Threads *

Jens Lechtenbörger

IT Systems, Summer Term 2024

This presentation is an introduction to threads in operating systems.

1 Introduction

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

1.1 Core Questions

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

(Based on Chapter 2 of (Hailperin 2019))
This presentation addresses the following questions: What exactly are threads? How does switching between threads work? Why and for what are they used? How are they created in Java? What impact does blocking or non-blocking I/O have on the use of threads?

1.2 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

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`

From your introductory programming course you should be able to explain interfaces, classes, and instances such as `this` in Java. Maybe see the URLs provided here.

1.3.2 Recall: Stack

- Stack = Classical data structure (abstract data type)
  - LIFO (last in, first out) principle
  - See Appendix A in (Hailperin 2019) 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)
  - With stack register pointing to top of stack

We take the basic data structure **stack** for granted. Recall that it comes with two elementary operations, namely push and pop. Most processors support this data structure with instructions in machine language. Then, a special purpose CPU register contains a stack pointer, i.e., the address of the top of the stack.

1.3.3 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.)

This drawing illustrates use cases for stacks as places where variables, function arguments, and return addresses for function calls are maintained. As threads execute functions, each thread has at least one stack.

1.3.4 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?
- What are blocking and non-blocking I/O?

Please recall the notions of thread, multitasking, scheduling, and blocking as well non-blocking I/O.

1.4 Terminology

The next two slides introduce basic terminology related to multitasking and threads.

1.4.1 Parallelism

- **Parallelism** = **simultaneous** execution
  - E.g., multi-core
  - Potential speedup for computations!
    - (Limited by Amdahl's law)

Parallelism refers to the simultaneous execution of multiple threads, such as in the case of multi-core processors that are capable of executing multiple instructions at the same time. This can potentially lead to a significant speedup in computations. Beyond class topics, Amdahl's law demonstrates that inherently serial parts of computations limit the maximum speedup.

- Note
  - Processors contain more and more cores
  - Individual cores do not become much faster any longer
    - Recall future of Moore's law
  - Consequence: Need parallel programming to take advantage of current hardware

As pointed out in the computer architecture part, there is a trend towards processors containing an increasing number of cores rather than individual cores becoming faster. This shift can be attributed to the fact that the rate of improvement in processor speeds predicted by Moore's law is no longer being met. As a result, taking advantage of the current hardware requires the use of parallel programming techniques to utilize the available resources and improve performance.

1.4.2 Concurrency

- Concurrency is **more general** term than parallelism
  - Concurrency includes

* **Parallel** threads (on multiple CPU cores)

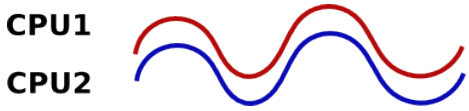

Figure 1: Figure under CC0 1.0

  · (Executing different code in general)
* **Interleaved** threads (taking turns on single CPU core)

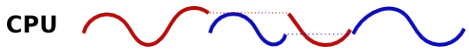

Figure 2: Figure under CC0 1.0

Concurrency is a broader concept than parallelism. While parallelism specifically refers to the simultaneous execution of multiple threads, concurrency encompasses both parallel and interleaved thread executions. In particular, concurrency arises when parallel threads run simultaneously on multiple CPU cores, as well as when the OS interleaves the execution of multiple threads with scheduling.

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

Despite these differences, many of the challenges and solutions associated with concurrent programming apply equally to both parallel and interleaved executions. The topics related to these challenges and solutions will be covered in greater detail during the upcoming presentations.

Agenda

The agenda for this presentation is as follows.

After this introduction, we investigate threads in more detail, and we revisit context switches in the form of thread switching. Afterwards, we look at the creation of threads in Java, before we discuss server models that make use of threads in different ways.

Conclusions end the presentation.

2 Threads

Let us take a closer look at threads.

2.1 Threads and Programs

- Program vs thread
  - Program contains instructions to be executed on CPU
  - OS schedules execution of threads

    Our programs are ultimately executed as machine instructions on the CPU. The operating system schedules and manages the execution of these instructions in the form of threads.

        * 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)

        By default, when a program begins executing, it runs with a single thread, where each thread represents an independent sequence of computational steps. However, the programmer determines the number of threads created within a program.
        The OS provides system calls for managing threads, and there also exist programming languages like Java that offer APIs for creating and manipulating threads.

