Mutual Exclusion *

Jens Lechtenbörger

IT Systems, Summer Term 2024

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

1 Introduction

Let us look at essential questions and terminology of our topic.

1.1 Core Questions

- What can go wrong with concurrent computations as result of race conditions?
- What mechanisms for Mutual Exclusion (MX) does the OS provide to help?

(Based on Chapter 4 of (Hailperin 2019))
This set of presentations addresses the two major questions:
First, what can go wrong with concurrent computations as result of race conditions?
You will see that subtle programming bugs may be related to timing issues when accessing shared resources.
Based on this observation, a second question arises, namely:
What mechanisms for Mutual Exclusion does the OS provide to help?
Briefly, locking mechanisms that you know from database systems are a special case of mechanisms for mutual exclusion, which guarantee that accesses to shared resources proceed in an orderly fashion with correct results.

1.2 Learning Objectives

- Recognize and explain race conditions
  - (Playing Deadlock Empire helps)
- Explain notions of critical section and mutual exclusion
  - Use mutexes (and monitors) to enforce mutual exclusion
  - Apply and explain MX and cooperation based on monitor concept in Java
- Explain and apply deadlock prevention and detection
- Explain starvation as MX challenge

Take some time to think about the learning objectives specified here.

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

1.3 Retrieval practice

Let us see what prior knowledge is involved subsequently.

1.3.1 Informatik 1

- 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`

You have seen these questions before. They are repeated here for emphasis.

1.3.2 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.

Recall update anomalies and the transaction concept as countermeasure in database systems.

Agenda

- 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

The agenda for this presentation is as follows.

After this introduction, we investigate race conditions as core challenge of concurrent executions, which can be addressed with mutual exclusion for critical sections. Importantly, we look at locking as popular mechanism to guarantee mutual exclusion. The first part ends with pointers beyond class topics, which hint at the variety of related topics beyond the scope of our course.

In the second part, we introduce the monitor concept and its use in Java to guarantee mutual exclusion beyond locking, also enhancing the cooperation of threads on shared data structures.

In the third part, we explore challenges related to mutual exclusion, in particular deadlocks and starvation.

2 Race Condition

- **Race (condition)**: a technical term
  - Concurrent **activities** on **shared resources**
    - At least one activity writes

    "Race condition", or "race" for short, is a technical term in OSs: When multiple threads access a shared resource, e.g., a shared variable, in a concurrent fashion and at least one of the threads modifies the resource, then phenomena such as dirty reads or lost updates may arise.

  - Overall outcome depends on **subtle timing** differences
    - Missing synchronization, "nondeterminism", **bug**

  Crucially, negative effects of race conditions depend on subtle timing issues regarding the current execution of multiple threads, and they may not occur under "normal" testing. In any case, the occurrence of races indicates a lack of proper synchronization among the concurrent threads.

  In this case, a program may appear to produce nondeterministic outcomes, which is just a programming bug.

- Analogy

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

  - **Cars** are activities
  - **Street segments** represent shared resources
  - Timing determines whether a **crash** occurs or not
  - Crash = misjudgment = missing synchronization

Here, you see an analogy from car races, where cars share street segments as resources, and drivers need to time their actions properly to avoid crashes.

2.1 Examples for Races Conditions

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

Race conditions may occur in database systems where SQL commands are activities, database objects are shared resources, and update anomalies are bugs to be avoided.

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

In OSs, we consider threads as activities, which access shared resources such as variables, memory regions, or files. Here, programmers are responsible for proper synchronization, and programs lacking proper synchronization are buggy, leading to anomalies just as in database systems.

2.2 Self-Study Tasks

Take a break for self-study.

2.2.1 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 **mutual exclusion** and **critical sections**, to be discussed next
    - At any point in time just **one** thread is allowed to execute under mutual exclusion inside a critical section

        * If you manage to lead two threads into a critical section simultaneously (or, in some levels, to execute `Assert(false)`), you demonstrate a race condition
    - You **win** a game if you demonstrate a race condition

If you did not play Deadlock Empire yet, do so now.

The game is about mutual exclusion and critical sections, to be discussed in detail next. For now, at any point in time just one thread is allowed to execute under mutual exclusion inside a critical section. If you manage to lead two threads into a critical section simultaneously (or, in some levels, to execute `Assert(false)`), you demonstrate a race condition and win the level.

- Really, start playing **now**! (Nothing to submit here)

Did you play?

2.2.2 Transfer of Deadlock Empire

Consider the following piece of Java code (from Sec. 4.2 of (Hailperin 2019)) 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 race conditions involving the counter `seatsRemaining`, which leads to "too many" tickets being sold.

