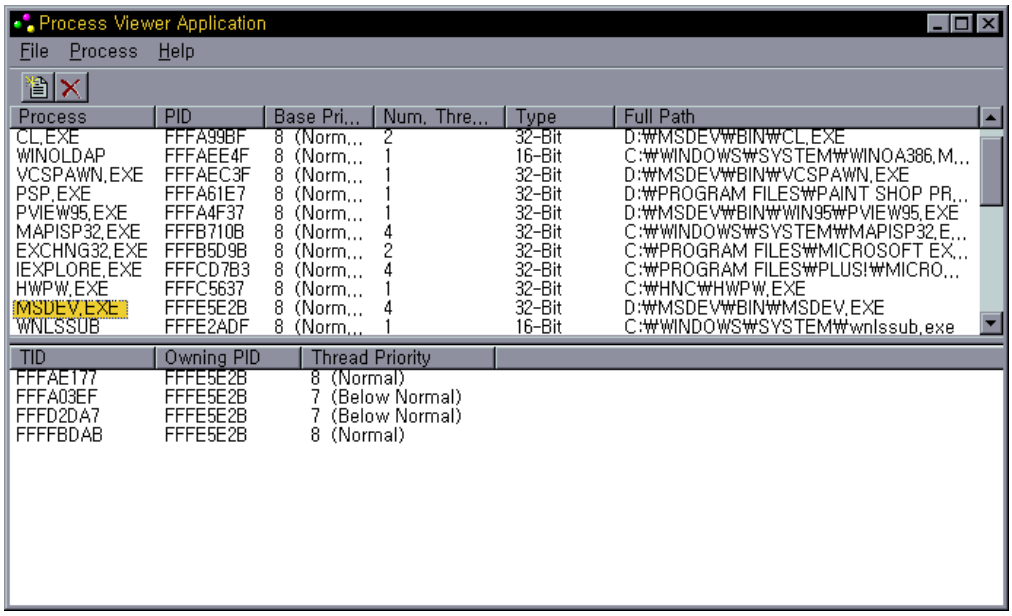
7 장 윈도 95 프로그래밍 활용

프로세스와 스레드

프로세스(process)란 자신의 주소 공간과 코드, 데이터 그리고 다른 운영체제의 자원으로 이루어진 실행중인 응용프로그램이다. 운영체제의 자원이란, 파일이나 동기화 객체같은 것을 말한다. 프로세스는 또한 프로세스 내에서 수행되는 하나 이상의 쓰레드를 포함하는데, 쓰레드(thread)란 운영체제가 CPU 시간을 할당하는 기본적인 단위이다. 쓰레드는 응용프로그램의 어떤 부분의 코드라도 수행할 수 있으며 여기에는 다른 쓰레드에 의해 수행되고 있는 부분도 포함된다. 프로세스의 모든 쓰레드는 주소공간, 전역변수, 운영체제의 자원을 공유한다.

도스나 윈도 3.1 같은 16 비트 운영체제에서는 하나의 주소공간에 모든 프로세스가 존재했다. 따라서 하나의 응용프로그램이 다른 응용프로그램의 주소공간을 마음대로 침범할 수가 있었다. 그렇기 때문에 윈도 3.1 에서는 실행중인 어떤 프로그램이 오동작하면 시스템 전체를 다시 기동시켜야 하는 일이 많았다. 왜냐하면 오동작하는 프로그램이 시스템의 주소공간에까지도 접근할 수 있고, 이 프로그램이 시스템 주소 영역에 엉뚱한 데이터를 써넣으면 시스템 전체가 오동작하게 되기 때문이었다. 하지만 윈도 95 에서는 오동작하는 프로그램만 CTRL+ALT+Del 키를 눌러 종료시키면 다른 프로그램들은 그대로 잘 동작한다. 언제나 그런 것은 아니지만 말이다. 그렇다면 어떻게 윈도 95 에서는 이런 일이 가능하게 되었을까? 윈도 95 에서는 각 프로세스마다 독립된 주소공간을 할당하기 때문에 어떤 프로세스가 다른 프로세스의 주소 공간에 데이터를 쓸 수 없게 되어버렸다. 그렇다면 윈도 95 에서 16 비트 응용프로그램들은 어떤방식으로 동작하게 될까? 16 비트 응용프로그램들이 실행되면, 16 비트 응용프로그램을 위한 주소공간이 할당되고, 다른 16 비트 응용프로그램들도 모두 이 할당된 주소공간 안에서 실행된다.

윈도 3.1 에서는 하나의 프로세스에는 하나의 쓰레드만이 존재하는 형식이었지만 윈도 95 는 하나의 프로세스 안에 여러개의 쓰레드가 존재할 수 있게 되었다. 이 때 다른 쓰레드를 생성하는 쓰레드를 주 쓰레드(primary thread)라고 한다.



윈도 95 가 다중 쓰레드를 지원한다고 해서, 다중 쓰레드를 함부로 사용해서도 안된다. 왜냐하면 대부분의 작업은 단일 쓰레드로도 충분하며, 사실 다중 쓰레드 프로그래밍은 그렇게 쉬운 것이 아니기 때문이다. 물론 디버깅하기도 아주 까다롭다. 그렇다면 어떤 경우에 다중 쓰레드를 사용하는 것이 좋을까? Microsoft Developet Studio 가 좋은 예가 되겠다. 컴파일을 시작하면 윈도 3.1 에서 사용되던 컴파일러와는 달리, Output 창에 메시지가 출력되면서 컴파일, 링크가 이루어지는 동안에도 메뉴에 접근할 수 있으며, 소스코드를 편집할 수도 있다. 각 창에서 이루어지는 일들이 서로 다른 쓰레드로 동작하기 때문에 가능한 일이라 하겠다.

Microsoft Developet Studio 에는 Process Viewer Application 이라는 프로그램이 있는데 이 프로그램은 현재 실행 중인 프로세스와 각 프로세스에 대한 쓰레드의 상태를 볼 수 있다.

프로세스 생성

윈도 3.1 에서는 LoadModule 이나 WinExec 함수를 이용해서 프로세스를 생성했다. 윈도 95 에도 이들 함수가 존재하기는 하지만, 이는 호환성을 위해 존재할 뿐이고 윈도 95 에서는 CreateProcess 함수를 사용한다.

```
BOOL CreateProcess(

  LPCTSTR  lpApplicationName,      // pointer to name of executable module
  LPTSTR  lpCommandLine,      // pointer to command line string
  LPSECURITY_ATTRIBUTES  lpProcessAttributes,
        // pointer to process security attributes
  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
        // pointer to thread security attributes
  BOOL  bInheritHandles,   // handle inheritance flag
  DWORD  dwCreationFlags,     // creation flags
  LPVOID  lpEnvironment,  // pointer to new environment block
  LPCTSTR  lpCurrentDirectory,// pointer to current directory name
  LPSTARTUPINFO  lpStartupInfo,     // pointer to STARTUPINFO
  LPPROCESS_INFORMATION  lpProcessInformation
        // pointer to PROCESS_INFORMATION
 );
```

[인수]

LPCTSTR lpApplicationName

실행할 모듈의 문자열 포인터. 이 문자열은 절대 경로(full-path)와 모듈의 파일이름으로 구성될 수도 있고, 상대 경로(partial-path)일 수도 있다. 상대 패쓰일 경우에는 물론 현재 드라이브와 현재 디렉토리가 적용된다.

이 문자열이 NULL 일 수도 있는데, 이런 경우에는 lpCommandLine 에서 첫 번째 공백이나 탭으로 구분되는 단어가 실행할 모듈의 이름이 된다.

윈도용 프로그램 뿐만 아니라, 도스용 프로그램도 가능하다.

LPTSTR lpCommandLine

실행시킬 모듈에 대한 명령행의 문자열 포인터. 이 값이 NULL 이면 lpApplicationName 만으로 명령행이 구성되어 사용된다. lpApplication 과 lpCommandLine 이 모두 NULL 이 아니면, *lpApplication 은 실행할 모듈의 이름이 되고, *lpCommandLine 은 그 실행할 모듈에 대한 명령행이 된다. 전체 명령행을 읽어오려면 GetCommandLine 함수를 이용하면 된다.

LPTSTR GetCommandLine(VOID)

lpApplicationName 이 NULL 이면 첫 번째 공백이나 탭으로 구분되는 단어가 모듈의 이름이 된다. 확장자가 없다면 .EXE 로 간주하며, 확장자 없이 파일이름 끝에 '.'이 있거나 파일이름에 경로(path)가 포함되어 있다면 .EXE 를 붙이지 않는다. 파일이름이 디렉토리 경로를 포함하지 않는다면 다음과 같은 순서로 실행파일을 찾는다.

1. 응용프로그램이 기동된 디렉토리

2. 부모 프로세스의 현재 디렉토리

3. 윈도 시스템 디렉토리.(GetSystemDirectory 로 얻을 수 있다)

4. 윈도 디렉토리(GetWindowsDirectory 로 얻을 수 있다)

5. 환경 변수 PATH 에 있는 디렉토리

LPSECURITY_ATTRIBUTES lpProcessAttributes

만들 프로세스의 보안 속성을 지정하기 위한 SECURITY_ATTRIBUTES 구조체에 대한 포인터.

이 값이 NULL 이면 기본 보안 속성을 갖는다.

```
typedef struct_SECURITY_ATTRIBUTES { // sa
    DWORD  nLength;
    LPVOID lpSecurityDescriptor;
    BOOL   bInheritHandle;
} SECURITY_ATTRIBUTES;
```

nLength 는 이 구조체의 크기.

lpSecurityDescriptor 는 프로세스의 소유자와 그룹을 식별하는 정보가 들어 있음

bInheritHandle 는 지금 생성하는 객체의 핸들이 상속가능한 지를 나타낸다. 이 값이 거짓이면 아래에 설명하게 될 bInheritHandles 의 값은 무시된다. 따라서 부모 프로세스의 객체가 자식 프로세스 객체에 상속되려면 이 SECURITY_ATTRIBUTES 의 bInheritHandle 가 참이고 아래에 나올 bInheritHandles 의 값이 참이여야 한다.

