Threads, Multithreading & Thread Synchronisation

Amitava Ghosh*

1. THREADS:

1.1 The philosophy underneath : Threads represent a compromise between 2 different Operating System philosophies :

i) Traditional heavy state-laden processes that offer protection but can impair real-time performance, such as in Unix.

ii) Lean and fast real-time OS that sacrifice protection for performance.

1.2 What is a thread? : A thread is a lightweight process with a reduced state. In thread based systems, threads take over the role of the processes as the smallest individual unit of scheduling. Each thread belongs to exactly one process and no thread can exist outside a process. However, one process may contain multiple threads in a multithreaded system.

In an operating system, implementing threads (e.g.. Windows NT) or process comprises of

— A private memory space in which the process code and data are stored.

— An access taken for security checks.

— System resources such as files and windows.

— At least one thread to execute the code and a thread comprises of—

— A processor state including the current instruction pointer.

— A stack for use when running in user mode.

— A stack for use when running in kernet mode.

Since processes (and not threads) own the access token, system resource handles, and address space, all of the threads in process share the same memory, access token and address space when a process is created in such an environment, an initial thread is also created in it. This thread can create new threads in the same process. These threads can exit at any time and the process will exit when the last thread of the process exits.

The threads can execute anywhere in the code space of the process. They might execute the exactly same code at the same time (or at different times), but even when they do execute the same code, it is not the same execution, just the same instructions—i.e. they are fetching and executing the same instructions, but in different contexts (with different registers and stack).

1.3 Why should we use threads? : Advantages of thread-based systems :

i) Threads are cheaper to create and destroy because they do not need allocation and deallocation of a new address space [or for that matter other process resources (such as duplicate open files)]. So using multiple threads or creating a thread to do a small task is more justified.

ii) Switching between threads (rather than processes) can be faster if a thread is dispatched to a running program. This is because memory mapping does not have to be set up and address translation cachis do not have to be invalidated.

iii) As threads share memory they do not need system calls (and consequently context switches), they communicate faster and through memory. This makes them especially suitable for parallel activities that are tightly coupled and use the same data structure.

* Final Year Student of Computer Science & Technology

When an O.S. implements both processes and threads, the programmer can choose to have parallelism at 2 levels :

i) the process level

ii) the thread level.

Advantages of using threads to partition process activities—

i) The program is more responsive to the user, as one of the threads will always accept input from the user.

ii) Programming the system is easier.

iii) It is efficient to wait for things with threads. A thread is but a part of a program, and if a thread is blocked, the rest of the threads are still active.

2. IMPLEMENTATIONS :

Threads have been successfully used in implementing network servers. They also provide a suitable foundation for parallel execution of applications or shared-memory multiprocessors. Inter-thread synchronisation increases program portability and significant execution speedups may be obtained on a multiprocessor by allocating separate processor to individual threads.

They may be found in Mach and OS/2. Besides, we present the following case studies where threads have been implemented.

2.1 Visual C++ 5.0

One of the best aspects of 32-bit programming is true multitasking. Each of the programs running at once has its own thread or execution stream. Windows multitasks between threads, sharing access to the CPU between them.

There are 2 types of threads in Visual C++ —user I/O threads and Worker threads.

A user I/O thread supports the creation of Windows and can seem to the user much as he has launched a new Windows program. The more common worker threads stay invisible but allow a program to devote available time to a specific problem. With the possibility of a no. of execution streams, and since in Win 32 programs are usually multithreaded, when the OS can switch control away from threads without asking them first (unlike in Win 16), where a program decides when to relinquish the processor, thread synchronisation becomes an important issue.

There are primarily 4 methods available in Visual C++ for thread and synchronisation.

2.1.1 Critical Section Synchronisation : We define Critical Section of a program as that part where some shared variable or any other type of shared resources is accessed.

This kind of synchronisation deals only with threads in the same process and is used when a program has a particular section of code that not more than one thread should execute simultaneously.

To use this technique, an object of the MFC class "CCritical Section" is created. The critical section can be entered when the constructor returns. For repeated access, an object of class "CSingleLock" (in case of one critical section) or "CMultiLock" (in case of multiple critical sections) is created and the object's Lock ( ) and Unlock ( ) functions are used to gain access of the critical section. Besides, a thread can use the Non-MFC functions, Initialize Critical Section ( ) and Enter Critical Section ( ) to enter the critical section of the code, ensuring no other thread from the same process is allowed to enter the Critical Section, while the first thread is still in it. When the thread leaves the cirtical section, it calls Leave Critical Section ( ) which opens the way for other threads waiting to get in.

