Auteur : J. ALVAREZ
Catégorie : Cours
Département : IRIST

Les API WIN32

Sujet : Langage (V 2)

Institution Saint Joseph
413, avenue de Boufflers
54524 LAXOU-NANCY

Email : jalvarez@instit-st-jo.asso.fr
URL : www.instit-st-jo.asso.fr
m.a.j : ***17/10/2002 18:17***

An application consists of one or more processes. A *process*, in the simplest terms, is an executing program. One or more threads run in the context of the process. A *thread* is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread. A *fiber* is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them.
A *job object* allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on the job object affect all processes associated with the job object.

1.1 ABOUT PROCESSES AND THREADS

Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, data, object handles, environment variables, a base priority, and minimum and maximum working set sizes. Each process is started with a single thread, often called the *primary thread*, but can create additional threads from any of its threads.
All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, and a set of structures the system will use to save the thread context until it is scheduled. The *thread context* includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process.
Windows NT supports *preemptive multitasking*, which creates the effect of simultaneous execution of multiple threads from multiple processes. On a multiprocessor computer, Windows NT can simultaneously execute as many threads as there are processors on the computer.

1.2 MULTITASKING

A multitasking operating system divides the available processor time among the processes or threads that need it. The system is designed for preemptive multitasking; it allocates a processor *time slice* to each thread it executes. The currently executing thread is suspended when its time slice elapses, allowing another thread to run. When the system switches from one thread to another, it saves the context of the preempted thread and restores the saved context of the next thread in the queue.
The length of the time slice depends on the operating system and the processor. Because each time slice is small (approximately 20 milliseconds), multiple threads appear to be executing at the same time. This is actually the case on multiprocessor systems, where the executable threads are distributed among the available processors. However, you must use caution when using multiple threads in an application, because system performance can decrease if there are too many threads.

To the user, the advantage of multitasking is the ability to have several applications open and working at the same time. For example, a user can edit a file with one application while another application is recalculating a spreadsheet.
To the application developer, the advantage of multitasking is the ability to create applications that use more than one process and to create processes that use more than one thread of execution. For example, a process can have a user interface thread that manages interactions with the user (keyboard and mouse input), and worker threads that perform other tasks while the user interface thread waits for user input. If you give the user interface thread a higher priority, the application will be more responsive to the user, while the worker threads use the processor efficiently during the times when there is no user input.

There are two ways to implement multitasking: as a single process with multiple threads or as multiple processes, each with one or more threads. An application can put each thread that requires a private address space and private resources into its own process, to protect it from the activities of other process threads.
A multithreaded process can manage mutually exclusive tasks with threads, such as providing a user interface and performing background calculations. Creating a multithreaded process can also be a convenient way to structure a program that performs several similar or identical tasks concurrently. For example, a named pipe server can create a thread for each client process that attaches to the pipe. This thread manages the communication between the server and the client.
Your process could use multiple threads to accomplish the following tasks:

- Manage input for multiple windows.
- Manage input from several communications devices.
- Distinguish tasks of varying priority. For example, a high-priority thread manages time-critical tasks, and a low-priority thread performs other tasks.
- Allow the user interface to remain responsive, while allocating time to background tasks.

It is typically more efficient for an application to implement multitasking by creating a single, multithreaded process, rather than creating multiple processes, for the following reasons:

- The system can perform a context switch more quickly for threads than processes, because a process has more overhead than a thread does (the process context is larger than the thread context).
- All threads of a process share the same address space and can access the process's global variables, which can simplify communication between threads.
- All threads of a process can share open handles to resources, such as files and pipes.

There are other techniques you can use in the place of multithreading. The most significant of these are as follows: asynchronous input and output (I/O), I/O completion ports, asynchronous procedure calls (APC), and the ability to wait for multiple events.
A single thread can initiate multiple time-consuming I/O requests that can run concurrently using asynchronous I/O. Asynchronous I/O can be performed on files, pipes, and serial communication devices. For more information, see Synchronization and Overlapped Input and Output.
A single thread can block its own execution while waiting for any one or all of several events to occur. This is more efficient than using multiple threads, each waiting for a single event, and more efficient than using a single thread that consumes processor time by continually checking for events to occur. For more information, see Wait Functions.

The recommended guideline is to use as few threads as possible, thereby minimizing the use of system resources. This improves performance. Multitasking has resource requirements and potential conflicts to be considered when designing your application. The resource requirements are as follows:

- The system consumes memory for the context information required by both processes and threads. Therefore, the number of processes and threads that can be created is limited by available memory.
- Keeping track of a large number of threads consumes significant processor time. If there are too many threads, most of them will not be able to make significant progress. If most of the current threads are in one process, threads in other processes are scheduled less frequently.

Providing shared access to resources can create conflicts. To avoid them, you must synchronize access to shared resources. This is true for system resources (such as communications ports), resources shared by multiple processes (such as file handles), or the resources of a single process

(such as global variables) accessed by multiple threads. Failure to synchronize access properly (in the same or in different processes) can lead to problems such as *deadlock* and *race conditions*. The synchronization objects and functions you can use to coordinate resource sharing among multiple threads. For more information about synchronization, see Synchronizing Execution of Multiple Threads. Reducing the number of threads makes it easier and more effective to synchronize resources.
A good design for a multithreaded application is the pipeline server. In this design, you create one thread per processor and build queues of requests for which the application maintains the context information. A thread would process all requests in a queue before processing requests in the next queue.

1.3 SCHEDULING

The system scheduler controls multitasking by determining which of the competing threads receives the next processor time slice. The scheduler determines which thread runs next using its scheduling priority.

1.4 MULTIPLE THREADS

Each process is started with a single thread, but can create additional threads from any of its threads.

The CreateThread function creates a new thread for a process. The creating thread must specify the starting address of the code that the new thread is to execute. Typically, the starting address is the name of a function defined in the program code. This function takes a single parameter and returns a DWORD value. A process can have multiple threads simultaneously executing the same function. The following example demonstrates how to create a new thread that executes the locally defined function, ThreadFunc.

```
#include <windows.h>
#include <conio.h>

DWORD WINAPI ThreadFunc( LPVOID lpParam )
{
   char szMsg[80];

   wsprintf( szMsg, "Parameter = %d.", *(DWORD*)lpParam );
   MessageBox( NULL, szMsg, "ThreadFunc", MB_OK );

   return 0;
}

VOID main( VOID )
{
   DWORD dwThreadId, dwThrdParam = 1;
   HANDLE hThread;
   char szMsg[80];

   hThread = CreateThread(
      NULL,                   // no security attributes
      0,                      // use default stack size
      ThreadFunc,             // thread function
      &dwThrdParam,           // argument to thread function
      0,                      // use default creation flags
      &dwThreadId);           // returns the thread identifier

  // Check the return value for success.

  if (hThread == NULL)
  {
```

```
    wsprintf( szMsg, "CreateThread failed." );
    MessageBox( NULL, szMsg, "main", MB_OK );
  }
  else
  {
    _getch();
    CloseHandle( hThread );
  }
}
```

For simplicity, this example passes a pointer to a value as an argument to the thread function. This could be a pointer to any type of data or structure, or it could be omitted altogether by passing a NULL pointer and deleting the references to the parameter in ThreadFunc.
It is risky to pass the address of a local variable if the creating thread exits before the new thread, because the pointer becomes invalid. Instead, either pass a pointer to dynamically allocated memory or make the creating thread wait for the new thread to terminate. Data can also be passed from the creating thread to the new thread using global variables. With global variables, it is usually necessary to synchronize access by multiple threads. For more information about synchronization, see Synchronizing Execution of Multiple Threads.
In processes where a thread might create multiple threads to execute the same code, it is inconvenient to use global variables. For example, a process that enables the user to open several files at the same time can create a new thread for each file, with each of the threads executing the same thread function. The creating thread can pass the unique information (such as the file name) required by each instance of the thread function as an argument. You cannot use a single global variable for this purpose, but you could use a dynamically allocated string buffer.
The creating thread can use the arguments to CreateThread to specify the following:

- The security attributes for the handle to the new thread. These security attributes include an inheritance flag that determines whether the handle can be inherited by child processes. The security attributes also include a security descriptor, which the system uses to perform access checks on all subsequent uses of the thread's handle before access is granted.
- The initial stack size of the new thread. The thread's stack is allocated automatically in the memory space of the process; the system increases the stack as needed and frees it when the thread terminates.
- A creation flag that enables you to create the thread in a suspended state. When suspended, the thread does not run until the _____ function is called.

You can also create a thread by calling the CreateRemoteThread function. This function is used by debugger processes to create a thread that runs in the address space of the process being debugged.

Each new thread receives its own stack space, consisting of both committed and reserved memory. The system will commit one page blocks from the reserved stack memory as needed, until the stack cannot grow any farther.
The default size for committed and reserved memory is specified in the executable file header. To specify a different default stack size, use the STACKSIZE statement in the module definition (.DEF) file. Your linker may also support a command-line option for setting the stack size. For more information, see the documentation included with your linker.
To increase the amount of stack space which is to be initially committed for a thread, specify the value in the *dwStackSize* parameter of the CreateThread or CreateRemoteThread function. This value is rounded to the nearest page. The call to create the thread fails if there is not enough memory to commit or reserve the number of bytes requested. If *dwStackSize* is smaller than the default reserve size, the new thread uses the default reserve size. If *dwStackSize* is larger than the default reserve size, the reserve size is rounded up to the nearest multiple of 1 MB.

Windows XP: If the *dwCreationFlags* parameter of CreateThread or CreateRemoteThread is STACK_SIZE_PARAM_IS_A_RESERVATION, the *dwStackSize* parameter specifies the amount of stack space which is to be initially reserved for the thread.
The stack is freed when the thread terminates.

When a new thread is created by the CreateThread or CreateRemoteThread function, a handle to the thread is returned. By default, this handle has full access rights, and — subject to security access checking — can be used in any of the functions that accept a thread handle. This handle can be inherited by child processes, depending on the inheritance flag specified when it is created. The handle can be duplicated by DuplicateHandle, which enables you to create a thread handle with a subset of the access rights. The handle is valid until closed, even after the thread it represents has been terminated.
The CreateThread and CreateRemoteThread functions also return an identifier that uniquely identifies the thread throughout the system. A thread can use the GetCurrentThreadId function to get its own thread identifier. The identifiers are valid from the time the thread is created until the thread has been terminated.
Windows Me, Windows 2000/XP: If you have a thread identifier, you can get the thread handle by calling the OpenThread function. OpenThread enables you to specify the handle's access rights and whether it can be inherited.
Windows NT 4.0 and earlier, Windows 95/98/Me: There is no way to get the thread handle from the thread identifier. If the handles were made available this way, the owning process could fail because another process unexpectedly performed an operation on one of its threads, such as suspending it, resuming it, adjusting its priority, or terminating it. Instead, you must request the handle from the thread creator or the thread itself.
A thread can use the GetCurrentThread function to retrieve a *pseudo handle* to its own thread object. This pseudo handle is valid only for the calling process; it cannot be inherited or duplicated for use by other processes. To get the real handle to the thread, given a pseudo handle, use the DuplicateHandle function.

