<https://blog.csdn.net/weiyuefei/article/details/53006659>

* Author：Echo Chen（陈斌）
* Email：chenb19870707@gmail.com
* Blog：[Blog.csdn.net/chen19870707](http://blog.csdn.net/chen19870707)
* Date：Jan.7th, 2015

## 

## 1.什么是epoll

epoll是当前在Linux下开发大规模并发网络程序的热门人选，epoll 在Linux2.6内核中正式引入，和select相似，都是**I/O多路复用(IO multiplexing)技术**,按照man手册的说法：是为处理大批量句柄而作了改进的poll。

Linux下有以下几个经典的服务器模型：

#### 

#### ****①Apache模型（Process Per Connection，简称PPC） 和 TPC（Thread Per Connection）模型****

这两种模型思想类似，就是让每一个到来的连接都有一个进程/线程来服务。这种模型的代价是它要时间和空间。连接较多时，进程/线程切换的开销比较大。因此这类模型能接受的最大连接数都不会高，一般在几百个左右。

#### ②select模型

最大并发数限制：因为一个进程所打开的fd（文件描述符）是有限制的，由FD\_SETSIZE设置，默认值是1024/2048，因此select模型的最大并发数就被相应限制了。

效率问题：select每次调用都会线性扫描全部的fd集合，这样效率就会呈现线性下降，把FD\_SETSIZE改大可能造成这些fd都超时了。

内核/用户空间内存拷贝问题：如何让内核把fd消息通知给用户空间呢？在这个问题上select采取了内存拷贝方法。

#### ③poll模型

虽然解决了select 最大并发数的限制，但是依然存在select的效率问题，select缺点的2和3它都没有改掉。

#### ****④epoll模型****

对比其他模型的问题，epoll的改进如下：

**1.支持一个进程打开大数目的socket描述符(FD)**   
    select 最不能忍受的是一个进程所打开的FD是有一定限制的，由FD\_SETSIZE设置，默认值是2048。对于那些需要支持的上万连接数目的IM服务器来说显然太少了。这时候你一是可以选择修改这个宏然后重新编译内核，不过资料也同时指出这样会带来网络效率的下降，二是可以选择多进程的解决方案(传统的 Apache方案)，不过虽然linux上面创建进程的代价比较小，但仍旧是不可忽视的，加上进程间数据同步远比不上线程间同步的高效，所以也不是一种完美的方案。不过 epoll则没有这个限制，它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于2048,举个例子,在1GB内存的机器上大约是10万左右，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。   
    
   **2.IO效率不随FD数目增加而线性下降**   
    传统的select/poll另一个致命弱点就是当你拥有一个很大的socket集合，不过由于网络延时，任一时间只有部分的socket是"活跃"的，但是select/poll每次调用都会线性扫描全部的集合，导致效率呈现线性下降。但是epoll不存在这个问题，它只会对"活跃"的socket进行操作---这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。那么，只有"活跃"的socket才会主动的去调用 callback函数，其他idle状态socket则不会，在这点上，epoll实现了一个"伪"AIO，因为这时候推动力在os内核。在一些 benchmark中，如果所有的socket基本上都是活跃的---比如一个高速LAN环境，epoll并不比select/poll有什么效率，相反，如果过多使用epoll\_ctl,效率相比还有稍微的下降。但是一旦使用idle connections模拟WAN环境,epoll的效率就远在select/poll之上了。  
    
**3.使用mmap加速内核与用户空间的消息传递**   
    这点实际上涉及到epoll的具体实现了。无论是select,poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存拷贝就很重要，在这点上，epoll是通过内核于用户空间mmap同一块内存实现的。而如果你想我一样从2.5内核就关注epoll的话，一定不会忘记手工 mmap这一步的。  
    
**4.内核微调**   
      这一点其实不算epoll的优点了，而是整个linux平台的优点。也许你可以怀疑linux平台，但是你无法回避linux平台赋予你微调内核的能力。比如，内核TCP/IP协议栈使用内存池管理sk\_buff结构，那么可以在运行时期动态调整这个内存pool(skb\_head\_pool)的大小--- 通过echo XXXX>/proc/sys/net/core/hot\_list\_length完成。再比如listen函数的第2个参数(TCP完成3次握手的数据包队列长度)，也可以根据你平台内存大小动态调整。更甚至在一个数据包面数目巨大但同时每个数据包本身大小却很小的特殊系统上尝试最新的NAPI网卡驱动[**架构**](http://lib.csdn.net/base/16)。

## 2.Epoll API

epoll只有epoll\_create,epoll\_ctl,epoll\_wait 3个系统调用。

1: #include  <sys/epoll.h>

2:

3: int  epoll\_create(int  size);

4:

5: int  epoll\_ctl(int epfd, int op, int fd, structepoll\_event \*event);

6:

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

8:

9:

#### ① int epoll\_create(int size);

创建一个epoll的句柄。自从linux2.6.8之后，size参数是被忽略的。需要注意的是，当创建好epoll句柄后，它就是会占用一个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。

#### ②int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

epoll的事件注册函数，它不同于select()是在监听事件时告诉内核要监听什么类型的事件，而是在这里先注册要监听的事件类型。   
第一个参数是epoll\_create()的返回值。   
第二个参数表示动作，用三个宏来表示：   
EPOLL\_CTL\_ADD：注册新的fd到epfd中；   
EPOLL\_CTL\_MOD：修改已经注册的fd的监听事件；   
EPOLL\_CTL\_DEL：从epfd中删除一个fd；   
    
第三个参数是需要监听的fd。   
第四个参数是告诉内核需要监听什么事，struct epoll\_event结构如下：

1: //保存触发事件的某个文件描述符相关的数据（与具体使用方式有关）

2:

3: typedef union epoll\_data {

4:     void \*ptr;

5:     int fd;

