Overlapped IO와 IOCP 이야기 - 4번째

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

좀 많이 늦었죠? 죄송합니다 ^^ 이것 저것 하는 일이 많아서 말이죠. 그리고 예제 코드도 만든다고 삽질한 지라…… ^^: 여하튼 시작해보겠습니다.

♠ IOCP를 이용한 프로그래밍의 실제

이번은 실제로 어떻게 프로그래밍을 해나가느냐에 초점을 맞추도록 하겠습니다. 예제코드를 분석하는 식으로 하겠습니다. 아마 이 정도만 하시면 Overlapped IO 그리고 IOCP의 개념과 흐름을 정리하는 시간이 될 수 있을 걸로 생각이 됩니다.

자 이제 작성한 제가 조금 허접하게 ^^: 작성한 예제 코드를 보시면서 하나하나 설명해 나가겠습니다.

1. IOCP 초기화 처리

```
// 초기화 처리
BOOL NetworkController::Init(const int Port)
{
	// 생략
	m_listenSocket=WSASocket(AF_INET,SOCK_STREAM,0,NULL,0,WSA_FLAG_OVERLAPPED);
	// 생략
	// IOCP 초기화 처리
	int ErrCode;
	if(!m_IocpHandler.Create(0,&ErrCode))
	{
		TRACE("Create IO Completion Port Error: %dWn",ErrCode);
		return FALSE;
	}
```

첨에 나오는 것은 초기화 시킬 때 처리입니다. 보시면 몇 줄 정도의 코드가 나오죠? 물론 클래스화 처리가 되어서 조금 간단합니다만 천천히 설명해보도록 하겠습니다. 일단 주의할 것 소켓 생성입니다. 이렇게 WSASocket으로 생성할 시에는 반드시 WSA_FLAG_OVERLAPPED로 옵션을 주어서 생성시켜주셔야 합니다. 안 그러면 리슨 소켓에서 Accept시에 나온 클라이언트 소켓에 Overlapped 속성이 붙질 않습니다. 물론 socket으로 하신다면 이 점은 신경 쓰실 필요가 없겠습니다.

그리고 이런 코드가 나왔네요?

```
If(!m_IocpHandler.Create(0,&ErrCode))
```

이건 보면 아시겠지만 IOCP 커널 객체를 생성하는 코드 입니다. 현재 클래스화 되어 있어서 무척 간단하게 보이실 겁니다. 그럼 이 내부 코드를 한 번 볼까요?

```
////////////////////////////////////////////////////////////////////////////////////////////////
//
// IOCP 생성 및 초기화
// nMaxNumberOfConcurrentThreads: Concurrent Thread 의 개수
// pnOutErrCode: 에러 상황 시에 외부로 던져질 에러코드, NULL 이 들어오면 에러코드를 던지지 않는다.
//
////////////////////////////////////////////////////////////////////////////////////////////////
BOOL IocpHandler::Create(int nMaxNumberOfConcurrentThreads, int* pnOutErrCode)
{
  m_hIOCP=CreateIoCompletionPort(INVALID_HANDLE_VALUE,NULL,0,nMaxNumberOfConcurrentThreads);
  if(m_hIOCP==NULL && pnOutErrCode!=NULL)
  {
       *pnOutErrCode=GetLastError();
  }
          return (m_hIOCP!=NULL);
}
```

위의 간단한 한 줄 코드가 호출하는 함수의 내부 모습입니다. 내부도 그리 복잡하게 보이지는 않아 보입니다. 그렇지 않으시나요? 동의 못하신다면 ^^ 어쩔 수 없고요. 자 살펴봅시다. 먼저 제가 작성한 이 IocpHandler란 클래스는 IOCP 객체에 해당되는 모든 동작을 클래스화 시킨 클래스 입니다.(☞ 모두 설명하지는 않을 겁니다. 하지만 보시면 어떤 작용을 하는지 파악하실 수 있을 거라고 생각됩니다.)

그리고 그 중에서도 가장 중요한 함수가 바로 이 Create 함수입니다. Create 안 하면

처리를 시작할 수 없잖아요.

