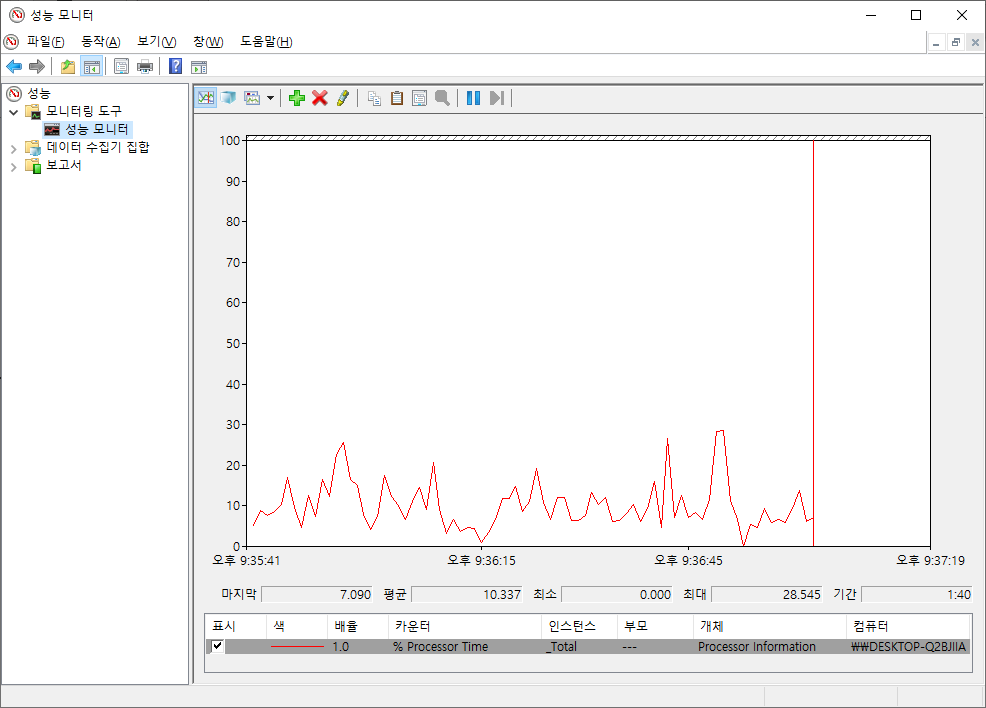
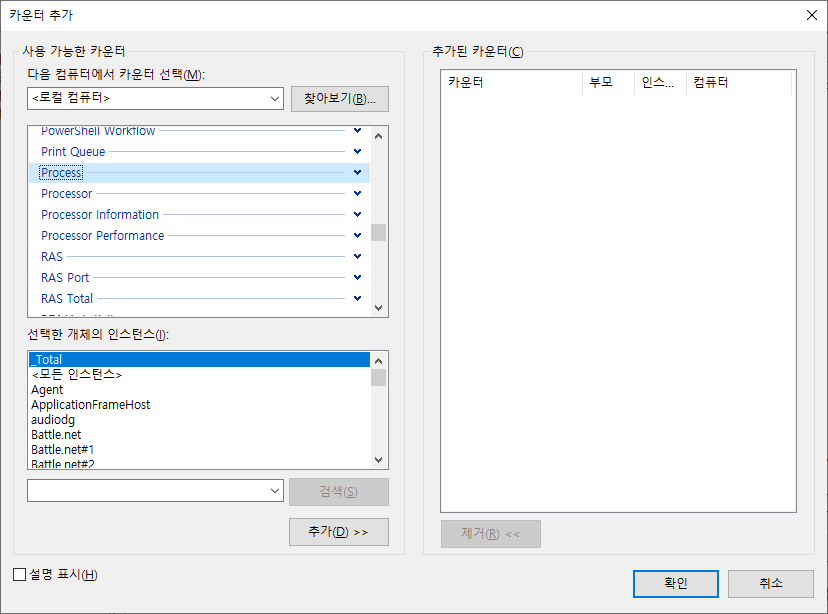
**CPU사용률을 체크하는 방법 1.**

**PDH의 사용**

**제어판 -> 관리도구 -> 성능 모니터**



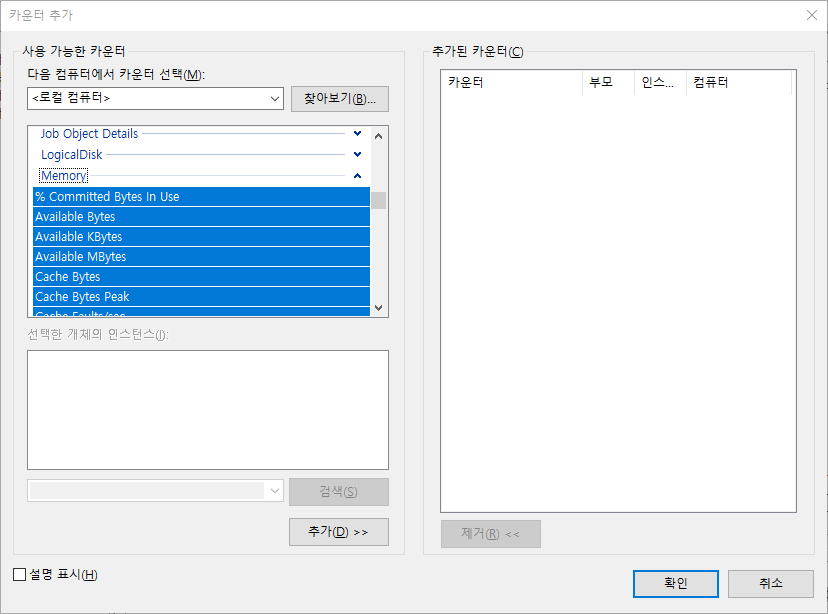
**오른쪽마우스 -> 카운터추가**



**Processor : CPU로, 프로세서에 대한 사용시간 같은것을 의미한다.**

**Process -> 프로세스. 프로세스에대한 CPU사용이나 핸들, IO입출력 pageFault등등.. 다나옵니다.**

그리고 메모리 파트 항목을 가면 메모리에 대한게 나온다.



(참고로 여기있는캐시는 우리가알고있는 캐시메모리가아나라 디스크에대한 캐시.)

이밖에도 TCP항목에는 재전송회수, 세그먼트(L4) 전송회수, 스레드항목에는 컨텍스트 스위칭 카운팅 등 우리가 상상하는 그 이상으로 모니터링이 가능하다.

따라서 대부분 서버 모니터링을 본격적으로 한다면 여기서 필요한 항목을 추가하여 그래프로 보는 것이다.

**이 모든 항목들은 윈도우 OS안에서 수집되게끔 작동되고있고, 제공되는 카운터를 WindowAPI를 통해 우리는 얻어낼 수 있다.**

**문제는 얻어내는 것이 windows7까지는 이상이 없었으나 8, 10에서부터 잘 되지않는 현상이 발견되었다.**

CPU를 0에가깝게 사용하는데 3,4,5의 수치가 나온다거나,

**CPU사용률은 높아지는데 비정상적으로 낮게 나온다는 등의 현상이 이것이다.**

**따라서 CPU사용률은 버리고, 윈도우8 이후부터는 CPU사용률쪽은 직접 만들어서 쓰는걸로 간다.**

만약 PDH로 시스템모니터링 데이터를 뽑았는데 CPU사용률이 정확하게 나왔다면 사용해도된다.

아직까지 어떤것이 문제인지 파악되지않음.

해당 수치자체가 문제가있는 것이 아니라, 내 코드상으로 얻어왔을 때 문제가됨.

**CPU사용률에 대해**

**CPU사용률은 특정시간동안 할당받은 스레드(또는 다른곳)에서 돌아가는 시간과 쉬는시간을 비율로 따져놓은 것이다.**

CPU는 사용된다 / 사용되지않는다 밖에없다. (0,100)

CPU사용률이 50% 라는것은 특정시간동안 50%의 시간동안 사용되었다는 소리다.

(결과적으로는 0.5초를 돌고 0.5초를 쉰것이지만 0.1초씩 돌고 쉬었는지 쉬고있다가 돌았는지에 대한것은 모름)

따라서 CPU사용률이라는 개념이 살짝 애매한 부분이 있다.

측정을 시작한 시점마다 다르므로

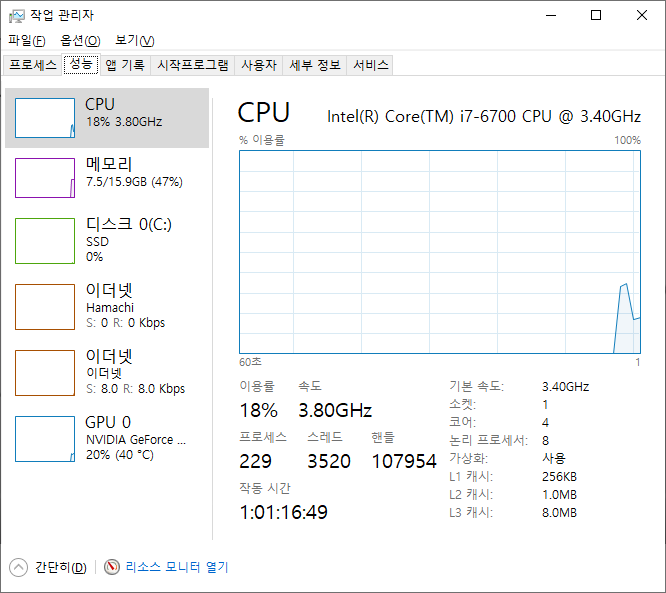
한쪽에서는 50 / 0 / 100 이 나왔는데,

다른쪽에서는 100 / 0 / 50 / 50 이 나올수도있다.



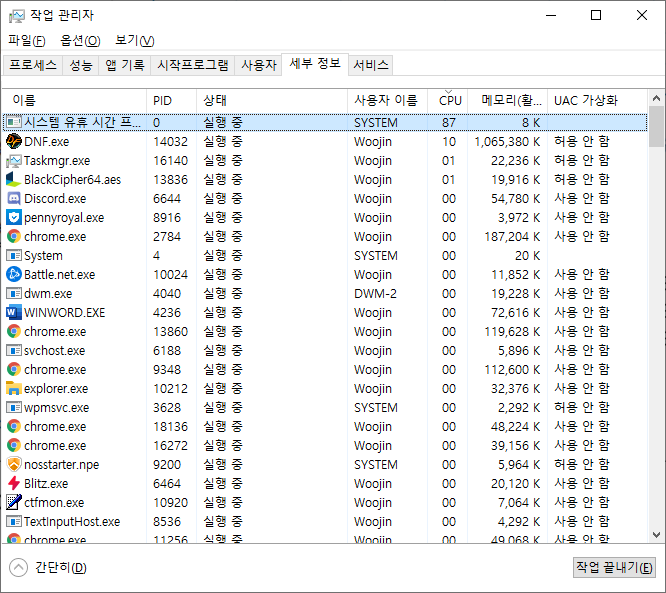
**CPU의 작업관리자 탭**

일단 우리는 작업관리자의 성능탭에서는 보지않는다.



