DEBUGGING GAME SERVER

Homepage: https://drv.tw/~sj6219@hotmail.com/od/Doc4Code/

Email: sj6219@hotmail.com

2021/3/6

여기에서는 게임 서버 프로그램을 디버깅하는 방법과 성능을 높이는 법에 대해서 알아보겠다. 디버깅을 하다 보면 서버의 병목 부분을 찾고 성능을 높이는 것까지 하게 된다.

보통 클라이언트 디버깅에는 익숙하고, 관련 서적이나 문서도 많이 찾아볼 수 있다. 클라이언트와 달리 서버를 디버깅할 때에는 프로그램을 한 줄 한 줄 실행해 가면서 디버깅할 수 없다. 그래서, 서버 디버깅의 경험이 없을 때는, 더 어렵다고 느끼기 쉽다.

그렇지만, 서버 디버깅에 대한 경험이 쌓이게 되면 클라이언트보다 편한 점도 많아진다. 서버 디버깅은 버그를 재현하기가 쉬운 장점이 있다. 어떤 버그라도 재현할 수만 있다면 찾아서 수정하는 것이 가능하다. 클라이언트에서는 특수한 경우에만 발생되는 버그가 있어서 재현하기가 어려운 경우가 많다. 서버의 경우에는 프로그래머가 버그를 재현하느라 수고할 필요가 없다. 의심되는 부분의 로그를 추가해 놓고, 다시 버그가 나올 때까지 기다리기만 하면 된다. 나중에 버그가 발생되면 또 로그를 추가해서, 조금씩 버그의 원인을 추적해 나가면 언젠가는 다 해결된다.

샘플 소스는

https://github.com/sj6219/EchoServer

에 있으니, 분석해보면 도움이 될 것이다.

EXCEPTION 처리

버그가 발생했을 때 그 원인을 찾는 것도 중요하지만, 버그를 찾기 쉽게 만드는 환경을 구축하는 게 더 중요하다. 버그가 발생해서 프로그램이 죽었을 때, 그 원인을 찾을 수 있는 시스템을 미리 만들어 놓아야 한다. 이 과정을 편리하게 만들어 놓을수록 디버깅이 수월해진다.

우선 Exception 이 발생했을 때, 그 정보를 자동적으로 Minidump 파일로 저장하는 기능이 있어야 한다. MiniDumpWriteDump 함수를 사용하면 간단히 Minidump 파일을 생성할 수 있다. 나중에 이 파일을 Visual Studio 나 Windbg 프로그램을 이용해 분석하면 버그의 원인을 알아낼 수 있다.

또, DbgHelp API 를 이용하면 디버그 정보를 직접 텍스트 형식으로 저장할 수도 있다. 텍스트 파일은 읽기가 편하기 때문에, 에러를 분석하는 시간을 줄일 수 있다. 텍스트 파일만 분석해서 원인을 찾아내는 경우가 많다. 그래서, 텍스트 파일과 Minidump 파일을 같이 생성해서, 먼저 텍스트 파일을 분석하고, 그걸로 부족한 경우 Minidump 파일을 이용하면 좋다.

다음은 실제로 생성된 디버그 텍스트 파일의 샘플이다.

샘플 프로그램에서는 Debug 메뉴 밑에 Exception 을 선택하면 Exception 폴더 안에 생성된다.

```
0xe80(3712) 2009/10/14 10:26:41> 'MainSvrT_CT' returns 0xc0000005
0xc24(3108) 2009/10/14 10:27:01> Execute 'MainSvrT_CT'
0xc24(3108) 2009/10/14 10:27:18> 'MainSvrT_CT' returns 0
0xf88(3976) 2009/10/14 10:27:26> Execute 'MainSvrT_CT'
0x8c0(2240) 2009/10/14 10:30:03> exception 0
start at 09/10/14 10:27:26
D:₩Immortal₩MainSvr₩MainSvrT_CT.exe, run by administrator.
2 processor(s), type 586.
2048 MBytes physical memory.
an Access Violation in module MainSvrT_CT.exe at 001b:0040EF15.
Read from location 00000000 caused an access violation.

Registers:
EAX=00000004 CS=001b EIP=0040ef15 EFLGS=00010202
EBX=00be6120 SS=0023 ESP=116cfa6c EBP=116cfa78
```

ECX=00000004 DS=0023 ESI=7c81be79 FS=003b
EDX=09ecc298 ES=0023 EDI=00000000 GS=0000

Bytes at CS:EIP:
8b 48 fc 89 4a 20 8b 55 fc 89 55 10 8b 45 10 83

Stack dump:
116CFA60: 00000000 00000000 00000000 09ecc298 04cfffd8 00000004 116cfc00 0040eb13
116CFA80: e5eacd9c 19d11eb3 00000000 3fffffff 2718ce50 0047d203 2718cb60 0000001e
116CFAA0: 00527cb4 000176cc 0000001d 000000fa 00000000 08ec1ac0 116cfacc 004e911e
116CFAC0: 0000000c 116cffa0 0000001d 116cfb14 0046d842 00000001 3fffffff 08ec1ad0
116CFAE0: 08ec1acc 08ec1ac8 2718ce50 08ec1ad0 08ec1ad0 08ec1acc 08ec1acc 09ecc298

