IO多路复用

从事服务端开发，少不了要接触网络编程。epoll作为linux下高性能网络服务器的必备技术至关重要，nginx、redis、skynet和大部分游戏服务器都使用到这一多路复用技术。

因为epoll的重要性，不少游戏公司在招聘服务端同学时，会问及epoll相关的问题。比如epoll和select的区别是什么？epoll高效率的原因是什么？如果只靠背诵，显然不能算上深刻的理解。

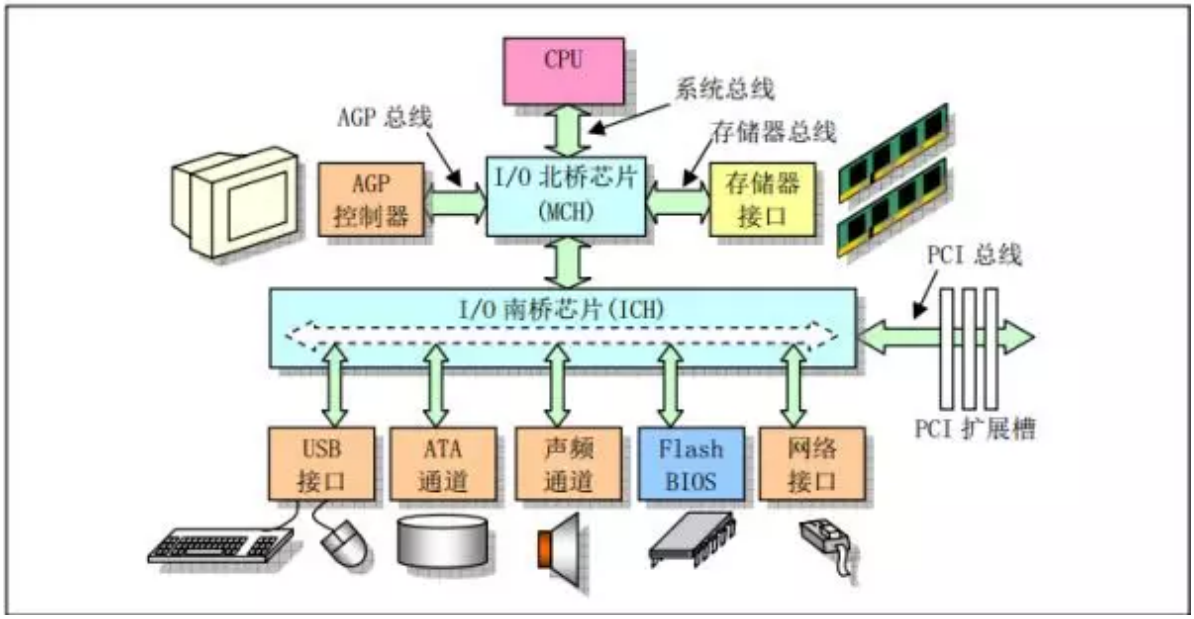
网上虽然有不少讲解epoll的文章，但要不是过于浅显，就是陷入源码解析，很少能有通俗易懂的。于是决定编写此文，让缺乏专业背景知识的读者也能够明白epoll的原理。文章核心思想是：

要让读者清晰明白EPOLL为什么性能好。

本文会从网卡接收数据的流程讲起，串联起CPU中断、操作系统进程调度等知识；再一步步分析阻塞接收数据、select到epoll的进化过程；最后探究epoll的实现细节。

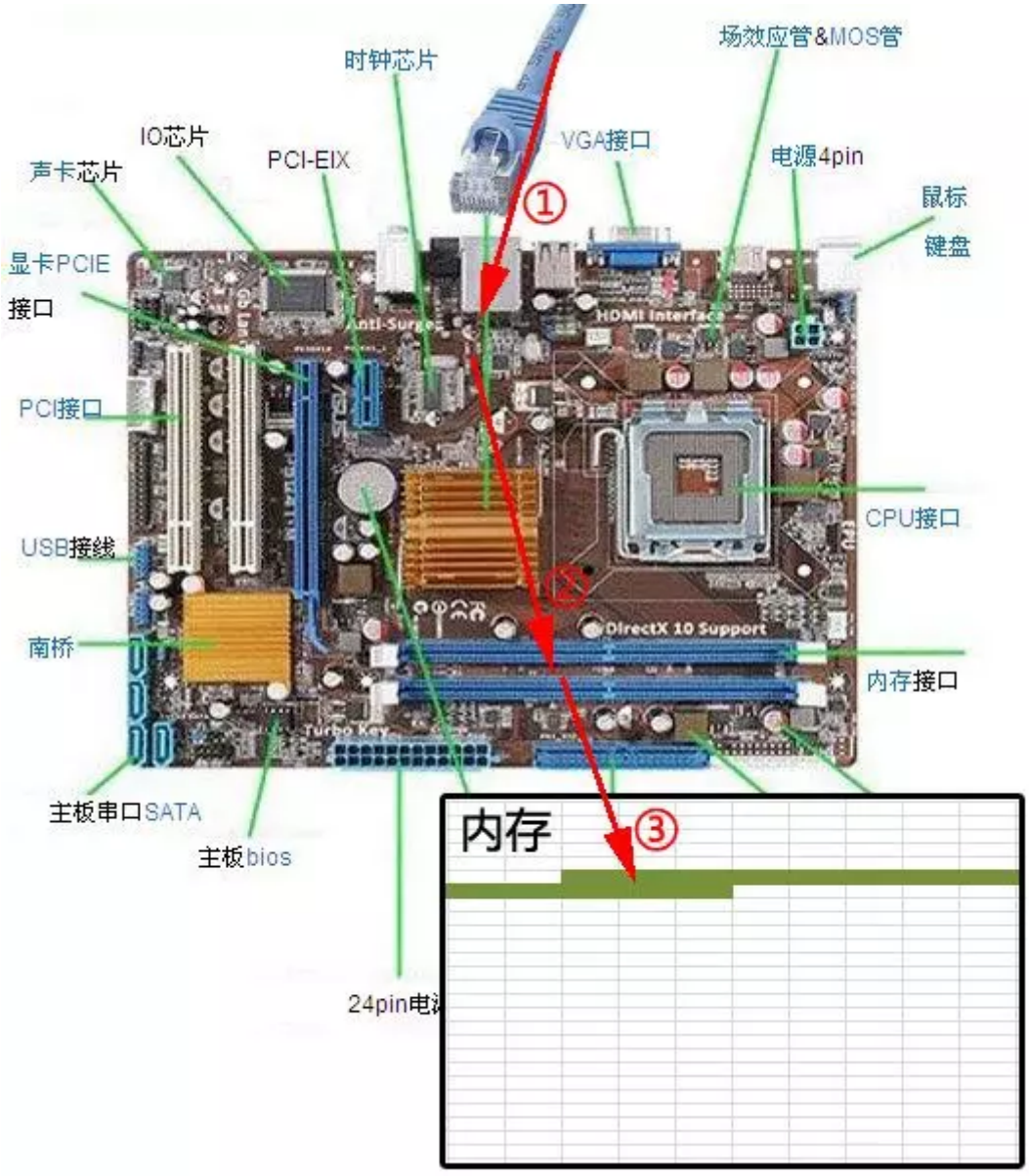
一、从网卡接收数据说起

下图是一个典型的计算机结构图，计算机由CPU、存储器（内存）、网络接口等部件组成。了解epoll本质的第一步，要从硬件的角度看计算机怎样接收网络数据。



计算机结构图（图片来源：linux内核完全注释之微型计算机组成结构）

下图展示了网卡接收数据的过程。在①阶段，网卡收到网线传来的数据；经过②阶段的硬件电路的传输；最终将数据写入到内存中的某个地址上（③阶段）。这个过程涉及到DMA传输、IO通路选择等硬件有关的知识，但我们只需知道：**网卡会把接收到的数据写入内存。**



网卡接收数据的过程

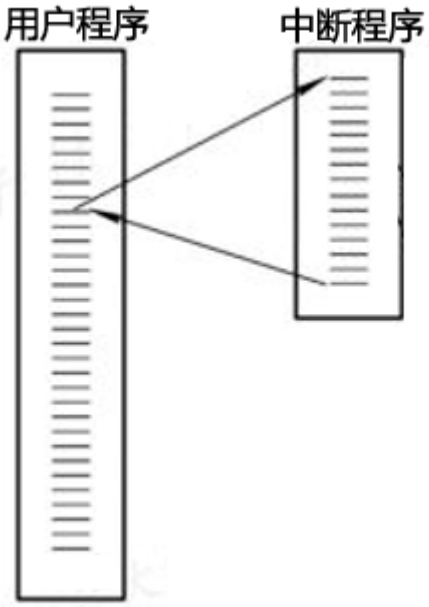
通过硬件传输，网卡接收的数据存放到内存中。操作系统就可以去读取它们。

二、如何知道接收了数据?

