实例浅析epoll的水平触发和边缘触发，以及边缘触发为什么要使用非阻塞IO

一.基本概念

我们通俗一点讲：

Level_triggered(水平触发)：当被监控的文件描述符上有可读写事件发生时，epoll_wait()会通知处理程序去读写。如果这次没有把数据一次性全部读写完(如读写缓冲区太小)，那么下次调用 epoll_wait()时，它还会通知你在上没读写完的文件描述符上继续读写，当然如果你一直不去读写，它会一直通知你！！！如果系统中有大量你不需要读写的就绪文件描述符，而它们每次都会返回，这样会大大降低处理程序检索自己关心的就绪文件描述符的效率！！！

Edge_triggered(边缘触发)：当被监控的文件描述符上有可读写事件发生时，epoll_wait()会通知处理程序去读写。如果这次没有把数据全部读写完(如读写缓冲区太小)，那么下次调用epoll_wait()时，它不会通知你，也就是它只会通知你一次，直到该文件描述符上出现第二次可读写事件才会通知你！！！这种模式比水平触发效率高，系统不会充斥大量你不关心的就绪文件描述符！！！

阻塞IO：当你去读一个阻塞的文件描述符时，如果在该文件描述符上没有数据可读，那么它会一直阻塞(通俗一点就是一直卡在调用函数那里)，直到有数据可读。当你去写一个阻塞的文件描述符时，如果在该文件描述符上没有空间(通常是缓冲区)可写，那么它会一直阻塞，直到有空间可写。以上的读和写我们统一指在某个文件描述符进行的操作，不单单指真正的读数据，写数据，还包括接收连接accept()，发起连接connect()等操作...

非阻塞IO：当你去读写一个非阻塞的文件描述符时，不管可不可以读写，它都会立即返回，返回成功说明读写操作完成了，返回失败会设置相应errno状态码，根据这个errno可以进一步执行其他处理。它不会像阻塞IO那样，卡在那里不动！！！

二.几种IO模型的触发方式

select(),poll()模型都是水平触发模式，信号驱动IO是边缘触发模式，epoll()模型即支持水平触发，也支持边缘触发，默认是水平触发。

这里我们要探讨epoll()的水平触发和边缘触发，以及阻塞IO和非阻塞IO对它们的影响！！！下面称水平触发为LT，边缘触发为ET。

对于监听的socket文件描述符我们用sockfd代替，对于accept()返回的文件描述符(即要读写的文件描述符)用connfd代替。

我们来验证以下几个内容：

1.水平触发的非阻塞sockfd

2.边缘触发的非阻塞sockfd

3.水平触发的阻塞connfd

4.水平触发的非阻塞connfd

5.边缘触发的阻塞connfd

6.边缘触发的非阻塞connfd

以上没有验证阻塞的sockfd，因为epoll_wait()返回必定是已就绪的连接，设不设置阻塞accept()都会立即返回。例外：UNP里面有个例子，在BSD上，使用select()模型。设置阻塞的监听sockfd时，当客户端发起连接请求，由于服务器繁忙没有来得及accept()，此时客户端自己又断开，当服务器到达accept()时，会出现阻塞。本机测试epoll()模型没有出现这种情况，我们就暂且忽略这种情况！！！

