

# C ++ THREAD LIBRARY

**Project Report 1999-2000**

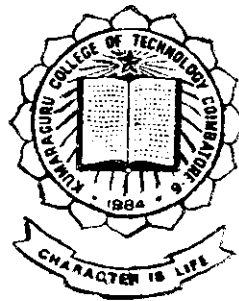
P-486

DISSERTATION SUBMITTED IN PARTIAL FULFILMENT  
OF THE REQUIREMENTS FOR THE DEGREE OF  
**MASTER OF ENGINEERING**  
IN COMPUTER SCIENCE AND ENGINEERING  
OF BHARATHIAR UNIVERSITY

By

**J. SARAVANAMATHY UMA KANCHANA**  
Reg. No. 9937K0010

Under the Guidance of  
**Mrs. S. DEVAKI, B.E., M.S.,**



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING  
**KUMARAGURU COLLEGE OF TECHNOLOGY**

# CERTIFICATE



**Department of Computer Science and Engineering  
Kumaraguru College of Technology  
(Affiliated to Bharathiar University)  
Coimbatore – 641 006**

**Project Work**

## **CERTIFICATE**

**Bonafide record of the project work**

**C + + THREAD LIBRARY**

Done by

**J.Saravanamathy Uma Kanchana  
(Reg. No. 9937K0010)**

Submitted in partial fulfilment of the requirements  
for the degree of Master of Engineering in  
Computer Science and Engineering  
of Bharathiar University



S. Sreenath  
Faculty Guide

S. Sreenath  
Head of the Department

Submitted for Viva-Voce held on 20-11-2011

S. Sreenath  
Internal Examiner

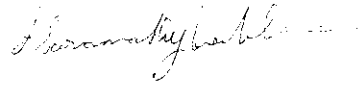
S. Sreenath  
External Examiner

## Declaration

I, J.Saravanamathy Uma Kanchana hereby declare that this project work entitled “C + + **THREAD LIBRARY** ” submitted to Kumaraguru College of Technology, Coimbatore(Affiliated to Bharathiar University) is a record of original work done by me under the supervision and guidance of Mrs.Devaki M.S., Department of Computer Science and Engineering.

Register Number  
9937K0010

Name of the Candidate



J.Saravanamathy Uma Kanchana

Signature of the Candidate

Countersigned by:

Staff in charge



Mrs.S.Devaki B.E., M.S.,

Assistant Professor

Department of Computer Science and Engineering

Kumaraguru College of Technology

Coimbatore – 641 006

Place : Coimbatore

Date :



*Dedicated to my  
Beloved Parents*



# CONTENTS

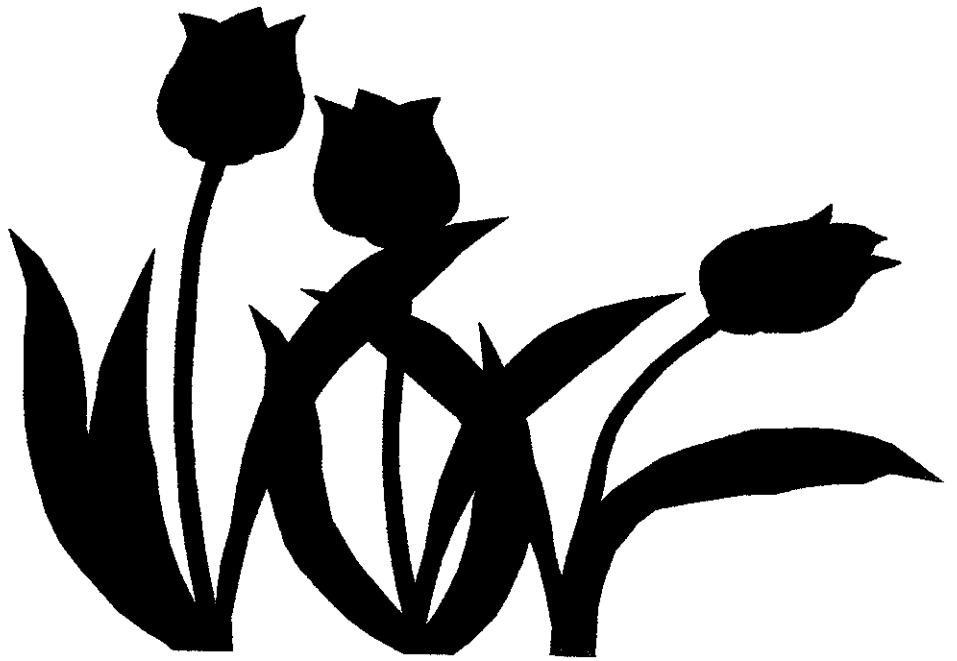
## ACKNOWLEDGEMENT

## SYNOPSIS

<b>1. INTRODUCTION</b>	<b>1</b>
<b>2. HARDWARE AND SOFTWARE ENVIRONMENT</b>	<b>4</b>
<b>3. THE CREATETHREAD FUNCTION</b>	<b>5</b>
3.1 psa	
3.2 cbstack	
3.3 pfnStartAddr and pvParam	
3.4 fdwCreate	
3.5 pdwThreadID	
<b>4. TERMINATING A THREAD</b>	<b>13</b>
4.1 The thread function returns	
4.2 The ExitThread function	
4.3 The Terminate Thread function	
4.4 Process termination	
4.5 Thread termination	
<b>5. THREAD INTERNALS</b>	<b>19</b>
<b>6. THREAD SCHEDULING</b>	<b>25</b>
6.1 Suspending and Resuming a Thread	
6.2 Sleeping	
6.3 Switching to Another Thread	
6.4 Context in Context	



<b>7. THREAD PRIORITIES</b>	<b>31</b>
7.1 Process priorities	
7.2 Relative Thread priorities	
7.3 Programming Priorities	
<b>8. THREAD SYNCHRONIZATION WITH   KERNEL OBJECTS</b>	<b>39</b>
8.1 Wait Functions	
8.2 Wait for Single Object	
8.3 Wait for Multiple Object	
<b>9. THREAD SYNCHRONIZATON USING   CRITICAL SECTION</b>	<b>45</b>
<b>10. CLOSING A KERNEL OBJECT</b>	<b>51</b>
<b>CONCLUSION</b>	<b>53</b>
Scope for future development	
<b>BIBLIOGRAPHY</b>	<b>54</b>
<b>APPENDIX – A</b>	<b>55</b>
Thread class diagram	
Thread class implementation	
<b>APPENDIX – B</b>	<b>77</b>
Producer Consumer Problem	



# ACKNOWLEDGEMENT





## ACKNOWLEDGEMENT

I would like to acknowledge all the persons who have contributed to a great extent towards the initialization, the development and the success of our project.

I wish to express my sincere thanks to our Principal **Dr.K.K.Padmanaban, B.Sc.(Engg), M.Tech., Ph.D** for giving us the needed encouragement in starting this project and carrying it out successfully.

I also thank our Head of the Department, **Dr.S.Thangasamy** for providing an opportunity to take up this project.

Words are inadequate to thank and express my heartfelt gratitude to our guide **Asst. Prof. S.Devaki, B.E., M.S.**, for her valuable suggestions and constant support during every stage of the development of the project.

I take an immense pleasure especially in thanking our class advisor **Prof. K.Kannan, B.E., M.E.**, who showed keen interest in the development of this project.

I would like to thank **Mr.A.Elangovan, M.C.A.**, and **Mr.M.Ramesh, M.C.A.**, for providing me an extensive support to complete this project.

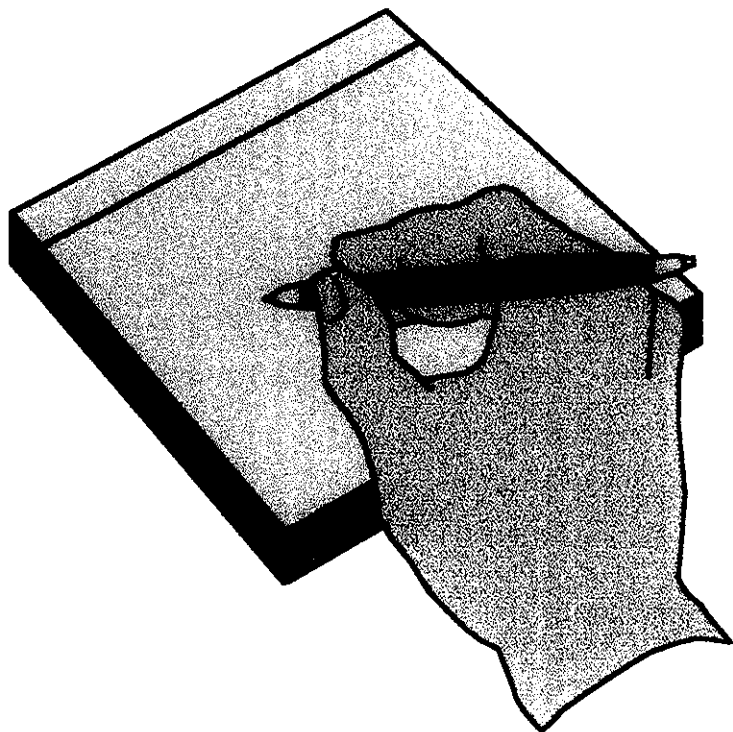
I would like to deeply thank my beloved parents, to whom this project is dedicated.



## SYNOPSIS

This Project is a wrapper class for platform specific thread APIs. If the application is not using our project they should directly use the APIs provided by the platform. So this will make this application unportable, because they have to recode the whole application according to the targeted platform. This will take more time, manpower and money.

In the application uses our thread library then the application can be ported to any platform as such. So the only module that we have to modify is our thread library, which will take very less time and manpower.



# INTRODUCTION



# 1. INTRODUCTION

A thread consists of two components:

- A kernel object that the operating system uses to manage the thread. The kernel object is also where the system keeps statistical information about the thread.
- A thread stack that maintains all the function parameters and local variables required as the thread executes code.

A process never executes anything; it is simply a container for threads. Threads are always created in the context of some process and live their entire life within that process. What this really means is that the thread executes code within its process's address space and manipulates data within its process's address space. So if we have two or more threads running in the context of a single process, the threads share a single address space. The threads can execute the same code and manipulate the same data. Threads can also share kernel object handles because the handle table exists for each process, not each thread.

Processes use a lot more system resources than threads do. The reason for this is the address space. Creating a virtual address space for a process requires a lot of system resources. A lot of record keeping takes place in the system, and this requires a lot of memory. Since .exe and .dll files get loaded into an address space, file resources are required as well. A thread, on the other hand, uses significantly fewer system resources.



A thread has just a kernel object and a stack; little record keeping is involved, and little memory is required.

