Overlapped IO와 IOCP 이야기 - 3번째

이 기탁 (Microsoft 2002 Asia MVP)
E-Mail: snaiper80@korea.com ,
nsnaiper@hotmail.com(MSN ID)
Devpia ID: snaiper

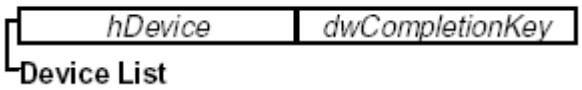
3번째 시간이네요. 변함없이 좋다고 해주시니 몸 둘 바를 모르겠네요. ^^: 여하튼 오늘은 별 잡담 없이 시작하죠.

♣ IOCP의 동작 원리

저번 강좌까지는 IOCP 커널 객체를 어떻게 생성시키고 이걸 socket 과 연결시키는 것과 이와 관련된 간단한 쓰레드 동작에 대해 알아 봤습니다. IOCP 초기에 시작하면서 하는 동작이 될 수 있지요. 이번에는 이제 동작원리에 대해 얘기를 해보고자 합니다. 아마 이번 강좌부터 좀 어렵게 느껴질 수도 있을 수도 있을 것 같네요. 좌우지간 최대한 쉽게 이해가 되도록 노력해보겠습니다.

IOCP 가 동작하기 위해서는 여러 가지 자료구조들이 필요합니다. 일전에 보았던 CreateIoCompletionPort 함수에서 Completion Key 와 socket 과의 연결이 있었지요? 그럼 이렇게 연결했다 라고 끝나면 안되겠죠? 그래서 이런 정보들을 저장하기 위한 자료구조들이 필요합니다. IOCP 동작에서 필요한 자료구조는 총 5가지 입니다. 일단 하나하나 보면서 살펴 가봅시다.

1) Device List



자 우선 Device List 입니다. 이것은 위에서 말씀 드렸던 연결에 관련된 자료구조입니다. List라고 이름 붙어 있는 거 보니 링크드 리스트로 관리 하지 않을까 추측해 볼 수도 있을 것 같네요. 그리고 이 리스트에 저장되는 정보들, 즉 레코드들의 내용은 위 그림과 같습니다. 이 레코드들이 리스트 형태로 주렁주렁 달려 있을 겁니다.

이 레코드의 내용을 가만히 들여다 보세요. 잘 보셨죠? 그럼 위에서 했던 얘기가 기억나실

겁니다. hDevice는 socket, dwCompletionKey는 socket과 연결된 Completion Key가 되겠지요? 자 대충 연결이 되시죠? 그럼 이런 생각의 연결 고리를 떠올리면서 이해해봅시다.

자 보죠. 이렇게 두 내용을 연결 해놓아야만 hDevice와 관련된 IO가 완료되었을 때 그에 관련된 Completion Key, 즉 dwCompletionKey를 우리한테 던져주겠죠? 만약 이렇게 안되고 따로 놀아버리면 어떻게 될까요? IOCP는 IO가 하나 완료되어서 처리하려고 하긴 하는데 이걸 누구에게 요청되었던 작업이 완료되었다고 하고 넘겨줘야 하는 거지? 하고 고민할 겁니다. 그렇겠죠?

그래서 우리는 위 그림과 같이 연결된 내용을 가지고 있다면 IOCP가 이 Completion Key를 찾고 우리에게 넘겨줄 수 있습니다. 그럼 우리는 어떤 device 가 작업을 완료했는지를 구분할 수 있게 되는 거죠.

그럼 Completion Key 에 대한 예를 한번 들어 볼까요? CK-SOCK라고 단지 #define CK-SOCK 1 이라고 해서 이걸 m_sock과 연결했다고 합시다. 그럼 위의 구조는 어떻게 되죠?

(m_sock, CK-SOCK)

이렇게 레코드가 하나 생성되겠죠? 그리고는 IOCP가 이 내용을 관리할 겁니다. 뭐 리스트에 뒷부분에 추가하던지 할 겁니다. 그럼 “hDevice에 어떤 IO가 완료되었다.” 라고 상황이 발생하게 된다고 해봅시다. 그럼 IOCP가 이 내용을 찾아서 이 지정된 CK- SOCK 와 함께 우리한테 던져주는 겁니다. 그럼 우리는 아 CK- SOCK 라는 Completion Key가 넘어왔네? m_sock에서 작업이 완료되었나 보다 라고 생각할 수 있는 겁니다.

여기까진 그냥 예로써의 상황이었습니다.

♤ Completion Key를 실제로는 어떻게 정하는가?

실제적인 프로그래밍에서는 어떻게 써 볼 수 있는지를 보겠습니다. 이 Completion key라는 것이 내용의 제한이 없습니다. 그래서 여러 가지로 쓸 수가 있지요. 뭐 Completion Key에 accept 된 Socket 을 지정했다고 합시다. 그럼 CreateIoCompletionPort 함수에는 소켓 변수가 두 번 쓰이겠죠? 그럼

(m_sock, m_sock)

이런 레코드가 생성될 수 있겠네요. 그리고는 리스트에 달려 있겠네요. 그럼 m_sock 에 어떤 IO를 요청한다고 해봅시다. 그럼 이 IO가 완료된다면 m_sock의 내용을 그대로 던져 주겠죠? 그럼 우리는 이걸 가지고 어떤 소켓이 IO 가 완료되었는데? 아하 m_sock 이었구나! 라고 알 수 있는 겁니다.

