MX in Java [1] [2]

IT Systems, Summer Term 2026

Dr. Matthes Elstermann

The topic of mutual exclusion is explained in three presentations, of which this is the second one.

1 Introduction

- Part 1
  - Introduction
  - Race Conditions
  - Critical Sections and Mutual Exclusion
  - Locking
  - Pointers beyond class topics
- Part 2
  - Monitors
  - MX with Monitors in Java
  - Cooperation with Monitors in Java
- Part 3
  - Deadlocks
  - Deadlock Strategies
  - Further Challenges
  - Conclusions

Based on the concepts introduced in the first part, we now look at monitors in general and their implementation in Java in particular. Concerning Java, we first see how to guarantee MX with monitors, before we look at an example for cooperation with monitors.

1.1 Thread Terminology

Take this quiz.

1.2 Thread States

Take this quiz.

1.3 Java Threads

Take this quiz.

1.4 Races

Take this quiz.

1.5 Mutual Exclusion

Take this quiz.

[1] This PDF document is an inferior version of an OER in HTML format; free/libre Org mode source repository.
[2] Material created by Jens Lechtenbörger; see end of document for license information.

1.6 The Deadlock Empire, Part 2

- Continue playing Deadlock Empire

Figure 1: "Logo for Deadlock Empire" under GPLv2; from GitHub

- Play "Insufficent Lock" and "Deadlock" at https://deadlockempire.github.io/
  - `Monitor` in game is just a lock, which is locked with `enter()` and unlocked with `exit()`
  - Differently from general monitor concept (and Java implementation) introduced subsequently

Continue with some levels of Deadlock Empire.

Beware though! What is called "Monitor" in the Game, is not the same as the Monitor concept introduced next and implemented in Java. The Monitor in the game is essentially used like a lock.

2 Monitors

The monitor concept offers a fundamental MX mechanism.

2.1 Monitor Idea

- Monitor ≈ instance of class with methods and attributes
- Equip **every** object (= class instance) with a lock
  - **Automatically**
    - Call `lock()` when method is entered
      - As usual: Thread is blocked if lock is already locked
      - Thus, **automatic MX**
      - We say that executing thread **entered the monitor** or **executes inside the monitor** when it has passed `lock()` and executes a method
    - Call `unlock()` when method is left
      - Thread **leaves the monitor**

The basic idea of monitors is as follows: Think of a monitor as a special class, whose instances are automatically protected with their **own locks**. The run-time system ensures that before a method is executed on such a monitor instance, the lock for that instance needs to be acquired.

We say that a thread that has successfully executed `lock()`, "entered the monitor" or "executes inside the monitor".

Thus, monitors automatically provide mutual exclusion for methods of the monitor class: If multiple threads share the same object (with a potential for race conditions), only one of them can execute inside the monitor at any point in time, while others are blocked.

Importantly, each object has its own lock. Thus, two threads that operate on different class instances can both acquire their different locks and execute **the same monitor method in parallel** (without the danger of races as they do not share resources). Thus, to explicitly point out a frequent misunderstanding: Methods are **not** locked; instead, objects are protected with locks. Then, a thread may be blocked when it attempts to execute a method that requires access to an already locked object.

The next slide explains the origin of monitors in terms of an abstract data type (instead of the more modern "class" formulation presented here). On that slide, you also see that monitors not only guarantee MX; in addition, they provide methods for cooperation of threads.

Subsequent slides then discuss how the monitor concept is implemented in Java with the keyword `synchronized` (which activates locking of the `this` object as explained here in general terms) and methods for cooperation.

2.2 Monitor Origin

- Monitors proposed by Hoare; 1974
- **Abstract data type** with MX guarantee

Monitors were proposed decades ago by famous computer scientist Tony Hoare. You may also know him as developer of quicksort.

Monitors can be understood as abstract data type that guarantees MX.

