过程中没有被写操作打断，
- **关闭内核抢占**
  内核是抢占性的，因此内核中的进程在任何时候都可能停下来以便让另一个具有更高优先级的进程运行。但是一个任务和被抢占的任务可能在同一个临界区运行。为了避免上述情况，当使用自旋锁时这个区域被标记为非抢占的，一个自旋锁被持有表示内核不能抢占调度。
  但是在一些情况下，就算不用自旋锁，也要关闭内核抢占。
  比如，对于只有一个处理器能够访问到数据，原理上是没有必要加自旋锁的，因为在任何时刻数据的访问者永远只有一位。但是，如果内核抢占没有关闭，则可能一个新调度的任务就可能访问同一个变量。
  所以这时候害怕的不是多个任务访问同问同一个变量，而是一个任务的访问还没有完成就转到了另一个任务。
- **RCU(Read, Copy, Update)**
  RCU(Read, Copy, Update)是一组Linux内核API，实现了一种同步机制，允许多个读者与写者并发操作而不需要任何锁，这种同步机制可以用于保护通过指针进行访问的数据。比较适合用在读操作很多而写操作极少的情况，可以用来替代读写锁。
  一个典型场景就是内核路由表，路由表的更新是由外部触发的，外部环境的延迟远比内核更新延迟高，在内核更新路由表前实际已经向旧路径转发了很多数据包，RCU读者按照旧路径再多转发几个数据包是完全可以接受的，而且由于RCU的无锁特性，实际上相比有锁的同步机制，内核可以更早生效新的路由表。路由表这个场景，以系统内部短时间的不一致为代价，降低了系统内部与外部世界的不一致时间，同时降低了读者成本。

  > Q: **为什么只能保护通过指针访问的数据？**
  >
  > A: 任何CPU架构下的Linux都可以保证指针操作的原子性，这是无锁并发的前提。也就是说，假设CPU A在修改指针，无论何时CPU B读取该指针，都可以

> 保证读取到的数据要么是旧的值，要么是新的值，绝不会是混合新旧值不同bit位的无意义值。因此使用RCU对更复杂的数据结构的保护都是基于对指向该数据结构的指针的保护。

死锁

> 概念：由于操作系统会产生并发，那会产生一个问题，就是多个进程因为争夺资源而互相陷入等待。

- **死锁产生的原因**
    i. 资源分配不当
    ii. 进程运行的顺序不合理
- **产生死锁的必要条件****
    i. 互斥。某个资源只允许一个进程访问，如果已经有进程访问该资源，则其他进程就不能访问，直到该进程访问结束。
    ii. 占有的同时等待。一个进程占有其他资源的同时，还有资源未得到，需要其他进程释放该资源。
    iii. 不可抢占。别的进程已经占有某资源，自己不能去抢。
    iv. 循环等待。存在一个循环，每个进程都需要下一个进程的资源。

  以上四个条件均满足必然会造成死锁。会导致cpu吞吐量下降，死锁会浪费系统资源造成计算机性能下降。
- **避免死锁的方法**

  我们要尽量避免四个条件同时产生，因此就要破坏。由于互斥条件是必须的，必须要保证的，因此从后面三条下手。
    i. 破坏“占有且等待条件”。
    ①所有进程在开始运行之前，一次性申请到所有所需要的资源。
    ②进程用完的资源释放掉，然后再去请求新的资源，提高利用率。
    ii. 破坏“不可抢占”条件。
    当进程提出在得到一些资源时候不被满足的情况下，必须释放自己已经保存的资源。
    iii. 破坏“循环等待”。
    实现资源有序分配策略，所有进程申请资源必须按照顺序执行。
    iv. 银行家算法
    参考链接

银行家算法的实质就是即每当进程提出资源请求且系统的资源能够满足该请求时，系统将判断满足此次资源请求后系统状态是否安全（安全状态是非死锁状态，而不安全状态并不一定是死锁状态。即系统处于安全状态一定可以避免死锁，而系统处于不安全状态则仅仅可能进入死锁状态。），如果判断结果为安全，则给该进程分配资源，否则不分配资源，申请资源的进程将阻塞。

银行家算法的执行有个前提条件，即要求进程预先提出自己的最大资源请求，并假设系统拥有固定的资源总量。

① 可用资源向量 available，它记录系统中各类资源的当前可利用数目。

② 最大需求矩阵 max，它记录每个进程对各类资源的最大需求量。

③ 分配矩阵 allocation，它记录每个进程对各类资源当前的占有量。

④ 需求矩阵 need，它记录每个进程对各类资源尚需要的数目，等于最大需求矩阵与分配矩阵的差。

⑤ 请求向量 request，它记录某个进程当前对各类资源的申请量，是银行家算法的入口参数。

银行家算法的描述如下（设进程 P_i 向系统提出 $request_i$ 资源请求）：

① 若 $request_i > need_i$，则进程 P_i 出错；

② 若 $request_i > available$，则进程 P_i 阻塞；

③ 系统试着把资源分配给进程 P_i，并对相应数据结构作如下修改：

$available - request_i$；$allocation_i + request_i$；$need_i - request_i$；

④ 系统执行安全性检测子算法，以判断试分配后系统状态是否安全；

⑤ 若第④步返回逻辑真值，即“安全”，则完成本次分配，返回；

⑥ 否则，撤销此次（即第③步中的）试分配，进程 P_i 阻塞。

虚拟内存

为什么需要虚拟内存？内存技术发展历程

- 1、无存储器抽象

  早期计算机没有存储器抽象，每一个程序都是直接访问物理内存。这种情况下在内存中同时运行两个程序是不可能的，因为如果第一个程序比如在内存2000的位置写入一个新的值，那么第二个程序运行的时候就会擦掉第一个程序的值。

  但是即使没有抽象概念，运行两个程序也是可能的。。。

  操作系统把当前内存中的内容保存在磁盘文件中，然后把下一个程序读入到内存中再运行就行，即某个时间内内存只有一个程序在运行就不会发生冲突。（最早期的交换的概念）

  还有就是借助硬件来实现并发运行多个程序，比如IBM360中内存被划分为2KB的块，每个块有一个4位的保护键，保护键存储在CPU的特殊寄存器中。一个运行中

的程序如果访问的保护键与操作系统中保存的不相符，则硬件会捕获到该异常，由于只有操作系统可以修改保护键，因此这样就可以防止用户进程之间、用户进程和操作系统间的干扰。

- 2、地址空间（存储抽象）

  > 把物理地址暴露给进程会带来几个问题：
  >
  > i. 如果用户进程可以寻址内存中的每一个字节，就可以很容易的破坏操作系统，从而时电脑出故障。
  > ii. 想要同时运行多个程序比较困难

  在之前所说的解决多个程序在内存中互不影响的办法主要有两个：保护和重定位，也就是说并不是所有内存你都能访问，因此出现了地址空间的概念。
  地址空间是一个进程可用于寻址内存的一套地址的集合。每个进程都有自己的地址空间，并且这个地址空间独立于其他进程的地址空间
  最开始的地址空间的解决办法是动态重定位，就是简单地把每个进程的地址空间映射到物理内存的不同部分。但是也有问题，即每个进程都有内存，如果同时运行多个进程，那所需要的内存是很庞大的，这个ram数远远超过存储器支持的。比如在Linux和Windows中，计算机完成引导后会启动50-100个进程，那你需要的空间可大了去了。
  处理上述内存超载问题有两个办法，就是交换技术和虚拟内存。
  交换技术就是把一个进程完整的调入内存，使得该进程完整运行一段时间后在调入磁盘。但是这样的话，你把一个完整的进程调入进去，可能很大程度只会使用一部分内存，这样是不划算的。
  所以有了虚拟内存，每个进程的一部分调入内存。

- 3、虚拟内存
  这个要重点说

什么是虚拟内存

根据上面所说，虚拟内存就是来解决“**程序大于内存的问题**”，即如何不把程序内存全部装进去。

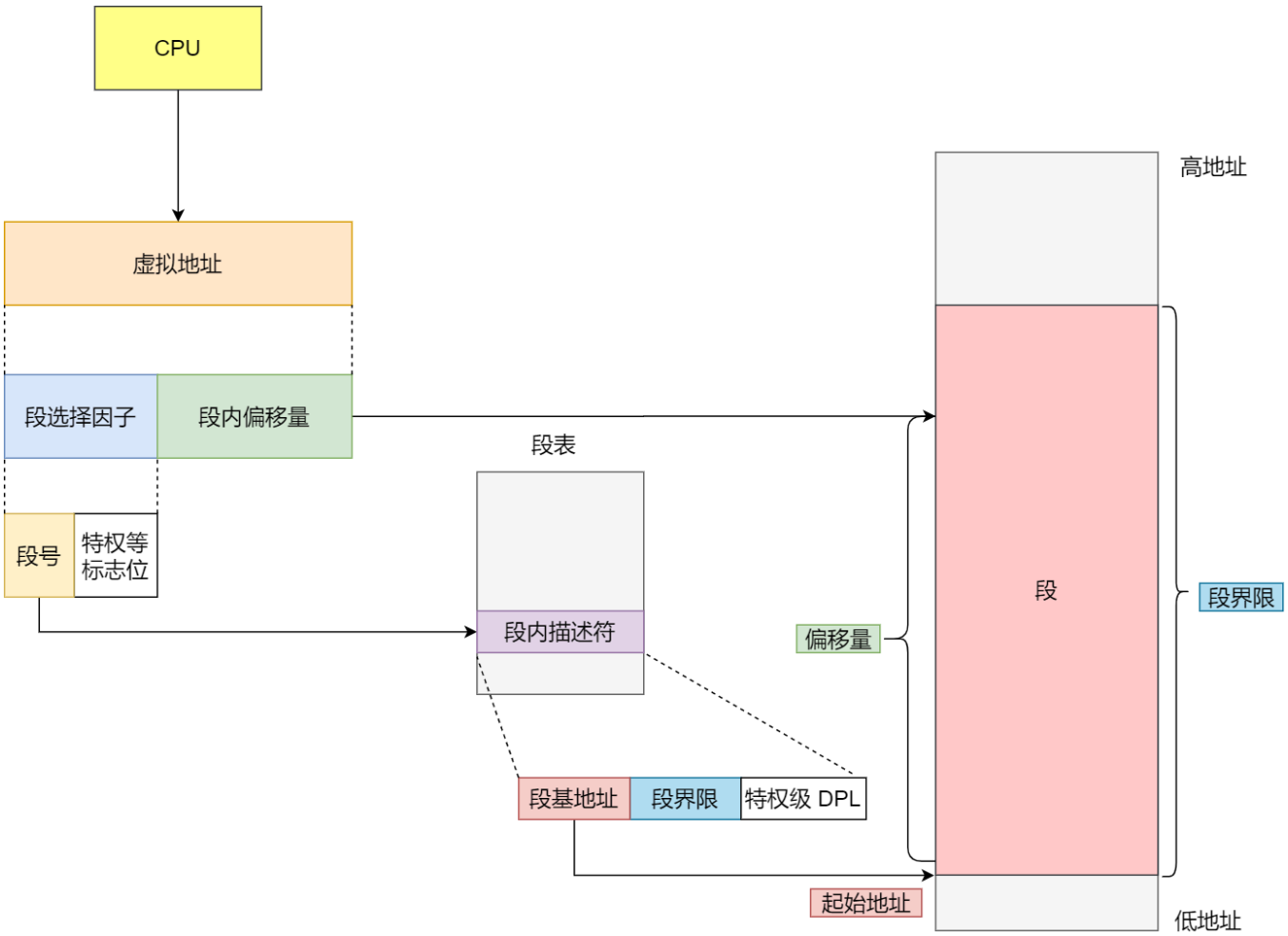
虚拟内存的基本思想是每个程序都拥有自己的地址空间，这些空间被分割成多个块儿。每一块儿被称作一页或者页面。每一个页面有连续的地址范围。这些页面被映射到物理内存，但是并不是一个程序的所有的页面都必须在内存中才能运行。当程序引用到一部分在物理内存中的地址空间时，由硬件立刻执行映射。当程序引用到一部分不在物理内存中的地址空间时，由操作系统负责将缺失部分装入物理内存并重新执行指令。

虚拟内存 使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。与没有使用虚拟内存技术的系统相比，使用这种技术的系统使得大型程序的编写变得更容易，对真正的物理内存（例如RAM）的使用也更有效率。

第一阶段：分段

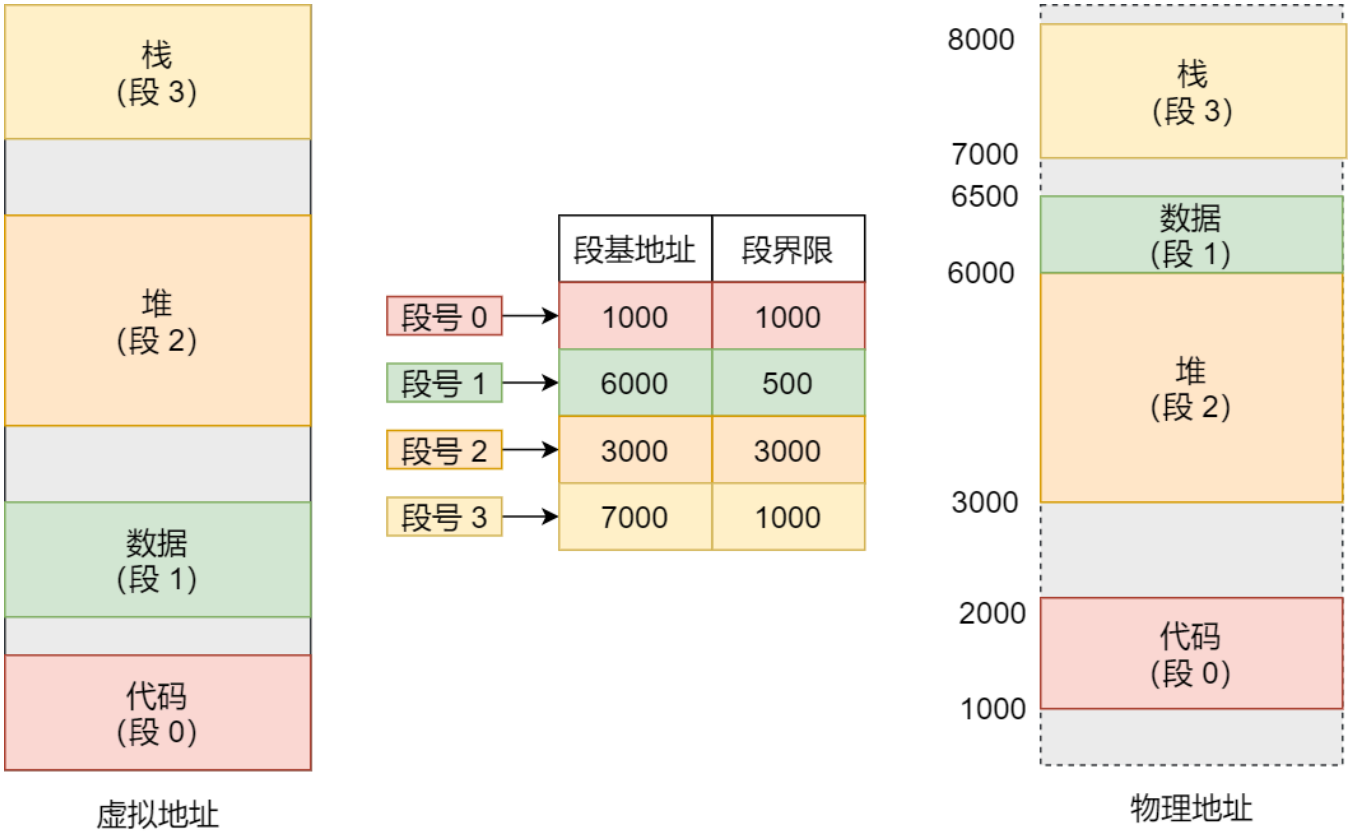
程序是由若干个逻辑分段组成的，如可由代码分段、数据分段、栈段、堆段组成。不同的段是有不同的属性的，所以就用分段的形式

分段机制下的虚拟地址由两部分组成，**段选择子**和**段内偏移量**。



- 段选择子就保存在段寄存器里面。段选择子里面最重要的是段号，用作段表的索引。段表里面保存的是这个段的基地址、段的界限和特权等级等。
- 虚拟地址中的段内偏移量应该位于 0 和段界限之间，如果段内偏移量是合法的，就将段基地址加上段内偏移量得到物理内存地址。

虚拟地址是通过段表与物理地址进行映射的，分段机制会把程序的虚拟地址分成 4 个段，每个段在段表中有一个项，在这一项找到段的基地址，再加上偏移量，于是就能找到物理内存中的地址，如下图：



如果要访问段 3 中偏移量 500 的虚拟地址，我们可以计算出物理地址为，段 3 基地址 7000 + 偏移量 500 = 7500。

分段的办法很好，解决了程序本身不需要关心具体的物理内存地址的问题，但它也有一些不足之处：

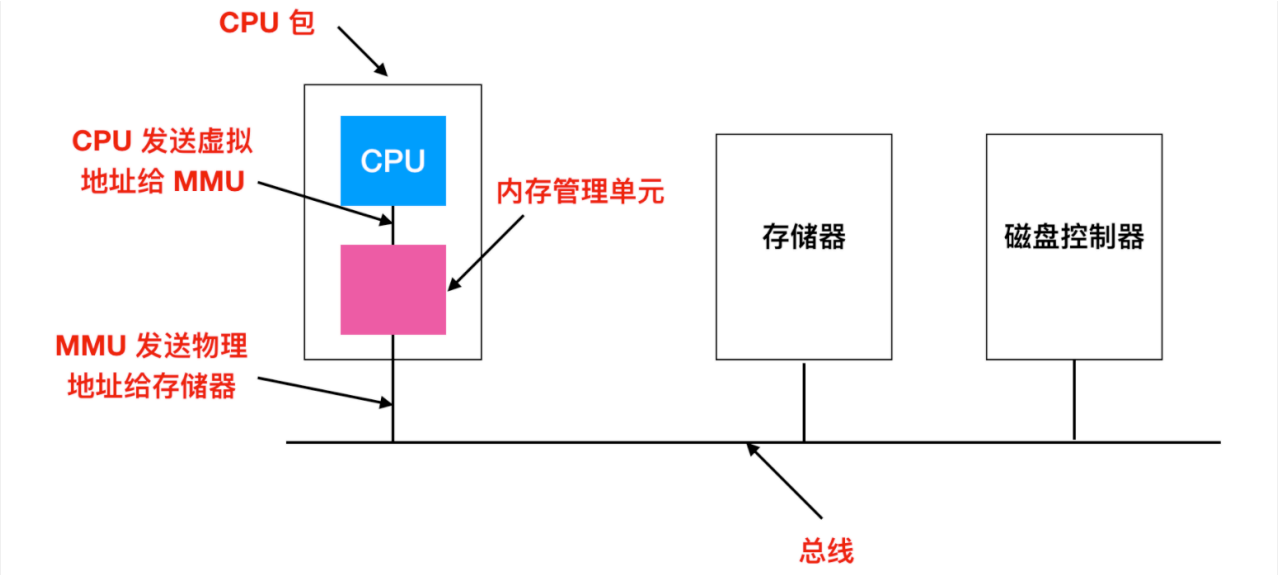
1. 第一个就是内存碎片的问题。
    - 外部内存碎片，也就是产生了多个不连续的小物理内存，导致新的程序无法被装载；
    - 内部内存碎片，程序所有的内存都被装载到了物理内存，但是这个程序有部分的内存可能并不是很常使用，这也会导致内存的浪费；
2. 第二个就是内存交换的效率低的问题。

解决外部内存碎片的问题就是内存交换。这个内存交换空间，在 Linux 系统里，也就是我们常看到的 Swap 空间，这块空间是从硬盘划分出来的，用于内存与硬盘的空间交换。对于多进程的系统来说，用分段的方式，内存碎片是很容易产生的，产生了内存碎片，那不得不重新 `Swap`

内存区域，这个过程会产生性能瓶颈。因为硬盘的访问速度要比内存慢太多了，每一次内存交换，我们都需要把一大段连续的内存数据写到硬盘上。所以，**如果内存交换的时候，交换的是一个占内存空间很大的程序，这样整个机器都会显得卡顿。**为了解决内存分段的内存碎片和内存交换效率低的问题，就出现了内存分页。

