Practical difference between epoll and Windows IO Completion Ports (IOCP)

January 20, 2014 George

Contents [hide]

Introduction

This article compares the difference between the use of epoll and Windows I/O Completion Port (hereby IOCP).

It may be of interest to system architects who need to create a high-performance cross-platform networking servers, and to software engineers porting such code from Windows to Linux or vice versa. It may also be of interest to the developers familiar with one technology who would like to learn more about another one.

Both epoll and IOCP are efficient technologies when you need to support high performance networking with a large number of connections. They differ from other polling methods in may ways such as:

- They have no practical limitations (besides system resources) on the total number of descriptors/operations to monitor;
- They scale well with a VERY large number of descriptors; each adds very little overhead comparing to other polling/notification methods;
- They are suitable to a thread pool based model, when a small thread pool handles a large number of connections through a state machine;
- They are not effective, burdensome and essentially useless if all you need is a couple of client connections; their purpose is to handle 1000+ simultaneous connection;

In short, those technologies are suitable for creating a networked server which has to concurrently serve a very large number of clients. However at the same time those technologies differ significantly, and it is important to understand how they differ.

Notification type

The first and most important difference between epoll and IOCP is when you can receive a notification.

- epoll tells you when a file descriptor is ready to perform a requested operation – such as **"you can read from this socket now"**.
- IOCP tells you when a requested operation is completed, or failed to complete – such as **"the requested read operation has been completed"**.

When using epoll an application:

- Decides what action is required to be performed on a specific descriptor (receive data, send data or both);
- Sets the polling mask for the descriptor via *epoll_ctl*
- Calls *epoll_wait* which blocks until at least one monitored event is triggered. If more than one event is triggered, the function picks as many as it could.
- Finds the event data pointer from the *data* union.
- If the specific bits are set in the associated *revent* structure, initiates the specific operation (such as read, write or both)
- After the operation completes (which should be immediate), proceeds with the data read or send more data if any;
- Notably, the descriptor may have both events set at the same time, so the application can perform both read

Search for:
Search

Navigation

Linux software
Kchmviewer
Features
Getting kchmviewer
Frequently Asked Questions
Reporting bugs
Screenshots
Kchmviewer integration reference
Commercial Usage
Karaoke Lyrics Editor
Features
Getting it
Tutorial
Screenshots
Frequently Asked Questions
Registration
Libircclient
Getting it
Documentation
Karaoke Disk Unpacker
Details
Karaoke Player for Android
Features
Missing features
Support
Reporting bugs

Recent Posts

QtMultimedia, FFMpeg, Gstreamer: comparing multimedia frameworks
Building a Raspberry Pi-based camera powered by Ethernet
Building a WiFi-connected streaming camera and video recorder using Raspberry Pi 3
The Biggest Myth of Technology Licensing
Why companies license technology to other companies?

Archives

June 2016
May 2016
March 2016
January 2016
December 2015
November 2015
September 2015
March 2015
December 2014
November 2014
March 2014
January 2014
March 2013
February 2013

and write.

With IOCP an application:

- Initiates the required action on a specific descriptor (*ReadFile* or *WriteFile*) with the nonzero *OVERLAPPED* argument. The system queues the operation and the function returns immediately (as a side note, the function may complete immediately but this doesn't change the logic since even the operation which completed immediately still posts the completion notification; this could be turned off on Vista+ though).
- Calls *GetQueuedCompletionStatus()* which blocks until exactly one operation is completed and posted the notification. It makes no difference if more than one operation completes, this function will only pick one.
- Finds the event data pointer from the completion key and the *OVERLAPPED* pointer.
- Proceed with the data read or send more data if any;
- Only one completed operation could be got from a queue at the same time;

The difference between notification types makes it possible – and fairly easy – to emulate IOCP with epoll while using a separate thread pool. Wine project does just that. However it is not that easy to do the reverse, and I don'tt know of any easy way to emulate epoll with IOCP, and it looks rather impossible to keep the same or close performance.

Data accessibility

If the read operation is planned, there should be a buffer in your code somewhere to read into. If the write operation is planned, there should be a buffer in your code with the data to write.

- epoll doesn't care about those buffers and does not use them
- IOCP cares. Because IOCP works as "read 256 bytes into this buffer" -> return to OS -> completion notification, **the buffer must not be touched from the moment read is invoked until the operation completion notification is received**. Which may take a while.

A networked server typically operates with connection objects which contain the socket descriptor as well as other linked data such as buffers. Typically those objects are destroyed when the relevant socket is closed. There are, however, certain limitations when using IOCP.

IOCP works by queuing the *ReadFile* and *WriteFile* operations, which will complete later. Both *read* and *write* operate on buffers, and require the buffers passed to them to be left intact until the operation completes. More, you are not allowed to touch the data in those buffers. This places several important restrictions:

- You cannot use a local (stack-allocated) buffer to read the data into, or send the data from , because the buffer must be valid until the operation completes, which typically happens after you leave the function and thus invalidate the buffer pointers;
- You cannot dynamically reallocate the output buffer, for example if there is more data to send, and you decide to increase the buffer size. You cannot do this if there is a pending operation on this buffer, because this would invalidate the buffer. You can create a new buffer, but cannot destroy the old one, and since you don't know how much data would be sent, this makes the code more complex.
- If you write a proxy application, most likely you would have to introduce the double buffering, because both sockets would always have an active operation pending, and you could not touch their buffers.
- If your connection manager class is designed the way it could destroy the connection class anytime (for example when the connection class reported an error while processing the received data), your class instance cannot be destroyed until all the pending IOCP operations complete.

