NT 의 Performance Counter 를 이용한 시스템 성능 평가

(1/2) NT 의 성능 평가단, 퍼포먼스 카운터!
(2/2) 보다 진보된 도구, PDH 를 사용하자.

김진홍, aquakim@gmail.com

본 문서는 2000 년 마이크로소프트 월간지에 기고된 것이며, 다른 사람들이 참고할 수 있도록 문서화 한 것입니다. 문서의 내용을 변경하지 않은 상태에서 재 배포 가능합니다.

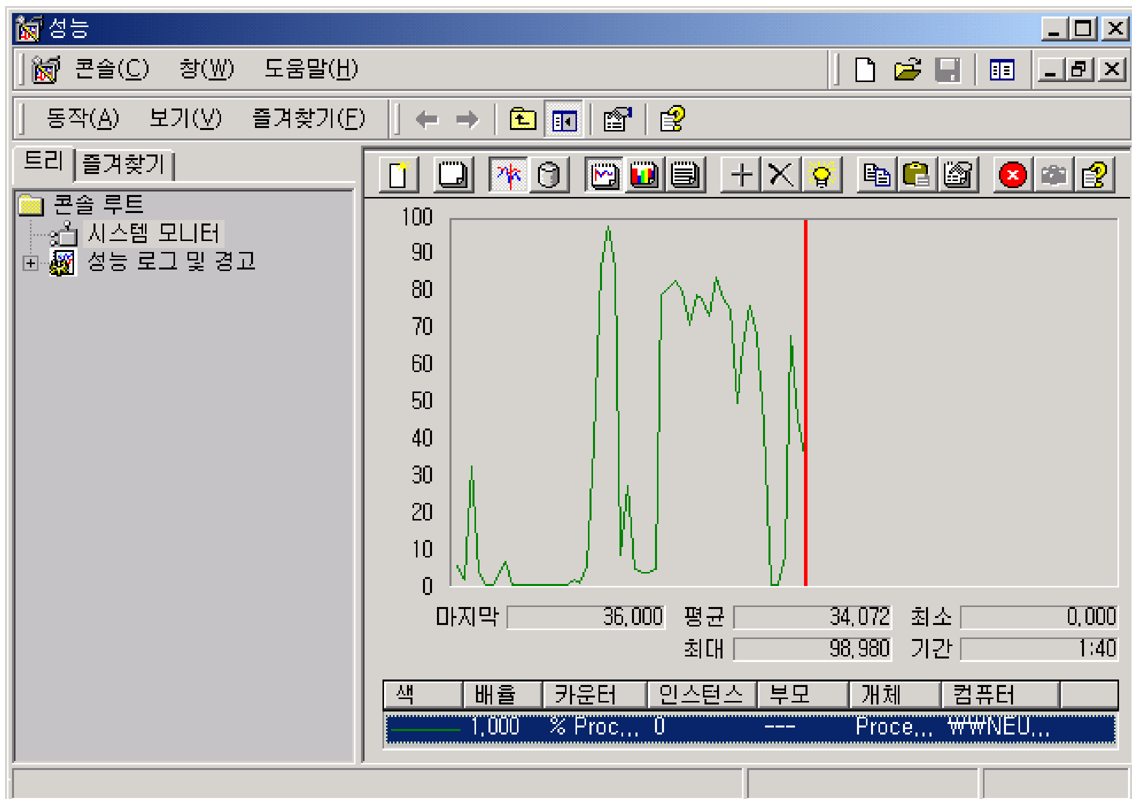
제 1 회: NT 의 성능 평가단, 퍼포먼스 카운터!

고성능의 소프트웨어를 제작하기 위해서는 시스템의 성능을 나타내는 세부적인 상태들을 분석해낼 줄 알아야 한다. 우리는 Windows NT 로 부터 운영체제와 주변장치들에 대한 세부 상태 정보를 얻어낼 수 있다. 실제로 NT 의 성능 데이타를 얻어보는 과정을 실습하고 더불어 시스템을 이해하는데 도움이 되도록 그 내부를 파해쳐 본다.

필자는 현재 네트워킹 모듈을 제작하고 있다. 서버용 네트워킹 모듈을 작성하면서, 호스트에 걸리는 각종 부하(Load)율을 알아볼 필요가 있었다. 시스템 부하율을 검사하기위해 가장 먼저 알아내려고 했던것은 바로, 시스템에 부착된 프로세서의 부하율과 네트웍 장비 자체의 부하율이었고, 특히 네트웍 장비가 처리한 IP(Internet Packet)와 애플리케이션으로 전송되는 TCP 혹은 UDP 의 갯수가 초당 얼마나 되는지 알고 싶었다. 그러나 우습게도 당연히 API 로 존재할거라 믿었던 프로세서의 부하율에 대한 API 마저 찾을 수 없었다.

