

OS05: Mutual Exclusion

Based on Chapter 4 of [Hai17]

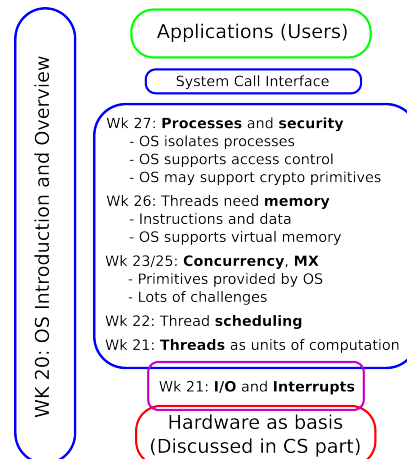
Jens Lechtenbörger

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 Questions

- What can go wrong with concurrent computations?
 - What is a race condition?
- How to avoid subtle programming bugs related to timing issues?
 - What mechanisms does the OS provide to help?

1.3 Learning Objectives

- Recognize and explain race conditions
 - (Playing [Deadlock Empire](#) helps)
- Explain notions of critical section and mutual exclusion
- Use mutexes and semaphores (and monitors after upcoming lectures) to enforce mutual exclusion

1.4 Retrieval Practice

1.4.1 Informatik 1

You have seen this advice before. It is repeated here for emphasis:

- What are **interfaces** and **classes** in Java, what is this?
- If you are not certain, consult a textbook; these self-check questions and preceding tutorials may help:
 - <https://docs.oracle.com/javase/tutorial/java/concepts/QandE/questions.html>
 - <https://docs.oracle.com/javase/tutorial/java/IandI/QandE/interfaces-questions.html>

1.4.2 Recall: Datenmanagement

- Give examples for **dirty read** and **lost update** anomalies.
- What is a database **transaction**?
- What does each of the four **ACID guarantees** mean?
- Explain **serializability** as notion of consistency.

The above is covered in this introduction to transaction processing.

Contents

1	Introduction	1
2	Race Conditions	3
3	Critical Sections and Mutual Exclusion	5
4	Locking	7
5	Semaphores	8
6	In-Class Meeting	11
7	Conclusions	14

2 Race Conditions

2.1 Central Challenge: Races



Figure 1: “Ferrari Kiss” by Antoine Valentini under CC BY-SA 2.0; from flickr

2.2 Races (1/2)

- **Race (condition):** a technical term
 - Multiple **activities** access **shared resources**
 - * At least one writes, in parallel or concurrently
 - Overall outcome depends on **subtle timing** differences
 - * “Nondeterminism” (recall Dijkstra on interrupts)
 - * Missing synchronization
 - * **Bug!**
- Previous picture
 - **Cars** are activities
 - **Street segments** represent shared resources
 - Timing determines whether a **crash** occurs or not
 - Crash = misjudgment = missing synchronization

2.3 Races (2/2)

- DBMS
 - **SQL commands** are activities
 - **Database objects** are shared resources
 - **Update anomalies** indicate missing synchronization
 - * Serializability requires synchronization and avoids anomalies
 - E.g., via locking

- OS
 - **Threads** are activities
 - **Variables, memory, files** are shared resources
 - Missing synchronization is a **bug**, leading to anomalies just as in database systems

2.4 JiTT Assignments

Work through the following tasks. Submit solutions in [Learnweb](#).

2.4.1 JiTT Assignment: The Deadlock Empire, Part 1

- Play “Tutorial 1,” “Tutorial 2,” and the three games for “Unsynchronized Code” at <https://deadlockempire.github.io/>
 - The games make use of C#
 - * (Which you do not need to know; the games include lots of explanations, also mouse-over helps)
- General idea
 - The game is about **critical sections**, which will be discussed next
 - * At any point in time just **one** thread is allowed to execute inside a critical section
 - * If you manage to lead two threads into a critical section simultaneously, you demonstrate a race condition
 - You **win** a game if you demonstrate a race condition
- Really, start playing **now!** (Nothing to submit here)

2.4.2 Ticket to Exam

Consider the following piece of Java code (from Sec. 4.2 of [Hai17]) to sell tickets as long as seats are available. (That code is embedded in [this sample program](#), which you can execute to see races yourself.)