IOCP operation also requires the pointer to an *OVERLAPPED* structure, which also has to be kept intact – and not reused – until the operation completes. This also means if you need to do both reading and writing at the same time, you cannot inherit your class from the *OVERLAPPED* structure – a common design pattern. You would have to keep two structures as members of your socket class instead, passing one for use with *ReadFile* and another one for use with *WriteFile*.

epoll, however, does not use any I/O buffers, and therefore none of those issues apply.

Waiting condition modification

When adding a new event condition (such as we waited for socked available for reading, but now we also want to wait until it is available for writing), both epoll and IOCP make it easy to add a new condition. Epoll allows the polling mask to be modified anytime, and IOCP allows to start the new operation anytime.

Modifying or removing an existing condition, however, is different though. Epoll allows to easily modify a condition by using a single *epoll_ctl* call. It could be performed from any thread, and will works safely even if other threads are waiting for the condition.

IOCP is much more burdensome. If an I/O operation is scheduled, it should be canceled first by calling the CancelIo function. This function could only be called by the thread which scheduled the operation (i.e. it cannot be called by a dedicated management thread), and the operation status is unknown until the *GetOverlappedResult* retrieves the operation status. As stated above, this also means the buffers are untouchable until it happens.

Another issue with IOCP is that once the operation is scheduled it cannot be changed. For example, you cannot modify the queued *ReadFile* and tell it you'd like now to read only 10 bytes and not 8192; you'd have to cancel and reissue it. This is not an issue with epoll which does not schedule the operations.

Non-blocking connect

Some server implementations (inter-linked servers, FTP, p2p) need to initiate outbound connections. Both epoll and IOCP have support for the non-blocking *connect*, although different.

Using epoll, the code is the same whether you use *select, poll* or *epoll*. You create a non-blocking socket, call *connect()* on it and monitor it for writing (EPOLLOUT) condition.

Using IOCP you need to use a dedicated function ConnectEx, because the *connect()* call does not accept the *OVERLAPPED* structure and therefore cannot generate the completion notification. So not only the implementation will be different from the epoll, it would be different from the regular Windows implementation which uses *select* or *poll*. However the needed code change is rather small and insignificant.

An interesting note that *accept()* works as usual with IOCP. There is an AcceptEx function, but its role is completely different. It is not a non-blocking *accept()*.

Event monitoring

Often there is more data has arrived while the original event condition is triggered. For example, the epoll which monitors the descriptor for reading, or IOCP waiting for the *ReadFile* to complete, are triggered by the socket receiving the first chunk of data. Now what happens if there are another few chunks of data arrived after that? Is it possible to safely retrieve them without polling?

With epoll it is possible. Even if you receive only one *read* event, you can loop *read()*ing from the socket until you either read less than a requested amount, or got the EAGAIN error (and if you use the epoll edge mode you **must** do exactly that). Same with sending the data, if your producer sends the data in small chunks, you can loop around the *write()* call until EAGAIN.

With IOCP it is not possible. To read more data from the socket you need to post another *ReadFile* or *WriteFile* operation, and wait until it completes. This may create additional level of complexity. Consider the following example:

1. A socket class posted the *ReadFile* operation, and threads A and B are waiting in *GetOverlappedResult*()
2. The operation has been completed, thread A got the operation result, and called the socket class to process the read data
3. The socket class decided to read more data, and posted another *ReadFile* operation
4. This operation completed immediately, thread B got the result and called the socket class to process the read data
5. Now the read processing function is being called by two threads at the same time, with the execution order unknown.

There are of course a few ways to avoid this. First would be to have a lock per-class, but this introduces another issue. Locks aren't unlimited, and if you need to support 100k concurrent connections, you may run out of locks. You would also lose some concurrency, because your execution path for processing the read data may have nothing in common with the execution path for processing the written data.

The usual solution is to have the connection manager class call the *ReadFile* or *WriteFile* for the class. This is better – and as a bonus, allows destroying the class if needed – but makes the code more complex.

Conclusion

Both epoll and IOCP are suitable for, and typically used to write high performance networking servers handling a large number of concurrent connections. However those technologies differ significantly enough to require different event processing code. This difference most likely will make a common implementation of connection/socket class meaningless, as the amount of duplicated code would be minimal. In several implementation I have done an attempt to unify the code resulted in a much less maintainable code comparing to separate implementations, and was always rejected.

Also when porting, it is usually easier to port the IOCP-based code to use *epoll* than vice versa.

So my suggestion:

- If you need to develop the cross-platform networking server, you should focus on Windows and start with IOCP support. Once it is done, it would be easy to add epoll-based backend.
- Usually it is futile to implement the single Connection and ConnectionMgr classes. You will end up not only with a whole lot of #ifdef's but also with different logic. Better create the base ConnectionMgr class and inherit from it. This way you can keep any shared code in the base class, if there's any.
- Watch out for the scope of your Connection, and make sure you do not delete the object which has read and/or write operations pending.

This entry was posted in Uncategorized.

« select / poll / epoll: practical difference for system architects

Running Mac OS X under qemu/KVM on AMD CPU »

One Response to Practical difference between epoll and Windows IO Completion Ports (IOCP)

Emjayen *says:* July 20, 2015 at 7:48 am

NT6 and later support dequeuing of multiple IO completion packets per system call via GetQueuedCompletionStatusEx — it's the preferred method as it reduces the number of context switches per packet.

Reply