OS07: MX Challenges *

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)

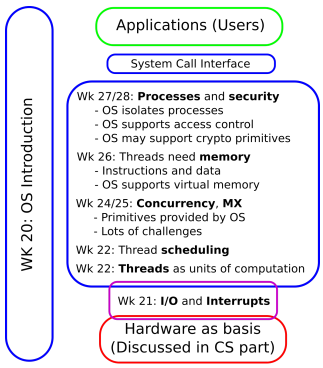

Figure 1: OS course plan, summer 2022

1.2 Today's Core Question

- Am I fine if I lock all shared resources before use?

(Short answer: No. Issues such as priority inversion, deadlocks, starvation may arise.)

1.3 Learning Objectives

- Explain priority inversion and counter measures

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

- Explain and apply deadlock prevention and detection
- Explain convoys and starvation as MX challenges

1.4 Previously on OS ...

- Mutexes may be based either on busy waiting (spinlocks) or on blocking (e.g., lock, mutex, semaphore, monitor)
- Threads may have different **priorities**
  - Lower priority threads are **preempted** for those with higher priority (e.g., with round robin scheduling), which may lead to starvation

1.4.1 Threads, again

1.5 Different Learning Styles

- In previous years, some students reported that Section 4.8.1 (pp. 135 – 137) of [Hai19] on Priority Inversion is quite easy to understand, while they perceived that section in this presentation to be confusing.
- Note that [Hai19] discusses Priority Inversion resulting from locks/mutexes with blocking, while the slides also contain a variant with spinlocks.

Table of Contents

2 Priority Inversion

In general, if threads with different priorities exist, the OS should run those with high priority in preference to those with lower priority.

The technical term "priority inversion" denotes phenomena, where low-priority threads hinder the progress of high-priority threads, which intuitively should not happen. The next slides demonstrate such phenomena, first a weaker variant where MX is enforced with spinlocks, then the more usual variant with MX based on blocking.

2.1 Priority Inversion with Spinlocks

- Example; single CPU core (visualization on next slide)
  1. Thread T_0 with low priority enters CS
  2. T_0 preempted by OS for T_1 with high priority
     - E.g., an important event occurs, to be handled by T_1
     - Note that T_0 is still inside CS, holds lock
  3. T_1 tries to enter same CS and **spins** on lock held by T_0
- This is a variant of **priority inversion**
  - High-priority thread T_1 cannot continue due to actions by low-priority thread
    * If just one CPU core exists: Low-priority thread T_0 cannot continue

  · As CPU occupied by T_1
  · Deadlock (discussed subsequently)
* If multiple cores exist: Low-priority thread T_0 runs although thread with higher priority does not make progress

2.1.1 Single Core

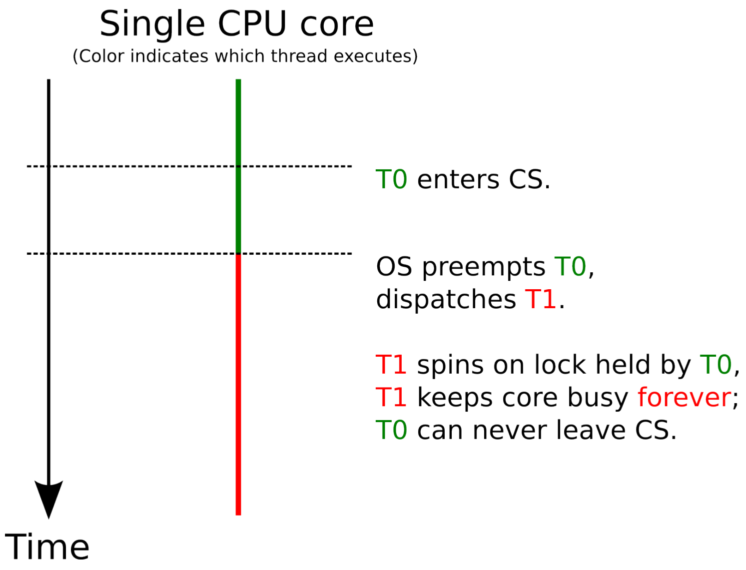

Figure 2: Figure under CC0 1.0

See previous slide or notes for explanations

Green thread T0 executes on the single CPU core where is enters a CS, holding a lock, until the red thread T1 with higher priority arrives. At that point in time, T0 is preempted (still inside the CS, holding a lock), and T1 runs, wanting to enter the CS. T1 now spins on the lock forever.

