Linux 对网络通信的实现

Linux 网络 IO 模型

同步和异步，阻塞和非阻塞

同步和异步

关注的是调用方是否主动获取结果

同步:同步的意思就是调用方需要主动等待结果的返回

异步:异步的意思就是不需要主动等待结果的返回，而是通过其他手段比如，状态通知，回调函数等。

阻塞和非阻塞

主要关注的是等待结果返回调用方的状态

阻塞:是指结果返回之前，当前线程被挂起，不做任何事

非阻塞:是指结果在返回之前，线程可以做一些其他事，不会被挂起。

两者的组合

1.同步阻塞:同步阻塞基本也是编程中最常见的模型，打个比方你去商店买衣服，你去了之后发现衣服卖完了，那你就在店里面一直等，期间不做任何事(包括看手机)，等着商家进货，直到有货为止，这个效率很低。

2.同步非阻塞:同步非阻塞在编程中可以抽象为一个轮询模式，你去了商店之后，发现衣服卖完了，这个时候不需要傻傻的等着，你可以去其他地方比如奶茶店，买杯水，但是你还是需要时不时的去商店问老板新衣服到了吗。

3.异步阻塞:异步阻塞这个编程里面用的较少，有点类似你写了个线程池,submit 然后马上 future.get()，这样线程其实还是挂起的。有点像你去商店买衣服，这个时候发现衣服没有了，这个时候你就给老板留给电话，说衣服到了就给我打电话，然后你就守着这个电话，一直等着他响什么事也不做。这样感觉的确有点傻，所以这个模式用得比较少。

4.异步非阻塞:异步非阻塞。好比你去商店买衣服，衣服没了，你只需要给老板说这是我的电话，衣服到了就打。然后你就随心所欲的去玩，也不用操心衣服什么时候到，衣服一到，电话一响就可以去买衣服了。

Linux 下的五种 I/O 模型

1)阻塞I/O（blocking I/O）

2)非阻塞I/O （nonblocking I/O）

3) I/O复用(select 、 poll和epoll) （I/O multiplexing）

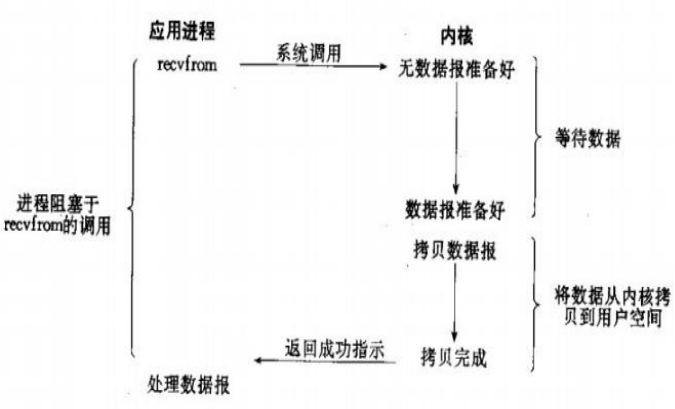
4)信号驱动I/O （signal driven I/O (SIGIO)）

5)异步I/O （asynchronous I/O ）

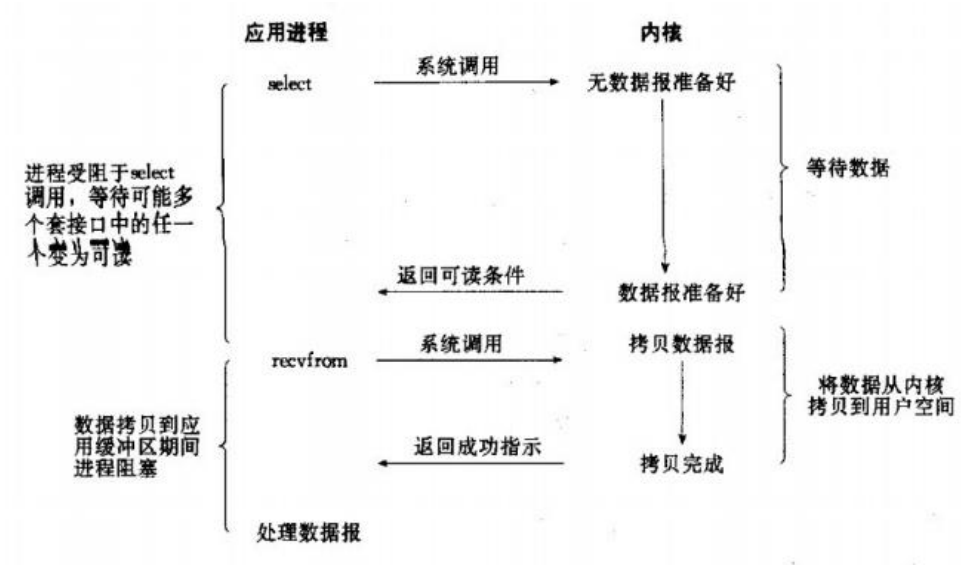
同步

异步

总的来说，阻塞 IO 就是 JDK 里的 BIO 编程，IO 复用就是 JDK 里的 NIO 编程，Linux 下异步 IO 的实现建立在 epoll 之上，是个伪异步实现，而且相比 IO 复用，没有体现出性能优势，使用不广。非阻塞 IO 使用轮询模式，会不断检测是否有数据到达，大量的占用 CPU 的时间，是绝不被推荐的模型。信号驱动 IO 需要在网络通信时额外安装信号处理函数，使用也不广泛。