    - Simple programs are single-threaded
    - More complex programs can be multi-threaded
        * 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

On the one hand, simple programs have only one thread and are referred to as single-threaded. On the other hand, more complex programs may require multiple independent sequences of computational steps, making them multi-threaded. For instance, consider an online game; its various components such as artificial intelligence, graphical user interface events, and network communication could all run concurrently using separate threads. Modern multi-core processors can take advantage of this design pattern by simultaneously executing multiple threads in parallel, thereby improving performance.

2.2 Thread Creation and Termination

- Different OSes and different languages provide different APIs to manage threads
    - Thread creation
        * Following example: Java
        * (Hailperin 2019): Java and POSIX threads

        Various operating systems and programming languages come equipped with their own unique APIs designed specifically for managing threads. These APIs enable developers to create new threads easily, depending upon the specific language being used. For instance, we will see the Java API subsequently, while the textbook also mentioned the famous POSIX API, which is beyond our scope.

    - 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)

Every created thread can also be terminated. Each API comes with its own set of functions dedicated to ending or destroying threads explicitly. In addition, threads might terminate implicitly, once they complete their final instruction. This happens when the primary function associated with a particular thread ends, e.g., when the `main` or `run` methods end in Java's case.

2.3 Thread Classification

- I/O bound vs CPU bound

  Two major classes of threads are I/O bound threads and CPU bound threads.

  - **I/O bound**
    - Threads spending most time submitting and waiting for I/O requests
    - Run only for short periods of time
      - Until next I/O operation
      - E.g., virus scanner, network server for web or chat

    An I/O bound thread is one that spends most of its time submitting I/O requests, and waiting for their completion. These threads typically only execute for short periods of time until they are blocked for the next I/O operation. Examples of I/O intensive threads are virus scanners or network servers.

  - **CPU bound**
    - Threads spending most time executing code
    - Run for longer periods of time
      - Until time slice runs out
      - E.g., graph rendering, compilation of source code, training for deep learning

A CPU bound thread is one that spends most of its time executing code rather than waiting for I/O operations to complete. CPU bound threads often run for full time slices. Examples of CPU bound tasks include rendering graphics, compiling source code, or training models for deep learning.

2.4 Reasons for Threads

- **Resource utilization**
  - Keep most of the hardware resources busy most of the time, e.g.:
    - While one thread is **idle** or **blocked** (e.g., waiting for external events such as user interaction, disk, or network I/O), allow other threads to continue
      - Next slide
    - Parallelism: Utilize multiple CPU cores
      - Different threads on different cores

  Maintaining high levels of resource utilization is an important goal in multithreaded programming, because it allows us to keep most of the computer's hardware resources busy most of the time. For example, while one thread is idle or blocked (for instance, waiting for external events like user interaction, disk I/O, or network communication), allowing other threads to continue executing can help ensure that your computer's CPUs remain fully utilized. Additionally, distributing work across multiple threads and running them on different CPU cores can further increase resource utilization and thereby the speed of programs.

- **Responsiveness**
  - Use separate threads to react quickly to external events
    - Think of game AI vs GUI

    - Other example on later slide: Web server

  Another key benefit of using multiple threads is improved responsiveness. By dedicating specific threads to handling particular types of external events, such as those triggered by user input or network activity, you can ensure that your program responds quickly to these stimuli. This can be especially important in applications where rapid response times are critical, such as video games or real-time systems. One example of this approach could involve having one thread handle non-player characters in a game, while another manages the graphical user interface.

  Similarly, we will look at web servers and their use of separate threads for the processing of incoming requests from multiple clients.

- More **modular design**

  Finally, adopting a more modular design through the use of multiple threads can make it easier to develop complex, yet maintainable programs with well-defined components that serve individual purposes.