Work on the task given here.

2.3 Non-Atomic Executions

- Races generally result from non-atomic executions

  Race conditions generally result from non-atomic executions. In other words, they arise from context switches between instructions which programmers meant to be executed atomically as one unit.

  Note that the word "atomic" is used in its literal sense here: An execution is **not** atomic if is really consists of multiple steps.

  - 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

    Even simple statements of high-level programming languages are not executed atomically, which may be the source of race conditions.

    As you saw in Deadlock Empire and as you know from programming in assembly language, "simple" increment statements really consist of multiple machine instructions.

  - 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 transactions: races, dirty reads, lost updates
    * Demo: Play a game as instructed previously

A context switch may happen after any of these machine instructions, i.e., "in the middle" of a high-level instruction. Then, intermediate results may become accessible elsewhere.

In database systems, resulting phenomena such as dirty reads and lost updates indicate a lack of isolation.

In programming, Deadlock Empire demonstrates negative effects.

3 Critical Sections and Mutual Exclusion

Critical sections and mutual exclusion are key concepts to address race conditions in OSs.

3.1 Goal and Solutions (1/3)

- Goal
  - Concurrent executions that access **shared resources** should be **isolated** from one another
    * Cf. **I** in ACID transactions

Our goal is to isolate concurrent executions on shared resources from each other.

- Conceptual solution
  - Identify **critical sections** (CSs)
    * CS = Block of code with potential for race conditions on shared resources
      · Cf. transaction as sequence of operations on shared data
  - Enforce **mutual exclusion** (MX) on CSs
    * At most one thread inside CS at any point in time
      · This avoids race conditions
      · Cf. serializability for transactions

To reach this goal, we first identify parts of the code as critical sections. Afterwards, we enforce mutual exclusion on those parts of code.

More precisely, a piece of code is a critical section if it accesses shared resources with the potential for race conditions.

Mutual exclusion is given if at most one thread executes inside a critical section at any point in time. In that case, race conditions are avoided.

3.2 MX with CSs: Ticket example

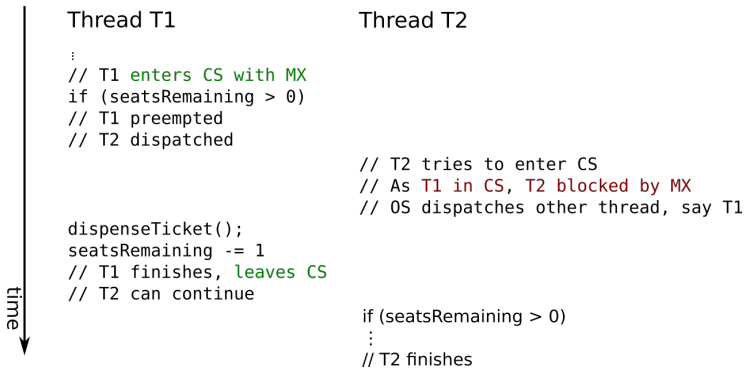

Figure 2: "Interleaved execution of threads with MX for code from Sec. 4.2 of book by Max Hailperin, CC BY-SA 3.0." by Jens Lechtenbörger under CC BY-SA 4.0; from GitLab

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 self-study assignment for that code fragment yet, please do so now and come back afterwards. Actually, this animation may also help you to solve the 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 or monitors is the topic of later slides.

Thread 1 enters the CS first. Suppose its time slice runs out after the check that seats are still remaining.

Now the OS dispatches Thread 2, which wants to execute the same CS. However, as Thread 1 is currently executing inside that CS and as MX is enforced for that CS, Thread 2 is blocked by the OS. Thus, the OS schedules another thread for execution, eventually Thread 1.

Consequently, Thread 1 can finish the CS without intermediate modifications by any other thread.

Afterwards, Thread 2 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 avoid races; on the other, they inhibit parallelism, which is generally bad for performance. Thus, CSs should be as short as possible.

3.3 Goal and Solutions (2/3)

- Refined goal
  - Implementations/mechanisms for MX on CS
- Solutions, to be discussed in detail
  - **Locks**, also called **mutexes**
    - Cf. 2PL for database transactions
    - Acquire lock/mutex at start of CS, release it at end
      - Exclusive "ownership" by one thread at a time
  - **Monitors**
    - Abstract data type, think of Java class, methods as CS with MX

    - Keyword `synchronized` turns Java method into CS with MX!
  - (**Semaphores**, beyond class)
    - Abstract data type, generalization of locks, blocking, signaling

To refine our goal, we need implementations or mechanisms that enforce MX for CS.