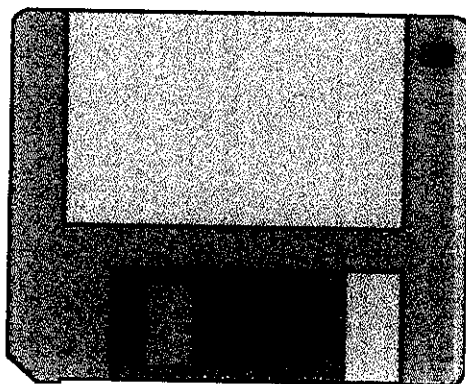
### **1.1 When to create a Thread**

A thread describes a path of execution within a process. Every time a process is initialized, the system creates a primary thread. This thread begins executing with the C/C++ run-time library's startup code, which in turn calls our entry-point function (*main*, *wmain*, *WinMain*, or *wWinMain*) and continues executing until the entry-point function returns and the C/C++ run-time library's startup code calls *ExitProcess*. For many applications, this primary thread is the only thread the application requires. However, processes can create additional threads to help them do their work.issues). To keep the CPU busy, we give it varied tasks to perform.

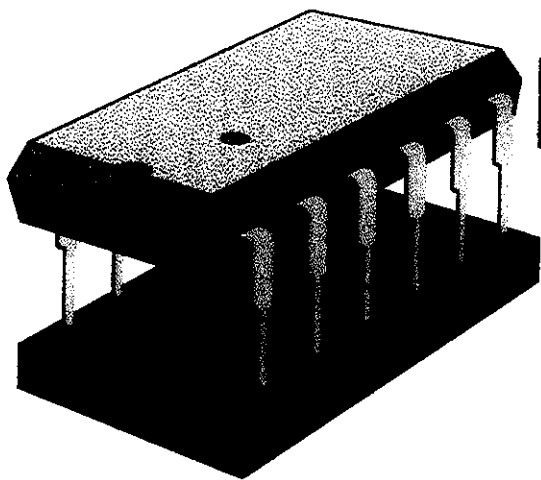
If we have two CPUs in your computer and two threads in our application, both CPUs will be busy. In effect, we get two tasks done in the time it would take for one.

### **1.2 Writing A First Thread Function**

Every thread must have an entry-point function where it begins execution. The entry-point function for our primary thread is: *main*, *wmain*, *WinMain*, or *wWinMain*. If we want to create a secondary thread in our process, it must also have an entry-point function. Our thread function can perform any task we want it to.



**HARDWARE & SOFTWARE**



**ENVIRONMENT**

## **2.HARDWARE AND SOFTWARE ENVIRONMENT**

### **2.1 Hardware environment**

The hardware environment under which the thread library was developed is as follows

Processor	—————>	X86
Hard disk drive	—————>	10.2 GB
RAM	—————>	32 MB

### **2.2 Software environment**

The minimum software environment under which the thread library can be executed is as follows

- \* Windows platform
- \* Any C ++ compiler





**THE  
CREATETHREAD  
FUNCTION**

### 3. THE *CreateThread* FUNCTION

A process's primary thread comes into being when *CreateProcess* is called. If we want to create one or more secondary threads, we simply have an already running thread call *CreateThread*:

```
HANDLE CreateThread(  
  
    PSECURITY_ATTRIBUTES psa,  
  
    DWORD cbStack,  
  
    PTHREAD_START_ROUTINE pfnStartAddr,  
  
    PVOID pvParam,  
  
    DWORD fdwCreate,  
  
    PDWORD pdwThreadId);
```

When *CreateThread* is called, the system creates a thread kernel object. This thread kernel object is not the thread itself but a small data structure that the operating system uses to manage the thread. We can think of the thread kernel object as a small data structure that consists of statistical information about the thread. This is identical to the way processes and process kernel objects relate to each other.

The system allocates memory out of the process's address space for use by the thread's stack.

The new thread runs in the same process context as the creating thread. The new thread therefore has access to all of the process's kernel object handles, all of the memory in the process, and the stacks of all other threads that are in this same process. This makes it really easy for multiple threads in a single process to communicate with each other.

### 3.1 *psa*

The *psa* parameter is a pointer to a SECURITY\_ATTRIBUTES structure. We can (and usually will) pass NULL if we want the default security attributes for the thread kernel object. If we want any child processes to be able to inherit a handle to this thread object, we must specify a SECURITY\_ATTRIBUTES structure, whose *InheritHandle* member is initialized to TRUE.

Security:

Kernel objects can be protected with a security descriptor. A security descriptor describes who created the object, who can gain access to or use the object, and who is denied access to the object. Security descriptors are usually used when writing server applications; we can ignore this feature of kernel objects if we are writing client-side applications.

Most applications will simply pass NULL for this argument so that the object is created with default security. Default security means that any member of the administrators group and the creator of the object have full access to the object; all others are denied access.

When we want to gain access to an existing kernel object (rather than create a new one), we must specify the operations we intend to perform on the object.

The function performs a security check first, before it returns a valid handle value. If the access is denied the function returns NULL.

Object Handle Inheritance:

Object handle inheritance can be used only when threads have a parent-child relationship. In this scenario, one or more kernel object handles are available to the parent thread, and the parent decides to spawn a child, giving the child access to the parent's kernel objects.

First, when the parent thread creates a kernel object, the parent must indicate to the system that it wants the object's handle to be inheritable. Although kernel object *handles* are inheritable, kernel objects themselves are not.

To create an inheritable handle, the parent thread must allocate and initialize a SECURITY\_ATTRIBUTES structure and pass the structure's address to the specific *Create* function. The following code creates a mutex object and returns an inheritable handle to it:

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

```
// Make the returned handle inheritable.  
  
HANDLE hMutex = CreateMutex(&sa, FALSE, NULL)  
;Error! Unknown switch argument.
```

Usually, when we spawn a process, we will pass `FALSE` for `bInheritHandles` this parameter. This value tells the system that you do not want the child thread to inherit the inheritable handles that are in the parent threads process's handle table. When we pass `TRUE`, the operating system creates the new child process, but does not allow the child process to begin executing its code right away. Of course, the system creates a new, empty process handle table for the child process—just as it would for any new process. But because we passed `TRUE` to `bInheritHandles` parameter, the system does one more thing: it walks the parent's handle table, and for each entry it finds that contains a valid inheritable handle, the system copies the entry exactly into the child's handle table. The entry is copied to the exact same position in the child's handle table as in the parent's handle table. This fact is important because it means that the handle value that identifies a kernel object is identical in both the parent and the child threads.

In addition to copying the handle table entry, the system increments the usage count of the kernel object because two threads are now using the object. For the kernel object to be destroyed, both the parent threads and the child thread must either call *CloseHandle* on the object or terminate.

### 3.2 *cbStack*

The *cbStack* parameter specifies how much address space the thread can use for its own stack. Every thread owns its own stack. When *CreateProcess* starts a process, it internally calls *CreateThread* to initialize the process's primary thread. For the *cbStack* parameter, *CreateProcess* uses a value stored inside the executable file. You can control this value using the linker's `/STACK` switch:

```
/STACK: [reserve] [, commit]
```

The *reserve* argument sets the amount of address space the system should reserve for the thread's stack. The default is **1 MB**. The *commit* argument specifies the amount of physical storage that should be initially committed to the stack's reserved region. The default is one page. As the code in our thread executes, we might require more than one page of storage. When our thread overflows its stack, an exception is generated. The system catches the exception and commits another page (or whatever we specified for the *commit* argument) to the reserved space, which allows a thread's stack to grow dynamically as needed.

When we call *CreateThread*, passing a value other than 0 causes the function to reserve and commit all storage for the thread's stack. Since all the storage is committed up front, the thread is guaranteed to have the specified amount of stack storage available. The amount of reserved space is either the amount specified by the `/STACK` linker switch or the value of *cbStack*, whichever is larger. The amount of storage committed matches the value we passed for *cbStack*.

If we pass 0 to the *cbStack* parameter, *CreateThread* reserves a region and commits the amount of storage indicated by the /STACK linker switch information embedded in the .exe file by the linker. The reserve amount sets an upper limit for the stack so that we can catch endless recursion bugs in our code.

### 3.3 *pfmStartAddr* and *pvParam*

The *pfmStartAddr* parameter indicates the address of the thread function that we want the new thread to execute. A thread function's *pvParam* parameter is the same as the *pvParam* parameter that we originally passed to *CreateThread*. *CreateThread* does nothing with this parameter except pass it on to the thread function when the thread starts executing. This parameter provides a way to pass an initialization value to the thread function. This initialization data can be either a numeric value or a pointer to a data structure that contains additional information.

It is quite useful to create multiple threads that have the same function address as their starting point. We can even implement a Web server that creates a new thread to handle each client's request. Each thread knows which client it is processing because we pass a different *pvParam* value as we create each thread.



### 3.4 *fdwCreate*

The *fdwCreate* parameter specifies additional flags that control the creation of the thread. It can be one of two values. If the value is 0, the thread is schedulable immediately after it is created. If the value is `CREATE_SUSPENDED`, the system fully creates and initializes the thread but suspends the thread so that it is not schedulable.

### 3.5 *PdwThreadID*

The `CREATE_SUSPENDED` flag allows an application to alter some properties of a thread before it has a chance to execute any code. Because this is rarely necessary, this flag is not commonly used. The last parameter of *CreateThread*, *pdwThreadID*, must be a valid address of a `DWORD` in which *CreateThread* stores the ID that the system assigns to the new thread.

When a process is created, the system automatically creates its first thread, called the *primary thread*. This thread can then create additional threads, and these can in turn create even more threads. When a thread kernel object is created, the system assigns the object a unique, system-wide ID number. Process IDs and thread IDs share the same number pool. This means that it is impossible for a process and a thread to have the same ID. In addition, an object is never assigned an ID of 0. Before `CreateProcess` returns, it fills the *dwProcessId* and *dwThreadId* members of the `PROCESS_INFORMATION` structure with these IDs. IDs simply make it easy for you to identify the processes and threads in the system.



If our application uses IDs to track processes and threads, we must be aware that the system reuses process and thread IDs immediately. If the process identified by the ID is freed and a new process is created and given the same ID.

To guarantee that a process or thread ID isn't reused is to make sure that the process or thread kernel object doesn't get destroyed. If we have just created a new process or thread, we can do this simply by not closing the handles to these objects. Then, once our application has finished using the ID, call *CloseHandle* to release the kernel object(s) and remember that it is no longer safe for us to use or rely on the process ID.



**TERMINATING A  
THREAD**

## 4. TERMINATING A THREAD

A thread can be terminated in four ways:

- The thread function returns.
- The thread kills itself by calling the *ExitThread* function.
- A thread in the same or in another process calls the *TerminateThread* function.
- The process containing the thread terminates.

