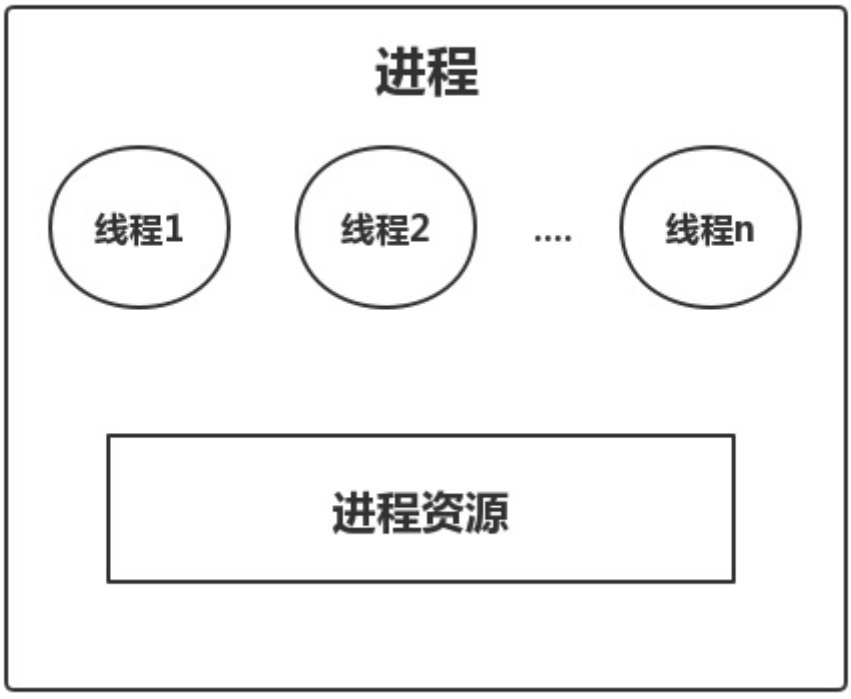
1. 为什么线程调度开销比进程调度小

here

协程与线程here

线程调度与进程调度

线程是调度的最小单位，进程是资源分配的最小单位



在Linux中，其实进程线程都是用`struct task_struct`来描述的

```
struct task_struct {
    ...
    struct mm_struct *mm;    /*内存资源*/
    ...
}
```

既然都是用`struct task_struct`描述，那么进程和线程的关系怎么体现？资源指针！例如上面代码块描述的结构成员`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

或许你有这样的疑问，既然在linux实现上，线程都是独立的`struct task_struct`，因此线程调度和进程调度这样区分

> 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

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

对于第一条，`进程调度`和`线程调度`其实都是必须的，二者差异主要在第二条。

再看回我们对“进程调度”和“线程调度”的定义，有没觉得灵光一闪？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。

线程上下文切换:

- 当两个线程不是属于同一个进程，则切换的过程就跟进程上下文切换一样；
- **当两个线程是属于同一个进程，因为虚拟内存是共享的，所以在切换时，虚拟内存这些资源就保持不动，只需要切换线程的私有数据、寄存器等不共享的数据**；

协程调度与线程调度

协程切换比线程切换快主要有两点：

（1）协程切换**完全在用户空间进行**，线程切换涉及**特权模式切换，需要在内核空间完成**；

（2）协程切换相比线程切换**做的事情更少**。

协程调度

关键:用户态，内核态

协程切换只涉及基本的**CPU上下文切换**，所谓的 CPU 上下文，就是一堆寄存器，里面保存了 CPU运行任务所需要的信息：从哪里开始运行（%rip：指令指针寄存器，标识 CPU 运行的下一条指令），栈顶的位置（**%rsp：** 是堆栈指针寄存器，通常会指向栈顶位置），当前栈帧在哪（%rbp 是栈帧指针，用于标识当前栈帧的起始位置）以及其它的CPU的中间状态或者结果（%rbx，%r12，%r13，%14，%15 等等）。协程切换非常简单，就是把**当前协程的 CPU 寄存器状态保存起来，然后将需要切换进来的协程的 CPU 寄存器状态加载的 CPU 寄存器上**就 ok 了。而且**完全在用户态进行**，一般来说一次协程上下文切换最多就是**几十ns** 这个量级。

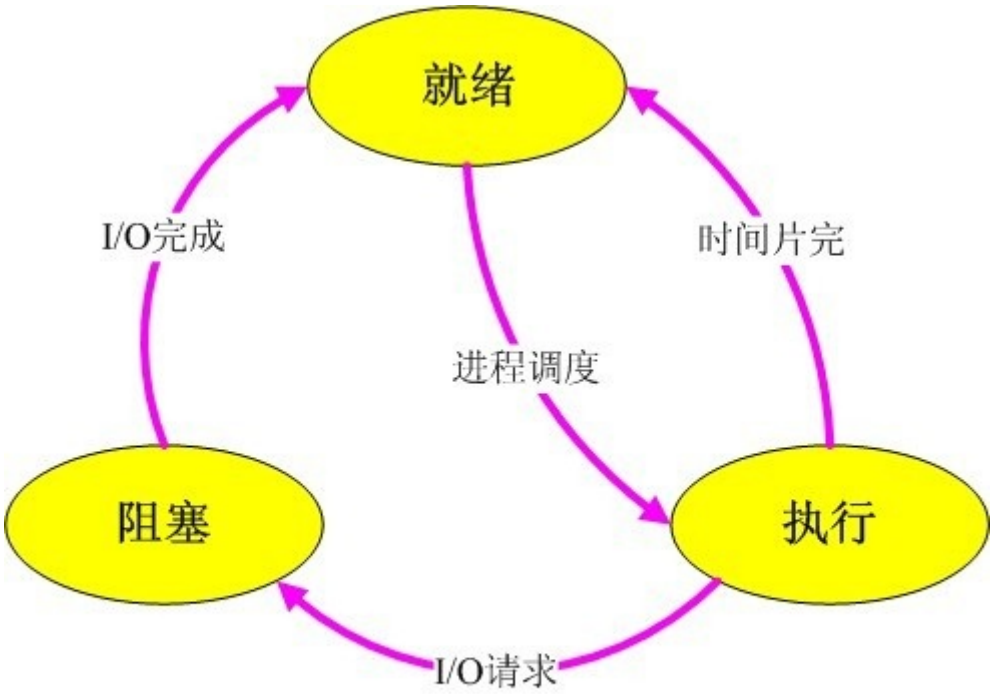
线程调度

系统内核调度的对象是线程，因为线程是调度的基本单元（进程是资源拥有的基本单元，进程的切换需要做的事情更多，这里占时不讨论进程切换），而**线程的调度只有拥有最高权限的内核空间才可以完成**，所以线程的切换涉及到**用户空间和内核空间的切换**，也就是特权模式切换，然后需要操作系统调度模块完成**线程调度（task *struct）**，而且除了和协程相同基本的 CPU 上下文，还有**线程私有的栈和寄存器**等，说白了就是上下文比协程多一些，其实简单比较下 task_strcut 和 任何一个协程库的 coroutine 的 struct 结构体大小就能明显区分出来。而且特权模式切换的开销确实不小，随便搜一组测试数据 [3]，随便算算都比协程切换开销大很多。

2. 进程，线程，协程的区别

1. **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。**



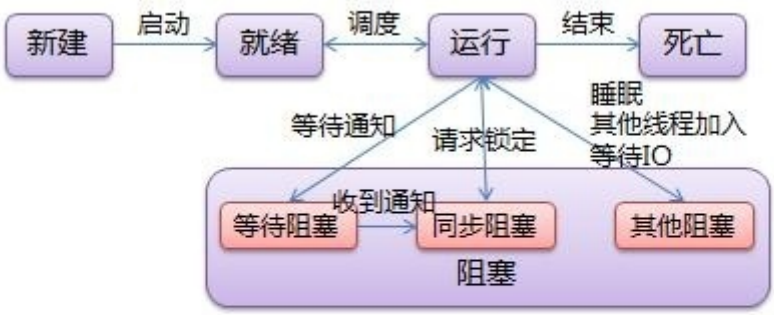
进程状态图

进程间的通信

- 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)

2. **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。



线程状态图

进程和线程的区别

1. 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；

- 系统开销：在创建或撤消进程时，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消线程时的开销。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，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，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。

2. 联系:

- **一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程**；
- 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。不同进程的线程间要利用消息通信的办法实现同步。

3. **协程(用户级线程)**

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。

线程和协程

- **极高的执行效率**：因为**子程序切换不是线程切换**，而是**由程序自身控制**，因此，**没有线程切换的开销**，和多线程比，线程数量越多，协程的性能优势就越明显；
- **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。

3. 进程控制块(PCB)

原文链接: https://mp.weixin.qq.com/s/YXl6WZVzRKCfxzerJWyfrg

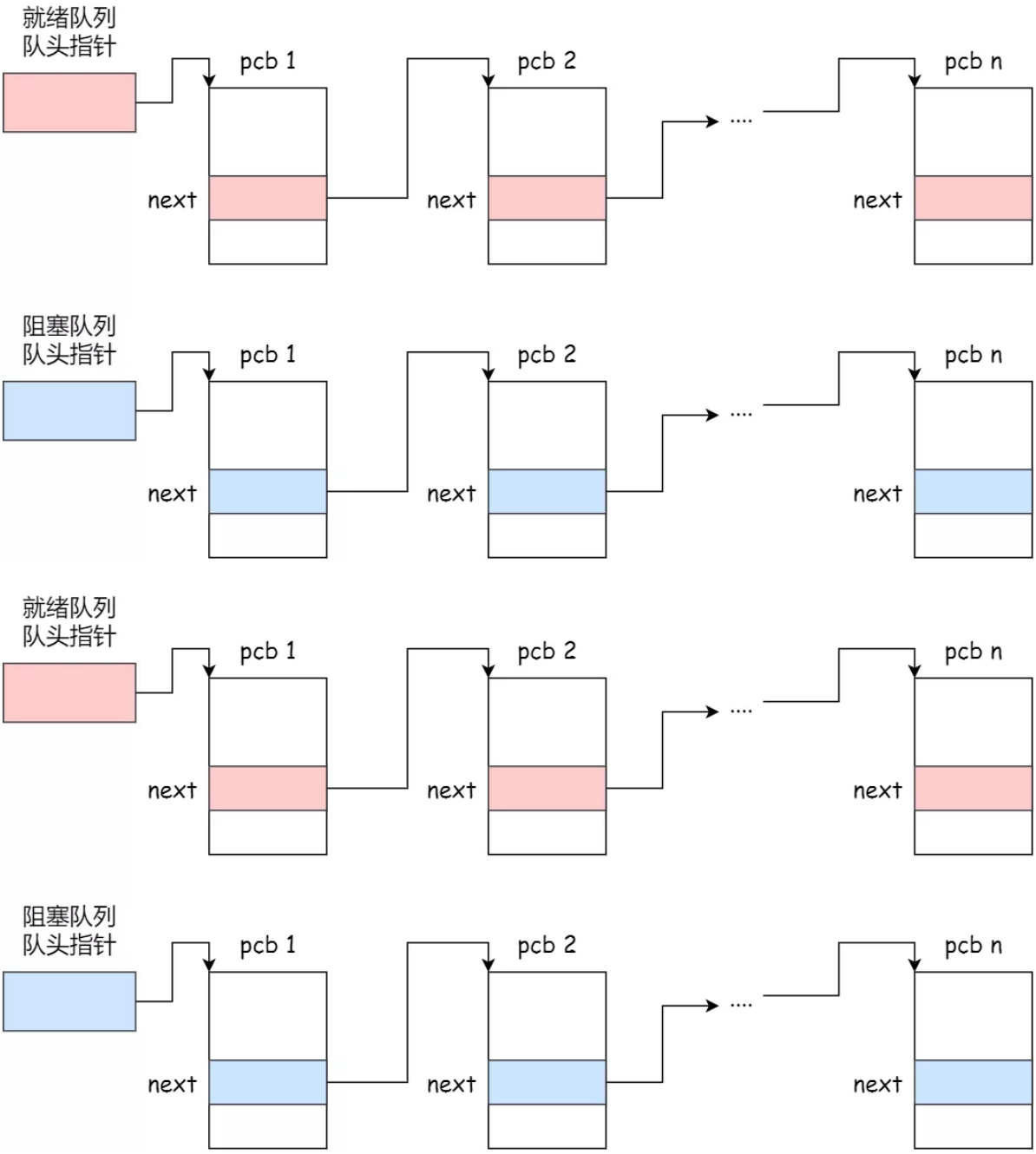
*进程控制块**（process control block，PCB*）数据结构来描述进程的。**PCB 是进程存在的唯一标识**，这意味着一个进程的存在，必然会有一个 PCB，如果进程消失了，那么 PCB 也会随之消失。

PCB 具体包含什么信息呢?

1. **进程描述信息：**
    - 进程标识符：标识各个进程，每个进程都有一个并且唯一的标识符；
    - 用户标识符：进程归属的用户，用户标识符主要为共享和保护服务；
2. **进程控制和管理信息：**
    - 进程当前状态，如 new、ready、running、waiting 或 blocked 等；
    - 进程优先级：进程抢占 CPU 时的优先级；
3. **资源分配清单：**
    - 有关内存地址空间或虚拟地址空间的信息，所打开文件的列表和所使用的 I/O 设备信息。
4. **CPU 相关信息：**
    - CPU 中各个寄存器的值，当进程被切换时，CPU 的状态信息都会被保存在相应的 PCB 中，以便进程重新执行时，能从断点处继续执行。

通常是通过**链表**的方式进行组织，把具有**相同状态的进程链在一起，组成各种队列**。比如：

- 将所有处于就绪状态的进程链在一起，称为**就绪队列**；
- 把所有因等待某事件而处于等待状态的进程链在一起就组成各种**阻塞队列**；
- 另外，对于运行队列在单核 CPU 系统中则只有一个运行指针了，因为单核 CPU 在某个时间，只能运行一个程序。



4. 进程的控制

原文链接: https://mp.weixin.qq.com/s/YXl6WZVzRKCfxzerJWyfrg

1. **创建进程**

操作系统允许一个进程创建另一个进程，而且允许子进程继承父进程所拥有的资源，当子进程被终止时，其在父进程处继承的资源应当还给父进程。同时，终止父进程时同时也会终止其所有的子进程。

创建进程的过程如下：

- 为新进程分配一个唯一的进程标识号，并申请一个空白的 PCB，PCB 是有限的，若申请失败则创建失败；
- 为进程分配资源，此处如果资源不足，进程就会进入等待状态，以等待资源；
- 初始化 PCB；
- 如果进程的调度队列能够接纳新进程，那就将进程插入到就绪队列，等待被调度运行；

2. **终止进程**

进程可以有 3 种终止方式：正常结束、异常结束以及外界干预（信号 `kill` 掉）。

终止进程的过程如下：

- 查找需要终止的进程的 PCB；
- 如果处于执行状态，则立即终止该进程的执行，然后将 CPU 资源分配给其他进程；
- 如果其还有子进程，则应将其所有子进程终止；
- 将该进程所拥有的全部资源都归还给父进程或操作系统；
- 将其从 PCB 所在队列中删除；

3. **阻塞进程**

当进程需要等待某一事件完成时，它可以调用阻塞语句把自己阻塞等待。而一旦被阻塞等待，它只能由另一个进程唤醒。

阻塞进程的过程如下：

- 找到将要被阻塞进程标识号对应的 PCB；
- 如果该进程为运行状态，则保护其现场，将其状态转为阻塞状态，停止运行；
- 将该 PCB 插入的阻塞队列中去；

4. **唤醒进程**

进程由「运行」转变为「阻塞」状态是由于进程必须等待某一事件的完成，所以处于阻塞状态的进程是绝对不可能叫醒自己的。

如果某进程正在等待 I/O 事件，需由别的进程发消息给它，则只有当该进程所期待的事件出现时，才由发现者进程用唤醒语句叫醒它。

唤醒进程的过程如下：

- 在该事件的阻塞队列中找到相应进程的 PCB；
- 将其从阻塞队列中移出，并置其状态为就绪状态；
- 把该 PCB 插入到就绪队列中，等待调度程序调度；

进程的阻塞和唤醒是一对功能相反的语句，如果某个进程调用了阻塞语句，则必有一个与之对应的唤醒语句。

5. 进程上下文切换

1. 什么是进程上下文切换

大多数操作系统都是多任务，通常支持大于 CPU 数量的任务同时运行。实际上，这些任务并不是同时运行的，只是因为系统在很短的时间内，让各个任务分别在 CPU 运行，于是就造成同时运行的错觉。

任务是交给 CPU 运行的，那么在每个任务运行前，CPU 需要知道任务从哪里加载，又从哪里开始运行。

所以，操作系统需要事先帮 CPU 设置好 **CPU 寄存器和程序计数器**。

CPU 寄存器是 CPU 内部一个容量小，但是速度极快的内存（缓存）。

再来，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。

所以说，**CPU 寄存器**和**程序计数**是 CPU 在运行任何任务前，所必须依赖的环境，这些环境就叫做 **CPU 上下文**。

既然知道了什么是 CPU 上下文，那理解 CPU 上下文切换就不难了。

CPU 上下文切换就是先把前一个任务的 CPU 上下文（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。

系统内核会存储保持下来的上下文信息，当此任务再次被分配给 CPU 运行时，CPU 会重新加载这些上下文，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。

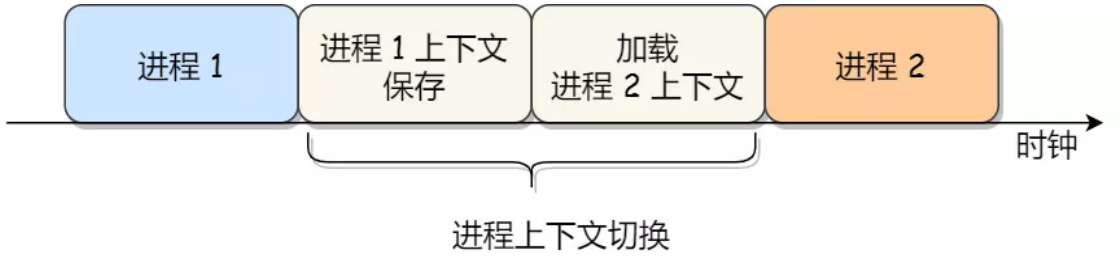
上面说到所谓的「任务」，主要包含进程、线程和中断。所以，可以根据任务的不同，把 CPU 上下文切换分成：**进程上下文切换、线程上下文切换和中断上下文切换**。

2. 进程上下文切换具体切换了什么

进程是由内核管理和调度的，所以进程的切换只能发生在内核态。

所以，**进程的上下文切换不仅包含了虚拟内存、栈、全局变量等用户空间的资源，还包括了内核堆栈、寄存器等内核空间的资源。**

通常，**会把交换的信息保存在进程的 PCB**，当要运行另外一个进程的时候，我们需要从这个进程的 PCB 取出上下文，然后恢复到 CPU 中，这使得这个进程可以继续执行，如下图所示：



3. 进程上下文切换的场景

   - (**CPU时间片完**)为了保证所有进程可以得到公平调度，CPU 时间被划分为一段段的时间片，这些时间片再被轮流分配给各个进程。这样，当某个进程的时间片耗尽了，就会被系统挂起，切换到其它正在等待 CPU 的进程运行；
   - (**资源不足**)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；
   - (**sleep**)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；
   - (**被抢占**)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；

- (**硬件中断**)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；

6. 线程的实现

- **用户线程（User Thread）**：在用户空间实现的线程，不是由内核管理的线程，是由用户态的线程库来完成线程的管理；
- **内核线程（Kernel Thread）**：在内核中实现的线程，是由内核管理的线程；
- **轻量级进程（LightWeight Process）**：在内核中来支持用户线程；

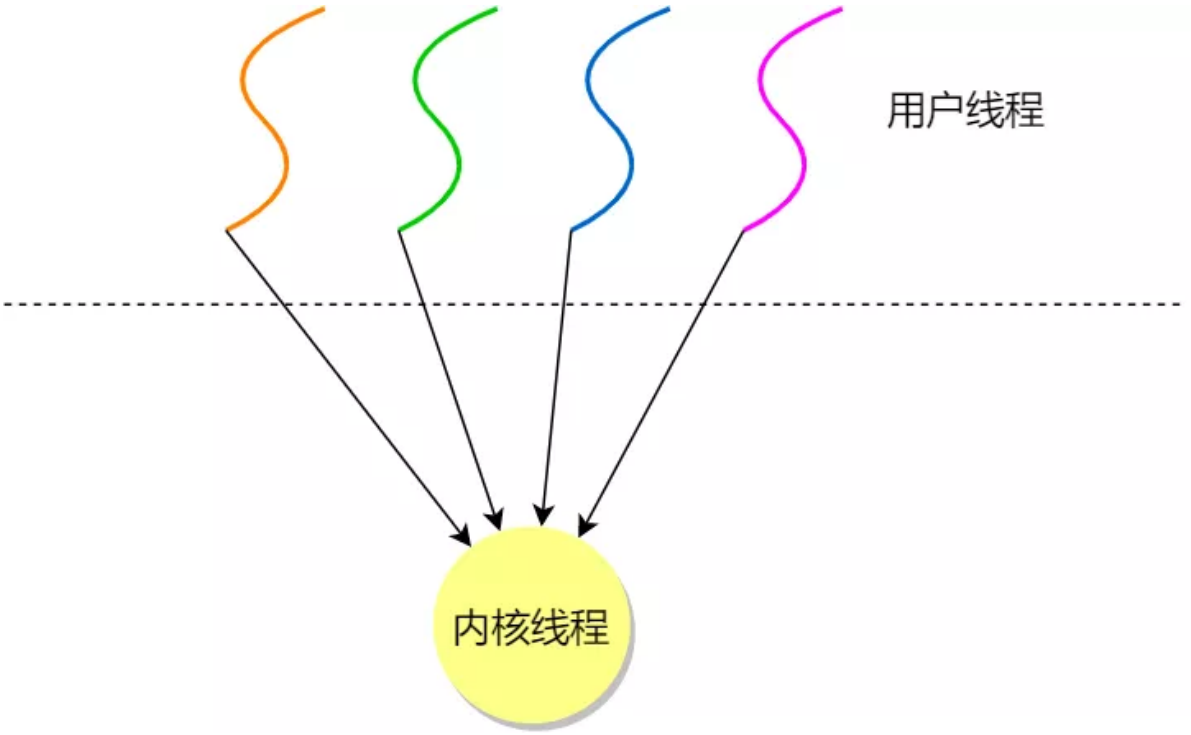
7. 用户线程(协程)

