OS03: Threads *

Based on Chapter 2 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)

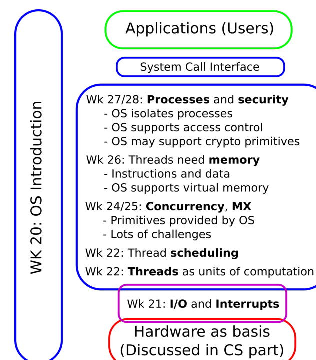

Figure 1: OS course plan, summer 2022

1.2 Today's Core Questions

- What exactly are threads?
  - Why and for what are they used?
  - How can I inspect them?
  - How are they created in Java?
  - What impact does blocking or non-blocking I/O have on the use of threads?
  - How does switching between threads work?

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

1.3 Learning Objectives

- Explain thread concept, thread switching, and multitasking
  - Including states (after upcoming presentation)
  - Explain distinctions between threads and processes
  - Explain advantages of a multithreaded organization in structuring applications and in performance
- Inspect threads on your system
- Create threads in Java
- Discuss differences between and use cases for blocking and non-blocking I/O

1.4 Retrieval practice

1.4.1 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`

1.4.2 Recall: Stack

- Stack = Classical data structure (abstract data type)
  - LIFO (last in, first out) principle
  - See Appendix A in [Hai19] if necessary
- Two elementary operations
  - `Stack.Push(o)`: place object `o` on top of `Stack`
  - `Stack.Pop()`: remove object from top of `Stack` and return it
- Supported in machine language of most processors (not in Hack, though)
  - Typically (e.g., x86), stack grows towards smaller addresses
    * Next object pushed gets smaller address than previous one
    * (Differently from stack of VM for Hack platform)

1.4.3 Drawing on Stack

Warning! External figure **not** included: "What's the stack?" © 2016 Julia Evans, all rights reserved from julia's drawings. Displayed here with personal permission.
(See HTML presentation instead.)

1.4.4 Previously on OS . . .

- What is a thread? **Warning!** External figure **not** included: "Threads!" © 2016 Julia Evans, all rights reserved from julia's drawings. Displayed here with personal permission.
  (See HTML presentation instead.)
- What are multitasking and scheduling?

1.4.5 Recall: Blocking vs Non-Blocking I/O

- For blocking as well as non-blocking I/O, thread invokes system call
  - OS is responsible for I/O
- **Blocking** I/O: OS initiates I/O and schedules **different** thread for execution
  - Calling thread is **blocked** for duration of I/O
  - After I/O is finished, OS un-blocks calling thread
    - Un-blocked thread to be scheduled later on, with result of I/O system call
- **Non-blocking** I/O: OS initiates I/O and returns (incomplete) result to calling thread

Table of Contents

2 Threads

2.1 Threads and Programs

- Program vs thread
  - Program contains instructions to be executed on CPU
  - OS schedules execution of programs
    - By default, program execution starts with one **thread**
      - **Thread** = unit of OS scheduling = independent sequence of computational steps
    - Programmer determines how many threads are created
      - (OS provides system calls for thread management, Java an API)
  - Simple programs are single-threaded
  - More complex programs can be multithreaded
    - Multiple independent sequences of computational steps
      - E.g., an online game: different threads for game AI, GUI events, network handling
    - Multi-core CPUs can execute multiple threads in parallel

2.2 Thread Creation and Termination

- Different OSes and different languages provide different APIs to manage threads
  - Thread creation
    - Following example: Java
    - [Hai19]: Java and POSIX threads
  - Thread termination
    - API-specific functions to end/destroy threads
    - Implicit termination when "last" instruction ends
      - E.g., in Java when methods `main()` (for main thread) or `run()` (for other threads) end (if at all)

2.3 Thread Terminology

- Parallelism vs concurrency
- Thread preemption
- I/O bound vs CPU bound threads

2.3.1 Parallelism

- **Parallelism** = **simultaneous** execution
  - E.g., multi-core
  - Potential speedup for computations!
    - (Limited by Amdahl's law)
- Note
  - Processors contain more and more cores
  - Individual cores do not become much faster any longer
    - Recall CS part about Moore's law
  - Consequence: Need parallel programming to take advantage of current hardware

2.3.2 Concurrency

- Concurrency is **more general** term than parallelism
  - Concurrency includes
    - **Parallel** threads (on multiple CPU cores)

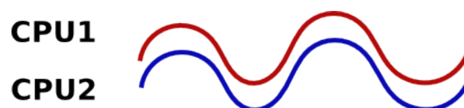

Figure 2: Figure under CC0 1.0

          - (Executing different code in general)
      - **Interleaved** threads (taking turns on single CPU core)
          - With gaps on single core!