그럼 이 부분은 어떻게 되는지 대충 이해가 되셨지 않았을까 하는군요. Completion Key에

대한 얘기를 조금만 더 해보죠. Completion Key가 인자에 들어갈 때 ULONG_PTR이라고 보셨을 겁니다. 이것의 정의는 각자 찾아보시고, 이것은 크기로 따지자면야 DWORD랑 틀리지 않습니다. 즉 4bytes 라는 거죠. 그리고 IOCP는 여기에 어떤 내용이 들어가든 관계하지 않는다고 말했습니다. 그럼 우리는 활용을 생각해 볼 수 있겠네요. 여기다 포인터를 넘겨줘도 되겠죠? 제가 예를 하나 들어보죠.

```
struct PerHandleData
{
    SOCKET sock;
    char clientid[8];
};
```

자 이런 식의 구조체가 있다고 생각해봅시다. 그럼 이것을 new로 하고 나온 포인터를 이 Completion Key 에 넘길 수 있겠죠? 어차피 포인터도 4bytes 이니까요. 그럼 완료되어 연결되어 나오는 Completion Key도 Pointer 값이고, 이거 가지고 접근하면 어느 소켓에서 그랬는지 이 소켓과 관련된 client의 id는 뭔지 등등의 정보를 얻어낼 수 있습니다. 그래서 나름대로 PerHandleData 라고 이름을 붙여봤습니다. PerSessionData라고도 해도 될 듯 하군요. ^^ 뭐 여하튼 이렇게 한다면 나름대로 유용하겠죠? 아니라고요? 동의 못하시겠다면 어쩔 수 없지만요. ^^::

물론 위에서 말했던 바대로 socket만 집어넣고 나머지는 다른 곳에서 얻는 방법도 있습니다. 뭐 여하튼 결론을 말해보자면 프로그래밍 하는 사람 맘이라는 거죠. 하지만 위와 같은 구조체를 사용한다면 얼마던지 필요한 내용을 꺼내서 쓸 수 있도록 할 수 있을 것입니다.

♤ 정리

그럼 슬슬 이 자료구조가 도대체 언제 생성되고 언제 없어지는 지도 알아봐야 할 것 같네요. 간단히 정리해보자면 다음과 같습니다.

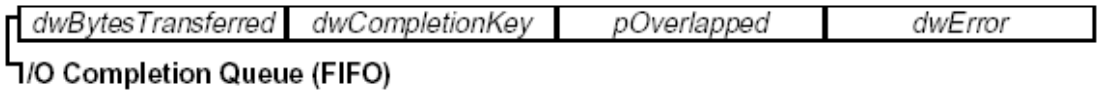
생성: CreateIoCompletionPort가 호출될 때
(기존 IOCP 포트와 socket등의 device 객체를 연결할 때를 말합니다.)

제거: hDevice에 지정된 핸들이 Close될 때
(소켓이라면 closesocket을 할 때이고 그 외라면 CloseHandle() 할 때입니다.)

2) IO Completion Queue

2번째 자료구조는 IO Completion Queue 입니다. 뭐 좀 줄이고 바꿔서 IOCP 큐라고 부르겠습니다.(이래도 되겠죠? ^^:)

이 자료구조는 IO가 완료가 되면 그 관련 정보를 저장하는 자료구조입니다. FIFO라고 적혀있고 또 큐라고 하니 처음 들어간 내용이 처음 나오는 그런 구조겠지요? 그리고 이 큐에서는 레코드 인스턴스들이 유지 됩니다. 무슨 레코드인데 라고 물으신다면 물론 방금 말한대로 IO하나가 완료되면 이 IO가 끝났을 때 그 끝난 결과를 레코드로 만들어서 가지고 있게 되는 겁니다. 그럼 그 내용을 한번 보죠.



위 그림의 첫 번째에 보세요.

dwBytesTransferred, 즉 얼마만큼 전송이 되었는가 입니다. 조금 바꿔 말하자면 IO가 이루어진 바이트 수는 얼마인가? 라는 겁니다. 여기서 이 값이 만약 0이 넘어온다면 어떻게 될까요? IO를 요청해서 완료했는데 0이더라? 그럼 소켓의 경우는 연결이 끊겨버린 경우겠네요? 그럼 이것을 보고 소켓 끊김을 체크해줘서 closesocket 해주시면 됩니다.