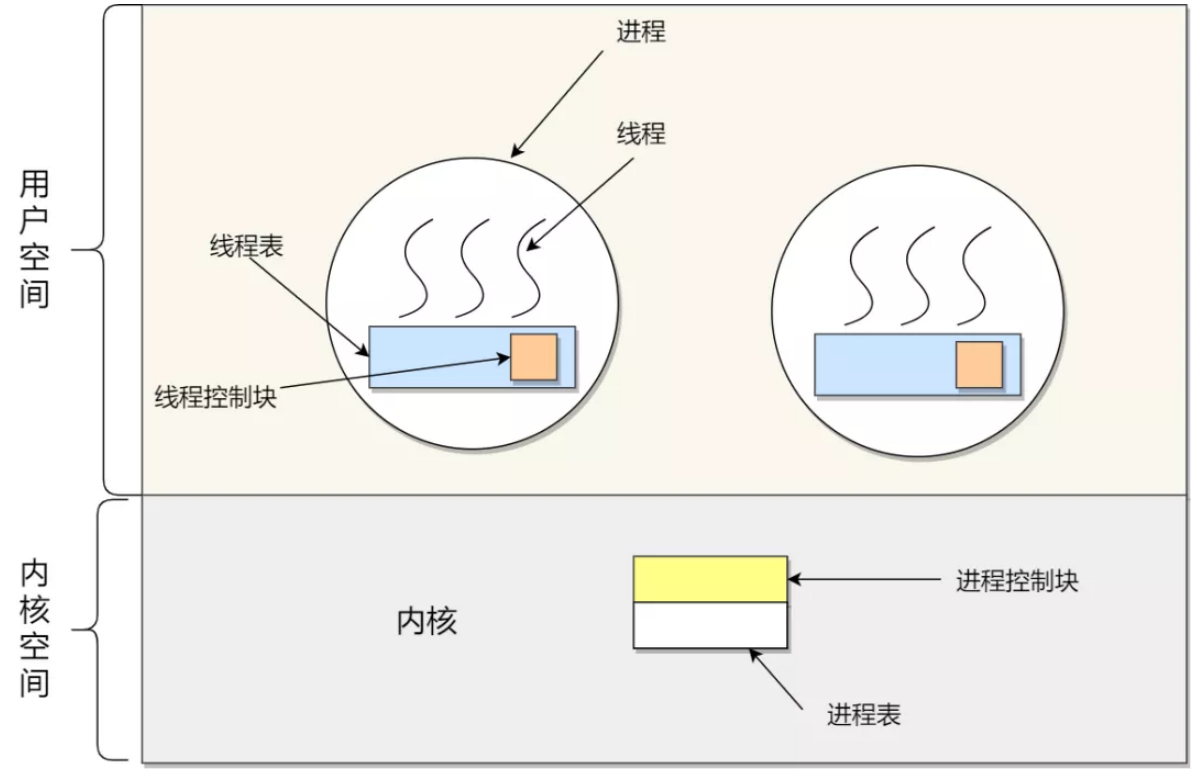
用户线程是基于用户态的线程管理库来实现的，那么**线程控制块（Thread Control Block, TCB）** 也是在库里面来实现的，对于操作系统而言是看不到这个 TCB 的，它只能看到整个进程的 PCB。

所以，**用户线程的整个线程管理和调度，操作系统是不直接参与的，而是由用户级线程库函数来完成线程的管理，包括线程的创建、终止、同步和调度等。**

用户级线程的一个缺点，**这些线程只能占用一个核**，所以做不到**并行加速**，而且由于用户线程的**透明性**，**操作系统是不能主动切换线程的**，换句话讲，如果 A，B 是同一个进程的两个线程的话， A 正在运行的时候，线程 B 想要运行的话，只能等待 A **主动放弃** CPU，也就是主动调用 `pthread_yield` 函数

用户级线程的模型，也就类似前面提到的**多对一(多个用户线程只对应一个内核线程)**的关系，即多个用户线程对应同一个内核线程，如下图所示：





用户线程的**优点**：

- 每个进程都需要有它私有的线程控制块（TCB）列表，用来跟踪记录它各个线程状态信息（PC、栈指针、寄存

器），TCB 由用户级线程库函数来维护，可用于不支持线程技术的操作系统；
- **用户线程的切换也是由线程库函数来完成的，无需用户态与内核态的切换**，所以速度特别快；

用户线程的**缺点**：

- 由于操作系统不参与线程的调度，如果一个线程发起了系统调用而阻塞，那进程所包含的用户线程都不能执行了。即对于操作系统还是只看到进程，而看不到进程里面的用户线程。即**用户线程对操作系统具有透明性**
- 当一个线程开始运行后，除非它主动地交出 CPU 的使用权，否则它所在的进程当中的其他线程无法运行，因为**用户态的线程没法打断当前运行中的线程**，它没有这个特权，只有操作系统才有，但是用户线程不是由操作系统管理的(**操作系统只分配一个核**)。
- 由于时间片分配给进程，故与其他进程比，在多线程执行时，每个线程得到的时间片较少，执行会比较慢；

8. 内核线程

内核线程是由操作系统管理的，线程对应的 TCB 自然是放在操作系统里的，这样线程的创建、终止和管理都是由操作系统负责。

内核线程的模型，也就类似前面提到的**一对一**的关系，即一个用户线程对应一个内核线程，如下图所示

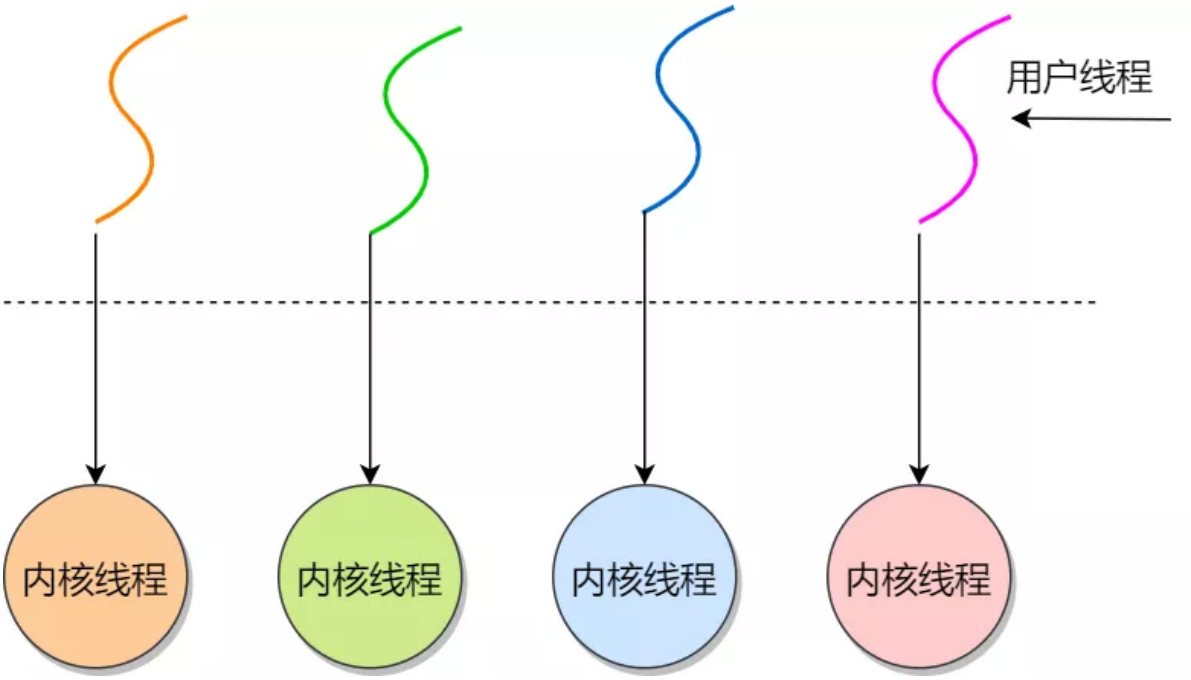
内核线程的**优点**：

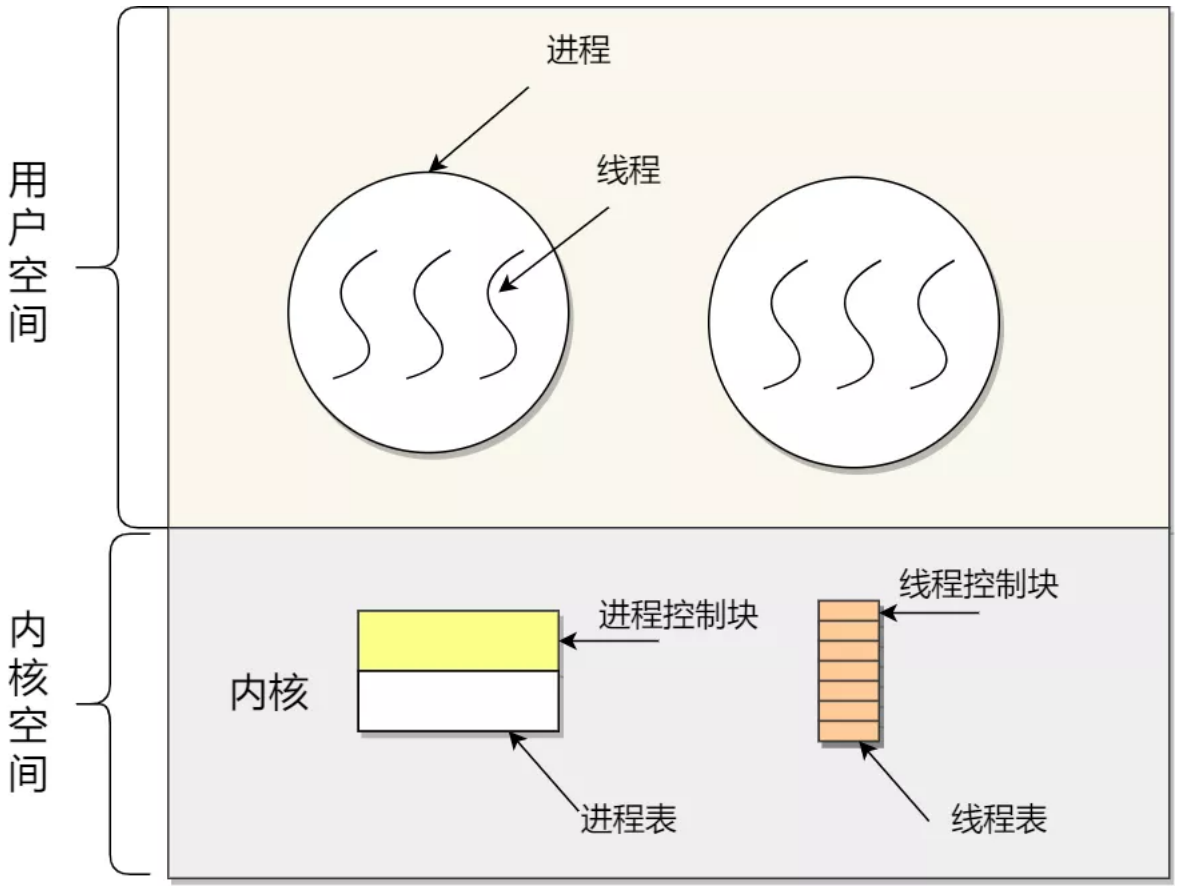
- 在一个进程当中，如果某个内核线程发起系统调用而被阻塞，并不会影响其他内核线程的运行；

- 分配给线程，多线程的进程获得更多的 CPU 运行时间；

内核线程的**缺点**：

- 在支持内核线程的操作系统中，由内核来维护进程和线程的上下问信息，如 PCB 和 TCB；
- 线程的创建、终止和切换都是通过系统调用的方式来进行，因此对于系统来说，系统开销比较大；





内核线程的**优点**：

- 在一个进程当中，如果某个内核线程发起系统调用而被阻塞，并不会影响其他内核线程的运行；
- 分配给线程，多线程的进程获得+更多的 CPU 运行时间；

内核线程的**缺点**：

- 在支持内核线程的操作系统中，由内核来维护进程和线程的上下问信息，如 PCB 和 TCB；
- 线程的创建、终止和切换都是通过系统调用的方式来进行，因此对于系统来说，系统开销比较大；

9. 轻量级进程

轻量级进程（Light-weight process，LWP）是**内核支持的用户线程**，一个进程可有一个或多个 LWP，**每个 LWP 是跟内核线程一对一映射的**，也就是 LWP 都是由一个内核线程支持。

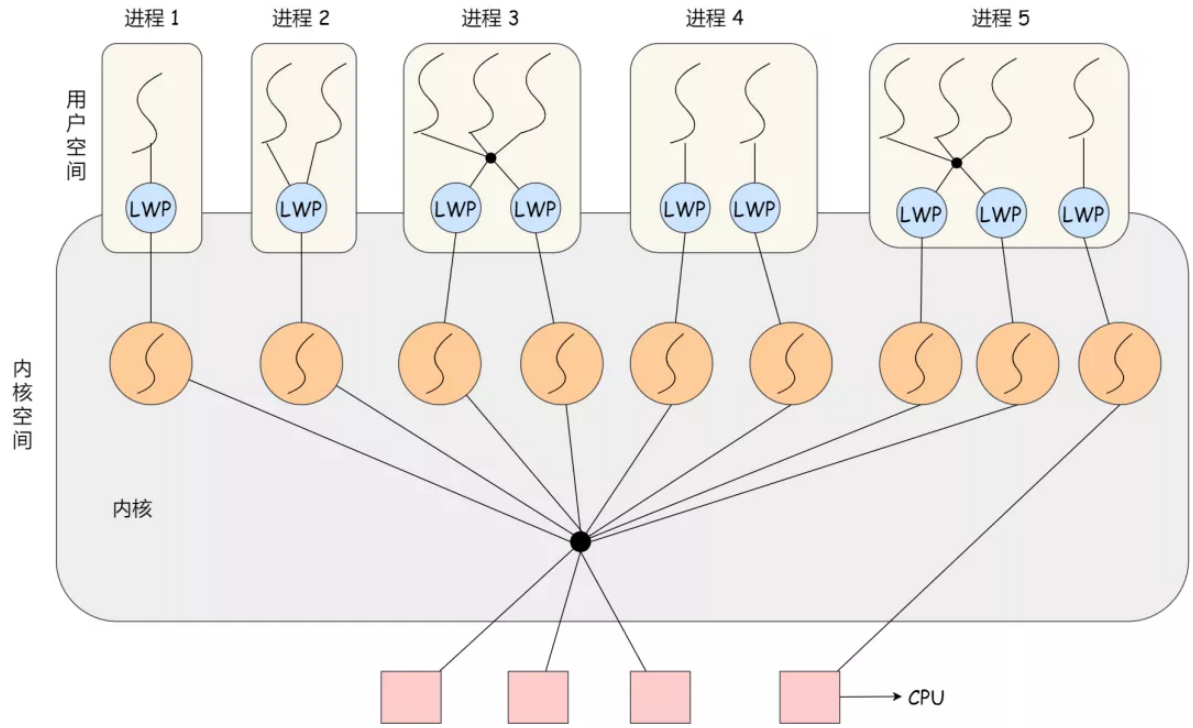
另外，LWP 只能由内核管理并像普通进程一样被调度，Linux 内核是支持 LWP 的典型例子。

在大多数系统中，**LWP与普通进程的区别也在于它只有一个最小的执行上下文和调度程序所需的统计信息。**一般来说，一个进程代表程序的一个实例，而 LWP 代表程序的执行线程，因为一个执行线程不像进程那样需要那么多状态信息，所以 LWP 也不带有这样的信息。

在 LWP 之上也是可以使用用户线程的，那么 LWP 与用户线程的对应关系就有三种：

- `1 : 1`，即一个 LWP 对应 一个用户线程；
- `N : 1`，即一个 LWP 对应多个用户线程；
- `N : N`，即多个 LMP 对应多个用户线程；

接下来针对上面这三种对应关系说明它们优缺点。先下图的 LWP 模型：



LWP 模型

1 : 1 模式

一个线程对应到一个 LWP 再对应到一个内核线程，如上图的进程 4，属于此模型。

- 优点：实现并行，当一个 LWP 阻塞，不会影响其他 LWP；
- 缺点：每一个用户线程，就产生一个内核线程，创建线程的开销较大。

N : 1 模式

多个用户线程对应一个 LWP 再对应一个内核线程，如上图的进程 2，线程管理是在用户空间完成的，此模式中用户的线程对操作系统不可见。

- 优点：用户线程要开几个都没问题，且上下文切换发生用户空间，切换的效率较高；
- 缺点：一个用户线程如果阻塞了，则整个进程都将会阻塞，另外在多核 CPU 中，是没办法充分利用 CPU 的。

M : N 模式

根据前面的两个模型混搭一起，就形成 `M:N` 模型，该模型提供了两级控制，首先多个用户线程对应到多个 LWP，LWP 再一一对应到内核线程，如上图的进程 3。

- 优点：综合了前两种优点，大部分的线程上下文发生在用户空间，且多个线程又可以充分利用多核 CPU 的资源。

组合模式

如上图的进程 5，此进程结合 `1:1` 模型和 `M:N` 模型。开发人员可以针对不同的应用特点调节内核线程的数目来达到物理并行性和逻辑并行性的最佳方案。

10. pthread_create创建的是用户线程还是内核线程

结论:与linux内核版本有关，1：使用2.6的内核的系统平台，2：你的gcc支持NPTL （现在一般都支持），那么你编译出来的多线程程序，就是“内核级”线程了。

原文链接:https://www.zhihu.com/question/35128513

11. fork

fork底层是调用了内核的函数来实现fork的功能的，即先create()先创建进程，此时进程内容为空，然后clone()复制父进程的内容到子进程中，此时子进程就诞生了，接着父进程就return返回了。**而子进程诞生后，是直接运行return返回**

的，然后接着执行后面的程序，这里注意：子进程是不会执行前面父进程已经执行过的程序了得，因为PCB中记录了当前进程运行到哪里，而子进程又是完全拷贝过来的，所以PCB的程序计数器也是和父进程相同的，所以是从fork()后面的程序继续执行。

写时复制

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。

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。

12. 进程调度算法

原文链接:https://zhuanlan.zhihu.com/p/225162322

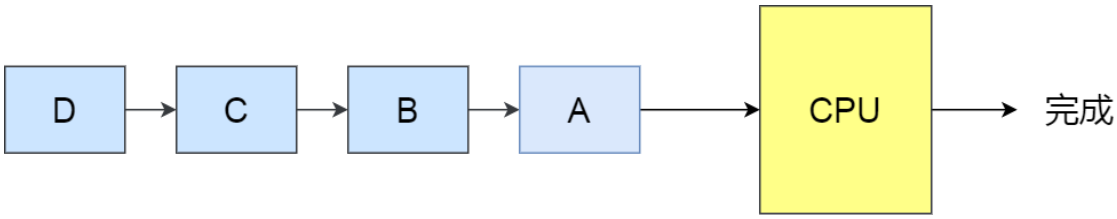
主要六种:

- 先来先服务FCFS
- 短作业优先SJF

- 高响应比优先
- 时间片轮转
- 最高优先级调度
- 多级反馈队列调度

1. **先来先服务FCFS**

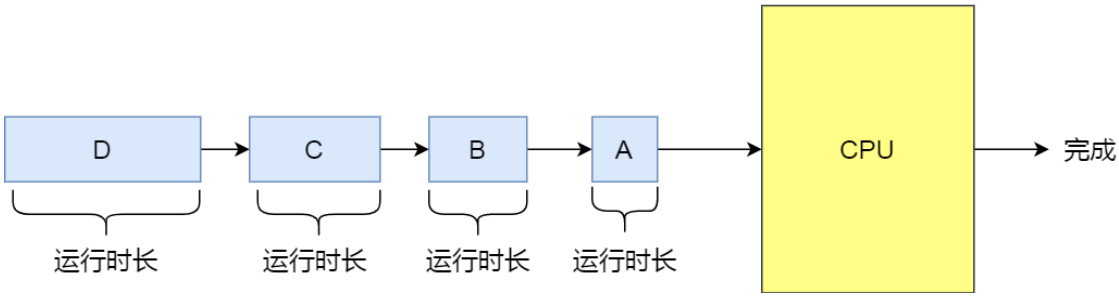
   非抢占式的**先来先服务（*First Come First Severd, FCFS*）算法**



   顾名思义，先来后到，**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。**

2. **最短作业优先调度算法**

   最短作业优先（*Shortest Job First, SJF*）调度算法同样也是顾名思义，它会**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。

3. **高相应比优先调度算法**

   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。

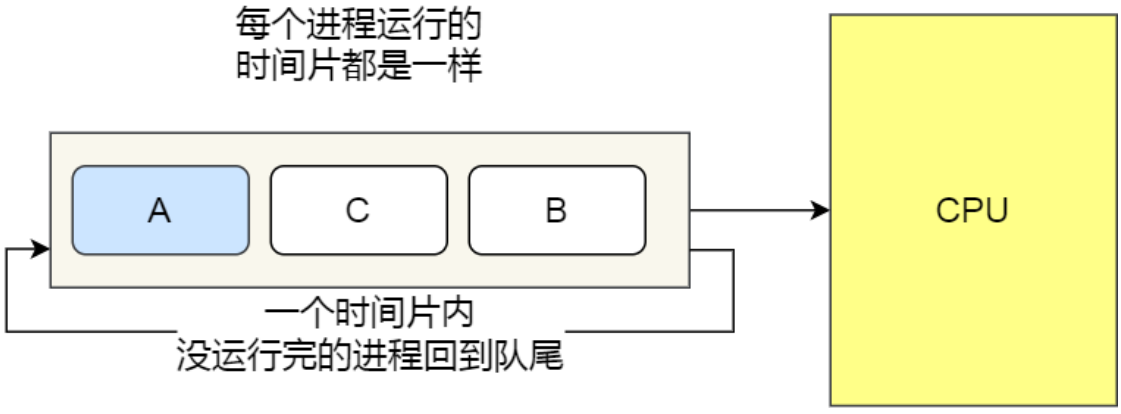
   那么，**高响应比优先 （*Highest Response Ratio Next, HRRN*）调度算法**主要是权衡了短作业和长作业。