A thread can suspend and resume the execution of another thread using the SuspendThread and ResumeThread functions. While a thread is suspended, it is not scheduled for time on the processor. The SuspendThread function is not particularly useful for synchronization because it does not control the point in the code at which the thread's execution is suspended. However, you might want to suspend a thread in a situation where you are waiting for user input that could cancel the work the thread is performing. If the user input cancels the work, have the thread exit; otherwise, call ResumeThread.
If a thread is created in a suspended state (with the CREATE_SUSPENDED flag), it does not begin to execute until another thread calls ResumeThread with a handle to the suspended thread. This can be useful for initializing the thread's state before it begins to execute. Suspending a thread at creation can be useful for one-time synchronization, because this ensures that the suspended thread will execute the starting point of its code when you call ResumeThread.
A thread can temporarily yield its execution for a specified interval by calling the Sleep or SleepEx functions This is useful particularly in cases where the thread responds to user interaction, because it can delay execution long enough to allow users to observe the results of their actions. During the sleep interval, the thread is not scheduled for time on the processor.
The SwitchToThread function is similar to Sleep and SleepEx, except that you cannot specify the interval. SwitchToThread allows the thread to give up its time slice.

To avoid race conditions and deadlocks, it is necessary to synchronize access by multiple threads to shared resources. Synchronization is also necessary to ensure that interdependent code is executed in the proper sequence.

There are a number of objects whose handles can be used to synchronize multiple threads. These objects include:

- Console input buffers
- Events
- Mutexes
- Processes
- Semaphores
- Threads
- Timers

The state of each of these objects is either signaled or not signaled. When you specify a handle to any of these objects in a call to one of the wait functions, the execution of the calling thread is blocked until the state of the specified object becomes signaled.
Some of these objects are useful in blocking a thread until some event occurs. For example, a console input buffer handle is signaled when there is unread input, such as a keystroke or mouse button click. Process and thread handles are signaled when the process or thread terminates. This allows a process, for example, to create a child process and then block its own execution until the new process has terminated.
Other objects are useful in protecting shared resources from simultaneous access. For example, multiple threads can each have a handle to a mutex object. Before accessing a shared resource, the threads must call one of the wait functions to wait for the state of the mutex to be signaled. When the mutex becomes signaled, only one waiting thread is released to access the resource. The state of the mutex is immediately reset to not signaled so any other waiting threads remain blocked. When the thread is finished with the resource, it must set the state of the mutex to signaled to allow other threads to access the resource.
For the threads of a single process, critical-section objects provide a more efficient means of synchronization than mutexes. A critical section is used like a mutex to enable one thread at a time to use the protected resource. A thread can use the EnterCriticalSection function to request ownership of a critical section. If it is already owned by another thread, the requesting thread is blocked. A thread can use the TryEnterCriticalSection function to request ownership of a critical section, without blocking upon failure to obtain the critical section. After it receives ownership, the thread is free to use the protected resource. The execution of the other threads of the process is not affected unless they attempt to enter the same critical section.
The WaitForInputIdle function makes a thread wait until a specified process is initialized and waiting for user input with no input pending. Calling WaitForInputIdle can be useful for synchronizing parent and child processes, because CreateProcess returns without waiting for the child process to complete its initialization.
For more information, see Synchronization.

A thread executes until one of the following events occurs:

- The thread calls the _____ function.
- Any thread of the process calls the _____ function.
- The thread function returns.
- Any thread calls the _____ function with a handle to the thread.
- Any thread calls the _____ function with a handle to the process.

The GetExitCodeThread function returns the termination status of a thread. While a thread is executing, its termination status is STILL_ACTIVE. When a thread terminates, its termination status changes from STILL_ACTIVE to the exit code of the thread. The exit code is either the value specified in the call to ExitThread, ExitProcess, TerminateThread, or TerminateProcess, or the value returned by the thread function.
When a thread terminates, the state of the thread object changes to signaled, releasing any other threads that had been waiting for the thread to terminate. For more about synchronization, see Synchronizing Execution of Multiple Threads.
If a thread is terminated by ExitThread, the system calls the entry-point function of each attached DLL with a value indicating that the thread is detaching from the DLL (unless you call the DisableThreadLibraryCalls function). If a thread is terminated by ExitProcess, the DLL entry-point functions are invoked once, to indicate that the process is detaching. DLLs are not notified when a thread is terminated by TerminateThread or TerminateProcess. For more information about DLLs, see Dynamic-Link Libraries.
Warning The TerminateThread and TerminateProcess functions should be used only in extreme circumstances, since they do not allow threads to clean up, do not notify attached DLLs, and do not free the initial stack. The following steps provide a better solution:

- Create an event object using the _____ function.
- Create the threads.
- Each thread monitors the event state by calling the _____ function. Use a wait time-out interval of zero.
- Each thread terminates its own execution when the event is set to the signaled state (returns WAIT_OBJECT_0).

_____

The GetThreadTimes function obtains timing information for a thread. It returns the thread creation time, how much time the thread has been executing in kernel mode, and how much time the thread has been executing in user mode. These times do not include time spent executing system threads or waiting in a suspended or blocked state. If the thread has exited, GetThreadTimes returns the thread exit time.

_____

The Windows NT security model enables you to control access to thread objects. For more information about security, see Access-Control Model.
You can specify a security descriptor for a thread when you call the CreateProcess, CreateProcessAsUser, CreateProcessWithLogonW, CreateThread, or CreateRemoteThread function. To retrieve a thread's security descriptor, call the GetSecurityInfo function. To change a thread's security descriptor, call the SetSecurityInfo function.
The handle returned by the CreateThread function has THREAD_ALL_ACCESS access to the thread object. When you call the GetCurrentThread function, the system returns a pseudohandle with the maximum access that the thread's security descriptor allows the caller.
The valid access rights for thread objects include the DELETE, READ_CONTROL, SYNCHRONIZE, WRITE_DAC, and WRITE_OWNER standard access rights, in addition to the following thread-specific access rights.

SYNCHRONIZE	A standard right required to wait for the thread to exit.
THREAD_ALL_ACCESS	Specifies all possible access rights for a thread object.
THREAD_DIRECT_IMPERSONATION	Required for a server thread that impersonates a client.
THREAD_GET_CONTEXT	Required to read the context of a thread using

THREAD_IMPERSONATE	Required to use a thread's security information directly without calling it by using a communication mechanism that provides impersonation services.
THREAD_QUERY_INFORMATION	Required to read certain information from the thread object.
THREAD_SET_CONTEXT	Required to write the context of a thread.
THREAD_SET_INFORMATION	Required to set certain information in the thread object.
THREAD_SET_THREAD_TOKEN	Required to set the impersonation token for a thread.
THREAD_SUSPEND_RESUME	Required to suspend or resume a thread.
THREAD_TERMINATE	Required to terminate a thread.

You can request the ACCESS_SYSTEM_SECURITY access right to a thread object if you want to read or write the object's SACL. For more information, see Access-Control Lists (ACLs) and SACL Access Right.

1.5 CHILD PROCESSES

A *child process* is a process that is created by another process, called the *parent process*

The CreateProcess function creates a new process, which runs independently of the creating process. However, for simplicity, the relationship is referred to as a parent-child relationship. The following code fragment demonstrates how to create a process.

```
void main( VOID )
{
   STARTUPINFO si;
   PROCESS_INFORMATION pi;

   ZeroMemory( &si, sizeof(si) );
   si.cb = sizeof(si);
   ZeroMemory( &pi, sizeof(pi) );

   // Start the child process.
   if( !CreateProcess( NULL, // No module name (use command line).
      "MyChildProcess", // Command line.
      NULL,             // Process handle not inheritable.
      NULL,             // Thread handle not inheritable.
      FALSE,            // Set handle inheritance to FALSE.
      0,              // No creation flags.
      NULL,             // Use parent's environment block.
      NULL,             // Use parent's starting directory.
      &si,             // Pointer to STARTUPINFO structure.
      &pi )            // Pointer to PROCESS_INFORMATION structure.
   )
   {
      ErrorExit( "CreateProcess failed." );
   }

   // Wait until child process exits.
   WaitForSingleObject( pi.hProcess, INFINITE );

   // Close process and thread handles.
   CloseHandle( pi.hProcess );
   CloseHandle( pi.hThread );
}
```

If CreateProcess succeeds, it returns a PROCESS_INFORMATION structure containing handles and identifiers for the new process and its primary thread. The thread and process handles are created with full access rights, although access can be restricted if you specify security descriptors. When you no longer need these handles, close them by using the CloseHandle function.
You can also create a process using the CreateProcessAsUser function. This function allows you to specify the security context of the user account in which the process will execute.

A parent process can specify properties associated with the main window of its child process. The CreateProcess function takes a pointer to a STARTUPINFO structure as one of its parameters. Use the members of this structure to specify characteristics of the child process's main window. The dwFlags member contains a bit field that determines which other members of the structure are used. This allows you to specify values for any subset of the window properties. The system uses default values for the properties you do not specify. The dwFlags member can also force a feedback cursor to be displayed during the initialization of the new process.
For GUI processes, the STARTUPINFO structure specifies the default values to be used the first time the new process calls the CreateWindow and ShowWindow functions to create and display an overlapped window. The following default values can be specified:

- The width and height, in pixels, of the window created by .
- The location, in screen coordinates of the window created by .
- The *nCmdShow* parameter of .

For console processes, use the STARTUPINFO structure to specify window properties only when creating a new console (either using CreateProcess with CREATE_NEW_CONSOLE or with the AllocConsole function). The STARTUPINFO structure can be used to specify the following console window properties:

- The size of the new console window, in character cells.
- The location of the new console window, in screen coordinates.
- The size, in character cells, of the new console's screen buffer.
- The text and background color attributes of the new console's screen buffer.
- The title of the new console's window.

When a new process is created by the CreateProcess function, handles of the new process and its primary thread are returned. These handles are created with full access rights, and — subject to security access checking — can be used in any of the functions that accept thread or process handles. These handles can be inherited by child processes, depending on the inheritance flag specified when they are created. The handles are valid until closed, even after the process or thread they represent has been terminated.
The CreateProcess function also returns an identifier that uniquely identifies the process throughout the system. A process can use the GetCurrentProcessId function to get its own process identifier. The identifier is valid from the time the process is created until the process has been terminated.
If you have a process identifier, you can get the process handle by calling the OpenProcess function. OpenProcess enables you to specify the handle's access rights and whether it can be inherited.
A process can use the GetCurrentProcess function to retrieve a pseudo handle to its own process object. This pseudo handle is valid only for the calling process; it cannot be inherited or duplicated for use by other processes. To get the real handle to the process, call the DuplicateHandle function.

There are a variety of functions for obtaining information about processes. Some of these functions can be used only for the calling process, because they do not take a process handle as a parameter. You can use functions that take a process handle to obtain information about other processes.

