Win32 System Programming Issues: Threads, Multitasking, and Memory Management

- The Windows environment can be accessed through a call-based interface called the **Application Program Interface** (**API**), also called the **Win32 API.**

- **Win32** supports a **32-bit architecture**, that is, **32-bit addressing** and **32-bit arguments** in its function calls.

- The **Win32 API** defines a set of functions that a Windows application can use to take advantage of the preemptive **multitasking** capabilities of Win32.

- The **Virtual Machine Manager** (**VMM**) is the core of the Windows OS and is used to conduct essential tasks such as **Memory Management**, **Interrupt Handling**, and **Thread Scheduling**.

- The VMM also sets up and manages **Virtual machines (VMs)** and **Virtual Device Drivers** (**VxDs**).

Processes

- A process can be defined as an instance of an executing program.

- In **Win32**, a process when owns a **4-GB** address space.

- By itself, the Win32 process is **inert**. That is, the process simply owns a 4-GB address space containing the code and data for an application's executable code.

- In order for the process to accomplish a useful function, the process must own a **thread**.

- The **thread** is responsible for executing the code contained in the process's address space.

- In fact, a single process might contain several threads, all of them executing code "simultaneously" in the process's address space.

- A new process is created when your application calls the **CreateProcess** function:

```
        BOOL CreateProcess (
                LPCTSTR  lpszImageName,   // address of executable module name
                LPCTSTR  lpszCommandLine,         // address of command line string
                LPSECURITY_ATTRIBUTES  lpsaProcess,     // security  attributes
                LPSECURITY_ATTRIBUTES  lpsaThread,      // address of thread
security                                                        //   attributes,
Default: NULL
                BOOL  fInheritHandles,        // new process inherits handles
                DWORD  fdwCreate,             // creation flags
                LPVOID  lpvEnvironment,    // address of new environment block
                LPCTSTR  lpszCurDir,          // address of current directory name
                LPSTARTUPINFO  lpsiStartInfo,       // address of STARTUPINFO
                LPPROCESS_INFORMATION  lppiProcInfo   // address of
                                                        PROCESS_INFORMATION
        );
```

lpszImageName

Points to a null-terminated string specifying the full path and filename of the module to execute. If this parameter is NULL, the module name must be the first white space-delimited token in the **lpszCommandLine** string.

lpszCommandLine

Points to a null-terminated string specifying the command line for the application to be executed. If this parameter is NULL, the **lpszImageName** string is used as the command line. If both lpszImageName and lpszCommandLine are non-NULL, lpszImageName specifies the module to execute and lpszCommandLine is used as the command line.

lpsaProcess

Points to a **SECURITY_ATTRIBUTES** structure that specifies the security attributes for the created process. If **lpsaProcess** is NULL, the process is created with a default security descriptor, and the resulting handle is not inherited.

lpsaThread

Points to a **SECURITY_ATTRIBUTES** structure that specifies the security attributes for the primary thread of the new process. If lpsaThread is NULL, the process is created with a default security descriptor, and the resulting handle is not inherited.

fInheritHandles

Indicates whether the **new process** inherits handles from the **calling process**. If TRUE, each inheritable open handle in the calling process is inherited by the new process. Inherited handles have the same value and access privileges as the original handles.

fdwCreate

Specifies additional flags that control the **priority class** and the creation of the process.

Priority Class	Flag Identifier
Idle	IDLE_PRIORITY_CLASS
Normal	NORMAL_PRIORITY_CLASS
High	HIGH_PRIORITY_CLASS
Realtime	REALTIME_PRIORITY_CLASS

lpszCurDir

Points to a null-terminated string that specifies the current drive and directory for the new process. The string must be a full path and filename that includes a drive letter. If this parameter is NULL, the new process is created with the same current drive and directory as the calling process.

lpsiStartInfo

Points to a **STARTUPINFO** structure that specifies how the main window for the new process should appear.

lppiProcInfo

Points to a **PROCESS_INFORMATION** structure that receives identification information about the new process.

- If the function succeeds, the return value is **TRUE.**
- If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**.
- In addition to creating a process, **CreateProcess** also creates a thread object. The thread is created with an initial stack whose size is described in the image header of the specified program's executable file. The thread begins execution at the image's entry point.
- The process is assigned a 32-bit process identifier. The ID is valid until the process terminates. It can be used to identify the process, or specified in the **OpenProcess** function to open a handle to the process.
- The initial thread in the process is also assigned a 32-bit thread identifier. The ID is valid until the thread terminates and can be used to uniquely identify the thread

within the system. These identifiers are returned in the **PROCESS_INFORMATION** structure.

