ISSN 2229-5518

Achieving a Real Multitasking, Multiprocessing and Multithreading by using Monitors

Authors- Mr. Manojkumar S. Sonawane , Ms.Mayura V. Gujarathi

Abstract :

Any object or thing in computer has its own “Monitor” So at a time only one task (program, process, or thread) can enter into monitor. So point to discuss is, at the depth or by looking from monitors view Where is the Multitasking (Multiprogramming, Multiprocessing, Multithreading)????? Even though there are DUAL Core Processors. So this paper discusses how we can achieve a real Multitasking, Multiprocessing & Multithreading by creating and maintaining number of monitors.

Index Term:

Monitor,Semaphor,Multitasking,Multithreading,Multiprocessing,Dual core processors, Synchronization mechanism.

1. INTRODUCTION

Today, a large number of software solutions are multi-threaded. Many of our desktop applications, such as word processors and web browsers, require multiple tasks executing concurrently to implement a seamless solution. Company services, such as purchasing or scheduling, use web-based, multi-tier solutions that must scale to allow millions of simultaneous users to perform transactions that access shared data. Though multi-threading enable the creation of complex systems, it introduces complexities in their development. When multiple threads are executed within a system, the execution order and time allotted for each is non-deterministic. As a result, they may display different behaviors from execution to execution. This non-determinism must be managed by means of thread synchronization to ensure that threads behave as expected.

One of the strengths of the programming language Java is its support for multithreading. This support centers on synchronization i.e.:- coordinating activities & data access among multiple threads. The mechanism used to support synchronization can be M*onitor*, Semaphore, Mutex and so on.

2. MONITORS

A monitor is like a building that contains one special room that can be occupied by only one thread at a time. The room usually contains some data. From the time a thread enters this room to the time it leaves, it has exclusive access to any data in the room. Entering the monitor building is called "entering the monitor." Entering the special room inside the building is called "acquiring the monitor." Occupying the room is called "owning the monitor," and leaving the room is called "releasing the monitor." Leaving the entire building is called "exiting the monitor." In addition to being associated with a bit of data, a monitor is associated with one or more bits of code, which in this paper will be called monitor regions. A monitor region is code that needs to be executed as one indivisible operation with respect to a particular monitor.

2.1 Why monitors?

Concurrency has always been an OS issue

Resource allocation is necessary among competing processes

Timer interrupts

Existing synchronization mechanisms (semaphores, locks) are subject to hard-to-find, subtle bugs.

One thread must be able to execute a monitor region from beginning to end without another thread concurrently executing a monitor region of the same monitor. A monitor enforces this one-thread-at-a-time execution of its monitor regions. The only way a thread can enter a monitor is by arriving at the beginning of one of the monitor regions associated with that monitor. The only way a thread can move forward and execute the monitor region is by acquiring the monitor. When a thread arrives at the beginning of a monitor region, it is placed into an entry set for the associated monitor. The entry set is like the front hallway of the monitor building. If no other thread is waiting in the entry set and no other thread currently owns the monitor, the thread acquires the monitor and continues executing the monitor region. When the thread finishes executing the monitor region, it exits (and releases) the monitor.

The synchronization supported by monitors is cooperation are mutual exclusion and Cooperation. Mutual exclusion helps keep threads from interfering with one

another while sharing data, cooperation helps threads to work together towards some common goal.

Cooperation is important when one thread needs some data to be in a particular state and another thread is responsible for getting the data into that state. For example, one thread, a "read thread," may be reading data from a buffer that another thread, a "write thread," is filling. The read thread needs the buffer to be in a "not empty" state before it can read any data out of the buffer. If the read thread discovers that the buffer is empty, it must wait. The write thread is responsible for filling the buffer with data. Once the write thread has done some more writing, the read thread can do some more reading. The form of monitor used by the Java virtual machine is called a "Wait and Notify" monitor. (It is also sometimes called a "Signal and Continue" monitor.) In this kind of monitor, a thread that currently owns the monitor can suspend itself inside the monitor by executing a wait command. When a thread executes a wait, it releases the monitor and enters a wait set. The thread will stay suspended in the wait set until some time after another thread executes a notify command inside the monitor. When a thread executes a notify, it continues to own the monitor until it releases the monitor of its own accord, either by executing a wait or by completing the monitor region. After the notifying thread has released the monitor, the waiting thread will be resurrected and will reacquire the monitor.