**(정확하지 않음)**

우리는 세부정보와, 위에 언급한 성능모니터로 본다.



**(성능모니터 수치에는 문제가 없으나 내 코드상에 얻어왔을때 문제가 됨)**

**PDH수치자체는 정확하다.**

**PDH수치중 CPU사용률을 API로 얻어오는것은 문제가 있으나,**

**작업관리자의 세부정보로 보면 거의 일치한다.**

(샘플링 되는 시간때문에 약간의 오차는 존재하므로, 전반적인 추이를 기준으로 볼 것.)

다행스러운건 모든 OS의 기본 API에는 프로세스의 시간을 구하는 API가 존재한다.

(이런거조차 지원되지않는다면 아예 불가능한 부분)

**CPU사용률을 체크하는 클래스**

그대로 사용하지말고, 개량할 것.

선언할때 CCpuUsage CPUTime(); // CPUTime(hProcess)

이렇게 선언하거나, 특정 프로세스 핸들을 지정하거나이다.

빈칸으로 넣으면 자기자신의 프로세스. GetCurrnetProcess핸들을 통해서 구함.

프로세스 핸들을 집어넣는다면 해당 프로세스에 대해 시간을 구한다.

**CPU\_USAGE 헤더**

#include <windows.h>

#ifndef \_\_CPU\_USAGE\_H\_\_

#define \_\_CPU\_USAGE\_H\_\_

/////////////////////////////////////////////////////////////////////////////

// CCpuUsage CPUTime(); // CPUTime(hProcess) 둘중 하나로 선언한다.

// 일반적인 클래스 선언 // 프로세스 핸들지정

//

// while ( 1 )

// {

// CPUTIme.UpdateCpuTime();

// wprintf(L"Processor:%f / Process:%f \n"

, CPUTime.ProcessorTotal(), CPUTime.ProcessTotal());

//

// wprintf(L"ProcessorKernel:%f / ProcessKernel:%f \n",

CPUTime.ProcessorKernel(), CPUTime.ProcessKernel());

//

// wprintf(L"ProcessorUser:%f / ProcessUser:%f \n",

CPUTime.ProcessorUser(), CPUTime.ProcessUser());

// Sleep(1000);

// }

/////////////////////////////////////////////////////////////////////////////

**ProcessorTotal : 하드웨어적인 차원의 CPU전체 사용율**

**ProcessTotal : 해당 프로세스의 CPU사용률**

이때 문제는, 이 클래스하나가 프로세스 하나와 하드웨어 CPU하나를 모니터링 한다는 것이다.

만약 서버컴퓨터(물리적인 하드웨어) 안에 두개의 서버를 넣었다고 한다면, 양쪽에서 할 필요는 없음.

**따라서 하드웨어 CPU사용률과 프로세스CPU(프로세스별) 사용률을 분리할것.**

**커널모드 사용률 / 유저모드 사용률**

작업관리자에서는 커널모드/유저모드로 구분하여 사용률을 보여주지않는다.

내 서버의 CPU사용률이 높다면 커널/유저 어느쪽이 높은지 살펴봐야한다.

유저쪽이 높다면 순전히 로직을 짠 내탓일 것이고, 커널쪽이 높다면 낮추기위한 우회적인 방안을 모색해봐야 할것. (커널함수의 콜 낮추기)

class CCpuUsage

{

public:

//----------------------------------------------------------------------

// 생성자, 확인대상 프로세스 핸들. 미입력시 자기 자신.

//----------------------------------------------------------------------

CCpuUsage(HANDLE hProcess = INVALID\_HANDLE\_VALUE);

void UpdateCpuTime(void);

// 샘플링하고자 하는 간격마다 얘를 호출된다. (대부분 초단위)

// 해당시점의 퍼센테이지를 구함.

float ProcessorTotal(void) { return \_fProcessorTotal; }

float ProcessorUser(void) { return \_fProcessorUser; }

float ProcessorKernel(void) { return \_fProcessorKernel; }

float ProcessTotal(void) { return \_fProcessTotal; }

float ProcessUser(void) { return \_fProcessUser; }

float ProcessKernel(void) { return \_fProcessKernel; }

private:

HANDLE \_hProcess;

int \_iNumberOfProcessors;

//현재나의 CPU사용개수. 하이퍼스레딩이면 논리스레드 포함.

float \_fProcessorTotal;

float \_fProcessorUser;

float \_fProcessorKernel;

float \_fProcessTotal;

float \_fProcessUser;

float \_fProcessKernel;

ULARGE\_INTEGER \_ftProcessor\_LastKernel;

ULARGE\_INTEGER \_ftProcessor\_LastUser;

ULARGE\_INTEGER \_ftProcessor\_LastIdle;

ULARGE\_INTEGER \_ftProcess\_LastKernel;

ULARGE\_INTEGER \_ftProcess\_LastUser;

ULARGE\_INTEGER \_ftProcess\_LastTime;

};

#endif

**멤버변수 iNumberOfProcessors**

어떤 프로그램 이 1초동안 샘플링했는데 1초를 썼다고하자.

“1초”라는 수치는 사용시간일 뿐, 사용률이 아니다.

완벽하게 100%가 되려면 CPU를 모두 써야한다.

그래서 Number of Prossors로, CPU개수를 구해서 퍼센테이지 구해서 나누기를 할것임.

**Last가 붙은 멤버변수**

사용시간을 측정해야 하기때문에 이전시간을 보관하는 용도

**CCpuUsage 생성자**

#include <windows.h>

#include "CpuUsage.h"

//----------------------------------------------------------------------

// 생성자, 확인대상 프로세스 핸들. 미입력시 자기 자신.

//----------------------------------------------------------------------

CCpuUsage::CCpuUsage(HANDLE hProcess)

{

//------------------------------------------------------------------

// 프로세스 핸들 입력이 없다면 자기 자신을 대상으로...

//------------------------------------------------------------------

if (hProcess == INVALID\_HANDLE\_VALUE)

{

\_hProcess = GetCurrentProcess();

//가상핸들 나오겠지만 잘작동이 되기때문에 똑같이가면된다.

}

//------------------------------------------------------------------

// 프로세서 개수를 확인한다.

//

// 프로세스 (exe) 실행률 계산시 cpu 개수로 나누기를 하여 실제 사용률을 구함.

//------------------------------------------------------------------

SYSTEM\_INFO SystemInfo;

GetSystemInfo(&SystemInfo);

\_iNumberOfProcessors = SystemInfo.dwNumberOfProcessors;

\_fProcessorTotal = 0;

\_fProcessorUser = 0;

\_fProcessorKernel = 0;

\_fProcessTotal = 0;

\_fProcessUser = 0;

\_fProcessKernel = 0;

\_ftProcessor\_LastKernel.QuadPart = 0;

\_ftProcessor\_LastUser.QuadPart = 0;

\_ftProcessor\_LastIdle.QuadPart = 0;

\_ftProcess\_LastUser.QuadPart = 0;

\_ftProcess\_LastKernel.QuadPart = 0;

\_ftProcess\_LastTime.QuadPart = 0;

UpdateCpuTime();

}

**UpdateCpuTime**

////////////////////////////////////////////////////////////////////////

// CPU 사용률을 갱신한다. 500ms ~ 1000ms 단위의 호출이 적절한듯.

//

//

////////////////////////////////////////////////////////////////////////

void CCpuUsage::UpdateCpuTime()

{

//---------------------------------------------------------

// 프로세서 사용률을 갱신한다.

//

// 본래의 사용 구조체는 FILETIME 이지만, ULARGE\_INTEGER 와 구조가 같으므로 이를 사용함.

// FILETIME 구조체는 100 나노세컨드 단위의 시간 단위를 표현하는 구조체임.

//---------------------------------------------------------

ULARGE\_INTEGER Idle;

ULARGE\_INTEGER Kernel;

ULARGE\_INTEGER User;

//---------------------------------------------------------

// 시스템 사용 시간을 구한다.

//

// 아이들 타임 / 커널 사용 타임 (아이들포함) / 유저 사용 타임

//---------------------------------------------------------

if (GetSystemTimes((PFILETIME)&Idle, (PFILETIME)&Kernel, (PFILETIME)&User) == false)

{

return;

}

**GetSystemTimes**

**실제 시간을 구해주는 API이다.** (인자로 FILETIME의 포인터형태를 받는다.)

**FileTime이란 100n/s로 단위를 표현하는 8byte 변수를 두개가진 구조체이다. (UTC시간기준)**

(윈도우환경에서 시간의 최소단위는 100n/s이다)

어떤 변수를 써도 상관없기때문에, QueryPerformanceCounter에서 사용되는 LARGE\_INTEGER를 사용해도 상관없다.

GetSystemTimes()을 호출하여 IDLE/커널/유저 타임을 구한다.FILETIME 구조체로 뽑아준다.

이때 문제는 커널타임에 IDLE이 포함되어 들어온다.

**이때, 커널타임에 IDLE이 포함되어 들어오기 때문에 주의해야한다.**

우리가 필요로하는것은 각각의 사용시간을 구해서 비율을 구하는것이 목적이다.

따라서 커널과 유저를 더한 총사용시간을 구해서 비율을 구한다.

ULONGLONG KernelDiff = Kernel.QuadPart - \_ftProcessor\_LastKernel.QuadPart;

ULONGLONG UserDiff = User.QuadPart - \_ftProcessor\_LastUser.QuadPart;

ULONGLONG IdleDiff = Idle.QuadPart - \_ftProcessor\_LastIdle.QuadPart;

ULONGLONG Total = KernelDiff + UserDiff; // 커널 타임에는 아이들 타임이 포함됨.

ULONGLONG TimeDiff;

if (Total == 0)

{

\_fProcessorUser = 0.0f;

\_fProcessorKernel = 0.0f;

\_fProcessorTotal = 0.0f;

}

else

{

// 커널 타임에 아이들 타임이 있으므로 빼서 계산.

\_fProcessorTotal = (float)((double)(Total - IdleDiff) / Total \* 100.0f);

\_fProcessorUser = (float)((double)UserDiff / Total \* 100.0f);

\_fProcessorKernel = (float)((double)(KernelDiff - IdleDiff) / Total \* 100.0f);

}

\_ftProcessor\_LastKernel = Kernel;

\_ftProcessor\_LastUser = User;

\_ftProcessor\_LastIdle = Idle;

**위는 프로세서(CPU),**

**이제나올 밑구문은 프로세스를 뜻한다.**

//---------------------------------------------------------

// 지정된 프로세스 사용률을 갱신한다.

//---------------------------------------------------------

ULARGE\_INTEGER None;

ULARGE\_INTEGER NowTime;

//---------------------------------------------------------

// 현재의 100 나노세컨드 단위 시간을 구한다. UTC 시간.

//

// 프로세스 사용률 판단의 공식

//

// a = 샘플간격의 시스템 시간을 구함. (그냥 실제로 지나간 시간)

// b = 프로세스의 CPU 사용 시간을 구함.

//

// a : 100 = b : 사용률 공식으로 사용률을 구함.