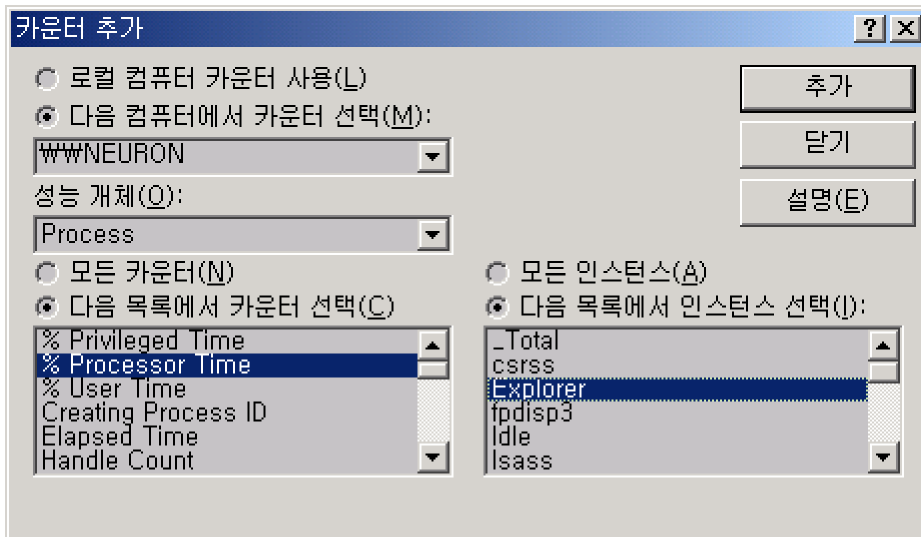
여기서는 NT 시스템의 정보를 동적인 성능에 관계된 소프트웨어에 사용하려고 프로그래밍을 하지만, 단순히 시스템의 상태를 관찰하려고만 한다면 반드시 복잡한 프로그래밍을 거칠 필요는 없다. NT 에서는 다음호에 소개할 PDH 라이브러리를 내장한 소프트웨어를 이미 제공하고 있기 때문이다(<화면 1> 참고). [시작] 단추를 누르면 나오는 '실행' 메뉴에서 'perfmon' 이라고 입력하면 제공되는 소프트웨어를 바로 사용할 수 있다. 우리는 Perfmon 을 통해 간단히 시스템의 다양한 정보를 얻을 수 있다. 예를들어 Perfmon 의 chart 에 http 나 IIS 를 추가하면 Web Server 의 성능을 관찰할 수 있다. Perfmon 말고도 태스크 매니저
(Left Ctrl + Left Shft + ESC)인 'taskmgr'과 시스템 정보를 표시해주는 'winmsd' 를 이용해 보다 다양한 정보를 얻을 수 있다. Windows 2000 에도 화면은 조금 다르지만 동일한 도구들이 있다. 조금은 다른 이야기지만 NT 의 성능에 관계된 각종 Tip 들을 얻으려면 아래 site 를 방문해 보기 바란다.

http://www.tuningandsizingnt.com/ NT의 성능에 관한 다양하고 전문적인 정보를 얻을 수 있다. 우리가 배우려는 Performance Counter를 주로 이용하고 있다.	
http://www.ntpro.org/techtips.html NT에 관한 Tip들이 있다.	
http://www.pureperformance.com 역시, Win9X, NT, Win2000 에 관한 Tip들이 있다.	



<화면 1> Performance Monitor

<Performance Counter>
결국 MSDN 과 한참을 씨름끝에, Performance Data Helper(이하 PDH)라는 라이브러리를 발견했고, 그 PDH 또한 시스템의 Registry 를 통해 Windows NT(이하 NT)와 시스템의 성능에 관계된 각종 – 거의 모든 – 정보를 얻어낸다는것을 알았다. NT 는 시스템과 운영체제의 성능에 관계되는 모든 정보를 Performance Counter 라는 구조체에 담아 Registry 를 통해 사용자로 전달한다. 이것은 NT 와 Windows 2000 시리즈에서 가능하다.



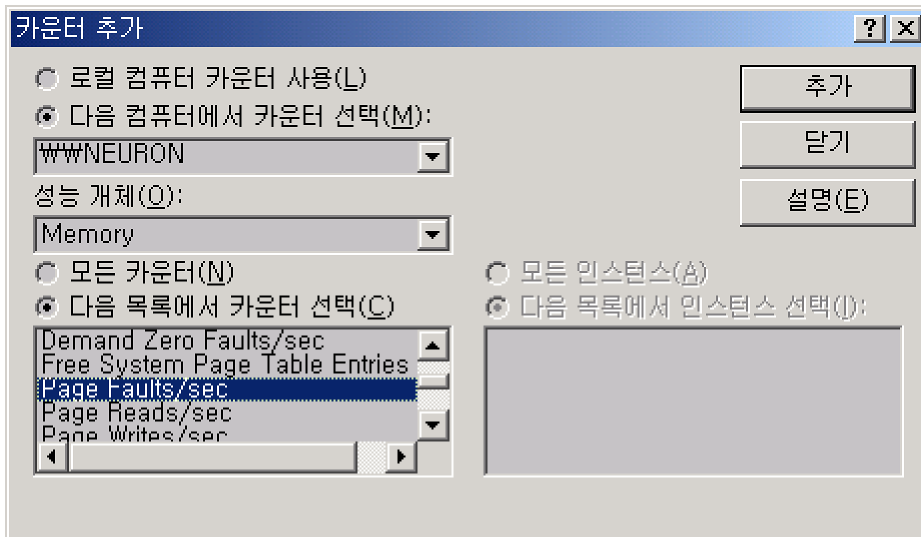
<화면 2> PDH 가 제공하는 기본 Dialog Interface

<PDH>
PDH 는 시스템의 성능에 관계된 정보를 관찰하는데 요구되는 복잡한 절차들을 사용자가 쉽게 사용할 수 있도록 꾸며놓은 Helper 라이브러리이다. PDH 는 Performance Monitor 를 위한 각종 함수들을 모아놓은 함수들의 집합이기도 하지만 <화면 2>와 같은 기본 PDH Dialog Interface 를 자체 지원하고 있으며 Visual Basic 에서도 사용 가능하도록 설계되어 있다(MSJ 1998 3 월호, UnderTheHood 참고). 이제 곧 알게 되겠지만 일반인들이 자신의 소프트웨어를 위해 Performance Counter 를 저수준 제어하는 것은 배보다 배꼽이 더 크게 되는 격이며 그리 만만하지도 않다. PDH 는 그러한 까다로운 저수준 제어를 대신해주고 대신 추상화된 고수준 인터페이스를 제공한다.

그러나 이번호에서는 고급 라이브러리인 PDH 를 이용하지않고, 복잡하지만 Performance Counter 가 무엇인지 그리고 어떻게 이루어져 있는지를 배우는 시간으로 하겠다. 다음호에서는 여기서 배운 Performance Counter 에 대한 지식을 바탕으로 PDH 를 사용해 보도록 하겠다.

<Windows NT Performance Counters>
시스템 성능 측정의 척도로 제공되는 Performance Counter 는 Cache, ICMP, IP, Logical Disk, Memory, NetBEUI, Network Interface, NWLink IPX, Objects, Paging File, Physical Disk, Process, Processor, RAS, Redirector, System, TCP, Thread, UDP 등을 커버하며 (나머지는 MSDN, 'Performance Data Helper' 를 참고) 이들을 Object 라 부른다. 각 Object 는 그것과 관련된 세부 정보를 담고 있는 Instance 와 Counter 들로 이루어져 있다.

