SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2001; **31**:667–695 (DOI: 10.1002/spe.383)

An enhanced thread synchronization mechanism for Java

Hsin-Ta Chiao and Shyan-Ming Yuan*,†

Department of Computer and Information Science, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu 300, Taiwan

SUMMARY

The thread synchronization mechanism of Java is derived from Hoare's monitor concept. In the authors' view, however, it is over simplified and suffers the following four drawbacks. First, it belongs to a category of no-priority monitor, the design of which, as reported in the literature on concurrent programming, is not well rated. Second, it offers only one condition queue. Where more than one long-term synchronization event is required, this restriction both degrades performance and further complicates the ordering problems that a no-priority monitor presents. Third, it lacks the support for building more elaborate scheduling programs. Fourth, during nested monitor invocations, deadlock may occur. In this paper, we first analyze these drawbacks in depth before proceeding to present our own proposal, which is a new monitor-based thread synchronization mechanism that we term EMonitor. This mechanism is implemented solely by Java, thus avoiding the need for any modification to the underlying Java Virtual Machine. A preprocessor is employed to translate the EMonitor syntax into the pure Java codes that invoke the EMonitor class libraries. We conclude with a comparison of the performance of the two monitors and allow the experimental results to demonstrate that, in most cases, replacing the Java version with the EMonitor version for developing concurrent Java objects is perfectly feasible. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: Java™; thread synchronization mechanism; monitor

THE JAVA MONITOR AND ITS DRAWBACKS

The thread synchronization mechanism of Java [1] is a simplification of Hoare's original monitor concept [2]. To implement the monitor, each Java object contains a monitor lock and a condition queue. The keyword `synchronized` can be inserted into the definition of a method for the purpose of specifying the method as a synchronized method. In a Java object, only one synchronized method

*Correspondence to: Shyan-Ming Yuan, Department of Computer and Information Science, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu 300, Taiwan.
†E-mail: smyuan@cis.nctu.edu.tw

Received 23 November 1999
Revised 3 February 2000 and 27 September 2000
Accepted 6 December 2000

can be run at a time. In addition, Java also offers synchronized blocks to reduce the size of critical sections. For condition synchronization, Java provides each object with the following three methods: `wait()`, `notify()`, and `notifyAll()`. (In this paper, the Java term *notify* and the conventional monitor term *signal* are interchangeable). These methods can only be invoked inside a synchronized method or synchronized block.

The design philosophy of the Java monitor is to keep things as simple as possible. However, we consider it over simplified. It has several drawbacks that both complicate the design of concurrent objects, and bring extra synchronization overhead when contention increases. Besides, deadlock may occur during nested monitor invocations, and can be prevented only by considerable effort. In this section, we will discuss the drawbacks in detail one by one.

No-priority monitor

Many monitor variations have been proposed since Hoare's first monitor [2]. Basically, they can be classified according to both their monitor locks and the signal semantics they offer [3]. Logically, a monitor lock contains three (or less) queues for storing the threads that intend to enter the monitor. The entry queue holds the threads that cannot enter the monitor immediately after they invoke any one of the monitor methods. The waiting queue stores the threads that have been signaled by other threads. The signaler queue contains signaler threads that have signaled another thread in any condition queue. To prevent starvation, the entry queue priority Ep should always be the lowest. For a monitor, if its Ep is equal to either the Sp (the signaler queue priority) or the Wp (the waiting queue priority), we refer to it as a *no-priority monitor*. In any other case it is a *priority monitor*.

Since both `notify()` and `notifyAll()` do not release the monitor lock, and merely resume either one or all of the pending threads in the condition queue, the signal semantic offered by the Java monitor belongs to the so-called *non-blocking signal* [3,4]. In addition, since no signaler thread has to be stored after either `notify()` or `notifyAll()` is invoked, the Java monitor lock logically contains no signaler queue. Furthermore, since the Java monitor lock combines entry and waiting queues into a single queue, it is reasonable to assume that the Ep and the Wp of the Java monitor lock are equal. Hence, we may classify the Java monitor as no-priority, but because of this property, it presents the following two problems.

The first is that the post-condition of `wait()` may be different from the pre-condition of `notify()` [3–5]. In fact, this condition-breakup problem is caused by another intruding thread that preempts a thread that has been notified, and enters the monitor to destroy the pre-condition of `notify()`. Hence, as shown in the following `Semaphore` Java class, the `P()` method has to enclose the `wait()` method in a while loop to determine whether it is allowed to proceed.

```
public class Semaphore {

  private int SemaphoreCounter = 0;

  public synchronized void P( ) throws Exception {
    while(SemaphoreCounter == 0)
      wait( );
    SemaphoreCounter--;
```



```
  }

  public synchronized void V( ) throws Exception {
    SemaphoreCounter++;
    notify( );
  }

}
```

The second problem concerns the execution order among the synchronized methods or synchronized blocks becoming more difficult to control. This may complicate the design of scheduling programs, where ordering is important [3,5,6]. For example, the above `Semaphore` class cannot guarantee the FIFO ordering of completing the `P()` method. (Here we assume that both the lock queue and the condition queue of the Java monitor are FIFO-ordered. In fact their ordering is unspecified in the specification of Java [1] and is implementation dependent.) Let us consider the following situation. A thread T_1 holds the monitor lock and begins to run the `V()` method. Another thread T_2 is a pending thread at the front of the condition queue. The third thread T_3 intends to invoke the `P()` method, and waits in the Java monitor's lock queue. Thread T_1 first signals thread T_2, sets the value of the `SemaphoreCounter` to one, and returns. Then, thread T_3 (the new invocation of `P()`) preempts thread T_2 (the old invocation), and the FIFO ordering is violated. The correct implementation of FIFO semaphore is shown below, but it is more complex and less intuitive.

```
public class FIFOSemaphore {

  private int SemaphoreCounter = 0, PendingCounter = 0;

  public synchronized void P( ) throws Exception {
    if(SemaphoreCounter > 0)
      SemaphoreCounter--;
    else { // Semaphore Counter == 0
      PendingCounter++;
      wait( );
    }
  }

  public synchronized void V( ) throws Exception {
    if(PendingCounter == 0)
      SemaphoreCounter++;
    else { // PendingCounter > 0
      PendingCounter--;
      notify( );
    }
  }

}
```

In fact, a no-priority monitor not only complicates concurrent program design, but also incurs more thread context switches. Consequently, it is not well rated in the concurrent programming literature [3].

Only one condition queue is offered

As stated previously, the Java monitor offers only one condition queue. When only one long-term synchronization event is required, such as the above FIFO semaphore, the complications caused by a no-priority monitor remain under control. However, if more than one long-term synchronization event is required, the execution order and fairness of the invoked methods become unmanageable. For example, the following `BoundedBuffer` class requires two long-term synchronization events: with the first, the `Buffer` becomes not full, and with the second, the `Buffer` becomes not empty. This example is still simple enough for us to ensure that, at any one time, all pending threads in the condition queue are waiting for the same synchronization event. However, we cannot ensure that the execution order of the `Put()` method is always FIFO-ordered (and neither can we ensure this for the `Get()` method). Unlike the previous FIFO semaphore, the plain FIFO-ordered implementation of a bounded buffer is almost impossible to construct.

```
public class BoundedBuffer {

  private int Front = 0, Rear = 0, Counter = 0;
  private static final int Size = 10;
  private Object Buffer[ ] = new Object[Size];

  public synchronized void Put(Object Obj) throws Exception {
    while(Counter == Size)
      wait( );
    Buffer[Rear] = Obj; Rear = (Rear + 1) % Size; Counter++;
    if(Counter == 1)
      notifyAll( );
  }

  public synchronized Object Get( ) throws Exception {
    Object Result;
    while(Counter == 0)
      wait( );
    Result = Buffer[Front]; Front = (Front + 1) % Size; Counter--;
    if(Counter == Size - 1)
      notifyAll( );
    return Result;
  }

}
```

Another problem of this implementation concerns performance. In order to ensure that all pending threads in the condition queue are waiting for the same synchronization event, when either of the long-

term synchronization events happens, the `notifyAll()` method has to be invoked to wake up all pending threads in the condition queue. This action may generate lots of thread context switches and lower the performance.

In brief, for any concurrent programs that require multiple long-term synchronization events, to construct them by using the Java monitor straightforwardly will incur the same problems with ordering, fairness, and performance.