Call Stack Information:

116CFA78 0040EF15 CBfVehicle::Init +35
d:₩project₩mainsvr₩buff.h 387 CBfVehicle::Init
Params: E5EACD9C 19D11EB3 00000000 3FFFFFFF
116CFBE0: 00ecfa18 116cfc08 0041226f 07eca4c8 187bc96a 04cfffd8 09ecc298 07d66888
116CFC00: 116cfc30 0040f4ff 0000002f e5eacd9c 19d11eb3 00000000 0043fb36 2718ce50

116CFC00 0040EB13 CBuffHandler::CreateBuff +c83
d:₩project₩mainsvr₩buff.cpp 429 CBuffHandler::CreateBuff
Params: 0000002F E5EACD9C 19D11EB3 00000000
116CFC20: 02ec8fb8 100693a8 00000000 00000000 116cfc5c 0040de79 2718cb60 0000002f

116CFC30 0040F4FF CBuffHandler::SetBuff +4f
d:₩project₩mainsvr₩buff.cpp 471 CBuffHandler::SetBuff
Params: 2718CB60 0000002F E5EACD9C 19D11EB3
116CFC40: e5eacd9c 19d11eb3 2718ce50 0000000f 19d11eb3 e5eacd9c 2f000003 116cfd08
116CFC60: 004811b9 2718cb60 000000cc 116cfcf8 0000003c 00000000 4ad5a464 00000000

116CFC5C 0040DE79 CBuffHandler::Load +89
d:₩project₩mainsvr₩buff.cpp 234 CBuffHandler::Load
Params: 2718CB60 000000CC 116CFCF8 0000003C
116CFCE0: 00000b26 00000000 00000362 00000a1a 00000fde 000006c4 13097ef1 00000000
116CFD00: 00000000 000000cc 116cfd88 004dc494 13097b50 116cfd80 116cfd44 005b591c

116CFD08 004811B9 CPlayer::OnLoadPc +639
d:₩project₩mainsvr₩player.cpp 3492 CPlayer::OnLoadPc
Params: 13097B50 116CFD80 116CFD44 005B591C
116CFD60: 07c1ea51 24613a2b 00525db7 116cfd80 004506f7 13097b50 116cfd98 0ecffa88
116CFD80: 00000005 2718cb60 116cff3c 0041d2f3 00000000 13097b50 00000004 00000009

116CFD88 004DC494 CSocket::OnLoadPc +b4
d:₩project₩mainsvr₩socket.cpp 949 CSocket::OnLoadPc
Params: 00000000 13097B50 00000004 00000009
116CFF00: 00428b8c 0000032a 005acdc0 00be6120 00001f40 2299f8f0 07c1e8e5 7c9677f9
116CFF20: 7c81bea2 00000364 116cff74 116cff60 116cff40 13097b50 00000004 116cff50

116CFF40: 0041cc02 13097b48 00d73508 00427936 116cff78 00427957 00000001 00000000

116CFF3C 0041D2F3 CDBSocket::Process +383
d:₩project₩mainsvr₩dbsocket.cpp 237 CDBSocket::Process
Params: 13097B48 00D73508 00427936 116CFF78

116CFF50 0041CC02 CDBPacket::OnIOCallback +12
d:₩project₩mainsvr₩dbsocket.cpp 26 CDBPacket::OnIOCallback
Params: 00000001 00000000 13097B48 00000000
116CFF60: 13097b48 00000000 00be6120 00000000 13097b48 00d73508 116cffb0 004ff479
116CFF80: 00000003 07c1e869 00000000 00be6120 00be6120 c0000005 116cff84 116cf69c

116CFF78 00427957 XIOSocket::IOThread +57
d:₩project₩mainsvr₩iolib.cpp 1296 XIOSocket::IOThread
Params: 00000003 07C1E869 00000000 00BE6120
116CFFA0: 116cffdc 004ff780 16f969b1 00000000 116cffec 004ff51e 00000000 7c82482f

116CFFB0 004FF479 _callthreadstartex +1b
f:₩dd₩vctools₩crt_bld₩self_x86₩crt₩src₩threadex.c 348 _callthreadstartex
Params: 00000000 7C82482F 00BE6120 00000000
116CFFC0: 00be6120 00000000 00000000 00be6120 c0000005 116cffc4 116cf698 ffffffff

116CFFB8 004FF51E _threadstartex +7f
f:₩dd₩vctools₩crt_bld₩self_x86₩crt₩src₩threadex.c 326 _threadstartex
Params: 00BE6120 00000000 00000000 00BE6120
116CFFE0: 7c821a60 7c824838 00000000 00000000 00000000 004ff49f 00be6120 00000000

116CFFEC 7C82482F GetModuleHandleA +df
C:₩WINDOWS₩system32₩kernel32.dll GetModuleHandleA
Params: 004FF49F 00BE6120 00000000 00000008

Module list:
D:₩Immortal₩MainSvr₩MainSvrT_CT.exe, loaded at 0x00400000 - 09/10/14 09:42:50 (KR)

위 파일을 보면 소스 몇 번째 줄에서 Exception 이 발생했는지 알 수 있다. 그리고, 텍스트 파일의 제일 마지막 줄을 보면, 그 서버 프로그램을 빌드한 날짜가 있다. 프로그램의 빌드 시각을 알면, 소스 제어 시스템을 이용해 빌드 당시의 소스도 구할 수 있다. 나중에 소스가 변경되었더라도 소스를 보면서 디버깅하는 게 가능하다.