프로세스란 별도의 주소공간을 갖기 때문에 여기서 말하는 객체의 핸들을 상속한다는 말은 부모 프로세스의 객체 핸들 변수 그 자체를 자식 프로세스도 사용할 수 있다는 말이 아니라, 부모 프로세스의 객체 핸들값 자체를 자식 프로세스도 사용할 수 있다는 뜻이다. 예를 들어 어떤 파일을 열어 그 핸들을 얻었는데 그 핸들값이 100 이라고 하면 자식 프로세스도 이 100 이라는 핸들값으로 부모 프로세스가 연 파일에 접근할 수 있다는 것이다. 하지만 부모 프로세스와 자식 프로세스는 별도의 주소공간에 있기 때문에 서로 상대방의 핸들값을 읽어올 수 없다. 따라서 이렇게 상속받은 핸들값을 이용하려면 프로세스간 통신을 이용해야 하는데, 프로세스간 통신에 대해서는 다음 단원에서 다루겠다.

LPSECURITY_ATTRIBUTES lpThreadAttributes

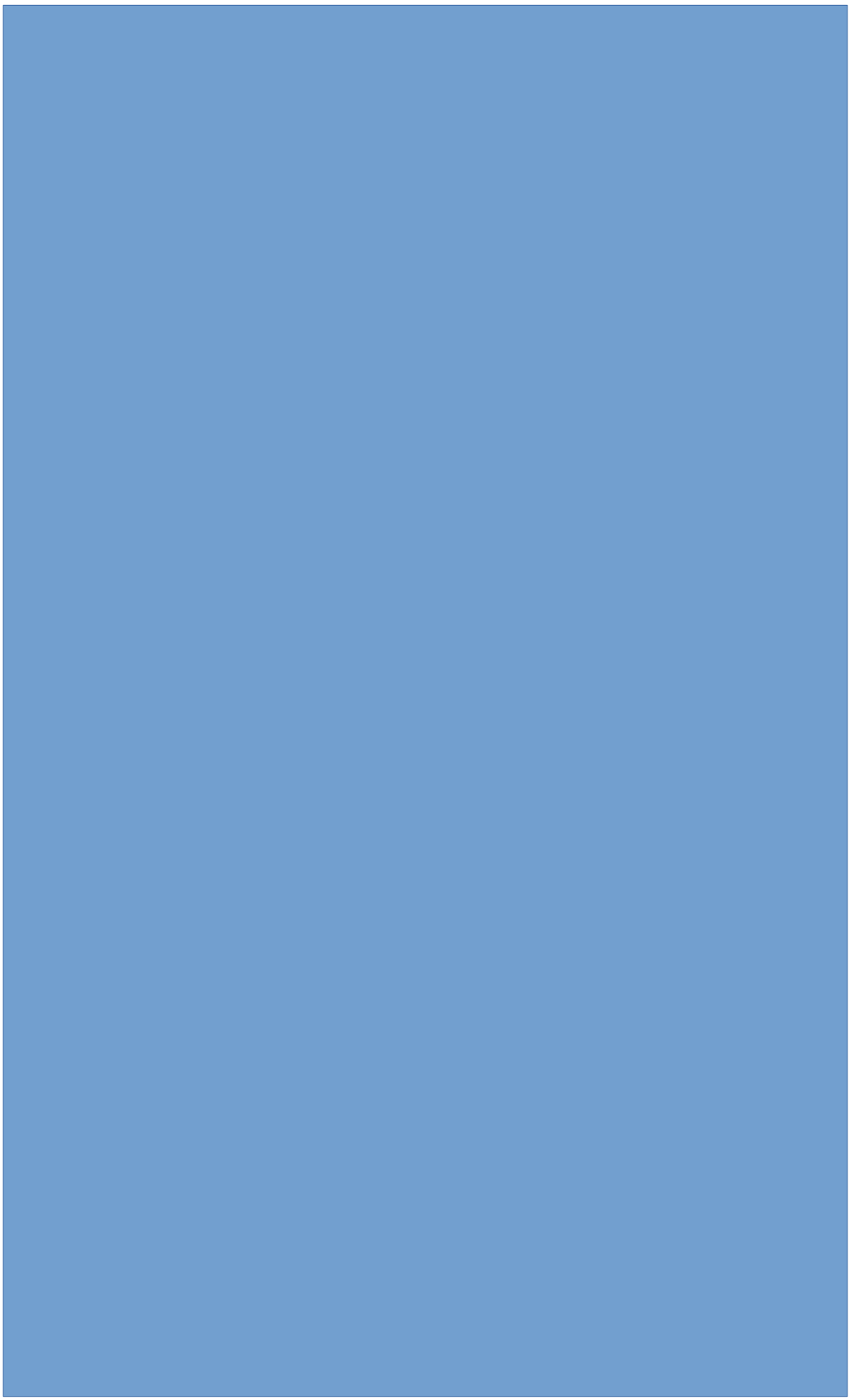
새로 만들어지는 프로세스의 주 쓰레드에 대한 보안속성을 지시하기 위한 SECURITY_ATTRIBUTES 구조체에 대한 포인터.

BOOL bInheritHandles

새로 만들어지는 프로세스가 이 프로세스를 만든 프로세스의 핸들을 상속할 것인지 결정. 이 값이 참이면 상속 가능한 핸들은 모두 새 프로세스에 상속된다. 상속된 핸들은 원래의 핸들과 같은 값과 같은 접근 특성을 갖는다.

DWORD dwCreationFlags

우선권 클래스와 프로세스의 생성 대한 제어를 위한 플래그



dwCreationFlags 는 새 프로세스의 우선권 클래스의 제어에도 사용되는데, 자세한 내용은 아래의 표를 참고한다.

LPVOID lpEnvironment

새로운 프로세스에 대한 환경 블록에 대한 포인터. 이 값이 NULL 이면 새 프로세스는 이 프로세스를 만든 프로세스의 환경 블록을 사용한다.

환경 블록은 NULL 로 끝나는 문자열들로 이루어 진다. 각 문자열은 다음과 같은 형식이다.

name=value

등호(=)가 구분자로 사용되며 등호는 환경변수의 이름에는 사용될 수 없다.

환경변수 블록은 유니코드 문자일 수도 있고 안시(ANSI) 문자일 수도 있는데, dwCreationFlag 가 CREATE_UNICODE_ENVIRONMENT 플래그가 설정되어 있으면 유니코드이고 설정되어 있지 않으면 안시이다. 안시는 두 개의 0 바이트로 끝나는데, 하나는 마지막 문자열의 끝을 표시하는 것이고 또 하나는 이 환경블록의 끝을 표시하는 것이다. 유니코드일 때는 4 개의 0 바이트로 끝나는데, 2 개는 마지막 문자열에 대한 것이고, 나머지 2 개는 환

경 블록에 대한 것이다.

LPCTSTR lpCurrentDirectory

자식 프로세스에 대한 현재 드라이브와 현재 디렉토리를 가리키는 문자열에 대한 포인터. 드라이브 이름까지 포함하는 절대경로(full path)여야 한다. 이 값이 NULL 이면 자식 프로세스는 부모 프로세스와 같은 현재 드라이브와 현재 디렉토리를 갖는다.

LPSTARTUPINFO lpStartupInfo

새로운 프로세스에 대한 주 윈도가 어떻게 나타날지를 지정한다. STARTUPINFO 구조체에 대한 포인터

```
typedef struct _STARTUPINFO { // si
    DWORD   cb;
    LPTSTR  lpReserved;
    LPTSTR  lpDesktop;
    LPTSTR  lpTitle;
    DWORD   dwX;
    DWORD   dwY;
    DWORD   dwXSize;
    DWORD   dwYSize;
    DWORD   dwXCountChars;
    DWORD   dwYCountChars;
    DWORD   dwFillAttribute;
    DWORD   dwFlags;
    WORD    wShowWindow;
    WORD    cbReserved2;
    LPBYTE  lpReserved2;
    HANDLE  hStdInput;
    HANDLE  hStdOutput;
    HANDLE  hStdError;
} STARTUPINFO, *LPSTARTUPINFO;
```

LPPROCESS_INFORMATION lpProcessInformation

새로운 프로세스에 대한 식별 정보를 읽어오기 위한 PROCESS_INFORMATION 구조체에 대한 포인터

```
typedef struct _PROCESS_INFORMATION { // pi
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION;
```

[반환값]

성공하면 TRUE, 실패하면 FALSE 를 반환한다. 오류의 종류를 알려면 GetLastError 를 호출하면 된다.

[설명]

CreateProcess 함수는 새 프로그램을 수행시키기 위해 사용된다. 윈도 95 환경에서도 WinExec 나 LoadModule 함수도 여전히 사용할 수 있지만, 이들 함수들도 내부적으로는 CreateProcess 함수로 구현되어 있다.

CreateProcess 함수는 프로세스를 뿐만 아니라 쓰레드 객체도 만든다.

새로 만들어진 프로세스에는 32 비트 프로세스 식별자가 할당 된다. 이 식별자는 프로세스가 끝날 때까지 유효하다. 어떤 프로세스에 대한 핸들을 열기 위해 OpenProcess 함수를 사용할 때 이 식별자가 이용된다. 프로세스의 초기 쓰레드에도 역시 32 비트 쓰레드 식별자가 할당되며 이 쓰레드 식별자는 시스템 내의 다른 식별자들과 구별되는 유일한 값을 갖는다. 프로세스와 쓰레드에 대한 식별자는 PROCESS_INFORMATION 구조체에 넣어 돌려준다.

lpApplication, lpCommandLine 에 오는 응용프로그램 이름은 확장자가 없어도 된다. 하지만 .COM 파일의 경우에는 확장자 .COM 을 명시해 주어야 한다.