This section discusses all four methods for terminating a thread and describes what happens when a thread ends.

### 4.1 The Thread Function Returns

We should always design our thread functions so that they return when we want the thread to terminate. This is the only way to guarantee that all our thread's resources are cleaned up properly.

Having our thread function return ensures the following:

- Any and all C++ objects created in our thread function will be destroyed properly via their destructors.
- The operating system will properly free the memory used by the thread's stack.
- The system will set the thread's exit code (maintained in the thread's kernel object) to our thread function's return value.

- The system will decrement the usage count of the thread's kernel Object.

## 4.2 The *ExitThread* Function

The *ExitThread* Function can force our thread to terminate by having it call *ExitThread*:

```
VOID ExitThread(DWORD dwExitCode);
```

This function terminates the thread and causes the operating system to clean up all of the operating system resources that were used by the thread. However, C/C++ resources (such as C++ class objects) will not be destroyed. For this reason, it is much better to simply return from our thread function instead of calling *ExitThread*.

We can use *ExitThread*'s *dwExitCode* parameter to tell the system what to set the thread's exit code to. The *ExitThread* function does not return a value because the thread has terminated and cannot execute any more code.

## 4.3 The *TerminateThread* function

A call to *TerminateThread* also kills a thread:

```
BOOL TerminateThread (  
  
HANDLE hthread,  
  
DWORD dwExitCode);
```

Unlike `ExitThread`, which always kills the calling thread, ***TerminateThread*** can kill any thread. The ***hThread*** parameter identifies the handle of the thread to be terminated. When the thread terminates, its exit code becomes the value you passed as the ***dwExitCode*** parameter. Also, the thread's kernel object has its usage count decremented.

The `TerminateThread` function is asynchronous. That is, it tells the system that we want the thread to terminate but the thread is not guaranteed to be killed by the time the function returns.

A well-designed application never uses this function because the thread being terminated receives no notification that it is dying. The thread cannot clean up properly and it cannot prevent itself from being killed.

DLLs usually receive notifications when a thread is terminating. If a thread is forcibly killed with ***TerminateThread***, however, the DLLs do not receive this notification, which can prevent proper cleanup.

#### 4.4 Process Termination

A process terminates when one of the threads in the process calls ***ExitProcess***:

**VOID ExitProcess (INT fuExitCode);**

This function terminates the process and sets the exit code of the process to ***fuExitCode***. `ExitProcess` doesn't return a value because the process has terminated.

A call to *TerminateProcess* also ends a process:

```
BOOL TerminateProcess(  
    HANDLE hProcess,  
    INT fuExitCode);
```

This function is different from *ExitProcess* in one major way: any thread can call **TerminateProcess** to terminate another process or its own process. The *hProcess* parameter identifies the handle of the process to be terminated. When the process terminates, its exit code becomes the value we passed as the *fuExitCode* parameter.

We should use **TerminateProcess** only if we can't force a process to exit by using another method. A process will not have a chance to do its own cleanup, the operating system does clean up completely after the process so that no operating system resources remain. This means that all memory used by the process is freed, any open files are closed, all kernel objects have their usage counts decremented, and all User and GDI objects are destroyed.

The **ExitProcess** and **TerminateProcess** functions terminate threads. The difference is that these functions terminate all the threads contained in the process being terminated. Also, since the entire process is being shut down, all resources in use by the process are guaranteed to be cleaned up. This certainly includes any and all thread stacks. These two functions cause the remaining threads in the process to be forcibly killed, as if *TerminateThread* were called for each remaining thread. This means that proper application cleanup does not occur: C++ object destructors are not called data is not flushed to disk.

## 4.5 Thread Termination

The following actions occur when a thread terminates:

- All User object handles owned by the thread are freed. A thread owns two User objects: windows and hooks. When a thread dies, the system automatically destroys any windows and uninstalls any hooks that were created or installed by the thread. Other objects are destroyed only when the owning process terminates.
- The thread's exit code changes from `STILL_ACTIVE` to the code passed to *ExitThread* or *TerminateThread*.
- The state of the thread kernel object becomes signaled.
- If the thread is the last active thread in the process, the system considers the process terminated as well.
- The thread kernel object's usage count is decremented by 1.

When a thread terminates, its associated thread kernel object does not automatically become freed until all the outstanding references to the object are closed.

Once a thread is no longer running, there isn't much any other thread in the system can do with the thread's handle. However, these other threads can call *GetExitCodeThread* to check whether the thread identified by *hThread* has terminated and, if it has, determine its exit code.

```
BOOL GetExitCodeThread(HANDLE hThread,  
  
                       PDWORD pdwExitCode);
```

The exit code value is returned in the DWORD pointed to by *pdwExitCode*. If the thread hasn't terminated when *GetExitCodeThread* is called, the function fills the DWORD with the STILL\_ACTIVE identifier. If the function is successful, TRUE is returned.





**THREAD  
INTERNALS**

## 5. THREAD INTERNALS

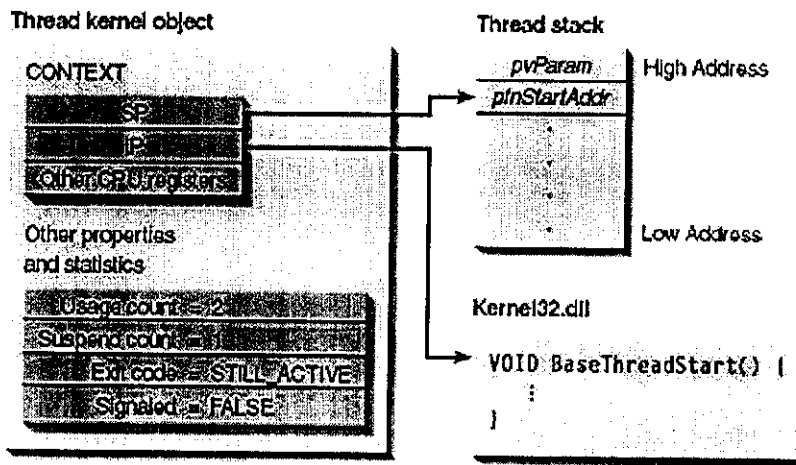
A call to *CreateThread* causes the system to create a thread kernel object. This object has an initial usage count of 2. (The thread kernel object is not destroyed until the thread stops running and the handle returned from *CreateThread* is closed.) Other properties of the thread's kernel object are also initialized:

- The suspension count is set to 1
- The exit code is set to *STILL\_ACTIVE* (0x103)
- The object is set to the nonsignaled state.

Once the kernel object has been created, the system allocates memory, which is used for the thread's stack. This memory is allocated from the process's address space since threads don't have an address space of their own. The system then writes two values to the upper end of the new thread's stack. (Thread stacks always build from high memory addresses to low memory addresses.) The first value written to the stack is the value of the *pvParam* parameter that we passed to *CreateThread*. Immediately below it is the *pfnStartAddr* value that we also passed to *CreateThread*.

Each thread has its own set of CPU registers, called the thread's context. The context reflects the state of the thread's CPU registers when the thread last executed.





*How a thread is created and initialized*

The set of CPU registers for the thread is saved in a CONTEXT structure (defined in the WinNT.h header file). The CONTEXT structure is itself contained in the thread's kernel object.

The instruction pointer and stack pointer registers are the two most important registers in the thread's context. The threads always run in the context of a process. So both these addresses identify memory in the owning process's address space. When the thread's kernel object is initialized, the CONTEXT structure's stack pointer register is set to the address of where *pfnStartAddr* was placed on the thread's stack. The instruction pointer register is set to the address of an undocumented (and unexported) function called *BaseThreadStart*. This function is contained inside the Kernel32.dll module (which is also where the *CreateThread* function is implemented). The above figure shows all of this.

Here is what *BaseThreadStart* basically does:

```
VOID BaseThreadStart(PTHREAD_START_ROUTINE
                    PfnStartAddr, PVOID pvParam)
{
    _try
    {
        ExitThread((pfnStartAddr)(pvParam));
    }
    _except(UnhandledExceptionFilter
           (GetExceptionInformation()))
    {
        ExitProcess(GetExceptionCode());
    }
    // NOTE: We never get here.
}
```

Inside a thread kernel object is a value that indicates the thread's suspend count. When you call `CreateProcess` or `CreateThread`, the thread kernel object is created and the *suspend count* is initialized to 1. This prevents the thread from being scheduled to a CPU. This is desirable because it takes time for the thread to be initialized and we don't want the system to start executing the thread before it is fully ready.

After the thread has completely initialized, the system checks to see whether the `CREATE_SUSPENDED` flag was passed to `CreateThread`. If this flag was not passed, the system decrements the thread's suspend count to 0 and the thread can be scheduled to a processor.

The system then loads the actual CPU registers with the values that were last saved in the thread's context. The thread can now execute code and manipulate data in its process's address space.

Because a new thread's instruction pointer is set to *BaseThreadStart*, this function is really where the thread begins execution. The new thread simply comes into existence and starts executing here. *BaseThreadStart* believes that it was called from another function because it has access to two parameters. But access to these parameters works because the operating system explicitly wrote the values to the thread's stack. The system initializes the proper registers correctly before allowing the thread to execute the *BaseThreadStart* function. When the new thread executes the *BaseThreadStart* function, the following things happen:

- A structured exception handling (SEH) frame is set up around your thread function so that any exceptions raised while our thread executes get some default handling by the system. The system calls our thread function, passing it the *pvParam* parameter that you passed to the *CreateThread* function.
- When the thread function returns, *BaseThreadStart* calls *ExitThread*, passing it the thread function's return value. The thread kernel object's usage count is decremented and the thread stops executing.
- If the thread raises an exception that is not handled, the SEH frame set up by the *BaseThreadStart* function handles the exception.

- Within `BaseThreadStart`, the thread calls either *ExitThread* or *ExitProcess*. This means that the thread cannot ever exit this function; it always dies inside it. This is why *BaseThreadStart* is prototyped as returning `VOID`—it never returns.

The thread function can return when it's done processing because of `BaseThreadStart`. When `BaseThreadStart` calls our thread function, it pushes its return address on the stack so your thread function knows where to return.

When a process's primary thread is initialized, its instruction pointer is set to another undocumented function called *BaseProcessStart*. This function is almost identical to *BaseThreadStart*

The only real difference is that there is no reference to the *pvParam* parameter. When `BaseProcessStart` begins executing, it calls the C/C++ run time library's startup code, which initializes and then calls `main`, `wmain`, `WinMain`, or `wWinMain` function. When your entry-point function returns, the C/C++ run-time library startup code calls `ExitProcess`.