6:     \_\_uint32\_t u32;

7:     \_\_uint64\_t u64;

8: } epoll\_data\_t;

9:  //感兴趣的事件和被触发的事件

10: struct epoll\_event {

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

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

13: };

events可以是以下几个宏的集合：   
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；   
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

#### ③ int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);

收集在epoll监控的事件中已经发送的事件。参数events是分配好的epoll\_event结构体数组，epoll将会把发生的事件赋值到events数组中（events不可以是空指针，内核只负责把数据复制到这个events数组中，不会去帮助我们在用户态中分配内存）。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）。如果函数调用成功，返回对应I/O上已准备好的文件描述符数目，如返回0表示已超时。

## 3.Epoll  工作模式

#### ①LT模式：Level Triggered水平触发

这个是缺省的工作模式。同时支持block socket和non-block socket。内核会告诉程序员一个文件描述符是否就绪了。如果程序员不作任何操作，内核仍会通知。

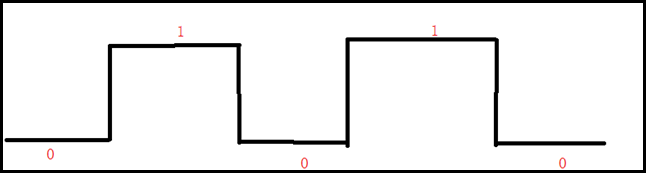
#### ②ET模式：Edge Triggered 边缘触发

是一种高速模式。仅当状态发生变化的时候才获得通知。这种模式假定程序员在收到一次通知后能够完整地处理事件，于是内核不再通知这一事件。注意：缓冲区中还有未处理的数据不算状态变化，所以ET模式下程序员只读取了一部分数据就再也得不到通知了，正确的用法是程序员自己确认读完了所有的字节（一直调用read/write直到出错EAGAIN为止）。

如下图：

0：表示文件描述符未准备就绪

1：表示文件描述符准备就绪

[](https://img-blog.csdn.net/20150108154507640)

对于水平触发模式(LT)：在1处，如果你不做任何操作，内核依旧会不断的通知进程文件描述符准备就绪。

对于边缘出发模式(ET): 只有在0变化到1处的时候，内核才会通知进程文件描述符准备就绪。之后如果不在发生文件描述符状态变化，内核就不会再通知进程文件描述符已准备就绪。

Nginx 默认采用的就是ET。

## 4.实例

1: #include <stdio.h>

2: #include <stdlib.h>

3: #include <unistd.h>

4: #include <sys/socket.h>

5: #include <errno.h>

6: #include <sys/epoll.h>

7: #include <netinet/in.h>

8: #include <fcntl.h>

9: #include <string.h>

10:  #include <netdb.h>

11:

12:

13:

14: struct epoll\_event  \*events = NULL;

15: int epollFd = -1;

16:

17: const int MAX\_SOCK\_NUM = 1024;

18:

19:

20: int epoll\_init();

21: int epoll\_socket(int domain, int type, int protocol);

22: int epoll\_cleanup();

23: int epoll\_new\_conn(int sfd);

24:

25:

26: int main()

27: {

28:       struct sockaddr\_in listenAddr;

29:       int listenFd = -1;

30:

31:       if(-1 == epoll\_init())

32:       {

33:           printf("epoll\_init err\n");

34:           return -1;

35:       }

36:

37:       if((listenFd = epoll\_socket(AF\_INET,SOCK\_STREAM,0)) == -1)

38:       {

39:           printf("epoll\_socket err\n");

40:           epoll\_cleanup();

41:           return -1;

42:       }

43:

44:       listenAddr.sin\_family = AF\_INET;

45:       listenAddr.sin\_port = htons(999);

46:       listenAddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

47:

48:       if(-1 == bind(listenFd,(struct sockaddr\*)&listenAddr,sizeof(listenAddr)))

49:       {

50:           printf("bind err %d\n",errno);

51:           epoll\_cleanup();

52:           return -1;

53:       }

54:

55:       if(-1 == listen(listenFd,1024))

56:       {

57:           printf("listen err\n");

58:           epoll\_cleanup();

59:           return -1;

60:       }

61:

62:       //Add ListenFd into epoll

63:       if(-1 == epoll\_new\_conn(listenFd))

64:       {

65:           printf("eph\_new\_conn err\n");

66:           close(listenFd);

67:         epoll\_cleanup();

68:         return -1;

69:       }

70:

71:

72:       //LOOP

73:       while(1)

74:       {

75:           int n;

76:           n = epoll\_wait(listenFd,events,MAX\_SOCK\_NUM,-1);

77:           for (int i = 0; i < n; i++)

78:           {

79:                if( (events[i].events & EPOLLERR) || ( events[i].events & EPOLLHUP ) || !(events[i].events & EPOLLIN) )

80:                {

81:                    printf("epoll err\n");

82:                    close(events[i].data.fd);

83:                    continue;

84:                }

85:                else if(events[i].data.fd == listenFd)

86:                {

87:                    while(1)

88:                    {

89:                        struct sockaddr inAddr;

90:                        char hbuf[1024],sbuf[NI\_MAXSERV];

91:                        socklen\_t inLen = -1;

92:                        int inFd = -1;

93:                        int s = 0;

94:                        int flag = 0;

95:

96:                        inLen = sizeof(inAddr);

97:                        inFd = accept(listenFd,&inAddr,&inLen);

98:

99:                        if(inFd == -1)

100:                        {

101:                            if( errno == EAGAIN || errno == EWOULDBLOCK )

102:                            {

103:                                break;

104:                            }

105:                            else

106:                            {

107:                                printf("accept error\n");

108:                                break;

109:                            }

110:                        }

111:

112:                     if (s ==  getnameinfo (&inAddr, inLen, hbuf, sizeof(hbuf), sbuf, sizeof(sbuf), NI\_NUMERICHOST | NI\_NUMERICSERV))

113:                     {

114:                         printf("Accepted connection on descriptor %d (host=%s, port=%s)\n", inFd, hbuf, sbuf);

115:                     }

116:

117:                     //Set Socket to non-block

118:                     if((flag = fcntl(inFd,F\_GETFL,0)) < 0 || fcntl(inFd,F\_SETFL,flag | O\_NONBLOCK) < 0)

119:                     {

120:                         close(inFd);

121:                         return -1;

122:                     }

123:

124:                     epoll\_new\_conn(inFd);

125:                    }

126:                }

127:                else

128:                {

129:                         while (1)

130:                         {

131:                         ssize\_t count;

132:                         char buf[512];

133:

134:                         count = read (events[i].data.fd, buf, sizeof buf);

135:

136:                         if (count == -1)

137:                         {

138:                             if (errno != EAGAIN)

139:                              {

140:                                 printf("read err\n");

141:                                 }

142:

143:                             break;

144:

145:                         }

146:                         else if (count == 0)

147:                         {

148:                             break;

149:                         }

150:

151:                         write (1, buf, count);

152:                     }

153:                 }

154:           }

155:

156:       }

157:

158:       epoll\_cleanup();

159: }

160:

161:

162: int epoll\_init()

163: {

164:     if(!(events = (struct epoll\_event\* ) malloc ( MAX\_SOCK\_NUM \* sizeof(struct epoll\_event))))

165:     {

166:         return -1;

167:     }

168:

169:     if( (epollFd = epoll\_create(MAX\_SOCK\_NUM)) < 0 )

170:     {

171:         return -1;

172:     }

173:

174:     return 0;

175: }

176:

177: int epoll\_socket(int domain, int type, int protocol)

178: {

179:     int sockFd = -1;

180:     int flag = -1;

181:

182:     if ((sockFd = socket(domain,type,protocol)) < 0)

183:     {

184:         return -1;

185:     }

186:

187:     //Set Socket to non-block

188:     if((flag = fcntl(sockFd,F\_GETFL,0)) < 0 || fcntl(sockFd,F\_SETFL,flag | O\_NONBLOCK) < 0)

189:     {

190:         close(sockFd);

191:         return -1;

192:     }

193:

194:     return sockFd;

195: }

196:

197: int epoll\_cleanup()

198: {

199:     free(events);

200:     close(epollFd);

201:     return 0;

202: }

203:

204: int epoll\_new\_conn(int sfd)

205: {

206:

207:       struct epoll\_event  epollEvent;

208:       memset(&epollEvent, 0, sizeof(struct epoll\_event));

209:       epollEvent.events = EPOLLIN | EPOLLERR | EPOLLHUP | EPOLLET;

210:       epollEvent.data.ptr = NULL;

211:       epollEvent.data.fd  = sfd;

212:

213:       if (epoll\_ctl(epollFd, EPOLL\_CTL\_ADD, sfd, &epollEvent) < 0)

214:       {

215:         return -1;

216:       }

217:

218:     epollEvent.data.fd  = sfd;

219:

220:     return 0;

221: }

## 5.Epoll为什么高效

Epoll高效主要体现在以下三个方面：

①从上面的调用方式就可以看出epoll比select/poll的一个优势：select/poll每次调用都要传递所要监控的所有fd给select/poll系统调用（这意味着每次调用都要将fd列表从用户态拷贝到内核态，当fd数目很多时，这会造成低效）。而每次调用epoll\_wait时（作用相当于调用select/poll），不需要再传递fd列表给内核，因为已经在epoll\_ctl中将需要监控的fd告诉了内核（epoll\_ctl不需要每次都拷贝所有的fd，只需要进行增量式操作）。所以，在调用epoll\_create之后，内核已经在内核态开始准备数据结构存放要监控的fd了。每次epoll\_ctl只是对这个数据结构进行简单的维护。

② 此外，**内核使用了slab机制，为epoll提供了快速的数据结构：**

在内核里，一切皆文件。所以，**epoll向内核注册了一个文件系统，用于存储上述的被监控的fd**。当你调用epoll\_create时，就会在这个虚拟的epoll文件系统里创建一个file结点。当然这个file不是普通文件，它只服务于epoll。epoll在被内核初始化时（操作系统启动），同时会开辟出epoll自己的内核高速cache区，用于安置每一个我们想监控的fd，**这些fd会以红黑树的形式保存在内核cache里，以支持快速的查找、插入、删除。**这个内核高速cache区，就是建立连续的物理内存页，然后在之上建立slab层，简单的说，就是物理上分配好你想要的size的内存对象，每次使用时都是使用空闲的已分配好的对象。

③ epoll的第三个优势在于：当我们调用epoll\_ctl往里塞入百万个fd时，epoll\_wait仍然可以飞快的返回，并有效的将发生事件的fd给我们用户。**这是由于我们在调用epoll\_create时，内核除了帮我们在epoll文件系统里建了个file结点，在内核cache里建了个红黑树用于存储以后epoll\_ctl传来的fd外，还会再建立一个list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list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。所以，epoll\_wait非常高效。而且，通常情况下即使我们要监控百万计的fd，大多一次也只返回很少量的准备就绪fd而已，所以，epoll\_wait仅需要从内核态copy少量的fd到用户态而已。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fd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fd的中断到了，就把它放到准备就绪list链表里。所以，当一个fd（例如socket）上有数据到了，内核在把设备（例如网卡）上的数据copy到内核中后就来把fd（socket）插入到准备就绪list链表里了。