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

Inspired by the Deadlock Empire, find and explain a race condition involving `seatsRemaining`, which leads to the last ticket being sold twice.

2.5 Non-Atomic Executions

- Races generally result from non-atomic executions
 - Even “single” instructions such as `i += 1` are **not atomic**
 - * Execution via **sequences of machine instructions**
 - Load variable’s value from RAM
 - Perform add in ALU
 - Write result to RAM
 - A context switch may happen after any of these machine instructions, i.e., “in the middle” of a high-level instruction
 - * Intermediate results accessible elsewhere
 - **No isolation** in the sense of ACID: races, dirty reads, lost updates
 - * Demo: Play a game as instructed previously

This slide highlights that even simple statements of high-level programming languages are not executed atomically, which may be the source of race conditions.

Note that the word “atomic” is used in its literal sense here. So, an execution is **not** atomic if it really consists of multiple steps.

Be careful not to confuse this with the notion of atomicity of ACID transactions. In the ACID context, atomicity means that transactions appear to be either executed entirely or not at all; whether they consist of multiple steps or not is not an issue.

3 Critical Sections and Mutual Exclusion

3.1 Goal and Solutions (1/2)

- Goal
 - Concurrent executions that access **shared resources** should be **isolated** from one another
 - * Cf. **I** in ACID
- Conceptual solution
 - Declare **critical sections** (CSs)
 - * Blocks of code with access to shared resources
 - Cf. transaction as sequence of operations
 - Enforce **mutual exclusion** (MX) on CSs
 - * At most one thread inside CS at any point in time
 - Cf. serializability for transactions

3.2 MX with CSs: Ticket example

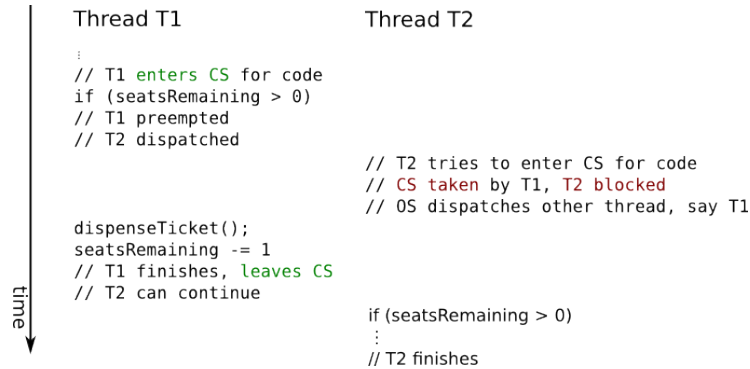


Figure 2: “Interleaved execution of threads with MX for code from Sec. 4.2 of book by Max Hailperin, CC BY-SA 3.0.” by Jens Lechtenbörger under CC BY-SA 4.0; from GitLab

1. The animation on this slide illustrates the effect of mutual exclusion on interleaved executions to avoid races based on a previously shown code fragment to sell tickets for seats. If you did not think about the JiTT assignment for that code fragment yet, please do so now and come back afterwards. Actually, this animation may also help you solving the JiTT assignment.
Consider two threads that simultaneously try to obtain seats, and suppose that the code fragment is executed as critical section under mutual exclusion. How MX can actually be enforced via locking, semaphores, or monitors is the topic of later slides.
2. T1 enters the CS first. Suppose its time slice runs out after the check that seats are still remaining. Now the OS dispatches T2, which wants to execute the same CS. However, as T1 is currently executing inside that CS and as MX is enforced for that CS, T2 is blocked by the OS. Thus, the OS schedules another thread for execution, say T1.
3. Consequently, T1 can finish the CS without intermediate modifications by any other thread.
4. Afterwards, T2 can enter the CS, check whether seats are still available etc. In essence, MX enforces the serial execution of CSs. On the one hand, serial executions are good as they avoids races; on the other, they inhibit parallelism, which is generally bad for performance.

