IN5170 Models of Concurrency

Lecture 1: Introduction

S. Lizeth Tapia Tarifa, Andrea Pferscher, Juliane Päßler, Tobias John

August 19th, 2024

University of Oslo

Motivation

Concurrency is Everywhere and Challenging

- Multiple computations at the same time
- Networks, programs, even single-threaded application!
- Source of serious flaws in systems: *safety*, *security*, *privacy*
- Hard or impossible to test concurrent systems – enormous amount of executions

Motivation

Concurrency is Everywhere and Challenging

- Multiple computations at the same time
- Networks, programs, even single-threaded application!
- Source of serious flaws in systems: *safety*, *security*, *privacy*
- Hard or impossible to test concurrent systems – enormous amount of executions

Concurrency has Many Facets

- Programming languages use many different concurrency mechanisms
  - Multi-threading with shared state
  - Message passing, channels
  - Go-routines, async/await, futures
  - ...
- Modern languages build on these concurrency mechanisms, cf. Go, Swift, Rust, ...
- What is the essence of these mechanisms and how should we use them?

What this course is not about . . .

- Insight into some basic concepts and techniques for specification and reasoning for parallel processes,
- with a focus on
  - collaborative threads
  - distributed and object-oriented systems

IN5170 - Models of Concurrency

What this course is not about ...

- Insight into some basic concepts and techniques for specification and reasoning for parallel processes,
- with a focus on
  - collaborative threads
  - distributed and object-oriented systems

Learning Outcomes

- know different mechanisms of parallelism, including shared variables and communication-based
- have good insight into typical problems with parallel systems like deadlock, fairness, etc.
- be able to specify, design and analyze a parallel system so that it meets the desired properties
- be able to contemplate the characteristics of a parallel system

General Info

Structure

- **Part 1:** Shared Memory (and Java)
- **Part 2:** Message Passing (and Go)
- **Part 3:** Analyses and Tool Support (and Rust)

General Info

Structure

- **Part 1:** Shared Memory (and Java)
- **Part 2:** Message Passing (and Go)
- **Part 3:** Analyses and Tool Support (and Rust)

Exercises, Obligs, Exam

- (Almost) weekly exercise sessions, two obligatory assignments, ...
- **Exam 28.11.2024**, check website for details

General Info

Structure

- **Part 1:** Shared Memory (and Java)
- **Part 2:** Message Passing (and Go)
- **Part 3:** Analyses and Tool Support (and Rust)

Exercises, Obligs, Exam

- (Almost) weekly exercise sessions, two obligatory assignments, ...
- **Exam 28.11.2024**, check website for details

Literature

- First part: **G. R. Andrews. Foundations of Multi-threaded, Parallel, and Distributed Programming**. Addison Wesley, 2000 (Chapters 1 to 10).
- Second and third part: Slides from the lectures, supplementary material

General Info

Rooms

- **Lectures:** 10:15 on Mondays in Postscript
- **Exercises:** 12:15 on Wednesday in Logo
- First exercise next week

Teaching Team

For questions, contact as follows:

- **Silvia Lizeth Tapia Tarifa** (Lectures)
- **Andrea Pferscher** (Lectures)
- **Juliane Päßler** (Exercises, Obligs)
- **Tobias John** (Exercises, Obligs)

Today's Agenda

- Overview ✓
- Motivation and Considerations
- Critical Sections and the `await`-language
- A taste of concurrency in Java

Reading material: Chapter 1 of Andrews, slides about Java

Shared Memory Concurrency

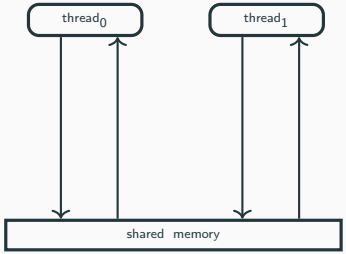
Parallel Processes

- **Sequential program:** one thread of control, full control over the whole memory
- **Parallel/concurrent program:** several threads of control, which *need to exchange information and coordinate execution*

Communication between processes

We will study two different ways to organize communication between processes:

- Reading from and writing to *shared variables* (Now)
- Communication with *messages* between processes (Part 2)



Course Overview — Part 1: Shared variables

Content

- Problems that occur in concurrent systems with shared variables
- Patterns to solve these problems

Course Overview — Part 1: Shared variables

Content

- Problems that occur in concurrent systems with shared variables
- Patterns to solve these problems

- Atomic operations
- Interference
- Deadlock, livelock, liveness, fairness
- Locks, critical sections and (active) waiting
- Semaphores and passive waiting
- Monitors
- Java: threads and synchronization
- Rust: ownership of program variables

Why Shared Variables?