   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：

$$优先权 = \frac{等待时间 + 要求服务时间}{要求服务时间}$$

4. **时间片轮转调度算法**

   最古老、最简单、最公平且使用最广的算法就是**时间片轮转（*Round Robin, RR*）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；

另外，时间片的长度就是一个很关键的点：

- 如果时间片设得太短会导致过多的进程上下文切换，降低了 CPU 效率；
- 如果设得太长又可能引起对短作业进程的响应时间变长。将通常时间片设为 `20ms~50ms` 通常是一个比较合理的折中值。

5. **最高优先级调度算法**

前面的「时间片轮转算法」做了个假设，即让所有的进程同等重要，也不偏袒谁，大家的运行时间都一样。

但是，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。**

进程的优先级可以分为，静态优先级或动态优先级：

- 静态优先级：创建进程时候，就已经确定了优先级了，然后整个运行时间优先级都不会变化；
- 动态优先级：根据进程的动态变化调整优先级，比如如果进程运行时间增加，则降低其优先级，如果进程等待时间（就绪队列的等待时间）增加，则升高其优先级，也就是**随着时间的推移增加等待进程的优先级。**

该算法也有两种处理优先级高的方法，非抢占式和抢占式：

- 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。

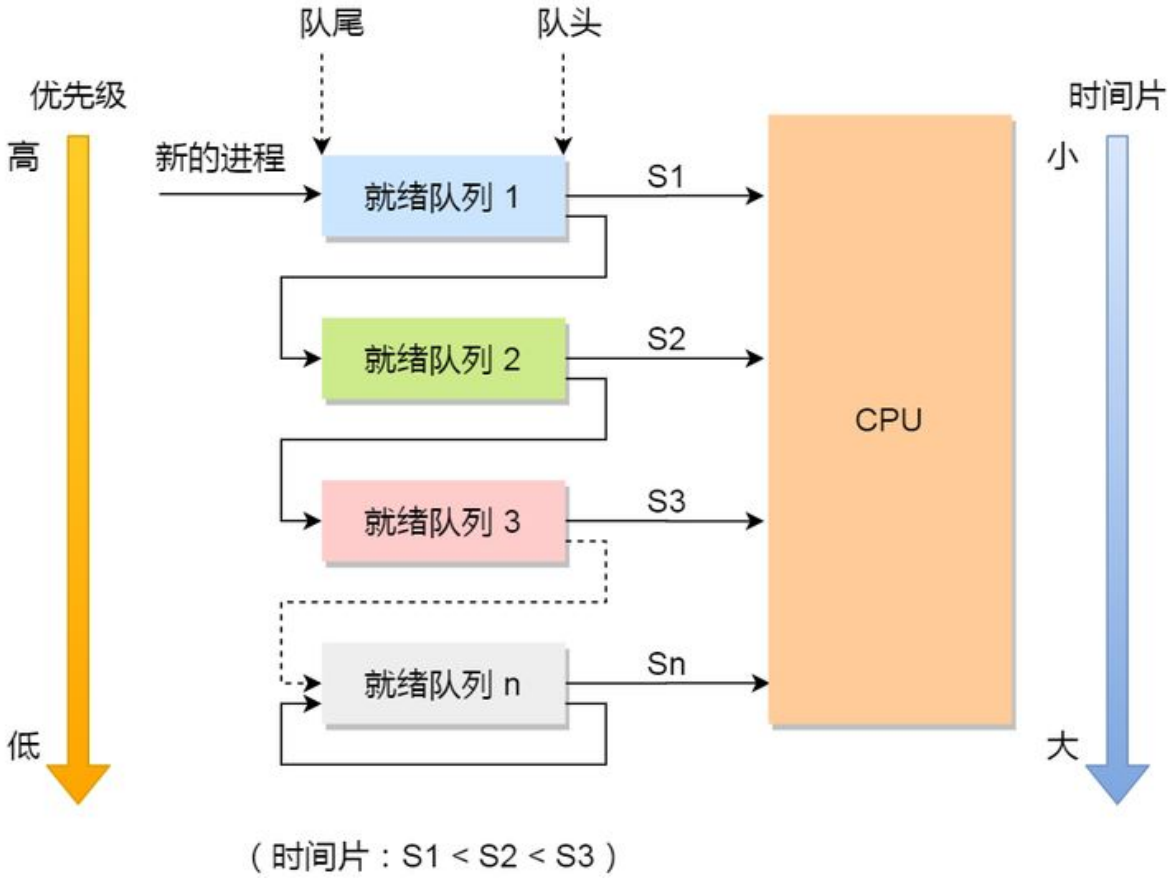
但是依然有缺点，可能会导致低优先级的进程永远不会运行。

6. **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。

   顾名思义：

   - 「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；



来看看，它是如何工作的：

- 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**；
- 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；
- 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；

  可以发现，对于短作业可能可以在第一级队列很快被处理完。对于长作业，如果在第一级队列处理不完，可以移入下次队列等待被执行，虽然等待的时间变长了，但是运行时间也会更长了，所以该算法很好的**兼顾了长短作业，同时有较好的响应时间。**

13. 虚拟内存

原文链接: https://blog.csdn.net/qq_34827674/article/details/107042163

如果程序直接操作内存的物理地址，假如第一个程序在 2000 的位置写入一个新的值，将会擦掉第二个程序存放在相同位置上的所有内容，所以同时运行两个程序是根本行不通的，这两个程序会立刻崩溃。

我们可以把进程所使用的地址「**隔离**」开来，即让操作系统为每个进程分配独立的一套「**虚拟地址**」，人人都有，大家自己玩自己的地址就行，互不干涉。但是有个前提每个进程都不能访问物理地址，至于虚拟地址最终怎么落到物理内存里，对进程来说是透明的，操作系统已经把这些都安排的明明白白了。

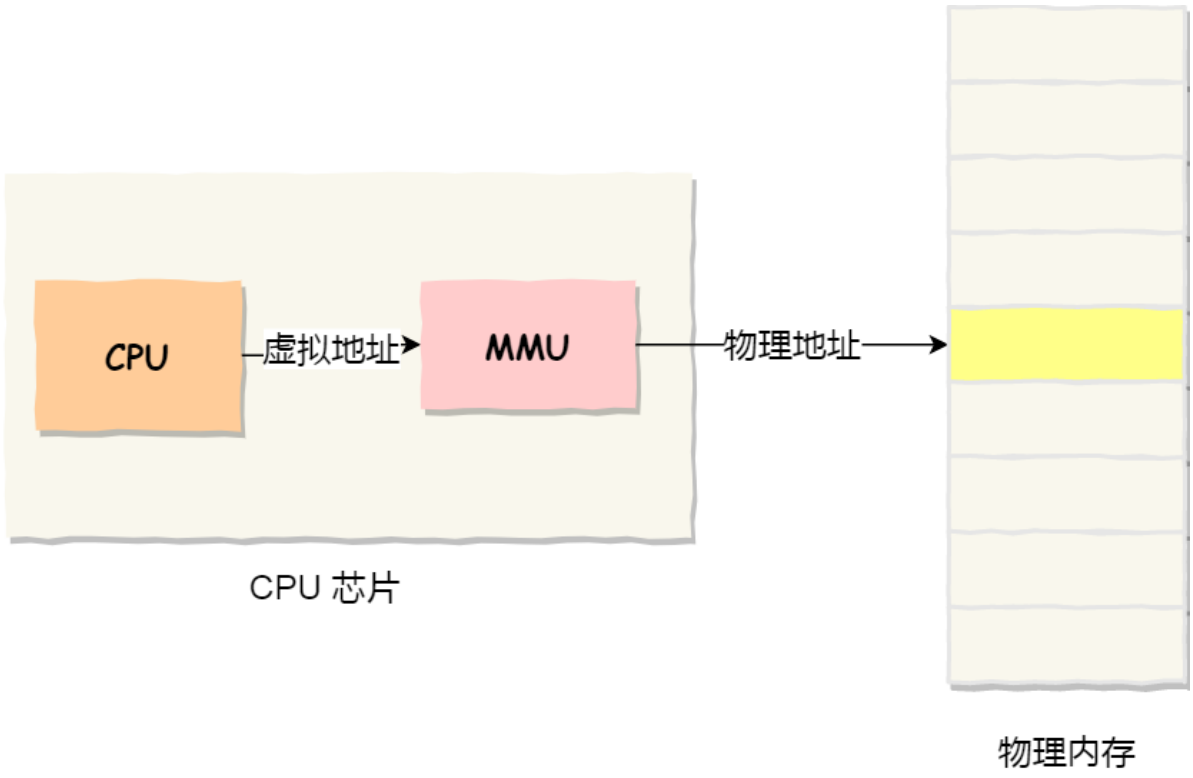
操作系统会提供一种机制，将不同进程的虚拟地址和不同内存的物理地址映射起来。

如果程序要访问虚拟地址的时候，由操作系统转换成不同的物理地址，这样不同的进程运行的时候，写入的是不同的物理地址，这样就**不会冲突**了。

于是，这里就引出了两种地址的概念：

- 我们程序所使用的内存地址叫做*虚拟内存地址**(Virtual Memory Address*)
- 实际存在硬件里面的空间地址叫*物理内存地址**(Physical Memory Address*)。

操作系统引入了虚拟内存，进程持有的虚拟地址会通过 CPU 芯片中的**内存管理单元（MMU）**的**映射关系**，来转换变成物理地址，然后再通过物理地址访问内存，如下图所示：



> 操作系统是如何管理虚拟地址与物理地址之间的关系？

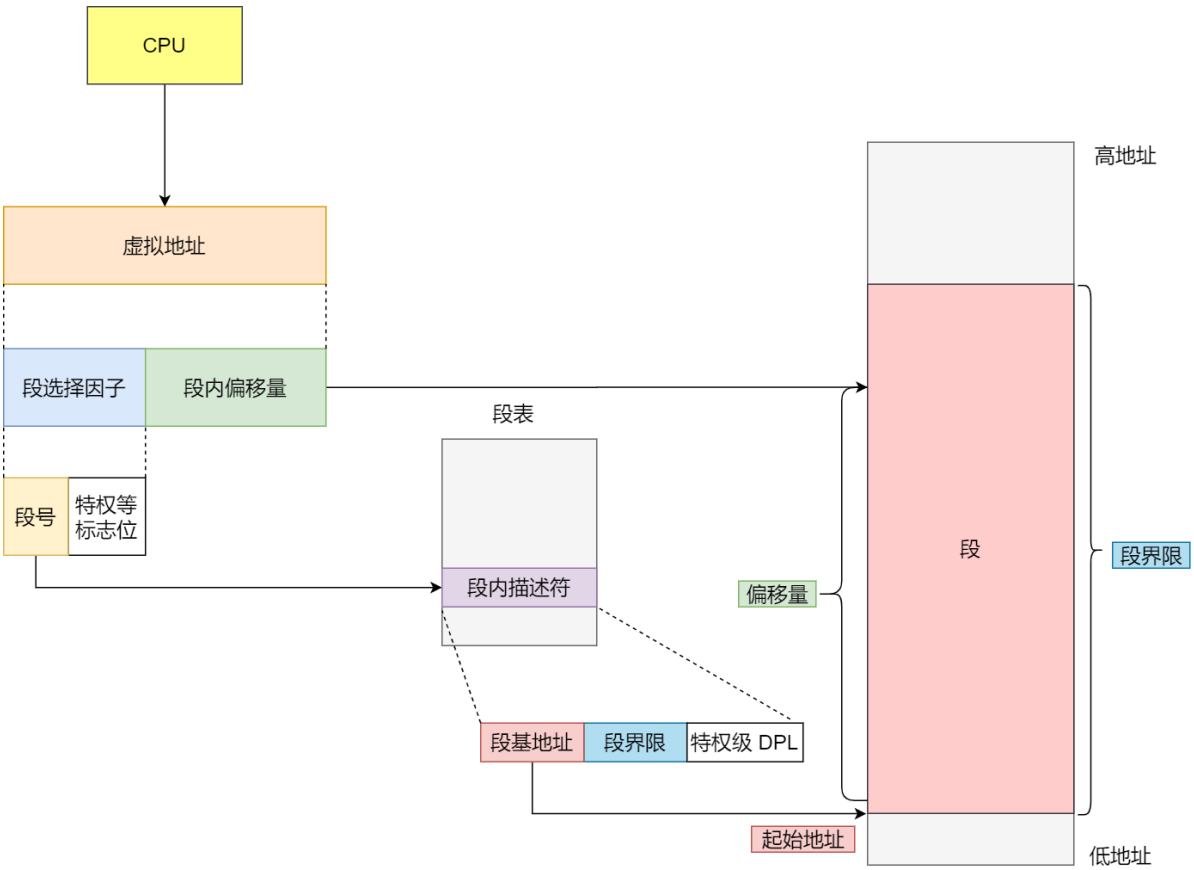
主要有两种方式，分别是**内存分段和内存分页**，分段是比较早提出的，我们先来看看内存分段。

14. 段式存储系统

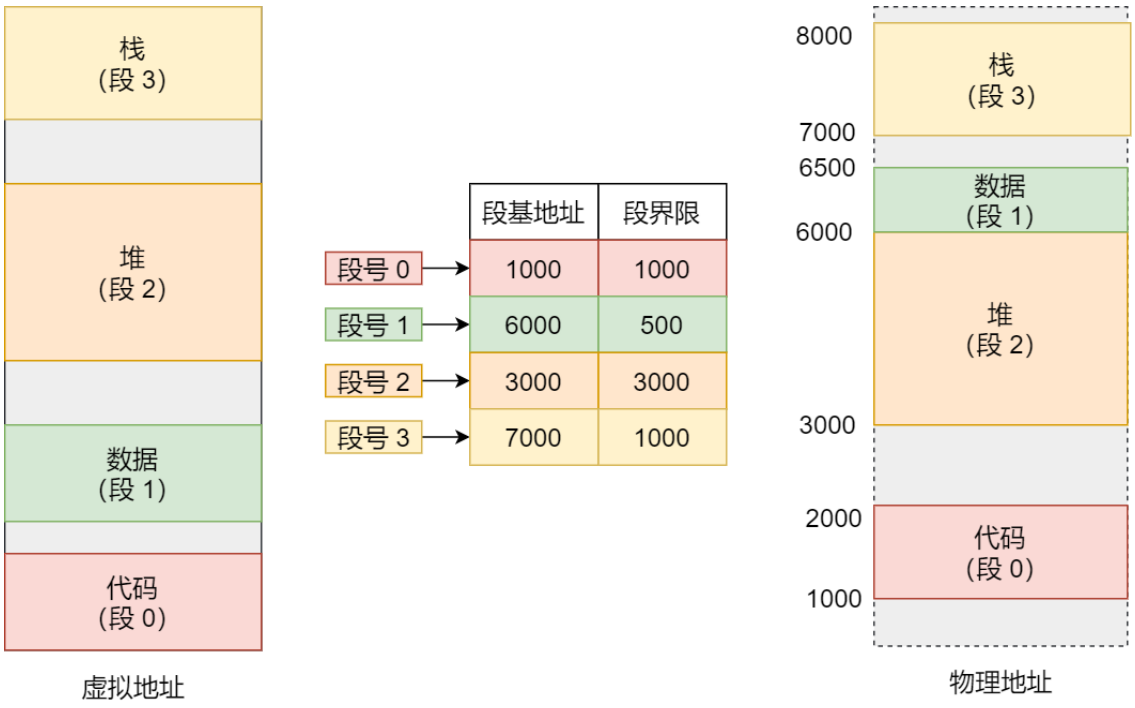
程序是由若干个逻辑分段组成的，如可由代码分段、数据分段、栈段、堆段组成。**不同的段是有不同的属性的，所以就用分段（*Segmentation*）的形式把这些段分离出来。**

即为了离散地管理了进程的逻辑空间

分段机制，虚拟地址和物理地址的映射



- **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。



出现的问题:

- 第一个就是**内存碎片**的问题。
- 第二个就是**内存交换的效率低**的问题。

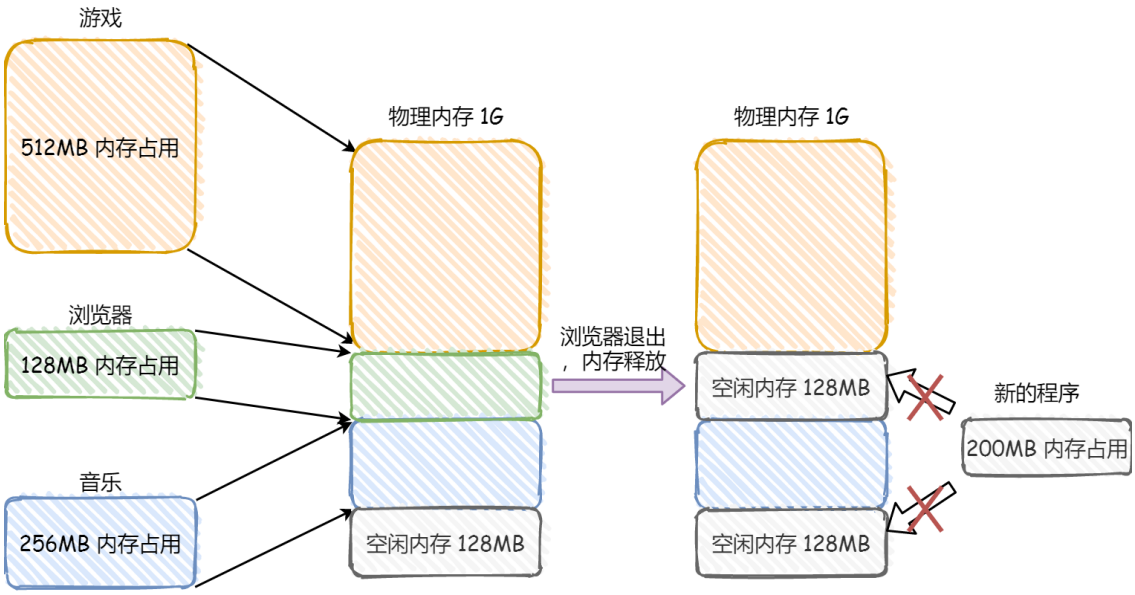
分段产生内存碎片的原因

我们来看看这样一个例子。假设有 1G 的物理内存，用户执行了多个程序，其中：

- 游戏占用了 512MB 内存
- 浏览器占用了 128MB 内存
- 音乐占用了 256 MB 内存。

这个时候，如果我们关闭了浏览器，则空闲内存还有 1024 - 512 - 256 = 256MB。

如果这个 256MB 不是连续的，被分成了两段 128 MB 内存，这就会导致没有空间再打开一个 200MB 的程序。



这里的内存碎片的问题共有两处地方：

- 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；

针对上面两种内存碎片的问题，解决的方式会有所不同。

解决**外部内存碎片**的问题就是**内存交换**。

可以把音乐程序占用的那 256MB 内存写到硬盘上，然后再从硬盘上读回来到内存里。不过再读回的时候，我们不能装载回原来的位置，而是紧紧跟着那已经被占用了的 512MB 内存后面。这样就能空缺出连续的 256MB 空间，于是新的 200MB 程序就可以装载进来。

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。**

> 再来看看，分段为什么会导致内存交换效率低的问题？

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，那不得不重新利用 `Swap` 内存区域，这个过程会产生性能瓶颈。

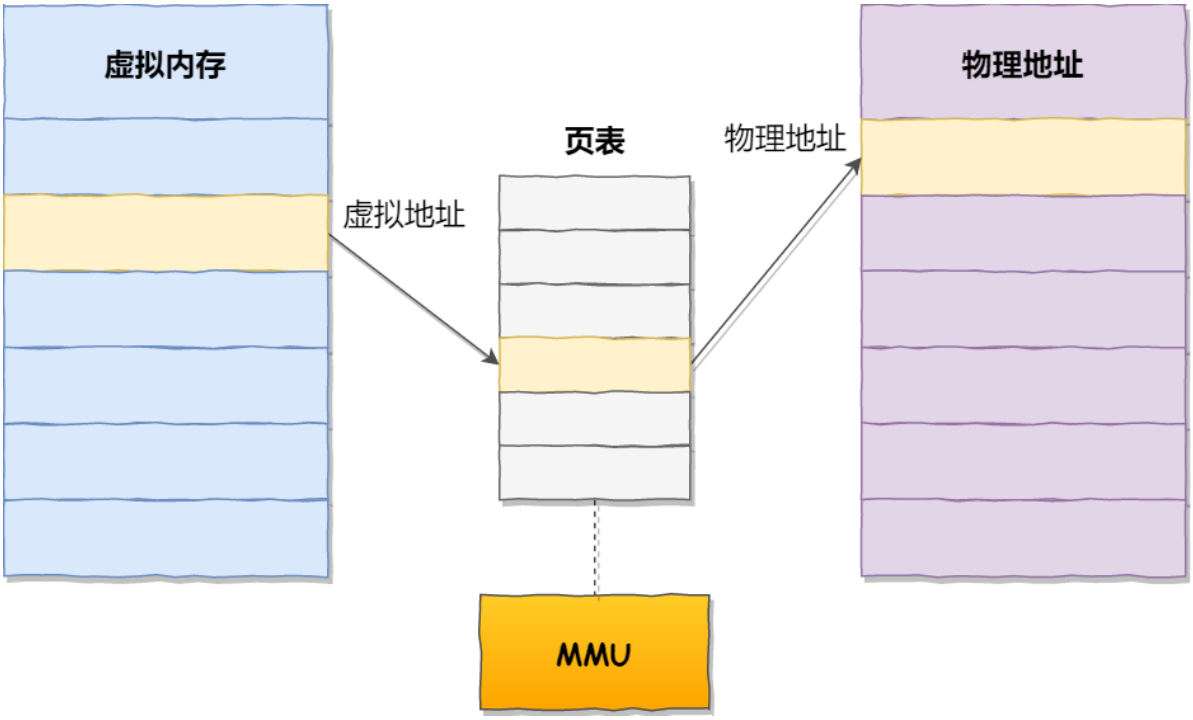
因为硬盘的访问速度要比内存慢太多了，每一次内存交换，我们都需要把一大段连续的内存数据写到硬盘上。

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。

15. 页式存储系统

*分页是把整个虚拟和物理内存空间切成一段段固定尺寸的大小。**这样一个连续并且尺寸固定的内存空间，我们叫**页**（Page*）。在 Linux 下，每一页的大小为 `4KB`。



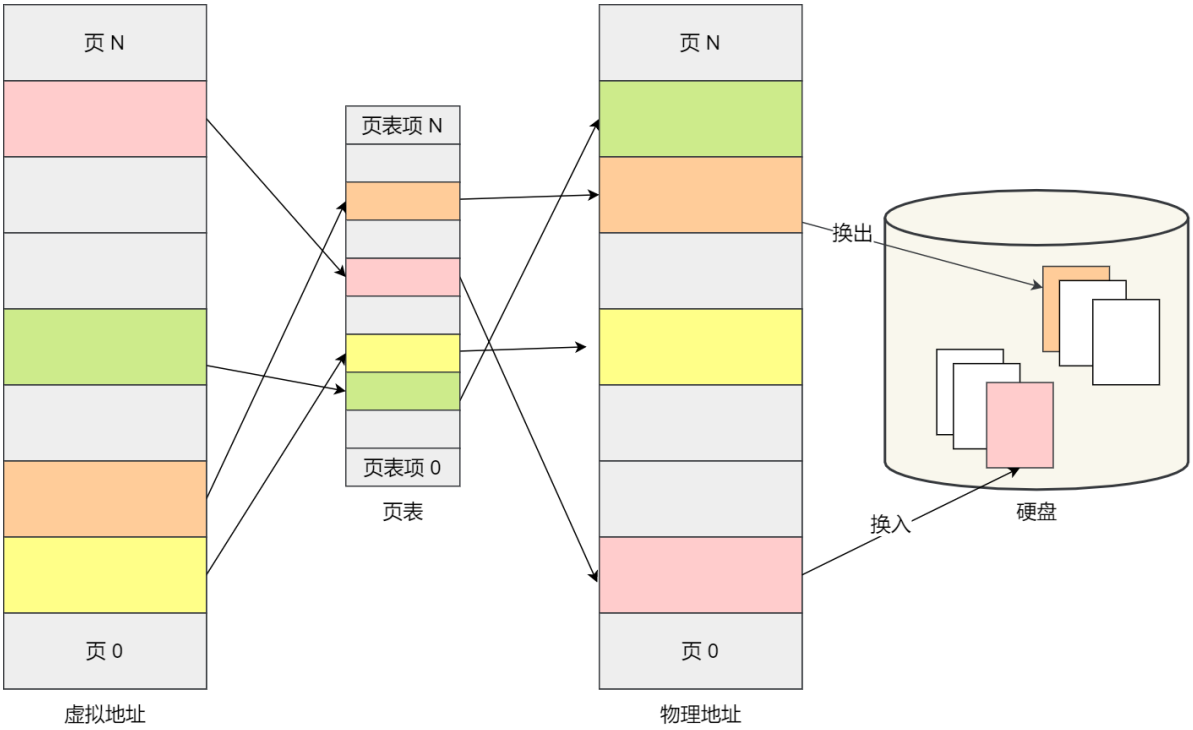
页表实际上存储在 CPU 的**内存管理单元（MMU）** 中，于是**CPU 就可以直接通过 MMU，找出要实际要访问的物理内存地址**。

> 分页是怎么解决分段的内存碎片、内存交换效率低的问题?