버그가 발생했는지 주기적으로 검사하는 것보다는 자동적으로 알려 주는 시스템을 구축해 놓는 게 좋다. 그래서, 서버에 Exception 이 발생할 때마다 개발자의 메일 계정으로 디버그 텍스트 파일을 보내주는 기능이 있으면 편리하다.

또, 강제적으로 디버그 파일을 생성하는 기능이 있으면 매우 편리하다. 서버 프로그램의 메뉴 버튼을 눌렀을 때 디버그 파일을 생성하는 기능이 있으면, dead lock 이나 서버의 병목(bottle neck)부분을 찾아내는데 유용하다.

소스는 XLib/IOException.cpp 에 있다.

컴파일 옵션

Release 설정으로 빌드한 프로그램을 디버깅 하기란 쉽지 않다. Assembler 에 대한 사전지식도 있어야 하고, Register 에 들어있는 값까지 분석해야 한다. 버그 하나 분석하려면 몇 십 분씩 걸리기 쉽다.

그래서, 내가 즐겨 쓰는 방법은 컴파일할 때 최적화를 하지 않는 것이다. 많은 사람들이 최적화하지 않으면 실행 속도가 느려지는 것으로 착각하고 있다. 그러나, 실제 경험한 바에 의하면, 그다지 성능 저하를 느낄 수 없다.

소스 중 버그가 없고 안정화된 부분만 최적화하고 나머지 대부분을 최적화하지 않으면, 성능 저하를 최소화하면서 디버깅을 할 수 있다. 조금 황당하게 느끼겠지만, 실제 해보면 디버깅이 정말로 쉬워진다. 지금까지 해보지 않은 사람은 왜 이제야 알았을까 후회할 것이다. ㅎㅎ

다음은 내가 서버 프로그램을 컴파일할 때 사용하는 설정이다.

```
#define _HAS_EXCEPTIONS 0

Optimization : Disabled (/Od)
Enable C++ Exception : No
Inline Function Expansion : Any Suitable (/Ob2)
```

서버에서는 try-catch 문을 사용하는 경우가 별로 없다. 게다가, C++에서 Exception 기능을 끄면, 더 최적화된 코드를 얻을 수 있다.

Inline 함수 확장을 안 하면 디버깅하기에는 유리하지만, 코드의 성능이 떨어진다. 따라서, 가능하면 Inline 확장을 사용하는 게 좋다.

또, Linker 옵션에서 **Generate Debug Info: Yes(/Debug)** 를 하면 PDB 파일을 생성한다. 이 파일을 실행파일과 같은 디렉토리에 놔두면 디버그가 편해진다.

LOG & ASSERT

프로그램을 디버깅하는 가장 기본적인 방법은 printf 같은 함수를 사용해 로그(log)하는 것이다.

디버깅 환경에서만 로그하고 실행 환경에서는 로그하지 않는 경우가 있는데, 실행 환경에서도 로그하는 함수를 만들면 유용하게 사용할 수 있다.

로그와 더불어 ASSERT 문도 유용하다. 특히, 서버 디버깅에서는 중요한 데, 서버에서는 버그가 초기에 발견되어야 디버깅하기가 쉽다. 버그가 있는 체로 그대로 동작하다가 나중에 발현되면 서버의 상태가 뒤죽박죽이 되기 때문에 찾기가 어려워진다. 그래서, 가급적 버그를 빨리 찾는 게 중요하다. 의심스러운 부분에 ASSERT 를 이용해서 버그를 조기에 찾아낸다.

이 때, ASSERT 의 조건이 거짓으로 판명된 경우, 앞에서 설명한 Minidump 파일을 생성하도록 하고, 프로그램은 그대로 진행되도록 하면 좋다. 함수 호출 순서와 로컬 변수까지 조사할 수 있어서 버그 분석이 쉬워진다. 나는 처음 한번만 Minidump 파일을 생성하도록 하였다. 여러 번 버그가 중첩되어 발생되는 경우가 발생하면 첫 번째 에러만 분석하면 되기 때문이다.

소스에서 ELOG 와 EASSERT 를 참조해라.

DEADLOCK

DeadLock 이 발생했을 때, 그 원인을 찾아서 수정하기는 까다롭다. CPU 사용량이 0%일 때, 프로그램이 입력을 처리하지 못하면 Deadlock 을 의심해 보아야 한다. Deadlock 의 원인을 찾는 가장 좋은 방법은 강제로 Minidump 파일을 생성하는 것이다. 작업관리자나 Windbg 프로그램에도 이런 기능이 있지만, 서버 프로그램의 메뉴에서 Minidump 파일을 생성하는 게 편리하다.