//---------------------------------------------------------

//---------------------------------------------------------

// 얼마의 시간이 지났는지 100 나노세컨드 시간을 구함,

//---------------------------------------------------------

GetSystemTimeAsFileTime((LPFILETIME)&NowTime);

//---------------------------------------------------------

// 해당 프로세스가 사용한 시간을 구함.

//

// 두번째, 세번째는 실행,종료 시간으로 미사용.

//---------------------------------------------------------

GetProcessTimes(\_hProcess, (LPFILETIME)&None, (LPFILETIME)&None, (LPFILETIME)&Kernel, (LPFILETIME)&User);

//---------------------------------------------------------

// 이전에 저장된 프로세스 시간과의 차를 구해서 실제로 얼마의 시간이 지났는지 확인.

//

// 그리고 실제 지나온 시간으로 나누면 사용률이 나옴.

//---------------------------------------------------------

TimeDiff = NowTime.QuadPart - \_ftProcess\_LastTime.QuadPart;

UserDiff = User.QuadPart - \_ftProcess\_LastUser.QuadPart;

KernelDiff = Kernel.QuadPart - \_ftProcess\_LastKernel.QuadPart;

Total = KernelDiff + UserDiff;

\_fProcessTotal = (float)(Total / (double)\_iNumberOfProcessors / (double)TimeDiff \* 100.0f);

\_fProcessKernel = (float)(KernelDiff / (double)\_iNumberOfProcessors / (double)TimeDiff \* 100.0f);

\_fProcessUser = (float)(UserDiff / (double)\_iNumberOfProcessors / (double)TimeDiff \* 100.0f);

\_ftProcess\_LastTime = NowTime;

\_ftProcess\_LastKernel = Kernel;

\_ftProcess\_LastUser = User;

}

**프로세스 대상으로 시간을 구하는 함수로는**

**GetProcessTime() ,**

**GetSystemTimeAsFileTime() 가 있다.**

**GetProcessTime**

GetProcessTime는 실제 사용된 프로세스 시작시간, 가동시간이 들어가지만 필요없으므로 None으로 잡아서 버린다.

애매한 부분은, CPU와는 똑같이 할수가 없다.

CPU에서는 커널과 유저밖에 들어오지않았다.

프로세스에서는 IDLE이라는 개념이 없고, 단순히 사용시간만이 존재한다.

**따라서 GetSystemTimeAsFileTime을 통해서, 파일타임 규격에맞춰 흐른시간을 구한다.**

(c++11에서 새로추가된 라이브러리인 chrono 역시 내부에서 GetSystem..()을 사용한다.)

**QueryPerformanceCounter vs GetSystemTimeAsFileTime**

이둘의 정확도는 완전히 같다.

**QPFC는 시스템 부팅시간부터의 카운팅이며,**

**GSTF는 날짜의 개념까지 들어간다. 시스템에 대한 시간과 날짜 기준으로 FILETIME을 구함.**

(위 약자는 임의로 쓴것이며 실제로는 이렇게 부르지않음)

성능테스트를 해보지않아서 뭐가 더 가벼운지에 대한 확신은 서지않음.

(날짜 시간을 구해야되는데 정교한 시간으로 구해야한다면 GSTF쓰면 됨)

**--**

Processor에 대한 실제 사용시간이었기 때문에, 프로세서기준으로 나누기를 해주지않음.

(CPU에 대한 CPU총 사용시간이 나왔기 때문에, 개수로 나누기를해서 실제 사용률을 구함)

Process의 경우 사용시간에서 나누기를 해줘야함.

**사용자 쪽**

이제 Update함수를 호출시키면서 사용률을 출력하거나 네트워크로 쏘는등의 행동을 할수 있을것.

CPU사용률 구하기 끝.

위 개념들은 어떤 OS든 다똑같다. (함수만 다름)

커널타임, IDLE타임 등등 구하는방식이 모두 같음.

**pdh (performance data helper)**

작업관리자의 성능모니터를 API를 통해 모두 얻어낼 것이다.

pdh라이브러리를 추가하고, 인클루드 한다.

https://docs.microsoft.com/en-us/windows/win32/perfctrs/browsing-performance-counters

위 링크게 있는걸 복붙한 샘플. (msdn)

#include <windows.h>

#include <stdio.h>

#include <conio.h>

**#include <pdh.h>**

#include <pdhmsg.h>

**#pragma comment(lib, "pdh.lib")**

CONST ULONG SAMPLE\_INTERVAL\_MS = 1000;

CONST PWSTR BROWSE\_DIALOG\_CAPTION = L"Select a counter to monitor.";

void wmain(void)

{

PDH\_STATUS Status;

HQUERY Query = NULL;

HCOUNTER Counter;

PDH\_FMT\_COUNTERVALUE DisplayValue;

DWORD CounterType;

SYSTEMTIME SampleTime;

PDH\_BROWSE\_DLG\_CONFIG BrowseDlgData;

WCHAR CounterPathBuffer[PDH\_MAX\_COUNTER\_PATH];

//

// Create a query.

//

Status = PdhOpenQuery(NULL, NULL, &Query);

**(먼저 여기 핸들을 받고)**

if (Status != ERROR\_SUCCESS)

{

wprintf(L"\nPdhOpenQuery failed with status 0x%x.", Status);

goto Cleanup;

}

//

// Initialize the browser dialog window settings.

//

ZeroMemory(&CounterPathBuffer, sizeof(CounterPathBuffer));

ZeroMemory(&BrowseDlgData, sizeof(PDH\_BROWSE\_DLG\_CONFIG));

BrowseDlgData.bIncludeInstanceIndex = FALSE;

BrowseDlgData.bSingleCounterPerAdd = TRUE;

BrowseDlgData.bSingleCounterPerDialog = TRUE;

BrowseDlgData.bLocalCountersOnly = FALSE;

BrowseDlgData.bWildCardInstances = TRUE;

BrowseDlgData.bHideDetailBox = TRUE;

BrowseDlgData.bInitializePath = FALSE;

BrowseDlgData.bDisableMachineSelection = FALSE;

BrowseDlgData.bIncludeCostlyObjects = FALSE;

BrowseDlgData.bShowObjectBrowser = FALSE;

BrowseDlgData.hWndOwner = NULL;

**BrowseDlgData.szReturnPathBuffer = CounterPathBuffer;**

BrowseDlgData.cchReturnPathLength = PDH\_MAX\_COUNTER\_PATH;

BrowseDlgData.pCallBack = NULL;

BrowseDlgData.dwCallBackArg = 0;

BrowseDlgData.CallBackStatus = ERROR\_SUCCESS;

BrowseDlgData.dwDefaultDetailLevel = PERF\_DETAIL\_WIZARD;

BrowseDlgData.szDialogBoxCaption = BROWSE\_DIALOG\_CAPTION;

//

// Display the counter browser window. The dialog is configured

// to return a single selection from the counter list.

//

Status = PdhBrowseCounters(&BrowseDlgData);

if (Status != ERROR\_SUCCESS)

{

if (Status == PDH\_DIALOG\_CANCELLED)

{

wprintf(L"\nDialog canceled by user.");

}

else

{

wprintf(L"\nPdhBrowseCounters failed with status 0x%x.", Status);

}

goto Cleanup;

}

else if (wcslen(CounterPathBuffer) == 0)

{

wprintf(L"\nUser did not select any counter.");

goto Cleanup;

}

else

{

wprintf(L"\nCounter selected: %s\n", CounterPathBuffer);

}

//wcscpy\_s(CounterPathBuffer, L"\\Process(GameServer#0)\\% Processor Time");

//wcscpy\_s(CounterPathBuffer, L"\\Process(\_Total)\\% Processor Time");

//

// Add the selected counter to the query.

//

Status = **PdhAddCounter**(Query, CounterPathBuffer, 0, &Counter);

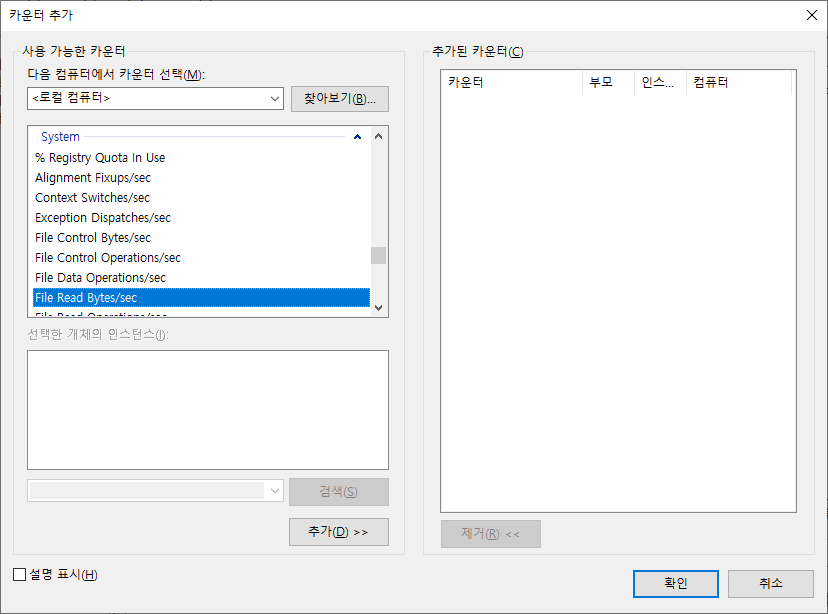
(카운터를 추가한다.

여기서 얘기하는 AddCounter는 주석처리된 wcscpy내부에 문자열을 말함.

**이것은 작업관리자에 존재하는 수많은 항목들을 스트링 쿼리라는 문장으로 요청하는 개념이다.**

항목마다 쿼리스트링이 정해져있고 그 쿼리스트링을 요청하면 등록이 되는 형태.)

예를들면, 성능모니터->카운터 추가에서

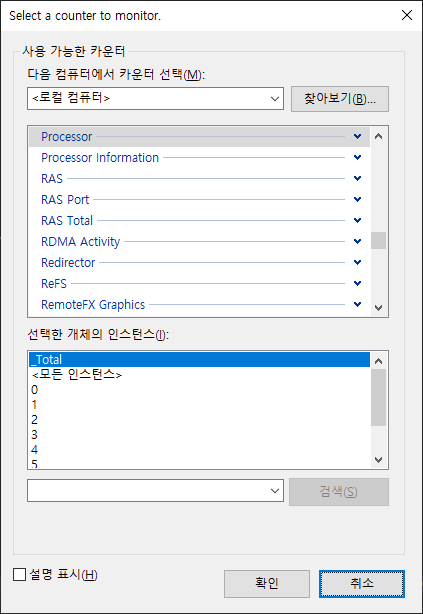


File Read Bytes/sec를 모니터링 하고싶다면, 해당 항목을 얻어내는 쿼리스트링을 알아야 한다.

이 쿼리스트링을 얻어내기위해, 아래 API함수를 사용한다.

**Status = PdhBrowseCounters(&BrowseDlgData);**

**이 API함수를 호출시키면 다이얼로그가 뜬다.**



다이얼로그에서 항목을 선택하고 확인을 누르면,

콘솔창에 스트링이 나오게 된다.