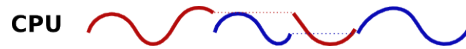

Figure 3: Figure under CC0 1.0

  - Challenges and solutions for concurrency apply to parallel and interleaved executions
      - Topics covered in upcoming presentations (mutual exclusion (MX), MX in Java, MX challenges)

2.3.3 Thread Preemption

- **Preemption** = temporary removal of thread from CPU by OS
  - Before thread is finished (with later continuation)
      - To allow others to continue after **scheduling** decision by OS
  - Typical technique in modern OSs
      - Run lots of threads for brief intervals per second; creates illusion of parallel executions, even on single-core CPU
- Later slides: Cooperative vs preemptive multitasking
- Upcoming presentation: Thread scheduling

2.3.4 Thread Classification

- **I/O bound**
  - Threads spending most time submitting and waiting for I/O requests
  - Run frequently by OS, but only for short periods of time
      - Until next I/O operation
      - E.g., virus scanner, network server (e.g., web, chat)
- **CPU bound**
  - Threads spending most time executing code
  - Run for longer periods of time
      - Until preempted by scheduler
      - E.g., graph rendering, compilation of source code, training for deep learning

3 Java Threads

3.1 Threads in Java

- Threads are created from **instances** of classes implementing the Runnable interface
  1. Implement `run()` method
  2. Create new Thread instance from `Runnable` instance
  3. Invoke `start()` method on `Thread` instance
- Alternatives (beyond the scope of this course)
  - Subclass of `Thread` (`Thread implements Runnable`)
    - If more than `run()` overwritten
  - java.util.concurrent.Executor
    - With `Callable<V>`, `Future<V>` and service methods of `Executor`
      - Worker Thread Pool
    - Virtual threads
      - Later slide with some ideas

Here you see one way for the creation of threads in Java with 3 basic steps. First, the code to be executed by a thread is specified by the method `run()` of the interface `Runnable`, and a thread itself is represented as instance of a class that implements this interface.

Note that programmers do not call `run()` directly: Instead, step (2) requires to create an instance of class `Thread`, which is then started in step (3) with method `start()`, which in turn eventually calls `run()`.

The next slide shows an example for these three steps.

3.2 Java Thread Example

```
public class Simpler2Threads { // Based on Fig. 2.3 of [Hai17]
  // "Simplified" by removing anonymous class.
  public static void main(String args[]){
    Thread childThread = new Thread(new MyThread());
    childThread.start();
    sleep(5000);
    System.out.println("Parent is done sleeping 5 seconds.");}

  static void sleep(int milliseconds){
    // Sleep milliseconds (blocked/removed from CPU).
    try{ Thread.sleep(milliseconds); } catch(InterruptedException e){
      // ignore this exception; it won't happen anyhow
    }}}

class MyThread implements Runnable {
  public void run(){
    Simpler2Threads.sleep(3000);
    System.out.println("Child is done sleeping 3 seconds.");
  }}
```

Connecting Java threads to earlier topics, when you execute `java`, the OS creates a process for the Java runtime. Within that process, one thread will be created to execute the `main()` method. In addition, the runtime creates an implementation-specific number of threads for Java management tasks. Such threads are not important for our purposes.

The program Simpler2Threads does not only run code in the main thread, but it is multi-threaded, as `start()` is invoked on the new `Thread` instance `childThread` (and `start()` automatically calls `run()` to execute the thread's code).

Importantly, the method `sleep()` used here involves a blocking system call to the OS, essentially stating "please take me away for the specified amount of milliseconds and give the CPU to someone else".

Which output do you expect at what points in time?

3.3 Self-Study Task: CPU Usage

This task is available for self-study in Learnweb.

