Efficient Java™Monitors

Blomdell, Anders

2001

Document Version:
Publisher's PDF, also known as Version of record

ISSN 0280-5316
ISRN LUTFD2/TFRT--7593--SE

Efficient Java ™ Monitors

Anders Blomdell

Department of Automatic Control
Lund Institute of Technology
November 2001

Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 Lund Sweden

Document name
INTERNAL REPORT

Date of issue
November 2001

Document Number
ISRN LUTFD2/TFRT--7593--SE

Author(s)
Anders Blomdell

Supervisor

Sponsoring organization

Title and subtitle
Efficient Java ™ Monitors. (Effektiva Java ™ monitorer)

Abstract
In most real world systems, objects vastly outnumbers threads. This paper shows how this characteristic can be used to implement efficient Java monitors in a way that reduces the number of needed monitors to be equal to the number of threads, while fulfilling the Java synchronized semantics.
Some additional benefits related to priority inheritance and hierarchical resource locking will also be briefly explored.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
10

Recipient's notes

Security classification

Efficient Java™ Monitors

Anders Blomdell
Department of Automatic Control
Lund Institute of Technology
Box 118, SE-221 00 Lund, Sweden
anders.blomdell@control.lth.se

Abstract

*In most real world systems, objects vastly outnumbers threads. This paper shows how this characteristic can be used to implement efficient Java™monitors in a way that reduces the number of needed monitors to be equal to the number of threads, while fulfilling the Java™**synchronized** semantics. Some additional benefits related to priority inheritance and hierarchical resource locking will also be briefly explored.*

Keywords Java™, concurrent programming, hierarchical resource locking, monitors, priority inheritance, real-time.

1. Introduction

In many embedded systems the memory is a scarce resource and every saved byte is important, in other the worst-case execution time is the limiting factor. In this paper, the fact that in most real world systems objects vastly outnumbers threads will be the basis for an efficient implementation of Java™monitors that fulfills the requirements of Java™**synchronized** semantics. The solution is probably near optimal when it comes to memory consumption and it has a small runtime penalty for systems where the number of objects locked by a thread at any instant is reasonably low.

Finally some additional benefits with the proposed implementation will be briefly mentioned.

2. Ordinary JVM monitors

According to ”The Java™Virtual Machine Specification” [6], ”There is a lock associated with each object”. This requirement together with the fact that many important methods in the Java™ runtime system are **synchronized**, makes efficient monitor implementation vital. The common way to implement Java monitors, is to associate one monitor with each object, see Figure 1, but there are some simple optimizations to reduce the number of needed locks. As

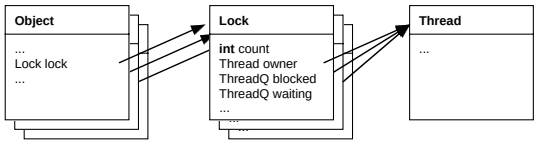

Figure 1 Common Java™ implementation with one monitor per object.

a background to the new monitor implementation, a quick overview of some of these techniques will be given. Most Java™Virtual Machines (JVM) and the associated monitors are not implemented in Java™, but in this paper all code is written in Java™.

2.1 Simple implementation

The obvious implementation of locks is to create a lock for each created object. The only problem with this is that neither the locks nor their constituent parts are allowed to have have locks or unbounded recursion will occur, see Listing 1 . This makes it hard to implement the Java runtime system entirely in Java. To avoid this problem we have to check if the Object about to be created is part of a **Monitor** or not. Apart from that small quirk, implementation of

```
class Object {
  Monitor lock;
  static boolean creatingLock = false;
  public Object() {
    if (! creatingLock) {
      // we have to avoid infinite recursion
      creatingLock = true;
      lock = new Monitor();
      creatingLock = false;
    }
```

Listing 1 Simple monitor.

the **Monitor** itself, and the **JVM** dispatching code is straightforward, see Appendix A. Unfortunately the resulting system fails to fulfill the requirement that each object has an associated monitor (since monitors doesn’t have monitors), but it is a minor

problem, since this fact is invisible for users of the runtime system.