Counter selected: \Netlogon\Semaphore Acquires

위 구문에서 버퍼를 위와같이 지정했는데,

(BrowseDlgData.szReturnPathBuffer = CounterPathBuffer;)

이 버퍼를 통해 스트링이 나오게된다.

위와같이 나온 스트링을 AddCount에 등록시킨다.

Status = **PdhAddCounter**(Query, CounterPathBuffer, 0, &Counter);

그럼 이 하나의 핸들 쿼리에는 내가 수집하고자하는 쿼리가 쌓이게된다.

다이얼로그 시작할때마다 해야되는 것은 아니므로 적어놓고 기억함.

(우리코드에는 wscpy\_s로 박음)

여러개가 필요하다면 카운터 핸들만 바꿔서 여러개를 등록하면 된다.

if (Status != ERROR\_SUCCESS)

{

wprintf(L"\nPdhAddCounter failed with status 0x%x.", Status);

goto Cleanup;

}

//

// Most counters require two sample values to display a formatted value.

// PDH stores the current sample value and the previously collected

// sample value. This call retrieves the first value that will be used

// by PdhGetFormattedCounterValue in the first iteration of the loop

// Note that this value is lost if the counter does not require two

// values to compute a displayable value.

//

Status = PdhCollectQueryData(Query);

if (Status != ERROR\_SUCCESS)

{

wprintf(L"\nPdhCollectQueryData failed with 0x%x.\n", Status);

goto Cleanup;

}

//

// Print counter values until a key is pressed.

//

while (!\_kbhit())

{

Sleep(SAMPLE\_INTERVAL\_MS);

GetLocalTime(&SampleTime);

Status = PdhCollectQueryData(Query);

if (Status != ERROR\_SUCCESS)

{

wprintf(L"\nPdhCollectQueryData failed with status 0x%x.", Status);

}

wprintf(L"\n\"%2.2d/%2.2d/%4.4d %2.2d:%2.2d:%2.2d.%3.3d\"",

SampleTime.wMonth,

SampleTime.wDay,

SampleTime.wYear,

SampleTime.wHour,

SampleTime.wMinute,

SampleTime.wSecond,

SampleTime.wMilliseconds);

//

// Compute a displayable value for the counter.

//

Status = PdhGetFormattedCounterValue(Counter,

PDH\_FMT\_DOUBLE,

&CounterType,

&DisplayValue);

if (Status != ERROR\_SUCCESS)

{

wprintf(L"\nPdhGetFormattedCounterValue failed with status 0x%x.", Status);

goto Cleanup;

}

wprintf(L",\"%.20g\"", DisplayValue.doubleValue);

}

Cleanup:

//

// Close the query.

//

if (Query)

{

PdhCloseQuery(Query);

}

}

**PDH관련 메일**

# Performance Counters

윈도우는 다양한 하드웨어 리소스 (Performance Counters) 를 모니터링 하고 있으며 이 데이터를 사용자에게 제공한다.

**[ 제어판 > 관리도구 > 성능모니터 ]**

에 제공되는 모든 수치는 API를 통해 수집이 가능하다.

[성능모니터 > 카운터 추가]를 해보면 굉장히 다양한 정보들이 모니터링 가능함.

이는 윈도우에서 자체적으로 수집하는 항목이며 각각의 하드웨어 드라이버들은 Performance Counters 에 데이터를 제공하는 것을 원칙으로 함

# Performance Data Helper (PDH) interface,

우리는 Performance Counters 의 모든 데이터를 PDH API로 데이터수집이 가능하다.

**간단한 샘플 예제**

#include <stdio.h>

#include <Pdh.h>

#pragma comment(lib,"Pdh.lib")

int main(void)

{

// PDH 쿼리 핸들 생성

PDH\_HQUERY cpuQuery;

PdhOpenQuery(NULL, NULL, &cpuQuery);

// PDH 리소스 카운터 생성 (여러개 수집시 이를 여러개 생성)

PDH\_HCOUNTER cpuTotal;

PdhAddCounter(cpuQuery, L"\\Processor(\_Total)\\% Processor Time", NULL, &cpuTotal);

// 첫 갱신

PdhCollectQueryData(cpuQuery);

// 갱심하는 함수. 내가 등록한 쿼리에대한 데이터를 수집해온다.

// pdh얻고싶을때 주기적으로 호출한다.

while (true)

{

// 1초마다 갱신

Sleep(1000);

//쿼리 핸들한테 남김.

PdhCollectQueryData(cpuQuery);

// 갱신 데이터 준비

PDH\_FMT\_COUNTERVALUE counterVal;

//카운터 핸들에서 갱신데이터 얻음

PdhGetFormattedCounterValue(cpuTotal, PDH\_FMT\_DOUBLE, NULL, &counterVal);

wprintf(L"CPU Usage : %f%%\n", counterVal.doubleValue);

}

return 0;

}

1. 쿼리핸들 생성 (pdhOpenQuery)

2. PDH 리소스 카운터 생성 (pdhAddCounter)

3. 등록한 쿼리에 대해 데이터 수집 PdhCollectQueryData();

while(true)

{

4. 쿼리핸들에 남김 (PdhCollectQueryData)

5. 카운터 핸들에서 갱신데이터 얻음 PdhGetFormattedCounterValue

6. 출력

}

string을 쿼리핸들(cpuQuery)에 등록하고,CpuTotal대상으로 카운터 핸들(counterVal)을 구함

PdhCollectQueryData를 호출할때는 쿼리핸들을 넣어주고 (실제 메인이되는 쿼리핸들)

실제 데이터를 얻을때는 카운터핸들(CputTotal)에서 얻음

그래서 처음에 PdhCollectQueryData할때는 쿼리핸들로 한번씩 날리고, 세부데이터 얻을대는 카운터핸들(CpuTotal)을 기준으로 얻어내면 됩니다.

PDH\_FMT\_COUNTERVALUE 이 구조체는

typedef struct \_PDH\_FMT\_COUNTERVALUE {

DWORD CStatus;

union {

LONG longValue;

double doubleValue;

LONGLONG largeValue;

LPCSTR AnsiStringValue;

LPCWSTR WideStringValue;

};

} PDH\_FMT\_COUNTERVALUE, \* PPDH\_FMT\_COUNTERVALUE;

하나의 데이터를 어떤방식으로 지정할건지 정하는 방식이다.

그래서저는 더블을 쓰기위해서 여기다가 포맷 더블을 지정한것입니다.

**PdhGetFormattedCounterValue(cpuTotal, PDH\_FMT\_DOUBLE, NULL, &counterVal);**

그럼 여기 구조체에 더블 멤버에만 값이 들어감

더블만 필요하기떄문에 더블만 넣은것임. 더블이면 왠만한 데이터 다 표현할수있기때문에.

위 코드로 CPU 사용율을 얻어볼 수 있습니다.

**\* PdhOpenQuery**

pdh 는 검색 대상의 정보를 문자열로 등록하여 요청한다

그리고 이를 쿼리라고 표현. 먼저 pdh 쿼리 핸들을 만들어야 합니다.

PDH\_FUNCTION

PdhOpenQueryA(

\_In\_opt\_ LPCSTR szDataSource,

\_In\_ DWORD\_PTR dwUserData,

\_Out\_ PDH\_HQUERY \* phQuery

);

1 param : 실시간 데이터를 얻을 것이므로 null

2 param : 카운터 추가정보를 등록하는 인자, 너무 복잡하여 미사용

3 param : 쿼리 핸들 out 인자

**\* PdhAddCounter**

이제 수집하고자 하는 쿼리를 하나하나 PDH\_HQUERY 에 등록하면 됩니다.

수집 대상 당 카운터 핸들이 1개씩 나오며 몇개를 등록하든 상관 없습니다.

PDH\_HCOUNTER cpuTotal;

PdhAddCounter(cpuQuery, L"\\Processor(\_Total)\\% Processor Time", NULL, &cpuTotal);

cpu 사용율 1개의 카운터를 등록 했습니다.

하나의 PDH\_HQUERY 에 여러개의 카운터 쿼리를 등록하고자 한다면 여러번 반복 등록 하면 됩니다.

**\* PdhCollectQueryData(cpuQuery);**

위 함수를 사용해서 등록한 카운터의 정보를 갱신 합니다.

우리는 이 함수를 1초마다 주기적으로 호출하여 갱신 합니다.

**\* PdhGetFormattedCounterValue**

등록된 카운터의 값을 얻어냅니다.

PDH\_FMT\_COUNTERVALUE counterVal;

PdhGetFormattedCounterValue(cpuTotal, PDH\_FMT\_DOUBLE, NULL, &counterVal);

PDH\_FMT\_COUNTERVALUE 값을 통해서 받게 됩니다.

이 안에는 double, long, long long, string 등 다양한 타입의 변수가 존재하며

이중 원하는 타입으로 값을 받을 수 있습니다.

지금 예제는 DOUBLE 타입만 얻도록 지정 했습니다.

3번째 인자의 Type 은 얻고자 하는 데이터의 종류 입니다. delta, max, min, avr 등의 값이 들어갈 수 있으나

우리는 그냥 현재 값을 얻고자 하는 것이므로 넣지 않습니다.

위 코드대로 호출하고 나면 counterVal.doubleValue 안에 해당 수치가 들어있게 됩니다.

----------------------------------------------------------------

pdh 사용법 자체는 간단합니다. 문제는 쿼리 스트링이며 몇몇 스트링을 살펴보면

------------------------------------------------------------------

**CPU 전체 사용률 : "\\Processor(\_Total)\\% Processor Time"**

**CPU 코어 사용률 : "\\Processor(0)\\% Processor Time"**

**CPU 코어 사용률 : "\\Processor(1)\\% Processor Time"**

**CPU 코어 사용률 : "\\Processor(2)\\% Processor Time"**

**CPU 코어 사용률 : "\\Processor(3)\\% Processor Time"**

**프로세스 CPU 유저 사용률 : "\\Process(NAME)\\% User Time"**

**프로세스 CPU 전체 사용률 : "\\Process(NAME)\\% Processor Time"**

NAME은 실제 프로세스이름을 말함.

------------------------------------------------------------------

\*이부분이 정상적으로 나오지않아서 위와같이 따로 CPU사용률을 뽑는 클래스를 마련한 것.

정상적으로 나온다면 코어별로 사용량을 파악할 수 있으므로 좋을 것.

CPU 사용량은 OS 에 따라서 정상적으로 얻어지지 않는 경우가 발생함

직접 사용시간을 구해서 사용하도록 함

**프로세스 핸들 수 : "\\Process(NAME)\\Handle Count"**

**프로세스 스레드 수 : "\\Process(NAME)\\Thread Count"**

**프로세스 가상메모리 사용 : "\\Process(NAME)\\Virtual Bytes"**

이는 가상메모리 테이블의 주소 용량으로 실제 사용메모리(Commit) 과는 무관하다.

모니터링 대상 X

**프로세스 유저할당 메모리 : "\\Process(NAME)\\Private Bytes"**

실제로 우리가 사용한 유저메모리 사용량이다. (프로세스 공용메모리/커널메모리는 제외)

이 항목을 주력으로 모니터링 한다.