No support for scheduling

Scheduling is an important aspect of thread synchronization [7,8]. To select out one from a number of received requests, a scheduler must have some global information about the requests. When a thread synchronization mechanism is able to provide tailor-made facilities to help retain scheduling information, constructing schedulers is greatly simplified. For example, many existing monitors [5,9–11] offer a prioritized condition queue to simplify the task of writing static scheduling programs. However, the condition queue in the Java monitor is primitive and offers no support for scheduling. Consequently, a Java program has to retain the required scheduling information in other more expensive ways.

Deadlock of inter-monitor nested calls

Since the Java monitor has a nested mutually exclusive lock, deadlock never occurs during an intra-monitor nested call, which means that a synchronized method calls any synchronized method within the same object. However, deadlock may arise in inter-monitor nested calls, which means that a synchronized method of an object invokes a synchronized method in another object. The possible deadlocks can be further divided into two categories. The first is *mutually dependent deadlock* [4,12,13]. Suppose that thread T_1 is running inside object M_1's synchronized method at the same time that thread T_2 is running inside object M_2's synchronized method. This kind of deadlock happens when thread T_1 intends to invoke object M_2's synchronized method at the same time as thread T_2 intends to invoke object M_1's synchronized method. The second is *condition-wait deadlock* [6]. Suppose that thread T_3 consecutively invokes N synchronized methods from objects $M_1, M_2, M_3, \ldots,$ to object M_N, after which the thread calls `wait()` to block itself in object M_N, and releases the monitor lock of M_N only. Then, assume that thread T_3 expects to be signaled by another thread T_4 that intends to invoke `notify()` in object M_N. If, before thread T_4 reaches object M_N, it goes through any object among object M_1 to the object M_{N-1}, the condition-wait deadlock arises.

THE EMonitor

To overcome the above-discussed drawbacks, we propose to replace the Java monitor with a new monitor-based thread synchronization mechanism. The new monitor, termed EMonitor, has the following design goals:

- it must successfully deal with all the above-stated Java monitor drawbacks;

- it should if possible offer similar syntaxes and features to those of the Java monitor, and thus be more acceptable to Java programmers, and reduce the difficulty of translating existing concurrent Java programs into the EMonitor's form; and
- it should be portable. Java is a programming language that strongly stresses the property of 'write once and run everywhere'. If it is to be a really practical solution, the EMonitor itself must be implemented solely by Java without any native codes.

The features of the EMonitor

The EMonitor is a language extension to Java. It is a priority monitor, and provides three kinds of condition queues for different scheduling requirements. Of course, inside an object protected by the EMonitor, more than one condition queue is allowed.

Conventionally, the semantic of Java inter-monitor nested calls belongs to the so-called *closed-call semantic*. To prevent the above-discussed deadlocks of inter-monitor nested calls, the EMonitor provides another semantic—the *open-call semantic* [5,14]. Before a thread performs an open inter-monitor nested call, it will completely release the caller object's monitor lock, and save the lock-nested count. After the inter-monitor nested call returns, this thread will acquire the monitor lock again, reenter the caller monitor, and then restore the previous lock-nested count. Since any thread at any one time holds at most one monitor lock, mutually dependent deadlock and condition-wait deadlock are eliminated. However, open calls enforce extra restrictions on the programming style. Before invoking an open call, the calling thread has to transfer the caller monitor's state to a consistent state, in which all the monitor invariants are true. Since preventing deadlock and keeping the programming style simple are conflicting requirements, the EMonitor offers both open and closed call semantics.

An open call seems to be able to be simulated by synchronized blocks of Java. For example, suppose a Java method `Source()` intends to issue an open call to another method `Dest.f()`. The following code fragment shows the implementation that employs two synchronized blocks:

```
void Source() {
  synchronized(this) {
    ...
  }
  Dest.f( );
  synchronized(this) {
    ...
  }
}
```

In fact, this implementation works correctly only when the nested count of the monitor lock is never larger than one. This restriction implies that intra-monitor nested calls are not allowed. Consider the situation where a thread that already holds the monitor lock intends to invoke the `Source()` method. Leaving the first synchronized block reduces the lock nested count only by one. Since the count is still larger than zero, the monitor lock will not be released. Consequently, mutually dependent deadlock and condition-wait deadlock may still occur.

The Java monitor provides a non-blocking signal semantic only. We choose to retain this semantic in our EMonitor because it is efficient and for reasons of compatibility, but we also feel that a blocking

signal semantic [3] is practical and should be included. Using a blocking signal means that the post condition of the wait method always matches the pre-condition of the corresponding signal method, making the blocking signal easier to use. Although, in theory, a blocking signal introduces more context switches than a non-blocking one, our experience indicates that this semantic is slightly less efficient than the non-blocking signal semantic and thus encourages us to include it in the EMonitor. After consideration, we decided not to include other possible signal semantics. For example, the quasi-blocking signal and the automatic signal, both of which are regarded as impractical semantics in the literature [3], were excluded. Neither, because of possible implementation issues, did we include the immediate-return signal semantic. The reasons for these decisions will be given later when we discuss the implementation of the EMonitor. In summary, the EMonitor supports both blocking signal and non-blocking signal semantics. The non-blocking signal-all semantic of the Java monitor is also included.

The syntax of the EMonitor

The EMonitor can be used in three ways, each with a different granularity of mutual exclusion. The first option is through an *EMonitored class*, by which a new class modifier, `EMonitored`, is added to Java. When this keyword is inserted into the declaration of a Java class, this class becomes an EMonitored class. In an instance of an EMonitored class, only one invoked method can run at the same time. However, for executing more than one method concurrently in an object, the *EMonitored method* can be employed. We overload the keyword `EMonitored` as a new method modifier for Java. Once the declaration of a method includes the keyword, it becomes an EMonitored method. Only one invoked EMonitored method can be run in an object at the same time. Since the implementation of an EMonitored class and an EMonitored method is similar, we consider only the EMonitored class in the rest of this section.

The third option is through *EMonitored blocks*, which are similar to synchronized blocks of Java. The syntax is

```
EMonitored(EMonitorObject) { ... }
```

The `EMonitoredObject` is an instance of the `EMonitor` class, which implements the monitor lock of the EMonitor (also called the EMonitor lock). All the statements inside the pair of braces belong to the EMonitored block. Only one of the EMonitored blocks employing the same `EMonitoredObject` is allowed to proceed at the same time.

The EMonitor provides three kinds of condition queues; FIFO, prioritized, and customizable condition queues. Their class hierarchies are shown in Figure 1. A description of each class in the hierarchies follows.

- `Condition` class. This abstract class defines the common properties of the other three public condition queue classes. The responsibility of this class is to maintain the association between a condition queue and an EMonitor lock. The condition queue can only be accessed within the critical section that is protected by the associated EMonitor lock. Otherwise, the `IllegalEMonitorStateException` defined in the EMonitor class libraries is thrown.
- `FIFOCondition` class: This class is for the FIFO condition queue, which is primitive and similar to that offered by Java.

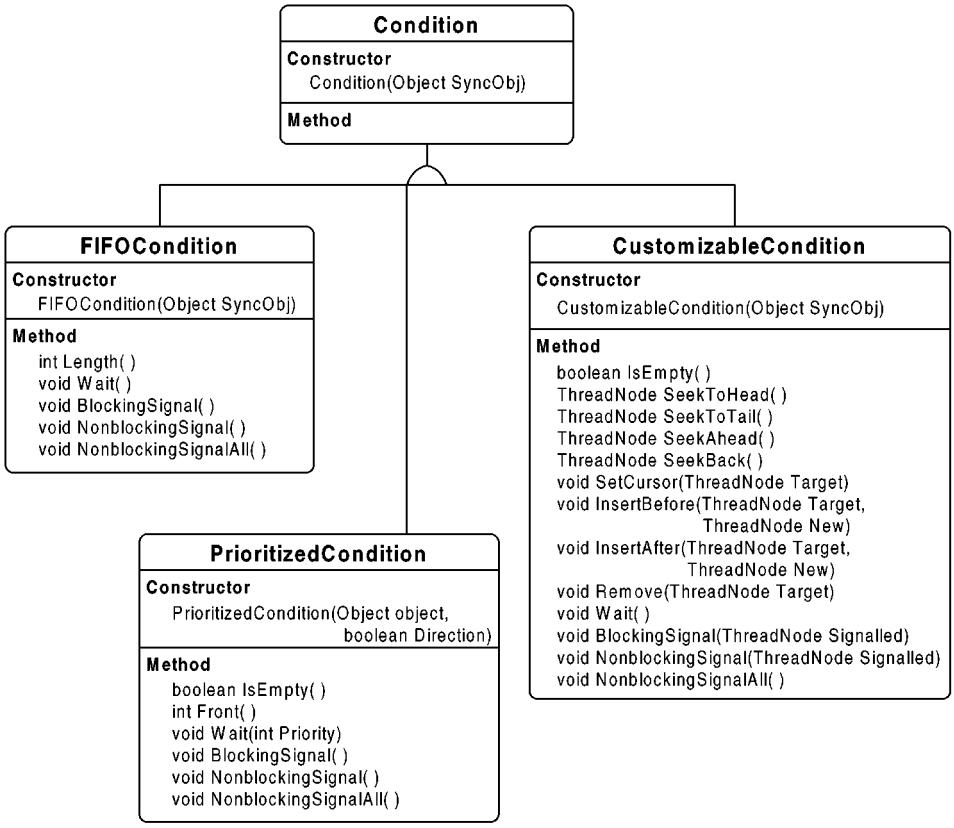


Figure 1. The class hierarchies of condition queues.

