MX Challenges [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 third 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

Now that we know general concepts for MX and their use in Java, let us investigate arising challenges.
The major part of this presentation concerns deadlocks, followed by further challenges, in particular starvation.

2 Deadlocks

- Did you play level "Deadlock" at https://deadlockempire.github.io/?

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

- There, you lead two threads into a deadlock...

Let us investigate deadlocks.
Previously, you were instructed to experience a deadlock in the Deadlock Empire. Recall this task.

[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.

2.1 Deadlock

- Permanent blocking of thread set

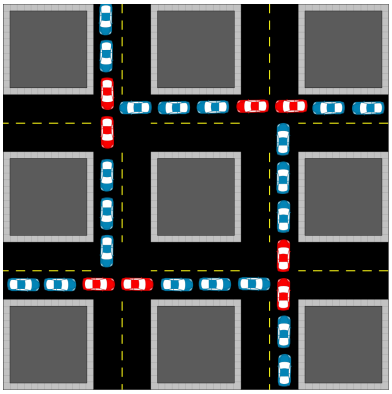

Figure 2: "Gridlock" by Interiot~commonswiki and Jeanacoa under CC BY-SA 2.5 Generic; from Wikimedia Commons

- Reason
  - **Cyclic waiting** for resources/locks/messages of other threads
  - (Formal definition on later slide)

A deadlock is a programming bug, which leads to multiple threads being stuck: In essence, the threads mutually wait for something from other threads which never arrives.

To get a feeling for deadlocks, note that some traffic situations can be interpreted as deadlocks. First, the image here shows a traffic situation where no car can move because other cars (namely, the red ones) block required street segments. In OS terms, the cars can be interpreted as threads, which are stuck, while street segments represent shared resources under MX that are exclusively owned by some threads while others also need them. This is an instance of cyclic waiting.

As a different example, consider priority to the right and a street crossing where four cars arrive from all four directions. Under "priority to the right", each driver needs to wait for another car to move first. Thus, neither can move, all are stuck.

- No generally accepted solution
  - Deadlocks can be perceived as programming bugs
    - Dealing with deadlocks causes overhead
      - Acceptable to deal with (hopefully rare) bugs?
  - Solutions depend on
    - Properties of resources (e.g., linearly ordered ones)
    - Properties of threads (transactions?)

In programming, we aim to avoid problematic algorithms or rules such a "priority to the right" so that deadlocks do not occur. However, there is no generally accepted solution, and you need to be particularly careful when using MX mechanisms.

As you will see, OSs typically ignore deadlocks, which is justified by the reasoning that programmers should avoid this type of bug; thus, there is no need to add additional complexity and overhead to the OS. Moreover, solutions also depend on the type of resources, e.g., you will see a strategy for linearly ordered resources later on.

As a side note, database systems may involve deadlock detection for transactions, which can be aborted to undo their effects, while this is less simple for threads in OSs. Thus, thread properties also play a role in deadlock considerations.

2.2 Deadlock Example

- Money transfers between bank accounts
  - Transfer from `myAccount` to `yourAccount` by thread 1; transfer in other direction by thread 2

This slide introduces deadlocks based on a programming example with MX.
Consider multiple transfers of money between bank accounts, where each transfer is managed by a separate thread.

- **Race conditions** on account balances
- Need **mutex** per account
  - Lock both accounts involved in transfer. What order?

Clearly, account balances are shared resources, on which race conditions may arise, e.g., leading to lost updates.
Thus, MX mechanisms are necessary, say locking. In what order should each thread lock its two accounts?

- "Natural" lock order: First, lock source account; then, lock destination account
  - Thread 1 locks `myAccount`, while thread 2 locks `yourAccount`
    - Each thread gets blocked once it attempts to acquire the second lock
      - Neither can continue
    - **Deadlock**

If the programmer implements a natural lock order where the lock for the source account is acquired first, followed by the lock for the target account, then two transfers in opposite directions can lead to a deadlock as stated here:
Each thread obtains just the lock for its source account, which is the target account for the other account. Thus, both threads are blocked when attempting to acquire their second lock.