了解epoll本质的第二步，要从CPU的角度来看数据接收。要理解这个问题，要先了解一个概念—中断。

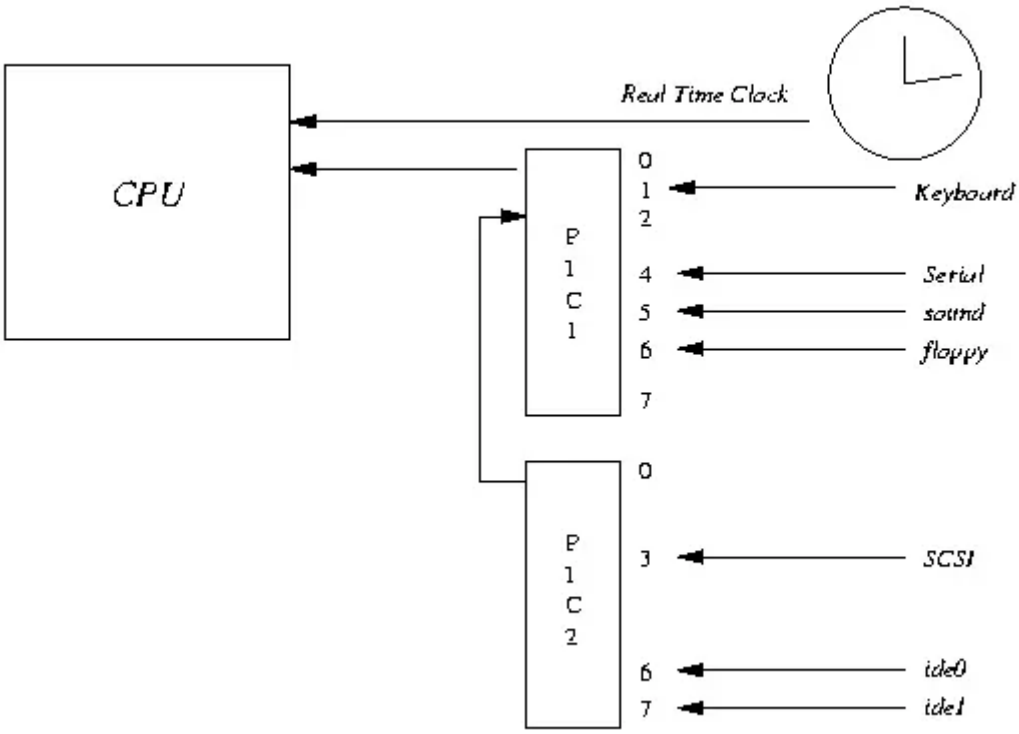
计算机执行程序时，会有优先级的需求。比如，当计算机收到断电信号时（电容可以保存少许电量，供CPU运行很短的一小段时间），它应立即去保存数据，保存数据的程序具有较高的优先级。

一般而言，由硬件产生的信号需要cpu立马做出回应（不然数据可能就丢失），所以它的优先级很高。cpu理应中断掉正在执行的程序，去做出响应；当cpu完成对硬件的响应后，再重新执行用户程序。中断的过程如下图，和函数调用差不多。只不过函数调用是事先定好位置，而中断的位置由“信号”决定。



中断程序调用

以键盘为例，当用户按下键盘某个按键时，键盘会给cpu的中断引脚发出一个高电平。cpu能够捕获这个信号，然后执行键盘中断程序。下图展示了各种硬件通过中断与cpu交互。



cpu中断

现在可以回答本节提出的问题了：当网卡把数据写入到内存后，**网卡向cpu发出一个中断信号，操作系统便能得知有新数据到来**，再通过网卡**中断程序**去处理数据。

三、进程阻塞为什么不占用cpu资源?

了解epoll本质的第三步，要从操作系统进程调度的角度来看数据接收。阻塞是进程调度的关键一环，指的是进程在等待某事件（如接收到网络数据）发生之前的等待状态，recv、select和epoll都是阻塞方法。

为简单起见，我们从普通的recv接收开始分析，先看看下面代码：

```
//创建socket
int s = socket(AF_INET, SOCK_STREAM, 0);
//绑定bind(s, ...)
//监听listen(s, ...)
//接受客户端连接int c = accept(s, ...)
//接收客户端数据recv(c, ...);
//将数据打印出来printf(...)
```

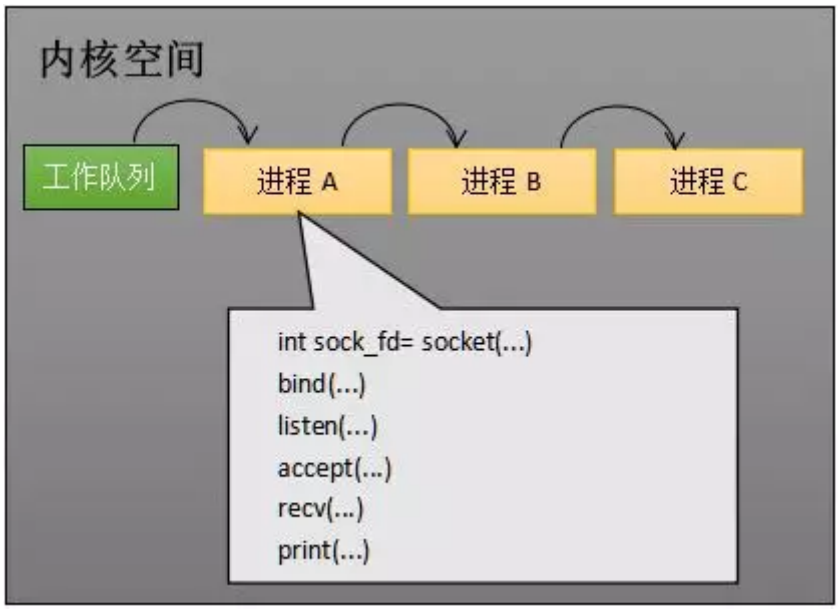
这是一段最基础的网络编程代码，先新建socket对象，依次调用bind、listen、accept，最后调用recv接收数据。recv是个阻塞方法，当程序运行到recv时，它会一直等待，直到接收到数据才往下执行。

那么阻塞的原理是什么?

工作队列

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。运行状态是进程获得cpu使用权，正在执行代码的状态；等待状态是阻塞状态，比如上述程序运行到recv时，程序会从运行状态变为等待状态，接收到数据后又变回运行状态。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。

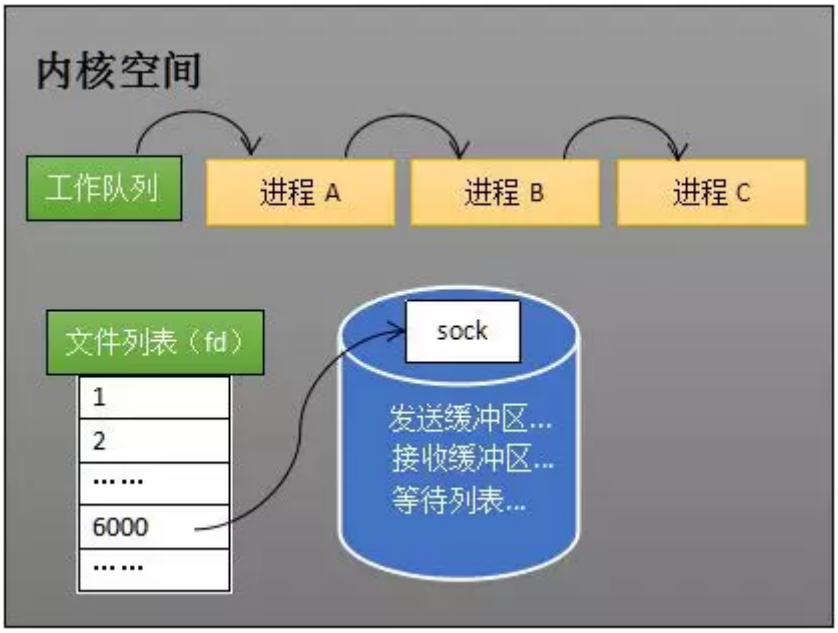
下图中的计算机中运行着A、B、C三个进程，其中进程A执行着上述基础网络程序，一开始，这3个进程都被操作系统的工作队列所引用，处于运行状态，会分时执行。