- To obtain the command-line string for the current process, use the _____ function.
- To parse a Unicode command-line string obtained from the Unicode version of , use the _____ function.
- To retrieve the structure specified when the current process was created, use the _____ function.
- To obtain the version information from the executable header, use the _____ function.
- To obtain the full path and file name for the executable file containing the process code, use the _____ function.
- To obtain the count of handles to graphical user interface (GUI) objects in use, use the _____ function.
- To determine whether a process is being debugged, use the _____ function.
- To retrieve accounting information for all I/O operations performed by the process, use the _____ function.

A child process can inherit several properties and resources from its parent process. You can also prevent a child process from inheriting properties from its parent process. The following can be inherited:

- Open handles returned by the _____ function. This includes handles to files, console input buffers, console screen buffers, named pipes, serial communication devices, and mailslots.
- Open handles to process, thread, mutex, event, semaphore, named-pipe, anonymous-pipe, and file-mapping objects.
- Environment variables.
- The current directory.
- The console, unless the process is detached or a new console is created. A child console process also inherits the parent's standard handles, as well as access to the input buffer and the active screen buffer.

The child process does not inherit the following:

- Priority class.
- Handles returned by _____, _____, _____, and _____.
- Pseudo handles, as in the handles returned by the _____ or _____ function. These handles are valid only for the calling process.
- DLL module handles returned by the _____ function.
- GDI or USER handles, such as or .

1.5.5.1 INHERITING HANDLES

To cause a handle to be inherited, you must do two things:

- Specify that the handle is to be inherited when you create, open, or duplicate the handle.
- Specify that inheritable handles are to be inherited when you call the _____ function.

This allows a child process to inherit some of its parent's handles, but not inherit others. For example, creation functions such as CreateProcess and CreateFile take a security attributes argument that determines whether the handle can be inherited. Open functions such as OpenMutex and OpenEvent take a handle inheritance flag that determines whether the handle can be inherited. The DuplicateHandle function takes a handle inheritance flag that determines whether the handle can be inherited.
When a child process is created, the *fInheritHandles* parameter of CreateProcess determines whether the inheritable handles of the parent process are inherited by the child process. An inherited handle refers to the same object in the child process as it does in the parent process. It also has the same value and access privileges. Therefore, when one process changes the state of the object, the change affects both processes. To use a handle, the child process must retrieve the handle value and "know" the object to which it refers. Usually, the parent process communicates this information to the child process through its command line, environment block, or some form of interprocess communication.
The DuplicateHandle function is useful if a process has an inheritable open handle that you do not want to be inherited by the child process. In this case, use DuplicateHandle to open a duplicate of the handle that cannot be inherited, then use the CloseHandle function to close the inheritable handle. You can also use the DuplicateHandle function to open an inheritable duplicate of a handle that cannot be inherited.

1.5.5.2 INHERITING ENVIRONMENT VARIABLES

A child process inherits the environment variables of its parent process by default. However, CreateProcess enables the parent process to specify a different block of environment variables. For more information, see Environment Variables.

1.5.5.3 INHERITING THE CURRENT DIRECTORY

The GetCurrentDirectory function retrieves the current directory of the calling process. A child process inherits the current directory of its parent process by default. However, CreateProcess enables the parent process to specify a different current directory for the child process. To change the current directory of the calling process, use the SetCurrentDirectory function.

Every process has an environment block that contains a set of environment variables and their values. The command processor provides the set command to display its environment block or to create new environment variables. Programs started by the command processor inherit the command processor's environment variables.
By default, a child process inherits the environment variables of its parent process. However, you can specify a different environment for the child process by creating a new environment block and passing a pointer to it as a parameter to the CreateProcess function.
The GetEnvironmentStrings function returns a pointer to the environment block of the calling process. This should be treated as a read-only block; do not modify it directly. Instead, use the SetEnvironmentVariable function to change an environment variable. When you are finished with the environment block obtained from GetEnvironmentStrings, call the FreeEnvironmentStrings function to free the block.
The GetEnvironmentVariable function determines whether a specified variable is defined in the environment of the calling process, and, if so, what its value is.
For more information, see the examples in Changing Environment Variables.

A process executes until one of the following events occurs:

- Any thread of the process calls the _____ function. This terminates all threads of the process.
- The primary thread of the process returns. The primary thread can avoid terminating other threads by explicitly calling _____ before it returns. One of the remaining threads can still call to ensure that all threads are terminated.
- The last thread of the process terminates.
- Any thread calls the _____ function with a handle to the process. This terminates all threads of the process, without allowing them to clean up or save data.
- For console processes, the default handler function calls _____ when the console receives a CTRL+C or CTRL+BREAK signal. All console processes attached to the console receive these signals. Detached processes and GUI processes are not affected by CTRL+C or CTRL+BREAK signals. For more information, see _____.
- The user shuts down the system or logs off. Use the _____ function to specify shutdown parameters, such as when a process should terminate relative to the other processes in the system. The _____ function retrieves the current shutdown priority of the process and other shutdown flags.

When a process is terminated, all threads of the process are terminated immediately with no chance to run additional code. This means that the process does not execute code in termination handler blocks. For more information, see Structured Exception Handling.
The GetExitCodeProcess function returns the termination status of a process. While a process is executing, its termination status is STILL_ACTIVE. When a process terminates, its termination status changes from STILL_ACTIVE to the exit code of the process. The exit code is either the value specified in the call to ExitProcess or TerminateProcess, or the value returned by the main or WinMain function of the process. If a process is terminated due to a fatal exception, the exit code is the value of the exception that caused the termination. In addition, this value is used as the exit code for all the threads that were executing when the exception occurred.
When a process terminates, the state of the process object becomes signaled, releasing any threads that had been waiting for the process to terminate. For more about synchronization, see Synchronizing Execution of Multiple Threads.
Open handles to files or other resources are closed automatically when a process terminates. However, the objects themselves exist until all open handles to them are closed. This means that an object remains valid after a process closes, if another process has a handle to it.
If a process is terminated by ExitProcess, the system calls the entry-point function of each attached DLL with a value indicating that the process is detaching from the DLL. DLLs are not notified when a process is terminated by TerminateProcess. For more information about DLLs, see Dynamic-Link Libraries.
The execution of the ExitProcess, ExitThread, CreateThread, CreateRemoteThread, and CreateProcess functions is serialized within an address space. The following restrictions apply:

- During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is finished for the process.
- Only one thread at a time can be in a DLL initialization or detach routine.
- The function does not return until there are no threads are in their DLL initialization or detach routines.

Warning The TerminateProcess function should be used only in extreme circumstances, since it does not allow threads to clean up or save data and does not notify attached DLLs. If you need to have one process terminate another process, the following steps provide a better solution:

- Have both processes call the _____ function to create a private message.
- One process can terminate the other process by broadcasting the private message using the _____ function as follows:

```
BroadcastSystemMessage(
   BSF_IGNORECURRENTTASK, // do not send message to this process
   BSM_APPLICATIONS,      // broadcast only to applications
   private message,       // message registered in previous step
   wParam,                // message-specific value
   lParam );              // message-specific value
```

- The process receiving the private message calls _____ to terminate its execution.

Note When the system is terminating a process, it does not terminate any child processes that the process has created.

The GetProcessTimes function obtains timing information for a process. It returns the process creation time, how much time the process has been executing in kernel mode, and how much time the process has been executing in user mode. These times do not include time spent executing system threads or waiting in a suspended or blocked state. If the process has exited, GetProcessTimes returns the process exit time.

The Windows NT security model enables you to control access to process objects. For more information about security, see Access-Control Model.
You can specify a security descriptor for a process when you call the CreateProcess, CreateProcessAsUser, or CreateProcessWithLogonW function. To retrieve a process's security descriptor, call the GetSecurityInfo function. To change a process's security descriptor, call the SetSecurityInfo function.
The handle returned by the CreateProcess function has PROCESS_ALL_ACCESS access to the process object. When you call the OpenProcess function, the system checks the requested access rights against the DACL in the process's security descriptor. When you call the GetCurrentProcess function, the system returns a pseudohandle with the maximum access that the DACL allows to the caller.
The valid access rights for process objects include the DELETE, READ_CONTROL, SYNCHRONIZE, WRITE_DAC, and WRITE_OWNER standard access rights, in addition to the following process-specific access rights.

PROCESS_ALL_ACCESS	Specifies all possible access rights for a process object.
PROCESS_CREATE_PROCESS	Required to create a process.
PROCESS_CREATE_THREAD	Required to create a thread.
PROCESS_DUP_HANDLE	Required to duplicate a handle.
PROCESS_QUERY_INFORMATION	Required to retrieve certain information about a process, such as its priority class.
PROCESS_SET_QUOTA	Required to set memory limits.
PROCESS_SET_INFORMATION	Required to set certain information about a process, such as its priority class.

PROCESS_TERMINATE	Required to terminate a process.
PROCESS_VM_OPERATION	Required to perform an operation on the address space of a process.
PROCESS_VM_READ	Required to read memory in a process.
PROCESS_VM_WRITE	Required to write to memory in a process.
SYNCHRONIZE	A standard right required to wait for the process to terminate.

You can request the ACCESS_SYSTEM_SECURITY access right to a process object if you want to read or write the object's SACL. For more information, see Access-Control Lists (ACLs) and SACL Access Right.

1.6 JOB OBJECTS

A *job object* allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on the job object affect all processes associated with the job object.
To create a job object, use the CreateJobObject function. When the job is created, there are no associated processes. To associate a process with a job, use the AssignProcessToJobObject function. After you associate a process with a job, the association cannot be broken. By default, processes created by a process associated with a job (child processes) are associated with the job. If the job has the extended limit JOB_OBJECT_LIMIT_BREAKAWAY_OK and the process was created with the CREATE_BREAKAWAY_FROM_JOB flag, its child processess are not associated with the job. If the job has the extended limit JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK, no child processes are associated with the job.
To determine if a process is running in a job, use the IsProcessInJob function.
A job can enforce limits on each associated process, such as the working set size, process priority, end-of-job time limit, and so on. To set limits for a job object, use the SetInformationJobObject function. If a process associated with a job attempts to increase its working set size or process priority, the function calls are silently ignored.
The job object also records basic accounting information for all its associated processes, including those that have terminated. To retrieve this accounting information, use the QueryInformationJobObject function.
To terminate all processes currently associated with a job object, use the TerminateJobObject function.
To close a job object handle, use the CloseHandle function. The job object is destroyed when its last handle has been closed. If there are running processes still associated with the job when it is destroyed, they will continue to run even after the job is destroyed.
If a tool is to manage a process tree that uses job objects, both the tool and the members of the process tree must cooperate. Use one of the following options:

1. The tool could use the JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK limit. If the tool uses this limit, it cannot monitor an entire process tree. The tool can monitor only the processes it adds to the job. If these processes create child processes, they are not associated with the job. In this option, child processes can be associated with other job objects.