- `PrioritizedCondition` class: This class is for the prioritized condition queue, which is similar to that offered by other existing monitors. A specialized wait method, the `Wait(int Priority)`, is provided. In a prioritized condition queue, the pending threads are sorted according to their priorities. The sorting direction depends on the `Direction` parameter of the constructor. In addition, a `Front()` method is offered for returning the priority of the first pending thread.
- `CustomizableCondition` class: This class is for the customizable condition queue, which is very generalized and has enough expressive power to deal with dynamic scheduling problems. Basically, it maintains a list of instances of the `ThreadNode` class (usually its subclass). Each instance of the `ThreadNode` class represents a pending thread, and holds the pending thread's object reference. A programmer can customize a subclass of the `ThreadNode` class, and add any new attributes that have to be associated with a pending thread. Scheduling information is

carried in these user-defined attributes. Each customizable condition queue has its own cursor, which can be manipulated by the following five methods: `SeekToHead()`, `SeekToTail()`, `SeekAhead()`, `SeekBack()`, and `SetCursor()`. A new instance of the `ThreadNode` class can be inserted into a designated position through either the `InsertBefore()` or the `InsertAfter()` method. Before a thread blocks itself in a customizable condition queue, it should create a new instance of the `ThreadNode`'s subclass, set the necessary attributes, and then traverse the condition queue to find the right position for storing the new instance. Once the new instance is inserted into the condition queue, the thread can invoke the `Wait()` method to suspend itself. Signaling a designated pending thread is done through either the `BlockingSignal()` or the `NonblockingSignal()` method, as long as the `ThreadNode` instance of the pending thread is provided. If an unwanted pending thread is discovered, it can be deleted by the `Remove()` method.

The EMonitor-version `BoundedBuffer` class is shown below. Here we implement it as an EMonitored class. This implementation avoids the ordering and performance problems of the Java version.

```
import EMonitor.lang.*;

public EMonitored class BoundedBuffer {

  private int Front = 0, Rear = 0, Counter = 0;
  private static final int Size = 10;
  private Object Buffer[ ] = new Object[Size];
  private FIFOCondition NotEmpty, NotFull;

  public BoundedBuffer( ) throws Exception {
    NotEmpty = new FIFOCondition(this);
    NotFull = new FIFOCondition(this);
  }

  public void Put(Object Obj) throws Exception {
    if(Counter == Size)
      NotFull.Wait( );
    Buffer[Rear] = Obj; Rear = (Rear + 1) % Size; Counter++;
    if(Counter == 1)
      NotEmpty.NonblockingSignal( );
  }
  public Object Get( ) throws Exception {
    Object Result;
    if(Counter == 0)
      NotEmpty.Wait( );
    Result = Buffer[Front]; Front = (Front + 1) % Size; Counter--;
    if(Counter == Size - 1)
      NotFull.NonblockingSignal( );
```

```
    return Result;
  }

}
```

Two points should be noted in this example. First, all classes in the EMonitor class libraries reside in the `EMonitor.lang` package, which last therefore must be imported before defining the `BoundedBuffer` class. Second, for each condition queue belonging to an EMonitored class, the instance of the EMonitored class should be passed to the constructor of the condition queue (for creating the association with the EMonitor lock that is embedded in the instance of the EMonitored class).

The remaining item of syntax to be introduced relates to inter-monitor nested calls. To issue an open call, we create a new keyword `OpenCall`. The syntax of invoking an open call from the `Source()` method (which belongs to an `EMonitored` class) to the `Dest.f()` method is:

```
void Source() {
    ...
  OpenCall Dest.f( );
    ...
}
```

If no prefix is specified for an inter-monitor nested call, the original closed call semantic is employed.

The implementation of the EMonitor

We use a preprocessor to translate the EMonitor syntax into several Java codes that invoke the EMonitor class libraries. The class libraries contain one Java interface, the `EMonitoredInterface`, and five public Java classes, which comprise the three public condition queue classes described in the previous subsection, an `EMonitoredThread` class, and an `EMonitor` class. Since both the preprocessor and the class libraries are implemented by pure Java codes, the underlying Java virtual machine needs no modification.

The `EMonitoredThread` class is a subclass of the `java.lang.Thread` class and any thread using the EMonitor has to be an instance of this class or its subclass. Otherwise, when using any other public class in the EMonitor class libraries, the `NotEMonitoredThreadException` will be thrown. The `EMonitoredThread` class contains a lock-count stack to store the lock-nested count of unreturned open calls. In addition, since the `Thread.resume()` and `Thread.suspend()` of Java are deadlock-prone [4], the `EMonitoredThread` class uses its own `Resume()` and `Suspend()` methods in place of the original ones. The source codes of the `EMonitoredThread` class are shown in Appendix A.

The `EMonitor` class implements a nested lock—the EMonitor lock, and its source codes are shown in Appendix B. Unlike the Java monitor lock, an EMonitor lock contains four FIFO queues for storing pending threads: the entry queue, the waiting queue, the signaler queue, and the return queue. The first three of these has been described in the previous section. The last, the return queue, holds the pending threads that have just returned from open calls. The relative priorities of these queues are: $Wp > Sp > Rp > Ep$ (Rp denotes the return queue priority). To prevent the breakup of the

post condition of the wait method, we give the highest priority to the pending threads in the waiting queue. For the discussion that follows below, we first show the translated codes of the EMonitored `BoundedBuffer` class, where the underscored codes are inserted by the preprocessor.

```
import EMonitor.lang.*;

public class BoundedBuffer implements EMonitoredInterface {

  protected EMonitor EMonitorInstance = new EMonitor();
  public EMonitor GetEMonitor() {
    return EMonitorInstance;
  }

  private int Front = 0, Rear = 0, Counter = 0;
  private static final int Size = 10;
  private Object Buffer[ ] = new Object[Size];
  private FIFOCondition NotEmpty, NotFull;

  public BoundedBuffer() throws Exception {
    NotEmpty = new FIFOCondition(this);
    NotFull = new FIFOCondition(this);
  }

  public void Put(Object Obj) throws Exception {
    EMonitorInstance.Enter();
    try
    if(Counter == Size)
      NotFull.Wait();
    Buffer[Rear] = Obj; Rear = (Rear + 1) % Size; Counter++;
    if(Counter == 1)
      NotEmpty.NonblockingSignal();
    } finally { EMonitorInstance.Leave(); }
  }

  public Object Get() throws Exception {
    EMonitorInstance.Enter();
    try {
    Object Result;
    if(Counter == 0)
      NotEmpty.Wait();
    Result = Buffer[Front]; Front = (Front + 1) % Size; Counter--;
    if(Counter == Size - 1)
      NotFull.NonblockingSignal();
    return Result;
```



```
    } finally { EMonitorInstance.Leave(); }
  }

}
```

For each instance of an EMonitored class, an EMonitor lock, `EMonitorInstance`, is created and embedded in the instance. In addition, to deal with inheritance, an interface, `EMonitoredInterface`, is implemented. If an EMonitored class inherited a predefined and compiled parent class, the preprocessor will determine whether the `EMonitoredInterface` has been implemented in the parent class. If the `EMonitoredInterface` is found in the parent class, the preprocessor will not create the EMonitor lock and implement the `EMonitoredInterface` for the child EMonitored class.

The EMonitor class offers four public methods: `Enter()`, `Leave()`, `IssueOpenCall()`, and `OpenCallReturn()`. `Enter()` is invoked before the first statement of an EMonitored block, or before the first statement of a method in an EMonitored class. This method acquires the EMonitor lock, and increases the lock's nested count by one. `Leave()` is invoked when leaving an EMonitored block, or when leaving a method in an EMonitored class. This is achieved by the Java exception handling mechanism `try{ } finally{ }`. `Leave()` reduces the lock's nested count by one, which if resulting in zero releases the EMonitor lock. Since the `NotEMonitoredThreadException` may be thrown by any of the above four methods, for each method of an EMonitored class (except the constructor), the preprocessor determines whether its throw clause covers the exception. If not, the preprocessor appends the exception to the throw clause.

