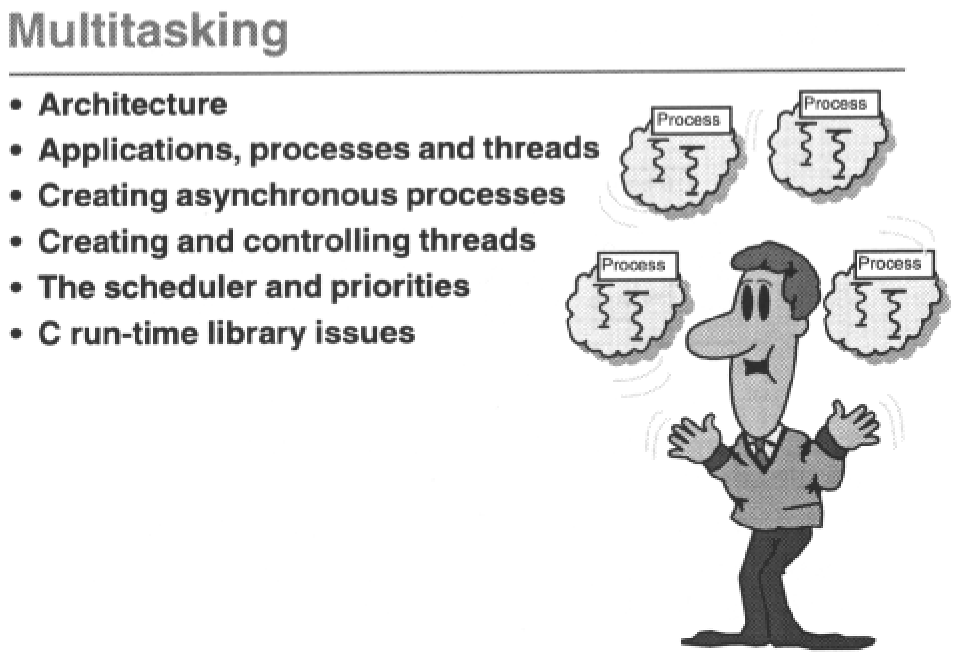


Multitasking

- Architecture
- Applications, processes and threads
- Creating asynchronous processes
- Creating and controlling threads
- The scheduler and priorities
- C run-time library issues

Process
Process
Process
Process

Multitasking is a key feature of Windows 9x and Windows NT. We look closely at two Win32 primitives; processes and threads. A process is an instance of an executing program. A thread is a unit of execution within a process.

In this chapter, we look at how to launch, control and close processes and threads from within an application. We study the operation of the scheduler, and examine the implications of writing multi-threaded applications.

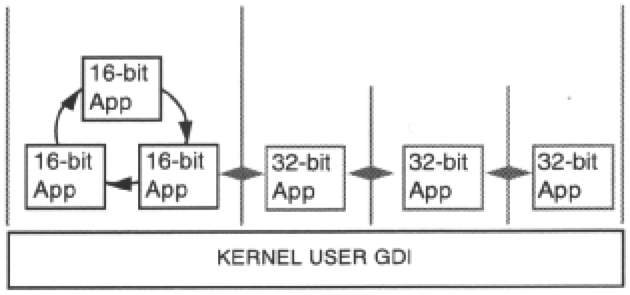
Objectives

When you have completed this chapter, you should be able to:

- Describe the architecture of Windows 95 multitasking.
- Describe the role of processes and threads.
- Write programs, which launch asynchronous processes.
- Create and control multiple threads within a process.
- Explain the operation of the scheduler, and the way that it distributes CPU cycles amongst threads according to their state and priority.

Use C run-time library support for multi-threaded applications.

16-Bit Tasking Vs 32-Bit Tasking



- **Non pre-emptive multitasking amongst 16-bit applications**
- **Pre-emptive multitasking for 32-bit applications**
- **A word of warning - badly-written 16-bit applications can hang Windows 95 user interface**

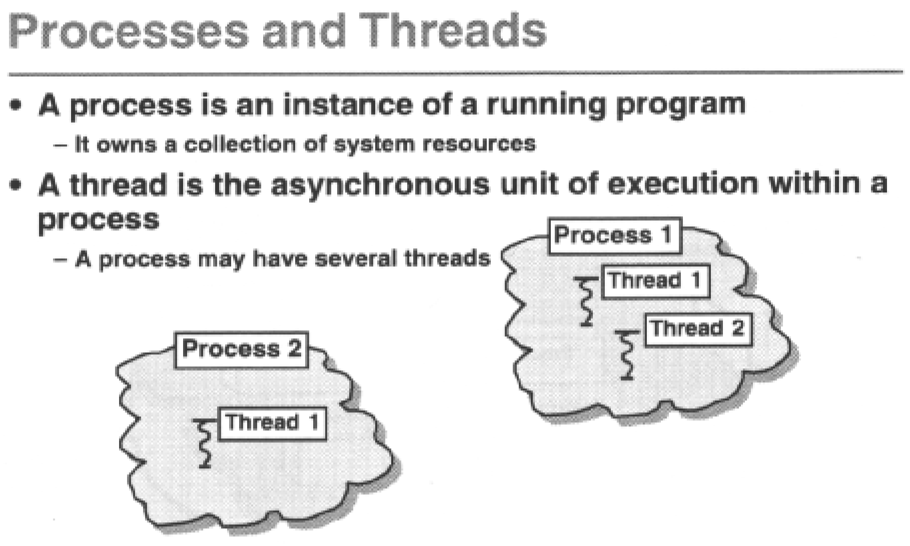
Existing 16-bit applications running under Windows 95 are non pre-emptively multitasked against each other as they were under Windows 3.x, but are collectively, pre-emptively multitasked against all other threads (some of which will include those belonging to 32-bit applications).

Under Windows 3.x, there was only ever one thread of execution, and applications were expected to be cooperative, something that caused problems when one application hogged the CPU. With the arrival of pre-emptive multitasking in Windows 95, supporting Windows 3.x applications posed problems.

Multitasking issues arise because Windows 3.x applications never expect to be pre-empted; the only instances where Windows took back control of the CPU were through control APIs such as GetMessage() or PeekMessage() This gave rise to a serious problem in the DLLs USER.EXE and GDI.EXE, which both serve 16-bit and 32-bit applications concurrently. The problem is this: code in these DLLs is non reentrant; that is, a thread cannot be in the midst of executing this code and be preempted by another thread that also wants to execute code in the same DLL. This is because of state data contained within the DLL, which cannot be restored to the original thread when it gets to run again. So, the designers chose to guard sensitive non-reentrant 16-bit code with a single mutex, which means only one thread can ever be executing that code at a time, effectively blocking all other threads wanting access until it has completed and released the mutex. When a 16-bit application gains access to the CPU to do some work, the mutex is immediately and transparently locked until the task yields through one of the control APIs mentioned earlier, at which point, the mutex is released. If a Win32 process needs to execute any of the 16-bit code mentioned, it must wait until the mutex is released. Should a badly behaved 16-bit application hang, then the user interface also hangs, as the user interface is sensitive 16-bit code and the application has the mutex locked!

This is not a problem for a system in which only 32-bit applications are running. When a 32-bit application makes a call that translates to sensitive 16-bit code, the mutex is only locked at the point of call and immediately released after the call, as opposed to locking the mutex for the duration the application is running.

It should be noted, however, that even though the user interface may be hung by a badly behaved 16-bit application, 32-bit application threads that do not make user interface calls will not suffer (as they never have to wait for exclusive ownership of the mutex) and will continue to execute as normal.



Processes and Threads

- **A process is an instance of a running program**
  - It owns a collection of system resources
- **A thread is the asynchronous unit of execution within a process**
  - A process may have several threads

Process 1
Thread 1
Thread 2
Process 2
Thread 1

Windows 95 allows a user to run several applications simultaneously, so that non-interactive tasks can be run in the background while the user continues with other work in the foreground. The user can also run multiple copies of the same program at the same time.

Less obviously, Windows 95’s facility for running several parts of a single application simultaneously allows the programmer to improve the performance of many applications by designing them so that individual programming tasks are carried out independently and in parallel (asynchronously), rather than in sequence.

The two key operating-system object types that have a major role to play in multitasking are the '*process’* and the *‘thread’*. A process is an instance of a running program. Each process owns its own resources (code, data and the like), which are located in its own private 4GB address space. Any such resources created by a process are destroyed when the process terminates. A thread is a unit of execution within a process. Each thread has a function to execute, and a CPU register state and stacks to enable the operating system to pre-emptively schedule them. Processes can contain several threads. Threads are used by the programmer to perform asynchronous subtasks that cooperate towards a common goal, their combined effect being the purpose of the process to which they belong.

The Virtual Machine Manager (VMM) component is responsible for determining how the operating system and applications use the processor. Multitasking under Windows 95 is a priority-based pre-emptive implementation.

The VMM *schedules* threads for execution and maintains a list of threads that are ready to execute, but are simply waiting their turn. When a currently executing thread reaches the end of its time slice, or *quantum* (20 ins.), the VMM selects the highest-priority thread from the ready list and performs a context switch to it. A context switch involves suspending the execution of the current thread, saving its state (e.g. the contents of the processor registers), restoring the state of the new thread and allowing the new thread to resume execution. In most cases, a thread will not run for its complete time quantum, either because it blocks on an API function, or because a higher-priority thread becomes ready. With the exception of the VMM and real-time threads, any thread can be pre-empted at any time.

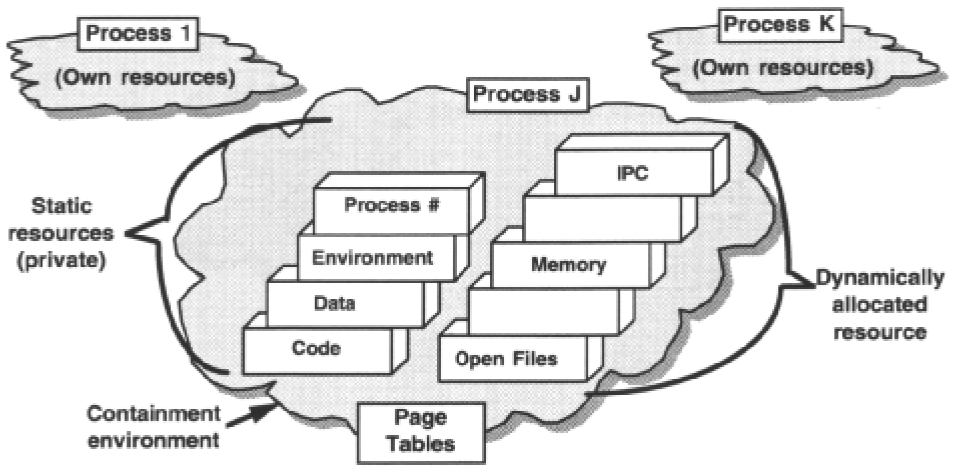
To facilitate the *'magic'* of multitasking, the VMM comprises two discrete software components: the primary scheduler and the time slice scheduler. These two entities work together to ensure that the highest-priority thread is always running and that multitasking is as *smooth* as possible.

