OS03: Threads

Based on Chapter 2 of [Hai17]

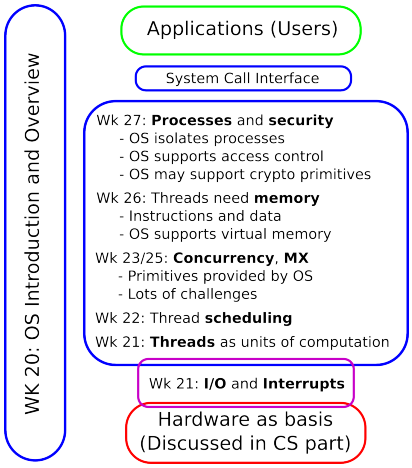
Jens Lechtenbörger

Computer Structures and Operating Systems 2019

1 Introduction

1.1 OS Plan

- OS Motivation (Wk 20)
- OS Introduction (Wk 20)
- Interrupts and I/O (Wk 21)
- Threads (Wk 21)
- Thread Scheduling (Wk 22)
- Mutual Exclusion (MX) (Wk 23)
- MX in Java (Wk 23)
- MX Challenges (Wk 25)
- Virtual Memory I (Wk 26)
- Virtual Memory II (Wk 26)
- Processes (Wk 27)
- Security (Wk 27)
- Wrap-up (Wk 28)



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?

1.3 Learning Objectives

- Explain thread concept, thread switching, and multitasking
  - Including reasons for threads
    - With learning objectives for BattleThreads, rephrased from [Hil+03]
      - Explain distinctions between threads and processes
      - Explain advantages of a multithreaded organization in structuring applications and in performance
  - Including stack for housekeeping
  - Including states (after upcoming presentation)
- 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 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?

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)
  - 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
    - [Hai17]: 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)

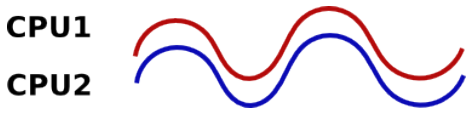


Figure 1: Figure under CC0 1.0

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

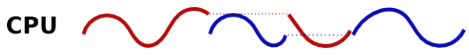


Figure 2: Figure under CC0 1.0

  - 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
- **CPU bound**
  - Threads spending most time executing code
  - Run for longer periods of time
    * Until preempted by scheduler
    * E.g., graph rendering

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
    * Since Java 5.0
    * With `Callable<V>`, `Future<V>` and service methods of `Executor`
      · Worker Thread Pool