2.3 Defining Conditions for Deadlocks

Deadlock if and only if (1) – (4) hold (Coffman, Elphick, and Shoshani 1971):

1. **Mutual exclusion**
   - Exclusive resource usage
2. **Hold and wait**
   - Threads hold some resources while waiting for others
3. **No preemption**
   - OS does not forcibly remove allocated resources
4. **Circular wait**
   - Circular chain of threads such that each thread holds resources that are requested by next thread in chain

As stated on this slide, a deadlock exists precisely when four conditions hold:
First, resources are used under MX.
Second, the threads involved in the deadlock hold some resources while waiting for others.
Third, the OS does not forcibly remove allocated resources.
Fourth, the threads can be arranged in a circular chain such that each thread holds resources for which the next thread in the chain waits.
In the deadlock example about money transfers above, we have two threads and two accounts protected by locks. The four conditions hold as follows: First, accounts are locked exclusively.
Second, each thread holds one lock while waiting for a second lock.
Third, locks are never removed by the OS. Instead, threads may release them.
Fourth, both threads wait for each other.

2.4 Resource Allocation Graphs

- Representation and visualization of resource allocation as directed graph
  - (Necessary prior knowledge: directed graphs and cycles)
  - Nodes

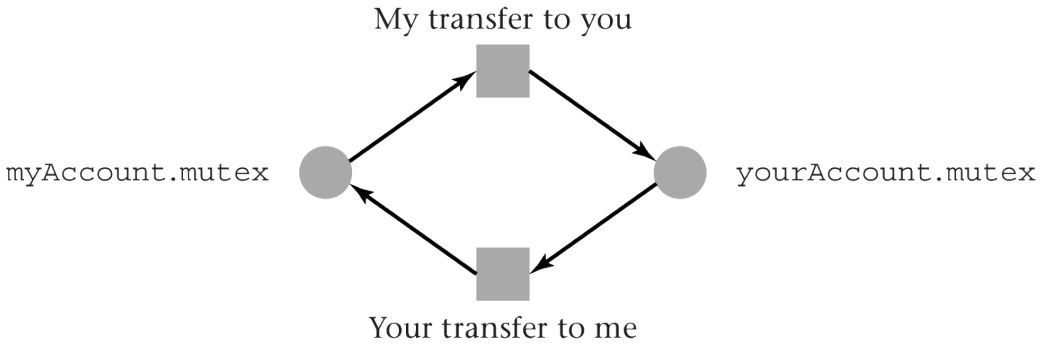

Figure 3: "Figure 4.22 of (Hailperin 2019)" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

  - Threads (squares)
  - Resources (circles)
  - (Choice of shapes is arbitrary, just for visualization purposes)
- Edges
  - From thread T to resource R if T is waiting for R
  - From resource R to thread T if R is allocated to T
- **Fact**: System in deadlock if and only if graph contains cycle

A resource allocation graph is a directed graph that visualizes the current state of resource allocations and of outstanding requests by threads. Threads and resources are nodes in this graph, and there are two types of edges:

First, there is an edge from a thread to a resource if the thread is waiting for the resource (because the resource is currently allocated to another thread).

Second, there is an edge from a resource to a thread if the resource is allocated to that thread.

Importantly, a cycle in this graph represents the circular wait condition of a deadlock. Thus, if a cycle exists, a deadlock exists. All threads on that cycle are involved in this deadlock.

Here you see a resource allocation graph for the previous deadlock example involving bank transfers.

3 Deadlock Strategies

- (Ostrich "Algorithm")
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection

Different deadlock strategies are employed in different types of systems. We consider the four strategies shown here.

3.1 Ostrich "Algorithm"

- A joke about missing deadlock handling
  - "Implemented" in most systems
    - Pretend nothing special is happening
    - (E.g., Java acts like ostrich)
  - Reasoning
    - Proper deadlock handling is complex
    - Deadlocks are rare, result from buggy programs

Figure 4: Drawing created by Adrian Lison for bonus task in summer term 2017; released into Public Domain; other excellent drawings.

The so-called ostrich "algorithm" is visualized here, in different variants if you refresh the HTML page.

This is not an algorithm but a joke about missing deadlock handling in most OSs. They do not care about deadlocks, but pretend that nothing bad is happening, keeping the blocked threads in a deadlock forever.

The reason to use this strategy is that proper deadlock handling is a complex topic, while deadlocks in OSs result from buggy programs. Thus, programmers should fix their programs, instead of adding the overhead for proper handling of deadlocks to OSs.

3.2 Deadlock Prevention

- Prevent a defining condition for deadlocks from becoming true
- Practical options

  A strategy for deadlocks is called prevention strategy if it prevents deadlocks from happening by making sure that one of the four defining deadlock conditions can never become true. Although there are four conditions, only two of them are used for practical purposes, and they are explained in the subsequent bullet points.