2.5 Interleaved Execution Example

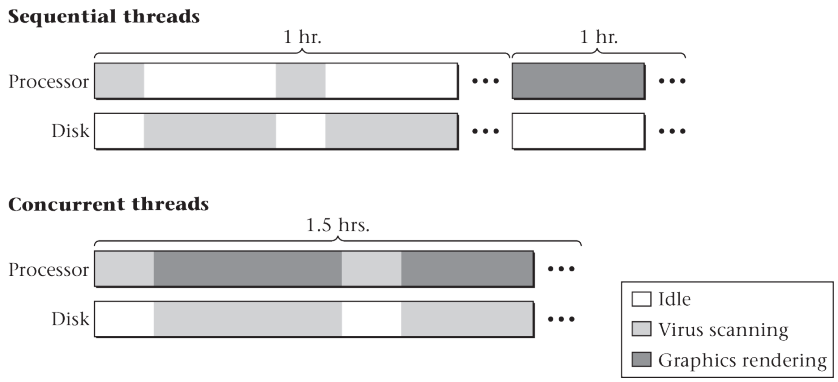

Figure 3: "Interleaved execution example" by Jens Lechtenbörger under CC BY-SA 4.0; SVG image refers to converted and cut parts of Figure 2.6 of a book by Max Hailperin under CC BY-SA 3.0. From GitLab

This figure illustrates the benefit of improved resource utilization resulting from multi-threading, which leads to higher overall throughput. Consider two threads and their resource demands, each taking 1 hour 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 does not need the disk at all. Clearly, the sequential execution of both threads takes 2 hours, which 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 second thread whenever the first one waits 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 1.5 hours.

Note that the total time of 1.5 hours 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 2 hours. Of course, both threads could also finish at different points in time.

3 Thread Switching

Let us now turn to details about thread switching.

3.1 Thread Switching for Multitasking

- OS schedules threads (details in 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

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 with time-slicing for execution on CPU cores.

By rapidly switching between various threads and allowing them to run for brief intervals, the OS can create the illusion of parallel executions even when only a single CPU core is available.

- Scheduling involves **context switches**
  - (Recall introduction and interrupts; now, as **thread switch**)
  - Remove current thread from CPU
    - **Remember computation state** (in TCB: program counter, register contents, stack, ...)
  - **Dispatch** next thread to CPU
    - **Restore computation state**

If the time slice for a thread ends, that thread 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, a pointer to the stack, and other information. To resume this computation later on, the OS needs to save the state of that thread somewhere, before another thread can be executed. Similarly, the new thread may be in the middle of its own computation, whose state was saved previously by the OS.

The switch from one thread, through the OS, to another thread with saving and restoring of state is an example of a context switch.

3.2 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
    - **program counter**; where to execute next machine instruction
    - **stack**; current position given by **stack pointer (SP)**
  - Besides: priority, scheduling information, blocking events (if any)

When multiple threads are executed on a single CPU core, they all use the same CPU registers, including program counter and stack pointer. Hence, it is crucial to save the values of these registers when transitioning between threads to prevent incorrect computation results.

Additionally, specific attributes such as priority level, scheduling details, and blocking events are associated with each individual thread.

- OS uses block of memory for housekeeping, called **thread control block** (TCB)
  - One for each thread
    - Storing register contents, stack pointer, program counter, ...

The operating system employs a region of memory known as the thread control block, TCB for short, for managing various aspects related to each thread's execution. A dedicated TCB exists for every thread, storing essential data like the content of CPU registers, stack pointer, and program counter.

4 Java Threads

Let us create threads in Java.

4.1 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

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, the second step requires creating an instance of class `Thread`, which is then started in the third step with method `start()`, which in turn eventually calls `run()`.

The next slide shows an example for these three steps.

- 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

Beyond the previous basic recipe for the creation of threads, Java features more functionality. Some pointers are listed here.

4.2 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 executes 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.

This sample program 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 the following: "Please take me away for the specified amount of milliseconds and give the CPU to someone else".

Which outputs do you expect at what points in time?

4.3 Self-Study Task: CPU Usage

Aspect of this task are available for self-study in Learnweb.

- Compile and run source code of Simpler2Threads
  - Explain its output. Maybe ask in Learnweb.
  - Maybe try out debugger, e.g., `jdb` with OpenJDK tools:
    - jdb Simpler2Threads
    - help
    - stop in Simpler2Threads.main
    - run
    - threads
    - stop in MyThread.run
    - step
    - threads

Please compile and run our sample program.