2.1.2 Two Cores

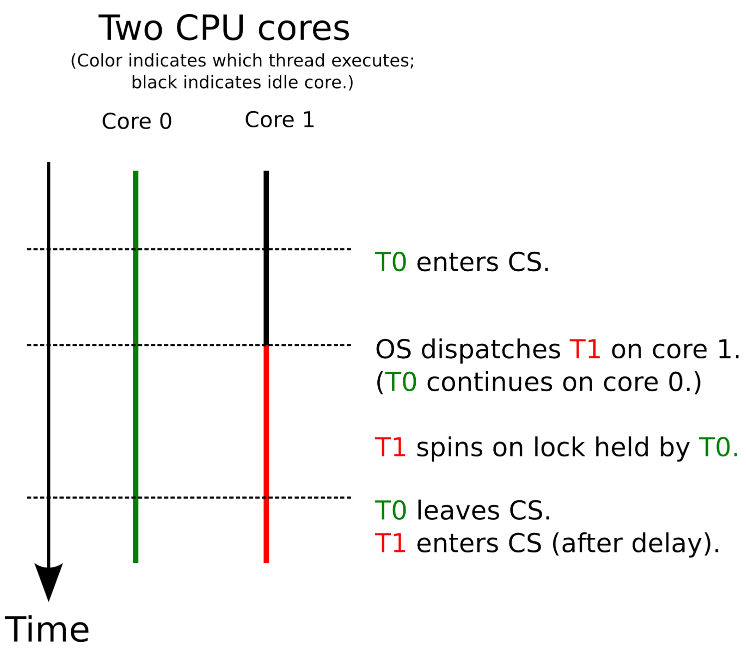

Figure 3: Figure under CC0 1.0

See earlier slide or notes for explanations

This time, two CPU cores are available. As before, green thread T0 executes and is inside a CS, holding a lock. When red thread T1 with higher priority arrives, it can be dispatched for execution on the second core. So, T0 and T1 run in parallel. T1 tries to enter the CS, but needs to spin on the lock held by T0 for some time. As T1 has higher priority than T0, T1 should intuitively not be delayed by T0. Thus, we say that this is a variant of priority inversion.

When T0 leaves the CS and releases the lock, T1 is able to enter the CS.

2.2 Priority Inversion with Blocking

- (Visualization on next slide)
- T_0 with low, T_M with medium, T_1 with high priority
  1. T_0 in CS
  2. An important event occurs, OS preempts T_0 for T_1
     - T_1 attempts entry into same CS, T_1 gets blocked
     - T_0 could continue if no higher priority thread existed
  3. Another, less important, event occurs, to be handled by T_M
     - Based on priority, OS favors T_M over T_0
     - T_M runs instead of more highly prioritized T_1 → **Priority inversion**
       * (T_M does unrelated work, without need to enter the CS)
     - T_0 cannot leave CS as long as T_M exists

- With long running or many threads of medium priority, T_1 (and important event) need to wait for a long time

