OS06: Monitors in Java

Based on Chapter 4 of [Hai17]

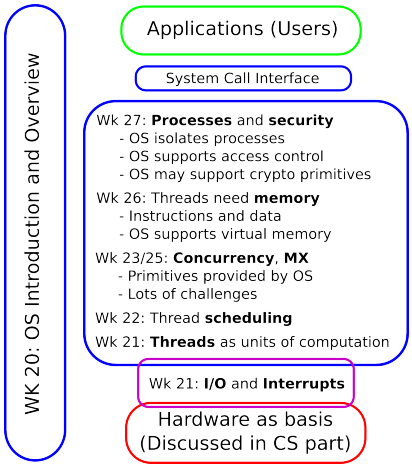
Jens Lechtenbörger

Computer Structures and Operating Systems 2019

1 Introduction

1.1 OS Plan

- OS Motivation (Wk 20)
- OS Introduction (Wk 20)
- Interrupts and I/O (Wk 21)
- Threads (Wk 21)
- Thread Scheduling (Wk 22)
- Mutual Exclusion (MX) (Wk 23)
- MX in Java (Wk 23)
- MX Challenges (Wk 25)
- Virtual Memory I (Wk 26)
- Virtual Memory II (Wk 26)
- Processes (Wk 27)
- Security (Wk 27)
- Wrap-up (Wk 28)



1.2 Today's Core Question

- How to achieve MX with monitors in Java?

1.3 Learning Objectives

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

Contents

2 Monitors

2.1 Monitor Idea

- Monitor ≈ class with methods and attributes
- Equip **every** object (= class instance) with a lock
  - **Automatically**
    - Call `lock()` when method is entered
    - Call `unlock()` when method is left

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
  - Turn method's body into CS
  - E.g., for sample code from [Hai17] (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`, **objects** are locked
  - For `synchronized` methods, the `this` object
- Methods **cannot** be locked

- Individual attributes of the `this` object (e.g., `seatsRemaining`) are **not** locked
  - (Which is not a problem as attributes should be **encapsulated**, i.e., not be accessed directly but through `synchronized` methods)

3.1.3 JiTT Assignment

1. Inspect and understand, compile, and run this sample program, which embeds the code to sell tickets, for which you found races previously.
2. Turn `sell()` into a monitor, 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
- **Every** Java object (and class) comes with
  - Monitor with **lock** (not activated by default)
    - Keyword `synchronized` activates lock
    - For method
      - `public synchronized methodAsCS(...) {...}`
      - Thread **locks** the **this** object upon call (Class object for static methods)
    - Or for block
      - `synchronized (syncObj) {...}`
      - Thread **locks** `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` turns method into critical section
  - Method executed under mutual exclusion
  - Threads need to acquire lock on `this` object before executing method
- Really, it is that simple!

3.4 Understanding Required

- Sample scenario (inspired by student's exam attempt)
  - All threads share some object `shared`
  - Every thread has its own `number`
- The following does **not** work! (Full source code)
  - This serves as **bad** example, don't do this

```
/* Following method called from run().
   Repeatedly assign this thread's own number to a
   shared object and race to read that number again.
   Exit, if other thread modified number in between. */
public synchronized void setGet() {
  while(true) {
    shared.setNo(number);
    int no = shared.getNo();
    if (no != number) {
      System.out.println("Thread " + number + " sees " + no);
      System.exit(no); }}}
```

The previous slide contains an example for MX with a synchronized method sell(). This slide seems to contain a similar example for MX with a synchronized method setGet(). However, this slide serves as counter example, and you should figure out why it does not work in the next JiTT assignment.

3.5 Ticket to Exam

Submit your solution in Learnweb.

Inspect and understand, compile, and run the Java program containing the code from the previous slide. The intention of introducing `synchronized` was to create a critical section containing `shared.setNo(number)` and `shared.getNo()`, with the effect that `no` always equals `number`.

Apparently, no such critical section exists, and races around `shared` arise. Why? What is locked by whom thanks to `synchronized` in that code?

4 Cooperation 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 size
  - E.g., a Java array
- As with any other data structure, MX is necessary when buffers are shared
  - In following
    - Java arrays used as underlying, shared data structure to implement buffer
    - Java's monitor concept for MX and cooperation

5.2 Sample 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();
}
```

5.2.1 Some Comments

- 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
* 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 JiTT Assignments

5.3.1 BoundedBuffer in Java

The previously shown code for `insert()` is contained in this archive. Use it to answer the following questions in Learnweb.

1. Is the following a possible beginning of output when executing `java BBTest synchronized`? Why (not)? (You need to analyze the scope of critical sections. Which statements are executed inside, which ones outside?)

   ```
   Consumer1 before retrieve(), size == 0
   Consumer2 before retrieve(), size == 0
   Producer1 before insert(), size == 0
   Consumer2 retrieved: 519
   Consumer2 before retrieve(), size == 0
   Producer1 entered: 519
   Producer1 before insert(), size == 0
   ```

2. For that program construct and explain a possible sequence of actions by producers and consumers (a so-called trace) that shows how `notifyAll()` may also wake up too many threads (where some or all of the awoken threads cannot do anything useful but `wait()` immediately, leading to a waste of CPU cycles). Would that trace also be possible with MX based on semaphores (also part of the archive)? Why (not)?

5.3.2 Questions, Feedback, and Improvements

Answer the following question in Learnweb.

What did you find difficult or confusing about the **contents** of the presentation? Please be as specific as possible. For example, you could describe your current understanding (which might allow us to identify misunderstandings), ask questions that allow us to help you, or suggest improvements (maybe on GitLab). You may submit individual questions as response to this task or ask questions in our Riot room and the Learnweb forum. Most questions turn out to be of general interest; please do not hesitate to ask and answer in forum and Riot room. If you created additional original content that might help others (e.g., a new exercise, an experiment, explanations concerning relationships with different courses, ...), please share.

6 In-Class Meeting

6.1 Inspect Java Code

The following piece of Java code is a student's exam attempt to explain MX and cooperation in Java. Grade that piece of code from an instructor's perspective.

```
public synchronized void run() {
  ...
  while(isWorking)
  wait();
  isWorking = true;
  /*Critical Section */
  isWorking = false;
  notify();
}
```

7 Conclusions

7.1 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

[Hai17] Max Hailperin. *Operating Systems and Middleware – Supporting Controlled Interaction*. revised edition 1.3, 2017. URL: https://gustavus.edu/mcs/max/os-book/.

License Information