It should be noted that only 32-bit applications can make calls to the Win32 API, and as such are the only type of application that can have multiple threads of execution. Existing 16-bit applications are inherently single threaded, but may of course be ported to become 32-bit, enabling them to make calls to the Win32 API.

Note, Windows 95 is a single CPU operating system, unlike Windows NT which is multiprocessor aware.

A Process

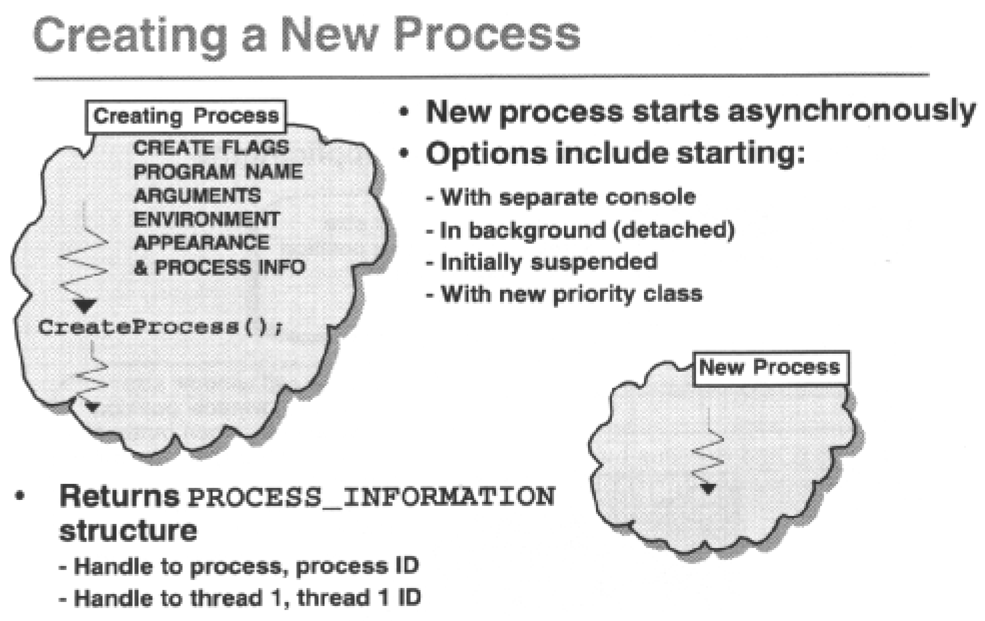
- **A resource-containment environment**



A process is a *'resource ownership and containment environment'* that defines a program that is loaded into memory and is executing. A process defines the context in which the execution occurs, and owns statically allocated resources, such as code, data and stack space, or dynamically allocated resources, such as files, memory and pipes.

Windows 95 maintains a unique set of *page tables* for each process that define a private 4GB logical, or virtual, address space and map it on to available physical memory. A process can only address memory that is referenced by an entry in its page table, so virtual memory owned by one process is invisible to all other processes, ensuring that processes are protected from each other.

The work of a process is done on its behalf by its threads of execution. Each process has at least one thread of execution. When a process is started up it has a single thread called the *'primary'* thread, but additional threads can be created. Because of policies imposed by the C run-time libraries and Win32, this primary thread has some attributes which differentiate it from subsequently created threads; it is, for instance, normally responsible for ending the process. However, Windows *95* does not consider the primary thread as anything special, and will only terminate the process when all threads have terminated. There is more on terminating processes and threads later.



Creating a New Process

Creating Process
CREATE FLAGS
PROGRAM NAME
ARGUMENTS
ENVIRONMENT
APPEARANCE
& PROCESS INFO
CreateProcess();

- **New process starts asynchronously**
- **Options include starting:**
  - With separate console
  - In background (detached)
  - Initially suspended
  - With new priority class

New Process

- **Returns** `PROCESS_INFORMATION` **structure**
  - Handle to process, process ID
  - Handle to thread 1, thread 1 ID

A thread in a process can start a new asynchronous process by using `CreateProcess()` . If successful, this causes the system to create a process object with a 4GB address space into which it maps an executable image. The system also creates a primary thread object, which is made known to the scheduler. The main thread of the new process won't necessarily be executing when Create Process() returns. It depends on a number of things; whether the new process was created with its primary thread initially suspended, the thread priorities of the calling thread and the primary thread of the new process. Do not make any assumptions. The primary thread of the new process first executes the C run-time start-up code, which then calls the entry point of the application, normally `main()` or `WinMain()`.

The initial execution environment of the new process is determined by the creating process in the parameters passed to CreateProcess() . To the system there is no special relationship between the two processes; for instance, the creating process can terminate without affecting the execution of the new process.

There are a number of parameters to `CreateProcess()`.The *'image name'* parameter is a null-terminated string defining a fully qualified path name for the executable image to be mapped into the address space of the new process. Win32 will not look down the directories specified by the PATH environment variable if it cannot find the file, or if it is not fully qualified.

If the image-name parameter is NULL, then the executable image is deduced from the first token (argv[0]) in the *'command line'* null-terminated string parameter, and Win32 will look down the PATH for the executable image file. Tokens in the command-line parameter are space separated, as you would type them in a normal command shell. This command line is available to the new process by calling `GetCommandLine()` . or by using the normal *argc /argv* mechanism in standard C programs. If the commandline parameter is NULL, then the new process has no command line.

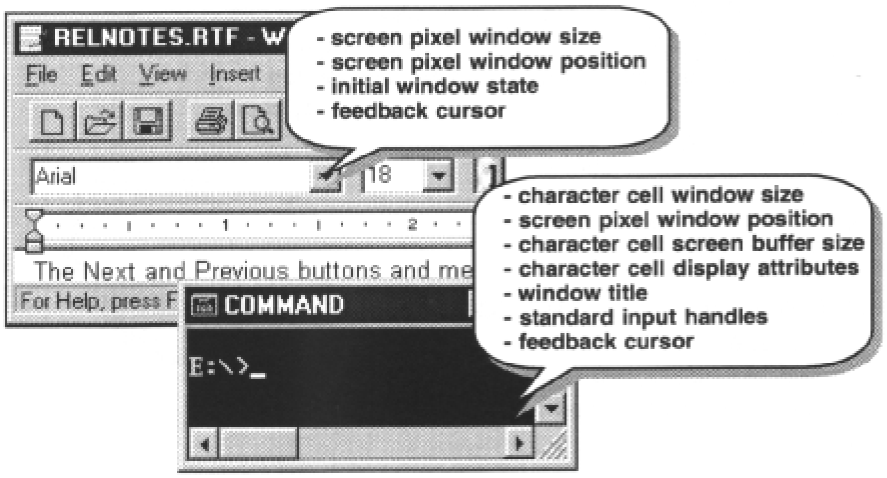
By default, a new process belongs to the same process group as the creating process. It can become the root process of a new process group (CREATE_NEW_PROCESS_GROUP).
The creating process can debug the new process only (DEBUG_ONLY_THI S_PROCESS), or the new process and any of it's descendants (DEBUG_PROCESS).

The new process can initially be started suspended (CREATE_SUSPENDED) so that it will not run until it is resumed by another thread by calling `ResumeThread()`.

The process priority class can be specified
(IDLE_PRIORITY_CLASS NORMAL_PRIORITY_CLASS,HIOH_PRIORITY CLASS, or REALTIME_PRIORITY_CLASS), which determines the range of priorities at which all threads within the new process will execute.
If no priority class is specified, and the creating process has class real time ‘high’ or ‘normal’, the priority class of the new process is normal. If no priority class is specified and the creating process has idle class then the priority class of the new process is idle.

Process Appearance

- **STARTUPINFO parameter of CreateProcess()**
- **Mostly display characteristics of applications**


RELNOTES.RTF - W
File Edit View Insert
Arial
18
The Next and Previous buttons and me
For Help, press F
COMMAND
E:\>_
- screen pixel window size
- screen pixel window position
- initial window state
- feedback cursor
- character cell window size
- screen pixel window position
- character cell screen buffer size
- character cell display attributes
- window title
- standard input handles
- feedback cursor

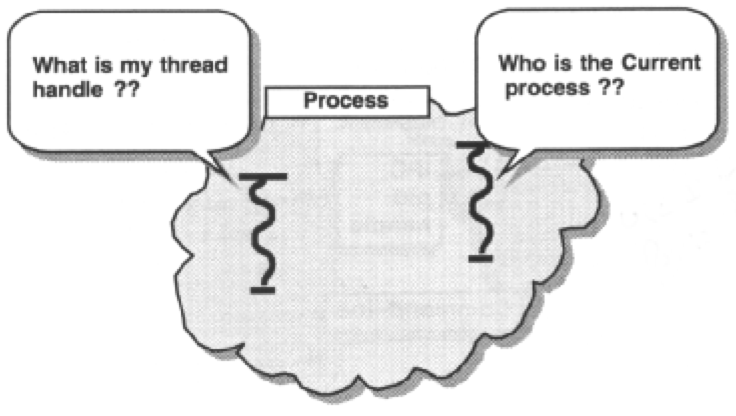
A creating process can specify display characteristics associated with a new process. `CreateProcess()` accepts a *'startup information'* pointer parameter which references a STARTUPINFO structure, whose members define these characteristics. The creating process can specify values for any subset of the characteristics; default values are used for those not specified. One useful application of the STARTUPINFO is to force a feedback cursor to be displayed during the initialization of the new process. Another is to facilitate standard handle redirection, which is discussed later.

Once started, a process can use `GetStartupinfo()` to retrieve the STARTUPINFO structure that was specified when it was created.