A graphical depiction of the kind of monitor used by a Java virtual machine is shown in Figure. This figure shows the monitor as three rectangles. In the center, a large rectangle contains a single thread, the monitor's owner. On the left, a small rectangle contains the entry set. On the right, another small rectangle contains the wait set. Active threads are shown as dark gray circles. Suspended threads are shown as light gray circles.

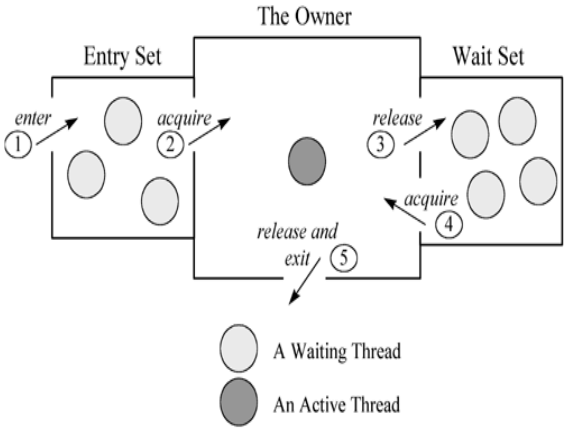

Figure 20-1. A Java monitor.

A Java monitor is a specialized class that is used to encapsulate application specific thread synchronization logic in a Java program. This class consists of one or more synchronized methods, each of which is a critical section of the monitor and guarded by a single lock object. This lock object is implicitly acquired and released each time a thread enters and exits a critical section. A model of a Java monitor can be seen in Figure.

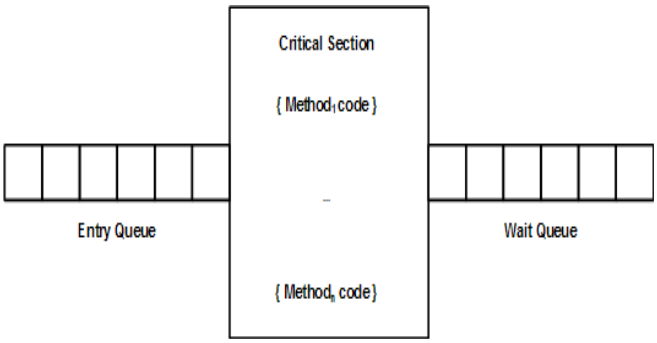

Fig: Model of a Java Monitor

Once a thread is executing in a synchronized method of the monitor, all other threads attempting to execute a synchronized method must wait in the monitor's entry. A thread that has entered a critical section, may exit the critical section either by successful completion of its method or by making a call to one of the wait primitives. If a wait primitive is called, the thread must release its lock after which it is moved to the monitor's wait queue. After a thread releases its lock and exits the critical section, a thread is selected from the entry queue, allowing it to acquire the lock and enter the critical section. For a threads to exit the wait queue, another thread executing in a critical section must make a notify or notifyAll primitive call. If the wait queue is not empty at the time these calls are made, one or more threads are moved from the wait queue to the entry queue before execution resumes in the calling thread. This type of signaling discipline is known as signal-and-continue. The only difference between these primitives is that the notify call will result in the selection of a random thread from the wait queue, while a notifyAll call affects all threads in the wait queue. It is also possible for a thread to exit the wait queue if it called a wait primitive with a timeout argument. In this event, if the thread is still waiting after the timeout period has expired, it will be moved to the entry queue. We will not address this type of wait primitive in our approach.

- **Condition variable queues:**

There may be any number of condition variable queues for a given monitor. Each queue has associated wait and notify methods for putting tasks in the queue and taking them out.

- **The Entry queue:**

Each monitor has one entry queue. When a task attempts to access a monitor method from outside the monitor, it is

put in the monitor's entry queue.

- **The Signaller queue:**

Each monitor has one signaller queue. When a task performs a notify, it is put in this queue.

- **The Waiting queue:**

Each monitor has one waiting queue. When a task is removed from one of the condition variable queues, it is put in this waiting queue.

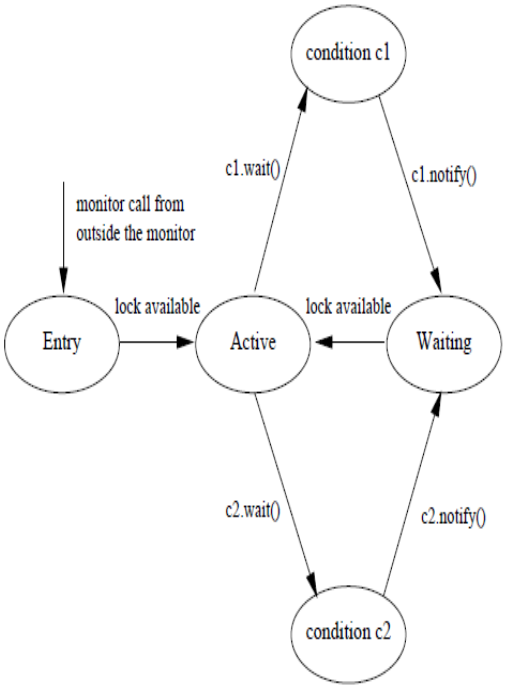

Figure 1: The queues associated with a monitor.