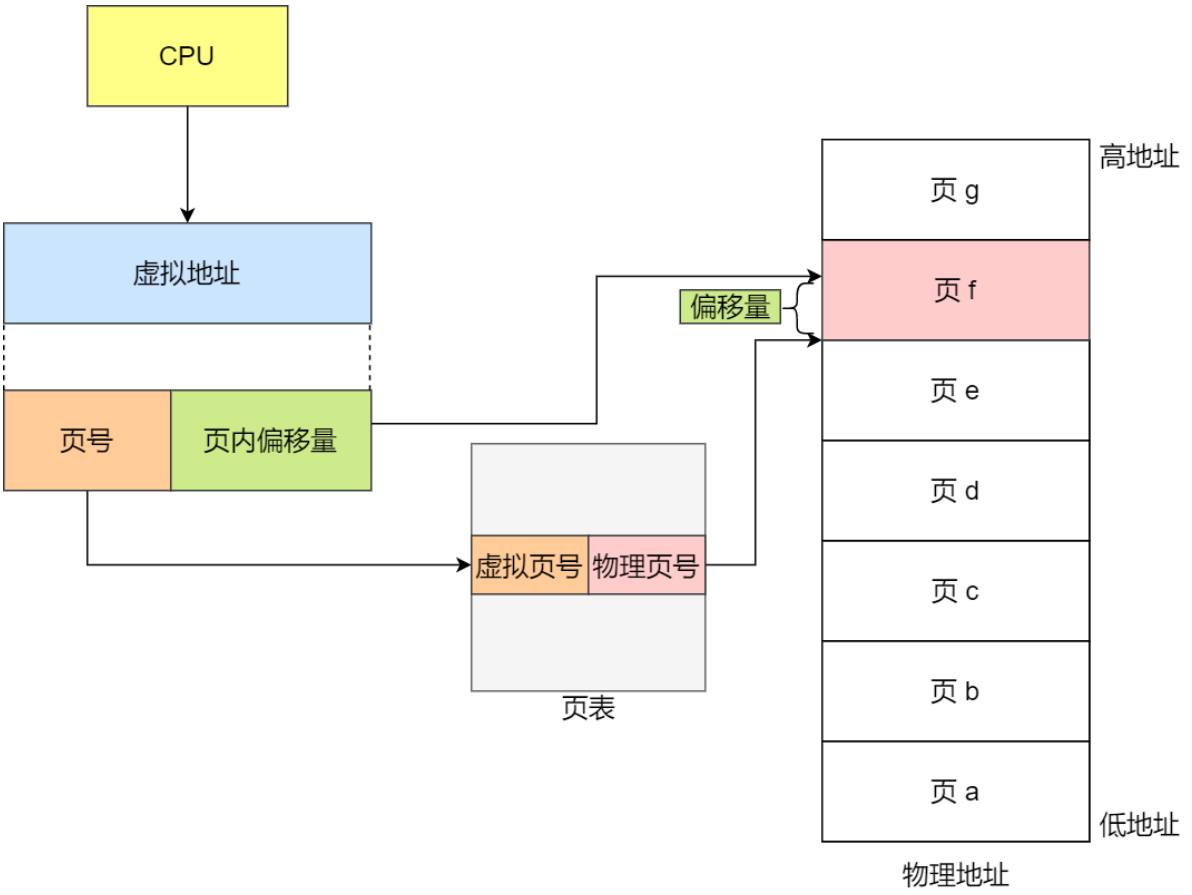
由于内存空间都是预先划分好的，也就不会像分段会产生间隙非常小的内存，这正是分段会产生内存碎片的原因。而**采用了分页，那么释放的内存都是以页为单位释放的，也就不会产生无法给进程使用的小内存。**

如果内存空间不够，操作系统会把其他正在运行的进程中的「最近没被使用」的内存页面给释放掉，也就是暂时写在硬盘上，称为*换出**（Swap Out）。一旦需要的时候，再加载进来，称为**换入**（Swap In*）。所以，一次性写入磁盘的

也只有少数的一个页或者几个页，不会花太多时间，**内存交换的效率就相对比较高。**

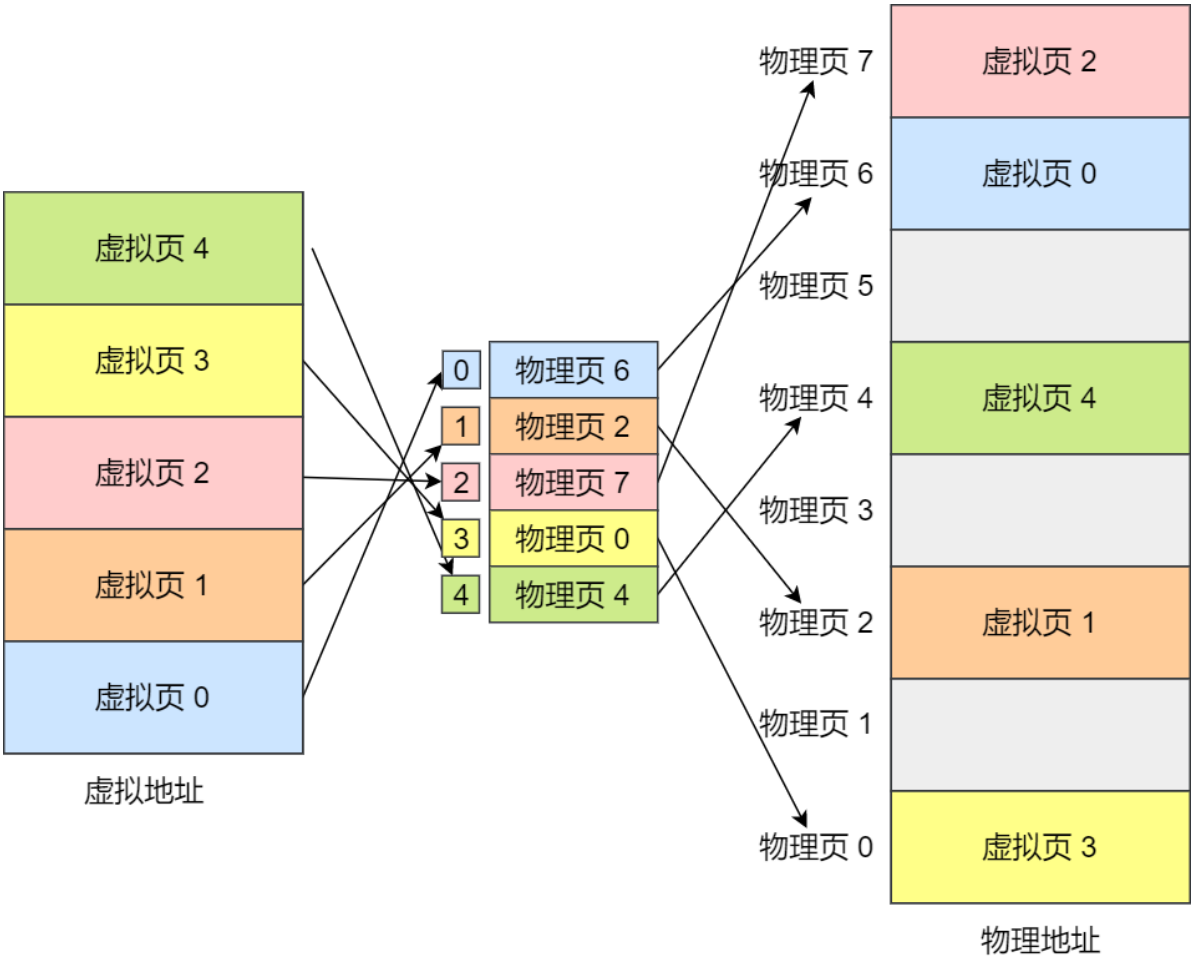


分页机制下，虚拟地址和物理地址的映射:



总结一下，对于一个内存地址转换，其实就是这样三个步骤：

- 把虚拟内存地址，切分成页号和偏移量；
- 根据页号，从页表里面，查询对应的物理页号；
- 直接拿物理页号，加上前面的偏移量，就得到了物理内存地址。



出现的问题

因为操作系统是可以同时运行非常多的进程的，那这不就意味着页表会非常的庞大。

为了理论上能够映射全部的内存地址，就需要在虚拟页表中建立起对所有物理内存地址的映射，这就意味着页表的大小会很大。

在 32 位的环境下，虚拟地址空间共有 4GB，假设一个页的大小是 4KB（2^12），那么就需要大约 100 万 （2^20） 个页，每个「页表项」需要 4 个字节大小来存储，那么整个 4GB 空间的映射就需要有 `4MB` 的内存来存储页表。

这 4MB 大小的页表，看起来也不是很大。但是要知道每个进程都是有自己的虚拟地址空间的，也就说都有自己的页表。

这样程序就可以映射所有物理内存地址了，但实际上由于程序局部性，程序并不需要用到（分配）每一个物理内存地址，大多数情况只需要（分配）一小部分内存物理地址，这时候如果还将全部的内存物理地址来放到页表中进行映射将会十分浪费。只要保证空间分配能够在物理内存中找到一块空间来对应这块被分配的虚拟空间即可，二级页表其实还是会映射了全部虚拟空间，只不过为了减少保存在内存中页表的大小，而再加一个页表(一级页表)映射原来的那个页表(二级页表)，以更小的体积保存在内存中。

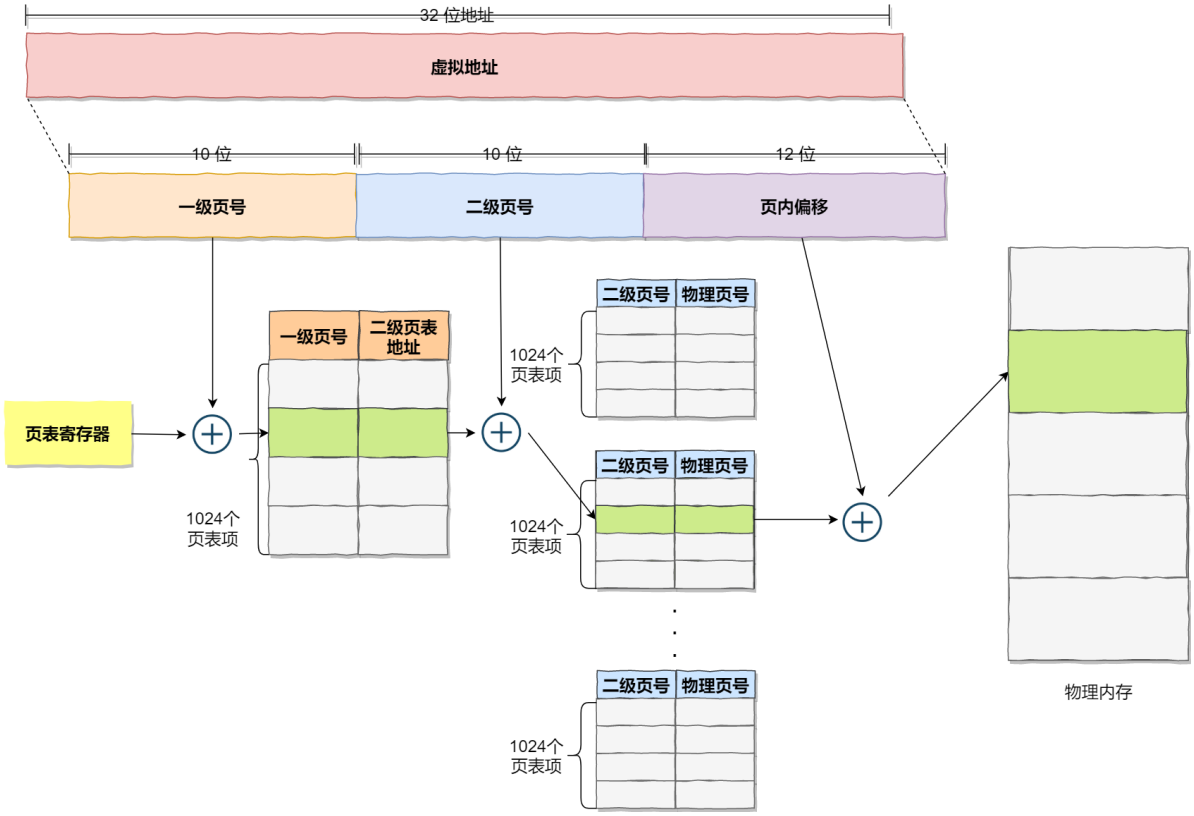
那么，`100` 个进程的话，就需要 `400MB` 的内存来存储页表，这是非常大的内存了，更别说 64 位的环境了。

多级页表

要解决上面的问题，就需要采用的是一种叫作*多级页表**（Multi-Level Page Table*）的解决方案。

在前面我们知道了，对于单页表的实现方式，在 32 位和页大小 `4KB` 的环境下，一个进程的页表需要装下 100 多万个「页表项」，并且每个页表项是占用 4 字节大小的，于是相当于每个页表需占用 4MB 大小的空间。

我们**把这个 100 多万个「页表项」的单级页表再分页**，将页表（一级页表）分为 `1024` 个页表（二级页表），每个表（二级页表）中包含 `1024` 个「页表项」，形成**二级分页**。如下图所示：



> 你可能会问，分了二级表，映射 4GB 地址空间就需要 4KB（一级页表）+ 4MB（二级页表）的内存，这样占用空间不是更大了吗？

对于32位的机器，**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**，因为会存在部分对应的页表项都是空的，根本没有分配（即已经给映射到页表中了，但是我们实际上根本就用不着），对于已分配的页表项，如果存在最近一定时间未访问的页表，在物理内存紧张的情况下，操作系统会将页面换出到硬盘，也就是说不会占用物理内存。

如果使用了二级分页，一级页表就可以覆盖整个 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。**做个简单的计算，假设只有 20% 的一级页表项被用到了，那么页表占用的内存空间就只有 4KB（一级页表） + 20% * 4MB（二级页表）= `0.804MB`，这对比单级页表的 `4MB` 是不是一个巨大的节约?

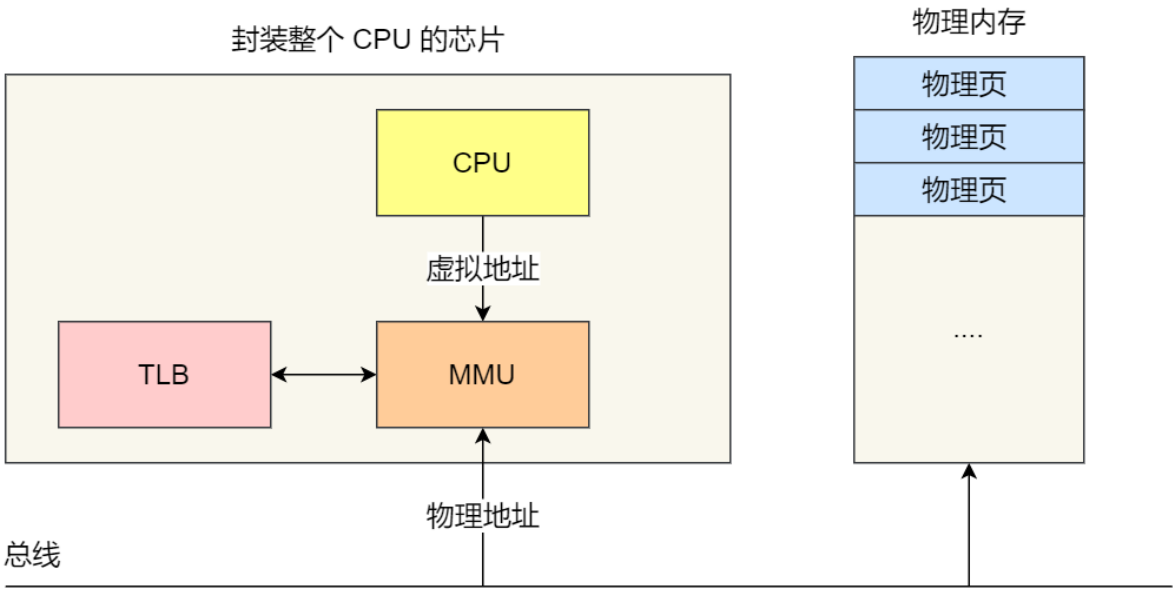
那么为什么不分级的页表就做不到这样节约内存呢？我们从页表的性质来看，保存在内存中的页表承担的职责是将虚拟地址翻译成物理地址。假如虚拟地址在页表中找不到对应的页表项，计算机系统就不能工作了。所以**页表一定要覆盖全部虚拟地址空间，不分级的页表就需要有 100 多万个页表项来映射，而二级分页则只需要 1024 个页表项**（此时一级页表覆盖到了全部虚拟地址空间，二级页表在需要时创建）。

> 这里重点是对于一个程序其需要覆盖所有的虚拟地址，如果单级页表只是部分载入就做不到，而二级页表就可以做到覆盖道全部的内存。只是取相关页的时候需要多一次转换，但是页表的空间大小也随之大大压缩。

引入多级页表结构是为了能离散式的存放页表，目的不在于减少或增加内存的多少。能够更好的利用局部性来减轻内存负担。

快表TLB

利用程序的局部性这一特性，把最常访问的几个页表项存储到访问速度更快的硬件，于是计算机科学家们，就在 CPU 芯片中，加入了一个专门存放程序最常访问的页表项的 Cache，这个 Cache 就是 TLB（*Translation Lookaside Buffer*），通常称为页表缓存、转址旁路缓存、快表等。



