Lab session 3

Sockets - Non-blocking operations

Receiving data from a socket is typically a blocking operation. Usually, if there is no data received by the socket, the thread calling *recv* blocks until there is something to be read. In the same way, sending data, accepting new connections (TCP protocol) and connecting to a remote socket (also TCP protocol), can also be blocking actions if the socket is not ready to execute such operations. Blocking the application loop is not optimal for an application that requires real-time performance, as it is the case in video games, where the frame rate must be high. Imagine a server with 5 connected clients. If the server calls *recv* on one of its sockets connected to a client and there is nothing received, this will block until some data arrives from the client. This avoids the server using this valuable time to perform other operations such as consulting other sockets, accepting new connections, or running the game simulation.

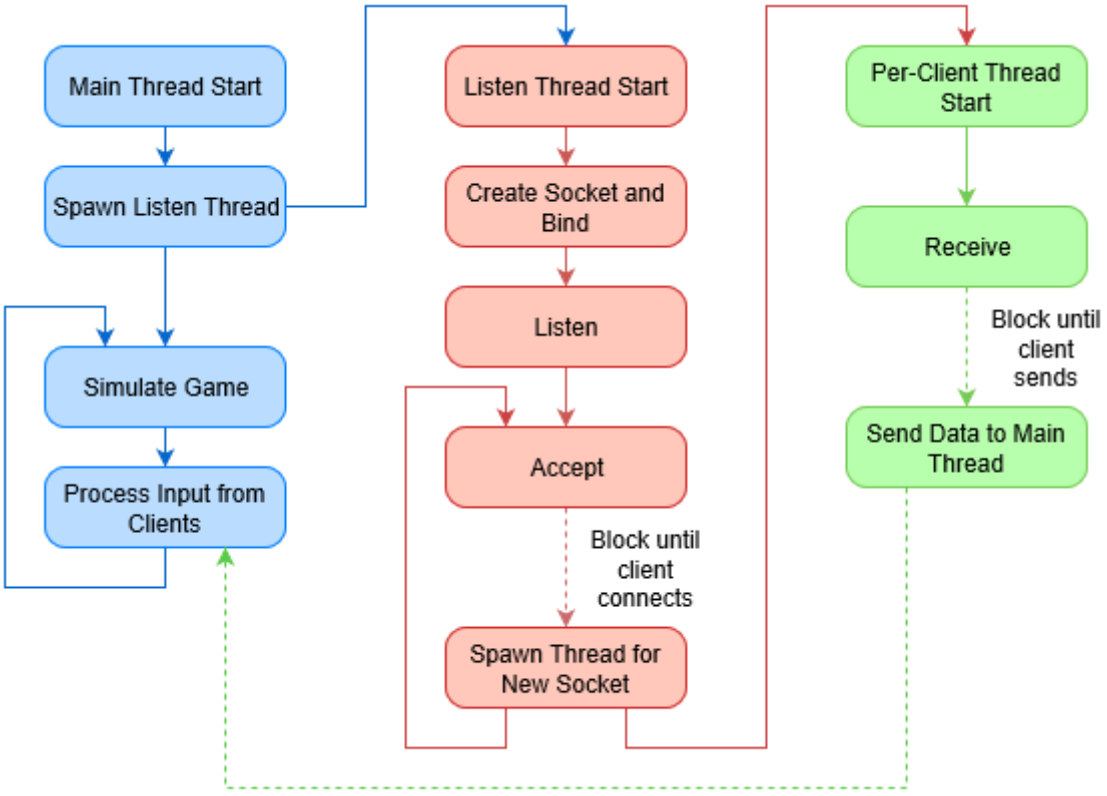
Luckily, we have three common ways to solve this problem:

1. Multithreading
2. Non-blocking I/O
3. *select()* function

Table of contents

Multithreading

One of the techniques to solve blocking in sockets is by putting each potentially blocking call in its own thread. In the previously mentioned example, the server would need at least 7 threads: one for each client connection, another for the listen socket accepting new incoming connections, and another more for the game simulation. The following image strives to picture the process:



This solution works, although it is not quite scalable when the number of client connections is too high. It can also be difficult to manage because data is received in parallel to other threads that can need this data, and for this reason, data needs to be transferred among threads in a safe way. There are several problems derived from this architecture. Of course, all of them can be solved, but as we will see in the following section, there are simpler alternatives.

Non-blocking I/O

By default, sockets are configured in blocking mode. However, they can be configured in **non-blocking mode** to avoid blocking. When a socket has been configured in non-blocking mode, function calls that should block (e.g. calling *recv* if there is no data to receive) will return *SOCKET_ERROR*, and (recalling the first class) *WSAGetLastError()* will return the more specific error code *WSAEWOULDBLOCK*.