다음 두 번째, dwCompletionKey 입니다. 이름에서도 예상하셨듯이 이건 Completion Key 입니다. 첫 번째 자료구조에서 말했던 바대로 우리가 IO 완료 상황을 처리하려면 어떤 device 에서 완료가 되었는지를 알아야 할 겁니다. 그래서 Completion Key 가 넘어온다면 우리는 어떤 디바이스에서 또는 Winsock 에서는 어떤 소켓에서 완료가 되었는지를 알 수 있는 겁니다. 그러면 이 완료된 내용에 따라 그에 따른 적절한 처리를 해 줄 수가 있겠죠?

pOverlapped, 이거 제가 누누이 얘기했던 그 Overlapped 구조체 입니다. IOCP가 동작하려면 overlapped IO가 필요하고, 이것은 기본적으로 이 overlapped 구조체를 사용하기 때문에 이것이 넘어오는 겁니다. 이 Overlapped 구조체는 보통 확장하여서 사용합니다. 그 얘기는 조금 있다 하죠.

dwError, 말이 필요 없겠죠? 에러에 대한 내용입니다.

자 그럼 이 자료구조와 관련되는 동작들을 봅시다. Overlapped IO를 하나 요청했습니다. 그리고는 그것이 완료되었죠. 그럼 커널이 IOCP와 연결된 device, winsock 에서는 socket 이겠죠? 이걸 찾습니다. 만약에 IOCP와 연결된 것이 존재한다면, 이 완료되는 IO에 대한 정보를 모아서 위 그림과 같은 레코드를 만듭니다. 그리고는 그 IOCP와 관련되는 IOCP 큐, 즉 IO Completion Queue에다 집어넣습니다. 물론 FIFO니까 뒤에다 집어넣습니다. 그럼 IO가 완료된 차례대로 들어가게 될 겁니다. 그럼 우리가 이것을 하나하나 빼서 처리하는 겁니다. 물론 이 처리는 저번 강좌에서 말했던 IOCP Worker Thread 에서 합니다.

예를 들어 볼까요? 예를 들어 sock이라는 소켓에 recv작업이 완료되었다고 하죠. 정보를 한번 정해볼까요? 한 10bytes recv 되었다고 하고, 에러는 없었다고 합시다. 아 그리고 sock과 관련된 Completion Key는 그냥 sock 이라고 합시다. 그럼

(10, sock, pOverlapped, 0)

이런 레코드가 하나 만들어 집니다. 그리고는 이 레코드 내용이 IOCP 큐로 들어가는 거죠. 그럼 IOCP Worker Thread 중 하나 이 레코드를 큐에서 꺼냅니다. 그럼 이 내용을 처리하는 거죠. 보자 10바이트가 전송되었네? 그리고 sock 라는 소켓에 요청한 recv 작업이 그렇게 된 거고, 에러는 없네. 라고 해석이 가능한 겁니다. 자 이렇게 예를 들었으니 어느 정도 이해하셨지 않았나 싶네요. 그럼 슬슬 궁금한 게 생기시는 분들이 있으실 겁니다. 그럼 정작 중요한 데이터는 어떻게 알아내지 하는 거 말입니다. 그게 이제 말할 Overlapped 구조체 확장에 관련된 내용입니다.

♤ OVERLAPPED 구조체의 확장

보통 IOCP 프로그래밍 하면서 Overlapped 구조체를 확장 많이 합니다. 물론 IOCP로 하지 않고도 그렇게 쓸 수 있는 통보 방식도 있습니다. 뭐 그건 중요한 것이 아니고 일단 예를 하나 들어보죠.

```
struct PerIoOperationData
{
    OVERLAPPED ov;
    WSABUF buf;
    char buffer[4096];
};
```

자 이렇게도 가능하고 또 다음과 같이 해도 좋습니다.

```
struct PerIoOperationData: public OVERLAPPED
{
    WSABUF buf;
    char buffer[4096];
};
```

어느 쪽을 하셔도 좋습니다. 이건 프로그래밍 스타일에 관련된 겁니다. 단 아래 것은 C++에서만 되겠죠?

그래서 위와 같이 이렇게 확장해서 쓸 수 있다는 겁니다. 그래서 여기 있는 버퍼를 WSARecv 나 WSASend할 때 지정해 주는 겁니다. 만약 recv 동작이고 IO 완료 레코드가 IOCP 큐에 들어가 있는 상황이라면, 이미 여기에 외부에서 온 데이터들이 저장되어 있는 것이 되죠. 그래서 이 내용을 가지고 IOCP Worker Thread 에서 처리해주면 됩니다.

그럼 위에서 한 얘기에서 Completion Key는 socket으로 하고 그 외 정보는 다른 곳에서 얻는 방법이 있다고 한 말 기억나시죠? 이 다른 곳이 여기 입니다. 여기에다 필요한 내용을 더 저장하고 뽑아 쓸 수 있습니다. 참 그리고 이거 인자로 지정해줄 때는 위의 구조체라면 내부 엑세스를 하여 Overlapped 구조체를 지정해 줄 수 있고, 아래의 구조체라면 캐스팅을 한다면 가능하겠죠? 이 정도는 금방 이해가시리라 봅니다.

♤ PostQueuedCompletionStatus?