Why shared (global) variables?

- Reflected in conventional hardware architectures, e.g., multi-core systems
- Reflected in most programming languages as a default (e.g., multi-threading).

Why Shared Variables?

Why shared (global) variables?

- Reflected in conventional hardware architectures, e.g., multi-core systems
- Reflected in most programming languages as a default (e.g., multi-threading).

Notes

- Even with a single processor and one thread, you may want to use many processes, in order to get a natural partitioning, e.g., several active windows at the same time

Why Shared Variables?

Why shared (global) variables?

- Reflected in conventional hardware architectures, e.g., multi-core systems
- Reflected in most programming languages as a default (e.g., multi-threading).

Notes

- Even with a single processor and one thread, you may want to use many processes, in order to get a natural partitioning, e.g., several active windows at the same time
- As all concurrency: potentially greater efficiency and/or better latency if several things happen/appear to happen "at the same time".

Why Shared Variables?

Why shared (global) variables?

- Reflected in conventional hardware architectures, e.g., multi-core systems
- Reflected in most programming languages as a default (e.g., multi-threading).

Notes

- Even with a single processor and one thread, you may want to use many processes, in order to get a natural partitioning, e.g., several active windows at the same time
- As all concurrency: potentially greater efficiency and/or better latency if several things happen/appear to happen "at the same time".
- Natural interaction for tightly coupled systems

Simple Example

Consider 3 global variables x, y, and z and the following program: x := x+z; y := y+z;

- Can we use parallelism here (without changing the results)?

Simple Example

Consider 3 global variables x, y, and z and the following program: x := x+z; y := y+z;

- Can we use parallelism here (without changing the results)?
- If operations can be performed *independently* then performance may increase

Simple Example

Consider 3 global variables x, y, and z and the following program: x := x+z; y := y+z;

- Can we use parallelism here (without changing the results)?
- If operations can be performed *independently* then performance may increase
- What are the results?

Simple Example

Consider 3 global variables x, y, and z and the following program: x := x+z; y := y+z;

- Can we use parallelism here (without changing the results)?
- If operations can be performed *independently* then performance may increase
- What are the results?

Await

```
Before: {x = a & y = b & z = c}
          x := x+z; y := y+z;
After:  {x = a+c & y = b+c & z = c}
```

Simple Example

Consider 3 global variables x, y, and z and the following program: x := x+z; y := y+z;

- Can we use parallelism here (without changing the results)?
- If operations can be performed *independently* then performance may increase
- What are the results?

Await

```
Before: {x = a & y = b & z = c}
          x := x+z; y := y+z;
After:  {x = a+c & y = b+c & z = c}
```

Pre/post-conditions

- We use brackets {...} to describe conditions before or after a statement
- These conditions are describing the state, but are not executed
- Java has `assert` that can check such conditions at runtime and the JML language to specify more complex conditions than expressions

Parallel Operator ||

- Consider shared and non-shared program variables, assignment.
- We extend the language with a construction for *parallel composition*:

Await

```
co S1 || S2 || ... || Sn oc
```

Parallel Operator ||

- Consider shared and non-shared program variables, assignment.
- We extend the language with a construction for *parallel composition*:

Await

```
co S1 || S2 || ... || Sn oc
```

- The execution of a parallel composition happens via the *concurrent* execution of the component processes $S_1, \ldots, S_n$.
- *Terminates* normally if all component processes terminate normally.

Parallel Operator ||

- Consider shared and non-shared program variables, assignment.
- We extend the language with a construction for *parallel composition*:

Await

```
co S1 || S2 || ... || Sn oc
```

- The execution of a parallel composition happens via the *concurrent* execution of the component processes $S_1, \ldots, S_n$.
- *Terminates* normally if all component processes terminate normally.

Example

Await

```
{x = a & y = b & z = c}
 x := x+z; y := y+z;
{x = a+c & y = b+c & z = c}
```

Await

```
{x = a & y = b & z = c}
 co x := x+z || y := y+z oc
{x = a+c & y = b+c & z = c}
```

Simple Example in Java

Java

```
class C { public static int x = 0,  y = 0,  z = 5 }
...
new Thread ( () -> { x = x + z } ).start();
new Thread ( () -> { y = y + z } ).start();
```

Interaction Between Parallel Processes

Processes can *interact* with each other in *two* different ways:

- *Cooperation* to obtain a result
- *Competition* for common resources

Interaction Between Parallel Processes

Processes can *interact* with each other in *two* different ways:

- *Cooperation* to obtain a result
- *Competition* for common resources

To organize their interactions, we use *synchronization*

Synchronization

Synchronization *restricts* the possible interleavings of parallel processes
to avoid unwanted behavior and enforce wanted behavior.