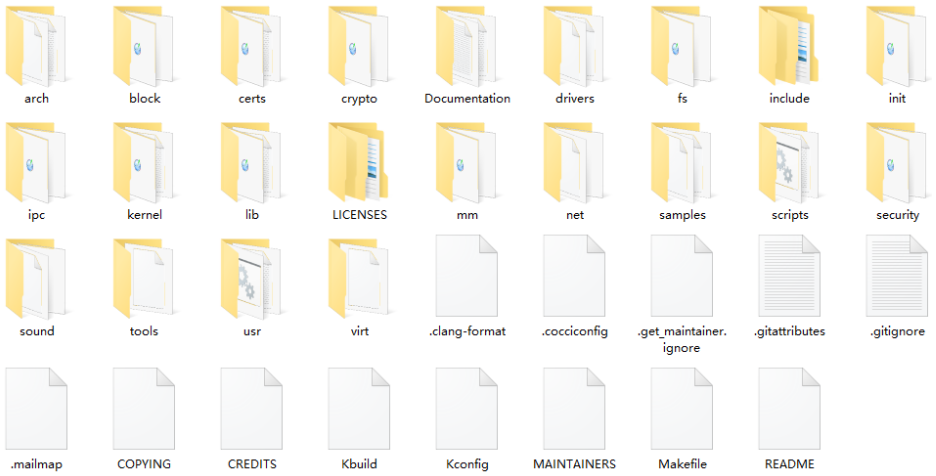
阻塞 IO 模型



I/O 复用模型

比较上面两张图，IO 复用需要使用两个系统调用(select 和 recvfrom)，而 blocking IO 只调用了一个系统调用(recvfrom)。但是，用 select 的优势在于它可以同时处理多个 connection。所以，如果处理的连接数不是很高的话，使用 select/epoll 的 web server 不一定比使用 multi-threading + blocking IO 的 web server 性能更好，可能延迟还更大。select/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。

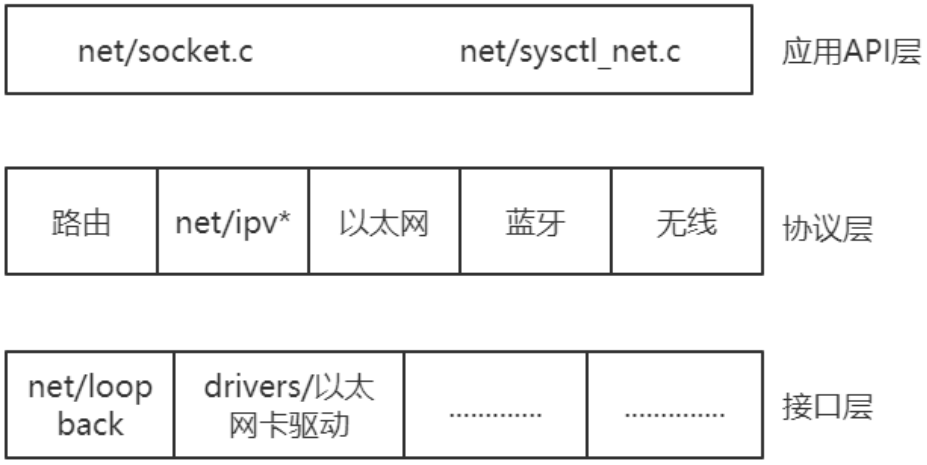
从 Linux 代码结构看网络通信



Linux 内核的源码包含的东西很多，在 Linux 的源代码中，网络设备驱动对应的逻辑位于 driver/net/ethernet，其中 intel 系列网卡的驱动在 driver/net/ethernet/intel 目录下。协议栈模块代码位于 kernel 和 net 目录。

其中 net 目录中包含 Linux 内核的网络协议栈的代码。子目录 ipv4 和 ipv6 为 TCP/IP 协议栈的 IPv4 和 IPv6 的实现，主要包含了 TCP、UDP、IP 协议的代码，还有 ARP 协议、ICMP 协议、IGMP 协议代码实现，以及如 proc、ioctl 等控制相关的代码。

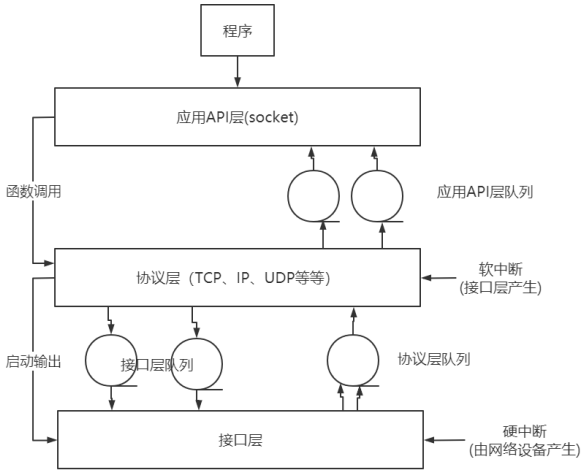
站在网络通信的角度，源代码组织的表现形式如下：



网络协议栈是由若干个层组成的，网络数据的流程主要是指在协议栈的各个层之间的传递。一个 TCP 服务器的流程按照建立 socket()函数，绑定地址端口 bind()函数，侦听端口 listen()函数，接收连接 accept()函数，发送数据 send()函数，接收数据 recv()函数，关闭 socket()函数的顺序来进行。

与此对应内核的处理过程也是按照此顺序进行的，网络数据在内核中的处理过程主要是在网卡和协议栈之间进行:从网卡接收数据，交给协议栈处理;协议栈将需要发送的数据通过网络发出去。

由下图中可以看出，数据的流向主要有两种。应用层输出数据时，数据按照自上而下的顺序，依次通过应用 API 层、协议层和接口层;当有数据到达的时候，自下而上依次通过接口层、协议层和应用 API 层的方式，在内核层传递。