三.验证代码

文件名：epoll_lt_et.c

```
1 /*
2  *url:http://www.cnblogs.com/yuuyuu/p/5103744.html
3  *
```

```
 4  */
 5
 6 #include <stdio.h>
 7 #include <stdlib.h>
 8 #include <string.h>
 9 #include <errno.h>
10 #include <unistd.h>
11 #include <fcntl.h>
12 #include <arpa/inet.h>
13 #include <netinet/in.h>
14 #include <sys/socket.h>
15 #include <sys/epoll.h>
16
17 /* 最大缓存区大小 */
18 #define MAX_BUFFER_SIZE 5
19 /* epoll最大监听数 */
20 #define MAX_EPOLL_EVENTS 20
21 /* LT模式 */
22 #define EPOLL_LT 0
23 /* ET模式 */
24 #define EPOLL_ET 1
25 /* 文件描述符设置阻塞 */
26 #define FD_BLOCK 0
27 /* 文件描述符设置非阻塞 */
28 #define FD_NONBLOCK 1
29
30 /* 设置文件为非阻塞 */
31 int set_nonblock(int fd)
32 {
33     int old_flags = fcntl(fd, F_GETFL);
34     fcntl(fd, F_SETFL, old_flags | O_NONBLOCK);
35     return old_flags;
36 }
37
38 /* 注册文件描述符到epoll，并设置其事件为EPOLLIN(可读事件) */
39 void addfd_to_epoll(int epoll_fd, int fd, int epoll_type, int block_typ
40 {
41     struct epoll_event ep_event;
42     ep_event.data.fd = fd;
43     ep_event.events = EPOLLIN;
44
45     /* 如果是ET模式，设置EPOLLET */
46     if (epoll_type == EPOLL_ET)
47         ep_event.events |= EPOLLET;
48
49     /* 设置是否阻塞 */
50     if (block_type == FD_NONBLOCK)
51         set_nonblock(fd);
```

```
52
53      epoll_ctl(epoll_fd, EPOLL_CTL_ADD, fd, &ep_event);
54  }
55
56  /* LT处理流程 */
57  void epoll_lt(int sockfd)
58  {
59      char buffer[MAX_BUFFER_SIZE];
60      int ret;
61
62      memset(buffer, 0, MAX_BUFFER_SIZE);
63      printf("开始recv()...\n");
64      ret = recv(sockfd, buffer, MAX_BUFFER_SIZE, 0);
65      printf("ret = %d\n", ret);
66      if (ret > 0)
67          printf("收到消息:%s, 共%d个字节\n", buffer, ret);
68      else
69      {
70          if (ret == 0)
71              printf("客户端主动关闭！！！\n");
72          close(sockfd);
73      }
74
75      printf("LT处理结束！！！\n");
76  }
77
78  /* 带循环的ET处理流程 */
79  void epoll_et_loop(int sockfd)
80  {
81      char buffer[MAX_BUFFER_SIZE];
82      int ret;
83
84      printf("带循环的ET读取数据开始...\n");
85      while (1)
86      {
87          memset(buffer, 0, MAX_BUFFER_SIZE);
88          ret = recv(sockfd, buffer, MAX_BUFFER_SIZE, 0);
89          if (ret == -1)
90          {
91              if (errno == EAGAIN || errno == EWOULDBLOCK)
92              {
93                  printf("循环读完所有数据！！！\n");
94                  break;
95              }
96              close(sockfd);
97              break;
98          }
99          else if (ret == 0)
```

```
100         {
101             printf("客户端主动关闭请求！！！\n");
102             close(sockfd);
103             break;
104         }
105         else
106             printf("收到消息:%s, 共%d个字节\n", buffer, ret);
107     }
108     printf("带循环的ET处理结束！！！\n");
109 }
110
111
112 /* 不带循环的ET处理流程，比epoll_et_loop少了一个while循环 */
113 void epoll_et_nonloop(int sockfd)
114 {
115     char buffer[MAX_BUFFER_SIZE];
116     int ret;
117
118     printf("不带循环的ET模式开始读取数据...\n");
119     memset(buffer, 0, MAX_BUFFER_SIZE);
120     ret = recv(sockfd, buffer, MAX_BUFFER_SIZE, 0);
121     if (ret > 0)
122     {
123         printf("收到消息:%s, 共%d个字节\n", buffer, ret);
124     }
125     else
126     {
127         if (ret == 0)
128             printf("客户端主动关闭连接！！！\n");
129         close(sockfd);
130     }
131
132     printf("不带循环的ET模式处理结束！！！\n");
133 }
134
135 /* 处理epoll的返回结果 */
136 void epoll_process(int epollfd, struct epoll_event *events, int number,
137 {
138     struct sockaddr_in client_addr;
139     socklen_t client_addrlen;
140     int newfd, connfd;
141     int i;
142
143     for (i = 0; i < number; i++)
144     {
145         newfd = events[i].data.fd;
146         if (newfd == sockfd)
147         {
```

```
148             printf("==================================新一轮accept()=====
149             printf("accept()开始...\n");
150 
151             /* 休眠3秒，模拟一个繁忙的服务器，不能立即处理accept连接 */
152             printf("开始休眠3秒...\n");
