I/O models

In chapter 3 I briefly touched **blocking** and **non-blocking** sockets, which play a role in the available winsock **I/O models**. An I/O model is the method you use to control the program flow of the code that deals with the network input and output. Winsock provides several functions to design an I/O strategy, I will discuss them all here in short to get an overview. Later in the tutorial I will deal with most models separately and show some examples of them.

1. The need for an I/O model

So why do you need an I/O model? We don't have infinite network speed, so when you send or receive data the operation you asked for may not be completed immediately. Especially with networks, which are slow compared to 'normal', local operations. How do you handle this? You could choose to do other things while you're waiting to try again, or let your program wait until the operation is done, etc. The best choice depends on the structure and requirements of your program.

Originally, Berkeley sockets used the blocking I/O model. This means that all socket functions operate synchronously, ie. they will not return before the operation is finished. This kind of behavior is often undesirable in the Windows environment, because often user input and output should still be processed even while network operations might occur (I explained this earlier in chapter 3). To solve this problem, non-blocking sockets were introduced.

2. Non-blocking mode

A socket can be set into non-blocking mode using **ioctlsocket** (with FIONBIO as its cmd parameter). Some functions used in I/O models implicitly set the socket in non-blocking mode (more on this later). When a socket is in non-blocking mode, winsock functions that operate on it will never block but always return immediately, possibly failing because there simply wasn't any time to perform the operation. Non-blocking sockets introduce a new winsock error code which - unlike other errors - is not exceptional. For now, keep the following in mind:

WSAEWOULDBLOCK

This constant is the error code a winsock function sets when it cannot immediately perform an operation on a non-blocking socket. You get this error code when you call **WSAGetLastError** after a winsock function failed. Its name literally says 'error, would block', meaning that the function would have to block to complete. Since a non-blocking socket should not block, the function can never do what you ask it to. Note that this isn't really an error. It can occur all the time when using non-blocking sockets. It just says: I can't do that right now, try again later. The I/O model usually provides a way to determine what's the best time to try again.

3. I/O models

I've made several attempts to find a categorical description of the several I/O models but I haven't really found a good one, mainly because the models' properties overlap and terms like (a)synchronous have slightly different meanings or apply to different things for each model. So I decided to just create a table with all the models to show the differences and explain the details later.

Model	**Blocking mode**	**Notification method**		
		none	*on network event*	*on completion*
Blocking sockets	blocking	**x**		
Polling	non-blocking	**x**		
Select	both		blocking select	
WSAAsyncSelect	non-blocking		window message	
WSAEventSelect	non-blocking		event objects	
Overlapped I/O: blocking	N/A			blocking call
Overlapped I/O: polling	N/A	**x**		
Overlapped I/O: completion routines	N/A			callback function
Overlapped I/O: completion ports	N/A			completion port

The first five models are commonly used and fairly easy to use. The last four actually use the same model (overlapped I/O), but use different implementation methods. Actually, you don't really need overlapped I/O unless you're writing network programs that should be able to handle thousands of connections. Most people won't write such programs but I included them because good information and tutorials about the overlapped I/O model is not easy to find on the web. If you're not interested in overlapped I/O you can safely skip the future chapters about them.

One way to divide the I/O models is based on the blocking mode it uses. The **blocking sockets** model naturally uses blocking mode, while the others use non-blocking mode (select may be used for both). The blocking mode is not applicable to overlapped I/O because these operations always operate asynchronously (the blocking mode cannot affect this nor the other way around).

Another way to divide them is using their differences in the notification method used (if any). There are three subtypes:

- **None**
  There is no notification of anything, an operation simply fails or succeeds (optionally blocking).
- **On network event**
  A notification is sent on a specific network event (data available, ready to send, incoming connection waiting to be accepted, etc.). Operations fail if they cannot complete immediately, the network event notification can be used to determine the best time to try again.
- **On completion**
  A notification is sent when a pending network operation has been completed. Operations either succeed immediately, or fail with an 'I/O pending' error code (assuming nothing else went wrong). You will be notified when the operation does complete, eliminating the need to try the operation again.