For Win32 GUI processes, the nCmdShow parameter is always set to SW_SHOWDEFAULT. This forces Win32 to use a field in the STARTUPINFO structure specified at process creation time, in order to decide how to show the application main window for the first time, e.g. hidden, minimized or maximized. If the `CreateWindow()` that created the main application window uses UW_USEDEFAULT parameters for initial size and position, fields in the same STARTUPINFO structure are used to decide the pixel width and height of the window, and the screen pixel location of the window.

Processes and Thread Handles

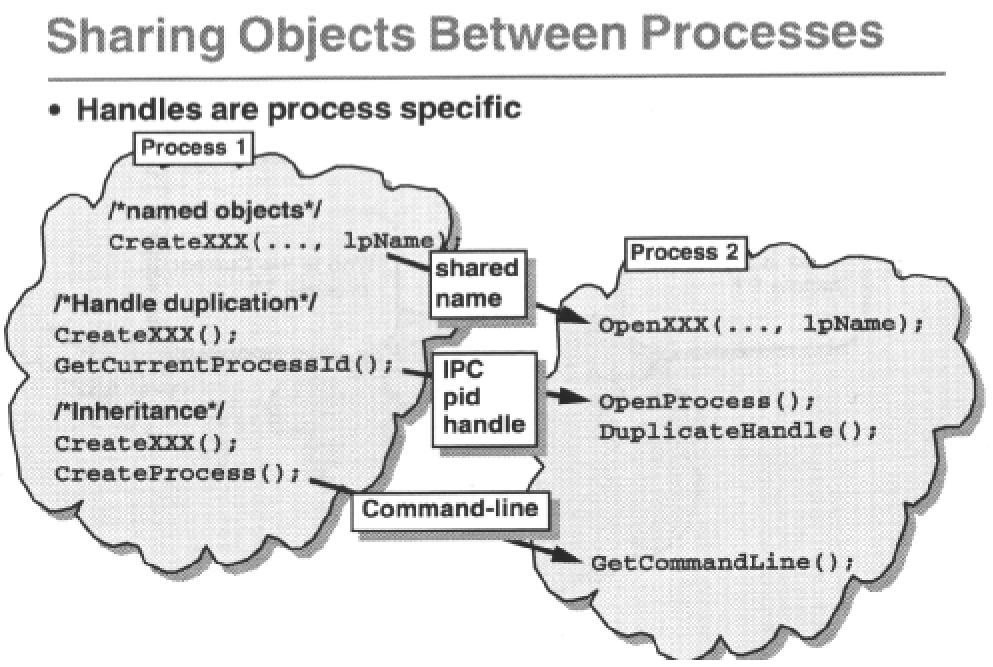
- Identify current thread - GetCurrentThread()
- Identify current process - GetCurrentProcess()



When a process obtains an open handle to a Windows 95 object through a Win32 API call, the reference count on the object is incremented. A handle so obtained is private to the process that obtained it, and references an entry in the handle table maintained for that process. It is meaningless to another process. The open handle is closed with a call to CloseHandle() which decrements the object-usage count and deletes the object from the system if the usage count goes to zero.

On successful process creation, CreateProcess() causes both a process and a thread object to be created with a reference count of I. However, it also fills in a *‘process information’* pointer parameter that references a PROCESS_INFORMATION structure supplied by the calling process. This contains the new process and primary thread details; the handle and ID of each. Both processes and threads are objects, accessed like any other by handle. The two new handles in the process information structure cause the usage count on the new process and thread objects to rise to 2. These new handles are private to the creating process, and are its reference to the new process and thread. If the creating process decides to close these handles, it will not cause the new process and thread objects to be unloaded from the system. The process ID and the thread ID, however, are globally unique identifiers that reference the new process and thread. These IDs are reused when threads and processes terminate, and are freed, so be careful about hanging onto them once they’re past their sell-by date. All the time you have an outstanding open handle to a process or thread you should be safe. Process and thread IDs are obtained by using `GetCurrentProcessId()` and `GetCurrentThread()`.

Calling `GetCurrentThread()` obtains a handle to the calling thread and GetCurrentProcess() obtains a handle to the calling threads process. These are often used for self identification. Be careful; these are pseudo handles, and obtaining them does not increment the usage count for the current process or thread object. They are ambiguously defined; they equate to system-defined values of -l and -2, which mean ‘the current process’ and ‘the current thread’. Their use is restricted. For instance, you could not pass a handle returned from `GetCurrentThread()` in one process to a thread in another process to duplicate and use, because it would reference the receiving thread. To obtain an unambiguous current process or thread handle, the pseudo handle would need to be duplicated.



Sharing Objects Between Processes

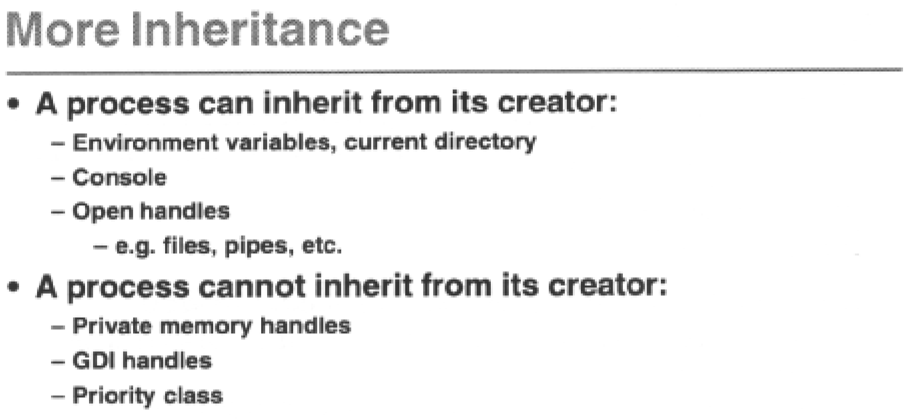
- **Handles are process specific**

So, object handles are process specific. How do we share objects?

One way is to share objects by name; one process creates the object by name and obtains a process-specific handle, and another opens it using the same name to obtain its process-specific handle. Beware; the names of event objects, semaphore objects, mutex objects, and file-mapping objects share the same name space. If a specified name matches the name of an existing object of a different type, an error occurs. Names cannot contain the null character. A NULL name specifies an unnamed object, which is then normally used privately within the process in which it is created.

Whenever an open handle is requested from a Win32 API, there will be an *'inherit handles'* parameter that specifies whether the resulting handle is to be inherited by new processes created by the calling process. Any open handles so marked as inheritable can be inherited by a new process, if the *'inherit handles'* parameter of CreateProcess() is TRUE. Such inherited handles are newly opened handles, even though they have the same value in the newly created inheriting process, and the same access rights. Thus, the reference count on an object referenced by an inherited handle does increase and the handle has its own separate state, so the creating processes and the newly created inheriting process can independently reference the same object.

The last mechanism relies on one process to pass one of its process-specific handles and its process ID to some other process via an IPC mechanism, to enable the handle to be duplicated to a new process-specific handle for the receiving process, using `DuplicateHandle()` Handles are more commonly duplicated in one process than across processes. We will see this later with standard handle redirection.



More Inheritance

- A process can inherit from its creator:
  - Environment variables, current directory
  - Console
  - Open handles
    - e.g. files, pipes, etc.
- A process cannot inherit from its creator:
  - Private memory handles
  - GDI handles
  - Priority class

If the *'environment'* memory block parameter to CreateProcess() is NULL, a process inherits the environment of its creating process. If the creating process passes a pointer to an environment memory block, which is a null-terminated block of null-terminated strings, then it can pass a brand new environment. Any process can obtain its environment variables by calling `GetEnvironmentStrings()`, or more specifically `GetEnvironmentVariable()`.

If the *'current directory'* string parameter to `CreateProcess()` is NULL, a process inherits the current directory of its creating process. Otherwise, the creating process can set the current directory of the new process with a fully qualified path name, including drive letter.

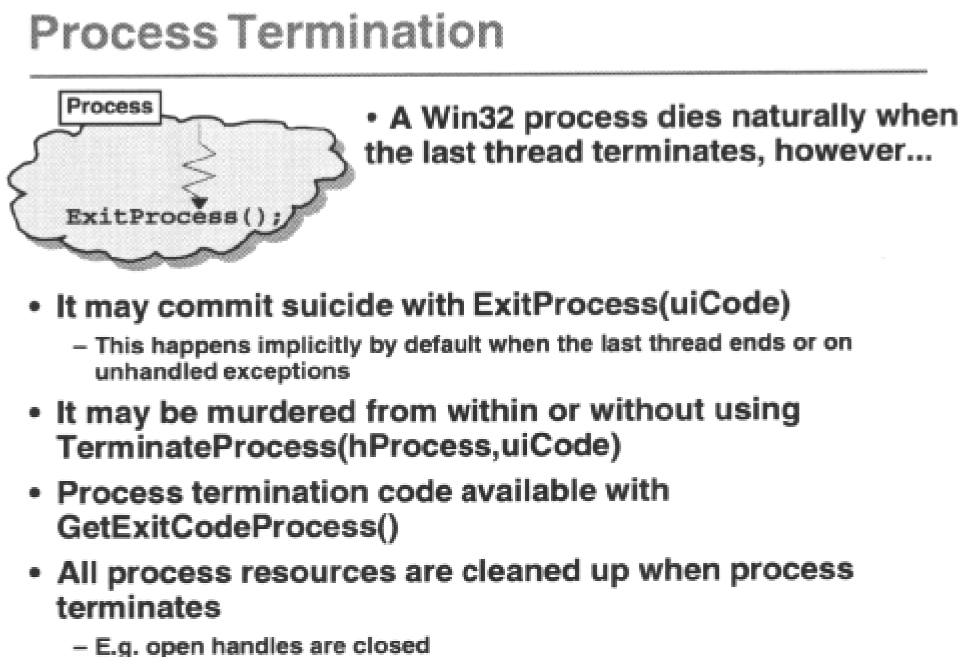
As discussed earlier, a creating process can also allow the new process to inherit any of its inheritable open handles by setting the *'inherit handles'* parameter of `CreateProcess()` to TRUE.
If a process has an inheritable handle it doesn't want to be inherited, or *vice versa,* then it can create a duplicate with `DuplicateHandle()` and close the original with `CloseHandle()`. That new process does not inherit any inheritable open handles obtained by a creating process after a new process is started.

