## Abstract

Though there are different ways to enumerate processes in Windows, none of them can be used on every version of the system. Some techinques that work perfectly on Windows NT fail on Windows 95 and vice versa. This article describes five different ways to enumerate Windows processes, which you can easily combine into one truly universal function.

# Enumerating Windows Processes

**Download source code for this article (.zip, 193KB)**
**Original article (in Russian)**

## Introduction

Win32 offers several methods to enumerate currently running processes. Unfortunately, none of them works on every Win32 platform. Software developers have to combine several methods in their applications to ensure they will run on every Windows version.

In this article we will examine the following enumeration methods:

- Using Process Status Helper (PSAPI) library
- Using ToolHelp32 API
- Using the undocumented **ZwQuerySystemInformation** function
- Using performance counters
- Using Windows Management Instrumentation interfaces

We will illustrate each method with a function that follows the prototype shown below:

```
// this callback function is called for each process in the enumeration
typedef BOOL (CALLBACK * PFNENUMPROC)(
    IN DWORD dwProcessId,       // process identifier
    IN PCTSTR pszProcessName,   // process name
    IN LPARAM lParam            // application-defined parameter
    );

// this function enumerates processes
BOOL EnumProcesses_Method(
    IN PCTSTR pszMachineName,   // computer name
    IN PFNENUMPROC pfnEnumProc, // application-defined callback function
    IN LPARAM lParam            // application-defined parameter
    );
```

The enumeration function calls an application-defined callback function for each running process, passing process information as the callback function arguments. The callback function is supposed to deal with the process information in according to the application logic. For example, in the demo application for this article, it adds an item to the list of processes.

Some of methods covered in this article allows enumeration of processes on remote computers, therefore the enumeration function prototype has the *pszMachineName* parameter. The NULL value for this parameter means enumeration of processes on the local machine.

## Method 1. Using Process Status Helper Library

The Process Status Helper library, also known as PSAPI, offers a set of functions that return certain information about processes and device drivers. The library is shipped with Windows 2000/XP and also available as a redistributable package for Windows NT 4.0.

For process enumeration purposes, PSAPI provides the **EnumProcesses** function that returns an array of identifiers of currently running processes. Below is the source code of the **EnumProcesses_PsApi** function that implements process enumeration using PSAPI.