Blocking mode doesn't use any notifications, the call will just block until the operation finished. WSAAsyncSelect is an example of a network event notification model as you will be notified by a window message when a specific network event occurred. The completion notification method is solely used by overlapped I/O, and is far more efficient. They are bound directly to the operations; the big difference between the network event and completion notification is that a completion notification will be about a specific operation you requested, while a network event can happen because of any reason. Also, overlapped I/O operations can - like its name says - overlap. That means multiple I/O requests can be queued.

In the next section I will show you the details of each model separately. To give you a more intuitive view of the models, I've created timeline images and used a conversation between the program and winsock as an analogy to how the model works.

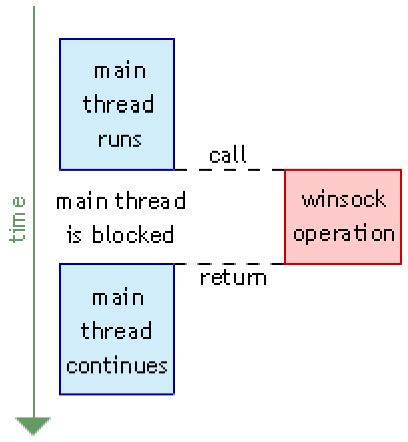
Note

In many of these timelines I've assumed the winsock operation fails (in a WSAEWOULDBLOCK way) because that is the interesting case. The function might as well succeed and return immediately if the operation has been done already. I've left this case out in most of the timelines in favor of clarity.

4. Blocking sockets

Blocking sockets are the easiest to use, they were already used in the first socket implementations. When an operation performed on a blocking socket cannot complete immediately, the socket will block (ie. halt execution) until it is completed. This implies that when you call a winsock function like send or recv, it might take quite a while (compared to other API calls) before it returns.

This is the timeline for a blocking socket:



As you can see, as soon as the main thread calls a winsock function that couldn't be completed immediately, the function will not return until it is completed. Naturally this keeps the program flow simple, since the operations can be sequenced easily.

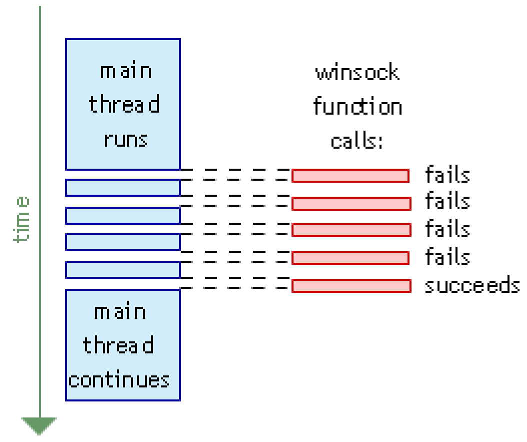
By default, a socket is in blocking mode and behaves as shown above. As I told earlier, I will also show each I/O model in the form of a conversation between the program and winsock. For blocking sockets, it's very simple:

```
program: send this data

winsock: okay, but it might take some time
         ...
```

```
...
...
done!
```

5. Polling

Polling is actually a very bad I/O model in windows, but for completeness' sake of I will describe it. Polling is an I/O model for non-blocking sockets, so the socket first has to be put into non-blocking mode. This can be done with **ioctlsocket**. Polling in general is repeating something until its status is the desired one, in this case repeating a winsock function until it returns successfully:



Because the socket is non-blocking, the function will not block until the operation is finished. If it cannot perform the operation it has to fail (with WSAEWOULDBLOCK as error code). The polling I/O model just keeps calling the function in a loop until it succeeds:

```
program: send this data

winsock: sorry can't do that right now, I would block

program: send this data

winsock: sorry can't do that right now, I would block

program: send this data

winsock: sorry can't do that right now, I would block

program: send this data

winsock: sorry can't do that right now, I would block
```