- Methods encapsulate local variables
  - Just like methods in Java classes

  Importantly, just as with object-oriented programming, monitors encapsulate local variables as embedded data. Thus, all accesses to embedded data require method invocations.
- Thread enters monitor via method
  - **Built-in MX**: At most one thread in monitor

  Threads executing monitor methods automatically try to acquire a lock for the embedded data. As only one thread can own the lock, monitor methods guarantee MX. As stated on the previous slide, we say that the successful thread "enters" the monitor. While one thread executes inside the monitor, other threads are blocked when they attempt to enter as well.
- In addition: Methods for cooperation
  - `cwait(x)`: Blocks calling thread until `csignal(x)`
    - Monitor free then
  - `csignal(x)`: Starts at most one thread waiting for x
    - If existing; otherwise, nothing happens

In addition to MX, monitors also offer methods that are meant for cooperation among threads. Here, a thread may find that it cannot continue before some other thread changed the state of the execution. Then, the first thread may call a wait method, which blocks it until some other thread sends a matching signal.

We will see an example in Java shortly.

3 MX with Monitors in Java

Let us see how to use monitors for MX in Java.

3.1 Monitors in Java: Overview

- In Java, classes and objects come with built-in locks
  - Which are ignored by default

  You may be surprised to learn that in Java, classes and objects come with built-in locks. However, those locks are ignored by default.
- Keyword `synchronized` activates locks
  - Automatic locking of `this` object during execution of method
    - Automatic MX for method's body
    - Useful if (large part of) body is a CS

    To activate those locks, the keyword `synchronized` is necessary. If you declare a method to be `synchronized`, a thread entering such a method automatically attempts to acquire the lock for the method's `this` object. Thus, based on usual locking functionality, MX is guaranteed for the object accessed in the method's body: The first thread acquires the lock for the duration of the method, while subsequent threads get blocked when they try to acquire the same lock.

    This approach is particularly useful if a large part of the method is a critical section.
  - E.g., for sample code from (Hailperin 2019) (for which you found races previously):

```
public synchronized void sell() {
  if (seatsRemaining > 0) {
    dispenseTicket();
    seatsRemaining = seatsRemaining - 1;
  } else displaySorrySoldOut();
}
```

E.g., for the ticket selling code, we can simply declare `sell()` to be a `synchronized` method, as shown here. Then, MX is guaranteed, and races around `seatsRemaining` no longer occur.

3.1.1 Java, `synchronized`, `this`

- Java basics, hopefully clear
  - Method `sell()` from previous slides invoked on some object, say `theater`
    - Each `theater` has its own attribute `seatsRemaining`
    - `seatsRemaining` is really `this.seatsRemaining`, which is the same as `theater.seatsRemaining`
      - Inside the method, the name `theater` is unknown, `theater` is the `this` object, which is used implicitly
- Without `synchronized`, races arise when two threads invoke `sell()` on the same object `theater`
  - With `synchronized`, only one of the threads obtains the lock on `theater`, so races are prevented

This slide lists basic facts about the code on the previous slide. Please think about them.
Ask if necessary.

3.1.2 Possible Sources of Confusion

- With `synchronized`, **locks** for **objects** are activated
  - For `synchronized` methods, thread needs to acquire lock for `this` object

To sum up, with `synchronized`, **locks** for **objects** are activated. When executing a `synchronized` method, the thread needs to acquire the lock for the `this` object. This procedure turns the method into a critical section with MX for the `this` object.

- Methods **cannot** be locked

  Note, again, that methods are not locked.
- Individual attributes of the `this` object (e.g., `seatsRemaining`) are **not** locked
  - (Which is not a problem as object-orientation recommends to **encapsulate** attributes, i.e., they cannot be accessed directly but only through `synchronized` methods)

Also, a lock for the `this` object is acquired. However, individual attributes of that object are not locked. If accesses to attributes are encapsulated by methods, as they should be, this does not pose a problem for MX.

3.1.3 Self-Study Task

1. Inspect and understand, compile, and run this sample program, which embeds the code to sell tickets, for which you found races previously.
2. Change `sell()` to use the monitor concept, recompile, and run again. Observe the expected outcome.

(Nothing to submit here; maybe ask questions online.)
Please convince yourself that `synchronized` is all you need to prevent races for the ticket selling code.

3.2 Java Monitors in Detail

- MX based on monitor concept
  - See Java specification if you are interested in details

Java implements the monitor concept, with details specified at the URL given here.

- **Every** Java object (and class) comes with
  - Monitor with **lock** (not activated by default)
    - Keyword `synchronized` activates lock

In essence, MX with Java is quite simple, as every Java object is equipped with a lock. By default, however, these locks are not used. Instead, you need to use the keyword `synchronized` if you want threads to acquire the locks for MX.