153             sleep(3);
154             printf("休眠3秒结束！！！\n");
155 
156             client_addrlen = sizeof(client_addr);
157             connfd = accept(sockfd, (struct sockaddr *)&client_addr, &c
158             printf("connfd = %d\n", connfd);
159 
160             /* 注册已链接的socket到epoll，并设置是LT还是ET，是阻塞还是非阻塞 *
161             addfd_to_epoll(epollfd, connfd, epoll_type, block_type);
162             printf("accept()结束！！！\n");
163         }
164         else if (events[i].events & EPOLLIN)
165         {
166             /* 可读事件处理流程 */
167 
168             if (epoll_type == EPOLL_LT)
169             {
170                 printf("============================>水平触发开始...\n");
171                 epoll_lt(newfd);
172             }
173             else if (epoll_type == EPOLL_ET)
174             {
175                 printf("============================>边缘触发开始...\n");
176 
177                 /* 带循环的ET模式 */
178                 epoll_et_loop(newfd);
179 
180                 /* 不带循环的ET模式 */
181                 //epoll_et_nonloop(newfd);
182             }
183         }
184         else
185             printf("其他事件发生...\n");
186     }
187 }
188 
189 /* 出错处理 */
190 void err_exit(char *msg)
191 {
192     perror(msg);
193     exit(1);
194 }
195 
```

```
196 /* 创建socket */
197 int create_socket(const char *ip, const int port_number)
198 {
199     struct sockaddr_in server_addr;
200     int sockfd, reuse = 1;
201 
202     memset(&server_addr, 0, sizeof(server_addr));
203     server_addr.sin_family = AF_INET;
204     server_addr.sin_port = htons(port_number);
205 
206     if (inet_pton(PF_INET, ip, &server_addr.sin_addr) == -1)
207         err_exit("inet_pton() error");
208 
209     if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
210         err_exit("socket() error");
211 
212     /* 设置复用socket地址 */
213     if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reu
214         err_exit("setsockopt() error");
215 
216     if (bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_add
217         err_exit("bind() error");
218 
219     if (listen(sockfd, 5) == -1)
220         err_exit("listen() error");
221 
222     return sockfd;
223 }
224 
225 /* main函数 */
226 int main(int argc, const char *argv[])
227 {
228     if (argc < 3)
229     {
230         fprintf(stderr, "usage:%s ip_address port_number\n", argv[0]);
231         exit(1);
232     }
233 
234     int sockfd, epollfd, number;
235 
236     sockfd = create_socket(argv[1], atoi(argv[2]));
237     struct epoll_event events[MAX_EPOLL_EVENTS];
238 
239     /* linux内核2.6.27版的新函数，和epoll_create(int size)一样的功能，并去掉
240     if ((epollfd = epoll_create1(0)) == -1)
241         err_exit("epoll_create1() error");
242 
243     /* 以下设置是针对监听的sockfd，当epoll_wait返回时，必定有事件发生，
```

```
244      * 所以这里我们忽略罕见的情况外设置阻塞IO没意义，我们设置为非阻塞IO */
245
246     /* sockfd：非阻塞的LT模式 */
247     addfd_to_epoll(epollfd, sockfd, EPOLL_LT, FD_NONBLOCK);
248
249     /* sockfd：非阻塞的ET模式 */
250     //addfd_to_epoll(epollfd, sockfd, EPOLL_ET, FD_NONBLOCK);
251
252
253     while (1)
254     {
255         number = epoll_wait(epollfd, events, MAX_EPOLL_EVENTS, -1);
256         if (number == -1)
257             err_exit("epoll_wait() error");
258         else
259         {
260             /* 以下的LT，ET，以及是否阻塞都是是针对accept()函数返回的文件描述符
261
262             /* connfd:阻塞的LT模式 */
263             epoll_process(epollfd, events, number, sockfd, EPOLL_LT, FD
264
265             /* connfd:非阻塞的LT模式 */
266             //epoll_process(epollfd, events, number, sockfd, EPOLL_LT,
267
268             /* connfd:阻塞的ET模式 */
269             //epoll_process(epollfd, events, number, sockfd, EPOLL_ET,
270
271             /* connfd:非阻塞的ET模式 */
272             //epoll_process(epollfd, events, number, sockfd, EPOLL_ET,
273         }
274     }
275
276     close(sockfd);
277     return 0;
278 }
```