在 CPU 芯片里面，封装了内存管理单元（*Memory Management Unit*）芯片，它用来完成地址转换和 TLB 的访问与交互。

有了 TLB 后，那么 CPU 在寻址时，会先查 TLB，如果没找到，才会继续查常规的页表。

TLB 的命中率其实是很高的，因为程序最常访问的页就那么几个。

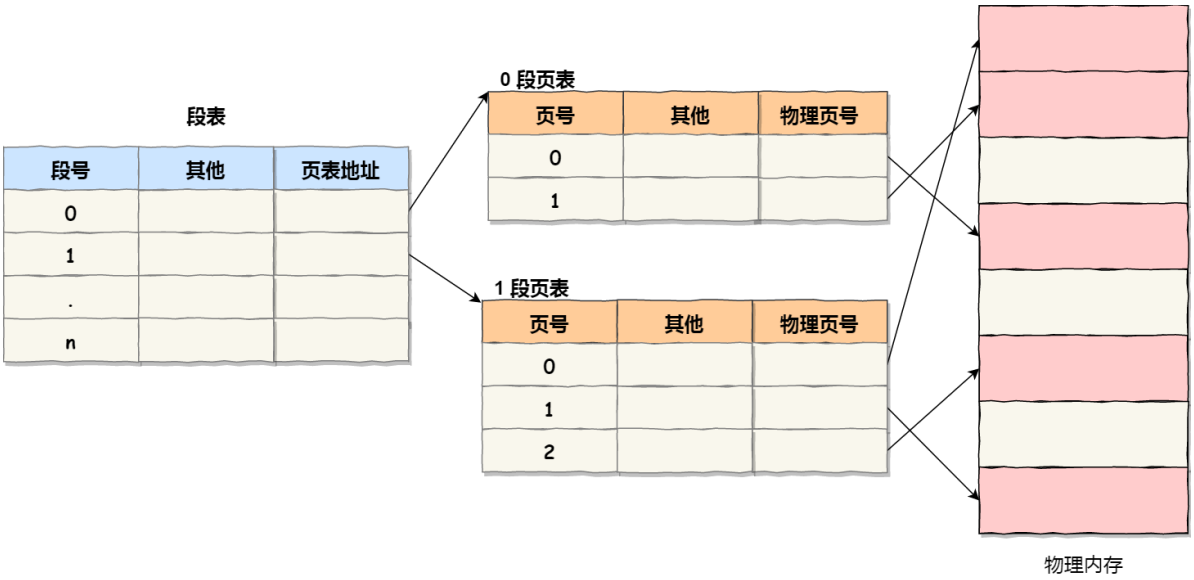
14. 段页式存储系统

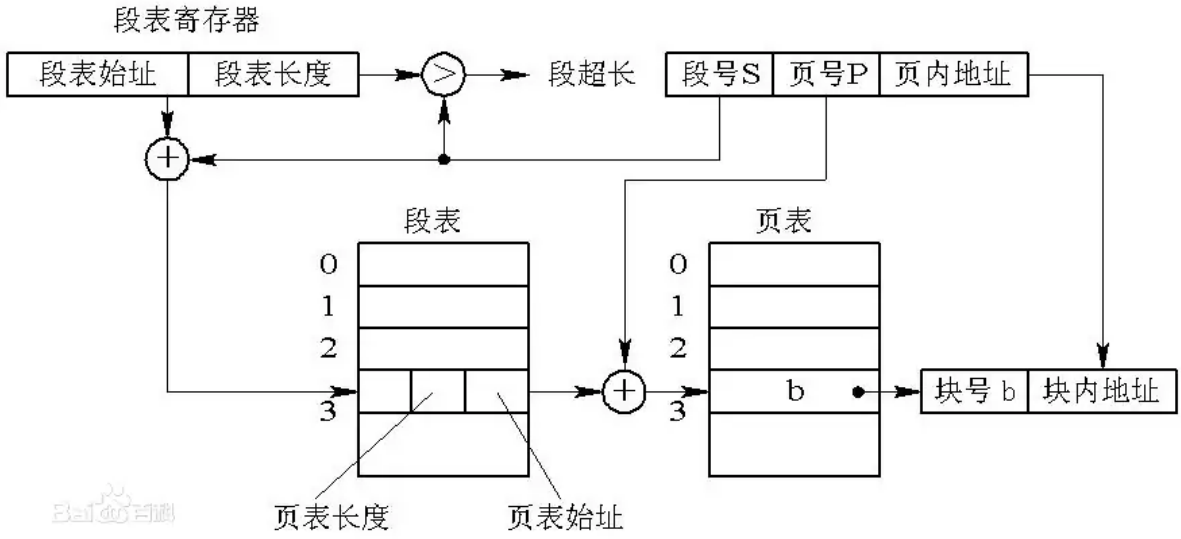
段页式内存管理实现的方式：

- 先将程序划分为多个有逻辑意义的段，也就是前面提到的分段机制；
- 接着再把每个段划分为多个页，也就是对分段划分出来的连续空间，再划分固定大小的页；

这样，地址结构就由**段号、段内页号和页内位移**三部分组成。

用于段页式地址变换的数据结构是每一个程序一张段表，每个段又建立一张页表，段表中的地址是页表的起始地址，而页表中的地址则为某页的物理页号，如图所示：





段页式地址变换中要得到物理地址须经过三次内存访问：

- 第一次访问段表，得到页表起始地址；
- 第二次访问页表，得到物理页号；
- 第三次将物理页号与页内位移组合，得到物理地址。

可用软、硬件相结合的方法实现段页式地址变换，这样虽然增加了硬件成本和系统开销，但提高了内存的利用率。

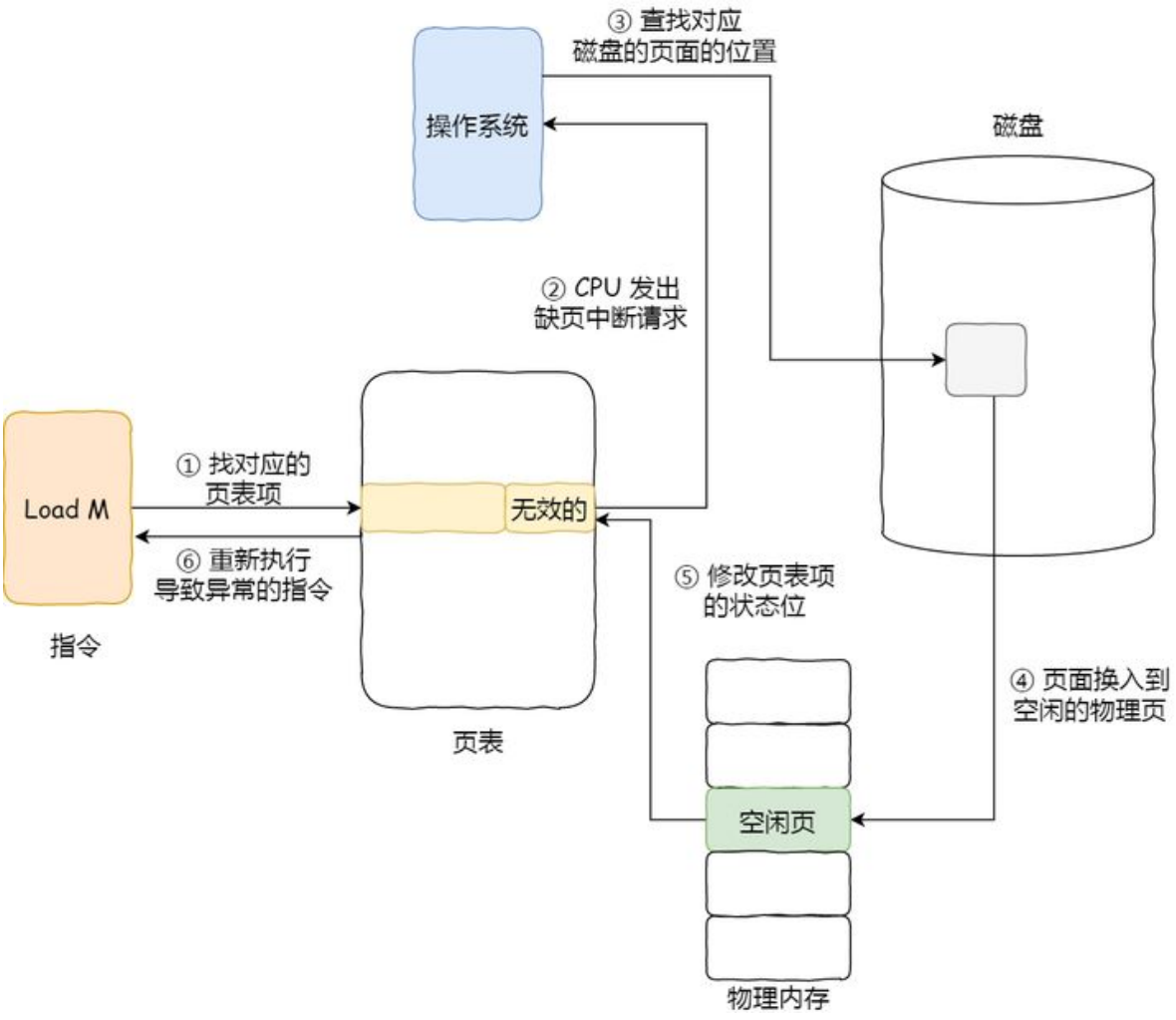
13. 内存页面置换算法

什么是缺页中断

当 CPU 访问的页面不在物理内存时，便会产生一个缺页中断，请求操作系统将所缺页调入到物理内存。那它与一般中断的主要区别在于：

- 缺页中断在指令执行「期间」产生和处理中断信号，而一般中断在一条指令执行「完成」后检查和处理中断信号。
- 缺页中断返回到该指令的开始重新执行「该指令」，而一般中断返回回到该指令的「下一个指令」执行。

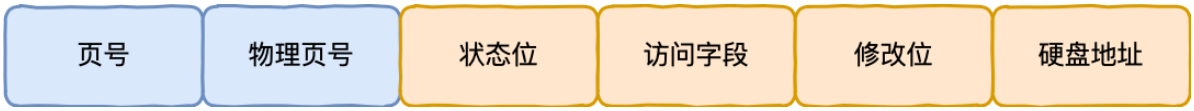
我们来看一下缺页中断的处理流程，如下图：



1. 在 CPU 里访问一条 Load M 指令，然后 CPU 会去找 M 所对应的页表项。
2. 如果该页表项的状态位是「有效的」，那 CPU 就可以直接去访问物理内存了，如果状态位是「无效的」，则 CPU 则会发送缺页中断请求。
3. 操作系统收到了缺页中断，则会执行缺页中断处理函数，先会查找该页面在磁盘中的页面的位置。
4. 找到磁盘中对应的页面后，需要把该页面换入到物理内存中，但是在换入前，需要在物理内存中找空闲页，如果找到空闲页，就把页面换入到物理内存中。
5. 页面从磁盘换入到物理内存完成后，则把页表项中的**状态位**修改为「有效的」。
6. 最后，CPU 重新执行导致缺页异常的指令。

在第四步中，如果这时候**内存页（注意是内存）**已经满了，则需要使用`页面置换算法`选择一个内存物理页进行淘汰。

这里提一下，页表项通常有如下图的字段：



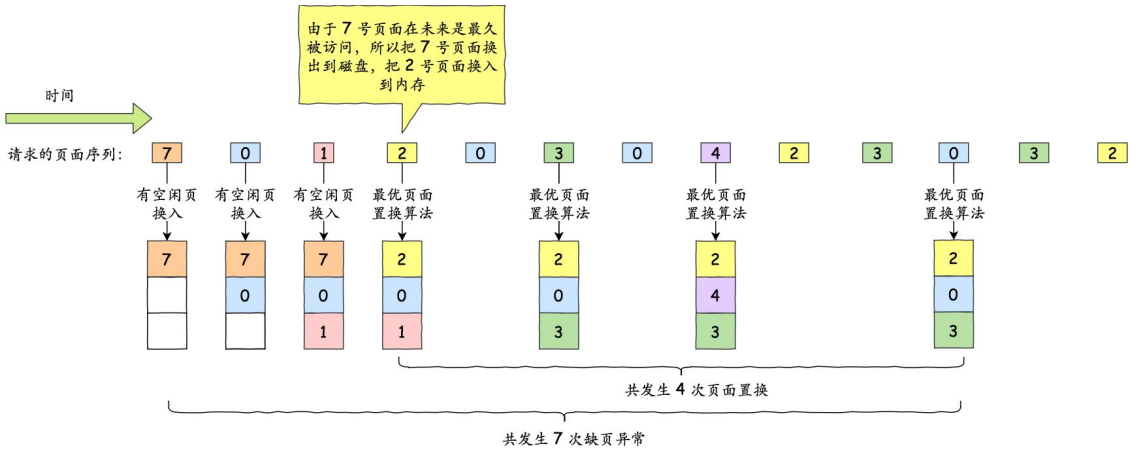
- `状态位`：用于表示该页**是否有效**，也就是说**是否在物理内存**中，供程序访问时参考。
- `访问字段`：用于记录该页在一段时间被访问的次数，供页面置换算法选择出页面时参考。
- `修改位`：表示该页在调入内存后是否有被修改过，由于内存中的每一页都在磁盘上保留一份副本，因此，如果没有修改，在置换该页时就不需要将该页写回到磁盘上，以减少系统的开销；如果已经被修改，则将该页重写到磁盘上，以保证磁盘中所保留的始终是最新的副本。

- `硬盘地址`：用于指出该页在硬盘上的地址，通常是物理块号，供调入该页时使用。

页面置换算法主要有:

1. 最佳页面置换算法(OPT)
2. 先进先出置换算法(FIFO)
3. **最近最久未使用算法(LRU)（重点，手撕代码）**
4. 时钟页面置换算法(Lock)
5. 最不常用置换算法(LFU)
6. **最佳页面置换算法(OPT)**

   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。**

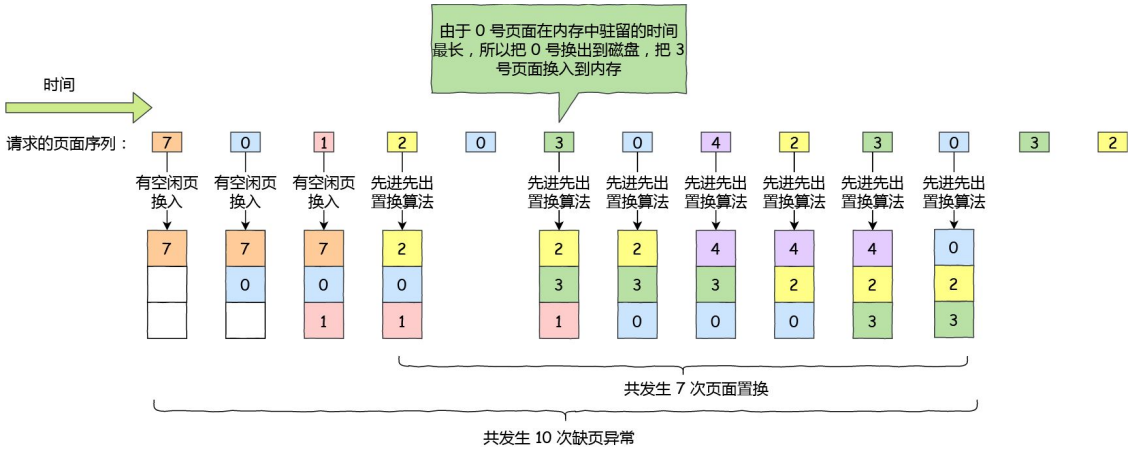


   这很理想，但是实际系统中无法实现，因为程序访问页面时是动态的，我们是无法预知每个页面在「下一次」访问前的等待时间。

   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。

7. **先进先出置换算法(FIFO)**

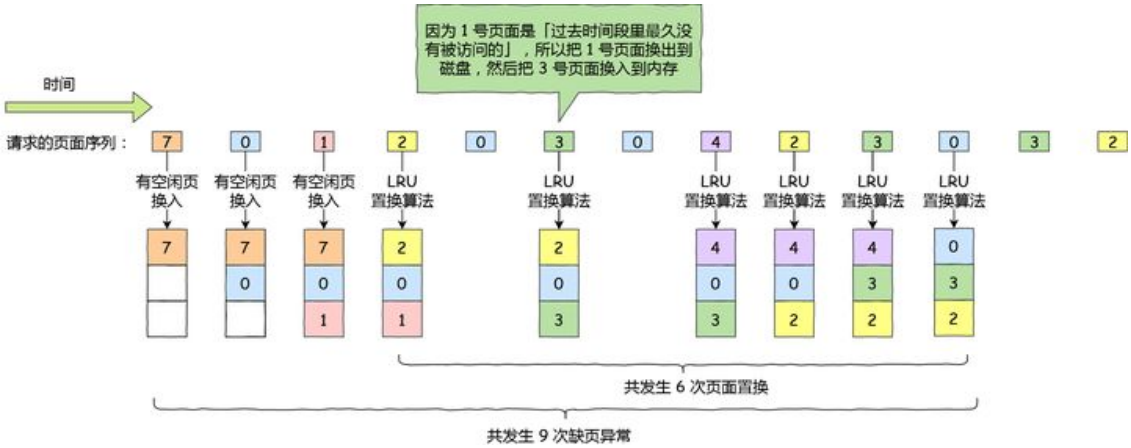
既然我们无法预知页面在下一次访问前所需的等待时间，那我们可以**选择在内存驻留时间很长的页面进行中置换**，这个就是「先进先出置换」算法的思想。



8. 最近最久未使用的置换算法（LRU）

最近最久未使用（*LRU*）的置换算法的基本思路是，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。

这种算法近似最优置换算法，最优置换算法是通过「未来」的使用情况来推测要淘汰的页面，而 LRU 则是通过「**历史**」的使用情况来推测要淘汰的页面。

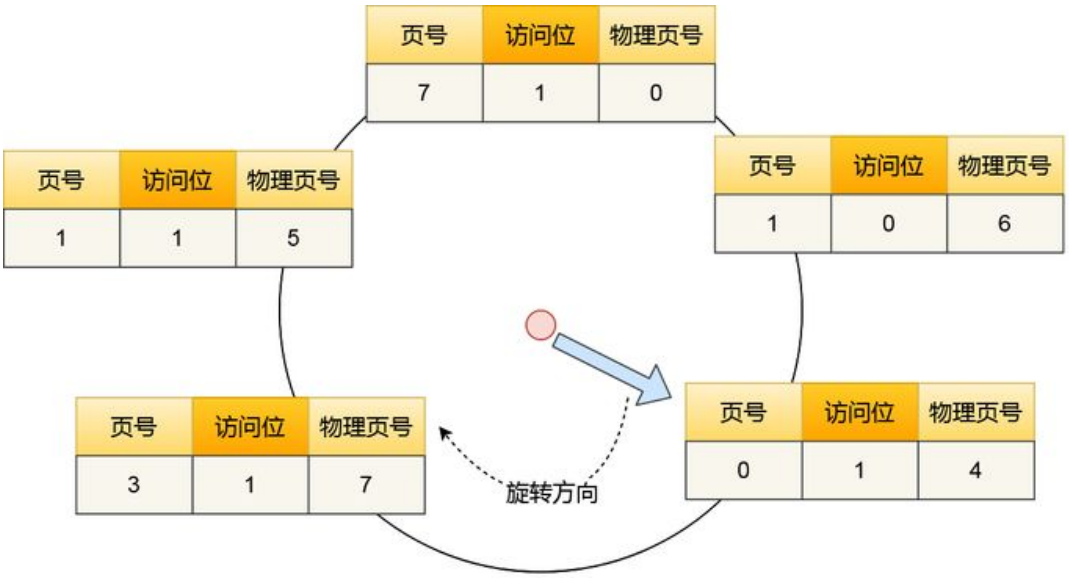


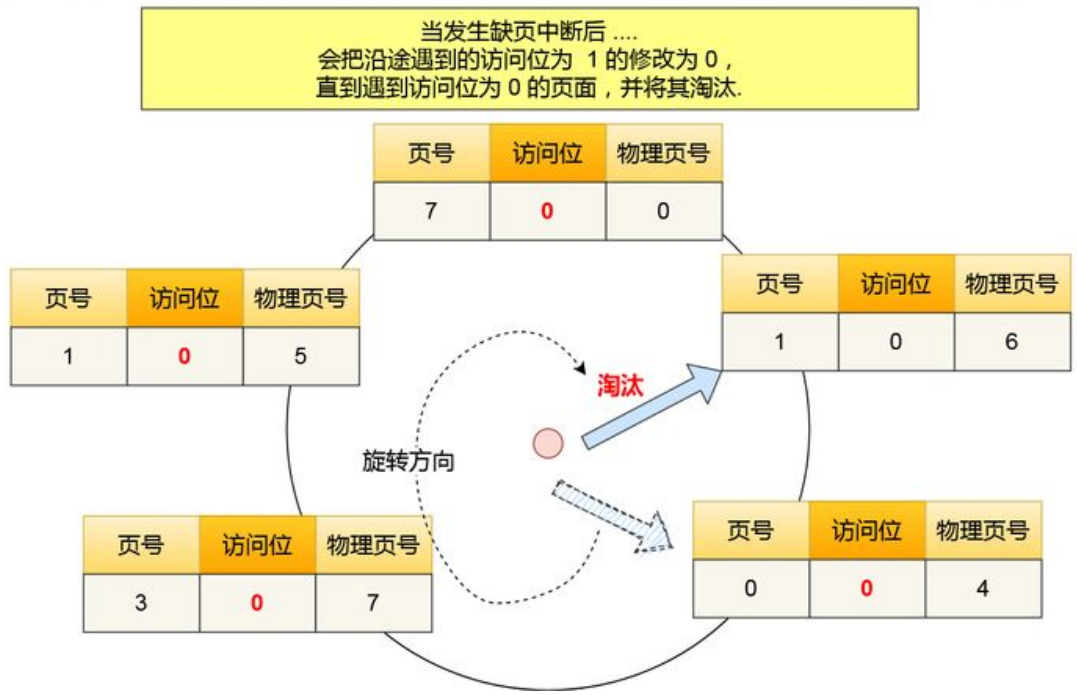
9. 时钟页面置换算法（Lock）

该算法的思路是，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；





10. **最不常用算法**

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。

它的实现方式是，对每个页面设置一个「访问计数器」，每当一个页面被访问时，该页面的访问计数器就累加 1。在发生缺页中断时，淘汰计数器值最小的那个页面。

缺点:

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。

但还有个问题，LFU 算法只考虑了频率问题，没考虑时间的问题，比如有些页面在过去时间里访问的频率很高，但是现在已经没有访问了，而当前频繁访问的页面由于没有这些页面访问的次数高，在发生缺页中断时，就会可能会误伤当前刚开始频繁访问，但访问次数还不高的页面。

那这个问题的解决的办法还是有的，可以定期减少访问的次数，比如当发生时间中断时，把过去时间访问的页面的访问次数除以 2，也就说，随着时间的流失，以前的高访问次数的页面会慢慢减少，相当于加大了被置换的概率。

14. 多级页表如何节约内存

原文链接: https://www.polarxiong.com/archives/%E5%A4%9A%E7%BA%A7%E9%A1%B5%E8%A1%A8%E5%A6%82%E4%BD%95%E8%8A%82%E7%BA%A6%E5%86%85%E5%AD%98.html

个人理解: 虚拟内存是为了能够离散的分配内存空间。由于程序局部性，并不是所有虚拟内存映射的物理内存空间都必须用得到（即分配），因此单级页表映射了所有的虚拟空间（实际程序运行时只会分配其中的一小部分）将会是一种浪

费，多级页表的存在是为了在能够满足所有虚拟地址得到映射的情况下（若没有全部映射，则MMU在使用时会找不到没有映射的物理内存地址无法为程序工作），更好的减小保存在内存中的页表大小。

15. 死锁

原文链接: https://blog.csdn.net/qq_34827674/article/details/115365866

两个线程为了保护两个不同的共享资源而使用了两个互斥锁，那么这两个互斥锁应用不当的时候，可能会造成**两个线程都在等待对方释放锁**，在没有外力的作用下，这些线程会一直相互等待，就没办法继续运行，这种情况就是发生了**死锁。**

死锁的条件(重要)

