Course 02158

The End

Hans Henrik Løvengreen

DTU Compute

Fall 2024

Wrapping up Concurrent Programming Fall 2024

What we didn't make

- Concurrency in language *XX*, e.g. Python, Rust
- The *coroutines* of Kotlin
- The *isolates* of Dart
- The *virtual threads* in Java

What we did cover well

- Classical *concurrent programming notions*
- Basic *concurrency threory*

What's next?

- Courses

Exam

- Form and tips

Python Concurrency

History

- Developed from the 1980'ies by Dutch computer scientist Guido van Rossum
- Version 2.0 (2000), version 3.0 (2008). version 3.13 (2024)
- Open source reference implementation CPython

Python characteristics

- Dynamically typed, interpreted (it has a cost)
- Elements of object-oriented and functional programming
- Myriads of libraries, especially for data processing

Python Concurrency

- A *threading library* providing *user space threads*
- Shared state with *locks* and *condition variables*
- Also *semaphores* and *barriers* are available
- Only a single OS thread is running the Python threads (GIL)
- An *executor* notion (using OS processes) is also provided.
- Various *message passing* libraries, e.g. MPI

The Rust Language

History

- Developed by a few people under auspices of Mozilla from 2010
- Open Source — version 1.0 May 2015 (now 1.83)
- Still under development: See `rust-lang.org`

Rust characteristics

- Combines functional and procedural styles with object notions
- Aims at efficient and *safe systems programming* (the "new C")
- Compiled language, no garbage collection, reference counting
- Tries to prevent memory and concurrency issues by *strong typing* and *ownership*
- Exceptions → explicit errors, classes → traits

Rust Concurrency

- Thread-based concurrency with *mutex'es* and *conditions*
- Message passing over *channels*, *selection*

Rust Example: Simple communication

```
fn main() {
    let (tx, rx) = mpsc::sync_channel(0);

    for i in 0..10 {
        let tx1 = tx.clone();

        thread::spawn(move || {
            let answer = i * i;

            tx1.send(answer).unwrap();
        });
    }

    for _ in 0..10 {
        println!("{}", rx.recv().unwrap());
    }
}
```

Concurrency in Kotlin

History

- Developed by JetBrains for Android development
- Runs on the JVM — blends with Java code
- Open Source — version 1.0 in February 2016 (now 2.1)
- From 2019 Google's official Android development language

Kotlin characteristics

- Combines functional and procedural styles with object notions
- Mostly syntactic simplifications

Kotlin Concurrency

- May use (Java) thread notions
- Has recently provided a light-weight concurrency notion: *coroutines*
- Coroutines are executed by *coroutine dispatchers* ($\sim$ thread pools)
- Couroutines use *suspending operations* for synchronization

Kotlin: Coroutines

- *Coroutines* are light-weight asynchronous executions
- Coroutines may be started by **launch** { ... } or **async** { ... }
- ```
  fun main() {
      suspend fun P(name : String ) { repeat(10) { println(name) } }
      runBlocking {
          launch { P("Huey") }
          launch { P("Dewey") }
          launch { P("Louie") }
      }
      println("Done")
  }
  ```
- Uses *scoped concurrent execution*
- Coroutines must use *suspending functions*
- Suspension = "lightweight blocking"

Kotlin: Synchronization

- Coroutines may *share data*
- Shared data may be protected by *mutual exclusion* (locks)
- ```
  fun main() {
    val region = Mutex()
    var x = 0;

    suspend fun P() {
      repeat (1000) { region.withLock { x++ } }
    }

    runBlocking {
        launch { P() }
        launch { P() }
    }
    println("x␣=␣\$x")
  }
  ```
- No condition queues (yet)

Kotlin: Message Passing

- Coroutines may communicate via *channels*
- Channels may be *synchronous*, *buffered*, *asynchronous* or *confloated*
- ```
  fun main() {
    val c = Channel<Int>()

    suspend fun P() {
      repeat (10) { c.send(it); delay(500) }
    }

    runBlocking {
        launch { P() }
        launch { P() }

        repeat(20) { println( c.receive() ) }
    }
  }
  ```
- Experimental *selection* construct

Isolates in Dart

Dart History

- Developed within Google by Lars Bak ('V8')
- Aiming at being a safe JavaScript substituion for Chrome — dropped in 2015
- Simple, OO-oriented language with both static and dynamic types
- From 2015 chosen for Flutter cross-platform app development (Google too!)

