View Socket.h (which contains the 3 implementations under a class called eventManager) from the root directory of this repository

Epoll Implementation

Implementing Epoll for event based servers involves initializing the epoll instance then setting it to monitor your server socket which you will be using to accept client connections. You do this by creating an epoll_event struct for your server socket then calling epoll_ctl to add the struct to be monitored. Next you need to create a structure to hold your client events once they are connected, you malloc an epoll_event array as large as the number of events you would allow at a time.

To monitor a client you add configure an epoll_event struct to the array mentioned earlier then tell epoll to notify you of activity by adding it to the monitoring using epoll_ctl. Similarly you can use epol_ctl to stop monitoring the socket using a different control value in the second parameter.

Waiting for an event involves calling epoll_wait then distinguishing between incoming data events and new connection events. You can do this by reading the file descriptor linked to the event, if it is the server socket, the event is a new connection, if it is one of the client sockets, there is new data to be read from that client.

When you exit you must free your epoll structure as well as your event array.

Poll Implementation

To use poll, you use a different structure called a pollfd. You construct this structure for each of the file descriptors you wish to monitor. To start off you need to add the server fd by declaring a pollfd variable, setting the .fd variable to your server socket, then specify the types of events you want to monitor, I believe it is sufficient to use POLLIN to monitor for incoming data or connections.

You need to create a vector of pollfd structures to hold all of the sockets you are monitoring. Use .push_back(pollfd) to add a new struct to the vector. You can use poll to scan the vector for activity until an event occurs. The fd corresponding to the event will be returned by the poll function. Again if the event corresponds to the server socket it is a new connection attempt, otherwise it is incoming data from a client.

To stop monitoring a socket you can use the erase function to remove it from the vector. In this implementation no data needs to be manually allocated for freed.

Select Implementation

To use Select, you need a fd_set structure and you use FD_SET function to add sockets to the struct. Just like before you need to add the server socket first then monitor it for incoming connections before adding new client fds to the structure. To check for events you use the select function on the set and you have to manually provide the number of sockets in the set to the function. The function is non-blocking and will return -1 if no event is available. So you must use

a while loop to continually call the function until an event is ready. Furthermore, you must re-add all client and server socket to the set after each use of the select function. Overall this is the least friendly and efficient method by far.