- **互斥条件**： 多个线程不能同时使用同一个资源
- **保持与请求条件**：线程等待资源的同时不会释放自己已有资源
- **不可抢占条件**：线程持有的资源在使用完之前不能被其他线程获取
- **循环等待条件**：两个线程获取资源的顺序构成环链

16. 互斥锁和自旋锁

多线程访问共享资源的时候，避免不了资源竞争而导致数据错乱的问题，所以我们通常为了解决这一问题，都会在访问共享资源之前加锁。

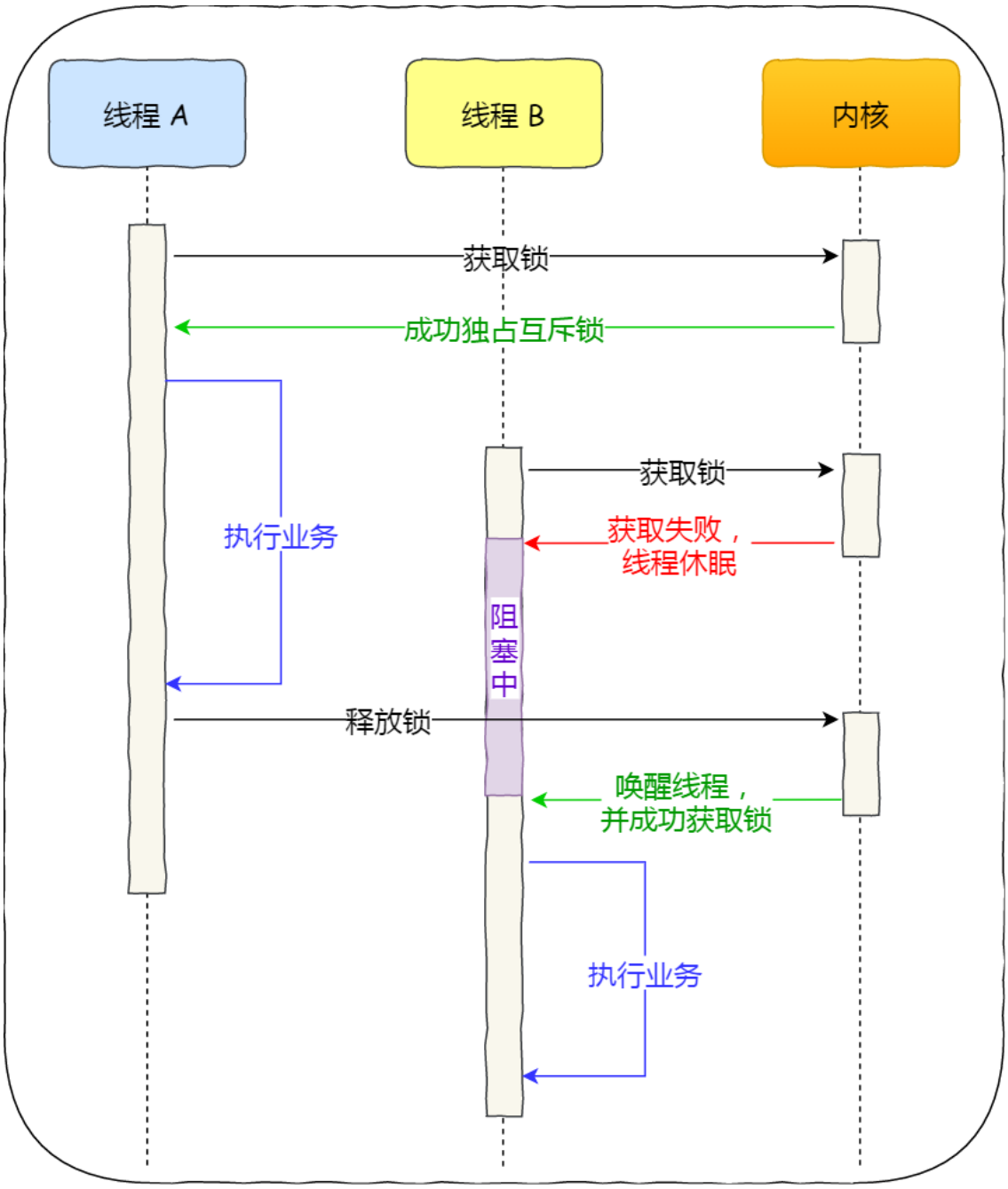
1. 互斥锁和自旋锁

   加锁的目的就是保证共享资源在任意时间里，只有一个线程访问，这样就可以避免多线程导致共享数据错乱的问题。

   - **互斥锁**加锁失败后，线程会**释放 CPU** ，给其他线程；
   - **自旋锁**加锁失败后，线程会**忙等待**，直到它拿到锁；

   互斥锁是一种「**独占锁**」，比如当线程 A 加锁成功后，此时互斥锁已经被线程 A 独占了，只要线程 A 没有释放手中的锁，线程 B 加锁就会失败，于是就会释放 CPU 让给其他线程，**既然线程 B 释放掉了 CPU，自然线程 B 加锁的代码就会被阻塞。**

   对于互斥锁加锁失败而阻塞的现象，是由操作系统内核实现的。当加锁失败时，内核会将线程置为「**睡眠**」状态，等到锁被释放后，内核会在合适的时机唤醒线程，当这个线程成功获取到锁后，于是就可以继续执行。如下图：



互斥锁加锁失败的开销:

会从用户态陷入到内核态，让内核帮我们切换线程，虽然简化了使用锁的难度，但是存在一定的性能开销成本。

那这个开销成本是什么呢？会有两次线程上下文切换的成本：

- 当线程加锁失败时，内核会把线程的状态从「**运行**」状态设置为「**睡眠**」状态，然后把 CPU 切换给其他线程运行；

- 接着，当锁被释放时，之前「**睡眠**」状态的线程会变为「**就绪**」状态，然后内核会在合适的时间，把 CPU 切换给该线程运行。

线程的上下文切换的是什么？当两个线程是属于同一个进程，**因为虚拟内存是共享的，所以在切换时，虚拟内存这些资源就保持不动，只需要切换线程的私有数据、寄存器等不共享的数据。**

上下切换的耗时有大佬统计过，大概在几十纳秒到几微秒之间，如果你锁住的代码执行时间比较短，那可能上下文切换的时间都比你锁住的代码执行时间还要长。

所以，**如果你能确定被锁住的代码执行时间很短，就不应该用互斥锁，而应该选用自旋锁，否则使用互斥锁。**

自旋锁的实现

自旋锁是通过 CPU 提供的 *`CAS`** 函数（Compare And Swap*），在「**用户态**」完成加锁和解锁操作，**不会主动产生线程上下文切换**，所以相比互斥锁来说，会快一些，开销也小一些。

一般加锁的过程，包含两个步骤：

- 第一步，查看锁的状态，如果锁是空闲的，则执行第二步；
- 第二步，将锁设置为当前线程持有；

CAS 函数就把这两个步骤合并成一条硬件级指令，形成原子指令，这样就保证了这两个步骤是不可分割的，要么一次性执行完两个步骤，要么两个步骤都不执行。

使用自旋锁的时候，当发生多线程竞争锁的情况，加锁失败的线程会「**忙等待**」，直到它拿到锁。这里的「忙等待」可以用 while 循环等待实现，不过最好是**使用 CPU 提供的 PAUSE 指令**来实现「忙等待」，因为可以减少循环等待时的耗电量。

自旋锁是最比较简单的一种锁，一直自旋，利用 CPU 周期，直到锁可用。需要注意，在单核 CPU 上，需要抢占式的调度器（即不断通过时钟中断一个线程，运行其他线程）。否则，自旋锁在单 CPU 上无法使用，因为一个自旋的线程永远不会放弃 CPU。

自旋锁开销少，在多核系统下一般不会主动产生线程切换，适合异步、协程等在用户态切换请求的编程方式，但如果被锁住的代码执行时间过长，自旋的线程会长时间占用 CPU 资源，所以自旋的时间和被锁住的代码执行的时间是成「正比」的关系，我们需要清楚的知道这一点。

自旋锁与互斥锁使用层面比较相似，但实现层面上完全不同：**当加锁失败时，互斥锁用「线程切换」来应对，自旋锁则用「忙等待」来应对。**

它俩是锁的最基本处理方式，更高级的锁都会选择其中一个来实现，比如读写锁既可以选择互斥锁实现，也可以基于自旋锁实现。

17. 读写锁

读写锁从字面意思我们也可以知道，它由「读锁」和「写锁」两部分构成，如果只读取共享资源用「读锁」加锁，如果要修改共享资源则用「写锁」加锁。

所以，**读写锁适用于能明确区分读操作和写操作的场景。**

读写锁的工作原理是：

- 当「写锁」没有被线程持有时，多个线程能够并发地持有读锁，这大大提高了共享资源的访问效率，因为「读锁」是用于读取共享资源的场景，所以多个线程同时持有读锁也不会破坏共享资源的数据。
- 但是，一旦「写锁」被线程持有后，读线程的获取读锁的操作会被阻塞，而且其他写线程的获取写锁的操作也会被阻塞。

所以说，写锁是独占锁，因为任何时刻只能有一个线程持有写锁，类似互斥锁和自旋锁，而读锁是共享锁，因为读锁可以被多个线程同时持有。

知道了读写锁的工作原理后，我们可以发现，**读写锁在读多写少的场景，能发挥出优势。**

读优先锁

当读线程 A 先持有了读锁，写线程 B 在获取写锁的时候，会被阻塞，并且在阻塞过程中，后续来的读线程 C 仍然可以成功获取读锁，最后直到读线程 A 和 C 释放读锁后，写线程 B 才可以成功获取写锁。

写优先锁

当读线程 A 先持有了读锁，写线程 B 在获取写锁的时候，会被阻塞，并且在阻塞过程中，后续来的读线程 C 获取读锁时会失败，于是读线程 C 将被阻塞在获取读锁的操作，这样只要读线程 A 释放读锁后，写线程 B 就可以成功获取读锁。

读优先锁对于读线程并发性更好，但也不是没有问题。我们试想一下，如果一直有读线程获取读锁，那么写线程将永远获取不到写锁，这就造成了写线程「饥饿」的现象。

写优先锁可以保证写线程不会饿死，但是如果一直有写线程获取写锁，读线程也会被「饿死」。

既然不管优先读锁还是写锁，对方可能会出现饿死问题，那么我们就不偏袒任何一方，搞个「公平读写锁」。

公平读写锁比较简单的一种方式是：**用队列把获取锁的线程排队，不管是写线程还是读线程都按照先进先出的原则加锁即可，这样读线程仍然可以并发，也不会出现「饥饿」的现象。**

18. 乐观锁悲观锁

前面提到的**互斥锁**、**自旋锁**、**读写锁**，都是属于**悲观锁**。

悲观锁做事比较悲观，它认为**多线程同时修改共享资源的概率比较高，于是很容易出现冲突，所以访问共享资源前，先要上锁。**

那相反的，如果多线程同时修改共享资源的概率比较低，就可以采用乐观锁。

乐观锁做事比较乐观，它假定冲突的概率很低，它的工作方式是：**先修改完共享资源，再验证这段时间内有没有发生冲突，如果没有其他线程在修改资源，那么操作完成，如果发现有其他线程已经修改过这个资源，就放弃本次操作。**

乐观锁全程并没有加锁，所以它也叫无锁编程。

我们常见的 SVN 和 Git 也是用了乐观锁的思想，先让用户编辑代码，然后提交的时候，通过版本号来判断是否产生了冲突，发生了冲突的地方，需要我们自己修改后，再重新提交。

乐观锁虽然去除了加锁解锁的操作，但是一旦发生冲突，重试的成本非常高，所以只有在冲突概率非常低，且加锁成本非常高的场景时，才考虑使用乐观锁。

19. 原子操作

"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。

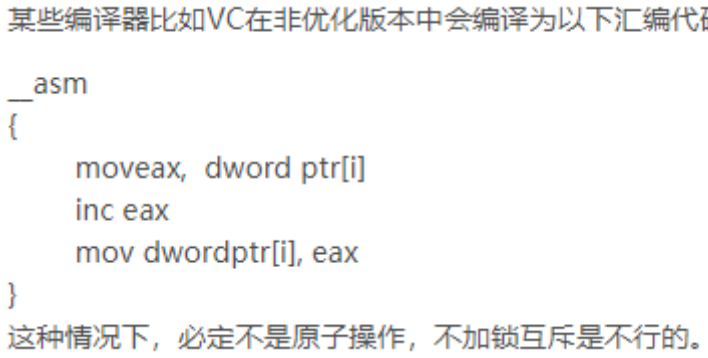
一个核在执行一个指令时，其他核同时执行的指令有可能操作同一块内存区域，从而出现数据竞争现象。多核系统中的原子操作通常使用**内存栅障**（memory barrier）来实现，即**一个CPU核在执行原子操作时，其他CPU核必须停止对内**

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。

20. i++和++i是原子操作码

不是



21. 进程间通信(IPC)

原文链接:https://www.cnblogs.com/xiaolincoding/p/13402297.html

在linux下的多个进程间的通信机制叫做IPC(Inter-Process Communication)，它是多个进程之间相互沟通的一种方法。在linux下有多种进程间通信的方法：半双工管道、命名管道、消息队列、信号、信号量、共享内存、内存映射文件，套接字等等。使用这些机制可以为linux下的网络服务器开发提供灵活而又坚固的框架。

1. **管道**
    - 匿名管道

```
ps auxf | grep mysql
```

- 命名管道

  有点像golang无缓冲channel

```
$ mkfifo myPipe
# shell1
$ echo "hello" > myPipe  // 将数据写进管道
                         // 停住了 ...
                         // shell2执行完后退出
# shell2
$ cat < myPipe  // 读取管道里的数据
hello
```

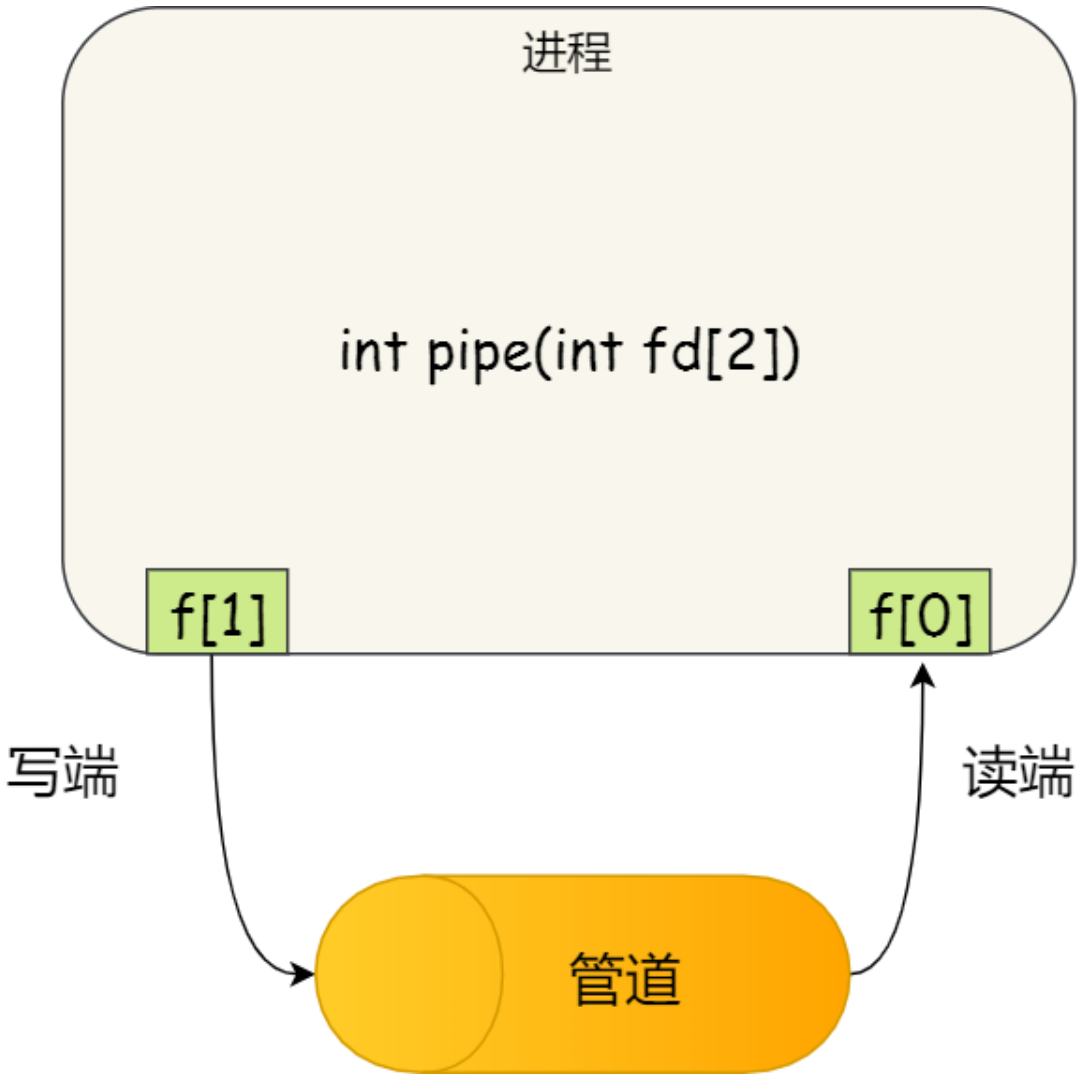
我们可以看出，**管道这种通信方式效率低，不适合进程间频繁地交换数据**。当然，它的好处，自然就是简单，同时也我们很容易得知管道里的数据已经被另一个进程读取了。

管道创建原理

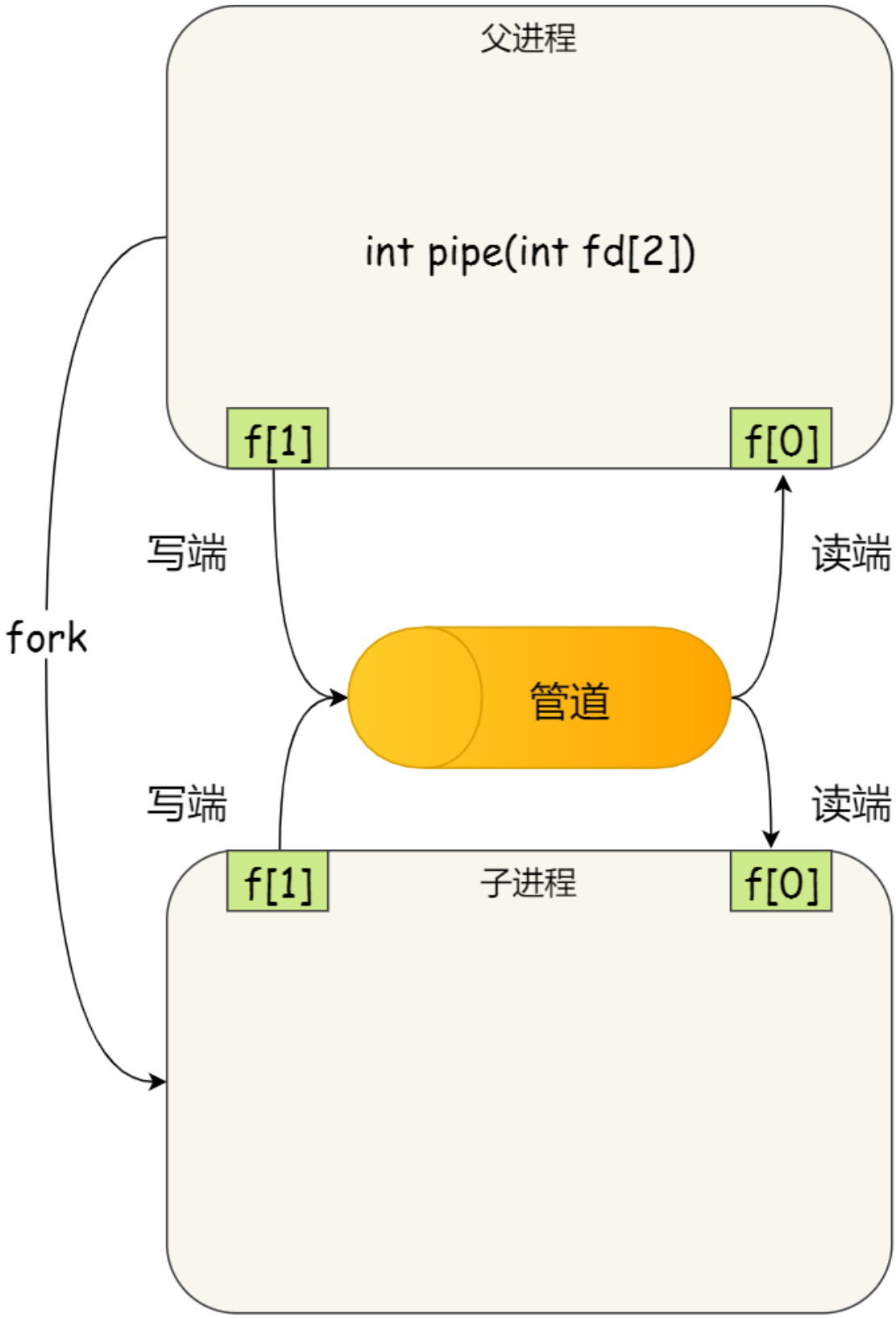
匿名管道的创建，需要通过下面这个系统调用：

```
init pipe(int fd[2]);
```

这里表示创建一个匿名管道，并返回了两个描述符，一个是管道的读取端描述符 `fd[0]`，另一个是管道的写入端描述符 `fd[1]`。注意，这个匿名管道是特殊的文件，只存在于内存，不存于文件系统中。