2. The tool could use the JOB_OBJECT_LIMIT_BREAKAWAY_OK limit. If the tool uses this limit, it can monitor the entire process tree, except for those processes that any member of the tree explicitly breaks away from the tree. A member of the tree can create a child process in a new job object by calling the _____ function with the CREATE_BREAKAWAY_FROM_JOB flag, then calling the _____ function. Otherwise, the member must handle cases in which fails.

The CREATE_BREAKAWAY_FROM_JOB flag has no effect if the tree is not being monitored by the tool. Therefore, this is the preferred option, but it requires advance knowledge of the processes being monitored.

3. The tool could prevent breakaways of any kind. In this option, the tool can monitor the entire process tree. However, if a process associated with the job tries to call , the call will fail. If the process was not designed to be associated with a job, this failure may be unexpected.

2.1 CREATING A CHILD PROCESS WITH REDIRECTED INPUT AND OUTPUT

The example in this topic demonstrates how to create a child process from a console process. It also demonstrates a technique for using anonymous pipes to redirect the child process's standard input and output handles.
The CreatePipe function uses the SECURITY_ATTRIBUTES structure to create inheritable handles to the read and write ends of two pipes. The read end of one pipe serves as standard input for the child process, and the write end of the other pipe is the standard output for the child process. These pipe handles are specified in the SetStdHandle function, which makes them the standard handles inherited by the child process. After the child process is created, SetStdHandle is used again to restore the original standard handles for the parent process.
The parent process uses the other ends of the pipes to write to the child process's input and read the child process's output. The handles to these ends of the pipe are also inheritable. However, the handle must not be inherited. Before creating the child process, the parent process must use DuplicateHandle to create a duplicate of the application-defined hChildStdinWr global variable that cannot be inherited. It then uses CloseHandle to close the inheritable handle. For more information, see Pipes.
The following is the parent process.

```
#include <stdio.h>
#include <windows.h>

#define BUFSIZE 4096

HANDLE hChildStdinRd, hChildStdinWr, hChildStdinWrDup,
   hChildStdoutRd, hChildStdoutWr, hChildStdoutRdDup,
   hInputFile, hSaveStdin, hSaveStdout;

BOOL CreateChildProcess(VOID);
VOID WriteToPipe(VOID);
VOID ReadFromPipe(VOID);
VOID ErrorExit(LPTSTR);
VOID ErrMsg(LPTSTR, BOOL);

DWORD main(int argc, char *argv[])
{
   SECURITY_ATTRIBUTES saAttr;
   BOOL fSuccess;

// Set the bInheritHandle flag so pipe handles are inherited.

   saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
   saAttr.bInheritHandle = TRUE;
   saAttr.lpSecurityDescriptor = NULL;

   // The steps for redirecting child process's STDOUT:
```

```
  //     1. Save current STDOUT, to be restored later.
  //     2. Create anonymous pipe to be STDOUT for child process.
  //     3. Set STDOUT of the parent process to be write handle to
  //        the pipe, so it is inherited by the child process.
  //     4. Create a noninheritable duplicate of the read handle and
  //        close the inheritable read handle.

// Save the handle to the current STDOUT.

   hSaveStdout = GetStdHandle(STD_OUTPUT_HANDLE);

// Create a pipe for the child process's STDOUT.

   if (! CreatePipe(&hChildStdoutRd, &hChildStdoutWr, &saAttr, 0))
      ErrorExit("Stdout pipe creation failed\n");

// Set a write handle to the pipe to be STDOUT.

   if (! SetStdHandle(STD_OUTPUT_HANDLE, hChildStdoutWr))
      ErrorExit("Redirecting STDOUT failed");

// Create noninheritable read handle and close the inheritable read
// handle.

    fSuccess = DuplicateHandle(GetCurrentProcess(), hChildStdoutRd,
        GetCurrentProcess(), &hChildStdoutRdDup , 0,
        FALSE,
        DUPLICATE_SAME_ACCESS);
    if( !fSuccess )
        ErrorExit("DuplicateHandle failed");
    CloseHandle(hChildStdoutRd);

   // The steps for redirecting child process's STDIN:
   //     1.  Save current STDIN, to be restored later.
   //     2.  Create anonymous pipe to be STDIN for child process.
   //     3.  Set STDIN of the parent to be the read handle to the
   //         pipe, so it is inherited by the child process.
   //     4.  Create a noninheritable duplicate of the write handle,
   //         and close the inheritable write handle.

// Save the handle to the current STDIN.

   hSaveStdin = GetStdHandle(STD_INPUT_HANDLE);

// Create a pipe for the child process's STDIN.

   if (! CreatePipe(&hChildStdinRd, &hChildStdinWr, &saAttr, 0))
      ErrorExit("Stdin pipe creation failed\n");

// Set a read handle to the pipe to be STDIN.

   if (! SetStdHandle(STD_INPUT_HANDLE, hChildStdinRd))
      ErrorExit("Redirecting Stdin failed");

// Duplicate the write handle to the pipe so it is not inherited.

   fSuccess = DuplicateHandle(GetCurrentProcess(), hChildStdinWr,
      GetCurrentProcess(), &hChildStdinWrDup, 0,
      FALSE,                  // not inherited
      DUPLICATE_SAME_ACCESS);
   if (! fSuccess)
      ErrorExit("DuplicateHandle failed");

   CloseHandle(hChildStdinWr);
```

```
// Now create the child process.

   if (! CreateChildProcess())
      ErrorExit("Create process failed");

// After process creation, restore the saved STDIN and STDOUT.

   if (! SetStdHandle(STD_INPUT_HANDLE, hSaveStdin))
      ErrorExit("Re-redirecting Stdin failed\n");

   if (! SetStdHandle(STD_OUTPUT_HANDLE, hSaveStdout))
      ErrorExit("Re-redirecting Stdout failed\n");

// Get a handle to the parent's input file.

   if (argc > 1)
      hInputFile = CreateFile(argv[1], GENERIC_READ, 0, NULL,
         OPEN_EXISTING, FILE_ATTRIBUTE_READONLY, NULL);
   else
      hInputFile = hSaveStdin;

   if (hInputFile == INVALID_HANDLE_VALUE)
      ErrorExit("no input file\n");

// Write to pipe that is the standard input for a child process.

   WriteToPipe();

// Read from pipe that is the standard output for child process.

   ReadFromPipe();

   return 0;
}

BOOL CreateChildProcess()
{
   PROCESS_INFORMATION piProcInfo;
   STARTUPINFO siStartInfo;

// Set up members of the PROCESS_INFORMATION structure.

   ZeroMemory( &piProcInfo, sizeof(PROCESS_INFORMATION) );

// Set up members of the STARTUPINFO structure.

   ZeroMemory( &siStartInfo, sizeof(STARTUPINFO) );
   siStartInfo.cb = sizeof(STARTUPINFO);

// Create the child process.

   return CreateProcess(NULL,
      "child",       // command line
      NULL,          // process security attributes
      NULL,          // primary thread security attributes
      TRUE,          // handles are inherited
      0,             // creation flags
      NULL,          // use parent's environment
      NULL,          // use parent's current directory
      &siStartInfo,  // STARTUPINFO pointer
      &piProcInfo);  // receives PROCESS_INFORMATION
}
```

```
VOID WriteToPipe(VOID)
{
   DWORD dwRead, dwWritten;
   CHAR chBuf[BUFSIZE];

// Read from a file and write its contents to a pipe.

   for (;;)
   {
      if (! ReadFile(hInputFile, chBuf, BUFSIZE, &dwRead, NULL) ||
         dwRead == 0) break;
      if (! WriteFile(hChildStdinWrDup, chBuf, dwRead,
         &dwWritten, NULL)) break;
   }

// Close the pipe handle so the child process stops reading.

   if (! CloseHandle(hChildStdinWrDup))
      ErrorExit("Close pipe failed\n");
}

VOID ReadFromPipe(VOID)
{
   DWORD dwRead, dwWritten;
   CHAR chBuf[BUFSIZE];
   HANDLE hStdout = GetStdHandle(STD_OUTPUT_HANDLE);

// Close the write end of the pipe before reading from the
// read end of the pipe.

   if (!CloseHandle(hChildStdoutWr))
      ErrorExit("Closing handle failed");

// Read output from the child process, and write to parent's STDOUT.

   for (;;)
   {
      if( !ReadFile( hChildStdoutRdDup, chBuf, BUFSIZE, &dwRead,
         NULL) || dwRead == 0) break;
      if (! WriteFile(hSaveStdout, chBuf, dwRead, &dwWritten, NULL))
         break;
   }
}

VOID ErrorExit (LPTSTR lpszMessage)
{
   fprintf(stderr, "%s\n", lpszMessage);
   ExitProcess(0);
}

// The code for the child process.

#include <windows.h>
#define BUFSIZE 4096

VOID main(VOID)
{
   CHAR chBuf[BUFSIZE];
   DWORD dwRead, dwWritten;
   HANDLE hStdin, hStdout;
   BOOL fSuccess;

   hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
   hStdin = GetStdHandle(STD_INPUT_HANDLE);
```

```
  if ((hStdout == INVALID_HANDLE_VALUE) ||
     (hStdin == INVALID_HANDLE_VALUE))
     ExitProcess(1);

  for (;;)
  {
  // Read from standard input.
     fSuccess = ReadFile(hStdin, chBuf, BUFSIZE, &dwRead, NULL);
     if (! fSuccess || dwRead == 0)
        break;

  // Write to standard output.
     fSuccess = WriteFile(hStdout, chBuf, dwRead, &dwWritten, NULL);
     if (! fSuccess)
        break;
  }
}
```

2.2 CHANGING ENVIRONMENT VARIABLES

Each process has an environment block associated with it. The environment block consists of a null-terminated block of null-terminated strings (meaning there are two null bytes at the end of the block), where each string is in the form:

name=value

All strings in the environment block must be sorted alphabetically by name. The sort is case-insensitive, Unicode order, without regard to locale. Because the equal sign is a separator, it must not be used in the name of an environment variable.