  - Prevent condition (2), "hold and wait": Request all necessary resources at once
    - Only possible in special cases, e.g., conservative/static 2PL in DBMS
    - Threads either have no incoming or no outgoing edges in resource allocation graph → Cycles cannot occur

    Some systems may prevent the hold-and-wait-condition from becoming true. One example are database systems with variants of the two-phase locking protocol, where transactions either acquire all resources they need, or none at all. Clearly, such transactions either hold or wait, preventing cycles in resource allocation graphs.

  - Prevent condition (4), "circular wait": Number resources, request resources according to **linear resource ordering**
    - Consider resources R_h and R_k with h $<$ k
      - Threads that need both resources must lock R_h first
      - Threads that already requested R_k do not request R_h afterwards
    - Requests for resources in ascending order → Cycles cannot occur

    For programmers, preventing the circular-wait-condition is a usual choice. This is possible if resources can be numbered linearly. If a thread needs multiple resources, it acquires them according to their linear order.

    Then, edges in resource allocation graphs can never "go back", which would be necessary to close cycles.

3.2.1 Linear Resource Ordering Example

- Money transfers between bank accounts revisited
- Locks acquired in **order of account numbers**
  - A programming contract, not known by OS
  - Suppose `myAccount` has number 42, `yourAccount` is 4711
    - Both threads try to lock `myAccount` first (as 42 $<$ 4711)
      - Only one succeeds, can also lock `yourAccount`
      - The other thread gets blocked
  - **No deadlock**
- (See Fig 4.21 in (Hailperin 2019) for an example of linear ordering in the context of the Linux scheduler)

In the context of transfers among bank accounts, each thread could just lock the account with the smaller number first. Then, no deadlocks arise.

Note that this strategy just requires a programming contract, to be followed by **all** programmers using the shared resources. The OS can then use the ostrich "algorithm".

3.3 Deadlock Avoidance

- (See stackexchange for difference between prevent and avoid)

A strategy for deadlocks is called avoidance strategy if it avoids deadlocks. See the hyperlink here for the difference between the words "prevent" and "avoid" if you are not sure about their difference.

Avoidance does not rule out any specific of the four defining deadlock conditions, but it still makes sure that deadlocks do not happen.

- Dynamic decision whether allocation may lead to deadlock
  - If a deadlock cannot be ruled out easily: Do not perform that allocation but **block** the requesting thread (or return error code or raise exception)
  - Consequently, deadlocks do never occur; they are **avoided**

The typical approach is to analyze resource requests by threads. If some deadlock avoidance algorithm is able to rule out a deadlock for the resulting state, the request will be granted. If the algorithm is not able to rule out deadlocks, the request will not be granted. Note that such algorithms generally err on the safe side. Thus, some requests might not be granted, although they would not cause any deadlock; the OS might be unable to detect this, though.

- Classical technique
  - Banker's algorithm by Dijkstra
    - Deny incremental allocation if "unsafe" state would arise
    - Not used in practice
      - Resources and threads' requirements need to be declared ahead of time

A famous deadlock avoidance technique is Dijkstra's banker's algorithm, which has quite restrictive preconditions and is therefore not used in practice.

3.4 Deadlock Detection

- Idea
  - Let deadlocks happen
  - Detect deadlocks, e.g., via cycle-check on resource allocation graph
    - Periodically or
    - After "unreasonably long" waiting time for lock or
    - Immediately when thread tries to acquire a locked mutex
  - Resolve deadlocks: typically, terminate some thread(s)

The final strategy for dealing with deadlocks is deadlock detection. Here, the system does not take special precautions to avoid or prevent deadlocks but lets them happen. To deal with deadlocks, they are detected, for example based on cycle checks on resource allocation graphs, and then resolved. Detection may take place periodically or after waiting times or even immediately upon resource requests; the latter actually prevents cyclic wait conditions, moving from deadlock detection to deadlock prevention.

To resolve deadlocks, the OS typically terminates some threads until no cycle exists any longer, and various strategies exist to select victim threads, which is beyond our topics.

- Prerequisite to build graph
  - Mutex records by which thread it is locked (if any)
  - OS records for what mutex a thread is waiting

Clearly, the OS needs to build suitable data structures for deadlock detection, in case of resource allocation graphs, each mutex can easily record by which thread it is locked, while the OS also keeps track of what threads are waiting for what mutexes.