<Instance>
Instance 는 Object Type 의 복사본이다. 예를들어 여러분이 가진 하드드라이브(Physical Disk Object)는 두개 이상일 수 있기 때문에 Multiple-Instance 를 지원한다. 하지만 메모리(Memory Object)라는 것은 하나뿐이기 때문에 Single-Instance 만을 지원한다. Instance 는 시간에 따라 변한다. 프로세스를 예로 들면 현재 시스템에 존재하는 많은 프로세스들은 모두 Process Object 의 Instance 이다. (이것으로 현재 활동중인 Process 들의 목록을 알 수 있다.) Counter 는 각 Instance 마다 존재하며 실제 성능 자료를 가지고 있는 곳이다. (Single-Instance 에서는 Instance 라는 개념없이 바로 Counter 에 접근한다.)



<화면 3> Single-Instance 인 Memory Object

<성능 관찰 방법>
NT 에서 대부분의 Performance Counter 는 계속 값이 증가하며 다시 0 으로 클리어되지 않는 형태 이므로 일반적으로 Performance Counter 를 이용해 성능을 관찰 하려면 일정한 시간간격의 시작과 끝에서 각각 한번씩 Counter 를 읽고 그 둘간의 차이를 Counter 마다 정해져 있는 특별한 방식으로 계산해야 한다.

<프로세서의 부하율>
예를들어 프로세서의 부하율을 계산해 보자. 운영체제는 프로세서마다 Idle 쓰레드를 할당해 다른 쓰레드가 실행될 준비가 되어있지않으면 Idle 쓰레드를 실행한다. 그런데 Idle 쓰레드가 전혀 실행되지 않는다면 그것은 프로세서가 모든 쓰레드들에 의해 100% 활용된다는 것을 의미한다. 이것은 프로세서의 부하율이 100% 이상이며 여기서 더 많은 쓰레드나 프로세스가 생겨나게 되면 전체적으로 시스템의 성능이 떨어질 수 밖에 없다. (실제로 부하율이 60% 정도를 넘어서면 긴 계산을 필요로 하는 애플리케이션은 상당히 느려지게 된다.) 하지만 Idle 쓰레드를 실행하는데에만 프로세서 Cycle 이 100% 소비된다면, 그것은 프로세서의 부하율(혹은 활용율)이 0%에 가깝다는것을 의미한다. 우리가 원하는 이 부하율에 해당하는 Counter 는 Processor Object 의 '0' Instance(첫번째 CPU 를 의미)에 있는 '% Processor Time' 이다.
'% Processor Time' 에는 운영체제가 Idle 쓰레드를 실행할때 100ns(Nano-Second, 10 억분의 1 초) 마다 1 씩 더해진 64 비트 정수 값이 들어있다. 이를 64 비트 변수 A 에 넣고 1 초가 지난후 다시 B 에 넣은 후 B 에서 A 를 빼면(카운터는 계속 증가) 1 초 동안에 Idle 쓰레드를 실행하는데 소비된 카운트 수를 알 수 있다. 이 카운트 수를 10,000,000(1 초동안의 카운트) 으로 나누면 Idle 쓰레드를 실행하는데 소요된 시간을 알 수 있다. 이 시간에 100 을 곱하면 백분율이 나오고 이 값을 100 에서 뺀것이 바로 우리가 지금까지 알고자 했던 '프로세서의 부하율' 이다.

NT 는 Performance Counter 들의 정보를 Registry 를 통해 애플리케이션으로 전달한다. 그러나 실제로 그것들이 Registry 의 데이타베이스에 저장되고 있는것은 아니다.

<리스트 1> Performance Counter 들의 ID 와 이름 얻기

```
dwBufferSize = 8192;
```

```
pStrbuf = (TCHAR*)malloc(dwBufferSize);

/* HKEY_PERFORMANCE_DATA, 이 키는 미리 Open 할 필요가 없다. */
while ( RegQueryValueEx(HKEY_PERFORMANCE_DATA,
            TEXT("Counter"), /* Counter 의 ID 와 이름을 요구 */
            NULL,
            NULL,
            (LPBYTE)pStrbuf,
            &dwBufferSize) == ERROR_MORE_DATA)
{
dwBufferSize += 1024;
pStrbuf = (TCHAR*)realloc(pStrbuf, dwBufferSize);
}
```

<Counter ID>
우리가 실제로 Counter 의 값을 읽으려고 NT 에 요구하려면 NT 에게 그 Counter 의 ID 가 무엇인지 전달해 주어야 한다. 하지만 그 ID 를 모를 경우 NT 를 뒤져(?) 보아야 한다. <리스트 1> 을 이용해 현재 시스템에 존재하는 Counter 들과 해당 ID 의 총 목록을 볼 수 있다. 리스트를 보자. 먼저 데이타를 얻기 위해 필요한 버퍼를 잡아야 한다. 그런데 NT 는 필요한 버퍼의 크기가 얼마나 되는지 알려주지 않는다. 다만 데이타를 넣기에 버퍼가 모자란 경우 ERROR_MORE_DATA 라는 에러값을 리턴해줄 뿐이다. 우리는 리스트에서 처럼 ERROR_MORE_DATA 에러가 나지 않을때까지 버퍼의 크기를 계속 늘려주어야 한다. RegQueryValueEx() 함수의 두번째 파라메타에 "Counter" 라는 문자열을 넣은것을 볼 수 있다. 그것은 NT 에게 Performance Counter 들의 ID 와 이름을 달라는 키 이다. NT 는 운영체제가 지원하는 언어가 무엇이든, NULL 로 끝나는 Unicode 문자열의 배열(REG_MULTI_SZ 형식)을 버퍼에 담아 돌려준다. 버퍼에는 아래와 같은 포맷으로 이루어져 있다.

<ID>₩0<카운터이름>₩0<ID>₩0<카운터이름>₩0<ID>...₩0<카운터이름>₩0₩0

뒤에서 이야기 하지만, 역시 RegQueryValueEx() 함수의 두번째 파라메타를 통해 원하는 Performance Counter 를 요구하게 된다. 하지만 카운터의 ID 로만 Counter 를 지정할 수 있으므로, 원하는 Counter 를 얻으려면 그것의 ID 를 알아야 한다.