By default, a child process inherits a copy of the environment block of the parent process. The following example demonstrates how to create a new environment block to pass to a child process.

```
LPTSTR lpszCurrentVariable;
BOOL fSuccess;

// Copy environment strings into an environment block.

lpszCurrentVariable = tchNewEnv;
if (lstrcpy(lpszCurrentVariable, "MyVersion=2") == NULL)
   ErrorExit("lstrcpy failed");

lpszCurrentVariable += lstrlen(lpszCurrentVariable) + 1;
if (lstrcpy(lpszCurrentVariable, "MySetting=A") == NULL)
   ErrorExit("lstrcpy failed");

// Terminate the block with a NULL byte.

lpszCurrentVariable += lstrlen(lpszCurrentVariable) + 1;
*lpszCurrentVariable = '\0';

// Create the child process, specifying a new environment block.

fSuccess = CreateProcess(NULL, "childenv", NULL, NULL, TRUE, 0,
   (LPVOID) tchNewEnv,       // new environment block
   NULL, &siStartInfo, &piProcInfo);

if (! fSuccess)
    ErrorExit("CreateProcess failed");
```

If you want the child process to inherit most of the parent's environment with only a few changes, save the current values, make changes for the child process to inherit, create the child process, and then restore the saved values, as shown following.

```
LPTSTR lpszOldValue;
TCHAR tchBuf[BUFSIZE];
BOOL fSuccess;
```

```
// lpszOldValue gets current value of "varname", or NULL if "varname"
// environment variable does not exist. Set "varname" to new value,
// create child process, then use SetEnvironmentVariable to restore
// original value of "varname". If lpszOldValue is NULL, the "varname"
// variable will be deleted.

lpszOldValue = ((GetEnvironmentVariable("varname",
    tchBuf, BUFSIZE) > 0) ? tchBuf : NULL);

// Set a value for the child process to inherit.

if (! SetEnvironmentVariable("varname", "newvalue"))
   ErrorExit("SetEnvironmentVariable failed");

// Create a child process.

fSuccess = CreateProcess(NULL, "childenv", NULL, NULL, TRUE, 0,
   NULL,     // inherit parent's environment
   NULL, &siStartInfo, &piProcInfo);
if (! fSuccess)
   ErrorExit("CreateProcess failed");

// Restore the parent's environment.

if (! SetEnvironmentVariable("varname", lpszOldValue))
   ErrorExit("SetEnvironmentVariable failed");
```

The following example, taken from a console process, prints the contents of the process's environment block.

```
LPTSTR lpszVariable;
LPVOID lpvEnv;

// Get a pointer to the environment block.

lpvEnv = GetEnvironmentStrings();

// Variable strings are separated by NULL byte, and the block is
// terminated by a NULL byte.

for (lpszVariable = (LPTSTR) lpvEnv; *lpszVariable; lpszVariable++)
{
   while (*lpszVariable)
      putchar(*lpszVariable++);
   putchar('\n');
}
```

2.3 USING THREAD LOCAL STORAGE

Thread local storage (TLS) enables multiple threads of the same process to use an index allocated by the TlsAlloc function to store and retrieve a value that is local to the thread. In this example, an index is allocated when the process starts. When each thread starts, it allocates a block of dynamic memory and stores a pointer to this memory by using the TLS index. The TLS index is used by the locally defined CommonFunc function to access the data local to the calling thread. Before each thread terminates, it releases its dynamic memory.

```
#include <stdio.h>
#include <windows.h>

#define THREADCOUNT 4
DWORD dwTlsIndex;

VOID ErrorExit(LPTSTR);

VOID CommonFunc(VOID)
```

```
{
  LPVOID lpvData;

// Retrieve a data pointer for the current thread.

  lpvData = TlsGetValue(dwTlsIndex);
  if ((lpvData == 0) && (GetLastError() != 0))
    ErrorExit("TlsGetValue error");

// Use the data stored for the current thread.

  printf("common: thread %d: lpvData=%lx\n",
    GetCurrentThreadId(), lpvData);

  Sleep(5000);
}

DWORD WINAPI ThreadFunc(VOID)
{
  LPVOID lpvData;

// Initialize the TLS index for this thread.

  lpvData = (LPVOID) LocalAlloc(LPTR, 256);
  if (! TlsSetValue(dwTlsIndex, lpvData))
    ErrorExit("TlsSetValue error");

  printf("thread %d: lpvData=%lx\n", GetCurrentThreadId(), lpvData);

  CommonFunc();

// Release the dynamic memory before the thread returns.

  lpvData = TlsGetValue(dwTlsIndex);
  if (lpvData != 0)
    LocalFree((HLOCAL) lpvData);

  return 0;
}

DWORD main(VOID)
{
  DWORD IDThread;
  HANDLE hThread[THREADCOUNT];
  int i;

// Allocate a TLS index.

  if ((dwTlsIndex = TlsAlloc()) == -1)
    ErrorExit("TlsAlloc failed");

// Create multiple threads.

  for (i = 0; i < THREADCOUNT; i++)
  {
    hThread[i] = CreateThread(NULL, // no security attributes
      0,                    // use default stack size
      (LPTHREAD_START_ROUTINE) ThreadFunc, // thread function
      NULL,                 // no thread function argument
      0,                  // use default creation flags
      &IDThread);           // returns thread identifier

  // Check the return value for success.
    if (hThread[i] == NULL)
```

```
        ErrorExit("CreateThread error\n");
    }

    for (i = 0; i < THREADCOUNT; i++)
      WaitForSingleObject(hThread[i], INFINITE);

    return 0;
}

VOID ErrorExit (LPTSTR lpszMessage)
{
    fprintf(stderr, "%s\n", lpszMessage);
    ExitProcess(0);
}
```

A *synchronization object* is an object whose handle can be specified in one of the wait functions to coordinate the execution of multiple threads. More than one process can have a handle to the same synchronization object, making interprocess synchronization possible.
The following object types are provided exclusively for synchronization.

Event	Notifies one or more waiting threads that an event has occurred. For more information, see Event Objects.
Mutex	Can be owned by only one thread at a time, enabling threads to coordinate mutually exclusive access to a shared resource. For more information, see Mutex Objects.
Semaphore	Maintains a count between zero and some maximum value, limiting the number of threads that are simultaneously accessing a shared resource. For more information, see Semaphore Objects.
Waitable timer	Notifies one or more waiting threads that a specified time has arrived. For more information, see Waitable Timer Objects.

Though available for other uses, the following objects can also be used for synchronization.

Change notification	Created by the _____ function, its state is set to signaled when a specified type of change occurs within a specified directory or directory tree. For more information, see File I/O.
Console input	Created when a console is created. The handle to console input is returned by the _____ function when CONIN$ is specified, or by the _____ function. Its state is set to signaled when there is unread input in the console's input buffer, and set to nonsignaled when the input buffer is empty. For more information about consoles, see Character-Mode Applications
Job	Created by calling the _____ function. The state of a job object is set to signaled when all its processes are terminated because the specified end-of-job time limit has been exceeded. For more information about job objects, see _____.
Process	Created by calling the _____ function. Its state is set to nonsignaled while the process is running, and set to signaled when the process terminates. For more information about processes, see Processes and Threads.
Thread	Created when a new thread is created by calling the , _____, or _____ function. Its state is set to nonsignaled while the thread is running, and set to signaled when the thread terminates. For more information about threads, see Processes and Threads.

In some circumstances, you can also use a file, named pipe, or communications device as a synchronization object; however, their use for this purpose is discouraged. Instead, use asynchronous I/O and wait on the event object set in the OVERLAPPED structure. It is safer to use the event object because of the confusion that can occur when multiple simultaneous overlapped operations are performed on the same file, named pipe, or communications device. In this situation, there is no way to know which operation caused the object's state to be signaled.
For additional information about I/O operations on files, named pipes, or communications, see Synchronization and Overlapped Input and Output.

3.1 EVENT OBJECTS

An *event object* is a synchronization object whose state can be explicitly set to signaled by use of the SetEvent or PulseEvent function. Following are the two types of event object.

Manual-reset event	An event object whose state remains signaled until it is explicitly reset to nonsignaled by the _____ function. While it is signaled, any number of waiting threads, or threads that subsequently specify the same event object in one of the wait functions, can be released.
Auto-reset event	An event object whose state remains signaled until a single waiting thread is released, at which time the system automatically sets the state to nonsignaled. If no threads are waiting, the event object's state remains signaled.

The event object is useful in sending a signal to a thread indicating that a particular event has occurred. For example, in overlapped input and output, the system sets a specified event object to the signaled state when the overlapped operation has been completed. A single thread can specify different event objects in several simultaneous overlapped operations, then use one of the multiple-object wait functions to wait for the state of any one of the event objects to be signaled.
A thread uses the CreateEvent function to create an event object. The creating thread specifies the initial state of the object and whether it is a manual-reset or auto-reset event object. The creating thread can also specify a name for the event object. Threads in other processes can open a handle to an existing event object by specifying its name in a call to the OpenEvent function. For additional information about names for mutex, event, semaphore, and timer objects, see Interprocess Synchronization.
A thread can use the PulseEvent function to set the state of an event object to signaled and then reset it to nonsignaled after releasing the appropriate number of waiting threads. For a manual-reset event object, all waiting threads are released. For an auto-reset event object, the function releases only a single waiting thread, even if multiple threads are waiting. If no threads are waiting, PulseEvent simply sets the state of the event object to nonsignaled and returns.

Using Event Objects

Applications use event objects in a number of situations to notify a waiting thread of the occurrence of an event. For example, overlapped I/O operations on files, named pipes, and communications devices use an event object to signal their completion. For more information about the use of event objects in overlapped I/O operations, see Synchronization and Overlapped Input and Output.
In the following example, an application uses event objects to prevent several threads from reading from a shared memory buffer while a master thread is writing to that buffer. First, the master thread uses the CreateEvent function to create a manual-reset event object. The master thread sets the event object to nonsignaled when it is writing to the buffer and then resets the object to signaled when it has finished writing. Then it creates several reader threads and an auto-reset event object for each thread. Each reader thread sets its event object to signaled when it is not reading from the buffer.

```
#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.
      }
   }
}
```

Before the master thread writes to the shared buffer, it uses the ResetEvent function to set the state of hGlobalWriteEvent (an application-defined global variable) to nonsignaled. This blocks the reader threads from starting a read operation. The master then uses the WaitForMultipleObjects function to wait for all reader threads to finish any current read operations. When WaitForMultipleObjects returns, the master thread can safely write to the buffer. After it has finished, it sets hGlobalWriteEvent and all the reader-thread events to signaled, enabling the reader threads to resume their read operations.

```
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.
    }
}
```

Before starting a read operation, each reader thread uses WaitForMultipleObjects to wait for the application-defined global variable hGlobalWriteEvent and its own read event to be signaled. When WaitForMultipleObjects returns, the reader thread's auto-reset event has been reset to nonsignaled. This blocks the master thread from writing to the buffer until the reader thread uses the SetEvent function to set the event's state back to signaled.

```
VOID ThreadFunction(LPVOID lpParam)
{
  DWORD dwWaitResult;
  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.
    }
}
```

3.2 MUTEX OBJECTS

A mutex object is a synchronization object whose state is set to signaled when it is not owned by any thread, and nonsignaled when it is owned. Only one thread at a time can own a mutex object, whose name comes from the fact that it is useful in coordinating mutually exclusive access to a shared resource. For example, to prevent two threads from writing to shared memory at the same time, each thread waits for ownership of a mutex object before executing the code that accesses the memory. After writing to the shared memory, the thread releases the mutex object.
A thread uses the CreateMutex function to create a mutex object. The creating thread can request immediate ownership of the mutex object and can also specify a name for the mutex object. Threads in other processes can open a handle to an existing mutex object by specifying its name in a call to the OpenMutex function. For additional information about names for mutex, event, semaphore, and timer objects, see Interprocess Synchronization.
Any thread with a handle to a mutex object can use one of the wait functions to request ownership of the mutex object. If the mutex object is owned by another thread, the wait function blocks the requesting thread until the owning thread releases the mutex object using the ReleaseMutex function. The return value of the wait function indicates whether the function returned for some reason other than the state of the mutex being set to signaled.
Threads that are waiting for ownership of a mutex are placed in a first in, first out (FIFO) queue. Therefore, the first thread to wait on the mutex will be the first to receive ownership of the mutex, regardless of thread priority. However, kernel-mode APCs and events that suspend a thread will cause the system to remove the thread from the queue. When the thread resumes its wait for the mutex, it is placed at the end of the queue.
After a thread obtains ownership of a mutex, it can specify the same mutex in repeated calls to the wait-functions without blocking its execution. This prevents a thread from deadlocking itself while waiting for a mutex that it already owns. To release its ownership under such circumstances, the thread must call ReleaseMutex once for each time that the mutex satisfied the conditions of a wait function.
If a thread terminates without releasing its ownership of a mutex object, the mutex object is considered to be abandoned. A waiting thread can acquire ownership of an abandoned mutex object, but the wait function's return value indicates that the mutex object is abandoned. It is best to assume that an abandoned mutex object indicates that an error has occurred and that any shared resource being protected by the mutex object is in an undefined state. If the thread proceeds as though the mutex object had not been abandoned, its "abandoned" flag is cleared when the thread releases its ownership. This restores normal behavior if a handle to the mutex object is subsequently specified in a wait function.