## Gaining a Sense of Thread's Own Identity

As threads execute, they frequently want to call Windows functions that change their execution environment. For example, a thread might want to alter its priority or its process's priority. Since it is common for a thread to alter its (or its process's) environment, Windows offers functions that make it easy for a thread to refer to its process kernel object or to its own thread kernel object:

```
HANDLE GetCurrentProcess();  
HANDLE GetCurrentThread();
```

Both of these functions return a pseudo-handle to the calling thread's process or thread kernel object. These functions do not create new handles in the calling process's handle table. Also, calling these functions has no effect on the usage count of the process or thread kernel object. If we call *CloseHandle*, passing a pseudo-handle as the parameter, *CloseHandle* simply ignores the call and returns FALSE. When we call a Windows function that requires a handle to a process or thread, we can pass a pseudo-handle, which causes the function to perform its action on the calling process or thread. The child thread to get its own CPU times, not the parent thread's CPU times. This happens because a thread pseudo-handle is a handle to the current thread—that is, a handle to whichever thread is making the function call.

A few Windows functions allow us to identify a specific process or thread by its unique system-wide ID. The following functions allow a thread to query its process's unique ID or its own unique ID:

```
DWORD GetCurrentProcessId();  
DWORD GetCurrentThreadId();
```

These functions are generally not as useful as the functions that return pseudo-handles, but occasionally they come in handy.



**THREAD  
SCHEDULING**





## 6. THREAD SCHEDULING

In a preemptive multithreaded operating system a thread can be stopped at any time and another thread can be scheduled. We have some control over this, but not much. We cannot guarantee that our thread will always be running, that our thread will get the whole processor, that no other thread will be allowed to run.

The system only schedules schedulable threads, but as it turns out, most of the threads in the system are not schedulable. For example, some thread objects might have a suspend count greater than 0. This means that the thread is suspended and should not be scheduled any CPU time. We can create a suspended thread by calling *CreateProcess* or *CreateThread* using the `CREATE_SUSPENDED` flag.

In addition to suspended threads, many other threads are not schedulable because they are waiting for something to happen. The system does not assign CPU time to threads that have nothing to do.

### 6.1 Suspending and Resuming a Thread

Inside a thread kernel object is a value that indicates the thread's suspend count. When we call *CreateProcess* or *CreateThread*, the thread kernel object is created and the suspend count is initialized to 1. This prevents the thread from being scheduled to a CPU. This is desirable because it takes time for the thread to be initialized and we don't want the system to start executing the thread before it is fully ready.

After the thread is fully initialized, *CreateProcess* or *CreateThread* checks to see whether we've passed the `CREATE_SUSPENDED` flag. If we have, the functions return and the new thread is left in the suspended state. If we have not, the function decrements the thread's suspend count to 0. When a thread's suspend count is 0, the thread is schedulable unless it is waiting for something else to happen (such as keyboard input).

Creating a thread in the suspended state allows us to alter the thread's environment (such as priority, discussed later in the chapter) before the thread has a chance to execute any code. Once we alter the thread's environment, we must make the thread schedulable. We do this by calling *ResumeThread* and passing it the thread handle returned by the call to *CreateThread*.

```
DWORD ResumeThread(HANDLE hThread);
```

If *ResumeThread* is successful, it returns the thread's previous suspend count; otherwise, it returns `0xFFFFFFFF`.

A single thread can be suspended several times. If a thread is suspended three times, it must be resumed three times before it is eligible for assignment to a CPU. In addition to using the `CREATE_SUSPENDED` flag when you create a thread, you can suspend a thread by calling *SuspendThread*:

```
DWORD SuspendThread(HANDLE hThread);
```

Any thread can call this function to suspend another thread (as long as we have the thread's handle).

A thread can suspend itself but cannot resume itself. Like `ResumeThread`, *`SuspendThread`* returns the thread's previous suspend count. A thread can be suspended as many as `MAXIMUM_SUSPEND_COUNT` times (defined as 127 in `WinNT.h`).

## 6.2 Sleeping

A thread can also tell the system that it does not want to be schedulable for a certain amount of time. This is accomplished by calling *`Sleep`*:

```
VOID Sleep(DWORD dwMilliseconds);
```

This function causes the thread to suspend itself until *`dwMilliseconds`* have elapsed. There are a few important things to notice about *`Sleep`*:

- Calling *`Sleep`* allows the thread to voluntarily give up the remainder of its time slice.
- The system makes the thread **not schedulable** for approximately the number of milliseconds specified, possibly several seconds or minutes more. Our thread will probably wake up at the right time, but whether it does depends on what else is going on in the system.
- We can call `Sleep` and pass `INFINITE` for the `dwMilliseconds` parameter. It is much better to have the thread exit and to recover its stack and kernel object.

- We can pass 0 to *Sleep*. This tells the system that the calling thread relinquishes the remainder of its time slice and forces the system to schedule another thread. However, the system can reschedule the thread that just called *Sleep*. This will happen if there are no more schedulable threads at the same priority.

### 6.3 Switching to Another Thread

The system offers a function called *SwitchToThread* that allows another schedulable thread to run if one exists:

```
BOOL SwitchToThread();
```

When we call this function, the system checks to see whether there is a thread that is being starved of CPU time. If no thread is starving, *SwitchToThread* returns immediately. If there is a starving thread, *SwitchToThread* schedules that thread (which might have a lower priority than the thread calling *SwitchToThread*). The starving thread is allowed to run for one time quantum and then the system scheduler operates as usual.

This function allows a thread that wants a resource to force a lower-priority thread that might currently own the resource to relinquish the resource. If no other thread can run when *SwitchToThread* is called, the function returns `FALSE`; otherwise, it Calling *SwitchToThread* is similar to calling *Sleep* and passing it a timeout of 0 milliseconds. The difference is that *SwitchToThread* allows lower-priority threads to execute. *Sleep* reschedules the calling thread immediately even if lower-priority threads are being starved. returns a nonzero value.

## 6.4 Context in Context

The context structure plays an important role thread scheduling. The context structure allows the system to remember a thread's state so that the thread can pick up where it left off the next time it has a CPU to run on.

A CONTEXT structure contains processor-specific register data. The system uses CONTEXT structures to perform various internal operations. Currently, there are CONTEXT structures defined for Intel, MIPS, Alpha, and PowerPC processors. The CONTEXT structure is the only data structure that is CPU-specific.

CONTEXT structure contains a data member for each register on the host CPU. On an x86 machine, the members are *Eax*, *Ebx*, *Ecx*, *Edx*, and so on. For the Alpha processor, the members are *IntV0*, *IntT0*, *IntT1*, *IntS0*, *IntRa*, *IntZero*, and so on.

A CONTEXT structure has several sections. CONTEXT\_CONTROL contains the control registers of the CPU, such as the instruction pointer, stack pointer, flags, and function return address.

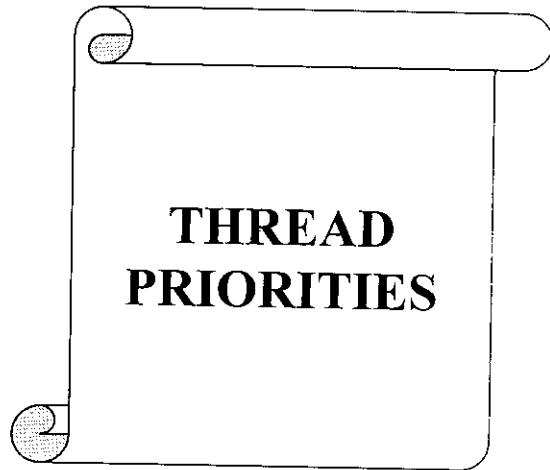
CONTEXT\_INTEGER identifies the CPU's integer registers;

CONTEXT\_FLOATING\_POINT identifies the CPU's floating-point registers;

CONTEXT\_SEGMENTS identifies the CPU's segment registers

CONTEXT\_DEBUG\_REGISTERS identifies the CPU's debug registers

CONTEXT\_EXTENDED\_REGISTERS identifies the CPU's extended registers.



**THREAD  
PRIORITIES**

## 7.THREAD PRIORITIES

Threads are assigned a lot of different priorities and this affects which thread the scheduler picks as the next thread to run. Every thread is assigned a priority number ranging from 0 (the lowest) to 31 (the highest). When the system decides which thread to assign to a CPU, it examines the priority 31 threads first and schedules them in a round-robin fashion. If a priority 31 thread is schedulable, it is assigned to a CPU. At the end of this thread's time slice, the system checks to see whether there is another priority 31 thread that can run; if so, it allows that thread to be assigned to a CPU.

As long as a priority 31 thread is schedulable, the system never assigns any thread with a priority of 0 through 30 to a CPU. This condition is called *starvation*. Starvation occurs when higher-priority threads use so much CPU time that they prevent lower-priority threads from executing. Starvation is much less likely to occur on a multiprocessor machine because on such a machine a priority 31 thread and a priority 30 thread can run simultaneously. The system always tries to keep the CPUs busy, and CPUs sit idle only if no threads are schedulable.

Higher-priority threads always preempt lower-priority threads, regardless of what the lower-priority threads are executing. For example, if a priority 5 thread is running and the system determines that a higher-priority thread is ready to run, the system immediately suspends the lower-priority thread and assigns the CPU to the higher-priority thread, which gets a full time slice.

By the way, when the system boots, it creates a special thread called the *zero page thread*. This thread is assigned priority 0 and is the only thread in the entire system that runs at priority 0. The zero page thread is responsible for zeroing any free pages of RAM in the system when there are no other threads that need to perform work. When we design an application, we should choose a priority class based on how responsive we need the threads in your application to be.

## 7.1 Process Priority

When we design an application, we should choose a priority class based on how responsive we need the threads in your application to be. Windows supports six priority classes: idle, below normal, normal, above normal, high, and real-time. Normal is the most common priority class and is used by 99 percent of the applications out there. The table below describes the priority classes.

Priority Class	Description
Real-time	The threads in this process must respond immediately to events in order to execute time-critical tasks. Threads in this process also preempt operating system components. Use this priority class with extreme caution.
High	The threads in this process must respond immediately to events in order to execute time-critical tasks. The Task Manager runs at this class so a user can kill runaway processes.
Above normal	The threads in this process run between the normal and high priority classes (new in Windows 2000).
Normal	The threads in this process have no special scheduling needs.
Below normal	The threads in this process run between the normal and idle priority classes (new in Windows 2000).
Idle	The threads in this process run when the system is otherwise idle. This process is typically used by screensavers or background utility and statistic-gathering software.

Table 1 : Priority Classes



The idle priority class is perfect for applications that run when the system is all but doing nothing. Statistic-tracking applications that periodically update some state about the system usually should not interfere with more critical tasks.