工作队列中有A、B和C三个进程

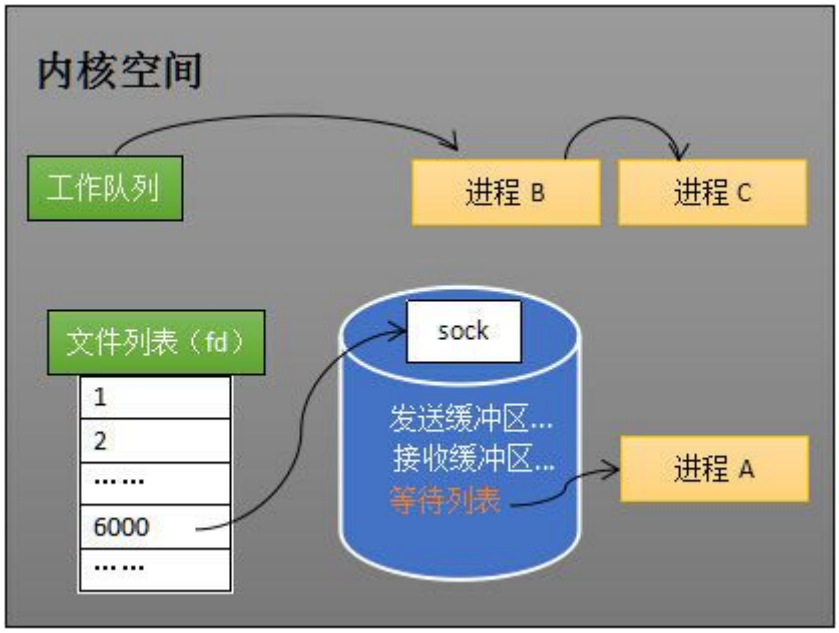
等待队列

当进程A执行到创建socket的语句时，操作系统会创建一个由文件系统管理的socket对象（如下图）。这个socket对象包含了发送缓冲区、接收缓冲区、等待队列等成员。等待队列是个非常重要的结构，它指向所有需要等待该socket事件的进程。



创建socket

当程序执行到recv时，操作系统会将进程A从工作队列移动到该socket的等待队列中（如下图）。由于工作队列只剩下了进程B和C，依据进程调度，cpu会轮流执行这两个进程的程序，不会执行进程A的程序。**所以进程A被阻塞，不会往下执行代码，也不会占用cpu资源。**



socket的等待队列

ps：操作系统添加等待队列只是添加了对这个“等待中”进程的引用，以便在接收到数据时获取进程对象、将其唤醒，而非直接将进程管理纳入自己之下。上图为了方便说明，直接将进程挂到等待队列之下。

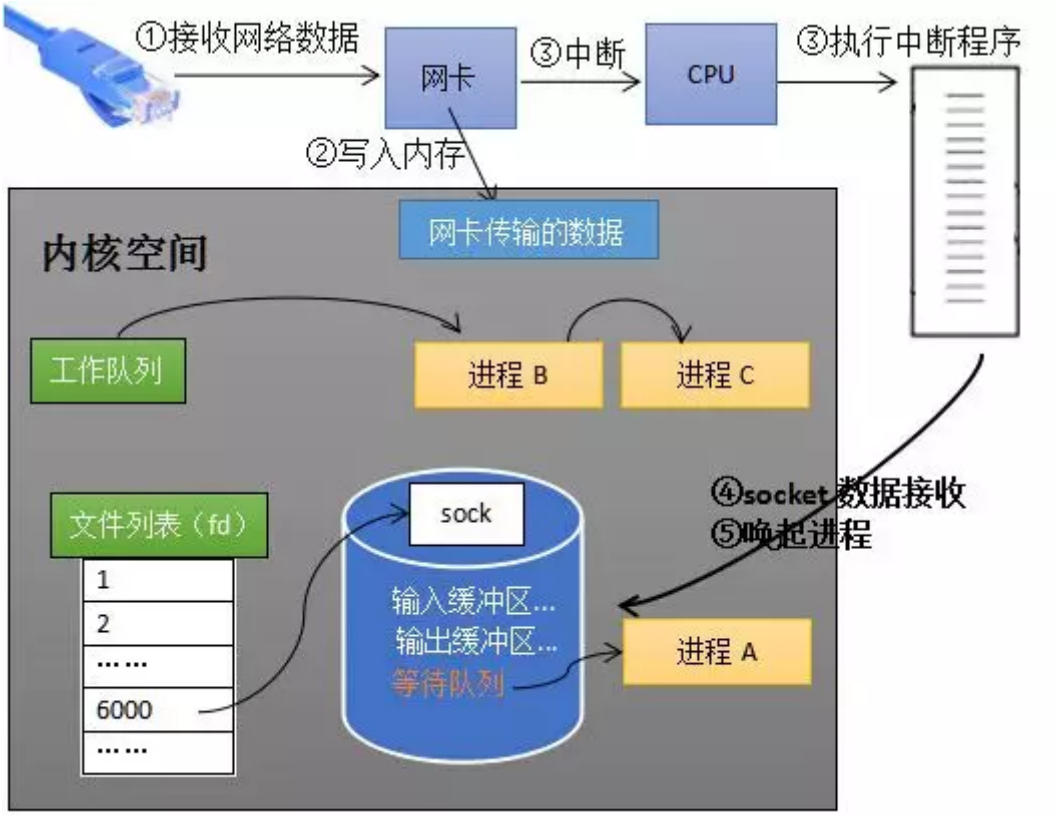
唤醒进程

当socket接收到数据后，操作系统将该socket等待队列上的进程重新放回到工作队列，该进程变成运行状态，继续执行代码。也由于socket的接收缓冲区已经有了数据，recv可以返回接收到的数据。

四、内核接收网络数据全过程

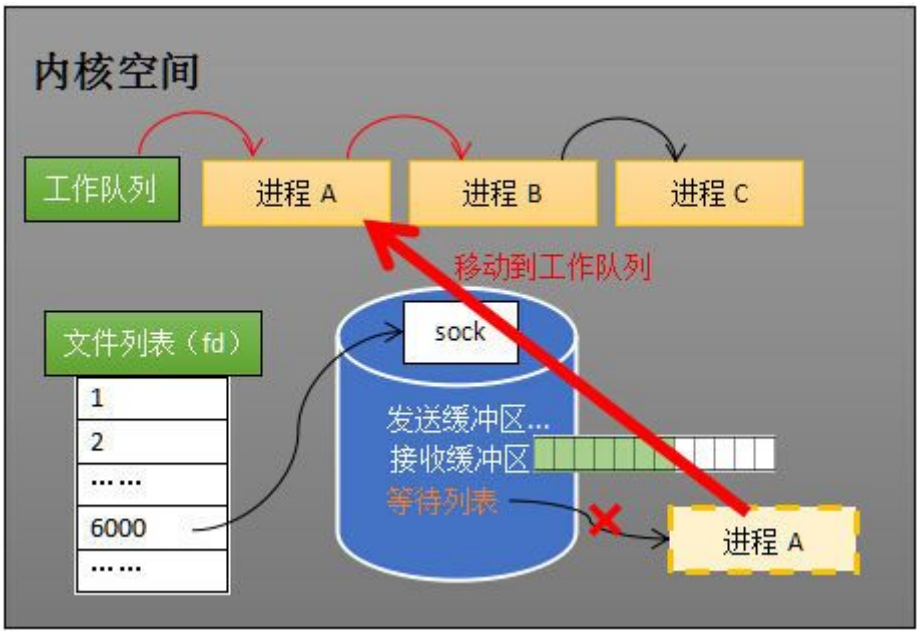
这一步，贯穿网卡、中断、进程调度的知识，叙述阻塞recv下，内核接收数据全过程。