Using Mutex Objects

You can use a mutex object to protect a shared resource from simultaneous access by multiple threads or processes. Each thread must wait for ownership of the mutex before it can execute the code that accesses the shared resource. For example, if several threads share access to a database, the threads can use a mutex object to permit only one thread at a time to write to the database.
In the following example, a process uses the CreateMutex function to create a named mutex object or open a handle to an existing mutex object.

```
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.
}
```

When a thread of this process writes to the database, as in the next example, it first requests ownership of the mutex. If it gets ownership, the thread writes to the database and then releases its ownership.
The example uses structured exception-handling syntax to ensure that the thread properly releases the mutex object. The __finally block of code is executed no matter how the __try block terminates (unless the __try block includes a call to the TerminateThread function). This prevents the mutex object from being abandoned inadvertently.

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

   // Request ownership of mutex.

   dwWaitResult = WaitForSingleObject(
      hMutex,   // handle to 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;

      // Got ownership of the abandoned mutex object.
      case WAIT_ABANDONED:
         return FALSE;
   }

   return TRUE;
}
```

3.3 SEMAPHORE OBJECTS

A *semaphore object* is a synchronization object that maintains a count between zero and a specified maximum value. The count is decremented each time a thread completes a wait for the semaphore object and incremented each time a thread releases the semaphore. When the count reaches zero, no more threads can successfully wait for the semaphore object state to become signaled. The state of a semaphore is set to signaled when its count is greater than zero, and nonsignaled when its count is zero.
The semaphore object is useful in controlling a shared resource that can support a limited number of users. It acts as a gate that limits the number of threads sharing the resource to a specified maximum number. For example, an application might place a limit on the number of windows that it creates. It uses a semaphore with a maximum count equal to the window limit, decrementing the

count whenever a window is created and incrementing it whenever a window is closed. The application specifies the semaphore object in call to one of the wait functions before each window is created. When the count is zero — indicating that the window limit has been reached — the wait function blocks execution of the window-creation code.
A thread uses the CreateSemaphore function to create a semaphore object. The creating thread specifies the initial count and the maximum value of the count for the object. The initial count must be neither less than zero nor greater than the maximum value. The creating thread can also specify a name for the semaphore object. Threads in other processes can open a handle to an existing semaphore object by specifying its name in a call to the OpenSemaphore function. For additional information about names for mutex, event, semaphore, and timer objects, see Interprocess Synchronization.
Threads that are waiting for a semaphore are placed in a first in, first out (FIFO) queue. Therefore, the first thread to wait on the semaphore will be the first to successfully complete the wait, regardless of thread priority. However, kernel-mode APCs and events that suspend a thread from waiting will cause the system to remove the thread from the queue. When the thread resumes its wait for the semaphore, it is placed at the end of the queue.
Each time one of the wait functions returns because the state of a semaphore was set to signaled, the count of the semaphore is decreased by one. The ReleaseSemaphore function increases a semaphore's count by a specified amount. The count can never be less than zero or greater than the maximum value.
The initial count of a semaphore is typically set to the maximum value. The count is then decremented from that level as the protected resource is consumed. Alternatively, you can create a semaphore with an initial count of zero to block access to the protected resource while the application is being initialized. After initialization, you can use ReleaseSemaphore to increment the count to the maximum value.
A thread that owns a mutex object can wait repeatedly for the same mutex object to become signaled without its execution becoming blocked. A thread that waits repeatedly for the same semaphore object, however, decrements the semaphore's count each time a wait operation is completed; the thread is blocked when the count gets to zero. Similarly, only the thread that owns a mutex can successfully call the ReleaseMutex function, though any thread can use ReleaseSemaphore to increase the count of a semaphore object.
A thread can decrement a semaphore's count more than once by repeatedly specifying the same semaphore object in calls to any of the wait functions. However, calling one of the multiple-object wait functions with an array that contains multiple handles of the same semaphore does not result in multiple decrements.

Using Semaphore Objects

In the following example, a process uses a semaphore object to limit the number of windows it creates. First, it uses the CreateSemaphore function to create the semaphore and to specify initial and maximum counts.

```
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.
}
```

Before any thread of the process creates a new window, it uses the WaitForSingleObject function to determine whether the semaphore's current count permits the creation of additional windows. The

wait function's time-out parameter is set to zero, so the function returns immediately if the semaphore is nonsignaled.

```
DWORD dwWaitResult;

// Try to enter the semaphore gate.

dwWaitResult = WaitForSingleObject(
      hSemaphore,   // handle to 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;
}
```

When a thread closes a window, it uses the ReleaseSemaphore function to increment the semaphore's count.

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

if (!ReleaseSemaphore(
      hSemaphore,  // handle to semaphore
      1,           // increase count by one
      NULL) )      // not interested in previous count
{
   // Deal with the error.
}
```

3.4 WAITABLE TIMER OBJECTS

A *waitable timer object* is a synchronization object whose state is set to signaled when the specified due time arrives. There are two types of waitable timers that can be created: manual-reset and synchronization. A timer of either type can also be a periodic timer.

manual-reset timer	A timer whose state remains signaled until _____ is called to establish a new due time.
synchronization timer	A timer whose state remains signaled until a thread completes a wait operation on the timer object.
periodic timer	A timer that is reactivated each time the specified period expires, until the timer is reset or canceled. A periodic timer is either a periodic manual-reset timer or a periodic synchronization timer.

A thread uses the CreateWaitableTimer function to create a timer object. Specify TRUE for the *bManualReset* parameter to create a manual-reset timer and FALSE to create a synchronization timer. The creating thread can specify a name for the timer object in the *lpTimerName* parameter. Threads in other processes can open a handle to an existing timer by specifying its name in a call to the OpenWaitableTimer function. Any thread with a handle to a timer object can use one of the wait functions to wait for the timer state to be set to signaled.

- The thread calls the _____ function to activate the timer. Note the use of the following parameters for :

- Use the *lpDueTime* parameter to specify the time at which the timer is to be set to the signaled state. When a manual-reset timer is set to the signaled state, it remains in this state until establishes a new due time. When a synchronization timer is set to the signaled state, it remains in this state until a thread completes a wait operation on the timer object.

- Use the *lPeriod* parameter of the _____ function to specify the timer period. If the period is not zero, the timer is a periodic timer; it is reactivated each time the period expires, until the timer is reset or canceled. If the period is zero, the timer is not a periodic timer; it is signaled once and then deactivated.

A thread can use the CancelWaitableTimer function to set the timer to the inactive state. To reset the timer, call SetWaitableTimer. When you are finished with the timer object, call CloseHandle to close the handle to the timer object.

Using Waitable Timer Objects

The following example creates a timer that will be signaled after a 10 second delay. First, the code uses the CreateWaitableTimer function to create a waitable timer object. Then it uses the SetWaitableTimer function to set the timer. The code uses the WaitForSingleObject function to determine when the timer has been signaled.

```
#include <windows.h>
#include <stdio.h>

int main()
{
   HANDLE hTimer = NULL;
   LARGE_INTEGER liDueTime;

   liDueTime.QuadPart=-100000000;

   // Create a waitable timer.
   hTimer = CreateWaitableTimer(NULL, TRUE, "WaitableTimer");
   if (!hTimer)
   {
      printf("CreateWaitableTimer failed (%d)\n", GetLastError());
      return 1;
   }

   printf("Waiting for 10 seconds...\n");

   // Set a timer to wait for 10 seconds.
   if (!SetWaitableTimer(
      hTimer, &liDueTime, 0, NULL, NULL, 0))
   {
      printf("SetWaitableTimer failed (%d)\n", GetLastError());
      return 2;
   }

   // Wait for the timer.

   if (WaitForSingleObject(hTimer, INFINITE) != WAIT_OBJECT_0)
      printf("WaitForSingleObject failed (%d)\n", GetLastError());
   else printf("Timer was signaled.\n");

   return 0;
}
```

3.5 TIMER QUEUES

The CreateTimerQueue function creates a queue for timers. Timers in this queue, known as *timer-queue timers*, are lightweight objects that enable you to specify a callback function to be called when the specified due time arrives. The wait operation is performed by a thread in the thread pool.
To add a timer to the queue, call the CreateTimerQueueTimer function. To update a timer-queue timer, call the ChangeTimerQueueTimer function. You can specify a callback function to be executed by a worker thread from the thread pool when the timer expires.
To cancel a pending timer, call the DeleteTimerQueueTimer function. When you are finished with the queue of timers, call the DeleteTimerQueueEx function to delete the timer queue. Any pending timers in the queue are canceled and deleted.