2.2 Lazy creation

If not all objects are expected to be subject to locking, it's an improvement to postpone the monitor creation until the monitor is really needed, see Listing 2. There are two slight disadvantages with

```
case monitorenter: {
  Object o = stack.pop()
  if (o.lock == null) {
    o.lock = new Monitor(o);
  }
  o.lock.enter(currentThread);
```

Listing 2 Lazy creation.

this solution; the **JVM** has check if the monitor exists before trying to lock the object and hence the first locking of an object takes longer time than the subsequent ones, since the monitor has to be created at the first locking. In a real-time system the delayed monitor creation have an adverse affect on the schedulability of the system since monitor creation time has to be taken into account when **synchronized** is used.

2.3 Eager destruction

If there is a fair chance of monitors to be used only a few times, immediate destruction of free monitors might be useful, see Listing 3. If this continuous create/destroy proves too costly, a pool of monitors

```
class Monitor {
  ...
    if (count == 0 &&
        blocked.empty() &&
        waiting.empty()) {
      // Delete lock
      object.lock = null;
```

Listing 3 Eager destruction.

and some bookkeeping of recently freed monitors might be useful. For real-time purposes, this means worse performance since processing power is used to keep memory consumption down, and if a monitor pool is not used, the worst case execution time is probably increased since load on the garbage collector is increased.

3. One monitor per thread

However, we can do better than in Section 2. Since a thread can lock arbitrarily many objects, but an object only can be locked by one thread at a time, we can keep the information of locked objects in one monitor that is associated with the thread instead of one monitor for each locked object. With this approach we reduce the overhead for each locked object from one separate [monitor] object to two equal sized arrays; one containing references to locked objects, and the other the associated counts for the locked objects, see Figure 2 and Appendix C. The size of these arrays can either be variable to accommodate any program, or of a constant size that can be determined by a global analysis provided that no unbounded recursive locking is done.

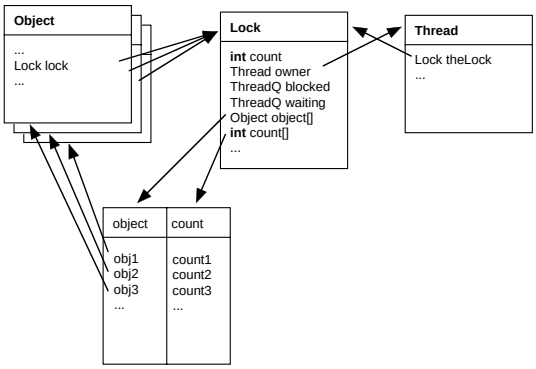

Figure 2 One monitor per thread.

The obvious implementation of blocking is to make the blocked queue a list of object-thread pairs, but as will be shown in Section 5 omitting the object reference will save some memory and gain an additional advantage at the same time. Java™ **wait** and **notify** poses a greater challenge since a wait implies that the object is unlocked, leaving the object without a natural place to keep the wait queue. The solution to this dilemma is to let **wait** keep the reference to the waiting monitor in the object and let threads trying to lock the object check if the monitor is active or waiting. If the monitor was in the waiting state, the queue of waiting objects is moved to the new monitor. When unlocking an object, the lock is set to point at the first monitor waiting for the object and the waiting queue is moved to that monitor. This will effectively mean that the waiting queue is passed between different threads as objects are locked and unlocked. Since each object can have its own waiting queue, and multiple objects could be locked simultaneously by a monitor, another array of the same size as the count and object arrays is needed to hold these queues.

Unless the "eager destruction" scheme is chosen, this gives a space saving if a thread ever locks

four different objects, since the overhead of three arrays in this implementation is less or equal than the overhead of the four monitors needed in the implementations in Section 2. Compared to ”eager destruction” there is still a time saving, and also a space saving if a thread locks more than four objects at the same time.

For threads that has deeply nested locking of the same object, the linear search through the array of locked objects adds to the running time, but first time locking of a previously free object takes constant time. From a scheduling perspective, it is beneficial that the monitor is created at the same time as the thread is created, and not at some later time when the thread locks an object for the first time.

```
...
case monitorenter: {
  currentThread.enter(stack.pop());
} break;
case monitorexit: {
  currentThread.leave(stack.pop());
} break;
...
```

Listing 4 One monitor per thread

Since there is a one to one correspondence between threads and locks, we can further reduce the memory usage by merging the lock with the thread, i.e. place the data and methods in the thread instead of a separate lock object, see Listing 4. Such a merge also has a slight runtime advantage, since there is a reduction of the number of references that need to be followed at runtime.