Dart Concurrency

- Single-threaded event-handling using external concurrency with async/await
- Several concurrent *isolates* that may serve each other
- Similar to the *actor* notion
  ```

Dart Asynchronous Programming

With futures

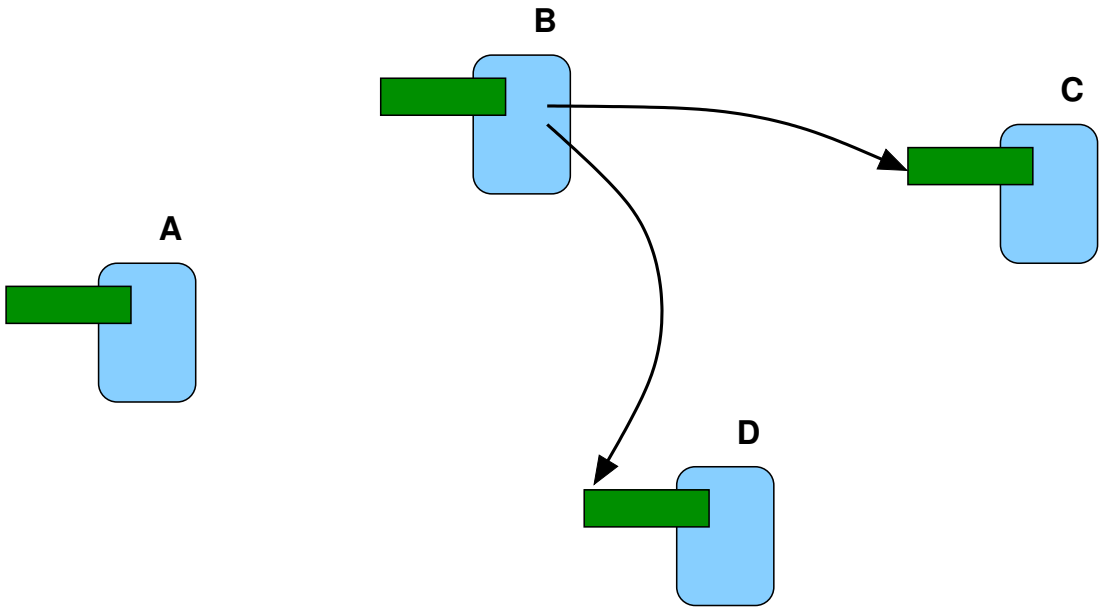
```
handleCall() {
  Future<String> name = getNameAsync();
  Future<int> no = name.then((nm) => lookupPhoneNoAsync(nm));
  no.then( (x) => makeCall(x) );
}
```

With async/await

```
handleCallAsync() async {
  String name = await getNameAsync();
  int no = await lookupPhoneNoAsync(nm);
  makeCall(no);
}
```

- An **async** function always returns a `Future`
- **await** can only be used within an **async** block

Actor Model



Java Virtual Threads

- Project [Loom] under OpenJDK
- Threads as they should always have been!
- Alternative to asynchronous programming with `async/await` syntax
- Introduces *virtual threads* which act like threads
  ... but without the cost of an OS thread
- Virtual threads are executed by scheduling underlying OS threads
- Virtual threads may block/synchronize (without a penalty)
- Most synchronization classes work — except **synchronized** (for now)
- Appeared first as a *preview* in Java 19 (Aug 2022)
- Virtual threads are (just) concurrent processes
- Enables a simple *thread-per-request* concurrency paradigm

Java Virtual Threads: Example

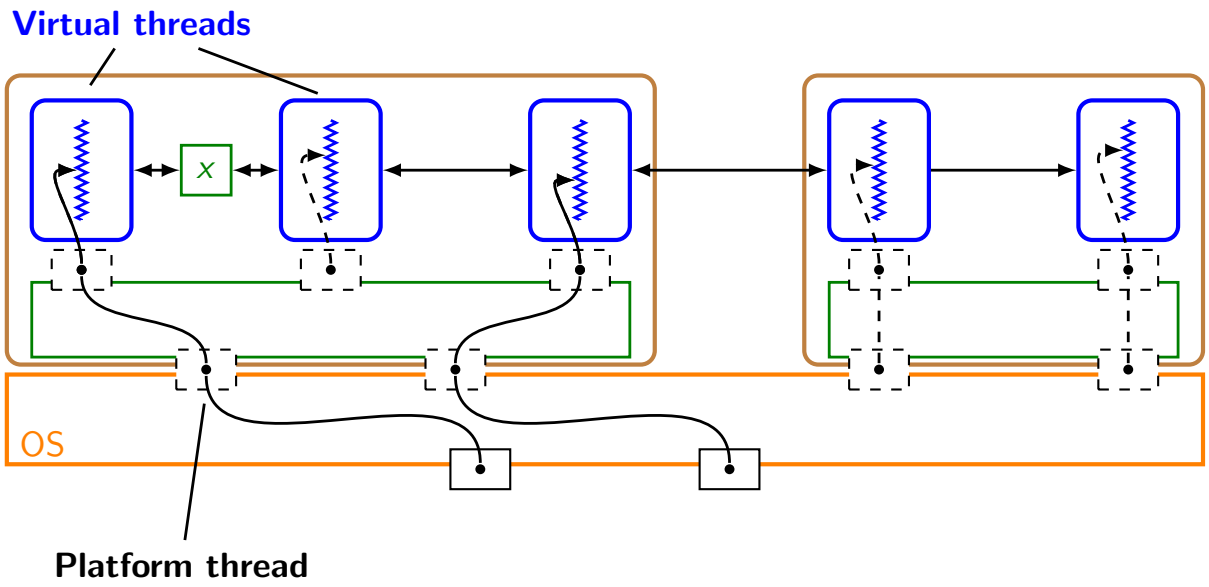
```
public class HelloWorld implements Runnable {

