Multithreaded Programming in a Microsoft Win32* Environment

By Soumya Guptha

Introduction

Through several generations of microprocessors, Intel has extended and enhanced the IA-32 architecture to improve performance. But applications typically only make use of about one-third of processors execution resources at any one time. To improve the utilization of execution resources, Intel has introduced Hyper-Threading Technology. The goal of Hyper-Threading Technology is to enable better processor utilization and try to achieve about 50% utilization of resources.

In order to take advantage of this innovative technology, we first need to understand the fundamentals of multithreading and see how multithreaded applications behave in order to reap the benefits of Hyper-Threading Technology.

So let's dive into understanding what threads are, when to use threads and how to synchronize them to prevent them from interfering with each other.

Overview of Multithreaded Programming

Multithreaded programming involves the implementation of software to perform two or more activities in parallel within the same application. This can be accomplished by creating threads to perform each activity. Threads are tasks that run independently of one another within the encompassing process. A thread is a path of execution through the software that has its own call stack and CPU state. Threads run within the context of a process, which has an address space consisting of code and data.

Why use Threads?

It is true; two people can mow a lawn faster than one IF each person has their own mower, IF the work is divided evenly and, IF the resources are shared efficiently between the two. IF the mowing pattern overlaps, there could be some slow down - catastrophe could occur IF the mowers run into each other. IF both share a gasoline can, they both could contend for it at the same time, or have to wait while the other fills up. However, IF the mowers have two mowers, both mowers communicate so that they don't overlap, share the resources efficiently, then the two can mow the lawn twice as fast as one.

Most of the time, programs need to accomplish more than one task. Using multiple threads *increases throughput*, which is measured by the number of computations a program can perform at a given time. Some events, like a user pressing a button or constantly interacting with the program, are independent activities. The performance of an application can be improved by creating a separate thread for performing each of these activities rather than using a single thread to perform all these activities in a serial manner.

Programs that are I/O intensive often benefit and better use the CPU by using multiple threads to handle individual tasks. For instance, if a thread performing activity '***A***' spends a significant amount of time waiting for an I/O operation to complete, another thread can be created to perform activity '***B***' that can accomplish some work while thread '***A***' is blocked. A lot of work can be done in short bursts in between long waits. Waiting for a block of data to read or write to, or from, a device can take a lot of time. By creating multiple threads to perform these activities, the operating system can do a better job of keeping the CPU busy doing useful work while I/O bound threads are waiting for these tasks to complete.

Using multiple threads to separate the user interface sections of the program from the rest of the program *increases responsiveness* to the user. If the main program is busy doing something, the other threads can handle the user inputs and perform the tasks. For example, if a user wants to cancel bringing in a large amount of data from a web page, a single threaded Internet browser application needs to have some process in place to periodically check for cancellation and interrupt the data transfer. By creating multiple threads, the user interface thread running at a higher priority can immediately react and cancel the operation.

When Not to Use Threads…

Using multiple threads in an application does not guarantee any kind of a performance gain. Just because an operating system supports the use of multiple threads, it does not mean that we should always create threads since at times there are certain disadvantages in using multiple threads to accomplish a task in a program.

The overhead of adding threads, scheduling them to run, communicating between each thread and context switching between threads may sometimes outweigh the actual work performed when threads are used in a serial manner. For example, a single thread trying to compute a square root of a huge number would probably run faster than two threads trying to perform the same operation. This is because it takes a finite amount of time for the processor to switch from one thread to another thread. To switch to a different thread, the operating system points the processor at the memory of the thread's process. Then the operating system restores the registers that were saved in the context structure of a new thread. This process is known as context switch.

It is important to make sure to use threads where they can have the most impact! It is hard to determine when threading helps and when threading does not help for better performance. Sometimes we may have to experiment via trial and error methods.

Benefits of using Multiple Threads over Multiple Processes

Used intelligently, threads are cheap, fast to start up, fast to shut down and have a minimal impact on system resources. Threads share ownership of most kernel objects such as file handles. In contrast, it is difficult to pass window handles using multiple processes because the operating system prohibits this to prevent one process from damaging the resources in another process. In a multithreaded program, threads can share window handles because both the threads and handles live in the same process.