四.验证

1.验证水平触发的非阻塞sockfd，关键代码在247行。编译运行

```
yuu@yuukali:~/c_code/linux_program/2016_01/04$ gcc -W -Wall epoll_lt_et.c -o epoll_lt_et
yuu@yuukali:~/c_code/linux_program/2016_01/04$ ./epoll_lt_et 127.0.0.1 7788
```

代码里面休眠了3秒，模拟繁忙服务器不能很快处理accept()请求。这里，我们开另一个终端快速用5个连接连到服务器：

```
yuu@yuukali:~/c_code/linux_program/2016_01/04$ nc 127.0.0.1 7788 &
[1] 13118
yuu@yuukali:~/c_code/linux_program/2016_01/04$ nc 127.0.0.1 7788 &
[2] 13119

[1]+  Stopped                 nc 127.0.0.1 7788
yuu@yuukali:~/c_code/linux_program/2016_01/04$ nc 127.0.0.1 7788 &
[3] 13120

[2]+  Stopped                 nc 127.0.0.1 7788
yuu@yuukali:~/c_code/linux_program/2016_01/04$ nc 127.0.0.1 7788 &
[4] 13121

[3]+  Stopped                 nc 127.0.0.1 7788
yuu@yuukali:~/c_code/linux_program/2016_01/04$ nc 127.0.0.1 7788 &
[5] 13122

[4]+  Stopped                 nc 127.0.0.1 7788
yuu@yuukali:~/c_code/linux_program/2016_01/04$ jobs
[1]   Stopped                 nc 127.0.0.1 7788
[2]   Stopped                 nc 127.0.0.1 7788
[3]   Stopped                 nc 127.0.0.1 7788
[4]-  Stopped                 nc 127.0.0.1 7788
[5]+  Stopped                 nc 127.0.0.1 7788
yuu@yuukali:~/c_code/linux_program/2016_01/04$
```

我们再看看服务器的反映，可以看到5个终端连接都处理完成了，返回的新connfd依次为5,6,7,8,9：

```
yuu@yuukali:~/c_code/linux_program/2016_01/04$ gcc -W -Wall epoll_lt_et.c -o epoll_lt_et
yuu@yuukali:~/c_code/linux_program/2016_01/04$ ./epoll_lt_et 127.0.0.1 7788
==================================新一轮accept()==================================
accept()开始...
开始休眠3秒...
休眠3秒结束！！！
connfd = 5
accept()结束！！！
==================================新一轮accept()==================================
accept()开始...
开始休眠3秒...
休眠3秒结束！！！
connfd = 6
accept()结束！！！
==================================新一轮accept()==================================
accept()开始...
开始休眠3秒...
休眠3秒结束！！！
connfd = 7
accept()结束！！！
==================================新一轮accept()==================================
accept()开始...
开始休眠3秒...
休眠3秒结束！！！
connfd = 8
accept()结束！！！
==================================新一轮accept()==================================
accept()开始...
开始休眠3秒...
休眠3秒结束！！！
connfd = 9
accept()结束！！！
```

上面测试完毕后，我们批量kill掉那5个客户端，方便后面的测试：

```
1 $:for i in {1..5};do kill %$i;done
```

2.边缘触发的非阻塞sockfd，我们注释掉247行的代码，放开250行的代码。编译运行后，用同样的方法，快速创建5个客户端连接，或者测试5个后再测试10个。再看服务器的反映，5个客户端只处理了2个。说明高并发时，会出现客户端连接不上的问题：

```
yuu@yuukali:~/c_code/linux_program/2016_01/04$ gcc -W -Wall epoll_lt_et.c -o epoll_lt_et
yuu@yuukali:~/c_code/linux_program/2016_01/04$ ./epoll_lt_et 127.0.0.1 7788
==================================新一轮accept()==================================
accept()开始...
开始休眠3秒...
休眠3秒结束！！！
connfd = 5
accept()结束！！！
==================================新一轮accept()==================================
accept()开始...
开始休眠3秒...
休眠3秒结束！！！
connfd = 6
accept()结束！！！
```