    public void run() {
        System.out.println("Hello World");
    }

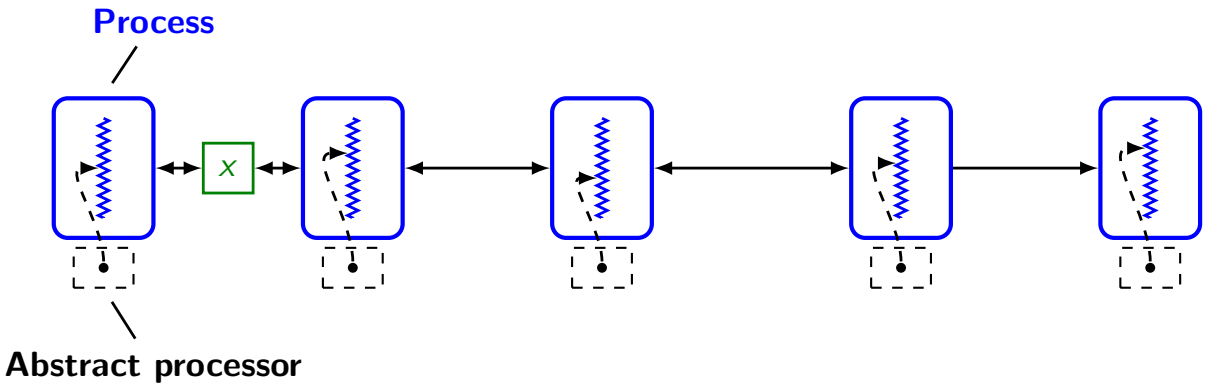
    public static void main(String[] args) throws InterruptedException {