Context switching in a multithreaded application is cheaper than context switching of multiple processes because switching processes carries a lot more overhead than switching threads.

Consider a web server that needs to service hundreds of requests at a time and a few million requests per day. Users of the web server typically make requests for a small amount of data. It would be easy, but impractical, to start a new process to service each request, the overhead would be tremendous. Each new process would require a complete copy of the server software, this would require huge amounts of memory to be allocated and would need to be initialized to the state of the first copy. This could result in each request taking several seconds. This is obviously a lot of extra work to just move small amounts of data to the user. Using a process per request, in this case, results in a bloated and inefficient web server. Similarly, using a single thread to service every single request results in the serialization of requests and ultimately poor performance. Creating multiple threads will result in better performance since there are threads that are always waiting for network I/O to complete.

Win32 Thread Handling Functions

Let's take a look at the various procedures provided by the Microsoft Win32 API for working with threads. Every process has one thread created when a process begins. To create additional threads, use the CreateThread( ) function documented below.

HANDLE CreateThread (
LPSECURITY_ATTRIBUTES lp Thread Attributes,
DWORD dwStackSize,
LPTHREAD_START_ROUTINE lpStartAddress,
LPVOID lpParameter,
DWORD dwCreationFlags,
LPDWORD lpThreadId **)**;

lp Thread Attributes	security attributes that should be applied to the new thread, this is for NT. Use NULL to get the default security attributes. Use NULL for win95.
dwStackSize	default size of 1MB can be passed by passing zero.
lpStartAddress	address of the function where the new thread starts.
lpParameter	pointer to the 32 –bit parameter that will be passed to the thread.
dwCreationFlags	flags to control the creation of the thread. Passing zero starts the thread immediately. Passing CREATE_SUSPENDED suspends the thread until the ResumeThread( ) function is called.
lpThreadId	pointer to a 32-bit variable that receives the thread identifier.

CreateThread ( ) function call.[1]

CreateThread( ) returns a handle to the thread if it succeeds**.**

BOOL CloseHandle (Handle hObject);
Parameters: hOject - identifies the handle to an open object
Return Value: returns true if it succeeds.

CloseHandle ( ) function call.[1]

[1] Win32® thread handling function definitions taken from Microsoft help

It is important to use the CloseHandle( ) API shown above. You need to use this to release kernel objects when you are done using them. If a process exits without closing the thread handle, the operating system drops the reference counts for those objects. But if a process frequently creates threads without closing the handles, there could be hundreds of thread kernel objects lying around and these resource leaks can have a big hit on performance.

Void ExitThread (DWORD dwExitCode);
dwExitCode specifies the exit code for the calling thread.

ExitThread ( ) function call.

[1]

There are several ways to terminate threads. One of the ways is to call TerminateThread(). Calling this function will kill the thread but does not deallocate the thread stack and any resources that were held by the thread. The preferred way to exit threads is by calling the ExitThread ( ) function. If the primary thread calls this function, the application exits.

DWORD SuspendThread (HANDLE hThread);
hThread Handle to the thread.

DWORD ResumeThread (HANDLE hThread);
hThread specifies the handle to the thread to be restarted.

SuspendThread( ) and ResumeThread ( ) function call.[1]

When the primary thread calls the SuspendThread( ) function, the thread stops executing the user-mode code until the function ResumeThread( ) is called to wake up the thread which then starts executing again.

Multithreaded Programs are Unpredictable

The program PrintNumbers.c (this is one 'project') below in Example 1 shows a program that creates multiple threads and displays the thread Ids. The output obtained and shown below may be surprising.