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.");
  }}
```

3.3 JiTT Assignments

Work through the following tasks. Submit solutions in Learnweb.

3.3.1 CPU Usage for Polling and Interrupts

- (Nothing to submit here; maybe ask questions)
- Which output messages do you expect for Simpler2Threads at what points in time?
  - E.g.: The first after 3s, the second after 5s; the first after 3s, the second after 8s; the first after 5s, the second after 8s; other.
  - Compile and run the program (source code) to verify your expectation.
  - 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

3.3.2 Behavior and Inspection of Threads

Change the code by adding infinite loops after the `println()` invocations so that the threads run endlessly.

- Try to identify the two threads (parent and child) using some tool to inspect processes and threads; typically they show numerical IDs, CPU usage, and other details.
- This task is challenging and requires research ("forschendes Lernen" in German).
  - Depending on your OS, default tools may hide threads from you. Then more powerful alternatives need to be found, installed, and used properly.
  - You also need to figure out how to identify *your* threads (in contrast to others that you may find).
  - If you find the identification of Java threads boring, note that using the same tools you can identify malware on your system, e.g., malware mining cryptocurrency.
- What OS and what tool do you use? What do you observe?

3.3.3 Ticket to Exam - Task 1

Suppose that (a) the code of Simpler2Threads (without infinite loops) is changed to create more threads that output strings after some sleep time (maybe with a loop in `main()` to create and start child threads) and (b) the CPU hardware is able to execute up to four threads in parallel. At what time do you expect the output strings to appear if up to 4 threads are created? What output behavior do you expect if 20 threads are created?

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 waiting for external event (e.g., disk or network I/O), allow other threads to continue
      - Next slide
    - Keep multiple 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

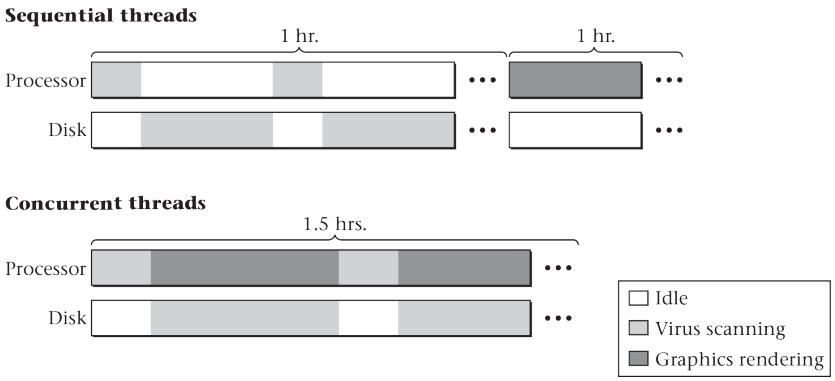


Figure 3: "Interleaved execution example" by Jens Lechtenbörger under CC BY-SA 4.0; SVG image refers to converted and cut parts of Figure 2.6 of a book by Max Hailperin under CC BY-SA 3.0. From GitLab

This figure illustrates the benefit of improved resource utilization resulting from 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

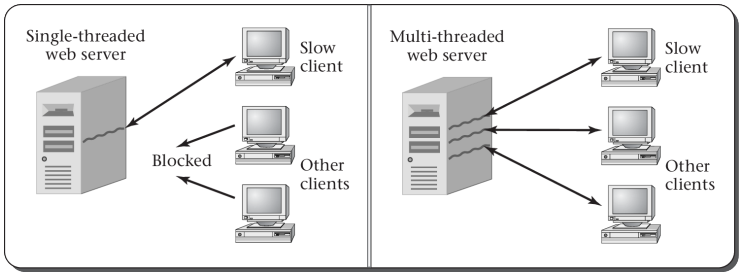


Figure 4: "Figure 2.5 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

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

- With multiple threads, OS needs to decide which to execute when → Scheduling (later lecture)
  - (Similar to machine scheduling for industrial production, which you may know from operations management)
- After that decision, a **context switch** (thread switch) takes place
  - Remove thread A from CPU
    - **Remember A's state** (instruction pointer/program counter, register contents, stack, ...)
  - **Dispatch** thread B to CPU
    - **Restore B's state**

5.1.1 Thread Switching with `yield`

In the following

- First, simplified setting of voluntary switch from thread A to thread B
  - Function `switchFromTo()`
    - Details discussed in class meeting

- 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

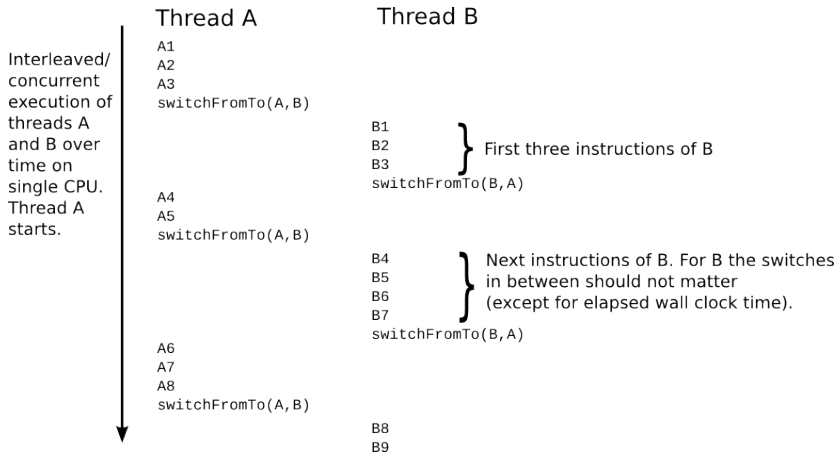


Figure 5: "Interleaved execution of threads. Based on Figure 2.7 of book by Max Hailperin, CC BY-SA 3.0." by Jens Lechtenbörger under CC BY-SA 4.0; from GitLab

5.3 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

5.4 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
      · Upcoming presentation

5.5 Multitasking Overhead

- OS performs scheduling, which takes time
- Thread switching creates **overhead**
  - Minor sources: Scheduling costs, saving state
  - Major sources: Cache pollution, cache coherence protocols
    * 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