应用层 Socket 的初始化、绑定(bind)和销毁是通过调用内核层的 socket()函数进行资源的申请和销毁的。

发送数据的时候，将数据由应用 API 层传递给协议层，协议层在 UDP 层添加 UDP 的首部、TCP 层添加 TCP 的首部、IP 层添加 IP 的首部，接口层的网卡则添加以太网相关的信息后，通过网卡的发送程序发送到网络上。

接收数据的过程是一个相反的过程，当有数据到来的时候，网卡的中断处理程序将数据从以太网网卡的 FIFO 对列中接收到内核,传递给协议层,协议层在 IP 层剥离 IP 的首部、UDP 层剥离 UDP 的首部、TCP 层剥离 TCP 的首部后传递给应用 API 层，应用 API 层查询 socket 的标识后，将数据送给用户层匹配的 socket。

在 Linux 内核实现中，链路层协议靠网卡驱动来实现，内核协议栈来实现网络层和传输层。内核对更上层的应用层提供 socket 接口来供用户进程访问。

Linux 下的 IO 复用编程

select，poll，epoll 都是 IO 多路复用的机制。I/O 多路复用就是通过一种机制，一个进程可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但 select，poll，epoll 本质上都是同步 I/O，因为他们都需要在读写事件就绪后自己负责进行读写，并等待读写完成。

文件描述符 FD

在 Linux 操作系统中，可以将一切都看作是文件，包括普通文件，目录文件，字符设备文件（如键盘，鼠标…），块设备文件（如硬盘，光驱…），套接字等等，所有一切均抽象成文件，提供了统一的接口，方便应用程序调用。

既然在 Linux 操作系统中，你将一切都抽象为了文件，那么对于一个打开的文件，我应用程序怎么对应上呢？文件描述符应运而生。

文件描述符：File descriptor,简称 fd，当应用程序请求内核打开/新建一个文件时，内核会返回一个文件描述符用于对应这个打开/新建的文件，其 fd 本质上就是一个非负整数。实际上，它是一个索引值，指向内核为每一个进程所维护的该进程打开文件的记录表。当程序打开一个现有文件或者创建一个新文件时，内核向进程返回一个文件描述符。在程序设计中，一些涉及底层的程序编写往往会围绕着文件描述符展开。但是文件描述符这一概念往往只适用于 UNIX、Linux 这样的操作系统。

系统为了维护文件描述符建立了 3 个表：进程级的文件描述符表、系统级的文件描述符表、文件系统的 i-node 表。所谓进程级的文件描述符表，指操作系统为每一个进程维护了一个文件描述符表，该表的索引值都从从 0 开始的，所以在不同的进程中可以看到相同的文件描述符，这种情况下相同的文件描述符可能指向同一个实际文件，也可能指向不同的实际文件。

select

```
int select (int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

select 函数监视的文件描述符分 3 类，分别是 writefds、readfds、和 exceptfds。调用后 select 函数会阻塞，直到有描述副就绪（有数据 可读、可写、或者有 except），或者超时（timeout 指定等待时间，如果立即返回设为 null 即可），函数返回。当 select 函数返回后，可以 通过遍历 fdset，来找到就绪的描述符。

select 目前几乎在所有的平台上支持，其良好跨平台支持也是它的一个优点。select 的一 个缺点在于单个进程能够监视的文件描述符的数量存在最大限制，在 Linux 上一般为 1024，可以通过修改宏定义甚至重新编译内核的方式提升这一限制，但是这样也会造成效率的降低。

poll

```
int poll (struct pollfd *fds, unsigned int nfds, int timeout);
```

不同与 select 使用三个位图来表示三个 fdset 的方式，poll 使用一个 pollfd 的指针实现。

pollfd 结构包含了要监视的 event 和发生的 event，不再使用 select “参数-值” 传递的方式。同时，pollfd 并没有最大数量限制（但是数量过大后性能也是会下降）。 和 select 函数一样，poll 返回后，需要轮询 pollfd 来获取就绪的描述符。

epoll

epoll 是在 2.6 内核中提出的，是之前的 select 和 poll 的增强版本。相对于 select 和 poll 来说，可以看到 epoll 做了更细致的分解，包含了三个方法，使用上更加灵活。