그 다음 어느 Thread 가 어떤 Lock 을 기다리고 있는지 분석해보면 Deadlock 을 찾을 수 있다.

Deadlock 을 회피하기 위해서는 Lock 마다 우선 순위를 정하고, 그 순서대로 Lock 을 획득하도록 하는 것이다. 예를 들면, 유저간의 아이템을 물물교환 할 때, 각 Lock 의 주소를 비교해서 작은 주소부터 Lock 을 걸고, 나머지 Lock 을 걸도록 구현하면 deadlock 을 회피할 수 있다.

그리고, Monster 클래스와 User 클래스 둘 다 Lock 을 걸어야 할 때, 항상 Monster 클래스부터 Lock 을 걸면 Deadlock 이 생기지 않는다.

내가 지금까지 경험했던 디버깅 중에 가장 어려웠던 것은 병목현상 때문에 생긴 버그였다.
완전한 Deadlock 은 아니고, Dead Lock 과 비슷한 상태였었다. 이럴 때는 여러 번 Minidump 파일을 생성해서, 기다리고 있는 경우가 많은 Critical Section 을 찾아야 한다. 분석이 어렵기는 하지만, 고생하면 병목 부분을 찾을 수 있다.

XIOSocket::DumpStack()를 참조해라.

PERFORMANCE IMPROVEMENT

자 이제 성능을 개선하는 방법을 알아보자. 서버 프로그램에서 가장 흔한 병목(bottle neck) 부분은 Critical Section 같은 Lock 과 연관되어 있다. Lock 에서 병목이 발생하면 한 순간에 하나의 Thread 만 실행될 수도 있기 때문에 성능이 떨어진다.

우선 서버의 병목 부분이 어디인지 찾아야 한다. 가장 쉬운 방법은 실행 중인 프로그램의 Minidump 를 2-3 개 생성하는 것이다. 프로그램 자체에 그런 기능을 구현해도 되고, Microsoft 사의 windbg 프로그램에 있는 기능을 이용해도 된다.
샘플 프로그램에서는 Debug 메뉴 밑에 stack dump 를 선택하면 Exception 폴더 안에 덤프파일이 생성된다.

Minidump 를 분석해서 어떤 Lock 때문에 병목이 발생하는지 찾아본다. Thread 가 실행 상태에 있지 않고, Lock 이 풀리기를 기다리는 상태인 경우가 있다. 이렇게 어떤 Lock 을 얻기 위해서

Thread 간에 경쟁이 일어나서 기다리는 경우가 많다면, 프로그램의 로직을 변경해서 그 Lock 의 충돌을 줄이도록 하면 된다.

원인을 찾기가 어렵지, 원인만 알면 해결하는 것은 쉽다.

MALLOC

C++에서 기본적으로 제공되는 malloc 함수를 게임 서버에서 사용하는 것은 적합하지 않다. 이 함수는 범용으로 개발되었기 때문에, multi thread 와 고성능을 요구하는 특수한 환경에서는 다르게 구현되어야 한다.

예전에 서버를 만들었을 때 경험했던 일이다. 작업 관리자에서 보면 서버의 CPU 사용량이 다 차지도 않았는데, 더 이상 서버가 입력을 버티지 못하는 상황이었다. 나는 서버 프로그램을 개발하다가, 우연히 실행 Thread 수가 최대인 순간을 디버깅해 보았다. 그런데, 예상치 못한 현상을 발견하게 되었다. Thread 1 개를 제외하고, Thread 반쯤은 malloc 함수 내부를 실행 중이었고, 나머지 반은 free 함수 내부를 실행 중이었다.

malloc, free 함수 내부에서 사용하는 critical section 에서 병목 현상이 발생해서, 대부분의 thread 가 같은 Critical Section 을 기다리고 있는 것이었다. malloc, free 함수가 동시에 한 Critical Section 내부에서만 실행되도록 구현되어 있어서, 실제로는 single thread 처럼 느리게 동작했었다.