5.6 JiTT Assignments

Answer the following questions in Learnweb.

5.6.1 Ticket to Exam - Task 2

- Submit your solutions to Exercise 2.3 of [Hai17].
  - In previous years, I received shocking complaints concerning this task. I decided to keep it unchanged. Yes, you need to locate the book, and you need to locate that numbered exercise in the book.

    By the way, lots of students like that book's explanations. If you see the reference "[Hai17]" in presentations (or source code), you now know where to look for further/different explanations.

5.6.2 Questions, Feedback, and Improvements

- What did you find difficult or confusing about the **contents** of the presentation? Please be as specific as possible. For example, you could describe your current understanding (which might allow us to identify misunderstandings), ask questions that allow us to help you, or suggest improvements (maybe on GitLab). You may submit individual questions as response to this task or ask questions in our Riot room and the Learnweb forum. Most questions turn out to be of general interest; please do not hesitate to ask and answer in forum and Riot room. If you created additional

original content that might help others (e.g., a new exercise, an experiment, explanations concerning relationships with different courses, ...), please share.

5.7 Recall: Stack

- Stack = Classical data structure (abstract data type)
  - LIFO (last in, first out) principle
  - See Appendix A in [Hai17] 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)

5.7.1 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.)

6 In-Class Meeting

6.1 Server Models

6.1.1 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

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

- **Single thread** (left-hand side of Figure 2.5; thought experiment, not for use)
- **Sequential** processing of client requests
  - Long idle time during I/O → Not suitable in practice

6.1.3 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

6.1.4 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)

6.1.5 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
  - If more client requests than worker threads, then potentially long delays

6.2 `switchFromTo()`

- Recall use of `switchFromTo()` in interleaved execution
- Some implementation ideas follow

6.2.1 Thread Control Blocks (TCBs)

- All threads share the same CPU registers
  - Obviously, register values need to be **saved** somewhere to avoid incorrect results
  - Also, each thread has its own
    - **stack**; current position given by **stack pointer (SP)**
    - **instruction pointer (IP)**; where to execute next machine instruction
  - Besides: priority, scheduling information, blocking events (if any)
- Use some 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

6.2.2 Major Stack Use Case

Housekeeping for **function calls**

- Pass **arguments** and **return values** of functions
- Before calling a function, `push` arguments
  - And other pieces of information
- In called function, `pop` and use arguments
- Before returning from function, `push` return value
- In calling function, `pop` and use return value

6.2.3 Details of `switchFromTo()`

As threads A (variable `outgoing`) and B (variable `next`) have separate stacks anyways, use them to store registers (see Sec. 2.4 in [Hai17]):

```
  push each register on the (outgoing thread's) stack
  store the stack pointer into outgoing->SP
  store label L's address into outgoing->IP
  load the stack pointer from next->SP
  load in next->IP and jump to that address
L:
  pop each register from the (resumed outgoing thread's) stack
```

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.

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

[Hil+03] John M. D. Hill et al. "Puzzles and Games: Addressing Different Learning Styles in Teaching Operating Systems Concepts". In: *SIGCSE Bull.* 35.1 (Jan. 2003), pp. 182–186. ISSN: 0097-8418. DOI: 10.1145/792548.611964. URL: https://dl.acm.org/citation.cfm?doid=792548.611964.

License Information