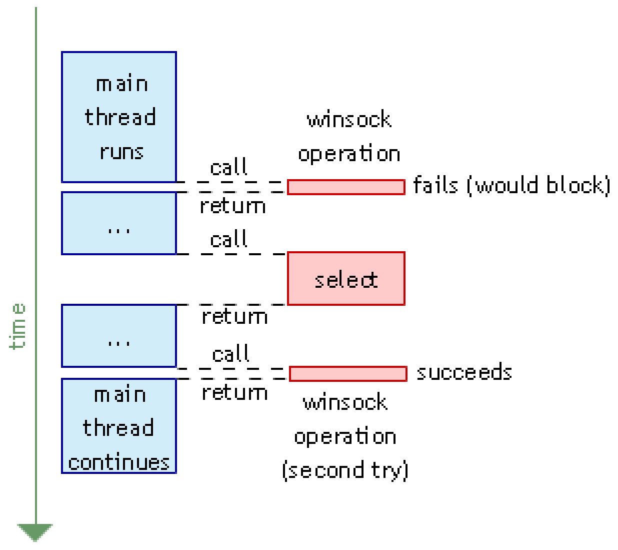
```
program: send this data
winsock: done!
```

As I said, this is a really bad method because its effect is the same as a blocking function, except that you have some control inside the loop so you could stop waiting when some variable is set, for example. This style of synchronization is called 'busy waiting', which means the program is continuously busy with waiting, wasting precious CPU time. Blocking sockets are far more efficient since they use an efficient wait state that requires nearly no CPU time until the operation completes.

Now you know how the polling I/O model works, forget about it immediately and avoid it by all means :)

6. Select

Select provides you a more controlled way of blocking. Although it can be used on blocking sockets too, I will only focus on the non-blocking socket usage. This is the timeline for select:



And the corresponding conversation:

```
program: send this data
winsock: sorry can't do that right now, I would block
program: okay, tell me when's the best time to try again (the select call)
```

```
winsock: sure, hang on a minute
         ...
         ...
         try again now!

program: send this data

winsock: done!
```

You might have noticed that the select call looks suspiciously similar to the blocking socket timeline. This is because the **select** function does block. The first call tries to perform the winsock operation. In this case, the operation would block but the function can't so it returns. Then at one point, select is called. Select will wait until the best time to retry the winsock operation. So instead of blocking winsock functions, we now have only one function that blocks, select.

If select blocks, why use it for non-blocking sockets then? Select is more powerful than blocking sockets because it can wait on multiple events. This is the prototype of select:

```
select PROTO nfds:DWORD, readfds:DWORD, writefds:DWORD, exceptfds:DWORD,
timeout:DWORD
```

Select determines the status of one or more sockets, performing synchronous I/O if necessary. The nfds parameter is ignored, select is one of the original Berkeley sockets functions, it is provided for compatibility. The timeout parameter can be used to specify an optional timeout for the function. The other three parameters all specify a set of sockets.

- readfds is a set of sockets that will be checked for readability
- writefds is a set of sockets that will be checked for writability
- exceptfds is a set of sockets that will be checked for errors

Readability means that data has arrived on a socket and that a call to **read** after select is likely to receive data. Writability means it's a good time to send data since the receiver is probably ready to receive it. **Exceptfds** is used to catch errors from a non-blocking connect call as well as out-of-band data (which is not discussed in this tutorial).