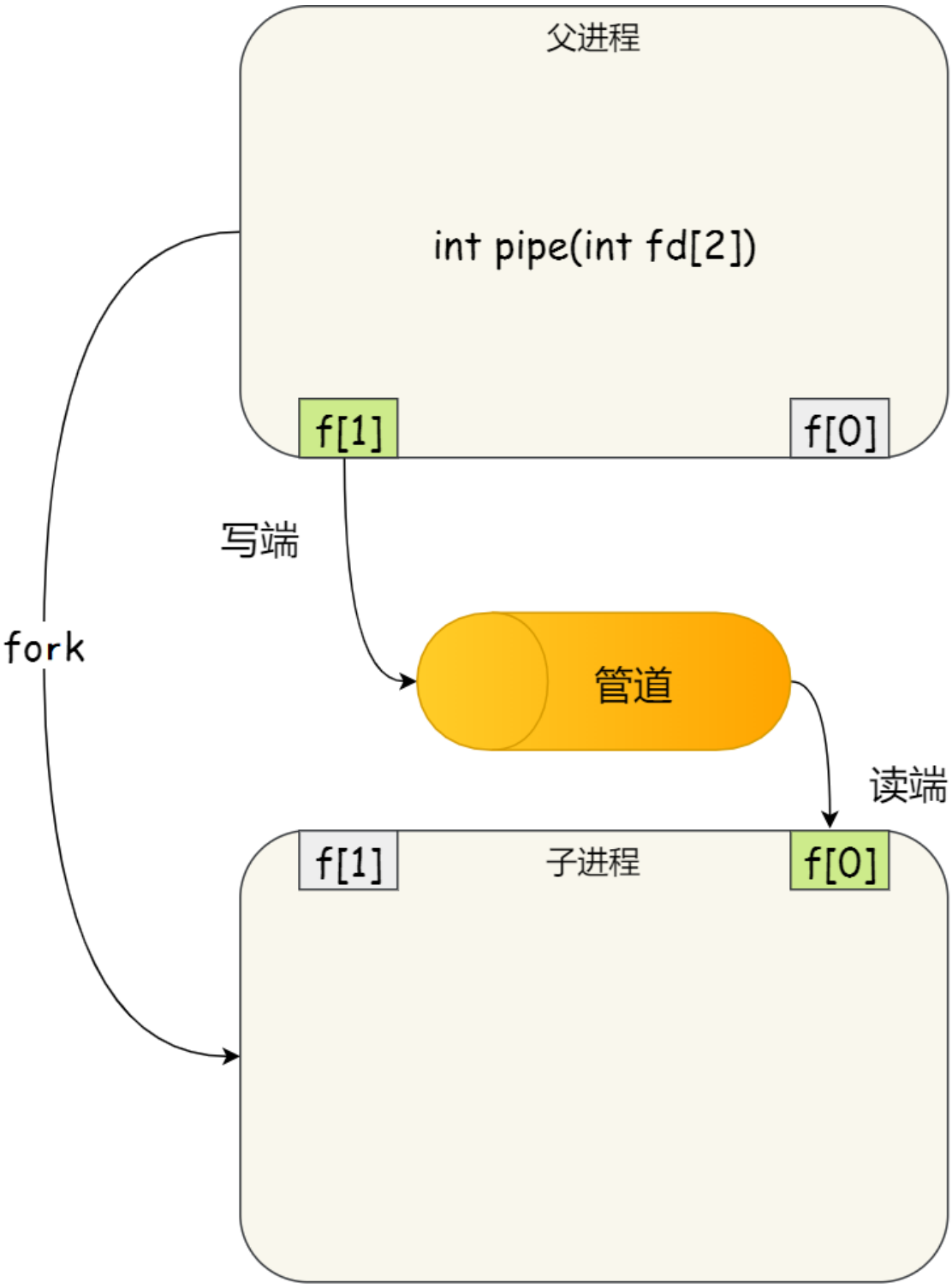
但是这样体现不了进程间通信，因此可以使用 `fork` 创建子进程，**创建的子进程会复制父进程的文件描述符**，这样就做到了两个进程各有两个「 `fd[0]` 与 `fd[1]`」，两个进程就可以通过各自的 fd 写入和读取同一个管道文件实现跨进程通信了



管道只能一端写入，另一端读出，所以上面这种模式容易造成混乱，因为父进程和子进程都可以同时写入，也都可以读出。那么，为了避免这种情况，通常的做法是：

- 父进程关闭读取的 fd[0]，只保留写入的 fd[1]；
- 子进程关闭写入的 fd[1]，只保留读取的 fd[0]；

```c
int close(int fd); //成功返回0，失败返回-1
```



所谓的管道，就是内核里面的一串缓存。从管道的一段写入的数据，实际上是缓存在内核中的，另一端读取，也就是从内核中读取这段数据。另外，管道传输的数据是无格式的流且大小受限。

我们可以得知，**对于匿名管道，它的通信范围是存在父子关系的进程**。因为管道没有实体，也就是没有管道文件，只能通过 fork 来复制父进程 fd 文件描述符，来达到通信的目的。

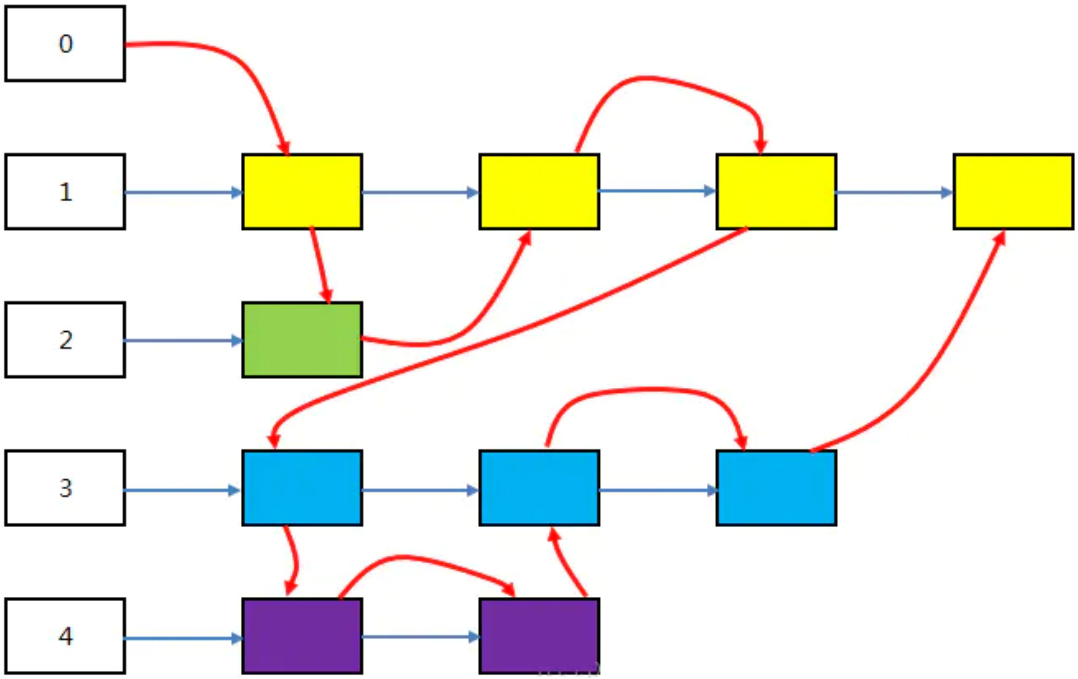
另外，**对于命名管道，它可以在不相关的进程间也能相互通信**。因为命令管道，提前创建了一个类型为管道的设备文件，在进程里只要使用这个设备文件，就可以相互通信。

不管是匿名管道还是命名管道，进程写入的数据都是缓存在内核中，另一个进程读取数据时候自然也是从内核中获取，同时通信数据都遵循**先进先出**原则，不支持 lseek 之类的文件定位操作。

---

2. **消息队列**

消息队列是保存在内核空间中的消息链表，在发送数据时，会分成一个一个独立的数据单元，也就是消息体（数据块），消息体是用户自定义的数据类型，消息的发送方和接收方要约定好消息体的数据类型，所以每个消息体都是固定大小的存储块，不像管道是无格式的字节流数据。如果进程从消息队列中读取了消息体，内核就会把这个消息体删除。每一条消息都有自己的消息类型，消息类型用整数来表示，而且必须大于 0。每种类型的消息都被对应的链表所维护



消息这种模型，两个进程之间的通信就像平时发邮件一样，你来一封，我回一封，可以频繁沟通了。

但邮件的通信方式存在不足的地方有两点，**一是通信不及时，二是附件也有大小限制**，这同样也是消息队列通信不足的点。

缺点

- **消息队列不适合比较大数据的传输**，因为在内核中每个消息体都有一个最大长度的限制，同时所有队列所包含的全部消息体的总长度也是有上限。在 Linux 内核中，会有两个宏定义 `MSGMAX` 和 `MSGMNB`，它们以字节为单位，分别定义了一条消息的最大长度和一个队列的最大长度。
- **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。

linux中相关系统调用

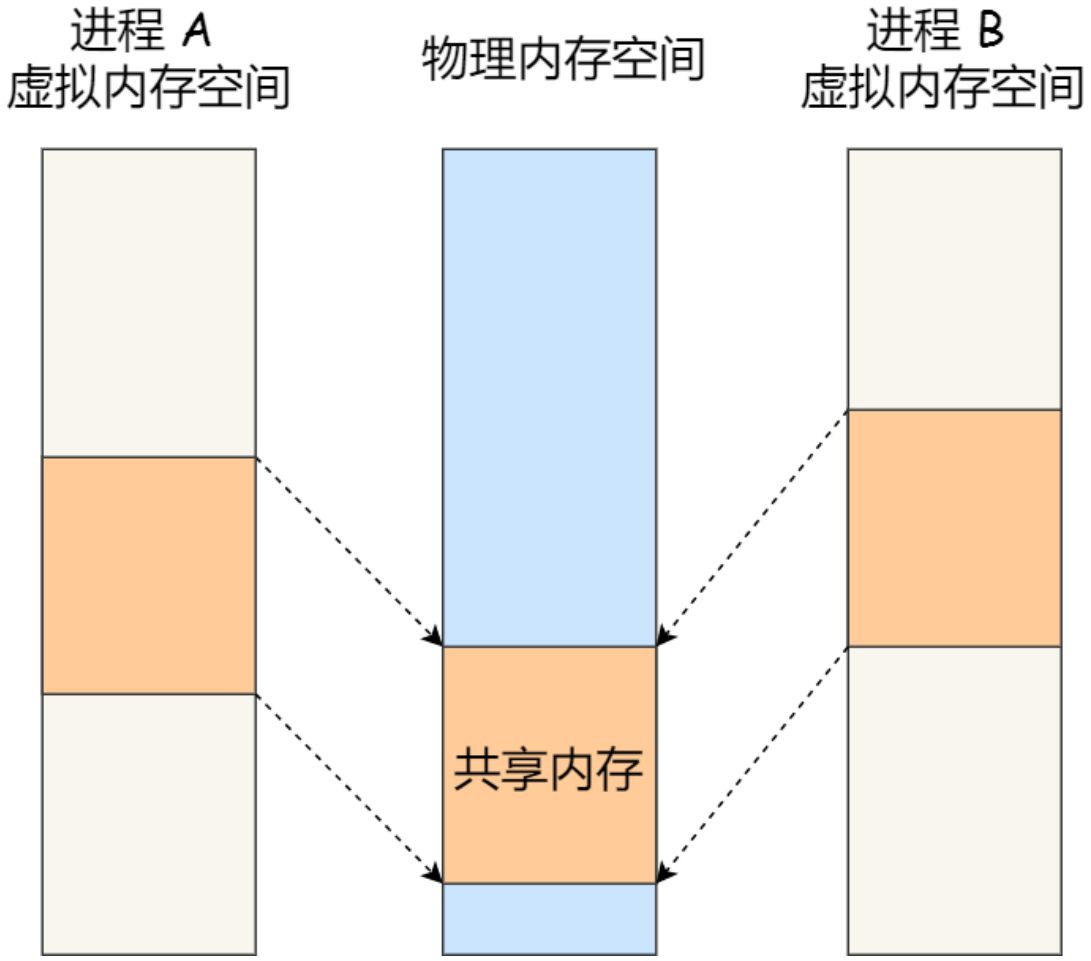
使用时要先有一个IPC对象

```
// 创建和获取 ipc 内核对象
int msgget(key_t key, int flags);
// 将消息发送到消息队列
int msgsnd(int msqid, const void *msgp,
size_t msgsz, int msgflg);
// 从消息队列获取消息
ssize_t msgrcv(int msqid, void *msgp,
size_t msgsz, long msgtyp, int msgflg);
// 查看、设置、删除 ipc 内核对象（用法和 shmctl
一样）
int msgctl(int msqid, int cmd, struct
msqid_ds *buf);
```

---

3. **共享内存**

共享内存的机制，就是拿出一块虚拟地址空间来，映射到相同的物理内存中。这样这个进程写入的东西，另外一个进程马上就能看到了，都不需要拷贝来拷贝去，传来传去，大大提高了进程间通信的速度。



4. **信号量**

   信号量其实是一个整型的计数器，主要用于实现进程间的互斥与同步，而不是用于缓存进程间通信的数据。

   信号量表示资源的数量，控制信号量的方式有两种原子操作：

   - 一个是 **P 操作**，这个操作会把信号量减去 -1，相减后如果信号量 < 0，则表明资源已被占用，进程需阻塞等待；相减后如果信号量 >= 0，则表明还有资源可使用，进程可正常继续执行。
   - 另一个是 **V 操作**，这个操作会把信号量加上 1，相加后如果信号量 <= 0，则表明当前有阻塞中的进程，于是会将该进程唤醒运行；相加后如果信号量 > 0，则表明当前没有阻塞中的进程；

5. **信号量**

上面说的进程间通信，都是常规状态下的工作模式。**对于异常情况下的工作模式，就需要用「信号」的方式来通知进程。**

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

运行在 shell 终端的进程，我们可以通过键盘输入某些组合键的时候，给进程发送信号。例如

- Ctrl+C 产生 `SIGINT` 信号，表示终止该进程；
- Ctrl+Z 产生 `SIGTSTP` 信号，表示停止该进程，但还未结束；

---

6. **socket**

```
int socket(int domain, int type, int protocal)
```

三个参数分别代表：

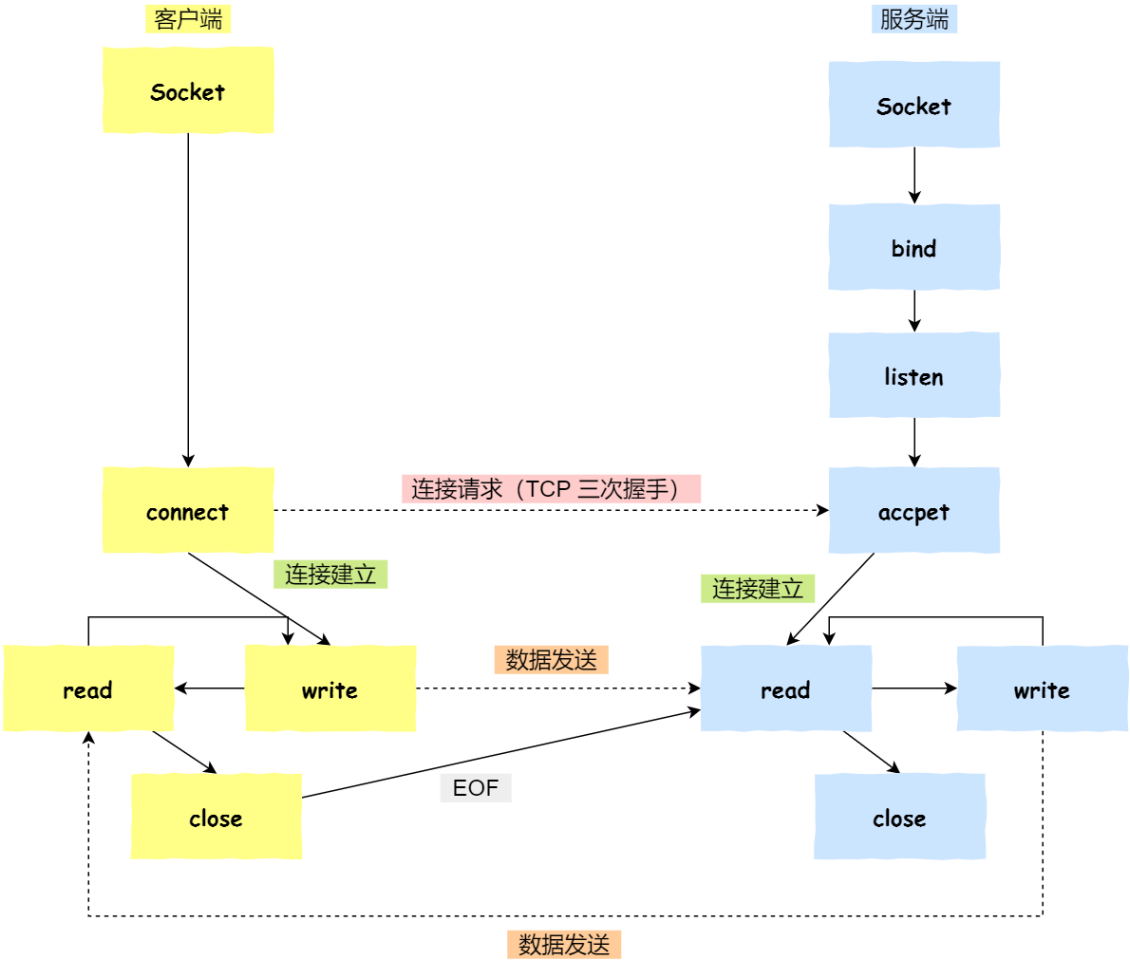
- domain 参数用来指定协议族，比如 AF_INET 用于 IPV4、AF_INET6 用于 IPV6、AF_LOCAL/AF_UNIX 用于本机；
- type 参数用来指定通信特性，比如 SOCK_STREAM 表示的是字节流，对应 TCP、SOCK_DGRAM 表示的是数据报，对应 UDP、SOCK_RAW 表示的是原始套接字；
- protocal 参数原本是用来指定通信协议的，但现在基本废弃。因为协议已经通过前面两个参数指定完成，protocol 目前一般写成 0 即可；

根据创建 socket 类型的不同，通信的方式也就不同：

- 实现 TCP 字节流通信： socket 类型是 AF_INET 和 SOCK_STREAM；
- 实现 UDP 数据报通信：socket 类型是 AF_INET 和 SOCK_DGRAM；
- 实现本地进程间通信： 「本地字节流 socket 」类型是 AF_LOCAL 和 SOCK_STREAM，「本地数据报 socket

」类型是 AF_LOCAL 和 SOCK_DGRAM。另外，AF_UNIX 和 AF_LOCAL 是等价的，所以 AF_UNIX 也属于本地 socket；

TCP协议通信的socket编程模型



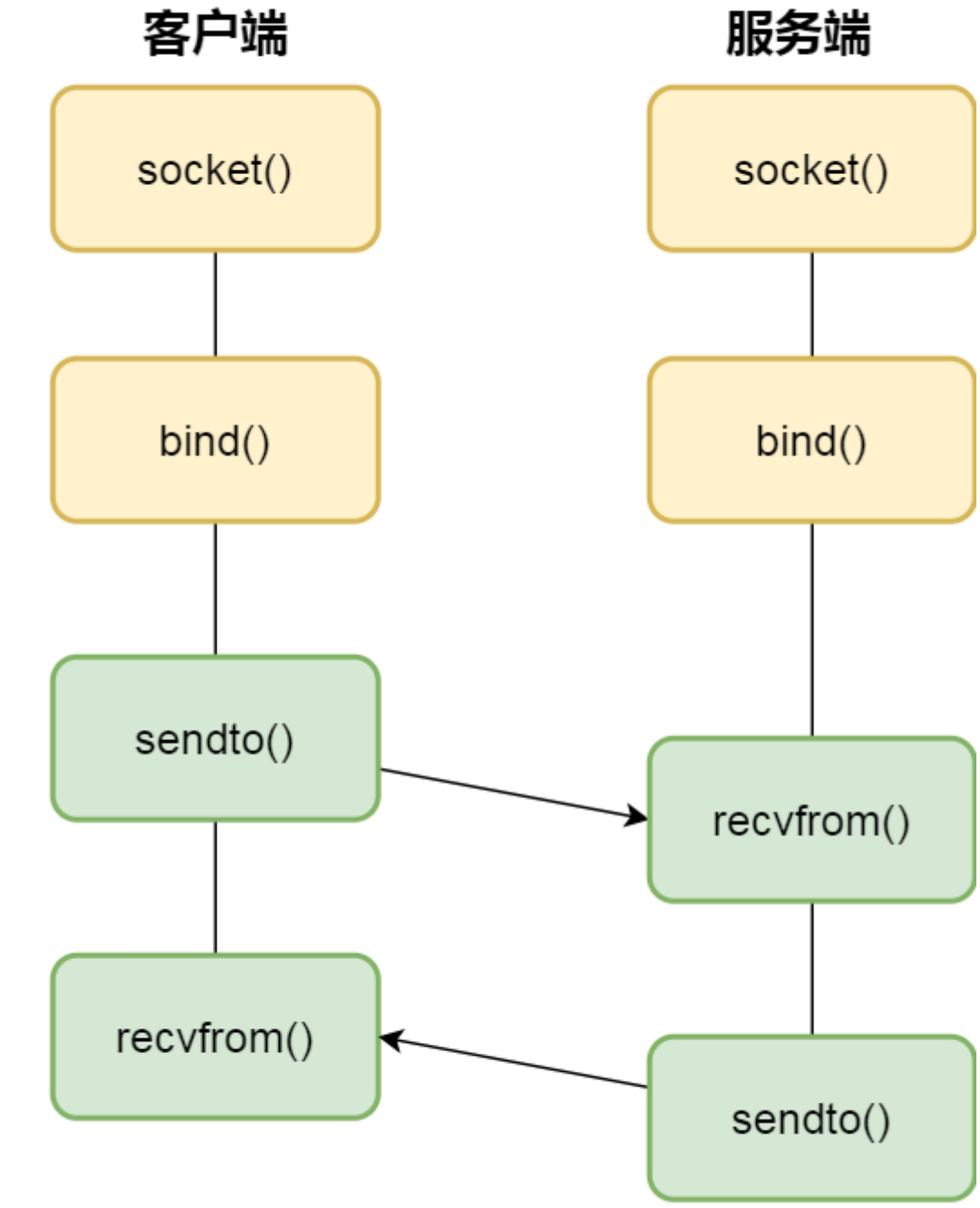
- 服务端和客户端初始化 `socket`，得到文件描述符；
- 服务端调用 `bind`，将绑定在 IP 地址和端口；
- 服务端调用 `listen`，进行监听；
- 服务端调用 `accept`，等待客户端连接；
- 客户端调用 `connect`，向服务器端的地址和端口发起连接请求；
- 服务端 `accept` 返回用于传输的 `socket` 的文件描述符；
- 客户端调用 `write` 写入数据；服务端调用 `read` 读取数据；