Here is an example of starting a process with a command line, a new environment and a new current directory:

```
STARTUPINFO si ={sizeof (si) } ; /*set this up with defaults*/
PROCESS_TNEORMATION p1;       /*system fills this in*/
char *env   = “paths c:\Q include=\inc\Olibs\lib\O\O”;
BOOL blnheritHandles    TRUE;

BOOL bSuccess = CreateProcess(NULL, “newprocess -a -d” NULL, NULL, FALSE,
NORMAL_PRIORITY_CLASS, any, “c:\”, &si, &pi);
```

Processes cannot inherit scheduler priorities, private memory handles or GDI object handles.



Process Termination

Process
ExitProcess();

- A Win32 process dies naturally when the last thread terminates, however...
- It may commit suicide with ExitProcess(uiCode)
  - This happens implicitly by default when the last thread ends or on unhandled exceptions
- It may be murdered from within or without using TerminateProcess(hProcess,uiCode)
- Process termination code available with GetExitCodeProcess()
- All process resources are cleaned up when process terminates
  - E.g. open handles are closed

A Win32 process will not terminate until the last thread terminates; the process remains in the system until all threads in the process are terminated, and all handles to the process and its threads have been closed by calling `CloseHandle()`.So natural process termination is by the last thread terminating naturally. In this case the process return code is the return code of the last thread.

However, a thread will also terminate under two other conditions:

Suicide. Any thread in a process can make an explicit call to `ExitProcess()` . The process return value is the parameter supplied to `ExitProcess()`.`ExitProcess()` will never return, and will terminate all running threads in the process abruptly.

The primary thread is normally responsible for ending the process. The C run-time clean-up code ensures that `ExitProcess`() is called on termination of the primary thread of execution (the end of the `main()` or `WinMain()` function) by a `return`, or `exit()` or hitting the ending }. The process return value is the return value from the primary thread.

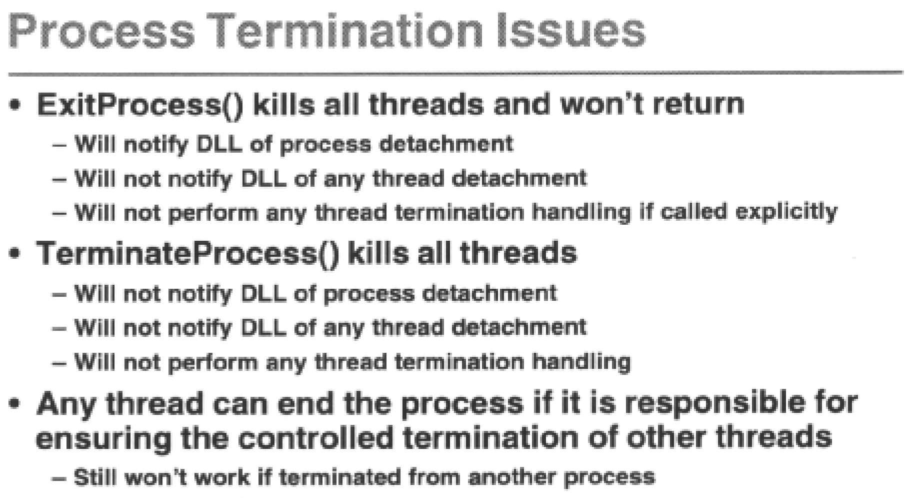
However, `if` the primary thread calls `ExitThread()` the process will not terminate until the last thread dies or unless another thread calls `ExitProcess()`.

A critical-error abort or an unintercepted execution fault, such as a general protection violation or a page fault, will also cause `ExitProcess()` to be called.
The return value is the value of any exception code. Of course, you can choose to handle exceptions yourself and decide whether terminating the process is the correct course of action, or whether the process can recover and carry on.

Remember to check the system-defined exception codes to ensure that your process return codes do not conflict. Seethe documentation on `GetExceptionInformation()` for more details.

Murder. One process may terminate another if it has the correct security access, by calling `TerminateProcess`() .The return value is the parameter supplied to `TerminateProcess()`. This should only be used under extreme circumstances. The reasons are discussed later.

When a process terminates, any open handles are closed automatically. When the process terminates, its termination status changes from STILL_ACTIVE to the termination code of the process, and the process object is set to a signaled state to satisfy any threads waiting on it to end. The process return value is available to another process by using `GetExitCodeProcess`() ; a return of STILL_ACTIVE means it hasn't terminated yet.



Process Termination Issues

- ExitProcess() kills all threads and won't return
  - Will notify DLL of process detachment
  - Will not notify DLL of any thread detachment
  - Will not perform any thread termination handling if called explicitly
- TerminateProcess() kills all threads
  - Will not notify DLL of process detachment
  - Will not notify DLL of any thread detachment
  - Will not perform any thread termination handling
- Any thread can end the process if it is responsible for ensuring the controlled termination of other threads
  - Still won't work if terminated from another process

Care should be taken in multi-threaded processes concerning the way in which the process is closed. If the primary thread, or any other thread, explicitly calls `ExitProcess()` or causes it to be implicitly called by the C run time, then all other threads will terminate abruptly. There are two problems here.

Processes use DLLs that execute in the context of the process. DLLs provide an entry/exit function, which notifies the DLL when a process is attaching to/detaching from the DLL. The entry point *process attach'* notification occurs when the process starts up. The entry point *'process detach'* notification occurs when ExitProcess() is called. The entry point *'thread attach'* notification occurs when a subsequent thread within the process starts up. The entry point *'thread detach'* notification occurs when `ExitThread()` is called. `ExitThread()` is discussed later.

The calling of this DLL function will normally perform DLL initialization and clean up functionality, so it is quite important that it is called when it expects to be! For instance, the DLL may be writing DLL state changes to disk on detach notification.

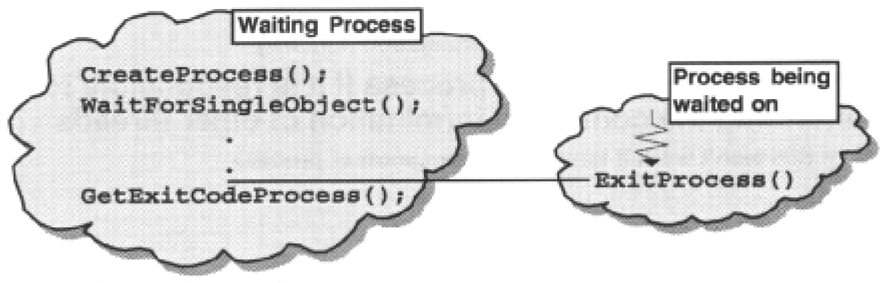
Calling `ExitProcess()` will trigger the DLL process detach notification, but not any thread detach notifications. Calling `TerminateProcess()` will not trigger the DLL process detach notification, or any thread detach notifications.

Any thread may choose to install a *'termination handler'* to perform clean-up operations when a thread function terminates. Because `ExitProcess()` never returns, and no more code is executed for the process in any of its threads, explicitly calling `ExitProcess()` will not cause any thread-termination handlers to be called. Also, because `ExitThread()` never returns and no more code is executed for that thread, explicitly calling `ExitThread()` will not cause the termination handler for that thread to be called. However, ending a thread with return, `exit()` or by hitting the thread function will work.

Care must be taken with process termination. One approach is to ensure that one thread is responsible for causing the termination of all other threads, waiting for them all to terminate in a controlled fashion before it terminates. Even this won't work if another process calls `TerminateProcess()` on you!

Waiting on a Process to Terminate

- **CreateProcess() returns immediately**
  - **With process details**
- **WaitForSingleObject() waits for process to terminate**
- **GetExitCodeProcess() returns termination status**



When designing an application solution it is very often desirable to adopt a tools approach, and split the solution up into many cooperating processes. The pros and cons of using processes versus threads is addressed later. It maybe necessary to synchronize the operation of cooperating processes, especially to know when they have terminated. Remember there is no process structure in Win32, and the termination of a process does not affect the process that started it.

Traditionally, in Windows 3.x, the only way to start another application and be notified of its termination was to use `WinExec()` and then install a hook to watch for tell-tale closing down messages. Waiting for a process to terminate in Win32 is much easier. A process is a '*synchronization*' object; unsignalled during its lifetime and signaled when its last thread terminates. Here is an example of waiting on a detached process:

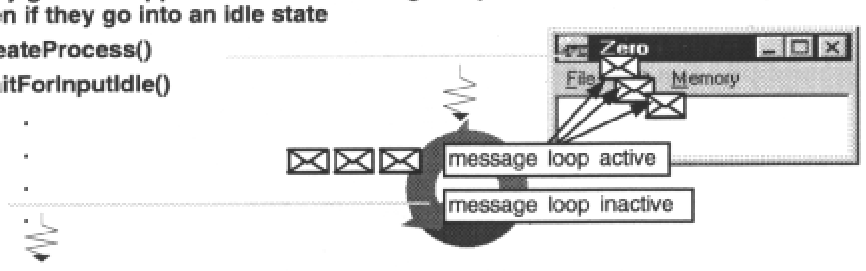
```
STARTUPINFO Si {sizeof (si) };      /*setthisupwithdefaults*/
PROCESS_INFORMATION pi;             /*system fills this in*/
BOOL bSuccess = CreateProcess( NULL,/*noimagename*/
      "newprocess",                 /*command line*/
      NULL,                         /*no process security*/
      NULL,                         /* no thread security*/
      FALSE,                        /*don't inherit handles*/
      IDLE_PRIORITY_CLASSIDETACHED_PROCESS,/*idle priority background task */
      NULL,                         /*inherit environment*/
      NULL,                         /*inherit current directory*/
      &si,                          /*initial appearance*/
      &pi ) ;                       /*processinformation*/

if ( bSuccess )
{
if ( WaitForSingleObject( pi.LProcess, INFINITE ) WAIT OBJECT ) {
      bSuccess = GetExitCodeProcess( pi.hProcess, &dwExicCode );
      CloseHandle( pi.hThread. );
      CloseHandle( pi.hProcess );
  }
}
```

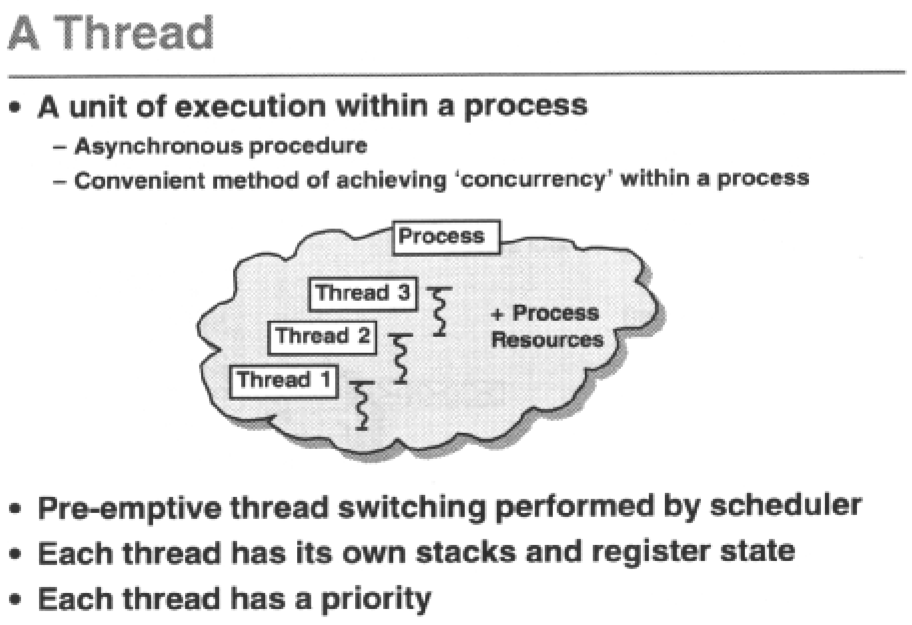
When the process terminates, its termination status changes from STILL_ACTIVE `to the` termination code of the process, and the process object is set to a signalled state to satisfy any threads waiting on it `to` end. The process return value is available to another process by using `GetExitCodeProcess()`;a return of STILL_ACTIVE means it hasn't terminated yet.