이 Create 함수 내부를 보시면 CreateIoCompletionPort() 란 함수가 보입니다. 앞에서도 설명했던 함수 입니다. 앞 강좌를 열심히 읽어보신 분들이면 아마 기억하실 겁니다. 그 때 인자를 설명하면서 뭐라고 그랬죠? IOCP 커널 객체를 생성하려면 첫 번째 인자에 INVALID_HANDLE_VALUE 라는 값을 줘야 한다고 했죠? 그래서 이렇게 첫 번째 인자에 저런 값이 들어간다면 아 생성하는 구나! 하고 생각하시면 되겠습니다. 그리고 두 번째 인자 Existing이라는 단어가 변수 명에 들어가 있었죠? 일전에 설명했듯이 이건 이미 생성된 IOCP 객체에 대한 핸들을 가리킵니다. 그래서 NULL 을 주었습니다. 막 생성하는데 줄 것도 없고 또 줄 것도 없고요. 그리고 세 번째 0으로 되어 있는 것은 Completion Key 입니다. 생성하는 곳에서는 Completion Key를 신경 쓰실 필요가 없습니다.

그리고 가장 중요하다고 느껴지는 Concurrent Thread 수입니다. 이것은 실제 얼만큼만 돌아가게 허용할 것인가를 정해주는 것이라고 했습니다. 3번째 강좌 문서를 열심히 보셨던 분이라면 아시겠네요. 일반적으로 0을 줍니다. 0으로 주면 CPU 개수를 Concurrent Thread 수로 잡습니다. 그래서 디폴트 인자로 0으로 주어서 특별하게 뭘 쓰지 않으면 0이 되도록 만들었습니다. 그래서 에러 코드를 쓰지 않으신다면 아무 인자도 주지 않아도 되겠죠? 다음 코드로 넘어가보죠.

```
if(!m_IocpHandler.CreateThreadPool(this))
{
		TRACE("Create Thread Pool Failed₩n");
		return FALSE;
}
TRACE("IOCP Initiation Success₩n");

m_pPerSocketCtxMemPool=new MemPooler<PerSocketContext>(MAX_USER);
m_pRecvMemPool=new MemPooler<PerIoContext>(MAX_USER*2);
m_pSendMemPool=new MemPooler<PerIoContext>(MAX_USER*2);
if(!m_pPerSocketCtxMemPool || !m_pRecvMemPool || !m_pSendMemPool)
{
		return FALSE;
}
TRACE("Memory Pool Create Success₩n");
return TRUE;
}
```

저번에 했던 강좌들을 생각해보면서 코드를 봅시다. 제가 객체를 생성하고 나서 해야 할 일

이 뭐라고 했나요? 쓰레드 풀을 생성 하는 것이라고 했죠? 그래서 CreateThreadPool이라는 함수를 호출하여 IOCP Worker 쓰레드 풀을 생성합니다. 이것의 내부를 한번 살펴봅시다.

```
////////////////////////////////////////////////////////////////////////////
// IOCP Worker Thread 풀을 만듬
// piProcessThread: IIocpProcessThread 인터페이스를 상속받은 클래스에 대한 포인터 즉 쓰레드 풀에 들어갈
//                  쓰레드 함수가 구현된 클래스의 포인터
// nNumOfPooledThredad: 풀링할 쓰레드 개수, 0이면 디폴트 값에 맞추어짐
////////////////////////////////////////////////////////////////////////////
BOOL IocpHandler::CreateThreadPool(IIocpProcessThread* piProcessThread, int nNumOfPooledThread)
{
        assert(piProcessThread);
        assert(nNumOfPooledThread>=0);

        if(nNumOfPooledThread==0)
        {
                SYSTEM_INFO si;
                GetSystemInfo(&si);
                // 디폴트 쓰레드 수로
                // 2 * 프로세서수 + 2 의 공식을 따랐음
                m_CreatedThreadNumber=si.dwNumberOfProcessors*2+2;
        }
        else
        {
               m_CreatedThreadNumber=nNumOfPooledThread;
        }

        for(int i=0;i<m_CreatedThreadNumber;i++)
        {       DWORD dwThreadId=0;

        CloseHandle(BEGINTHREADEX(NULL,0,IocpWorkerThreadStartingPoint,piProcessThread,0,&dwThreadId));
        }
        return TRUE;
}
```