`IssueOpenCall()` and `OpenCallReturn()` are methods for handling open calls. The translated codes of the previous `Source()` method are

```
void Source() throws NotEMonitoredThreadException {
  EMonitorInstance.Enter();
  try {
    ...
  EMonitorInstance.IssueOpenCall();
  Dest.f( );
  EMonitorInstance.OpenCallReturn( );
    ...
  } finally { EMonitorInstance.Leave( ); }
}
```

`IssueOpenCall()` first pushes the nested count of the EMonitor lock into the lock-count stack of the current thread, and then entirely releases the EMonitor lock. `OpenCallReturn()` reacquires the EMonitor lock after an issued open call returns. The EMonitor's lock-nested count is also restored to the value popped from the top of the lock-count stack.

As previously explained, the responsibility of the `Condition` class is to maintain the association between a condition queue and an EMonitor lock. The field `AssociateEMonitor` stores the associated EMonitor lock of a condition queue. Constructing a condition queue that will be accessed in an EMonitored block is simple. The association between the condition queue and the EMonitor lock of the EMonitored block can be set up by passing the object reference of the EMonitor lock to the constructor of the condition queue.


```
package EMonitor.lang;

abstract class Condition {

  protected EMonitor AssociatedEMonitor;

  public Condition(Object object) throws IllegalArgumentException{
    if(object instanceof EMonitor)
      AssociatedEMonitor = (EMonitor) object;
    else if(object instanceof EMonitoredInterface)
      AssociatedEMonitor = ((EMonitoredInterface) object).GetEMonitor();
    else
      throw new IllegalArgumentException();
  }

}
```

To create a condition queue that belongs to an instance of an EMonitored class, the object reference of the instance should be passed to the constructor of the condition queue. The constructor obtains the associated EMonitor lock through the `GetEMonitor()` method of the `EMonitoredInterface`.

In fact, the condition queue classes and the `EMonitor` class are tightly coupled. The `Wait()`, `BlockingSignal()`, `NonblockingSignal()`, and `NonblockingSignalAll()` methods of any condition queue class can directly access the internal data structure (lock ownership, lock counter, waiting queue, signaler queue) of the `EMonitor` class. We prefer efficiency of implementation over data encapsulation. Further detail regarding implementation can be had by referring to the source codes of the `FIFOCondition` class in Appendix C.

Finally in this subsection, we explain briefly why we excluded the immediate-return signal semantic, which can provide two kinds of methods. The first, the *signal-and-exit* method, reduces the overhead introduced by a thread that calls a signal method and then leaves the monitor immediately. However, since we use the Java exception handling mechanism to catch the leaving the monitor event, the `EMonitor.Leave()` method is always invoked at that point. Thus, the signal-and-exit method cannot be implemented as a shortcut that bypasses the `EMonitor.Leave()` and becomes useless. The second, is the *signal-and-wait* method. Implementing it, however, is troublesome, because the EMonitor has more than one kind of condition queue. For any combination of two different kinds of condition queues, the corresponding signal-and-wait method must be offered. In addition, once a new kind of condition queue is introduced, each existing condition queue must be modified for the purpose of adding a new signal-and-wait method.

PERFORMANCE

In this section, we compare the performance of our EMonitor with the Java version. All the experimental results were gathered in a 266 MHz, dual-Pentium II machine. The operating system is Windows NT 4.0, and the version of JDK is 1.2.

Table I. The overhead when no contention presents.

	Synchronized/ EMonitored method	Nested Synchronized/ EMonitored method	Synchronized/ EMonitored block	Open call
Java Monitor	868 ns	171.4 ns	831.3 ns	N/A
EMonitor	2600.2 ns	2513.5 ns	2545.9 ns	3260.1 ns

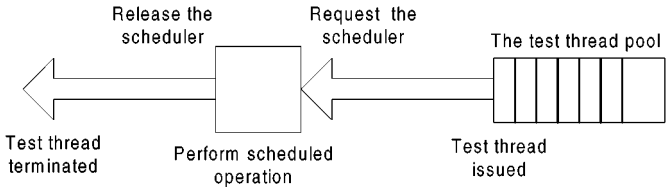
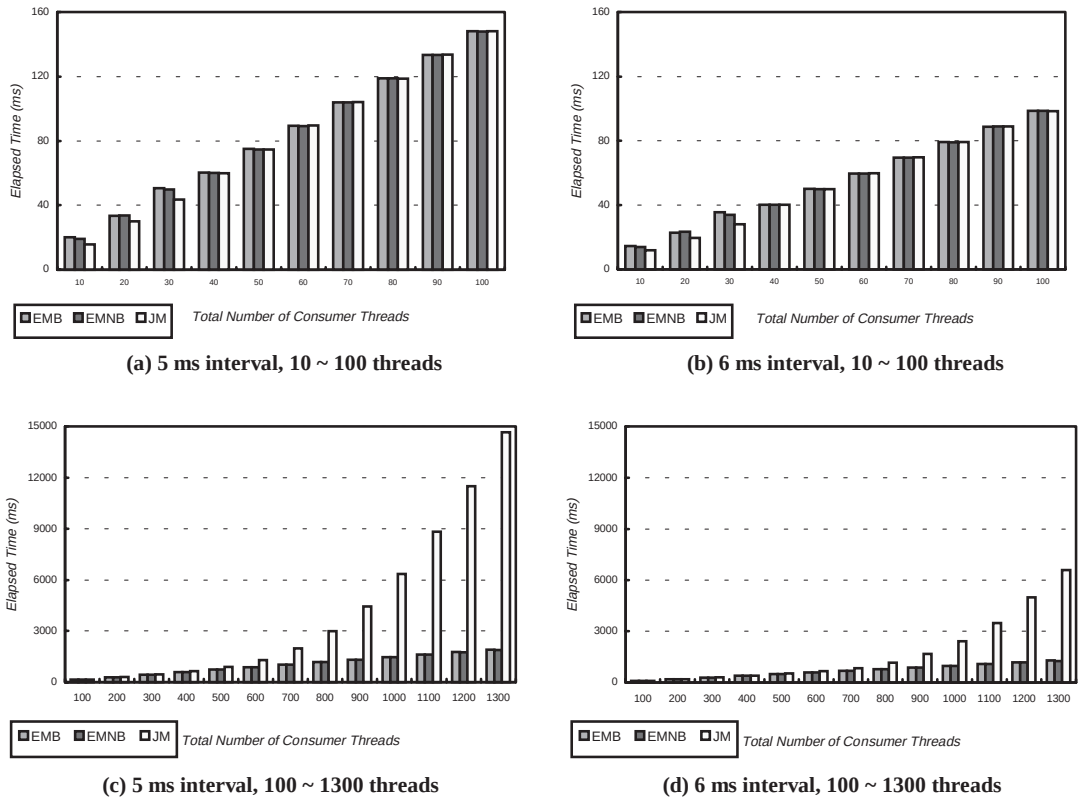


Figure 2. The experiment for testing scheduling programs.

We first compare the performance in the situation where no contention is present. We measure the overheads of a null synchronized method, a null EMonitored method, a null synchronized block, a null EMonitored block, and a null open call. The results are shown in Table I. Suppose that the overhead of acquiring a Java monitor lock and then releasing approximates to the overhead of a null synchronized block (as is the case for the EMonitor). In our experiments, the overhead of the Java monitor lock plus the overhead of `Thread.currentThread()` ($(831.3 \times 2) + 318.5 = 1981.1$ ns) is the dominant part of the overhead of the EMonitor lock. In addition, the open call overhead clearly exceeds other EMonitor overheads. This is due to a null open call having to invoke an extra `Thread.currentThread()`.

To compare performance in a contention situation, we conducted two different experiments. In the first, the aforementioned `BoundedBuffer` classes served as the test program. A producer thread and multiple consumer threads share a bounded buffer. All the consumer threads are created at once. To control the contention level, we varied the total number of consumer threads (from 10 to 100, and from 100 to 1300. The upper bound of 1300 was near the system limit of our test platform.) For every 8 ms, the producer thread creates a new instance of the `Object` class, and invokes the `BoundedBuffer.Put()` method to store the new instance in the shared buffer. The consumer threads are issued at a fixed interval of either 5 or 6 ms. Once issued, a consumer thread invokes the `BoundedBuffer.Get()` method to retrieve an object instance from the shared buffer. We measured the time elapsing between when the first consumer thread is issued and when all consumer threads obtain the desired object instances.