**프로세스 작업 메모리 : "\\Process(NAME)\\Working Set"**

현재 물리 메모리에 사용되는 크기일 뿐 할당 용량은 아닐 수 있음.

모니터링 대상 X

**프로세스 논페이지 메모리 : "\\Process(NAME)\\Pool Nonpaged Bytes"**

논페이지드 메모리로 문제가 터질때 이항목으로 잡히지않으므로 의미없는 항목이다.

**논페이지드 메모리 : L"\\Memory\\Pool Nonpaged Bytes"**

시스템전체에 대한 Nonpaged Pool메모리 용량이 나온다.

우리는 이 항목을 모니터링한다.

TCP Port, Overlapped I/O등을 사용하게되면서 수치는 계속 물결치게 될것인데,

만약 수치가 계속해서 상승곡선을 타게된다면 언제가는 서버가 터지게 된다.

NonPagedMemory는 우리가 제어할 수 없으며 이밖에 확인할 수 있는것이없음.

위 정보들을 좀더 체계적으로 보는 것이 poolmon.exe가 있다.

((

**Poolmon.exe**

풀몬은 커널관리 드라이버 코드기준으로 사용량을 볼 수 있다. (이 이상은 정보제공안됨)

그리고 논페이지드 메모리가 어디서 많이 쓰이는지 대략적인 힌트정도를 얻을 수 있음.

**Non-PagedPool Memory에 대해**

논페이지드 메모리가 증가된다고 하더라도 할 수 있는게 별로 없다.

사용자를 줄이거나 논페이지드 풀 메모리를 쓰는 행위를 빠르게 끝나도록 유도하는 수 밖에없음.

(Zero Byte Recv 등..)

논페이지드 풀 메모리는 사용이 끝나도 그 즉시 반환되지않고 약간의 텀을 두고 반환된다.

내부에서 버퍼링을 통해 시간을 잡아 해제하는 느낌이다.

))

**사용가능 메모리 : L"\\Memory\\Available MBytes"**

모니터링 대상에 들어가긴 하지만 큰 의미는 없음.

사용가능 메모리에 대한 어떤 시점만 가지고서는 정보를 파악하기 힘들다.

**우리가 최종적으로 모니터링 항목은 3~40개를 상회하므로, 중요도가 낮은 것은 제외하게 된다.**

**(사용자, 스레드에대한 FPS, MsgQ, JopQ.size(), DB size, DB초당 size, Accept수치 등..)**

따라서 중요도가 낮거나 크게 문제없는 서버들의 사용량은 보지 않을 것이다. 당연히 모든 서버의 프로세스 유저할당 메모리와 논페이지드 풀을 확인할 수있다면 완벽하게 모니터링 될 것.

**따라서 우리는 private bytes(프로세스 유저할당 메모리), Availale Mbytes(사용가능 메모리)로 모니터링한다.**

Private bytes는 늘지않는데 사용가능 메모리가 줄어든다면 어딘가에서 메모리가 누수되고있다는 것

이런식의 쿼리 스트링들이 존재 합니다.

모니터링이 필요한 쿼리 스트링을 미리 준비하여 하드코딩으로 구현 됩니다.

**\* 쿼리 스트링 알아내기**

쿼리 스트링은 msdn 등에서 제공되지 않으며 이 역시 api 함수를 통해 얻어낼 수 있습니다.

PdhBrowseCounters

위 함수가 그 기능을 해주는 함수이며, 위 함수를 실제로 호출 해보면 다이알로그가 뜹니다.

해당 다이알로그에서 원하는 모니터링 대상을 지정하면 쿼리를 얻어주는 방식 입니다.

해당 다이알로그는 성능 모니터링의 카운터 추가시 뜨는 다이알로그의 기능중 하나 입니다.

https://docs.microsoft.com/ko-kr/windows/win32/perfctrs/browsing-performance-counters

msdn 에서 Browsing Performance Counters 의 샘플 코드를 그대로 제공하고 있습니다.

더 다양한 쿼리 스트링을 알고자 한다면 위 url 의 샘플 코드를 그대로 복붙하여 실행시켜 보시기 바랍니다.

**# CPU 사용율**

이상하게도 윈도우 8 부터 pdh 를 통한 cpu 사용률 수집이 비정상적인 데이터가 자주 나타남.

원인은 파악되지 않지만 실제 cpu 사용율과는 조금 차이가 있는것으로 보임

그래서 cpu 사용율은 직접 구현하여 구하도록 할 것입니다.

관련 코드는 별도 pdf 로 드립니다.

cpu 사용률이란 ?

"4GHz cpu 에서 2GHz 만 사용했으니 50% 사용률이다" ( X )

클럭을 절반만 쓰는건 없으며, cpu 사용률이란 결국 '샘플링 시간동안 cpu사용시간이 얼마인가?'를 표현하는 수치이다.

따라서 사용시간을 구하는 것이 CPU사용률을 구하는 것이다.

CPU사용률은 OS가 제공해줘야만 측정이 가능한 부분이다. (모든 OS는 이런기능을 기본제공함.)

**GetSystemTimes(), GetProcessTimes()**

GetSystemTimes()는 CPU의 사용시간을 구해준다.

**프로세서 사용률과 프로세스 사용률은 다름**

**GetSystemTimes() - 프로세서 자체 사용시간**

**GetProcessTimes() - 프로세스의 cpu 사용시간을 얻기 위해선**

//---------------------------------------------------------

// 해당 프로세스가 사용한 시간을 구함.

//

// 두번째, 세번째는 실행,종료 시간으로 미사용.

//---------------------------------------------------------

GetProcessTimes(\_hProcess, (LPFILETIME)&None, (LPFILETIME)&None, (LPFILETIME)&Kernel, (LPFILETIME)&User);

GetProcessTimes()는 인자가 많은데 해당 프로세스의 시작시간, 전체 가동시간 인자가 있어 이런 부분은 사용하지 않음.

뒤 인자로 커널사용 시간, 유저사용 시간을 얻어볼 수 있습니다.

FILETIME 구조체는 64bit 형 시간 자료형으로 LARGE\_INTEGER 와 동일.

FILETIME이라고 표현하는 시간값은 100 nano Second를 기준으로하는 시간값 의미

**\* 특이점**

cpu 사용률을 구할 때 프로세서 개수를 얻어 나누기를 해야함.

\_ProcessTotal = (float)(Total / (double)\_NumberOfProcessors / (double)TimeDiff \* 100.0f);

\_ProcessKernel = (float)(KernelDiff / (double)\_NumberOfProcessors / (double)TimeDiff \* 100.0f);

\_ProcessUser = (float)(UserDiff / (double)\_NumberOfProcessors / (double)TimeDiff \* 100.0f);

듀얼코어에서 10초 중 10초를 내 프로세스가 모두 사용했다면,

사용률은 50%가되어야 하므로 /2를 하는 개념.

멀티스레드로 2개의 코어를 모두 사용했다면 1초동안 CPU사용 시간은 2초가 나오게된다.

그래서 항상 프로세서 개수 (논리 프로세서 개수, 하이퍼 스레딩 개수 포함) 로 나누기를 하여 사용률을 구하게 된다.

(위방식이 현재까지 가장 정확하게 확인된 CPU사용률 구하는 법)

////////////////////////////////////////////////////////////////////////////////////////////////////

제공해드린 자료들을 사용해서 모니터링 클래스를 만든다.

필수적인 모니터링 항목은

\* cpu total 사용률

\* cpu process 사용률

\* 프로세스 유저할당 메모리

\* 프로세스 논페이지 메모리

\* 사용가능 메모리

\* 논페이지드 메모리

+ 네트워크 사용량

네트워크 사용량은 한번 해보시면 아시겠지만 뭔가 좀 애매하다.

차후 별도언급.

**회선 계약**

서버를 만들고 회선계약을 얼마나 해야하는지 고민해봐야한다.

어떤 IDC나 어떤 클라우드에 가느냐에 따라 다르지만, 고전적인 IDC들은 회선으로 계약을 함.

Ex) 동접 n명을 목표로 하기 때문에 회선을 얼마로 하면 되겠다.

실제로 테스트환경을 구축해 트래픽이 어느정도 나오는지 알아야 한다.

**내 로직에서 트래픽 계산**

비트는 보통 Mb(메가비트 : 1초에 100만번의 비트체크)로 간다.

우리 로직에서는 WSASend 의 카운팅정도를 할 수 있겠다. (완료통지랑 1:1대응)

하지만 우리로직에서는 정확한 측정이 불가능하다.

**Nagle작동유무,상대방의 윈도우 사이즈가 가득찬 경우, 내 송신버퍼가 가득 찬 경우, MTU(1460)을 초과하는 데이터를 보낼경우 L4에서 조각내게 되는경우 등등의 이유때문.**

**[(WSASend CallCount + SendBytes) \* (IP헤더 + TCP헤더) ]**

**로 측정한다면, 조금 더 과하게 잡힐 것.**

(거의 ACK가 바로바로 오기 때문에 송신버퍼에 쌓일 염려가 있을까..)

(서버는 Nagle을 끄면 헤더부담이 너무 커지기 때문에 보통은 키고가는 것을 선택한다.)

**위 방식대로 측정한다면 트래픽의 최대치로 참고정도는 가능할 것.**

만약 이 수치를 초과하는 경우 더미테스트 자체가 잘 못되었거나, 유저들의 행동패턴 예측이 잘못된 경우라고 볼 수 있음.

**IP통신이나 TCP통신으로 트래픽계산**

우리서버 말고도 많은 송수신이 있고, 이것들도 모두 수집되기 때문에 애매함.

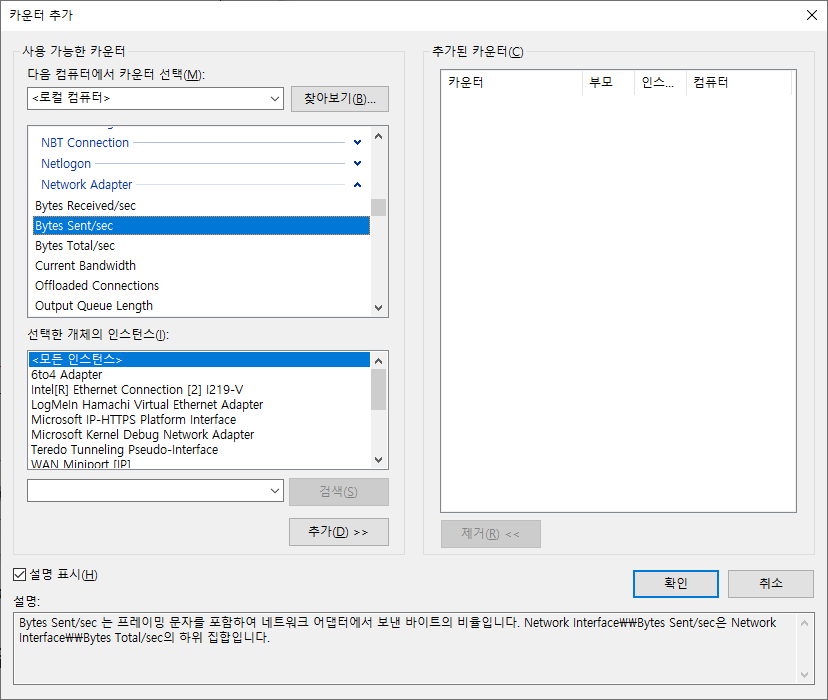
**이더넷에서 트래픽 계산**

하드웨어에 접근(pdh)하여 트래픽을 알아 낼 수도 있다.