<리스트 2> Performance Counter 의 ID 와 이름 출력

```
TCHAR **pCounterNames; /* 카운터들의 이름을 저장 */
TCHAR **pCounterIDs;      /* 카운터들의 ID 를 저장 */
DWORD dwTracks[5000]; /* 카운터 이름의 위치를 저장. 카운터의 수가 5000 개를 넘지 않는다고 가정 */
DWORD dwTotalTrack;

/* 총 카운터의 수가 몇개인지 돌려줌 */
int GetNumOfCounters()
{
    HKEY hKeyPerflib;
    DWORD dwCounters;
    DWORD dwSize = sizeof(dwCounters);

    /* Performance Counter 의 몇가지 정보가 저장된 곳 */
    RegOpenKeyEx( HKEY_LOCAL_MACHINE,
        TEXT("SOFTWARE₩₩Microsoft₩₩Windows NT₩₩CurrentVersion₩₩Perflib"),
        0,
        KEY_READ,
        &hKeyPerflib);

    RegQueryValueEx( hKeyPerflib,
        TEXT("Last Counter"),                    /* 총 카운터의 수 */
```

```
            NULL,
            NULL,
            (LPBYTE)&dwCounters,
            &dwSize);

      RegCloseKey( hKeyPerflib );
      return dwCounters;
}


void main()
{
... (생략)
dwCounter = GetNumOfCounters();
pCounterNames = (TCHAR**)malloc(sizeof(TCHAR*)*dwCounter);
pCounterIDs = (TCHAR**)malloc(sizeof(TCHAR*)*dwCounter);

dwTotalTrack = 0;
for( pCurrentString = pStrbuf; *pCurrentString;
        pCurrentString += (lstrlen((LPCTSTR)pCurrentString)+1) )
{
            dwCounter = _ttoi(pCurrentString);          /* 카운터 ID */
             dwTracks[dwTotalTrack++] = dwCounter; /* 밑에서 카운터 ID 를 Index 로 한 위치에 카운터의 ID 와 이
름을 넣게됨으로, 그 위치를 dwTracks 배열에 기록해 놓는다. */
             pCounterIDs[dwCounter] = pCurrentString; /* 카운터 ID 저장 */
            pCurrentString += (lstrlen((LPCTSTR)pCurrentString)+1); /* ID 다음의 위치로 이동*/
             pCounterNames[dwCounter] = pCurrentString; /* 카운터 이름 저장 */
}
...(생략)
```

<리스트 2>의 pCounterIDs[]와 pCounterNames[]를 출력하면 Counter 의 ID 와 이름을 알 수 있다. 중요한 것은 Counter 의 이름에 해당하는 ID 이다. 이제 그 ID 를 이용해 원하는 Counter 의 실제 정보를 얻어보자. 참고로 디버깅시 WatchWindow 에서 Unicode 문자열을 보기 위해서는 Unicode 변수 뒤에 ", su" 라고 쓰면 된다.

<리스트 3> Counter 정보 얻기

```
dwBufferSize = 8192;
PerfData = (PPERF_DATA_BLOCK) malloc( dwBufferSize );

while( RegQueryValueEx( HKEY_PERFORMANCE_DATA,
                                    TEXT("원하는 카운터의 ID")
                                    NULL,
                                    NULL,
                                    (LPBYTE)PerfData,
                                    &dwBufferSize ) == ERROR_MORE_DATA )
{
        dwBufferSize += 1024;
        PerfData = (PPERF_DATA_BLOCK) realloc( PerfData, dwBufferSize );
}
```

<또 다른 의미의 Key>

<리스트 3> 에서 처럼, 원하는 Counter 의 ID 를 넣어주면 그 Counter Object 의 Instance 와 Counter Data 들에 대한 각종 정보를 'Performance Data Format' 형태(<그림 1> 참조)로 버퍼에 저장해 준다. 예를들어 Processor Object 에 대한 정보를 얻고 싶다면, TEXT("238") 을 그 두번째 파라메타에 넣어주면 된다. 물론 이 ID 는 <리스트 1,2>를 통해 얻을 수 있다. 참고로 ID 대신 특별한 의미의 Key 를 대신 줄 수도 있으며 다음과 같다.

TEXT("Counter") : 모든 Counter Object, Instance, Counter 들의 이름을 요구.

TEXT(“Global”) : 모든 Counter Object 들의 Counter 데이타를 요구.
TEXT(“ID1 ID2 ID3…”) : 한번에 여러개의 Counter 데이타를 요구.
TEXT(“Foreign 아무개 ID1 ID2…”) : ‘아무개’ 호스트에서 한번에 여러개의 데이타를 요구.

<그림 1> Performance Data Format

버퍼의 전체 구조

PERF_DATA_BLOCK
Object 1
Object 2
…
Object N

Object 의 구조: Multiple-Instance 의 경우

PERF_OBJECT_TYPE
PERF_COUNTER_DEFINITION 1
PERF_COUNTER_DEFINITION 2
…
PERF_COUNTER_DEFINITION N
Instance 1
Instance 2
…
Instance M

Object 의 구조: Single-Instance 의 경우

PERF_OBJECT_TYPE
PERF_COUNTER_DEFINITION 1
PERF_COUNTER_DEFINITION 2
…
PERF_COUNTER_DEFINITION N
PERF_COUNTER_BLOCK
Counter 1 Data
Counter 2 Data
…
Counter N Data

Instance 의 구조

PERF_INSTANCE_DEFINITON
Instance Name
PERF_COUNTER_BLOCK
Counter 1 Data
Counter 2 Data
…
Counter N Data

음영 처리된 각 Structure 는 winperf.h 라는 파일에 정의되어 있으며 아주 자세한 코멘트가 달려 있으니 확인해 보기 바란다.

실제 Counter 데이타를 읽기 전에 그 자료가 어떤 구조로 되어 있는지 <그림 1>을 살펴

보자.