We should use the high priority class only when absolutely necessary. We should avoid the real-time priority class if possible. Real-time priority is extremely high and can interfere with operating systems tasks because most operating system threads execute at a lower priority. So real-time threads can prevent required disk I/O and network traffic from occurring. In addition, keyboard and mouse input are not processed in a timely manner; the user might think that the system is hung. Basically, we should have a good reason for using real-time priority, such as the need to respond to hardware events with short latency or to perform some short-lived task that just can't be interrupted.

A process cannot run in the real-time priority class unless the user has the Increase Scheduling Priority privilege. Any user designated an administrator or a power user has this privilege by default.

Most processes are part of the normal priority class. The two other priority classes, below normal and above normal along with these priority classes offer enough flexibility.

## 7.2 Thread Priority

Windows supports seven relative thread priorities: idle, lowest, below normal, normal, above normal, highest, and time-critical. These priorities are relative to the process's priority class. Our process is part of a priority class and we assign the threads within the process relative thread priorities. Application developers never work with priority levels. Instead, the system maps the process's priority class and a thread's relative priority to a priority level. Again, most threads use the normal thread priority.

The table below describes the relative thread priorities.

Relative Thread Priority	Description
Time-critical	Thread runs at 31 for the real-time priority class and at 15 for all other priority classes.
Highest	Thread runs two levels above normal.
Above normal	Thread runs one level above normal.
Normal	Thread runs normally for the process's priority class.
Below normal	Thread runs one level below normal.
Lowest	Thread runs two levels below normal.
Idle	Thread runs at 16 for the real-time priority class and at 1 for all other priority classes.

Table 2 : Relative Thread Priorities

For example, a normal thread in a normal process is assigned a priority level of 8. Since most processes are of the normal priority class and most threads are of normal thread priority, most threads in the system have a priority level of 8.

If we have a normal thread in a high-priority process, the thread will have a priority level of 13. If you change the process's priority class to idle, the thread's priority level becomes 4. Remember that thread priorities are relative to the process's priority class. If we change a process's priority class, the thread's relative priority will not change but its priority level will.

Relative Thread Priority	Process Priority Class					
	Idle	Below Normal	Normal	Above Normal	High	Real-time
Time-critical	15	15	15	15	15	31
Highest	6	8	10	12	15	26
Above normal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
Below normal	3	5	7	9	12	23
Lowest	2	4	6	8	11	22
Idle	1	1	1	1	1	16

Table 3 : Process Priority with corresponding Thread priorities

The table above does not show any way for a thread to have a priority level of 0. This is because the 0 priority is reserved for the zero page thread and the system does not allow any other thread to have a priority of 0. Also, the following priority levels are not obtainable: 17, 18, 19, 20, 21, 27, 28, 29, or 30. If we are writing a device driver that runs in kernel mode, we can obtain these levels; a user-mode application cannot. Also note that a thread in the real-time priority class can't be below priority level 16. Likewise, a thread in a non-real-time priority class cannot be above 15.

In general, a thread with a high priority level should not be schedulable most of the time. When the thread has something to do, it quickly gets CPU time.

At this point, the thread should execute as few CPU instructions as possible and go back to sleep, waiting to be schedulable again. In contrast, a thread with a low priority level can remain schedulable and execute a lot of CPU instructions to do its work.

### 7.3 Programming Priorities

A process is assigned a priority class when we call *CreateProcess*, we can pass the desired priority class in the *fdwCreate* parameter. The table below shows the priority class identifiers.

Priority Class	Symbolic Identifiers
Real-time	REALTIME_PRIORITY_CLASS
High	HIGH_PRIORITY_CLASS
Above normal	ABOVE_NORMAL_PRIORITY_CLASS
Normal	NORMAL_PRIORITY_CLASS
Below normal	BELOW_NORMAL_PRIORITY_CLASS
Idle	IDLE_PRIORITY_CLASS

Table 4 : Symbolic Identifiers for Threads

It might seem odd that the process that creates a child process chooses the priority class at which the child process runs. However once the child process is running, it can change its own priority class by calling *SetPriorityClass*:

```

BOOL SetPriorityClass(HANDLE hProcess,
                    DWORD fdwPriority);

```

This function changes the priority class identified by `hProcess` to the value specified in the `fdwPriority` parameter. The `fdwPriority` parameter can be one of the identifiers shown in the table above.

Because this function takes a process handle, we can alter the priority class of any process running in the system as long as we have a handle to it and sufficient access.

Here is the complementary function used to retrieve the priority class of a process:

```
DWORD GetPriorityClass(HANDLE hProcess);
```

This function returns one of the identifiers listed in the table above.

When a thread is first created, its relative thread priority is always set to normal. To set and get a thread's relative priority, we must call these functions:

```
BOOL SetThreadPriority(  
    HANDLE hThread,  
    int nPriority);
```

The ***hThread*** parameter identifies the single thread whose priority we want to change, and the ***nPriority*** parameter is one of the seven identifiers listed in the following table.

Relative Thread Priority	Symbolic Constant
Time-critical	THREAD_PRIORITY_TIME_CRITICAL
Highest	THREAD_PRIORITY_HIGHEST
Above normal	THREAD_PRIORITY_ABOVE_NORMAL
Normal	THREAD_PRIORITY_NORMAL
Below normal	THREAD_PRIORITY_BELOW_NORMAL
Lowest	THREAD_PRIORITY_LOWEST
Idle	THREAD_PRIORITY_IDLE

Table 5 : Relative Thread Priorities and Symbolic Constants

Here is the complementary function for retrieving a thread's relative priority:

```
Int GetThreadPriority(HANDLE hThread);
```

The *CreateThread* always creates a new thread with a normal relative thread priority. To have the thread execute using idle priority, we pass the `CREATE_SUSPENDED` flag to `CreateThread`; this prevents the thread from executing any code at all. Then we call `SetThreadPriority` to change the thread to an idle relative thread priority. We then call `ResumeThread` so that the thread can be schedulable. The scheduler takes into account the fact that this thread has an idle thread priority. Finally, we close the handle to the new thread so that the kernel object can be destroyed as soon as the thread terminates.

## 8. THREAD SYNCHRONIZATION WITH KERNEL OBJECTS

Kernel objects can be used to synchronize threads. We've discussed kernel objects of threads. For thread synchronization, each of these kernel objects is said to be in a signaled or nonsignaled state. The toggling of this state is determined by rules that Microsoft has created for each object. Process kernel objects are always created in the nonsignaled state. When the process terminates, the operating system automatically makes the process kernel object signaled. Once a process kernel object is signaled, it remains that way forever; its state never changes back to nonsignaled.

A process kernel object is nonsignaled while the process is running, and it becomes signaled when the process terminates. Inside a process kernel object is a Boolean value that is initialized to FALSE (nonsignaled) when the object is created. When the process terminates, the operating system automatically changes the corresponding object's Boolean value to TRUE, indicating that the object is signaled.

If we want to write code that checks whether a process is still running, all we do is call a function that asks the operating system to check the process object's Boolean value. We may also tell the system to put our thread in a wait state and wake it up automatically when the Boolean changes from FALSE to TRUE.

This way, we can write code in which a thread in a parent process that needs to wait for the child process to terminate can simply put itself to sleep until the kernel object identifying the child process becomes signaled.

Thread kernel objects are always created in the nonsignaled state. When the thread terminates, the operating system automatically changes the thread object's state to signaled. Therefore, we can use the same technique in our application to determine whether a thread is no longer executing. Just like process kernel objects, thread kernel objects never return to the nonsignaled state.

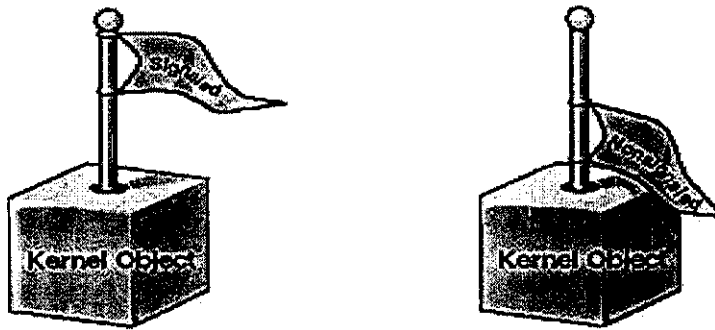
The following kernel objects can be in a signaled or nonsignaled state:

- Processes
- Threads
- Jobs
- Files
- Console input
- File change notifications
- Events
- Waitable timers
- Semaphores
- Mutexes

Threads can put themselves into a wait state until an object becomes signaled. The rules that govern the signaled/nonsignaled state of each object depend on the type of object. We'll look at the functions that allow a thread to wait for a specific kernel object to become signaled.

Kernel objects can be imagined as flag (the wave-in-the-air kind, not the bit kind). When the object was signaled, the flag was raised; when the object was nonsignaled, the flag was lowered.





Signaled & Nonsignaled States of Thread

## 8.1 Wait Functions

*Wait functions* cause a thread to voluntarily place itself into a wait state until a specific kernel object becomes signaled. By far the most common of these functions is *WaitForSingleObject*:

```
DWORD WaitForSingleObject(  
    HANDLE hObject,  
    DWORD dwMilliseconds);
```

When a thread calls this function, the first parameter, *hObject*, identifies a kernel object that supports being signaled/nonsignaled. The second parameter, *dwMilliseconds*, allows the thread to indicate how long it is willing to wait for the object to become signaled.

The following function call tells the system that the calling thread wants to wait until the process identified by the *hProcess* handle terminates:

```
WaitForSingleObject(hProcess, INFINITE);
```

The second parameter tells the system that the calling thread is willing to wait forever (an infinite amount of time) until this process terminates. Usually, INFINITE is passed as the second parameter to *WaitForSingleObject*, but we can pass any value (in milliseconds). By the way, INFINITE is defined as 0xFFFFFFFF (or -1).

Passing INFINITE can be a little dangerous. If the object never becomes signaled, the calling thread never wakes up—it is forever deadlocked but, fortunately, not wasting precious CPU time. We can pass 0 for the dwMilliseconds parameter. If we do this, WaitForSingleObject always returns immediately.

*WaitForSingleObject's* return value indicates why the calling thread became schedulable again. If the object the thread is waiting on became signaled, the return value is WAIT\_OBJECT\_0; if the timeout expires, the return value is WAIT\_TIMEOUT. If we pass a bad parameter (such as an invalid handle) to WaitForSingleObject, the return value is WAIT\_FAILED.

The function *WaitForMultipleObjects*, is similar to *WaitForSingleObject* except that it allows the calling thread to check the signaled state of several kernel objects simultaneously:

```
DWORD WaitForMultipleObjects(  
    DWORD dwCount,  
    CONST HANDLE* phObjects,  
    BOOL fWaitAll,  
    DWORD dwMilliseconds);
```

The *dwCount* parameter indicates the number of kernel objects we want the function to check. This value must be between 1 and `MAXIMUM_WAIT_OBJECTS` (defined as 64 in the Windows header files). The *phObjects* parameter is a pointer to an array of kernel object handles.