새로운 프로세스가 그 초기화를 마치고 대기 중인 사용자 입력이 없는 상태에서 사용자의 입력을 기다리는 상태가 되기까지 기다리고 싶다면 WaitForInputIdle 함수를 이용한다.

```
DWORD WaitForInputIdle(
    HANDLE  hProcess,  // handle to process
   DWORD  dwMilliseconds  // time-out interval in milliseconds
  );
```

이 함수는 부모 프로세스와 자식 프로세스 사이의 동기를 맞추는데 사용된다. 왜냐하면 CreateProcess 함수는 새 프로세스가 그 초기화를 끝낼 때까지 기다리지 않고 바로 돌아오기기 때문이다. 예를 들어 어떤 부모 프로세스가 새로운 자식 프로세스를 만들고, 그 새로 만들어진 자식 프로세스의 창을 이용해야 한다면 WaitForInputIdle 함수가 아주 유용할 것이다.

프로세스를 종료할 때는 ExitProcess 함수를 사용하는 것이 좋다. 왜냐하면 이 함수는 종료하려는 프로세스에 붙은(attach) 모든 DLL 들에게도 프로세스가 종료한다는 사실을 알려주기 때문이다. 쓰레드가 ExitProcess 함수를 호출하면 프로세스의 다른 쓰데드들은 사용중인 DLL 에 대한 종료과정도 거치지 않은 상태로 즉시 종료된다는 사실에 주의해야 한다.

ExitProcess, ExitThread, CreateThread 와 시작 중인 프로세스는 각각 하나의 프로세스 안에서 "순서대로 나열"된다. 순서대로 나열된다는 말은 한번에 하나의 주소공간 안에서 쓰레드의 생성이나 종료에 대한 사건이 동시에 일어날 수는 없다는 뜻이다.

(가) 프로세스가 기동되고 DLL 이 초기화되는 도중에는 새로운 쓰레드가 생성되더라도 프로세스의 초기화 작업이 끝날 때까지 쓰레드의 수행이 시작되지 않는다.

(나) 하나의 프로세스안에서 한번에 오직 하나의 쓰레드만이 DLL 의 초기화나 DLL 해지 과정 중에 있을 수 있다.

(다) ExitProcess 함수는 쓰레드가 DLL 초기화나 해지 과정 중일 때

새로 만들어진 프로세스는 그 프로세스에 포함된 모든 쓰레드가 종료될 때까지 그리고 그 프로세스와 쓰레드에 대한 모든 핸들이 CloseHandle 함수를 통해 닫힐 때까지 시스템에 그대로 남아있게 된다. 프로세스와 주 쓰레드에 대한 핸들은 CloseHandle 함수를 이용해서 꼭 닫아주어야 한다. 이 핸들이 필요하지 않다면 프로세스를 만든 직후에 곧바로 닫아주는 것이 가장 좋은 방법이다.

프로세스 내의 마지막 쓰레드가 끝날 때, 다음과 같은 일이 일어난다.

(가) 프로세스가 열어 둔 모든 객체는 닫힌다.

(나) GetExitCodeProcess 함수로 얻은 프로세스의 종료상태는 STILL_ACTIVE 인 초기 상태에서 종료되는 마지막 쓰레드의 종료상태로 바뀐다.

(다) 주 쓰레드의 쓰레드 객체는 신호상태(signaled status)로 된다. 따라서 더 이상 대기 중인 쓰레드는 없다.

(라) 프로세스 객체는 신호상태로 된다. 따라서 더 이상 대기 중인 쓰레드는 없다.

드라이브 C 의 현재 디렉토리가 \HNC\APP 라면, 환경 변수 =C:는 그 값이 C:\HNC\APP 이다. lpEnvironment 부분에서 말했던 것처럼 CreateProcess 함수의 lpEnvironment 인수가 NULL 이 아니면 시스템의 현재 디렉토리 정보가 새 프로세스로 자동으로 전달되지 않는다. 이런 경우, 응용프로그램은 현재 디렉토리 정보를 넘겨주어야 하는데, 이렇게 하기 위해서는 =X 환경 변수 문자열을 만들어야 한다. 그 다음에 이 변수를 lpEnvironment 가 지시하는 환경 변수 블록에 넣어 주어야 한다.

드라이브 X 에 대한 현재 디렉토리 정보를 얻기 위한 또 다른 방법으로는 GetFullPathName("X:", ...)이 있다.

```
DWORD GetFullPathName(
    LPCTSTR  lpFileName,     // address of name of file to find path for
   DWORD  nBufferLength,  // size, in characters, of path buffer
   LPTSTR  lpBuffer,       // address of path buffer
   LPTSTR  *lpFilePart    // address of filename in path
  );
```

새로운 프로세스의 기본 설정 현재 디렉토리는 루트 디렉토리이기 때문에 GetFullPathName 이 돌려주는 환경변수가 X:\이면 그 값을 새 프로세스에 넘겨 줄 필요는 없다.

CreateProcess 함수가 돌려주는 핸들은 프로세스 객체에 대해 PROCESS_ALL_ACCESS 접근 방식을 갖는다.

lpCurrentDirectory 인수가 지시하는 현재 디렉토리는 자식 프로세스의 현재 디렉토리가 된다. lpCommandLine 인수의 2 번째 항목이 지시하는 현재 디렉토리는 부모 프로세스의 현재 디렉토리이다.

부모 프로세스와 자식 프로세스

CreateProcess 함수로 프로세스를 생성하면 생성한 프로세스와 생성된 프로세스 간에는 부모-자식 관계가 생긴다. 하지만, 이들 프로세스가 서로 독립적으로 동작하도록 하고 싶다면, 앞에서도 잠시 언급한 것 처럼, 생성된 프로세스의 핸들을 곧바로 닫아주면 된다.

```
PROCESSS_INFORMATION ProcessInfomation;
BOOL fSuccess = CreateProcess(...., &ProcessInformation);
if(fSuccess)
{
    CloseHandle(ProcessInformation.hThread);
    CloseHandle(ProcessInformation.hProcess);
}
```

Win32 의 DLL 은 자신을 사용하는 프로세스의 개수를 참조횟수(reference count)라고 해서 기억해 두는데, 이 참조횟수가 0 이 되면 DLL 은 메모리에서 제거된다. CreateProcess 함수로 프로세스를 만들면 새로 만들어진 프로세스와 쓰레드의 참조횟수는 1 이 된다. 그 다음, 이 프로세스와 쓰레드는 부모 프로세스에 의해 참조되기 때문에 참조횟수는 2 가 된다. 따라서 자식 프로세스가 종료되더라도 참조횟수는 하나가 준 1 이 되기 때문에 메모리에서

제거되지 않는다. 따라서 부모 프로세스에서 CloseHandle 을 이용해서 참조횟수를 하나 더 줄여 0 으로 만들어야 메모리에서 제거된다. 자식 프로세스를 생성한 다음 CloseHandle 을 해 주지 않으면 메모리에서 완전히 제거되지 않기 때문에 문제가 생길 수 있으므로 주의해야 한다.

부모 프로세스가 자식 프로세스의 실행이 모두 끝난 다음에 다음 일을 수행하도록 하고 싶다면 다음과 같이 하면 된다.

```
PROCESS_INFORMATION ProcessInformation;
DWORD dwExitCode;
BOOL fSuccess, fExit;
fSuccess = CreateProcess(....,&ProcessInformation);

if(fSuccess)
{
    HANDLE hProcess = ProcessInformation.hProcess;
    if(WaitForSingleObject(hProcess, INFINITE) != 0xFFFFFFFF)
    {
        fExit = GetExitCodeProcess(hProcess, &dwExitCode);
    }
    CloseHandle(ProcessInformation.hThread);
    CloseHandle(hProcess);
}
```

대기함수

WaitForSingleObject 함수는 지정한 객체가 신호상태가 되거나 시간초과가 일어나면 반환되는데, 프로세스나 쓰레드의 경우에 그 프로세스나 쓰레드가 종료하기 전까지는 반환되지 않고 시간초과가 일어날 때까지 기다리게 된다.

```
```

```
DWORD WaitForSingleObject(
  HANDLE  hHandle,    // handle of object to wait for
  DWORD  dwMilliseconds // time-out interval in milliseconds
 );
```

첫 번째 인수인 hHandle 은 프로세스나 쓰레드 같은 객체의 핸들 값인데, WaitForSingleObject 함수는 이 핸들이 가리키는 객체가 종료될 때까지, 즉 신호(signaled) 상태가 될 때까지 기다리게 된다. 두 번째 인수인 dwMilliseconds 는 프로세스나 쓰레드가 종료될 때까지 얼마동안 기다릴 것인가를 밀리 초 단위로 지정해 준다. dwMilliseconds 가 INFINITE 이면 시간 제한 없이 계속 기다리게 된다. 함수가 성공하면 반환값은 함수가 복귀하게 된 원인(즉, 신호상태가 된 또는 종료된)이 된 사건(event)을 가리키게 된다. 실패하면 WAIT_FAILED 를 반환하며, 자세한 정보를 알고 싶다면 GetLastError 함수를 이용하면 된다. 만약 성공했을 때에는 GetLastError 함수는 WAIT_OBJECT_0 와 WAIT_TIMEOUT 등을 반환하게 된다. WAIT_OBJECT_0 는 지정한 객체의 상태가 신호상태가 되었음을 나타내고, WAIT_TIMEOUT 은 시간초과가 발생했으며 객체 상태는 여전히 비신호상태임을 나타낸다. 이런 종류의 함수를 대기함수라 부른다.

프로세스 종료

프로세스는 프로세스 내의 모든 쓰레드(물론 주 쓰레드도 포함)가 정상적으로 종료되면 비로소 종료하게 된다. 하지만 ExitProcess 와 TerminateProcess 를 이용하면 강제로 프로세스를 종료시킬 수 있다.

```
VOID ExitProcess(
  UINT  uExitCode // exit code for all threads
 );

BOOL TerminateThread(
  HANDLE  hThread,    // handle to the thread
  DWORD  dwExitCode      // exit code for the thread
 );
```

ExitProcess 는 프로세스가 프로세스 자신을 종료시키기 위해 사용된다. uExitCode 는 GetExitCodeProcess 로 구하는 값이 된다. ExitProcess 로 프로세스 자신을 종료시키면 그 프로세스가 사용했던 객체들과 DLL 들이 모두 안전하게 메모리에서 내려오게 된다.

