

# OS06: MX in Java \*

Based on Chapter 4 of [Hai19]

Jens Lechtenbörger

Computer Structures and Operating Systems 2023

## 1 Introduction

### 1.1 OS Plan

- [OS Overview](#) (Wk 20)
- [OS Introduction](#) (Wk 21)
- [Interrupts and I/O](#) (Wk 21)
- [Threads](#) (Wk 23)
- [Thread Scheduling](#) (Wk 24)
- [Mutual Exclusion \(MX\)](#) (Wk 25)
- [MX in Java](#) (Wk 25)
- [MX Challenges](#) (Wk 25)
- [Virtual Memory I](#) (Wk 26)
- [Virtual Memory II](#) (Wk 26)
- [Processes](#) (Wk 27)
- [Security](#) (Wk 28)

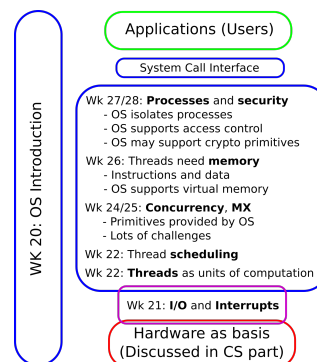


Figure 1: OS course plan, summer 2022

### 1.2 Today's Core Question

- How to achieve MX with monitors in Java?
- (Also: Semaphores revisited with Java as alternative)

### 1.3 Learning Objectives

- Apply and explain MX and cooperation based on monitor concept in Java

\*This PDF document is an inferior version of an OER HTML page; free/libre Org mode source repository.

- Give example
- Discuss incorrect attempts

## 1.4 Retrieval Practice

### 1.4.1 Thread Terminology

### 1.4.2 Thread States

### 1.4.3 Java Threads

### 1.4.4 Races

### 1.4.5 Mutual Exclusion

## Table of Contents

# 2 Monitors

## 2.1 Monitor Idea

- Monitor  $\approx$  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 an instance of a special type of class, where each instance is automatically equipped with its **own lock**. The run-time system ensures that before a method of such a class is executed on a class instance (which is `this` in Java), the lock for that class 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 MX 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 monitor methods in parallel (without the danger of races as they do not share resources).

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**
  - Methods encapsulate local variables
    - \* Just like methods in Java classes
  - Thread enters monitor via method
    - \* **Built-in MX**: At most one thread in monitor
  - 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
      - Difference to semaphore: signal may get lost

## 3 MX in Java

### 3.1 Monitors in Java: Overview

- In Java, classes and objects come with built-in locks
  - Which 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
  - E.g., for sample code from [Hai19] (for which you [found races](#) previously):

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

#### 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

### 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
- Methods **cannot** be 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)

### 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.)

## 3.2 Java Monitors in Detail

- MX based on monitor concept
  - See Java specification if you are interested in details
- **Every** Java object (and class) comes with
  - Monitor with **lock** (not activated by default)
    - \* Keyword `synchronized` activates lock
    - \* For method
      - `public synchronized methodAsCS(...) {...}`
      - Thread acquires **lock** for `this` object upon call (Class object for static methods)
    - \* Or for block
      - `synchronized (syncObj) {...}`
      - Thread acquires **lock** for `syncObj`
    - \* First thread **acquires lock** for duration of method/block

- \* Further threads get **blocked**
- **Wait set** (set of threads; `wait()` and `notify()`, explained later; ignore for now)

1. Java provides all methods for mutual exclusion discussed in the previous presentation, including the monitor concept, whose details can be found at the URL given here.
2. 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.

The simplest way to enforce MX is to declare methods operating on shared resources as `synchronized`. If a thread T1 wants to execute such a synchronized method on some object, then thread T1 will automatically try to acquire the lock for that object. If that lock has been taken, say by thread T0, then T1 will be blocked until T0 leaves the method and releases the lock.

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.

Finally, the Java monitor concept includes a mechanism for cooperation of threads based on wait sets, which will be explained later.

### 3.3 Recall: synchronized Example

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

- As you observed above, `synchronized` avoids races
  - Method executed under MX
  - Threads need to acquire lock on **this** object before executing method
- Really, it is that simple!