그래서, multi thread 에서도 성능이 좋은 malloc 을 개발하게 되었다. 나는 Visual Studio 에서 제공하는 malloc CRT 소스를 변형했다. 원래 malloc 함수를 class 의 멤버 함수로 변경하고, 그 class 의 인스턴스를 여러 개 만들어서, 돌아가면서 호출되도록 수정했다. 16 개의 메모리 풀을 사용해서, critical section 의 충돌 가능성을 줄였다.

```
class XIOMemory
{
```

```
public:
        void *   _malloc(size_t size);
        void     _free (void * pBlock);
        ...
};
struct HEAP_HEADER
{
        UINT_PTR        size;
        UINT_PTR        key;
};

#define MEMORY_HEAP_SIZE      16
#define KEY                   0x97f9e200

static XIOMemory g_memory[MEMORY_HEAP_SIZE];
static long g_nMemoryCount;

void *   _malloc( size_t size)
{

        DWORD nIndex = InterlockedIncrement(&g_nMemoryCount) &
                (MEMORY_HEAP_SIZE-1);
        HEAP_HEADER *pHeader = (HEAP_HEADER*) g_memory[nIndex]._malloc (
                size+ sizeof( HEAP_HEADER));
        if ( pHeader) {
                pHeader->size = size;
                pHeader->key = KEY + nIndex;
                memset(++pHeader, 0xfc, size);
                return pHeader;
        }
        else {
                ASSERT(0);
                return 0;
        }
}

void _free( void * pBlock)
{
        if (pBlock == NULL)
```

```
                return;
        HEAP_HEADER   *pHeader = ((HEAP_HEADER *)pBlock) - 1;
        HEAP_HEADER header = *pHeader;

        ASSERT(header.key & -MEMORY_HEAP_SIZE) == KEY);
        memset(pHeader, 0xfd, pHeader->size + sizeof( HEAP_HEADER));
        g_memory[header.key & ( MEMORY_HEAP_SIZE-1)]._free(pHeader);
}
```

그리고, 기존 malloc, free 대신에 새로 만든 함수를 호출하도록 변경하였다.

```
#define malloc( s)              _malloc( s)
#define free( p)                _free( p)

void * operator new( size_t cb)
{
        return _malloc( cb);
}

void operator delete( void * p)
{
        _free( p );
}
```

소스를 분석해보면 malloc 한 직후, 메모리의 내용을 0xfc 로 초기화한다. 그래서, 초기화가 아직 안 된 변수의 포인터를 액세스(access)할 때 Exception 이 발생되도록 한다. 또, free 직후에 메모리를 0xfd 로 바꾼다. 어떤 thread 에서 이미 free 된 메모리의 포인터 변수를 액세스를 한다면 그 순간 Exception 이 발생하게 된다. 이것은, 좀 더 빨리 버그가 발현되도록 도와준다. 나중에, 메모리 시스템이 깨진 후에 디버깅하는 것은 어렵기 때문에, 즉시 버그가 발생되도록 한 것이다. 실제 적용해보면 매우 유용하다는 것을 느낄 것이다.

소스는 XLib/IOMemory.cpp 를 참고해라.

CONTEXT SWITCHING

프로그램의 성능을 높이기 위해서는 하드웨어와 OS 시스템에 대한 기초 지식이 필요하다.

여기에서는 그 중 가장 중요한 Context switching 에 대하여 설명하겠다.

Context Switching 은 스택(stack)과 register 의 값을 다른 것으로 교환하는 작업을 한다. Context Switching 은 세가지로 분류할 수 있다.

- User Mode 와 Kernel Mode 의 switching
- Thread 간 Switching
- Process 간 Switching

OS 에는 User Mode 와 Kernel Mode 두 가지가 있다. 우리가 일반적으로 만드는 프로그램은 User Mode 에서 실행되는 것이다. 그리고, 디바이스 드라이버 같은 프로그램들은 Kernel Mode 에서 실행된다.

User Mode 에서 할 수 있는 일은 제한되어있다. Kernel Mode 에서만 다른 프로세스에 속한 메모리에 접근하거나, 입출력 디바이스에 접근할 수 있는 등 특별한 권한이 필요한 작업을 할 수 있다. 이렇게 두 가지 모드로 분리하여 응용 프로그램에서 OS 내부를 파괴시키는 것을 방지한다.

OS 는 각 thread 별로 두 개의 스택을 만든다. 하나는 User Mode 용 스택이고, 또 하나는 Kernel Mode 용 스택이다. malloc 같은 함수 내부를 분석하면 결국 VirtualAlloc 같은 Kernel 함수를 호출하게 된다. VirtualAlloc 함수를 호출하게 되면, 우선 Software Interrupt 를 발생시켜 User Mode 에서 Kernel Mode 로 바꾼다. 이 때 Interrupt 를 처리하는 코드에서 스택을 User 용에서 Kernel 용으로 바꾼다. 그 다음, 호출한 프로세스에 새로운 메모리를 할당한다. 메모리를 할당한 후에, 다시 Kernel Mode 에서 User Mode 로 돌아오게 된다.

또, 응용 프로그램 실행 도중 마우스 입력이 들어왔다면, Hardware Interrupt 가 발생된다. Hardware Interrupt 를 처리하는 코드에서 실행중인 thread 를 중단시키고 User Mode 에서 Kernel Mode 로 바꾼다. 그 다음 마우스 디바이스 드라이버에서 마우스 입력을 처리한다. 마우스 드라이버가 실행될 때는 원래 실행중이던 thread 의 kernel 용 스택을 사용한다. 입력이 처리된 후,

다시 Kernel Mode 에서 User Mode 로 바뀐다. 그리고, 원래 thread 의 중단된 곳부터 다시 시작된다.

이제 Thread 간 Context switching 에 대하여 알아보자. WaitForSingleObject 같은 함수 내부나 Timer Interrupt 를 처리하는 과정에서 Thread 전환이 일어난다. Thread 1 에서 Thread 2 로 전환되는 절차에 대해서 설명하겠다.