Waiting on a 'Windowed' Process

- **To finish initialisation, or processing specific input**
  - WaitForInputIdle() returns when process waiting for user input with no input pending
  - Only good for applications with message loop in main thread, and only then if they go into an idle state
  - CreateProcess()
  - WaitForInputIdle()



- **At other times, use IPC synchronisation objects**
  - Events, semaphores



A Thread

- **A unit of execution within a process**
  - Asynchronous procedure
  - Convenient method of achieving 'concurrency' within a process

Process
Thread 3
Thread 2
Thread 1
+ Process Resources

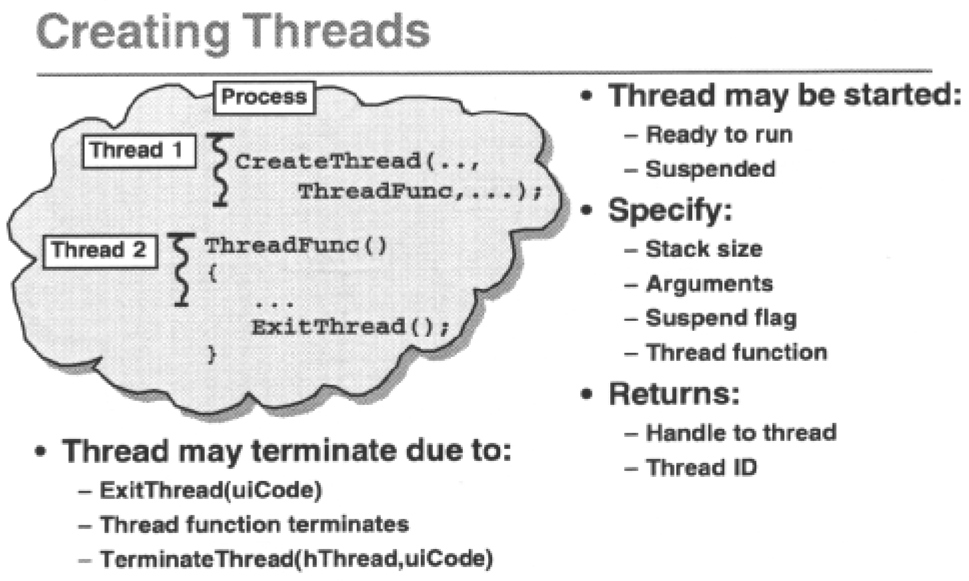
- **Pre-emptive thread switching performed by scheduler**
- **Each thread has its own stacks and register state**
- **Each thread has a priority**

A thread is a dispatchable unit of execution in Windows 9x and Windows NT; it is the basic entity to which the operating system allocates CPU time. Threads provide a mechanism for carrying out several programming tasks simultaneously within a process. Each thread has a priority, and the operating system scheduler carries out switching between threads pre-emptively. Each thread runs independently and maintains a set of data structures for saving its context while waiting to be scheduled for processing time. These structures include its own set of machine registers, its own kernel stack, a thread environment block, and a user stack in the address space of its process. A typical Win32 application will consist of many threads.

From a programming language perspective, a thread may be considered as an asynchronous function. When a normal C function is called, the flow of execution transfers from the calling environment to the function, and then returns when the function ends. But when one thread starts another, the original continues to run while the new thread executes concurrently and independently. There may thus be many copies of a function running within a process as different threads. Threads of the same process can execute any part of the program's code, including a part being executed by another thread.

The operating system divides the available CPU time among the threads that need it. This pre-emptive multitasking means that the system allocates small slices of CPU time among competing threads. The currently executing thread is suspended when its *'time quantum'* elapses, or if it gets pre-empted by a more important thread, allowing another thread to run. When the system switches from one thread to another it saves the context of the suspended thread and restores the saved context of the thread to be run. Although it appears that multiple threads are executing at the same time, in fact they are not, as there will only be one CPU.

Creating new threads is 'cheap and fast' in comparison to processes, in terms of system overhead and time to initialize internal data structures. Threads have access to the virtual address space of the process to which they belong. Its threads share global resources of the process and so they communicate simply, but they are not protected from each other. For this reason, and because threads run asynchronously, the programmer often needs to serialize access to data using synchronization objects such as critical sections, events, mutexes and semaphores, to coordinate their activities.



Creating Threads

Process

Thread 1

```
CreateThread(..,
    ThreadFunc,...);
```

Thread 2

```
ThreadFunc()
{
    ...
    ExitThread();
}
```

- Thread may be started:
  - Ready to run
  - Suspended
- Specify:
  - Stack size
  - Arguments
  - Suspend flag
  - Thread function
- Returns:
  - Handle to thread
  - Thread ID
- Thread may terminate due to:
  - ExitThread(uiCode)
  - Thread function terminates
  - TerminateThread(hThread,uiCode)

Every process has a primary thread of execution, started when the process is created. Any thread can create both processes and threads. Threads are created with fewer overheads than processes, although they have less protection between them. Applications are more likely to multitask by using multiple threads in the same process rather than multiple processes, unless threads with protected, private address spaces are required.

Asynchronous tasks can be carried out on behalf of a process by creating new threads of execution, using `CreateThread()` ,specifying:

- The size of the stack allocated from the process address space when the thread is created, and freed when the thread terminates; if 0 is specified, the default application stack size is used, i.e. that size used by the primary thread.
- The address of the thread function code to execute. and optionally:
- The arguments passed to the thread function.
- If not specified, the thread handle cannot be inherited.

The system passes back a thread handle and ID. The size of the stack for a thread depends on whether it is a GUI or console application, and on program behavior, e.g. reentrancy.

No assumptions about the sequence and timing of the thread starting execution should be made. `CreateThread()` returns asynchronously to inform us that a thread object has been created. This thread is waiting, along with all others in the system, to be executed by the scheduler, according to its priority; it may or may not already have been executed by the time `CreateThread()` returns.

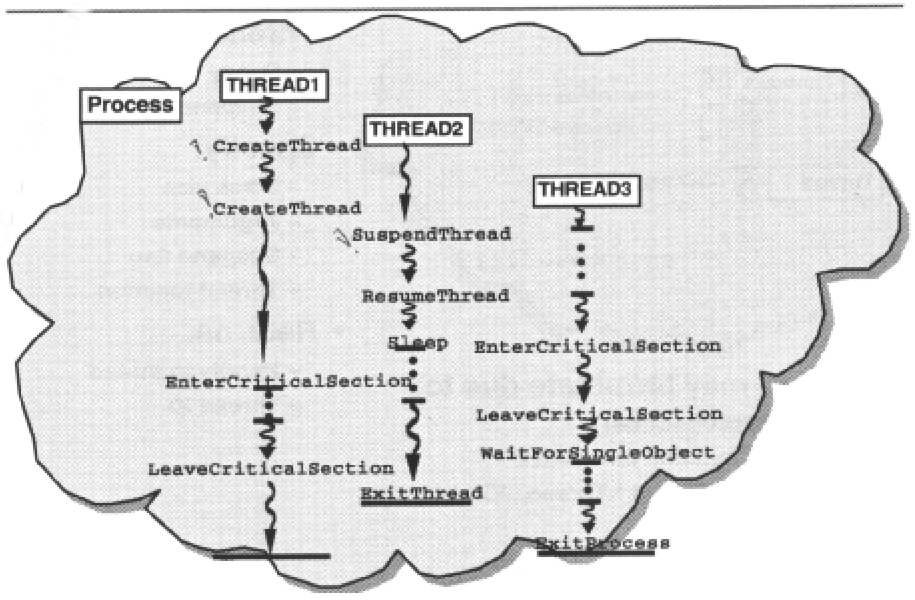
This is an important issue when passing pointer arguments to a new thread; if the memory they reference exists on the stack, will arguments still be valid when the thread executes?

`A thread executes until:`

- It calls `ExitThread()` or the thread function returns; the thread-return value is the parameter supplied to `ExitThread()` or the return from the thread function. If the process main thread calls `ExitThread()` , the process will not terminate unless another thread calls `ExitProcess()`
- Another thread terminates it using `TerminateThread`() the thread-return value is the parameter supplied to `TerminateThread()`.

The thread-return code is available to another thread by using `GetExitCodeThread()` . Note that a created thread remains in the system until the thread is terminated and all handles to the threads have been closed by calling `CloseHandle()` . Until then, the thread object is set to a signaled state to satisfy all waits on it. No clean up of resources takes place, except for the thread stack being deallocated.