- Compile and run the source code of Simpler2Threads and make sure that you can explain its output. Maybe ask in Learnweb.
  - In general, if a program behaves unexpectedly, a debugger helps to understand what is happening when. Your favorite IDE probably includes a debugger. Also, several Java implementations come with a simple debugger called `jdb`, e.g., the OpenJDK tools. (The notes on this slide contain sample commands for `jdb`.)

- jdb Simpler2Threads
- stop in Simpler2Threads.main
- run
- threads
- stop in MyThread.run
- step
- threads

4 Reasons for Threads

4.1 Main Reasons

- **Resource utilization**
  - Keep most of the hardware resources busy most of the time, e.g.:
    - While one thread is **idle** (e.g., waiting for external events such as user interaction, disk, or network I/O), allow other threads to continue
      - Next slide
    - Keep multiple CPU cores busy
      - E.g., OS housekeeping such as zeroing of memory on second core
- **Responsiveness**
  - Use separate threads to react quickly to external events
    - Think of game AI vs GUI
    - Other example on later slide: Web server
- More **modular design**

4.2 Interleaved Execution Example

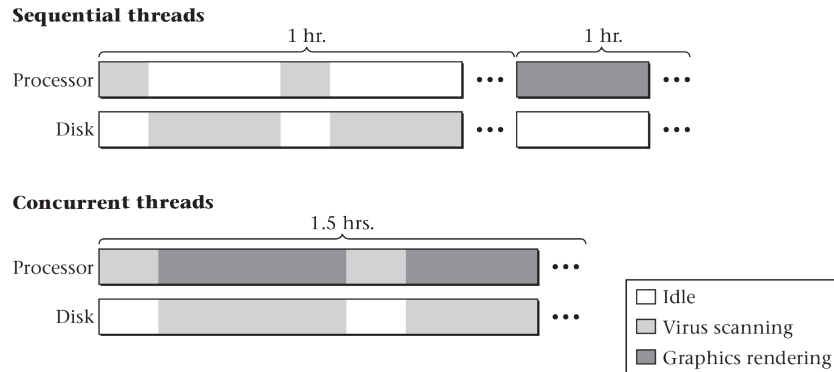

Figure 4: "Interleaved execution example" by Jens Lechtenbörger under CC BY-SA 4.0; SVG image refers to converted and cut parts of Figure 2.6 of a book by Max Hailperin under CC BY-SA 3.0. From GitLab

This figure illustrates the benefit of improved resource utilization resulting from multithreading, which leads to higher overall throughput. Consider two threads and their resource demands, each taking 1h to finish. The first thread, shown on the left, is I/O bound, in this case a virus scanner, which uses the CPU only for brief periods of time, whereas it mostly waits for new data to arrive from disk. In contrast, the other thread, shown to the right, is CPU bound, performing complex graph rendering; it doesn't need the disk at all. Clearly, the sequential execution of both threads, which takes 2h, is a waste of resources, namely a waste of CPU time.

In fact, an OS that is equipped with a scheduling mechanism might be able to schedule the 2nd thread whenever the 1st one is idle waiting for new data to arrive from disk. In that case, both threads can be executed in an interleaved fashion on a single CPU core, keeping the core busy all the time. In the example shown here, both threads now finish after 1.5h.

Note that the total time of 1.5h is an arbitrary example, without underlying calculation. The point is that idle times of the virus scanner can now be used for real work, which leads to a total time of less than 2h. Of course, both threads could also finish at different points in time (but earlier than 2h).

4.3 Example: Web Server

- Web server "talks" HTTP with browsers
- Simplified
  - Lots of browsers ask per GET method for various web pages
  - Server responds with HTML files
- How many threads on server side?

4.4 Single- vs Multithreaded Web Server

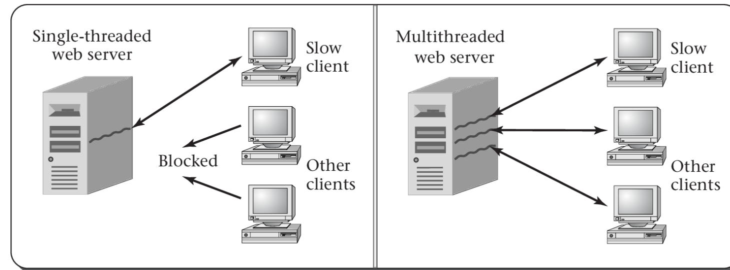

Figure 5: “Figure 2.5 of [Hai19]” by Max Hailperin under CC BY-SA 3.0; converted from GitHub

This figure illustrates a thought experiment.