  - For method: `public synchronized methodAsCS(...) {...}`
    - First thread acquires **lock** for `this` object upon call (Class object for static methods)
    - Further threads get **blocked**

The simplest way to enforce MX is to declare methods operating on shared resources as `synchronized`. If a thread wants to execute such a synchronized method on some object, then the thread automatically attempts to acquire the lock for that object. If that lock has been taken by another thread, then the locking attempt is not successful but leads to blocking of the thread. Once the thread holding the lock leaves the method, it automatically releases the lock, and other threads blocked on that lock are made runnable by the OS. Thus, locking attempts of previously blocked threads can continue when they are scheduled again.
MX with monitors is really that simple.

- Or for block: `synchronized (syncObj) {...}`
  - Thread acquires **lock** for `syncObj`

    Besides, you can also use other objects for synchronization if you want to turn blocks of code into critical sections. We will not use this, however.

- **Wait set** (set of threads; `wait()` and `notify()`, explained next)

  Finally, the Java monitor concept also includes a mechanism for cooperation based on waiting and signaling. For this purpose, Java adds a wait set to each object, to be explained subsequently.

4 Cooperation with Monitors in Java

Let us see an example for cooperation with monitors in Java.

4.1 Producer/Consumer problems

- Classical synchronization problems
  - Producers produce data, to be consumed by consumers
  - Data in shared data structure, e.g., Java array
    - Synchronization for data structure necessary

      So-called producer/consumer problems are classical examples for mutual exclusion and thread synchronization in OS contexts.

      Here, two types of threads cooperate: Producers produce some data, e.g., records, messages, or tasks, which are then consumed by consumers. Producers place the generated data into a shared data structure, from which consumers retrieve data. Thus, all these threads race around the shared data structure, with accessing code in critical sections, for which mutual exclusion is necessary.
  - One or more **producers**
    - Generate data, e.g., records, messages, tasks
    - Place data into **buffer** (shared resource)
      - Two buffer variants: unbounded or bounded
      - Producer **blocks**, if bounded buffer is full
  - One or more **consumers**
    - Consume data
      - Take data out of **buffer**
      - Consumer **blocks**, if buffer is empty

In general, arbitrary numbers of producers and consumers may exist, and different data structures may be used to manage the data exchange between them. Frequently, buffers are used as abstraction for such data structures. Importantly, buffers may be bounded, i.e., have limited capacity, or they may be unbounded.

If a buffer is bounded, an insert operation by a producer may be a blocking operation: If the buffer is full, the thread is blocked until a consumer frees a slot in the buffer.

Subsequently, you will see a method to insert data into a bounded buffer, which is based on a Java array.

A get or retrieve operation by a consumer may also be blocking: If the buffer is empty, the thread is blocked until a producer filled a slot in the buffer.

4.2 Ideas for Cooperation

- Use waiting and signaling of monitors
- Threads may work with different roles on shared data structures
  - E.g., producer/consumer problems on previous slide
- Some may find that they cannot continue before others did their work
  - The former call `wait()` and hope for `notify()` by the latter
  - Cooperation (**orthogonal to and not necessary for MX!**)
    - **Wait set** mentioned above and explained subsequently

Recall that monitors come with a mechanism that allow threads to wait for signals by others. This mechanism is particularly useful if threads work with different roles that fulfil complementary tasks.

Producers and consumers are a prime example for such roles.

As just explained, consumers may find an empty buffer, which means that they cannot do anything useful. Thus, they should be taken aside and not be considered by the scheduler, until a producer placed new data into the buffer. With monitors in Java, such a consumer thread invokes the method `wait()`, which blocks it until some thread, ideally a producer, calls `notify()`.

In Java, such blocked threads are collected in the wait set of the `this` object, for which the next slide provides more details.

4.3 `wait()` and `notify()` in Java

Cooperation between threads sharing resources can be managed with the methods `wait()` and `notify()` (or `notifyAll()`). A thread can only invoke these methods on an object if it has acquired the lock for that object, i.e., if it currently executes inside the object's monitor. Thus, usually, you see invocations of `wait()` and `notify()` in synchronized methods.