Controlling Threads



One thread can suspend or resume the execution of another by calling SuspendThread() or ResumeThread().) . A suspended thread is not given CPU time by the scheduler. A thread can suspend itself, but it had better make sure that another thread is going to resume it, because it can't resume itself! You can never really be sure where a suspended thread will suspend execution, so it is not a good synchronization technique. A thread can be created suspended, to be resumed later by the creating thread. A thread can use Sleep() to suspend its execution for a specified interval. Here is an example of creating a suspended thread:

```
DWORD dwID;
HANDLE hThread CreateThread(  NULL,             /*nosecurity*/
                              p                 /* default stack*/
                              finThread,        /*thread function*/
                              0,                /*arguments*/
                              CREATE_SUSPENDED, /*initially suspended*/
                              &dwID )           /*threadlD*/
if ( hThread)
{
      SetThreadPriority( hThread, THREAD_PRIORITY LOWEST);
      ResumeThread( hThread );
}
```

Some Win32 objects are *'synchronization objects'*. A thread waiting on such an object is blocked if the object is not signaled. Blocked threads, like suspended threads, are not scheduled, but it is possible to specify the exact place at which the thread will wait.
A thread is a synchronization object; unsignalled during its lifetime and signalled when it terminates. One thread can wait for another to terminate.

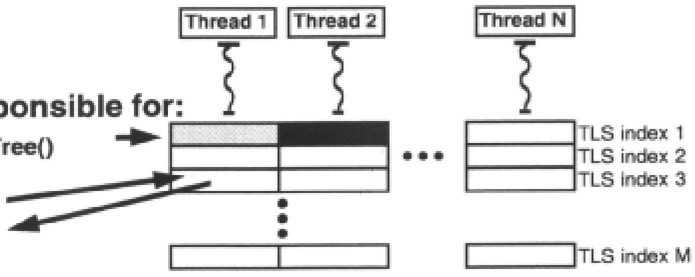
For threads of a single process that are all sharing the same resources, it is necessary to control access to these resources and to coordinate and synchronize thread actions.

To do this, threads typically use synchronization objects, like *‘mutexes’, ‘semaphores’* and *‘events’,* to signal each other in order to synchronize their activities.
These objects can be *‘waited on’,* until they attain a signaled state, using APIs like `WaitForSingleObject()`, `WaitForMultipleObjects()` .Threads can also enter ’criticalsections’ of code, where all other threads in the process are blocked if they attempt to enter the same critical section, until the critical section is left.
A thread calls `EnterCriticalSection()` to request ownership of the critical section. If another thread already owns it, the requesting thread blocks until it is released.
When the critical section is unowned, the calling thread is granted ownership and can access it. The thread then uses `LeaveCriticalSection()` to relinquish ownership.

Thread Local Storage

- **Allows different threads to have different thread-specific values for the same logical data item**
- **The __declspec(thread) extended attribute syntax in the MS 32-bit C/C++ compiler is equivalent**

- **API**
- **Thread 1 responsible for:**
  - **TlsAlloc()/TlsFree()**
- **Any thread:**
  - **TlsSetValue()**
  - **TlsGetValue()**



Thread local storage *(TLS)* is a method by which each thread in a process is given a location(s) in which to store thread-specific data. An application or DLL can use these to store separate instances of the same logical data on a per-thread basis. The TLS functions manipulate TLS indexes, which refer to storage areas for each thread in a process. A given TLS index is valid only within the process that allocated it; it cannot be shared across processes.

A call to `TlsAlloc()` allocates a global TLS *'index'* to represent a logical piece of data that can have different values for different threads. This one TLS index is valid for every thread within the process that allocated it, and should therefore be saved in a global or static variable. When `TlsAlloc()` is called, every thread within the process has its own private DWORD-sized space reserved for it (in its stack space, but this is implementation specific). However, only one Tls index is returned. This single ILS index may be used by each and every thread in the process to refer to the unique space that `TlsAlloc()` reserved for it.

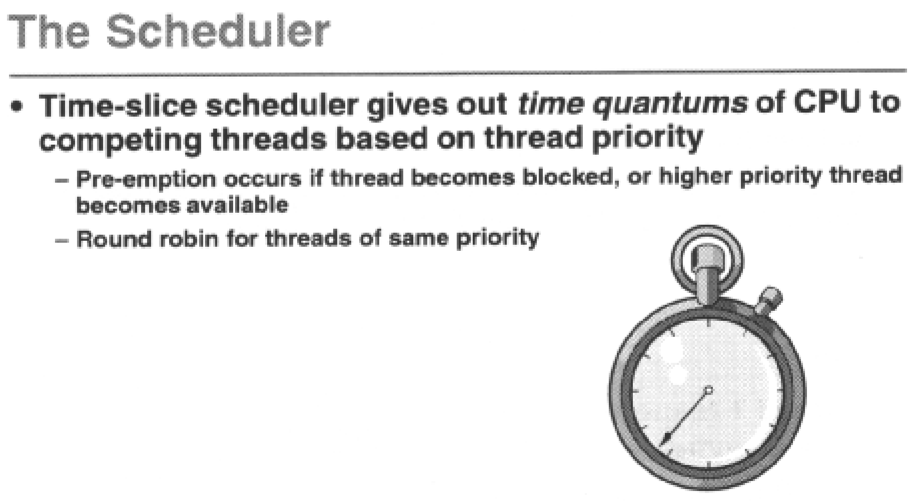
For this reason, `TlsAlloC()` is called only once, often by the main thread of a process. This is convenient for DLLs, which can distinguish between the first process's thread connecting to the DLL and subsequent threads of that process attaching. The thread that calls `TlsAlloc()` might store the TLS index in a global or static variable, and every other thread could refer to the global variable to access their local storage space.

Thread 1, and then subsequent threads of the same process, can store different thread-specific values in the same TLS index, using `TlsSetValue()`, specifying the index. Win32 guarantees that there are a minimum number, TLS_MINIMUM_AVAILABLE, of TLS indices per process; currently the value is 64. Often, only one TLS index is sufficient, as it can be used to store a pointer to some dynamic memory. As threads need to obtain their thread-specific data, they call `TlsSetValue()` specifying the TLS index; the thread context will dictate which thread-specific data item gets passed back.

A process should free TLS indexes with `TlsFree()`, specifying the TLS index, when it has finished using them. However, if any threads in the process have stored a pointer to dynamically allocated memory within their local storage spaces, it is important to free the memory or retrieve the pointer to it before freeing the TLS index, or it will be lost.

The compiler also has extended attribute syntax. It uses a keyword, *declspec*, to specify that an instance of a given type is to be stored with a Microsoft-specific storage class attribute. For example, to declare an integer variable which is used for storing thread-specific data:

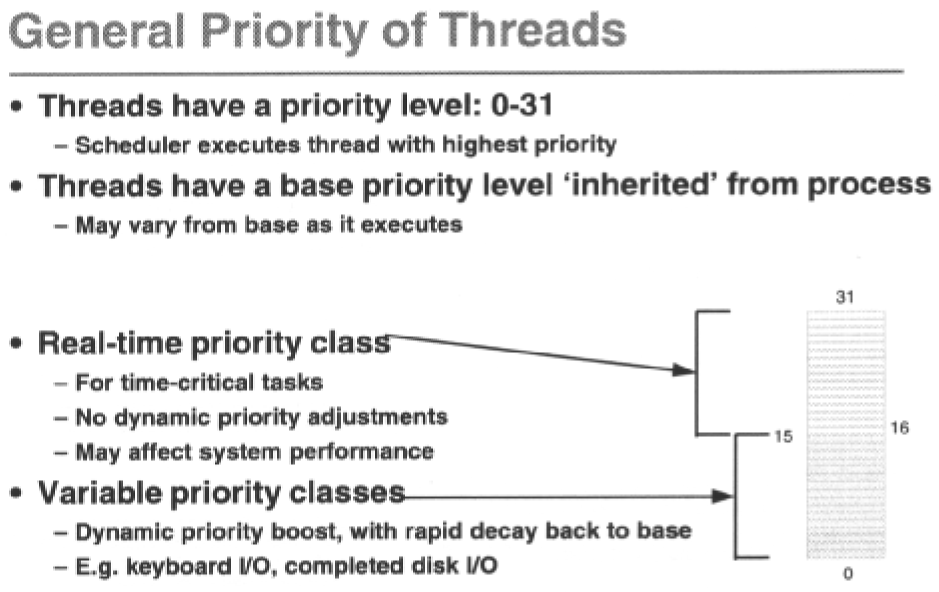
```
__declspec( thread ) int tls_i = 1;
```



The Scheduler

- **Time-slice scheduler gives out *time quantums* of CPU to competing threads based on thread priority**
  - Pre-emption occurs if thread becomes blocked, or higher priority thread becomes available
  - Round robin for threads of same priority

Threads are given CPU time by the time-slice scheduler according to their *priority,* scheduling higher-priority threads before those of lower priority. The scheduler gives out *'time quantums'* of CPU time to competing threads based on their priorities. The scheduler decides which thread receives the next CPU time quantum, at the end of each time quantum, or if the currently executing thread becomes blocked, or if a higher-priority thread becomes runnable. It always gives control to the highest priority runnable thread. If several runnable threads have the same priority, the one that has been waiting longest gets the time slice, i.e *'round-robin'.* A runnable thread is one that is not waiting, e.g. blocked on input, or waiting for a semaphore to clear. Non-runnable threads are not considered for CPU time until they become runnable.



General Priority of Threads

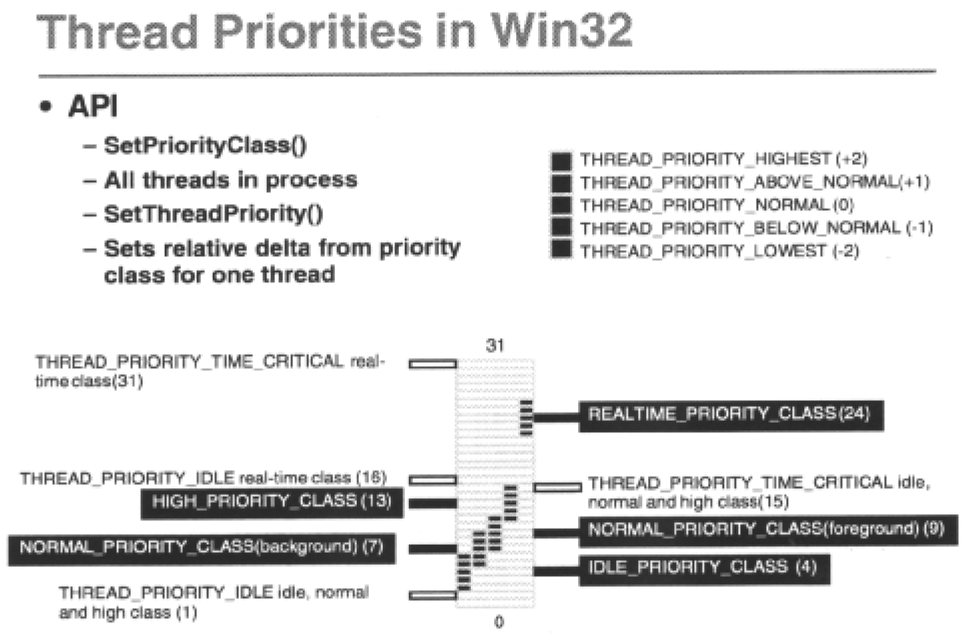
- **Threads have a priority level: 0-31**
  - Scheduler executes thread with highest priority
- **Threads have a base priority level 'inherited' from process**
  - May vary from base as it executes
- **Real-time priority class**
  - For time-critical tasks
  - No dynamic priority adjustments
  - May affect system performance
- **Variable priority classes**
  - Dynamic priority boost, with rapid decay back to base
  - E.g. keyboard I/O, completed disk I/O

There are 32 *priority levels* divided into two classes - *variable* class (1-15) and *real-time* class (16-31), with 31 being highest priority. Priority 0 is reserved for system use, used for tasks like garbage collection.

