<https://blog.csdn.net/weiyuefei/article/details/52128556>

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <sys/types.h>
5. #include <sys/socket.h>
6. #include <netdb.h>
7. #include <unistd.h>
8. #include <fcntl.h>
9. #include <sys/epoll.h>
10. #include <errno.h>
12. #define MAXEVENTS 64
14. static int
15. make\_socket\_non\_blocking (int sfd)
16. {
17. int flags, s;
19. flags = fcntl (sfd, F\_GETFL, 0);
20. if (flags == -1)
21. {
22. perror ("fcntl");
23. return -1;
24. }
26. flags |= O\_NONBLOCK;
27. s = fcntl (sfd, F\_SETFL, flags);
28. if (s == -1)
29. {
30. perror ("fcntl");
31. return -1;
32. }
34. return 0;
35. }
37. static int
38. create\_and\_bind (char \*port)
39. {
40. struct addrinfo hints;
41. struct addrinfo \*result, \*rp;
42. int s, sfd;
44. memset (&hints, 0, sizeof (struct addrinfo));
45. hints.ai\_family = AF\_UNSPEC; /\* Return IPv4 and IPv6 choices \*/
46. hints.ai\_socktype = SOCK\_STREAM; /\* We want a TCP socket \*/
47. hints.ai\_flags = AI\_PASSIVE; /\* All interfaces \*/
49. s = getaddrinfo (NULL, port, &hints, &result);
50. if (s != 0)
51. {
52. fprintf (stderr, "getaddrinfo: %s\n", gai\_strerror (s));
53. return -1;
54. }
56. for (rp = result; rp != NULL; rp = rp->ai\_next)
57. {
58. sfd = socket (rp->ai\_family, rp->ai\_socktype, rp->ai\_protocol);
59. if (sfd == -1)
60. continue;
62. s = bind (sfd, rp->ai\_addr, rp->ai\_addrlen);
63. if (s == 0)
64. {
65. /\* We managed to bind successfully! \*/
66. break;
67. }
69. close (sfd);
70. }
72. if (rp == NULL)
73. {
74. fprintf (stderr, "Could not bind\n");
75. return -1;
76. }
78. freeaddrinfo (result);
80. return sfd;
81. }
83. int
84. main (int argc, char \*argv[])
85. {
86. int sfd, s;
87. int efd;
88. struct epoll\_event event;
89. struct epoll\_event \*events;
91. if (argc != 2)
92. {
93. fprintf (stderr, "Usage: %s [port]\n", argv[0]);
94. exit (EXIT\_FAILURE);
95. }
97. sfd = create\_and\_bind (argv[1]);
98. if (sfd == -1)
99. abort ();
101. s = make\_socket\_non\_blocking (sfd);
102. if (s == -1)
103. abort ();
105. s = listen (sfd, SOMAXCONN);
106. if (s == -1)
107. {
108. perror ("listen");
109. abort ();
110. }
112. efd = epoll\_create1 (0);
113. if (efd == -1)
114. {
115. perror ("epoll\_create");
116. abort ();
117. }
119. event.data.fd = sfd;
120. event.events = EPOLLIN | EPOLLET;
121. s = epoll\_ctl (efd, EPOLL\_CTL\_ADD, sfd, &event);
122. if (s == -1)
123. {
124. perror ("epoll\_ctl");
125. abort ();
126. }
128. /\* Buffer where events are returned \*/
129. events = calloc (MAXEVENTS, sizeof event);
131. /\* The event loop \*/
132. while (1)
133. {
134. int n, i;
136. n = epoll\_wait (efd, events, MAXEVENTS, -1);
137. for (i = 0; i < n; i++)
138. {
139. if ((events[i].events & EPOLLERR) ||
140. (events[i].events & EPOLLHUP) ||
141. (!(events[i].events & EPOLLIN)))
142. {
143. /\* An error has occured on this fd, or the socket is not
144. ready for reading (why were we notified then?) \*/
145. fprintf (stderr, "epoll error\n");
146. close (events[i].data.fd);
147. continue;
148. }
150. else if (sfd == events[i].data.fd)
151. {
152. /\* We have a notification on the listening socket, which
153. means one or more incoming connections. \*/
154. while (1)
155. {
156. struct sockaddr in\_addr;
157. socklen\_t in\_len;
158. int infd;
159. char hbuf[NI\_MAXHOST], sbuf[NI\_MAXSERV];
161. in\_len = sizeof in\_addr;
162. infd = accept (sfd, &in\_addr, &in\_len);
163. if (infd == -1)
164. {
165. if ((errno == EAGAIN) ||
166. (errno == EWOULDBLOCK))
167. {
168. /\* We have processed all incoming
169. connections. \*/
170. break;
171. }
172. else
173. {
174. perror ("accept");
175. break;
176. }
177. }
179. s = getnameinfo (&in\_addr, in\_len,
180. hbuf, sizeof hbuf,
181. sbuf, sizeof sbuf,
182. NI\_NUMERICHOST | NI\_NUMERICSERV);
183. if (s == 0)
184. {
185. printf("Accepted connection on descriptor %d "
186. "(host=%s, port=%s)\n", infd, hbuf, sbuf);
187. }
189. /\* Make the incoming socket non-blocking and add it to the
190. list of fds to monitor. \*/
191. s = make\_socket\_non\_blocking (infd);
192. if (s == -1)
193. abort ();
195. event.data.fd = infd;
196. event.events = EPOLLIN | EPOLLET;
197. s = epoll\_ctl (efd, EPOLL\_CTL\_ADD, infd, &event);
198. if (s == -1)
199. {
200. perror ("epoll\_ctl");
201. abort ();
202. }
203. }
204. continue;
205. }
206. else
207. {
208. /\* We have data on the fd waiting to be read. Read and
209. display it. We must read whatever data is available
210. completely, as we are running in edge-triggered mode
211. and won't get a notification again for the same
212. data. \*/
213. int done = 0;
215. while (1)
216. {
217. ssize\_t count;
218. char buf[512];
220. count = read (events[i].data.fd, buf, sizeof buf);
221. if (count == -1)
222. {
223. /\* If errno == EAGAIN, that means we have read all
224. data. So go back to the main loop. \*/
225. if (errno != EAGAIN)
226. {
227. perror ("read");
228. done = 1;
229. }
230. break;
231. }
232. else if (count == 0)
233. {
234. /\* End of file. The remote has closed the
235. connection. \*/
236. done = 1;
237. break;
238. }
240. /\* Write the buffer to standard output \*/
241. s = write (1, buf, count);
242. if (s == -1)
243. {
244. perror ("write");
245. abort ();
246. }
247. }
249. if (done)
250. {
251. printf ("Closed connection on descriptor %d\n",
252. events[i].data.fd);
254. /\* Closing the descriptor will make epoll remove it
255. from the set of descriptors which are monitored. \*/
256. close (events[i].data.fd);
257. }
258. }
259. }
260. }
262. free (events);
264. close (sfd);
266. return EXIT\_SUCCESS;
267. }