4. Deadlocks

```
static Resource A, B;
void f() {
  synchronized (A) {
    try { A.wait(100); } catch (Exception e) {}
...
public void run() {
  synchronized (A) {
    synchronized (B) {
      f();
```

Listing 5 Possible deadlock

A common way to avoid deadlocks, is to assign every resource a unique number, and then always claim the resources in ascending order [3]. Unfortunately Java™ programs that **seem** to follow this paradigm may still deadlock, see Listing 5. In this program the intention was to always claim resource A before resource B, but while coding the method f() it was not realized that resources might already have been locked somewhere else, so when f() gets called from the run() method, resource A is released while the thread is still holding resource B. The net effect is that if more than one copy of this code executes at the same time, the program will (somewhat unexpectedly) deadlock on resource B.

In the case of ”ordinary Java™ monitors”, there is not much to do about this problem, but with the ”monitor per thread” approach there are a number of possible approaches that can be used. Each of these have their advantages and drawbacks, which we will come back to after a brief overview of possible deadlock avoidance methods.

1. Throw an exception if the object waited on is not the last acquired one, this is easily checked by ensuring that the object waited on matches the last object in the list of locked objects.
2. Release all objects that were locked after the object waited on were locked, this is easily done by releasing the lock of the object waited for and all objects located after it in the list of locked objects.
3. Detect deadlocks by traversing the blocking chain when blocking on an object, to ensure that the thread holding the object we are waiting for is not blocked waiting for some object that is held by the thread just about to block. If a deadlock is detected, an exception is thrown.

Combinations of (1) and (3) or (2) and (3) are also possible.

Throwing an exception (1) when we try to wait for an object that was not the last one acquired is simple to implement, and if we put the additional requirement that the locking counter should be equal to one, we will also be able to detect potential problems like the one in Listing 6. The disadvantages are that it only prevents deadlocks if objects are locked in the same order everywhere and it will break perfectly legal Java™ code. Together with (3) it is a good way to ensure that deadlocks are always detected and that coding follows a stricter set of programming rules that make sense for critical real-time systems.

Releasing all objects (2) locked after the one waited for, is also easy to implement and it will reduce

```
int attribute;
void g() {
  synchronized(A) {
    try { A.wait(); } catch (Exception e) {}
  }
}
void f() {
  synchronized(A) {
    int tmp = attribute;
    g();
    attribute = tmp + 1;
  }
}
```

Listing 6 Dubious caching

blocking in the running system. The disadvantage is that it breaks Java™ semantics, even though the kind of problems that are exposed by this breaking can just as well occur in normal JVM's, see Listing 6, although not as frequently. If it is combined with (3) it will probably give good performance and still detect all true deadlocks.

Detecting deadlocks (3) does not break any correct Java™ programs, but will throw exceptions when deadlocks occur, regardless of what policy is used for resource locking. A possible drawback is that traversing the locking chain may be time consuming, neither does it give any hints about dangerous constructs that might give problems in the future.

5. Priority inheritance

When unlocking an object, it is natural to assume that only the threads blocked on that object should be released, leading to a blocking queue that is a list of object–thread pairs. But by releasing all threads waiting for any of the held objects (not only those waiting for the unlocked one), we will only need half the amount of memory for the blocking queue and gain an additional implementation advantage, namely priority inheritance. This is so because the threads waiting for the current thread falls in four categories:

1. Lower priority threads waiting for the released lock.
2. Higher priority threads waiting for the released lock.
3. Lower priority threads waiting for some other lock held by the current thread.
4. Higher priority threads waiting for some other lock held by the current thread.

Higher priority threads (2) that are no longer blocked by this thread should start execution immediately and will not affect the current thread any more (unless it gets blocked again). Lower priority threads (2) no longer blocked should eventually run. Releasing any higher priority threads (4) that will immediately start execution just to be blocked again on the same object might seem like a bad idea, but if the current thread's effective priority is reset to its base priority before rescheduling, the higher priority thread will re-raise priority of the current thread before it blocks again, and the net effect is that we have painlessly implemented priority inheritance. Lower priority threads (3) may just as well be released since they should eventually run, and won't affect the current thread unless they become subject to priority inheritance, which will then be propagated to the current thread.

In a uni-processor the worst case behavior is that one higher priority thread is woken up and then immediately blocked again. In a multi-processor every processor may run threads that are immediately blocked, but since the entire monitor implementation is aimed at small embedded systems, this is [currently] not a major drawback.