3.3 Goal and Solutions (2/2)

- New goal
 - Implementations/mechanisms for MX on CS
- Solutions
 - **Locks**, also called **mutexes**
 - * Cf. 2PL in database systems
 - * Acquire lock/mutex at start of CS, release it at end
 - Choices: Busy waiting (spinning) or blocking when lock/mutex not free?
 - **Semaphores**

- * Abstract datatype, generalization of locks, blocking, signaling
- **Monitors**
 - * Abstract datatype, think of Java class, methods as CS with MX
 - * Keyword `synchronized` turns Java method into CS!

3.4 Challenges

- Above solutions restrict entry to CS
 - Thus, they restrict access to the resource CPU
- Major synchronization challenges arise
 - **Starvation**
 - * Thread never enters CS
 - (More generally: never receives some resource, e.g., CPU under scheduling)
 - **Deadlock** (discussed in later presentation)
 - * Set of threads is stuck
 - * Circular wait for additional locks/semaphores/resources/messages
 - In addition, programming errors
 - * Difficult to locate, time-dependent
 - * Difficult to reproduce, “non-determinism”

4 Locking

4.1 Mutexes

Warning! External figure **not** included: “Mutexes” © 2016 Julia Evans, all rights reserved from [julia’s drawings](#). Displayed here with personal permission. (See HTML presentation instead.)

4.2 Locks and Mutexes

- Lock = mutex = object with methods
 - `lock()` or `acquire()`: Lock/acquire/take the object
 - * A lock can only be `lock()`’ed by one thread at a time
 - * Further threads trying to `lock()` need to wait for `unlock()`
 - `unlock()` or `release()`: Unlock/release the object
 - * Can be `lock()`’ed again afterwards
 - E.g., interface `java.util.concurrent.locks.Lock` in Java.

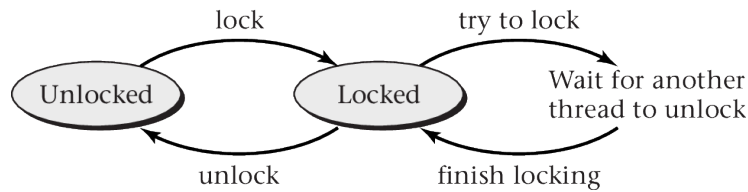


Figure 3: “Figure 4.4 of [Hai17]” by Max Hailperin under CC BY-SA 3.0; converted from GitHub

First of all, note that the terms “lock” and “mutex” are synonyms.

Locks (and mutexes) are special-purpose objects, which essentially have two states, namely Unlocked and Locked, which you can see in the figure here, along with possible state transitions when the lock’s methods `lock()` and `unlock()` are invoked. These methods are explained in the bullet points.

When mutual exclusion (MX) is necessary to prevent races for a critical section (CS), a lock object shared by the racing threads can be used, for example `seatlock` on the next slide. To enforce MX, method `lock()` needs to be invoked on the lock object at the beginning of the CS, and `unlock()` at the end.

When the first thread executes `lock()`, the lock’s state changes from Unlocked to Locked. If other threads try to execute `lock()` in state Locked, these threads get blocked until the first thread executes `unlock()`, which changes the lock’s state to Unlocked and which allows the blocked threads to continue their locking attempts; of course, only one of them will be successful.

The question whether the lock really changes its state from Locked to Unlocked upon `unlock()` or whether it is immediately reassigned to one of the blocked threads (e.g., in FIFO order) is a design decision, which will be revisited later in the context of the so-called convoy problem.