如下图所示，进程在recv阻塞期间，计算机收到了对端传送的数据（步骤①）。数据经由网卡传送到内存（步骤②），然后网卡通过中断信号通知cpu有数据到达，cpu执行中断程序（步骤③）。此处的中断程序主要有两项功能，先将网络数据写入到对应socket的接收缓冲区里面（步骤④），再唤醒进程A（步骤⑤），重新将进程A放入工作队列中。



内核接收数据全过程

唤醒进程的过程如下图所示。



唤醒进程

以上是内核接收数据全过程

这里留有两个思考题，大家先想一想。

其一，操作系统如何知道网络数据对应于哪个socket？

其二，如何同时监视多个socket的数据？

公布答案的时刻到了。

第一个问题：因为一个socket对应着一个端口号，而网络数据包中包含了ip和端口的信息，内核可以通过端口号找到对应的socket。当然，为了提高处理速度，操作系统会维护端口号到socket的索引结构，以快速读取。

第二个问题是**多路复用的重中之重**，是本文后半部分的重点！

五、同时监视多个socket的简单方法

服务端需要管理多个客户端连接，而recv只能监视单个socket，这种矛盾下，人们开始寻找监视多个socket的方法。epoll的要义是高效的监视多个socket。从历史发展角度看，必然先出现一种不太高效的方法，人们再加以改进。只有先理解了不太高效的方法，才能够理解epoll的本质。

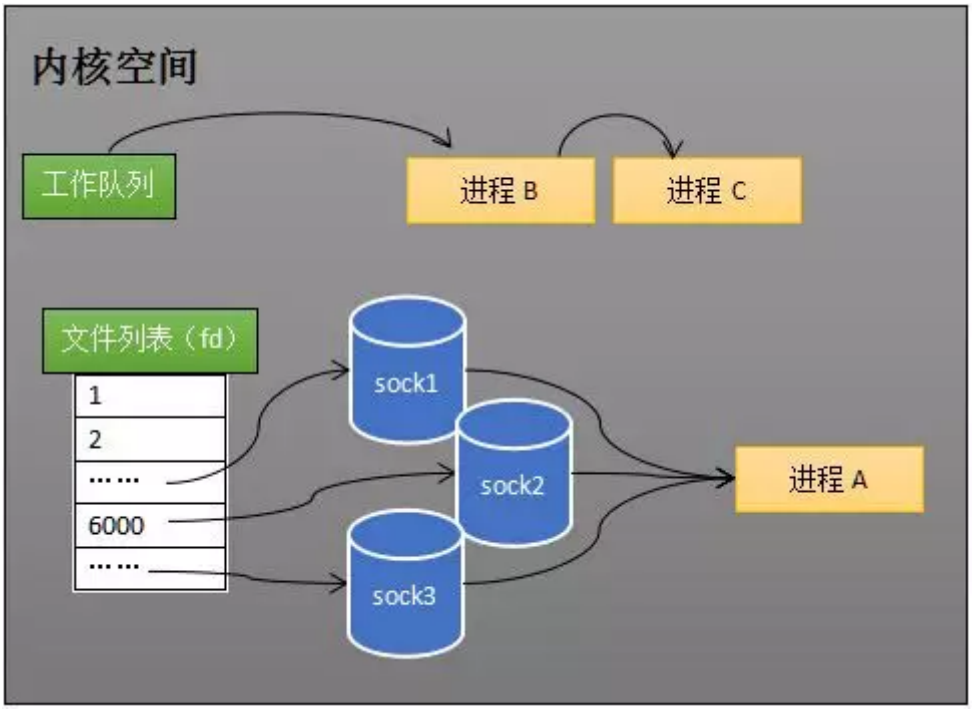
假如能够预先传入一个socket列表，**如果列表中的socket都没有数据，挂起进程，直到有一个socket收到数据，唤醒进程**。这种方法很直接，也是select的设计思想。

为方便理解，我们先复习select的用法。在如下的代码中，先准备一个数组（下面代码中的fds），让fds存放着所有需要监视的socket。然后调用select，如果fds中的所有socket都没有数据，select会阻塞，直到有一个socket接收到数据，select返回，唤醒进程。用户可以遍历fds，通过FD_ISSET判断具体哪个socket收到数据，然后做出处理。

```
int s = socket(AF_INET, SOCK_STREAM, 0);  bind(s, ...)listen(s, ...)
int fds[] =  存放需要监听的socket
while(1){
    int n = select(..., fds, ...)
    for(int i=0; i < fds.count; i++){
        if(FD_ISSET(fds[i], ...)){
            //fds[i]的数据处理
        }
    }}
```

select

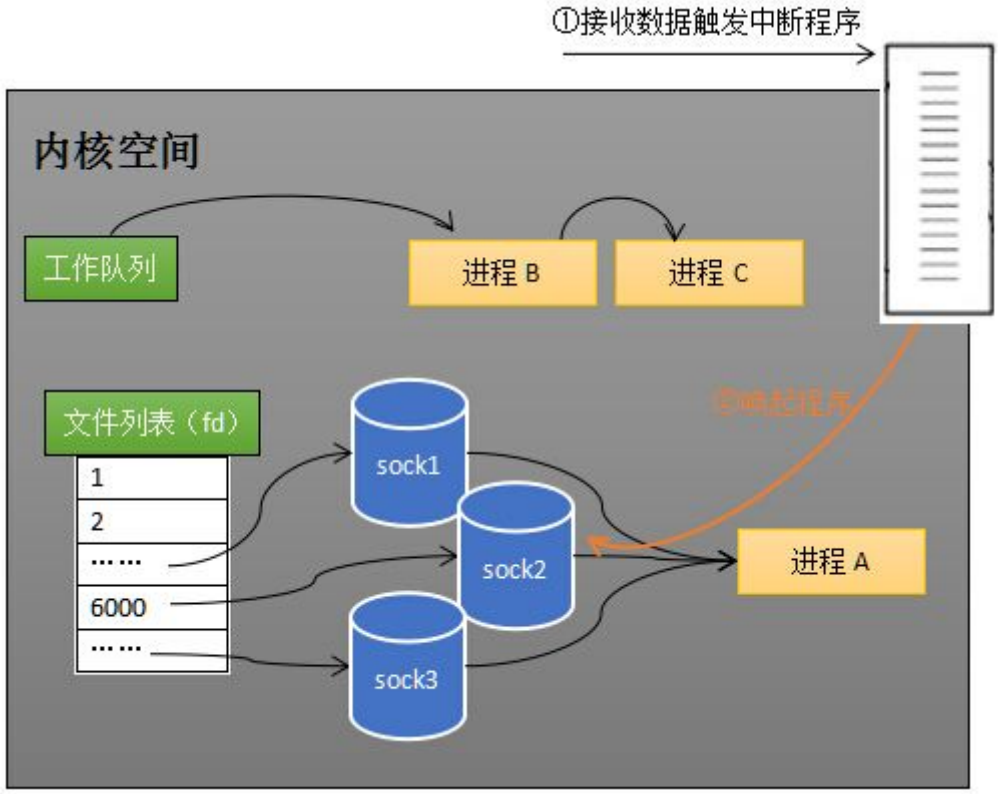
select的实现思路很直接。假如程序同时监视如下图的sock1、sock2和sock3三个socket，那么在调用select之后，操作系统把进程A分别加入这三个socket的等待队列中。