3.水平触发的阻塞connfd，我们先把sockfd改回到水平触发，注释250行的代码，放开247行。重点代码在263行。

编译运行后，用一个客户端连接，并发送1-9这几个数：

```
yuu@yuukali:~/c_code/linux_program/2016_01/04$ nc 127.0.0.1 7788
123456789
```

再看服务器的反映，可以看到水平触发触发了2次。因为我们代码里面设置的缓冲区是5字节，处理代码一次接收不完，水平触发一直触发，直到数据全部读取完毕：

```
yuu@yuukali:~/c_code/linux_program/2016_01/04$ gcc -W -Wall epoll_lt_et.c -o epoll_lt_et
yuu@yuukali:~/c_code/linux_program/2016_01/04$ ./epoll_lt_et 127.0.0.1 7788
==================================新一轮accept()==================================
accept()开始...
开始休眠3秒...
休眠3秒结束！！！
connfd = 5
accept()结束！！！
============================>水平触发开始...
开始recv()...
ret = 5
收到消息:12345, 共5个字节
LT处理结束！！！
============================>水平触发开始...
开始recv()...
ret = 5
收到消息:6789
, 共5个字节
LT处理结束！！！
```

4.水平触发的非阻塞connfd。注释263行的代码，放开266行的代码。同上面那样测试，我们可以看到服务器反馈的消息跟上面测试一样。这里我就不再截图。

5.边缘触发的阻塞connfd，注释其他测试代码，放开269行的代码。先测试不带循环的ET模式(即不循环读取数据，跟水平触发读取一样)，注释178行的代码，放开181行的代码。

编译运行后，开启一个客户端连接，并发送1-9这几个数字，再看看服务器的反映，可以看到边缘触发只触发了一次，只读取了5个字节：

```
yuu@yuukali:~/c_code/linux_program/2016_01/04$ gcc -W -Wall epoll_lt_et.c -o epoll_lt_et
yuu@yuukali:~/c_code/linux_program/2016_01/04$ ./epoll_lt_et 127.0.0.1 7788
==================================新一轮accept()==================================
accept()开始...
开始休眠3秒...
休眠3秒结束！！！
connfd = 5
accept()结束！！！
============================>边缘触发开始...
不带循环的ET模式开始读取数据...
收到消息:12345, 共5个字节
不带循环的ET模式处理结束！！！
```

我们继续在刚才的客户端发送一个字符a，告诉epoll_wait()，有新的可读事件发生：

```
yuu@yuukali:~/c_code/linux_program/2016_01/04$ nc 127.0.0.1 7788
123456789
a
```

再看看服务器，服务器又触发了一次新的边缘触发，并继续读取上次没读完的6789加一个回车符：

```
yuu@yuukali:~/c_code/linux_program/2016_01/04$ gcc -W -Wall epoll_lt_et.c -o epoll_lt_et
yuu@yuukali:~/c_code/linux_program/2016_01/04$ ./epoll_lt_et 127.0.0.1 7788
==================================新一轮accept()==================================
accept()开始...
开始休眠3秒...
休眠3秒结束！！！
connfd = 5
accept()结束！！！
============================>边缘触发开始...
不带循环的ET模式开始读取数据...
收到消息:12345，共5个字节
不带循环的ET模式处理结束！！！
============================>边缘触发开始...
不带循环的ET模式开始读取数据...
收到消息:6789
，共5个字节
不带循环的ET模式处理结束！！！
```

这个时候，如果继续在刚刚的客户端再发送一个a，客户端这个时候就会读取上次没读完的a加上次的回车符，2个字节，还剩3个字节的缓冲区就可以读取本次的a加本次的回车符共4个字节：

```
yuu@yuukali:~/c_code/linux_program/2016_01/04$ gcc -W -Wall epoll_lt_et.c -o epoll_lt_et
yuu@yuukali:~/c_code/linux_program/2016_01/04$ ./epoll_lt_et 127.0.0.1 7788
==================================新一轮accept()==================================
accept()开始...
开始休眠3秒...
休眠3秒结束！！！
connfd = 5
accept()结束！！！
============================>边缘触发开始...
不带循环的ET模式开始读取数据...
收到消息:12345，共5个字节
不带循环的ET模式处理结束！！！
============================>边缘触发开始...
不带循环的ET模式开始读取数据...
收到消息:6789
，共5个字节
不带循环的ET模式处理结束！！！
============================>边缘触发开始...
不带循环的ET模式开始读取数据...
收到消息:a
a
，共4个字节
不带循环的ET模式处理结束！！！
```