Example 1

```
/*********************************************************
 * Program: PrintNumbers.c
 *********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
DWORD WINAPI PrintThreads (LPVOID);

int main ()
{
        HANDLE hThread;
        DWORD dwThreadID;
        int i;

        for (i=0; i<5; i++)
        {
                hThread=CreateThread(NULL,   //default security attributes
                                0,                  //default stack size
                                PrintThreads,  //function name
                                (LPVOID)i,    // parameter
                                0,          // start the thread immediately after creation
                                &dwThreadID);
                if (hThread)
                {
                        printf ("Thread launched successfully\n");
                        CloseHandle (hThread);
```

```
                }
        }
        Sleep (1000);
        return (0);
}
//function PrintThreads
DWORD WINAPI PrintThreads (LPVOID num)
{
        int i;
        for (i=0; i<3; i++)
        printf ("Thread Number is %d%d%d\n", num,num,num);
        return 0;
}
```

As you can see from the example Output for program PrintnNumbers.c, the results of the program running multiple threads are very unpredictable. Every time the program is run you get a different output. Even though thread 3 is created before thread 4, thread 4 finished executing before thread 3 started. Sometimes there are also cases where a context switch happens while the thread is in a process of displaying results and displays a row as 433 instead of 444. You also see from the output that the threads do not start immediately. Thread 0 started printing immediately when it was created, but both threads 2 and 3 were created after thread 0 executed completely and then started printing numbers.

```
/*****************************************
 * Output for program PrintNumbers.c
 *****************************************/
```

Run #1

```
Thread launched successfully
Thread Number is 000
Thread Number is 000
Thread Number is 000
Thread launched successfully
Thread launched successfully
Thread Number is 111
Thread Number is 111
Thread Number is 111
Thread Number is 222
Thread Number is 222
Thread Number is 222
Thread launched successfully
Thread launched successfully
Thread Number is 444
Thread Number is 444
```

Thread Number is 433

```
Thread Number is 333
Thread Number is 333
Thread Number is 333
```

As you can see from the output above, the execution order of threads can be very random with unpredictable results. Because of this, we need to examine ways to synchronize threads. In the next section, we'll look at how to provide thread synchronization to ensure that we get the results we expect from a program.

Thread Synchronization Tools

Synchronization is essential because concurrent access to shared data may result in data inconsistency since threads normally run asynchronously. Access to these data needs to be synchronized. Multithreaded programming involves writing software to synchronize resource access between threads by using multiple threads in a useful and efficient manner. In a preemptive multitasking system, the operating system ensures that every thread gets to run. But the order of execution of multiple threads is unpredictable. This is known as a *race condition.* It is the responsibility of the programmer to use synchronization objects to ensure correct order of execution of threads. There are various synchronization mechanisms in the Microsoft Win32 API that help in using threads in an efficient manner. These are Critical Sections, Mutexes, Semaphores and Events.

Consider an example of two threads trying to access the same data structure, say a stack. Example 2 demonstrates two threads trying to add a node at the same time.

Example 2

```
struct Node
{
        struct Node *next;
        int data;
};
struct Stack
{
        struct Node *head;
};
void Push (struct Stack *stk, struct Node * new node)
{
        node->next = stk->head;
        stk->head = new node;
}
Node* Pop (struct Stack*stk)
{
        Node *temp = stk->head;
        stk->head = stk->head->next;
        return temp;
}
```

Suppose we have a stack of one node as shown in Figure 1. Thread 1 calls the function Push() to add Node B, then a context switch happens and the control is passed to Thread 2 as you can see in Figure 2.

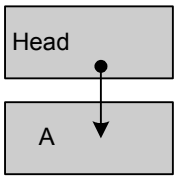

Fig 1. Stack with one Node

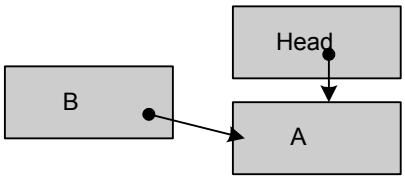

Fig 2. Stack after Context Switch

Thread 2 tries to add (push) Node C and it successfully completes adding Node C as you can see in Figure 3.