如此，一颗红黑树，一张准备就绪fd链表，少量的内核cache，就帮我们解决了大并发下的fd（socket）处理问题。

**1.执行epoll\_create时，创建了红黑树和就绪list链表。**

**2.执行epoll\_ctl时，如果增加fd（socket），则检查在红黑树中是否存在，存在立即返回，不存在则添加到红黑树上，然后向内核注册回调函数，用于当中断事件来临时向准备就绪list链表中插入数据。**

**3.执行epoll\_wait时立刻返回准备就绪链表里的数据即可。**

## 6.Epoll源码分析

1: static int \_\_init eventpoll\_init(void)

2: {

3:   mutex\_init(&pmutex);

4:

5:   ep\_poll\_safewake\_init(&psw);

6:

7:   epi\_cache = kmem\_cache\_create("eventpoll\_epi", sizeof(struct epitem), 0, SLAB\_HWCACHE\_ALIGN|EPI\_SLAB\_DEBUG|SLAB\_PANIC, NULL);

8:

9:   pwq\_cache = kmem\_cache\_create("eventpoll\_pwq", sizeof(struct eppoll\_entry), 0, EPI\_SLAB\_DEBUG|SLAB\_PANIC, NULL);

10:

11:   return 0;

12: }

epoll用kmem\_cache\_create（slab分配器）分配内存用来存放**struct** epitem和**struct** eppoll\_entry。

当向系统中添加一个fd时，就创建一个epitem结构体，这是内核管理epoll的基本数据结构：

1: struct epitem

2: {

3:     struct rb\_node  rbn;        //用于主结构管理的红黑树

4:

5:     struct list\_head  rdllink;  //事件就绪队列

6:

7:     struct epitem  \*next;       //用于主结构体中的链表

8:

9:     struct epoll\_filefd  ffd;   //这个结构体对应的被监听的文件描述符信息

10:

11:     int  nwait;                 //poll操作中事件的个数

12:

13:     struct list\_head  pwqlist;  //双向链表，保存着被监视文件的等待队列，功能类似于select/poll中的poll\_table

14:

15:     struct eventpoll  \*ep;      //该项属于哪个主结构体（多个epitm从属于一个eventpoll）

16:

17:     struct list\_head  fllink;   //双向链表，用来链接被监视的文件描述符对应的struct file。因为file里有f\_ep\_link,用来保存所有监视这个文件的epoll节点

18:

19:     struct epoll\_event  event;  //注册的感兴趣的事件,也就是用户空间的epoll\_event

20:

21: }

而每个epoll fd（epfd）对应的主要数据结构为：

1: struct eventpoll

2: {

3:     spin\_lock\_t       lock;             //对本数据结构的访问

4:

5:     struct mutex      mtx;              //防止使用时被删除

6:

7:     wait\_queue\_head\_t     wq;           //sys\_epoll\_wait() 使用的等待队列

8:

9:     wait\_queue\_head\_t   poll\_wait;      //file->poll()使用的等待队列

10:

11:     struct list\_head    rdllist;        //事件满足条件的链表

12:

13:     struct rb\_root      rbr;            //用于管理所有fd的红黑树（树根）

14:

15:     struct epitem      \*ovflist;       //将事件到达的fd进行链接起来发送至用户空间

16:

17: }

18:

eventpoll在epoll\_create时创建:

1: long sys\_epoll\_create(int size)

2: {

3:

4:     struct eventpoll \*ep;

5:

6:     ...

7:

8:     ep\_alloc(&ep); //为ep分配内存并进行初始化

9:

10: /\* 调用anon\_inode\_getfd 新建一个file instance，也就是epoll可以看成一个文件（匿名文件）。因此我们可以看到epoll\_create会返回一个fd。epoll所管理的所有的fd都是放在一个大的结构eventpoll(红黑树)中，

11: 将主结构体struct eventpoll \*ep放入file->private项中进行保存（sys\_epoll\_ctl会取用）\*/

12:

13:  fd = anon\_inode\_getfd("[eventpoll]", &eventpoll\_fops, ep, O\_RDWR | (flags & O\_CLOEXEC));

14:

15:      return fd;

16:

17: }

其中，**ep\_alloc**(**struct** eventpoll \*\*pep)为pep分配内存，并初始化。

其中，上面注册的操作eventpoll\_fops定义如下：

1: static const struct file\_operations eventpoll\_fops = {

2:

3:     .release=  ep\_eventpoll\_release,

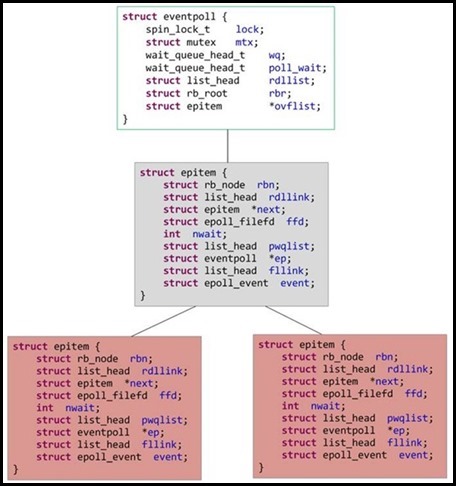
4:

5:     .poll    =  ep\_eventpoll\_poll,

6:

7: };

这样说来，内核中维护了一棵红黑树，大致的结构如下：

[](https://img-blog.csdn.net/20150108154531844)

接着是epoll\_ctl函数（省略了出错检查等代码）：

1: asmlinkage long sys\_epoll\_ctl(int epfd,int op,int fd,struct epoll\_event \_\_user \*event) {

2:

3:    int error;

4:

5:    struct file \*file,\*tfile;

6:

7:    struct eventpoll \*ep;

8:

9:    struct epoll\_event epds;

10:

11:

12:

13:    error = -FAULT;

14:

15:    //判断参数的合法性，将 \_\_user \*event 复制给 epds。

16:

17:    if(ep\_op\_has\_event(op) && copy\_from\_user(&epds,event,sizeof(struct epoll\_event)))

18:

19:            goto error\_return; //省略跳转到的代码

20:

21:

22:

23:    file  = fget (epfd); // epoll fd 对应的文件对象

24:

25:    tfile = fget(fd);    // fd 对应的文件对象

26:

27:

28:

29:    //在create时存入进去的（anon\_inode\_getfd），现在取用。

30:

31:    ep = file->private->data;

32:

33:

34:

35:    mutex\_lock(&ep->mtx);

36:

37:

38:

39:    //防止重复添加（在ep的红黑树中查找是否已经存在这个fd）

40:

41:    epi = epi\_find(ep,tfile,fd);

42:

43:

44:

45:    switch(op)

46:

47:    {

48:

49:       ...