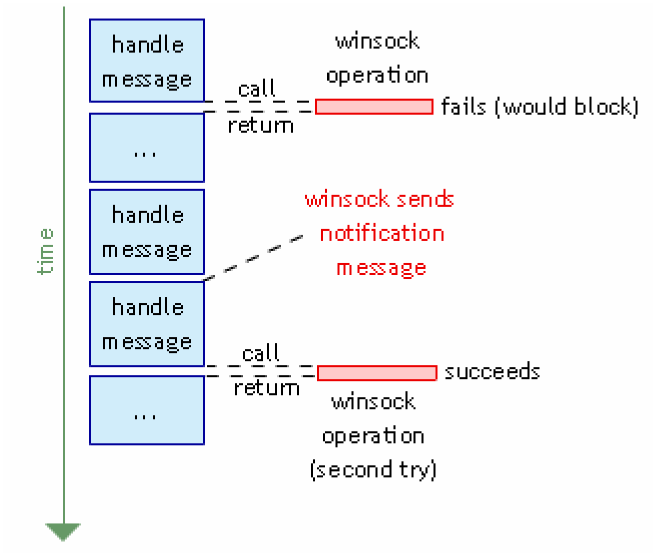
So while select may block you have more control over it since you can specify more than one socket to wait on for a specific event, and multiple types of events (data waiting, ready to send or some error that has occurred). Select will be explained more detailed in later chapters.

7. Windows messages (WSAASyncSelect)

Many windows programs have some kind of window to get input from and give information to the user. Winsock provides a way to integrate the network event notification with a windows's message handling. The **WSAAsyncSelect** function will register notification for the specified network events in the form of a custom window message.

```
WSAAsyncSelect PROTO s:DWORD, hWnd:DWORD, wMsg:DWORD, lEvent:DWORD
```

This function requires a custom message (wMsg) that the user chooses and the window procedure should handle. lEvent is a bit mask that selects the events to be notified about. The timeline is as follows:



Let's say the first message wants to write some data to the socket using **send**. Because the socket is non-blocking, send will return immediately. The call might succeed immediately, but here it didn't (it would need to block). Assuming WSAAsyncSelect was setup to notify you about the **FD_WRITE** event, you will eventually get a message from winsock telling you a network event has happened. In this case it's the FD_WRITE event which means something like: "I'm ready again, try resending your data now". So in the handler of that message, the program tries to send the data again, and this is likely to succeed.

The conversation between the program and winsock is much like the one with select, the difference is in the method of notification: a window message instead of a synchronous select call. While select blocks waiting until an event happens, a program using WSAASyncSelect can continue to process windows messages as long as no events happen.

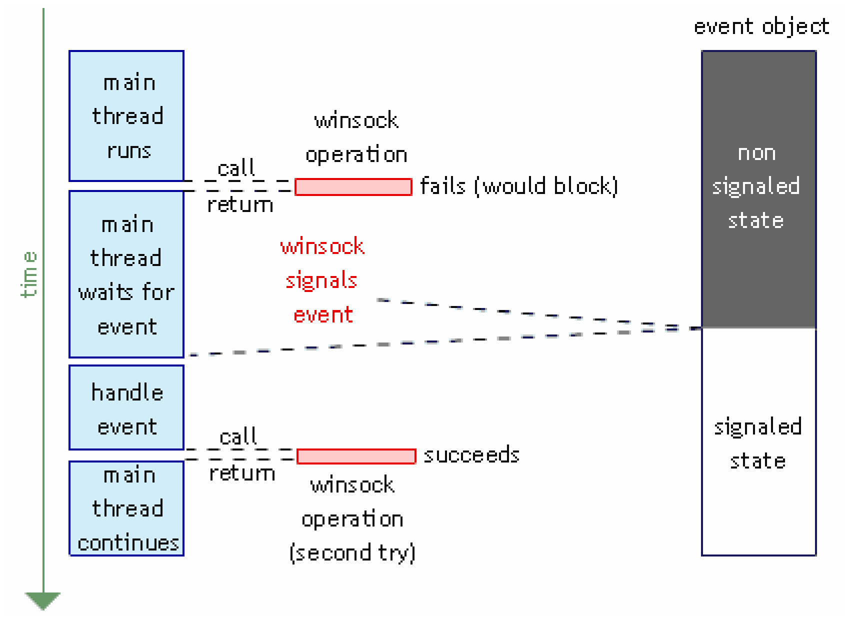
```
program registers for network event notification via window messages

program:   send this data

winsock:   sorry can't do that right now, I would block

program handles some message

program handles some other message

program gets a notification window message from winsock

program:   send this data

winsock:   done!
```

WSAAsyncSelect provide a more 'Windows natural' way of event notification and is fairly easy to use. For low traffic servers (ie. < 1000 connections) it efficient enough as well. The drawback is that window messages aren't really fast and that you'll need a window in order to use it.