<버퍼 구조>
먼저 '버퍼의 전체 구조'를 보자. 우리가 요구한 Counter Object 에 대해 NT 가 돌려준 버퍼의 구조는 그것과 같으며, 그 전체 구조의 헤더에 해당하는 PERF_DATA_BLOCK 부분을 읽으면 필요한 정보를 알 수 있다. 우리는 버퍼에서 각 Object 에 해당하는 위치로 정확히 위치를 이동해야 하므로 각 구조체가 가지고 있는 정보를 이용해야 한다. 먼저 이 버퍼에서 첫번째 Object 를 찾아보자. 구조를 보면 버퍼의 시작지점에서 PERF_DATA_BLOCK 구조체의 크기만큼만 건너뛰면 바로 첫번째 Counter Object 를 만날 수 있다.

```
/* 첫번째 Object 로 이동 */
PPERF_OBJECT_TYPE FirstObject( PPERF_DATA_BLOCK PerfData )
{
    /* PERF_DATA_BLOCK 구조체 길이만큼 뒤로 이동 */
    return( (PPERF_OBJECT_TYPE)((PBYTE)PerfData + PerfData->HeaderLength) );
}
```

그 다음의 Object 는 그 Object 의 길이를 알면 찾을 수 있을것이다. 그것은 Object 블럭의 헤더인 PERF_OBJECT_TYPE 에 있다.

```
/* 다음 Object 로 이동 */
PPERF_OBJECT_TYPE NextObject( PPERF_OBJECT_TYPE PerfObj )
{
    return( (PPERF_OBJECT_TYPE)((PBYTE)PerfObj + PerfObj->TotalByteLength) );
}
```

우리가 탐색해야 할 Object 의 총갯수는 PPERF_DATA_BLOCK 의 NumObjectTypes 라는 필드에 저장되어 있다.

이제 우리는 버퍼의 전체 구조에서 Object 단위로 이동할 수 있다. 이제는 각 Object 를 만날때 그 내부의 Instance(Multiple-Instance 를 지원할 경우)와, Counter 들간을 이동해보자.
위에서 언급한대로 Multiple-Instance 를 지원하는 Object 만 Instance 가 존재하며 Single-Instance 인 경우에는 아예 Instance 라는 구조 없이 바로 Counter 들을 만나게 된다. 그럼 먼저 Multiple-Instance 인 경우를 살펴보자.

Object 의 구조를 보면, 헤더 부분에 PERF_OBJECT_TYPE 라는 Object 에 대한 정보를 가지는 구조체와, 그 밑에 PERF_COUNTER_DEFINITION 라는 Counter 들의 정보를 가지는 구조체들이 있다. 첫번째 Instance 를 만나려면 어떻게 해야할까? 위에서 처럼 PERF_OBJECT_TYPE 과 PERF_COUNTER_DEFINITION 구조체들의 크기를 알고 그만큼 뛰어 넘으면 된다. 다행히 PERF_OBJECT_TYPE 이라는 구조체의 DefinitionLength 필드에 위의 구조체들의 크기를 모두 더한 값이 저장되어 있다. 그러므로 우리가 찾은 Object 의 시작 부분에서 DefinitionLength 값을 더하면 그 Object 의 첫번째 Instance 를 접하게 된다.

```
PPERF_INSTANCE_DEFINITION FirstInstance( PPERF_OBJECT_TYPE PerfObj )
{
    return( (PPERF_INSTANCE_DEFINITION)((PBYTE)PerfObj +
        PerfObj->DefinitionLength) );
}
```

위의 Instance 의 구조를 보면 시작부분에 Instance 에 대한 정보를 담고 있는 PERF_INSTANCE_DEFINITION 과 그 Instance 의 이름을 담고 있는 블럭, 그리고 그 Instance 에 존재하는 Counter 들에 대한 정보를 담고 있는 PERF_COUNTER_BLOCK 이 있

음을 알 수 있다. PERF_INSTANCE_DEFINITION 의 ByteLength 필드에는 그 구조체와 뒤따르는 Instance Name 블럭의 크기를 합친 값이 저장되어 있으며, PERF_COUNTER_BLOCK 의 ByteLength 필드에는 그 구조체의 크기와 뒤따르는 Counter Data 들의 총 크기를 합친 값이 저장되어 있으므로 그 두 값을 더하면 다음 Instance 의 위치를 알 수 있다.

```
PPERF_INSTANCE_DEFINITION NextInstance(
    PPERF_INSTANCE_DEFINITION PerfInst )
{
    PPERF_COUNTER_BLOCK PerfCntrBlk;
    /* PERF_COUNTER_BLOCK 위치 계산 (Counter 들의 총 길이를 알기위해) */
    PerfCntrBlk = (PPERF_COUNTER_BLOCK)((PBYTE)PerfInst +
        PerfInst->ByteLength);

    /* PERF_COUNTER_BLOCK 의 위치에서 뒤따르는 Counter 들의 총 길이를 더함 */
    return( (PPERF_INSTANCE_DEFINITION)((PBYTE)PerfCntrBlk +
        PerfCntrBlk->ByteLength) );
}
```

이제 우리는 Instance 사이를 넘나들 수 있다. Performance Data 를 담고 있는 Counter Data 에 거의 접근했다. 꽤 복잡하다. 그렇지 않은가? NT 가 워낙 복잡한 운영체제다 보니까 자신에 대한 정보를 알려주는것도 복잡한것 같다. 그래도 필자는 정보가 많은것이 부족한 것보다 덜 답답하다(MSDN 이 참으로 방대하지만 매우 유용하듯이). MS 에서는 이런점 때문에 Performance Counter Data 에 대한 정보에 쉽게 접근할 수 있도록 PDH 를 제공하고 있다. 이것은 다음호에서 자세히 설명하겠다.

Counter Data 의 길이는 DWORD(32bit), LARGE(64bit), VARIABLE_LEN 가 될 수 있고, 그 Data 또한 계산과정없이 표시해야할 'NUMBER 형식', 특별한 계산이 요구되는 'COUNTER 형식', 혹은 '문자열 형식' 일 수 도 있다. 이런 정보들은 PERF_COUNTER_DEFINITION 구조체에 있으며 이곳에는 Data 의 수준을 나타내는 정보와, 계산방법, 화면 표시방법등의 세부 정보가 들어 있다. 다시, Counter Data 의 길이는 가변적이다. 그래서 특정한 Counter Data 에 접근하려면 그에 해당하는 PERF_COUNTER_DEFINITON 정보를 읽어야 한다. (Counter 2 Data 를 읽으려면 PERF_COUNTER_DEFINITION 2 를 읽어야 함) 그러므로 우리는 Counter Data 사이를 움직이는 것이 아니라, 그것에 대한 정보를 담고 있는 PERF_COUNTER_DEFINITION 구조체들 사이를 돌아다녀야 한다.