Pdh의 문제는 이더넷에서 잡는다는 것에 있다.

**아래는 이더넷에 접근하여 초당 SendBytes / RecvBytes로 구하는것**

이더넷의 경우 어댑터 단위로 나오기 때문에, 사용하는 어댑터가 오타없이 정확하게 지정되어야한다.



프로세스의 경우 아래와같이 이름을 적었으면 됐었음

---------------------------------------------------------------------

프로세스 핸들 수 : "\\Process(**NAME**)\\Handle Count"

프로세스 스레드 수 : "\\Process(**NAME**)\\Thread Count"

프로세스 가상메모리 사용 : "\\Process(**NAME**)\\Virtual Bytes"

----------------------------------------------------------------------

그런데 이더넷의 경우는 우리가 이름을 알 수 없을뿐더러 아래와 같은 이유로 쓰기가 곤란하다.

**- 설치할때마다 이름이 달라짐 (=랜카드를 드라이버에 설치될 때 이름)**

**- 하나의 컴파일된 소스코드로 수십대 서버에 들어가는 경우 일일이 다 고칠 수 없음.**

**- 똑같은 사양으로 랜카드를 맞췄다고 하더라도 어떤게 먼저 넘버링 될지 알수없음.**

(만약에 위 문제들에도 불구하고 하드코딩할 자신이 있다면 해도 상관없다.)

**만약 이더넷하나를 특정지어 내 소스코드에 그대로 넣을 수 있다면 가장 정확할 것이나, 위와같은 이유로 어렵기 때문에 전체를 수집하는 방식을 택한다.**

(NAME)괄호에 \*을 넣으면 모든 인스턴스가 선택되지만 실제로는 첫번째꺼 하나만 잡히므로 이것은 활용하기 애매하다.

**전체 트래픽 수집**

내부라인은 애초에 수집대상이 아니다.

외부 공인 IP를 가지고 사용자들과 송신하는 송신량이 필요한 것.

**하지만 위 이유로 어려움이 많으므로 내부라인을 포함한 전체 인스턴스를 수집하여 합산한다.**

(이방식도 마음에 들지는 않기 때문에 실제로 사용할때는 개량하는 것을 권장한다)

**PDH 네트워크 이더넷 목록 얻어 사용하기**

#include <windows.h>

#include <tchar.h>

#include <pdh.h>

#define df\_PDH\_ETHERNET\_MAX 8

//--------------------------------------------------------------

// 이더넷 하나에 대한 Send,Recv PDH 쿼리 정보.

//--------------------------------------------------------------

struct st\_ETHERNET

{

bool \_bUse;

WCHAR \_szName[128];

PDH\_HCOUNTER \_pdh\_Counter\_Network\_RecvBytes;

PDH\_HCOUNTER \_pdh\_Counter\_Network\_SendBytes;

};

**이름을 찾아내 하나씩 등록시키고, 업데이트 갱신때마다 RecvBytes카운터, SendBytes카운터를 모두 수집을해서 다 합산하는 방식이다.**

**(모든 이더넷 트래픽 합산)**

st\_ETHERNET \_EthernetStruct[df\_PDH\_ETHERNET\_MAX]; // 랜카드 별 PDH 정보

double \_pdh\_value\_Network\_RecvBytes;

// 총 Recv Bytes 모든 이더넷의 Recv 수치 합산

double \_pdh\_value\_Network\_SendBytes;

// 총 Send Bytes 모든 이더넷의 Send 수치 합산

//////////////////////////////////////////////////////////////////////////////////////

int iCnt = 0;

bool bErr = false;

WCHAR\* szCur = NULL;

WCHAR\* szCounters = NULL;

WCHAR\* szInterfaces = NULL;

DWORD dwCounterSize = 0, dwInterfaceSize = 0;

WCHAR szQuery[1024] = { 0, };

// PDH enum Object 를 사용하는 방법.

// 모든 이더넷 이름이 나오지만 실제 사용중인 이더넷, 가상이더넷 등등을 확인불가 함.

//---------------------------------------------------------------------------------------

// PdhEnumObjectItems 을 통해서 "NetworkInterface" 항목에서 얻을 수 있는

// 특정항목(Counters) / 인터페이스 항목(Interfaces) 를 얻음. 그런데 그 개수나 길이를 모르기 때문에

// 먼저 버퍼의 길이를 알기 위해서 Out Buffer 인자들을 NULL 포인터로 넣어서 사이즈만 확인.

//---------------------------------------------------------------------------------------

PdhEnumObjectItems(NULL, NULL, L"Network Interface", szCounters, &dwCounterSize, szInterfaces, &dwInterfaceSize, PERF\_DETAIL\_WIZARD, 0);

**PdhEnumObjectItems**

먼저 성능모니터에서 본 항목을 알아야 해당 정보를 수집해올 것이다.

(CPU같은 경우 0,1,2,3,4 등으로 번호목록이 떴었음)

PdhEnumObjectItems(NULL, NULL, L"Network Interface", szCounters, &dwCounterSize, szInterfaces, &dwInterfaceSize, PERF\_DETAIL\_WIZARD, 0);

PdhEnumObjectItems()함수는 목록을 나열시켜, 내 네트워크 인터페이스에 있는 코드상에서 얻어낼 때 사용.

szCounters = new WCHAR[dwCounterSize];

szInterfaces = new WCHAR[dwInterfaceSize];

//---------------------------------------------------------------------------------------

// 버퍼의 동적할당 후 다시 호출!

//

// szCounters 와 szInterfaces 버퍼에는 여러개의 문자열이 쭉쭉쭉 들어온다. 2차원 배열도 아니고,

// 그냥 NULL 포인터로 끝나는 문자열들이 dwCounterSize, dwInterfaceSize 길이만큼 줄줄이 들어있음.

// 이를 문자열 단위로 끊어서 개수를 확인 해야 함. aaa\0bbb\0ccc\0ddd 이런 식

//---------------------------------------------------------------------------------------

if (PdhEnumObjectItems(NULL, NULL, L"Network Interface", szCounters, &dwCounterSize, szInterfaces, &dwInterfaceSize, PERF\_DETAIL\_WIZARD, 0) != ERROR\_SUCCESS)

{

delete[] szCounters;

delete[] szInterfaces;

return false;

}

iCnt = 0;

szCur = szInterfaces;

**szCounters - 하위에 수집할 수 있는 카운터에 대한 종류가 뜰 것**

**szInterfaces - 인터페이스 목록 나열**

**문제는 용량 예측이 전혀 되지않는것에 있다.**

따라서 WINAPI에서 많이 사용하는 방식을 사용한다.

(UTF-8인코딩할 때, 멀티바이트 -> 와이드캐릭터 또는 와이드캐릭터 -> 멀티바이트 할때도 함)

버퍼 포인터에 NULL을 넣어 인자로 전달한다면 szCounters 버퍼를 사용하지않고 뒤 인자에 크기만 넣어 준다.

이제 얻은 크기를 가지고 동적할당하여 이 크기를 가지고 다시금 호출한다.

문자열에는 aaa\0bbb\0ccc\0ddd 이와 같은 형태로 들어오기 때문에

포인터 단위로 NULL문자를 끊어가면서 문자열을 얻어내면 된다.

아래는 반복문을 돌면서 뽑아내는 로직.

//---------------------------------------------------------

// szInterfaces 에서 문자열 단위로 끊으면서 , 이름을 복사받는다.

//---------------------------------------------------------

for (; \*szCur != L'\0' && iCnt < df\_PDH\_ETHERNET\_MAX; szCur += wcslen(szCur) + 1, iCnt++)

{

\_EthernetStruct[iCnt].\_bUse = true;

\_EthernetStruct[iCnt].\_szName[0] = L'\0';

wcscpy\_s(\_EthernetStruct[iCnt].\_szName, szCur);

szQuery[0] = L'\0';

StringCbPrintf(szQuery, sizeof(WCHAR) \* 1024, L"\\Network Interface(%s)\\Bytes Received/sec", szCur);

PdhAddCounter(\_pdh\_Query, szQuery, NULL, &\_EthernetStruct[iCnt].\_pdh\_Counter\_Network\_RecvBytes);

스트링 잡아서, 바로 COunter로 등록.

szQuery[0] = L'\0';

StringCbPrintf(szQuery, sizeof(WCHAR) \* 1024, L"\\Network Interface(%s)\\Bytes Sent/sec", szCur);

PdhAddCounter(\_pdh\_Query, szQuery, NULL, &\_EthernetStruct[iCnt].\_pdh\_Counter\_Network\_SendBytes);

}

///////////////////////////////////////////////////////////////////////////

위에서 만들어진

\_EthernetStruct[iCnt].\_pdh\_Counter\_Network\_SendBytes

\_EthernetStruct[iCnt].\_pdh\_Counter\_Network\_RecvBytes

PDH 카운터를 다른 PDH 카운터와 같은 방법으로 사용 해주면 됨.

//-----------------------------------------------------------------------------------------------

// 이더넷 개수만큼 돌면서 총 합을 뽑음.

//-----------------------------------------------------------------------------------------------

for (int iCnt = 0; iCnt < df\_PDH\_ETHERNET\_MAX; iCnt++)

{

if (\_EthernetStruct[iCnt].\_bUse)

{

Status = PdhGetFormattedCounterValue(\_EthernetStruct[iCnt].\_pdh\_Counter\_Network\_RecvBytes,

PDH\_FMT\_DOUBLE, NULL, &CounterValue);

if (Status == 0) \_pdh\_value\_Network\_RecvBytes += CounterValue.doubleValue;

Status = PdhGetFormattedCounterValue(\_EthernetStruct[iCnt].\_pdh\_Counter\_Network\_SendBytes,

PDH\_FMT\_DOUBLE, NULL, &CounterValue);

if (Status == 0) \_pdh\_value\_Network\_SendBytes += CounterValue.doubleValue;

}

}

갱신하고 싶다면, 먼저 PDH컬렉터 데이터로 갱신한번

그리고 FOR문 돌면서 PdhGetFormattedCountervalue().

여기서는 리시브바이트 핸들을 대상으로 카운트밸류해서 밑줄로 넘기고,

샌드바이트에 대한 카운터핸들해서 겟카운트밸류해서 누적. 전체를 대상으로 유지.

그래서 초당 몇바이트를 보냈고 몇바이트를 받았는가 뽑아내고있음.

**주의해야 할 것**

실제 학원에서는 15000명을 테스트하는데, 실제로는 1U에 더미하나, 2U에 더미두개를 넣는다.

로컬통신은 이더넷(L2)이 아닌 라우팅테이블(L3)까지갔다가 다시 돌아온다.

