**MONITORS WITH IMPLICIT SIGNALS**

By Michael Xing, Jivan Luu, Taylor Morrow, Yu-Chun Huang

**ABSTRACT**

This document presents the implementation of a new type of monitor lock that uses implicit signals. While normal monitors require the user to signal correctly for programs to work properly, implicit signaling implements and hides this step for the user. However, the cost of signaling for the user incurs overhead, and this overhead varies under different design implementations. In this paper, we discuss the idea and concept of monitor with implicit signaling. We provide the implementation for the implicit monitor and evaluate the performances of various design strategies.

**INTRODUCTION**

A monitor is a synchronization primitive used in concurrent programming to ensure mutual exclusion of data access between concurrent threads. Since multiple threads may attempt to access shared data at the same time, they use the interface provided by the monitor to guarantee that no more than one thread accesses a piece of shared data at any one time. In particular, the normal concept of a monitor in the context of Java includes locking and unlocking implicitly surrounding the execution of sections of code within **synchronized** blocks, as well as explicit **wait** and **notify** functionality for communicating between threads. **Wait** inserts the thread into a waiting queue, and **notify** wakes up a thread in a waiting queue.

The explicit monitor provided by Java, while sufficient to achieve mutual exclusion, has potential costs, because it does not distinguish between threads that wait for different reasons. The monitor cannot distinguish between different types of waiting threads; if the programmer only notifies one thread, that thread may not be able to do anything, resulting in deadlock, but if the programmer notifies all threads, then many threads will be woken up unnecessarily. Monitors with explicit conditional signaling can be used to solve this problem. Java provides the classes **ReentrentLock** and **Condition** to achieve conditional signaling by allowing the programmer to break up waiting threads into groups and to signal a specific type of thread. This mechanism is “explicit,” meaning that the programmer decides the actual signaling.

In this paper, we will present a monitor with “implicit” signaling, one that frees the programmer from the duty of signaling. We will first describe the operations of a monitor as well as the difference between explicit- and implicit-signaling. Then we will present our implementation of the implicit-signaling monitor. Finally, we explore different design alternatives and evaluate the performance of each.

**PROJECT DESCRIPTION**

A normal java monitor has four basic operations provided for the user program: **lock**, **unlock**, **wait**, and **signal**. The user program calls **lock** prior to accessing the data structure and **unlock** right before exiting the critical section. When the program needs to wait for certain conditions to continue executing, it calls **wait** to suspend itself and allowing other threads to access the data structure. After the program executes a section of code that may change the waiting conditions of threads, it calls **signal** to wake up the possible waiting threads. Incorrect signaling can cause deadlock or violation of mutual exclusion. Therefore, there is incentive for a monitor implementation that hides the signaling mechanism from the user. That way, the user does not have to signal too many or too little threads to be woken up. With an implicit-signaling monitor, the signal operation is hidden from the user. Instead, the **unlock** operation both unlocks the monitor and signals the corresponding waiting threads.

**IMPLEMENTATION**

The implementation of the implicit monitor class is shown below in figure 1. The lock() function uses the ReentrantLock to satisfy the requirement of mutual exclusion. If there is another thread that has the lock, the lock() is blocked until it acquires the lock. Notice that the lock() and unlock() also emulates the use of “synchronized” functions when using normal monitor.

After the user has acquired the lock and entered the critical section, the user may use the wait(ImplicitCondition c) method for conditional synchronization. In other words, the user will provide an implicitCondition object and the method will check if the condition has become true. The wait() method will block until the condition becomes true. Inside the wait() method, it first checks if the ImplicitCondition provided by the user is part of ArrayList of previous ImplicitConditions. If it’s not, then it’s added to the conditions ArrayList. Next, a while loop checks if the ImplicitCondition is true. If not, the thread waits and lets other threads execute until it is woken up. The while loop terminates once the thread is woken up and the ImplicitCondition returns true.

After the wait(c) returns, the thread may continue executing in the critical section. When it is ready to exit the critical section, it must call unlock() to allow other threads to enter the critical section. This is where the actual implicit signal takes place. Inside the unlock() function, the ImplicitMonitor goes through the array of conditions and checks if each condition has become true. For each condition that is true, the monitor wakes up one corresponding waiting thread.