Multiple-Instance 나 Single-Instance 둘 모두 PERF_COUNTER_DEFINITON 의 위치는 같으므로 같은 함수가 적용된다. 첫번째 PERF_COUNTER_DEFINITION 을 얻으려면 Object 의 구조에서 PERF_OBJECT_TYPE 의 크기 만큼만 이동하면 된다.

```
PPERF_COUNTER_DEFINITION FirstCounter( PPERF_OBJECT_TYPE PerfObj )
{
    return( (PPERF_COUNTER_DEFINITION) ((PBYTE)PerfObj + PerfObj->HeaderLength) );
}
```

다음 PERF_COUNTER_DEFINITION 은 그 위치에서 PERF_COUNTER_DEFINITION 의 크기 만큼 이동한다.

```
PPERF_COUNTER_DEFINITION NextCounter( PPERF_COUNTER_DEFINITION PerfCntr )
{
    return( (PPERF_COUNTER_DEFINITION)((PBYTE)PerfCntr + PerfCntr->ByteLength) );
}
```

이제 우리는 모든 이동 수단을 가졌고, PERF_COUNTER_DEFINITION 구조체들 사이를 돌아다니며 원하는 Counter Data 를 수집하면 된다.

<Counter 계산>
모든 Object 들의 Counter 를 완전히 계산하려면 처리해야 할것이 한 두개가 아니다. 여기서는 Performance Counter 에 대한 기본지식을 배우고 간단한 예를 통해 실습하는 정도로 마치겠다. 실제 상황에서 Performance Counter 를 모두 사용하는 일은 매우 드물고 필요한 정보만 몇개 사용하는것이 보통이므로 전체 과정이 일반적일 필요는 없다. 그래도 모든 Data 가 필요한 사람은 여기서 배운 지식으로 나머지를 완성하거나 아니면 다음호에 소개될 PDH 를 이용하는것이 나을 것이다.

Counter Data 의 길이는 가변적이지만 계산을 편하게 하기 위해 4 혹은 8 바이트로 가정하자(대부분이 4 혹은 8 바이트임). 그리고 Data 는 'COUNTER 형식'으로 특별한 계산이 필요한 것으로 가정하자.

<Counter Data 의 위치>
우선 Counter Data 의 위치를 찾아야 하는데 그 위치는 PERF_COUNTER_DEFINITION 의 CounterOffset 필드에 있다. 그리고 PERF_COUNTER_BLOCK 의 위치에 그 CounterOffset 을 더하면 해당하는 Counter Data 의 위치가 나온다. 주의할것은 PERF_COUNTER_DEFINITION 의 CounterType 필드에 PERF_DISPLAY_NOSHOW 비트가 셋 되어 있으면 계산하지 말고 다음 블럭으로 넘어가야 한다는 것이다.

```
PerfObj = <원하는 Object 의 PERF_OBJECT_TYPE 의 위치>;
PerfCntr = <원하는 PERF_COUNTER_DEFINITION 의 위치>;
/* PERF_COUNTER_BLOCK 의 위치 */
PtrToCntr = (PPERF_COUNTER_BLOCK) ((PBYTE)PerfObj + PerfObj->DefinitionLength );
```

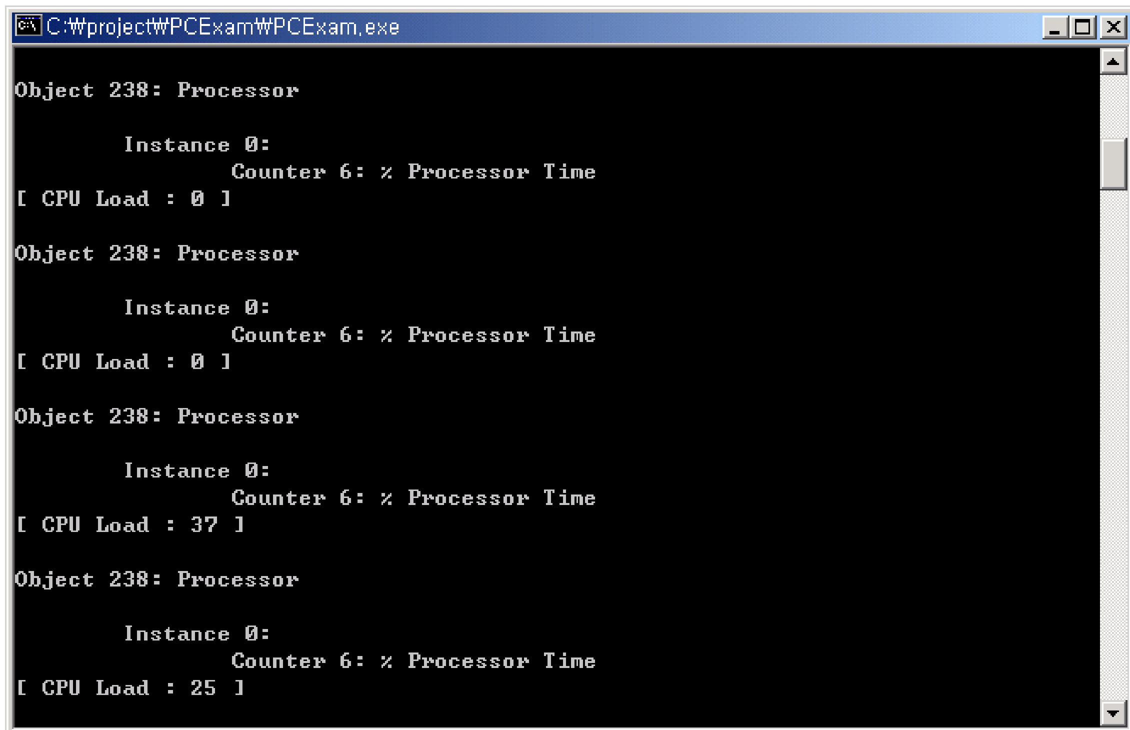
이제 위치를 알았으니 Data 를 읽자. Data 의 크기가 4 바이트면,

res = *(int*)((char*)PtrToCntr + PerfCntr->CounterOffset); 로 읽고, 8 바이트면,
res = *(__int64*)((char*)PtrToCntr + PerfCntr->CounterOffset); 로 읽는다.