자 그럼 IO가 완료되고 이것이 어떻게 내용이 우리에게 알려질 수 있는 가에 대한 얘기를 했습니다. 물론 이와 관련된 자료구조들도 얘기했죠? 그럼 한가지 생각이 떠오르시는 분도 있지 않을까 하네요. IOCP 큐가 있는데 여기에 우리가 내용을 직접 넣을 수는 없을까 하는 그런 의문이요. 물론 저번 강좌를 열심히 읽으셨던 분이라면 제가 InterThread Communication 에 대한 얘기를 하면서 이에 대한 얘기를 조금 내비쳤던 걸 기억하실 겁니다. 물론 이것에 대한 대비가 되어 있죠. 이에 관련된 API가 PostQueuedCompletionStatus 입니다. 이름을 봐도 Post한다. 보낸다 라는 느낌이 들지요?

그럼 이 API도 한 번 살펴보죠.

```
BOOL PostQueuedCompletionStatus (
  HANDLE CompletionPort,                // handle to an I/O completion port
  DWORD dwNumberOfBytesTransferred, // bytes transferred
  ULONG_PTR dwCompletionKey,            // completion key
  LPOVERLAPPED lpOverlapped             // overlapped buffer
);
```

MSDN 에 있는 내용 가져왔습니다. 보시면 어떤 느낌이 드시나요? 왠지 위에 있던 그림에 있는 레코드 내용이랑 같다고 생각이 들지 않으시나요? 예 그렇습니다. 거의 같습니다. 이렇게 되야 큐에 레코드를 만들어 집어넣는 것이 되죠. 인자도 위에 있는 설명이니까 말씀드릴 필요가 없겠죠? 단 첫 번째 인자는 설명을 해야 되겠네요. 아마 다 이해하시지 않을까 합니다만 큐에다 집어넣는데 어느 IOCP 객체의 큐냐를 말해주는 인자입니다. 저기다 집어넣을 IOCP 객체의 핸들을 집어넣으면 그 IOCP 객체와 관련된 큐로 다음 3 개의 인자와 관련된 내용이 레코드로 만들어져서 가는 겁니다.

♤ PostQueuedCompletionStatus 를 어떻게 활용할까?

그럼 이것의 활용을 조금만 더 생각해볼까요? 저번 강좌에서 말했던 경우에는 뭐 예를 들어 Completion Key 부분에 보낼 내용을 집어넣고 반대쪽의 처리 쓰레드에서 이 Completion Key 의 내용만을 처리한다면 훌륭한 쓰레드간의 큐가 될 수 있을 겁니다.

또 다른 활용을 생각한다면 IOCP Worker Thread 의 종료 처리에 쓰일 수 있습니다. 서버 종료 시에 이 API 로 어떤 특정한 Completion Key 를 보낸다면 이것을 읽은 Thread 는 리턴 하여 쓰레드를 종료하게 만들어 버리는 겁니다. 그러면 안전하고도 멋지게 쓰레드를 종료시킬 수 있겠죠? 단 잊지 않으셔야 할 점은 다음에 말하겠지만 레코드가 하나 빠지면 그걸로 끝이라는 겁니다. 계속 남아 있지 않는다는 거죠? 그렇다면 쓰레드 여러 개면 당연히 종료 레코드도 쓰레드 개수만큼 보내 야겠죠?

♤ 정리

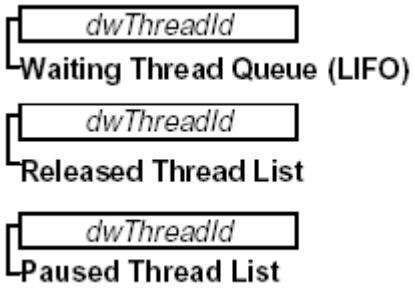
자 이제 어느 정도 얘기가 끝난 것 같군요. 이제 이것이 생성되고 삭제되는 경우에 대해 알아볼 시간이네요.

생성: IO 요청이 완료되었을 때
PostQueuedCompletionStatus 를 호출하였을 때
제거: Waiting Thread Queue 로부터 Entry 를 하나 제거할 때
(이 WTQ 는 뒤에 말하겠습니다. 즉 말하자면 IOCP Worker Thread 를 하나 깨우고 이 쓰레드에서 레코드 하나를 꺼낼 때 라고 말할 수 있겠습니다.)

3) Waiting Thread Queue, Released Thread List, Paused Thread List

한꺼번에 3개를 말해야 할 것 같군요. 이것은 서로 연관되어 있거든요.
이 세 개는 Concurrent Thread 숫자와 관계가 있습니다. 정확히 말해서는 이 3가지 자료구조를 IOCP Worker Thread 들이 옮겨 다니는 도중에 이 숫자가 관여한다고 말해야 할 것 같네요. 참 자료 구조에 Thread, Thread 들어가 있는 것 보니까 다 내부에 Thread랑 관련된 내용이 들어있을 것 같으시죠?

일단 자료구조와 그 레코드 내용을 봅시다.



