epoll

int epoll\_create(int size);

这就是创建一个epoll句柄，同时也占用一个文件描述符．size指明这个epoll监听的数目有多大．

typedef union epoll\_data {

void \*ptr;

int fd;

\_\_uint32\_t u32;

\_\_uint64\_t u64;

} epoll\_data\_t;

struct epoll\_event {

\_\_uint32\_t events; /\* Epoll events \*/

epoll\_data\_t data; /\* User data variable \*/

};

------------------------------------------------------------------

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

EPOLLIN 表示对应的文件描述符可读(包括对端socket关闭)

EPOLLOUT：表示对应的文件描述符可以写；

EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）

EPOLLERR：表示对应的文件描述符发生错误；

EPOLLHUP：表示对应的文件描述符被挂断；

EPOLLET： 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的；

EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里。

int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);

这个函数用于等待事件的发生．第二个参数是用户自己开辟的一块事件数组，用于存储就绪的事件，第三个参数为这个数组的最大值，就是第二个参数事件数组的最大值，用户想监听fd的个数，第四个参数为超时时间(0表示立即返回，-1表示永久阻塞，直到有就绪事件)

epoll经常使用框架包括监听listenfd以及clientfd，当epoll\_wait返回时，迭代每个事件，如果是listenfd,则接收客户端fd，并在epoll注册一个读事件；如果是clientfd的可读事件，则先读取数据，然后处理数据，将数据写进输出缓冲区，最后将clientfd可读事件改为可写事件，这也是异步写的精髓；如果是clientfd可写事件，则先发送数据，然后将可写事件改为可读事件．

for( ; ; )

{

nfds = epoll\_wait(epfd,events,20,500);

for(i=0;i<nfds;++i)

{

if(events[i].data.fd==listenfd)

//如果是主socket的事件，则表示有新的连接

{

connfd = accept(listenfd,(sockaddr \*)&clientaddr, &clilen);

//accept这个连接

ev.data.fd=connfd;

ev.events=EPOLLIN|EPOLLET;

epoll\_ctl(epfd,EPOLL\_CTL\_ADD,connfd,&ev);

//将新的fd添加到epoll的监听队列中

}

else if( events[i].events&EPOLLIN ) //接收到数据，读socket

{

if ( (sockfd = events[i].data.fd) < 0) continue;

n = read(sockfd, line, MAXLINE)< 0 //读

ev.data.ptr = md;

//md为自定义类型，添加数据

ev.events=EPOLLOUT|EPOLLET;

epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);

//修改标识符，等待下一个循环时发送数据，异步处理的精髓

}

else if(events[i].events&EPOLLOUT)

//有数据待发送，写socket

{

struct myepoll\_data\* md = (myepoll\_data\*)events[i].data.ptr;

//取数据

sockfd = md->fd;

send( sockfd, md->ptr, strlen((char\*)md->ptr), 0 );

//发送数据

ev.data.fd=sockfd;

ev.events=EPOLLIN|EPOLLET;

epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);

//修改标识符，等待下一个循环时接收数据

}

else

{

//其他情况的处理

}

}

}

在linux，一切皆文件．所以当调用epoll\_create时，内核给这个epoll分配一个file，但是这个不是普通的文件，而是只服务于epoll．

所以当内核初始化epoll时，会开辟一块内核高速cache区，用于安置我们监听的socket，这些socket会以红黑树的形式保存在内核的cache里，以支持快速的查找，插入，删除．同时，建立了一盒list链表，用于存储准备就绪的事件．所以调用epoll\_wait时，在timeout时间内，只是简单的观察这个list链表是否有数据，如果没有，则睡眠至超时时间到返回；如果有数据，则在超时时间到，拷贝至用户态events数组中．

那么，这个准备就绪list链表是怎么维护的呢？当我们执行epoll\_ctl时，除了把socket放到epoll文件系统里file对象对应的红黑树上之外，还会给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪list链表里。所以，当一个socket上有数据到了，内核在把网卡上的数据copy到内核中后就来把socket插入到准备就绪链表里了。

epoll有两种模式LT(水平触发)和ET(边缘触发)，LT模式下，主要缓冲区数据一次没有处理完，那么下次epoll\_wait返回时，还会返回这个句柄；而ET模式下，缓冲区数据一次没处理结束，那么下次是不会再通知了，只在第一次返回．所以在ET模式下，一般是通过while循环，一次性读完全部数据．epoll默认使用的是LT．

这件事怎么做到的呢？当一个socket句柄上有事件时，内核会把该句柄插入上面所说的准备就绪list链表，这时我们调用epoll\_wait，会把准备就绪的socket拷贝到用户态内存，然后清空准备就绪list链表，最后，epoll\_wait干了件事，就是检查这些socket，如果不是ET模式（就是LT模式的句柄了），并且这些socket上确实有未处理的事件时，又把该句柄放回到刚刚清空的准备就绪链表了。所以，非ET的句柄，只要它上面还有事件，epoll\_wait每次都会返回。而ET模式的句柄，除非有新中断到，即使socket上的事件没有处理完，也是不会次次从epoll\_wait返回的．

经常看到比较ET和LT模式到底哪个效率高的问题．有一个回答是说ET模式下减少epoll系统调用．这话没错，也可以理解，但是在ET模式下，为了避免数据饿死问题，用户态必须用一个循环，将所有的数据一次性处理结束．所以在ET模式下下，虽然epoll系统调用减少了，但是用户态的逻辑复杂了，write/read调用增多了．所以这不好判断，要看用户的性能瓶颈在哪．

epoll与select最后需要说明的就是epoll与select/poll相比的优点．

首先select/poll监听的文件描述符个数受限．select的文件描述符默认为2048，而现在的服务器连接数在轻轻松松就超过2048个；epoll支持的fd个数不受限制，它支持的fd上限是最大可以打开文件的数目，一般远大于2048，1G内存的机器上是大约10万左右．

select和poll需要循环检测所有fd是否就绪，当fd数量百万或者更多时，这是很耗时的，根据前面原理分析可知，epoll只处理就绪的fd，而一般一次epoll\_wait返回时，就绪的fd是不多的，所以处理起来不是很耗时．

还有两点是关于用户态和内核态复制文件描述符，epoll使用的是共享内存，select全部复制，所以效率更低；epoll支持内核微调．