4 Further Challenges

Let us briefly look at additional challenges.

4.1 Starvation

- A thread **starves** if its resource requests are repeatedly denied
- Examples in previous presentations
  - Interrupt livelock
  - Thread with low priority in presence of high priority threads
  - Thread which cannot enter CS
    - Famous illustration: Dining philosophers (next slide)
    - No simple solutions

The term **starvation** occurred on several occasions already, where threads could not continue their execution as expected but were preempted or blocked frequently or for prolonged periods of time. When locking is involved, avoidance of starvation is a hard problem without simple solutions as illustrated next with the famous example of dining philosophers.

4.1.1 Dining Philosophers

- MX problem proposed by Dijkstra
- Philosophers sit in circle; **eat** and think repeatedly
  - Two **forks** required for eating
    - **MX** for forks

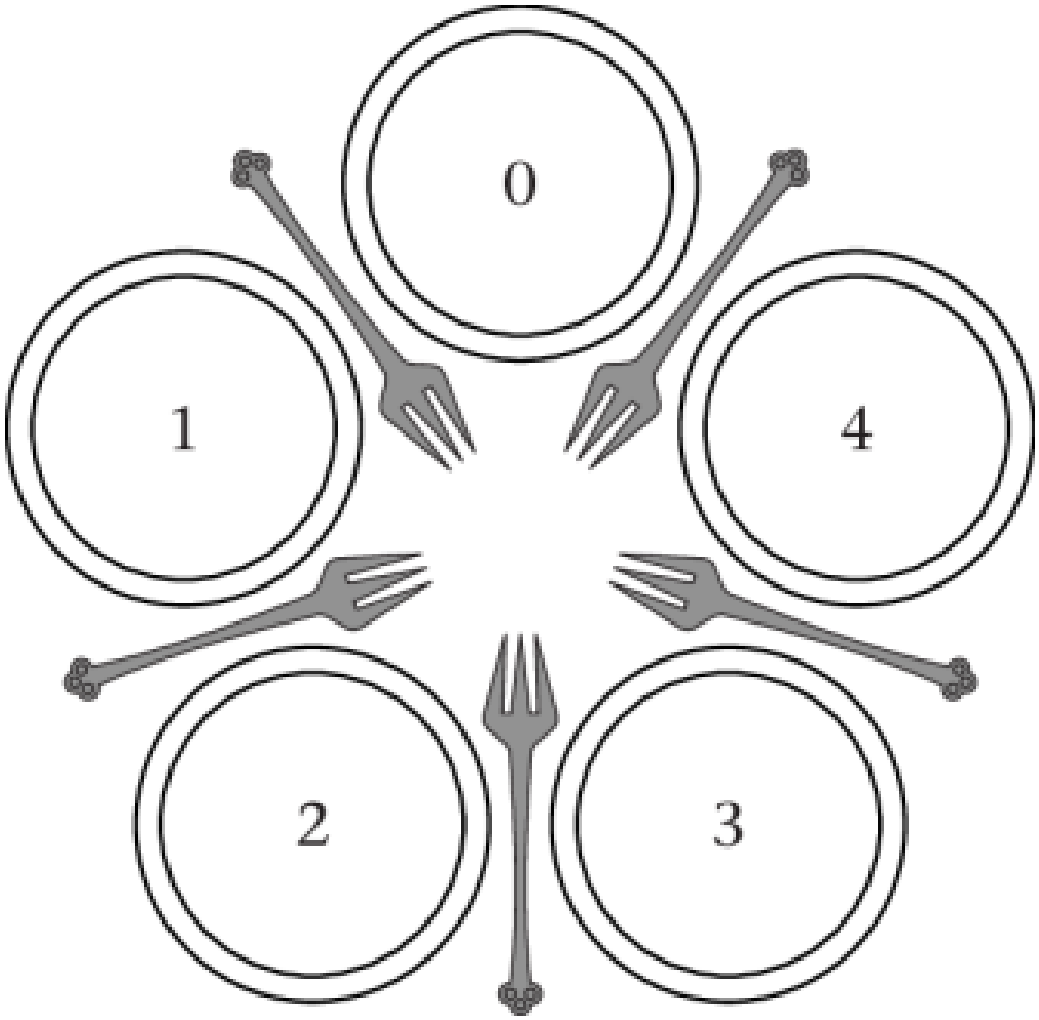

Figure 5: Dining Philosophers ("Figure 4.20 of (Hailperin 2019)" by Max Hailperin under CC BY-SA 3.0; converted from GitHub)

A famous illustration of starvation, which alludes to the literal meaning of the word, goes back to Dijkstra. Here, philosophers need forks to eat, and forks are protected by some MX mechanism. If the underlying algorithm to protect and reassign forks does not prevent starvation, one or more philosophers may die from hunger as they do not receive forks frequently enough.