⊟ Listing 1. Enumerating processes with PSAPI

```
#include <psapi.h>

BOOL WINAPI EnumProcesses_PsApi(
    IN LPCTSTR pszMachineName,
    IN PFNENUMPROC pfnEnumProc,
    IN LPARAM lParam
    )
{
    _ASSERTE(pfnEnumProc != NULL);

    if (pszMachineName != NULL)
        return SetLastError(ERROR_INVALID_PARAMETER), FALSE;

    HINSTANCE hPsApi;
    BOOL (WINAPI * pEnumProcesses)(DWORD *, DWORD, DWORD *);
    BOOL (WINAPI * pEnumProcessModules)(HANDLE, HMODULE*, DWORD, DWORD*);
    BOOL (WINAPI * pGetModuleFileNameEx)(HANDLE, HMODULE, LPTSTR, DWORD);

    // load PSAPI.DLL
    hPsApi = LoadLibrary(_T("psapi.dll"));
    if (hPsApi == NULL)
        return FALSE;

    // find necessary functions in PSAPI.DLL
    *(FARPROC *)&pEnumProcesses =
        GetProcAddress(hPsApi, "EnumProcesses");
    *(FARPROC *)&pEnumProcessModules =
        GetProcAddress(hPsApi, "EnumProcessModules");
#ifdef UNICODE
    *(FARPROC *)&pGetModuleFileNameEx =
        GetProcAddress(hPsApi, "GetModuleFileNameExW");
#else
    *(FARPROC *)&pGetModuleFileNameEx =
        GetProcAddress(hPsApi, "GetModuleFileNameExA");
#endif

    if (pEnumProcesses == NULL ||
        pEnumProcessModules == NULL ||
        pGetModuleFileNameEx == NULL)
    {
        FreeLibrary(hPsApi);
        return SetLastError(ERROR_PROC_NOT_FOUND), FALSE;
    }

    // get default process heap handle
    HANDLE hHeap = GetProcessHeap();

    DWORD dwError;
    DWORD cbReturned;
    DWORD cbAlloc = 128;
    DWORD * pdwIds = NULL;

    OSVERSIONINFO osvi;
    osvi.dwOSVersionInfoSize = sizeof(osvi);
    GetVersionEx(&osvi);

    DWORD dwSystemId = 8;
    if (osvi.dwMajorVersion >= 5)
        dwSystemId = (osvi.dwMinorVersion == 0) ? 2 : 4;

    do
    {
        cbAlloc *= 2;

        if (pdwIds != NULL)
            HeapFree(hHeap, 0, pdwIds);

        // allocate memory for the array of identifiers
        pdwIds = (DWORD *)HeapAlloc(hHeap, 0, cbAlloc);
        if (pdwIds == NULL)
        {
            FreeLibrary(hPsApi);
            return SetLastError(ERROR_NOT_ENOUGH_MEMORY), FALSE;
        }

        // get processes identifiers
        if (!pEnumProcesses(pdwIds, cbAlloc, &cbReturned))
        {
            dwError = GetLastError();

            HeapFree(hHeap, 0, pdwIds);
            FreeLibrary(hPsApi);
            return SetLastError(dwError), FALSE;
        }
    }
    while (cbReturned == cbAlloc);

    // now enumerate all identifiers in the array and call
    // the user-defined callback function for every process
    for (DWORD i = 0; i < cbReturned / sizeof(DWORD); i++)
    {
        BOOL bContinue;
        DWORD dwProcessId = pdwIds[i];

        // handle two special cases: Idle process (0) and
        // System process, which identifier depends on the
        // operating system version
        if (dwProcessId == 0)
        {
            bContinue = pfnEnumProc(0, _T("Idle"), lParam);
        }
        else if (dwProcessId == dwSystemId)
        {
            bContinue = pfnEnumProc(dwSystemId, _T("System"), lParam);
        }
        else
        {
            HANDLE hProcess;
            HMODULE hExeModule;
            DWORD cbNeeded;
            TCHAR szModulePath[MAX_PATH];
            LPTSTR pszProcessName = NULL;

            // open process handle
            hProcess = OpenProcess(
                            PROCESS_QUERY_INFORMATION|PROCESS_VM_READ,
                            FALSE, dwProcessId);
            if (hProcess != NULL)
            {
                if (pEnumProcessModules(hProcess, &hExeModule,
                                        sizeof(HMODULE), &cbNeeded))
                {
                    if (pGetModuleFileNameEx(hProcess, hExeModule,
                                             szModulePath, MAX_PATH))
                    {
                        pszProcessName =
                            _tcsrchr(szModulePath, _T('\\'));
                        if (pszProcessName == NULL)
                            pszProcessName = szModulePath;
                        else
                            pszProcessName++;
                    }
                }
            }

            CloseHandle(hProcess);

            // call the callback function
            bContinue = pfnEnumProc(dwProcessId, pszProcessName, lParam);
        }

        if (!bContinue)
            break;
    }

    HeapFree(hHeap, 0, pdwIds);
    FreeLibrary(hPsApi);

    return TRUE;
}
```

Note that we link with PSAPI.DLL dynamically, loading the library with **LoadLibrary** and then obtaining addresses of necessary functions using **GetProcAddress**. This will allow us to include this function into an application that has to run on

Windows 9x/Me, where PSAPI.DLL is not available. We will use this technique when implementing other enumeration methods, too.

The **EnumProcesses** function does not provide a way to know how much space is needed to receive all the identifiers. To deal with this limitation, we call **EnumProcesses** in a loop, increasing the the buffer size until the returned array size becomes less than the buffer size we allocated.

Since we want to obtain not only identifiers of processes but also their names, we have to do some additional work. For each process, we first get its handle with **OpenProcess** and then call **EnumProcessModules**, which returns a list of modules loaded into the process address space. The first module in the list is always the module representing the EXE-file used to create the process, so we get this handle and pass it to the **GetModuleFileNameEx** function (which is also a part of PSAPI) to obtain a path to the EXE-file. Finally, we strip path information from the file name and use this value as a process name.