4.3 Use of Locks/Mutexes

- Programming discipline **required** to prevent races
 - Create one (shared) lock for each shared data structure
 - Take lock before operating on shared data structure
 - Release lock afterwards
- See [Hai17] for sample code following POSIX standard
- Ticket example modified (leading to MX behavior):

```

seatlock.lock();
if (seatsRemaining > 0) {
    dispenseTicket();
    seatsRemaining = seatsRemaining - 1;
} else displaySorrySoldOut();
seatlock.unlock();

```

5 Semaphores

5.1 Semaphore Origin

- Proposed by Dijkstra, 1965
 - Based on **waiting** (sleeping) for **signals** (wake-up calls)

- * Thread waiting for signal is **blocked**

- **Abstract data type**

- Non-negative integer
 - * Number of available resources; 1 for MX on CPU
- Queue for blocked threads
- **Atomic** operations
 - * Initialize integer
 - * **acquire** (wait, sleep, down, P [passeren, proberen]): entry into CS
 - Decrement integer by 1
 - As integer must be non-negative, **block** when 0
 - * **release** (signal, wakeup, up, V [vrijgeven, verlaten]): exit from CS
 - Increment integer by 1 (value may grow without bound)
 - Potentially **unblock** blocked thread

5.2 Use of Semaphores for MX

- Programming discipline **required** similarly to locks
 - Create semaphore for shared data structure
 - **acquire()** before CS, **release()** after
- Ticket example modified with `seatSem` initialized to 1 (leading to MX behavior):
 - (The semaphore initialized to 1 behaves exactly like a lock here)

```
seatSem.acquire();
if (seatsRemaining > 0) {
    dispenseTicket();
    seatsRemaining = seatsRemaining - 1;
} else displaySorrySoldOut();
seatSem.release();
```

5.2.1 Semaphores for Capacity Control

- Semaphores initialized to other values than 1 are typically used to model capacities
- Example from [stack overflow](#): [bouncer in nightclub](#)
 - Nightclub has limited capacity, i.e., number of people allowed in club at any moment: n
 - * Bouncer (semaphore initialized to n) makes sure that no more than n people can be inside
 - * If club is full, a queue collects waiting people
 - Guests (threads) call **acquire()** on bouncer/semaphore to enter
 - Guests (threads) call **release()** on bouncer/semaphore to leave
- Different example in in-class session: `SemaphoreBoundedBuffer`

5.3 JiTT Assignments

5.3.1 The Deadlock Empire – Remarks

- The games at <https://deadlockempire.github.io/> make use of C#
- What is called “monitor” in C# is **not** a Hoare style monitor to be discussed in the upcoming presentation
 - In C#, a “monitor” needs to be (un)locked explicitly
 - Whereas Hoare style monitors are locked implicitly and automatically
- Class context
 - Hoare style monitors in Java
 - Producer/consumer scenarios mentioned in games
 - * Classical synchronization problems (this presentation)
 - * Revisited as `BoundedBuffer` in Java

5.3.2 The Deadlock Empire, Part 2

Play the following games at <https://deadlockempire.github.io/>

- “Locks” and “Semaphores”
 - Incorrect use of both, sometimes leading to deadlocks. For locks, `Enter()` and `Exit()` represent `lock()` and `unlock()`
- “Condition Variables”
 - Here, `if (queue.Count == 0)` is meant to avoid removal attempts from empty queues. However, `PulseAll()` wakes up all waiting (blocked) threads (similarly to `notifyAll()` for Java later on)

5.3.3 Search for Race Conditions

- Submit solutions for the following tasks in Learnweb.
 1. Explain a concurrent execution of the following two threads (starting with the shared variables `one` and `two` initialized to 0) such that Thread 1 executes the statement `Debug.Assert(false)`.
 2. How could you prevent such race conditions with locks or semaphores? Where would you add what statements?