**(따라서 제대로 테스트하려면 별도 컴퓨터를 활용해서 이더넷을 송신하는 형태로 해야 할 것.)**

**안타까운건 내외부가 모두 통합이 되기 때문에 완전히 정확하지않음.(단순히 많고 적음만 판단)**

**합산된 Recv/Send수치 활용**

라이브서비스 하는 과정에서 일상적으로 모니터링 하는 용도이다.

**일반적인 상태의 트래픽이 나와야하는데 이유모르게 늘었다면 이상한낌새를 눈치채는 용도이다.**

실제 프로젝트 할때는 DB,더미 등이 추가로 잡힌다. (DB송신이 제일 클것임)

통합수치이므로 이를 구분할 수는 없음.

그런데 똑 같은 코드에 똑 같은 더미동작인데 1.5~2배정도 차이가 나기도한다.

**트래픽이 적다는 것은 우리로직차원에서 얼마나 모아서 보내는지, 노딜레이 옵션유무, IO팬딩 유무 등이 있겠다.**

회사에서는 하나만 만들기떄문에 비교대상이 없음

**실제 회선계약을 위한 트래픽계산**

트래픽을 예상하고자한다면 이더넷을 특정하고 가야한다.

**그외에도 코드상에서도 구하고 [내 메시지의 크기 + (SendCount \* 헤더)],**

**PDH수치 등 활용할 수 있는 데이터는 모두 수집, 취합하여 결정권자 한테 줘야한다.**

물론 안전하게 간다면 높은걸 보는 것이 좋을 것.

**대역폭계약과 종량제방식**

**대부분 IDC센터들이 대역폭을 계약해서 가고, 일부는 종량제방식을 선택한다.**

대역폭 계약의 경우 회선자체에 제한이 걸리는 것이므로 그이상으로는 송신하지못한다.

종량제의 경우 10GB를 열고 월 600GB또는 900GB까지가 무료.

그이상은 MB단위로 비용을 측정하게된다. 이를 초과할 경우 요금폭탄이 떨어진다.

AWS역시 나가는 OutBand에 대해 요금이 나가게 된다.

(간단한 웹사이트를 만들었는데 요금폭탄이 떨어지는 피해사례가 많음)

따라서 어느정도 트래픽을 예상할 필요가 있다.

**테스트환경에서 잠깐 하는 경우**

우리는 PDH데이터를 코드에 박아넣고 주기적으로 수집하기위해 위와 같은 로직을 짰다.

만약 컴퓨터 테스트환경에서 잠깐 하고자한다면 단순히 성능모니터 그래프로 봐도 문제없음

**채팅서버에 적용**

채팅서버에 적용해서 Console에 출력하도록 한다.

이후 모니터링 서버에 연결한다면 이 데이터들을 모니터링 서버에 쏘는 것.

모니터링 서버는DB에도 저장하고, 모니터링 클라한테도 쏜다.

**메모리사용량 모니터링**

프로세스 메모리 워킹셋API, 메모리관련 Getter함수들이 존재하긴하지만, 어차피 PDH로 얻을수있으므로 굳이 선택할 필요성은 못느낌.

**<채팅서버 관련 채팅서버파일로 넘김>**

**1.22**

**<채팅서버 관련 채팅서버파일로 넘김>**

**대칭키 암호화**

**대칭키 - 인코딩을 하는 Key - 디코딩을 하는 Key가 같은 키를 가짐.**

**대칭키 암호화는 KEY를 주고받기 때문에 치명적인 단점이 존재한다.**

**키는 상수화 하여 사용할수 없으므로 패킷을 통해 주고받음.**

**따라서 누군가 패킷을 가로챈다면 무용지물이 된다.**

이것이 대칭키의 한계.

키를 나만이 소유하고 나만이 암호화/복호화한다면 아무런 상관이없다.

문제는 원격지에 있는 과정에서 암호화하는 쪽/복호화 하는쪽이 다르면 어쩔수 없이 문제가된다.

**비대칭키 암호화**

비대칭키는 암호화 키 / 복호화 키가 다르므로 대칭키와 같은 문제가 없다.

암호화 키가 노출된다고 하더라도 복호화가 될 수 없음.

비대칭키는 성능이 굉장히 떨어지기 때문에 모든 통신에 이 방법을 사용할 수는 없다.

**Diffie–Hellman key exchange**

**디피헬만 키교환 알고리즘(=키공유 알고리즘)은 키 공유를 하는 방법에 대해 말한다.**

**비대칭키와 디피헬만은 동의어가 아니다.**

**대부분이 디피헬만 키공유 방법을 이용해 비대칭키를 구현한다.**

----------------------------------------------------------------------------------

**P = 소수 G = p보다 작은 정수**

----------------------------------------------------------------------------------

P와 G를 준비한다.

----------------------------------------------------------------------------------

**엘리스 밥**

**A(값) B(값)**

**GA  % P GB % P**

**(결과값 밥에 전달) (결과값 엘리스에 전달)**

----------------------------------------------------------------------------------

**엘리스와 밥은 각자가 값(A,B)를 가지고, 계산한 결과값을 공유한다.**

----------------------------------------------------------------------------------

**엘리스 밥**

**A(값) B(값)**

받은값에 A승한다. 받은값에 B승한다

**GAB % PA GAB  % PB**

**(GB % P )A (GA  % P)B**

----------------------------------------------------------------------------------

**각자가 전달받은 결과값에 본인(A, B)값을 승수로 올린다.**

----------------------------------------------------------------------------------

**엘리스 밥**

**A(값) B(값)**

**(GB % P )A % p (GA  % P)B % p**

----------------------------------------------------------------------------------

**마지막으로 P를 나머지연산(mod)한다.**

**결과적으로 사전에 정의된 P와 G는 노출된다.**

**노출되지 않은 값은 A, B두개의 값뿐이다.**

**디피헬만을 색깔로 구현**

----------------------------------------------------------------------------------  
**P(노랑) G(파랑**) <- 노출되는값

----------------------------------------------------------------------------------

**노출되는 두개의 값을 정의함 (소수P와 P보다 작은 정수G)**

----------------------------------------------------------------------------------

**엘리스 밥**

A(빨강) B(초록)

**노파빨 노파초 <- 상대에게 공유할 값 연산**

----------------------------------------------------------------------------------

**A(빨강)는 P(노랑), G(파랑)를 연산해 노파빨을 만듬**

**B(초록)는 P(노랑), G(파랑)를 연산해 노파초를 만듬**

**각 결과값을 상대에게 전달할 예정**

----------------------------------------------------------------------------------

**엘리스 밥**

A(빨강) B(초록)

**PGA(노파빨) PGB(노파초)** **<- 교환하면서 노출된다**

**PGB(노파초) PGA(노파빨) <- 서로에게 받은 값**

----------------------------------------------------------------------------------

각자가 계산한 결과값을 상대에게 전달한다. (결과값들은 노출된다)

----------------------------------------------------------------------------------

**엘리스 밥**

A(빨강) B(초록)

**PGA(노파빨) PGB(노파초)** <- (교환하면서 노출)

**PGB + A(노파초빨) PGA + B(노파빨초)** <- (노출된적없음)

----------------------------------------------------------------------------------

**최종적으로 [노,파, 노파빨, 노파초]가 노출되었다.**

결국엔 빨,초를 알아낸다면 복호화가 가능하지만

이미 섞여낸 노파빨에서 빨을 추출하거나

노파초에서 초를 알아내는 것은 쉽지않다.

결국 브루트포스에 100% 안전한 것은 아니나, 그 확률을 불가능에 가깝게 만드는 것

(몇십년씩 걸리게)

**디피헬만 증명**

개념설명은 위 설명으로 끝이다. 이제 어떻게 이런 계산이 가능한지 살펴본다.

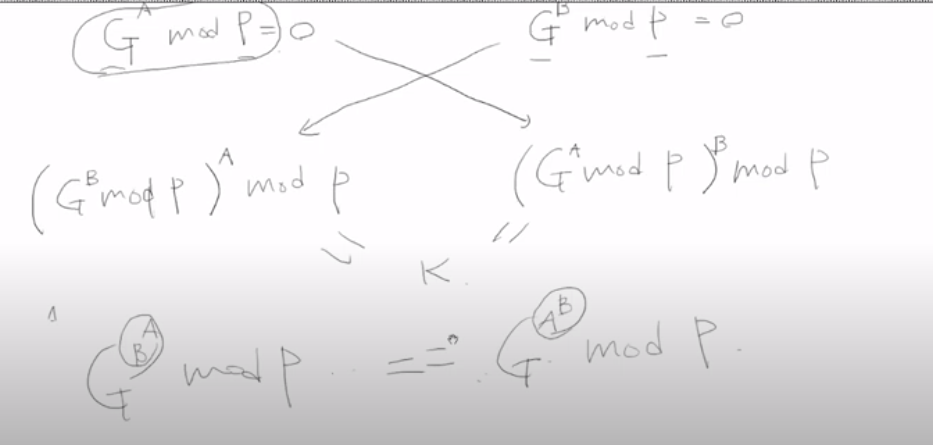
A의 2승의 3승은, A의 6승과 같다.

2\*3 == 6

나머지연산은 교환법칙과 분배법칙이 적용된다.

A \* 3 + B \* 3 = ( A + B ) \* 3;

아까 식으로 보자면,



(mod는 나머지 계산을 의미한다.)

현재 G는 P보다 작은값이다.

G % P 를 해서 나온값을, 다시 승수해서 나머지한다고해도 값은 달라지지않는다.

P는 소수임을 가정했는데,

P가 소수가 아닌경우 판단이 안되는경우가 나오기 때문이다.

소수가 아니라고해서 아예 안되는것은 아니지만 여기도 전제조건들이 붙는다.

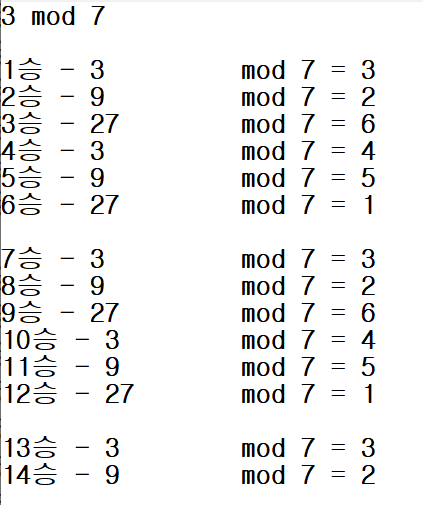
G < P인데,

G > P일 경우 결과가 달라진다.

그럼 직접 계산을 해보자.

**(GB % P )A % P == (GA  % P)B % P**

결국 해킹하는입장에서는 A,B만모름. 결국 문제는 G? mod P에 몇승을 하느냐에 대한 문제이다.



3n % 7을 하는경우 값을 나타냈다.