자 이렇게 되어 있습니다. 레코드 내용은 아주 간단하네요. dwThreadID 즉 IOCP Worker Thread의 Thread ID 군요. 즉 아 아이디로만 모든 쓰레드를 관리한다는 뜻이겠죠?

♤ Waiting Thread Queue부터 알아보자

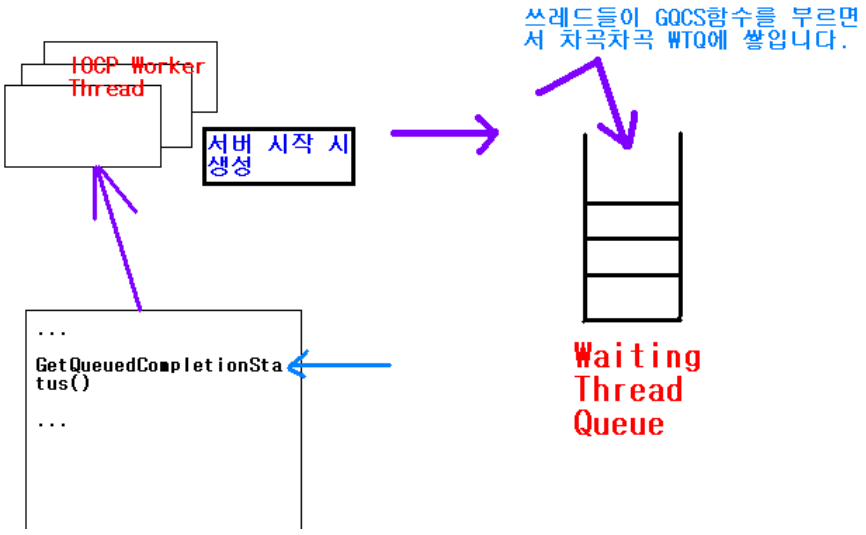
먼저 Waiting Thread Queue 에 대해 알아봅시다. 이건 제가 저번에 그림까지 그려가면서 말씀 드렸던 쓰레드 풀이라고도 불 수 있겠습니다. IOCP Worker Thread 들이 생성되면 여기에 차곡차곡 저장이 되는 겁니다. 아니 저장이라고 말하긴 좀 그럴까요? 하지만 그렇게 하는 것이 더 이해가 쉬울 것 같네요.
일단 이 쓰레드 풀이 어떻게 생성되는지부터 살펴봅시다. IOCP 소스를 한번이라도 분석해보신 분들은 알겠지만 쓰레드 함수에 위쪽 근처에 있는 코드에서 가장 먼저 불리는 함수가 GetQueuedCompletionStatus라는 함수가 라는 걸 기억하실 겁니다. 이 함수 이름을 보니 어떤 생각이 드세요? 이전에 말했던 PostQueuedCompletionStatus라는 함수와 이름이 대비된다고 느끼실 겁니다. 네 그렇습니다. 이 두 함수의 작용은 서로 대비됩니다.
제가 PostQueuedCompletionStatus 함수는 큐에다 어떤 레코드를 집어넣어 주는 거라고 말씀 드렸습니다. 그럼 GetQueuedCompletionStatus 함수는 반대로 큐에서 빼는 거라고 생각하시겠죠? 맞습니다. 큐에서 IO 완료가 오면 이 함수가 리턴 되면서 하나를 뺴오는 겁니다. 그럼 레코드의 정보가 GetQueuedCompletionStatus 함수의 인자로 나오는 거죠.

그럼 GetQueuedCompletionStatus 함수를 한번 볼까요? (이하, GetQueuedCompletionStatus 함수는 GQCS 함수라고 부르겠습니다.)

```
BOOL GetQueuedCompletionStatus(
  HANDLE CompletionPort,        // handle to completion port
  LPDWORD lpNumberOfBytes,      // bytes transferred
  PULONG_PTR lpCompletionKey,   // file completion key
  LPOVERLAPPED *lpOverlapped,   // buffer
  DWORD dwMilliseconds          // optional timeout value
);
```

자 GQCS 함수입니다. 보시면 두 번째에 나온 자료구조의 레코드 내용이 있다는 것을 볼 수 있으실 겁니다. 2,3,4 번째가 되겠네요. 2,3,4 번째가 포인터로 되어 있는 이유는 아시겠죠? Out Parameter 이기 때문입니다. 이 강좌 보실 분들이라면 이 정도는 아실 거라고 생각합니다. 물론 첫 번째 인자는 빼내올 IOCP 큐를 지니고 있는 IOCP 객체에 대한 핸들이 되겠습니다. 그리고 마지막 인자는 얼만큼 함수가 기다릴 건지에 대한 내용입니다. 보통 INFINITE 를 집어넣습니다. 그러니까 완료 레코드가 없다면 이 함수는 영원히 블럭되어 있다는 얘기입니다. 그리고 리턴값은 MSDN 을 참조하시길 바랍니다. IOCP 에서 가장 중요한 함수이기 때문에 꼭 MSDN 을 살펴보세요. 리턴 값은 뭐가 되는지 어떻게 될 때 에러가 뭐고 하는지를 살펴보시길 바랍니다.