The second experiment concerned scheduling programs (see Figure 2). All the test threads were created in advance, and gathered in a test thread pool. From the starting point of the experiment, the



(a) 5 ms interval, 10 ~ 100 threads

(b) 6 ms interval, 10 ~ 100 threads

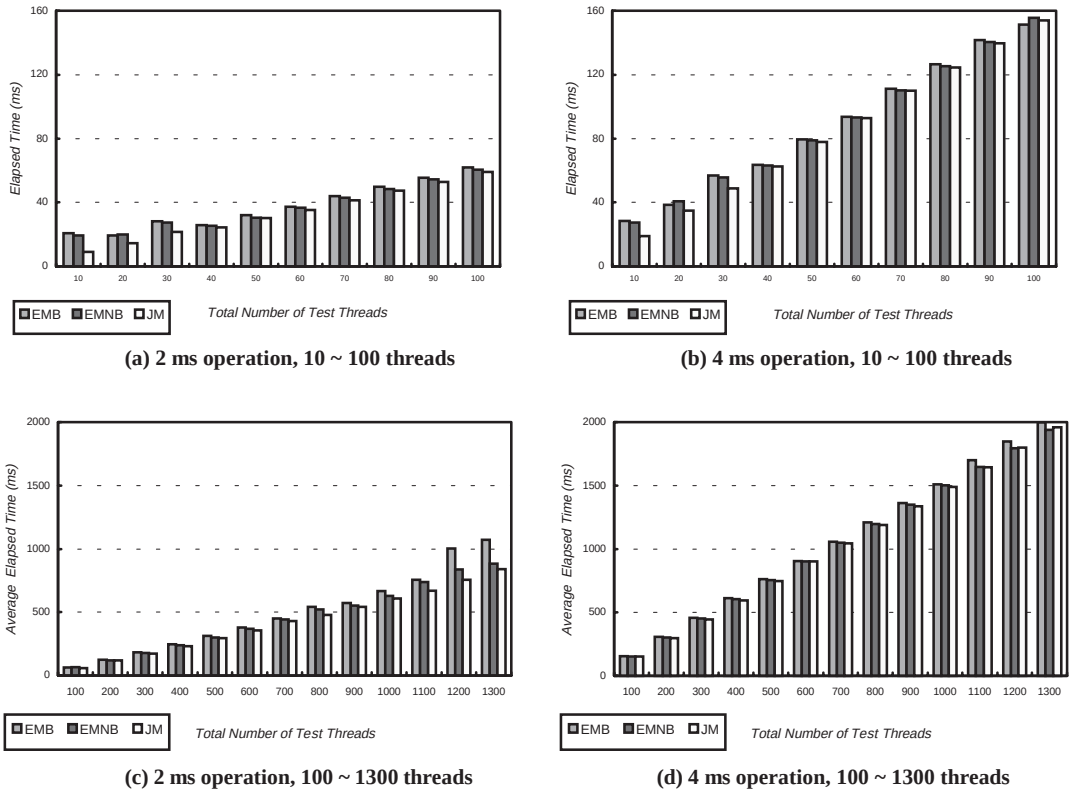
(c) 5 ms interval, 100 ~ 1300 threads

(d) 6 ms interval, 100 ~ 1300 threads

Figure 3. The experiment results of the bounded buffer.

threads were issued one by one from the pool. The interval separating the issue of the threads was set at 1 ms. On issue, each thread sent a request to the scheduler. Once permission from the scheduler was received, the thread took either 2 or 4 ms to perform a scheduled operation, before releasing the scheduler. For each thread, the time elapsing between issuing the request to the scheduler and releasing the scheduler was measured. Then, the average elapsed time was calculated.

Three test scheduling programs were used. The first is the above-mentioned `FIFOSemaphore` class, which represents the type of problem simple enough to be straightforwardly dealt with by the Java monitor. The program is basically the same for both the EMonitor and the Java-version, except that the count of pending threads need not be explicitly maintained in the EMonitor-version. The second is an elevator disk scheduler [2], which represents the type of static scheduling problem that can be handled by the EMonitor's prioritized condition queues. The third is an SSF (shortest-scan-first) disk scheduler [6], which represents the type of dynamic scheduling problem suitable for the EMonitor's customizable condition queues. In this paper, we only show the EMonitor implementations



(a) 2 ms operation, 10 ~ 100 threads

(b) 4 ms operation, 10 ~ 100 threads

(c) 2 ms operation, 100 ~ 1300 threads

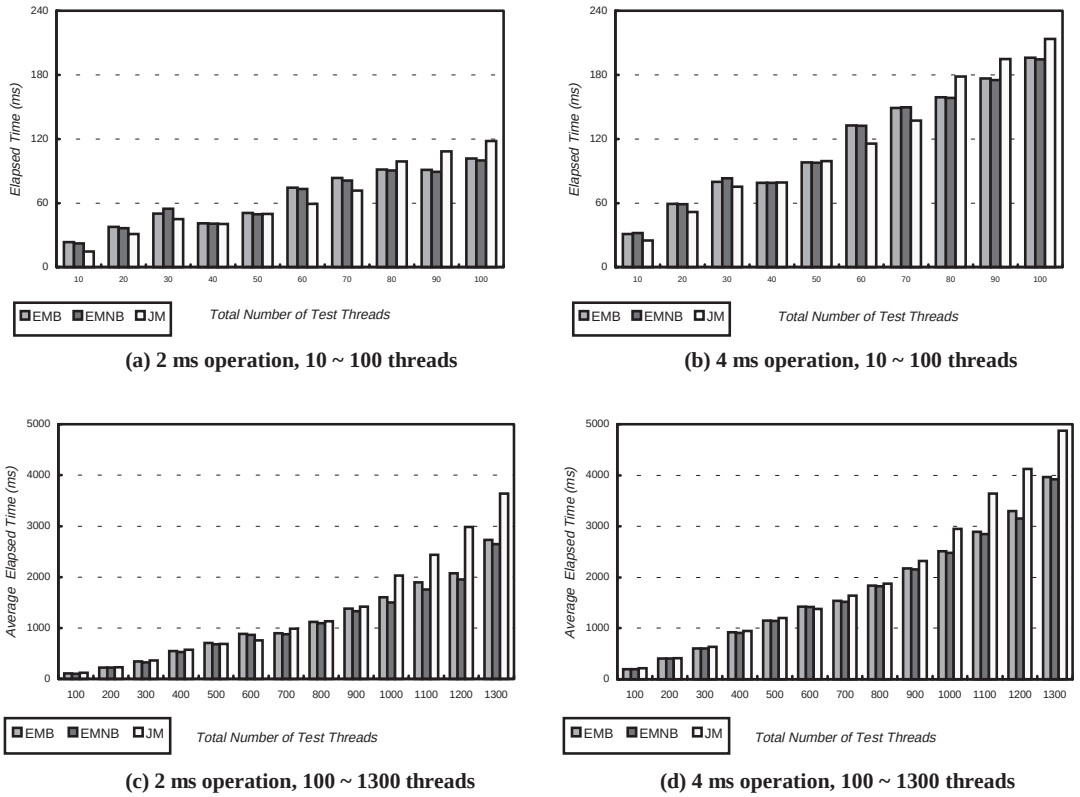
(d) 4 ms operation, 100 ~ 1300 threads

Figure 4. The experiment results of the FIFO semaphore.

(non-blocking signal) of the two disk schedulers in Appendix D and Appendix E, respectively. In addition, one point needs to be mentioned about the Java-version disk schedulers. Since the monitor lock of Java cannot be acquired or released explicitly, it cannot be used directly as the monitor lock of the Java-version disk schedulers [4,15]. Here we use another mutually exclusive lock implemented by Java (the `BusyFlag` class [4], a no-priority lock) in place of the Java monitor.

The experimental results of the bounded buffer are shown in Figure 3. In addition, the results of the three scheduling programs are shown in Figures 4, 5 and 6, respectively. In these figures, the term EMB denotes *EMonitor—blocking signal*, the term EMNB *EMonitor—non-blocking signal* and the term JM *Java monitor*.

