Concurrent Programming in Windows Vista

Hadeel Tariq Al-Rayes
M.Sc. Computer Science
Basic Education College-Computer Science Department
Diyala University
hadeelalrayes@yahoo.com

***Abstract*— In general, writing concurrent programs is extremely difficult because the multiplicity of possible interleaving of operations among threads means that program execution is non-deterministic. For this reason, program bugs may be difficult to reproduce. Furthermore, the complexity introduced by multiple threads and their potential interactions makes programs much more difficult to analyze and reason about. Fortunately, many concurrent programs including most GUI applications follow stylized design patterns that control the underlying complexity.**

There are many features related to concurrent programming used in Windows operating system. This paper will illustrate the main features that used in Window Vista specifically. Concurrent programming concepts in Windows Vista differs from that used in the earliest releases of Windows operating system, this paper will explain these differences.

***Index Term*— Concurrent programming, Operating Systems, Threads, Windows Vista, Windows 7.**

I. Introduction

In software implementations, concurrent execution is simulated by running multiple threads of execution within an operating system (OS) process. The rules that govern time multiplexing between threads of execution are called threading semantics. e mandates that threading is non-preemptive: the running thread can yield control, but control cannot be taken away. Threads in e yield control when they become blocked.
Threads can block when they wait on, or synchronize to, a temporal expression (TE); attempt to perform a port operation that blocks; or attempt to access a shared resource that is occupied. The programmer can assume atomic execution for sequences of actions not containing these operations.
Once a thread blocks, the runtime engine selects the next thread to run. This is called scheduling. Threads shall be scheduled to run as long as they are not blocked. When all threads are blocked, simulated time is advanced according to the mode of execution in effect.

II. Concurrent Programming & Threads Definitions

- Concurrent Programming: the act of running several programs apparently simultaneously, achieved by executing small sections from each program in turn.[1]
- Concurrent computing is the concurrent (simultaneous) execution of multiple interacting computational tasks. These tasks may be implemented as separate programs, or as a set of processes or threads created by a single program. The tasks may also be executing on a single processor, several processors in close proximity, or distributed across a network. Concurrent computing is related to parallel computing, but focuses more on the interactions between tasks. Correct sequencing of the interactions or communications between different tasks, and the coordination of access to resources that are shared between tasks, are key concerns during the design of concurrent computing systems.[2]
- A Thread is runnable unless it executes a special operation requiring synchronization that waits until a particular condition occurs. If more than one thread is runnable, all but one thread may starve (make no progress because none of its operations are being executed) unless the language makes a fairness guarantee. A fairness guarantee states that the next operation in a runnable thread eventually will execute. The Java language specification currently makes no fairness guarantees but most Java Virtual Machines guarantee fairness.[3]

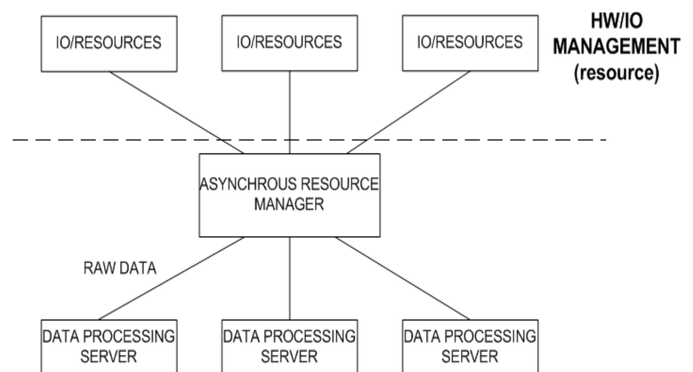


Fig. 1. Asynchronous resource manager.[2]