操作系统把进程A分别加入这三个socket的等待队列中

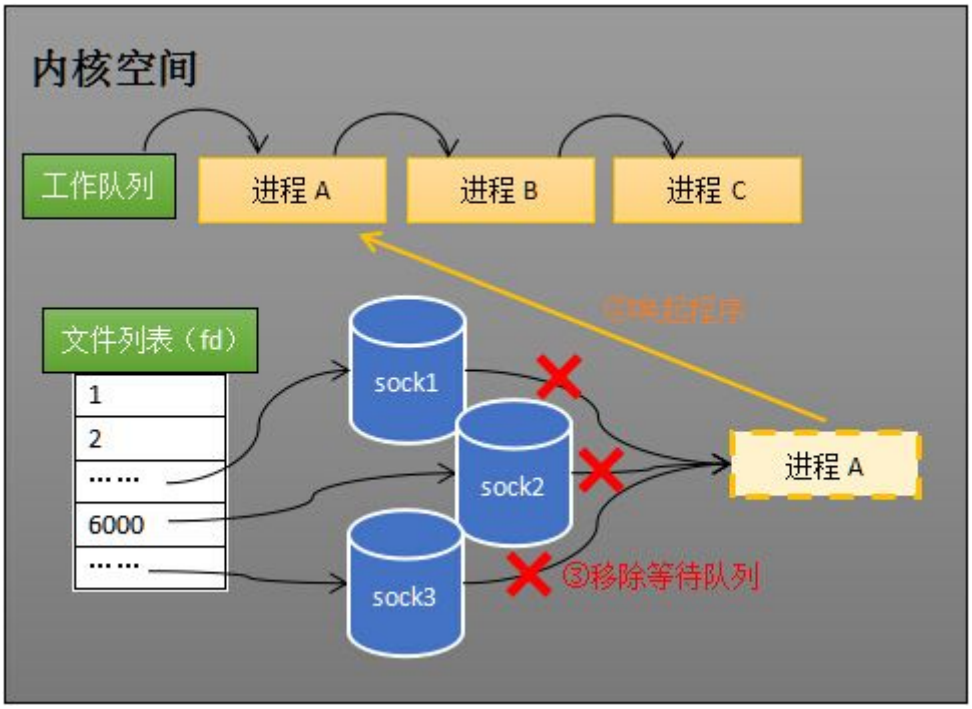
当任何一个socket收到数据后，中断程序将唤起进程。下图展示了sock2接收到了数据的处理流程。

ps：recv和select的中断回调可以设置成不同的内容。



sock2接收到了数据，中断程序唤起进程A

所谓唤起进程，就是将进程从所有的等待队列中移除，加入到工作队列里面。如下图所示。



将进程A从所有等待队列中移除，再加入到工作队列里面

经由这些步骤，当进程A被唤醒后，它知道至少有一个socket接收了数据。程序只需遍历一遍socket列表，就可以得到就绪的socket。

这种简单方式行之有效，在几乎所有操作系统都有对应的实现。

但是简单的方法往往有缺点，主要是：

其一，每次调用select都需要将进程加入到所有监视socket的等待队列，每次唤醒都需要从每个队列中移除。这里涉及了两次遍历，而且每次都要将整个fds列表传递给内核，有一定的开销。正是因为遍历操作开销大，出于效率的考量，才会规定select的最大监视数量，默认只能监视1024个socket。

其二，进程被唤醒后，程序并不知道哪些socket收到数据，还需要遍历一次。

那么，有没有减少遍历的方法？有没有保存就绪socket的方法？这两个问题便是epoll技术要解决的。

补充说明：本节只解释了select的一种情形。当程序调用select时，内核会先遍历一遍socket，如果有一个以上的socket接收缓冲区有数据，那么select直接返回，不会阻塞。这也是为什么select的返回值有可能大于1的原因之一。如果没有socket有数据，进程才会阻塞。

poll

elect()和poll()系统调用的大体一样，处理多个描述符也是使用轮询的方式，根据描述符的状态进行处理，一样需要把fd集合从用户态拷贝到内核态，并进行遍历

最大的区别是：poll没有最大文件描述符限制(使用链表的方式存储fd，因为链表可以无限的去扩展，单/双向链表都可以无限扩展)

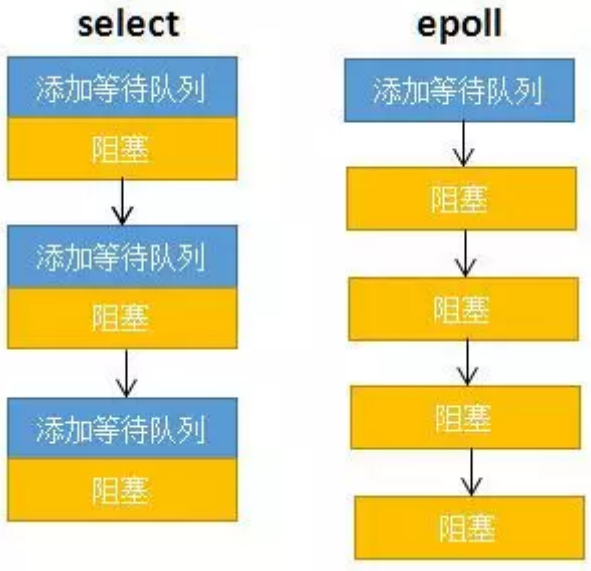
poll技术使用链表结构的方式来存储fdset的集合,相比select而言,链表不受限于FD_SIZE的个数限制,但是对于select存在的性能并没有解决,即一个是存在大内存数据拷贝的问题,一个是轮询遍历整个等待队列的每个节点并逐个通过回调函数来实现读取任务的唤醒

六、epoll的设计思路

epoll是在select出现N多年后才被发明的，是select和poll的增强版本。epoll通过以下一些措施来改进效率。

措施一：功能分离

select低效的原因之一是将“维护等待队列”和“阻塞进程”两个步骤合二为一。如下图所示，每次调用select都需要这两步操作，然而大多数应用场景中，需要监视的socket相对固定，并不需要每次都修改。epoll将这两个操作分开，先用epoll_ctl维护等待队列，再调用epoll_wait阻塞进程。显而易见的，效率就能得到提升。



相比select，epoll拆分了功能

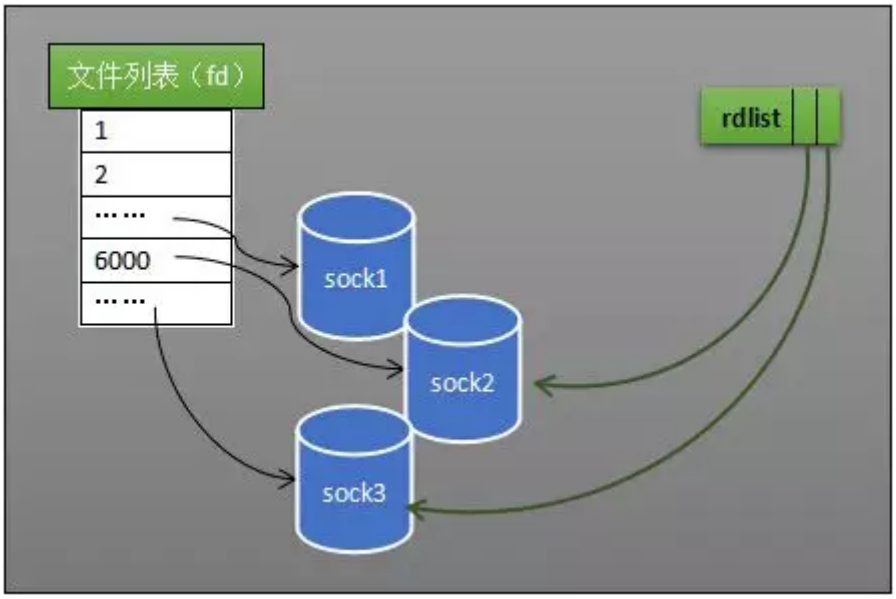
为方便理解后续的内容，我们先复习下epoll的用法。如下的代码中，先用epoll_create创建一个epoll对象epfd，再通过epoll_ctl将需要监视的socket添加到epfd中，最后调用epoll_wait等待数据。

```
int s = socket(AF_INET, SOCK_STREAM, 0);   bind(s, ...)listen(s, ...)
int epfd = epoll_create(...);epoll_ctl(epfd, ...); //将所有需要监听的socket添加到epfd中
while(1){
    int n = epoll_wait(...)
    for(接收到数据的socket){
        //处理
    }}
```

功能分离，使得epoll有了优化的可能。

措施二：就绪列表

select低效的另一个原因在于程序不知道哪些socket收到数据，只能一个个遍历。如果内核维护一个“就绪列表”，引用收到数据的socket，就能避免遍历。如下图所示，计算机共有三个socket，收到数据的sock2和sock3被rdlist（就绪列表）所引用。当进程被唤醒后，只要获取rdlist的内容，就能够知道哪些socket收到数据。