```
int epoll_create(int size)；

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)；

int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
```

int epoll_create(int size);

创建一个 epoll 的句柄，size 用来告诉内核这个监听的数目一共有多大，这个参数不同于 select()中的第一个参数，给出最大监听的 fd+1 的值，参数 size 并不是限制了 epoll 所能监听的描述符最大个数，只是对内核初始分配内部数据结构的一个建议。当创建好 epoll 句柄后，它就会占用一个 fd 值，在 linux 下如果查看/proc/进程 id/fd/，是能够看到这个 fd 的，所以在使用完 epoll 后，必须调用 close()关闭，否则可能导致 fd 被耗尽。

作为类比，可以理解为对应于 JDK NIO 编程里的 selector = Selector.open();

*int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event) ;*

函数是对指定描述符 fd 执行 op 操作。

epfd：是 epoll_create()的返回值。

op：表示 op 操作，用三个宏来表示：添加 EPOLL_CTL_ADD，删除 EPOLL_CTL_DEL，修改 EPOLL_CTL_MOD。分别添加、删除和修改对 fd 的监听事件。

fd：是需要监听的 fd（文件描述符）

epoll_event：是告诉内核需要监听什么事，有具体的宏可以使用，比如 EPOLLIN ：表示对应的文件描述符可以读（包括对端 SOCKET 正常关闭）；EPOLLOUT：表示对应的文件描述符可以写；

作为类比，可以理解为对应于 JDK NIO 编程里的 socketChannel.register();

*int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);*

等待 epfd 上的 io 事件，最多返回 maxevents 个事件。

参数 events 用来从内核得到事件的集合，maxevents 告之内核这个 events 有多大，这个 maxevents 的值不能大于创建 epoll_create()时的 size，参数 timeout 是超时时间（毫秒，0 会立即返回，-1 将不确定，也有说法说是永久阻塞）。该函数返回需要处理的事件数目，如返回 0 表示已超时。

作为类比，可以理解为对应于 JDK NIO 编程里的 selector.select();

select、poll、epoll 的比较

select，poll，epoll 都是 操作系统实现 IO 多路复用的机制。 我们知道，I/O 多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。那么这三种机制有什么区别呢。

1、支持一个进程所能打开的最大连接数

select	单个进程所能打开的最大连接数有 FD_SETSIZE 宏定义，其大小是 32 个整数的大小（在 32 位的机器上，大小就是 32*32，同理 64 位机器上 FD_SETSIZE 为 32*64），当然我们可以对进行修改，然后重新编译内核，但是性能可能会受到影响。
poll	poll 本质上和 select 没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的

epoll	虽然连接数基本上只受限于机器的内存大小

2、FD 剧增后带来的 IO 效率问题

select	因为每次调用时都会对连接进行线性遍历，所以随着 FD 的增加会造成遍历速度慢的“线性下降性能问题”。
poll	同上
epoll	因为 epoll 内核中实现是根据每个 fd 上的 callback 函数来实现的，只有活跃的 socket 才会主动调用 callback，所以在活跃 socket 较少的情况下，使用 epoll 没有前面两者的线性下降的性能问题，但是所有 socket 都很活跃的情况下，可能会有性能问题。

3、 消息传递方式

select	内核需要将消息传递到用户空间，都需要内核拷贝动作
poll	同上
epoll	epoll 通过内核和用户空间共享一块内存来实现的。

总结：

综上，在选择 select，poll，epoll 时要根据具体的使用场合以及这三种方式的自身特点。

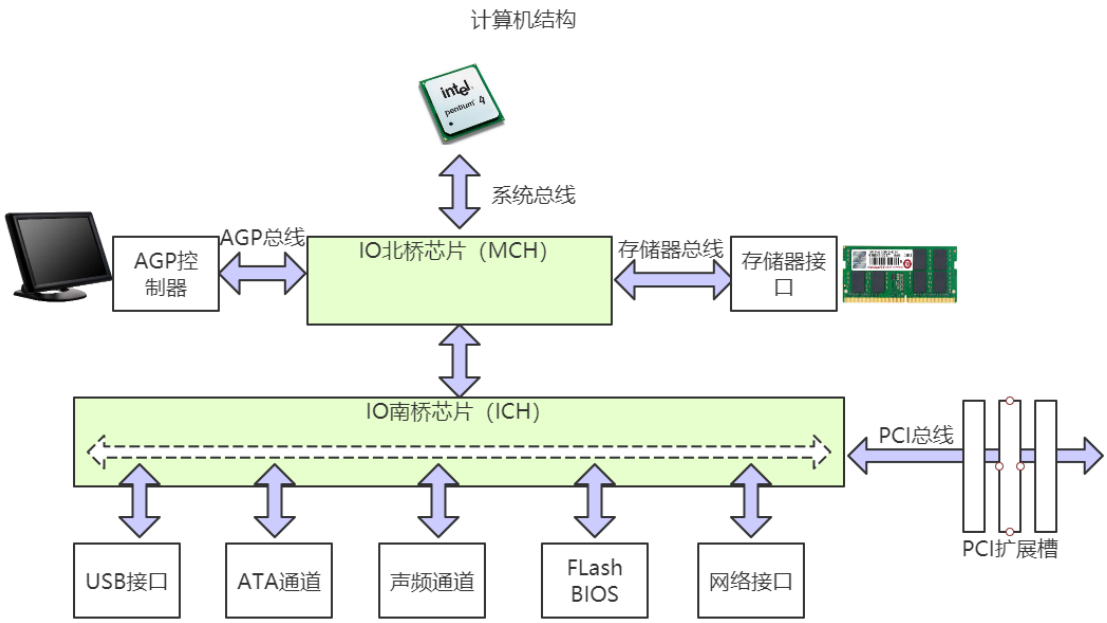
1、表面上看 epoll 的性能最好，但是在连接数少并且连接都十分活跃的情况下，select 和 poll 的性能可能比 epoll 好，毕竟 epoll 的通知机制需要很多函数回调。

2、select 低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善。

epoll 高效原理和底层机制分析

从网卡接收数据说起

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。



网卡收到网线传来的数据；经过硬件电路的传输；最终将数据写入到内存中的某个地址上。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。操作系统就可以去读取它们。

如何知道接收了数据?

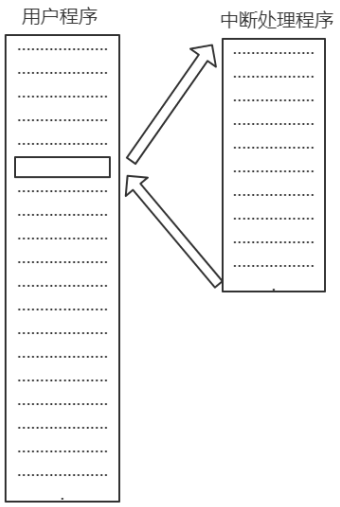
CPU 和操作系统如何知道网络上有数据要接收？很简单，使用中断机制。

中断、上半部、下半部

内核和设备驱动是通过中断的方式来处理的。所谓中断，可以理解为当设备上有数据到达的时候，会给 CPU 的相关引脚上触发一个电压变化，以通知 CPU 来处理数据。

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。



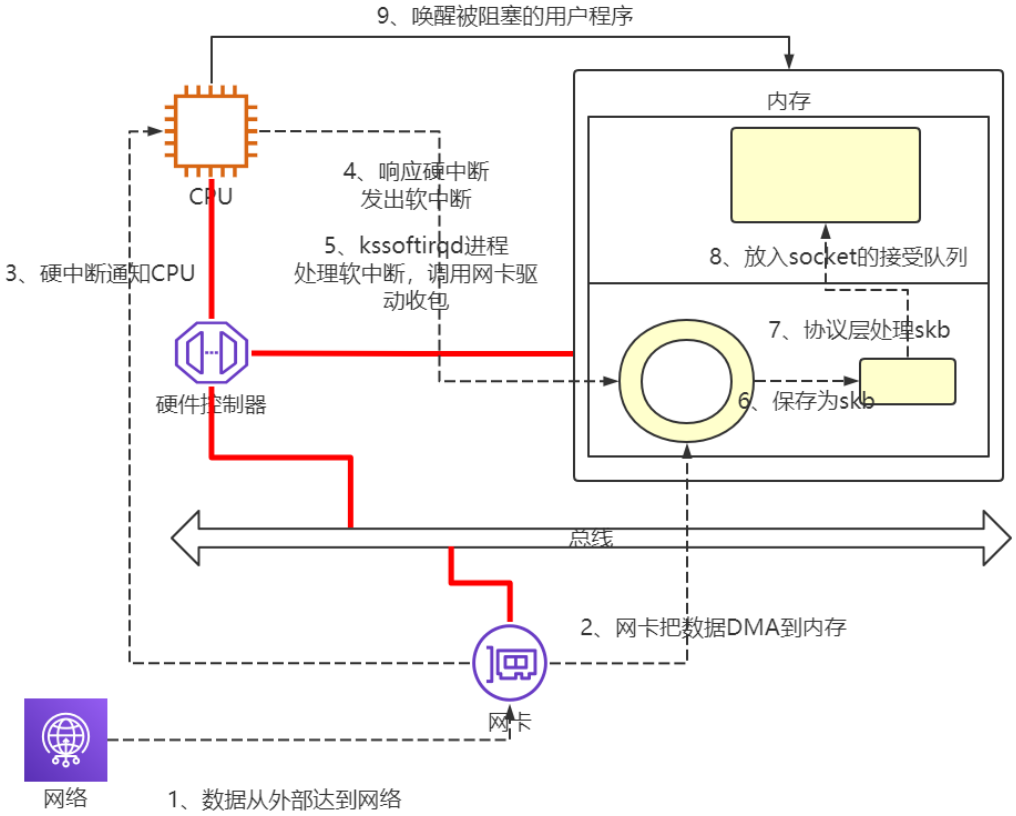
以键盘为例，当用户按下键盘某个按键时，键盘会给 cpu 的中断引脚发出一个高电平。cpu 能够捕获这个信号，然后执行键盘中断程序。

同样，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。

对于网络模块来说，由于处理过程比较复杂和耗时，如果在中断函数中完成所有的处理，将会导致中断处理函数（优先级过高）将过度占据 CPU，将导致 CPU 无法响应其它设备，例如鼠标和键盘的消息。

因此 Linux 中断处理函数是分上半部和下半部的。上半部是只进行最简单的工作，快速处理然后释放 CPU，接着 CPU 就可以允许其它中断进来。剩下将绝大部分的工作都放到下半部中，可以慢慢从容处理。2.4 以后的内核版本采用的下半部实现方式是软中断，由 ksoftirqd 内核线程全权处理。和硬中断不同的是，硬中断是通过给 CPU 物理引脚施加电压变化，而软中断是通过给内存中的一个变量的二进制值以通知软中断处理程序。

内核收包的概览



当网卡上收到数据以后，Linux 中第一个工作的模块是网络驱动。 网络驱动会以 DMA 的方式把网卡上收到的帧写到内存里。再向 CPU 发起一个中断，以通知 CPU 有数据到达。第二，当 CPU 收到中断请求后，会去调用网络驱动注册的中断处理函数。 网卡的中断处理函数并不做过多工作，发出软中断请求，然后尽快释放 CPU。ksoftirqd 检测到有软中断请求到达，调用 poll 开始轮询收包，收到后交由各级协议栈处理。最后会被放到用户 socket 的接收队列中。

进程阻塞

了解 epoll 本质，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。了解“进程阻塞为什么不占用 cpu 资源？”，也就能够了解这一步。

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
```