Subsequently, we look into locking and monitors as MX mechanisms. A lock is a data structure that can be acquired by exactly one thread at a time at the start of a critical section, to be released at its end.

Monitors are abstract data types that offer richer functionalities beyond pure mutual exclusion. In essence, a monitor is like a class, and its methods are automatically executed under MX. In particular, you will see that Java implements the monitor concept with the keyword `synchronized`.

Another popular MX mechanism is offered by semaphores, which we do not cover here.

3.4 Challenges

- Major synchronization challenges arise
  - Programming errors, e.g., races involving `seatsRemaining`
    - Difficult to locate and reproduce; time-dependent, "non-deterministic"

While MX mechanisms avoid race conditions, their proper use requires some care. Importantly, programmers need to be aware where races may arise, to apply appropriate MX mechanism. As mentioned, races depend on timing issues, which may make them difficult to locate and reproduce.

  - Above solutions restrict entry to CS
    - Thus, they restrict access to the resource CPU

      Moreover, MX mechanisms restrict access into critical sections, which implies that they may block threads, effectively denying access to the resource CPU.
    - **Deadlock**
      - Set of threads is stuck
      - Circular wait for additional locks/resources/messages
      - Definition, detection, prevention in later presentation

      Without some care, multiple threads may block each other in such a way that all of them are stuck in a circular wait condition. This situation is called deadlock and requires proper counter measures such as detection or prevention techniques.
    - **Starvation** (related to (un-) fairness)
      - Thread never enters CS
      - (More generally: never receives some resource, e.g., CPU under scheduling)
      - (Revisited in later presentation)

      Moreover, with MX, some threads may never be allowed to enter a critical section, which is another example of starvation.

3.5 Goal and Solutions (3/3)

- Avoid race conditions with MX
  - Recall above loose definition

        * MX = At most one thread inside CS at any point in time

To sum up our goal, we strive to avoid race conditions, and we do so with MX mechanisms which guarantee that at most one thread executes inside a critical section at any point in time.

- Stricter definitions also address deadlocks, starvation, failures
  - Our **definition**: Solution **ensures MX** if
    * At most one thread inside CS at any point in time
    * Deadlocks are ruled out
  - (Not our focus: Starvation does not occur)
    * (E.g., requests granted under fairness guarantees such as first-come first-serve or with "luck" based on randomness)
  - ((Lamport 1986) provides detailed discussion, also addressing failures)

In fact, we aim for mechanisms that not only guarantee mutual exclusion but that also rule out deadlocks.

In other literature, you will also find that MX mechanisms should rule out starvation, which may happen with fairness guarantees or based on randomness. We do not consider these issues.

4 Locking

Let us turn to 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.)

A "mutex" is an object that can guarantee mutual exclusion on shared resources. Here, you see an integer variable and an array as examples for shared resources.

Essentially, the mutex can be taken, or "owned", by only one thread. The owning thread then proceeds to work on the shared resource. When the thread finished its work, it makes the mutex available for other threads. That way, only one thread at a time works on the shared resource.

The final part of the drawing shows some terminology, which is beyond the scope of the drawing and which will also only be mentioned in passing subsequently.

4.2 Locks and Mutexes

- Lock = mutex
  - Object with **atomic** methods

    First, 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 is shown in the figure here, along with possible state transitions when the lock's methods `lock()` and `unlock()` are invoked as explained subsequently.

    Importantly, the methods of locks are executed under atomicity guarantees. Thus, if multiple threads attempt to execute such a method at the same time, the implementation must ensure that exactly one thread executes the method without interference by others.

Beyond our topics, implementations usually make use of atomic machine instructions such as "compare and swap" mentioned in the previous drawing. Such machine instructions perform more complex functionality, e.g., the comparison of some values followed by swapping of values, but the computer architecture makes sure that no race conditions arise when multiple threads execute them concurrently. By clever use of such atomic building blocks, atomic methods can be implemented.

* `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()` are blocked until `unlock()`
* `unlock()` or `release()`: Unlock/release the object
  - Can be `lock()`'ed again afterwards

The methods of locks can have various names such as `lock`, `acquire`, or `take`, and `unlock` or `release`.

Initially, a lock is Unlocked. Only one thread will be able to perform `lock()` successfully, causing the lock's state to change to Locked. We also say that the locking thread takes or owns the lock. Other threads invoking `lock()` will then be blocked until the first thread performs `unlock()`, changing the lock's state to Unlocked. Of course, in state Unlocked, again only one thread will be successful in a `lock()` attempt.

- Use in your own code (next slide)
  - E.g., interface java.util.concurrent.locks.Lock in Java.