1 단계. 앞에서 설명한 것처럼 Thread 1 의 User Mode 에서 Kernel Mode 로 전환된다. 이 때, 스택도 Thread 1 의 Kernel 용 스택으로 바뀐다.

2 단계. 각 Thread 마다 TEB(Thread Environment Block)이라는 구조체를 가진다. 현재 register 의 값과 상태를 Thread 1 의 TEB 에 보관한다.

3 단계. Thread 2 의 TEB 에 보관되어 있는 reister 값을 읽어 들인다. 이제 Thread 1 의 Kernel Mode 에서 Thread 2 의 Kernel Mode 로 전환되었다.

4 단계. Thread 2 의 Kernel Mode 에서 User Mode 로 전환된다. 지난번에 Thread 2 가 중단된 지점부터 다시 실행되기 시작한다.

위의 내용은 Thread 1 과 Thread 2 가 같은 프로세스에 속할 때의 경우에 해당한다.

두 Thread 가 다른 프로세스라면 조금 더 복잡해진다.

우선 linear address 와 physical address 간의 관계를 이해해야 한다.

프로그램에서 사용하는 주소를 linear address 라고 부른다. 그리고, 실제 하드웨어상의 메모리 주소를 physical address 라고 부른다. 그런데, linear address 와 physical address 는 같지 않다.

하드웨어에서 linear address 를 physical address 로 자동 변환해 준다.

Unix 의 fork() 함수나 Memory Mapped I/O 에서는 이렇게 주소를 변환하는 기능을 이용해서 구현되어 있다.

그리고, 각 process 마다 PEB(Process Environment Block)이라는 구조체가 있는데, 이 PEB 내부에

Address 변환 테이블을 가지고 있다. 그래서, 앞에서 설명한 2 단계와 3 단계 사이에서 Address 변환 테이블을 전환해야 한다. Thread 2 가 속한 process 의 PEB 에서 주소 변환 테이블을 읽어 들인다.

또한, 하드웨어에는 주소 변환을 빠르게 하기 위해 TLB(Translation Lookaside Buffer)라는 cache 가 있다. Process 간 Context Switching 을 할 때에는 이 cache 에 있는 내용을 비우게 된다. 그러므로, Process 간 Context Switching 을 자주하는 것은 비효율적이다.

지금까지, Context switching 에 대해 간략하게 설명하였다. 불필요한 Context switching 은 프로그램의 성능을 저하시키므로, Context switching 을 줄이도록 노력해야 한다.

IOCP(IO COMPLETION PORT)

Microsoft 가 Context switching 을 줄이기 위해 개발한 것이 IOCP 이다. IIS 웹 서버를 개발하다가 Context switching 에서 CPU 가 낭비되는 것을 발견하고 개발하였다고 한다.

많은 사람들이 IOCP 를 Network API 로만 알고 있는데, Thread 를 효율적으로 관리해 주는데도 유용하다. PostQueuedCompletionStatus 함수를 사용하면 작업을 여러 개의 Thread 에 분배할 수 있다.

나는 IOCP Queue 를 처리하는 Thread 를 보통 8 개로 설정한다. 가장, 이상적인 경우에는 Thread 갯수와 processor 갯수를 같게 하는 게 좋다. Thread 의 숫자가 많으면 Context switching 때문에 오히려 성능이 떨어진다. 그렇지만, 실제로는 thread 가 입력을 처리하는 도중 I/O 나 Critical Section 같은 것 때문에 중단(suspend)될 수 있다. 그러므로, processor 보다 약간 많게 설정하는 게 좋다. 내 경우에는 minidump 파일 분석을 쉽게 하기 위해 가급적 적게 설정한다.

내가 만든 서버의 thread 는 다음과 같이 이루어져 있다.

- IOCP 처리용 Thread 8 개
- Event 와 Timer 처리용 Thread 1 개

- UI 담당 Thread 1 개

이렇게 총 10 개의 Thread 로 만들었다.

READ WRITE LOCK

어떤 데이터는 읽기는 많이 하지만 쓰기는 거의 안 하는 경우가 있다. 이런 경우에는 Read Write Lock 를 사용하면, Lock 의 충돌을 줄여서 성능이 좋아진다.

Read Write Lock 을 사용하면 Critical Section 을 사용했을 때보다는 확실한 성능향상을 느낄 수 있다.

소스에서 XRWLock 를 참조해라.

ADDREF, RELEASE

Multi-thread 프로그래밍에서는 AddRef, Release 를 사용해서 오브젝트의 반환(free)을 구현하는 경우가 많다.

```
class XIOObject
{
public:
        long m_nRef;

        void AddRef() { InterlockedIncrement(&m_nRef);}
        void Release() { if (InterlockedDecrement(&m_nRef) <= 0) OnFree(); }
```

```
        virtual void OnFree( ) { delete this; }
};
```

Reference Count 를 사용하면, 언제 오브젝트를 free 할 지 결정할 수 있다.

그런데, 여러 모듈에서 AddRef, Release 를 하면 버그를 찾기 어렵다.