We can use *WaitForMultipleObjects* in two different ways—to allow a thread to enter a wait state until any one of the specified kernel objects becomes signaled, or to allow a thread to wait until the entire specified kernel objects become signaled. The *fWaitAll* parameter tells the function which way we want it to work. If we pass `TRUE` for this parameter, the function will not allow the calling thread to execute until all of the objects have become signaled.

The *dwMilliseconds* parameter works exactly as it does for *WaitForSingleObject*. If, while waiting, the specified time expires, the function returns anyway. Again, `INFINITE` is usually passed for this parameter, but you should write our code carefully to avoid the possibility of deadlock.

The *WaitForMultipleObjects* function's return value tells the caller why it got rescheduled. The possible return values are `WAIT_FAILED` and `WAIT_TIMEOUT`. If we pass `TRUE` for *fWaitAll* and all of the objects become signaled, the return value is `WAIT_OBJECT_0`. If we pass `FALSE` for *fWaitAll*, the function returns as soon as any of the objects becomes signaled. In this case, we probably want to know which object became signaled. The return value is a value between `WAIT_OBJECT_0` and  $(\text{WAIT\_OBJECT\_0} + \text{dwCount} - 1)$ .

In other words, if the return value is not `WAIT_TIMEOUT` and is not `WAIT_FAILED`, we should subtract `WAIT_OBJECT_0` from the return value. The resulting number is an index into the array of handles that we passed as the second parameter to *WaitForMultipleObjects*. The index tells us which object became signaled.

For example, our thread might be waiting for three child processes to terminate by passing three process handles to this function. If the process at index 0 in the array terminates, *WaitForMultipleObjects* returns. Now the thread can do whatever it needs to and then loop back around, waiting for another process to terminate. If the thread passes the same three handles, the function returns immediately with `WAIT_OBJECT_0` again. Unless we remove the handles that we've already received notifications from, our code will not work correctly.

A graphic of a scroll with a title. The scroll is rectangular with rounded corners and a horizontal bar at the top. The top-left and bottom-left corners are rolled up, showing a shaded interior. The title is centered on the scroll.

**THREAD  
SYNCHRONIZATION  
USING  
CRITICAL SECTION**

## 9. THREAD SYNCHRONIZATION USING CRITICAL SECTION

A *critical section* is a small section of code that requires exclusive access to some shared resource before the code can execute. This is a way to have several lines of code "atomically" manipulate a resource. By atomic, we mean that the code knows that no other thread will access the resource. Of course, the system can still preempt our thread and schedule other threads. However, it will not schedule any other threads that want to access the same resource until our thread leaves the critical section.

While managing a linked list of objects, if access to the linked list is not synchronized, one thread can add an item to the list while another thread is trying to search for an item in the list. The situation can become more chaotic if the two threads add items to the list at the same time. By using critical sections, we can ensure that access to the data structures is coordinated among threads.

When we have a resource that is accessed by multiple threads, we should create a `CRITICAL_SECTION` structure. Since the critical section is small, only one thread at a time can be inside the critical section using the protected resource.

If we have multiple resources that are always used together, we can place them all in a single critical section: we should create just one `CRITICAL_SECTION` structure to guard them all.

If we have multiple resources that are not always used together—for example, threads 1 and 2 access one resource and threads 1 and 3 access another resource—you should create a separate `CRITICAL_SECTION` structure, for each resource.

Now, wherever we have code that touches a resource, we must place a call to *EnterCriticalSection*, passing it the address of the `CRITICAL_SECTION` structure that identifies the resource. This is like saying that when a thread wants to access a resource. The `CRITICAL_SECTION` structure identifies which thread wants to enter and the *EnterCriticalSection* function is what the thread uses to check the occupied sign.

If *EnterCriticalSection* sees that no other thread is in the critical section, the calling thread is allowed to use it. If *EnterCriticalSection* sees that another thread is in the critical section, the calling thread must wait outside the critical section until the other thread in the critical section leaves.

When a thread no longer executes code that touches the resource, it should call *LeaveCriticalSection*. This is how the thread tells the system that it has left the critical section containing the resource. If we forget to call *LeaveCriticalSection*, the system will think that the resource is still in the Critical Section and will not allow any waiting threads in.

Any code we write that touches a shared resource must be wrapped inside *EnterCriticalSection* and *LeaveCriticalSection* functions. If we forget to wrap your code in just one place, the shared resource will be subject to corruption.

If we forget calls to *EnterCriticalSection* & *LeaveCriticalSection*, the thread just muscles its way in and manipulates the resource. If just one thread exhibits this, the resource is corrupted.

We can solve our synchronization problem using critical sections. Critical sections are easy to use and they use the interlocked functions internally, so they execute quickly. The major disadvantage of critical sections is that we cannot use them to synchronize threads in multiple processes.

To manipulate a `CRITICAL_SECTION` structure, we call a Windows function, passing it the address of the structure. The function knows how to manipulate the members and guarantees that the structure's state is always consistent. So now, let's turn our attention to these functions.

Normally, `CRITICAL_SECTION` structures are allocated as global variables to allow all threads in the process an easy way to refer the structure by variable name. However, `CRITICAL_SECTION` structures can be allocated as local variables or dynamically allocated from a heap. There are just two requirements. The first is that all threads that want to access the resource must know the address of the `CRITICAL_SECTION` structure that protects the resource. We can get this address to these threads using any mechanism we like.





*EnterCriticalSection* examines the member variables inside the structure. The variables indicate which thread, if any, is currently accessing the resource. *EnterCriticalSection* performs the following tests:

- If no thread is accessing the resource, *EnterCriticalSection* updates the member variables to indicate that the calling thread has been granted access and returns immediately, allowing the thread to continue executing (accessing the resource).
- If the member variables indicate that the calling thread was already granted access to the resource, *EnterCriticalSection* updates the variables to indicate how many times the calling thread was granted access and returns immediately, allowing the thread to continue executing. This situation is rare and occurs only if the thread calls *EnterCriticalSection* twice in a row without an intervening call to *LeaveCriticalSection*.
- If the member variables indicate that a thread (other than the calling thread) was granted access to the resource, *EnterCriticalSection* places the calling thread in a wait state. The waiting thread does not waste any CPU time. The system remembers that the thread wants access to the resource and automatically updates the `CRITICAL_SECTION`'s member variables and allows the thread to be schedulable as soon as the thread currently accessing the resource calls *LeaveCriticalSection*.



*EnterCriticalSection* is not too complicated internally; it performs just a few simple tests. This function is so valuable is that it can perform all of these tests atomically. If two threads call *EnterCriticalSection* at exactly the same time on a multiprocessor machine, the function still behaves correctly: one thread is granted access to the resource, and the other thread is placed in a wait state.

If *EnterCriticalSection* places a thread in a wait state, the thread might not be scheduled again for a long time. In fact, in a poorly written application, the thread might never be scheduled CPU time again. If this happens, the thread is said to be *starved*.

At the end of our code that touches the shared resource, we must call :

```
VOID LeaveCriticalSection(  
    PCRITICAL_SECTION pcs);
```

*LeaveCriticalSection* examines the member variables inside the structure. The function decrements by 1 a counter that indicates how many times the calling thread was granted access to the shared resource. If the counter is greater than 0, *LeaveCriticalSection* does nothing else and simply returns.

If the counter becomes 0, it checks to see whether any other threads are waiting in a call to *EnterCriticalSection*. If at least one thread is waiting, it updates the member variables and makes one of the waiting threads schedulable again. If no threads are waiting, *LeaveCriticalSection* updates the member variables to indicate that no thread is accessing the resource. Like *EnterCriticalSection*, *LeaveCriticalSection* performs all of these tests and updates atomically.

If we forget to call *CloseHandle*, it is possible for a process to leak resources (such as kernel objects) while the process runs. However, when the process terminates, the operating system ensures that any and all resources used by the process are freed—this is guaranteed.

For kernel objects, the system performs the following actions: When our process terminates, the system automatically scans the process's handle table. If the table has any valid entries (objects that we didn't close before terminating), the system closes these object handles for us. If the usage count of any of these objects goes to zero, the kernel destroys the object.

So, our application can leak kernel objects while it runs, but when our process terminates, the system guarantees that everything is cleaned up properly. By the way, this is true for all objects, resources, and memory blocks: when a process terminates, the system ensures that our process leaves nothing behind.



**CONCLUSION**

## **CONCLUSION**

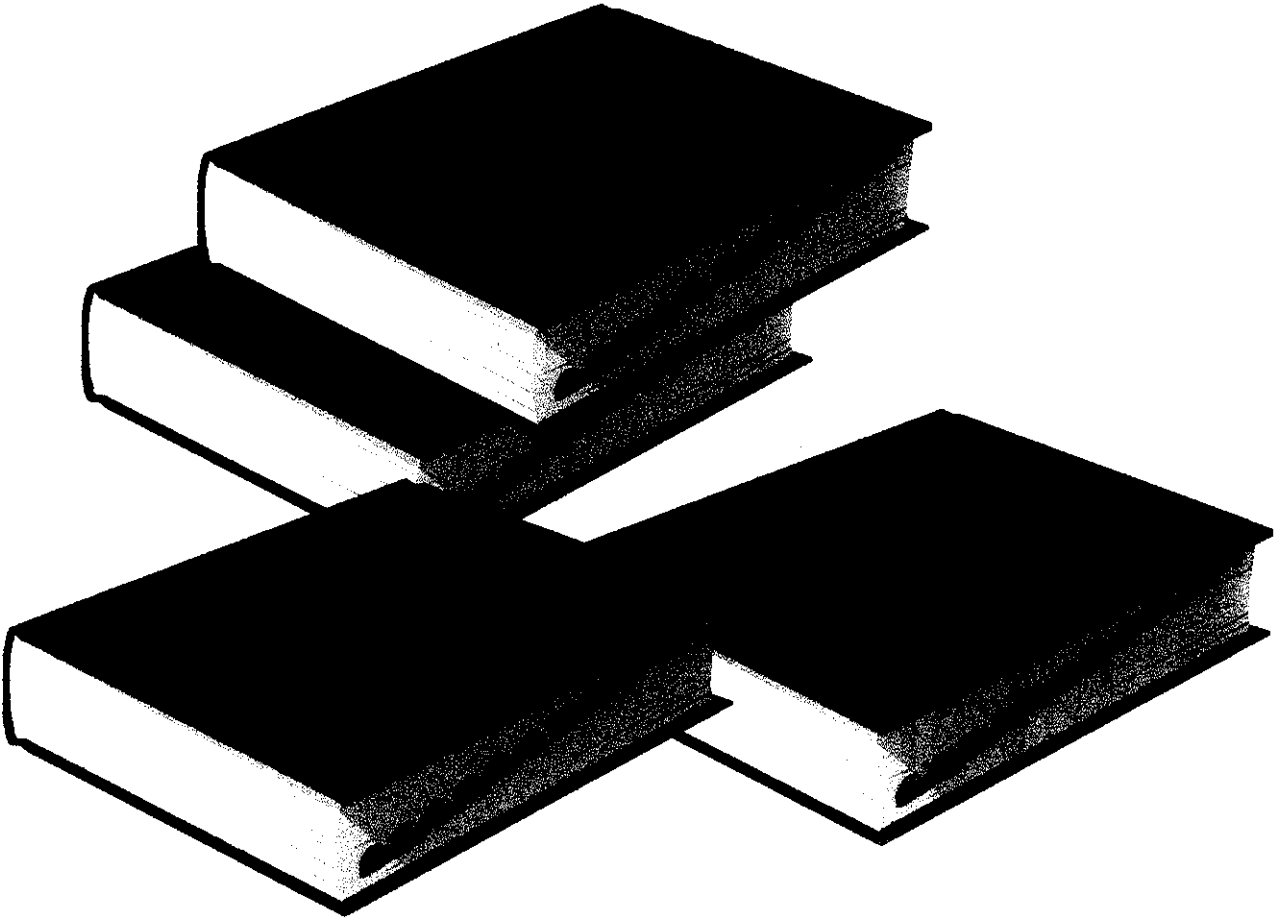
A wrapper class for platform specific thread APIs is developed. If the application is using our thread library the application can be ported to any platform as such. So the only module that we have to modify is our thread library, which will take very less time and manpower. Since the application has to be recoded according to the working platform it makes the application unportable, where as the thread library makes the application portable.