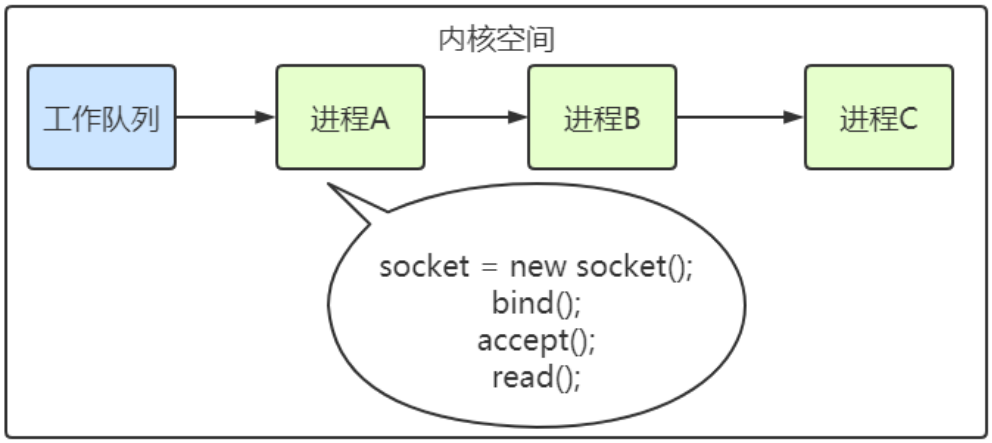
```
int c = accept(s, ...)
//接收客户端数据
recv(c, ...);
//将数据打印出来
printf(...)
```