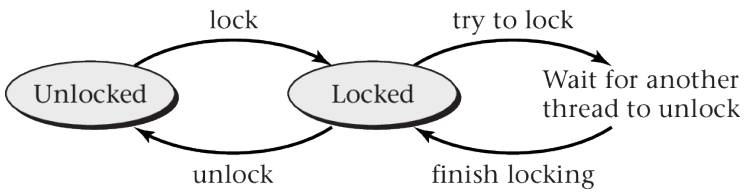

Figure 3: "Figure 4.4 of cite:Hai17" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

When MX is necessary to prevent races for a critical section, a lock object shared by the racing threads can be used as shown on the next slide.

4.3 Use of Locks/Mutexes

- Programming discipline **required** to prevent races
  1. Create one (shared) lock for each shared data structure
  2. Take lock before operating on shared data structure
  3. Release lock afterwards
- ((Hailperin 2019) has 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();
```

As explained here, programmers can prevent races in critical sections for shared data structures with locks, which requires programming discipline:

First, create a shared lock for the data structure. Second, before using the data structure, take the lock. Third, when done with the data structure, release the lock again.

The sample code here shows how the shared variable `seatsRemaining` can be protected with a lock. Note how `lock()` and `unlock` surround the entire `if` statement as critical section.

4.4 Quiz on MX Vocabulary

Take this quiz.

5 Pointers beyond class topics

This section offers pointers beyond class topics.

5.1 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
- 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)

The Linux kernel provides a mutex variant called futex, short for fast user space mutex. The futex is also mentioned in a previous drawing. The futex builds upon atomic machine instructions and is fast for the first thread, while kernel interaction with context switches is only necessary when a thread needs to be blocked. Documentation is provided by a manual page and under the web page linked here.

Notably, the futex offers priority inheritance, which is a counter measure against priority inversion. Shortly, priority inversion is a bad situation when using MX with threads of different priorities, where your high-priority threads may not work as expected when they need to wait for resources owned by low-priority threads.

When you program threads of different priorities that share resources, you **must** follow the links provided here.

5.2 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 (Levandoski, Lomet, and Sengupta 2013)
    - See also Section 4.9 of (Hailperin 2019)
  - Transactional memory, see (Larus and Kozyrakis 2008)

For high-performance environments, lock-free data structures exist. These enable concurrent access without blocking of threads. For example, one paper cited here describes a lock-free tree structure that is used in the database system Microsoft SQL Server.

Besides, "transactional memory attempts to simplify concurrent programming by allowing a group of load and store instructions to execute in an atomic way. [...] Transactional memory systems provide high-level abstraction as an alternative to low-level thread synchronization."

5.3 "Safer" Programming Languages

- High-level programming languages may help with MX
- See (Jung et al. 2021) for introduction to Rust
  - Strong type system for detection of common bugs at **compile** time
    - Thread safety (absence of race conditions) for shared data structures with compile-time checks
  - Ongoing research into safety proofs
- (Besides, the OS Redox is implemented in Rust)

Different programming languages provide different concepts to counter race conditions. For example, the language Rust comes with a strong type system, where the compiler can detect common bugs. In particular, it provides thread safety for shared data structures with compile-time checks.

5.4 Massively Parallel Programming

- For massively parallel (big data) processing in clusters or cloud environments, specialized frameworks exist
  - Breaking down "large" tasks with partitioning into smaller ones that are processed in parallel
    - Smaller tasks usually independent of each other (no race conditions)
    - (Built-in fault tolerance with replication)
  - E.g., Apache Hadoop (MapReduce), Apache Spark, Apache Flink

For massively parallel processing in clusters or cloud environments, specialized frameworks exist. They make use of partitioning to break down "large" tasks into smaller ones that are processed in parallel.

The slide contains hyperlinks for popular frameworks.

Bibliography

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

Jung, Ralf, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. "Safe Systems Programming in Rust." *Commun. Acm* 64 (4): 144–52. https://doi.org/10.1145/3418295.

Lamport, Leslie. 1986. "The Mutual Exclusion Problem: Part II—Statement and Solutions." *J. Acm* 33 (2): 327–48. https://doi.org/10.1145/5383.5385.

Larus, James, and Christos Kozyrakis. 2008. "Transactional Memory." *Cacm* 51 (7): 80–88. https://doi.org/10.1145/1364782.1364800.

Levandoski, Justin, David Lomet, and Sudipta Sengupta. 2013. "The Bw-Tree: A B-Tree for New Hardware Platforms." In *Icde 2013*. IEEE. https://www.microsoft.com/en-us/research/publication/the-bw-tree-a-b-tree-for-new-hardware/.

The bibliography contains references used in this presentation.

License Information