        Thread thread = Thread.ofVirtual().start(new HelloWorld());
        thread.join();
    }
}
```

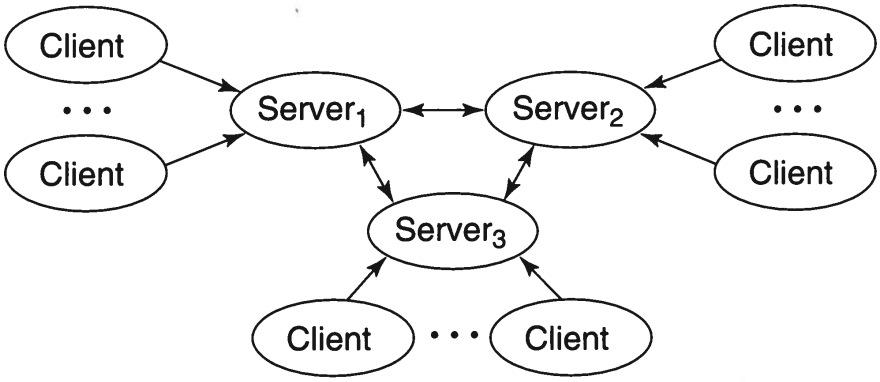
Java Virtual Threads



Abstract Concurrency Model



Replicated Data [Andrews 8.4.2]



Consistency notions

Sequential Consistency [Lamport 1978]

- All participants see same global sequence of operations
- May be implemented by a *central server* — does not scale well

Eventual Consistency [Shapiro et.al. 2011]

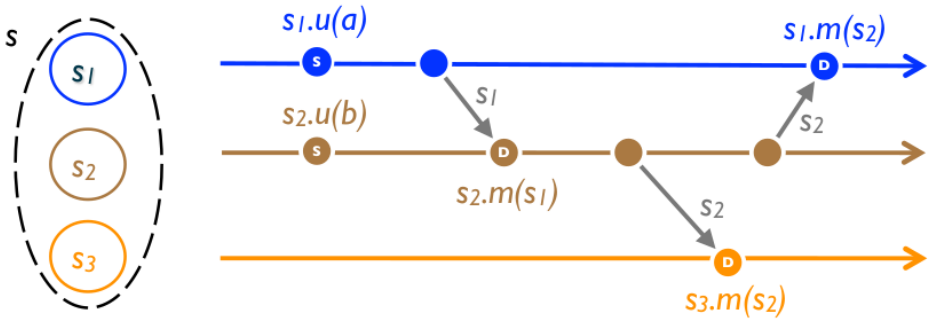
- Make local updates, spread state, merge external states
- EC: All participants eventually see the same stable data

> **Definition 2.2** (Eventual Consistency (EC)). **Eventual delivery:** *An update delivered at some correct replica is eventually delivered to all correct replicas:* $\forall i, j : f \in c_i \Rightarrow \Diamond f \in c_j$.
> **Convergence:** *Correct replicas that have delivered the same updates eventually reach equivalent state:* $\forall i, j : \Box c_i = c_j \Rightarrow \Diamond\Box s_i \equiv s_j$.

- External updates may *conflict* with local ones ⇒ *roll-back*

Conflict-free Replicated Data Type (CRDT)

- Strong EC: If updates *commute* (do not conflict), the merged state is final



Examples of CRDTs

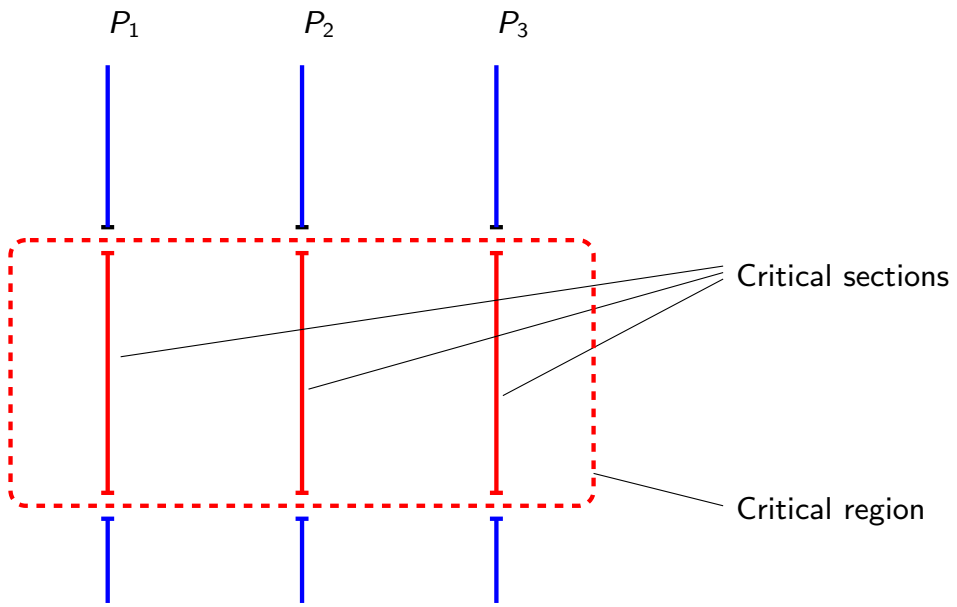
- A counter with atomic increments
- Tree-structured text for cooperative editing

What is Concurrent Programming?

ChatGPT

> *Concurrent programming refers to a computing technique where multiple tasks are executed simultaneously, enabling efficient utilization of system resources. It involves managing the execution of tasks to make progress on several operations concurrently. This approach is crucial in modern software development, utilizing threads, processes, or other mechanisms to execute tasks concurrently, improving performance and responsiveness. Efficient management and synchronization are essential to prevent conflicts and ensure seamless operation of concurrent tasks within a program.*

Critical Regions and Critical Sections



Critical Region vs. Critical Section?

ChatGPT

Critical Region: *Refers to any part of the code where shared resources are accessed or modified by multiple concurrent processes or threads. It's a broader term encompassing any segment of code where data is shared and requires exclusive access to prevent conflicts.*

Critical Section: *Specifically denotes the segment within a program where shared resources are being accessed or modified, requiring mutual exclusion to ensure only one process or thread can access it at a time. It's a more focused term referring to the actual part of the code that needs synchronized access.*

In essence, a critical section is a specific instance of a critical region. All critical sections are critical regions, but not all critical regions may represent critical sections, as critical regions can encompass multiple sections where shared resources are utilized.

Concurrent Programming Aims

After the course you should

- Understand *concepts* and *notions* of concurrency.
- Know *abstract models* of concurrency and principles of *verification*
- Be well versed in synchronization and communication *mechanisms*
- Know about underlying *implementation principles*
- Be aware of concurrency *pitfalls* and principles for avoiding them
- Know how to *test* concurrent programs
- Be skilled in writing *multi-threaded Java* programs
- Know a number of concurrency *SW-architectures*

Concurrent Programming

What did you learn?

- Basic *notions*: processes/threads, synchronization, monitors, message passing
- Concurrency awareness — especially of pitfalls