IJENS

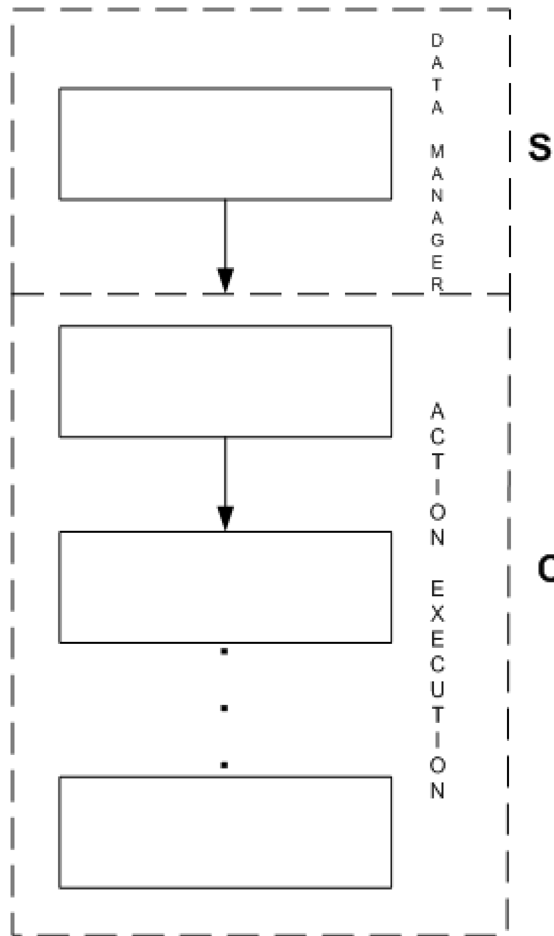


Fig. 2. Execution of Thread[2]

III. CONCURRENT ASPECTS OF PROGRAMMING

A concurrent program is a collection of processes that communicate by reading and writing from a shared memory. Processes in a such program often need to synchronize their actions. Synchronization between processes is classified as either cooperation or contention. A typical example for cooperation is the case in which there are two sets of processes, called the producers and the consumers, where the producers produce data items and then the consumers consume. Contention arises when several processes compete for exclusive use of shared resources. For example, the integrity of the data may be destroyed if two processes update a common file at the same time, and as a result, deposits and withdrawals could be lost, confirmed reservations might have disappeared, etc. In such cases it is sometimes essential to allow at most one process to use a given resource at any given time. The problem of concurrent program in practice is, how to resolve conflicts resulting when several processes are trying to use shared resources. Let's take an example of a concurrent program, this program includes two threads, thread T1 sets up the balance for two accounts A and B, thread T2 transfers 50$ from A to B (Figure 3). Depending on the order of the commands in these two threads, the result of the module may be different. This is an data race error example appeared in concurrent programs.[3]

T1	T2
$A := 50$ $B := 100$	A := A - 50 B := B + 50

Fig. 3. A Concurrent Programs

IV. WHY CONCURRENCY?

There are many reasons why concurrency may be interesting to you:

1- You are programming in an environment where concurrency is already pervasive. This is common in real-time systems, OS programming, and server-side programming. It is the reason, for example, that most database programmers must become deeply familiar with the notion of a transaction before they can truly be effective at their jobs.

2- You need to maintain a responsive user interface (UI) while performing some compute- or I/O-intensive activity in response to some user input. In such cases, running this work on the UI thread will lead to poor responsiveness and frustrated end users. Instead, concurrency can be used to move work elsewhere, dramatically improving the responsiveness and user experience.

3- You'd like to exploit the asynchrony that already exists in the relationship between the CPU running your program and other hardware devices. (They are, after all, separately operating and independent pieces of hardware.) Windows and many device drivers cooperate to ensure that large I/O latencies do not severely impact program performance. Using these capabilities requires that you rewrite code to deal with concurrent orchestration of events.

4- Some problems are more naturally modeled using concurrency. Games, AI, and scientific simulations often need to model interactions among many agents that operate mostly independently of one another, much like objects in the real world. These interactions are inherently concurrent. Stream processing of real-time data feeds, where the data is being generated in the physical world, typically requires the use of concurrency. Telephony switches are inherently massively concurrent, leading to special purpose languages, such as Erlang, that deal specifically with concurrency as a first class concept.