Lots of textbooks on OS include algorithms for the dining philosophers to explain MX, deadlocks, and starvation. The next slide points out the difficulty of associated challenges.

4.1.2 Starving Philosophers

- Starvation of P0

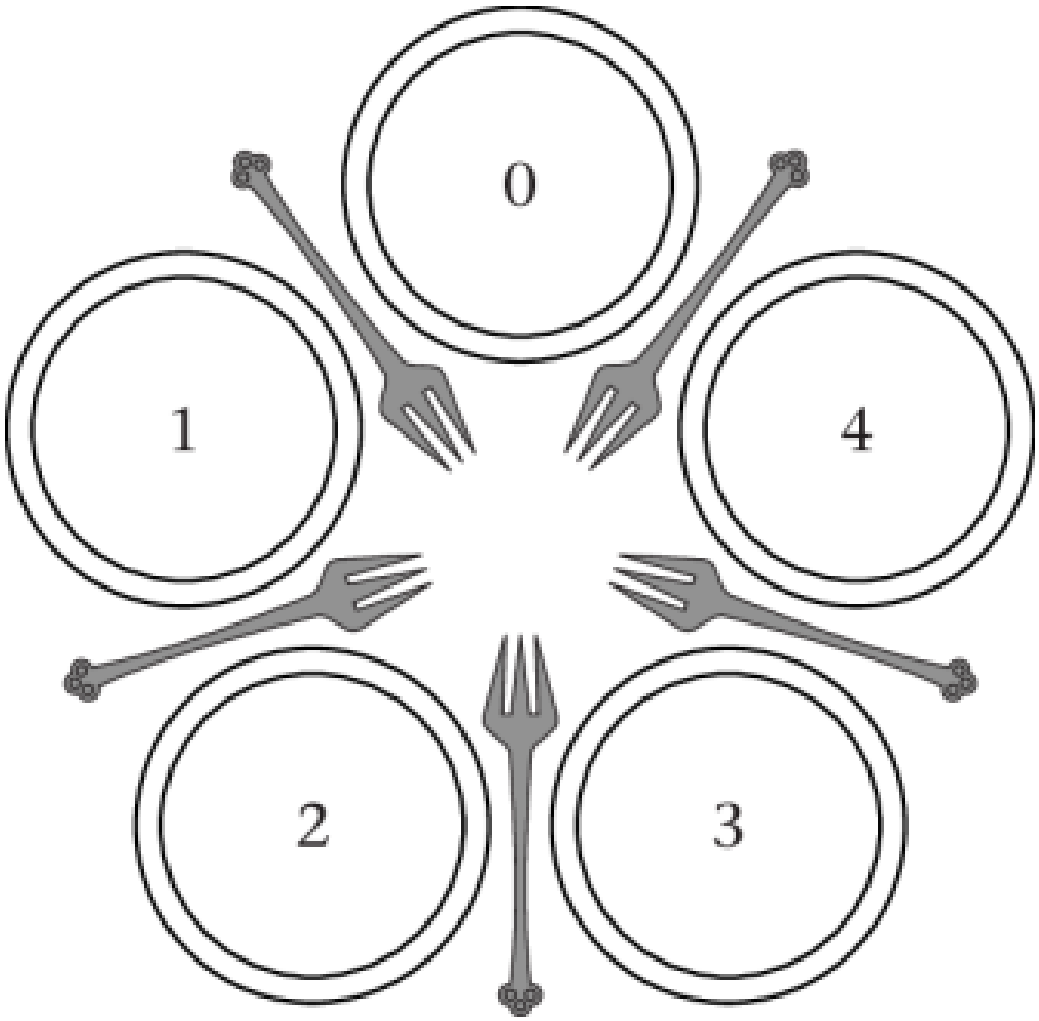

Figure 6: “Figure 4.20 of (Hailperin 2019)” by Max Hailperin under CC BY-SA 3.0; converted from GitHub

- P1 and P3 or P2 and P4 eat in parallel
- Then they wake the other pair
  - P1 wakes P2; P3 wakes P4
  - P2 wakes P1; P4 wakes P3
- Iterate
- (Above sequence possible for algorithm in (Tanenbaum 2001); inspired by exercise in (Stallings 2001))

The sequence of events shown here is possible for a textbook algorithm on the dining philosophers. In this case, philosopher 0 starves as the other four philosophers form stable pairs. The philosophers in these pairs wake up each other, but nobody ever wakes up philosopher 0.