## **Scope for future development**

We can create a thread manager, which contains the centralized control of all the thread in the application. We can also create some utilities like queues and we can implement the thread synchronization algorithm in it,so that the data inside those queues can be protected from data corruption which can be caused by the scenario like two thread attempting to write data in to the queue.

## **BIBLIOGRAPHY**

1. Jeffrey Richter "Programming Applications for Microsoft Windows" Microsoft Press, Fourth Edition.
2. Charles Petzold "Programming Windows" Microsoft Press, Fifth Edition.
3. Robert Lafore "Windows Programming Made Easy", Galgotia Publications pvt. Ltd.
4. Richard J. Simon "Win32 API SuperBible", Wait Group Press.

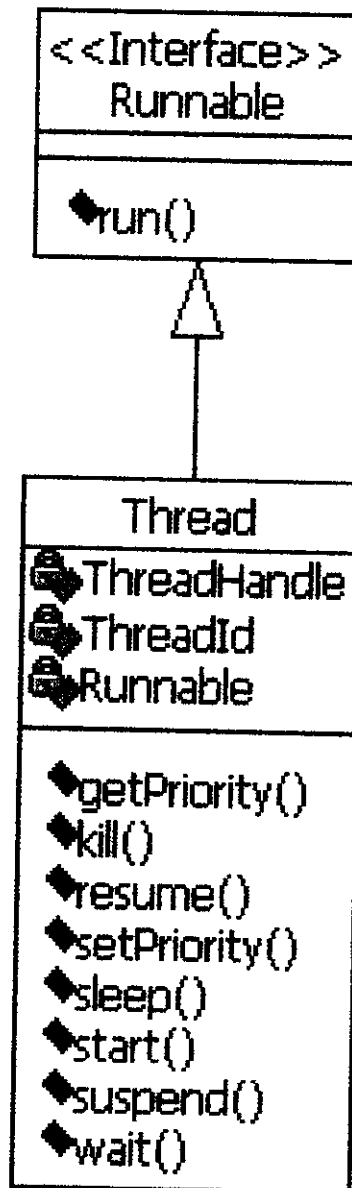


# **BIBLIOGRAPHY**



## APPENDIX – A

### The Thread class diagram



## Thread class implementation

### The Thread class

The function `StartProc` does Start up procedure for thread. Its input argument is a long Pointer to `Void`, no output arguments and the return value is a `DWORD`.

The function `Thread` has a default constructor; it has no input, output arguments and no return value.

The `Thread` function also has a constructor with initializer whose input argument is a pointer to `Runnable` interface. It has no input, output arguments.

The function `start()` Starts the thread; it has no input, output arguments and no return value.

The function `kill()` ends the thread life; it has no input, output arguments and no return value.

The function `suspend()` suspends the execution of the thread; it has no input, output arguments and no return value.

The function `resume()` resumes the execution of the thread; it has no input, output arguments and no return value.

The function `setPriority()` sets the priority for the thread; its input argument is an integer value, which specifies the priority for the thread.

The function `getPriority()` returns the priority of a thread which is an integer value; it has no input, output arguments.

The function `sleep()` makes the thread to move to the sleeping state for specific duration. Its input argument is an integer value, which specifies the duration for which the thread has to sleep and it is an integer value.

The function `wait()` makes the other threads to wait until this thread ends. Its input argument is an integer value, which specifies the duration of time for which the other threads has to wait.

### The Runnable class

This class is an interface which defines the signature for `run()`. The `run()` forces to provide the task for the thread to execute. We can Create a class by extending this class `Runnable` and provide an implementation for `run()`.

### **Thread header file (Thread.h)**

The `Thread` class represents the thread. This provides a lot of members to manipulate the threads in a more effective way. This class can be used to create a thread by passing `Runnable` object pointer to the `Thread`'s constructor. After the thread object creation, start the thread by invoking `start()` method of the thread class. The class `Runnable` can be used to implement the thread.

/\*\* **THREAD.CPP** \*\*\*/

```
#include <Thread.h>
```

```
#include <iostream>
```

```
using namespace std;
```

```
////////////////////////////////////////////////////////////////
```

```
// Function Name: StartProc
```

```
// Description : Start up procedure for thread.
```

```
// Input Args : Long Pointer to Void
```

```
// Output Args : None
```

```
// Return Value : DWORD
```

```
////////////////////////////////////////////////////////////////
```

```
DWORD WINAPI Thread::StartProc(LPVOID vpv)
```

```
{ Runnable* r = (Runnable*) vpv;
```

```
  r->run();
```

```
  return 0;
```

```
}
```

```
////////////////////////////////////
```

```
// Function Name: Thread
```

```
// Description : Constuctor - Default.
```

```
// Input Args : None
```

```
// Output Args : None
```

```
// Return Value : None
```

```
////////////////////////////////////
```

```
Thread::Thread()
```

```
{    m_Runable = this;
```

```
}
```

```
////////////////////////////////////
```

```
// Function Name: Thread
```

```
// Description : constructor with initializer
```

```
// Input Args : Runnable*
```

```
// Output Args : None
```

```
// Return Value : None
```

```
////////////////////////////////////
```

```

Thread::Thread(Runnable* a_Runnable)

{
    m_Runnable = a_Runnable;
}

////////////////////////////////////

// Function Name: start

// Description : Starts the thread.

// Input Args : None

// Output Args : None

// Return Value : None

////////////////////////////////////

void Thread::start()

{ m_ThrdHdl = CreateThread(NULL,

                            0,

                            StartProc,

                            m_Runnable,

                            0,

                            &m_ThreadId);

```

```
if(m_ThrdHdl == NULL)
{
    cout << "ERROR: Thread Creation Failed" << endl;
}
}
```

```
////////////////////////////////////
```

```
// Function Name: kill
// Description : Ends thread's life.
// Input Args : None
// Output Args : None
// Return Value : None
```

```
////////////////////////////////////
```

```
void Thread::kill()
{ if(m_ThrdHdl != NULL)
{
    ExitThread(10);
}
}
```



```
////////////////////////////////////
```

```
// Function Name: suspend
```

```
// Description : Suspends the execution of this thread.
```

```
// Input Args : None
```

```
// Output Args : None
```

```
// Return Value : None
```

```
////////////////////////////////////
```

```
void Thread::suspend()
```

```
{
```

```
    if (m_ThrdHdl != NULL)
```

```
    {
```

```
        int rv = SuspendThread(m_ThrdHdl);
```

```
        if (rv == -1)
```

```
        {
```

```
            cout << "Suspend Thread operation failed" << endl;
```

```
        }
```



```
else
{
    cout << "Suspend count = "
        << rv
        << " Max. suspend count = "
        << MAXIMUM_SUSPEND_COUNT << endl;
}
}
}
```

////////////////////////////////////

```
// Function Name: resume
// Description : Resumes the execution of this thread.
// Input Args : None
// Output Args : None
// Return Value : None
```

////////////////////////////////////

```
void Thread::resume()
{
    if (m_ThrdHdl != NULL)
    {
        int rv = ResumeThread(m_ThrdHdl);

        if (rv == -1)
        {
            cout << "Resume Thread operation failed" << endl;
        } else
        {
            cout << "Thread resumed. Suspend count = "
                << rv << endl;
        }
    }
}
```

```
////////////////////////////////////
```

```
// Function Name: setPriority
```

```
// Description : Sets the priority for this thread.
```

```
// Input Args : int - thread priority
```

```
// Output Args : None
```

```
// Return Value : None
```

```
////////////////////////////////////
```

```
void Thread::setPriority(int a_nPriority)
```

```
{
```

```
    if (m_ThrdHdl != NULL)
```

```
    {
```

```
        SetThreadPriority(m_ThrdHdl, a_nPriority);
```

```
    }
```

```
}
```

```
////////////////////////////////////
```

```
// Function Name: getPriority
```

```
// Description : Returns the priority of this thread.
```

```
// Input Args : None
```

```
// Output Args : None
```

```
// Return Value : int - Thread priority
```

```
////////////////////////////////////
```

```
int Thread::getPriority()
```

```
{
```

```
    int rv = 0;
```

```
    if (m_ThrdHdl != NULL)
```

```
    {
```

```
        rv = GetThreadPriority(m_ThrdHdl);
```

```
    } return rv;
```

```
}
```

```
////////////////////////////////////
```

```
// Function Name: sleep
```

```
// Description : Makes the thread to move to the Sleeping state for  
specific duration.
```

```
// Input Args : int - duration to sleep
```

```
// Output Args : None
```

```
// Return Value : None
```

```
////////////////////////////////////
```

```
void Thread::sleep(int m_nDuration)
```

```
{
```

```
    if (m_ThrdHdl != NULL)
```

```
    {
```

```
        Sleep(m_nDuration);
```

```
    }
```

```
}
```

```
////////////////////////////////////
```

```
// Function Name: wait
```

```
// Description : Makes the other threads to wait until this thread ends.
```

```
// Input Args : int - duration to wait
```

```
// Output Args : None
```

```
// Return Value : None
```

```
////////////////////////////////////
```

```
void Thread::wait(int m_nDuration)
```

```
{    if (m_ThrdHdl != NULL)
```

```
    {    WaitForSingleObject(m_ThrdHdl,m_nDuration);
```

```
    }
```

```
}
```

\*\*\*RUNNABLE.H\*\*\*

#ifndef Runnable\_h

#define Runnable\_h

/\*  
\*\*\*\*\*  
\*/

\* CLASS:

\* Runnable

\* DESCRIPTION:

\* This class is an interface which defines the signature for

\* run(). The run() forces to provide the task for the thread

\* to execute.