5- You'd like to utilize the processing power made available by multiprocessor architectures, such as multicore, which requires a form of concurrency called parallelism to be used. This requires individual operations to be decomposed into independent parts that can run on separate processors.[4]

V. NEW CONCURRENT PROGRAMMING'S FEATURES IN WINDOWS VISTA

1- On Windows Vista and Server 2008, a new feature called I/O Prioritization has been added. This regulates the scheduling of I/Os because contention

 IJENS

for the disk can artificially boost the priority of lower priority processes and threads by allowing them to interfere with higher priority ones. Five priorities are used: Critical, High, Medium, Low, and Very Low.[5]

2- Assignment of priority to an I/O request is handled primarily by the OS and drivers, although you have some control over it by assigning thread priorities. By default, all 1/ 0 under a priority of Medium, but you may pass the value PROCESS_MODE_BACKGROUND_BEGIN to SetPriorityClass to lower the I/O Priority to Very Low, and PROCESS_MODE_BACKGROUND_END to revert it. Similarly, you can pass THREAD_MODE_BACKGROUND_BEGIN to the SetThreadPriority function to lower I/O Priority for that particular thread, and THREAD_MODE_BACKGROUND_END to revert this change. This is used by programs such as the Windows Search Indexer to prevent it from interfering with other interactive applications.
3- Modifications to the thread scheduler's quantum accounting algorithm were made in Windows Vista and Server 2008. Two problems existed on previous versions of Windows that could lead to unfairness and unpredictability in the way that thread execution times were measured. The first is that interrupts that executed in the context of a thread would count towards that thread's quantum. Say that a thread's quantum was 15 milliseconds and 5 milliseconds of that time were spent executing interrupts; in this case, the thread would only be running its code for 10 milliseconds.
4- Vista no longer accounts for interrupt time when deciding whether to switch out a thread. The second problem was that the scheduler didn't account for threads being scheduled in the middle of a quantum interval.
5- The OS uses a timer interrupt routine to account for execution time. If this timer was set to execute every 15 milliseconds and some thread was scheduled in the middle of such an interval, say after 5 milliseconds, then when the timer fired next the OS would charge the thread for the full 15 milliseconds, when in fact it only ran for 10 milliseconds. Vista prefers to undercharge threads instead. This same thread would run for nearly a full timer interval longer than it should-since the granularity of the timer routine remains the same-but ensures threads are not unfairly starved.
6- As of Windows Vista, a new multimedia thread scheduler has been added to the system, called the multimedia class scheduler service (MMCSS). This is not really a thread scheduler per se, it's simply a service running in svchost.exe at a very high priority that monitors the activity of multimedia programs that have been registered with the system. It cooperates with them to boost priorities to ensure smoother multimedia playback. The service boosts threads inside of a multimedia program into the real-time range while it is actively playing media, but throttles this boosting periodically to avoid starving other processes on the system.
7- Windows Media Player 11 automatically registers itself, but any third party programs can also register programs with MMCSS. Programs do so by adding an entry to the KEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Multimedia\SystemProfile\Tasks registry key.[6]