First, we compare the performance of the Java monitor (no-priority, non-blocking signal) with that of the EMonitor (except that here we temporarily consider the non-blocking signal only). Basically, the Java version outperforms the EMonitor where thread contention is very low. However, with the exception of FIFO semaphore, as contention increases the EMonitor correspondingly outperforms the



(a) 2 ms operation, 10 ~ 100 threads

(b) 4 ms operation, 10 ~ 100 threads

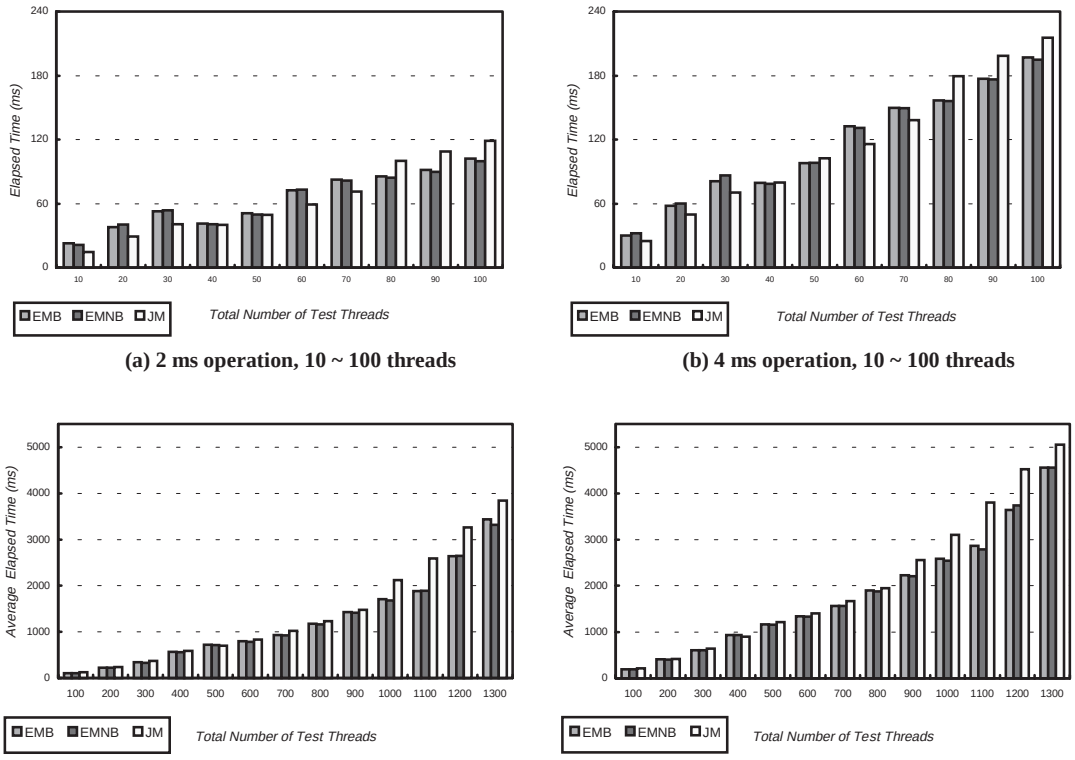
(c) 2 ms operation, 100 ~ 1300 threads

(d) 4 ms operation, 100 ~ 1300 threads

Figure 5. The experiment results of the elevator disk scheduler.

Java version. The turning point in performance is application-dependent. In the bounded buffer, it occurs at around 40 threads. In contrast, the exact turning points with the other two disk schedulers are more difficult to pinpoint. The EMonitor outdid the Java monitor at more than 80 threads. With the FIFO semaphore, although the Java monitor consistently outdid the EMonitor, we found a turning point (around 40 threads) above which the difference between them greatly narrowed. In Table II, we summarize the difference in performance of the two monitors in terms of the $(\text{elapsed time}_{JM} - \text{elapsed time}_{EMNB})/\text{elapsed time}_{EMNB}$.

The bounded buffer program is used to exploit the effect of the single-condition-queue restriction of the Java monitor. It can be observed that at around 700 plus consumer threads the Java monitor's average elapsed time increases dramatically. This also means that the Java monitor's single-condition-queue restriction will have significant impact on performance in situations of medium or high contention.



(a) 2 ms operation, 10 ~ 100 threads

(b) 4 ms operation, 10 ~ 100 threads

(c) 2 ms operation, 100 ~ 1300 threads

(d) 4 ms operation, 100 ~ 1300 threads

Figure 6. The experiment results of the SSF disk scheduler.

Table II. The performance differences between the Java monitor and the EMonitor.

	Bounded buffer		FIFO semaphore		Elevator disk scheduler		SSF disk scheduler	
	5 ms Inv.	6 ms Inv.	2 ms Op.	4 ms Op.	2 ms Op.	4 ms Op.	2 ms Op.	4 ms Op.
Before the turning point	−14%	−16.1%	−34.4%	−19.4%	−13.9%	−9%	−16.7%	−10.2%
After the turning point	183.9%	93.3%	−4.6%	−0.6%	16.7%	11.2%	13.8%	11.1%

In theory, the performance of a no-priority monitor fails to match that of a priority monitor. However, in the case of FIFO semaphore we find the Java monitor slightly outperforming the EMonitor, a phenomenon we attribute to implementation. Since the EMonitor is implemented by the Java monitor, its implementation overhead outweighs the performance gain of being a priority monitor. Contrast this with the other two scheduling programs, whose monitor lock, the `BusyFlag` class, is also implemented by the Java monitor, and observe that here the EMonitor outperforms the Java version. However, for these scheduling programs, we consider that the difference in performance of both monitors is not obvious except where thread contention increases considerably (around 1000 plus threads) .

In summary, where thread contention is either absent or very low and performance is the consideration, we recommend the Java monitor for the task of constructing simple concurrent objects that can be straightforwardly implemented. However, where thread contention exceeds the turning point, we recommend the EMonitor for the same task because it both avoids the inconvenience of a no-priority monitor and provides a performance almost equal to that of the Java monitor. We further recommend the EMonitor for concurrent objects that require multiple or more elaborate condition queues, tasks which the Java monitor handles only with great difficulty. We found the EMonitor outperforming Java in dealing with these complex concurrent objects.

Finally in this section we compare the non-blocking and blocking signals of the EMonitor. In fact, with the exception of the FIFO semaphore (with the 2 ms operation), there is only a light difference in performance between the two signal semantics (within 4% in most cases). Furthermore, with the above exception, the performance of the blocking signal is demonstrably poor only when the contention level nears the system limit (above 1200 test threads). Consequently, since the blocking signal is both easier to use and offers acceptable performance, we recommend employing it as necessary.

RELATED WORK

Several pieces of research work about thread synchronization mechanism on Java have been proposed. Basically, they may be categorized into following three paradigms: monitor, active object [16], and CSP [17]. In fact, among these, only the first is closely related to the EMonitor. Since the focus of this paper is on offering an improved replacement for the Java monitor, detailed comparison between the EMonitor and the other two paradigms is beyond our present scope. Hence, they are described only briefly. Interested readers can find more information and details in the References.

Monitor

Two pieces of research have appeared in which a design pattern [15] and a package [4] are proposed for providing more than one condition queue inside a Java object. Since how to construct a monitor is well known, the implementation of the above two proposals is somewhat similar to the EMonitor. However, apart from dealing with the restriction of a single condition queue, the above pieces of research do not seriously explore the other drawbacks of the Java monitor. For example, the `BusyFlag` class [4] proposed in the above package is merely a poor no-priority monitor lock. In contrast, our paper presents a comprehensive analysis of the Java monitor and demonstrates that the EMonitor is better than the proposals contained in the above pieces of research.

Active object

The following pieces of research belong to this paradigm: CJava [18], CORRELATE [19], and an active object design pattern [20]. An active object offers public methods that can be invoked by other objects, in the same way as a monitor does. However, unlike a monitor, an active object employs declarative synchronization mechanisms (usually method guards), by which the synchronization policy of each public method can be specified at a high level and in an implementation-independent way. The synchronization between method invocations is automatically managed by the run-time system of the active object. Hence, constructing concurrent objects by active objects is easier. Besides, since well-designed active objects can fully separate the sequential part from the synchronization part program [8], they are also less sensitive to the so-called inheritance anomaly [21]. However, the performance of active objects is poor, especially for pure active implementation [7]. Unfortunately, all the above Java-based active objects fall into this category. Even when no contention occurs, invoking a method on a Java-based active object inevitably generates context switches. In addition, since these Java-based active objects use method guards, they also incur the fast-growing guard evaluation overheads. Therefore, we consider these Java-based active objects suitable only for coarse-grain concurrent systems.