예를 들어, 모듈 A 에서 다음과 같이 obj 를 액세스한다고 가정하자. 모듈 A 에는 잘못된 부분이 없다.

```
Function A()
{
…
A1:     obj->AddRef();
A2:     obj->func();
A3:     obj->Release();
…
}
```

그런데, 모듈 B 에 버그가 있어서, Release 를 한번 더 한다고 하자.

```
Function B()
{
…
B1:     obj->AddRef();
B2:     obj->Release();
B3:     obj->Release();
…
}
```

모듈 A 에서 A1 줄까지 실행되고, 모듈 B 가 실행된다면, B3 줄에서 메모리가 free 된다.

그 후, 모듈 A 가 다시 실행되면 A2 줄(또는 A3 줄)에서 Access Violation 버그가 발생된다. 하지만, 버그가 발생된 곳(모듈 A)에는 잘못된 코드가 없기 때문에, 버그의 원인을 찾을 수가 없다. 나도 이것 때문에 오랫동안 고생했었다. 특히, 여러 사람이 협동해 코딩하는 경우 더 찾기가 어렵다.

그래서, 모듈 별로 따로 Reference Count 를 따로 관리하도록 수정했다.

```
class XIOObject
{
public:

	long m_nRef;
	long m_nSystemRef;
	long m_nGeneralRef;
	long m_nTempRef;

	XIOObject() : m_nRef(1), m_nGeneralRef(0), m_nSystemRef(1), m_nTempRef(0) {}

	void AddRef() { _AddRef(&m_nGeneralRef); }
	void Release() { _Release(&m_nGeneralRef); }
	void AddRefSystem() { _AddRef(&m_nSystemRef); }
	void ReleaseSystem() { _Release(&m_nSystemRef); }
	void AddRefTemp() { _AddRef(&m_nTempRef); }
	void ReleaseTemp() { _Release(&m_nTempRef); }

	void _AddRef(LONG volatile *pRef)
	{
		if (InterlockedIncrement(&m_nRef) == 1) {
			ASSERT(0);
			m_nRef = 100;
		}
		InterlockedIncrement(pRef);
	}
	void _Release(LONG volatile *pRef)
	{
		long nRef1 = InterlockedDecrement(pRef);
		if (nRef1 < 0)
```

```
                {
                        LOG(_T("XIOObject::Release %p %p %x %d %d"), this,
                                *( DWORD *)this, ( char *)pRef - ( char *)this, nRef1,
                                m_nRef);
                        ASSERT(0);
                        *pRef = 100;
                        m_nRef = 100;
                        return;
                }
                long nRef2 = InterlockedDecrement(&m_nRef);
                if (nRef2 > 0)
                        return;
                OnFree();
        }
        ...
};
```

이제 모듈 B 의 B3 줄을 실행하는 순간 _Release 의 ASSERT(0)문에 걸리게 된다.

```
Function A()
{
...
A1:     obj->AddRefSystem();
A2:     obj->func();
A3:     obj->ReleaseSystem();
...
}

Function B()
{
...
B1:     obj->AddRefTemp();
B2:     obj->ReleaseTemp();
B3:     obj->ReleaseTemp();
...
}
```

MEMORY LEAKAGE

함수 malloc() 또는 new 를 이용해 메모리를 사용하다가 반환 안 하는 실수를 하는 경우가 있다. 이런 식으로 누출된 메모리가 많아지면 결국 서버에 문제가 생길 것이다. 그래서 메모리가 제대로 반환되는지 검사해 봐야 한다. 유용한 방법은 디버그 버전을 실행하면서 프로그램이 끝났을 때 메모리가 모두 반환되었는 지 확인하는 것이다. Microsoft 에서 제공하는_CRTDBG_MAP_ALLOC 과 `_CrtSetDbgFlag` 기능을 이용하면 좋다.

또한, 주기적으로 메모리 총 사용량을 화면에 표시해서 메모리 유출이 일어나는지 확인한다.

EchoServer₩Status.cpp 에서 XMemory::GetTotalSize()를 호출하는 부분을 참고해라.

버퍼 크기 제어 (FLOW-CONTROL)

프로그램을 구현할 때, 중간 단계에서 사용하는 버퍼의 크기가 너무 커지게 되면 이상한 현상이 발생하고, 이럴 때, 그 원인을 찾기가 쉽지 않다. 이 때, 흐름을 제어해서 버퍼 크기를 조절해야 한다. 즉 버퍼의 크기가 일정 수준 이상 커지면 더 이상 입력을 받지 않게 설계해야 한다. 최소한 버퍼의 크기를 모니터 하는 시스템을 준비해서 버그가 발생했을 때, 그 원인을 찾을 수 있게 해야 한다.

Flow-Control 은 디버깅하기가 어려우므로, 문제가 발생하기 전에 설계 단계에서 미리 준비해야 한다. 버퍼가 넘치는 것을 대비하지 않는 경우가 있는데, 문제가 생기면 원인 자체를 알 수 없으므로 주의해야 한다.

우선 서버에서 클라이언트로 보내는 경우에 Flow Control 에 대해 생각해 보자.