Note that such cases of starvation do not necessarily arise for executions in practice. The point is that the textbook algorithm does not rule out such executions.

Apparently, addressing starvation is no simple task.

5 Conclusions

Let us conclude.

5.1 Priority Inversion Example

- Mars Pathfinder, 1997; Wikipedia offers details
  - Robotic spacecraft named Pathfinder

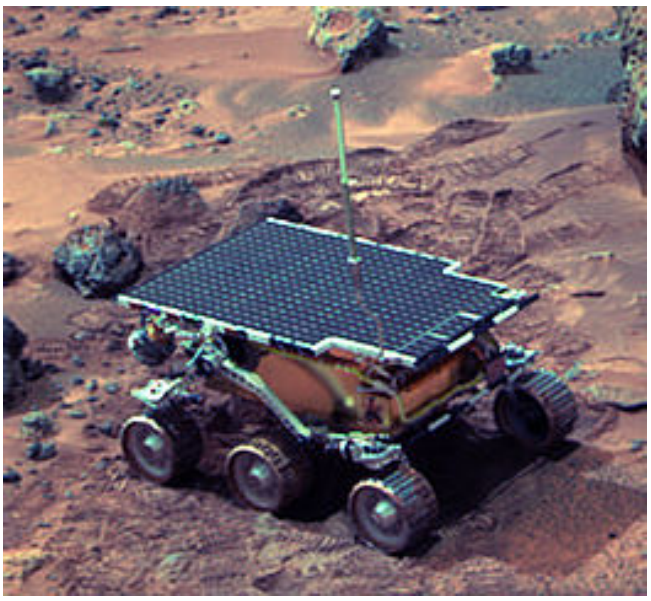

Figure 7: "Sojourner Rover" by NASA under Public domain; from Wikimedia Commons

    - With rover named Sojourner (shown to right)
  - A "low-cost" mission at $280 million
- Bug (priority inversion) caused repeated resets
  - "found in preflight testing but was deemed a **glitch** and therefore given a low priority as it only occurred in certain **unanticipated** heavy-load conditions"
- Priority inversion had been known for a long time
  - E.g.: (Lampson and Redell 1980)

As mentioned for the Linux Futex, priority inversion may arise when threads with different priorities share resources.

This slide points to the Mars Pathfinder mission as famous example for the occurrence of priority inversion, which almost led to a failure of the mission.

Please take a look at the bullet points.

Thus, to repeat: When you program threads of different priorities that share resources, you **must** learn about priority inversion. Simple counter-measures are available. You just have to use them.

5.2 Summary

- Concurrent access to resources leads to races
- Mutual exclusion for critical section prevents races
  - Locks, monitors
  - Keyword `synchronized` in Java
    - Cooperation via `wait()` and `notify()`
- Challenges such as deadlocks, starvation, priority inversion

To sum up, concurrent access to shared resources leads to race conditions, and mutual exclusion for critical sections can prevent them. Locks and monitors are typical MX mechanisms, with support by the OS. In particular, Java implements the monitor concept with the keyword `synchronized` for MX of CSs, and the methods `wait()` and `notify()` for cooperation.

As programmers, we need to be aware of challenges such as deadlocks, starvation, and priority inversion.

Bibliography

Coffman, E. G., M. Elphick, and A. Shoshani. 1971. "System Deadlocks." *Acm Comput. Surv.* 3 (2): 67–78. https://doi.org/10.1145/356586.356588.

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.

Lampson, Butler W., and David D. Redell. 1980. "Experience with Processes and Monitors in Mesa." *Commun. Acm* 23 (2): 105–17. https://doi.org/10.1145/358818.358824.
Stallings, William. 2001. *Operating Systems: Internals and Design Principles*. 4th ed. Prentice Hall.
Tanenbaum, Andrew S. 2001. *Modern Operating Systems*. 2nd ed. Prentice-Hall.
The bibliography contains references used in this presentation.

License Information

This presentation is distributed as Open Educational Resource under freedom granting license terms.

Source files are available on GitLab, where the author would be happy about contributions, e.g., in terms of issues and merge requests.