8- Windows Vista has a new dynamic spin count adjustment feature. While this is used inside the as, it is an undocumented feature. It's possible that this feature will be officially documented and supported in an upcoming Windows SDK, but that may not happen, so we wouldn't recommend taking a dependency on it. If the InitializeCriticalSectionEx API is used, passing a Flags value containing the RTLJRITICAL_SECTION_DYNAMIC_SPIN value, the resulting critical section will use a dynamic spinning algorithm.
9- Keyed Events to the Rescue. As of Windows XP, this is no longer an issue.Windows contains a new kernel object type, called a keyed event, to handle low-resource conditions. Keyed events are hidden inside the kernel and are not exposed directly, though we'll see that they are used heavily in the new Windows Vista synchronization primitives (as with condition variables and slim reader /writer locks). And they used by EnterCriticalSection when memory is not available to allocate a true event. Keyed events have improved quite a bit in Windows Vista. Instead of storing waiters in a linked list, they now use a hash table keyed by the key K, trading the possibility of hash collisions (and hence, some amount of contention unpredictability) in favor of improved lookup performance. This improvement led to performance good enough that it allows them to be used as the sole event mechanism for the new Vista slim reader /writer lock, condition variable, and one-time initialization APIs. None of these new features use traditional events-they use keyed events exclusively, which is why the new primitives are so lightweight, often taking up only a pointer-sized bit of data and not requiring any dedicated kernel objects whatsoever. The improvement that keyed events offer to reliability and the alleviation of HANDLE and non pageable pressure is overall very welcome and will pave the way for new synchronization OS features in the future. They are accessible most directly with the condition variable APIs because they internally wrap access to the keyed event object. We'll get to those in a few more sections.
10- Windows Vista now offers a "slim" RWL with these precise characteristics. The .NET Framework offers two, one of which has been available since the .NET Framework 1.1, while the other is new with 3.5.

That's where the new Windows Vista condition variable feature comes in handy. It integrates with both critical sections and SRWLs to enable waiting and signaling on a logical condition variable related to a particular lock. As with critical sections, condition variables are local to a process and, as with SRWLs, they are extremely lightweight: each one is the size of a pointer, and uses keyed events as the sole waiting and signaling mechanism, meaning no allocation of separate kernel event objects is required.

11- There is an important distinction between the Vista and legacy thread pools that will become apparent when we compare the APIs further. With the old thread pool, any callbacks that had to perform asynchronous I/O needed to get queued to a separate set of threads. That's because the pool reserved the right to retire ordinary callback threads while outstanding asynchronous I/O and APCs were running asynchronously with that thread, effectively canceling them. All of the threads in the Vista thread pool remain alive until asynchronous I/O operations and APCs have completed, so you need not worry about choosing one or the other.

12- Before Windows Vista there was no way, other than the ERROR_ALREADY_FIBER error, to determine whether a thread had already been fiberized. The new IsThreadAFiber function allows you to inquire about this. If the thread has already been converted to a fiber, this function returns TRUE, and otherwise it returns FALSE.[7]

13- Windows Vista Wait Chain Traversal (WCT). Windows Vista ships with a new set of Win32 APIs that fall under a single common feature, wait chain traversal, or WCT for short. WCT is meant to enable debuggers to capture wait graphs, much like what was shown earlier, in a nonintrusive way. Nonintrusive means that the debugged program need not be rewritten to support constructing an on demand wait graph: the WCT APIs gather and work with information already available in user-mode and the Windows kernel to produce a wait graph when requested to do so.[8]

14- A new API was added to Windows Vista and Server 2008 to take advantage of the fact that many I/Os use caches of OVERLAPPED data structures. When an I/O completes in the Windows kernel, it needs to lock the virtual memory pages containing the OVERLAPPEDs to guarantee they don't get paged out while devices are copying data to them. But all of this locking adds overhead to each I/O completion. The SetFileIoOverlappedRange function tells the kernel to lock the memory associated with a particular file's OVERLAPPED structures, so that it can avoid this overhead on subsequent I/Os.[9]

VI. Related Works

1- "Concurrent Programming Method for Digital Signal Processing"

The task of programming concurrent systems is substantially more difficult than the task of programming sequential systems with respect to both correctness and efficiency. The tendency in development of embedded, DSP systems and processors are shifting to multi core and multiprocessor setups as well. The problem of easy concurrency and algorithm development is an important for embedded and DSP systems as well. The goal of this paper is to define and present a high level language that allows description and development of signal processing algorithms. With the usage of a domain specific language, we can create compact and easy to understand definition of algorithms. In the paper the authors present the advantages granted by DSL for DSP applications. The created definitions are hardware independent can be executed and functionally verified. Efficient code can be generated for various targets without porting. The design of the presented DSL allows code generation for multi-core targets in case of computing-intensive algorithms, code generation for multiple streams, threads. Code reuse is supported by merging, re-grouping, and splitting of algorithms and groups of algorithms. [10]