함수 내용을 살펴보면 먼저 두 번째 인자로 들어온 지정할 쓰레드 개수에 따라 쓰레드를 생

성합니다. 하지만 0(0은 디폴트 값입니다.)으로 들어오게 된다면 2 * 프로세서 수 + 2 의 공식에 의하여 쓰레드를 생성합니다. 이 공식은 꼭 이렇게 만들어야 하는 것은 아니지만 이렇게 해보니 일반적으로는 괜찮더라 라고 하는 공식입니다. 물론 혹자는 뒤에 + 2를 빼서 하는 경우도 있습니다. 하지만 이런 공식보다는 자신의 서버 어플리케이션의 성격을 파악해서 더 필요하다면 더 만들어 주는 것이 좋을 것입니다. 그리고 IIocpProcessThread* piProcessThread 이 인자가 궁금하실 것으로 보입니다. 이것은 구현의 분리를 위해 IIocpProcessThread 라는 인터페이스를 정의하고 그 인터페이스를 상속 받은 클래스에서 쓰레드 함수를 구현하기만 하도록 만든 것입니다. 아주 간단한 클래스 설계이니 한번 보시면 아실 것이라고 생각합니다. 그래서 여기서는 IocpWorkerThreadStartingPoint 이라는 static 함수에서 저 인터페이스를 상속, 구현한 클래스의 함수를 호출하여 쓰레드를 만들게 됩니다. 이 구현 함수가 IOCP 에서 가장 중요하다고 할 수 있는 Worker 쓰레드입니다. 이건 조금 있다 설명하겠습니다.

그리고 밑에서는 MemPooler라는 템플릿 클래스를 쓰는 것을 볼 수 있습니다. 이것은 메모리 풀링을 위해서 쓰는 것입니다. 이건 하이텔의 안기찬(아이디 빌려 쓰신다고 하신 것 같은데 확실치 않군요.)님이 올린 소스 코드를 참조하고 조금 고쳤습니다. 구현을 잘 해 놓으셨더군요. 여하튼 서버에서 일일이 new, delete 한다는 것은 상당한 시간 손실, 오버헤드를 발생시킵니다. 그래서 이렇게 메모리 풀링을 하는 것이죠. 메모리 풀링이라는 것은 어렵게 생각하실 필욘 없습니다. 일전에 설명 드렸던 쓰레드 풀링의 개념을 생각하시면 됩니다. 미리 만들어 놓고 필요할 때 마다 불러 쓰고 필요 없으면 다시 풀에다 돌려 놓는 것이죠. 일종의 고정 메모리라고 생각하시면 되겠습니다. 뭐 아주 간단히 구현하지만 고정 배열을 많이 잡아두셔도 되겠네요. 여하튼 메모리 할당을 이런 식으로 해서 처리를 합니다. 그래서 일일이 new, delete 등을 하지 않도록 하고 있습니다.

그럼 Init() 란 함수가 끝났습니다. 실질적으로 IOCP 서버에서 초기화 과정이 이 함수에서 처리되는 겁니다. 위에는 코드가 생략되어 있지만 리슨 소켓 생성하고 bind 시키고 그리고 listen 모드로 해놓고, 그리고 IOCP 객체를 생성하고 쓰레드 풀을 만드는 과정까지가 IOCP 서버가 초기에 하는 일이 되겠습니다. 여기서 메모리 풀을 만드는 것은 선택 사항이 되겠죠. 간단하게 공부를 위해서 만드시는 것이라면 하실 필요가 없겠습니다. 저는 어디까지나 이런 방식을 사용한다는 것을 보여드리고 위해서 한번 사용해 봤습니다.

2. Accept 시에 해야 될 일은?

그럼 다음에는 해야 할 일이 뭐라고 생각하시나요? 소켓도 초기화 하고 IOCP 객체도 생성하고 했으니 이제는 외부 접속을 받아들여야 하겠죠? 그럼 일반적인 서버에서 외부 접속을 뭘 했죠? 예! 그렇습니다. Accept를 했죠. 만약 이 말이 0.5초 만에 튀어 나오지 않는 분이라면 아마 소켓 프로그래밍을 공부 안 하신 분일 겁니다. 그런 분은 소켓 프로그래밍부터 공부해보시길 권해드립니다.

자 그럼 accept 시에도 IOCP 서버라면 뭔가 특별한 일을 더 해야 하겠죠? 그 부분을 살펴봅시다.

```
clientsocket=accept(m_listenSocket,
                                (LPSOCKADDR)&clientsockaddr,
                                 &sockaddr_size);
            // 생략
int nZero=0;
if(SOCKET_ERROR==setsockopt(clientsocket,
                                    SOL_SOCKET,
                                    SO_RCVBUF,
                                    (const char*)&nZero,
                                    sizeof(int)))
nZero=0;
if(SOCKET_ERROR==setsockopt(clientsocket,
                                    SOL_SOCKET,
                                    SO_SNDBUF,
                                    (const char*)&nZero,
                                     sizeof(int)))

// 소켓 컨텍스트 할당 -> Completion Key
 pPerSocketCtx=AllocPerSocketContext(clientsocket);

// IOCP 커널 객체와 연결
if(!m_IocpHandler.Associate(clientsocket, reinterpret_cast<ULONG_PTR>(pPerSocketCtx), &ErrCode))

// 초기 Recv 요청
BOOL bRet=RecvPost(pPerSocketCtx);
```