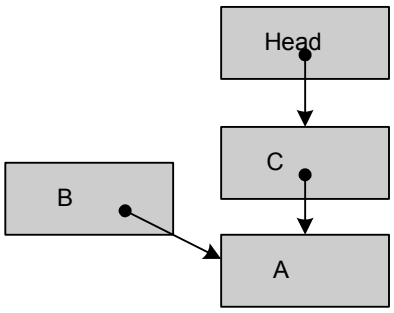

Fig 3. Stack after thread 2 completes

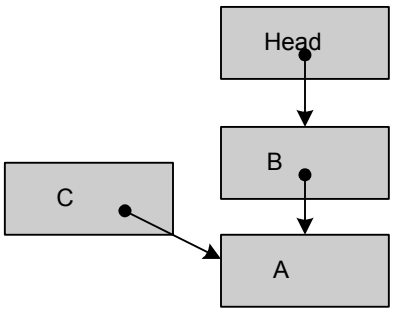

Fig 4 Stack after thread 1 completes

Thread 1 is allowed to finish as shown in Figure 4. Thread 2 sets its head to Node B and points to Node A. When a context switch happens, the current state of a thread is saved and resumed. As you can see from the output in Figure 4, the node C that thread1 tried to add has not been added. Node C is cut out of the stack. The problem like this may happen very rarely, but this could crash the program and more importantly, it produces incorrect results. As a result of this we need to examine the ways of synchronizing threads.

Critical Sections

A critical section is a portion of the code that can access a shared resource, which could be a memory location, file, data structure, or any resource where only one thread can access at a time. Only one thread can be inside the critical section at a time. Other threads are blocked from entering the critical section. They have to wait for the thread in the critical section to leave. Critical sections are used to synchronize threads within the same process and not different processes. Critical sections are used to protect areas of code or memory. Critical sections are not kernel objects which is why they are limited to synchronizing threads of a single process.

In Win32, critical sections are declared as a variable of type CRITICAL_SECTION for each resource that needs to be protected. We need to initialize by calling InitializeCriticalSection( ). After we are done with the critical section, use DeleteCriticalSection( ) to clean up. After initializing the critical section, a thread can enter a critical section by calling EnterCriticalSection ( ) and call LeaveCriticalSection ( ) to leave the critical section of the code.

Example 3 shows how to use critical sections. The problem of multiple threads adding nodes in a Stack that we saw earlier in Example 2 where Node C was cut off the list can be fixed by using critical sections. The problem with the code in Example 2 the function Push( ) was called to add a node by multiple threads at the same time, resulting in the corruption of the list. Thread2 was called before thread1 was completed. Example 3 shows how each access to the stack is surrounded by a request to enter and leave the critical section to overcome the problem we saw in example 2.

There are several problems encountered while using critical sections. One common problem is if a thread inside a critical section suddenly crashes or exits without calling the LeaveCriticalSection ( ), there is no way to say if the thread inside the critical section is alive. Since critical sections are not kernel objects, the kernel does not clean it up if a thread exits or crashes. We can overcome this problem by using a mutex.

Example 3

```
struct Node
{
        struct Node *next;
        int data;
};
struct Stack
{
        struct Node *head;
        CRITICAL_SECTION critical_sec;
};
void Push (struct Stack *stk, struct Node * new node)
{
        //enter critical section, add a new node and then
        //leave critical section
        EnterCriticalSection (&stk->critical_sec);
        node->next = stk->head;
        stk->head = new node;
        LeaveCriticalSection (&stk->critical_sec);
}
Node* Pop (struct Stack*stk)
{
        EnterCriticalSection (&stk->critical_sec);
        Node *temp = stk->head;
        stk->head = stk->head->next;
        LeaveCriticalSection (&stk->critical_sec);
        return temp;
}
```

Mutex

A mutex is a kernel object that allows any thread in the system to acquire mutually exclusive ownership of a resource. Only one thread at a time can own a mutex object. Unlike critical sections, mutexes can be used between processes, mutexes can be named and a timeout can be specified when waiting on a mutex. But the disadvantage is it takes about 100 times longer to lock an unowned mutex than it does to lock an unowned critical section.

In Win32, a mutex can be created by calling CreateMutex( ) or OpenMutex( ) if it already exists. After you are done with the mutex you need to close the handle by calling CloseHandle( ). Mutexes have a reference count that is decremented whenever a CloseHandle( ) is called or when the thread exits. When the reference count reaches zero, the mutex is deleted like all kernel objects where as this is not true in case of a critical section since critical sections are not kernel objects.