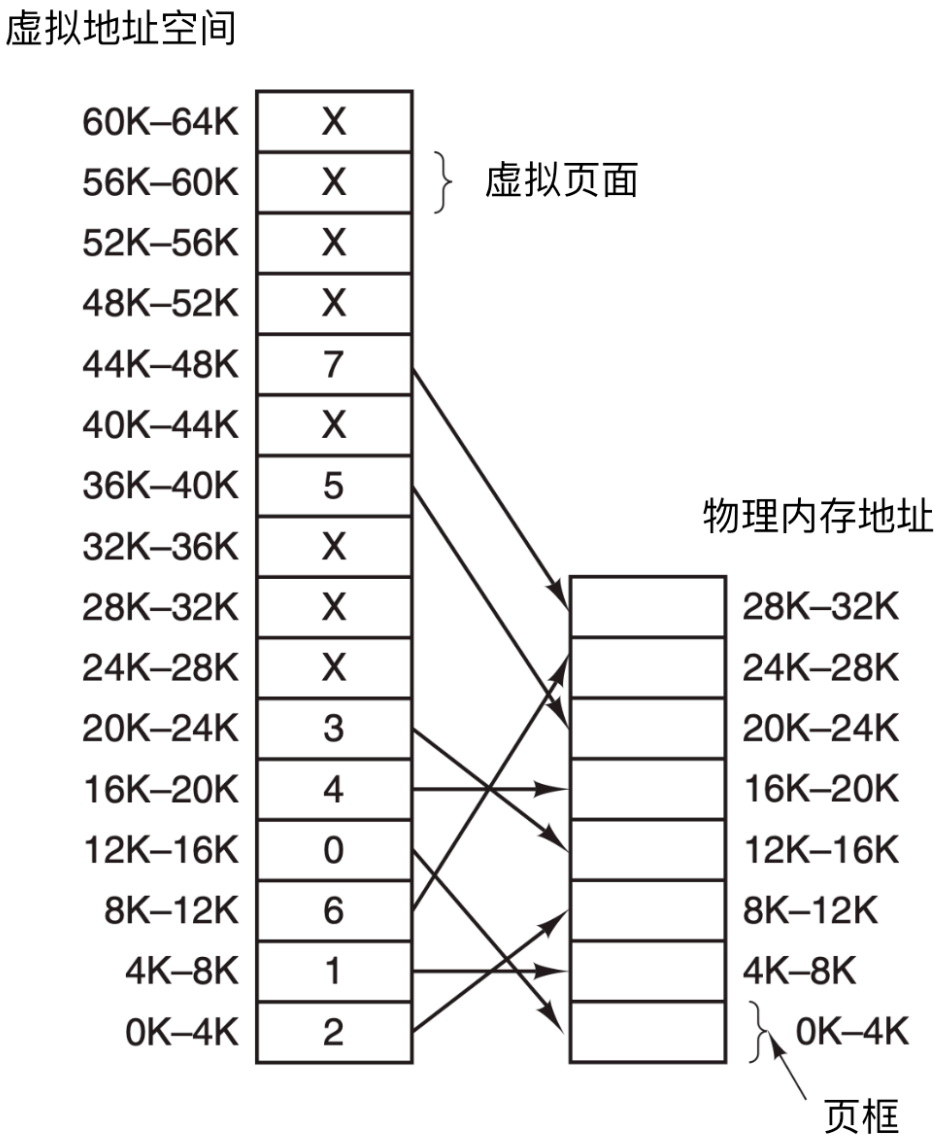
第二阶段：分页

在没有虚拟内存的计算机上，系统直接将虚拟地址送到内存总线上，读写操作使用具有同样地址的物理内存字。在使用虚拟内存的情况下，虚拟地址不是直接被送到内存总线上的，而是被送往内存管理单元MMU，然后MMU把虚拟地址映射为物理内存地址



在这里MMU是作为CPU芯片的一部分，通常就是这样的。

具体页表中虚拟地址与物理内存地址之间的映射关系如下：



上图可以看到虚拟地址被划分成多个页面组成的单位，而物理内存地址中对应的是页框的概念。页面和页框的大小通常是一样的，在这里是4KB作为例子，实际系统中可能是512KB-1GB。上图虚拟地址空间有64KB，物理内存有32KB，因此由16个虚拟页面和8个页框。请记住，ram和也磁盘之间的交换总是以整个页面为单位进行的。

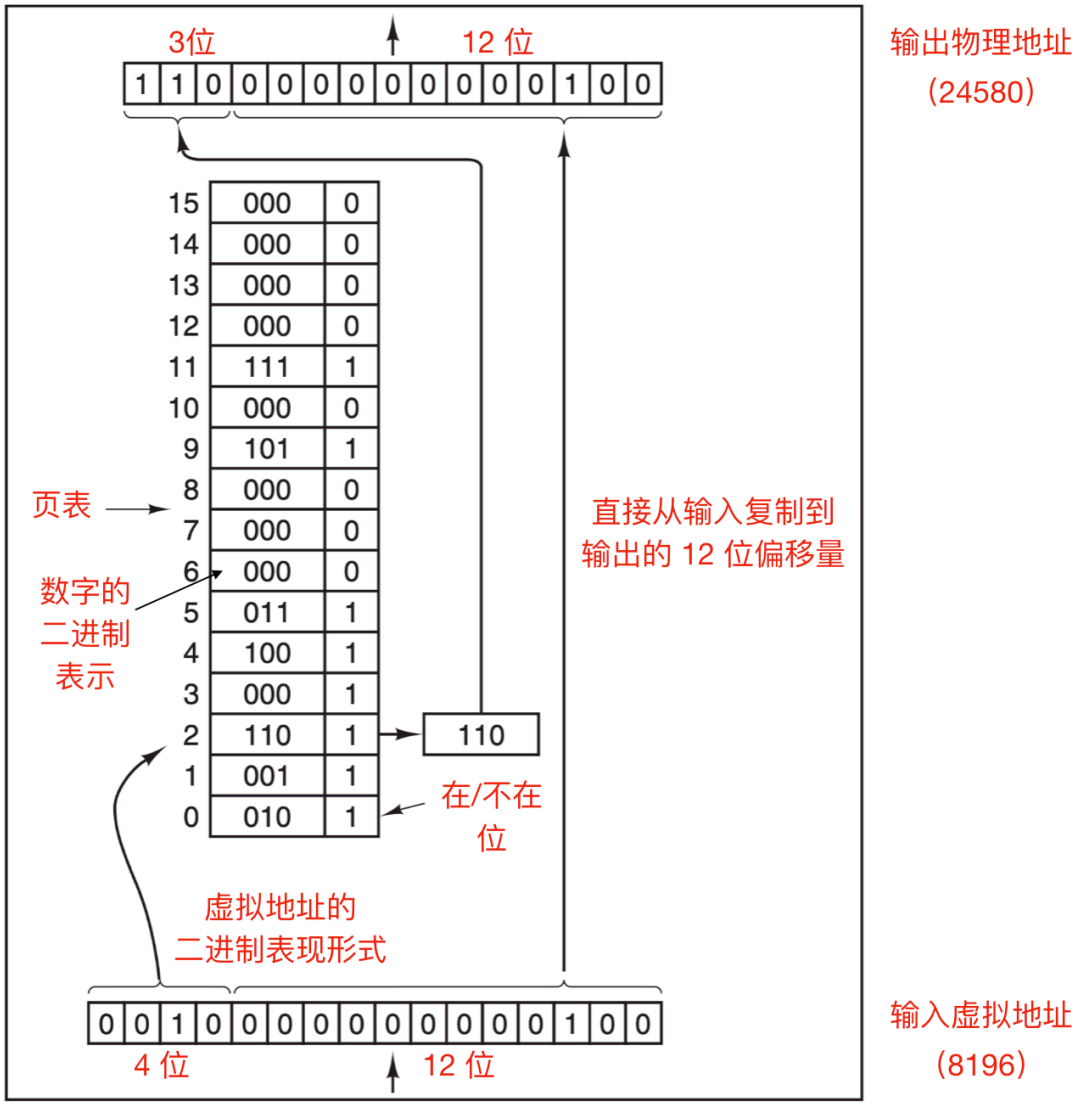
比如当程序访问地址0的时候，其实是访问虚拟地址，然后将虚拟地址0送到MMU，MMU根据映射关系发现虚拟地址0在页面0-4095这个页面上，然后根据映射找到实际物理内存地址是第二个页框，即8192，然后把地址8192送到总线上。内存对MMU是一无所知的，他只看到了一个读写8192的请求并执行。

当程序访问了一个未被映射的页面，即虚拟地址没有对应的页框索引。此时MMU注意到该页面没有映射，使CPU陷入到操作系统，即缺页中断或者缺页错误（page fault）。随后操作系统找到了一个很少使用的页框并把该页框内容写入磁盘，然后把需要访问的页面读到

刚才被回收的那个页框上，修改映射关系，重新启动引起中断的指令就好。

例如如果操作系统放弃页框1，即重新映射页框1，那么重新改变映射关系，将页面8装入物理地址4096（页框1），并且对MMU映射做两处修改：①由于页框1之前被页面1映射，因此要将页面1标记为未映射，使得以后对页面1的访问都将导致缺页中断。②把虚拟页面8的叉号改为1，表明虚拟页面8映射到页框1上去，当指令重新启动时候就会产生新的映射。

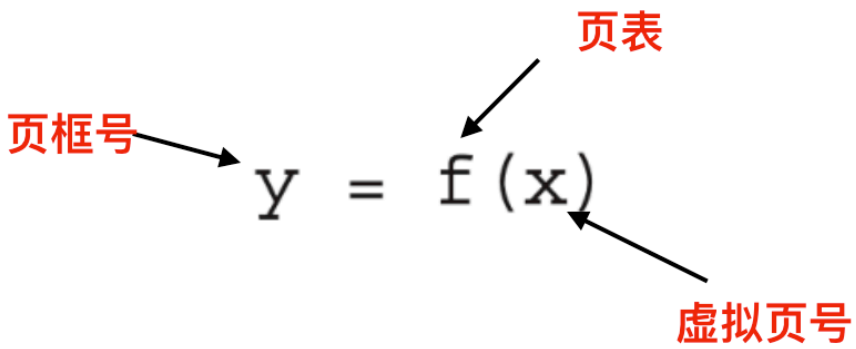
MMU的内部操作：



输入虚拟地址8196的二进制在最底下即0010000000000100，用MMU映射机进行映射，这16位虚拟地址被分解成4位的页号+12位的偏移量。4位页号表示16个页面，是页面的索引，12位的位偏移可以为一页内的全部4096个字节编址。

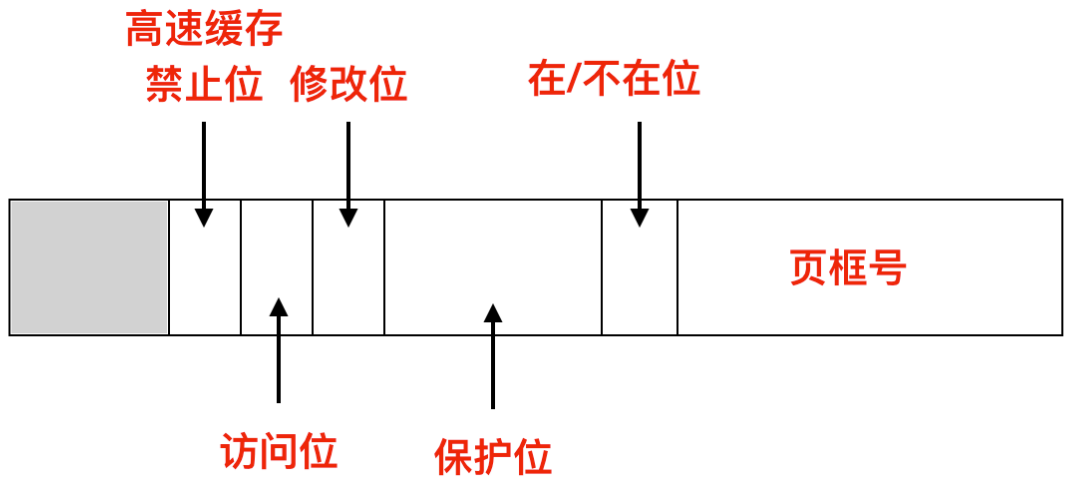
第二阶段：分表

虚拟地址到物理地址的映射可以概括如下：虚拟地址被分成虚拟页号+偏移量。虚拟页号是一个页表的索引，可以有该索引即页面号找到对应的页框号，然后将该页框号放到16位地址的前4位，替换掉虚拟页号，就形成了送往内存的地址。可以参考上面那个图中，两个二进制字符串的替换形式。如下图：



页表的目的是将虚拟页面映射为页框，从数学角度说页表是一个函数，输入参数是虚拟页号，输出结果是物理页框号。

页表的结构如下：



页表项中最重要的字段就是`页框号(Page frame number)`。毕竟，页表到页框最重要的一步操作就是要把此值映射过去。下一个比较重要的就是`在/不在`位，如果此位上的值是 1，那么页表项是有效的并且能够被`使用`。如果此值是 0 的话，则表示该页表项对应的虚拟页面`不在`内存中，访问该页面会引起一个`缺页异常(page fault)`。`保护位(Protection)`告诉我们哪一种访问是允许的，啥意思呢？最简单的表示形式是这个域只有一位，**0 表示可读可写，1 表示的是只读**。`修改位(Modified)`和`访问位(Referenced)`会跟踪页面的使用情况。当一个页面被写入时，硬件会自动的设置修改位。修改位在页面重新分配页框时很有用。如果一个页面已经被修改过

（即它是 `脏` 的），则必须把它写回磁盘。如果一个页面没有被修改过（即它是 `干净` 的），那么重新分配时这个页框会被直接丢弃，因为磁盘上的副本仍然是有效的。这个位有时也叫做 `脏位(dirty bit)`，因为它反映了页面的状态。`访问位(Referenced)` 在页面被访问时被设置，不管是读还是写。这个值能够帮助操作系统在发生缺页中断时选择要淘汰的页。不再使用的页要比正在使用的页更适合被淘汰。这个位在后面要讨论的 `页面置换` 算法中作用很大。最后一位用于禁止该页面被高速缓存，这个功能对于映射到设备寄存器还是内存中起到了关键作用。通过这一位可以禁用高速缓存。具有独立的 I/O 空间而不是用内存映射 I/O 的机器来说，并不需要这一位。

分页带来的问题

分页系统中要考虑两个问题：

1. 虚拟地址到物理地址的映射必须非常快（速度问题）
   由于每次访问内存都需要进行虚拟地址到物理地址的映射，所有的指令最终都必须来自内存，并且很多指令也会访问内存中的操作数。因此每条指令进行多次页表访问是必要的。如果执行一条指令需要1ns，则页表查询必须在0.2ns之内完成，以避免映射成为主要的瓶颈。
2. 如果虚拟地址空间很大，页表也会很大（空间问题）
   现代计算机使用的虚拟地址至少为32位，而且越来越多的64位。假设一个页面大小为4KB，则32位的地址空间将有100万页，那么64位地址空间更多了。一个页表有100万条表项，你个存储开销就很大。而且每个进程都有自己的页表还

所以我们接下来主要解决的就是这两个问题。

解决速度问题

大多数优化技术都是从内存中的页表开始的，因为这里面会存在这有巨大影响的问题。比如一条1字节指令要把一个寄存器中的数据复制到另一个寄存器，在不分页的情况下这条指令只访问一次内存。有了分页机制之后，会因为要访问页表而引起多次的内存访问。同时，CPU的执行速度会被内存中取指令执行和取数据的速度拉低，所以会使性能下降。解决方案如下：

由于大多数程序总是对少量页面进行多次访问，因此只有很少的页表项会被反复读取，而其他页表项很少会被访问。针对这个问题，为计算机设置一个小型的硬件设备，将虚拟地址直接映射到物理地址，而不必再去访问页表通过MMU得到物理地址，这个设备叫做**转换检测缓冲区**又叫做**快表（TLB）**。通常在MMU中，包含少量的表项，实际中应该有256个，每一个表项都记录了

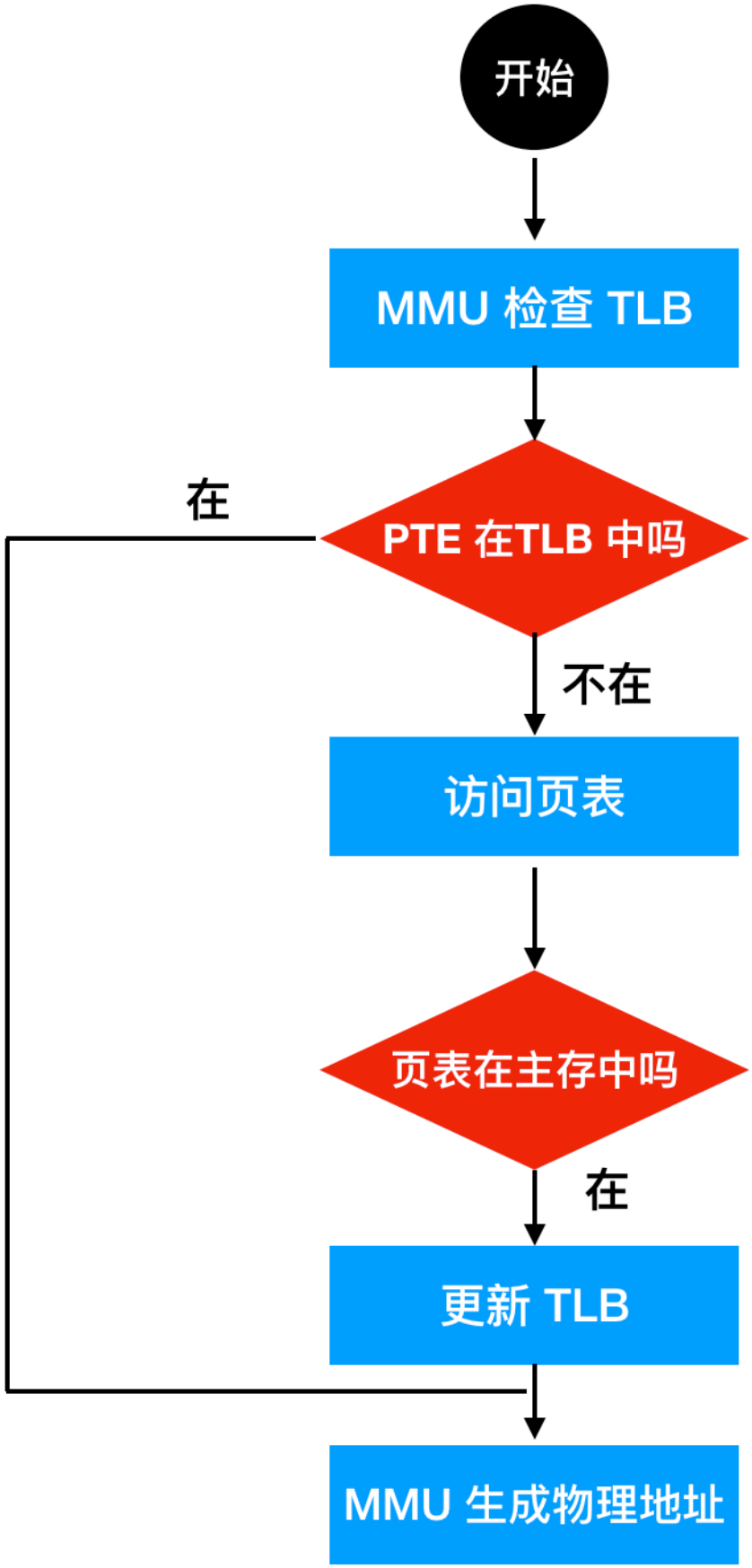
一个页面相关信息，即虚拟页号、修改为、保护码、对应的物理页框号，如下表所示：

有效位	虚拟页面号	修改位	保护位	页框号
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

TLB 其实就是一种内存缓存，用于减少访问内存所需要的时间，它就是 MMU 的一部分，TLB 会将虚拟地址到物理地址的转换存储起来，通常可以称为`地址翻译缓存(address-translation cache)`。TLB 通常位于 CPU 和 CPU 缓存之间，它与 CPU 缓存是不同的缓存级别。下面我们来看一下 TLB 是如何工作的。

当一个 MMU 中的虚拟地址需要进行转换时，硬件首先检查虚拟页号与 TLB 中所有表项进行并行匹配，判断虚拟页是否在 TLB 中。如果找到了有效匹配项，并且要进行的访问操作没有违反保护位的话，则将页框号直接从 TLB 中取出而不用再直接访问页表。如果虚拟页在 TLB 中但是违反了保护位的权限的话（比如只允许读但是是一个写指令），则会生成一个`保护错误(protection fault)`返回。上面探讨的是虚拟地址在 TLB 中的情况，那么如果虚拟地址不再 TLB 中该怎么办？如果 MMU 检测到没有有效的匹配项，就会进行正常的页表查找，然后从 TLB 中逐出一个表项然后把从页表中找到的项放在 TLB 中。当一个表项被从 TLB 中清除出，将修改位复制到内存中页表项，除了访问位之外，其他位保持不变。当页表项从页表装入 TLB 中时，所有的值都来自于内存。