2.2.1 Blocking CS

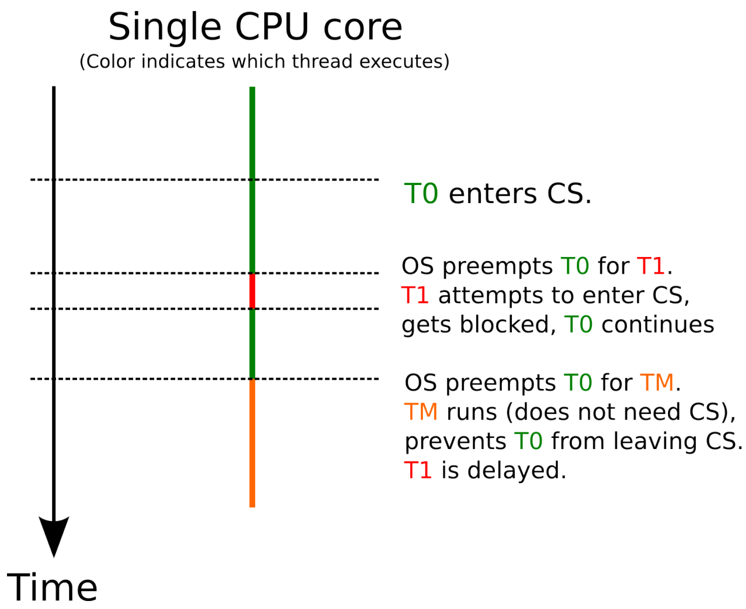

Figure 4: Figure under CC0 1.0

See previous slide for explanations

2.3 Priority Inversion Example

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

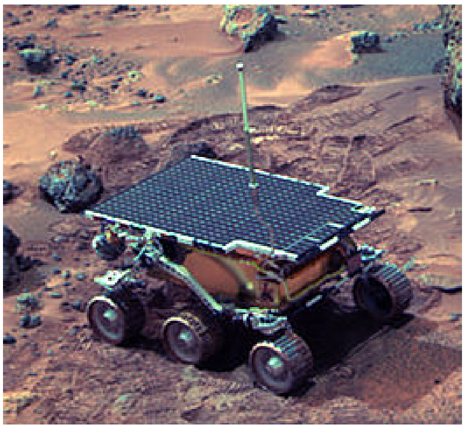

Figure 5: “Sojourner Rover” by NASA under Public domain; from Wikimedia Commons

    * 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.: [LR80]

2.4 Priority Inversion Solutions

- **Priority Inheritance (PI)**
  - Thread of low priority inherits priority of waiting thread
    - E.g., PI-futex in Linux
    - E.g., remote update for Mars Pathfinder
      - Mutex of Pathfinder OS had flag to activate PI
      - Initially, PI was off ...
- **Priority Ceiling (PC)**
  - Every resource has priority (new concept; so far only threads had priorities)
    - (Highest priority that “normal” threads can have) + 1
  - Accessing thread runs with that priority
- In both cases: Restore old priority after access

An intuitive explanation of Priority Inheritance is that if an important task, i.e., a thread with high priority, needs to wait for "something else", then this "something else" immediately gains in importance, as the success of the important task depends on it.

Thus, with Priority Inheritance a thread of low priority holding some lock, mutex, semaphore, or monitor, for which a high priority threads waits, inherits the priority of the high priority waiting thread.

If you think again about the negative effects of medium priority threads for priority inversion, those negative effects do no longer occur, as the thread with inherited priority is now scheduled before medium priority threads. Thus, it finishes fast, allowing the high priority thread to continue quickly.

For Priority Ceiling, each resource is assigned a priority. The priority of a thread accessing that resource is then increased to the resource's priority. Usually, the priority for resources is set to be slightly higher than that of ordinary threads. Thus, for the duration of resource accesses, threads run with highest priority and finish quickly.

In both cases, Priority Inheritance and Priority Ceiling, priorities are restored after resource access.

To conclude, you should think twice whether you want to create threads with different priorities that share resources. If yes, priority inversion may happen. Then, you need to check the documentation for whatever MX mechanism you are about to apply whether it supports PI or PC. If neither is documented, do not use that mechanism.