- The preferred way to shut down a process is by using the **ExitProcess** function, because this function notifies all DLLs attached to the process of the approaching termination.
- Other means of shutting down a process do not notify the attached DLLs. Note that when a thread calls **ExitProcess**, other threads of the process are terminated without an opportunity to execute any additional code (including the thread termination code of attached DLLs).
- Study the process creation example in your text

Threads

- A **thread** describes a path of execution within a process.
- Every time a process is initialized, the system creates a **primary thread** which starts at the C run-time's startup code.
- The thread calls your **WinMain** function and continues executing until the **WinMain** function returns and the C run-time's startup code calls **ExitProcess**.
- For most applications, the primary thread is the only thread required.

However, processes can create additional threads to help them perform the various tasks performed by the application.

Creating a Thread

- The **CreateThread()** function is used to create a new thread.
- The **CreateThread** function creates a thread to execute within the address space of the calling process.

```
HANDLE CreateThread (
        LPSECURITY_ATTRIBUTES  lpsa,    // security attributes
        DWORD  cbStack,                  // initial thread stack size, in bytes
        LPTHREAD_START_ROUTINE  lpStartAddr,    // address
                                           of thread function
        LPVOID  lpvThreadParm,          // argument for new thread
        DWORD  fdwCreate,               // creation flags
        LPDWORD  lpIDThread            // address of returned thread ID
);
```

lpsa

Points to a **SECURITY_ATTRIBUTES** structure that specifies the security attributes for the thread.

cbStack

Specifies how much address space the thread is allowed to use for its own stack. If 0 is specified, the stack size defaults to the same size as that of the primary thread of the process.

The stack is allocated automatically in the memory space of the process and it is freed when the thread terminates. Note that the stack size grows, if necessary.

lpStartAddr

Points to the application-supplied function to be executed by the thread and represents the starting address of the thread.

The function accepts a single 32-bit argument and returns a 32-bit exit value.

It is quite legal (and perhaps useful) to create multiple threads that all have the same function address as their starting point.

lpvThreadParm

Specifies a single 32-bit parameter value passed to the thread. This is usually some initialization data, a pointer to a data structure that contains additional information.

fdwCreate

Specifies additional flags that control the creation of the thread. It can be one of two values.

If the **CREATE_SUSPENDED** flag is specified, the thread is created in a suspended state, and will not run until the **ResumeThread()** function is called.

If this value is **zero**, the thread runs immediately after creation.

lpIDThread

Points to a 32-bit variable that receives the thread identifier.

This must be a valid address of a **DWORD** in which **CreateThread()** will store the ID that the system assigns to the new thread.

- If the function succeeds, the return value is a handle to the new thread.
- If the function fails, the return value is NULL. To get extended error information, call GetLastError.

- All threads begin executing at a function that you must specify.
- The function must have the following prototype:

  DWORD WINAPI YourThreadFunc(LPVOID lpvThreadParm);

- The system calls your thread function, and passes it the 32-bit **lpvThreadParm** parameter that was originally passed to the **CreateThread** function.

Terminating a Thread

- A thread can be terminated using one of the following system calls:
- The **ExitThread** function:

  VOID ExitThread (DWORD dwExitCode);

  - This function terminates the thread and sets the thread's exit code to **dwExitCode**.

    - This is the most common method because it releases all the thread's resources (deallocates the stack, etc.) when it terminates.

- The **TerminateThread** Function:

    BOOL TerminateThread (HANDLE hThread, DWORD dwExitCode);

    - This function terminates the thread identified by **hThread** and sets its exit code to **dwExitCode**.

    - This function is used only as a last resort when the thread is no longer responding.

    - Upon termination the thread has no chance to execute any user-mode code and its initial stack is not deallocated. DLLs attached to the thread are not notified that the thread is terminating.

<u>Thread Scheduling</u>

- All active threads are scheduled by the system based on their **priority levels**.
- The **primary** and **time-slicing schedulers** use different priority schemes and the **execution priority** of a thread determines whether the scheduler will grant it control next.
- Each thread is assigned a priority level ranging from **0 (lowest**) to **31** (**highest**).
- Priority level 0 is assigned to a special thread in the system called the **zero page** thread.
- The zero page thread is responsible for zeroing any free pages in the system when there are no other threads that need to perform tasks in a system.
- No other thread is allowed to have a priority level of 0.
- The system allocates a CPU slice to all threads of **priority 31** first, until there are no more priority 31 threads.
- Then all threads with decreasing priority (30, 29, etc.) are given CPU slices.
- This implies that if you always have at least on priority 31 thread for each CPU, other threads with a priority less than 31 will never execute.
- This is known as **starvation**. Some threads use so much of the CPU time that other threads are starved for CPU time slices.
- **Higher-priority** threads always preempt **lower-priority** threads regardless of what the lower-priority threads are executing.