下面给出流程图：



MMU 检查 TLB 过程

软件 TLB 管理

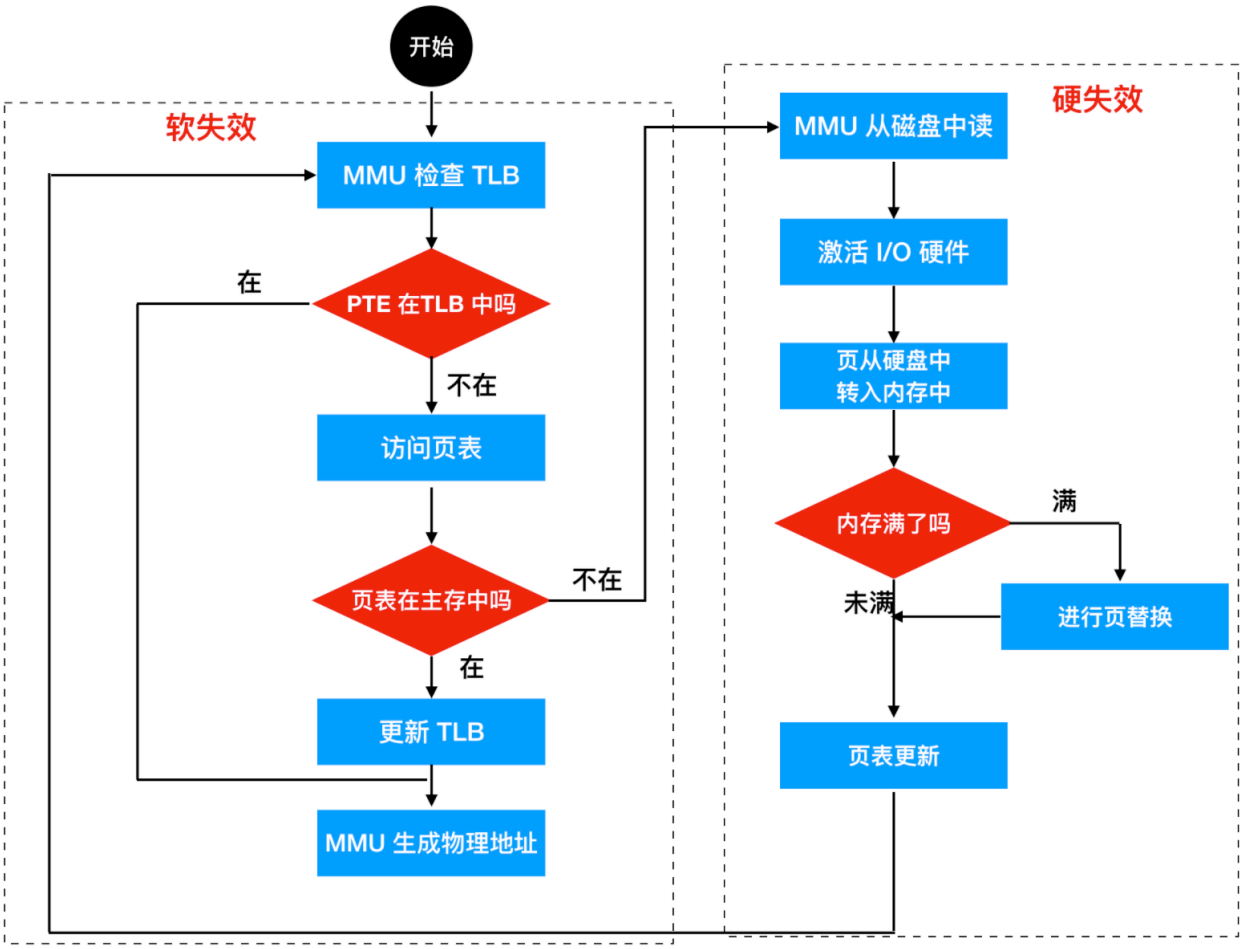
直到现在，我们假设每台电脑都有可以被硬件识别的页表，外加一个 TLB。在这个设计中，TLB 管理和处理 TLB 错误完全由硬件来完成。仅仅当页面不在内存中时，才会发生操作系统的`陷入(trap)`。

但是有些机器几乎所有的页面管理都是在软件中完成的。

在这些计算机上，TLB 条目由操作系统显示加载。当发生 TLB 访问丢失时，**不再是由 MMU 到页表中查找并取出需要的页表项，而是生成一个 TLB 失效并将问题交给操作系统解决**。操作系统必须找到该页，把它从 TLB 中移除（移除页表中的一项），然后把新找到的页放在 TLB 中，最后再执行先前出错的指令。然而，所有这些操作都必须通过少量指令完成，因为 TLB 丢失的发生率要比出错率高很多。

无论是用硬件还是用软件来处理 TLB 失效，常见的方式都是找到页表并执行索引操作以定位到将要访问的页面，在软件中进行搜索的问题是保存页表的页可能不在 TLB 中，这将在处理过程中导致其他 TLB 错误。改善方法是可以在内存中的固定位置维护一个大的 TLB 表项的高速缓存来减少 TLB 失效。通过首先检查软件的高速缓存，`操作系统`能够有效的减少 TLB 失效问题。

TLB 软件管理会有两种 TLB 失效问题，当一个页访问在内存中而不在 TLB 中时，将产生`软失效(soft miss)`，那么此时要做的就是把页表更新到 TLB 中（我们上面探讨的过程），而不会产生磁盘 I/O，处理仅仅需要一些机器指令在几纳秒的时间内完成。然而，当页本身不在内存中时，将会产生`硬失效(hard miss)`，那么此时就需要从磁盘中进行页表提取，硬失效的处理时间通常是软失效的百万倍。在页表结构中查找映射的过程称为`页表遍历(page table walk)`。如图：



上面的这两种情况都是理想情况下出现的现象，但是在实际应用过程中情况会更加复杂，未命中的情况可能既不是硬失效又不是软失效。一些未命中可能更`软`或更`硬`。比如，如果页表遍历的过程中没有找到所需要的页，那么此时会出现三种情况：

1. 所需的页面就在内存中，但是却没有记录在进程的页表中，这种情况可能是由其他进程从磁盘掉入内存，这种情况只需要把页正确映射就可以了，而不需要在从硬盘调入，这是一种软失效，称为`次要缺页错误(minor page fault)`。
2. 基于上述情况，如果需要从硬盘直接调入页面，这就是`严重缺页错误(major page falut)`。
3. 还有一种情况是，程序可能访问了一个非法地址，根本无需向 TLB 中增加映射。此时，操作系统会报告一个`段错误(segmentation fault)`来终止程序。只有第三种缺页属于程序错误，其他缺页情况都会被硬件或操作系统以降低程序性能为代价来修复

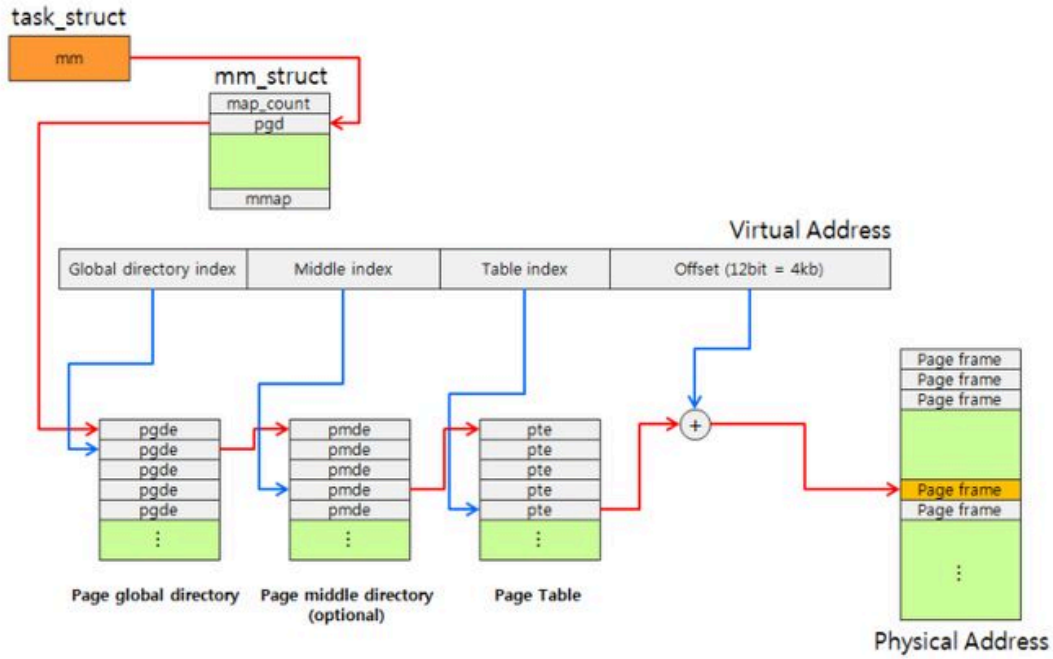
解决内存太大问题

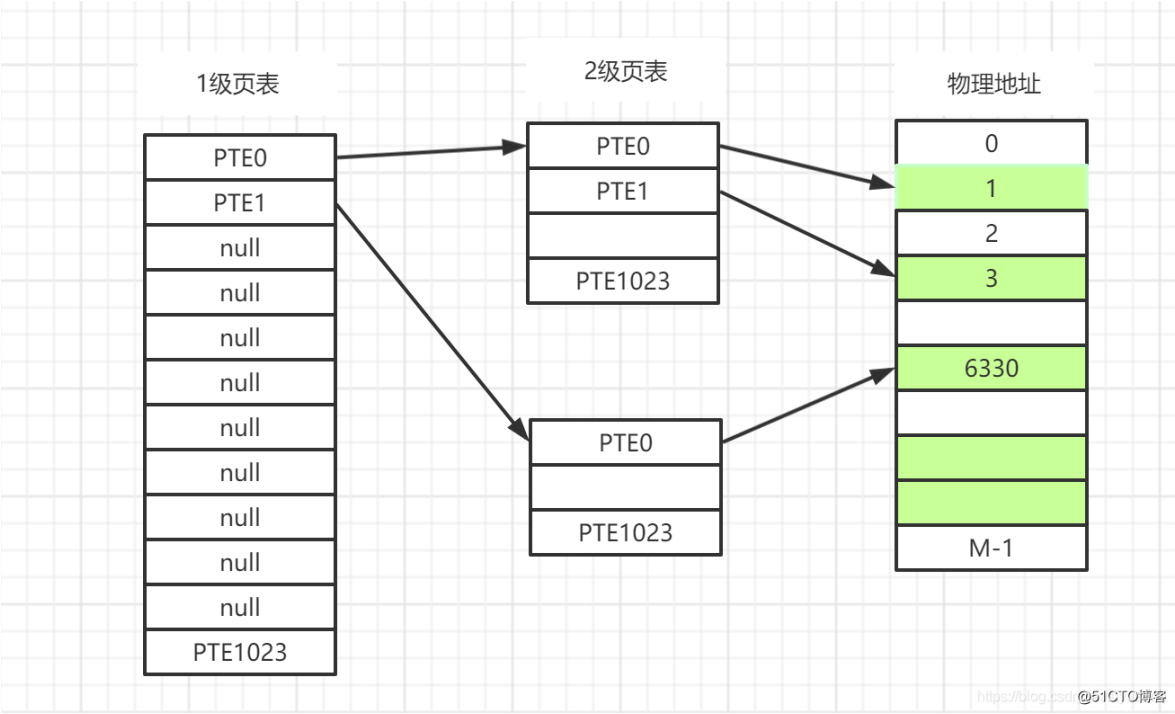
上面引入TLB加快虚拟地址到物理地址的转换，另一个要解决的问题就是处理巨大的虚拟空间，有两种解决方法：多级页表和倒排页表。

多级页表

> 从整体来过一遍虚拟地址的概念，虚拟存储器的基本思想是：程序、数据和堆栈的总大小可能超过可用的物理内存的大小。由操作系统把程序当前使用的那些部分保留在主存中，而把其他部分保存在磁盘上。例如，对于一个16MB的程序，通过仔细地选择在每个时刻将哪4MB内容保留在内存中，并在需要时在内存和磁盘间交换程序的片段，这样这个程序就可以在一个4MB的机器上运行。
>
> 由程序产生的地址被称为虚拟地址，它们构成了一个虚拟地址空间。在使用虚拟存储器的情况下，虚拟地址不是被直接送到内存总线上，而且是被送到内存管理单元(Memory Management Unt,MMU)，MMU把虚拟地址映射为物理内存地址。
>
> 虚拟地址空间以页面为单位划分。在物理内存中对应的单位称为页帧。页面和页帧的大小总是一样的。

- **页表在哪儿**
  任何进程的切换都会更换活动页表集，Linux中为每一个进程维护了一个tast_struct结构体（进程描述符PCB），其中tast_struct->mm_struct结构体成员用来保存该进程页表。在进程切换的过程中，内核把新的页表的地址写入CR3控制寄存器。CR3中含有页目录表的物理内存基地址，如下图：
- **为什么省空间？**
  假如每个进程都有4GB的虚拟地址空间，而显然对于大多数程序来说，其使用到的空间远未达到4GB，何必去映射不可能用到的空间呢？一级页表覆盖了整个4GB虚拟地址空间，但如果某个一级页表的页表项没有被用到，也就不需要创建这个页表项对应的二级页表了，即可以在需要时才创建二级页表。
  如果每个页表项都存在对应的映射地址那也就算了，但是，绝大部分程序仅仅使用了几个页，也就是说，只需要几个页的映射就可以了，如下图（左），进程1的页表，只用到了0,1,1024三个页，剩下的1048573页表项是空的，这就造成了巨大的浪费，为了避免内存浪费，计算机系统开发人员想出了一个方案，多级页表。


task_struct
mm
mm_struct
map_count
pgd
mmap
Virtual Address
Global directory index
Middle index
Table index
Offset (12bit = 4kb)
pgde
pgde
pgde
pgde
pgde
pmde
pmde
pmde
pmde
pmde
pte
pte
pte
pte
pte
Page frame
Page frame
Page frame
Page frame
Page frame
Page global directory
Page middle directory (optional)
Page Table
Physical Address



下面计算一下上图（右）的内存占用情况，对于一级页表来说，假设一个表项是4B，则

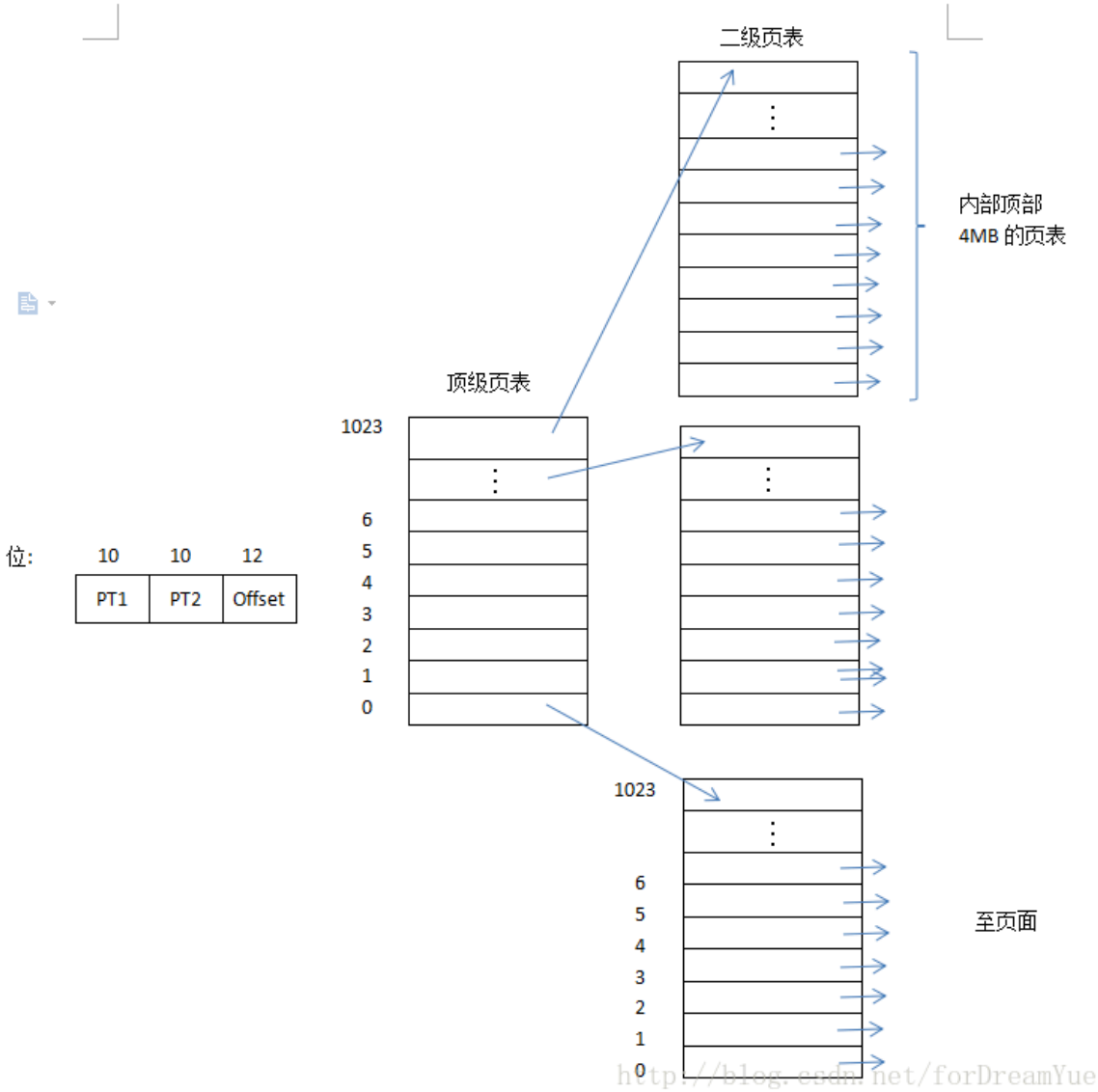
一级页表占用：1024 * 4 B= 4K

2级页表占用 = (1024 * 4 B) * 2 = 8K。

总共的占用情况是 12K，相比只有一个页表 4M，节省了99.7%的内存占用。

因此引入多级页表的原因就是避免把全部页表一直保存在内存中。

- **《深入理解计算机操作系统》 中的解释**



假设32位的虚拟地址被划分为10位的PT1域、10位的PT2域和12位的偏移量域，工作过程如下：当一个虚拟地址被送到MMU时，MMU首先提取PT1域并把该值作为访问顶级页表的索引。由于虚拟空间为32G，顶级页表有$2^{10(PT1)}$=1024个表项，则二级页表有$2^{10(PT2)}$=1024个表项，每一个二级页表都含有虚拟地址对应物理地址的页框号，该页框号与偏移量结合便形成物理地址，放到地址总线送入内存中。
如果没有多级页表，则32位虚拟内存应该有100万个页面，这个存储开销是很大的，而有了多级页表，则实际只需要4个页表，顶级页表、0、1、1024这四个，顶级页表中的其他表项的“在/不在”为被设置为0，访问他们是强制产生一个缺页中断。

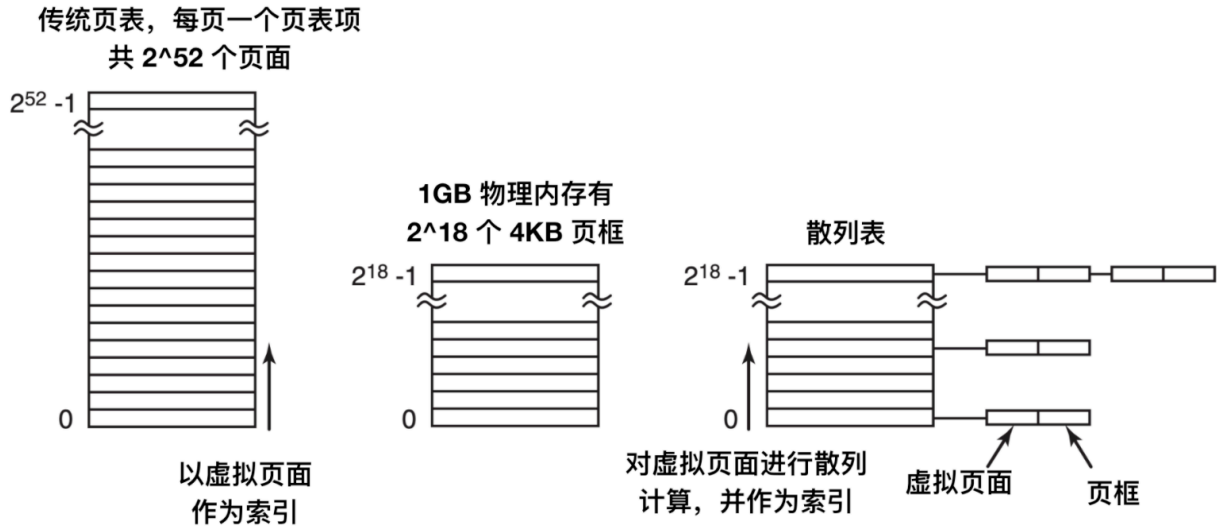
倒排页表

针对不断分级的页表的替代方案是**倒排页表**，实际内存中的每个页框对应一个表项，而不是每个虚拟页面对应一个表项。

对于64位虚拟地址，4KB的页，4GB的RAM，一个倒排页表仅需要1048576个表项。

由于4KB的页面，因此需要$2^{64}/2^{12}=2^{52}$个页面，但是1GB物理内存只能有2^{18}个4KB页框

> 虽然倒排页表节省了大量的空间，但是它也有自己的缺陷：那就是从虚拟地址到物理地址的转换会变得很困难。当进程 n 访问虚拟页面 p 时，硬件不能再通过把 p 当作指向页表的一个索引来查找物理页。而是必须搜索整个倒排表来查找某个表项。另外，搜索必须对每一个内存访问操作都执行一次，而不是在发生缺页中断时执行。解决这一问题的方式时使用 TLB。当发生 TLB 失效时，需要用软件搜索整个倒排页表。一个可行的方式是建立一个散列表，用虚拟地址来散列。当前所有内存中的具有相同散列值的虚拟页面被链接在一起。如下图所示
>
> 如果散列表中的槽数与机器中物理页面数一样多，那么散列表的冲突链的长度将会是 1 个表项的长度，这将会大大提高映射速度。一旦页框被找到，新的（虚拟页号，物理页框号）就会被装在到 TLB 中。



逻辑地址到物理地址的转换

页式存储管理的逻辑地址分为两部分：页号和页内地址。物理地址分为两部分：块号＋页内地址；

逻辑地址＝　页号＋页内地址

物理地址＝　块号＋页内地址；

举例子

某虚拟存储器的用户编程空间共32个页面，每页为1KB，内存为16KB。假定某时刻一用户页表中已调入内存的页面的页号和物理块号的对照表如下，则逻辑地址0A5C(H)所对应的物理地址是什么？要求：写出主要计算过程。

页号	物理块号
0	3
1	7
2	11
3	8

解答

用户编程空间共32个页面，$2^5 = 32$得知页号部分占５位，由“每页为1KB”，$1K = 2^{10}$，可知内页地址占10位。

由“内存为16KB”，$2^4 = 16$得知块号占4位。

然后找页号部分，转换成二进制查表，得到物理块号，然后转为二进制拼接页内地址就ok了。

逻辑地址0A5C（H）所对应的二进制表示形式是：００００１０１００１０１１１００，后十位１００１０１１１００是页内地址，

０００１０为为页号，页号化为十进制是２，在对照表中找到２对应的物理块号是１１,１１转换二进制是１０１１，即可求出物理地址为１０１１１００１０１１１００，化成十六进制为２Ｅ５Ｃ；