TerminateProcess 는 다른 프로세스를 종료시키는데 사용되는데, 종료 대상이 되는 프로세스가 사용하던 객체들과 DLL 들에 어떤 조치도 취하지 않은 상태에서 그냥 종료시켜 버리기 때문에 매우 위험하다. 따라서 매우 제한된 범위 내에서만 사용해야 한다.

쓰레드 생성

한 프로세스의 주소공간 내에서 수행되는 쓰레드를 만든다.

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
        // pointer to thread security attributes
  DWORD  dwStackSize,     // initial thread stack size, in bytes
  LPTHREAD_START_ROUTINE  lpStartAddress,   // pointer to thread function
  LPVOID  lpParameter,      // argument for new thread
  DWORD  dwCreationFlags,     // creation flags
  LPDWORD  lpThreadId     // pointer to returned thread identifier
 );
```

[인수]

lpThreadAttributes

보안 속성을 나타내는 SECURITY_ATTRIBUTE 구조체를 가리킨다. 이 값이 NULL 이면 기본 보안속성을 갖는다.

```
typedef struct _SECURITY_ATTRIBUTES { // sa
```

```
    DWORD  nLength;
    LPVOID lpSecurityDescriptor;
    BOOL   bInheritHandle;
} SECURITY_ATTRIBUTES;
```

dwStackSize

새 쓰레드의 스택 크기를 바이트 단위로 지정한다. 0 이면 스택 크기는 프로세스의 주 쓰레드의 스택 크기와 같은 값을 갖는다. 스택은 프로세스의 메모리 공간에 자동적으로 할당되고, 쓰레드가 끝날 때 자동적으로 해제된다.

lpStartAddress

쓰레드에 의해 수행될 함수의 포인터. 쓰레드의 시작 주소. 함수는 하나의 32 비트 인수를 갖는다.

lpParameter

쓰레드에 넘겨 줄 하나의 32 비트 변수에 대한 포인터

dwCreateFlag

CREATE_SUSPENDED 플래그를 지정하면 쓰레드는 일시정지 상태로 생성되어 ResumeThread 를 호출하기 전까지는 수행되지 않는다. 이 값이 0 이면 생성 즉시 수행된다.

lpThreadId

쓰레드의 식별자를 받는 32 비트 변수에 대한 포인터

[반환값]

성공하면 새로 만들어진 쓰레드의 핸들을 돌려준다. 실패하면 NULL 을 돌려 주며 자세한 정보를 알고 싶다면 GetLastError 을 호출한다.

[설명]

쓰레드 수행은 lpStartAddress 로 지정한 함수에서 시작된다. lpStartAddress 에 지정된 함수가 종료하게 되면 그 DWORD 반환값이 바로 쓰레드의 탈출코드(exit code)가 된다. 쓰레드 탈출코드는 쓰레드 종료 함수인 ExitThread 의 인수로 지정한 값이다. 이 쓰레드의 반환값을 알아내려면 GetExitCodeThread 를 이용한다.

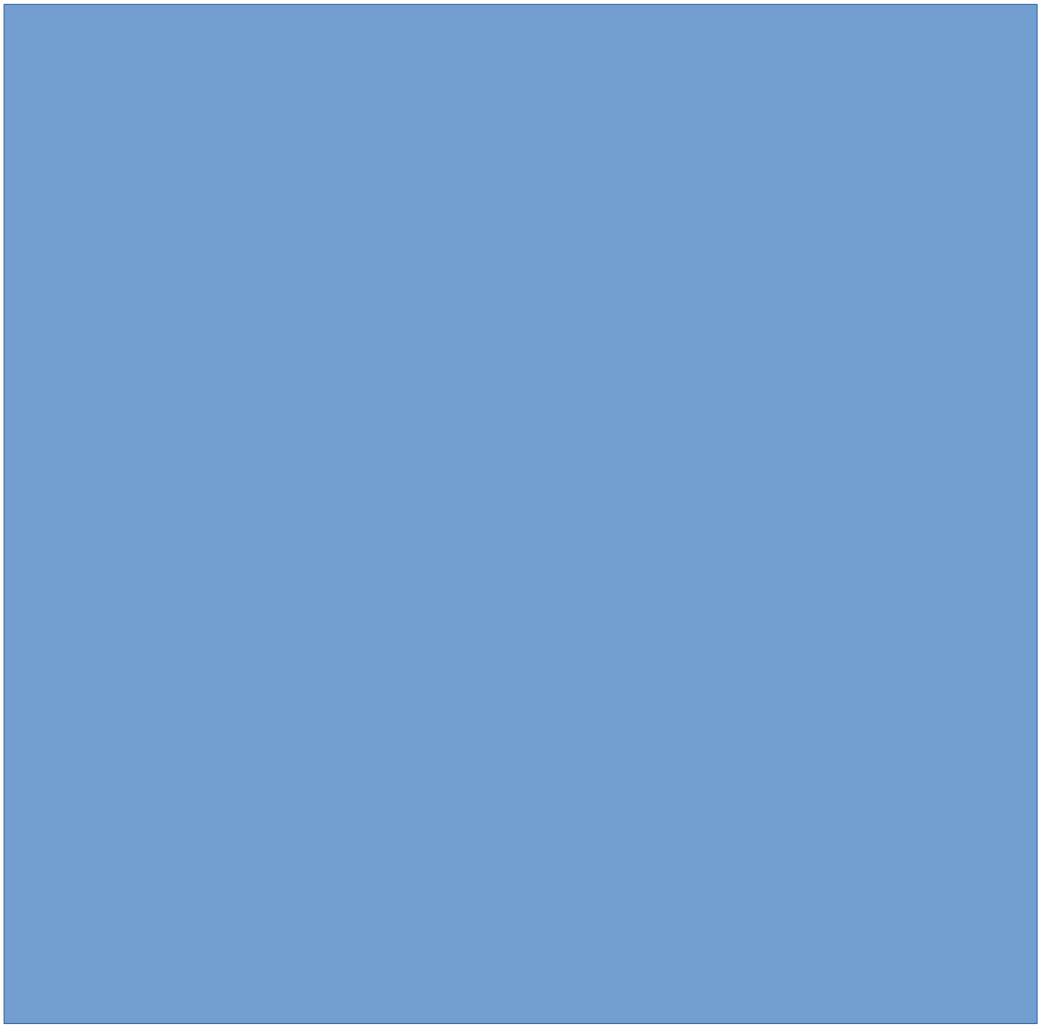
```
BOOL GetExitCodeThread(
   HANDLE  hThread,    // handle to the thread
   LPDWORD  lpExitCode     // address to receive termination status
  );
```

쓰레드는 우선순위 THREAD_PRIORITY_NORMAL 로 생성된다. 쓰레드의 우선순위를 얻으려면 GetThreadPriority 를 사용하고 우선순위값을 지정하려면 SetThreadPriority 를 지정한다.

```
int GetThreadPriority(
     HANDLE  hThread    // handle to thread
  );

BOOL SetThreadPriority(
     HANDLE  hThread,  // handle to the thread
   int  nPriority // thread priority level
  );
```

쓰레드의 우선순위 값에는 다음과 같은 것들이 있는데, 쓰레드의 우선순위는 프로세스의 우선순위에 대한 상대적인 값이다.



쓰레드 종료

쓰레드의 종료도 프로세스와 비슷하게 ExitThread 와 TerminateThread 두가지가 사용된다.

```
VOID ExitThread(
    DWORD  dwExitCode      // exit code for this thread
  );

BOOL TerminateThread(
```

```
  HANDLE  hThread,    // handle to the thread
  DWORD  dwExitCode       // exit code for the thread
);
```

프로세스의 종료와 마찬가지로, 쓰레드의 종료에서도 TerminateThread 는 다른 쓰레드를 종료시키는 함수로서, 종료되는 쓰레드가 사용하던 DLL 의 참조횟수따위는 모두 그대로 두기 때문에 아주 제한된 범위내에서만 사용하는 것이 좋다. 물론 ExitThread 함수는 쓰레드 자신을 종료시키는 함수이다.

프로세스간 통신

동기화

동기화에 대하여

대기함수, 동기화 객체, 중첩 입출력같은 동기화 기구(mechanism)에 대한 개략적인 설명을 해 보겠다. 임계 부분(critical section)과 연결된 변수 접근(interlocked variable access)도 동기화 기구에 포함된다.

대기 함수

쓰레드(thread)는 그 실행을 막기 위해서는 대기함수를 이용한다. 예를 들어 어떤 함수는 대기 함수의 인수(parameter)에 의해 지정된 조건 집합이 만족할 때까지 복귀하지 않는다. 시간 초과 간격(time-out interval)과 하나 이상의 동기화 객체가 대기 함수의 인수로 사용된다.

대기 함수가 호출되면 대기 함수는 그 자신을 마치기 위해 지정된 동기화 객체의 상태와 그

밖에 대기함수를 마칠 수 있는 다른 조건들을 검사한다. 초기 상태가 그 지정된 조건에 만족하지 않고 시간도 시간 초과 간격을 경과하지 않았다면 대기함수를 호출한 쓰레드는 일명 "효과적인 대기 상태"(efficient wait state)에 들어간다. 효과적인 대기상태란 조건이 만족하기 기다리는데, 아주 작은 프로세스 시간만을 소비하는 상태를 말한다. 시간 초과 간격을 정하기 어렵다면 그 값은 그냥 INFINITE 를 사용할 수도 있다.

WaitForSingleObject, WaitForSingleObjectEx 함수가 바로 우리가 사용할 수 있는 대기함수이다. 이들 대기함수는 하나의 동기화 객체에 대한 핸들을 필요로 한다. 이 함수들은 동기화 객체가 신호상태(signaled)이거나 시간 초과 간격이 경과했을 때 복귀한다. (동기화 객체의 상태는 신호상태(signaled)와 비신호상태(non-signaled) 두가지 중 하나이다)

```
DWORD WaitForSingleObject(
  HANDLE hHandle,     // handle of object to wait for
  DWORD dwMilliseconds   // time-out interval in milliseconds
 );
```

```
DWORD WaitForSingleObjectEx(
  HANDLE hHandle,     // handle of object to wait for
  DWORD dwMilliseconds,  // time-out interval in milliseconds
  BOOL bAlertable       // return to execute I/O completion routine if TRUE
 );
```

WaitForMultipleObjects, WaitForMultipleObjectsEx, MsgWaitForMultipleObjects 함수는 이 함수를 호출하는 쓰레드가 하나 이상의 동기화 객체를 지정할 수 있도록 해 준다. 하나 이상의 동기화 객체에 대한 배열이 이들 함수의 인수로서 사용된다. 이 함수들은 여러 동기화 객체 중에 어느 하나만 신호상태가 되면 바로 그 신호상태가 된 동기화 객체의 배열 인덱스를 복귀값(return value)으로 해서 마칠 수도 있으며, 반대로 모든 동기화 객체가 한꺼번에 신호상태가 되어야 지정된 조건이 만족되었다는 복귀값으로 마칠 수도 있다.