<프로세서 부하율 계산>
자. 원하는 것을 얻었다. 꽤 복잡한 편인데 수고들 하셨다. 위의 방식으로 자신의 시스템에 있는 프로세서의 부하율을 알아보자. 프로세서의 부하율을 계산하는 법은 처음부분에 소개했다. MSDN 의 Performance Data Helper 의 서브-카테고리중에 'The Registry Interface' 에 있는 'Performance Objects and Counters' 를 보면 모든 Counter 들의 용도와 계산법(CounterType)이 나와 있다. 우리가 원하는 '부하율' 에 대한 정보는 '% Processor Time' 이라는 Counter 에 있다. 그리고 Performance Counter 들의 ID 와 이름을 살펴보면, '% Processor Time' Counter 를 담고 있는 Object 의 이름은 'Processor' 라는것을 알 수 있다. 프로세서가 하나라고 가정하고 '0' Instance 를 택하겠다. 계산법은 PERF_COUNTER_DEFINITION 의 CounterType 필드의 조합된 정보를 따라야 하는데 매우 복잡하다. 다행이 winperf.h 에 각종 Flag 들을 조합하여 추린 몇가지의 매크로가 존재한다. 우리는 그 매크로에 맞추어 처리하면 된다. '% Processor Time' 의 CounterType 은 매크로로 PERF_100NSEC_TIMER_INV 라고 정의되어 있다. 이것은 7 개의 다른 매크로가 조합된 것이다. 대략적인 의미는 이렇다. 이 Counter Data 는 64 비트 정수이며, 100ns 마다 1 씩 증가한다. 주어진 시간간격의 시작과 끝에서 Data 를 읽고 두차를 계산해 10,000,000 으로 나눈다. 그것에 100 을 곱하면 Idle 쓰레드를 실행하는데 들어간 시간 비율을 알 수 있다. 매크로의 마지막에 INV 라고 쓰여있는것은 프로세서의 부하율은 Idle 쓰레드의 실행율의 Inverse 라는것을 의미한다. 즉, 계산된 Idle 쓰레드의 실행율을 100 에서 빼면 프로세서의 부하율이 나온다. 두번 Data 를 읽어야 하므로 RegQueryValueEx() 를 통해 두번 요구해야 한다. (물론 Processor Object 의 ID 를 두번 알아낼 필요는 없다.)

아래에 프로세서의 부하율을 1 초마다 한번씩 알려주는 완성된 코드가 있다.

지면관계상 위에서 언급한 FirstObject, NextObject, FirstInstance, NextInstance, FirstCounter, NextCounter, GetNumOfCounters 함수는 다시 적지 않았다. 아래 프로그램의 결과는 <화면 4> 와 같다.



<화면 4> 프로그램 결과 화면

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include <tchar.h>


TCHAR *pStrbuf;
TCHAR **pCounterNames;
TCHAR **pCounterIDs;
DWORD dwTracks[5000];
DWORD dwTotalTrack;
__int64 freq;

void Initialize()
{
    TCHAR* pCurrentString; // pointer for enumerating data strings
    DWORD dwCounter;          // current counter index
    DWORD dwBufferSize;

    dwCounter = GetNumOfCounters();
    pCounterNames = (TCHAR**)malloc(sizeof(TCHAR*)*dwCounter);
    pCounterIDs = (TCHAR**)malloc(sizeof(TCHAR*)*dwCounter);
```

```
    // load compounded counter string.
    dwBufferSize = 8192;
        pStrbuf = (TCHAR*)malloc(dwBufferSize);

        while ( RegQueryValueEx(HKEY_PERFORMANCE_DATA,
                TEXT("Counter"),
                NULL,
                NULL,
                (LPBYTE)pStrbuf,
                &dwBufferSize) == ERROR_MORE_DATA)
        {
        dwBufferSize += 1024;
        pStrbuf = (TCHAR*)realloc(pStrbuf, dwBufferSize);
        }

        dwTotalTrack = 0;
    for( pCurrentString = pStrbuf; *pCurrentString;
        pCurrentString += (lstrlen((LPCTSTR)pCurrentString)+1) )
    {
        dwCounter = _ttoi(pCurrentString);
                dwTracks[dwTotalTrack++] = dwCounter;
                pCounterIDs[dwCounter] = pCurrentString;
        pCurrentString += (lstrlen((LPCTSTR)pCurrentString)+1);
                pCounterNames[dwCounter] = pCurrentString;
    }
}

void MakeCounterID(TCHAR* pCounter, TCHAR* pID)
{
        int i;

        for (i=0; i<dwTotalTrack; i++)
                if (CompareString(LOCALE_SYSTEM_DEFAULT, 0,
                        pCounter, -1, pCounterNames[dwTracks[i]], -1) == CSTR_EQUAL)
                {
                        _tcscpy(pID, pCounterIDs[dwTracks[i]]);
                        break;
                }
}

void Uninitialize()
{
        free(pStrbuf);
}

DWORD QueryPerformanceData(TCHAR* pObject, TCHAR* pInst, TCHAR* pCounter, __int64* res)
{
    PPERF_DATA_BLOCK            PerfData = NULL;
    PPERF_OBJECT_TYPE           PerfObj;
    PPERF_INSTANCE_DEFINITION   PerfInst;
    PPERF_COUNTER_DEFINITION    PerfCntr, CurCntr;
    PPERF_COUNTER_BLOCK         PtrToCntr;
    DWORD                       i, j, k;
    DWORD                       dwBufferSize;
    TCHAR*                      pStr;

    // Allocate the buffer for the performance data.
    dwBufferSize = 8192;
    PerfData = (PPERF_DATA_BLOCK) malloc( dwBufferSize );

    while( RegQueryValueEx( HKEY_PERFORMANCE_DATA,
                            pObject, // TEXT("Global")
```

```
                              NULL,
                              NULL,
                              (LPBYTE)PerfData,
                              &dwBufferSize ) == ERROR_MORE_DATA )
{
    dwBufferSize += 1024;
    PerfData = (PPERF_DATA_BLOCK) realloc( PerfData, dwBufferSize );
}