accept 시에 처리하는 코드입니다. 아 참고로 여기 그리고 위에 있는 코드들은 생략된 것이 많습니다. 에러 처리라던가 등등 이런 것은 생략되어 있으니 생략되어 있는 부분은 같이 있는 예제 코드를 참조하시길 바랍니다. 그럼 보죠. 일단 accept를 합니다. 그럼 클라이언트가 올 때까지 대기하다가 접속해오면 클라이언트에 대한 소켓을 리턴 할 겁니다. 여기까지는 일반적인 얘기겠죠? 그럼 그 다음부터 조금 다릅니다. 보통은 이 소켓을 인자로 넘기는 쓰레드를 만들거나 할 터인데, 여기서는 약간의 옵션을 처리합니다. 위에 보시면 setsockopt 함수로 소켓 옵션을 조절하는데 인자 내용을 보시면 아시겠지만 소켓 버퍼 크

기를 조절합니다. 반드시 해야 된다고 하는 강제성은 없지만 하면 좋다고 하는 권고사항이 되겠습니다. 아시다시피 IOCP가 Overlapped IO에 대한 결과를 통보 받는 메커니즘이기 때문에 커널 단 버퍼를 사용하지 않고 직접 제공된 버퍼를 사용한다고 일전에 설명했습니다. 그래서 이 버퍼를 사용하지 않기 때문에 0으로 만들어 버리는 것이죠. 그래서 저런 코드가 들어간 것입니다. 그런데 지금 코드상으로 recv, send 버퍼 두 개를 모두 0으로 만들었지만 recv 는 해도 소용이 없다거나 오히려 성능 안 좋게 한다는 얘기도 더러 있습니다. 확인된 사실은 아닙니다만 보시는 분들은 사용하실 때 한번 정도 실험을 해보심이 좋을 듯 합니다.

그리고 다음에 할 일은 뭐냐 하면 소켓 컨텍스트를 할당하는 일입니다. 이것은 서버 구성에 따라 틀려질 수 있는 일입니다. 뭐 일단은 다른 분들은 확장 오버랩드 구조체를 하나 정의하고 new 해서 사용하시거나 하십니다. 그런데 저는 소켓 컨텍스트, 그리고 IO 컨텍스트 두 가지를 사용합니다. 그리고 이 소켓 컨텍스트를 Completion Key로 사용합니다. 이것 또한 설계에 따라 틀려지는 일이기 때문에 옵션이라고 볼 수 있습니다만 클라이언트에 대한 어떤 확장 Overlapped 구조체를 할당하는 작업만은 거의 필수라고 보시면 되겠습니다. 아참 컨텍스트, 컨텍스트 하는데 이거 뭐냐는 분들도 있으실 것 같군요. Context 이 것은 뭐라고 해야할까요? 일종의 정보의 단위라고 보시면 되겠습니다. Socket Context는 소켓과 관련된 정보의 모음, IO Context 는 IO 와 관련된 정보의 모음이라고 생각하시면 되겠습니다.

그럼 기왕에 컨텍스트 얘기가 나왔으니 이걸 한번 보죠.

```
typedef struct tagPerIoContext
{
          WSAOVERLAPPED overlapped;
          WSABUF wsaBuf;
          char Buffer[MAX_BUFFER];
} PerIoContext, *PPerIoContext;
typedef struct tagPerSocketContext
{
  SOCKET socket;
  PPerIoContext recvContext;
  PPerIoContext sendContext;

} PerSocketContext, *PPerSocketContext
```

이것은 제가 정의한 Context 의 정의입니다. 위의 PerIoContext 라는 것은 IO와 관련된 정보를 저장하는 구조체 입니다. 보시면 내부에 OVERLAPPED 구조체가 있는 것을 보실 수

있을 겁니다. 이렇게 멤버로 두거나 상속해서 처리하시면 되겠습니다. 그리고 그 외에는 IO와 관련된 정보를 두는데, IO를 한 결과를 담는 Buffer 라던지, 아니면 이 IO가 어떤 IO다라는 것을 알려주는 Flag 라던지 이런 변수들을 담아두는 겁니다. 다른 분들은 보니까 여기다 Socket 변수도 두시고 그리고 게임이라면 x, y 위치라던가 뭐 등등의 정보를 담아두시는 분들도 있었습니다. 이건 서버의 성격에 따라, 그리고 프로그래머가 필요로 하는 정보에 따라 틀려집니다.