Suppose that you implemented a web server using a single thread. When a browser connects, it typically asks for a sequence of resources such as images, CSS, JavaScript, and HTML files. These resources need to be retrieved from disk and transmitted over the Internet, before an entire page can be rendered by the browser. If the server processes this entire sequence before turning to the next client, complex pages, network latency, and slow clients will cause long delays for other clients. Consequently, web servers are not built this way.

At another extreme, which is also not used in practice for reasons to be discussed later, a multithreaded server could create a new thread to handle each incoming request separately. That way, requests can be processed in parallel or concurrently, which improves responsiveness and resource usage. In particular, the server would no longer be slowed down by slow clients.

5 Thread Switching

5.1 Thread Switching for Multitasking

- With multiple threads, OS decides which to execute when → Scheduling (later lecture)
  - (Similar to machine scheduling for industrial production, which you may know from operations management)
  - Recall multitasking
    - OS may use time-slicing to schedule threads for short intervals, illusion of parallelism on single CPU core
- After that decision, a **context switch** takes place
  - (Recall introduction and interrupts; now, as **thread switch**)
  - Remove thread A from CPU
    - **Remember A’s state** (in TCB: instruction pointer, register contents, stack, . . .)
  - **Dispatch** thread B to CPU
    - **Restore B’s state**

Recall how code is executed on the CPU (e.g., with Hack). A special register, the program counter, specifies what instruction to execute next, and instructions may modify CPU registers. You may think of one assembly language program as being executed in one thread.

Recall that with multitasking, the OS manages multiple threads and schedules them for execution on CPU cores, usually with time-slicing.

If the time slice for thread A ends, A is usually in the middle of some computation. The state of that computation is defined by the current value of the program counter, by values stored in registers, and other information. To resume this computation later on, the OS needs to save the state of thread A somewhere, before another thread B can be executed. Similarly, thread B may be in the middle of its own computation, whose state was saved previously by the OS.

The switch from thread A via the OS to thread B with saving of A's and restoring of B's state is called a *context switch*. Subsequent slides provide more details how such context switches happen.

5.1.1 Thread Switching with `yield`

In the following

- First, simplified setting of voluntary switch from thread A to thread B
  - Function `switchFromTo()` on next slide
    - For details, see Sec. 2.4 in [Hai19]
  - Leaving the CPU voluntarily is called **yielding**; `yield()` may really be an OS system call
- Afterwards, the real thing: **Preemption** by the OS

5.2 Interleaved Instruction Sequence

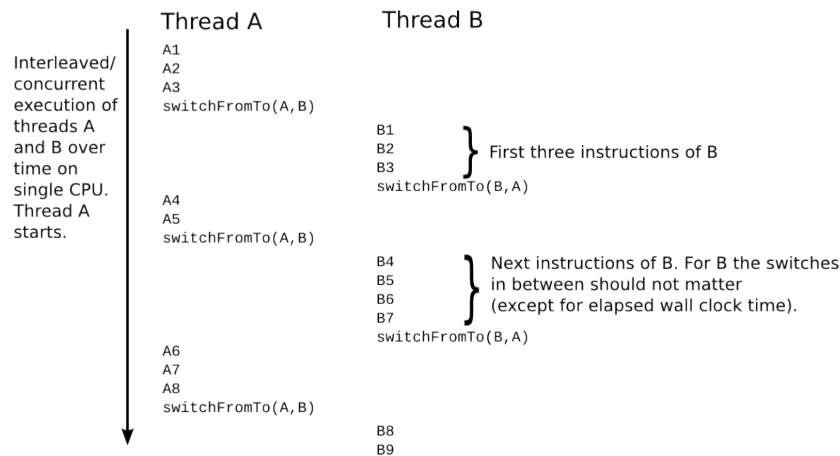

Figure 6: "Interleaved execution of threads. Based on Figure 2.7 of book by Max Hailperin, CC BY-SA 3.0." by Jens Lechtenbörger under CC BY-SA 4.0; from GitLab