50:

51:        case EPOLL\_CTL\_ADD:  //增加监听一个fd

52:

53:            if(!epi)

54:

55:            {

56:

57:                epds.events |= EPOLLERR | POLLHUP;     //默认包含POLLERR和POLLHUP事件

58:

59:                error = ep\_insert(ep,&epds,tfile,fd);  //在ep的红黑树中插入这个fd对应的epitm结构体。

60:

61:            } else  //重复添加（在ep的红黑树中查找已经存在这个fd）。

62:

63:                error = -EEXIST;

64:

65:            break;

66:

67:        ...

68:

69:    }

70:

71:    return error;

72:

73:

74:

ep\_insert的实现如下：

1: static int ep\_insert(struct eventpoll \*ep, struct epoll\_event \*event, struct file \*tfile, int fd)

2:

3: {

4:

5:    int error ,revents,pwake = 0;

6:

7:    unsigned long flags ;

8:

9:    struct epitem \*epi;

10:

11:    /\*

12:

13:       struct ep\_queue{

14:

15:          poll\_table pt;

16:

17:          struct epitem \*epi;

18:

19:       }   \*/

20:

21:

22:

23:    struct ep\_pqueue epq;

24:

25:

26:

27:    //分配一个epitem结构体来保存每个加入的fd

28:

29:    if(!(epi = kmem\_cache\_alloc(epi\_cache,GFP\_KERNEL)))

30:

31:       goto error\_return;

32:

33:    //初始化该结构体

34:

35:    ep\_rb\_initnode(&epi->rbn);

36:

37:    INIT\_LIST\_HEAD(&epi->rdllink);

38:

39:    INIT\_LIST\_HEAD(&epi->fllink);

40:

41:    INIT\_LIST\_HEAD(&epi->pwqlist);

42:

43:    epi->ep = ep;

44:

45:    ep\_set\_ffd(&epi->ffd,tfile,fd);

46:

47:    epi->event = \*event;

48:

49:    epi->nwait = 0;

50:

51:    epi->next = EP\_UNACTIVE\_PTR;

52:

53:

54:

55:    epq.epi = epi;

56:

57:    //安装poll回调函数

58:

59:    init\_poll\_funcptr(&epq.pt, ep\_ptable\_queue\_proc );

60:

61:    /\* 调用poll函数来获取当前事件位，其实是利用它来调用注册函数ep\_ptable\_queue\_proc（poll\_wait中调用）。

62:

63:        如果fd是套接字，f\_op为socket\_file\_ops，poll函数是

64:

65:        sock\_poll()。如果是TCP套接字的话，进而会调用

66:

67:        到tcp\_poll()函数。此处调用poll函数查看当前

68:

69:        文件描述符的状态，存储在revents中。

70:

71:        在poll的处理函数(tcp\_poll())中，会调用sock\_poll\_wait()，

72:

73:        在sock\_poll\_wait()中会调用到epq.pt.qproc指向的函数，

74:

75:        也就是ep\_ptable\_queue\_proc()。  \*/

76:

77:

78:

79:    revents = tfile->f\_op->poll(tfile, &epq.pt);

80:

81:

82:

83:    spin\_lock(&tfile->f\_ep\_lock);

84:

85:    list\_add\_tail(&epi->fllink,&tfile->f\_ep\_lilnks);

86:

87:    spin\_unlock(&tfile->f\_ep\_lock);

88:

89:

90:

91:    ep\_rbtree\_insert(ep,epi); //将该epi插入到ep的红黑树中

92:

93:

94:

95:    spin\_lock\_irqsave(&ep->lock,flags);

96:

97:

98:

99: //  revents & event->events：刚才fop->poll的返回值中标识的事件有用户event关心的事件发生。

100:

101: // !ep\_is\_linked(&epi->rdllink)：epi的ready队列中有数据。ep\_is\_linked用于判断队列是否为空。

102:

103: /\*  如果要监视的文件状态已经就绪并且还没有加入到就绪队列中,则将当前的

104:

105:     epitem加入到就绪队列中.如果有进程正在等待该文件的状态就绪,则

106:

107:     唤醒一个等待的进程。  \*/

108:

109:

110:

111: if((revents & event->events) && !ep\_is\_linked(&epi->rdllink)) {

112:

113:       list\_add\_tail(&epi->rdllink,&ep->rdllist); //将当前epi插入到ep->ready队列中。

114:

115: /\* 如果有进程正在等待文件的状态就绪，

116:

117: 也就是调用epoll\_wait睡眠的进程正在等待，

118:

119: 则唤醒一个等待进程。

120:

121: waitqueue\_active(q) 等待队列q中有等待的进程返回1，否则返回0。

122:

123: \*/

124:

125:

126:

127:       if(waitqueue\_active(&ep->wq))

128:

129:          \_\_wake\_up\_locked(&ep->wq,TAKS\_UNINTERRUPTIBLE | TASK\_INTERRUPTIBLE);

130:

131:

132:

133: /\*  如果有进程等待eventpoll文件本身（???）的事件就绪，

134:

135:            则增加临时变量pwake的值，pwake的值不为0时，

136:

137:            在释放lock后，会唤醒等待进程。 \*/

138:

139:

140:

141:       if(waitqueue\_active(&ep->poll\_wait))

142:

143:          pwake++;

144:

145:    }

146:

147:    spin\_unlock\_irqrestore(&ep->lock,flags);

148:

149:

150:

151:

152:

153: if(pwake)

154:

155:       ep\_poll\_safewake(&psw,&ep->poll\_wait);//唤醒等待eventpoll文件状态就绪的进程

156:

157:    return 0;

158:

159: }

**init\_poll\_funcptr**(&epq.pt, ep\_ptable\_queue\_proc);

revents = tfile->f\_op->**poll**(tfile, &epq.pt);

这两个函数将ep\_ptable\_queue\_proc注册到epq.pt中的qproc。

1: typedef struct poll\_table\_struct {

2:

3: poll\_queue\_proc qproc;

4:

5: unsigned long key;

6:

7: }poll\_table;

执行f\_op->**poll**(tfile, &epq.pt)时，XXX\_poll(tfile, &epq.pt)函数会执行poll\_wait()，poll\_wait()会调用epq.pt.qproc函数，即ep\_ptable\_queue\_proc。

ep\_ptable\_queue\_proc函数如下：

1: /\*  在文件操作中的poll函数中调用，将epoll的回调函数加入到目标文件的唤醒队列中。

2:

3:     如果监视的文件是套接字，参数whead则是sock结构的sk\_sleep成员的地址。  \*/

4:

5: static void ep\_ptable\_queue\_proc(struct file \*file, wait\_queue\_head\_t \*whead, poll\_table \*pt) {

6:

7: /\* struct ep\_queue{

8:

9:          poll\_table pt;

10:

11:          struct epitem \*epi;

12:

13:       } \*/

14:

15:     struct epitem \*epi = ep\_item\_from\_epqueue(pt); //pt获取struct ep\_queue的epi字段。

16:

17:     struct eppoll\_entry \*pwq;

18:

19:

20:

21:     if (epi->nwait >= 0 && (pwq = kmem\_cache\_alloc(pwq\_cache, GFP\_KERNEL))) {

22:

23:         init\_waitqueue\_func\_entry(&pwq->wait, ep\_poll\_callback);

24:

25:         pwq->whead = whead;

26:

27:         pwq->base = epi;

28:

29:         add\_wait\_queue(whead, &pwq->wait);

30:

31:         list\_add\_tail(&pwq->llink, &epi->pwqlist);

32:

33:         epi->nwait++;

34:

35:     } else {

36:

37:         /\* We have to signal that an error occurred \*/

38:

39:         /\*

40:

41:          \* 如果分配内存失败，则将nwait置为-1，表示

42:

43:          \* 发生错误，即内存分配失败，或者已发生错误

44:

45:          \*/

46:

47:         epi->nwait = -1;

48:

49:     }

50:

51: }

其中**struct** eppoll\_entry定义如下：

1: struct eppoll\_entry {

2:

3: struct list\_head llink;

4:

5: struct epitem \*base;

6:

7: wait\_queue\_t wait;

8:

9: wait\_queue\_head\_t \*whead;

10:

11: };

ep\_ptable\_queue\_proc 函数完成 epitem 加入到特定文件的wait队列任务。

ep\_ptable\_queue\_proc有三个参数：

**struct** file \*file; 该fd对应的文件对象

wait\_queue\_head\_t \*whead; 该fd对应的设备等待队列（同select中的mydev->wait\_address）

poll\_table \*pt; f\_op->**poll**(tfile, &epq.pt)中的epq.pt

在ep\_ptable\_queue\_proc函数中，引入了另外一个非常重要的数据结构eppoll\_entry。eppoll\_entry主要完成epitem和epitem事件发生时的callback（**ep\_poll\_callback**）函数之间的关联。首先将eppoll\_entry的whead指向fd的设备等待队列（同select中的wait\_address），然后初始化eppoll\_entry的base变量指向epitem，最后通过add\_wait\_queue将epoll\_entry挂载到fd的设备等待队列上。完成这个动作后，epoll\_entry已经被挂载到fd的设备等待队列。

由于ep\_ptable\_queue\_proc函数设置了等待队列的**ep\_poll\_callback**回调函数。所以在设备硬件数据到来时，硬件中断处理函数中会唤醒该等待队列上等待的进程时，会调用唤醒函数**ep\_poll\_callback**

1: static int ep\_poll\_callback(wait\_queue\_t \*wait, unsigned mode, int sync, void \*key) {

2:

3:    int pwake = 0;

4:

5:    unsigned long flags;

6:

7:    struct epitem \*epi = ep\_item\_from\_wait(wait);

8:

9:    struct eventpoll \*ep = epi->ep;

10:

11:

12:

13:    spin\_lock\_irqsave(&ep->lock, flags);

14:

15:    //判断注册的感兴趣事件

16:

17: //#define EP\_PRIVATE\_BITS  (EPOLLONESHOT | EPOLLET)

18:

19: //有非EPOLLONESHONT或EPOLLET事件

20:

21:    if (!(epi->event.events & ~EP\_PRIVATE\_BITS))

22:

23:       goto out\_unlock;

24:

25:

26:

27:    if (unlikely(ep->ovflist != EP\_UNACTIVE\_PTR)) {

28:

29:       if (epi->next == EP\_UNACTIVE\_PTR) {

30:

31:          epi->next = ep->ovflist;

32:

33:          ep->ovflist = epi;

34:

35:       }

36:

37:       goto out\_unlock;

38:

39:    }

40:

41:

42:

43:    if (ep\_is\_linked(&epi->rdllink))

44:

45:       goto is\_linked;

46:

47:     //\*\*\*关键\*\*\*，将该fd加入到epoll监听的就绪链表中

48:

49:    list\_add\_tail(&epi->rdllink, &ep->rdllist);

50:

51:    //唤醒调用epoll\_wait()函数时睡眠的进程。用户层epoll\_wait(...) 超时前返回。

52:

53: if (waitqueue\_active(&ep->wq))

54:

55:       \_\_wake\_up\_locked(&ep->wq, TASK\_UNINTERRUPTIBLE | TASK\_INTERRUPTIBLE);

56:

57:    if (waitqueue\_active(&ep->poll\_wait))

58:

59:       pwake++;

60:

61:    out\_unlock: spin\_unlock\_irqrestore(&ep->lock, flags);

62:

63:    if (pwake)

64:

65:       ep\_poll\_safewake(&psw, &ep->poll\_wait);

66:

67:    return 1;

68:

69: }

所以**ep\_poll\_callback**函数主要的功能是将被监视文件的等待事件就绪时，将文件对应的epitem实例添加到就绪队列中，当用户调用epoll\_wait()时，内核会将就绪队列中的事件报告给用户。

epoll\_wait实现如下：

1: SYSCALL\_DEFINE4(epoll\_wait, int, epfd, struct epoll\_event \_\_user \*, events, int, maxevents, int, timeout)  {

2:

3:    int error;

4:

5:    struct file \*file;

6:

7:    struct eventpoll \*ep;

8:

9:     /\* 检查maxevents参数。 \*/

10:

11:    if (maxevents <= 0 || maxevents > EP\_MAX\_EVENTS)

12:

13:       return -EINVAL;

14:

15:     /\* 检查用户空间传入的events指向的内存是否可写。参见\_\_range\_not\_ok()。 \*/

16:

17:    if (!access\_ok(VERIFY\_WRITE, events, maxevents \* sizeof(struct epoll\_event))) {

18:

19:       error = -EFAULT;

20:

21:       goto error\_return;

22:

23:    }

24:

25:     /\* 获取epfd对应的eventpoll文件的file实例，file结构是在epoll\_create中创建。 \*/

26:

27:    error = -EBADF;

28:

29:    file = fget(epfd);

30:

31:    if (!file)

32:

33:       goto error\_return;

34:

35:     /\* 通过检查epfd对应的文件操作是不是eventpoll\_fops 来判断epfd是否是一个eventpoll文件。如果不是则返回EINVAL错误。 \*/

36:

37:    error = -EINVAL;

38:

39:    if (!is\_file\_epoll(file))

40:

41:       goto error\_fput;

42:

43:     /\* At this point it is safe to assume that the "private\_data" contains  \*/

44:

45:    ep = file->private\_data;

46:

47:     /\* Time to fish for events ... \*/

48:

49:    error = ep\_poll(ep, events, maxevents, timeout);

50:

51:     error\_fput:

52:

53:    fput(file);

54:

55: error\_return:

56:

57:    return error;

58:

59: }

60:

61:

62:

63: epoll\_wait调用ep\_poll，ep\_poll实现如下：

64:

65:  static int ep\_poll(struct eventpoll \*ep, struct epoll\_event \_\_user \*events, int maxevents, long timeout) {

66:

67:     int res, eavail;

68:

69:    unsigned long flags;

70:

71:    long jtimeout;

72:

73:    wait\_queue\_t wait;

74:

75:     /\* timeout是以毫秒为单位，这里是要转换为jiffies时间。这里加上999(即1000-1)，是为了向上取整。 \*/

76:

77:    jtimeout = (timeout < 0 || timeout >= EP\_MAX\_MSTIMEO) ?MAX\_SCHEDULE\_TIMEOUT : (timeout \* HZ + 999) / 1000;

78:

79:  retry:

80:

81:    spin\_lock\_irqsave(&ep->lock, flags);

82:

83:     res = 0;

84:

85:    if (list\_empty(&ep->rdllist)) {

86:

87:       /\* 没有事件，所以需要睡眠。当有事件到来时，睡眠会被ep\_poll\_callback函数唤醒。\*/

88:

89:       init\_waitqueue\_entry(&wait, current); //将current进程放在wait这个等待队列中。

90:

91:       wait.flags |= WQ\_FLAG\_EXCLUSIVE;

92:

93:       /\* 将当前进程加入到eventpoll的等待队列中，等待文件状态就绪或直到超时，或被信号中断。 \*/

94:

95:       \_\_add\_wait\_queue(&ep->wq, &wait);

96:

97:        for (;;) {

98:

99:          /\* 执行ep\_poll\_callback()唤醒时应当需要将当前进程唤醒，所以当前进程状态应该为“可唤醒”TASK\_INTERRUPTIBLE  \*/

100:

101:          set\_current\_state(TASK\_INTERRUPTIBLE);

102:

103:          /\* 如果就绪队列不为空，也就是说已经有文件的状态就绪或者超时，则退出循环。\*/

104:

105:          if (!list\_empty(&ep->rdllist) || !jtimeout)

106:

107:             break;

108:

109:          /\* 如果当前进程接收到信号，则退出循环，返回EINTR错误 \*/

110:

111:          if (signal\_pending(current)) {

112:

113:             res = -EINTR;

114:

115:             break;

116:

117:          }

118:

119:           spin\_unlock\_irqrestore(&ep->lock, flags);

120:

121:          /\* 主动让出处理器，等待ep\_poll\_callback()将当前进程唤醒或者超时,返回值是剩余的时间。

122:

123: 从这里开始当前进程会进入睡眠状态，直到某些文件的状态就绪或者超时。

124:

125: 当文件状态就绪时，eventpoll的回调函数ep\_poll\_callback()会唤醒在ep->wq指向的等待队列中的进程。\*/

126:

127:          jtimeout = schedule\_timeout(jtimeout);