- 客户端断开连接时，会调用 `close`，那么服务端 `read` 读取数据的时候，就会读取到了 `EOF`，待处理完数据后，服务端调用 `close`，表示连接关闭。

这里需要注意的是，**服务端调用 `accept` 时，连接成功了会返回一个已完成连接的 socket**，后续用来传输数据。

所以，监听的 socket 和真正用来传送数据的 socket，是「**两个**」 socket，一个叫作**监听 socket**，一个叫作**已完成连接 socket**。

UDP协议通信的socket编程模型



UDP 是没有连接的，所以不需要三次握手，也就**不需要像 TCP 调用 listen 和 connect**，但是 UDP 的交互仍然需要 IP 地址和端口号，因此也需要 bind。

对于 UDP 来说，不需要要维护连接，那么也就没有所谓的发送方和接收方，甚至都不存在客户端和服务端的概念，只要有一个 socket 多台机器就可以任意通信，因此每一个 UDP 的 socket 都需要 bind。

另外，每次通信时，调用 sendto 和 recvfrom，都要传入目标主机的 IP 地址和端口。

本地进程通信的socket编程模型

本地 socket 被用于在**同一台主机上进程间通信**的场景：

- 本地 socket 的编程接口和 IPv4 、IPv6 套接字编程接口是一致的，可以支持「字节流」和「数据报」两种协议；
- 本地 socket 的实现效率大大高于 IPv4 和 IPv6 的字节流、数据报 socket 实现；

对于本地字节流 socket，其 socket 类型是 AF_LOCAL 和 SOCK_STREAM。

对于本地数据报 socket，其 socket 类型是 AF_LOCAL 和 SOCK_DGRAM。

本地字节流 socket 和 本地数据报 socket 在 bind 的时候，不像 TCP 和 UDP 要绑定 IP 地址和端口，而是**绑定一个本地文件**，这也就是它们之间的最大区别。

22. IO多路复用

一句话的解释: **单线程**或**单进程**同时**监测若干个文件描述符**是否可以执行IO操作的能力。

`多路复用`这个词多出现在网络编程，首先理解`多路`.
`多路`：有多个客户端连接，一路就是一个连接
`复用`：一个进程或线程处理上面所有的连接。如果不复用又需要同时服务多个客户端，需要多线程或多进程，这个就不是复用了

IO多路复用的方案

Linux: select、poll、epoll

MacOS/FreeBSD: kqueue

Windows/Solaris: IOCP

23. epoll,select,poll

1. **select**

   `select`中的时间轮询机制是基于比特位的，每次查询都要**遍历整个时间列表**

   `select`处理的数据结构是`fd_set`，每个select都要处理一个fd_set结构。

   fd_set可以理解为是一个长度为`1024`的**比特位**(可以设置，由FD_SETSIZE参数决定)，每一个比特位代表一个需要处理的文件描述符`fd`，如果是1，那么代表这个位对应的fd有需要处理的i/o时间。为了简化对fd_set位操作，linux提供了一组**宏函数**来处理。

```
void FD_CLR(int fd, fd_set *set);      // 清除fdset的所有位
int  FD_ISSET(int fd, fd_set *set);    // 检测fdset的位fd是否被设置
void FD_SET(int fd, fd_set *set);      // 设置fdset的位fd
void FD_ZERO(fd_set *set);             // 清除fdset的所有位，用来初始化
```

`select`的调用方式

```
#include <sys/select.h>
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

- `nfds`：表示表示文件描述符最大的数目+1，这个数目是指读事件和写事件中数目最大的，+1是为了全面检查
- `readfds`：表示需要监视的会发生读事件的fd，没有设置为NULL
- `writefds`：表示需要监视的会发生写事件的fd，没有设置为NULL
- `exceptfds`：表示异常处理的，暂时没用到。。。
- `timeout`：表示阻塞的时间，如果是0表示非阻塞模型，NULL表示永远阻塞，直到有数据来

```
struct timeval {
    long    tv_sec;         /* seconds */
    long    tv_usec;        /* microseconds */
};
```

有三个类型的返回值：

- 正数： 表示`readfds`和`writefds`就绪事件的总数
- 0：超时返回0
- -1：出现错误

eg:

```
int main() {


  fd_set read_fs, write_fs;
  struct timeval timeout;
  int max_sd = 0;  // 用于记录最大的fd，在轮询中时刻更新即可

  /*
   * 这里进行一些初始化的设置，
   * 包括socket建立，地址的设置等，
   * 同时记得初始化max_sd
   */

  // 初始化比特位
  FD_ZERO(&read_fs);
  FD_ZERO(&write_fs);

  int rc = 0;
  int desc_ready = 0; // 记录就绪的事件，可以减少遍历的次数
  while (1) {
    // 这里进行阻塞
    rc = select(max_sd + 1, &read_fd, &write_fd, NULL, &timeout);
    if (rc < 0) {
```

```
24          // 这里进行错误处理机制
25        }
26        if (rc == 0) {
27          // 这里进行超时处理机制
28        }
29
30        desc_ready = rc;
31        // 遍历所有的比特位，轮询事件
32        for (int i = 0; i <= max_sd &&
  desc_ready; ++i) {
33          if (FD_ISSET(i, &read_fd)) {
34            --desc_ready;
35            // 这里处理read事件，别忘了更新max_sd
36          }
37          if (FD_ISSET(i, &write_fd)) {
38            // 这里处理write事件，别忘了更新max_sd
39          }
40        }
41      }
42  }
```

这只是一个简单的模型，有时候还可能需要使用`FD_CTL`和`FD_SET`增加或者减少fd，根据实际情况灵活处理即可。

2. **poll**

`poll`是`select`的增强版，因为`select`中比特位限制了其处理的fd个数。而poll优化了这一点。

`poll`底层操作的数据结构`pollfd`

```
struct pollfd {
    int fd;           // 需要监视的文件描述符
    short events;     // 需要内核监视的事件，即注册的事件
    short revents;    // 实际发生的事件,由内核填充
};
```

其中`fd`成员指定文件描述符，`events`成员告诉poll需要监听fd上的**哪些事件**，他是一系列事件的按位或。`revents`成员则由内核修改，告诉应用程序fd上实际发生了什么事件。

表 9-1 poll 事件类型

事　件	描　述	是否可作为输入	是否可作为输出
POLLIN	数据（包括普通数据和优先数据）可读	是	是
POLLRDNORM	普通数据可读	是	是
POLLRDBAND	优先级带数据可读（Linux 不支持）	是	是
POLLPRI	高优先级数据可读，比如 TCP 带外数据	是	是
POLLOUT	数据（包括普通数据和优先数据）可写	是	是

（续）

事　件	描　述	是否可作为输入	是否可作为输出
POLLWRNORM	普通数据可写	是	是
POLLWRBAND	优先级带数据可写	是	是
POLLRDHUP	TCP 连接被对方关闭，或者对方关闭了写操作。它由 GNU 引入	是	是
POLLERR	错误	否	是
POLLHUP	挂起。比如管道的写端被关闭后，读端描述符上将收到 POLLHUP 事件	否	是
POLLNVAL	文件描述符没有打开	否	是

poll的操作

```
#include <poll.h>
int poll(struct pollfd* fds, nfds_t nfds, int timeout);
```

`fds`: 一个`pollfd`队列的队头指针，我们先把需要监视的文件描述符和他们上面的事件放到这个队列中

`nfds`: 指定被监听事件集合fds的大小，即**队列长度**

```
typedef unsigned long int nfds_t;
```

`timeout`: 指定poll的超时值，单位毫秒。-1表示永远阻塞，0表示立即返回。

eg

```
// 先宏定义长度
#define MAX_POLLFD_LEN 200

int main() {
  /*
   * 在这里进行一些初始化的操作，
   * 比如初始化数据和socket等。
   */

  int rc = 0;
  pollfd fds[MAX_POLL_LEN];
  memset(fds, 0, sizeof(fds));
  int ndfs  = 1;  // 队列的实际长度，是一个随时更新的，也可以自定义其他的
  int timeout = 0;
  /*
   * 在这里进行一些感兴趣事件的注册，
   * 每个pollfd可以注册多个类型的事件，
   * 使用 | 操作即可，就行博文提到的那样。
   * 根据需要设置阻塞时间
   */

  int current_size = ndfs;
  int compress_array = 0;  // 压缩队列的标记
  while (1) {
```

```
25      rc = poll(fds, nfds, timeout);
26      if (rc < 0) {
27      // 这里进行错误处理
28      }
29      if (rc == 0) {
30      // 这里进行超时处理
31      }
32
33      for (int i = 0; i < current_size; ++i)
{
34        if (fds[i].revents == 0){  // 没有事
件可以处理
35          continue;
36        }
37        if (fds[i].revents & POLLIN) {  //
简单的例子，比如处理写事件
38
39        }
40        /*
41         * current_size 是为了降低复杂度的，可以
随时进行更新
42         * ndfs如果要更新，应该是最后统一进行
43         */
44      }
45
46      if (compress_array) {  // 如果需要压缩队
列
47        compress_array = 0;
48        for (int i = 0; i < ndfs; ++i) {
49          for (int j = i; j < ndfs; ++j) {
50            fds[i].fd = fds[j + i].fd;
51          }
52          --i;
```

```
            --ndfs;
        }
      }
    }
}
```

3. **epoll**

`epoll`是linux特有的I/O函数。`epoll`使用一组函数来完成，而不是单个函数。其次，epoll把用户关心的文件描述符上的事件放在内核的一个事件表中。从而无须像select和poll那样每次调用都需要重复传入文件描述符或事件集。但是`epoll`需要使用一个**额外的文件描述符**，来**唯一标识内核中的这个事件表**。即直接申请一个`epollfd`的文件，对这些进行统一的管理。

这个文件描述符用如下函数创建:

```
#include <sys/epoll.h>
int epoll_create(int size);
```

size参数现在并不起作用，只是给内核一个提示，告诉他时间表需要多大。该函数返回的**文件描述符**将用做其他所有epoll系统调用的第一个参数，指定要访问的内核事件表。

还有另外一个创建函数

```
#include <sys/epoll.h>
int epoll_create1(int flag);
```

`flag==0`时，功能同上，另一个选项是`EPOLL_CLOEXEC`。这个选项的作用是当父进程fork出一个子进程的时候，**子进程不会包含`epoll`的`fd`**，这在多进程编程时十分有用。

`epoll`操作的底层数据结构

```
struct epoll_event {
    uint32_t events;
    epoll_data_t data;
};

typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;
```

注意到，`epoll_data`是一个`union`类型。fd很容易理解，是文件描述符；而文件描述符本质上是一个索引内核中资源地址的一个下标描述，因此也可以用ptr指针代替；同样的这些数据可以用整数代替。
再来看`epoll_event`，有一个data用于表示fd，之后又有一个events表示注册的事件。

操作epoll内核事件表

```
#include <sys.epoll.h>
int epoll_ctl(int epfd, int op, int fd,
struct epoll_event* event);
```

`epfd`: create的epoll事件管理文件描述符

`fd`: 要操作的文件描述符

`op`: 指定操作类型

- EPOLL_CTL_ADD: 往事件表注册fd上监听事件
- EPOLL_CTL_MOD: 修改fd上的注册事件
- EPOLL_CTL_DEL: 删除fd上的注册事件

`event`: 事件类型

使用方法:

```
//第三个参数是否设置et
void addfd(int epollfd, int fd, bool enable_et) {
    epoll_event event;
    event.data.fd = fd;
    event.events = EPOLLIN;
    if(enable_et) {
        event.evnets |= EPOLLET;
    }
    epoll_ctl(epollfd, EPOLL_CTL_ADD, fd, &event);
    setnonblocking(fd);      //自定义函数，设置非阻塞
}
```

等待epoll事件返回

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

- `epfd`： create的`epoll`的文件描述符
- `events`: epoll_wait如果检测到事件，就把所有就绪事件从内核事件表(由epfd指定的那个)中**复制到这个参数中**。这个列表**只输出epoll_wait检测到的就绪事件**。

- `maxevents`: 最多监听多少个事件
- `timeout`: 指定epoll的超时值，单位毫秒。-1表示永远阻塞，0表示立即返回。

eg:

```
#define MAX_EVENTS 10
int main() {
    struct epoll_event ev, events[MAX_EVENTS];
    int listen_sock, conn_sock, nfds, epollfd;

    /* Code to set up listening socket, 'listen_sock',
     (socket(), bind(), listen()) omitted */

    epollfd = epoll_create1(0);
    if (epollfd == -1) {
        perror("epoll_create1");
        exit(EXIT_FAILURE);
    }

    ev.events = EPOLLIN;
    ev.data.fd = listen_sock;
    if (epoll_ctl(epollfd, EPOLL_CTL_ADD, listen_sock, &ev) == -1) {
        perror("epoll_ctl: listen_sock");
        exit(EXIT_FAILURE);
    }

    for (;;) {
```

```
23          // 永久阻塞等待epoll_wait返回，直到有事件，事件复制到第二个参数中
24          nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);
25          if (nfds == -1) {  // 处理错误
26              perror("epoll_wait");
27              exit(EXIT_FAILURE);
28          }
29
30          // nfds监听到的有I/O的fd个数
31          for (n = 0; n < nfds; ++n) {
32              //如果这个是监听连接的socket
33              if (events[n].data.fd == listen_sock) {
34                  conn_sock = accept(listen_sock, (struct sockaddr *) &addr, &addrlen);
35                  if (conn_sock == -1) {
36                      perror("accept");
37                      exit(EXIT_FAILURE);
38                  }
39                  setnonblocking(conn_sock);
40                  ev.events = EPOLLIN | EPOLLET;
41                  ev.data.fd = conn_sock;
42                  if (epoll_ctl(epollfd, EPOLL_CTL_ADD, conn_sock, &ev) == -1) {
43                      perror("epoll_ctl: conn_sock");
44                      exit(EXIT_FAILURE);
45                  }
46              } else {
47                  //否则对应处理
```

```
48
    do_use_fd(events[n].data.fd);
49              }
50          }
51      }
52      return 0;
53  }
```

poll和epoll使用上区别

代码清单 9-2　poll 和 epoll 在使用上的差别

```
/* 如何索引 poll 返回的就绪文件描述符 */
int ret = poll( fds, MAX_EVENT_NUMBER, -1 );
/* 必须遍历所有已注册文件描述符并找到其中的就绪者（当然，可以利用 ret 来稍做优化）*/
for( int i = 0; i < MAX_EVENT_NUMBER; ++i )
{
      if( fds[i].revents & POLLIN )  /* 判断第 i 个文件描述符是否就绪 */
      {
          int sockfd = fds[i].fd;
          /* 处理 sockfd */
      }
}

/* 如何索引 epoll 返回的就绪文件描述符 */
int ret = epoll_wait( epollfd, events, MAX_EVENT_NUMBER, -1 );
/* 仅遍历就绪的 ret 个文件描述符 */
for ( int i = 0; i < ret; i++ )
{
      int sockfd = events[i].data.fd;
      /* sockfd 肯定就绪，直接处理 */
}
```

三者比较

表 9-2　select、poll 和 epoll 的区别

系统调用	select	poll	epoll
事件集合	用户通过 3 个参数分别传入感兴趣的可读、可写及异常等事件，内核通过对这些参数的在线修改来反馈其中的就绪事件。这使得用户每次调用 select 都要重置这 3 个参数	统一处理所有事件类型，因此只需一个事件集参数，用户通过 pollfd.events 传入感兴趣的事件，内核通过修改 pollfd.revents 反馈其中就绪的事件	内核通过一个事件表直接管理用户感兴趣的所有事件，因此每次调用 epoll_wait 时，无须反复传入用户感兴趣的事件，epoll_wait 系统调用的参数 events 仅用来反馈就绪的事件
应用程序索引就绪文件描述符的时间复杂度	O(n)	O(n)	O(1)
最大支持文件描述符数	一般有最大值限制	65 535	65 535
工作模式	LT	LT	支持 ET 高效模式
内核实现和工作效率	采用轮询方式来检测就绪事件，算法时间复杂度为 O(n)	采用轮询方式来检测就绪事件，算法时间复杂度为 O(n)	采用回调方式来检测就绪事件，算法时间复杂度为 O(1)

24. ET和LT

ET和LT是epoll对文件描述符操作的两种模式。LT是默认的，这种情况下的EPOLL相当于效率较高的POLL

`ET（边沿触发）`: epoll_wait检测到有事件发生后，我们的编写的应用程序应该立刻处理该事件，如在while完读完全部数据，因为这种模式下若读写没完成，下次该事件不会再被捕获。

`LT（水平触发）`:与ET相反，如果单次事件没有处理完成，不要紧，下次他还会在epoll_wait中再次被捕获。

因此ET模式很大程度降低了对同一个epoll事件被重复触发的次数，效率比LT要高。

25. EPOLLONESHOT事件

问题:

ET模式下，如果对socket上一次读操作没完成，则一个线程（或进程）在处理这部分已经读出来的数据时，这时候该socket在下次epoll_wait时又有新数据可读，此时又开了一个新线程来读取这部分数据，造成多个线程同时操作一个socket的局面。

`EPOLLONESHOT`是`epoll`特有的事件，操作系统上最多触发文件描述符上注册的一个可读、可写或者异常事件，只能触发一次，除非使用`epoll_ctl`重置该描述符。这在多线程编程时常用到，处理完毕后需要重新复原。

这样当一个线程处理某个socket时，其他线程就不能再操作该socket了。

26. epoll底层实现

原文链接:https://www.cnblogs.com/developing/articles/10849288.html

https://blog.csdn.net/qq_29108585/article/details/60468522

当某一进程调用epoll_create方法时，Linux内核会创建一个eventpoll结构体，这个结构体中有两个成员与epoll的使用方式密切相关。eventpoll结构体如下所示：

```
struct eventpoll{
    ....
    /*红黑树的根节点，这颗树中存储着所有添加到epoll中的需要监控的事件*/
    struct rb_root  rbr;
    /*就绪链表，双链表中则存放着将要通过epoll_wait返回给用户的满足条件的事件*/
    struct list_head rdlist;
    ....
};
```

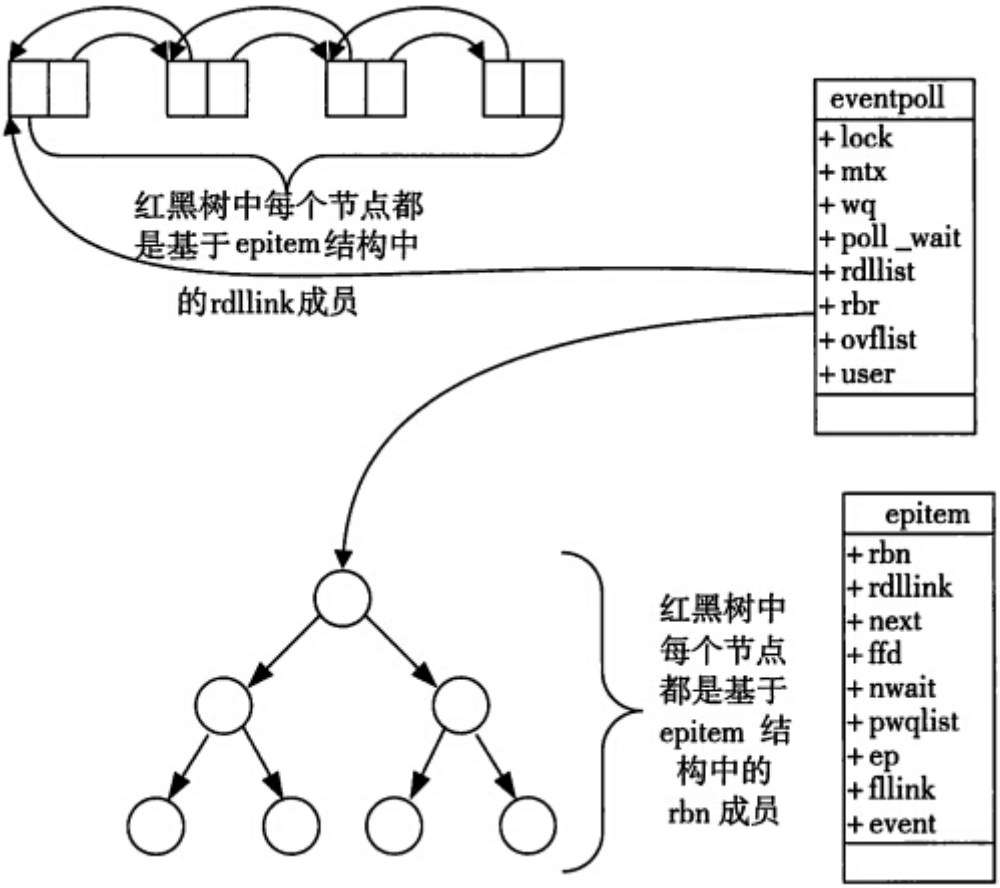
首先epoll_create创建一个epoll文件描述符，底层同时创建一个**红黑树**，和一个**就绪链表**；红黑树存储所监控的文件描述符的节点数据，就绪链表存储就绪的文件描述符的节点数据；epoll_ctl将会添加新的描述符，首先判断是红黑树上是否有此文件描述符节点，如果有，则立即返回。如果没有， 则在树干上插入新的节点，并且告知**内核注册回调函数**。当接收到某个文件描述符过来数据时，那么内核将该节点插入到就绪链表里面。epoll_wait将会接收到消息，并且将数据拷贝到用户空间，清空链表。对于LT模式epoll_wait清空就绪链表

之后会检查该文件描述符是哪一种模式，**如果为LT模式，且必须该节点确实有事件未处理，那么就会把该节点重新放入到刚刚删除掉的且刚准备好的就绪链表，epoll_wait马上返回。ET模式不会检查，只会调用一次**

在epoll中，对于每一个事件，都会建立一个epitem结构体，如下所示：

```
struct epitem{
    struct rb_node  rbn;//红黑树节点
    struct list_head    rdllink;//双向链表节点
    struct epoll_filefd  ffd;  //事件句柄信息
    struct eventpoll *ep;    //指向其所属的eventpoll对象
    struct epoll_event event; //期待发生的事件类型
}
```

当调用epoll_wait检查是否有事件发生时，只需要检查**eventpoll对象中的rdlist双链表中是否有epitem元素**即可。如果rdlist不为空，则把发生的事件复制到用户态，同时将事件数量返回给用户，然后清空链表。



从上面的讲解可知：通过红黑树和双链表数据结构，并结合回调机制，造就了epoll的高效。