Note the special handling of two processes in the source code of **EnumProcesses_PsApi**. We need to handle the System Idle Process, which identifier is always zero, and the System Process, which identifier depends on the operating system version, separately, because **OpenProcess** does not allow to obtain a handle to these processes and fails with ERROR_ACCESS_DENIED error.

If you run the demo application accompanying this article, you'll find that for at least one process the name is returned as "(name unavailable)". Using the Task Manager, it is easy to determine that this process is CSRSS.EXE. As it turns out, the system assigns such a security descriptor for this process, so it doesn't allow opening the process with access rights necessary to obtain its name.

On one hand, we could use the same approach as with two system processes. However, there is no guarantee that the identifier of this process is always the same. On the other hand, this issue can be resolved by enabling the SE_DEBUG_NAME privilege. When this privilege is turned on, the calling thread can open process handles with any access rights regardless the security descriptor assigned to the process. Since this privilege opens great opportunities to peneterate system security, it is normally granted only to system administrators. Because of this, we decided to not include the code for enabling this privilege into the **EnumProcesses_PsApi** function. If necessary, you can enable this privilege before calling **EnumProcesses_PsApi**, and the function will be able to return a name for the CSRSS.EXE process as well.

## Method 2. Using ToolHelp32 API

Microsoft Corporation had added a set of functions named ToolHelp API to Windows 3.1 to get independed software vendors access to system information which was previously available only to Microsoft engineers. When Windows 95 was born, these functions migrated into the new system with ToolHelp32 name.

The Windows NT operating system from very beginning provides an interface known as "performance data" to obtain similar information. This interface is very intricate and inconvenient (to be honest, beginning with Windows NT 4.0, Microsoft provides the Performance Data Helper library that greatly simplifies accessing performance data; we will use this library in one of our methods). Rumors are the Windows NT development team was refusing to include ToolHelp32 API into the system for long time, but anyway, starting with Windows 2000, you can see this API in the Windows NT family of operating systems.

When using ToolHelp32 API, we first create a snapshot of the list of processes with the **CreateToolhelp32Snapshot** function and then walk through the list using **Process32First** and **Process32Next** functions. The **PROCESSENTRY32** structure, which is filled by these functions, contains all the information we need. Below is the source code of the **EnumProcesses_ToolHelp** function that implements process enumeration using ToolHelp32 API.

⊟ Listing 2. Enumerating processes with ToolHelp32 API

```
#include <tlhelp32.h>

BOOL EnumProcesses_ToolHelp(
    IN LPCTSTR pszMachineName,
    IN PFNENUMPROC pfnEnumProc,
    IN LPARAM lParam
    )
{
    _ASSERTE(pfnEnumProc != NULL);

    if (pszMachineName != NULL)
        return SetLastError(ERROR_INVALID_PARAMETER), FALSE;

    HINSTANCE hKernel;
    HANDLE (WINAPI * pCreateToolhelp32Snapshot)(DWORD, DWORD);
    BOOL (WINAPI * pProcess32First)(HANDLE, PROCESSENTRY32 *);
    BOOL (WINAPI * pProcess32Next)(HANDLE, PROCESSENTRY32 *);

    // get handle of KERNEL32.DLL
    hKernel = GetModuleHandle(_T("kernel32.dll"));
    _ASSERTE(hKernel != NULL);

    // find necessary entrypoints in KERNEL32.DLL
    *(FARPROC *)&pCreateToolhelp32Snapshot =
        GetProcAddress(hKernel, "CreateToolhelp32Snapshot");
#ifdef UNICODE
    *(FARPROC *)&pProcess32First =
        GetProcAddress(hKernel, "Process32FirstW");
    *(FARPROC *)&pProcess32Next =
        GetProcAddress(hKernel, "Process32NextW");
#else
    *(FARPROC *)&pProcess32First =
        GetProcAddress(hKernel, "Process32First");
    *(FARPROC *)&pProcess32Next =
        GetProcAddress(hKernel, "Process32Next");
#endif

    if (pCreateToolhelp32Snapshot == NULL ||
        pProcess32First == NULL ||
        pProcess32Next == NULL)
        return SetLastError(ERROR_PROC_NOT_FOUND), FALSE;

    HANDLE hSnapshot;
    PROCESSENTRY32 Entry;

    // create a snapshot
    hSnapshot = pCreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (hSnapshot == INVALID_HANDLE_VALUE)
        return FALSE;

    // get information about the first process
    Entry.dwSize = sizeof(Entry);
    if (!pProcess32First(hSnapshot, &Entry))
        return FALSE;

    // enumerate other processes
    do
    {
        LPTSTR pszProcessName = NULL;

        if (Entry.dwSize > offsetof(PROCESSENTRY32, szExeFile))
        {
            pszProcessName = _tcsrchr(Entry.szExeFile, _T('\\'));
            if (pszProcessName == NULL)
                pszProcessName = Entry.szExeFile;
        }

        if (!pfnEnumProc(Entry.th32ProcessID, pszProcessName, lParam))
            break;

        Entry.dwSize = sizeof(Entry);
    }
    while (pProcess32Next(hSnapshot, &Entry));

    CloseHandle(hSnapshot);
    return TRUE;
}
```