8. Event objects (WSAEventSelect)

WSAAsyncSelect brother is WSAEventSelect, which works in a very similar way but uses event objects instead of windows messages. This has some advantages, including a better separation of the network code and normal program flow and better efficiency (event objects work faster than window messages).

Have a good look at the timeline and conversation, it looks a bit complicated but it really isn't:



```
program registers for network event notification via event objects

program:   send this data

winsock:   sorry can't do that right now, I would block

program waits for the event object to signal

program:   send this data

winsock:   done!
```

It's hard to draw a timeline for this function since event objects are a very powerful mechanism that can be used in many ways. I chose for a simple example here as this I/O model will be explained in great detail later in this tutorial.

At first, this model seems a lot like blocking: you **wait** for an event object to be signaled. This is true, but you can also wait for multiple events at the same time and create your own event objects. Event objects are part of the windows API, winsock uses the same objects. Winsock does have special functions to create the event objects but they are just wrappers around the usual functions.

All that winsock does with this model is signaling an event object when a winsock event happens. How you use this notification method is up to you. That makes it a very flexible model.

The function used to register for network events is WSAEventSelect. It is much like WSAAsyncSelect:

```
WSAEventSelect PROTO s:DWORD, hEventObject:DWORD, lNetworkEvents:DWORD
```

WSAAsyncSelect will send you a custom message with the network event that happened (FD_READ, FD_WRITE, etc.). Unlike WSAAsyncSelect, WSAEventSelect has only one way of notification: signaling the event object. When the object is signaled, **one or more** events may have happened. Which events exactly can be found out with WSAEnumNetworkEvents.

9. Use with threads

Before starting with the overlapped I/O models I first want to explain some things about the use of threads. Some of the models explained can show different behavior when threads come into play. For example, blocking sockets in a single threaded application will block the whole application. But when the blocking sockets are used in a separate thread, the main thread continues to run while the helper thread blocks. For low traffic servers (let's say 10 connections or so), an easy to implement method is to use the select model with one thread per client. Each running thread is bound to a specific connection, handling requests and responses for that particular connection. Other ways of using threads are possible too, like handling multiple connections per thread to limit the number of threads (this is useful for servers with many connections), or just one main thread to handle the user input/GUI and one worker thread that deals with all the socket I/O.

The same thing holds for the other models, although some combine better with threads than others. For example, WSAAsyncSelect uses window messages. You could use threads but you somehow have to pass the received messages to the worker threads. Easier to use is WSAEventSelect, since threads can wait on events (even multiple) so notifications can be directly acted on in the thread. Pure blocking sockets can be used as well, but it's hard to get some control over a thread that is blocked on a winsock function (select has the same problem). With events, you can create a custom event (not winsock related) and use that to notify the thread about something that hasn't got to do with socket I/O like shutting down the server.

As you can see, threads can be very powerful and change the abilities of an I/O model radically. Many servers need to handle multiple requests at the same time so that's why threads are a logical choice to implement this; threads all run at the same time. In later chapters I will discuss the use of threads, for now it's enough to know you can use them.

10. Introduction to Overlapped I/O

Overlapped I/O is very efficient and when implemented well also very scalable (allowing many, many connections to be handled). This is especially true for overlapped I/O in combination with completion ports. I said before that for most uses overlapped I/O is a bit overkill but I will explain them anyway.