자 그럼 이 함수를 왜 쓰레드 풀 얘기하는 데서 말할까요? 그 이유가 뭐냐고 하면 이것이 쓰레드에서 불릴 때 Waiting Thread Queue(WTQ) 에 들어가는 겁니다. 즉 GQCS 함수가 쓰레드에서 불린다면 이 쓰레드는 쓰레드 풀로 들어간다고 얘기할 수 있게 되는 겁니다.



(위 그림의 상황같이 이렇게 된다는 겁니다.)

그래서 이 GQCS 함수가 쓰레드에서 불린다면 INFINITE 옵션 시에 블로킹이 됩니다. 그리고는 쓰레드 풀에 들어가죠. 그래서 IO 가 요청되어서 완료되고 IO Completion Queue 에 레코드 들어가면 IOCP 가 이를 알아채어서 쓰레드 중 하나를 깨웁니다. 그리고는 GQCS 함수를 리턴 시킵니다. 좀 전에 얘기했듯이 리턴 할 때는 레코드를 하나 빼서 돌아옵니다. 그리고는 처리하고 다시 GQCS 를 부릅니다. 그럼 또 블로킹이 되겠지요?

자 이렇게 한다면 IO 가 완료되기를 기다리는 대부분의 시간 동안 쓰레드가 suspend 되어서 CPU 도 안 잡아먹고 있게 되죠? 그럼으로써 CPU 도 안 잡아먹는 훌륭한 쓰레드 풀이 완성되는 겁니다.

♤ GetQueuedCompletionStatus 함수가 리턴 될 때의 제약 사항

자 그런데 리턴 될 때 제약되는 사항이 있습니다. 그게 뭘까요? 이미 눈치채신 분도 있겠지만 Concurrent Thread 수에 대한 내용입니다. 이 GQCS 함수는 큐에 내용이 있으면 IOCP 가 쓰레드를 깨우고 그리고는 레코드 하나 가지고 리턴 된다고 했습니다. 그런데 리턴 되는 조건 중에 하나가 더 있습니다. 즉 현재 GQCS 함수가 리턴 되어 돌아가고 있는 쓰레드 수가 지정해준 Concurrent Thread 수를 넘어서면 이 함수는 큐에 내용이 있더라도 리턴 이 되질 않습니다. 이 점 때문에 Thread Switching 을 효율적으로 관리한다는 것이죠. 즉 Concurrent Thread 수를 넘어서지 않도록 IOCP 가 조절하여서 쓰레드를 깨운다는 말입니다. 하지만 이렇다고 하여서 돌아가고 있는 쓰레드 수가 Concurrent Thread 수보다 항상 작지는 않습니다. 이보다 클 수도 있죠. 그것이 IOCP 가 Smart 하게 처리된다는 장점이죠. 이 얘기는 좀 있다 하겠습니다.

♤ 정리

그럼 언제 이 WTQ 에 쓰레드가 들어오고 또 언제 나가는 지를 한번 정리해볼까요?

WTQ 로 들어올 때: ① 쓰레드함수가 GetQueuedCompletionStatus()를 불렀을 때
(즉 IOCP 서버 시작 시에 쓰레드를 만들어 놓을 때에 이 함수가 불리죠?
이 함수가 불림으로써 쓰레드 풀이 만들어지는 겁니다.
그럼 저번 강좌에서 그 쓰레드 풀이 어느 코드에서 만들어지는 거야
하고 궁금하셨던 분들은 이제 풀리실 겁니다.)
② GetQueuedCompletionStatus 함수가 리턴 되어 IO 완료 레코드를
처리한 후에 다시 GetQueuedCompletionStatus 함수를 불렀을 때
(이것 또한 별로 어려움 없이 이해되시겠죠?)

WTQ 에서 나갈 때: IO Completion Queue 가 비어있지 않고(And)

Release Thread List 에 있는 쓰레드 수가 지정해준 Concurrent Thread 수를 넘지 않았을 때
(이런 조건을 만족하여 나가면 IO Completion Queue 에서 레코드 entry 가 하나 제거되고, WTQ 에 있던 dwThreadID 값은 Release Thread List 로 옮겨 가게 됩니다. 그리고 나서는 GQCS 함수가 리턴 하면서 그 레코드 값을 GQCS 함수의 Out Parameter 로 내용들을 내보내게 되는 거죠.)