A mutex is signaled when no thread owns the mutex. A mutex can be owned in Win32 by calling one of the Wait…( ) functions such as WaitForSingleObject( ) or WaitForMultipleObjects( ). This call succeeds when no thread owns a mutex. Once a thread owns a mutex, the mutex goes into a nonsignaled state so that no other threads can have an ownership. After a thread is done with the mutex which is the same as saying a thread leaves the critical section, it can release the ownership by calling ReleaseMutex ( ). Only the thread that owns the mutex can release the mutex. When a mutex is in a nonsignaled state, a call to one of the Wait…( ) functions makes the thread *block* which means that the thread cannot run until the mutex is released and signaled. If a thread that owns a mutex exits or terminates without calling ReleaseMutex ( ), the mutex is not destroyed but it is marked as unowned and nonsignaled and the next thread waiting on it is notified by the flag WAIT_ABANDONED_0.

The program primes.c on the next page shows how to create multiple threads and use Mutexes. This program creates multiple threads to calculate the prime numbers for any given range. The purpose of this program is not to efficiently calculate primes but to show how to use mutexes, wait for a thread to complete, and how to release mutexes.

Example 4: Program Primes.c

```
/*************************************************************
 * Program: primes.c - The program creates 2 threads and calculates
 * the prime numbers. The main function takes the upper bound as an
 * argument to compute primes within the upper bound range and displays
 * all the prime numbers found.
 * ************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <windows.h>
#include <process.h>

HANDLE g_hMutex = NULL;
int *g_PrimeArr = NULL;
int g_primeMax = 0;
int g_primeIndex = 3;


DWORD WINAPI ComputePrimes (LPVOID);

int main (int argc, char **argv)
{
   int Max = 0;
          HANDLE hThread1 = NULL, hThread2 = NULL;
          int i, thd1=1,thd2=2;

   if(argc < 2)
   {
    printf("Usage error: The program needs the upper bound of search range.\n");
    return 0;
   }

   g_primeMax = Max = atoi(argv[1]);
   if(Max <= 2)
   {
     printf("Error: Upper Bound value has to be greater than 2\n");
     return 0;
   }
          //Allocate memory for the main thread's elements and initialize
           g_PrimeArr = (int *) malloc(sizeof (int) * (Max+1));
          for (i = 0; i<=Max; i++)
                    g_PrimeArr[i] = 0;
```

```
        //Set the first prime number
        g_PrimeArr[1] = 2;

        //create mutex
        g_hMutex = CreateMutex (NULL,           //default security attributes
                                TRUE,           //initial owner of the mutex
                                NULL);          //name of the mutex.
        if (g_hMutex == NULL)
        {
                printf("Error creating Mutex\n");
                return 0;
        }
        else
                ReleaseMutex(g_hMutex);

        //create 2 threads
        if((hThread1=CreateThread (NULL, 0, ComputePrimes,
                                        (LPVOID)thd1,0,NULL)) == NULL)
                printf("Error: Failed to create Thread1\n");

         if((hThread2=CreateThread(NULL, 0, ComputePrimes,
                                        (LPVOID)thd2,0,NULL)) == NULL)
                printf("Error: Failed to create Thread2\n");

        // wait for the thread to finish computing
        WaitForSingleObject (hThread1,    //handle for thread
                                INFINITE);      //time out interval
        WaitForSingleObject(hThread2, INFINITE);


        // Print the computed Prime Numbers
          printf ("Displaying primes\n");

    printf("Primes numbers in the range 1 to %d are: \n", Max);
             printf("---------------------------------------- \n");
             for(i = 0; i <= Max; i++)
    {
      if(g_PrimeArr[i] != 0)
         printf("%d ", g_PrimeArr[i]);
    }
             printf("\n");

   //close handles
  CloseHandle (g_hMutex);
  CloseHandle (hThread1);
  CloseHandle (hThread2);

        // free allocated memory
        if(g_PrimeArr)
        {
                free(g_PrimeArr);
                g_PrimeArr = NULL;
        }

return 0;
}

/*********************************************
 * ComputePrimes(..): Thread Function used to compute primes
 *********************************************/

DWORD WINAPI ComputePrimes(LPVOID idx)
{
        DWORD  dwresult;
```

```
        int currPrime = 0, i, sqroot;
        BOOL isPrime = TRUE;

        while (TRUE)
        {
                isPrime = TRUE;

                if (g_primeIndex <= g_primeMax)
                {
                        dwresult = WaitForSingleObject (g_hMutex, INFINITE);
                        if (dwresult == WAIT_OBJECT_0)
                        {
                                //if mutex acquired is successful
                                currPrime = g_primeIndex;
                                g_primeIndex++;

                                //release the mutex
                                ReleaseMutex (g_hMutex);
                        }
                        sqroot = (int) sqrt (currPrime)+1;
                        printf ("Thread %d checking if number %d"
                                " is prime\n",(int)idx,currPrime);
                        for (i = 2; i<= sqroot; i++)
                        {
                                if((currPrime%i) == 0)
                                {
                                        isPrime = FALSE;
                                        break;
                                }
                        }
                        dwresult = WaitForSingleObject (g_hMutex, INFINITE);
                        if (dwresult == WAIT_OBJECT_0)
                        {
                                if(isPrime)
                                        g_PrimeArr[currPrime] = currPrime;
                                ReleaseMutex (g_hMutex);
                        }
                }
                else
                  return 0;
        }
 return 0;
}
```

```
/*****************************************************
 * Sample Output for Progam Primes.c
*****************************************************/
C:\..\Release>primes 15
Thread 1 checking if 3 is prime
Thread 1 checking if 4 is prime
Thread 1 checking if 5 is prime
Thread 1 checking if 6 is prime
Thread 1 checking if 7 is prime
Thread 2 checking if 9 is prime
Thread 2 checking if 10 is prime
Thread 2 checking if 11 is prime
Thread 2 checking if 12 is prime
Thread 2 checking if 13 is prime
Thread 2 checking if 14 is prime
Thread 2 checking if 15 is prime
Thread 1 checking if 8 is prime
```