我们可以看到，阻塞的边缘触发，如果不一次性读取一个事件上的数据，会干扰下一个事件！！！

接下来，我们就一次性读取数据，即带循环的ET模式。注意：我们这里测试的还是边缘触发的阻塞connfd，只是换个读取数据的方式。

注释181行代码，放开178的代码。编译运行，依然用一个客户端连接，发送1-9。看看服务器，可以看到数据全部读取完毕：

```
yuu@yuukali:~/c_code/linux_program/2016_01/04$ gcc -W -Wall epoll_lt_et.c -o epoll_lt_et
yuu@yuukali:~/c_code/linux_program/2016_01/04$ ./epoll_lt_et 127.0.0.1 7788
==================================新一轮accept()==================================
accept()开始...
开始休眠3秒...
休眠3秒结束！！！
connfd = 5
accept()结束！！！
============================>边缘触发开始...
带循环的ET读取数据开始...
收到消息:12345，共5个字节
收到消息:6789
，共5个字节
```

细心的朋友肯定发现了问题，程序没有输出"带循环的ET处理结束"，是因为程序一直卡在了88行的recv()函数上，因为是阻塞IO，如果没数据可读，它会一直等在那里，直到有数据可读。如果这个时候，用另一个客户端去连接，服务器不能受理这个新的客户端！！！

6.边缘触发的非阻塞connfd，不带循环的ET测试同上面一样，数据不会读取完。这里我们就只需要测试带循环的ET处理，即正规的边缘触发用法。注释其他测试代码，放开272行代码。编译运行，用一个客户端连接，并发送1-9。再观测服务器的反映，可以看到数据全部读取完毕，处理函数也退出了，因为非阻塞IO如果没有数据可读时，会立即返回，并设置error，这里我们根据EAGAIN和EWOULDBLOCK来判断数据全部读取完毕了，可以退出循环了：

```
yuu@yuukali:~/c_code/linux_program/2016_01/04$ gcc -W -Wall epoll_lt_et.c -o epoll_lt_et
yuu@yuukali:~/c_code/linux_program/2016_01/04$ ./epoll_lt_et 127.0.0.1 7788
==================================新一轮accept()==================================
accept()开始...
开始休眠3秒...
休眠3秒结束！！！
connfd = 5
accept()结束！！！
============================>边缘触发开始...
带循环的ET读取数据开始...
收到消息:12345，共5个字节
收到消息:6789
，共5个字节
循环读完所有数据！！！
带循环的ET处理结束！！！
```

这个时候，我们用另一个客户端去连接，服务器依然可以正常接收请求：

```
yuu@yuukali:~/c_code/linux_program/2016_01/04$ gcc -W -Wall epoll_lt_et.c -o epoll_lt_et
yuu@yuukali:~/c_code/linux_program/2016_01/04$ ./epoll_lt_et 127.0.0.1 7788
==================================新一轮accept()==================================
accept()开始...
开始休眠3秒...
休眠3秒结束！！！
connfd = 5
accept()结束！！！
============================>边缘触发开始...
带循环的ET读取数据开始...
收到消息:12345，共5个字节
收到消息:6789
，共5个字节
循环读完所有数据！！！
带循环的ET处理结束！！！
==================================新一轮accept()==================================
accept()开始...
开始休眠3秒...
休眠3秒结束！！！
connfd = 6
accept()结束！！！
```

五.总结

1.对于监听的sockfd，最好使用水平触发模式，边缘触发模式会导致高并发情况下，有的客户端会连接不上。如果非要使用边缘触发，网上有的方案是用while来循环accept()。

2.对于读写的connfd，水平触发模式下，阻塞和非阻塞效果都一样，不过为了防止特殊情况，还是建议设置非阻塞。

3.对于读写的connfd，边缘触发模式下，必须使用非阻塞IO，并要一次性全

部读写完数据。