Before calling **Process32First** or **Process32Next**, we have to initialize the **dwSize** field of the **PROCESSENTRY32** structure with the structure size. The functions, in their turn, fill this field with the number of bytes stored into the structure. We compare this value with the offset of the **szExeFile** field to determine whether the process name was returned.

## Method 3. Using the ZwQuerySystemInformation function

Despite the fact that a documented way exists to obtain the list of processes using performance data, the Windows NT Task Manager never used it. Instead, it calls the undocumented **ZwQuerySystemInformation** function. This function is exported from NTDLL.DLL and provides access to various system information and, in particular, to the list of currently running processes.

The **ZwQuerySystemInformation** function has the following prototype [2]:

```
NTSTATUS ZwQuerySystemInformation(
    IN ULONG SystemInformationClass,    // information class
    IN OUT PVOID SystemInformation,     // information buffer
    IN ULONG SystemInformationLength,   // size of information buffer
    OUT PULONG ReturnLength OPTIONAL    // receives information length
    );
```

**SystemInformationClass**

Specifies the type of the information to retrieve. We are interested in the SystemProcessesAndThreadsInformation (5).

**SystemInformation**

Pointer to a buffer that receives the information requested.

**SystemInformationLength**

Specifies the size of the receiving buffer.

**ReturnLength**

Optional pointer to a variable into which the function stores the number of bytes actually written into the output buffer.

The format of processes and threads information buffer is described by the **SYSTEM_PROCESS_INFORMATION** structure, which contains almost all the information displayed by the Task Manager. Below you can find the source code for the **EnumProcesses_NtApi** function that implements process enumeration using **ZwQuerySystemInformation**.