2.2 Monitors: a language construct

- Monitors are a programming language construct
- Anonymous lock issues handled by compiler and OS
- Detection of invalid accesses to critical sections happens at compile time.
- Process of detection can be automated by compiler by scanning the text of the program.

2.3 An Abstract Monitor

```
name: monitor
…local declarations
…initialize local data
 proc1 (…parameters)
 …statement list
 proc2 (…parameters)
 …statement list
proc3 (…parameters)
        …statement list
```

Monitor Example:-

```
car: monitor
//monitor declaration
   occupied: Boolean;  occupied := false;      //local
variables / initializations
   nonOccupied: condition;
   procedure enterCar()
//procedure
          if occupied then nonOccupied.wait;
   occupied = true;
     procedure exitCar()
//procedure
       occupied = false;
       nonOccupied.signal;
```

2.4 Problems solved by monitors

- Mutual exclusion.
- Encapsulation of data.
- Compiler can automatically scan program text for some types of synchronization bugs.
- Synchronization of shared data access simplified vs. semaphores and locks.
- Good for problems that require course granularity.
- Invariants are guaranteed after waits
- Theoretically, a process that waits on a condition doesn't have to retest the condition when it is awakened.

2.5 Advantages of Monitor

- Data access synchronization simplified (vs. semaphores or locks)
- Better encapsulation.

3. PROPOSED WORK

So my proposed work suggests we can achieve real multitasking, multiprocessing, multithreading by creating number of monitors for object which will execute multiple threads at a time.

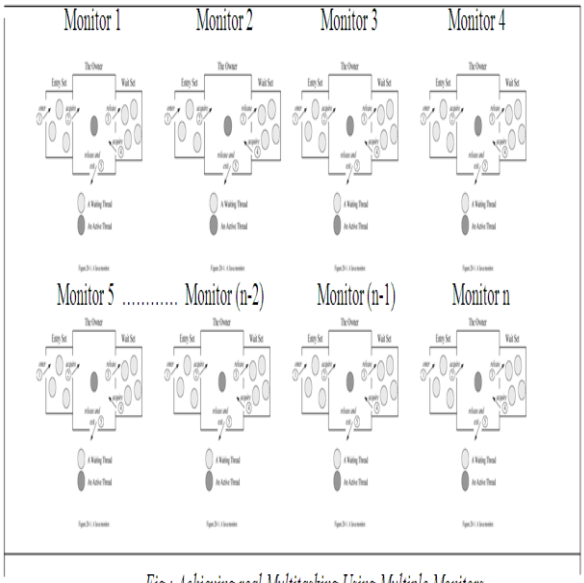

Fig : Achieving real Multitasking Using Multiple Monitors

A "ready" or "waiting" process has been loaded into main memory and is awaiting execution on a CPU (to be context switched onto the CPU by the dispatcher, or short-term scheduler). There may be many "ready" processes at any one point of the systems execution ,all other "concurrently executing" processes will be waiting for execution. A ready queue is used in computer scheduling. Modern computers are capable of running many different programs or processes at the same time. Processes that are ready for the CPU are kept in a queue for "ready" processes. Other processes that are waiting for an event to occur, such as loading information from a hard drive or waiting on an internet connection are not in the ready queue, they are putted in waiting queue.