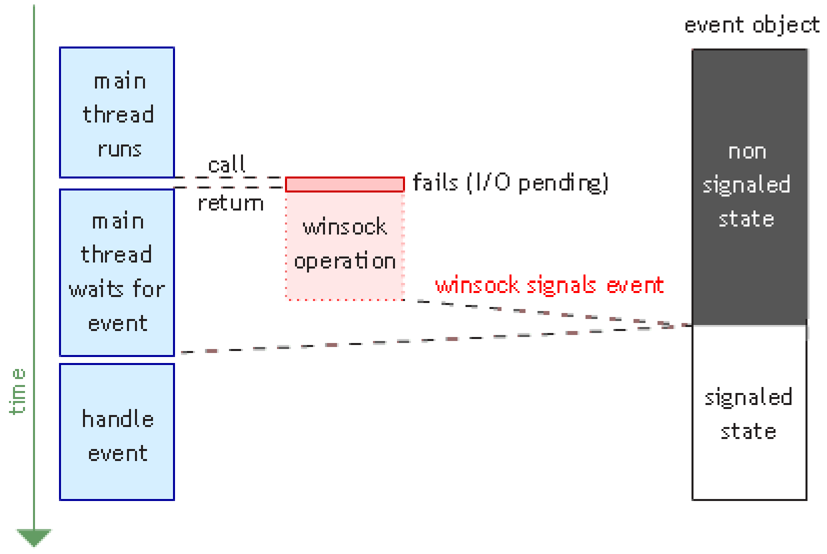
The asynchronous models discussed so far all send some kind of notification on the occurrence of a network event like 'data available' or 'ready to send again'. The overlapped I/O models also notify you, but about **completion** instead of a network event. When requesting a winsock operation, it might either complete immediately or fail with WSA_IO_PENDING as the winsock error code. In the latter case, you will be notified when the operation is finished. This means you don't have to try again like with the other models, you just wait until you're told it's done.

The price to pay for this efficient model is that overlapped I/O is a bit tricky to implement. Usually one of the other models can stand up to the task as well, prefer those if you don't need really high performance and scalability. Also, the windows 9x/ME series do not fully support all overlapped I/O models. While NT4/2K/XP has full kernel support for overlapped I/O, win9x/ME has none. However for some devices (including sockets), overlapped I/O is emulated by the windows API in win9x/ME. This means you can use overlapped I/O with winsock for win9x/ME, but NT+ has a much greater support for it and provides more functionality. For example, I/O completion ports are not available at all on win9x systems. Besides, if you're writing high-performance applications that require overlapped I/O I strongly recommend running it on an NT+ system.

As with the network event notification models, overlapped I/O can be implemented in different ways too. They differ in the method of notification: blocking, polling, completion routines and completion ports.

11. Overlapped I/O: blocking on event

The first overlapped I/O model I'm going to explain is using an event object to signal completion. This is much like WSAEventSelect, except that the object is set into the signaled state on completion of an operation, not on some network event. Here's the timeline:



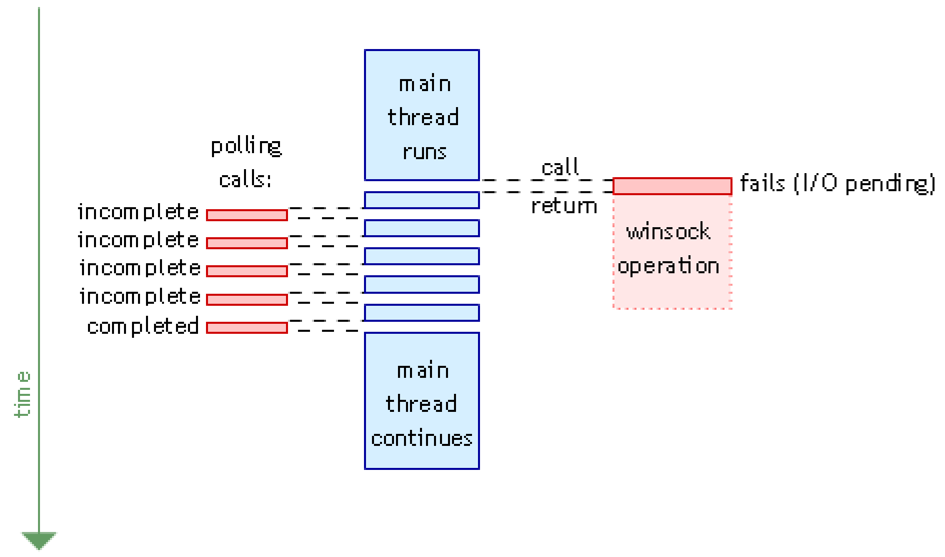
As with WSAEventSelect, there are many ways to use the event object. You could just wait for it, you could wait for multiple objects, etc. In the timeline above a blocking wait is used, matching this simple conversation:

```
program:       send this data

winsock:       okay, but I couldn't send it right now

program waits for the event object to signal, indicating completion of the
operation
```