- Waiting via blocking
  - `wait()`: thread **unlocks** and **leaves** monitor, enters **wait set**
    - Thread enters state **blocked**
    - Called by thread that cannot continue (without work/help of another thread)

  If a thread finds that it cannot make use of the shared resource in the current state, it can invoke `wait()` to release the lock on that resource and leave its monitor. At that point in time, the thread's state changes to blocked, and the thread is recorded in a special data structure associated with the object, called wait set. In the wait set, Java keeps track of all threads that have invoked `wait()` on the object.

  Once a thread has executed `wait()`, the object's lock is released, and other threads can acquire the object's lock and modify the object's state.

- Notifications
  - `notify()`
    - Remove one thread from **wait set** (if such a thread exists)
      - Change its state from blocked to runnable
    - Called by thread whose work may help another thread to continue
  - `notifyAll()`
    - Remove all threads from **wait set**
      - Only one can lock and enter the monitor, of course
      - Only after the notifying thread has left the monitor, of course
      - Overhead (may be avoidable with appropriate synchronization objects)

  If a thread has modified the object's state in such a way that there is reason to believe that waiting threads might now be able to continue, the thread invokes `notify()` on the object, which removes one thread from the wait set and makes it runnable. When that runnable thread is scheduled for execution later on, it can again try to enter the monitor by locking the object; once the lock has been acquired, the thread resumes execution after the `wait()` method.

  The method `notifyAll()` is an alternative to `notify()` that removes all threads from the wait set, not just one. You may want to think about advantages and disadvantage of notifying all waiting threads yourself.

4.4 Sample `synchronized` Java Method

```
// Based on Fig. 4.17 of [Hai17]
public synchronized void insert(Object o)
  throws InterruptedException
// Called by producer thread
{
  while(numOccupied == buffer.length)
      // block thread as buffer is full;
      // cooperation from consumer required to unblock
      wait();
  buffer[(firstOccupied + numOccupied) % buffer.length] = o;
  numOccupied++;
  // in case any retrieves are waiting for data, wake/unblock them
  notifyAll();
}
```

(Part of SynchronizedBoundedBuffer.java)

This slide shows the `synchronized` method `insert()` of a bounded buffer, which is called by producers to insert some object. The URL points to the full example, which also includes a method `retrieve()` for consumers. In the same directory, thread classes are available as well.

As explained earlier, the method is `synchronized` to guarantee MX, which prevents races on the shared data structure.

Initially, the producer checks whether the buffer is full. If so, it calls `wait()`, hoping for a notification from a consumer thread. Otherwise, the producer inserts the object at a free position.

This buffer implementation makes use of an array, which has a fixed `length`. Note how two variables, `firstOccupied` and `numOccupied` combined with modulo arithmetic, determine a free slot in the array.

Eventually, the producer calls `notifyAll()`, which unblocks all threads waiting on the buffer. In particular, consumers that were waiting for a new object in the buffer can now try again to retrieve an object.

4.5 Comments on `synchronized`

- Previous method in larger program: bb.zip
  - `SynchronizedBoundedBuffer` as shared resource
  - Different threads (Producer instances and Consumer instances) call `synchronized` methods on that bounded buffer
    - Before methods are executed, lock of buffer needs to be acquired
      - This enforces MX for methods `insert()` and `retrieve()`
    - In methods, threads call `wait()` on buffer if unable to continue
      - `this` object used implicitly as target of `wait()`
      - Thread enters wait set of buffer
      - Until `notifyAll()` on same buffer
    - Note that thread classes contain neither `synchronized` nor `wait/notify`

The first URL here points to a ZIP archive which contains a test program that you can compile and run to try out the code examples in these slides. Note that the archive not only contains an implementation based on the monitor concept but also with semaphores. You can ignore the latter.

In any case, note how the threads call `synchronized` methods on the shared bounded buffer, which guarantees MX. In addition, `wait()` and `notify()` are used in these methods for cooperation of producers and consumers as explained on the previous slide.

In contrast, thread classes do neither contain `synchronized` nor `wait()` nor `notify()`. Indeed, *resources* need to be protected with MX, not threads.

Bibliography

Hailperin, Max. 2019. *Operating Systems and Middleware – Supporting Controlled Interaction*. revised edition 1.3.1. https://github.com/Max-Hailperin/Operating-Systems-and-Middleware--Supporting-Controlled-Interaction.

The bibliography contains references used in this presentation.

License Information