Listing 3. Enumerating processes using **ZwQuerySystemInformation**

```
typedef LONG    NTSTATUS;
typedef LONG    KPRIORITY;

#define NT_SUCCESS(Status) ((NTSTATUS)(Status) >= 0)

#define STATUS_INFO_LENGTH_MISMATCH      ((NTSTATUS)0xC0000004L)

#define SystemProcessesAndThreadsInformation    5

typedef struct _CLIENT_ID {
    DWORD         UniqueProcess;
    DWORD         UniqueThread;
} CLIENT_ID;

typedef struct _UNICODE_STRING {
    USHORT        Length;
    USHORT        MaximumLength;
    PWSTR         Buffer;
} UNICODE_STRING;

typedef struct _VM_COUNTERS {
    SIZE_T        PeakVirtualSize;
    SIZE_T        VirtualSize;
    ULONG         PageFaultCount;
    SIZE_T        PeakWorkingSetSize;
    SIZE_T        WorkingSetSize;
    SIZE_T        QuotaPeakPagedPoolUsage;
    SIZE_T        QuotaPagedPoolUsage;
    SIZE_T        QuotaPeakNonPagedPoolUsage;
    SIZE_T        QuotaNonPagedPoolUsage;
    SIZE_T        PagefileUsage;
    SIZE_T        PeakPagefileUsage;
} VM_COUNTERS;

typedef struct _SYSTEM_THREAD_INFORMATION {
    LARGE_INTEGER KernelTime;
    LARGE_INTEGER UserTime;
    LARGE_INTEGER CreateTime;
    ULONG         WaitTime;
    PVOID         StartAddress;
    CLIENT_ID     ClientId;
    KPRIORITY     Priority;
    KPRIORITY     BasePriority;
    ULONG         ContextSwitchCount;
    LONG          State;
    LONG          WaitReason;
} SYSTEM_THREAD_INFORMATION, * PSYSTEM_THREAD_INFORMATION;

typedef struct _SYSTEM_PROCESS_INFORMATION {
    ULONG             NextEntryDelta;
    ULONG             ThreadCount;
    ULONG             Reserved1[6];
    LARGE_INTEGER     CreateTime;
    LARGE_INTEGER     UserTime;
    LARGE_INTEGER     KernelTime;
    UNICODE_STRING    ProcessName;
    KPRIORITY         BasePriority;
    ULONG             ProcessId;
    ULONG             InheritedFromProcessId;
    ULONG             HandleCount;
    ULONG             Reserved2[2];
    VM_COUNTERS       VmCounters;
#if _WIN32_WINNT >= 0x500
    IO_COUNTERS       IoCounters;
#endif
    SYSTEM_THREAD_INFORMATION Threads[1];
} SYSTEM_PROCESS_INFORMATION, * PSYSTEM_PROCESS_INFORMATION;

BOOL EnumProcesses_NtApi(
    IN PCTSTR pszMachineName,
    IN PFNENUMPROC pfnEnumProc,
    IN LPARAM lParam
    )
{
    _ASSERTE(pfnEnumProc != NULL);

    if (pszMachineName != NULL)
         return SetLastError(ERROR_INVALID_PARAMETER), FALSE;

    HINSTANCE hNtDll;
    NTSTATUS (WINAPI * pZwQuerySystemInformation)(UINT, PVOID,
                                  ULONG, PULONG);

    // get NTDLL.DLL handle
    hNtDll = GetModuleHandle(_T("ntdll.dll"));
    _ASSERTE(hNtDll != NULL);

    // find ZwQuerySystemInformation address
    *(FARPROC *)&_ZwQuerySystemInformation =
        GetProcAddress(hNtDll, "ZwQuerySystemInformation");
    if (_ZwQuerySystemInformation == NULL)
        return SetLastError(ERROR_PROC_NOT_FOUND), FALSE;

    // get default process heap handle
    HANDLE hHeap = GetProcessHeap();

    NTSTATUS Status;
    ULONG cbBuffer = 0x8000;
    PVOID pBuffer = NULL;

    // it is difficult to predict what buffer size will be
    // enough, so we start with 32K buffer and increase its
    // size as needed
    do
    {
        pBuffer = HeapAlloc(hHeap, 0, cbBuffer);
        if (pBuffer == NULL)
            return SetLastError(ERROR_NOT_ENOUGH_MEMORY), FALSE;

        Status = pZwQuerySystemInformation(
                    SystemProcessesAndThreadsInformation,
                    pBuffer, cbBuffer, NULL);

        if (Status == STATUS_INFO_LENGTH_MISMATCH)
        {
            HeapFree(hHeap, 0, pBuffer);
            cbBuffer *= 2;
        }
        else if (!NT_SUCCESS(Status))
        {
            HeapFree(hHeap, 0, pBuffer);
            return SetLastError(Status), FALSE;
        }
    }
    while (Status == STATUS_INFO_LENGTH_MISMATCH);

    PSYSTEM_PROCESS_INFORMATION pProcesses =
        (PSYSTEM_PROCESS_INFORMATION)pBuffer;

    for (;;)
    {
        PCWSTR pszProcessName = pProcesses->ProcessName.Buffer;
        if (pszProcessName == NULL)
            pszProcessName = L"Idle";

#ifdef UNICODE

        if (!pfnEnumProc(pProcesses->ProcessId, pszProcessName, lParam))
            break;

#else

        CHAR szProcessName[MAX_PATH];
        WideCharToMultiByte(CP_ACP, 0, pszProcessName, -1,
                            szProcessName, MAX_PATH, NULL, NULL);

        if (!pfnEnumProc(pProcesses->ProcessId, szProcessName, lParam))
            break;

#endif
        if (pProcesses->NextEntryDelta == 0)
            break;

        // find the address of the next process structure
        pProcesses = (PSYSTEM_PROCESS_INFORMATION)(((LPBYTE)pProcesses)
                        + pProcesses->NextEntryDelta);
    }
    HeapFree(hHeap, 0, pBuffer);
    return TRUE;
}
```

When calling **ZwQuerySystemInformation**, it is difficult to predict which buffer size will be enough to receive all the information. Thus, we start with a 32K buffer and increase its size until the function returns success.