Displaying primes
Primes numbers in the range 1 to 15 are:
---
2 3 5 7 11 13

Semaphores

A semaphore is a kernel object that has a numerical count associated with it to manage a finite number of system resources. Consider a semaphore as an integer variable on which two indivisible operations can be performed. These operations are wait and signal. Semaphores are used to ensure that only one process at any time can be utilizing a resource. They can provide *mutual exclusion* between processes and synchronization. Semaphores are used mainly for solving producer/consumer problems where reads and writes are done at the same time. When the count is greater than zero the semaphore is in a signaled state, when the count is zero the semaphore is in a nonsignaled state.

In Win32, a semaphore can be created by calling CreateSemaphore( ) or OpenSemaphore( ) if it already exists. The current value of the semaphore represents the number of locks currently available. A lock on a semaphore can be obtained by calling any of the Wait…( ) functions such as WaitForSingleObject( ) or WaitForMultipleObjects( ). If nobody owns a semaphore Wait..( ) returns immediately. If a semaphore succeeds in acquiring a lock it does not mean that the semaphore has an exclusive lock since unlike mutex there is no concept of ownership. A thread that repeatedly calls Wait..( ) function will acquire a new lock for every call made. When the semaphore object is in a nonsignaled state, then all the available resources are currently used by other threads. If a thread calls Wait..( ) on a semaphore that is in a nonsignaled state, the thread is blocked until one of the locks are released. Unlike mutexes there is no state such as “wait abandoned” that can be detected by other threads. To release a lock ReleaseSemaphore( ) function is called which increments the count. Any thread can call this function to release a lock. To close a handle to a semaphore object, the function CloseHandle( ) is used. When the last thread closes the handle, the semaphore object is destroyed.

The code snippet in Example 5 shows how to create a semaphore, wait for multiple objects and release a semaphore.