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？

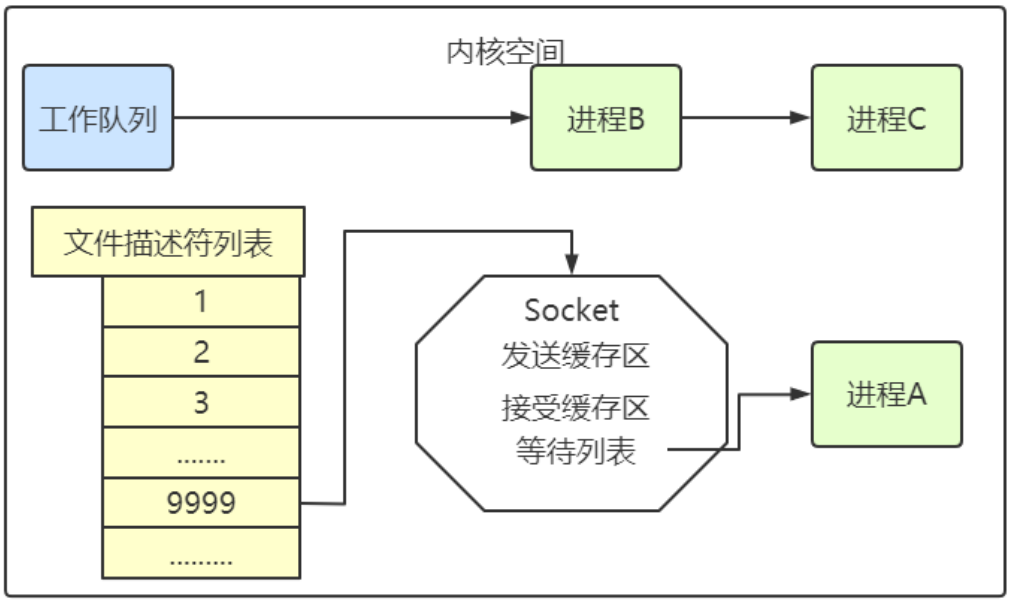
操作系统为了支持多任务，实现了进程调度的功能，会把进程分为“运行”和“等待”等几种状态。运行状态是进程获得 cpu 使用权，正在执行代码的状态；等待状态是阻塞状态，比如上述程序运行到 recv 时，程序会从运行状态变为等待状态，接收到数据后又变回运行状态。操作系统会分时执行各个运行状态的进程，由于速度很快，看上去就像是同时执行多个任务。

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。



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。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。

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。

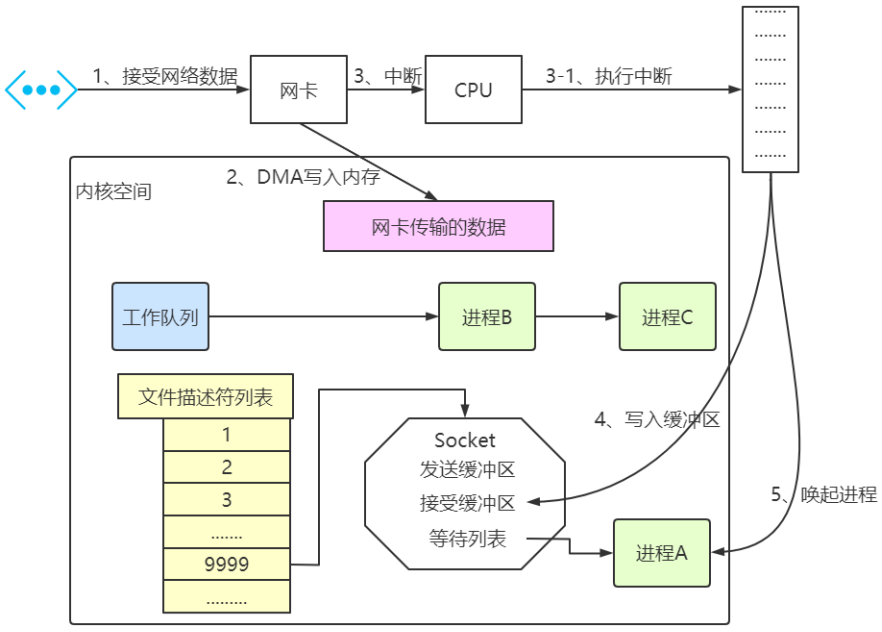


操作系统添加等待队列只是添加了对这个“等待中”进程的引用，以便在接收到数据时获取进程对象、将其唤醒，而非直接将进程管理纳入自己之下。上图为了方便说明，直接将进程挂到等待队列之下。