The **SYSTEM_PROCESS_INFORMATION** structure is a variable-size structure, the **NextEntryDelta** field specifies the offset to the next structure in the array. This field allows us to ignore the difference in size of the fixed part of the structure between Windows NT 4.0 and Windows 2000, where the **IoCounters** field was added to the structure.

## Method 4. Using Performance Counters

As it was already noted, the Windows NT operating system from very beginning had an interface to obtain various system information in the form of performance counters. This interface is far from intuitive. In order to get the information, you have to read a value with a specially formed name from the HKEY_PERFORMANCE_DATA registry key. The information is returned in the form of deeply nested structures, many of which are of variable size. Dealing with these data structures requires a certain degree of diligence.

Fortunately, the situation had changed with introducing the Performance Data Helper (PDH) library in Windows NT 4.0. This library provides more convenient interface to performance data. However, the library wasn't being shipped with Windows NT 4.0, it was available as a redistributable in the Microsoft Platform SDK. Beginning with Windows 2000, PDH became a part of the system.

A detailed discussion of performance monitoring is beyond the scope of this article. Just let me note that the Windows NT performance monitoring architecture defines a concept of an *object*, for which performance counting is made. Examples of objects are processor and disk drive. Each object can have one or more *instances* along with a set of *performance counters*. Our ultimate goal is to enumerate all instances of the object named "Process", finding the value of the performance counter "ID Process" for each instance. Below is the code of the **EnumProcesses_PerfData** function, which uses PDH for process enumeration.

⊞ Listing 4. Enumerating processes using PDH

We return instance names as process names. If you run the demo application, you'll see that instance names are names of EXE-files of corresponding processes without an extension.

Object names and performance counter names are localizable. That means that, for example, on Russian version of Windows NT, process object and process identifier counter are no longer called "Process" and "ID Process", localized names are used instead. To get the localized names and format a full path to the performance counter we are interested in, we use a helper function named **PerfFormatCounterPath**. Its source code is provided below.

⊟ Listing 5. **PerfFormatCounterPath** source code

```
static PDH_STATUS PerfFormatCounterPath(
    IN HINSTANCE hPdh,
    IN LPCTSTR pszMachineName,
    IN LPWSTR pszPath,
    IN ULONG cchPath
    )
{
    _ASSERTE(hPdh != NULL);
    _ASSERTE(_CrtIsValidPointer(pszPath, cchPath * sizeof(WCHAR), 1));

    PDH_STATUS (WINAPI * pPdhMakeCounterPath)(
        PDH_COUNTER_PATH_ELEMENTS_W *, LPWSTR, LPDWORD, DWORD);
    PDH_STATUS (WINAPI * pPdhLookupPerfNameByIndex)(
        LPCWSTR, DWORD, LPWSTR, LPDWORD);

    *(FARPROC *)&pPdhMakeCounterPath =
        GetProcAddress(hPdh, "PdhMakeCounterPathW");
    *(FARPROC *)&pPdhLookupPerfNameByIndex =
        GetProcAddress(hPdh, "PdhLookupPerfNameByIndexW");

    if (pPdhMakeCounterPath == NULL ||
        pPdhLookupPerfNameByIndex == NULL)
        return ERROR_PROC_NOT_FOUND;

    PDH_STATUS Status;
    DWORD cbName;
    WCHAR szObjectName[80];
    WCHAR szCounterName[80];

    LPWSTR pszMachineW = NULL;

#ifndef _UNICODE
    WCHAR szMachineName[256];
    if (pszMachineName != NULL)
    {
        MultiByteToWideChar(CP_ACP, 0, pszMachineName, -1,
                            szMachineName, 256);
        pszMachineW = szMachineName;
    }
#else
    pszMachineW = (LPWSTR)pszMachineName;
#endif

    // find localize Process object name
    cbName = sizeof(szObjectName);
    Status = pPdhLookupPerfNameByIndex(pszMachineW, 230,
                                       szObjectName, &cbName);
    if (Status != ERROR_SUCCESS)
        return Status;

    // find localized ID Process counter name
    cbName = sizeof(szCounterName);
    Status = pPdhLookupPerfNameByIndex(pszMachineW, 784,
                                       szCounterName, &cbName);
    if (Status != ERROR_SUCCESS)
        return Status;

    PDH_COUNTER_PATH_ELEMENTS_W pcpe;
    pcpe.szMachineName = pszMachineW;
    pcpe.szObjectName = szObjectName;
    pcpe.szInstanceName = L"*";
    pcpe.szParentInstance = NULL;
    pcpe.dwInstanceIndex = 0;
    pcpe.szCounterName = szCounterName;
    // format full counter path
    return pPdhMakeCounterPath(&pcpe, pszPath, &cchPath, 0);
}
```