Example 5

```
//start main function
HANDLE hSemaphore, hThread;

//create semaphore
hSemaphore = CreateSemaphore (NULL,  // default security attributes
                              2,              // initial count
                              2,     //maximum count
                              NULL);     // object name
if (hSemaphore == NULL)
{
        printf (“Error creating the semaphore\n”);
        exit (1);
}
//create child threads
for (i=0; i<3; i++)
{
        hThread=CreateThread(NULL,0,foo,(LPVOID)i,0,&dwThreadID);
        if (hThread)
                printf ("Thread launched successfully\n");
        else
                //print error message and exit (-1)
}

//wait until child threads have exited
DWORD dwCThd = WaitForMultipleObjects (3,              //count of objects
                                       hThread,        //thread handle
                                       TRUE,           //wait for all
                                       INFINITE);      //time out interval
If (dwCThd != WAIT_OBJECT_0)
        //Print error message and exit (-1)

//close handle for semaphore
CloseHandle (hSemaphore);
//close child thread handles
```

```
for (i=0; i<3; i++)
        CloseHandle (hThread[i]);

//end of main function

/************
 * function foo
 ************/
DWORD WINAPI foo (LPVOID num)
{
    for( j = 0; j<3; j++)
   {
         //wait for single object
         DWORD dwRes = WaitForSingleObject (hSemaphore, INFINITE);
        If (dwRes == WAIT_OBJECT_0)
            //acquired the semaphore
         else
           //print error

         // do something

       // release semaphore
       Long lpPreviousCount;
         ReleaseSemaphore (hSemaphore,
                                  1,          // count of the semaphore object is to be increased
                                  &lpPreviousCount);   //pointer to a 32-bit variable for the
                                                       // previous count of the semaphore
      Sleep (1000);
   }
 return 0;
}
```

Events

The event object is a kernel object that stays nonsignaled until a condition is met. The programmer has the control over setting the event object to a signaled or a nonsignaled state unlike a mutex or semaphore where the operating system governs the signaled and nonsignaled state of the object. The concept of an event object is very similar to a *condition variable* giving the programmer the maximum flexibility to define complex synchronization objects. There are two types of events, manual reset event and auto reset event. A manual reset event can be returned to a nonsignaled state only when reset by the programmer where as an auto reset event is set back to the nonsignaled state after the completion of a wait.

In Win32, an event can be created by calling CreateEvent( ) to create a new event or OpenEvent( ) if it already exists. To close a handle call CloseHandle( ). Win32 provides SetEvent( ), ResetEvent( ) and PulseEvent( ) functions.

The SetEvent( ) function is used to set an event to the signaled state. When the SetEvent( ) function or PulseEvent( ) function is called on an auto-reset event, only one waiting thread is released and the event is reset to the nonsignaled state. If no threads are waiting, then the event will stay signaled until one thread waits on it and then the event goes into the nonsignaled state.
When the SetEvent( ) function is called on a manual-reset event, all waiting threads are awakened and the event object is in the signaled state unless specifically reset by the ResetEvent( ) function.
When the PulseEvent( ) function is called on a manual-reset event, all waiting threads are awakened and the event object is reset to the nonsignaled state.

The code snippet in Example 6 shows how to use events.

Example 6

```
//global variables
HANDLE g_hEvent = NULL;

//local variables
HANDLE hThread;
DWORD dwThreadID;
BOOL seteventStatus;
```

```
//create auto-reset event
g_hEvent = CreateEvent (NULL, // no security attributes
                        FALSE,          //auto-reset event
                        FALSE,          //initial state non-signaled
                        NULL);          /name of the event object
If (g_hEvent)
        //print error message
        //exit (-1)

//create child threads
for (i=0; i<3; i++)
{
        hThread=CreateThread(NULL,0,foo,(LPVOID)i,0,&dwThreadID);
        if (hThread)
                printf ("Thread launched successfully\n");
        else
          //print error message and exit (-1)
}
Sleep (1000);

//signal for auto reset event
seteventStatus = SetEvent (g_hEvent);
if (!seteventStatus)
        //print error message for failing in calling set event and exit (-1)

//wait for child threads to exit
DWORD dwCThd = WaitForMultipleObjects (3,               //count of objects
                                       hThread,         //thread handle
                                       TRUE,            //wait for all
                                       INFINITE);       //time out interval
If (dwCThd != WAIT_OBJECT_0)
        //Print error message and exit (-1)

//close handles
CloseHandle (hEvent);
//close child thread handles
for (i=0; i<3; i++)
        CloseHandle (hThread[i]);

//end of main function

/************************
 * function foo
 ************************/

DWORD WINAPI foo (LPVOID num)
{
    for( j = 0; j<3; j++)
   {
        //wait for single object
        DWORD dwRes = WaitForSingleObject (hEvent, INFINITE);
       If (dwRes == WAIT_OBJECT_0)
            //print wait for set event succeeded
        else
          //print error
        // do something
   }
return 0;
}
```