2.5 Self-Study and Exercise Tasks

- This task is a **variant** of Exercise 4.10 of [Hai19]. Solve part (1) as self-study in Learnweb. Another variant as exercise task.
  - Suppose a computer with only one processor runs a program that creates three threads, which are assigned high, medium, and low fixed priorities. (Assume that no other threads are competing for the same processor.) The threads of high and medium priority are currently blocked, waiting for different events, while the thread with low priority is runnable. Some threads share a single mutex (to protect shared resources that are not shown). Pseudocode for each of the threads is shown subsequently.
    1. Suppose that the mutex does not provide priority inheritance. How soon would you expect the program to terminate? Why?
    2. Suppose that the mutex provides priority inheritance. How soon would you expect the program to terminate? Why?

```
High-priority thread:
  "initially blocked; unblocked to handle event after 200 milliseconds"
  perform lock() on mutex
  run for 2 milliseconds on CPU
  perform unlock() on mutex
  terminate execution of the whole program

Medium-priority thread:
  "initially blocked; unblocked to handle event after 200 milliseconds"
  run for 500 milliseconds on CPU

Low-priority thread:
  "initially runnable"
  perform lock() on mutex
  perform I/O operation which leads to blocking for 600 milliseconds
```

```
run for 3 milliseconds on CPU
perform unlock() on mutex
```

3 Deadlocks

3.1 Deadlock

- Permanent blocking of thread set

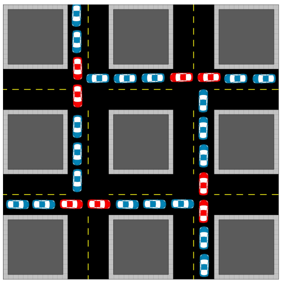

Figure 6: "Gridlock" by Interiot~commonswiki and Jeanacoa under CC BY-SA 2.5 Generic; from Wikimedia Commons

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

- 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?)

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 (in particular, 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.

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.

3.2 Deadlock Example

- Money transfers between bank accounts
  - Transfer from `myAccount` to `yourAccount` by thread 1; transfer in other direction by thread 2
- **Race conditions** on account balances
- Need **mutex** per account
  - Lock both accounts involved in transfer. What order?
- "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**

3.3 Defining Conditions for Deadlocks

Deadlock if and only if (1) – (4) hold [CES71]:

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

3.4 Resource Allocation Graphs

- Representation and visualization of resource allocation as directed graph
  - (Necessary prior knowledge: directed graphs and cycles in such graphs)
  - Nodes
    * Threads (squares on next slide)
    * Resources (circles on next slide)
  - 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
  - Example on next slide
- **Fact**: System in deadlock if and only if graph contains cycle

3.5 Resource Allocation Graph Example

Visualization of deadlock: cyclic resource allocation graph for previous example

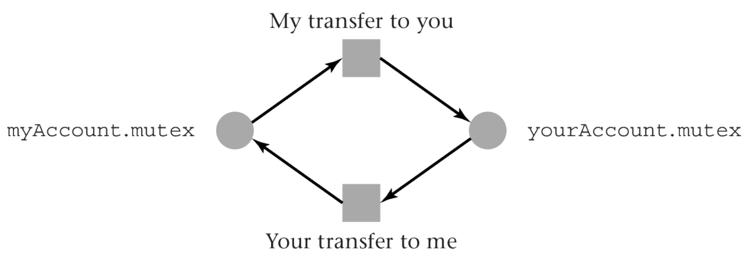

Figure 7: "Figure 4.22 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

(Note: Choice of shapes is arbitrary; just for visualization purposes)

4 Deadlock Strategies

4.1 Deadlock Strategies

- (Ostrich algorithm)
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection

These strategies are covered in subsequent slides.

4.2 Ostrich "Algorithm"

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

(Refresh HTML presentation for other drawings)

4.3 Deadlock Prevention

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

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. Please think about those bullet points on your own.

4.3.1 Linear Resource Ordering Example

- Money transfers between bank accounts revisited
- Locks acquired in **order of account numbers**

Figure 8: Drawing created by Adrian Lison for bonus task in summer term 2017; released into Public Domain; other excellent drawings.

    - 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 [Hai19] for an example of linear ordering in the context of the Linux scheduler)

4.4 Deadlock Avoidance

- (See stackexchange for difference between prevent and avoid)
- 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**
- 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 strategy for deadlocks is called avoidance strategy if it avoids deadlocks. Personally, I don't see much difference between the words "prevent" and "avoid", but this terminology is accepted in the literature on deadlocks.