即则逻辑地址0A5C(H)所对应的物理地址是２Ｅ５Ｃ；

页面置换算法

缺页中断：一个进程所有地址空间里的页面不必全部常驻内存，在执行一条指令时，如果发现他要访问的页没有在内存中（即存在位为0），那么停止该指令的执行，并产生一个页不存在的异常，对应的故障处理程序可通过从物理内存加载该页的方法来排除故障，之后，原先引起的异常的指令就可以继续执行，而不再产生异常。

- 最佳页面置换算法（*OPT*）（理想算法）
  最佳页面置换算法基本思路是，置换在「未来」最长时间不访问的页面。所以，该算法实现需要计算内存中每个逻辑页面的「下一次」访问时间，然后比较，选择未来最长时间不访问的页面。这很理想，但是实际系统中无法实现，因为程序访问页面时是动态的，我们是无法预知每个页面在「下一次」访问前的等待时间。所以，最佳页面置换算法作用是为了衡量你的算法的效率，你的算法效率越接近该算法的效率，那么说明你的算法是高效的。
- 先进先出置换算法（*FIFO*）
  选择在内存驻留时间很长的页面进行中置换
- 最近最久未使用的置换算法（*LRU*）
  发生缺页时，选择最长时间没有被访问的页面进行置换，也就是说，该算法假设已经很久没有使用的页面很有可能在未来较长的一段时间内仍然不会被使用。
  虽然 LRU 在理论上是可以实现的，但代价很高。为了完全实现 LRU，需要在内存中维护一个所有页面的链表，最近最多使用的页面在表头，最近最少使用的页面在表尾。困难的是，在每次访问内存时都必须要更新「整个链表」。在链表中找到一个页面，删除它，然后把它移动到表头是一个非常费时的操作。所以，LRU 虽然看上去不错，但是由于开销比较大，实际应用中比较少使用。
- 时钟页面置换算法（*Lock*）
  把所有的页面都保存在一个类似钟面的「环形链表」中，一个表针指向最老的页面。
  当发生缺页中断时，算法首先检查表针指向的页面：
    - 如果它的访问位位是 0 就淘汰该页面，并把新的页面插入这个位置，然后把表针前移一个位置；
    - 如果访问位是 1 就清除访问位，并把表针前移一个位置，重复这个过程直到找到了一个访问位为 0 的页面为止；
- 最不常用置换算法（*LFU*）
  当发生缺页中断时，选择「访问次数」最少的那个页面，并将其淘汰。
  要增加一个计数器来实现，这个硬件成本是比较高的，另外如果要对这个计数器查找哪个页面访问次数最小，查找链表本身，如果链表长度很大，是非常耗时的，效率不高。

对缺页中断的处理

1. 硬件陷入内核，在堆栈中保存程序计数器。大多数机器将当前的指令，各种状态信

息保存在特殊的CPU寄存器中。
2. 启动一个汇编代码保存通用寄存器和其他易失信息，防止被操作系统破坏
3. 当操作系统收到缺页中断信号后，定位到需要的虚拟页面。
4. 找到发生缺页中断的虚拟地址，操作系统检查这个地址是否有效，并检查存取与保护是否一致。
   如果不一致则杀掉该进程
   如果地址有效且没有保护错误发生，系统会检查是否有空闲页框。如果没有空闲页框就执行页面置换算法淘汰一个页面。
5. 如果选择的页框对应的页面发生了修改，即为“脏页面”，需要写回磁盘，并发生一次上下文切换，挂起产生缺页中断的进程，让其他进程运行直至全部把内容写到磁盘。
6. 一旦页框是干净的，则OS会查找要发生置换的页面对应磁盘上的地址，通过磁盘操作将其装入。在装入该页面的时候，产生缺页中断的进程仍然被挂起，运行其他可运行的进程
7. 当发生磁盘中断时表明该页面已经被装入，页表已经更新可以反映其位置，页框也被标记为正常状态。
8. 恢复发生缺页中断指令以前的状态，程序计数器重新指向引起缺页中断的指令
9. 调度引发缺页中断的进程
10. 该进程恢复寄存器和其他状态信息，返回用户空间继续执行。

Linux可执行文件如何装载进虚拟内存

源代码通过预处理，编译，汇编，链接后形成可执行文件，因为计算机的操作系统的启动程序是写死在硬件上的，每次计算机上电时，都将自动加载启动程序，之后的每一个程序，每一个应用，都是不断的 fork 出来的新进程。那么我们的可执行文件，以linux 系统为例，也是由shell 进程 fork 出一个新进程，在新进程中调用exec函数装载我们的可执行文件并执行。

1. execve()。当shell中敲入执行程序的指令之后，shell进程获取到敲入的指令，并执行execve()函数，该函数的参数是敲入的可执行文件名和形参，还有就是环境变量信息。execve()函数对进程栈进行初始化，即压栈环境变量值，并压栈传入的参数值，最后压栈可执行文件名。初始化完成后调用 sys_execve()
2. sys_execve()。该函数进行一些参数的检查与复制，而后调用 do_execve()
3. do_execve()。该函数在当前路径与环境变量的路径中寻找给定的可执行文件名，找到文件后读取该文件的前128字节。读取这128个字节的目的是为了判断文件的格

式，每个文件的开头几个字节都是魔数，可以用来判断文件类型。读取了前128字节的文件头部后，将调用 search_binary_handle()

4. search_binary_handle()。该函数将去搜索和匹配合适的可执行文件装载处理程序。Linux 中所有被支持的可执行文件格式都有相应的装在处理程序。以Linux 中的ELF文件为例，接下来将会调用elf 文件的处理程序：load_elf_binary()
5. load_elf_binary()。
   该函数执行以下三个步骤:
   a）创建虚拟地址空间：实际上指的是建立从虚拟地址空间到物理内存的映射函数所需要的相应的数据结构。（即创建一个空的页表）
   b）读取可执行文件的文件头，建立可执行文件到虚拟地址空间之间的映射关系
   c）将CPU指令寄存器设置为可执行文件入口（虚拟空间中的一个地址）
   load_elf_binary()函数执行完毕，事实上装载函数执行完毕后，可执行文件真正的指令和数据都没有被装入内存中，只是建立了可执行文件与虚拟内存之间的映射关系，以及分配了一个空的页表，用来存储虚拟内存与物理内存之间的映射关系。
6. 程序返回到execve()中。此时从内核态返回到用户态，且寄存器的地址被设置为了ELF 的入口地址，于是新的程序开始启动，发现程序入口对应的页面并没有加载（因为初始时是空页面），则此时引发一个缺页错误，操作系统根据可执行文件和虚拟内存之间的映射关系，在磁盘上找到缺的页，并申请物理内存，将其加载到物理内存中，并在页表中填入该虚拟内存页与物理内存页之间的映射关系。之后程序正常运行，直至结束后回到shell 父进程中，结束回到 shell。

说一说内存对齐

- 由一个例子引出内存对齐

```
//32位系统
#include<stdio.h>
struct{
    int x;
    char y;
}s;

int main()
{
    printf("%d\n",sizeof(s);  // 输出8
    return 0;
}
```

上述代码实际得到的值是8byte，这就是内存对齐所导致的。

> 从理论上来讲，任何类型的变量可以从任意地址开始存放。但是实际上计算机对基本数据类型在内存中的存放位置有限制，会要求数据首地址的值是某个数（通常是4或8）的整数倍，这就是内存对齐。

- 为什么要内存对齐呢？
  计算机的内存是以字节为单位的，但是大部分处理器并不是按照字节块来存取内存的。计算机有个内存存取粒度的概念，就是说一般以2字节，4字节，8字节这样的单位来存取内存。
  性能原因：加入没有内存对齐机制，数据可以任意存放。地址排列为01234567现在读取一个int型变量，这个变量存放地址从1开始，那我们从0字节开始找，找到1后读取第一个4字节，把0扔掉，然后从地址4开始读下4个字节，扔掉567地址，int变量就存储在1234这里，可以看到这样很浪费开销，因为访问了两次内存。
- 内存对齐的规则
  对其规则如下：
  i. 基本类型的对齐值就是sizeof值。如果该成员是c++自带类型如int、char、double等，那么其对齐字节数=该类型在内存中所占的字节数；如果该成员是自定义类型如某个class或者struct，那个它的对齐字节数 = 该类型内最大的成员对齐字节数
  ii. 结构体。结构体本身也要对齐，按照最大成员长度来参照的。
  iii. 编译器可以设置最大对齐值，gcc中默认是#pragma pack(4)。但是类型的实际对齐值与默认对齐值取最小值来
  iv. 如果设置了对齐字节数，就另说。①如果定义的字节数为1，就是所有

默认字节数直接相加。②定义的字节数大于任何一个成员大小时候，不产生任何效果。如果定义的对齐字节数大于结构体内最小的，小于结构体内最大的话，就按照定义的字节数来计算

内存空间的堆和栈的区别是什么？

- 程序内存布局场景下，堆与栈表示两种内存管理方式；
    - 栈是由操作系统自动分配的，用于存放函数参数值，局部变量。存储在栈中的数据的生命周期随着函数的执行结束而结束。栈的内存生长方向与堆相反，由高到低，按照变量定义的先后顺序入栈。
    - 堆是由用户自己分配的。如果用户不回收，程序结束后由操作系统自动回收。堆的内存地址生长方向与栈相反，由低到高。

      > **堆上分配内存的过程**：
      >
      > 操作系统有一个记录空闲内存地址的链表，当系统收到程序的开辟内存申请时候，会遍历该链表，寻找第一个空间大于所申请内存空间的节点。接着把该节点从空闲链表中删除，同时将该空间分配出去给程序使用。同时大多数系统会在内存空间中的首地址记录此次分配的大小，这样delete才能正确释放内存空间。由于找到的节点所对应的内存大小不一定正好等于申请内存的大小，OS会自动的将多余的部分放入空闲链表。

    - 堆与栈区别的总结
        a. 管理方式不同
        b. 空间大小不同
        c. 分配方式不同。堆都是动态分配。栈的静态分配由操作系统完成，回收也是自动的。栈的动态分配有alloca函数分配，由操作系统自动回收。
        d. 分配效率不同。栈由操作系统自动分配，会在硬件层级对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是由C/C++提供的库函数或运算符来完成申请与管理，实现机制较为复杂，频繁的内存申请容易产生内存碎片。显然，堆的效率比栈要低得多。
- 数据结构场景下，堆与栈表示两种常用的数据结构。

栈是线性结构
堆是一种特殊的完全二叉树

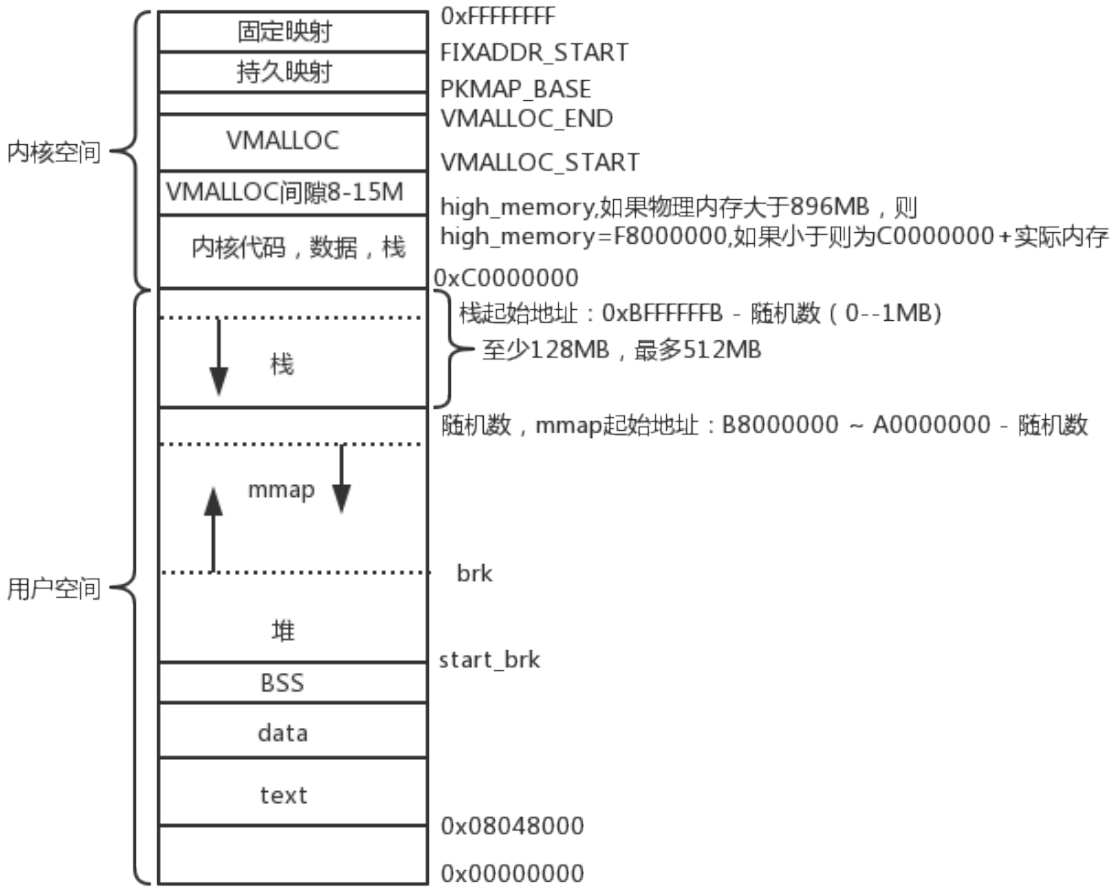
内存为什么要分堆栈在编程里，要是全部只用堆或者全部只用栈可以吗

程序的结构是前部是栈，中部是程序体，后部是堆。前部和中部是链接的时候由编译器算好了的，执行过程中是固定不变的。而后部则可以随着程序的执行而改变长度。

栈是用来存储程序初期设定的变量的，这些变量在程序执行之前就要准备好。因此栈要放在程序的前部并且固定位置，否则程序就不知道该到那里去找这些变量。而程序的运行结果往往是不能预先确定的，所以把堆放在后部以便可以提供足够的内存保存运算结果。

进程虚拟空间是怎么布局的？进程内存模型

linux进程在32位处理器下的虚拟空间内存布局，从高地址到低地址



- 内核空间，从C000000-FFFFFFFF
- 栈区。有以下用途
    i. 存储函数局部变量
    ii. 记录函数调用过程的相关信息，成为栈帧，主要包括函数返回地址，一些不适合放在寄存器中的函数参数。
    iii. 临时存储区，暂存算术表达式的计算结果和allocation函数分配的栈内存。
- 内存映射段（mmap）。内核将硬盘文件的内容直接映射到内存，是一种方便高校的文件I/O方式，用于装在动态共享库。
- 堆。分配的堆内存是经过字节对齐的空间，以适合原子操作。堆管理器通过链表管理每个申请的内存，由于堆申请和释放是无序的，最终会产生内存碎片。堆内存一般由应用程序分配释放，回收的内存可供重新使用。若程序员不释放，程序结束时操作系统可能会自动回收。
- BSS段。通常存放以下内容：
    i. 未初始化的全局变量和静态局部变量

ii. 初始化值为0的全局变量和静态局部变量
iii. 未定义且初值不为0的符号

- 数据段。通常存放程序中已经初始化且初值不为0的全局变量。数据段属于静态存储区，可读可写。
  数据段和BSS段的区别如下：
  i. BSS段不占用物理文件尺寸，但占用内存空间（不在可执行文件中）。数据段在可执行文件中，也占用内存空间。
  ii. 当程序读取数据段的数据时候，系统会发生缺页故障，从而分配物理内存。当程序读取BSS段数据的时候，内核会将其转到一个全零页面，不会发生缺页故障，也不会为期分配物理内存。
  iii. bss是不占用.exe文件（可执行文件）空间的，其内容由**操作系统初始化**（清零）；而data却需要占用，其内容由**程序初始化**，因此造成了上述情况。
- 代码段。代码段也称正文段或文本段，通常用于存放程序执行代码(即CPU执行的机器指令)。一般C语言执行语句都编译成机器代码保存在代码段。通常代码段是可共享的，因此频繁执行的程序只需要在内存中拥有一份拷贝即可。
- 保留区。位于虚拟地址空间的最低部分，未赋予物理地址。任何对它的引用都是非法的，用于捕捉使用空指针和小整型值指针引用内存的异常情况。

系统调用和进程上下文切换的区别

系统调用过程中，并不会涉及到虚拟内存等进程用户态的资源，也不会切换进程。这跟我们通常所说的进程上下文切换是不一样的：

进程上下文切换，是指从一个进程切换到另一个进程运行；而系统调用过程中一直是同一个进程在运行。

上下文切换

> **CPU 寄存器和程序计数器就是 CPU 上下文，因为它们都是 CPU 在运行任何任务前，必须的依赖环境。**
>
> - CPU 寄存器是 CPU 内置的容量小、但速度极快的内存。
> - 程序计数器则是用来存储 CPU 正在执行的指令位置、或者即将执行的下一条

> 指令位置。

什么是CPU上下文切换

就是先把前一个任务的 CPU 上下文（也就是 CPU 寄存器和程序计数器）保存起来，然后加载新任务的上下文到这些寄存器和程序计数器，最后再跳转到程序计数器所指的新位置，运行新任务。而这些保存下来的上下文，会存储在系统内核中，并在任务重新调度执行时再次加载进来。这样就能保证任务原来的状态不受影响，让任务看起来还是连续运行。

CPU 上下文切换的类型

- **进程上下文切换**
    i. .切换页目录以使用新的地址空间
    ii. 切换内核栈
    iii. 切换硬件上下文
    iv. 刷新TLB
    v. 系统调度器的代码执行
- **线程上下文切换**
  两个线程属于不同进程
  前后两个线程属于不同进程。此时，因为资源不共享，所以切换过程就跟进程上下文切换是一样。
  两个线程属于相同进程
  线程上下文切换和进程上下文切换一个最主要的区别是线程的切换虚拟内存空间依然是相同的，
  但是进程切换是不同的。这两种上下文切换的处理都是通过操作系统内核来完成的。
  内核的这种切换过程伴随的最显著的性能损耗是将寄存器中的内容切换出。
  另外一个隐藏的损耗是上下文的切换会扰乱处理器的缓存机制。
  简单的说，一旦去切换上下文，处理器中所有已经缓存的内存地址一瞬间都作废了。
- **中断上下文切换**
  为了快速响应硬件的事件,中断处理会打断进程的正常调度和执行，转而调用中断处理程序，响应设备事件。而在打断其他进程时，就需要将进程当前的状态保存下来，这样在中断结束后，进程仍然可以从原来的状态恢复运行。

中断上下文切换并不涉及到进程的用户态。即便中断过程打断了一个正处在用户态的进程，也不需要保存和恢复这个进程的虚拟内存、全局变量等用户态资源。只需要关注内核资源就行，CPU寄存器，内核堆栈，硬件中断参数啥的。

- **进程上下文切换的场景**：
    i. 时间片轮转技术下，该进程分配到的时间片耗尽，就会被系统挂起，切换到其他进程
    ii. 进程在系统资源不足（比如内存不足）时，要等到资源满足后才可以运行，这个时候进程也会被挂起，并由系统调度其他进程运行。
    iii. 当进程通过睡眠函数 sleep 这样的方法将自己主动挂起时，自然也会重新调度。
    iv. 当有优先级更高的进程运行时，为了保证高优先级进程的运行，当前进程会被挂起，由高优先级进程来运行
    v. 发生硬件中断时，CPU 上的进程会被中断挂起，转而执行内核中的中断服务程序。

什么是大端字节，什么是小端字节？如何转换字节序？

大端序，小端序是计算机存储数据的两种方式。

大端字节序：高位字节在前，低位字节在后，符合人类读写数值的习惯（参考普通的十进制数）

小端字节序：低位字节在前，高位字节在后。

- 为什么要有大小端字节序？统一不行吗？
  答：①内存的低地址处存放低字节，所以在强制转换数据时不需要调整字节的内容（注解：比如把int的4字节强制转换成short的2字节时，就直接把int数据存储的前两个字节给short就行，因为其前两个字节刚好就是最低的两个字节，符合转换逻辑）；②CPU做数值运算时从内存中依顺序依次从低位到高位取数据进行运算，直到最后刷新最高位的符号位，这样的运算方式会更高效。但是大端序更符合人类的习惯，主要用在网络传输和文件存储方面，符号位在所表示的数据的内存的第一个字节中，便于快速判断数据的正负和大小。
  其各自的优点就是对方的缺点，正因为两者彼此不分伯仲，再加上一些硬件厂商的坚持，因此在多字节存储顺序上始终没有一个统一的标准

- 转换字节序
  主要是针对主机字节序和网络字节序来说的。我们用的x86架构的处理器一般都是小端序存储数据，但是网络字节序是TCP/IP中规定好的数据表示格式，RFC1700规定使用“大端”字节序为网络字节序，独立于处理器操作系统。
  在Linux网络编程中，会使用下列C标准库函数进行字节之间的转换