그런데 저는 위와 같이 하질 않고 PerSocketContext 라는 구조체를 따로 정의해서 소켓과 관련된 정보를 따로 정리했습니다. 그리고 이것을 Completion Key로도 활용하는 것이죠.
그리고는 IO Context를 두 개 연결해두는 겁니다. Recv 에 대한 IO Context 하나, Send 에 대한 IO Context 하나. 이렇게 두 개 둔 이유는 Full Duplex라 할까요? Recv 와 Send를 동시에 하기 위해서입니다.

그럼 제가 정의한 것은 어느 정도 설명된 듯 합니다. 하지만 알아 두실 점은 이것은 정답이 아니라는 겁니다. 확장 오버랩드 구조체를 하나 정의해서 그것만 쓸 수 있습니다. Completion Key로는 소켓을 넘긴다던 지 해서 말입니다. 저는 이런 스타일을 즐깁니다만 보시는 분들은 님 편하게 쓰시기 바랍니다. 저는 이런 스타일이다라고 예시를 보여드렸을 뿐입니다.

자 그럼 다시 저 위에 코드로 넘어가봅시다. 이렇게 Socket Context를 하나 할당했습니다. 위의 PerSocketContext만큼의 공간을 메모리 풀에서 얻고 여기다 방금 받은 클라이언트 소켓을 대입해서 받아왔습니다. 소스를 자세히 분석해보신다면 아실 겁니다. 그래서 이 할당된 내용 가지고 뭘 해야 되겠죠? 저번에 제가 뭐라고 했던가요? IOCP 에서는 Completion Key와 socket 또는 device를 연결해야 된다고 했죠? 그걸 지금 하는 겁니다. IocpHandler의 멤버 Associate 라는 함수를 호출하여 연결합니다. 여기서 Completion Key가 PerSocketContext 즉 방금 할당했던 것이 넘어가도록 만들었습니다. 그럼 Associate 이 것의 내용이 궁금하실 터인데 이것은 여러분께 넘기겠습니다. 별로 어려운 내용은 아닐 겁니다. 함수의 핵심은 CreateIoCompletionPort에다 2번째 인자를 사용한다는 것이죠. 저번 강좌에서도 언급했던 사항이니 특별한 설명이 필요치 않을 것이라고 봅니다. 아마 저번 내용을 기억하시는 분들이라면 내용을 안보셔도 대충 그렇게 되겠구나 하고 추측하실 수 있을 겁니다.

그리고 그 다음은 가장 중요한 초기 Recv 걸기 입니다. 지금은 RecvPost 란 함수로 래핑되어 있지만 실제 내용은 WSARecv를 하는 것입니다. 그 코드는 다음과 같습니다.

```
// RECV 요청
BOOL NetworkController::RecvPost(PPerSocketContext pPerSocketCtx)
{
```

```
	DWORD dwRecvBytes=0;
	DWORD dwFlags=0;

	ZeroMemory(&pPerSocketCtx->recvContext->overlapped,sizeof(WSAOVERLAPPED));

	int ret=WSARecv(pPerSocketCtx->socket,&(pPerSocketCtx->recvContext->wsaBuf),1,
		&dwRecvBytes,&dwFlags,&(pPerSocketCtx->recvContext->overlapped),NULL);

	if(SOCKET_ERROR==ret)
	{
		int ErrCode=WSAGetLastError();
		if(ErrCode!=WSA_IO_PENDING)
		{
			TRACE("[%d] Recv Request Error(WSASend Function
Failed): %d₩n",GetTickCount(),ErrCode);;
			TRACE("Client will Close.₩n");
			return FALSE;
		}
	}

	return TRUE;
}
```

오버랩드 구조체를 초기화 시키고 WSARecv를 호출하는 것이죠. 참고로 WSARecv 할때 인자를 잘 넣으시길 바랍니다. 제가 예전에 실수한 부분이었는데 이거 잘못 넣었다 왜 안되지 하던 경험이 있었습니다. 인자의 포인터를 잘 보시고 넣으시길.....

