Unix system

my_irc Practical

Contact b-psu-330@epitech.eu

Abstract:
The goal of this practical is to show you how the `select` *syscall works.*
An optimal use of this syscall will be needed for the next projects to come, like `my_irc` *and the* `zappy`*.*

Contents

.1 Step 1 : Generalization

We will begin by generalizing the use of `select(2)` on clients sockets, but also on those of the server.

We have seen that when select(2) indicates that a data is available for reading on the client socket, `read(2)` is nonblocking.
But what happens if we use `select(2)` on the server's socket?

As you might guess, we never have to read any data on a server socket, we just use `accept(2)`. That's precisely to avoid a blocking call of this function that `select(2)` will be useful.
Indeed, when a new connection request comes in, `select(2)` flags the server's filedescriptor as having available data to read, and then it gives the control back to the program. From then, we can call `accept(2)` because we know that it won't be blocking.

An example code is available on the next page.

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/time.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  int s;
  int cs;
  struct sockaddr_in sin;
  struct sockaddr_in client_sin;
  int client_sin_len;
  int port;

  s = socket(PF_INET, SOCK_STREAM, 0);
  port = atoi(argv[1]);
  sin.sin_family = AF_INET;
  sin.sin_port = htons(port);
  sin.sin_addr.s_addr = INADDR_ANY;
  bind(s, (struct sockaddr*)&sin, sizeof(sin));
  listen(s, 42);
  client_sin_len = sizeof(client_sin);
  while (1)
    {
      fd_set readf;
      struct timeval tv;

      tv.tv_sec = 5;
      tv.tv_usec = 0;
      FD_ZERO(&readf);
      FD_SET(s, &readf);
      if (select(s + 1, &readf, NULL, NULL, &tv) == -1)
        perror("select");
      if (FD_ISSET(s, &readf))
    {
      printf("New client\n");
      cs = accept(s, (struct sockaddr *)&client_sin, &client_sin_len);
      /*...*/
      close(cs);
    }
      else
    {
      printf("waiting...\n");
    }
    }
  close(s);
  return (0);
```

```
51 }
```

Let's test the test program:

```
$> ./server 4242
waiting...
waiting...
waiting...
New client
^C
$>
```

```
$> telnet 127.0.0.1 4242
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
^]
$>
```

.2 Step 2 : Applying this concept to N clients

To be able to handle a variable client number, we will use an array which we will fill with the client's filedescriptors.
We'll also generalize the use of this array with the servers' filedescriptors. To know what each filedescriptor is, we'll affect flags to each of them:

- FD_FREE for an unused fd
- FD_CLIENT for a client fd
- FD_SERVER for a server fd

To simplify the code, we'll handle all the filedescriptors in the same way, whether its a client or a server.
However, the way of handling them is different, that's why we'll use function pointers corresponding to the filedescriptor's type.

An example code is available in the next page:

```
1  #include <sys/types.h>
2  #include <sys/socket.h>
3  #include <netinet/in.h>
4  #include <arpa/inet.h>
5  #include <sys/time.h>
6  #include <unistd.h>
7  #include <stdio.h>
8  #include <string.h>
9
10 #define FD_FREE 0
11 #define FD_CLIENT 1
12 #define FD_SERVER 2
13
14 #define MAX_FD 255
15
16 typedef void(*fct)();
17
18 typedef struct s_env
19 {
20   char fd_type[MAX_FD];
21   fct fct_read[MAX_FD];
22   fct fct_write[MAX_FD];
23   int port;
24 }t_env;
25
26 void client_read(t_env *e, int fd)
27 {
28   int r;
29   char buf[4096];
30
31   r = read(fd, buf, 4096);
32   if (r > 0)
33     {
34       buf[r] = '\0';
35       printf("%d: %s\n", fd, buf);
36     }
37   else
38     {
39       printf("%d: Connection closed\n");
40       close(fd);
41       e->fd_type[fd] = FD_FREE;
42     }
43 }
44
45 void add_client(t_env *e, int s)
46 {
47   int   cs;
48   struct sockaddr_in client_sin;
49   int client_sin_len;
50
```

```
51   client_sin_len = sizeof(client_sin);
52   cs = accept(s, (struct sockaddr *)&client_sin, &client_sin_len);
53   e->fd_type[cs] = FD_CLIENT;
54   e->fct_read[cs] = client_read;
55   e->fct_write[cs] = NULL;
56 }
57
58 void server_read(t_env *e, int fd)
59 {
60   printf("New client\n");
61   add_client(e, fd);
62 }
63
64 void add_server(t_env *e)
65 {
66   int s;
67   struct sockaddr_in sin;
68
69   s = socket(PF_INET, SOCK_STREAM, 0);
70   sin.sin_family = AF_INET;
71   sin.sin_port = htons(e->port);
72   sin.sin_addr.s_addr = INADDR_ANY;
73   bind(s, (struct sockaddr*)&sin, sizeof(sin));
74   listen(s, 42);
75   e->fd_type[s] = FD_SERVER;
76   e->fct_read[s] = server_read;
77   e->fct_write[s] = NULL;
78 }
79
80 int main(int argc, char **argv)
81 {
82   t_env   e;
83   int   i, fd_max;
84   fd_set  fd_read;
85   struct timeval tv;
86
87   memset(e.fd_type, FD_FREE, MAX_FD);
88   e.port = atoi(argv[1]);
89   add_server(&e);
90   tv.tv_sec = 20;
91   tv.tv_usec = 0;
92   while (1)
93     {
94       FD_ZERO(&fd_read);
95       fd_max = 0;
96       for (i = 0; i < MAX_FD; i++)
97     if (e.fd_type[i] != FD_FREE)
98       {
99         FD_SET(i, &fd_read);
100        fd_max = i;
101      }
```

```
102       if (select(fd_max + 1, &fd_read, NULL, NULL, &tv) == -1)
103     perror("select");
104       for (i = 0; i < MAX_FD; i++)
105     if (FD_ISSET(i, &fd_read))
106       e.fct_read[i](&e, i);
107       printf("waiting...\n");
108     }
109   return (0);
110 }
```

From this example:

- Write a tiny irc that returns to all the clients a message sent by one client
- Then, modify it to make your server works with 2 simultaneous ports
- Add using `select(2)` the verification that we can write on a filedescriptor before doing it, in order to make your server completely non-blocking whatever happens.

Thank you for following this practical and good luck for your project!