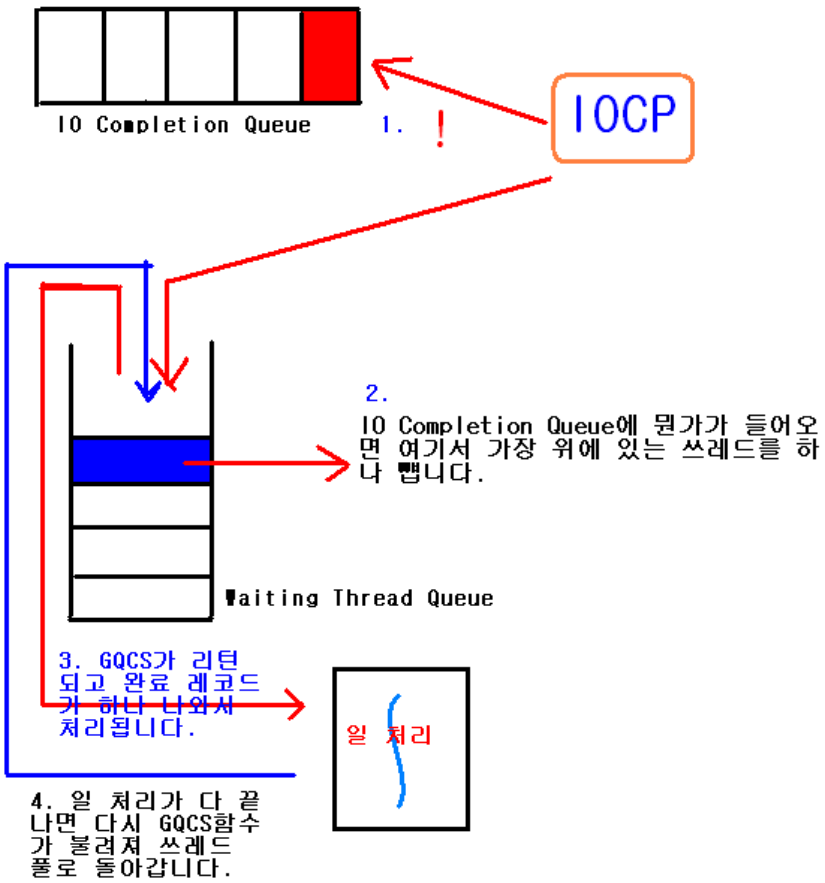
그럼 정리가 끝났죠? 정리가 끝나긴 했는데 뭔가 빼 먹은 것 같은 느낌이 들지 않으세요? 아마 읽으시는 분들 중에는 왜 이건 이렇지? 하는 궁금증이 생기시는 부분이 있으실 겁니다. 그걸 얘기해보죠.

♤ 왜 Waiting Thread Queue가 LIFO 구조를 가질까?

위에서 보면 Waiting Thread Queue해놓고 LIFO 라고 적어 놓았습니다. Queue라고 해놓으면서 왜 LIFO지 저도 궁금합니다만, 그 사람들 그렇게 해놓은 건 저도 알 수가 없군요. 여하튼 이 Queue라면서 LIFO 라고 해 놓은 것도 좀 고개를 기웃거리게 하지만 왜 LIFO가 되야 하는 지가 더 궁금하시죠? 그럼 이걸 연구해 보죠.

LIFO 라고 하면 다들 Stack 이 머리에 떠 오르실 겁니다. 안 떠오르시는 분이 있다면 어디 가서 자료 구조 책을 한번이라도 살펴 보고 오시기 바랍니다. 그런데 제가 WTQ가 쓰레드 풀이라고 얘기를 했습니다. 그럼 쓰레드 풀이 FIFO를 가지고 리스트를 가지든 별로 상관없는 게 아닌가 하는 의문도 가지실 분이 있으시겠네요? 메모리 풀을 구현해보셨던 분들이라면 그거 리스트로 하는 것이 아닌가? 하는 분들도 있지 않으실까 합니다.

Stack 의 구조를 생각해보세요. 가장 위의 것이 먼저 나가고 또 그 다음 것이 나가죠? 이런 구조와 풀이라는 특성을 결합시켜 보세요. 쓰레드가 Stack으로 쌓여 있다고 봅시다. 그럼 쓰레드가 하나 필요하다면 Stack이니까 가장 위에 있던 쓰레드(실제로는 dwThreadID만 거기에 있겠죠?)가 나갈 겁니다. 그리고는 쓰고 나서는 다시 돌아오면 또 가장 위에 쌓이게 됩니다. 그럼 이런 작업이 여러 번 반복되다 보면 항상 쓰던 쓰레드만 쓰일 겁니다. 그렇죠? 가장 위에 있는 것부터 쓰고 또 가장 위로 다시 들어오니까요. 그림으로 그려보시면 가장 쉽게 파악이 되실 겁니다.



(이렇게 그림 그려보았습니다. 정리할 때 도움이 되실런지요?)

자 그럼 이렇게 하는 이유는 뭘까요? IOCP 쓰는 이유랑 비슷하겠죠? 네 바로 성능 때문입니다. 항상 쓰는 쓰레드만 쓰게 처리된다면 쓰레드 풀에 있는 쓰레드들 중에서 Stack 아래 쪽에 있는 쓰레드들은 거의 쓸 일이 없을 겁니다. 즉 Scheduling될 일이 없다는 겁니다. 그럼 OS에서는 이런 쓰레드들이 가지고 있는 자원들 뭐 그러니까 메모리 같은 거 말입니다. (쓰레드 마다 stack 이 따로 있죠? 그 stack 메모리 등등을 가리킵니다.) 이런 메모리들의 내용이 하드로 Swap 되게 됩니다. 그러니까 안의 내용은 하드 디스크에 다 써지고 실제 메모리에 있던 것들은 없어진다는 거죠. 그리고 또한 프로세서의 캐쉬에서도 flush되고요.