여하튼 중요한 것은 초기 Recv를 반드시 해줘야 한다는 사실 입니다. 처음 짜보시는 분들이 빼먹는 부분이 이것인데요. Overlapped IO 라는 것을 한마디로 축약하자면 **선 요청 후 결과처리** 입니다. 그러니까 미리 recv를 socket 에 걸어줘야만 리시브가 된다는 것입니다. 안 그러면 워커 쓰레드로 처리결과가 오질 않고 recv 도 되질 않습니다. 그래서 접속만 되고 왜 내용은 안 받느냐 하시는 분들은 이 부분을 빼먹은 겁니다. 이렇게 초기 recv 를 걸어두면 recv 를 받을 수 있습니다. 단 이 부분에서 알아두실 것은 반드시 recv를 먼저 해야 하는 것은 아니라는 점입니다. 보통 서버에서는 recv를 먼저 하지만 때에 따라서는 send를 먼저 할 수도 있습니다. 그래서 그럴 경우에는 send를 걸어두셔도 됩니다. 즉 그러니까 정리하자면 어떤 경우든 간에 send,recv 든 IO를 하나 초기에 걸어주셔야 나머지 처리가 된다는 겁니다. 그래야 뭐든가 되겠죠? (물론 극소수의 예겠지만 안 하셔야 할 때도 있겠습니

다. 이건 서버 프로그램의 설계에 따라 달라지는 사항이겠죠.)

자 그럼 Accept 과정 처리 부분이 끝났습니다. 그럼 다시 정리해볼까요? 처음 초기화를 끝냈습니다. 그리고는 accept를 하죠. 그리고 클라이언트가 접속해서 accept 함수가 리턴되면 그 소켓과 관련된 여러 가지 정보를 확장 오버랩드 구조체에 집어넣습니다. 그리고는 그것을 이용하거나 또는 소켓 등을 Completion Key 로 삼아 IOCP 객체와 연결을 시킵니다. 그럼 이 때부터 소켓에 요구하는 IO는 IOCP를 통해 통보를 받을 수 있습니다. 그런 다음에는 뭘 한다고 했죠? 예 초기 IO를 요청합니다. 그래야 recv를 받든, send를 하든 처리할 수 있습니다.

3. Worker Thread 처리

3번째로 워커 쓰레드에 대한 처리가 되겠습니다. IOCP 서버를 짜면서 신경 쓰셔야 할 분은 첫 번째 초기화에서의 처리, 그리고 Accept 시의 처리, 그리고 어째 보면 가장 중요한 이 Worker Thread 에서의 처리입니다. 이 워커 쓰레드의 역할은 아시겠지만 정리하자면 이전의 요청한 IO의 완료 결과를 통보 받고 그 결과에 따라 어떤 IO를 더 요청하거나 아님 다른 작업을 하거나 합니다. 거의 중심적인 IO 처리는 여기가 될 것입니다.

그럼 코드를 볼까요?

```
// 완료 패킷 처리 함수
void NetworkController::ProcessingThread(void)
{
	PPerSocketContext pPerSocketCtx=NULL;
	PPerIoContext pPerIoCtx=NULL;
	DWORD dwBytesTransferred=0;
	int ErrCode=0;

	while(TRUE)
	{
	// IO Completion Packet 얻어온다.
	BOOL bRet=m_IocpHandler.GetCompletionStatus(reinterpret_cast<ULONG_PTR*>(&pPerSocketCtx),
						&dwBytesTransferred,
						reinterpret_cast<LPOVERLAPPED*>(&pPerIoCtx),
						&ErrCode);
```

워커 쓰레드 함수의 구현입니다. 아까 얘기했던 인터페이스에서 정의했던 함수가 이 void ProcessingThread(void) 함수입니다. 아까 CreateThreadPool 에서 쓰레드 만들 때 이 함수를 부르게 되는 거죠.

그래서 여기에서는 결과 패킷을 받아오고 그것을 처리하게 됩니다. 그래서 그런 정보를

받기 위해서 PerSocketContext의 포인터 변수, PerIoContext 포인터 변수, 그리고 전송 받은 바이트 수를 저장하는 변수를 선언합니다. 그리고 while 루프 안에서는 GetCompletionStatus 함수를 호출합니다. 눈치 채셨겠지만 이것은 내부적으로 GetQueuedCompletionStatus 함수를 싸고 있습니다. 이 함수는 결과 패킷을 없을 때는 블러킹 된다는 것은 알고 계시죠? 그리고는 쓰레드 풀에 돌아갑니다.

그래서 리턴 된 정보를 가지고 오는데 첫째 Completion Key, 이것은 제가 넣은 PerSocketContext 구조체에 대한 인스턴스가 될 겁니다. 그리고 그 외의 정보들 그리고 Overlapped 구조체에 대한 포인터가 리턴 되는데 이것은 PerIoContext의 주소를 집어 넣습니다. 그리고는 LPOVERLAPPED로 캐스팅을 하죠. LPOVERLAPPED 인 이유는 Out 파라미터이기 때문입니다. 그래서 이 인자에서 완료된 IO에 대한 오버랩드 구조체의 인스턴스에 대한 포인터가 리턴 되는데요. 이 포인터가 가리키는 것은 제 프로그램에서는 PerIoContext로 IO를 요청하므로 실제적으로는 PerIoContext에 대한 포인터가 리턴됩니다. 그래서 저렇게 캐스팅을 해서 집어넣는 것입니다. 좀 이해되셨나요? 자 그럼 내용이 다 넘어왔습니다.

