6.5. Blocking vs. non-blocking sockets

So far in this chapter, you've seen that select() can be used to detect when data is available to read from a socket. However, there are times when its useful to be able to call send(), recv(), connect(), accept(), etc without having to wait for the result.

For example, let's say that you're writing a web browser. You try to connect to a web server, but the server isn't responding. When a user presses (or clicks) a stop button, you want the connect() API to stop trying to connect.

With what you've learned so far, that can't be done. When you issue a call to connect(), your program doesn't regain control until either the connection is made, or an error occurs.

The solution to this problem is called "non-blocking sockets".

By default, TCP sockets are in "blocking" mode. For example, when you call recv() to read from a stream, control isn't returned to your program until at least one byte of data is read from the remote site. This process of waiting for data to appear is referred to as "blocking". The same is true for the write() API, the connect() API, etc. When you run them, the connection "blocks" until the operation is complete.

Its possible to set a descriptor so that it is placed in "non-blocking" mode. When placed in non-blocking mode, you never wait for an operation to complete. This is an invaluable tool if you need to switch between many different connected sockets, and want to ensure that none of them cause the program to "lock up."

If you call "recv()" in non-blocking mode, it will return any data that the system has in it's read buffer for that socket. But, it won't wait for that data. If the read buffer is empty, the system will return from recv() immediately saying ``"Operation Would Block!"".

The same is true of the send() API. When you call send(), it puts the data into a buffer, and as it's read by the remote site, it's removed from the buffer. If the buffer ever gets "full", the system will return the error 'Operation Would Block" the next time you try to write to it.

Non-blocking sockets have a similar effect on the accept() API. When you call accept(), and there isn't already a client connecting to you, it will return 'Operation Would Block', to tell you that it can't complete the accept() without waiting...

The connect() API is a little different. If you try to call connect() in non-blocking mode, and the API can't connect instantly, it will return the error code for 'Operation In Progress'. When you call connect() again, later, it may tell you 'Operation Already In Progress' to let you know that it's still trying to connect, or it may give you a successful return code, telling you that the connect has been made.

Going back to the "web browser" example, if you put the socket that was connecting to the web server into non-blocking mode, you could then call connect(), print a message saying "connecting to host www.floofy.com..." then maybe do something else, and them come back to connect() again. If connect() works the second time, you might print "Host contacted, waiting for reply..." and then start calling send() and recv(). If the connect() is still pending, you might check to see if the user has pressed a "abort" button, and if so, call close() to stop trying to connect.

Non-blocking sockets can also be used in conjunction with the select() API. In fact, if you reach a point where you actually WANT to wait for data on a socket that was previously marked as "non-blocking", you could simulate a blocking recv() just by calling select() first, followed by recv().

The "non-blocking" mode is set by changing one of the socket's "flags". The flags are a series of bits, each one representing a different capability of the socket. So, to turn on non-blocking mode requires three steps:

1. Call the fcntl() API to retrieve the socket descriptor's current flag settings into a local variable.
2. In our local variable, set the O_NONBLOCK (non-blocking) flag on. (being careful, of course, not to tamper with the other flags)
3. Call the fcntl() API to set the flags for the descriptor to the value in our local variable.