2- "A Middleware for Concurrent Programming in MPI Applications"

A wide range of computationally intensive applications such as information retrieval, on-line analytical processing and data mining inherently require concurrency, because concurrent data maintenance, query processing and multi-user operation are functional requirements. Therefore, concurrent programming is a prerequisite for such systems. However, existing tools for parallel programming fail to meet these demands for concurrency and the adoption of parallel processing for these application domains is thus hindered. In this paper, we discuss the use of threads and concurrent programming constructs in the state of the art in parallel programming tools and environments.

We find that the necessary functionality is available, but often in an inconvenient and unreliable manner. Due to the fact that the programmability and maintainability of parallel programs is a major concern, we consider the existing solutions inadequate or insufficient. We argue that an additional layer of middleware for threads and inter-thread communication and synchronization is necessary to support the effective development of persistently deployed parallel services for our targeted application domain and present the MPI Threads (MPIT) interface specification.

We give several real-world examples to demonstrate its use and present performance benchmarks to illustrate the cost of the additional layer of indirection.[11]

3- "Design of an Empirical Study for Comparing the Usability of Concurrent Programming Languages"

The recent turn towards multicore processing architectures has made concurrency an important part of mainstream software development. As a result, an increasing number of developers have to learn to write concurrent programs, a task that is known to be hard even for the expert.

Language designers are therefore working on languages that promise to make concurrent programming “easier”. However, the claim that a new language is more usable than another

cannot be supported by purely theoretical considerations, but calls for empirical studies. In this paper, we present the design of a study to compare concurrent programming languages with respect to comprehending and debugging existing programs and writing correct new programs. A critical challenge for such a study is avoiding the bias that might be introduced during the training phase and when interpreting participants' solutions.

We address these issues by the use of self-study material and an evaluation scheme that exposes any subjective decisions of the corrector, or eliminates them altogether.We apply our design to a comparison of two object-oriented languages for concurrency, multithreaded Java and SCOOP (Simple Concurrent Object-Oriented Programming), in an academic setting. We obtain results in favor of SCOOP even though the study participants had previous training in writing multithreaded Java programs.[12]

VII. Concurrency in Windows 7

Mainly, there are two new features in Windows 7:

1. Support for more than 64 processors.
2. User-Mode Scheduled Threads.

Both of the new features author is going to talk about will be supported in the Microsoft Concurrency Runtime, which will be delivered as part of Visual Studio 10.

An important note about each of these features is that they're only supported on the 64-bit Windows 7 platform.

1) More Than 64 Processors

The most straightforward of these new features is Windows 7's support for more than 64 processors. With earlier Windows OS's, even high end servers could only schedule threads among a maximum of 64 processors. Windows 7 will allow threads to run on more than 64 processors by allowing threads to be affinitized to both a processor group, and a processor index within that group. Each group can have up to 64 processors, and Windows 7 supports a maximum of 4 processor groups.

However, unless you actively modify your application to affinitize work amongst other processor groups, you'll still be stuck with a maximum of 64 processors. The good news is that if you use the Microsoft Concurrency Runtime on Windows 7, you don't need to be concerned at all with these gory details. As always, the runtime takes care of everything for you, and will automatically determine the total amount of available concurrency (e.g., total number of cores), and utilize as many as it can during any parallel computation. This is an example of what we call a "light-up" scenario. Compile your Concurrency Runtime-enabled application once, and you can run it on everything, from your Core2-Duo up to your monster Win7 256-core server.[13]