G = 3

P = 7

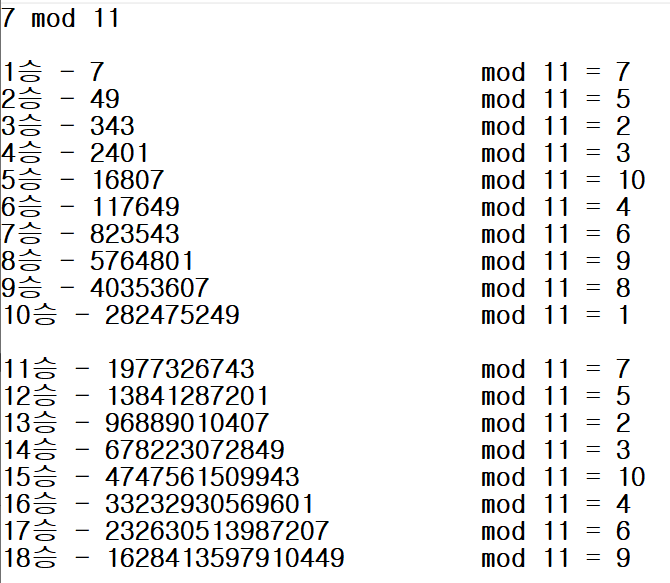
A = ?

B = ?

그렇다면 나올수있는 경우의 수는 3,2,6,4,5,1 중에 하나일 것이다.

왜냐면 P는 노출이되어있기 떄문이다.

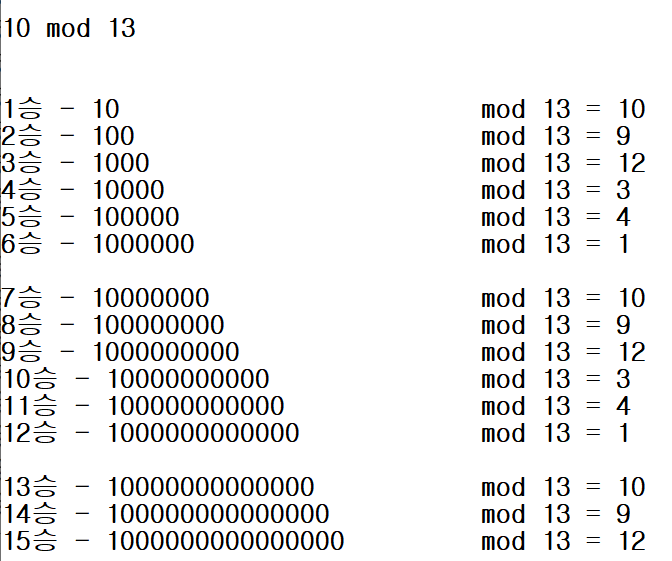
그러면 하나씩 때려맞추면 6분의 1확률로 나올것이다.



**우리는 G값과 P값을 잘구해야한다. 대부분 설명에는 [ G(정수) < P ]가 전제라고 함.**

**G가 정수일 때도 이 공식이 성립을 하긴한다.**

근데 만약에 13보다 작은 10을 잡는다면?



13이 모두 돌지않는다.

결국은 이 G(소수) < P 가 되면 가장좋다. 그래야 경우의 수를 최대한 넓힐수 있음.

(키를 13으로 한사람은 12개중 하나가 나오기를 기대하지만, 실제로는 6개의 패턴이 나온다.)

**브루트포스로 디피헬만 복호화할 확률**

결국 키라는 값을 40억으로 한다면 40억분의 1이 될 것.

일반적으로 암호학에서 사용하는 소수들은 256, 512, 1024비트.

64비트 수치만해도 0~ 1800경정도 되므로 거의 상상할 수 없는 수이며, 이는 현존하는 컴퓨터로브루트포스(무작위대입)하여 풀 수 있는 방법은 없다고봐도 무방.

**결국 RSA 비대칭키도 그렇고, 디피헬만도 그렇고 경우의 수를 불가능할 정도로 낮추는것이다.**

**네트워크 단에서 패킷을 분석해서 매크로를 만든다고 한다고 가정한다면,**

(클라조작이 아닌 프로토콜 송신)

**클라입장에서는 키를 알수가 있고 클라를 분석하면 키를 똑같이 만들 수있다.**

**지금얘기하는 암호화는 중간에 도청해가는 제 3자로부터 지키기 위함이지, 밥과 엘리스가 통신하는데 밥이 못열게하게끔 하는 것이 아니다.**

우리의 유저는 우리가 방어해야 할 대상이 아니다.

**RSA 암호화**

**디피헬만을 활용하여 이를 발전시킨 것이 RSA암호화이다.**

(디피헬만은 실제로도 여기저기서 활용한다)

**중요한건 노출되는 P, G값이며 A,B는 얼마를 넣던 상관없다.**

(A,B를 알아내도 P를 알아낼 수 있긴하겠지만 확률은 같음)

**디피헬만의 활용**

https로 먼저 키를 로그인서버로 보내기로 했다.

**우리는 현존하는 암호화를 대체하여 패스워드를 안전하게 암호화 하고자하는 것이 아니다.**

이를 구현하고자한다면 빅인트라는 초대형 숫자를 다루는것부터 구현이 되어야 할 것.

(단순 64비트로는 X)

**디피헬만 취약점**

디피헬만의 취약점으로는 중간자공격이 허용된다는 것이 있다.

중간자공격이란 상호간에 패킷을 주고받는 과정에서 누군가 끼어들어 그사람 행세를 하는 것을 뜻함.

따라서 우리는 외적으로 이러한 취약점을 막아야한다.

TCP를 사용한다면 '연결'이라는 개념이 들어가기 때문에 중간에 껴들수가 없음.

**https 사용**

Stateless로 서버를 만들때는 웹의 경우 https를 많이 썼었는데, 그냥 https를 써도 상관없다.

**[ 웹서버 - 클라(유니티) ]**

**https는 이미 다 만들어져있음. 단점으로는 송신자체의 부하가 크고, 디버깅이 불편하다.**

따라서 http를쓰되 나만의 암호화를 하기위해서 AES암호화를 채택하여 사용한다.

이 경우 키교환이 필요한 상황인데 키교환을 어떻게 할것인가?

디피헬만은 너무 복잡한데다 취약점이 존재한다. 따라서 OTP방식을 고민해볼 수 있음.

**휴대폰 앱 OTP**

클라가 서버에 로그인하려고 한다.

**서버는 별도장치로 키를 송신하고, 사용자는 별도장치에서 키를받아 서버에 로그인하는 방식.**

**별도 기기 OTP**

은행OTP는 조그맣게 기기로 제공되고, 버튼을 누르면 숫자가 찍힌다.

**이 기기는 네트워크/인터넷이 전혀되지않는데도 나온 찍힌 숫자를 입력하면 서버가 검증을 해줌.**

역시 수학적인 접근이다.

**별도 기기 OTP 검증방법**

OTP장치에는 시리얼번호가 정해져있고, 이를 검증하는 서버측에서는 이 사용자만의 고유의 시리얼 번호가 기입이 되어있어야한다. (따라서 처음OTP사용시 시리얼 번호 입력해야함)

**대부분 시간을 기준으로 검증을 거치게되는데, 현재 시간과 시리얼 코드에 대한 공식을 통해 임의의값을 만들어 낸다.** (서로의 시간이 같음을 전제함)

이렇게 만들어낸 값을 기기와 서버가 같은지 비교한다.

(해당공식은 알수없음. 알려지면 OTP가 다뚫릴것임)

**문제는 언제 버튼을 눌렀는지 시간을 정확히 파악할 수 없으므로, 허용범위를 두고 검증한다.**

1분간격으로 해도 5시 59분 59초에 눌렀다면 OTP가 바뀌게되므로, 서버에서 58분~ 2분까지를 허용해주는 식이다.

재발급하여 다시 검증하는 것이 더 낫겠지만, 사용자가 싫어하므로 이러한 방식을 택함.

**서버-기기간 시간이 틀어지는 경우**

기기를 3~4년을 썼다면 시간이 엄청나게 많이 틀어졌을 것이다.

(컴퓨터도 시간동기화를 하지않으면 3~4일만 지나도 10초이상 틀어진다)

**따라서 서버가 기기한테 맞추는 형식으로 구현한다.**

**기기로 OTP에 접속할때마다 서버에서 기기마다 틀어진시간을 갱신하여 값을 맞춘다.**

단, 시간이 너무많이 틀어진 경우에는 서버에서 갱신시켜주지않음.

**다시 키교환으로 돌아옴**

https를 쓰되 나만의 암호화를 하기위해 AES암호화를 채택하여 사용하는 경우.

키교환이 필요했고 여기에 디피헬만을 사용하기에는 너무 복잡하고 취약점도 존재하므로 OTP를 활용하면 되겠다.

**게임서버에서의 해킹**

**하지만 어차피 클라는 누구나 다운받을 수 있고, 클라에 키가 있기 때문에 모두 뚫어서 값을 조작하여 쏠 수있음.**

**이 조작패킷이 내 서버에 적용만 되지않는다면 컨펌이다.**

클라 대부분은 C#유니티로 만드는데 이 역시 모두 디컴파일되어 소스코드가 나오게된다.

이와 같은 이유로 프로토콜 암호화를 하지않아도 문제가 없음.

그래도 암호화를 하겠다고한다면 전세계에서 검증받은 암호화를 쓰는걸 권장한다.

어쩔수없이 내가만들어 사용해야 하는 경우에는 디피헬만과 같은 방식을 만들어 볼수있겠음.

(512비트 크기를 구현하는 방법도있겠고,

64비트를 10번 계산해서 10번의키를 만드는 방법도 있겠다.)

**특정 유저를 밴시키려면**

**이외에도 IP, 계정, 이더넷(MAC)등 다양한 것들을 밴하지만 핵을 쓰는 악성유저라고 하여 100%막을 수 없다. (해결 불가능)**

윈도우의 경우 OS라이센스가 한 컴퓨터당 하나밖에 안되는데, 하드웨어에 대한 스펙을 저장한다.  
(디스크, CPU메인보드.. 등)

디스크 CPU메인보드 이런것들.

어느하드웨어에서는 그게먹힌다. 전에 세팅했던게 먹혔음.

하드웨어가 컴퓨터 자체가 바뀌었지만 MS에서 인식을 못함, 얘는 맥같은걸로 안하고 CPU이런걸로해서.

근데 CPU가 바꾸면 인식을 못한다.

하드디스크 추가도 상관없지만 메인 C드라이브가 해당 시리얼번호가 바뀐다거나. 뭐이런거

이런식으로도 할수있고

보통 게임쪽에서는 맥을 가져다 쓰는거고

거의그거일것이다.

이거는 암호화, 지금설명드리는거는 밴.

**서버에서 IP가 바뀐경우**

로그인을 해놓고 쓰고있었는데 갑자기 IP가 바뀐경우, 옛날에는 누군가 개입해서 들어왔다고 판단하고 세션을 모두 날렸다.

하지만 지금은 모바일 환경이다 보니 걸어다님녀서 IP가 바뀐다. 따라서 이 계정에서 쓰는 IP를 DB에 저장하고 이미 저장된 IP일 경우 별도 인증없이 패스.

처음보는 IP라면 추가인증을 요구하거나 이메일을 보내도록 한다.