The following code configures the socket *s* to perform in non-blocking mode.

```
// Set non-blocking socket
u_long nonBlocking = 1;
res = ioctlsocket(s, FIONBIO, &nonBlocking);
if (res == SOCKET_ERROR) {
 // Log socket error and exit
}
```

After configuring the socket as non-blocking, we can invoke functions being sure that they will return immediately without blocking. We will have to check which error code *WSAGetLastError()* returned just to be sure that it was *WSAEWOULDBLOCK* and not other error code.

```
// Recv
int bytesRecv = recv(s, inputBuffer, inputBufferLen, 0);
if (bytesRecv == SOCKET_ERROR)
{
 int lastError = WSAGetLastError();
 if (lastError == WSAEWOULDBLOCK) {
  // Do nothing special, there was no data to receive
 }
 else {
  // Other error handling
 }
}
else // Success
{
 // Process received data
}
```

Consulting sockets this way is simple and direct. The previous example can be appropriate for programs that work with a unique socket (or with few sockets). However, when the amount of sockets to manage grows, this method may be a bit inefficient. The method explained in the following section solves this inefficiency issue.

NOTE: Use non-blocking sockets only when we are dealing with one or very few sockets (e.g. client applications usually have only one single socket connected to the server).

select() function

The alternative method to consult the available operations (those that will not block) on a set of sockets is the *select* function:

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, const timeval *timeout);
```

Nfds, in Windows platforms, does not mean anything and can be ignored.

Readfds, *writefds,* and *exceptfds* are input/output parameters, so they will provide input data to the function, but they will also be modified to provide an output when necessary.

Readfds is a pointer to a collection of sockets. All sockets contained in this collection will be verified to know whether or not they have an available reading operation (*recv* or *accept*, for instance). The *select* function, after returning, will have removed from *readfds* all those sockets without any available reading operations, ensuring that the following read operation on the remaining sockets will not be blocking.

Writefds is a pointer to a collection of sockets. Select will verify which sockets in this collection allow a writing operation (*send*, for instance) without blocking. Those sockets remaining in this collection after the execution of *select* are guaranteed not to block if a writing operation like *send* is invoked.

Exceptfds is a pointer to a collection of sockets. Only sockets with pending exceptions remain in the collection after the execution of *select*.

NOTE: We can pass *nullptr* to any of the three previous parameters to indicate that we do not need to query that kind of operation on any socket.

Timeout is a timer that allows controlling how long *select()* will wait until returning. If passing nullptr, the function will never return and will remain blocked until some socket has any of the required operations available. To avoid blocking and return immediately, set the *timeout* values to 0, indicating that we do not want to wait.

Furthermore, the *select* function returns the total number of sockets available with some of the required operations. If the *timeout* expires, the returned value will be 0, meaning that there are not available operations on any of the input sockets.

To create a collection of sockets:

```
fd_set readSet;
FD_ZERO(&readSet);
```

To add a socket to the collection:

```
FD_SET(socket, &readSet);
```

To check if a socket is still contained in the collection after the execution of *select*:

```
FD_ISSET(socket, &readSet);
```

The following code uses *select* to perform a query over a set of sockets to know which ones of them are prepared to be read without blocking, whether they are *listen sockets* with new incoming connections (TCP-only), or *sockets* with incoming data.

```
// New socket set
fd_set readfds;
FD_ZERO(&readfds);

// Fill the set
for (auto s : sockets) {
 FD_SET(s, &readfds);
}

// Timeout (return immediately)
struct timeval timeout;
timeout.tv_sec = 0;
timeout.tv_usec = 0;

// Select (check for readability)
int res = select(0, &readfds, nullptr, nullptr, &timeout);
if (res == SOCKET_ERROR) {
 logSocketErrorAndExit("select 4 read");
}

// Fill this array with disconnected sockets
std::list<SOCKET> disconnectedSockets;

// Read selected sockets
for (auto s : sockets)
{
 if (FD_ISSET(s, &readfds)) {
  if (s == serverSocket) { // Is the server socket
   // Accept stuff
  } else { // Is a client socket
   // Recv stuff
  }
 }
}

// Remove all disconnectedSockets from our list (sockets)
// TODO
```

In this example we can see how all sockets (see the variable *sockets* in the code) are initially stored in a c++ std collection (*std::vector<SOCKET>*, for instance). By iterating over this collection, we can fill collections of the type *fd_set*, which is the type needed by the *select* function.

NOTE: The *select* function should be called only when we have at least one socket to query. Calling *select* without passing any set of sockets (all the parameters set to *nullptr*) or passing empty sets, are operations without any sense that could generate unexpected results.