// save the high-resolution frequency
freq = PerfData->PerfFreq.QuadPart;

// Get the first object type.
PerfObj = FirstObject( PerfData );

// Process all objects.
for( i=0; i < PerfData->NumObjectTypes; i++ )
{
    // Display the object by index and name.
    _tprintf( TEXT("₩nObject %ld: %s₩n"), PerfObj->ObjectNameTitleIndex,
                pCounterNames[PerfObj->ObjectNameTitleIndex] );

     // Get the first counter.
    PerfCntr = FirstCounter( PerfObj );

    if( PerfObj->NumInstances > 0 )
    {
        // Get the first instance.
        PerfInst = FirstInstance( PerfObj );

         // Retrieve all instances.
        for( k=0; k < PerfObj->NumInstances; k++ )
        {
              // indicate the instance name.
              pStr = (TCHAR*)((PBYTE)PerfInst + PerfInst->NameOffset);

              if (CompareString(LOCALE_SYSTEM_DEFAULT, 0,
                        pInst, -1, pStr, -1) != CSTR_EQUAL)
              {
                     PerfInst = NextInstance( PerfInst );
                        continue;
              }

              // Display the instance by name.
              _tprintf( TEXT("₩n₩tInstance %s: ₩n"), pStr);

             CurCntr = PerfCntr;
             PtrToCntr = (PPERF_COUNTER_BLOCK) ((PBYTE)PerfInst + PerfInst->ByteLength );

            // Retrieve all counters.
            for( j=0; j < PerfObj->NumCounters; j++ )
            {
              if ((CurCntr->CounterType & PERF_DISPLAY_NOSHOW) == PERF_DISPLAY_NOSHOW)
              {
                        CurCntr = NextCounter( CurCntr );
                        continue;
              }

              // indicate the counter name.
              pStr = pCounterNames[CurCntr->CounterNameTitleIndex];
              if (CompareString(LOCALE_SYSTEM_DEFAULT, 0, pCounter, -1, pStr, -1) != CSTR_EQUAL)
              {
                         CurCntr = NextCounter( CurCntr );
```

```
                continue;
            }

            // Display the counter by index and name.
            _tprintf(TEXT("₩t₩tCounter %ld: %s₩n"), CurCntr->CounterNameTitleIndex, pStr);

            if ((CurCntr->CounterType & PERF_100NSEC_TIMER_INV) == PERF_100NSEC_TIMER_INV)
            {
                *res = *(__int64*)((char*)PtrToCntr + CurCntr->CounterOffset);
                return PERF_100NSEC_TIMER_INV;
            }

            // Get the next counter.
            CurCntr = NextCounter( CurCntr );
           }

           // Get the next instance.
           PerfInst = NextInstance( PerfInst );
        }
    }
    else
    {
        // Get the counter block.
        PtrToCntr = (PPERF_COUNTER_BLOCK) ((PBYTE)PerfObj + PerfObj->DefinitionLength );

        // Retrieve all counters.
        for( j=0; j < PerfObj->NumCounters; j++ )
        {
            if ((PerfCntr->CounterType & PERF_DISPLAY_NOSHOW) == PERF_DISPLAY_NOSHOW)
            {
                PerfCntr = NextCounter( PerfCntr );
                continue;
            }
            // indicate the counter name.
            pStr = pCounterNames[PerfCntr->CounterNameTitleIndex];
            if (CompareString(LOCALE_SYSTEM_DEFAULT, 0, pCounter, -1, pStr, -1) != CSTR_EQUAL)
            {
                PerfCntr = NextCounter( PerfCntr );
                continue;
            }

            // Display the counter by index and name.
            _tprintf(TEXT("₩t₩tCounter %ld: %s₩n"),
            PerfCntr->CounterNameTitleIndex, pStr);

            if (PerfCntr->CounterType == PERF_100NSEC_TIMER_INV)
            {
                *res = *(__int64*)((char*)PtrToCntr + PerfCntr->CounterOffset);
                return PERF_100NSEC_TIMER_INV;
            }

            // Get the next counter.
            PerfCntr = NextCounter( PerfCntr );
        }
    }

     // Get the next object type.
    PerfObj = NextObject( PerfObj );
  }
}

void main()
{
```

```
	TCHAR	pIndexName[20];
	__int64 st, res;
	DWORD	type;

	Initialize();

	MakeCounterID(TEXT("Processor"), pIndexName);
	while (1)
	{
		Sleep(1000);
		type = QueryPerformanceData(pIndexName, TEXT("0"), TEXT("% Processor Time"), &st);
		if (type == PERF_100NSEC_TIMER_INV)
		{
			static __int64 last = st;
			res = 100 - (__int64)((float)(st - last) / 10000000.0f * 100.0f);
			last = st;
		}
		_tprintf(TEXT("[ CPU Load : %I64u ]₩n"), res);
	}
	Uninitialize();
}
```

위 프로그램을 돌리면 사용중인 시스템의 CPU 사용율이 나온다. 태스크 매니저의 결과와 비교하면 태스크 매니저의 결과가 조금 높게 나올 수 있는데 그것은 위 프로그램 자체가 차지하는 CPU 부하율이 태스크 매니저의 결과에 영향을 미치기 때문이다. 이 때에는 태스크 매니저의 업-데이트 속도를 빠르게 하면 정상적인 결과를 볼 수 있다.

지면 관계상 설명에서 제외한 것들이 많다. Performance Counter 를 이용하면 '작업 관리자' 에서 보아오던 다양한 정보들 이상의 것을 알 수 있다. 필자는 이것을 통해 네트웍 장치의 바쁜 모습을 지켜 보았다. 실제로 통신하고있는 응용 프로그램 없었음에도 불구하고, 로컬 네트웍에서 발생하는 많은 데이타들을 계속 채질해 걸러내고있는 안타까운(?) 장면이었다. 그것으로 로컬 네트웍 라인의 부하율도 알게 되었다.

다음호에서는 지금까지 배운것을 완전하게 통합해놓은 그리고 매우 간편하게 구성한 Performance Data Helper (PDH) 에 대해 배우도록 하겠다.