Each process has a default base priority level affecting its threads when it is created. A new thread for the process inherits this base priority level when it starts executing. The thread's priority level may vary from this base as it executes.

Threads with a priority in the real-time class do not have their priority altered by the kernel. Generally, this class is used for threads used by time-critical programs that need immediate attention from the processor. They will always pre-empt lower-priority threads of all other processes, including system processes. Don't execute a real-time priority thread for more than a very brief interval, else the system will become unresponsive.

The priority of threads in the variable class is altered dynamically by the priority scheduler to optimize system response time; their priorities can be boosted or decremented depending on the execution profile of the thread. For instance, the scheduler boosts a thread's priority after releasing it from a wait (the size of the boost depends on what it was waiting for). If the event was keyboard I/O it would be given a large boost, if the event was a disk I/O request being satisfied it would be smaller. The dispatcher interrupts a thread after each time quantum to make a scheduling decision. If it is a variable-class thread, it will decrement the thread's priority until it reaches its base priority level. Thus the priority of a compute-bound thread will gradually decay. Generally speaking, the scheduler is responsive to the user, so interactive threads run at a higher priority than I/O-bound threads, which run at a higher priority than compute-bound threads.



Thread Priorities in Win32

- **API**
  - **SetPriorityClass()**
  - **All threads in process**
  - **SetThreadPriority()**
  - **Sets relative delta from priority class for one thread**

■ THREAD_PRIORITY_HIGHEST (+2)
■ THREAD_PRIORITY_ABOVE_NORMAL(+1)
■ THREAD_PRIORITY_NORMAL (0)
■ THREAD_PRIORITY_BELOW_NORMAL (-1)
■ THREAD_PRIORITY_LOWEST (-2)
31
THREAD_PRIORITY_TIME_CRITICAL real-time class(31)
REALTIME_PRIORITY_CLASS (24)
THREAD_PRIORITY_IDLE real-time class (16)
HIGH_PRIORITY_CLASS (13)
THREAD_PRIORITY_TIME_CRITICAL idle, normal and high class(15)
NORMAL_PRIORITY_CLASS(foreground) (9)
NORMAL_PRIORITY_CLASS(background) (7)
IDLE_PRIORITY_CLASS (4)
THREAD_PRIORITY_IDLE idle, normal and high class (1)
0

In Win32, things are a little more complicated. Each thread has a base priority level which can be combined with a relative delta (+ or - 2) from this base, to provide a thread priority. Each process has a priority class which affects the base priority level of all threads in the process.

The Win32 variable priority class of threads spans 3 ‘synthetic’ Win32 priority classes:

IDLE_PRIORITY_CLASS	Only run when system is idle. Pre-empted when a process of higher priority class becomes runnable. This class equates to a base priority level of 4.
NORMAL_PRIORITY_CLASS	Normal scheduling needs. Will always pre-empt idle processes. This class equates to a base priority level of 9 if a process window is in the foreground, 7 if not.
HIGH_PRIORITY_CLASS	Special scheduling needs. Will always pre-empt normal and idle processes. Don’t execute high priority threads for extended periods as this will starve lower priority threads of the CPU. Typically used for threads that remain blocked for most of the time, and respond to a time-critical event with a flurry of CPU activity. This class equates to a base priority level of 13.

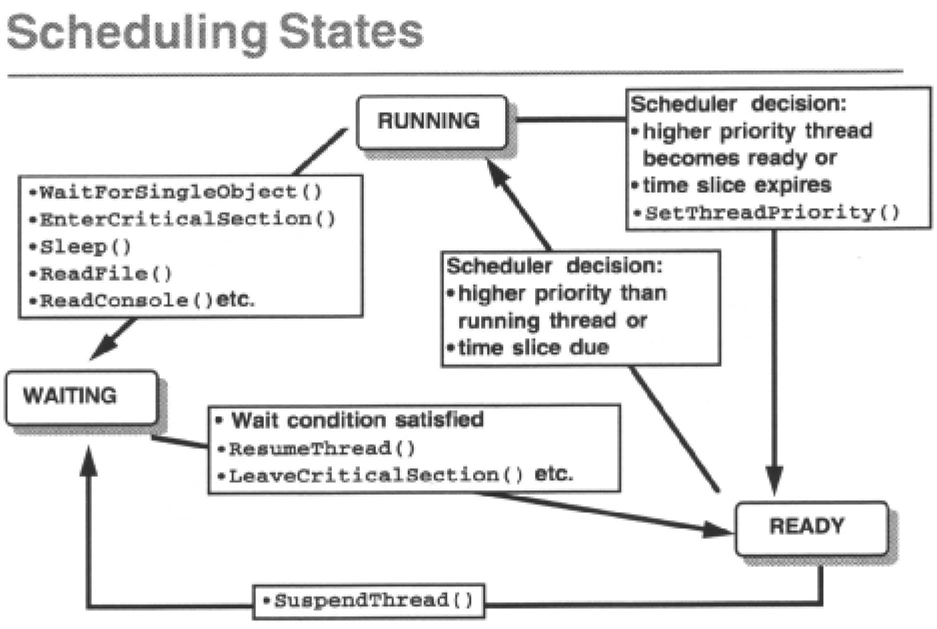
The Win32 real-time priority class of threads is represented by one priority class:

REALTIME_PRIORITY_CLASS	Generally this class is used for threads used by time-critical programs that need immediate attention from the processor. They will always pre-empt lower-priority threads of all other processes, including system processes. Don’t execute a real-time priority thread for more than a very brief interval, else the system will become unresponsive. This class equates to a base priority level of 24.

The relative priority deltas combine with the base priority levels dictated by the priority classes to allow thread priorities spanning the range 1-15 for the variable priority classes and 16,22-26,31 for the real-time class. Predefined deltas are THREAD_PRIORITY_LOWEST(-2), THREAD_PRIORITY_BELOW_NORMAL(-l), THREAD_PRIORITY_NORMAL(O), THREAD_PRIORITY_ABOVE NORMAL(l), THREAD_PRIORITY_HIGHEST(2).

Other predefined increments are THREAD_PRIORITY_IDLE, which combines with any variable class to give a thread priority of 1 and with the real-time class to give a thread priority of 16, and THREAD_PRIORITY_TIME_CRITICAL which combines with any variable class to give a thread priority of I 5, and with the real-time class to give a thread priority of 31.

One process can spawn another with a specific priority class. If not specified, the default priority class is normal, unless the creating process is of the idle priority class, in which case it is idle. All threads are started with a priority delta of normal, but an application can change this.



This diagram represents a simple conceptual view of scheduling states; the real implementation is slightly more complicated.

Each time the CPU time slice is awarded to a different thread, there is a *'context switch'* ; the context of the currently executing thread is saved, and the processor state is restored to the context of the thread to he executed. There are two reasons for a context switch:

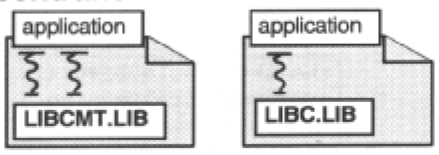
Voluntary switch where a thread gives up the CPU by waiting on an object, or terminating, or setting its priority lower etc.

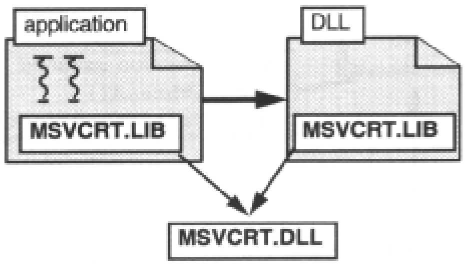
Preemption a higher priority thread has become available to run and takes over the CPU.

Watch for deadlock. If you have a higher-priority thread waiting for a lower priority thread to complete some task, be sure to block the execution of the waiting thread, using a wait function (discussed in the next chapter), rather than having it iterate in a loop. Otherwise, the process will deadlock, since the lower-priority thread will never get scheduled.

Thread Safety

- **Standard C run-time library not reentrant**
  - ***LIBC.LIB***
- **Multi-threaded library support**
  - ***LIBCMT.LIB* for applications**
  - ***CRTDLL.DLL* for applications and DLLs**
- **Use special thread creation/deletion**
  - **_beginthreadex()**
  - **_endthreadex()**





The normal C run-time library for applications is the statically linked LIEC LIB. Many functions in the standard C run-time libraries, including print f() and scanf(), share global data and other resources and therefore cannot be used in multi-threaded programs. They are *'non-reentrant'*. If a thread were to be pre-empted in the middle of using such a function, the data it needed might well have been changed by another thread by the time it was scheduled to run again. This is no good for DLLs which need to be able to service many threads from many clients, and multi-threaded clients.

There are special C libraries to support multi-threaded applications. These include reentrant adaptations of non-reentrant functions that use per-thread data when needed and use critical sections to protect shared data structures from concurrent thread access.

Applications can link to LIBCMT.LIB, which is a statically linked, re-entrant C-runtime library.

To build a multi-threaded application that uses C-Runtime, compile with the /MT option and link with

LIBCMT. LIB.