그럼 우리는 이 것을 가지고 처리를 해야 되겠지요? 그 처리를 한번 봅시다.

```
// 클라이언트가 연결 끊음
if(dwBytesTransferred==0)
{
        TRACE("Client(%d) Connection Closed.₩n",pPerSocketCtx->socket);
        throw "dwBytestransferred==0₩n";
}
// IO 성격에 따라 그에 따른 처리
if(pPerIoCtx==pPerSocketCtx->recvContext)
{
    // RECV Operation
    if(!RecvCompleteEvent(pPerSocketCtx,dwBytesTransferred))
    {
        throw "RecvCompleteEvent Error₩n";
    }
}
else if(pPerIoCtx==pPerSocketCtx->sendContext)
{
   // SEND Operation
   if(!SendCompleteEvent(pPerSocketCtx,dwBytesTransferred))
   {
        throw "SendCompleteEvent Error₩n";
```

일단 첨에 보이는 것은 dwBytesTransferred가 0이면 클라이언트가 Close 되었다는 처리입니다. 이렇게 IO를 요청했는데 그 결과가 0이었다는 것은 즉 클라이언트가 접속을 끊었다는 것입니다. 그래서 소켓을 닫아주는 처리를 해야 합니다. 물론 이 때에는 연결된 메모리를 해제하는 작업도 병행되어야 하겠죠. 저의 프로그램 같은 거라면 메모리 풀에다 다시 메모리를 돌려주는 작업이 되겠습니다.

그리고 다음과 같은 코드가 보이실 겁니다. 주석으로 IO 성격에 따른 처리라고 되어 있습니다.

```
// IO 성격에 따라 그에 따른 처리
if(pPerIoCtx==pPerSocketCtx->recvContext)
```

즉 지금 완료된 결과가 recv완료냐 send완료냐를 결정하는 겁니다. 실제로 GetQueuedCompletionStatus 함수에서 리턴 되는 정보로는 IO 가 어떤 IO인지 알 방법이 전혀 없으므로 이렇게 우리가 그에 따른 처리를 직접 해줘야 합니다. 즉 어떤 IO인지를 플래그를 통해 알아내던 가를 해야 합니다. 그래서 보통 사람들은 Flag 를 두어서 IO_READ 이면 READ 라고 처리하고 IO_WRITE라면 WRITE 라고 처리하는 등의 간단한 상수를 정의해서 씁니다. 하지만 저는 최인호 님이 쓰셨던 방식처럼 이렇게 나온 포인터와 소켓 컨텍스트의 recv 쪽 IO Context의 포인터를 비교합니다. 이렇게 함으로써 플래그를 쓰지 않고 처리 어떤 IO인지를 알 수 있게 되는 겁니다. 그런데 이런 처리는 Context 구조체를 어떻게 설계했느냐에 좌우 됩니다. 저처럼 설계 했으면 이렇게 가능할 수 있고요. 확장 오버랩드 구조체를 하나만 정의했다면 불가능 할 수 있습니다. 즉 자신의 어떻게 정의했느냐에 따라 달라질 수 있다는 말입니다.

자 여하튼 초기에 recv를 걸었을 터이니 아마 첫 번째 if 문을 먼저 들어올 것입니다. 그럼 여기서 Recv 가 완료된 상황을 처리하는 겁니다. 이 완료라는 의미를 잘 생각해보시길 바랍니다. 이 예제는 에코 서버 이므로 단순히 첨 Recv 가 완료되면 그것을 보내주는 역할을 하게 됩니다. 그것은 RecvCompleteEvent란 함수 안에서 처리하게 되는데 저처럼 하지 않고 if 문 안에 바로 넣을 수도 있겠습니다. 그건 그렇고 이 함수에서는 받은 값을 그대로 Send IO Context 에 복사하고 Send를 요청합니다. 아 이 쪽을 보실 때에는 소스 코드를 한번 보시면서 읽어보세요. 그럼 다음에는 ProcessingThread 함수로 들어온다면 else if 로 들어가겠죠? 하지만 에코의 경우 Send 완료시에는 할 일이 없으므로 아무 일도 안 합니다.