Using Timer Queues

The following example creates a timer routine that will be executed by a timer-queue thread after a 10 second delay. First, the code uses the CreateEvent function to create an event object that is signaled when the timer-queue thread completes. Then it creates a timer queue and a timer-queue timer, using the CreateTimerQueue and CreateTimerQueueTimer functions, respectively. The code uses the WaitForSingleObject function to determine when the timer routine has completed. Finally, the code calls DeleteTimerQueue to clean up.
For more information on the timer routine, see WaitOrTimerCallback.

```
#include <windows.h>
#include <stdio.h>

HANDLE gDoneEvent;

VOID CALLBACK TimerRoutine(PVOID lpParam, BOOL TimerOrWaitFired)
{
    if (lpParam == NULL)
    {
        printf("TimerRoutine lpParam is NULL\n");
    }
    else
    {
        // lpParam points to the argument; in this case it is an int

        printf("Timer routine called. Parameter is %d.\n",
                *(int*)lpParam);
    }

    SetEvent(gDoneEvent);
}

int main()
{
    HANDLE hTimer = NULL;
    HANDLE hTimerQueue = NULL;
    int arg = 123;

    // Use an event object to track the TimerRoutine execution
    gDoneEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
    if (!gDoneEvent)
    {
        printf("CreateEvent failed (%d)\n", GetLastError());
        return 1;
    }

    // Create the timer queue.
    hTimerQueue = CreateTimerQueue();
    if (!hTimerQueue)
    {
        printf("CreateTimerQueue failed (%d)\n", GetLastError());
```

```
    return 2;
  }

  // Set a timer to call the timer routine in 10 seconds.
  if (!CreateTimerQueueTimer(
     &hTimer, hTimerQueue, TimerRoutine, &arg , 10000, 0, 0))
  {
     printf("CreateTimerQueueTimer failed (%d)\n", GetLastError());
     return 3;
  }

  // Do other useful work here

  printf("Call timer routine in 10 seconds...\n");

  // Wait for the timer-queue thread to complete using an event
  // object. The thread will signal the event at that time.

  if (WaitForSingleObject(gDoneEvent, INFINITE) != WAIT_OBJECT_0)
     printf("WaitForSingleObject failed (%d)\n", GetLastError());

  // Delete all timers in the timer queue.
  if (!DeleteTimerQueue(hTimerQueue))
     printf("DeleteTimerQueue failed (%d)\n", GetLastError());

  return 0;
}
```

3.6 CRITICAL SECTION OBJECTS

Critical section **objects provide synchronization similar to that provided by mutex objects, except that critical section objects can be used only by the threads of a single process. Event, mutex, and semaphore objects can also be used in a single-process application, but critical section objects provide a slightly faster, more efficient mechanism for mutual-exclusion synchronization. Like a mutex object, a critical section object can be owned by only one thread at a time, which makes it useful for protecting a shared resource from simultaneous access. There is no guarantee about the order in which threads will obtain ownership of the critical section, however, the system will be fair to all threads.**

The process is responsible for allocating the memory used by a critical section. Typically, this is done by simply declaring a variable of type CRITICAL_SECTION. Before the threads of the process can use it, initialize the critical section by using the InitializeCriticalSection **or** InitializeCriticalSectionAndSpinCount **function.**

A thread uses the EnterCriticalSection **or** TryEnterCriticalSection **function to request ownership of a critical section. It uses the** LeaveCriticalSection **function to release ownership of a critical section. If the critical section object is currently owned by another thread, EnterCriticalSection waits indefinitely for ownership. In contrast, when a mutex object is used for mutual exclusion, the wait functions accept a specified time-out interval. The TryEnterCriticalSection function attempts to enter a critical section without blocking the calling thread.**

Once a thread owns a critical section, it can make additional calls to EnterCriticalSection or TryEnterCriticalSection without blocking its execution. This prevents a thread from deadlocking itself while waiting for a critical section that it already owns. To release its ownership, the thread must call LeaveCriticalSection once for each time that it entered the critical section.

A thread uses the InitializeCriticalSectionAndSpinCount **or** SetCriticalSectionSpinCount **function to specify a spin count for the critical section object. On single-processor systems, the spin count is ignored and the critical section spin count is set to 0. On multiprocessor systems, if the critical section is unavailable, the calling thread will spin *dwSpinCount* times before performing a wait operation on a semaphore associated with the critical section. If the critical section becomes free during the spin operation, the calling thread avoids the wait operation.**

Any thread of the process can use the DeleteCriticalSection function to release the system resources that were allocated when the critical section object was initialized. After this function has been called, the critical section object can no longer be used for synchronization.
When a critical section object is owned, the only other threads affected are those waiting for ownership in a call to EnterCriticalSection. Threads that are not waiting are free to continue running.

Using Critical Section Objects

The following example shows how a thread initializes, enters, and leaves a critical section. As with the mutex example (see Using Mutex Objects), this example uses structured exception-handling syntax to ensure that the thread calls the LeaveCriticalSection function to release its ownership of the critical section object.

```
// Global variable
CRITICAL_SECTION CriticalSection;

void main()
{
   ...

   // Initialize the critical section one time only.
   InitializeCriticalSection(&CriticalSection);

   ...

   // Release resources used by the critical section object.
   DeleteCriticalSection(&CriticalSection)
}

DWORD WINAPI ThreadProc( LPVOID lpParameter )
{
   ...

   // Request ownership of the critical section.
   __try
   {
      EnterCriticalSection(&CriticalSection);

      // Access the shared resource.
   }
   __finally
   {
      // Release ownership of the critical section.
      LeaveCriticalSection(&CriticalSection);
   }

   ...

}
```

3.7 INTERLOCKED VARIABLE ACCESS

The interlocked functions provide a simple mechanism for synchronizing access to a variable that is shared by multiple threads. The threads of different processes can use this mechanism if the variable is in shared memory.
Simple reads and writes to properly-aligned 32-bit variables are atomic. In other words, when one thread is updating a 32-bit variable, you will not end up with only one portion of the variable updated; all 32 bits are updated in an atomic fashion. However, access is not guaranteed to be synchronized. If two threads are reading and writing from the same variable, you cannot determine if one thread will perform its read operation before the other performs its write operation.

Simple reads and writes to properly-aligned 64-bit variables are atomic on 64-bit Windows. Reads and writes to 64-bit values are not guaranteed to be atomic on 32-bit Windows. Reads and writes to variables of other sizes are not guaranteed to be atomic on any platform.
The interlocked functions should be used to perform complex operations in an atomic manner. The InterlockedIncrement and InterlockedDecrement functions combine the operations of incrementing or decrementing the variable and checking the resulting value. This atomic operation is useful in a multitasking operating system, in which the system can interrupt one thread's execution to grant a slice of processor time to another thread. Without such synchronization, one thread could increment a variable but be interrupted by the system before it can check the resulting value of the variable. A second thread could then increment the same variable. When the first thread receives its next time slice, it will check the value of the variable, which has now been incremented not once but twice. The interlocked variable-access functions protect against this kind of error.
The InterlockedExchangePointer function atomically exchanges the values of the specified variables.
The InterlockedExchangeAdd function combines two operations: adding two variables together and storing the result in one of the variables.
The InterlockedCompareExchangePointer function combines two operations: comparing two values and storing a third value in one of the variables, based on the outcome of the comparison.

3.8 WAIT FUNCTIONS

The *wait functions* to allow a thread to block its own execution. The wait functions do not return until the specified criteria have been met. The type of wait function determines the set of criteria used. When a wait function is called, it checks whether the wait criteria have been met. If the criteria have not been met, the calling thread enters the wait state. It uses no processor time while waiting for the criteria to be met.
There are four types of wait functions:

- single-object
- multiple-object
- alertable
- registered

The SignalObjectAndWait, WaitForSingleObject, and WaitForSingleObjectEx functions require a handle to one synchronization object. These functions return when one of the following occurs:

- The specified object is in the signaled state.
- The time-out interval elapses. The time-out interval can be set to INFINITE to specify that the wait will not time out.

The SignalObjectAndWait function enables the calling thread to atomically set the state of an object to signaled and wait for the state of another object to be set to signaled.

The WaitForMultipleObjects, WaitForMultipleObjectsEx, MsgWaitForMultipleObjects, and MsgWaitForMultipleObjectsEx functions enable the calling thread to specify an array containing one or more synchronization object handles. These functions return when one of the following occurs:

- The state of any one of the specified objects is set to signaled or the states of all objects have been set to signaled. You control whether one or all of the states will be used in the function call.
- The time-out interval elapses. The time-out interval can be set to INFINITE to specify that the wait will not time out.

The MsgWaitForMultipleObjects and MsgWaitForMultipleObjectsEx function allow you to specify input event objects in the object handle array. This is done when you specify the type of input to wait for in the thread's input queue.
For example, a thread could use MsgWaitForMultipleObjects to block its execution until the state of a specified object has been set to signaled and there is mouse input available in the thread's input queue. The thread can use the GetMessage or PeekMessage function to retrieve the input.
When waiting for the states of all objects to be set to signaled, these multiple-object functions do not modify the states of the specified objects until the states of all objects have been set signaled. For example, the state of a mutex object can be signaled, but the calling thread does not get ownership until the states of the other objects specified in the array have also been set to signaled. In the meantime, some other thread may get ownership of the mutex object, thereby setting its state to nonsignaled.

The MsgWaitForMultipleObjectsEx, SignalObjectAndWait, WaitForMultipleObjectsEx, and WaitForSingleObjectEx functions differ from the other wait functions in that they can optionally perform an *alertable wait operation*. In an alertable wait operation, the function can return when the specified conditions are met, but it can also return if the system queues an I/O completion routine or an APC for execution by the waiting thread. For more information about alertable wait operations and I/O completion routines, see Synchronization and Overlapped Input and Output. For more information about APCs, see Asynchronous Procedure Calls.

The RegisterWaitForSingleObject function differs from the other wait functions in that the wait operation is performed by a thread from the thread pool. When the specified conditions are met, the callback function is executed by a worker thread from the thread pool.
By default, a registered wait operation is a multiple-wait operation. The system resets the timer every time the event is signaled (or the time-out interval elapses) until you call the UnregisterWaitEx function to cancel the operation. To specify that a wait operation should be executed only once, set the *dwFlags* parameter of RegisterWaitForSingleObject to WT_EXECUTEONLYONCE.

The wait functions can modify the states of some types of synchronization objects. Modification occurs only for the object or objects whose signaled state caused the function to return. Wait functions can modify the states of synchronization objects as follows:

- The count of a semaphore object decreases by one, and the state of the semaphore is set to nonsignaled if its count is zero.
- The states of mutex, auto-reset event, and change-notification objects are set to nonsignaled.
- The state of a synchronization timer is set to nonsignaled.
- The states of manual-reset event, manual-reset timer, process, thread, and console input objects are not affected by a wait function.

You have to be careful when using the wait functions and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. If you have a thread that uses a wait function with no time-out interval, the system will deadlock. Two examples of code that indirectly creates windows are DDE and COM CoInitialize. Therefore, if you have a thread that creates windows, use MsgWaitForMultipleObjects or MsgWaitForMultipleObjectsEx, rather than the other wait functions.

Well-written applications include error-handling code that allows them to recover gracefully from unexpected errors. When an error occurs, the application may need to request user intervention, or it may be able to recover on its own. In extreme cases, the application may log the user off or shut down the system.

4.1 ABOUT ERROR HANDLING

The error handling functions enable you to receive and display error information for your application. For more information, see the following topics:

Each process has an associated error mode that indicates to the system how the application is going to respond to serious errors. Serious errors include disk failure, drive-not-ready errors, data misalignment, and unhandled exceptions. An application can let the system display a message box informing the user that an error has occurred, or it can handle the errors. To handle these errors without user intervention, use the SetErrorMode function. After calling SetErrorMode and specifying appropriate flags, the system will not display the corresponding error message boxes.