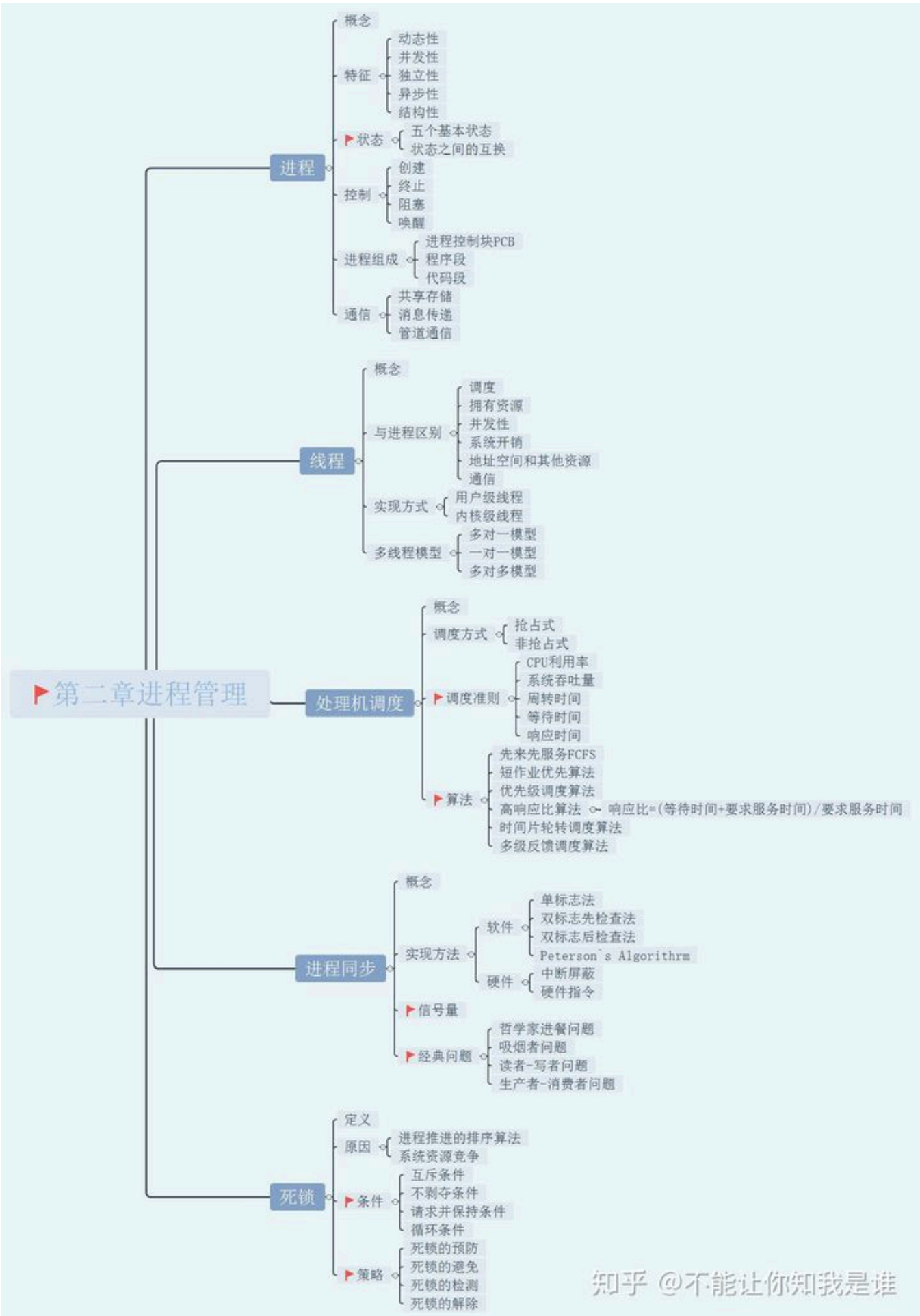
```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);                  //把uint32_t类型从主机序转换到网络序
uint16_t htons(uint16_t hostshort);                  //把uint16_t类型从主机序转换到网络序
uint32_t ntohl(uint32_t netlong);                  //把uint32_t类型从网络序转换到主机序
uint16_t ntohs(uint16_t netshort);                  //把uint16_t类型从网络序转换到主机序
```

- 如何判断本机是大端序还是小端序？

```
int i=1;
char *p=(char *)&i;
if(*p == 1)
    printf("小端模式"); //小端序是低位字节在前，高位字节在后
else // (*p == 0)
    printf("大端模式");
```

操作系统是怎么进行进程管理的？


第二章进程管理
进程
概念
特征
动态性
并发性
独立性
异步性
结构性
状态
五个基本状态
状态之间的互换
控制
创建
终止
阻塞
唤醒
进程组成
进程控制块PCB
程序段
代码段
通信
共享存储
消息传递
管道通信
线程
概念
与进程区别
调度
拥有资源
并发性
系统开销
地址空间和其他资源
通信
实现方式
用户级线程
内核级线程
多线程模型
多对一模型
一对一模型
多对多模型
处理机调度
概念
调度方式
抢占式
非抢占式
调度准则
CPU利用率
系统吞吐量
周转时间
等待时间
响应时间
算法
先来先服务FCFS
短作业优先算法
优先级调度算法
高响应比算法
响应比=(等待时间+要求服务时间)/要求服务时间
时间片轮转调度算法
多级反馈调度算法
进程同步
概念
实现方法
软件
单标志法
双标志先检查法
双标志后检查法
Peterson`s Algorithrm
硬件
中断屏蔽
硬件指令
信号量
经典问题
哲学家进餐问题
吸烟者问题
读者-写者问题
生产者-消费者问题
死锁
定义
原因
进程推进的排序算法
系统资源竞争
条件
互斥条件
不剥夺条件
请求并保持条件
循环条件
策略
死锁的预防
死锁的避免
死锁的检测
死锁的解除

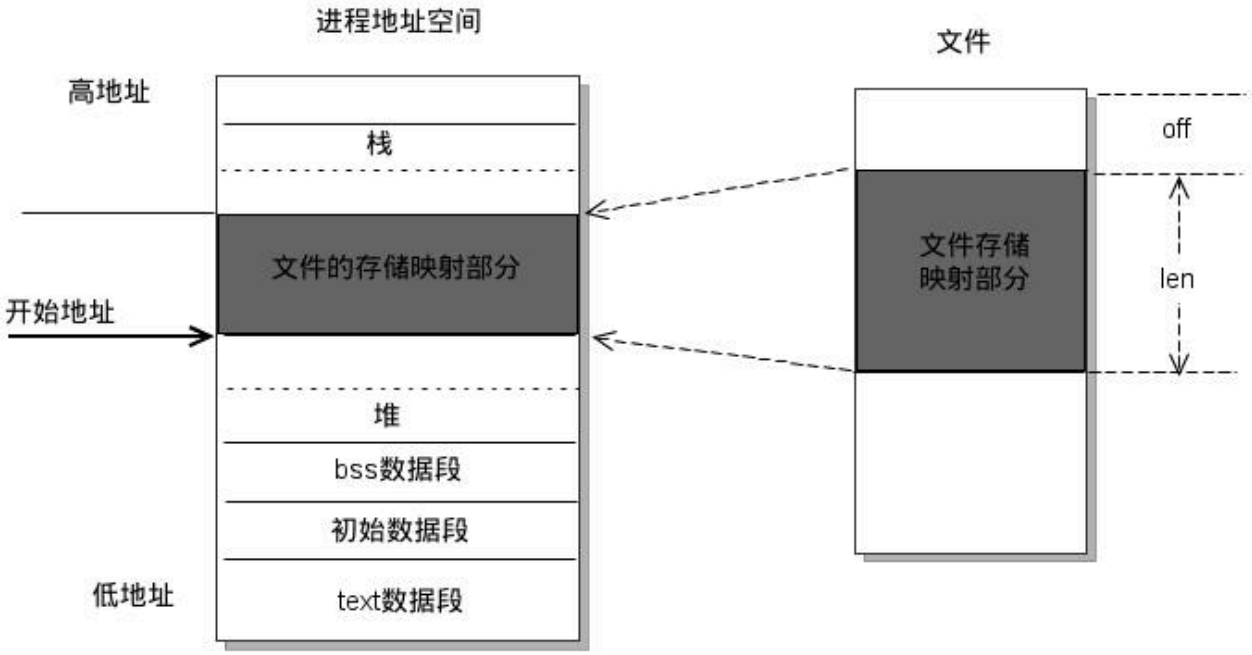
如果问到进程管理，就从进程通信，同步和死锁三大块来说！

- [进程间通信](# 进程间通信方式)
- 进程同步
- 死锁
- 进程调度

mmap（内存映射）

什么是mmap？

mmap是一种内存映射文件的方法，即将一个文件或者其它对象映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对映关系。实现这样的映射关系后，进程就可以采用指针的方式读写操作这一段内存，而系统会自动回写脏页面到对应的文件磁盘上，即完成了对文件的操作而不必再调用read,write等系统调用函数。相反，内核空间对这段区域的修改也直接反映用户空间，从而可以实现不同进程间的文件共享。



mmap的过程—原理

1. 进程启动映射过程，并在虚拟地址空间中为映射创建虚拟映射区域
2. 调用内核空间的系统调用函数mmap（不同于用户空间函数），实现文件物理地址和进程虚拟地址的一一映射关系
3. 进程发起对这片映射空间的访问，引发缺页异常，实现文件内容到物理内存（主存）的拷贝

mmap和常规文件操作的区别

常规文件的操作如下：

1. 进程发起读文件请求。
2. 内核查找文件描述符，定位到内核已打开的文件信息，找到文件的inode。
3. 查看文件页是否在缓存中，如果存在则直接返回这片页面
4. 如果不存在，缺页中断，需要定位到该文件的磁盘地址处，将数据从磁盘复制到页缓存中，然后发起页面读写过程，将页缓存中的数据发送给用户

常规文件需要先将文件页从磁盘拷贝到页缓存中，由于页缓存处在内核空间，不能被用户进程直接寻址，所以还需要将页缓存中数据页再次拷贝到内存对应的用户空间中。这样，通过了两次数据拷贝过程，才能完成进程对文件内容的获取任务。

而使用mmap操作文件中，创建新的虚拟内存区域和建立文件磁盘地址和虚拟内存区域映射这两步，没有任何文件拷贝操作。而之后访问数据时发现内存中并无数据而发起的缺页异常过程，可以通过已经建立好的映射关系，只使用一次数据拷贝，就从磁盘中将数据传入内存的用户空间中，供进程使用。

共享内存的原理

共享内存可以说是最有用的进程间通信方式，也是最快的IPC形式。两个不同进程A、B共享内存的意思是，同一块物理内存被映射到进程A、B各自的进程地址空间。进程A可以即时看到进程B对共享内存中数据的更新，反之亦然。由于多个进程共享同一块内存区域，必然需要某种同步机制，互斥锁和信号量都可以。

这种 IPC 机制无需内核介入！

采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝(用户空间buf到内核，内核把数据拷贝到内存，内存拷贝到内核，内核到用户空间)，而共享内存则只拷贝两次数据(一次从输入文件到共享内存区，另一次从共享内存区到输出文件。)

实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

共享内存位于进程空间的什么位置？

不同进程之间共享的内存通常为同一段物理内存。进程可以将同一段物理内存连接到他们自己的地址空间中，所有的进程都可以访问共享内存中的地址。

共享内存段最大限制是多少？

单个共享内存段最大字节数，一般为32M

共享内存段最大个数，一般为4096

系统中共享内存页总数默认值：2097152*4096=8GB

共享内存不保证同步，可以使用信号量来保证共享内存同步

> 注意共享内存和内存映射是不同的

参考

共享内存和内存映射的区别

1. 共享内存可以直接创建，内存映射需要磁盘文件（匿名映射除外）
2. 共享内存效率更高
3. 内存。
   共享内存，所有的进程操作的是同一块共享内存。
   内存映射，每个进程在自己的虚拟地址空间中有一个独立的内存。
4. 数据安全
   进程突然退出时，共享内存还存在，内存映射区消失

运行进程的电脑死机，宕机时。在共享内存中的数据会消失。内存映射区的数据，由于磁盘文件中的数据还在，所以内存映射区的数据还存在。

5. 生命周期
   内存映射区：进程退出，内存映射区销毁
   共享内存：进程退出，共享内存还在，标记删除（所有的关联的进程数为0），或者关机
   如果一个进程退出，会自动和共享内存进行取消关联。

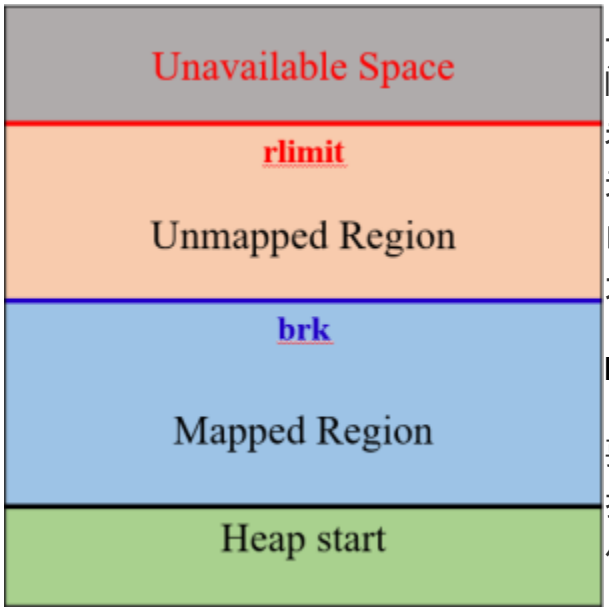
malloc是如何实现内存管理的

参考链接

C语言中使用malloc可以分配一段连续的内存空间。在c/c++开发中，因为malloc属于C标准库函数，经常会使用其分配内存。malloc是在堆中分配一块可用内存给用户。作为一个使用频繁的基础函数，理解清楚其实现原理很有必要，因此本文主要探讨malloc的具体实现原理，以及在linux系统中这该函数的实现方式。

内存分配涉及到的系统调用

跟brk/sbrk/mmap这三个系统调用函数有关



上图是一个堆内存的分布，其实在堆内存有三段空间，第一段是映射好的，指针可以访问的；第二段是未映射的地址空间，访问这段空间会报错；第三段是无法使用的空间。其中映射好的空间和未映射的空间由一个brk指针分割。所以如果要增加实际堆的可用大小，就可以移动brk指针。

brk()和sbrk()

要增加一个进程实际的可用堆大小，就需要将break指针向高地址移动。Linux通过**brk和sbrk系统调用**操作break指针。两个系统调用的原型如下：

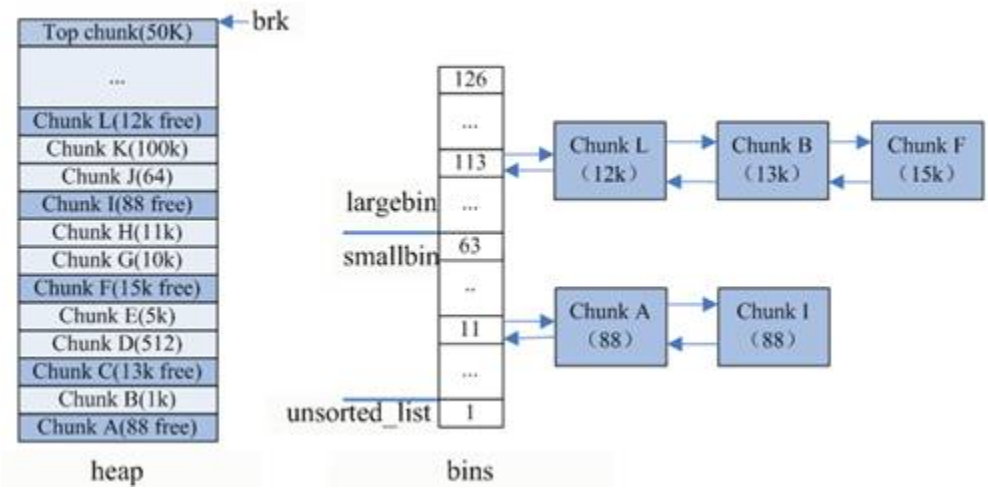
```
#include <unistd.h>
int brk(void *addr);
void *sbrk(intptr_t increment);
```

brk函数将break指针直接设置为某个地址，而sbrk将break指针从当前位置移动increment所指定的增量。所以我们使用sbrk获得brk的地址。brk在执行成功时返回0，否则返回-1并设置errno为ENOMEM；sbrk成功时返回break指针移动之前所指向的地址，否则返回(void *)-1。

mmap函数：mmap函数第一种用法是映射磁盘文件到内存中；而malloc使用的mmap函数的第二种用法，即匿名映射，匿名映射不映射磁盘文件，而是向映射区申请一块内存。当申请小内存的时，malloc使用sbrk分配内存；当申请大内存时，使用mmap函数申请内存；但是这只是分配了虚拟内存，还没有映射到物理内存，当访问申请的内存时，才会因为缺页异常，内核分配物理内存。

malloc实现方案

由于brk/sbrk/mmap属于系统调用，如果每次申请内存，都调用这三个函数中的一个，那么每次都要产生系统调用开销（即cpu从用户态切换到内核态的上下文切换，这里要保存用户态数据，等会还要切换回用户态），这是非常影响性能的；其次，这样申请的内存容易产生碎片，因为堆是从低地址到高地址，如果低地址的内存没有被释放，高地址的内存就不能被回收。鉴于此，**malloc采用的是内存池的实现方式**，malloc内存池实现方式更类似于STL分配器和memcached的内存池，先申请一大块内存，然后将内存分成不同大小的内存块，然后用户申请内存时，直接从内存池中选择一块相近的内存块即可。



如上图，内存池保存在bins这个长128的数组中，每个元素都是一双向个链表。

malloc将内存分成了大小不同的chunk，然后通过bins来组织起来。malloc将相似大小的chunk（图中可以看出同一链表上的chunk大小差不多）用双向链表链接起来，这样一个链表被称为一个bin。malloc一共维护了128个bin，并使用一个数组

来存储这些bin。数组中第一个为**unsorted bin**，数组编号前2到前64的bin为**small bins**，同一个small bin中的chunk具有相同的大小，两个相邻的small bin中的chunk大小相差8bytes。small bins后面的bin被称作**large bins**。large bins中的每一个bin分别包含了一个给定范围内的chunk，其中的chunk按大小序排列。large bin的每个bin相差64字节。

malloc除了有unsorted bin，small bin，large bin三个bin之外，还有一个**fast bin**。一般的情况是，程序在运行时会经常需要申请和释放一些较小的内存空间。当分配器合并了相邻的几个小的 chunk 之后，也许马上就会有另一个小块内存的请求，这样分配器又需要从大的空闲内存中切分出一块，这样无疑是比较低效的，故而，malloc 中在分配过程中引入了 fast bins，不大于 max_fast(默认值为 64B)的 chunk 被释放后，首先会被放到 fast bins中，fast bins 中的 chunk 并不改变它的使用标志 P。这样也就无法将它们合并，当需要给用户分配的 chunk 小于或等于 max_fast 时，malloc 首先会在 fast bins 中查找相应的空闲块，然后才会去查找 bins 中的空闲 chunk。在某个特定的时候，malloc 会遍历 fast bins 中的 chunk，将相邻的空闲 chunk 进行合并，并将合并后的 chunk 加入 unsorted bin 中，然后再将 unsorted bin 里的 chunk 加入 bins 中。

unsorted bin 的队列使用 bins 数组的第一个，如果被用户释放的 chunk 大于 max_fast，或者 fast bins 中的空闲 chunk 合并后，这些 chunk 首先会被放到 unsorted bin 队列中，在进行 malloc 操作的时候，如果在 fast bins 中没有找到合适的 chunk，则malloc 会先在 unsorted bin 中查找合适的空闲 chunk，然后才查找 bins。如果 unsorted bin 不能满足分配要求。 malloc便会将 unsorted bin 中的 chunk 加入 bins 中。然后再从 bins 中继续进行查找和分配过程。从这个过程可以看出来，**unsorted bin 可以看做是 bins 的一个缓冲区，增加它只是为了加快分配的速度。**（其实感觉在这里还利用了局部性原理，常用的内存块大小差不多，从unsorted bin这里取就行了，这个和TLB之类的都是异曲同工之妙啊！）

除了上述四种bins之外，malloc还有三种内存区。

- 当fast bin和bins都不能满足内存需求时，malloc会设法在**top chunk**中分配一块内存给用户；top chunk为在mmap区域分配一块较大的空闲内存模拟sub-heap。（比较大的时候） >top chunk是堆顶的chunk，堆顶指针brk位于top chunk的顶部。移动brk指针，即可扩充top chunk的大小。当top chunk大小超过128k(可配置)时，会触发malloc_trim操作，调用sbrk(-size)将内存归还操作系统。
- 当chunk足够大，fast bin和bins都不能满足要求，甚至top chunk都不能满足时，malloc会从mmap来直接使用内存映射来将页映射到进程空间，这样的chunk释放时，直接解除映射，归还给操作系统。（极限大的时候）
- Last remainder是另外一种特殊的chunk，就像top chunk和mmaped chunk一样，不会在任何bins中找到这种chunk。当需要分配一个small chunk,但在small bins中找不

到合适的chunk，如果last remainder chunk的大小大于所需要的small chunk大小，last remainder chunk被分裂成两个chunk，其中一个chunk返回给用户，另一个chunk变成新的last remainder chunk。（这个应该是fast bins中也找不到合适的时候，用于极限小的）