As you can see, the winsock operation is actually performed at the same time as the main thread is running (or waiting in this case). When the event is signaled, the operation is complete and the main thread can perform the next I/O operation. With network event notification models, you probably had to retry the operation. This is not necessary here.

12. Overlapped I/O: polling

Just like the polling model mentioned earlier, the status of an overlapped I/O operation can be polled too. The WSAGetOverlappedResult function can be used to determine the status of a pending operation. The timeline and conversation are pretty much the same as the other polling model, except for that the operation happens at the same time as the polling, and that the status is the completion of the operation, not whether the operation succeeded immediately or would have blocked.



```
program: send this data
winsock: okay, but I couldn't send it right now
program: are you done yet?
winsock: no
program: are you done yet?
winsock: no
program: are you done yet?
winsock: no
program: are you done yet?
winsock: no
program: are you done yet?
winsock: yes!
```

Again, polling isn't very good as it puts too much stress on the CPU. Continuously asking if an operation completes is less efficient than just waiting for it in an efficient, little CPU consuming wait state. So I don't consider this a very good I/O model either. This doesn't render WSAGetOverlappedResult useless though, it has more uses, which I will show when the tutorial comes to the chapters about overlapped I/O.

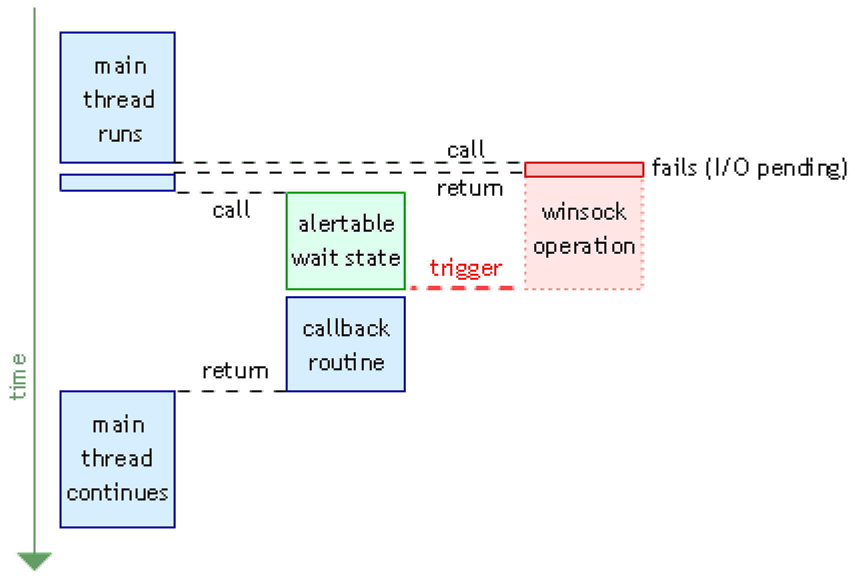
13. Overlapped I/O: completion routines

Completion routines are callback routines that get called when an operation (which you associated with the routine) completes. This looks quite simple but there is a tricky part: the callback routine is called in the context of the thread that initiated the operation. What does that mean? Imagine a thread just asked for an overlapped write operation. Winsock will perform this operation while your main thread continues to run. So winsock has its own thread for the operation. When the operation finishes, winsock will have to call the callback routine. If it would just call it, the routine would be run in the context of the winsock thread. This means the calling thread (the thread that asked for the operation) would be running at the same time as the callback routine. The problem with that is that you don't have synchronization with the calling thread, it doesn't know the operation completed unless the callback tells him somehow.