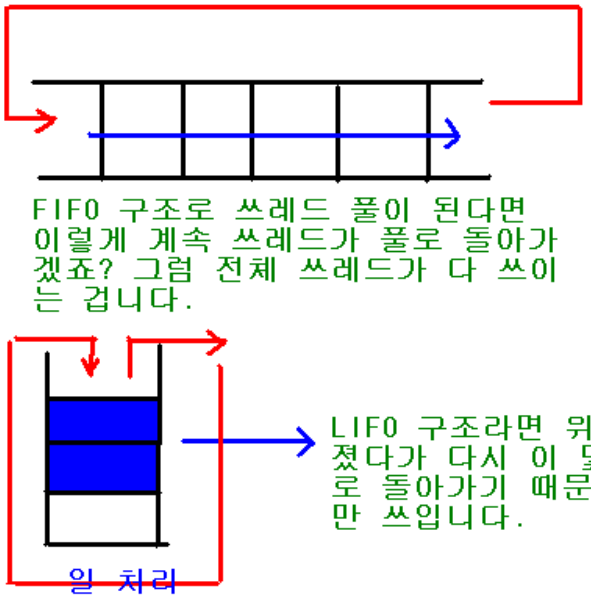
이러면 쓰레드 풀에 아무리 많은 쓰레드 들이 있다고 해도 실제로 메모리 차지하는 건 자주 쓰이는 위에 몇 개의 메모리 뿐이지요. 그래서 대비로 쓰레드 풀에 여러 개를 많이 넣어 놨다고 해서 그게 성능에 영향을 미치지 않는다는 말입니다. 만약에 IO가 완료되는 것이 아주 느려서 쓰레드 하나만 사용된다고 해보세요. 실제로 우리가 thread들을 100개 만들어 놓는다고 하더라도 쓰이는 건 하나가 되 버리니까 메모리에 유지하고 있는 건 쓰레드 하나면 된다는 겁니다.

그런데 아직 잘 이해가 되시지 않는 분들도 있으실 지도 모르겠군요. 그럼 거꾸로 생각해

봅시다. 만약에 LIFO 가 아니라 FIFO 구조로 있다고 생각해보세요. 그럼 쓰레드가 큐 front 에서 하나씩 빠져나가겠네요. 그럼 LIFO처럼 쓰던 것이 계속 쓰이는 것이 아니라 그 큐에 들어 있는 모든 쓰레드들이 언젠가는 다 한번씩은 쓰여지는 것이 되겠죠? 그럼 쓰레드들이 좀 많다고 생각해보면 큐 저 뒤 쪽에 있던 것들은 오랜 시간 쓰이지 않았으니까 OS에 의해 Swap이 됩니다. 그런데 FIFO 구조이니까 이 쓰레드들이 반드시 한번은 쓰여지게 되는 겁니다. 그럼 하드로 Swap된 메모리 내용을 다시 불러와야 하겠죠? 그럼 디스크 IO가 이루어지게 됩니다. 디스크 IO가 이루어진다는 뜻은 곧 성능이 떨어진다는 것을 의미합니다. 하드가 메모리나 CPU보다 느린 거는 아시죠? 첫 번째 강좌에서도 얘기를 해드렸습니다. 그럼 문제가 되는 겁니다. 이것이 계속 진행되다 보면 계속 Disk IO가 이루어지게 되는 거죠. IO가 이루어져서 하나 빼오고, 그리고 또 다음 번에는 또 하드에 있을 것이므로 또 빼오고요. 운영체제를 배우신 분들이 보면 LRU 등등이 나올 때 FIFO 가 나오죠? 그 때의 FIFO의 단점을 생각해보셔도 좋을 듯 합니다.

이것을 LIFO로 해보면 아무리 많이 있다고 해도 최악의 상황이 아니고서는(물론 이 최악의 상황을 위해서 쓰레드 풀에 쓰레드 들을 만들어 놓는 것이지만요.) Stack 아래쪽에 있는 것은 거의 쓰이지 않을 겁니다. FIFO에 비해서 Disk IO가 상대적으로 많이 줄어들겠죠? 한번 두 자료 구조를 그림을 그려놓고 생각해보시면 아실 거라고 봅니다.

제가 한번 그림을 그려 보았는데 이해가 되게 잘 그렸는지 모르겠군요. 한 번 참고 삼아 보시길……



자 이런 얘기입니다. 그런데 전달이 제대로 되는지는 모르겠네요 ^^:

이번 3번째 강좌는 여기까지 하죠. 얘기가 더 이어져야 하는데 끊겼으니까 다음 쓰는 대로 바로 올리겠습니다. 그럼 다음 강좌에서 ……