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。

内核接收网络数据

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。



思考下，操作系统如何知道网络数据对应于哪个 socket？

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。

思考下，如何同时监视多个 socket 的数据？

同时监视多个 socket 的简单方法

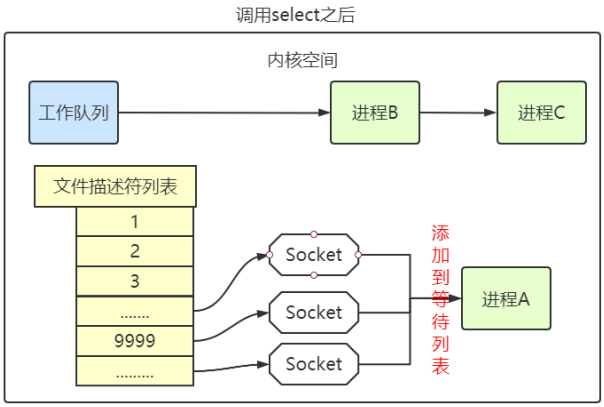
服务端需要管理多个客户端连接，而 recv 只能监视单个 socket，这种矛盾下，人们开始寻找监视多个 socket 的方法。epoll 的要义是高效的监视多个 socket。从历史发展角度看，必然先出现一种不太高效的方法，人们再加以改进。只有先理解了不太高效的方法，才能够理解 epoll 的本质。

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，我们先看看 Linux 中 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 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 的实现思路很直接。假如程序同时监视 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。



当任何一个 socket 收到数据后，中断程序将唤起进程。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。

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被监视 socket 的等待队列，每次唤醒都需要从每个队列中移除，都必须要进行遍历。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。

当然，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。

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。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，我们再来看看 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 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。