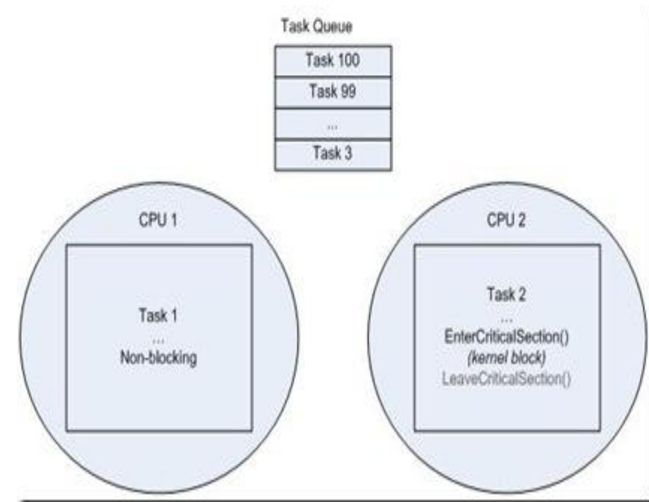
2) User Mode Scheduling

User Mode Scheduled Threads (UMS Threads) is another Windows 7 feature that "lights up" in the Concurrency Runtime.

As the name implies, UMS Threads are threads that are scheduled by a user-mode scheduler (like Concurrency Runtime's scheduler), instead of by the kernel. Scheduling threads in user mode has a couple of advantages:

1. A UMS Thread can be scheduled without a kernel transition, which can provide a performance boost.
2. Full use of the OS's quantum can be achieved if a UMS Thread blocks for any reason.

To illustrate the 2nd point, let's assume a very simple scheduler with a single work queue. In this example, I'll also assume that we have 100 tasks that can be run in parallel on 2 CPU's.



Here we've started with 100 items in our task queue, and two threads have picked up Task 1 and Task 2 and are running them in parallel. Unfortunately, Task 2 is going to block on a critical section. Obviously, we would like the scheduler (i.e. the Concurrency Runtime) to use CPU 2 to run the other queued tasks while Task 2 is blocked. Alas, with ordinary Win32 threads, the scheduler cannot tell the difference between a task that is performing a very long computation and a task that is simply blocked in the kernel. The end result is that until Task 2 unblocks, the Concurrency Runtime will not schedule any more tasks on CPU 2. Our 2-core machine just became a 1-core machine, and in the worst case, all 99 remaining tasks will be executed serially on CPU 1.

This situation can be improved somewhat by using the Concurrency Runtime's cooperative synchronization primitives (critical_section, reader_writer_lock, event) instead of Win32's kernel primitives. These runtime-aware primitives will cooperatively block a thread, informing the Concurrency Runtime that other work can be run on the CPU. In the above example, Task 2 will cooperatively block, but Task 3 can be run on another thread on CPU 2. All this involves several trips through the kernel to block one thread and unblock another, but it's certainly better than wasting the CPU.

The situation is improved even further on Windows 7 with UMS threads. When Task 2 blocks, the OS gives control back to Concurrency Runtime. It can now make a scheduling

decision and create a new thread to run Task 3 from the task queue. The new thread is scheduled in user-mode by the Concurrency Runtime, not by the OS, so the switch is very fast. Now both CPU 1 and CPU 2 can now be kept busy with the remaining 99 non-blocking tasks in the queue. When Task 2 gets unblocked, Win7 places its host thread back on a runnable list so that the Concurrency Runtime can schedule it – again, from user-mode – and Task 2 can be continued on any available CPU.

You might say, “hey my task doesn’t do any kernel blocking, so does this still help me?” The answer is yes. First, it’s really difficult to know whether your task will block at all. If you call “new” or “malloc” you may block on a heap lock. Even if you didn’t block, the operation might page-fault. An I/O operation will also cause a kernel transition. All these occurrences can take significant time and can stall forward progress on the core upon which they occur. These are opportunities for the scheduler to execute additional work on the now-idle CPU. Windows 7 UMS Threads enables these opportunities, and the result is greater throughput of tasks and more efficient CPU utilization.