We use **PdhLookupPerfNameByIndex** to obtain localized names from corresponding indices. Indices of standard objects and performance counters are well-defined and don't depend on the operating system version, thus it is safe to specify them directly in the code.

Note, among all process enumeration methods we discussed so far, this is the first method that allows to enumerate processes on another machine. All we have to do is to set the machine name in the **szMachineName** field of the **PDH_COUNTER_PATH_ELEMENTS** structure.

## Method 5. Using Windows Management Instrumentation

Windows Management Instrumentation (WMI) is Microsoft implementation of the Web-Based Enterprise Management (WBEM) initiative. WBEM provides a point of integration through which data from different management sources can be uniformly accessed, and it complements and extends existing management protocols such as SNMP. WBEM is based on the Common Information Model (CIM) schema, which is an industry standard maintained by DMTF (Distributed Management Task Force). WMI is included in Windows 2000 and Windows XP, but is also available as a redistributable package for Windows 9x/Me and Windows NT 4.0.

Any detailed discussion of WMI extends beyond the scope of this article. I'll just present the code that enumerates processes using WMI interfaces.

⊟ Listing 6. Enumerating processes using WMI interfaces

```
#include <comdef.h>
#include <SshWbemHelpers.h>

BOOL EnumProcesses_Wmi(
    IN LPCTSTR pszMachineName,
    IN PFNENUMPROC pfnEnumProc,
    IN LPARAM lParam
    )
{
    _ASSERTE(pfnEnumProc != NULL);

    try
    {
        HRESULT hRes;
        TCHAR szServer[256];
        _bstr_t bstrServer;
          IWbemLocatorPtr spLocator;
        IWbemServicesPtr spServices;
        IEnumWbemClassObjectPtr spEnum;
        IWbemClassObjectPtr spObject;
        ULONG ulCount;

        // create WBEM locator object
        hRes = CoCreateInstance(__uuidof(WbemLocator), NULL,
                    CLSCTX_INPROC_SERVER, __uuidof(IWbemLocator),
                    (PVOID *)&spLocator);
        if (FAILED(hRes))
            _com_issue_error(hRes);

        if (pszMachineName == NULL)
            lstrcpy(szServer, _T("root\\cimv2"));
        else
        {
             wsprintf(szServer, _T("\\\\%s\\root\\cimv2"),
                     pszMachineName);
        }

          // connect with the specified machine
        hRes = spLocator->ConnectServer(_bstr_t(L"root\\cimv2"), NULL,
                                NULL, NULL, 0, NULL, NULL,
                                &spServices);
        if (FAILED(hRes))
            _com_issue_error(hRes);

        // create enumerator
        hRes = spServices->CreateInstanceEnum(_bstr_t(L"Win32_Process"),
                      WBEM_FLAG_SHALLOW|WBEM_FLAG_FORWARD_ONLY,
                      NULL, &spEnum);
        if (FAILED(hRes))
            _com_issue_error(hRes);

        // enumerate processes
        while (spEnum->Next(WBEM_INFINITE, 1,
                    &spObject, &ulCount) == S_OK)
        {
            _variant_t valId;
            _variant_t valName;

            // get process identifier
            hRes = spObject->Get(L"ProcessId", 0, &valId, NULL, NULL);
            if (FAILED(hRes))
            {
                spObject = NULL;
                continue;
            }

            // get process name
            hRes = spObject->Get(L"Name", 0, &valName, NULL, NULL);
            if (FAILED(hRes))
            {
                spObject = NULL;
                continue;
            }

            _ASSERTE(valId.vt == VT_I4);
            _ASSERTE(valName.vt == VT_BSTR);

            if (!pfnEnumProc(valId.lVal, (LPCTSTR)_bstr_t(valName),
                     lParam))
                break;
            spObject = NULL;
        }
    }
    catch (_com_error& err)
    {
        return SetLastError(err.Error()), FALSE;
    }

    return TRUE;
}
```