\* USAGE:

\* Create a class by extending this class Runnable. Provide an

\* implementation for run().

\* RELATED CLASSES:

\* Runnable

/\*  
\*\*\*\*\*  
\*/

```
class Runnable
{
public:
    ///
    // Provides Task for thread to execute.
    //
    virtual void run() = 0;
};

#endif Runnable_h
```

\*\*\*THREAD.H\*\*\*

```
#ifndef Thread_h
```

```
#define Thread_h
```

```
#include <Runnable.h>
```

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

```
/***/
```

```
* CLASS: Thread
```

```
DESCRIPTION:
```

- \* This class represents the thread. This provides a lot of members to
- \* manipulate with the threads in a more effective way.

```
* USAGE:
```

- \* Create a thread by passing Runnable object pointer to the
- \* Thread's constructor. After the thread object creation, start
- \* the thread by invoking start() method of the thread class.

```
* RELATED CLASSES:
```

```
* Runnable
```

```
/***/
```



```
class Thread : public Runnable
{
public:
    ///
    // Constuctor - Default.
    //
    Thread();
    ///
    // Constructor with initializer.
    //
    Thread(Runnable *a_Runnable);
    ///
    // Starts the thread.
    //
    void start();
    ///
    // Ends thread's life.
    //
```

```
void kill();

///

// Makes the thread to move to the Sleeping state for specific

// duration.

//

void sleep(int m_nDuration=INFINITE);

///

// Makes the other threads to wait until this thread ends.

//

void wait(int m_nDuration=INFINITE);

///

// Suspends the execution of this thread.

//

void suspend();

///

// Resumes the execution of this thread.

//

void resume();

///
```

```
// Sets the priority for this thread.  
  
//  
void setPriority(int a_priority);  
  
///  
// Returns the priority of this thread.  
  
//  
int getPriority();  
  
///  
// Null implementation for Runnable's run method.  
  
//  
void run()  
  
{  
  
}
```

private:

// Start up procedure for thread.

static DWORD WINAPI StartProc(LPVOID vpv);

// Runnable object pointer.

Runnable\* m\_Runnable;

// Handle to the Thread object.

HANDLE m\_ThrdHdl;

// Unique thread identifier.

unsigned long m\_ThreadId;

};

\*\*\*TEST.CPP\*\*\*

```
#include <Thread.h>

#include <Nos.h>

void main()

{

    Nos *no1 = new Nos("Even", 0, 2);

    Nos *no2 = new Nos("Odd", 1, 2);

    Nos *no3 = new Nos("* of 5", 0, 5);

    Nos *no4 = new Nos("* of 10", 0, 10);

    Thread *t1 = new Thread(no1);

    Thread *t2 = new Thread(no2);

    Thread *t3 = new Thread(no3);

    Thread *t4 = new Thread(no4);

    t1->start();

    t2->start();

    t3->start();

    t4->start();
```

```
t1->setPriority(THREAD_PRIORITY_NORMAL);
```

```
t2->setPriority(THREAD_PRIORITY_HIGHEST);
```

```
t3->setPriority(THREAD_PRIORITY_LOWEST);
```

```
t4->setPriority(THREAD_PRIORITY_IDLE);
```

```
t1->wait();
```

```
t2->wait();
```

```
t3->wait();
```

```
t4->wait();
```

```
}
```

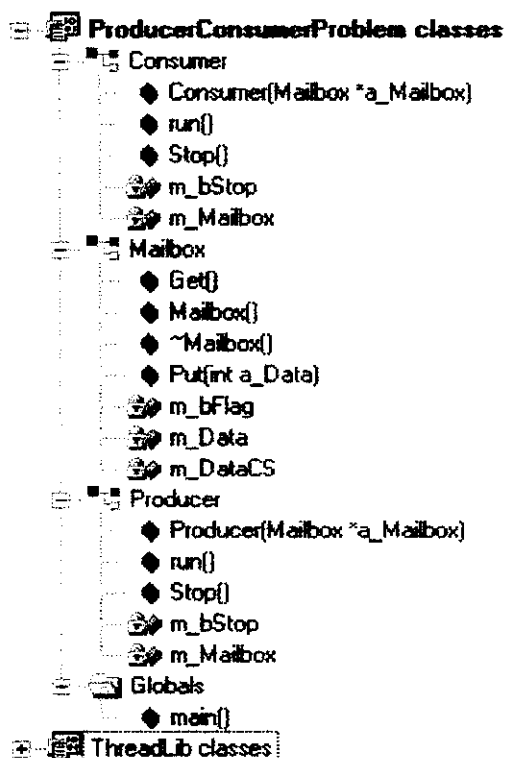
## APPENDIX – B

### Producer Consumer problem

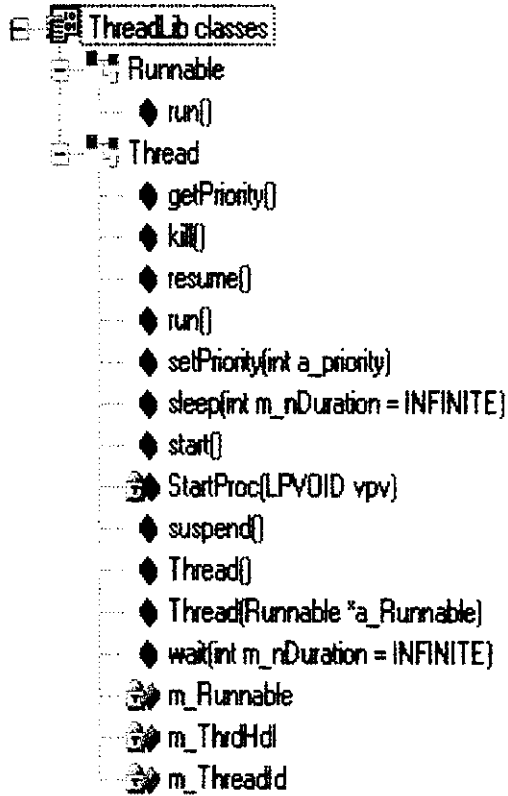
Producer produces a lot of data (here, it is natural numbers). The produced data is stored in mailbox. The mailbox can have only one data at any time. The consumer consumes the data from the mailbox. The consumer should consumes all the data produced by the producer. No data should be left unconsumed.

Thread synchronization win32 APIs are used for this.

### Class diagram for Producer Consumer problem



## Thread Library classes





/\*\* PRODUCE.H \*\*/

```
#ifndef _Producer_h
#define _Producer_h
#include <Runnable.h>
#include <Mailbox.h>
#include <string.h>
using namespace std;
///
// Producer class.
//
class Producer : public Runnable
{
public:
///
// Constructor with intializer.
//
Producer(Mailbox* a_Mailbox)
: m_Mailbox(a_Mailbox), m_bStop(false)
{
}

///
// Post new messages in to the mail box.
//
```

```
void Run()
{
    int count = 1;
    ///
    // Post new messages until the Stop flag is set. //
    while (!m_bStop)
    {
        m_Mailbox->Put(count++);
    }
}
///
// Stop the producer. //
```

```
void Stop()
{
    m_bStop = true;
}
```

```
private:
```

```
///
// Reference to the Mailbox object. //
Mailbox* m_Mailbox;
///
// Thread exit flag. //
bool m_bStop;
};
#endif _Producer_h
```

/\*\*CONSUMER.H\*\*/

```
#ifndef _Consumer_h
#define _Consumer_h
#include <Runnable.h>
#include <Mailbox.h>
#include <iostream>
using namespace std;
///
// Consumer class.//
class Consumer : public Runnable
{
public:
// Constructor with intializer.
Consumer(Mailbox* a_Mailbox): m_Mailbox(a_Mailbox),
m_bStop(false)
{ }
///
// Consumes the messages in the mail box. //
void Run()
{
int count = 0;
while (!m_bStop)
{
cout << "Data - " << m_Mailbox->Get() << endl;
}
}
}
```



```
// Stops the consumer. //  
void Stop()  
{  
    m_bStop = true;  
}  
private:  
///  
// Pointer to the Mailbox object. //  
Mailbox* m_Mailbox;  
///  
// Thread exit flag. //  
bool m_bStop;  
};  
#endif _Consumer_h
```

```
// Posts new message. //
void Put(int a_Data)
{
    // Sleep until the consumer consumes the existing message //
    while (m_bFlag)
    {
        Sleep(200);
    }

    // Lock the Data. //
    EnterCriticalSection(&m_DataCS);
    ///

    // Assign new Data. //
    m_Data = a_Data;
    ///

    // Enable New Message flag. //
    m_bFlag = true;
    ///

    // UnLock the Data. //
    LeaveCriticalSection(&m_DataCS);
}
///

// Consumes message. //
int Get()
{
    int data;
    ///
```

```
// Wait until new message is posted. //
while (!m_bFlag)
{
    Sleep(200); }
///
// Lock the Data. //
EnterCriticalSection(&m_DataCS);
// Take local copy.//
data = m_Data;
// Disable new message flag.//
m_bFlag = false;
// UnLock the Data.//
LeaveCriticalSection(&m_DataCS);
// Return the data to the consumer.//
return data;
}

private:
// Data produced by the Producer.//
int m_Data;
// Message Flag //
bool m_bFlag;
// Critical Section structure for Data.//
CRITICAL_SECTION m_DataCS;
};

#endif _Mailbox_h
```

/\*\*MAIN.CPP\*\*/

```
#include <Producer.h>
#include <Consumer.h>
#include <Thread.h>
#include <conio.h>
void main() {
    ///
    // Create mailbox object. //
    Mailbox* m = new Mailbox();
    ///
    // Create Producer object. //
    Producer* p = new Producer(m);
    ///
    // Create Consumer object. //
    Consumer* c = new Consumer(m);
    ///
    // Create thread object. //
    Thread* t1 = new Thread();
    Thread* t2 = new Thread();
    ///
    // Start the thread object. //
    t1->Start(p);
    t2->Start(c);
    cout << "Press any key to stop this Program" << endl;
    getch();
}
```