Thread 1	Thread 2
<code>business_logic();</code>	<code>business_logic();</code>
<code>one++;</code>	<code>one++;</code>
<code>two++;</code>	<code>two++;</code>
<code>if (two == 2 && one != 2)</code>	
<code> Debug.Assert(false);</code>	

5.3.4 Questions, Feedback, and Improvements

- 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 Busy Waiting vs Blocking

6.1.1 Atomic Instructions

- Typical building block for MX: **Atomic** machine instruction
 - Several memory accesses with guarantee of isolation/no interference
- E.g., `exchange`, which exchanges contents of register and memory location

6.1.2 Mutex with Simplistic Spinlock Implementation

- Single memory location called `mutex`
 - Value 1: unlocked
 - Value 0: locked
- Operations
 - `unlock()`: Store 1 into `mutex`
 - `lock()`: Atomically check for 1 and store 0 as follows ([Hai17]):

```
to lock mutex:
let temp = 0
repeat
  atomically exchange temp and mutex
until temp = 1
```

See [Hai17] for a cache-conscious spinlock variant (beyond scope of class).

6.1.3 On Spinlocks

- **Spinlock**: Thread spins **actively** in loop on CPU while waiting for lock to be released
 - **Busy waiting**

- Note: Spinning thread keeps CPU core busy
 - No blocking by OS
 - **Waste** of CPU resources unless lock periods are short

6.1.4 Mutex with Queue

- Mutex as OS mechanism
 - When `lock()` fails on a locked mutex, OS **blocks** thread
 - Blocked threads are collected in queue
 - **No** busy waiting
 - * Thus, CPU time not wasted for long waiting periods
 - * However, scheduling with its own **overhead** required
 - Wasteful, if waiting periods are short
- Different variants of `unlock()`
 1. Unblock thread in FIFO order from queue (if one exists)
 - And reassign mutex to that thread
 2. Make all threads runnable without reassigning mutex
 - Upcoming presentation: [convoy problem](#) and its solution
- Pseudocode in [Hai17]

6.2 Producer/Consumer problems

- Classical synchronization problems
 - One or more **producers**
 - * Generate data
 - 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
- Synchronization for **buffer manipulations** necessary

6.3 Use of Semaphores to Track Resources

```
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();
    }
}
```

}

6.4 GNU/Linux: Futex

- Fast user space mutex
 - No system call for single user (fastpath)
 - System calls for blocking/waiting (slowpath)
- Meant as building block for libraries
- Like semaphore: Integer with `up()` and `down()`
 - Assembler code with atomic instructions for integer access
- Documentation
 - `man futex`
 - `PI-futex.txt`
 - * (Topic for upcoming presentation: PI stands for priority inheritance, a counter-measure against priority inversion)

6.5 Lock-free Data Structures

- Core idea: Handle critical sections without locks
- Typically based on hardware support for atomicity guarantees
 - Atomic instructions as explained above
 - * E.g., Bw-Tree, see [LLS13]
 - Transactional memory, see [LK08]
- See Section 4.9 of [Hai17]

7 Conclusions

7.1 Summary

- Parallelism is a fact of life
 - Multi-core, multi-threaded programming
 - Race conditions arise
 - Synchronization is necessary
- Mutual exclusion for critical section
 - Locking
 - Monitors
 - Semaphores

Bibliography

- [Hai17] Max Hailperin. *Operating Systems and Middleware – Supporting Controlled Interaction*. revised edition 1.3, 2017. URL: <https://gustavus.edu/mcs/max/os-book/>.
- [LK08] James Larus and Christos Kozyrakis. “Transactional Memory”. In: *CACM* 51.7 (July 2008), pp. 80–88. ISSN: 0001-0782. DOI: 10.1145/1364782.1364800. URL: <https://dl.acm.org/citation.cfm?doid=1364782.1364800>.
- [LLS13] Justin Levandoski, David Lomet, and Sudipta Sengupta. “The Bw-Tree: A B-tree for New Hardware Platforms”. In: *ICDE 2013*. IEEE, Apr. 2013. URL: <https://www.microsoft.com/en-us/research/publication/the-bw-tree-a-b-tree-for-new-hardware/>.

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, this work, “OS05: Mutual Exclusion”, is © 2017, 2018, 2019 by Jens Lechtenbörger, 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.