由之前的分析可知malloc利用chunk结构来管理内存块，malloc就是由不同大小的chunk链表组成的。malloc会给用户分配的空间的前后加上一些控制信息，用这样的方法来记录分配的信息，以便完成分配和释放工作。chunk指针指向chunk开始的地方,图中的mem指针才是真正返回给用户的内存指针。

malloc 内存分配流程

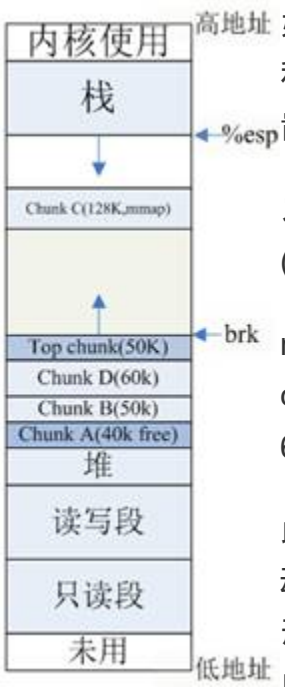
1. 如果分配内存<512字节，则通过内存大小定位到smallbins对应的index上(floor(size/8))
    - 如果smallbins[index]为空，进入步骤3
    - 如果smallbins[index]非空，直接返回第一个chunk
2. 如果分配内存>512字节，则定位到largebins对应的index上
    - 如果largebins[index]为空，进入步骤3
    - 如果largebins[index]非空，扫描链表，找到第一个大小最合适的chunk，如size=12.5K，则使用chunk B，剩下的0.5k放入unsorted_list中
3. 遍历unsorted_list，查找合适size的chunk，如果找到则返回；否则，将这些chunk都归类放到smallbins和largebins里面
4. index++从更大的链表中查找，直到找到合适大小的chunk为止，找到后将chunk拆分，并将剩余的加入到unsorted_list中
5. 如果还没有找到，那么使用top chunk
6. 或者，内存<128k，使用brk；内存>128k，使用mmap获取新内存

此外，调用free函数时，它将用户释放的内存块连接到空闲链上。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，并开始在空闲链上翻箱倒柜地检查各内存片段，对它们进行整理，将相邻的小空闲块合并成较大的内存块。

内存碎片

free释放内存时，有两种情况：

1. chunk和top chunk相邻，则和top chunk合并
2. chunk和top chunk不相邻，则直接插入到unsorted_list中



如上图示: top chunk是堆顶的chunk，堆顶指针brk位于top chunk的顶部。移动brk指针，即可扩充top chunk的大小。当top chunk大小超过128k(可配置)时，会触发malloc_trim操作，调用sbrk(-size)将内存归还操作系统。

以上图chunk分布图为例，按照glibc的内存分配策略，我们考虑下如下场景(假设brk其实地址是512k)：

malloc 40k内存，即chunkA，brk = 512k + 40k = 552k malloc 50k内存，即chunkB，brk = 552k + 50k = 602k malloc 60k内存，即chunkC，brk = 602k + 60k = 662k free chunkA。

此时，由于brk = 662k，而释放的内存是位于[512k, 552k]之间，无法通过移动brk指针，将区域内内存交还操作系统，因此，在[512k, 552k]的区域内便形成了一个内存空洞即内存碎片。 按照glibc的策略，free后的chunkA区域由于不和top chunk相邻，因此，无法和top chunk 合并，应该挂在unsorted_list链表上。

信号量和互斥量的区别？

我觉得主要区别在两方面：**第一方面是所有权的概念，第二个方面是用途。**

- 先说第一个所有权。
  解铃还须系铃人，一个锁住临界区的锁必须由上锁的线程解开，因此mutex的功能也就限制在了构造临界区上。
  对于信号量来说，任意多线程都可以对信号量执行PV操作。
- 第二个是用途
  也就是同步和互斥的用处。
  互斥很好说了，当我占有使用权的时候别人不能进入，独占式访问某段程序和内存。

同步就是**调度线程**，即一些线程生产一些线程消费，让生产和消费线程保持合理执行顺序。因为semaphore本意是信号灯，含量了通知的意味，只要是我的信号量值大于等于1，那么就可以有线程或者进程来使用。
『同步』这个词也可以拆开看，一侧是等待数据的『事件』或者『通知』，一侧是保护数据的 『临界区』，所以同步也即**同步+互斥**。信号量可以满足这两个功能，但是可以注意到两个功能的应用场景还是蛮大的，有 **do one thing and do it best** 的空间。linux 内核曾将 semaphore 作为同步原语，后面代码变得较难维护，刷了一把 mutex 变简单了不少还变快了，需要『通知』 的场景则替换为了 completion variable。

- 举个例子
  公司的咖啡机，互斥就是我一个人独占咖啡机，做完了咖啡然后撤走，咖啡机可以被别人使用。
  信号量是同步+互斥，有一个人不停地生产咖啡，每个人排队拿，如果有了就拿，没有了就阻塞等待。等有咖啡了唤醒阻塞的线程拿咖啡。
  所以更像是mutex+condition_variable
- 英文的解释

> A mutex
> is essentially the same thing as a binary semaphore and sometimes uses the same basic implementation. The differences between them are in how they are used. While a binary semaphore may be used as a mutex, a mutex is a more specific use-case, in that only the thread that locked the mutex is supposed to unlock it. This constraint makes it possible to implement some additional features in mutexes:
>
> i. Since only the thread that locked the mutex is supposed to unlock it, a mutex may store the id of thread that locked it and verify the same thread unlocks it.
> ii. Mutexes may provide priority inversion
> safety. If the mutex knows who locked it and is supposed to unlock it,
> it is possible to promote the priority of that thread whenever a higher-priority task starts waiting on the mutex.
> iii. Mutexes may also provide deletion safety, where the thread holding the mutex cannot be accidentally deleted.
> iv. Alternately, if the thread holding the mutex is deleted (perhaps due

> to an unrecoverable error), the mutex can be automatically released.
> v. A mutex may be recursive: a thread is allowed to lock it multiple times without causing a deadlock.

写时拷贝底层原理

在 Linux 系统中，调用 fork 系统调用创建子进程时，并不会把父进程所有占用的内存页复制一份，而是与父进程共用相同的内存页，而当子进程或者父进程对内存页进行修改时才会进行复制——这就是著名的 写时复制 机制。如果有多个调用者同时请求相同资源（如内存或磁盘上的数据存储），他们会共同获取相同的指针指向相同的资源，直到某个调用者试图修改资源的内容时，系统才会真正复制一份专用副本（private copy）给该调用者，而其他调用者所见到的最初的资源仍然保持不变。

传统的fork系统调用直接把所有资源复制给新创建的进程，这种实现过于简单并且效率低下，如果父进程中有很多数据的话依次复制个子进程，那么fork函数肯定是非常慢的。因此使用写时拷贝会快很多。

原理

首先要了解一下内存共享机制。不同进程的 虚拟内存地址 映射到相同的 物理内存地址，那么就实现了共享内存的机制。我们可以用用这种思想来实现写时拷贝。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。这样父进程和子进程都有了属于自己独立的页。

子进程可以执行exec()来做自己想要的功能。

栈区分配内存快还是堆区分配内存快 ？

参考链接

毫无疑问，显然从栈上分配内存更快，因为从栈上分配内存仅仅就是栈指针的移动而已

在堆区上申请与释放内存是一个相对复杂的过程，因为堆本身是需要程序员(内存分配器实现者)自己管理的，而栈是编译器来维护的，堆区的维护同样涉及内存的分配与释放，但这里的内存分配与释放显然不会像栈区那样简单，一句话，这里是**按需进行内存的分配与释放**，**本质在于堆区中每一块被分配出去的内存其生命周期都不一样**，这是由程序员决定的，我倾向于把内存动态分配释放想象成去停车场找停车位。

申请内存时底层发生了什么？

就是malloc如何实现内存管理的

Cache

参考链接

什么是cache

Cache存储器，是位于CPU和主存储器DRAM之间的一块高速缓冲存储器，规模较小，但是速度很快，通常由SRAM（静态存储器）组成。Cache的功能是提高CPU数据输入输出的速率。Cache容量小但速度快，内存速度较低但容量大，通过优化调度算法，可以让系统的性能大大改善，感觉就像是有了主存储器的内存，又有了Cache的访问速度。

工作方式

CPU在访问存储器的时候，会同时把虚拟地发送给MMU中的TLB以及Cache，CPU会在TLB中查找最终的RPN（Real Page Number），也就是真实的物理页面，如果找到了，就会返回相应的物理地址。同时，CPU通过cache编码地址中的Index，也可以很快找到相应的Cache line组，但是这个cache line 中存储的数据不一定是CPU所需要的，需要进一步检查，前面我们说了，如果TLB命中后，会返回一个真实的物理地址，将cache line中存放的地址和这个转换出来的物理地址进行比较，如果相同并且状态位匹配，那么就会发生cache命中。如果cache miss，那么CPU就需要重新从存储器中获取数据，然后再将其存放在cache line中。

多级cache存储结构

cache的速度在一定程度上同样影响着系统的性能。一般情况下cache的速度可以达到1ns，几乎可以和CPU寄存器速度媲美。但是，这就满足了人们对性能的追求了吗？并没有。当cache中没有缓存我们想要的数据的时候，依然需要漫长的等待从主存中load数据。为了进一步提升性能，引入多级cache。前面提到的cache，称为L1 cache（第一级cache）。我们在L1 cache后面连接L2 cache，在L2 cache和主存之间连接L3 cache。等级越高，速度越慢，容量越大。 但是速度和主存相比而言，依然很快。

内存屏障

参考链接

现在大多数现代计算机为了提高性能而采取乱序执行，这可能会导致程序运行不符合我们预期，内存屏障就是一类同步屏障指令，是CPU或者编译器在对内存随机访问的操作中的一个同步点，只有在此点之前的所有读写操作都执行后才可以执行此点之后的操作。

Fork函数相关知识

fork（）函数通过系统调用创建一个与原来进程几乎完全相同的进程，也就是两个进程可以做完全相同的事，但如果初始参数或者传入的变量不同，两个进程也可以做不同的事。

一个进程调用fork（）函数后，系统先给新的进程分配资源，例如存储数据和代码的空间。然后把原来的进程的所有值都复制到新的新进程中，只有少数值与原来的进程的值不同。相当于克隆了一个自己。

返回值

fork调用的一个奇妙之处就是它仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：

1. 在父进程中，fork返回新创建子进程的进程ID；
2. 在子进程中，fork返回0；
3. 如果出现错误，fork返回一个负值；

包含内容

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。

子进程与父进程之间除了代码是共享的之外，堆栈数据和全局数据均是独立的。

写时复制

fork子进程完全复制父进程的栈空间，也复制了页表，但没有复制物理页面，所以这时虚拟地址相同，物理地址也相同，但是会把父子共享的页面标记为“只读”（类似mmap的private的方式），如果父子进程一直对这个页面是同一个页面，知道其中任何一个进程要对共享的页面“写操作”，这时内核会复制一个物理页面给这个进程使用，同时修改页表。而把原来的只读页面标记为“可写”，留给另外一个进程使用。

执行顺序

fork之后内核会通过将子进程放在队列的前面，以让子进程先执行，以免父进程执行导致写时复制，而后子进程执行exec系统调用，因无意义的复制而造成效率的下降。

。正因为fork采用了这种写时复制的机制，所以fork出来子进程之后，父子进程哪个先调度呢？内核一般会先调度子进程，因为很多情况下子进程是要马上执行exec，会清空栈、堆。。这些和父进程共享的空间，加载新的代码段。。。，这就避免了“写时复制”拷贝共享页面的机会。如果父进程先调度很可能写共享页面，会产生“写时复制”的无用功。所以，一般是子进程先调度滴。

fork复制内部，为什么fork返回0？

参考链接

fork宏函数展开后代码

```
int fork(void) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_fork)); \
    if (__res >= 0) \
        return (type) __res; \
    errno = -__res; \
    return -1; \
}
```

fork是一个宏，内部关键指令就是0x80中断，调用中断就会执行相应的中断服务程序`kernel/sched.c`中的`sched_init`方法中就对0x80号中断进行了配置,就是说发生该中断就会调用`system_call`方法。会进入system_cal这个函数，然后有一个`sys_cal_table`。这个表就是一个系统调用函数表，存的都是函数指针。

```
//sys_call_table位于linux/sys.h
fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, ... };
```

根据索引找到sys_fork这个函数，能找到这个函数就是因为由寄存器eax传过来个值2，__NR_fork = 2,，索引为2的函数指针就是sys_fork。

```
sys_fork:
        call find_empty_process
        testl %eax,%eax             # 在eax中返回进程号pid。若返回负数则退出。
        js 1f
        push %gs
        pushl %esi
        pushl %edi
        pushl %ebp
        pushl %eax
        call copy_process
        addl $20,%esp               # 丢弃这里所有压栈内容。
1:        ret
```

首先`find_empty_process`找到一个可用的进程id，这个可用的进程id在`%eax`里面，接者调

用 `copy_process`

```
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
                 long ebx,long ecx,long edx,
                 long fs,long es,long ds,
                 long eip,long cs,long eflags,long esp,long ss)
{
        struct task_struct *p;
        int i;
        struct file *f;

    // 首先为新任务数据结构分配内存。如果内存分配出错，则返回出错码并退出。
    // 然后将新任务结构指针放入任务数组的nr项中。其中nr为任务号，由前面
    // find_empty_process()返回。接着把当前进程任务结构内容复制到刚申请到
    // 的内存页面p开始处。
        p = (struct task_struct *) get_free_page();
        if (!p)
                return -EAGAIN;
        task[nr] = p;
        *p = *current;        /* NOTE! this doesn't copy the supervisor stack */
    // 随后对复制来的进程结构内容进行一些修改，作为新进程的任务结构。先将
    // 进程的状态置为不可中断等待状态，以防止内核调度其执行。然后设置新进程
    // 的进程号pid和父进程号father，并初始化进程运行时间片值等于其priority值
    // 接着复位新进程的信号位图、报警定时值、会话(session)领导标志leader、进程
    // 及其子进程在内核和用户态运行时间统计值，还设置进程开始运行的系统时间start_time.
        p->state = TASK_UNINTERRUPTIBLE;
        p->pid = last_pid;              // 新进程号。也由find_empty_process()得到。
        p->father = current->pid;       // 设置父进程
        p->counter = p->priority;       // 运行时间片值
        p->signal = 0;                  // 信号位图置0
        p->alarm = 0;                   // 报警定时值(滴答数)
        p->leader = 0;                /* process leadership doesn't inherit */
        p->utime = p->stime = 0;        // 用户态时间和和心态运行时间
        p->cutime = p->cstime = 0;      // 子进程用户态和和心态运行时间
        p->start_time = jiffies;        // 进程开始运行时间(当前时间滴答数)
    // 再修改任务状态段TSS数据，由于系统给任务结构p分配了1页新内存，所以(PAGE_SIZE+
    // (long)p)让esp0正好指向该页顶端。ss0:esp0用作程序在内核态执行时的栈。另外，
    // 每个任务在GDT表中都有两个段描述符，一个是任务的TSS段描述符，另一个是任务的LDT
    // 表描述符。下面语句就是把GDT中本任务LDT段描述符和选择符保存在本任务的TSS段中。
    // 当CPU执行切换任务时，会自动从TSS中把LDT段描述符的选择符加载到ldtr寄存器中。
        p->tss.back_link = 0;
```

```
    p->tss.esp0 = PAGE_SIZE + (long) p;     // 任务内核态栈指针。
    p->tss.ss0 = 0x10;                      // 内核态栈的段选择符(与内核数据段相同)
    p->tss.eip = eip;                       // 指令代码指针
    p->tss.eflags = eflags;                 // 标志寄存器
    p->tss.eax = 0;                         // 这是当fork()返回时新进程会返回0的原因所在
    p->tss.ecx = ecx;
    p->tss.edx = edx;
    p->tss.ebx = ebx;
    p->tss.esp = esp;
    p->tss.ebp = ebp;
    p->tss.esi = esi;
    p->tss.edi = edi;
    p->tss.es = es & 0xffff;                // 段寄存器仅16位有效
    p->tss.cs = cs & 0xffff;
    p->tss.ss = ss & 0xffff;
    p->tss.ds = ds & 0xffff;
    p->tss.fs = fs & 0xffff;
    p->tss.gs = gs & 0xffff;
    p->tss.ldt = _LDT(nr);                  // 任务局部表描述符的选择符(LDT描述符在GDT中)
    p->tss.trace_bitmap = 0x80000000;       // 高16位有效
// 如果当前任务使用了协处理器，就保存其上下文。汇编指令clts用于清除控制寄存器CRO中
// 的任务已交换(TS)标志。每当发生任务切换，CPU都会设置该标志。该标志用于管理数学协
// 处理器：如果该标志置位，那么每个ESC指令都会被捕获(异常7)。如果协处理器存在标志MP
// 也同时置位的话，那么WAIT指令也会捕获。因此，如果任务切换发生在一个ESC指令开始执行
// 之后，则协处理器中的内容就可能需要在执行新的ESC指令之前保存起来。捕获处理句柄会
// 保存协处理器的内容并复位TS标志。指令fnsave用于把协处理器的所有状态保存到目的操作数
// 指定的内存区域中。
    if (last_task_used_math == current)
            __asm__("clts ; fnsave %0"::"m" (p->tss.i387));
// 接下来复制进程页表。即在线性地址空间中设置新任务代码段和数据段描述符中的基址和限长，
// 并复制页表。如果出错(返回值不是0)，则复位任务数组中相应项并释放为该新任务分配的用于
// 任务结构的内存页。
    if (copy_mem(nr,p)) {
            task[nr] = NULL;
            free_page((long) p);
            return -EAGAIN;
    }
// 如果父进程中有文件是打开的，则将对应文件的打开次数增1，因为这里创建的子进程会与父
// 进程共享这些打开的文件。将当前进程(父进程)的pwd，root和executable引用次数均增1.
```

```
    // 与上面同样的道理，子进程也引用了这些i节点。
        for (i=0; i<NR_OPEN;i++)
                if ((f=p->filp[i]))
                        f->f_count++;
        if (current->pwd)
                current->pwd->i_count++;
        if (current->root)
                current->root->i_count++;
        if (current->executable)
                current->executable->i_count++;
    // 随后GDT表中设置新任务TSS段和LDT段描述符项。这两个段的限长均被设置成104字节。
    // set_tss_desc()和set_ldt_desc()在system.h中定义。"gdt+(nr<<1)+FIRST_TSS_ENTRY"是
    // 任务nr的TSS描述符项在全局表中的地址。因为每个任务占用GDT表中2项，因此上式中
    // 要包括'(nr<<1)'.程序然后把新进程设置成就绪态。另外在任务切换时，任务寄存器tr由
    // CPU自动加载。最后返回新进程号。
        set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
        set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
        p->state = TASK_RUNNING;        /* do this last, just in case */
        return last_pid;
}
```

`copy_process`定义在`kernel/fork.c`里面，就是把父进程资源复制到子进程中，设置子进程的内存页等信息，如果父进程调用fork结束，就会返回`__res`,因为`__res`与`%eax`寄存器是绑定的，寄存器里面存的就是进程的pid，因为`copy_process`的返回值就是`last_pid`，也就是子进程pid，注意这里是父进程返回，子进程还没执行呢，但是子进程也会返回`__res`，`__res`的值就是寄存器`%eax`的值。睁大眼睛看代码力里有这么一段`p->tss.eax = 0;`,父进程已经把子进程`%eax`寄存器中的值设置为0了。明白为什么使用fork时，返回0就代表是子进程了吧。

通俗的解释，可以这样看待：“其实就相当于链表，进程形成了链表，父进程的fork函数返回的值指向子进程的进程id, 因为子进程没有子进程，所以其fork函数返回的值为0.

为什么子进程先运行?

