<https://blog.csdn.net/weiyuefei/article/details/44004373>

在一个非阻塞的socket上调用read/write函数, 返回EAGAIN或者EWOULDBLOCK(注: EAGAIN就是EWOULDBLOCK)

从字面上看, 意思是:

\* EAGAIN: 再试一次

\* EWOULDBLOCK: 如果这是一个阻塞socket, 操作将被block

\* perror输出:  Resource temporarily unavailable

总结:

这个错误表示资源暂时不够, 可能read时, 读缓冲区没有数据, 或者, write时，写缓冲区满了 。

遇到这种情况, 如果是阻塞socket, read/write就要阻塞掉。

而如果是非阻塞socket, read/write立即返回-1, 同 时errno设置为EAGAIN.

所以, 对于阻塞socket, read/write返回-1代表网络出错了.

但对于非阻塞socket, read/write返回-1不一定网络真的出错了.

可能是Resource temporarily unavailable. 这时你应该再试, 直到Resource available.

综上, 对于non-blocking的socket,  正确的读写操作为:

读: 忽略掉errno = EAGAIN的错误, 下次继续读

写: 忽略掉errno = EAGAIN的错误, 下次继续写

对于select和epoll的LT模式, 这种读写方式是没有问题的. 但对于epoll的ET模式, 这种方式还有漏洞.

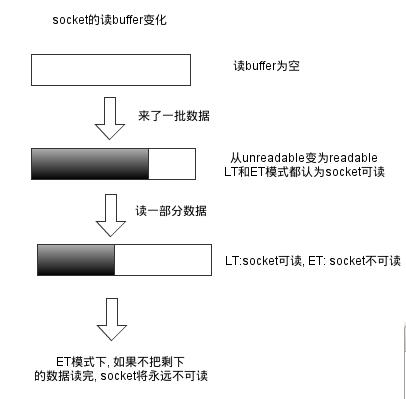
**epoll的两种模式 LT 和 ET**

二者的差异在于 level-trigger 模式下只要某个 socket 处于 readable/writable 状态，无论什么时候

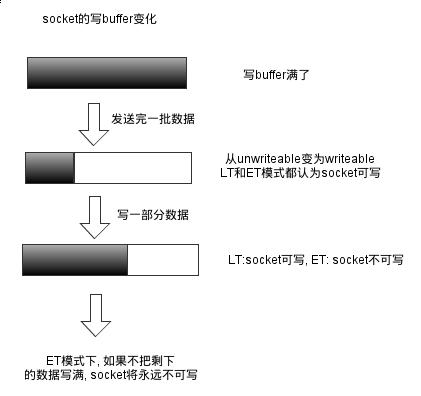
进行 epoll\_wait 都会返回该 socket；而 edge-trigger 模式下只有某个 socket 从 unreadable 变为 readable 或从

unwritable 变为 writable 时，epoll\_wait 才会返回该 socket。如下两个示意图:

从socket读数据:



往socket写数据



所以, 在epoll的ET模式下, 正确的读写方式为:

读: 只要可读, 就一直读, 直到返回0, 或者 errno = EAGAIN

写: 只要可写, 就一直写, 直到数据发送完, 或者 errno = EAGAIN

正确的读:

1. n = 0;
2. while ((nread = read(fd, buf + n, BUFSIZ-1)) > 0) {
3. n += nread;
4. }
5. if (nread == -1 && errno != EAGAIN) {
6. perror("read error");
7. }

 正确的写:

1. int nwrite, data\_size = strlen(buf);
2. n = data\_size;
3. while (n > 0) {
4. nwrite = write(fd, buf + data\_size - n, n);
5. if (nwrite < n) {
6. if (nwrite == -1 && errno != EAGAIN) {
7. perror("write error");
8. }
9. break;
10. }
11. n -= nwrite;
12. }

正确的accept，accept 要考虑 2 个问题

(1) 阻塞模式 accept 存在的问题

考虑这种情况： TCP 连接被客户端夭折，即在服务器调用 accept 之前，客户端主动发送 RST 终止

连接，导致刚刚建立的连接从就绪队列中移出，如果套接口被设置成阻塞模式，服务器就会一直阻塞

在 accept 调用上，直到其他某个客户建立一个新的连接为止。但是在此期间，服务器单纯地阻塞在

accept 调用上，就绪队列中的其他描述符都得不到处理.

解决办法是把监听套接口设置为非阻塞，当客户在服务器调用 accept 之前中止某个连接时，accept 调用

可以立即返回 -1， 这时源自 Berkeley 的实现会在内核中处理该事件，并不会将该事件通知给 epool，

而其他实现把 errno 设置为 ECONNABORTED 或者 EPROTO 错误，我们应该忽略这两个错误。

(2) ET 模式下 accept 存在的问题

考虑这种情况：多个连接同时到达，服务器的 TCP 就绪队列瞬间积累多个就绪连接，由于是边缘触发模式，

 epoll 只会通知一次，accept 只处理一个连接，导致 TCP 就绪队列中剩下的连接都得不到处理。

 解决办法是用 while 循环抱住 accept 调用，处理完 TCP 就绪队列中的所有连接后再退出循环。如何知道

 是否处理完就绪队列中的所有连接呢？ accept  返回 -1 并且 errno 设置为 EAGAIN 就表示所有连接都处理完。

综合以上两种情况，服务器应该使用非阻塞地 accept， accept 在 ET 模式下 的正确使用方式为：

1. while ((conn\_sock = accept(listenfd,(struct sockaddr \*) &remote,
2. (size\_t \*)&addrlen)) > 0)
3. {
4. handle\_client(conn\_sock);
5. }
6. if (conn\_sock == -1) {
7. if (errno != EAGAIN && errno != ECONNABORTED
8. && errno != EPROTO && errno != EINTR)
9. perror("accept");
10. }