5.3 Thread Control Blocks (TCBs)

- All threads share the same CPU registers
  - Obviously, register values need to be **saved** somewhere to avoid incorrect results when switching threads

- Also, each thread has its own
  - **stack**; current position given by **stack pointer (SP)**
  - **instruction pointer (IP)** (program counter); where to execute next machine instruction
- Besides: priority, scheduling information, blocking events (if any)

- OS uses block of memory for housekeeping, called **thread control block** (TCB)
  - One for each thread
    - Storing register contents, stack pointer, instruction pointer, ...
  - Arguments of `switchFromTo()` are really (pointers to) TCBs

5.4 Cooperative Multitasking

- Approach based on `switchFromTo()` is **cooperative**
  - Thread A decides to yield CPU (voluntarily)
    - A hands over to B
- Disadvantages
  - Inflexible: A and B are hard-coded
  - No parallelism, just interleaved execution
  - What if A contains a bug and enters an infinite loop?
- Advantages
  - Programmed, so full control over when and where of switches
  - Programmed, so usable even in restricted environments/OSs without support for multitasking/preemption

5.5 Preemptive Multitasking

- Preemption: OS removes thread **forcefully** (but only temporarily) from CPU
  - Housekeeping on stacks to allow seamless continuation later on similar to cooperative approach
  - OS schedules different thread for execution afterwards
- Additional mechanism: Timer interrupts
  - OS defines **time slice** (**quantum**), e.g., 30ms
    - Interrupt fires every 30ms
    - Interrupt handler invokes OS scheduler to determine next thread
      - Details in upcoming presentation

5.6 Multitasking Overhead

- OS performs scheduling, which takes time
- Thread switching creates **overhead**
  - Minor sources: Scheduling costs, saving and restoring state
  - Major sources: Recall cache pollution
    - After a context switch, the CPU's cache quite likely misses necessary data
      - Necessary data needs to be fetched from RAM
    - Accessing data in RAM takes hundreds of clock cycles
      - See estimates on Stack Overflow

6 Server Models

6.1 Server Models with Blocking I/O: Single-Threaded

- **Single thread** (left-hand side of Figure 2.5; thought experiment, not for use)

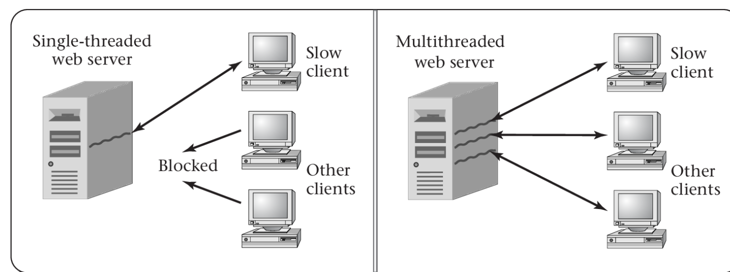

Figure 7: "Figure 2.5 of [Hai19]" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

- **Sequential** processing of client requests
  - Long idle time during I/O → Not suitable in practice

Recall Figure 2.5, where you saw that using a single thread to serve lots of clients may lead to long delays.

Think of the thread as a single employee in a self-service restaurant, who always processes each client entirely (e.g., order, cooking, payment) before turning to the next client.

6.2 Server Models with Blocking I/O: Multi-Threaded

- One **thread** (or process) **per client connection**
- **Parallel** processing of client connections
  - No idle time for I/O (switch to different client)
  - Limited scalability (thousands or millions of clients?)

        * Creation of threads causes overhead
            · Each thread allocates resources
        * OS needs to identify "correct" thread for incoming data

- Worker Thread Pool as compromise

In a conceptually simple server model, a new thread is created to serve each incoming connection. Thus, different clients can be served in parallel, keeping all CPU cores busy (if enough connections are active). Also, slow clients do not slow down other clients, and the OS can take a thread waiting for I/O aside and allow another thread to execute instead. Think of a hypothetical self-service restaurant, where each customer is served by their own employee.