6. Related work

Many Java™implementations have spent efforts on keeping monitor overhead as low as possible when there is no lock contention, among these are *Electric Fire* [4], *Kaffe* [1], *Latte* [7], [2], but when contention occurs they all fall back to hash-based allocation of one monitor per object.

7. Conclusions and future work

A characteristic of most Java™ programs has been used to implement monitors more space efficiently than is usually done. Apart from the space savings, the new monitor model is easy to implement and makes deadlock detection and priority inheritance easy to implement. The presented model is meant to be used as part of an experimental real-time virtual machine [5], where most of the run-time system will be implemented in Java™. The feasabilty of the proposed scheme for multi-processor systems also needs investigation.

8. Acknowledgments

This work has been supported by Lucas – Center for Applied Software (`http://www.lucas.lth.se/`).

References

[1] "Kaffe: Java Virtual Machine ." `http://www.transvirtual.com/`.

[2] "LaTTe : An Open-Source Java Virtual Machine and Just-in-Time Compiler." `http://latte.snu.ac.kr/`.

[3] A. Burns and A. Wellings. *Real–Time Systems and Their Programming Languages*. Addison–Wesley, 1997.

[4] S. Furman. "Electrical Fire." `http://www.mozilla.org`.

[5] A. Ive. *Implementation of an embedded real-time Java virtual machine prototype*. licenciate thesis, Department of Computer Science, Lund Institute of Technology, 2001.

[6] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification, Second Edition*. Addison–Wesley, 1999.

[7] B.-S. Yang, J. Lee, J. Park, S.-M. Moon, K. Ebcioglu, and E. Altman. "Lightweight Monitor for Java VM." *ACM SIGARCH Computer Architecture News*, March, March 1999.

Appendices

A. A straightforward monitor implementation

```
class Object {
  Monitor lock;
  static boolean creatingLock = false;
  public Object() {
    if (! creatingLock) {
      // we have to avoid infinite recursion
      creatingLock = true;
      lock = new Monitor();
      creatingLock = false;
    }
  }
}
class Thread {
  final int running = 0;
  final int ready = 1;
  final int blocked = 2;
  int state;
}
class ThreadQueue {
  /** Put thread in queue */
  void add(Thread thread);
  /** Remove a thread from queue */
  Thread remove();
  /** Check if the queue is empty */
  boolean empty();
}
class Monitor {
  Thread owner;
  int count;
  ThreadQueue blocked;
  ThreadQueue waiting;
  Monitor() {
    owner = null;
    count = 0;
    blocked = new ThreadQueue();
    waiting = new ThreadQueue();
  }
  void enter(Thread thread) {
    disableScheduling();
    while (true) {
      if (owner == null || owner == thread) {
        owner = thread;
        count++;
        break;
      } else {
        thread.state = Thread.blocked;
        blocked.add(thread);
        reschedule();
      }
    }
    enableScheduling();
  }
  void leave() {
    disableScheduling();
    count– –;
    if (count == 0) {
      owner = null;
      Thread next = blocked.remove();
      next.state = Thread.ready;
      reschedule();
    }
    enableScheduling();
  }
  void wait() {
    disableScheduling();
    waiting.add(owner);
    int oldCount = count;
    Thread oldOwner = owner;
    count = 0;
    owner = null;
    while (owner != null) {
      oldOwner.state = Thread.blocked;
      reschedule();
    }
    count = oldCount;
    owner = oldOwner;
    enableScheduling();
  }
  void notify() {
    disableScheduling();
    Thread toBeNotified = waiting.remove();
    toBeNotified.state = Thread.ready;
    reschedule();
    enableScheduling();
  }
  void notifyAll() {
    while (!waiting.empty()) {
      notify();
    }
  }
}
```

```
class JVM {
  void interpret() {
    ...
    switch (bytecode) {
      ...
      case monitorenter: {
        stack.pop().lock.enter(currentThread);
      } break;
      case monitorexit: {
        stack.pop().lock.leave();
      } break;
      ...
    }
}
```