参考链接该连接中说在多核是可以同时执行的，单核的话不确定，但是从cow来说确实应该子进程先执行。

复制进程的函数是`copy_process`，如果`copy_process`调用成功的话，那么系统会有意让新开辟的进程运行，这是因为子进程一般都会马上调用exec()函数来执行其他的任务，这样就可以避免

写是复制造成的开销，或者从另一个角度说，如果其首先执行父进程，而父进程在执行的过程中，可能会向地址空间中写入数据，那么这个时候，系统就会为子进程拷贝父进程原来的数据，而当子进程调用的时候，其紧接着执行拉exec()操作，那么此时，系统又会为子进程拷贝新的数据，这样的话，相比优先执行子程序，就进行了一次“多余”的拷贝。

锁是什么？

本质思想

并发编程中，为了保证数据操作的一致性，操作系统引入了锁机制，为了保证临界区代码的安全。锁其实是一种思想

锁的本质就是计算机中的一块内存。当这块内存空间被赋值为1的时候表示加锁了，当被赋值为0的时候表示解锁了，当然这块内存空间在哪里并不重要。多个线程抢占一个锁，就是要把这块内存赋为1。

在单核的情况下，关闭 CPU 中断，使其不能暂停当前请求而处理其他请求，从而达到赋值“锁”对应的内存空间的目的。

在多核的情况下，使用锁总线和缓存一致性技术（详情看这里），可以实现在单一时刻，只有某个CPU里面某一个核能够赋值“锁”对应的内存空间，从而达到锁的目的。

> 锁总线和缓存一致性
>
> 随着多核时代的到来，并发操作已经成了很正常的现象，操作系统必须要有一些机制和原语，以保证某些基本操作的原子性，比如处理器需要保证读一个字节或写一个字节是原子的，那么它是如何实现的呢？有两种机制：**总线锁定、缓存一致性**。
>
> CPU 和物理内存之间的通信速度远慢于 CPU 的处理速度，所以 CPU 有自己的内部缓存，根据一些规则将内存中的数据读取到内部缓存中来，以加快频繁读取的速度。那么这样就会造成cpu寄存器中的值和内存中的值出现不匹配的现象。（比如两核，i=0，第一个核i+1，第二个核取i的时候i还是0）
>
> **总线锁定机制：**在 CPU1 要做 i++ 操作的时候，其在总线上发出一个 LOCK 信号，其他处理器就不能操作缓存了该变量内存地址的缓存，也就是阻塞了其他CPU，使该处理器可以独享此共享内存。

> **缓存一致性：**当某块 CPU 对缓存中的数据进行操作了之后，就通知其他 CPU 放弃储存在它们内部的缓存，或者从主内存中重新读取

锁的开销

参考

所有锁的本质：

我们针对的是多线程环境下的锁机制，基于linux做测试。每种编程语言提供的锁机制都不太一样，不过无论如何，最终都会落实到两种机制上，**一是处理器提供的原子操作指令（现在一般是CAS—compare and swap），处理器会用轮询的方式试图获得锁，在处理器（包括多核）架构里这是必不可少的机制；二是内核提供的锁系统调用，在被锁住的时候会把当前线程置于睡眠（阻塞）状态。**

实际上我们在编程的时候并不会直接调用这两种机制，而是使用编程语言所带函数库里的锁方法，锁方法内部混合使用这两种机制。以pthread库（NPTL）的pthread_mutex来举例，一把锁本质上只是一个int类型的变量，占用4个字节内存并且内存边界按4字节对齐。加锁的时候先用trylock方法（内部使用的是CAS指令）来尝试获得锁，如果无法获得锁，则调用系统调用sys_futex来试图获得锁，这时候如果还不能获得锁，当前线程就会被阻塞。(**futex的知识**)

所以很容易得到一个结论，如果锁不存在冲突，每次获得锁和释放锁的处理器开销仅仅是CAS指令的开销，在x86-64处理器上，这个开销只比一次内存访问（无cache）高一点（大概是1.3倍）。一般的电脑上一次没有缓存的内存访问大概是十几纳秒

测试：

无冲突的时候：运行了 10 亿次，平摊到每次加锁/解锁操作大概是 14ns

锁冲突的情况：运行的结果是双核机器上消耗大约3400ns，所以锁冲突的开销大概是不冲突开销的两百倍了，相差出乎意料的大。

锁的开销

总结：锁的开销有好几部分，分别是：线程上下文切换的开销，调度器开销（把线程从睡眠改成就绪或者把就运行态改成阻塞），还有后续上下文切换带来的缓存不命中开销，跨处理器调度的开销等等。

锁的优化：

从上面可以知道，真正消耗时间的不是上锁的次数，而是锁冲突的次数。减少锁冲突的次数才是提升性能的关键。使用更细粒度的锁，可以减少锁冲突。这里说的粒度包括时间和空间，比如哈希表包含一系列哈希桶，为每个桶设置一把锁，空间粒度就会小很多——哈希值相互不冲突的访问不会导致锁冲突，这比为整个哈希表维护一把锁的冲突机率低很多。减少时间粒度也很容易理解，加锁的范围只包含必要的代码段，尽量缩短获得锁到释放锁之间的时间，最重要的是，绝对不要在锁中进行任何可能会阻塞的操作。使用读写锁也是一个很好的减少冲突的方式，读操作之间不互斥，大大减少了冲突。

假设单向链表中的插入/删除操作很少，主要操作是搜索，那么基于单一锁的方法性能会很差。在这种情况下，应该考虑使用读写锁，即 pthread_rwlock_t，这么做就允许多个线程同时搜索链表。插入和删除操作仍然会锁住整个链表。假设执行的插入和搜索操作数量差不多相同，但是删除操作很少，那么在插入期间锁住整个链表是不合适的，在这种情况下，最好允许在链表中的分离点（disjoint point）上执行并发插入，同样使用基于读写锁的方式。在两个级别上执行锁定，链表有一个读写锁，各个节点包含一个互斥锁，在插入期间，写线程在链表上建立读锁，然后继续处理。在插入数据之前，锁住要在其后添加新数据的节点，插入之后释放此节点，然后释放读写锁。删除操作在链表上建立写锁。不需要获得与节点相关的锁；互斥锁只建立在某一个操作节点之上，大大减少锁冲突的次数。

锁本身的行为也存在进一步优化的可能性，sys_futex系统调用的作用在于让被锁住的当前线程睡眠，让出处理器供其它线程使用，既然这个过程的消耗很高，也就是说如果被锁定的时间不超过这个数值的话，根本没有必要进内核加锁——释放的处理器时间还不够消耗的。sys_futex的时间消耗够跑很多次 CAS 的，也就是说，对于一个锁冲突比较频繁而且平均锁定时间比较短的系统，一个值得考虑的优化方式是先循环调用 CAS 来尝试获得锁（这个操作也被称作自旋锁），在若干次失败后再进入内核真正加锁。当然这个优化只能在多处理器的系统里起作用（得有另一个处理器来解锁，否则自旋锁无意义）。在glibc的pthread实现里，通过对pthread_mutex设置PTHREAD_MUTEX_ADAPTIVE_NP属性就可以使用这个机制。

读多写一的情况用double buffer

> 注：CAS指令是线程数据同步的原子指令。

什么是futex

linux的pthreads mutex采用futex实现

Futex 是 Fast Userspace Mutexes 的缩写，现在锁的机制一般使用这个，内核态和用户态的混合机制。其设计思想其实不难理解。

在传统的 Unix 系统中，System V IPC（inter process communication），如 semaphores，msgqueues，sockets 等进程间同步机制都是对一个**内核对象**操作来完成的，这个内核对象对要同步的进程都是可见的，其提供了共享的状态信息和原子操作，用来管理互斥锁并且通知阻塞的进程。当进程间要同步的时候必须要通过系统调用（如semop()）在内核中完成。比如进程A要进入临界区，先去内核查看这个对象，有没有别的进程在占用这个临界区，出临界区的时候，也去内核查看这个对象，有没有别的进程在等待进入临界区，然后根据一定的策略唤醒等待的进程。同时经研究发现，很多同步是无竞争的，即某个进程进入互斥区，到再从某个互斥区出来这段时间，常常是没有进程也要进这个互斥区或者请求同一同步变量的。但是在这种情况下，这个进程也要陷入内核去看看有没有人和它竞争，退出的时侯还要陷入内核去看看有没有进程等待在同一同步变量上，有的话需要唤醒等待的进程。这些不必要的系统调用(或者说内核陷入)造成了大量的性能开销。为了解决这个问题，Futex就应运而生。

为了解决上述这个问题，Futex 就应运而生，Futex 是一种用户态和内核态混合的同步机制。首先，同步的进程间通过 mmap 共享一段内存，futex 变量就位于这段共享的内存中且操作是原子的，当进程尝试进入互斥区或者退出互斥区的时候，先去查看共享内存中的 futex 变量，如果没有竞争发生，就不用再执行系统调用了。当通过访问 futex 变量后进程发现有竞争发生，则还是得执行系统调用去完成相应的处理（wait 或者 wake up）。简单的说，futex 就是通过在用户态的检查，（motivation）如果了解到没有竞争就不用陷入内核了，大大提高了 low-contention 时候的效率。

mutex 是在 futex 的基础上用的内存共享变量来实现的，如果共享变量建立在进程内，它就是一个线程锁，如果它建立在进程间共享内存上，那么它是一个进程锁。pthread_mutex_t 中的 `_lock` 字段用于标记占用情况，先使用CAS判断 `_lock` 是否占用，若未占用，直接返回。否则，通过 `__lll_lock_wait_private` 调用 `SYS_futex` 系统调用迫使线程进入沉睡。 CAS是用户态的 CPU 指令，若无竞争，简单修改锁状态即返回，非常高效，只有发现竞争，才通过系统调用陷入内核态。所以，FUTEX是一种用户态和内核态混合的同步机制，它保证了低竞争情况下的锁获取效率。

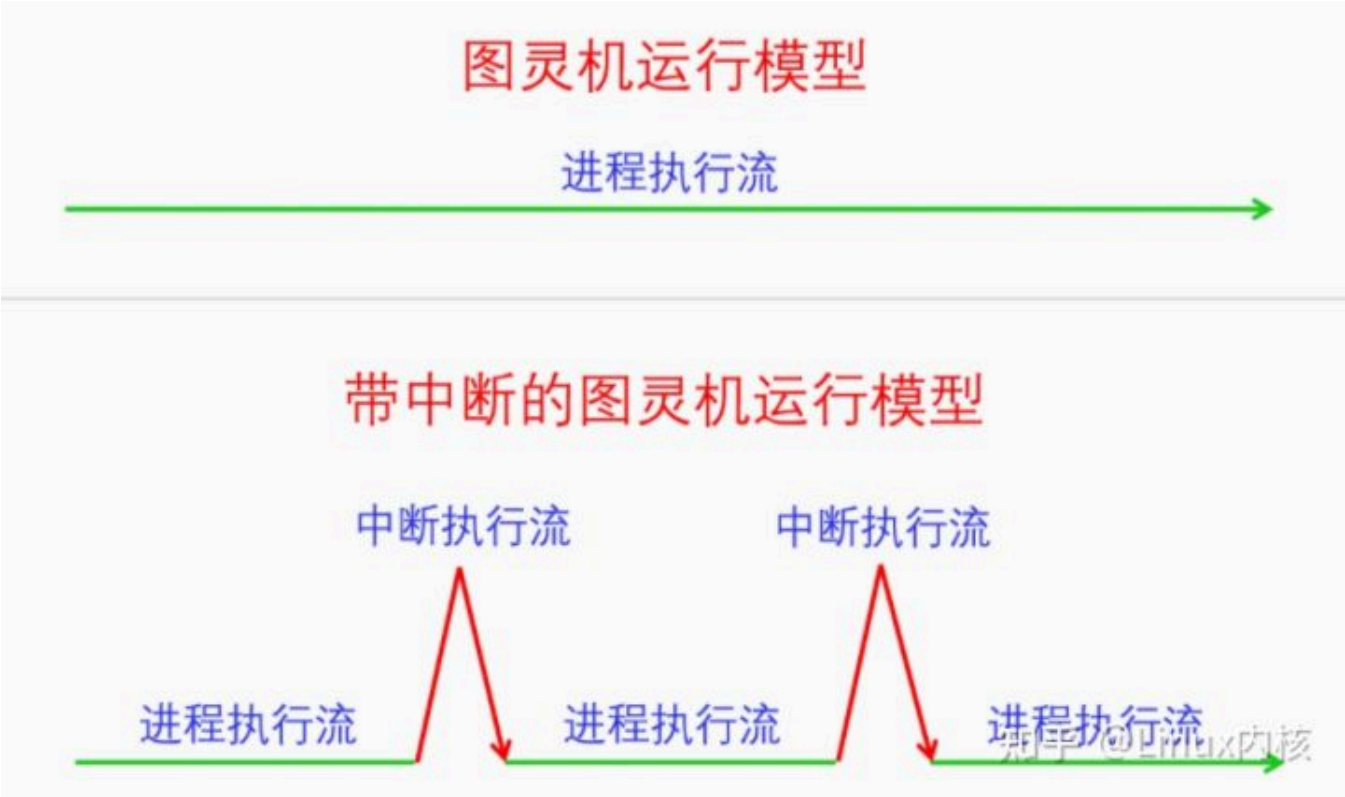
中断

参考链接

中断基本原理

中断定义

中断机制：CPU在执行指令时，收到某个中断信号转而去执行预先设定好的代码，然后再返回到原指令流中继续执行，这就是中断机制。



中断作用

外设异步通知CPU：外设发生了什么事情或者完成了什么任务或者有什么消息要告诉CPU，都可以异步给CPU发通知。例如，网卡收到了网络包，磁盘完成了IO任务，定时器的间隔时间到了，都可以给CPU发中断信号。

CPU之间发送消息：在SMP系统中，一个CPU想要给另一个CPU发送消息，可以给其发送IPI(处理器间中断)。

处理CPU异常：CPU在执行指令的过程中遇到了异常会给自己发送中断信号来处理异常。例如，做整数除法运算的时候发现被除数是0，访问虚拟内存的时候发现虚拟内存没有映射到物理内存上。

实现系统调用：早期的系统调用就是靠中断指令来实现的，后期虽然开发了专用的系统调用

指令，但是其基本原理还是相似的。

中断产生

中断信号的产生有以下4个来源：

1.外设。外设产生的中断信号是异步的，一般也叫做硬件中断(注意硬中断是另外一个概念)。硬件中断按照是否可以屏蔽分为可屏蔽中断和不可屏蔽中断。例如，网卡、磁盘、定时器都可以产生硬件中断。

2.CPU。这里指的是一个CPU向另一个CPU发送中断，这种中断叫做IPI(处理器间中断)。IPI也可以看出是一种特殊的硬件中断，因为它和硬件中断的模式差不多，都是异步的

3.CPU异常。CPU在执行指令的过程中发现异常会向自己发送中断信号，这种中断是同步的，一般也叫做软件中断(注意软中断是另外一个概念)。CPU异常按照是否需要修复以及是否能修复分为3类：1.陷阱(trap)，不需要修复，中断处理完成后继续执行下一条指令，2.故障(fault)，需要修复也有可能修复，中断处理完成后重新执行之前的指令，3.中止(abort)，需要修复但是无法修复，中断处理完成后，进程或者内核将会崩溃。例如，缺页异常是一种故障，所以也叫缺页故障，缺页异常处理完成后会重新执行刚才的指令。

4.中断指令。直接用CPU指令来产生中断信号，这种中断和CPU异常一样是同步的，也可以叫做软件中断。例如，中断指令int 0x80可以用来实现系统调用。

中断信号的4个来源正好对应着中断的4个作用。前两种中断都可以叫做硬件中断，都是异步的；后两种中断都可以叫做软件中断，都是同步的。很多书上也把硬件中断叫做中断(异步中断)，把软件中断叫做异常(同步中断)。

> **硬件中断、软件中断，硬中断、软中断是不同的概念，分别指的是中断的来源和中断的处理方式。**

中断处理

有了中断之后，CPU就分为两个执行场景了，进程执行场景(process context)和中断执行场景(interrupt context)。进程的执行是进程执行场景，同步中断的处理也是进程执行场景，异步中断的处理是中断执行场景。可能有的人会对同步中断的处理是进程执行场景感到疑惑，但是这也很好理解，因为同步中断处理是和当前指令相关的，可以看做是进程执行的一部分。而异步中断的处理和当前指令没有关系，所以不是进程执行场景。

进程执行场景和中断执行场景有两个区别：一是进程执行场景是可以调度、可以休眠的，而中断执行场景是不可以调度不可用休眠的；二是在进程执行场景中是可以接受中断信号的，而在中断执行场景中是屏蔽中断信号的。所以如果中断执行场景的执行时间太长的话，就会影响我们对新的中断信号的响应性，所以我们需要尽量缩短中断执行场景的时间。

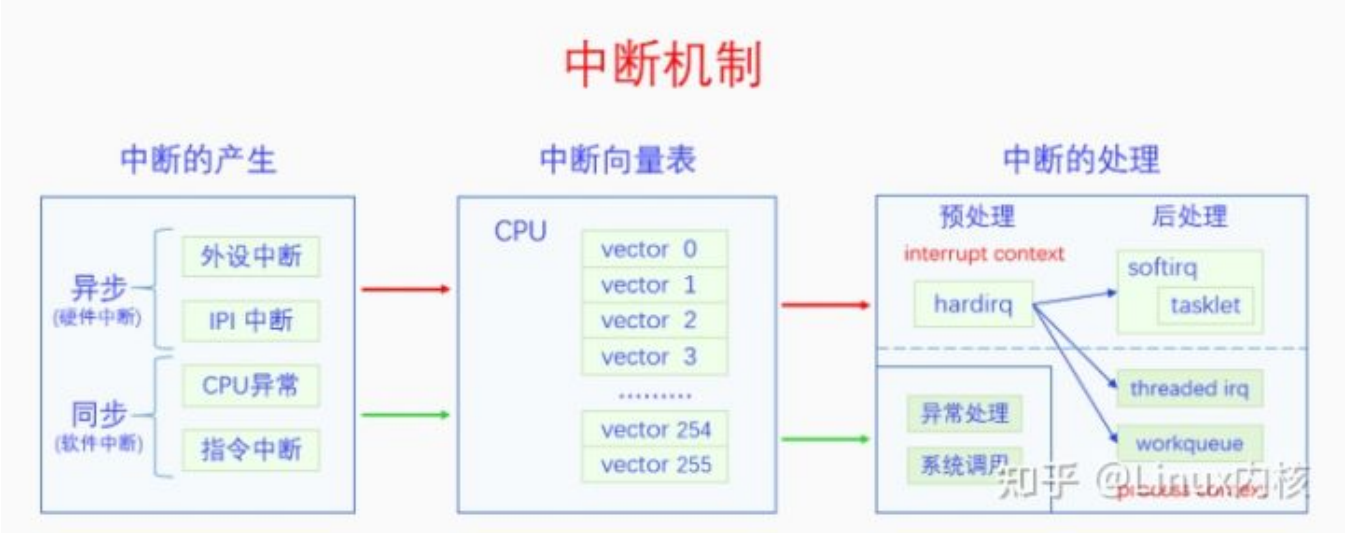
由于同步中断是软件产生的因此可以看做是进程执行的一部分，但是硬件中断在执行中断处理程序的时候会屏蔽其他中断序号，因此需要①立即快速处理。②如果比较耗时可以先预处理然后再完全处理。

中断向量号

不同的中断信号需要有不同的处理方式，那么系统是怎么区分不同的中断信号呢？是靠中断向量号。每一个中断信号都有一个中断向量号，中断向量号是一个整数。CPU收到一个中断信号会根据这个信号的中断的向量号去查询中断向量表，根据向量表里面的指示去调用相应的处理函数。

中断信号和中断向量号是如何对应的呢？对于CPU异常来说，其向量号是由CPU架构标准规定的。对于外设来说，其向量号是由设备驱动动态申请的。对于IPI中断和指令中断来说，其向量号是由内核规定的。

中断框架结构



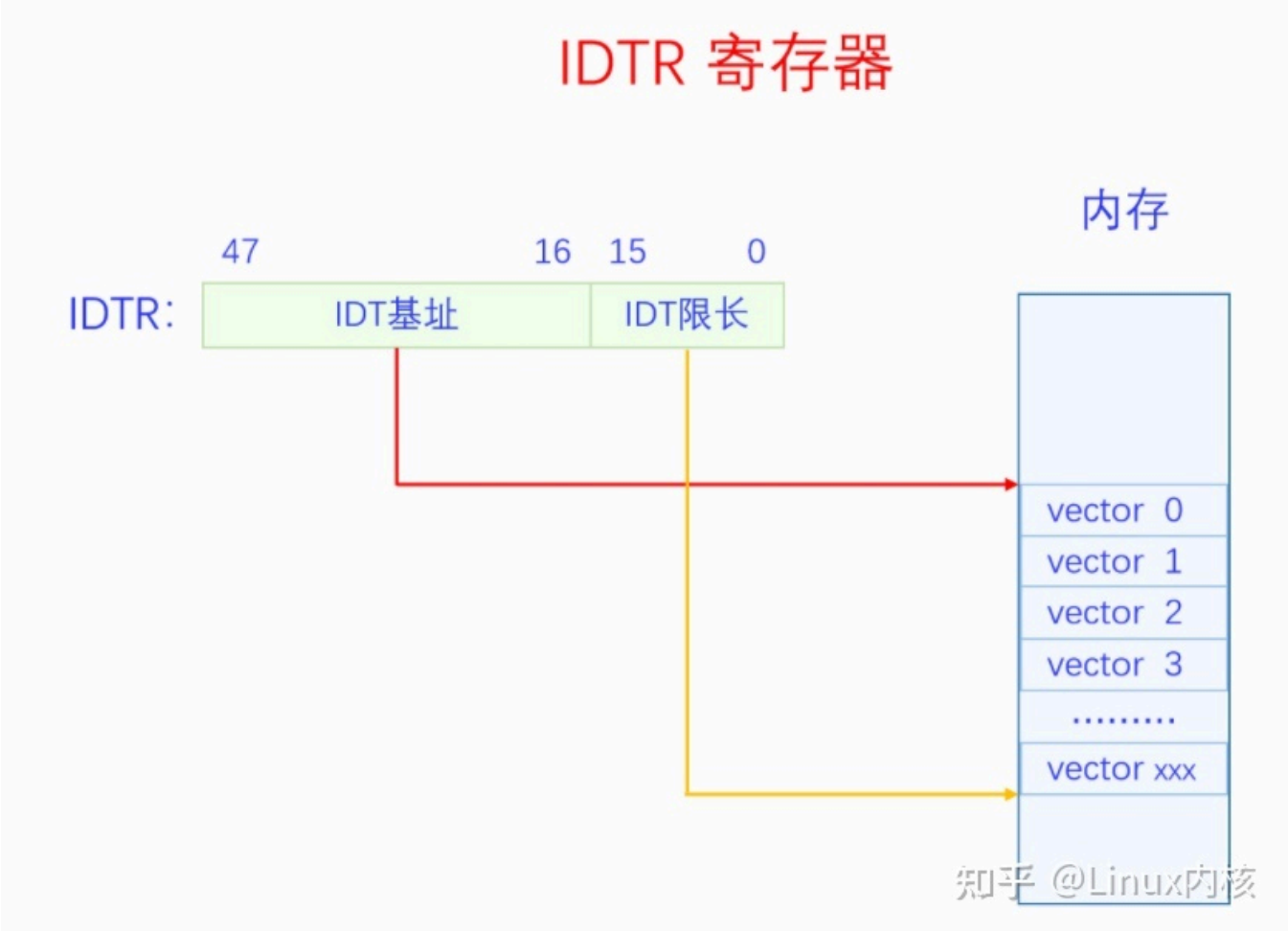
中断流程

保存现场

CPU收到中断信号后会首先把一些数据push到内核栈上，保存的数据是和当前执行点相关的，这样中断完成后就可以返回到原执行点。如果CPU当前处于用户态，则会先切换到内核态，把用户栈切换为内核栈再去保存数据

查找向量表

保存完被中断程序的信息之后，就要去执行中断处理程序了。CPU会根据当前中断信号的向量号去查询中断向量表找到中断处理程序。CPU是如何获得当前中断信号的向量号的呢，如果是CPU异常可以在CPU内部获取，如果是指令中断，在指令中就有向量号，如果是硬件中断，则可以从中断控制器中获取中断向量号。那CPU又是怎么找到中断向量表呢，是通过IDTR寄存器。如下