The fact that WMI is based on the COM technology saves us from explicit loading of necessary libraries as we did in all previous methods. The same fact requires you to initialize COM run-time before calling the process enumeration function. In an MFC application you can do it with **AfxOleInit**, in other cases you should use **CoInitialize** or **CoInitializeEx** functions.

Furthermore, using WMI requires initialization of the COM security with a call to **CoInitializeSecurity**:

```
    CoInitializeSecurity(
          NULL,
          -1,
          NULL,
          NULL,
          RPC_C_AUTHN_LEVEL_CONNECT,
          RPC_C_IMP_LEVEL_IMPERSONATE,
          NULL,
          EOAC_NONE,
          0);
```

Note, same as the previous process enumeration method, this method allows enumeration of processes on remote machines. To enumerate processes remotely, you should specify computer name to **IWbemLocator::ConnectServer**.

**Important note:** Remote process enumeration in the form as it presented in this article will work only if the current user of the local machine has administrative privileges on the remote machine. The samples provided above do not allow to change credentials with which a connection to the remote machine is made. To specify alternative credentials in PDH method, you should call **NetUseAdd** or **WNetAddConnection2** to establish a connection with IPC$ resource on the remote machine prior to calling the enumeration function. In WMI method, you should use standard COM methods for supplying alternative credentials.

## Wrap Up

Well, we've seen five different methods to enumerate processes on Windows. None of them is universal, since there is alvays a Windows version there a particular method won't run. The table below summarizes applicability of the described methods.

| | **Windows 9x/Me** | **Windows NT 4.0** | **Windows 2000/XP** |
|---|---|---|---|
| **Method 1** | No | Yes* | Yes |
| **Method 2** | Yes | No | Yes |
| **Method 3** | No | Yes | Yes |
| **Method 4** | No | Yes* | Yes |
| **Method 5** | Yes* | Yes* | Yes |

* Requires installation of additional components.

Using this table and the functions provided in the previous sections, it is easy to make a function for process enumeration, which will work on all Windows versions. For example, if you want to use PSAPI on Windows NT/2000/XP and ToolHelp32 API on Windows 9x/Me, such a function can look like following:

```
BOOL MyEnumProcesses(
    IN PFNENUMPROC pfnEnumProc,
    IN LPARAM lParam
    )
{
    OSVERSIONINFO osvi;
    osvi.dwOSVersionInfoSize = sizeof(osvi);

    if (osvi.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS)
        return EnumProcesses_ToolHelp(pfnEnumProc, lParam);
    else if (osvi.dwPlatformId == VER_PLATFORM_WIN32_NT)
        return EnumProcesses_PsApi(pfnEnumProc, lParam);
    else
        return SetLastError(ERROR_CALL_NOT_IMPLEMENTED), FALSE;
}
```

This article is accompanied by a small demo application, ambitiously named Process Viewer, that demonstrates all five methods presented in the article. The implementation of all methods is in **enumproc.h** and **enumproc.cpp** files. I tried to arrange sources in such a way to make easier their reuse in other projects. To build the application you will need Microsoft Platform SDK. Happy coding!

## References

1. *HOWTO: Enumerate Applications in Win32*, Q175030, Microsoft Knowledge Base.
2. Gary Nebbett, *Windows NT/2000 Native API Reference*. New Riders Publishing, 2000.
3. *INFO: Using PDH APIs Correctly in a Localized Language*, Q287159, Microsoft Knowledge Base.

## Leave your comment

| | |
|---|---|
| **From:** | |
| **Subject:** | |
| **Comment:** | |

Submit