CSP

Two other pieces of research belong to this paradigm—JCSP [22] and CTJ [23]. Both of them are Java class libraries that provide the primitives of the CSP algebra with several extensions. A CSP process can be regarded as an object with its own thread of control. In addition, a CSP process has no externally invocable methods, and its own data and algorithm are private. A process can communicate to another process only by reading and writing data through channels. Basically, channels are one-to-one and zero-buffered. However, buffered, multiple-reader and multiple-writer channels may also be offered. A higher-level process can be built through composition. Since the composition of processes can be nested, to construct an elaborate and powerful process is possible. The chief advantage of a CSP-based program is that its behavior can be formally verified. This enables the reasoning about race-hazards, deadlock, starvation, and livelock. Furthermore, the behavior of the program can also be guaranteed. In fact, the loosely coupled nature of a CSP-based program may generate more overheads. However, when developing large-scale, reliable systems, performance is not a primary focus. We consider the CSP-based libraries suitable for these circumstances.

CONCLUSION

We began this paper with a detailed analysis of four known drawbacks of the Java monitor. We then proposed our own monitor-based replacement—the EMonitor. However, since only Java was used to implement the EMonitor, no modification to the Java virtual machine was required. A comparative analysis of the performance of the two monitors was made. We concluded that the Java monitor is suitable only for constructing simple concurrent objects in situations where thread contention is absent or very low. That apart, we recommend replacing the Java monitor with the EMonitor for developing concurrent Java objects.

APPENDIX A. EMonitoredThread CLASS

```
package EMonitor.lang;

public class EMonitoredThread extends java.lang.Thread {

  private IntegerStack LockCountStack = new IntegerStack();
  private Object SyncObject = new Object();
  private boolean IsSuspended = false;
  private boolean IsResumed = false;

  /* The IntegerStack class is a variable-size integer stack without internal
      synchronization. */

  public void StackPush(int LockNestedCount) {
    LockCountStack.Push(LockNestedCount);
  }

    public int StackPop() {
    return LockCountStack.Pop();
  }

  public static EMonitoredThread CurrentEMonitoredThread() throws
                                                    NotEMonitoredThreadException {
    try {
      return (EMonitoredThread) Thread.currentThread();
    }
    catch(ClassCastException E) {
      throw new NotEMonitoredThreadException();
    }
  }

  public void Resume() {
    synchronized(SyncObject) {
      if(IsSuspended) {
        IsSuspended = false;
        try { SyncObject.notify(); } catch (Exception E) { E.printStackTrace(); }
      }
      else {
        IsResumed = true;
        return;
      }
    }
  }

  public void Suspend() {
    synchronized(SyncObject) {
      if(!IsResumed) {
        IsSuspended = true;
        try { SyncObject.wait(); } catch (Exception E) { E.printStackTrace(); }
```

```
      }
      else {
        IsResumed = false;
        return;
      }
    }
  }

}
```

APPENDIX B. EMonitor CLASS

```
package EMonitor.lang;

public class EMonitor {

  protected int LockNestedCount = 0;
  protected EMonitoredThread LockOwner = null;
  private FIFOThreadQueue EntryQueue = new FIFOThreadQueue();
  private FIFOThreadQueue ReturnQueue = new FIFOThreadQueue();
  protected FIFOThreadQueue WaitingQueue = new FIFOThreadQueue();
  protected FIFOThreadQueue SignallerQueue  = new FIFOThreadQueue();

  /* The FIFOThreadQueue class is an unsynchronized, variable-size FIFO queue
      that stores the instances of the EMonitoredThread class. */

  protected EMonitoredThread GetResumedThread() {

    EMonitoredThread ResumedThread = WaitingQueue.Delete();

    if(ResumedThread == null) {
      ResumedThread = SignallerQueue.Delete();
      if(ResumedThread == null) {
        ResumedThread = ReturnQueue.Delete();
        if(ResumedThread == null)
          ResumedThread = EntryQueue.Delete();
      }
    }
    return ResumedThread;
  }

  public void Enter() throws NotEMonitoredThreadException {

    boolean IsSuspend;
    EMonitoredThread CurrentThread = EMonitoredThread.CurrentEMonitoredThread();

    synchronized(this) {
      if(LockOwner == null) {
        LockOwner = CurrentThread;
```



```
      LockNestedCount = 1;
      IsSuspend = false;
    }
    else if(LockOwner == CurrentThread) {
      LockNestedCount++;
      IsSuspend = false;
    }
    else {
      EntryQueue.Insert(CurrentThread);
      CurrentThread.StackPush(1);
      IsSuspend = true;
    }
  }

  if(IsSuspend)
    CurrentThread.Suspend();
  else
    return;
}

public synchronized void Leave() throws NotEMonitoredThreadException {

  EMonitoredThread ResumedThread;

  LockNestedCount--;
  if(LockNestedCount == 0) {
    ResumedThread = GetResumedThread();
    if(ResumedThread != null) {
      LockOwner = ResumedThread;
      LockNestedCount = ResumedThread.StackPop();
      ResumedThread.Resume();
    }
    else
      LockOwner = null;
  }
  return;
}

public synchronized void IssueOpenCall() throws NotEMonitoredThreadException {

  EMonitoredThread ResumedThread;

  (EMonitoredThread.CurrentEMonitoredThread()).StackPush(LockNestedCount);
  ResumedThread = GetResumedThread();
  if(ResumedThread != null) {
    LockOwner = ResumedThread;
    LockNestedCount = ResumedThread.StackPop();
    ResumedThread.Resume();
  }
  else {
    LockOwner = null;
```

```
      LockNestedCount = 0;
    }
    return;
  }

  public void OpenCallReturn() throws NotEMonitoredThreadException {

    boolean IsSuspend;
    EMonitoredThread CurrentThread = EMonitoredThread.CurrentEMonitoredThread();

    synchronized(this) {
      if(LockOwner == null) {
        LockOwner = CurrentThread;
        LockNestedCount = CurrentThread.StackPop();
        IsSuspend = false;
      }
      else {
        ReturnQueue.Insert(CurrentThread);
        IsSuspend = true;
      }
    }

    if(IsSuspend)
      CurrentThread.Suspend();
    else
      return;
  }

}
```

APPENDIX C. FIFOCondition CLASS

```
package EMonitor.lang;

public class FIFOCondition extends Condition {

  private FIFOThreadQueue ConditionQueue = new FIFOThreadQueue();

  // A protected field AssociatedEMonitor is inherited from the Condition class.

  public FIFOCondition(Object object) throws IllegalArgumentException {
    super(object);
  }

  public int Length() {
    return  ConditionQueue.Length();
  }

  public void Wait() throws IllegalEMonitorStateException,
                                                NotEMonitoredThreadException {
```



```
    EMonitoredThread CurrentThread = EMonitoredThread.CurrentEMonitoredThread();
    EMonitoredThread ResumedThread;

    synchronized(AssociatedEMonitor) {

      if(AssociatedEMonitor.LockOwner != CurrentThread)
        throw new IllegalEMonitorStateException();

      CurrentThread.StackPush(AssociatedEMonitor.LockNestedCount);
      ConditionQueue.Insert(CurrentThread);

      ResumedThread = AssociatedEMonitor.GetResumedThread();
      if(ResumedThread != null) {
        AssociatedEMonitor.LockOwner = ResumedThread;
        AssociatedEMonitor.LockNestedCount = ResumedThread.StackPop();
        ResumedThread.Resume();
      }
      else {
        AssociatedEMonitor.LockOwner = null;
        AssociatedEMonitor.LockNestedCount = 0;
      }
    }
    CurrentThread.Suspend();
  }

  public void BlockingSignal() throws IllegalEMonitorStateException,
                                                  NotEMonitoredThreadException {

    EMonitoredThread CurrentThread = EMonitoredThread.CurrentEMonitoredThread();
    EMonitoredThread SignalledThread;

    synchronized(AssociatedEMonitor) {

      if(AssociatedEMonitor.LockOwner != CurrentThread)
        throw new IllegalEMonitorStateException();

      SignalledThread = ConditionQueue.Delete();
      if(SignalledThread != null) {
        CurrentThread.StackPush(AssociatedEMonitor.LockNestedCount);
        (AssociatedEMonitor.SignallerQueue).Insert(CurrentThread);
        AssociatedEMonitor.LockOwner = SignalledThread;
        AssociatedEMonitor.LockNestedCount = SignalledThread.StackPop();
        SignalledThread.Resume();
      }
    }
    if(SignalledThread != null)
      CurrentThread.Suspend();
  }

  public void NonblockingSignal() throws IllegalEMonitorStateException,
                                                  NotEMonitoredThreadException {
```



```
    EMonitoredThread CurrentThread = EMonitoredThread.CurrentEMonitoredThread();
    EMonitoredThread SignalledThread;

    synchronized(AssociatedEMonitor) {

      if(AssociatedEMonitor.LockOwner != CurrentThread)
        throw new IllegalEMonitorStateException();

      SignalledThread = ConditionQueue.Delete();
      if(SignalledThread != null)
        (AssociatedEMonitor.WaitingQueue).Insert(SignalledThread);
    }
  }

  public void NonblockingSignalAll() throws IllegalEMonitorStateException,
                                                    NotEMonitoredThreadException {

    EMonitoredThread CurrentThread = EMonitoredThread.CurrentEMonitoredThread();
    EMonitoredThread SignalledThread;

    synchronized(AssociatedEMonitor) {

      if(AssociatedEMonitor.LockOwner != CurrentThread)
        throw new IllegalEMonitorStateException();

      SignalledThread = ConditionQueue.Delete();
      while(SignalledThread != null) {
        (AssociatedEMonitor.WaitingQueue).Insert(SignalledThread);
        SignalledThread = ConditionQueue.Delete();
      }
    }
  }

}
```