B. Lazy creation and eager destruction

```
class Object {
  Monitor lock = null;
}
class Monitor {
  Object object;
  Monitor(Object object) {
    this.object = object;
  }
  ...
  void enter(Thread thread) {
    disableScheduling();
    while (true) {
      if (owner == null || owner == thread) {
        owner = thread;
        count++;
        break;
      } else {
        thread.state = Thread.blocked;
        blocked.add(thread);
        reschedule();
      }
    }
    enableScheduling();
  }
  void leave() {
    disableScheduling();
    count– –;
    if (count == 0) {
      owner = null;
      Thread next = o.lock.blocked.remove();
      next.state = Thread.ready;
      if (count == 0 &&
          blocked.empty() &&
          waiting.empty()) {
        // Delete lock
        object.lock = null;
      reschedule();
    }
    enableScheduling();
  }
  ...
}
class JVM {
      ...
      case monitorenter: {
        Object o = stack.pop()
        if (o.lock == null) {
          o.lock = new Monitor(o);
        }
        o.lock.enter(currentThread);
      } break;
      case monitorexit: {
        stack.pop.leave();
      } break;
      ...
}
```

C. One monitor per thread

```
class Object {
  Thread lock = null;
}
class Thread {
  private Thread nextBlocked;
  private Thread nextWaiting;
  private Object waitingFor;
  private int last;
  private int count[];
  private Object object[];
  private Thread waiting[];
  private int state;
  private int assignedPri;
  private int effectivePri;
  final int running = 0;
  final int ready = 1;
  final int blocked = 2;
  final int waiting = 3;
  public Thread() {
    waitingFor = null;
    last = 0;
    count = new int[initialSize];
    object = new Object[initialSize];
    waiting = new Thread[initialSize];
  }
  final int objectPos(Object o) {
    int pos;
    for (pos = 0 ; pos < last ; pos++) {
      if (object[pos] == o) {
        break;
      }
    }
    return pos;
  }
  void enter(Object o) {
    disableScheduling();
    int pos = objectPos(o);
    object[pos] = o;
    if (pos >= last) {
      count[pos] = 0;
      last++;
    }
    while (true) {
      if (o.lock == null || o.lock == this) {
        break;
      } else if (o.lock.waitingFor == o) {
        // Grab the waiting queue
        waiting[pos] = o.lock;
        break;
      } else {
        if (o.lock.effectivePri < effectivePri) {
```

```
            o.lock.effectivePri = effectivePri;
            if (o.lock.state == blocked) {
              o.lock.state = ready;
            }
          }
          // Put thread in the blocked queue
          nextBlocked = o.lock.nextBlocked;
          o.lock.nextBlocked = this;
          state = Thread.blocked;
          reschedule();
        }
      }
      o.lock = this;
      count[pos]++;
      enableScheduling();
    }
    void leave(Object o) {
      disableScheduling();
      int pos = objectPos(o);
      count[pos]– –;
      if (count[pos] == 0) {
        // Transfer waiting threads to
        // one of the waiting threads.
        o.lock = waiting[pos];
        // Wake up all blocked threads
        effectivePriority = assignedPriority;
        last– –;
        object[pos] = null;
        while (nextBlocked != null) {
          nextBlocked.state = Thread.ready;
          nextBlocked = nextBlocked.nextBlocked;
        }
        reschedule();
      }
      enableScheduling();
    }
    void wait(Object o) {
      disableScheduling();
      waitingFor = o;
      int pos = objectPos(o);
      int oldCount = count[pos];
      count[pos] = 0;
      state = Thread.waiting;
      reschedule();
      waitingFor = null;
      while (true) {
        if (o.lock == null) {
          break;
        } else if (o.lock.waitingFor == o) {
          // Grab the waiting queue
          waiting[pos] = o.lock;
          break;
        } else {
          if (o.lock.effectivePri < effectivePri) {
            o.lock.effectivePri = effectivePri;
            if (o.lock.state == blocked) {
              o.lock.state = ready;
            }
          }
          // Put thread in the blocked queue
          nextBlocked = o.lock.nextBlocked;
          o.lock.nextBlocked = this;
          state = Thread.blocked;
          reschedule();
        }
      }
      o.lock = this;
      count[pos] = oldCount;
      enableScheduling();
    }
    void notify(Object o) {
      disableScheduling();
      int pos = objectPos(o);
      if (waiting[pos] != null) {
        Thread toBeNotified = waiting[pos];
        waiting[pos] = waiting[pos].nextWaiting;
        toBeNotified.state = Thread.ready;
        reschedule();
      }
      enableScheduling();
    }
  }
  class JVM {
      ...
      case monitorenter: {
        currentThread.enter(stack.pop());
      } break;
      case monitorexit: {
        currentThread.leave(stack.pop());
      } break;
      ...
```