Obviously the above example is highly simplified. A UMS Thread scheduler is an extremely complex piece of software to write, and managing state when an arbitrary page fault can swap you out is challenging to say the least. However, once again users of the Concurrency Runtime don’t have to be concerned with any of these gory details. Write your programs once using PPL or Agents, and your code will run using Win32 threads or UMS Threads.[14]

VIII. CONCLUSIONS

Concurrency is a double-edged sword. It can be used to do amazing new things and to enable new compute-intensive experiences that will only become possible with the amount of computing power available in the next generation of microprocessor architecture. Concurrent programming is complex and hard to achieve. In most cases the parallelization of software is not a straightforward and easy task. The realized concurrent programs usually have safety and performance issues. And in some situations concurrency is unavoidable. But it must also be used responsibly so as not to negatively impact software robustness and reliability. This paper's aim is to help you decide when it is appropriate, in what ways it is appropriate, specially, and, once you've answered those questions for your situation, to aid you in developing, testing, and maintaining concurrent software in Windows Vista.
The main advantages of Windows 7 in concurrency issues are:
1. Support for more than 64 processors.
2. User-Mode Scheduled Threads

IX. REFERENCES

[1] computer-dictionary.com.
[2] Anita Sabo, Norbert Schramm, "Abstractions for Concurrent Programming in Embedded Systems " Polytechnical Engineering College, Subotica, Serbia UVA, Subotica, Serbia . (2011)
[3] M. Ben-Ari. Principles of Concurrent and Distributed Programming, Second Edition. Addition Wesley, 2006.
[4] http://www.cs.rice.edu/~cork/book/node96.html
[5] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer,D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley, EECS.
[6] J. Larus. Spending Moore's Dividend. Microsoft Technical Report, MSR-TR-2008-69 (May 2008).
[7] M. Russinovich. Inside the Windows Vista Kernel: Part 1. TechNet Magazine,http://www.microsoft.com/technet/technetmag/issues/2007 /02/VistaKernel (2007).
[8] R. Saccone, A. Taskov. Concurrency: Synchronization Primitives New to Windows Vista. MSDN Magazine (2007).
[9] Microsoft. I/O Prioritization in Windows Vista: Recommendations for Application, Driver, and Device Developers for Supporting I/O Prioritization in Windows Vista. Microsoft.com Whitepaper (2006).
[10] Anita Sabo, Norbert Schramm, " Concurrent Programming Method for Digital Signal Processing " Polytechnical Engineering College, Subotica, Serbia, UVA, Subotica, Serbia. (2011)
[11] Tobias Berka, Helge Hagenauer, Nikolaos Papaspyrou, ” A Middleware for Concurrent Programming in MPI Applications " , Department of Computer Sciences, University of Salzburg , Austria. (2011)
[12] Sebastian Nanz, Faraz Torshizi, " Design of an Empirical Study for Comparing the Usability of Concurrent Programming Languages", (2011)
[13] http://www.microsoft.com/whdc/system/Sysinternals/MoreThan64proc. mspx.

[14] Don McCrady, "Concurrency Runtime and Windows 7",Parallel Programming in Native Code, PARALLEL PROGRAMMING USING C++ AMP, PPL AND AGENTS LIBRARIES.*4 FEB 2009*

Author's biography

Hadeel Tariq Ibrahim Al-Rayes. obtained her Bachelor in Computer Science from Baghdad University, College of Science and Master’s degrees in Computer Science/Information Technology from Iraqi Committee for computers and Informatic, Informatic Institute for Postgraduate Studies.Iraq,Baghdad. She is currently working as a head of Computer Science Dept. in Basic Education Col., Diyala University, Iraq now.
E-Mail: hadeelalrayes@yahoo.com
Mobile: +964 7902 162334