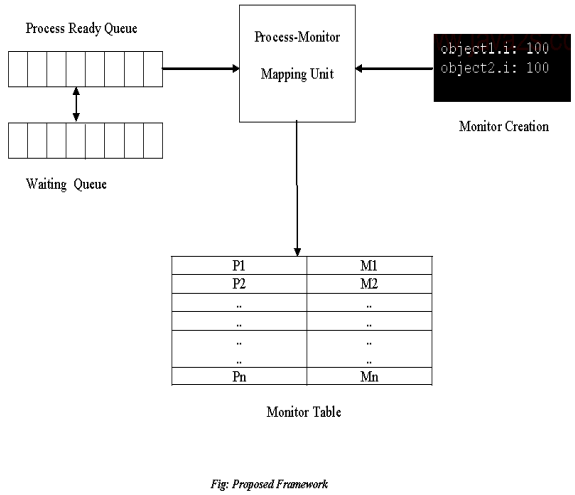

Fig: Proposed Framework

As shown in framework CPU will fetch one by one

pr
obj
Cr
ma
wi
obj
by

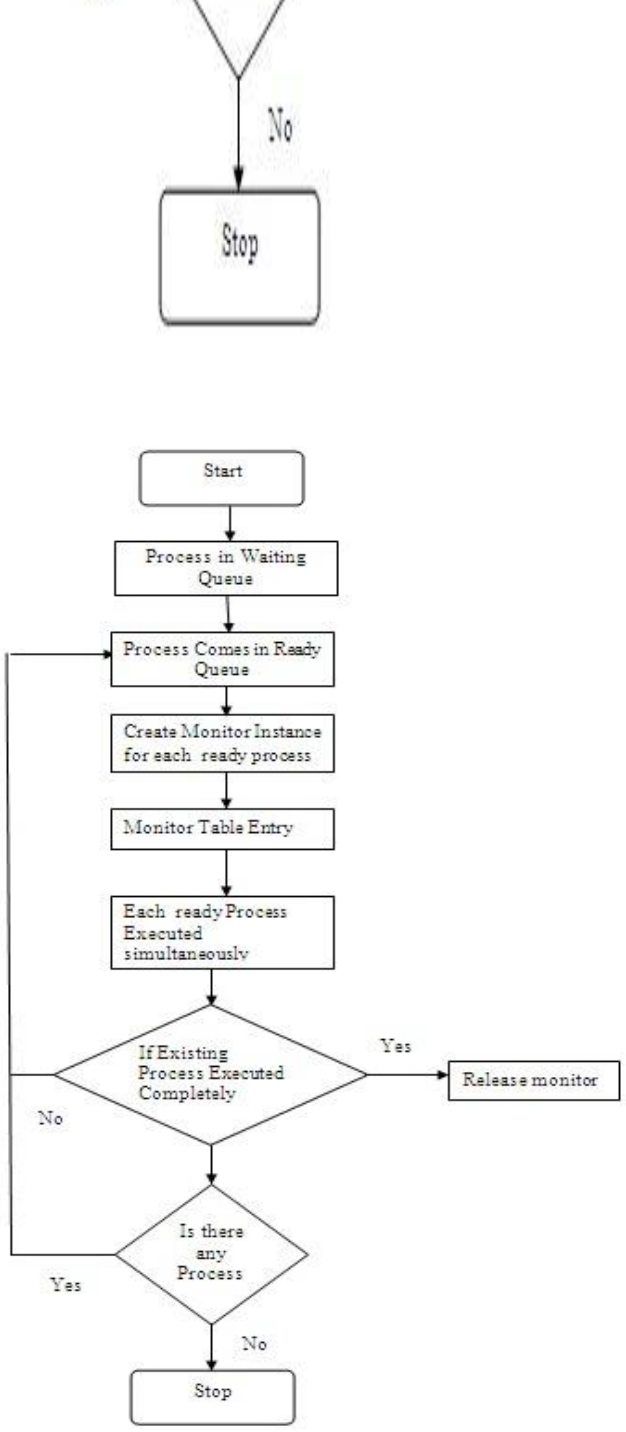

Fig: Workflow Diagram

4. CONCLUSION

Even though monitor is a higher level, easier to use abstraction, better encapsulation as compare to semaphores/locks. It has several drawbacks like in conventional multitasking, multiprocessing, multithreading at a time only one process is able to use the object's monitor. So problem arises when more than one processes needs to access same monitor at the same time. Because of this we are not getting actual multitasking.

Using this framework process will get required monitor whenever needed because we have created multiple monitors of an object. So the proposed work suggests we

can achieve real multitasking, multiprocessing, multithreading by creating number of monitors for object which will used by multiple threads or processes at the same time.By creating a number of monitors real multitasking, multiprocessing, multithreading achieved with faster speed.

5. REFERENCES

"Monitors: An Operating System Structuring Concept," Hoare.

1. Modern Operating Systems, Second Edition, Tannenbaum, pp. 115-119.
2. Jon Walpole, correspondence.
3. Emerson Murphy-Hill presentation from CS533, Winter 2005
4. http://en.wikipedia.org/wiki/C. A. R. Hoare