One difficulty with ImplicitMonitor is the different types of conditions that might be given by the user for conditional synchronization. To solve this problem, the user of the ImplicitMonitor is required to implement the ImplicitCondition interface for each type of condition the user requires. More specifically, the user is required to write the check function required by the ImplicitCondition interface. This way, the ImplicitMonitor will be able use the check() method to return a boolean depending on whether the ImplicitCondition is true or false.

public class ImplicitMonitor

{

public void lock();

public void unlock();

public void wait(ImplicitCondition c);

}

**DESIGN ALTERNATIVES**

To evaluate the effectiveness of an implicit monitor, several monitors were created for the purpose of observing the message complexity and time efficiency differences. There were three main types of monitors evaluated, which vary by the action taken by the monitor when a process has finished:

1. Standard monitor that uses notifyAll() to wake all threads
2. Monitor that checks which conditions are true and uses signalAll() to wake threads waiting on that condition
3. Monitor that checks which conditions are true and uses signal() to wake a single thread waiting for that condition

An explicit monitor using monitor type 1 is created, but no equivalent implicit monitor is created. This is because we believe that a monitor that notifies all threads (without conditions) is simple enough that an equivalent implicit monitor would be unnecessary. For monitor types 2 and 3 both an implicit and explicit monitor are created to compare with the type 1 monitor to see the differences between the implicit versus explicit implementations.

**PERFORMANCE**

To evaluate the performance of the monitors, we implemented a test based on the producer/consumer model with a FIFO queue interface between producers and consumers. Testing conditions were devised such that each operation would take a negligible amount of time to complete. Threads would have to wait for a notEmpty or notFull condition to execute; we wanted our test to allow for sufficient threads to be waiting on each condition. This is accomplished by making a large number of producer and consumer threads and having each thread do a large number of their assigned operation. The performance is significantly dependent on the size of the FIFO. At a FIFO size of one, there is more wait time because put and get operations must be perfectly interleaved; for every put, there must be a corresponding get, and so forth. However, at larger FIFO sizes it is possible to get away with multiple puts/gets in a row, meaning there is a better chance for threads to be able to access the critical section.

Testing conditions used:

1,000 producer threads (put)

1,000 consumer threads (get)

Each thread will perform its operation 100 times.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Queue Size 1 | Queue Size 2 | Queue Size 3 |
| Explicit Monitor, notifyAll | 6319 | 3238 | 1036 |
| Explicit Monitor w/ Condtion | 2717 | 2718 | 2043 |
| Implicit Monitor w/ Condition | 2757 | 2352 | 1588 |
| Explicit Monitor w/ Condtion, signalAll | 546329 | 299730 | 63900 |
| Implicit Monitor w/ Condition, signalAll | 636000 | 332238 | 66843 |

Queue size 1:

Analyzing the results of testing each monitor three times and averaging the data, the standard notifyAll explicit monitor was found to be significantly slower than both the implicit and explicit monitors with conditions. This is expected because of the large cost incurred when notifying all the threads for the explicit monitor. The implicit monitor with condition is slightly slower than the explicit monitor with condition because they will both notify the same number of threads, and there is overhead required in the implicit monitor to check which conditions are true.

Queue size 2:

At a queue size of two, the changes in run time can be attributed to the ability to wake multiple threads after a process has finished. The explicit monitor with notifyAll is still slower than the implicit and explicit condition monitors, but displayed the largest speedup compared to the results with queue size of 1. The implicit monitor with condition is observably faster than the explicit monitor with condition despite being slower at queue size one, because when there is one element in the queue, both conditions will be true causing two threads to be woken up. This leads to two processes being able to put/get from the FIFO in succession despite requiring extra overhead to be woken up.

Queue size 10:

At a large queue size of ten, it is observed that the monitors that notify the most threads complete fastest. The difference in overhead is negligible compared to the speedup gained by being able to notify multiple threads at the end of an operation. It is observed that explicit notifyAll monitor, which was relatively slow at queue size one due to its large cost, becomes the fastest at a large queue size because it notifies the most threads. The implicit monitor with condition continues to be faster than the explicit monitor with condition because of its ability to wake multiple threads - one for each condition.

**CONCLUSION**

This paper documented the concept and implementation of monitor with implicit signaling. Moreover, the paper explored different design implementations and measured the performance of each design option. Monitor with implicit signaling demands more overhead to check conditions and signaling threads. At the same time, implicit signaling allows easier use by the user since the signaling mechanism is hidden from the user. While we have demonstrated the performance of implementation for the implicit signaling mechanism, we have also opened many interesting future directions. More performances testing should be performed for other applications besides put() and get() to evaluate the place of usage for implicit signaling. Ultimately, Implicit signaling, at the expense of overhead time, prevents thread deadlocks and achieves more efficient signaling under specific cases.