<u>Assigning Priority Levels Using the Win32 API</u>

- The system determines a thread's priority level using a **two-step** process.
- The first step is to assign a priority class to a **process** compared to other running processes.
- The second step is to assign relative priority levels to **threads** owned by the process.

<u>Process Priority Classes</u>

- You can assign a priority class to a process by Oring one of the **CreatProcess** flags listed below with the other **fdwCreate** flags when calling **CreateProcess**.

Class	CreateProcess Flag	Level
Idle	IDLE_PRIORITY_CLASS	4
	BELOW NORMAL_PRIORITY_CLASS	5
Normal	NORMAL_PRIORITY_CLASS	7 background)- 9 (foreground)
	ABOVE NORMAL_PRIORITY_CLASS	11
High	HIGH_PRIORITY_CLASS	13
Realtime	REALTIME_PRIORITY_CLASS	24

- The importance of carefully selecting a process's priority class cannot be stressed enough.
- When calling **CreateProcess**, most applications should either not specify a priority class or use the **NORMAL_PRIORITY_CLASS** flag.
- If the priority class is not specified, the system will assume normal priority unless the parent has an idle priority class.
- When a user is working with a process, that process is said to be a **foreground** process and all other processes are called **background** processes.
- When a normal process is brought to the foreground, Windows NT automatically boosts all of that process's threads by 2.
- The reason for this is simply to make the foreground process react much faster to the user's input.

- There are some general guidelines in assigning priorities to different applications:
- **IDLE_PRIORITY_CLASS**
    - This is ideal for **system monitoring** applications or **screen saver** applications.
    - Indicates a process whose threads run only when the system is **idle** and are preempted by the threads of any process running in a higher priority class.
- **HIGH_PRIORITY_CLASS**
    - Must only be used when absolutely necessary because a high-priority class CPU-bound application can use nearly all available cycles.
    - The **Windows Task Manager** runs at high priority. The **Windows Task List** must respond quickly when called by the user, regardless of the load on the operating system.
    - Indicates a process that performs time-critical tasks that must be executed immediately for it to run correctly. The threads of a high-priority class process preempt the threads of normal or idle priority class processes.
- **NORMAL_PRIORITY_CLASS**
    - This is the most common priority class which indicates a normal process with no special scheduling needs.
- **REALTIME_PRIORITY_CLASS**
    - Indicates a process that has the **highest possible priority**. The threads of a real-time priority class process preempt the threads of all other processes, including **operating system processes** performing important tasks.
    - This must almost never be used because every other thread in the system will have lower priority, including **mouse control**, **keyboard**, **background disk flushing**, and **CTRL+ALT_DEL**.
    - This class might be used if you are writing an application that talk directly to **hardware**, or if you need to perform some **short-lived tasks** which cannot be interrupted.

<u>Modifying a Process's Priority Class</u>

- Once a child process is running, it can change its own priority class by calling the **SetPriorityClass** function:

  BOOL SetPriorityClass (HANDLE hProcess, DWORD fdwPriority);

  - **hProcess**

    Identifies the process. The handle must have **PROCESS_SET_INFORMATION** access.

  - **fdwPriority**

    Specifies the priority class for the process.

- The **fdwPriority** parameter can be one of the following values: **HIGH_PRIORITY_CLASS**, **IDLE_PRIORITY_CLASS**, **NORMAL_PRIORITY_CLASS**, and **REALTIME_PRIORITY_CLASS.**

- The complementary function **GetPriorityClass** function returns the priority class for the specified process:

  DWORD GetPriorityClass (HANDLE hProcess);

  - **hProcess**

  Identifies the process. The handle must have **PROCESS_QUERY_INFORMATION** access.

- The function returns one of the **CreateProcess** flags. This value, together with the priority value of each thread of the process, determines each thread's base priority level.

Thread Relative Priority

- When a **thread** is first created, its priority level is that of the **process's priority class.**
- It is possible to raise or lower the **priority** of an individual **thread**.
- A thread's priority is always **relative** to the priority class of the process that owns it.
- You can change a thread's relative priority within a single process by calling the **SetThreadPriority** function :

  BOOL SetThreadPriority (HANDLE hThread, int nPriority);

  - **hThread**

  Identifies the thread. The handle must have **THREAD_SET_INFORMATION** access.