128:

129:          spin\_lock\_irqsave(&ep->lock, flags);

130:

131:       }

132:

133:       \_\_remove\_wait\_queue(&ep->wq, &wait);

134:

135:        set\_current\_state(TASK\_RUNNING);

136:

137:    }

138:

139:     /\* ep->ovflist链表存储的向用户传递事件时暂存就绪的文件。

140:

141:     \* 所以不管是就绪队列ep->rdllist不为空，或者ep->ovflist不等于

142:

143:     \* EP\_UNACTIVE\_PTR，都有可能现在已经有文件的状态就绪。

144:

145:     \* ep->ovflist不等于EP\_UNACTIVE\_PTR有两种情况，一种是NULL，此时

146:

147:     \* 可能正在向用户传递事件，不一定就有文件状态就绪，

148:

149:     \* 一种情况时不为NULL，此时可以肯定有文件状态就绪，

150:

151:     \* 参见ep\_send\_events()。

152:

153:     \*/

154:

155:    eavail = !list\_empty(&ep->rdllist) || ep->ovflist != EP\_UNACTIVE\_PTR;

156:

157:     spin\_unlock\_irqrestore(&ep->lock, flags);

158:

159:     /\* Try to transfer events to user space. In case we get 0 events and there's still timeout left over, we go trying again in search of more luck. \*/

160:

161:    /\* 如果没有被信号中断，并且有事件就绪，但是没有获取到事件(有可能被其他进程获取到了)，并且没有超时，则跳转到retry标签处，重新等待文件状态就绪。 \*/

162:

163:    if (!res && eavail && !(res = ep\_send\_events(ep, events, maxevents)) && jtimeout)

164:

165:       goto retry;

166:

167:     /\* 返回获取到的事件的个数或者错误码 \*/

168:

169:    return res;

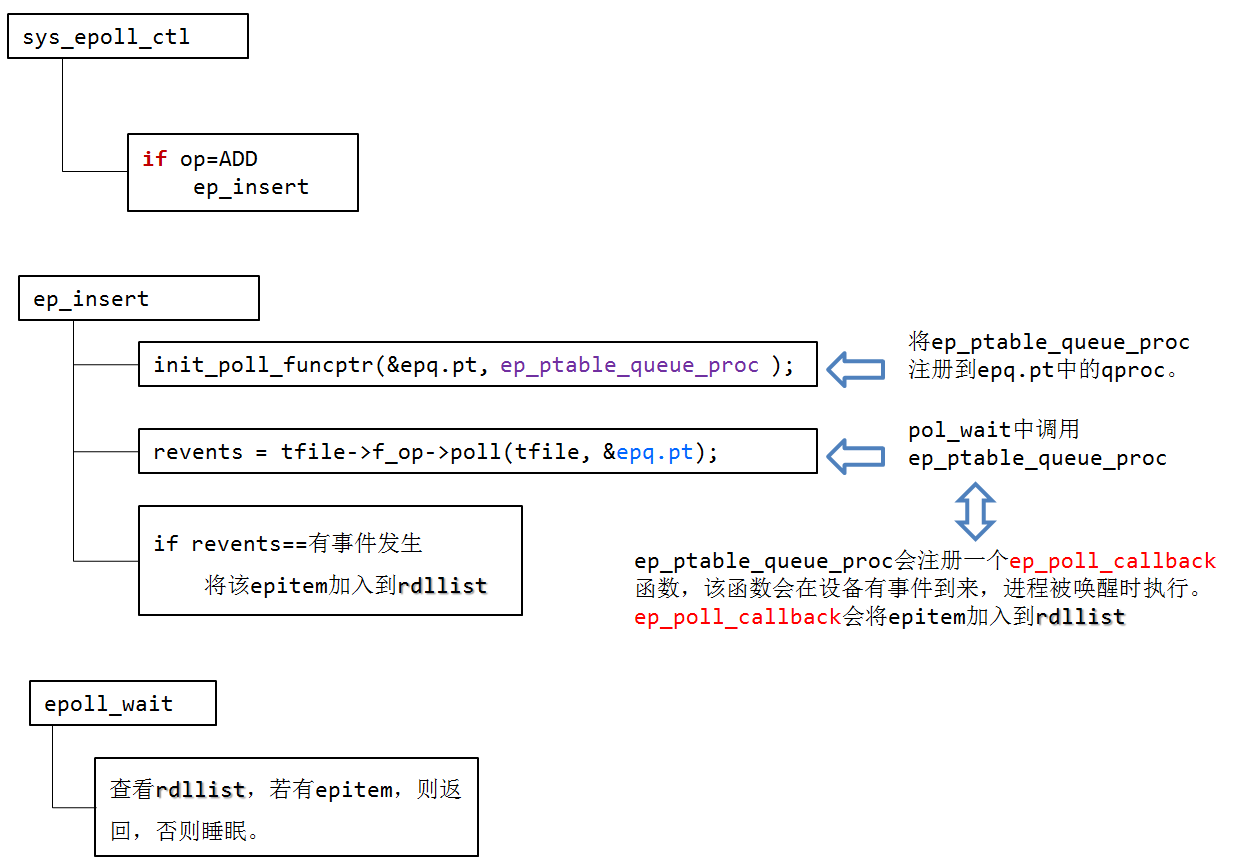
170:

171: }

**ep\_send\_events**函数向用户空间发送就绪事件。

ep\_send\_events()函数将用户传入的内存简单封装到ep\_send\_events\_data结构中，然后调用ep\_scan\_ready\_list() 将就绪队列中的事件传入用户空间的内存。

用户空间访问这个结果，进行处理。



## 7.参考

1.<http://www.cnblogs.com/apprentice89/p/3234677.html>

2.<http://www.cnblogs.com/apprentice89/archive/2013/05/06/3063039.html>