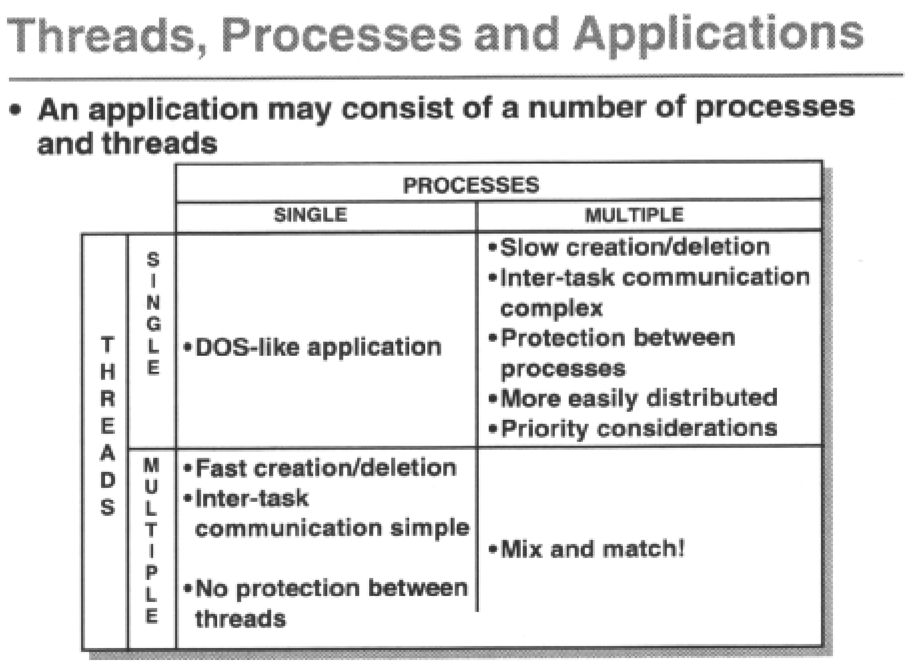
Applications and DLLs can link to MSVCRT. LIB, an import library for MSVCRT2 0 DLL, which is a dynamically linked reentrant C run-time dynamic link library. DLLs which use C run-time and have multi-threaded client applications must use this library.

To build an application or DLL which uses reentrant C run-time libraries, the code must be compiled with the /MD compiler flag. This forces C-runtime header files to define their contents differently in order to support the reentrant versions of the C run-time library.

Note that there is a caveat to using MSVCRT2 C . DLL. If you have an application which uses the services of a DLL, and you want to link either the .EXE or the .DLL with MSVCRT. LIB to use MSVCRT2 0 .DLL, then you must link both with MSVCRT. LIB. If you don't, the .EXE and the .DLL will not get the same initialization of variables, and thus calls to the C run-time from the .DLL with parameters from the .EXE will fail.

These reentrant libraries also include `beginthreadex()` and `endthreadex()` a simpler C run-time alternative to creating and destroying threads. These **must** be used instead of CreateThread() and ExitThread() if the multi-threaded C-runtime functions are used, because they set up the synchronization environment needed to ensure re-entrancy. They map down onto `CreateThread()` and `ExitThread()`.

Note. Do not assume that an application is single threaded just because it avoids the explicit creation of additional threads. That is because supporting code for an application may implicitly create additional threads. For example, the Control-C processing done for console applications creates an additional thread to call the applications Control-C handler.



Threads, Processes and Applications

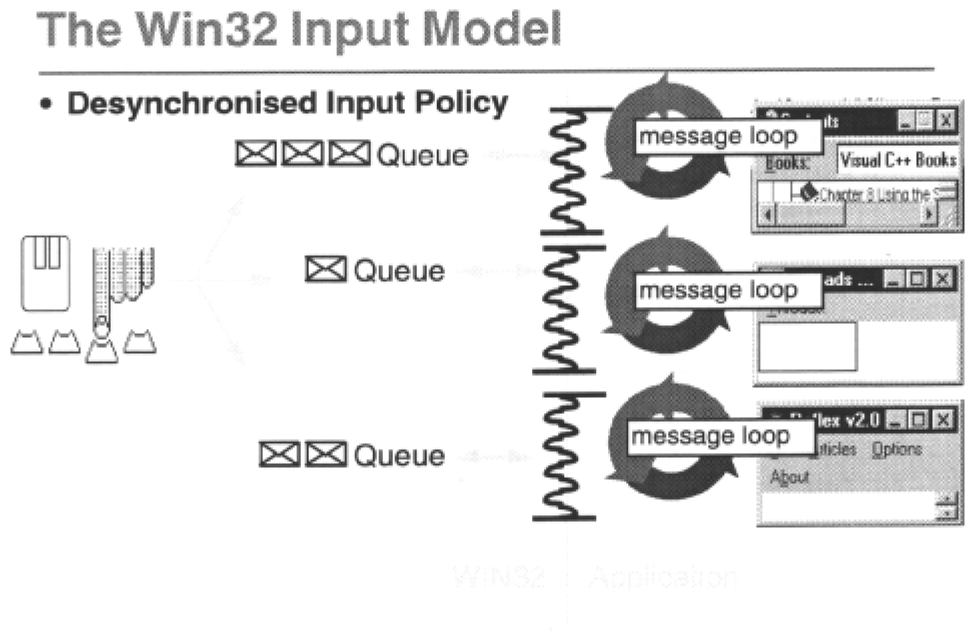
- **An application may consist of a number of processes and threads**

		PROCESSES	
		SINGLE	MULTIPLE
THREADS	SINGLE	•DOS-like application	•Slow creation/deletion •Inter-task communication complex •Protection between processes •More easily distributed •Priority considerations
	MULTIPLE	•Fast creation/deletion •Inter-task communication simple •No protection between threads	•Mix and match!

A process may contain one or more threads, and can create and terminate other processes. A Win32 application may therefore use multiple processes, each containing multiple threads.

The design considerations for a Win32 application are summarized in the diagram above. Unlike threads, processes offer a protected environment, but take much longer to create than threads. It is easier and quicker to share data between threads within a process, rather than to share data between processes by any of the means of interprocess communication. Processes are therefore best used for major, functionally separate, sections of an application that are not constantly invoked.

The maximum number of threads that can be created simultaneously in a process is only limited by available memory.



The Win32 input model is different from that of Windows 3.x; input ownership is decided at input time, instead of at the time it is read from the system queue. Each Win32 thread has its own input queue and associated input status (functions such as mouse capture, input focus and activation).

This means that the input of each thread is desynchronized with respect to all other threads. Of course, now the Win32 user interface is multi threaded. If one thread takes an inordinate amount of time to process a message, it will lock out only the windows owned by that thread. Windows in other threads will be unaffected.

Keyboard input goes to the focus window via the input queue of the thread that owns the focus window. Mouse input goes to the mouse under the window or the capture window via the input queue of the thread that owns that window.

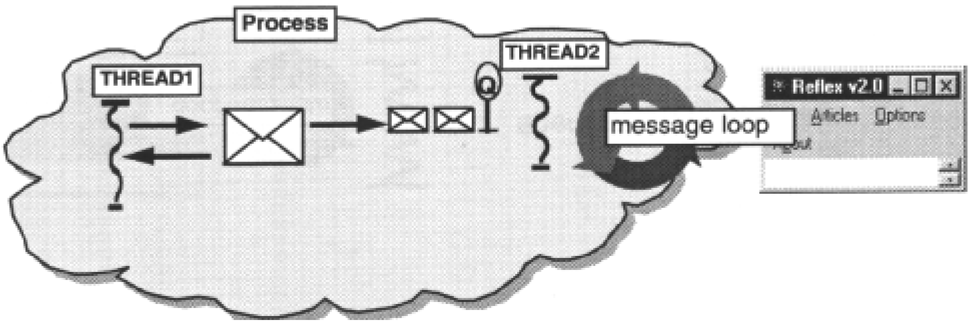
NULL is a valid return from `GetFocus()` and `GetActiveWindow()` if the current thread doesn't own the focus window or active window.

Win32 cannot know ahead of time whether an input thread will set the mouse capture. So by default, all mouse input between a button down and matching button up is captured, and goes to the window in which the button down occurred - traditionally what `SetCapture()` was used for in Windows 3.x.

Now, if `SetCapture()` is called while a mouse button is down, the window will have capture until specifically released with ReleaseCapture() or until the mouse button is released. This is what you might expect. More interestingly, If `SetCapture()` is called while a mouse button is up, the window will have capture only while it is over a window owned by the capturing thread.

Threads and Message Queues

- Messages can be posted to a threads message queue
  - PostMessage(); window handle
  - PostThreadMessage(); thread ID
    - replaces PostAppMessage()
- Main thread can continue execution immediately
  - Messages posted to worker threads



In Windows 3.x, the system enforces serialisation of message processing by implementing a shared message queue. This causes a problem if one application decides to perform a lengthy task in response to a message, without cooperatively yielding control to Windows 3.x periodically; all other Windows 3.x applications grind to a halt.

In a multi-threaded application, any thread can call the `CreateWindow()` API to create a window. There are no restrictions on which thread(s) can create windows. Each thread created by a Win32 application has its own message queue. When windows are created, they are *'owned'* by the thread that created them, and all their queued messages will go via that queue. All queued and non-queued messages for a window will be processed in the context of the owning thread. This means that if a thread fails to service its queue, then only the windows owned by that thread would suffer; windows belonging to other threads will behave normally. A lengthy operation in response to a message in a Win32 application will cause the hourglass cursor to appear only over the windows owned by the offending thread, and the normal selection cursor to appear over all other windows. An application performing a lengthy task should generally spawn a new thread.

Because window handles are unique across the system, simply posting / sending a message to a given window will locate the correct thread context, based on thread ownership. Posting messages to a thread without any windows is also possible. `PostThreadMessage()` replaces `PostAppMessage()` and works in an identical fashion. The message is posted to the queue of the thread specified. When the message is retrieved at a later date by the receiving thread calling `GetMessage()`, the `HWND` field of the MSC structure is set to NULL.

Using `MsgWaitForMultipleObjects()` is possible to wait on the calling threads input queue. The calling threads input event attains a signaled state when suitable input is available for the thread. A mask parameter determines the suitable types of input. Possibilities for queue events to wait on are:
- Character messages, mouse messages, paint messages, posted messages, messages sent by another thread or application, timer messages, hot-key messages, any input message or any message.

As mentioned above, windows created on different threads will process messages independently of each other. They have their own input state (focus, active window, capture window, keystate, queue status etc.), and they are not synchronized with respect to the input processing of other threads. Just to confuse matters, by using the `AttachThreadInput()` function, a thread can serialize its inputprocessing to that of other threads. This also allows the threads to share input state, so it can actually make `SetFocus()` calls to windows of different threads, and get keystate information. Normally these things are impossible.