```
DWORD WaitForMultipleObjects(
  DWORD nCount, // number of handles in handle array
  CONST HANDLE *lpHandles,  // address of object-handle array
  BOOL bWaitAll,   // wait flag
  DWORD dwMilliseconds   // time-out interval in milliseconds
 );
```

```
DWORD WaitForMultipleObjectsEx(
  DWORD nCount, // number of handles in handle array
  CONST HANDLE *lpHandles,  // address of object-handle array
  BOOL bWaitAll,   // wait flag
  DWORD dwMilliseconds,  // time-out interval in milliseconds
  BOOL bAlertable       // alertable wait flag
 );
```

WaitForSingleObjectEx, WaitForMultipleObjectsEx 는 경계대기(alertable wait)가 가능하다는 점에서 다른 대기함수들과 다르다. 경계대기란 어떤 조건이 만족했을 때 바로 복귀할 수가 있다는 뜻이다.

MsgWaitForMultipleObjects 함수는 지정된 입력의 종류가 이 함수를 호출한 쓰레드의 입력 큐에서 사용가능할 때 복귀할 수 있다는 점을 빼고는 WaitForMultipleObjects 함수와 같다. 어떤 쓰레드가 지정한 객체의 상태가 신호상태가 되거나 마우스 입력이 그 쓰레드의 입력 큐에 들어와 사용 가능한 상태가 될 때 까지 그 실행을 중단하고 싶다면 바로 이 WaitForMultipleObjects 함수를 사용하면 된다. 이렇게 되면 쓰레드는 GetMessage 나 PeekMessage 함수로 그 입력을 읽어올 수 있다.

```
DWORD MsgWaitForMultipleObjects(
  DWORD nCount, // number of handles in handle array
  LPHANDLE pHandles,      // address of object-handle array
  BOOL fWaitAll,    // wait for all or wait for one
  DWORD dwMilliseconds,  // time-out interval in milliseconds
  DWORD dwWakeMask     // type of input events to wait for
 );
```

대기함수는 복귀하기 전에 동기화 객체의 몇가지 상태를 수정하는데, 이런 수정은 오직 객체가 신호상태가 되어서 함수가 복귀했을 때만 일어난다. 대기 함수는 다음과 같이 동기화 객체의 상태를 수정할 수 있다.

(가) 신호기(semaphore) 객체의 수를 하나 감소시킨다. 신호기의 수가 0 이 되면, 신호기의 상태는 비신호상태(nonsignaled)로 된다.

(나) 뮤텍스(mutex)의 상태, 자동-리셋(auto-reset) 사건, 변화 통지(notification) 객체

가 비신호상태로 변한다.

(다) 수동-리셋(maual-reset) 사건, 프로세스, 쓰레드와 콘솔 입력 객체는 대기함수에 영향을 받지 않는다.

동기화 객체

동기화 객체는 여러 쓰레드의 수행을 통합 조정하기 위해, 대기 함수들에 그 핸들을 사용한다. 동기화 객체의 상태는 대기 함수에서 복귀하는 것을 허용하는 신호상태(signaled)와 복귀하는 것을 허용하지 않는 비신호상태(nonsignaled)중 하나이다. 하나 이상의 프로세스가 같은 동기화 객체의 핸들을 가질 수 있는데 이는 프로세스간 동기화(interprocess synchronization)를 가능하게 한다.

뮤텍스 객체(Mutex Object)

뮤텍스 객체는 어떤 쓰레드에 소유되지 않으면 신호상태(signaled state)로 되고 소유되면 비신호상태(nonsignaled state)로 되는 동기화 객체이다. 단지 한번에 한 쓰레드만이 뮤텍스를 소유할 수 있다. 뮤텍스란 이름은 공유 자원에 대한 상호 배타적(MUTually EXclusive) 접근이란 말에서 나온 용어이다. 예를 들어, 두 개의 쓰레드가서로 공유된 메모

리에 동시에 쓰기 작업을 수행하려는 걸 막기 위해 각 쓰레드는 쓰기 작업을 수행하기 전에 뮤텍스의 소유권을 얻기를 기다리도록 해야 한다. 공유 메모리에 쓰기를 마친 후에는 그 뮤텍스의 소유권을 풀어 준다.

뮤텍스 객체를 만들려면 CreateMutex 함수를 사용한다. 뮤텍스 객체를 를 만든 쓰레드는 그 뮤텍스 객체의 소유권을 즉시 요구할 수도 있고 또한 뮤텍스 객체의 이름을 지정할 수도 있다.

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,  // pointer to security attributes
  BOOL bInitialOwner, // flag for initial ownership
  LPCTSTR lpName      // pointer to mutex-object name
 );
```

다른 프로세스에 있는 쓰레드들도 OpenMutex 함수를 이용하면 뮤텍스 객체의 이름으로 이미 만들어져 있는 뮤텍스 객체의 핸들을 얻을 수 있다.

```
HANDLE OpenMutex(
  DWORD dwDesiredAccess,    // access flag
  BOOL bInheritHandle,      // inherit flag
  LPCTSTR lpName      // pointer to mutex-object name
 );
```

뮤텍스 객체의 핸들을 가진 쓰레드는 그 뮤텍스 객체의 소유권을 요구하기 위해 대기함수(wait-function) 중 하나를 사용할 수 있다. 만약 뮤텍스 객체를 다른 쓰레드가 소유하고 있다면 대기 함수는 그 뮤텍스 객체 소유권를 요구한 쓰레드의 실행을 뮤텍스 객체의 소유권을 가진 쓰레드가 그 소유권을 놓을 때까지 일시 중지시킨다. 소유권을 놓을 때는 ReleaseMutex를 사용한다.

```
BOOL ReleaseMutex(
  HANDLE hMutex // handle of mutex object
 );
```

뮤텍스 객체의 소유권을 놓지 않은 상태로 그 소유권을 가진 쓰레드가 종료되면 그 뮤텍스

객체는 버려진 것으로 생각된다. 대기하고 있던 쓰레드는 그 버려진 뮤텍스 객체의 소유권을 가질 수 있다.

세마포어 객체(Semaphore Object)

세마포어 객체는 0 에서 지정된 최대 값 사이의 총수(count)를 유지하기 위한 동기화 객체이다 세마포어의 상태는 그 총수(count)가 0 보다 커지면 신호상태(signaled state)가 되고, 그 총수가 0 이면 비신호상태(nonsignaled state)가 된다.

세마포어 객체는 어떤 공유 자원을 제한된 수만을 지원하도록 제어하는데 유용하다. 예를 들어 응용프로그램이 제한된 개수의 창을 만들고 싶다면 그 최대 창의 개수를 세마포어 객체의 총수로 한다. 창이 하나 생길 때마다 그 총수는 하나씩 줄어들고, 0 이되면 대기 함수는 창 생성 코드의 실행을 중지시킨다.

CreateSemaphore 함수로 세마포어 객체를 만드는데 이 때 초기 총수값과 최대 총수값을 지정한다. 초기값은 0 보다 작아서도 최대값보다 커서도 안된다. 물론 세마포어 객체의 이름도 지정하게 되는데 다른 프로세스에 있는 쓰레드들은 이 이름을 통해 이미 만들어진 세마포어 객체를 열 수 있다. 이 때는 OpenSemaphore 함수를 이용한다.

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, //  pointer  to  security attributes
  LONG lInitialCount,  // initial count
  LONG lMaximumCount,  // maximum count
  LPCTSTR lpName      // pointer to semaphore-object name
 );
```

```
HANDLE OpenSemaphore(
  DWORD dwDesiredAccess,    // access flag
  BOOL bInheritHandle,     // inherit flag
  LPCTSTR lpName      // pointer to semaphore-object name
 );
```

세마포어 객체가 신호상태가 되어 대기함수가 복귀할 때마다 세마포어의 총수가 하나씩 감소한다. ReleaseSemaphore 함수는 지정한 수만큼 세마포어의 총수를 증가시킨다.