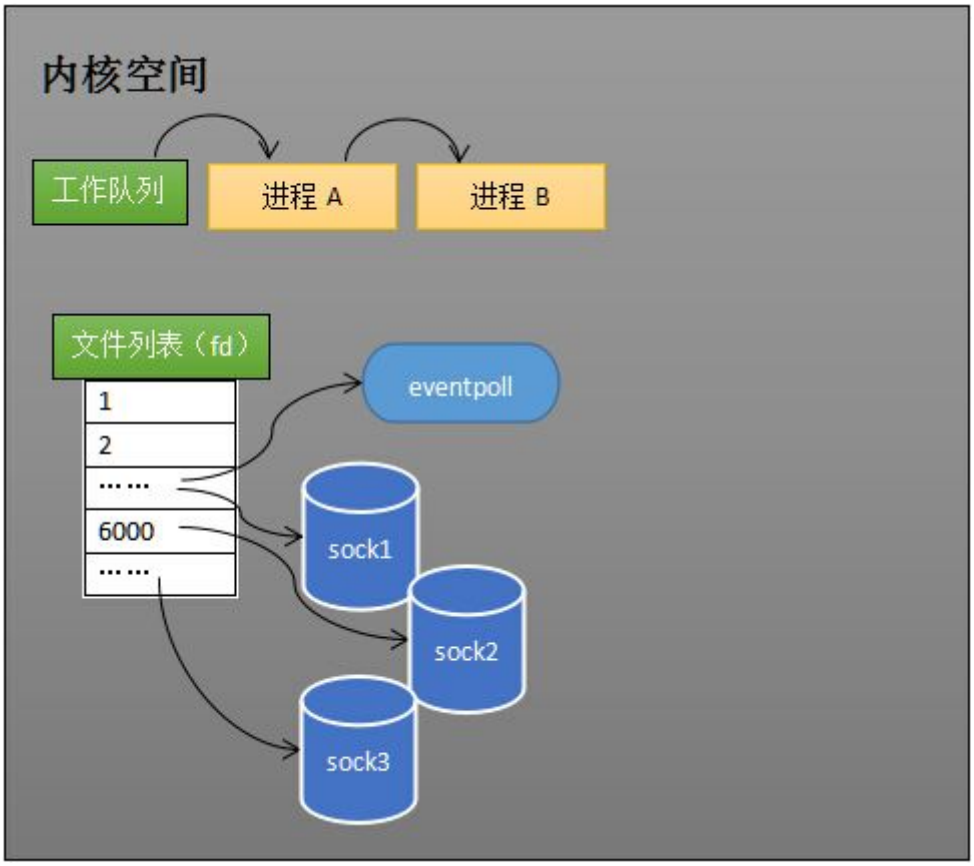
就绪列表示意图

七、epoll的原理和流程

本节会以示例和图表来讲解epoll的原理和流程。

创建epoll对象

如下所示，当某个进程调用epoll_create方法时，内核会创建一个eventpoll对象（也就是程序中epfd所代表的对象）。eventpoll对象也是文件系统中的一员，和socket一样，它也会有等待队列。

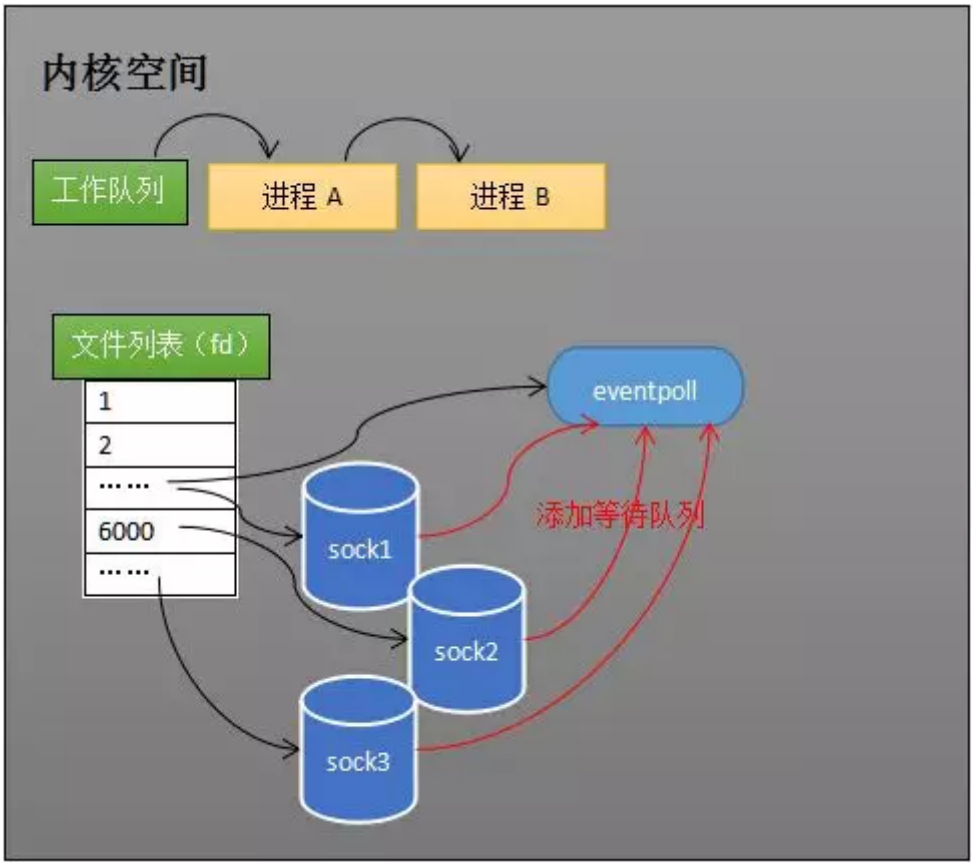


内核创建eventpoll对象

创建一个代表该epoll的eventpoll对象是必须的，因为内核要维护“就绪列表”等数据，“就绪列表”可以作为eventpoll的成员。

维护监视列表

创建epoll对象后，可以用epoll_ctl添加或删除所要监听的socket。以添加socket为例，如下图，如果通过epoll_ctl添加sock1、sock2和sock3的监视，内核会将eventpoll添加到这三个socket的等待队列中。

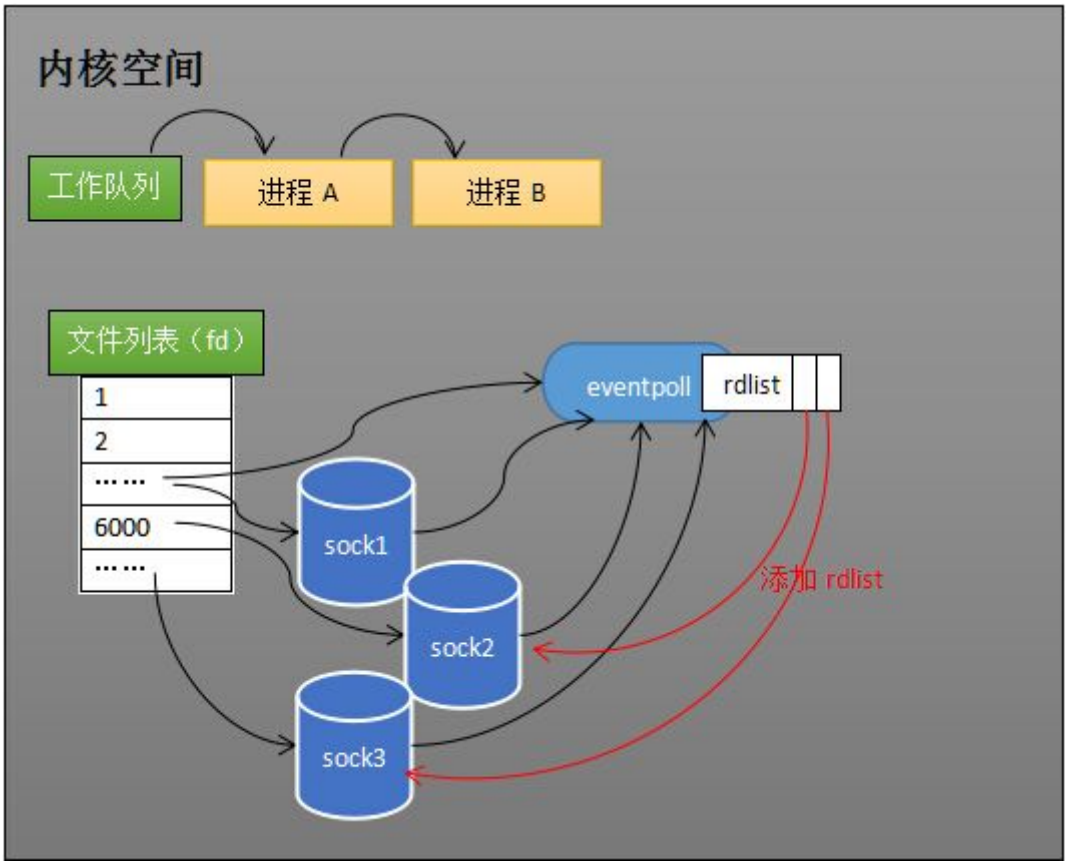


添加所要监听的socket

当socket收到数据后，中断程序会操作eventpoll对象，而不是直接操作进程。

接收数据

当socket收到数据后，中断程序会给eventpoll的“就绪列表”添加socket引用。如下图展示的是sock2和sock3收到数据后，中断程序让rdlist引用这两个socket。