2.1.2 Event Synchronisation : This deals with Windows events only. This kind of synchronisation averts the busy waiting at the thread execution level. [For the uninitiated, busy waiting refers to the continuous testing of a variable, waiting for some particular value to

appear. It wastes CPU time and should be avoided, at least until the atomic level.] An Windows event is created with Create Event ( ). Then thread B, which awaits some result from thread A, calls the function Wait For Single Object ( ) and Windows will simply suspend thread B until the function Set Event ( ) is called in thread A, which sets the event and execution of thread B is resumed. Alternately, the function Wait For Multiple Objects ( ) can be used for a number of events.

2.1.3 Mutex Synchronisation : In many ways, it works as critical section synchronisation does, except that mutex synchronisation can work with threads from different processes. Using the MFC class library, an object of class CMutex, as well as an object of either class CSingleLock or CMultiLock is created, followed by the use of CSingleLock or CMultiLock Lock ( ) and Unlock ( ) functions. Alternately, the non-MFC functions—Create Mutex ( ), Open Mutex ( ) and Release Mutex ( ) can be used.

2.1.4 Semaphore Synchronisation : A semaphore works much like a mutex, except that it can allow a no. of threads—upto a maximum number that a user specifies—access to the restricted code or resources.

When a program wants to declare a resource as restricted, it declares an object of class CSemaphore as well as an object of either class, CSingleLock or CMultiLock and uses the Lock ( ) and UnLock ( ) member functions. The non-MFC way to do this is to call Create Semaphore ( ) to get a handle to the semaphore and to specify how many threads are to have access to that resource concurrently. Various threads can then use Open Semaphore ( ) and Release Semaphore ( ) to get access to the restricted resource. To begin waiting for a restricted resource, they call Wait For Single Object ( ). In this way access can be restricted to a speficied no. of threads.

2.2 Windows NT : Windows NT is the first operating system to provide a consistent multithreading API across multiple platforms. Whenever a process is created in Windows NT, memory is allocated for it, a state is set up in the system and a thread object is created. To start a thread in a currently executing process, the Create Thread ( ) call is used as a function passed through 'lp Start Addr', this address may be any valid procedure address in an application.

2.2.1 A thread is born when a process is created : When a Windows NT process is created, its first thread, called the primary thread, is automatically created by the system. Every time a process is initialized, Windows NT creates a primary thread. This thread starts at the Win Main function and continues executing until the Win Main function returns. For many applications, this may be the only thread that the application requires, though this primary threads as well as other threads are also capable of creating new threads.

2.2.2 Priority and Scheduling : The Windows NT kernel's dispatcher performs thread scheduling and context switching. The kernel's dispatcher schedules threads to run based on a 32-level priority scheme. Windows NT ensures that the threads that are ready and have the highest priority will be running at any given time. The range of priorities (0-31) is divided in half with the upper 16 reserved for real-time threads and the lower 16 for variable priority threads.

Real-time threads run at the same priority for their entire lifetime. They are commonly used to monitor or control systems that require action to be taken at precise intervals and should be used sparingly as they run at higher priority levels than all variable priority threads.

Variable priority threads are assigned a base priority when they are created (which is determined by the process parent to the thread). The priority of such threads can be adjusted dynamically by the kernel's dispatcher and can vary upto two priority levels above or below its base priority.

The dispatcher maintains a priority queue of ready tasks. The scheduling is done in a pre-emptive, round-robin manner.The dynamic adjustment of thread priorities helps in performance turning.

2.3 Java : In Java, the multithreading concept is supported by the classes of java. lang. The thread class is used to construct and access individual threads of execution, that are executed as part of a multithreaded program. In java. (as well as in most of the other softwares having a good GUI), one of the most immediately competing reasons for multithreading is to produce a responsive user interface. The most important method for the thread class is run ( ) which is executed simultaneously with the other threads in the program, and must be overridden to make thread do your bidding. A run ( ) method virtually always has some kind of loop that continues until the thread is no longer necessary. So it is the program's responsibility to establish the breakout condition. Very often, run ( ) is cast in the form of an infinite loop and runs, barring some external call to stop ( ) or destroy ( ) until the program completes.

2.3.1 Problems with garbage collection : When main ( ) creates the thread objects it is not capturing the handles of any of them. As each thread "registers" itself there is actually a reference to it some place and the garbage collection cannot clear it up.