## 4 Cooperation with Monitors in Java

### 4.1 General Idea

- Threads may work with different roles on shared data structures
  - E.g., [producer/consumer problems seen earlier](#)
- 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!**)
    - \* General monitor concept
    - \* **Wait set** mentioned above and explained subsequently

## 4.2 wait() and notify() in Java

- Waiting via blocking
    - wait(): thread **unlocks** and **leaves** monitor, enters **wait set**
      - \* Thread enters **state blocked** (no busy waiting)
      - \* Called by thread that cannot continue (without work/help of another thread)
  - Notifications
    - notify()
      - \* Remove one thread from **wait set** (if such a thread exists)
        - Change 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 or with [semaphores as seen previously](#))
1. 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. So, usually, you see invocations of wait and notify in synchronized methods.
  2. If a thread finds that it cannot make use of the shared resource in the resource's 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. So, 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.
  3. 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 to 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.

## 5 BoundedBuffer in Java

### 5.1 Bounded Buffers

- A **buffer** is a data structure to store items, requests, responses, etc.
  - Lots of buffer variants exist
  - A **bounded** buffer has a limited capacity

- \* E.g., a `Java array` or any other data structure of limited capacity
- As with any other data structure, MX is necessary when buffers are shared
  - Subsequently, two alternative buffer implementations with MX
    - \* Java's monitor concept (with array as underlying, shared data structure)
    - \* Java Semaphore (with list as underlying, shared data structure)

## 5.2 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`)

### 5.2.1 Comments on synchronized

- Previous method in larger program: `BBTest.java`
  - `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`

## 5.3 Sample Semaphore Use in Java

```
import java.util.concurrent.Semaphore;
/*
    This code is based on Figure 4.18 of the following book:
    Max Hailperin, Operating Systems and Middleware - Supporting
```

Controlled Interaction, revised edition 1.3, 2017.  
<https://gustavus.edu/mcs/max/os-book/>

In Figure 4.18, `synchronizedList()` is used, whereas here a plain `LinkedList` is used, which is protected by the additional semaphore mutex.

Also, the class here is renamed and implements a new interface.  
\*/

```
public class SemaphoreBoundedBuffer implements BoundedBuffer {
    private java.util.List<Object> buffer =
        new java.util.LinkedList<Object>();

    private static final int SIZE = 20; // arbitrary

    private Semaphore mutex = new Semaphore(1);
    private Semaphore occupiedSem = new Semaphore(0);
    private Semaphore freeSem = new Semaphore(SIZE);

    /* invariant: occupiedSem + freeSem = SIZE
       buffer.size() = occupiedSem
       buffer contains entries from oldest to youngest */

    public void insert(Object o) throws InterruptedException {
        // Called by producer thread
        freeSem.acquire();
        mutex.acquire();
        buffer.add(o);
        mutex.release();
        occupiedSem.release();
    }

    public Object retrieve() throws InterruptedException {
        // Called by consumer thread
        occupiedSem.acquire();
        mutex.acquire();
        Object retrieved = buffer.remove(0);
        mutex.release();
        freeSem.release();
        return retrieved;
    }

    public int size() {
        return buffer.size();
    }
}
```

### 5.3.1 Comments on Java Semaphore

- Java provides `java.util.concurrent.Semaphore`

- Implements [semaphore concept](#)
  - \* Constructor with integer argument to track resources
  - \* Methods `acquire()` and `release()`
- `SemaphoreBoundedBuffer` implements same interface as `SynchronizedBoundedBuffer`
  - Use whichever you want with `BBTest.java`
  - Bounded buffer uses three semaphores
    - \* One (initialized to 1) acting as mutex
      - Note `acquire()` and `release()` around buffer accesses for `MX`
    - \* Other two counting occupied and free places

## 6 Conclusions

### 6.1 Summary

- Java objects can act as monitors
  - Keyword `synchronized`
    - \* `MX` for `CS` (method/block of code)
      - No flags, no explicit locks!
  - Cooperation via `wait()` and `notify()`

### Bibliography

- [Hai19] Max Hailperin. *Operating Systems and Middleware – Supporting Controlled Interaction*. revised edition 1.3.1, 2019. URL: <https://gustavus.edu/mcs/max/os-book/>.

### License Information

This document is part of an Open Educational Resource (OER) course on Operating Systems. Source code and source files are available on GitLab under free licenses.

Except where otherwise noted, the work “OS06: MX in Java”, © 2017-2023 Jens Lechtenbörger, is published under the Creative Commons license CC BY-SA 4.0.

No warranties are given. The license may not give you all of the permissions necessary for your intended use.

In particular, trademark rights are *not* licensed under this license. Thus, rights concerning third party logos (e.g., on the title slide) and other (trade-) marks (e.g., “Creative Commons” itself) remain with their respective holders.