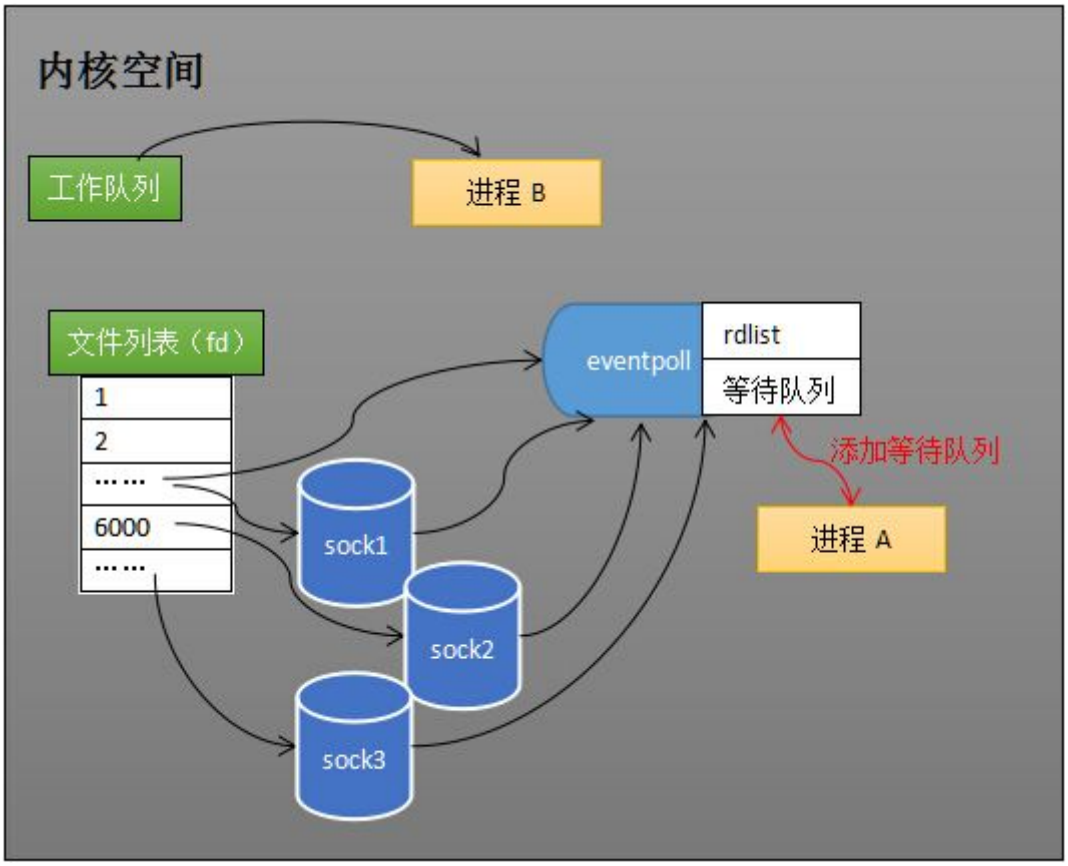
给就绪列表添加引用

eventpoll对象相当于是socket和进程之间的中介，socket的数据接收并不直接影响进程，而是通过改变eventpoll的就绪列表来改变进程状态。

当程序执行到epoll_wait时，如果rdlist已经引用了socket，那么epoll_wait直接返回，如果rdlist为空，阻塞进程。

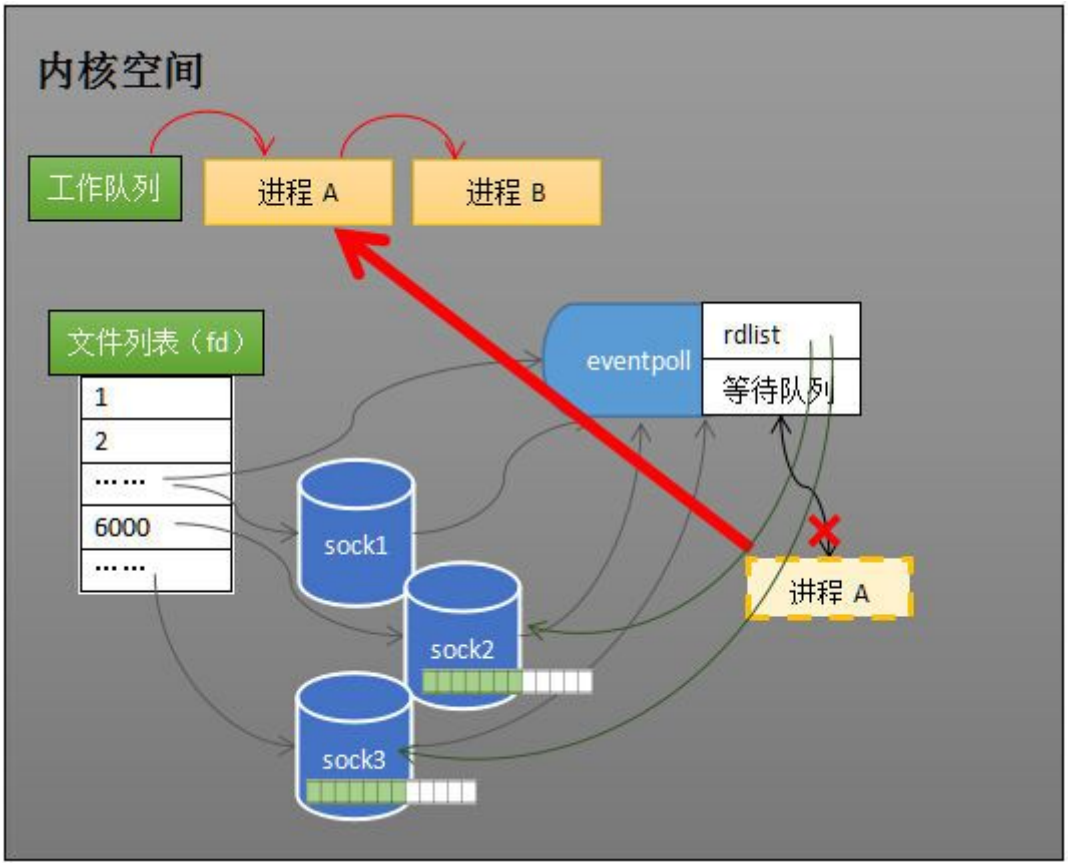
阻塞和唤醒进程

假设计算机中正在运行进程A和进程B，在某时刻进程A运行到了epoll_wait语句。如下图所示，内核会将进程A放入eventpoll的等待队列中，阻塞进程。



epoll_wait阻塞进程

当socket接收到数据，中断程序一方面修改rdlist，另一方面唤醒eventpoll等待队列中的进程，进程A再次进入运行状态（如下图）。也因为rdlist的存在，进程A可以知道哪些socket发生了变化。