네트워크에서 처리하는 속도보다 서버에서 보내는 데이터가 많다면 결국 Buffer Overflow 가 발생하게 된다. 그래서, 내 경우에는 서버의 출력 Buffer 가 가변적으로 커질 수 있게 구현하였다.

그러다가, 일정 사이즈 이상 버퍼가 늘어나면, 로그를 기록하고 네트워크를 강제적으로 끊었다. 나중에 로그를 분석하면 네트워크에 문제가 있었는 지 알 수 있다.

가장 흔하게 범하는 오류로 SQL 서버의 작업 처리 속도보다 입력 속도가 빠른 경우에 대해 생각해보자. 결국 서버에 문제가 생기게 되는데, 더 심각한 것은 왜 버그가 발생했는지 원인을 모른다는 것이다. SQL 쪽에 문제가 있다는 것을 알아야 고칠 텐데, SQL 이 작업을 안 하는 게 아니라 느린 것이므로 문제가 있다는 것을 알기 어렵다. 대부분의 돈 복사 버그도 이것 때문에 일어난다.

문제를 찾기 위해서는, SQL 에서 처리되기를 기다리고 있는 입력 Queue 의 크기를 모니터해야 한다. 그래서, 나는 DB 서버 화면에서 입력 Queue 의 크기를 화면에 보여주는 기능을 만들었다. 그리고, 입력 큐의 크기가 너무 커지면, 아예 사용자 접속을 막아서 더 커지지 않도록 하는 게 좋다.

또, DB 서버로 입력을 보내는 다른 서버에서도 출력 Buffer 의 크기를 화면에 보여주는 기능도 만들었다. DB 서버의 처리속도가 느려져서, 네트워크 입력을 느리게 읽으면 문제가 발생하기 때문이다.

이렇게 문제가 발생할 소지가 있는 모든 버퍼에 대해 모니터링을 해야 한다. 내 경우에는 UI 전용 쓰레드에서 각종 버퍼의 크기를 화면에 표시하도록 처리하였다. 하지만, 서버 설계 당시에 이를 고려하지 않아서 화면에 표시하기 어렵다면, Microsoft 에서 제공하는 Performance counter 등을 이용하는 것도 좋겠다.

Performance counter 는 EchoServer₩Status.cpp 를 참조해라.

DB(DATABASE) 서버

서버를 기능별로 분산해서 처리하면 서버의 성능을 향상시킬 수 있다. 하지만, 서버가 많아지면 디버깅하기가 어려워 개발이 복잡해지는 단점이 있다.

DB 서버의 경우 분리하면 여러 가지 장점이 있다.

첫째는 SQL 서버를 외부 Network 에 노출되지 않아서 보안상 안전하다는 점이다. 직접 SQL 서버를 Network 에 노출시키는 것보다는 DB 서버를 통하여 접근하는 게 더 안전하다.

둘째 이유는 성능을 올리기 위해서이다. SQL 작업의 경우 처리 시간이 오래 걸리고 응답이 올 때까지 기다려야 한다. 그래서, SQL 작업을 처리하는 별도의 서버를 만들고 Event-driven 방식으로 구현하면 성능이 올라간다. DB Server 의 경우 작업당 처리 시간이 더 길기 때문에 Thread 의 개수를 20-30 개 정도로 늘려주는 게 좋다.

REMOTE DEBUGGING

개발용 기계에서는 잘 돌아가다가, 실제 서버에서만 에러가 발생하는 경우가 있다. 이럴 때는 Remote Debugging 를 사용하는 게 좋다. Visual Studio 의 Remote Debugging 을 하면 편리하다. 네트워크 상황이 안 좋은 경우나 Firewall 에 막혀 있을 경우에는 Windbg 프로그램이 더 좋다. Visual Studio 가 사용하기에는 편리하지만, Windbg 에 더 많은 기능이 있으므로 둘 다 사용법을 알아두도록 하자.

다른 프로세스에 의해 프로그램 시작 중에 에러가 발생하는 경우가 있다. 이럴 경우엔 MS 가 제공하는 gflags.exe 을 이용하여 디버깅 할 수 있다. 그게 여의치 않으면 프로그램 시작시에

```
for (int loop=0; !loop;)
        Sleep(1000);
```

를 첨가해서 실행한다. 디버거에서 attach 한 후, loop 변수의 값을 1 로 바꾸면 된다.

참고 서적

[1] John Robbins. Debugging Applications for Microsoft . Net and Microsoft Windows

이 책에는 Minidump 파일을 생성하는 법에 대해 자세히 나와있다.

[2] Mario Hewardt. Advanced Windows Debugging.

Windbg 를 사용하는 방법에 대해 나와있다. 특히, 고급 디버깅 기술에 대해 설명하고 있다.

[3] Daniel P. Bovet. Understanding the Linux Kernel, Third Edition

OS 에 대해서 알아보는 가장 확실한 방법은 소스를 분석하는 것이다. 혼자서 소스를 분석하기는 어렵고, 이 책을 보면서 분석하면 더 쉽게 할 수 있다.