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

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

4.5 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)
- Prerequisite to build graph
  - Mutex records by which thread it is locked (if any)
  - OS records for what mutex a thread is waiting

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.

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.

5 Further Challenges

5.1 Convoy Problem

- Suppose a central shared resource R exists
  - **Frequently accessed** by lots of threads, protected by **mutex M**
- Preemption of thread T1 holding that mutex is likely
  - Other threads wind up in wait queue of mutex, the **convoy**
    * Thread switches **without much progress**
    * (See [Bla+79] for origin of "convoy" in context of database transactions)
- Suppose T1 continues
  - T1 releases lock, which is reassigned to T2
  - During its time slice, T1 wants R again, but M is now held by T2
    * T1 gets blocked **without much progress**
- The same happens to the other threads
  - The convoy **persists** for a long time

The previous explanations of `unlock()` (here and there) hinted at two alternatives to reassign unlocked locks.

The bullet points here argue that immediate reassignment of locks is bad for frequently used resources: Here, lots of threads—among them T1 and T2—need the same shared resource, say R. T1 currently holds the mutex M that is used for MX on R, and T2 is blocked, waiting for M to be unlocked. If (a) T1 unlocks M and (b) the OS immediately assigns the mutex to T2 (making T2 runnable but without any scheduling decision yet, so T1 continues to run), then T1 cannot access R again during its time slice, but will be blocked when trying to lock M.

If R is accessed frequently, threads will not be able to use much of their time slices, leading to frequent context switches without much progress in individual threads. The threads blocked on M are collected in a queue, which is called **convoy**.

A simple fix is explained on the next slide: When M is unlocked by T1, the OS (a) changes the states of all threads that are blocked on M (including T2) to runnable, but (b) does not reassign the mutex. Thus, when T1 needs R again during its time slice, it can simply lock M and proceed without problems.

5.1.1 A Convoy Solution

- Change mutex behavior
  - Proposed in [Bla+79]
  - Do no immediately reassign mutex upon `unlock()`
  - Instead, make **all** waiting threads runnable
    * Without reassigning mutex
  - (In addition, for performance reasons in case of failing locking attempt [Bla+79] suggests "to spin for a few instructions in the hope that the lock will become free")
- Effect: T1 can `lock()` M repeatedly during its time slice

5.2 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.

5.2.1 Dining Philosophers

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

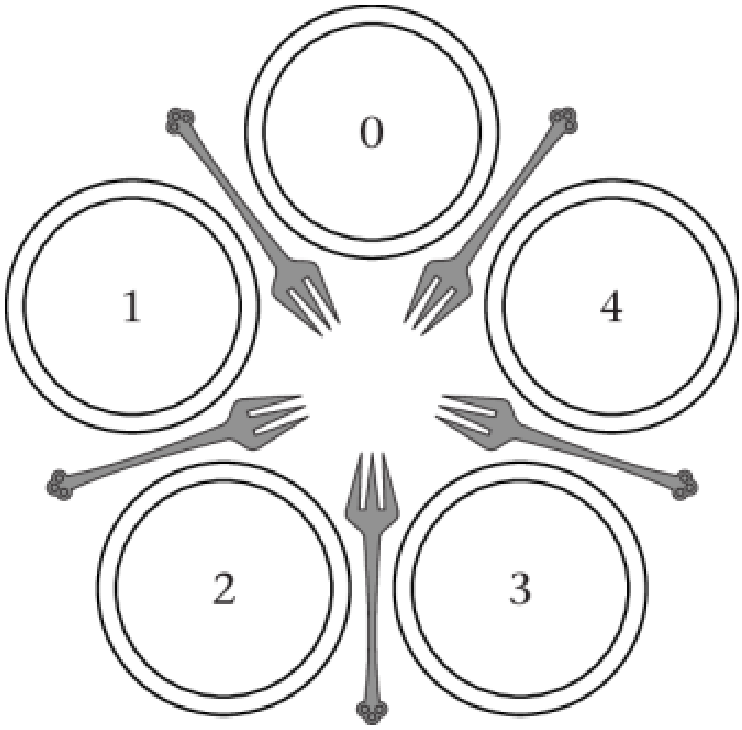

Figure 9: Dining Philosophers ("Figure 4.20 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; converted from GitHub)

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. Inspired by an exercise in [Sta01], the following slide shows a sequence of events that can happen for the deadlock-free algorithm presented in [Tan01], leading to starvation of a philosopher.