- Skills in using Java threads ($\sim$ C$^{\#}$, Python, ...)
- Should be able to use concurrency in many other languages, Pthreads, ...
- Able to utilize *multi-core* and *HPC* machines

Where can you learn more?

- Parallel Computation: **02258 Parallel Computer Systems** (E5A)
  02614 High-perf. Computing (GPU, OpenMP) (Jan)
  02616 Large Scale Modelling (MPI) (F3A)
- Real-time and embedded systems: **02226 Networked Embedded Syst.** (E1B)
  Robotics (DTU Electro)
- Distributed systems: **02148 Coordination in Distributed Appl.** (Jan)
  02225 Distributed Real-Time Systems (F4B)
- Operating systems: **02159 Operating Systems** (E1A)

Concurrency Theory

What did you learn?

- Models of concurrency: *Petri-nets*, *interleaving of atomic actions*
- Basic properties: *Invariants*
- Knowledge of property languages, e.g. *Temporal Logic*
- Knowledge of modelling languages: *CSP* (process algebras, ...)
- Knowledge of verification tools: SPIN, (∼ UPPAAL, ...)

Where can you learn more?

- Petri Nets: **02162 Software Engineering 2** (E3)
  02269 Process Mining (E5A)
- Verification tools: **02245 Program Verification** (E1B)
  02246 Model Checking (E4B)
  02256 Automated Reasoning (F1B)
- Temporal (modal) logic: **02287 Logical Theories for Uncertainty** (E2B)

10 Rules of Concurrent Programming

1. Use concurrency deliberately, sparingly and *with care*
2. Stick to *message passing* if possible
3. Analyze which *objects* may be *reached* by each thread
4. Always *protect shared data* (if mutable) preferably by *monitors*
5. Give away only *immmutable data objects*
6. *Document* the *concurrency behaviour* of classes (thread-safety)
7. Use *update functions* rather than getters/setters on shared data
8. Associate an (informal) *invariant* with each critical region (lock)
9. Think and design in terms of *large atomic operations*
10. Optimize (e.g. parallelize) only if *need demonstrated*

If only one rule?

Beware of race conditions

Race Conditions

- Race conditions are *latent bombs* in your program

According to Murphy

- Race conditions will never show up in testing
  - ... but will appear in production
    - ... at the worst possible moment
- May jeopardize both *safety* and *security*

So

- Race conditions will *haunt you* (like I did) ... till they are eliminated
- Avoid them by
  - ... keeping your solution *simple and obviously without errors*
    - ... rather than *complex but with no obvious errors*

[∼ C.A.R. Hoare]

Concurrent Programming Exam

Form

- **4 hours**
- 4-6 problems within concurrent prog. concepts and basic theory
- Smaller and larger questions
- Coding of smaller program parts using pseudo-code (not Java)
- Often one problem with a larger concurrency setting

Preparation hints

- See the *exam readings (syllabus)*
- Do exercise classes and home works (again?)
- Do not look at solutions till you have tried yourself
- Practice a couple of exam sets in real time

Exam Guidelines 2024

Form

- Exam will take place in **Building 116/127 at 9.00** [see eksamensplan.dtu.dk]
- Only written material (books, notes, . . .) — i.e. **no computers**
- Exam paper is distributed **on paper only**
- Hand-in is physical on paper — practice your handwriting
- Use of pencil is acceptable
- Do not repeat the problem text
- Use time propertional to weight of problems

Contents

- Questions with "Determine . . . " should be justified (why or how)
- If in doubt: Read the question again — how would it make sense?
- If still in doubt — write down your understanding

The Universal Pieces of Advice for Exams

Read the questions carefully!

Don't panic!

Good luck