It is also to be noted that the order in which an existing set of threads execute is indeterminate, unless they are prioritized in some predefined order using the set Priority ( ) method of the thread class.

2.3.2 Daemon Threads : Java borrows the concept of daemons from Unix. A "Daemon" thread is supposed to provide a general service in the background as long as the program is running, but is not part of the essence of the program. Thus a program terminates off all the daemon threads complete execution.

Whither a particular thread is a daemon can be determind by calling is Daemon ( ) and daemonhood of a thread can be set on and off with set Daemon ( ). If a thread is a daemon, then any thread it creates will automatically be daemons.

2.3.3 Thread synchronisation in Java : Java has built-in support to prevent "collision"s over one kind of resource—the memory in an object. As we typically make the data elements private and access that method through methods only we can secure a method from collision by making that method synchronised. Only one thread at a time may call a synchronised method for a particular object, although that thread may call more than one of the object's synchronised method.

Each object contains a single lock (also called a monitor) that is automatically part of the object. When any synchronised method is called, that object is locked and no other synchronised method can be called, until the first one finishes and thus releases the lock.

There is also a single lock per class, so that synchronised static methods can lock each other out from a static data on a class-wide basis.

2.3.3.1 Why all methods are not synchronised by default? : As acquiring a lock is pretty costly in terms of the efficiency of the execution (it multiplies the cost of a method call by at least 4 times) it is expedient to leave off the synchronised key word, whenever it is known that a particular method will not cause contention problems.

2.3.4 Thread States in Java : A Java thread can be in any of four states :

(1) New : The thread object has been created but it has not been started yet, so it cannot run.

(2) Runnable : This thread can run when CPU cycles are available to it.

(3) Dead : This thread has terminated on completion of its execution. [Some authors find the term 'dead' used here very morose.] The normal way for a thread to die is by return from its run ( ) method. Alternately, the stop ( ) method can be called [But stop ( ) throws an exception, which is a subclass of Error, drastic in nature and which does not release object lock.

(4) Blocked : The thread could be run but something is preventing it from running. Until the thread re-enters the runnable state, the scheduler does not allocate it any CPU cycle.

2.3.4.1 Causes of thread block : There are five reasons for a thread to become blocked :

(1) The thread has been put to sleep by calling steep (milliseconds).

(2) The execution of the thread has been suspended by calling suspend ( ) and will not become runnable until the resume ( ) message is sent.

(3) The execution of the thread has been suspended by calling wait ( ) and it will not become runnable until the notify ( ) or notify All ( ) message is sent.

(4) The thread is waiting for some IO to complete.

(5) The thread is trying to call a synchronised method on another object and that object's lock is not available.

2.3.4.2 Deadlock : Because threads can block and objects can have synchronised method that prevent threads from accessing that object until the synchronisation lock is released, deadlocks may occur in a multithreaded environment. There is no language support to help prevent deadlock, but Java 1.2 deprecates the stop ( ), suspend ( ), resume ( ), and destroy ( ) methods to reduce the possibility of a deadlock.

2.3.5 Priorities : In Java, the priority of a thread can be read with get Priority ( ) and altered by using set Priority ( ).

2.3.6 Thread groups : In Java, all threads belong to a thread group. This may be either the default thread group or a group explicitly specified by the programmer. At creation, the thread is bound to a group and cannot change group subsequently. Each application possesses at least one thread that belongs to the thread group.

Thread groups must also belong to other thread groups. The thread group that a new one belongs to must be specified in the constructor. If a thread group is created without specifying its parent, it is placed under the system thread group. Thus all thread groups in any application ultimately have the system thread group as the parent.

2.3.6.1 Advantages of thread groups : Besides "security reasons", thread groups provide more control to the programmer. Certain operations on a group of threads can be performed using a single command.

3. PARTING WORDS :

It seems the future development of cyberworld hangs on threads. With renewed thrust in areas like thread security and synchronisation in multithreading environments, threads are going to stay for quite some time not merely as an important OS concept, but also as a powerful conceptual tool to the present generation system designers.

Reference & Bibliography :

1) Advanced Visual C++ — Steven Holtzner

2) Operating System : A design oriented approach—Charles Crowley

3) Operating System : Concept and Design—Milan Milewkovic

4) Thinking in Java—Bruce Eckel (Edition—www.eckelobjects.com)

5) Windows NT architecture and programming—Synergetics

6) Java 1.2 unleashed—Jamie Jaworski