In general, a debugger helps to understand what is happening when. Your favorite programming environment probably includes a debugger. Also, several Java implementations come with a simple debugger, for which sample commands are shown here. Note that `step` allows executing an individual step of the program. Please try this out.

5 Server Models

- Web server "talks" HTTP with browsers
  - Simplified
    - Lots of browsers ask per GET method for various web page parts
    - Server responds with HTML files and other resources

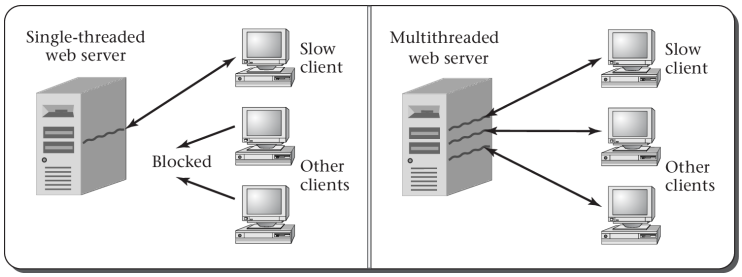

Figure 4: "Figure 2.5 of cite:Hai19" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

Let us now look at sample use cases for threads to implement servers, based on the example of web servers. (The reasoning outlined subsequently also applies other servers, e.g., database servers or messaging servers.)

Briefly, web servers talk with web browsers and other clients using the protocol HTTP. In that protocol, clients may use so-called GET requests to ask for resources, e.g., images, JavaScript or HTML files, or pieces of data in general.

As a thought experiment, suppose that you implemented a web server using a single thread. For a single web page, the browser typically asks for a sequence of resources. These resources need to be retrieved from disk and transmitted over the Internet, before an entire page can be rendered by the browser. Suppose that the server processes this entire sequence using blocking I/O operations before turning to the next client; then, complex pages, network latency, and slow clients will cause long delays for other clients. Consequently, web servers are not built this way.

Let us consider potential alternatives next.

5.1 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.

5.2 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 (Barroso et al. 2017)
- 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 some web servers. 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 full 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."

5.3 Worker Thread Pool

- Upon program start: Create **set** of **worker** threads
- Client requests received by **dispatcher** thread
  - Requests recorded in data structure with work items
- 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 data structure with work items.

If a worker is currently idle, i.e., it has nothing to do, it checks whether outstanding work items exist. If so, the worker picks one, removes it from the data structure, 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.)

5.3.1 Aside: Virtual Threads in Java

- Beyond class topics
- Java 19 introduced **virtual threads** in preview API
  - See JEP 436 or blog post for details
  - Updates for Java 21 in JEP 444
  - 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

In addition to course topics, there have been significant advancements related to multi-threading in Java. For instance, Java 19 introduced virtual threads, which were refined for Java 21.

Virtual threads aim at limiting overhead costs typically encountered during conventional multi-threading approaches. They facilitate the implementation of a thread-per-client model

while using blocking I/O operations. By mapping numerous virtual threads onto a limited pool of OS threads, the runtime limits the overhead incurred by the number of OS threads.

When a virtual thread encounters a blocking I/O event, the Java runtime handles this situation by performing a non-blocking OS call instead. Consequently, the virtual thread gets suspended temporarily until it can be resumed at a later time. Meanwhile, another unrelated virtual thread can be executed on the same underlying OS thread.

6 Conclusions

Let us conclude.

6.1 Summary

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

Threads are OS management units that represent individual instruction execution sequences within a program.

Multithreading can improve resource utilization, responsiveness, and modular design in the presence of concurrency.

The use of preemptive multithreading with housekeeping performed by the OS allows for thread switching, but this comes with some overhead.

When designing systems, one must make decisions such as whether to implement blocking or non-blocking I/O mechanisms and how many threads to use for servers.

Bibliography

Barroso, Luiz, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. "Attack of the Killer Microseconds." *Cacm* 60 (4): 48–54. https://dl.acm.org/citation.cfm?id=3015146.

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

The bibliography contains references used in this presentation.

License Information

Source files are available on GitLab (check out embedded submodules) under free licenses. Icons of custom controls are by @fontawesome, released under CC BY 4.0.

Except where otherwise noted, the work "Threads", © 2017-2024 Jens Lechtenbörger, is published under the Creative Commons license CC BY-SA 4.0.

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