To prevent this from happening, winsock makes sure the callback is run in the context of the calling thread by using the APC (Asynchronous Procedure Call) mechanism included in windows. You can look at this as 'injecting' a routine into a threads program flow so it will run the routine and then continue with what it was doing. Of course the system can't just say to a thread: "Stop doing whatever you were doing, and run this routine first!". A thread can't just be intervened at any point.

In order to deal with this, the APC mechanism requires the thread to be in a so-called **alertable wait state**. Each thread has its own APC queue where APCs are waiting to be called. When the thread enters an alertable wait state it indicates that it's willing to run an APC. The function that put the thread in this wait state (for example SleepEx, WaitForMultipleObjectsEx and more) either returns on the normal events for that function (timeout, triggered event etc.) or when an APC was executed.

Overlapped I/O with completion routines use the APC mechanism (though slightly wrapped) to notify you about completion of an operation. The timeline and conversation are:

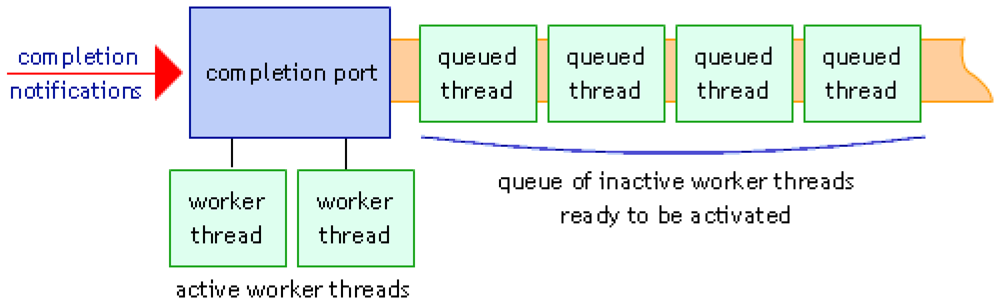


```
program:   send this data

winsock:   okay, but I couldn't send it right now

program enters an alertable wait state

the operation completes

winsock:   system, queue this completion routine for that thread

the wait state the program is in is alerted

the wait function executes the queued completion routine and returns to the
program
```

APCs can be a bit hard to understand but don't worry, this is just an introduction. Usually a thread is in the alertable wait state until the callback is called, which handles the event and returns to the thread. The thread then does some operations if necessary and finally loops back to the wait state again.

14. Overlapped I/O: completion ports

We've finally come to the last and probably most efficient winsock I/O model: overlapped I/O with completion ports. A completion port is a mechanism available in NT kernels (win9x/ME has no support for it) to allow efficient management of threads. Unlike the other models discussed so far, completion ports have their own thread management. I didn't draw a timeline nor made a conversation for this model, as it probably wouldn't make things clearer. I did draw an image of the mechanism itself, have a good look at it first:



The idea behind completion ports is the following. After creating the completion port, multiple sockets (or files) can be associated with it. At that point, when an overlapped I/O operation completes, a completion packet is sent to the completion port. The completion port has a pool of similar worker threads, each of which are blocking on the completion port. On arrival of a completion packet, the port takes one of the inactive queued threads and activates it. The activated thread handles the completion event and then blocks again on the port.

The management of threads is done by the completion port. There are a certain number of threads running (waiting on the completion port actually), but usually not all of them are active at the same time. When creating the completion port you can specify how many threads are active at the same time. This value defaults to the number of CPUs in the system.

Completion ports are a bit counter intuitive. There is no relation between a thread and a connection or operation. Each thread has to be able to act on any completion event that happened on the completion port. I/O completion ports (IOCP) are not easy to implement but provide a very good scalability. You will be able to handle thousands of connections with IOCP.