APPENDIX D. ELEVATOR DISK SCHEDULER

```
import EMonitor.lang.*;

public EMonitored class DiskHeadScheduler {

  private boolean IsBusy = false;
  private int HeadPosition = 0;
  private static final boolean UpDirection = false;
  private static final boolean DownDirection = true;
  private boolean Direction = UpDirection;

  private PrioritizedCondition UpSweep;
  private PrioritizedCondition DownSweep;
```



```
  /* The pending threads in UpSweep are sorted ascendantly.
      In contrast, the pending threads in DownSweep are sorted descendently. */

  public DiskHeadScheduler() throws Exception {
    UpSweep = new PrioritizedCondition(this, UpDirection);
    DownSweep = new PrioritizedCondition(this, DownDirection);
  }

  public void Request(int TargetCylinder) throws Exception {
    if(IsBusy) {
      if((HeadPosition <= TargetCylinder) && (Direction == UpDirection))
        UpSweep.Wait(TargetCylinder);
      else
        DownSweep.Wait(TargetCylinder);
    }
    IsBusy = true;
    HeadPosition = TargetCylinder;
  }

  public void Release() throws Exception {
    IsBusy = false;
    if(Direction == UpDirection) {
      if(!UpSweep.IsEmpty())
        UpSweep.NonblockingSignal();
      else {
        Direction = DownDirection;
        DownSweep.NonblockingSignal();
      }
    }
    else { // Direction == DownDirection
      if(!DownSweep.IsEmpty())
        DownSweep.NonblockingSignal();
      else {
        Direction = UpDirection;
        UpSweep.NonblockingSignal();
      }
    }
  }

}
```

APPENDIX E. SSF DISK SCHEDULER

```
import EMonitor.lang.*;

class CylinderThreadNode extends ThreadNode {
  protected int Cylinder;
}
```

```
public EMonitored class DiskHeadScheduler {

  private boolean IsBusy = false;
  private int HeadPosition = 0;
  private CustomizableCondition PendingRequest;

  public DiskHeadScheduler() throws Exception {
    PendingRequest = new CustomizableCondition(this);
  }

  public void Request(int TargetCylinder) throws Exception {

    CylinderThreadNode NewNode;
    ThreadNode TailNode;

    if(IsBusy) {
      NewNode = new CylinderThreadNode();
      NewNode.Cylinder = TargetCylinder;
      TailNode = PendingRequest.SeekToTail();
      PendingRequest.InsertAfter(TailNode, NewNode);
      PendingRequest.Wait();
    }
    IsBusy = true;
    HeadPosition = TargetCylinder;
  }

  public void Release() throws Exception {

    CylinderThreadNode CurrentNode;
    CylinderThreadNode SignalledNode;

    IsBusy = false;
    if(!PendingRequest.IsEmpty()) {
      SignalledNode = (CylinderThreadNode) PendingRequest.SeekToHead();
      CurrentNode = (CylinderThreadNode) PendingRequest.SeekAhead();

      /* If the cursor has already pointed to the last ThreadNode instance
          of a customizable queue, the SeekAhead() will return null. */

      while(CurrentNode != null) {
        if(Math.abs(CurrentNode.Cylinder - HeadPosition) <
            Math.abs(SignalledNode.Cylinder - HeadPosition))
           SignalledNode = CurrentNode;
        CurrentNode = (CylinderThreadNode) PendingRequest.SeekAhead();
      }
      PendingRequest.NonblockingSignal(SignalledNode);
    }
  }

}
```


ACKNOWLEDGEMENTS

The authors wish to thank the two anonymous reviewers for their comments on an earlier version of this paper. In addition, we gratefully acknowledge the financial support of both the National Science Council (grant NSC88-2213-E-009-087 and grant NSC89-2213-E-009-069) and the ROC Economic Bureau (industrial research program 89-EC-2-A-17-0285-006). We are also much obliged to our colleague, Pin-Huang Hsin, who helped solve problems during the coding of the EMonitor's preprocessor.

REFERENCES

1. Gosling J, Joy B, Steele G. *The Java Language Specification*. Addison-Wesley: Reading, MA, 1996.
2. Hoare C. Monitor: An operating system constructing concept. *CACM* 1974; **17**(10):549–557.
3. Buhr P, Fortier M, Coffin M. Monitor classification. *ACM Computing Surveys* 1995; **27**(1):63–107.
4. Oaks S, Wong H. *Java Threads* (2nd edn). O'Reilly & Associates: Sebastopol, CA, 1999.
5. Andrews G. *Concurrent Programming—Principles and Practice*. The Benjamin/Cummings: Redwood City, CA, 1991.
6. Gehani N. Capsules: A shared memory access mechanism for concurrent C/C++. *IEEE Transactions on Parallel and Distributed Systems* 1993; **4**(7):795–811.
7. Holmes D, Noble J, Potter J. Towards reusable synchronization for object-oriented languages. *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP '98*, 1998.
8. McHale C. Synchronization in concurrent, object-oriented languages: Expressive power, genericity and inheritance. *PhD Thesis*, 1994, Trinity College, Dublin, Ireland.
9. Olsson R, McNamee C. Experience using the C preprocessor to implement CCR, Monitor, and CSP preprocessors for SR. *Software—Practice and Experience* 1996; **26**(2):125–134.
10. Stubbs S, Carver D, Hoppe A. IPCC++: A concurrent C++ based on a shared-memory model. *Journal of Object-Oriented Programming* 1995; **8**(2):45–50, 66.
11. Yuan S, Hsu Y. Design and implementation of a distributed monitor facility. *Computer Systems Science and Engineering* 1997; **12**(1):43–51.
12. Brosgol B. A comparison of the concurrency features of Ada 95 and Java. *ACM Ada Letters* 1998; **18**(6):175–192.
13. Varela C, Agha G. What after Java? From objects to actors. *Computer Networks and ISDN Systems* 1998; **30**(1–7):573–577.
14. Kotulski L. About the semantic nested monitor calls. *SIGPLAN Notices* 1987; **22**(4):80–82.
15. Lea D. *Concurrent Programming in Java—Design Principles and Patterns* (2nd edn). Addison-Wesley: Reading, MA, 1999.
16. Agha G. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press: Cambridge, MA, 1986.
17. Hoare C. Communicating sequential processes. *CACM* 1978; **21**(8):666–677.
18. Cugola G, Ghezzi C. CJava: Introducing concurrent objects in Java. *Proceedings of International Conference on Object-Oriented Information Systems*, Brisbane, Australia, November 1997; 504–514.
19. Robben B, Matthijs F, Vanhaute B. Experience with CORELLATE. *Proceedings of OOPSLA '97 Workshop on Java-Based Paradigms for Mobile Objects*, 1997.
20. Carroll M. Active objects made easy. *Software—Practice and Experience* 1998; **28**(1):1–21.
21. Matsuoka S, Yonezawa A. Analysis of inheritance anomaly in object-oriented concurrent programming languages. *Research Directions in Concurrent Object-Oriented Programming*. MIT Press: Cambridge, MA, 1993; 107–150.
22. Welch P. *CSP for Java (What, Why, and How Much?)*. JCSP Project Homepage http://www.cs.ukc.ac.ul/projects/ofa/jcsp [November 1999].
23. Hilderink G. *Communicating Threads for Java—Tutorial for the CSP Package Version 0.9, Revision 10*. University of Twente, Netherlands, http://www.rt.el.utwente.nl/javapp/ [1998].