```
BOOL ReleaseSemaphore(
  HANDLE hSemaphore,     // handle of the semaphore object
  LONG lReleaseCount,    // amount to add to current count
  LPLONG lpPreviousCount       // address of previous count
 );
```

세마포어의 초기 총수는 보통 최대값으로 설정된다. 그 총수는 세마포어 객체가 보호하는 자원이 소비되는 수준에 따라 감소하게 된다. 세마포어가 그 초기 총수를 0 으로 하면 응용프로그램이 초기화 될 때 보호 자원에 접근하지 못하도록 막는 역할을 한다. 초기화 이후에 ReleaseSemaphore 함수로 그 총수를 최대값으로 증가시킬 수도 있다. 쓰레드는 뮤텍스 객체와는 달리 세마포어 객체의 소유권을 얻을 필요가 없다. 뮤텍스 객체를 소유한 쓰레드는 실행 중단 없이 같은 뮤텍스 객체를 반복해서 기다릴 수 있다. 하지만 세마포어의 경우에는 약간 다르다. 같은 세마포어 객체에 대해 반복해서 기다리는 쓰레드는 그 세마포어의 총수(count)가 대기함수가 만족할 때마다 하나씩 감소한다. 그리고 이 총수가 0 이 되면 그 실행은 중지된다.

쓰레드는 대기함수 호출에서 에서 같은 세마포어 객체를 반복해서 지정함으로써 세마포어의 총수를 하나 이상 감소시킬 수 있지만 같은 세마포어 객체를 배열에 포함하고 있는 상태에서 WaitForMultipleObjects 함수를 사용해도 그 총수가 여러개 감소하는 것은 아니다.

사건 객체(Event Object)

가장 주의깊게 살펴보아야 할 부분이다. 사건 객체는 SetEvent 나 PulseEvent 함수를 이용해서 신호상태로 설정할 수 있는 동기화 객체이다. 사건 객체에는 수동-리셋 객체와 자동-리셋 객체 두가지 종류가 있다.

사건 객체는 특정한 사건이 발생하게 되는 쓰레드에서 신호를 보내는데 사용하면 유용하다. 가장 좋은 예가 중첩 입출력이다. 시스템은 중첩작업이 완료되었을 때 지정된 사건 객체를 신호상태로 만든다. 따라서 중첩작업이 완료되었다는 것은 대기함수를 이용해서 사건객체가 신호상태가 되는 것을 보고 알 수가 있다. 하나의 쓰레드는 여러개의 동시에 일어나는 중첩작업에 대해 서로 다른 사건 객체를 설정할 수 있으며, 이렇게 여러개의 사건객체에 대해서 어느 하나라도 신호상태가 되기를 기다리려면 WaitForMultipleObject 함수를 이용하면 된다.

쓰레드에서 사건객체를 만들려면 CreateEvent 함수를 이용한다. 사건객체를 만드는 쓰레드는 수동리셋 사건이건 자동리셋 사건이건 간에 사건객체의 초기 상태를 설정해 주고, 그 사건 객체에 대한 이름도 지정할 수 있다. 다른 프로세스에 있는 쓰레드들은 이 객체이름으로 사건객체의 핸들을 열 수 있다. 이 때는 OpenEvent 함수를 이용한다.

PulseEvent 함수는 사건객체의 상태를 신호상태로 설정한 다음, 적당한 수의 대기 쓰레드를 풀고 난 다음에 비신호상태로 리셋 시키는데 사용한다. 수동 리셋 사건 객체에 대해서는 풀릴 수 있는 모든 대기 쓰레드는 즉시, 대기에서 풀린다(release). 자동 리셋 사건에 대해서는 여러개의 대기 쓰레드가 있더라도 하나의 대기 쓰레드만이 풀린다. 대기 중인 쓰레드가 없고 즉시 풀릴 수 있는 쓰레드가 없다면 PulseEvent 함수는 단지 사건객체를 비신호 상태로만 설정하게 된다.

프로세스간 동기화

여러개의 프로세스들은 똑같은 뮤텍스, 세마포어 또는 사건 객체를 가질 수 있다. 따라서 이들 객체는 프로세스간 동기화에 사용될 수 있는데, 객체를 만든 프로세스들은 그 객체 생성함수(CreateMutex, CreateSemaphore, CreateEvent)들이 돌려주는 핸들을 이용하게 된

다. 다른 프로세스들은 객체의 이름으로 그 객체에 대한 핸들을 열 수 있다. 또는 상속이나 복제를 통해서도 그 객체의 핸들을 열 수 있다.

객체 이름

객체 이름은 객체 핸들을 공유하기 위한 가장 쉬운 길이다. 객체를 생성한 프로세스가 지정한 이름은 MAX_PATH 개 문자로 제한되며, 객체 이름에는 역슬래시(\)를 제외한 어떠한 문자도 올 수 있다. 어떤 한 프로세스에서 뮤텍스, 세마포어 또는 객체를 만들었다면 다른 프로세스에서는 OpenMutex, OpenSemaphore, OpenEvent 함수를 이용해서 그 객체에 대한 핸들을 열 수 있다. 이 이름은 대소문자 구별이 있다. 사건, 세마포어, 뮤텍스, 파일-매핑 객체의 이름은 같은 이름 공간을 공유하기 때문에 다른 형식의 객체로서 같은 이름을 가지면 오류가 발생한다. 따라서 객체를 생성할 때는 중복되는 이름이 없는지 확인한 후에 유일무이한 이름을 사용해야 한다.

만약에 이미 존재하는 객체의 이름으로 Create...했다면 이는 사실 Open...한 것과 똑같다.

객체 상속

CreateProcess 함수로 만든 자식 프로세스는 뮤텍스, 세마포어 또는사건의 핸들을 상속받을 수 있다. 물론 SECURITY_ATTRIBUTE 구조체에 객체 핸들들에 대해 상속 가능을 설정해 놓은 경우에 대해서 그렇다. 자식 프로세스로 상속된 핸들은 부모 프로세스와 같은 접근권한을 갖는다. 상속된 핸들은 자식프로세스의 핸들로서 기능하게 되고 물론 부모 프로세스도 이 핸들을 사용할 수 있기 때문에, 이 핸들을 이용해서 부모 프로세스와 자식 프로세스간에 통신이 이루어 질 수 있게 된다. 부모 프로세스는 CreateProcess 함수를 이용할 때 명령행 인수로 값들을 정해줄 수 있는데, 자식 프로세스는 GetCommandLine 함수로 명령행 인수를 읽어, 사용할 수 있는 핸들로 변환하게 된다.

```
LPTSTR GetCommandLine(VOID)
```

객체 복사

DuplicateHandle 은 다른 프로세스에서 객체에 대한 핸들을 사용할 수 있도록 복사하는 함수이다. 이 방법은 위의 두 방법(객체 이름에 의한 방법, 상속에 의한 방법)에 비해 사용법이 좀 복잡하다. 객체를 만든 프로세스와 복사된 핸들이 들어가게 될 프로세스 사이에 통신이 필요하다. 핸들값과 프로세스 식별자같은 필요한 정보를 파이프나 공유파일, 공유메모리 등의 프로세스간 통신방법을 통해 넘겨주거나 받아야 한다.

동기화

Win32 API 는 파일, 파이프, 직렬통신 장치등에 대해 동기(synchronus)입출력과 비동기(asynchronus) 입출력 모두 가능하다. WriteFile, ReadFile, DeviceIoControl, WaitCommEvent, ConnectNamedPipe, TransactNamePipe 함수는 동기, 비동기로 모두 동작하고 WriteFileEx, ReadFileEx 는 비동기로만 동작한다.

함수가 동기적으로 실행된다는 말은 함수가 하는 작업이 모두 끝나기 전에는 함수가 복귀하지 않는다는 뜻이다. 이것은 시간이 많이 걸리는 작업의 경우에 이 작업이 마치기 전까지는 이 작업을 호출한 쓰레드는 그 실행이 막혀 있게 된다는 것을 의미한다. 즉, 시간이 많이 걸리는 작업이 끝날 때까지 그 함수안에서 계속 대기상태에 있게 된다.

중첩입출력

중첩작업, 즉 비동기적으로 동작하는 함수는 실행 후 즉시 복귀한다. 비록 그 작업이 완료되지 않았더라도 말이다. 이것은 시간이 많이 걸리는 작업은 배경(background)작업으로 이루어질 수 있도록 만들고 이 함수를 호출한 프로세스가 다른 일을 할 수 있도록 해준다. 예를 들어, 하나의 쓰레드가 다른 핸들 상에서 동시에 입출력을 수행할 수 있거나 또는 같은 핸들 상에서 동시에 읽기 쓰기 작업이 수행 될 수 있는 경우를 생각해 보자. 이런 중첩작업을 동기화하기 위해서는 이런 작업을 호출한 쓰레드가 GetOverlappedResult 같은 대기 함수를 사용하거나 또는 중첩작업이 완료될 때를 결정할 수 있는 대기 함수를 사용하면 된다.

중첩작업이 이루어지려면 FILE_FLAG_OVERLAPPED 플래그가 설정된 상태로 파일이나 통신장치 핸들을 만들어야 한다. 중첩작업으로 함수가 동작하도록 하려면 OVERLAPPED 구조체에 대한 포인터를 넘겨주어야 한다. 만약 이 포인터가 NULL 이면 비록, 핸들을 만들 때 FILE_FLA_OVERLAPPED 로 해주었다하더라도 중첩작업으로 동작하지 않게된다. OVERLAPPED 구조체는 수동-리셋 사건 객체에 대한 핸들을 가져야 한다. 시스템은 입출력 함수가 호출되면 그 직후에 바로 복귀하기 위해 입출력작업을 완료되기 전에 사건객체를 비신호상태로 설정한다. 그 다음 시스템은 이 입출력 작업이 완료되면 사건 객체 핸들을 신호상태로 만든다.

중첩작업의 경우에는 함수가 복귀하기 전에 작업이 완료되어 버릴 수도 있는데, 이런 때 그 결과는 마치 그 동작자체가 동기적으로 이루어진 것처럼 다루어 진다. 반면 작업은 아직 완료되지 않고 함수는 FALSE 를 반환하면 GetLastError 함수는 ERROR_IO_PENDING 을 반환한다.

쓰레드는 다음의 두가지 방법으로 중첩작업을 관리할 수 있다. 하나는 GetOverlappedResult 함수를 이용해서 중첩작업이 완료되기까지 기다리는 것이다. 두 번째 방법은 OVERLAPPED 구조체의 수동-리셋 사건객체의 핸들을 대기함수들 중 하나에 지정한 다음, 대기함수가 끝나면 GetOverlappedResult 함수를 호출하는 방법이다. GetOverlappedResult 는 완료된 중첩작업의 결과를 반환하며 결과가 성공적이면 실제로 전송된 바이트 수를 돌려준다.

그렇다면 중첩작업이 동시에 여러 가지가 일어나는 경우는 어떤가? 이 경우 역시 기본적으로 하나의 중첩작업고 마찬가지이다. 단지 사건객체가 신호상태가 되기를 기다리는 대기함수가 약간 다를 뿐이다. 각 중첩작업에 대해 수동-리셋 사건객체를 OVERLAPPED 구조체에 지정해 준다. 쓰레드는 사건객체의 핸들에 대한 배열을 인자로 갖는 WaitForMutipleObject 함수를 이용해서 대기하고자 하는 사건객체들을 지정해 줄 수 있다. 이 함수의 반환값은 어느 사건 객체가 신호상태가 되었는지를 나타낸다. 따라서 쓰레드는 어느 작업이 완료되었는지를 판단할 수 있다.

임계부분 객체(Critical Section Objects)

임계부분 객체는 하나의 프로세스내의 쓰레드들에서만 동기화 객체로 사용된다는 점만 빼고는 뮤텍스 객체와 같다. 사건 객체, 뮤텍스 객체, 세마포어 객체 모두 단일 프로세스에도 사용할 수 있긴 하지만, 임계부분 객체가 속도면에서 약간 더 빠르며 뮤텍스 동기화보다 좀 더

효율적인 구조로 되어 있다. 뮤텍스 객체처럼 임계부분 객체도 한번에 하나의 쓰레드에 의해서만 소유될 수 있다. 따라서 임계부분 객체는 공유하는 자원에 한 프로세스 내에서 속한 여러개의 프로세스가 동시에 접근하는 것을 막는데 사용된다. 예를 하나 든다면 하나의 프로세스 내에 있는 전역 자료 구조를 여러개의 쓰레드에서 동시에 수정하려고 하는 것을 막기 위해서 사용하면 좋다.

임계부분 객체가 사용하는 메모리는 프로세스가 관리하는데, 보통은 변수를 CRITICAL_SECTION 형으로 선언하면 간단히 된다. 프로세스의 한 쓰레드에서 이 변수를 사용하기 전에는 먼저 InitializeCriticalSection 함수를 불러 임계부분 객체를 초기화 해주어야 한다.

쓰레드는 EnterCriticalSection 함수를 불러 임계부분 객체의 소유권을 요구하고 LeaveCriticalSection 함수를 통해 소유권을 해제한다.

