Event-Driven Chat Server

In this exercise you will implement an event-driven *chat* server. The function of the chat is to forward each incoming message over all client connections (i.e., to all clients) except for the client connection over which the message was received. The challenge in such a server lies in implementing this behavior in an entirely event-driven manner (no threads).

In this exercise you'll use "select" to check from which socket descriptor is ready for read or write. You should call "select" inside a loop, but it should appear only once in your exercise.

When checking the read set, you should distinguish the main socket, which gets new connections, from the other sockets, that negotiate data with the clients.

When the main socket is ready for reading, you should call accept, and when any other socket is ready for reading you should call read or recv.

You'll maintain a queue for each connection. This queue will hold all msgs that must be written on that connection.

When checking the write set, note that if a socket descriptor is ready for writing, it means that we can write once to the socket without the risk to be blocked. Since we would like to write all msgs in queue to the socket descriptor, we will set the socket to be non-blocking.

In order to make a socket with descriptor "fd" to be non-blocking, you can use ioctl function:

```
int on = 1;
rc = ioctl(fd, (int)FIONBIO, (char *)&on)
```

Message Handling

Client messages are stored within `struct msg` objects. Each time the server reads a message from one client, it adds a msg object to all other clients. See chatServer.h and chatServer.c for details.
You can assume that at each read operation from a client socket, a full line is read (until the \r\n). Once one line was read, you'll not try to read immediately another line. If there is more to read from the client, you'll read it after the next call to select. Note that a socket should be added to the write set only if there is something to write to the socket.

Connection Handling

Your chat server will listen at a specified port number for incoming connections. The port is given as a parameter to the program. When a new connection request arrives from some client, the server accepts the connection and creates a new `struct conn` connection object for that client. See chatServer.h and chatServer.c for details.

If the server ends in CTRL-C, you should catch the SIG_INT signal, clean everything and exit.

Output:

Just before calling to select, print:

```
printf("Waiting on select()...\nMaxFd %d\n", pool->maxfd);
```

when accepting a new connection with descriptor “sd”, print:

```
printf("New incoming connection on sd %d\n", sd);
```

before reading data from socket descriptor “sd”, print:

```
printf("Descriptor %d is readable\n", sd);
```

after reading data of size “len” from socket descriptor “sd”, print:

```
printf("%d bytes received from sd %d\n", len, sd);
```

when connection from a client is closed by the client, print:

```
printf("Connection closed for sd %d\n",sd);
```

when the server removes connection from any reason (either the client closes the connection, or the server closes the client’s connection), print:

```
printf("removing connection with sd %d \n", sd);
```

Error Handling

For each failure until select is called, use perror and exit the server.

If select fails, exit the loop, clean the memory and exit.

For any other error, skip the operation and continue. I.e., if read from a socket or write to socket fail, do nothing, just continue, and skip this read or write operations.