CPU现在已经把被中断的程序现场保存到内核栈上了，又得到了中断向量号，然后就根据中断向量号从中断向量表中找到对应的门描述符，对描述符做一番安全检查之后，CPU就开始执行中断处理函数。

中断处理

硬中断(hardirq)

硬件中断的中断处理和软件中断有一部分是相同的，有一部分却有很大的不同。对于IPI中断和per CPU中断，其设置是和软件中断相同的，都是一步到位设置到具体的处理函数。但是对于余下的外设中断，只是设置了入口函数，并没有设置具体的处理函数，而且是所有的外设中断的处理函数都统一到了同一个入口函数。然后在这个入口函数处会调用相应的irq描述符的handler函数，这个handler函数是中断控制器设置的。中断控制器设置的这个handler函数会处理与这个中断控制器相关的一些事物，然后再调用具体设备注册的irqaction的handler函数进行具体的中断处理。

对于外设中断为什么要采取这样的处理方式呢？有两个原因，1是因为外设中断和中断控制器相关联，这样可以统一处理与中断控制器相关的事物，2是因为外设中断的驱动执行比较晚，有些设备还是可以热插拔的，直接把它们放到中断向量表上比较麻烦。有个irq_desc这个中间层，设备驱动后面只需要调用函数request_irq来注册ISR，只处理与设备相关的业务就可以了，而不用考虑和中断控制器硬件相关的处理。

软中断(softirq)

软中断是把中断处理程序分成了两段：前一段叫做硬中断，执行驱动的ISR，处理与硬件密切相关的事，在此期间是禁止中断的；后一段叫做软中断，软中断中处理和硬件不太密切的事物，在此期间是开中断的，可以继续接受硬件中断。软中断的设计提高了系统对中断的响应性。下面我们先说软中断的执行时机，然后再说软中断的使用接口。

软中断也是中断处理程序的一部分，是在ISR执行完成之后运行的，在ISR中可以向软中断中添加任务，然后软中断有事要做就会运行了。有些时候当软中断过多，处理不过来的时候，也会唤醒ksoftirqd/x线程来执行软中断。

所有软中断的处理函数都是在系统启动的初始化函数里面用open_softirq接口设置的。raise_softirq一般是在硬中断或者软中断中用来往软中断上push work使得软中断可以被触发执行或者继续执行。

Linux零拷贝技术

splice( )函数

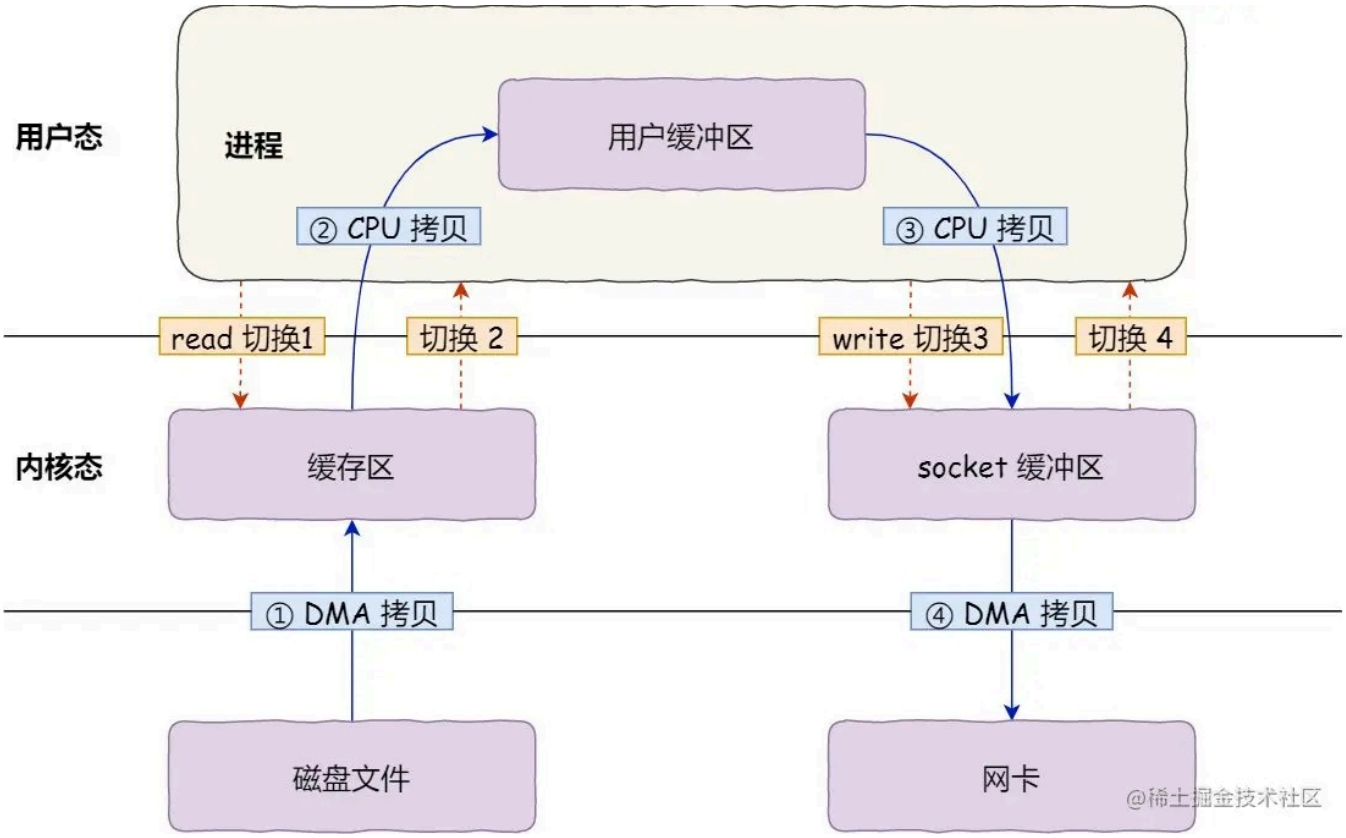
tee( )函数

概述：

零拷贝（Zero-Copy）是一种 `I/O` 操作优化技术，可以快速高效地将数据从文件系统移动到网络接口，而不需要将其从内核空间复制到用户空间。其在 `FTP` 或者 `HTTP` 等协议中可以显著地提升性能。

由来：

如果服务端要提供文件传输的功能，我们能想到的最简单的方式是：将磁盘上的文件读取出来，然后通过网络协议发送给客户端。传统 I/O 的工作方式是，数据读取和写入是从用户空间到内核空间的复制，而内核空间的数据是通过操作系统层面的 I/O 接口从磁盘读取或写入。其过程如下图所示：



可以想想一下这个过程。服务器读从磁盘读取文件的时候，发生一次系统调用，产生用户态到内核态的转换，将磁盘文件拷贝到内核的内存中。然后将位于内核内存中的文件数据拷贝到用户的缓冲区中。用户应用缓冲区需要将这些数据发送到socket缓冲区中，进行一次用户态到内核态的转换，复制这些数据。此时这些数据在内核的socket的缓冲区中，在进行一次拷贝放到网卡上发送出去。

所以整个过程一共进行了四次拷贝，四次内核和用户态的切换。这种简单又传统的文件传输方式，存在冗余的上文切换和数据拷贝，在高并发系统里是非常糟糕的，多了很多不必要的开销，会严重影响系统性能。

零拷贝原理

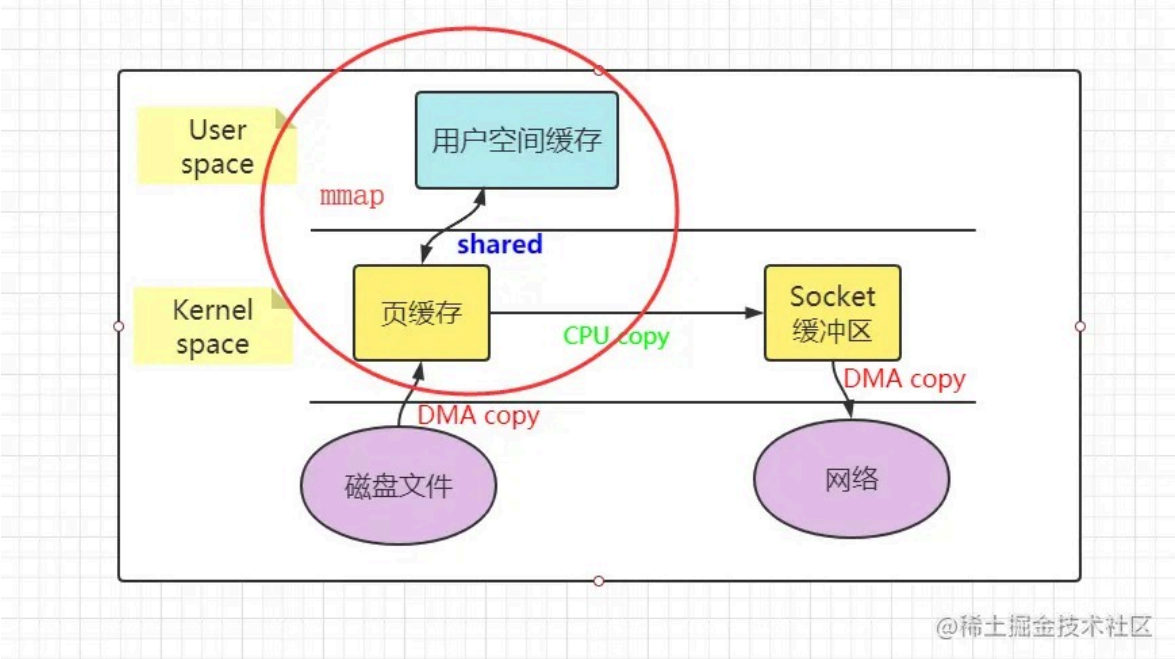
零拷贝主要是用来解决操作系统在处理 I/O 操作时，频繁复制数据的问题。关于零拷贝主要技术有 `mmap+write`、`sendfile`和`splice`等几种方式。

> DMA 技术很容易理解，本质上，DMA 技术就是我们在主板上放一块独立的芯片。在进行内存和 I/O 设备的数据传输的时候，我们不再通过 CPU 来控制数据传输，而直接通过 DMA 控制器（DMA Controller，简称 DMAC）。这块芯片，我们可以认为它其实就是一个协处理器（Co-Processor）。DMAC 的价值在如下情况中尤其明显：当我们要传输的数据

> 特别大、速度特别快，或者传输的数据特别小、速度特别慢的时候。

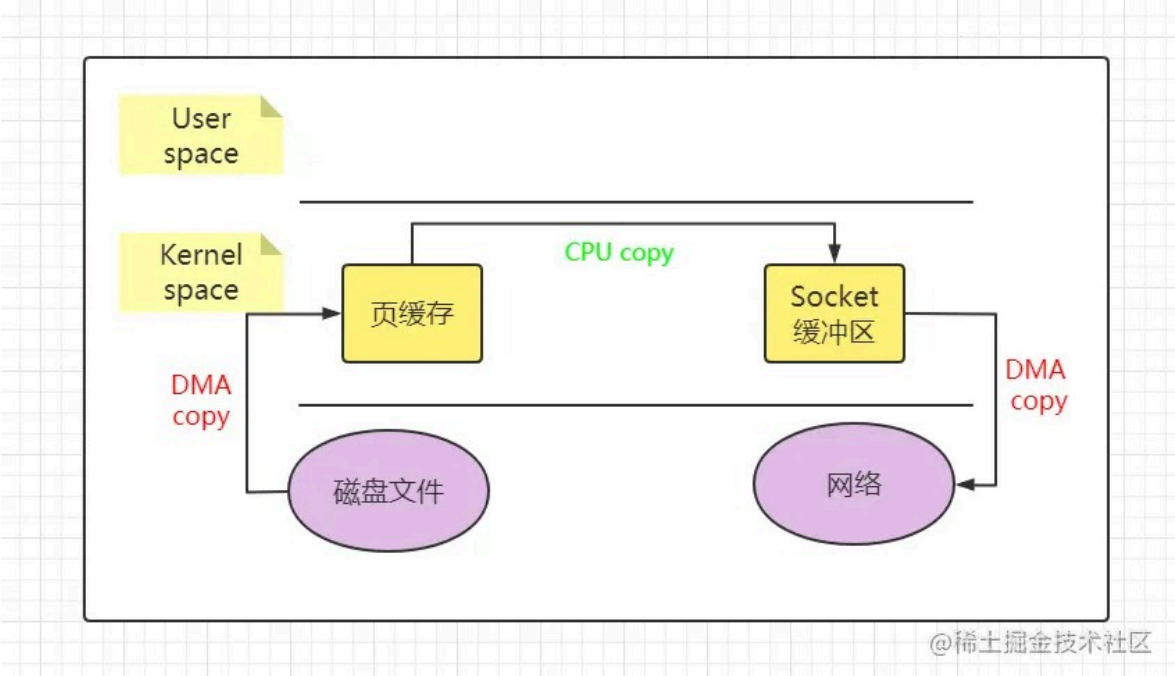
看完下图会发现其实零拷贝就是少了CPU拷贝这一步，磁盘拷贝还是要有的

- mmap/write 方式



把数据读取到内核缓冲区后，应用程序进行写入操作时，直接把内核的`Read Buffer`的数据复制到`Socket Buffer`以便写入，这次内核之间的复制也是需要CPU的参与的。

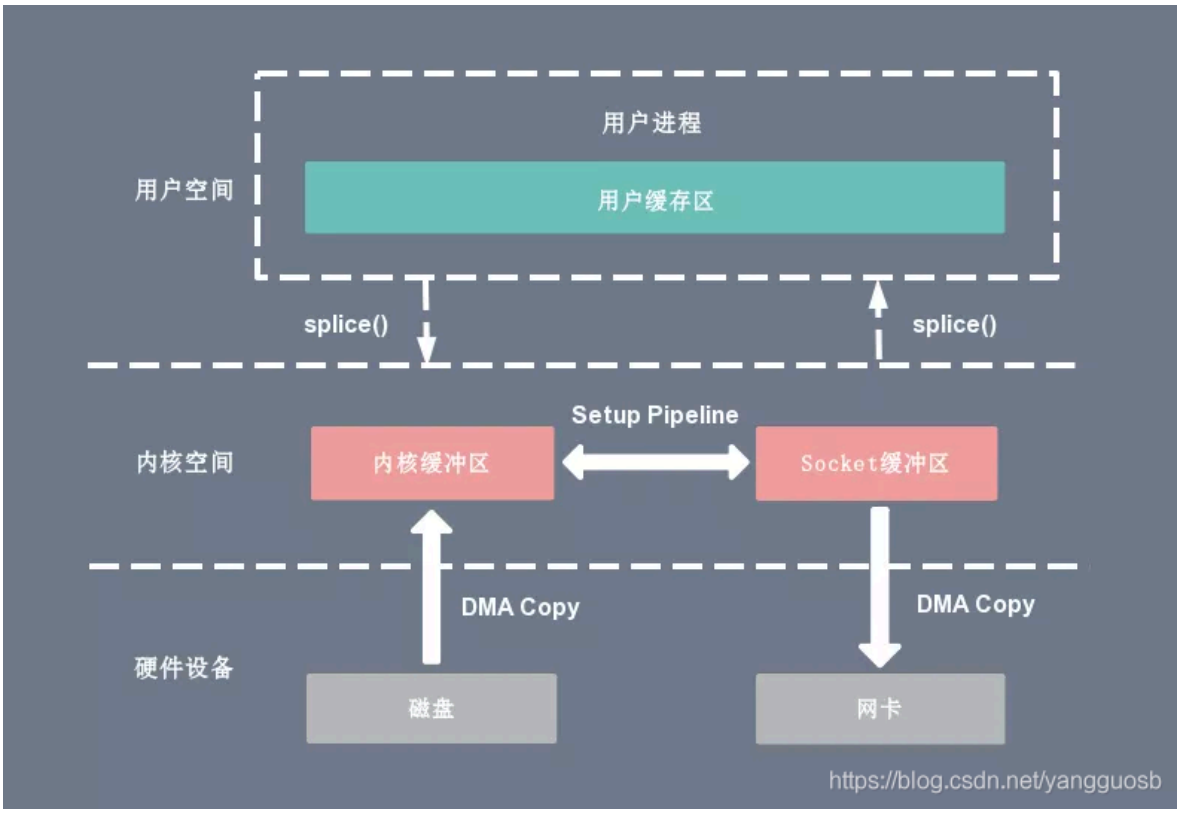
- sendfile 方式



可以看到使用sendfile后，没有用户空间的参与，一切操作都在内核中进行。但是还

是需要1次拷贝

- 带有 scatter/gather 的 sendfile方式
  Linux 2.4 内核进行了优化，提供了带有 `scatter/gather` 的 sendfile 操作，这个操作可以把最后一次 `CPU COPY` 去除。其原理就是在内核空间 Read BUffer 和 Socket Buffer 不做数据复制，而是将 Read Buffer 的内存地址、偏移量记录到相应的 Socket Buffer 中，这样就不需要复制。其本质和虚拟内存的解决方法思路一致，就是内存地址的记录。
  注意： sendfile适用于文件数据到网卡的传输过程，并且用户程序对数据没有修改的场景；
- splice 方式



其实就是CPU 在内核空间的读缓冲区（read buffer）和网络缓冲区（socket buffer）之间建立管道（pipeline）直接把数据传过去了，不去要CPU复制了

常见的零拷贝实现

mmap + write

mmap 将内核中读缓冲区（read buffer）的地址与用户空间的缓冲区（user buffer）进行映射，从而实现内核缓冲区与应用程序内存的共享，省去了将数据从内核读缓冲区（read buffer）拷贝

到用户缓冲区（user buffer）的过程，整个拷贝过程会发生 4 次上下文切换，1 次 CPU 拷贝和 2 次 DMA 拷贝。

sendfile()

通过 sendfile 系统调用，数据可以直接在内核空间内部进行 I/O 传输，从而省去了数据在用户空间和内核空间之间的来回拷贝，sendfile 调用中 I/O 数据对用户空间是完全不可见的，整个拷贝过程会发生 2 次上下文切换，1 次 CPU 拷贝和 2 次 DMA 拷贝。

splice()

在内核空间的读缓冲区（read buffer）和网络缓冲区（socket buffer）之间建立管道（pipeline），从而避免了两者之间的 CPU 拷贝操作，2 次上下文切换，0 次 CPU 拷贝以及 2 次 DMA 拷贝。

写时复制

通过尽量延迟产生私有对象中的副本，写时复制最充分地利用了稀有的物理资源。

写时拷贝底层原理

在 Linux 系统中，调用 fork 系统调用创建子进程时，并不会把父进程所有占用的内存页复制一份，而是与父进程共用相同的内存页，而当子进程或者父进程对内存页进行修改时才会进行复制——这就是著名的 写时复制 机制。如果有多个调用者同时请求相同资源（如内存或磁盘上的数据存储），他们会共同获取相同的指针指向相同的资源，直到某个调用者试图修改资源的内容时，系统才会真正复制一份专用副本（private copy）给该调用者，而其他调用者所见到的最初的资源仍然保持不变。

传统的fork系统调用直接把所有资源复制给新创建的进程，这种实现过于简单并且效率低下，如果父进程中有很多数据的话依次复制个子进程，那么fork函数肯定是非常慢的。因此使用写时拷贝会快很多。

原理

首先要了解一下内存共享机制。不同进程的 虚拟内存地址 映射到相同的 物理内存地址，那么就实现了共享内存的机制。我们可以用用这种思想来实现写时拷贝。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。这样父进程和子进程都有了属于自己独立的页。

子进程可以执行exec()来做自己想要的功能。

什么是DMA？

冯.诺依曼结构

冯.诺依曼提出了计算机“存储程序”的计算机设计理念，即将计算机指令进行编码后存储在计算机的存储器中，需要的时候可以顺序地执行程序代码，从而控制计算机运行，这就是冯.诺依曼计算机体系的开端。

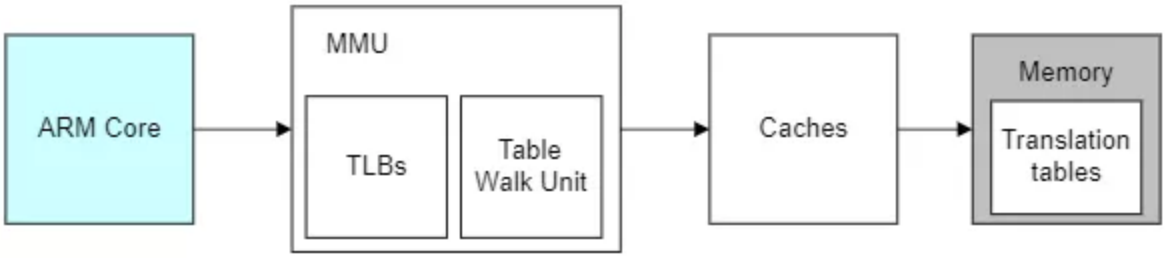
核心设计思想主要体现在如下三个方面：

- 程序、数据的最终形态都是二进制编码，程序和数据都是以二进制方式存储在存储器中的，二进制编码也是计算机能够所识别和执行的编码。（可执行二进制文件：.bin文件）
- 程序、数据和指令序列，都是事先存在主（内）存储器中，以便于计算机在工作时能够高速地从存储器中提取指令并加以分析和执行。
- 确定了计算机的五个基本组成部分：运算器、控制器、存储器、输入设备、输出设备

Linux内存管理

参考链接

CPU访问内存的过程



Linux任务调度

https://ty-chen.github.io/linux-kernel-schedule/

DMA等