Apparently, avoidance of starvation is no simple task.

5.2.2 Starving Philosophers

- Starvation of P0

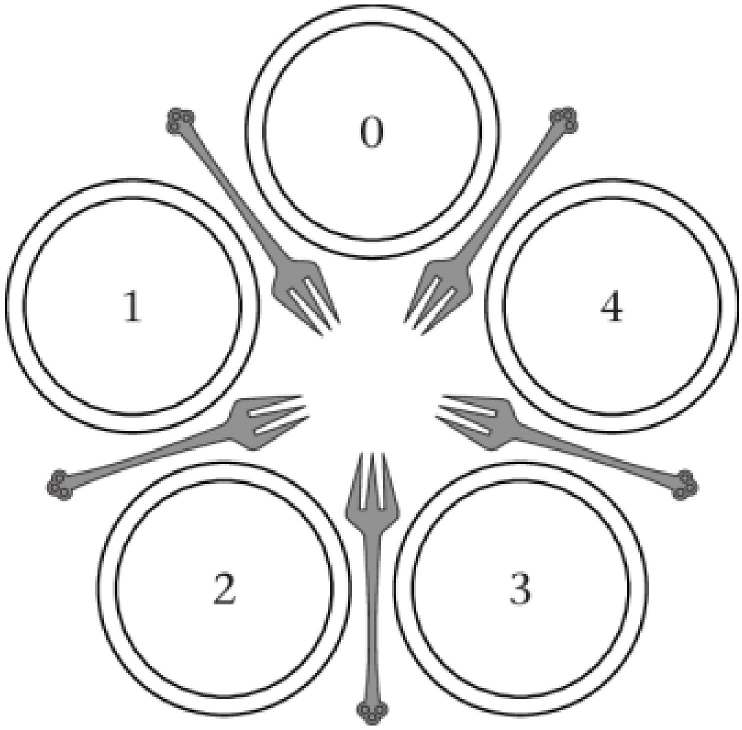

Figure 10: “Figure 4.20 of [Hai17]” by Max Hailperin under CC BY-SA 3.0; converted from GitHub

  - 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

6 Conclusions

6.1 Summary

- MX to avoid race conditions
- Challenges
  - Priority inversion
  - Deadlocks
  - Convoys
  - Starvation

Bibliography

[Bla+79] Mike Blasgen et al. "The Convoy Phenomenon". In: *SIGOPS Oper. Syst. Rev.* 13.2 (1979), pp. 20–25. DOI: 10.1145/850657.850659. URL: https://doi.org/10.1145/850657.850659.

[CES71] E. G. Coffman, M. Elphick, and A. Shoshani. "System Deadlocks". In: *ACM Comput. Surv.* 3.2 (June 1971), pp. 67–78. ISSN: 0360-0300. DOI: 10.1145/356586.356588. URL: https://dl.acm.org/citation.cfm?doid=356586.356588.356588.

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

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

[LR80] Butler W. Lampson and David D. Redell. "Experience with Processes and Monitors in Mesa". In: *Commun. ACM* 23.2 (Feb. 1980), pp. 105–117. ISSN: 0001-0782. DOI: 10.1145/358818.358824. URL: https://dl.acm.org/citation.cfm?doid=358818.358824.

[Sta01] William Stallings. *Operating Systems: Internals and Design Principles*. 4th ed. Prentice Hall, 2001.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. 2nd ed. Prentice-Hall, 2001.

License Information