  - **nPriority**

    Specifies the priority value for the thread.

- The **npriority** parameter can be one of the values given in the following table.

Identifier	Meaning
THREAD_PRIORITY_IDLE (-4)	Indicates a base priority level of 1 for IDLE_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, or HIGH_PRIORITY_CLASS processes, and a base priority level of 16 for REALTIME_PRIORITY_CLASS processes.
THREAD_PRIORITY_ABOVE_IDLE (-3)	
THREAD_PRIORITY_LOWEST (-2)	The thread's priority should be 2 less than the process's priority class
THREAD_PRIORITY_BELOW_NORMAL (-1)	The thread's priority should be 1 less than the process's priority class
THREAD_PRIORITY_NORMAL (0)	The thread's priority should be the same as the process's priority class
THREAD_PRIORITY_ABOVE_NORMAL (+1)	The thread's priority should be 1 more than the process's priority class
THREAD_PRIORITY_HIGHEST (+2)	The thread's priority should be 2 more than the process's priority class

THREAD_PRIORITY_TIME_CRITICAL (+3)	Indicates a base priority level of 15 for IDLE_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, or HIGH_PRIORITY_CLASS processes, and a base priority level of 31 for REALTIME_PRIORITY_CLASS processes.

- The system combines a **process's priority class** with a **thread's relative priority** to determine a **thread's base priority level**. This is shown in the table below.

Relative Thread Priority	**Process Priority Class**					
	Idle	Normal, in Background	Normal, in foreground (Boost +1)	Normal, in foreground (Boost +2)	High	Realtime
Time Critical	15	15	15	15	15	31
Highest	6	9	10	11	15	26
Above Normal	5	8	9	10	14	25
Normal	4	7	8	9	13	24
Below Normal	3	6	7	8	12	23
Lowest	2	5	6	7	11	22
Idle	1	1	1	1	1	16

- The complementary function **GetThreadPriority** returns the priority value for the specified thread:

int GetThreadPriority (HANDLE hThread);

- **hThread**

Identifies the thread. The handle must have **THREAD_QUERY_INFORMATION** access.

<u>Suspending and Resuming Threads</u>

- A thread can be created in a suspended state by passing the **CREATE_SUSPENDED** flag to **CreateProcess** or **CreateThread.**

- A thread can also be suspended by calling the **SuspendThread** function:

  DWORD SuspendThread (HANDLE hThread);

  - **hThread**

  - Identifies the thread. The handle must have **THREAD_SUSPEND_RESUME** access.

- Each thread has a **suspend count** (with a maximum value of **MAXIMUM_SUSPEND_COUNT**).

- If the suspend count is greater than zero, the thread is suspended; otherwise, the thread is not suspended and is eligible for execution.

- Calling **SuspendThread** causes the target thread's suspend count to be **incremented**. Attempting to increment past the maximum suspend count causes an error without incrementing the count.

- A thread can suspend itself but it cannot resume itself.

- To allow the thread to begin execution, another thread must call the ResumeThread function:

  DWORD ResumeThread (HANDLE hThread);

  - **hThread**

  Specifies a handle for the thread to be restarted. The handle must have **THREAD_SUSPEND_RESUME** access to the thread.

- The **ResumeThread** function **decrements** a thread's suspend count. When the suspend count is decremented to zero, the execution of the thread is resumed.

- That is, if the suspend count is 3, the thread must be resumed 3 times before it is eligible for a CPU execution slice.

- The function returns the thread's previous suspend count if successful.

<u>SUMMARY</u>

- The **time-slicing scheduler** enables all executing threads to **share the CPU** by giving each **highest priority thread** a **time slice** in **round robin** fashion.

- If a thread does not run within a "reasonable" period of time (determined internally by the VMM), the scheduler will boost its priority to prevent **starvation**.

- A **blocked thread** will have its **priority boosted** by the scheduler when it unblocks. This will ensure that it receives the same average CPU time when competing for resources.

- The scheduler **decays** a priority from a boosted value back to the base value when the thread time slice expires.

- Thus, the scheduler obeys the Win32 API instructions given to it as well as its own OS algorithms to generate **absolute priorities**.

- **Hardware interrupts** and **critical section** boosts will modify execution priorities quite substantially, thus effectively overriding the Win32 priority scheme.