一道腾讯后台开发的面试题

使用Linux epoll模型，水平触发模式；当socket可写时，会不停的触发 socket 可写的事件，如何处理？

第一种最普遍的方式：

需要向 socket 写数据的时候才把 socket 加入 epoll ，等待可写事件。

接受到可写事件后，调用 write 或者 send 发送数据。。。

当所有数据都写完后，把 socket 移出 epoll。

这种方式的缺点是，即使发送很少的数据，也要把 socket 加入 epoll，写完后在移出 epoll，有一定操作代价。

一种改进的方式：

开始不把 socket 加入 epoll，需要向 socket 写数据的时候，直接调用 write 或者 send 发送数据。

如果返回 EAGAIN，把 socket 加入 epoll，在 epoll 的驱动下写数据，全部数据发送完毕后，再移出 epoll。

这种方式的优点是：数据不多的时候可以避免 epoll 的事件处理，提高效率。

最后贴一个使用epoll, ET模式的简单HTTP服务器代码:

1. #include <sys/socket.h>
2. #include <sys/wait.h>
3. #include <netinet/in.h>
4. #include <netinet/tcp.h>
5. #include <sys/epoll.h>
6. #include <sys/sendfile.h>
7. #include <sys/stat.h>
8. #include <unistd.h>
9. #include <stdio.h>
10. #include <stdlib.h>
11. #include <string.h>
12. #include <strings.h>
13. #include <fcntl.h>
14. #include <errno.h>
16. #define MAX\_EVENTS 10
17. #define PORT 8080
19. //设置socket连接为非阻塞模式
20. void setnonblocking(int sockfd) {
21. int opts;
23. opts = fcntl(sockfd, F\_GETFL);
24. if(opts < 0) {
25. perror("fcntl(F\_GETFL)\n");
26. exit(1);
27. }
28. opts = (opts | O\_NONBLOCK);
29. if(fcntl(sockfd, F\_SETFL, opts) < 0) {
30. perror("fcntl(F\_SETFL)\n");
31. exit(1);
32. }
33. }
35. int main(){
36. struct epoll\_event ev, events[MAX\_EVENTS];
37. int addrlen, listenfd, conn\_sock, nfds, epfd, fd, i, nread, n;
38. struct sockaddr\_in local, remote;
39. char buf[BUFSIZ];
41. //创建listen socket
42. if( (listenfd = socket(AF\_INET, SOCK\_STREAM, 0)) < 0) {
43. perror("sockfd\n");
44. exit(1);
45. }
46. setnonblocking(listenfd);
47. bzero(&local, sizeof(local));
48. local.sin\_family = AF\_INET;
49. local.sin\_addr.s\_addr = htonl(INADDR\_ANY);;
50. local.sin\_port = htons(PORT);
51. if( bind(listenfd, (struct sockaddr \*) &local, sizeof(local)) < 0) {
52. perror("bind\n");
53. exit(1);
54. }
55. listen(listenfd, 20);
57. epfd = epoll\_create(MAX\_EVENTS);
58. if (epfd == -1) {
59. perror("epoll\_create");
60. exit(EXIT\_FAILURE);
61. }
63. ev.events = EPOLLIN;
64. ev.data.fd = listenfd;
65. if (epoll\_ctl(epfd, EPOLL\_CTL\_ADD, listenfd, &ev) == -1) {
66. perror("epoll\_ctl: listen\_sock");
67. exit(EXIT\_FAILURE);
68. }
70. for (;;) {
71. nfds = epoll\_wait(epfd, events, MAX\_EVENTS, -1);
72. if (nfds == -1) {
73. perror("epoll\_pwait");
74. exit(EXIT\_FAILURE);
75. }
77. for (i = 0; i < nfds; ++i) {
78. fd = events[i].data.fd;
79. if (fd == listenfd) {
80. while ((conn\_sock = accept(listenfd,(struct sockaddr \*) &remote,
81. (size\_t \*)&addrlen)) > 0) {
82. setnonblocking(conn\_sock);
83. ev.events = EPOLLIN | EPOLLET;
84. ev.data.fd = conn\_sock;
85. if (epoll\_ctl(epfd, EPOLL\_CTL\_ADD, conn\_sock,
86. &ev) == -1) {
87. perror("epoll\_ctl: add");
88. exit(EXIT\_FAILURE);
89. }
90. }
91. if (conn\_sock == -1) {
92. if (errno != EAGAIN && errno != ECONNABORTED
93. && errno != EPROTO && errno != EINTR)
94. perror("accept");
95. }
96. continue;
97. }
98. if (events[i].events & EPOLLIN) {
99. n = 0;
100. while ((nread = read(fd, buf + n, BUFSIZ-1)) > 0) {
101. n += nread;
102. }
103. if (nread == -1 && errno != EAGAIN) {
104. perror("read error");
105. }
106. ev.data.fd = fd;
107. ev.events = events[i].events | EPOLLOUT;
108. if (epoll\_ctl(epfd, EPOLL\_CTL\_MOD, fd, &ev) == -1) {
109. perror("epoll\_ctl: mod");
110. }
111. }
112. if (events[i].events & EPOLLOUT) {
113. sprintf(buf, "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\nHello World", 11);
114. int nwrite, data\_size = strlen(buf);
115. n = data\_size;
116. while (n > 0) {
117. nwrite = write(fd, buf + data\_size - n, n);
118. if (nwrite < n) {
119. if (nwrite == -1 && errno != EAGAIN) {
120. perror("write error");
121. }
122. break;
123. }
124. n -= nwrite;
125. }
126. close(fd);
127. }
128. }
129. }
131. return 0;
132. }