```
VOID InitializeCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection // address of critical section object
  );

VOID EnterCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection    // pointer to critical section object
  );

VOID LeaveCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection    // address of critical section object
  );
```

EnterCriticalSection 은 소유권을 얻을 때까지 무작정 기다리는데 이것은 뮤텍스 객체와 다른 점이다. 뮤텍스 객체는 시간초과값을 지정할 수 있는 대기함수를 사용할 수 있다.

그렇다면, EnterCriticalSection 한 객체에 대해서 다시 또 EnterCriticalSection 할 수 있을까? 정답은 "그렇다"이다. 만약 EnterCriticalSection 한 객체에 대해서 다시 또 EnterCriticalSection 할 수 없다면 교착상태에 빠지게 된다. 그 이유는 다음과 같다. 만약 EnterCriticalSection 해서 임계부분 객체에 대한 소유권을 갖게 된 쓰레드가 다시 바로 그 임계부분 객체에 대한 소유권을 얻기 위해 EnterCriticalSection 을 하게 되면, 두 번째 EnterCriticalSection 은 그 소유권을 얻기 위해 무작정 기다리게 되고, 첫 번째 EnterCriticalSection 으로 받게된 임계부분 객체의 소유권은 반환되지 않고 있기 때문에 일

종의 교착상태에 빠지게 되는 것이다. 소유권을 내놓으려면 EnterCriticalSection 한 횟수만큼 LeaveCriticalSection 해주어야 한다.

프로세스의 어떤 쓰레드라도 DeleteCriticalSection 함수를 이용해서 임계부분 객체가 초기화될 때 사용된 시스템 자원을 반환하도록 할 수 있는데, 이 함수가 호출되고 나면 더 이상 임계부분 객체는 사용될 수 없다.

```
VOID DeleteCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection     // pointer to critical section object
  );
```

어떤 한 쓰레드가 임계부분 객체를 소유하게 되면(EnterCriticalSection 함수를 사용) EnterCriticalSection 함수를 호출해서 그 소유권을 얻기를 기다리는 쓰레드들은 대기상태에 있게되지만 그 이외의 쓰레드들은 계속 다른 일을 수해하게 된다.

공유변수에 안전하게 접근(Interlocked Variable Access)

같은 프로세스 또는 다른 프로세스 내의 여러 쓰레드들이 공유하는 32 비트 변수에 대한 접근을 동기화시키려면 InterlockedIncrement 와 InterlockedDecrement 함수를 사용한다. 이 두 개의 함수는 변수의 증가 또는 감소 그리고 그 결과값을 검사하게 된다. 다중작업 환경에서는 하나의 쓰레드가 실행 중일 때, 다른 쓰레드가 수행될 수도 있다. "가"라는 쓰레드에서 다른 쓰레드와 공유하는 32 비트 변수의 값을 증가시켰다고 하자. 증가시킨 다음, 증가된 값을 이용하기 전에 "나"라는 쓰레드에서 다시 이 변수의 값을 증가시키고, 그 직후에 다시 "가"라는 쓰레드로 실행이 넘어왔다고 해보자. 이렇게 되면 "가"라는 쓰레드에서는 분명 값이 한번 증가시켰지만, 그 결과값을 읽어보면 값이 두 번 증가한 것으로 된다. 이런 종류의 오류를 방지하기 위해 사용한다.

```
LONG InterlockedIncrement(
    LPLONG  lpAddend    // address of the variable to increment
```

```
  );

LONG InterlockedDecrement(
   LPLONG  lpAddend    // address of the variable to decrement
  );
```

interlock : 현재 처리 중인 동작이 끝날 때까지 다른 동작이 개시되지 않도록 하는 일

동기화 사용

이름붙인 객체 사용

다음은 두 개의 프로세스가 객체의 이름을 이용해서 같은 뮤텍스 객체의 핸들을 여는 예제이다.

첫 번째 프로세스는 CreateMutex 함수를 이용해서 뮤텍스 객체를 만든다. 같은 이름의 뮤텍스 객체가 있더라도 CreateMutex 함수는 성공한다.

```
/* One process creates the mutex object. */

HANDLE hMutex;
DWORD dwErr;

hMutex = CreateMutex(
   NULL,                /* no security descriptor */
   FALSE,               /* mutex not owned        */
   "NameOfMutexObject");  /* object name            */
if (hMutex == NULL)
   printf("CreateMutex error: %d\n", GetLastError() );
else
   if ( GetLastError() == ERROR_ALREADY_EXISTS )
      printf("CreateMutex opened existing mutex\n");
   else
      printf("CreateMutex created new mutex\n");
```

두 번째 프로세스는 OpenMutex 함수를 이용해서 첫 번째 프로세스가 만든 뮤텍스 객체의 핸들을 연다. 이 함수는 지정한 이름의 뮤텍스 객체가 존재하지 않으면 실패한다.

```
/* Another process opens a handle of the existing mutex. */

HANDLE hMutex;

hMutex = OpenMutex(
    MUTEX_ALL_ACCESS,      /* request full access    */
    FALSE,                 /* handle not inheritable */
    "NameOfMutexObject");  /* object name            */
if (hMutex == NULL)
    printf("OpenMutex error: %d\n", GetLastError() );
```

다음은 이름 붙인 객체를 이용하는 CreateSemaphore 함수의 사용 예제이다.

```
HANDLE CreateNewSemaphore(LPSECURITY_ATTRIBUTES lpsa,
    LONG cInitial, LONG cMax, LPTSTR lpszName) {
HANDLE hSem;

/* Create or open a named semaphore. */

hSem = CreateSemaphore(
    lpsa,         /* security attributes */
    cInitial,     /* initial count       */
    cMax,         /* maximum count       */
    lpszName);    /* semaphore name      */

/* Close handle, and return NULL if existing semaphore opened. */

if (hSem != NULL &&
        GetLastError() == ERROR_ALREADY_EXISTS) {
    CloseHandle(hSem);
    return NULL;
}
```

```
/* If new semaphore was created, return the handle. */

return hSem;

}
```

다중 객체에 대한 대기

다음은 CreateEvent 함수로 두 개의 객체를 만들어서 WaitForMultipleObjects 함수로 그 두 객체 중 하나가 신호상태가 되기를 기다리는 예이다.

```
HANDLE hEvents[2];
DWORD i, dwEvent;

/* Create two event objects. */

for (i = 0; i < 2; i++) {
   hEvents[i] = CreateEvent(
      NULL,   /* no security attributes        */
      FALSE,  /* auto-reset event object       */
      FALSE,  /* initial state is nonsignaled  */
      NULL);  /* unnamed object                */
   if (hEvents[i] == NULL) {
      printf("CreateEvent error: %d\n", GetLastError() );
      ExitProcess(0);
   }
}

/*
 * The creating thread waits for other threads or processes
 * to signal the event objects.
 */

dwEvent = WaitForMultipleObjects(
   2,          /* number of objects in array */
   hEvents,     /* array of objects          */
   FALSE,       /* wait for any              */
   INFINITE);   /* indefinite wait           */

/* Return value indicates which event is signaled. */
```

```
switch (dwEvent) {
   /* hEvent[0] was signaled. */
   case WAIT_OBJECT_0 + 0:
      /* Perform tasks required by this event. */
      break;
   /* hEvent[1] was signaled. */
   case WAIT_OBJECT_0 + 1:
      /* Perform tasks required by this event. */
      break;
   /* Return value is invalid. */
   default:
      printf("Wait error: %d\n", GetLastError());
      ExitProcess(0);
}
```

뮤텍스 객체 사용

뮤텍스 객체는 여러개의 프로세스나 쓰레드가 공유하는 자원에 동시에 접근하게 되는 것을 막기 위해 사용한다. 각 쓰레드는 공유하는 자원에 접근하는 코드를 수행하기 전에 뮤텍스 객체의 소유권을 받아와야 한다.

다음은 CreateMutex 함수로 뮤텍스 객체를 만들거나, 이미 존재하는 뮤텍스 객체를 여는 예이다.