However, each thread needs some resources (e.g., RAM) and needs to be managed by the OS. Also, when data arrives in an ongoing connection, the OS first needs to identify the correct thread to handle incoming data before then performing a context switch to the correct thread. These types of overhead limit the scalability of this model, which is why the compromise of a Worker Thread Pool exists, to be discussed later.

6.3 Server Models with Non-Blocking I/O

- Single thread
    - **Event** loop, event-driven programming
    - E.g., web servers such as lighttpd, nginx
- Finite automaton to keep track of state per client
    - State of automaton records state of interaction
    - Complex code
        * See Google's experience mentioned in [Bar+17]
- Avoids overhead of context switches
    - Scalable (may be combined with Worker Thread Pool)

Another option to implement servers lies in the use of non-blocking I/O operations, which is used by web servers such as lighttpd or nginx. Here, a single thread serves lots of incoming requests in an interleaved fashion.

Processing of a request starts as in any other model. When I/O is necessary, e.g., to fetch HTML code from disk before network transfer, the thread executes a non-blocking I/O operation. Thus, the OS does *not* take the thread aside (as it would for blocking I/O operations) but immediately returns an incomplete result. (Think of you as thread in a self-service restaurant. You place your order and receive some receipt or order number instead of your meal.)

Hence, the thread sees that it should do something else for some time. So, the thread remembers the state of the current interaction, typically in some finite automaton, and continues to work on another request. (You might take the receipt in the self-service restaurant and continue your work on a self-study task while your meal is prepared. In addition, employees taking orders can again be perceived as threads with non-blocking calls into the kitchen; while meals are prepared, employees turn to other customers.)

This model avoids the overhead of context switches as a single thread can continue with 100% CPU usage. (You do not just sit there, wasting your time, but you order, work on exercises, eat, etc.)

Also, we can create one thread per CPU core for parallel processing with up to 100% CPU utilization across all cores.

However, server code is complex and error prone. The research paper cited here includes experiences at Google, stating that blocking "code is a lot simpler, hence easier to write, tune, and debug."

6.4 Worker Thread Pool

- Upon program start: Create **set** of **worker** threads
- Client requests received by **dispatcher** thread
  - Requests recorded in to-do data structure
- Idle worker threads process requests
- Note
  - **Re-use** of worker threads
  - **Limited** resource usage
  - How to tune for load?
    - Dispatcher may be bottleneck
    - If more client requests than worker threads, then potentially long delays

With worker thread pools, a fixed number of threads, namely one dispatcher and several workers, are created ahead of time. The dispatcher just records incoming requests in a to-do data structure.

If a worker is currently idle (has nothing to do), it checks whether the to-do list contains requests. If so, the worker picks one, removes it from the list, and processes it.

Think of a self-service restaurant with a single employee as dispatcher and multiple cooks as workers.

With this model, the overhead compared to the thread-per-connection is limited as (a) the number of threads is fixed and (b) workers do not need to be created and destroyed dynamically. Also note that dispatcher and workers can be assigned to different CPU cores for parallel work. It may happen, though, that dispatcher or workers are overloaded by the number of incoming requests, which leads to potentially long delays (again, think of self-service restaurants).

6.4.1 Aside: Virtual Threads in Java

- Beyond class topics
- Java 19 introduced **virtual threads** in preview API
  - See JEP 436 or blog post for details
  - Overhead reduced to enable thread per client model with blocking I/O
    - Map large number of virtual threads to pool of small number of OS threads
    - When virtual thread blocks on I/O, Java runtime performs non-blocking OS call and
      - suspends virtual thread until it can be resumed later and
      - executes different virtual thread on same OS thread

7 Conclusions

7.1 Summary

- Threads represent individual instruction execution sequences
- Multithreading improves
  - Resource utilization
  - Responsiveness
  - Modular design in presence of concurrency
- Preemptive multithreading with housekeeping by OS
  - Thread switching with overhead
- Design choices: I/O blocking or not, servers with multiple threads or not

Bibliography

[Bar+17] Luiz Barroso et al. "Attack of the Killer Microseconds". In: *CACM* 60.4 (2017), pp. 48–54. URL: https://dl.acm.org/citation.cfm?id=3015146.

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

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, the work "OS03: Threads", © 2017-2023 Jens Lechtenbörger, is 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.