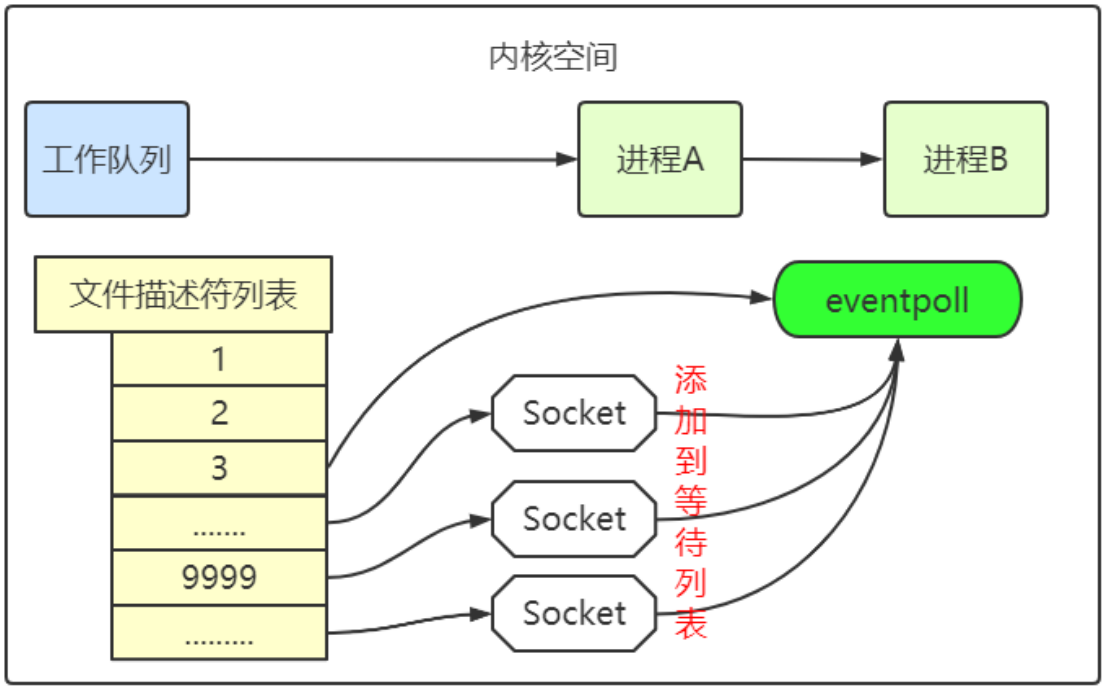
措施二：就绪列表

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。

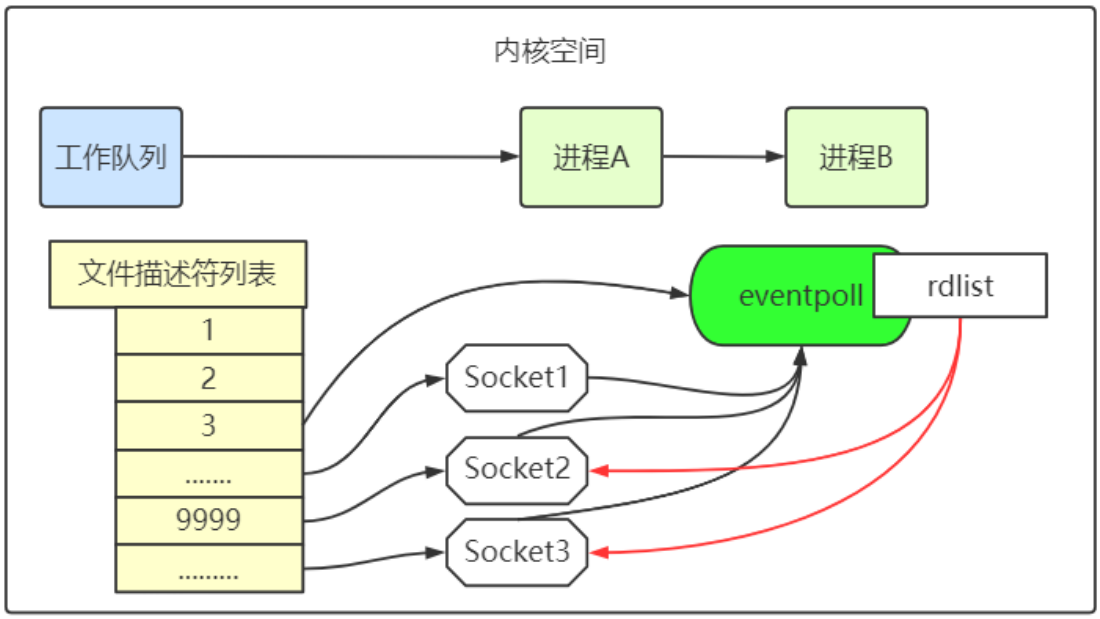
epoll 的原理和流程

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。

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。



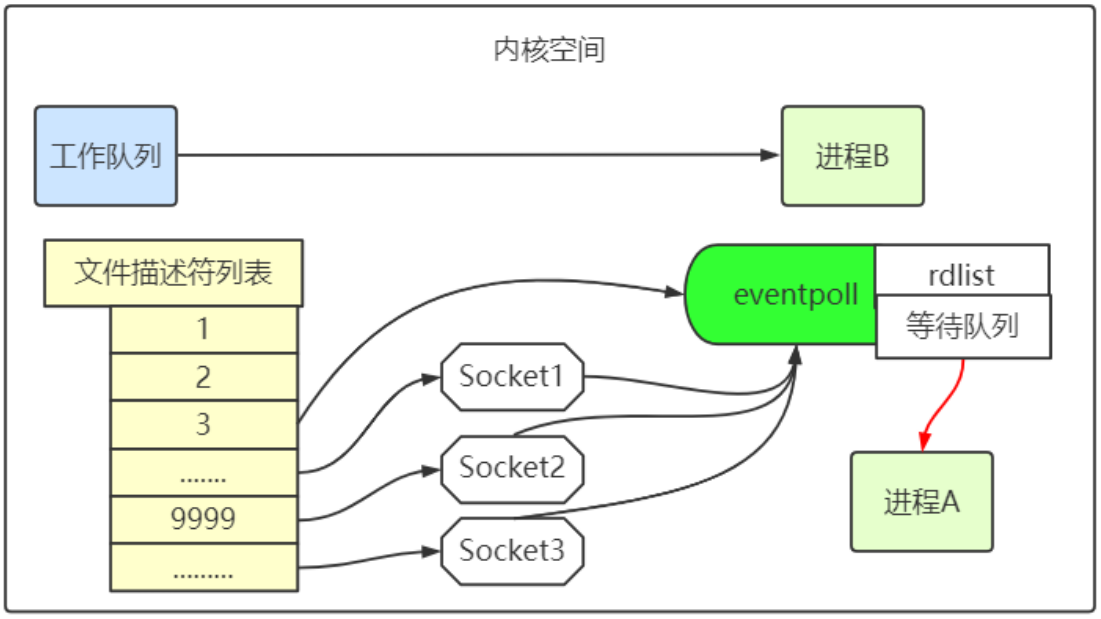
当 socket 收到数据后，中断程序会操作 eventpoll 对象，而不是直接操作进程。中断程序会给 eventpoll 的“就绪列表”添加 socket 引用。如下图展示的是 sock2 和 sock3 收到数据后，中断程序让 rdlist 引用这两个 socket。



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。

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。



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。也因为 rdlist 的存在，进程 A 可以知道哪些 socket 发生了变化。

epoll 的实现细节

现在对 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？

思考两个问题，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？

```
struct eventpoll {
    /*
     * This mutex is used to ensure that files are not removed
     * while epoll is using them. This is held during the event
     * collection loop, the file cleanup path, the epoll file exit
     * code and the ctl operations.
     */
    struct mutex mtx;

    /* Wait queue used by sys_epoll_wait() */
    wait_queue_head_t wq;

    /* Wait queue used by file->poll() */
    wait_queue_head_t poll_wait;

    /* List of ready file descriptors */
    struct list_head rdllist;

    /* Lock which protects rdllist and ovflist */
    rwlock_t lock;

    /* RB tree root used to store monitored fd structs */
    struct rb_root_cached rbr;

    /*
     * This is a single linked list that chains all the "struct epitem" that
     * happened while transferring ready events to userspace w/out
     * holding ->lock.
     */
    struct epitem *ovflist;

    /* wakeup_source used when ep_scan_ready_list is running */
    struct wakeup_source *ws;

    /* The user that created the eventpoll descriptor */
    struct user_struct *user;

    struct file *file;

    /* used to optimize loop detection check */
    u64 gen;
```

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，也就是 Linux 源码中的

```
/* List of ready file descriptors */
struct list_head rdllist;
```

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，也就是 Linux 源码中的

```
/* RB tree root used to store monitored fd structs */
struct rb_root_cached rbr;
```

总结

当某一进程调用 epoll_create 方法时，Linux 内核会创建一个 eventpoll 结构体，在内核 cache 里建了个红黑树用于存储以后 epoll_ctl 传来的 socket 外，还会再建立一个 rdllist 双向链表，用于存储准备就绪的事件，当 epoll_wait 调用时，仅仅观察这个 rdllist 双向链表里有没有数据即可。有数据就返回，没有数据就 sleep，等到 timeout 时间到后即使链表没数据也返回。

同时，所有添加到 epoll 中的事件都会与设备(如网卡)驱动程序建立回调关系，也就是说相应事件的发生时会调用这里的回调方法。这个回调方法在内核中叫做 ep_poll_callback，它会把这样的事件放到上面的 rdllist 双向链表中。

当调用 epoll_wait 检查是否有发生事件的连接时，只是检查 eventpoll 对象中的 rdllist 双向链表是否有 epitem 元素而已，如果 rdllist 链表不为空，则这里的事件复制到用户态内存（使用共享内存提高效率）中，同时将事件数量返回给用户。因此 epoll_waitx 效率非常高，可以轻易地处理百万级别的并发连接。