```
HANDLE hMutex;
/* Create a mutex with no initial owner. */
```

```
hMutex = CreateMutex(
   NULL,                       /* no security attributes */
   FALSE,                      /* initially not owned      */
   "MutexToProtectDatabase");  /* name of mutex          */
if (hMutex == NULL) {
   /* check for error */
}
```

프로세스의 한 쓰레드가 데이터베이스에 자료를 써 넣는 예를 생각해 보자. 먼저 뮤텍스 객체의 소유권을 요구해서 얻은 다음, 데이터베이스에 자료를 써넣고, 그 다음에는 소유권을 해제한다.

다음 예는 쓰레드가 적절하게 뮤텍스 객체를 해제하도록 하기 위해 try-funally 구조의 예외처리를 사용했다. 코드에서 finally 블록은 try 블록이 끝날 때까지는 수행되지 않는다. 이렇게 함으로써 뮤텍스 객체가 안전하지 않은 상태에서 해제되는 것을 막을 수 있다.

```
BOOL FunctionToWriteToDatabase(HANDLE hMutex) {
DWORD dwWaitResult;

/* Request ownership of mutex. */

dwWaitResult = WaitForSingleObject(
      hMutex,   /* handle of mutex           */
      5000L);   /* five-second time-out interval */

switch (dwWaitResult) {

   /* The thread got mutex ownership. */

   case WAIT_OBJECT_0:

      try {

         .
         . /* Write to the database. */
         .

      }

      finally {

         /* Release ownership of the mutex object. */

         if (! ReleaseMutex(hMutex)) {
```

```
            /* Deal with error. */
        }

        break;
    }

  /* Cannot get mutex ownership due to time-out. */

  case WAIT_TIMEOUT:

    return FALSE;
```

세마포어 객체 사용

만들어 지는 창의 수를 제한하기 위해 세마포어 객체를 사용하는 예이다. 먼저, CreateSemaphore 함수로 세마포어를 만든다음, 그 초기값과 최대 수를 지정해 준다.

```
HANDLE hSemaphore;
LONG cMax = 10;
LONG cPreviousCount;

/* Create a semaphore with initial and max. counts of 10. */

hSemaphore = CreateSemaphore(
    NULL,   /* no security attributes */
    cMax,   /* initial count         */
    cMax,   /* maximum count         */
    NULL);  /* unnamed semaphore     */
if (hSemaphore == NULL) {
    /* Check for error. */
}
```

쓰레드들은 새로운 창을 만들기 전에 WaitForSingleObject 함수를 사용해서 세마포어의 현재 값이 새로운 창을 만들 수 있는 상태인지를 검사한다. 대기함수의 시간 초과 인수가 0 이면, 함수는 세마포어가 비신호상태라면 즉시 복귀한다.

```
DWORD dwWaitResult;

/* Try to enter the semaphore gate. */

dwWaitResult = WaitForSingleObject(
        hSemaphore,   /* handle of semaphore          */
        0L);          /* zero-second time-out interval */

switch (dwWaitResult) {

   /* The semaphore object was signaled. */

   case WAIT_OBJECT_0:
        .
        . /* OK to open another window. */
        .
        break;

   /* Semaphore was nonsignaled, so a time-out occurred. */

   case WAIT_TIMEOUT:
        .
        . /* Cannot open another window. */
        .
        break;

}
```

쓰레드가 창을 하나 닫을 때는 세마포어의 수를 하나 증가시키기 위해 ReleaseSemaphore 함수를 사용한다.

```
/* Increment the count of the semaphore. */

if (! ReleaseSemaphore(
        hSemaphore,  /* handle of semaphore             */
        1,           /* increase count by one           */
        NULL) ) {    /* not interested in previous count */

   /* Deal with the error. */

}
```

사건 객체 사용

다음 예는 주 쓰레드(master thread)가 공유 메모리 버퍼에 자료를 쓰고 있을 때, 다른 쓰레드들이 그 공유메모리 버퍼를 읽는 것을 막기 위해 사건 객체를 사용하는 예이다. 먼저, CreateEvent 함수로 수동 리셋 사건 객체를 하나 만든다. 주 쓰레드는 버퍼에 자료를 쓸 때 사건 객체를 비신호 상태로 만들고 자료 쓰기가 완료되면 그 사건 객체를 신호상태로 돌려 놓는다. 그 버퍼를 읽는 여러개의 쓰레드를 만드는데, 각 쓰레드에 대해 자동 리셋 객체를 만든다. 이렇게 만들어진 쓰레드들은 버퍼에서 자료를 읽지 않을 때 사건 객체를 신호상태로 설정한다.

```
#define NUMTHREADS 4

HANDLE hGlobalWriteEvent;

void CreateEventsAndThreads(void) {

HANDLE hReadEvents[NUMTHREADS], hThread;
DWORD i, IDThread;

/*
 * Create a manual-reset event object. The master thread sets
 * this to nonsignaled when it writes to the shared buffer.
 */

hGlobalWriteEvent = CreateEvent(
    NULL,        /* no security attributes    */
    TRUE,        /* manual-reset event        */
    TRUE,        /* initial state is signaled */
    "WriteEvent" /* object name               */
    );
if (hGlobalWriteEvent == NULL) {
    /* error exit */
}

/*
 * Create multiple threads and an auto-reset event object
 * for each thread. Each thread sets its event object to
 * signaled when it is not reading from the shared buffer.
 */

for(i = 1; i <= NUMTHREADS; i++) {

    /* Create the auto-reset event. */
```

```
    hReadEvents[i] = CreateEvent(
        NULL,     /* no security attributes    */
        FALSE,    /* auto-reset event          */
        TRUE,     /* initial state is signaled */
        NULL);    /* object not named          */
    if (hReadEvents[i] == NULL) {
        /* error exit */
    }

    hThread = CreateThread(NULL, 0,
            (LPTHREAD_START_ROUTINE) ThreadFunction,
            &hReadEvents[i], /* pass event handle  */
            0, &IDThread);
    if (hThread == NULL) {
        /* error exit */
    }
}

}
```

주 쓰레드가 공유 버퍼에 쓰기 전에 ResetEvent 함수를 사용해서 hGlobalWriteEvent(위의 예에서 사용되는 전역 변수)의 상태를 비신호 상태로 만든다. 이것은 버퍼를 읽는 쓰레드들이 읽기 작업을 시작하는 것을 막기 위해서이다. 주 쓰레드는 버퍼를 읽는 모든 쓰레드들의 현재 읽기 작업이 끝나기를 기다리기 위해 WaitForMultipleObjects 함수를 사용한다. WaitForMultipleObjects 가 끝나면, 주 쓰레드는 안전하게 버퍼에 기록할 수 있게 된다. 그 다음에 hGlobalWriteEvent 와 모든 읽기 작업 쓰레드 사건 객체를 신호 상태로 만들어 버퍼를 읽으려는 쓰레드가 읽기 작업을 계속할 수 있도록 한다.

```
VOID WriteToBuffer(VOID) {
DWORD dwWaitResult, i;

/* Reset hGlobalWriteEvent to nonsignaled, to block readers. */

if (! ResetEvent(hGlobalWriteEvent) ) {
        /* error exit */
}

/* Wait for all reading threads to finish reading. */

dwWaitResult = WaitForMultipleObjects(
        NUMTHREADS,   /* number of handles in array   */
        hReadEvents,  /* array of read-event handles  */
        TRUE,         /* wait until all are signaled  */
        INFINITE);    /* indefinite wait              */
```

```
switch (dwWaitResult) {

   /* All read-event objects were signaled. */

   case WAIT_OBJECT_0:
       .
       . /* Write to the shared buffer. */
       .
       break;

   /* An error occurred. */

   default:

       printf("Wait error: %d\n", GetLastError());
       ExitProcess(0);

}

/* Set hGlobalWriteEvent to signaled. */

if (! SetEvent(hGlobalWriteEvent) ) {
       /* error exit */
}

/* Set all read events to signaled. */

for(i = 1; i <= NUMTHREADS; i++)
   if (! SetEvent(hReadEvents[i]) ) {
          /* error exit */
   }

}
```

읽기 작업을 시작하기 전에 공유버퍼를 읽는 각 쓰레드들은 WaitForMultipleObject 함수를 사용해서 위의 예에서 사용되는 전역변수 hGlobalWriteEvent 가 신호상태가 되기까지 기다린다. WaitForMultipleObject 가 끝나면, 읽기 쓰레드의 자동 리셋 사건 객체는 비신호상태로 재설정된다. 이것은 읽기 쓰레드가 SetEvent 함수를 사용해서 사건 객체의 상태를 신호상태로 되돌리기 전에 주 쓰레드가 버퍼에 쓰는 것을 막는다.

```
VOID ThreadFunction(LPVOID lpParam) {
DWORD dwWaitResult, i;
HANDLE hEvents[2];

hEvents[0] = (HANDLE) *lpParam;  /* thread's read event */
hEvents[1] = hGlobalWriteEvent;

dwWaitResult = WaitForMultipleObjects(
        2,           /* number of handles in array   */
        hEvents,      /* array of event handles       */
        TRUE,        /* wait till all are signaled   */
        INFINITE);    /* indefinite wait              */

switch (dwWaitResult) {

   /* Both event objects were signaled. */

   case WAIT_OBJECT_0:
        .
        . /* Read from the shared buffer. */
        .
        break;

   /* An error occurred. */

   default:

        printf("Wait error: %d\n", GetLastError());
        ExitThread(0);

}

/* Set the read event to signaled. */

if (! SetEvent(hEvents[0]) ) {
        /* error exit */
}


}
```

임계 부분 객체 사용

다음은 쓰레드가 임계부분을 어떻게 초기화하고, 들어가고, 나오는지를 보인 예이다. 뮤텍

스의 예에서 처럼, 이 예에서도 try-finally 구조의 예외처리를 사용한다. 이는 임계 부분 객체의 소유권을 풀어 주는 LeaveCriticalSection 함수를 꼭 호출하도록 만든다.

```
CRITICAL_SECTION GlobalCriticalSection;

/* Initialize the critical section. */

InitializeCriticalSection(&GlobalCriticalSection);

/* Request ownership of the critical section. */

try {

   EnterCriticalSection(&GlobalCriticalSection);
      .
      . /* Access the shared resource. */
      .
}
finally {

   /* Release ownership of the critical section. */

   LeaveCriticalSection(&GlobalCriticalSection);

}
```