Recent Threading Innovations

Now that we have a basic understanding of how to build threaded applications, let's examine Intel's Hyper-Threading Technology. The basic concept behind Hyper-Threading is that it enables a single physical processor to execute two threads concurrently. Hyper-Threading enables two logical processors sharing the same core processor resources to increase the utilization of the execution engine. Hyper-Threading Technology is designed to improve the utilization of IA-32 processors, thereby improving application performance. Software sees Hyper-Threading technology as 2 processors. To get the performance benefits of Hyper-Threading, the application has to be multithreaded. We will look into this more and discuss on some of the performance concerns involved in a future article.

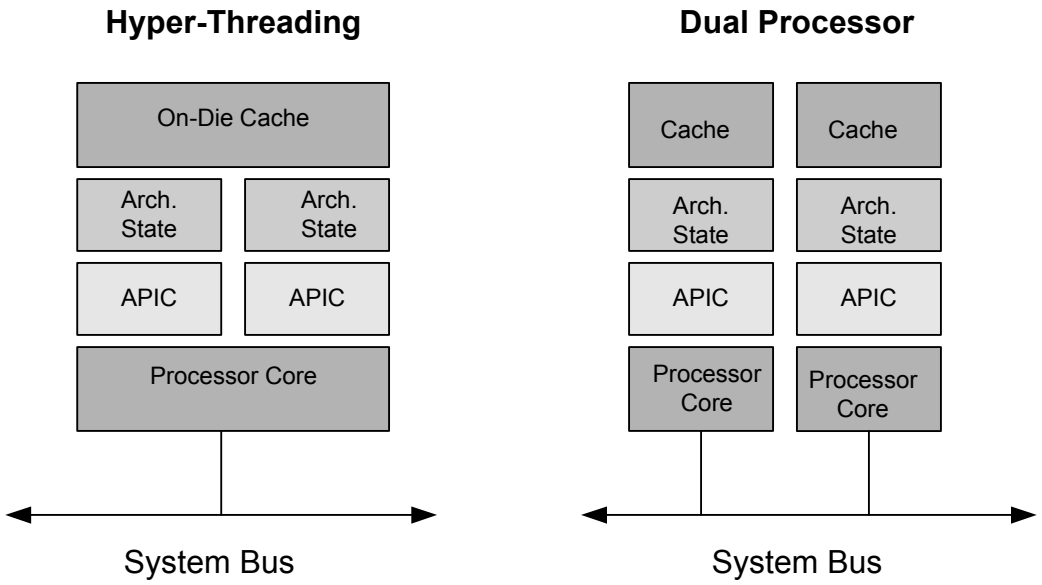

Fig 5. Representation of an IA-32 processor with Hyper-Threading technology and dual processor system.

Conclusion

We have covered the fundamentals of multithreading, the importance of using threads, the problems associated with it and the various synchronization mechanisms. You should have a pretty good idea by now how to write simple, multithreaded applications.

In a future article, we will look into writing applications using thread priorities and discuss process affinity and thread affinity for restricting processes and threads to a particular processor. We will also see some performance concerns and discuss ways to determine the ideal number of threads to be created for performance gains in multithreaded applications using Hyper-Threading technology.

References

Win32 Multithreaded Programming, Jan 1998. Aaron Cohen & Mike Woodring, O'Reilly.
Multithreaded Applications in Win32, dec 1998. Jim Beveridge & Robert Wiener, Addison Wesly.

Author Information:

Soumya Guptha

Soumya Gupta is a technical marketing engineer with Intel Corporation in the Software Solutions Group. If you have any questions, comments, suggestions or feedback, please email her at soumya.k.guptha@intel.com