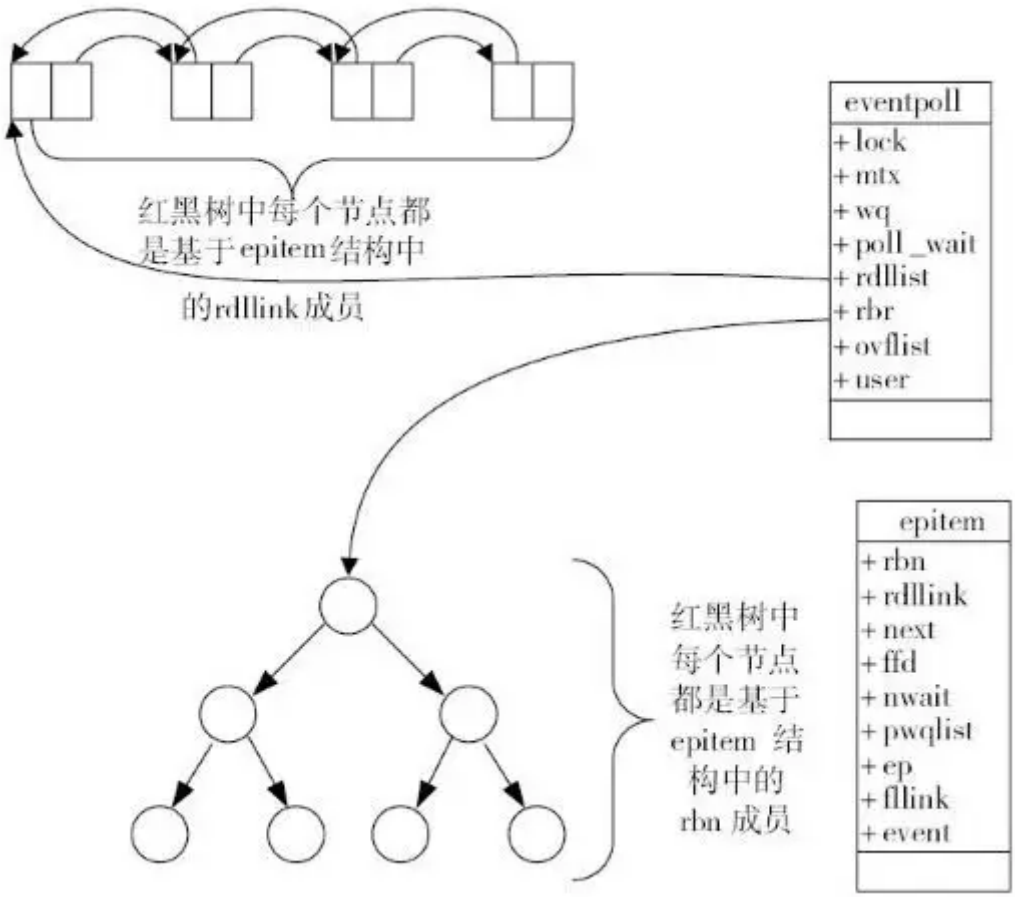
epoll唤醒进程

八、epoll的实现细节

至此，相信读者对epoll的本质已经有一定的了解。但我们还留有一个问题，eventpoll的数据结构是什么样子？

再留两个问题，就绪队列应该应使用什么数据结构？eventpoll应使用什么数据结构来管理通过epoll_ctl添加或删除的socket？

如下图所示，eventpoll包含了lock、mtx、wq（等待队列）、rdlist等成员。rdlist和rbr是我们所关心的。



epoll原理示意图，图片来源：《深入理解Nginx：模块开发与架构解析(第二版)》，陶辉

就绪列表的数据结构

就绪列表引用着就绪的socket，所以它应能够快速的插入数据。

程序可能随时调用epoll_ctl添加监视socket，也可能随时删除。当删除时，若该socket已经存放在就绪列表中，它也应该被移除。

所以就绪列表应是一种能够快速插入和删除的数据结构。双向链表就是这样一种数据结构，epoll使用双向链表来实现就绪队列（对应上图的rdllist）。

索引结构

既然epoll将“维护监视队列”和“进程阻塞”分离，也意味着需要有个数据结构来保存监视的socket。至少要方便的添加和移除，还要便于搜索，以避免重复添加。红黑树是一种自平衡二叉查找树，搜索、插入和删除时间复杂度都是O(log(N))，效率较好。epoll使用了红黑树作为索引结构（对应上图的rbr）。

ps：因为操作系统要兼顾多种功能，以及由更多需要保存的数据，rdlist并非直接引用socket，而是通过epitem间接引用，红黑树的节点也是epitem对象。同样，文件系统也并非直接引用着socket。为方便理解，本文中省略了一些间接结构。

九、结论

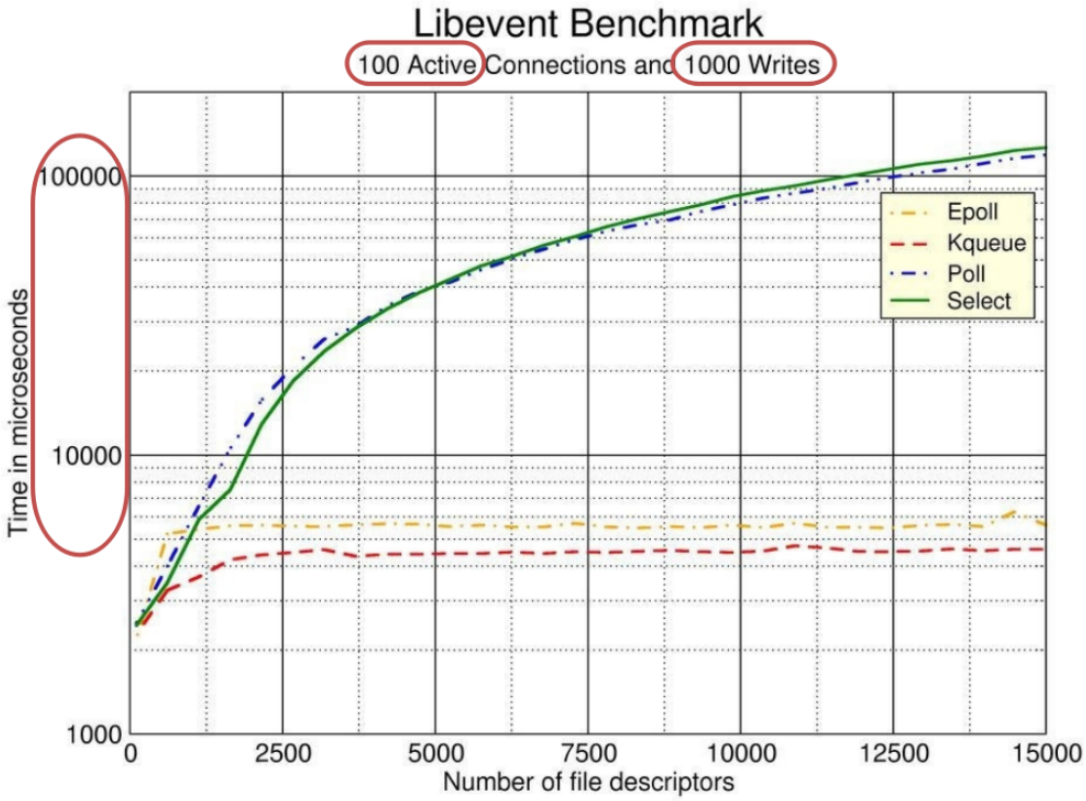
epoll在select和poll（poll和select基本一样，有少量改进）的基础引入了eventpoll作为中间层，使用了先进的数据结构，是一种高效的多路复用技术。

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)

select、poll和epoll的区别。图片来源《Linux高性能服务器编程》

	select	poll	epoll
操作方式	遍历	遍历	回调
底层实现	数组	链表	哈希表
IO效率	每次调用都进行线性遍历，时间复杂度为O(n)	每次调用都进行线性遍历，时间复杂度为O(n)	事件通知方式，每当fd就绪，系统注册的回调函数就会被调用，将就绪fd放到readyList里面，时间复杂度O(1)
最大连接数	1024（x86）或2048（x64）	无上限	无上限
fd拷贝	每次调用select，都需要把fd集合从用户态拷贝到内核态	每次调用poll，都需要把fd集合从用户态拷贝到内核态	调用epoll_ctl时拷贝进内核并保存，之后每次epoll_wait不拷贝

下图是 **libevent** (一个知名的异步事件处理软件库)对 select，poll，epoll ，kqueue 这几个 I/O 多路复用技术做的性能测试。



这是一个限制了 100 个活跃连接的基准测试，每个连接发生 1000 次读写操作为止。纵轴是请求的响应时间，横轴是持有的 socket 句柄数量。随着句柄数量的增加，epoll 和 kqueue 响应时间几乎无变化，而 poll 和 select 的响应时间却增长了非常多。
可以看出来，epoll 性能是很高的，并且随着监听的文件描述符的增加，epoll 的优势更加明显。
不过，这里限制的 100 个连接很重要。**epoll 在应对大量网络连接时，只有活跃连接很少的情况下才能表现的性能优异。换句话说，epoll 在处理大量非活跃的连接时性能才会表现的优异。如果 15000 个 socket 都是活跃的，epoll 和 select 其实差不了太多。**

四种网络模型中，IO多路复用模型 select / poll /epoll 的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。