When an error occurs, most functions return an error code, usually zero, NULL, or –1. Many functions also set an internal error code called the *last-error code*. When a function succeeds, the last-error code is not reset. The error code is maintained separately for each running thread; an error in one thread does not overwrite the last-error code in another thread. An application can retrieve the last-error code by using the GetLastError function; the error code may tell more about what actually occurred to make the function fail.
The SetLastError function sets the error code for the current thread. The SetLastErrorEx function also allows the caller to set an error type indicating the severity of the error. These functions are intended primarily for dynamic-link libraries (DLL), so they can provide information to the calling application. The system defines a set of error codes that can be set as last-error codes or be returned by these functions. Error codes are 32-bit values (bit 31 is the most significant bit). Bit 29 is reserved for application-defined error codes; no system error code has this bit set. If you define error codes for your application, set this bit to indicate that the error code has been defined by an application and to ensure that the error codes do not conflict with any system-defined error codes. For more information, see WinError.h and System Error Codes.

To notify the user that some kind of error has occurred, many applications simply produce a sound by using the Beep or MessageBeep function or flash the window by using the FlashWindow or FlashWindowEx function. An application can also use these functions to call attention to an error and then display a message box or an error message containing details about the error.

Message tables are special string resources used when displaying error messages. They are declared in a resource file using the MESSAGETABLE resource-definition statement. To access the message strings, use the FormatMessage function.
The system provides a message table for the system error codes. To retrieve the string that corresponds to the error code, call FormatMessage with the FORMAT_MESSAGE_FROM_SYSTEM flag.
To provide a message table for your application, follow the instructions in About Message Text Files. To retrieve strings from your message table, call FormatMessage with the FORMAT_MESSAGE_FROM_HMODULE flag.

The FatalAppExit function displays a message box and terminates the application when the user closes the message box. This function should only be used as a last resort, because it may not free the memory or files owned by the application.

4.2 USING ERROR HANDLING

The following example uses FlashWindow to flash a window and MessageBeep to play the system exclamation sound.

```
FlashWindow(hwnd, TRUE); // invert the title bar
Sleep(500);              // wait a bit
FlashWindow(hwnd, TRUE); // invert again

// Play the system exclamation sound.

MessageBeep(MB_ICONEXCLAMATION);
```

When many system functions fail, they set the last-error code. If your application needs more details about an error, it can retrieve the last-error code using the GetLastError function.
The following example shows an error-handling function.

```
void error(LPSTR lpszFunction)
{
   CHAR szBuf[80];
   DWORD dw = GetLastError();

   sprintf(szBuf, "%s failed: GetLastError returned %u\n",
      lpszFunction, dw);

   MessageBox(NULL, szBuf, "Error", MB_OK);
   ExitProcess(dw);
}
```

4.3 ERROR HANDLING REFERENCE

4.4 BEEP

The Beep function generates simple tones on the speaker. The function is synchronous; it does not return control to its caller until the sound finishes.

```
BOOL Beep(
  DWORD dwFreq,      // sound frequency
  DWORD dwDuration   // sound duration
);
```

Parameters

dwFreq

[in] Specifies the frequency, in hertz, of the sound. This parameter must be in the range 37 through 32,767 (0x25 through 0x7FFF).

Windows 95/98/Me: The Beep function ignores this parameter.

dwDuration

[in] Specifies the duration, in milliseconds, of the sound.

Windows 95/98/Me: The Beep function ignores this parameter.

Return Values

If the function succeeds, the return value is nonzero.
If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Terminal Services: The beep is redirected to the client.
Windows 95/98/Me: On computers with a sound card, the function plays the default sound event. On computers without a sound card, the function plays the standard system beep.

Example Code

For an example, see Registering a Control Handler Function.

Requirements

Windows NT/2000/XP: Included in Windows NT 3.1 and later.
Windows 95/98/Me: Included in Windows 95 and later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Error Handling Overview, Error Handling Functions, MessageBeep

4.5 GETLASTERROR

The GetLastError function retrieves the calling thread's last-error code value. The last-error code is maintained on a per-thread basis. Multiple threads do not overwrite each other's last-error code.
DWORD GetLastError(VOID);

Parameters

This function has no parameters.

Return Values

The return value is the calling thread's last-error code value. Functions set this value by calling the SetLastError **function. The Return Value section of each reference page notes the conditions under which the function sets the last-error code.**
Windows 95/98/Me: Functions that are actually implemented in 16-bit code do not set the last-error code. You should ignore the last-error code when you call these functions. They include window management functions, GDI functions, and Multimedia functions.

Remarks

To obtain an error string for system error codes, use the FormatMessage **function. For a complete list of error codes, see System Error Codes.**
You should call the GetLastError function immediately when a function's return value indicates that such a call will return useful data. That is because some functions call SetLastError with a zero when they succeed, wiping out the error code set by the most recently failed function.
Most functions that set the thread's last error code value set it when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value error code such as zero, NULL, or –1. Some functions call SetLastError under conditions of success; those cases are noted in each function's reference page.
Error codes are 32-bit values (bit 31 is the most significant bit). Bit 29 is reserved for application-defined error codes; no system error code has this bit set. If you are defining an error code for your

application, set this bit to one. That indicates that the error code has been defined by an application, and ensures that your error code does not conflict with any error codes defined by the system.

Example Code

For an example, see Retrieving the Last-Error Code.

Requirements

Windows NT/2000/XP: Included in Windows NT 3.1 and later.

Windows 95/98/Me: Included in Windows 95 and later.

Header: Declared in Winbase.h; include Windows.h.

Library: Use Kernel32.lib.

See Also

Error Handling Overview, **Error Handling Functions**, FormatMessage, SetLastError, SetLastErrorEx

4.6 SETLASTERROR

The SetLastError function sets the last-error code for the calling thread.

```
VOID SetLastError(
  DWORD dwErrCode   // per-thread error code
);
```

Parameters

dwErrCode
[in] Specifies the last-error code for the thread.

Return Values

This function does not return a value.

Remarks

Error codes are 32-bit values (bit 31 is the most significant bit). Bit 29 is reserved for application-defined error codes; no system error code has this bit set. If you are defining an error code for your application, set this bit to indicate that the error code has been defined by your application and to ensure that your error code does not conflict with any system-defined error codes.
This function is intended primarily for dynamic-link libraries (DLL). Calling this function after an error occurs lets the DLL notify the calling application.
Most functions call SetLastError when they fail. Function failure is typically indicated by a return value error code such as zero, NULL, or –1. Some functions call SetLastError under conditions of success; those cases are noted in each function's reference topic.
Applications can retrieve the value saved by this function by using the GetLastError function. The use of GetLastError is optional; an application can call it to find out the specific reason for a function failure.
The last-error code is kept in thread local storage so that multiple threads do not overwrite each other's values.

Example Code

For an example, see Displaying the User for an Event.

Requirements

Windows NT/2000/XP: Included in Windows NT 3.1 and later.
Windows 95/98/Me: Included in Windows 95 and later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

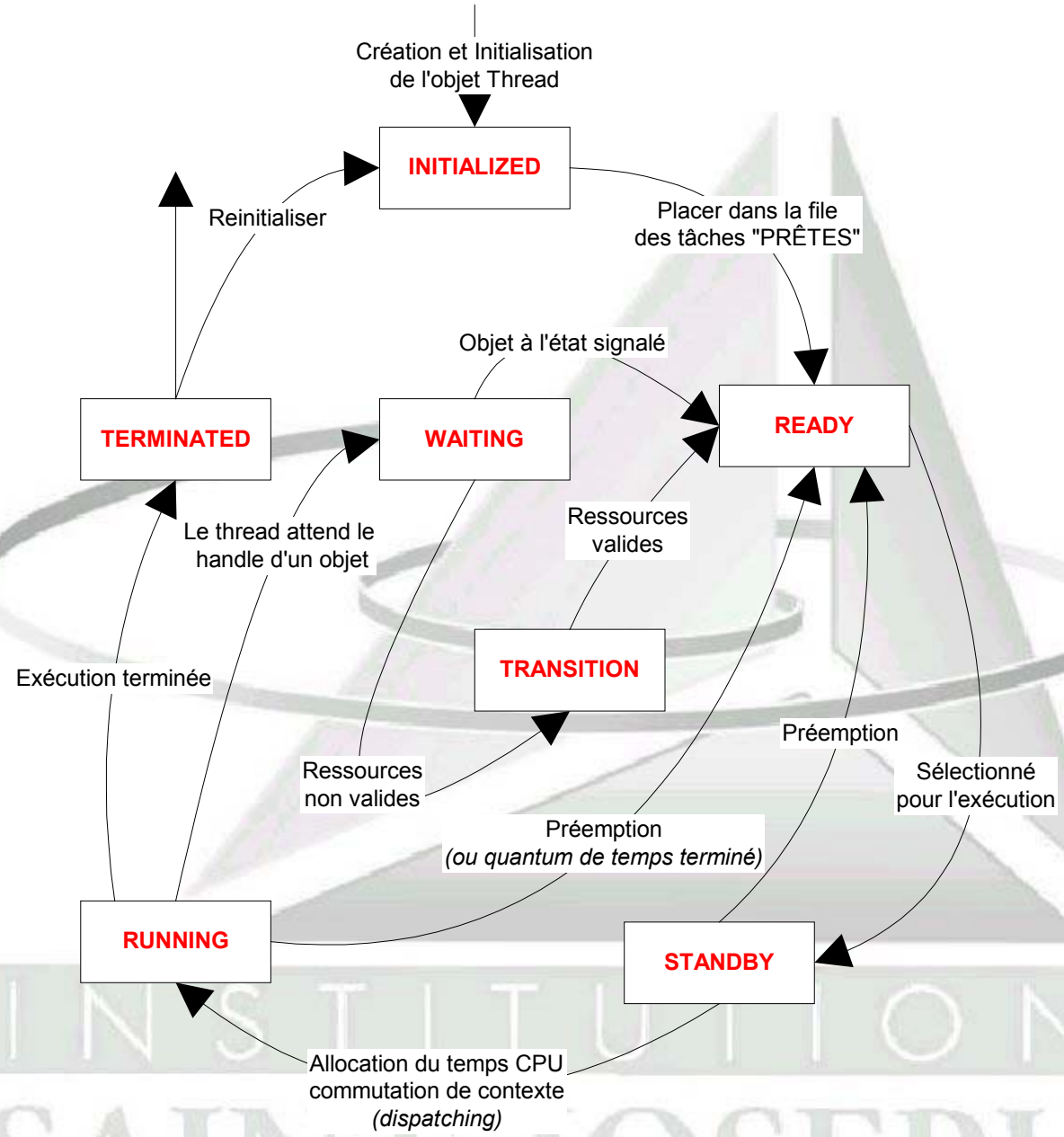
See Also

Error Handling Overview, **Error Handling Functions**, GetLastError, SetLastErrorEx

ANNEXE 1

INSIDE WINDOWS NT

Jeudi 17 octobre 2002



Les états d'un Thread

Institution Saint Joseph
413, avenue de Boufflers
54524 LAXOU-NANCY

Email : jalvarez@instit-st-jo.asso.fr
URL : www.instit-st-jo.asso.fr
m.a.j : ***17/10/2002 18:17***