그리고는 기억할 사항이 다시 Recv 를 요청해 두어야 한다는 사실입니다.

만약 다시 요청해두지 않는 다면 다음 번에 보내는 패킷을 받을 수 없습니다. 왜냐하면 IO 요청은 일회성입니다. 이걸 두 번 세 번 우려먹을 수 없습니다. 한번 요청하고 완료 결과를 받으면 그걸로 끝이지요. 그래서 마지막에 RecvPost해서 Recv 를 요청하는 겁니다.

자 그럼 정리해봅시다. Worker 쓰레드에서 하는 일을 무엇이던가요? 완료된 IO 결과를 뽑아

내고 그 내용을 분석하여서 그 분석된 결과에 따라 Send 면 Send, Recv면 Recv완료 처리를 하는 것입니다. 그래서 처음 프로그래밍 하실 때에는 초기 IO에 따른 흐름을 생각해보시면서 이건 이렇게 저건 저렇게 순서도를 그려놓으시고 그에 따라 프로그래밍 하시면 되겠습니다. 단 이것을 여러 번 불려지게 되므로 한번하고 말 그런 내용으로 프로그래밍하시면 안되겠죠? 여러 가지 상황에 대처할 수 있도록 프로그래밍 해주셔야 한다는 겁니다.

그래서 에코 서버 말고 좀 더 일반적인 상황에서 처리를 보면 recv 완료 후에 처리는 패킷(요 패킷은 데이터를 주고 받는 그 패킷입니다.) 받을 것을 보고 덜 받았다. 그럼 다시 recv를 요청합니다. 그리고 만약 다 받은 것 같다. 그럼 그 패킷에 따라 다른 작업을 하는 것입니다. 그리고 send 완료는 이번 완료된 send 는 제대로 다 간 것인가? 라는 것을 체크하고 아니라면 더 요청하는 식으로 합니다.

이렇게 IO 하나 처리에 따라 정보가 다 받아졌는지 덜 받아졌는지 체크를 해서 그에 따라 처리하는 루틴이 이 쪽 Worker Thread 쪽에서 할 일이 되겠습니다.

4. 정 리

이제 제 강좌를 총 정리하겠습니다. 첨에 우리는 IOCP에서 IO를 하는 중심이 되는 Overlapped IO라는 것을 알아 봤습니다. 이번 강좌까지 그 이 Overlapped IO 에 대한 말을 하면서 중요한 것을 한 마디로 정리하자면 Overlapped IO 라는 것이 **미리 요청한다. 그리고 그 결과는 나중에 받는다** 라는 것이었습니다. 그리고 2번째 강좌에서는 뭘 말했었나요? 흠…… 생각해보니 IOCP API를 정리하고 그에 대한 주변 이야기를 했었지요. 그리고 3번째 강좌는 IOCP 메커니즘 얘기였습니다. 그 5개 구조체와 그 사이의 관계. 이걸 알아두시면 소스 코드를 이해하시는데 큰 문젠 없을 겁니다. 그리고 마지막 이번 강좌에서는 간단한 예제를 보고 어떻게 프로그래밍 해나가는가에 대한 순서를 봤습니다. 아마 강좌 설명으로는 조금 부족할지도 모르겠습니다. 예제 소스 코드를 좀 더 보시면 될 것 같습니다. 일단 IOCP 핵심 3부분을 설명은 했으니 큰 어려움을 없을 것이라고 생각합니다. 그럼 이번 강좌를 끝으로 저의 강좌는 마치겠습니다.

드디어 허접한 강좌가 끝을 맺는군요. 그 동안 슬럼프에도 빠지고, 놀기만 하고 또 예제 코드도 제대로 작성해볼 거라고 노력 좀 했는데…… 역시나 예제도 제 생각엔 허접 코드이고 그러네요. 여하튼 저의 첫 번째 강좌를 종료하면서 여러분의 최종 평가를 듣고 싶네요. 나중에 혹시라도 강좌를 하면 참고하게 말입니다. 물론 그 때까지 기억할 수나 있을는지 모르겠지만요 ^^:

참 이번 소스 코드에 기본적인 쓰레드 풀링 외에 메모리 풀링도 넣고 커넥션 풀링도 넣어보려고 했습니다만 그럼 이해의 측면에서도 어려워 지고 막상 해놓으니 고치기 어렵더군요, 이 쪽은 다른 소스 코드를 보시면서 해보심이 좋을 듯 합니다.

그럼 다음 강좌를 한다면 그 때 뵙죠.