Interaction Between Parallel Processes

Processes can *interact* with each other in *two* different ways:

- *Cooperation* to obtain a result
- *Competition* for common resources

To organize their interactions, we use *synchronization*

Synchronization

Synchronization *restricts* the possible interleavings of parallel processes
to avoid unwanted behavior and enforce wanted behavior.

Example

- Increasing *atomicity* and *mutual exclusion* (Mutex) to introduce *critical sections* which can *not* be executed concurrently
- *Condition synchronization* enforces that processes must wait for a specific condition to be satisfied before execution can continue.

Atomic Expressions JVM bytecode

JVM

Java is compiled down to JVM bytecode, which does not correspond 1-to-1 to machine instructions

Java

```
x++; // x is only local variable, declared as long
```

Atomic Expressions JVM bytecode

JVM

Java is compiled down to JVM bytecode, which does not correspond 1-to-1 to machine instructions

Java

```
x++; // x is only local variable, declared as long
```

How many machine instructions will this be?

Atomic Expressions JVM bytecode

JVM

Java is compiled down to JVM bytecode, which does not correspond 1-to-1 to machine instructions

Java

```
x++; // x is only local variable, declared as long
```

How many machine instructions will this be? **4-6**

Atomic Expressions JVM bytecode

JVM

Java is compiled down to JVM bytecode, which does not correspond 1-to-1 to machine instructions

Java

```
x++; // x is only local variable, declared as long
```

How many machine instructions will this be? **4-6**

```
LLOAD_1      // push value from local variable #1
LCONST_1     // push value 1
LADD         // add 2 top-most values
LSTORE_1     // store value into local variable #1
```

Atomic Expressions JVM bytecode

JVM

Java is compiled down to JVM bytecode, which does not correspond 1-to-1 to machine instructions

Java

```
x++; // x is only local variable, declared as long
```

How many machine instructions will this be? **4-6**

- Reference reads and writes are atomic
- Basic type reads and writes except `long` and `double` are guaranteed to be atomic
- On 64bit machines, `long` and `double` reads and writes *might* be atomic, *might* be two instructions

Concurrent Processes: Atomic Operations

Definition (Atomic)

An operation is atomic if it cannot be subdivided into smaller operations.

- We can ignore concurrency inside atomic operations as they cannot be interleaved
- A statement with at most one atomic operation, in addition to operations on local variables, can be considered atomic

Concurrent Processes: Atomic Operations

Definition (Atomic)

An operation is atomic if it cannot be subdivided into smaller operations.

- We can ignore concurrency inside atomic operations as they cannot be interleaved
- A statement with at most one atomic operation, in addition to operations on local variables, can be considered atomic

- What is atomic depends on the language/setting:
  fine-grained and coarse-grained atomicity.
- Accessing global variables is atomic for this lecture.
  (In general, this may not be the case, e.g., for `long int`.)
- Assignments `x := e` are *not* atomic

Atomic Operations on Global Variables

Enabling atomic operations on global variables is fundamental for shared memory concurrency

Atomic Operations on Global Variables

Enabling atomic operations on global variables is fundamental for shared memory concurrency

- Process *communication* may be realized by variables:
  a communication channel corresponds to a variable of type vector (or similar)
- Associated with global variables is a set of *atomic operations*
- Typically: read and write, in hardware, e.g. LOAD/STORE to registers
- Channels can also be seen as global variables: *send* and *receive*
- Atomic operations on a variable x are called x-operations

Atomic Operations on Global Variables

Enabling atomic operations on global variables is fundamental for shared memory concurrency

- Process *communication* may be realized by variables:
  a communication channel corresponds to a variable of type vector (or similar)
- Associated with global variables is a set of *atomic operations*
- Typically: read and write, in hardware, e.g. LOAD/STORE to registers
- Channels can also be seen as global variables: *send* and *receive*
- Atomic operations on a variable x are called x-operations

Our goal:

Mutual exclusion

Make *composed* statements atomic so they cannot happen simultaneously.

Atomic Operations on Global Variables

Enabling atomic operations on global variables is fundamental for shared memory concurrency

- Process *communication* may be realized by variables:
  a communication channel corresponds to a variable of type vector (or similar)
- Associated with global variables is a set of *atomic operations*
- Typically: read and write, in hardware, e.g. LOAD/STORE to registers
- Channels can also be seen as global variables: *send* and *receive*
- Atomic operations on a variable x are called x-operations

Our goal:

Mutual exclusion

Make *composed* statements atomic so they cannot happen simultaneously.

. . . but observe: the more atomic we make the program, the less parallel execution can occur!

Example

Await

```
                    P1              P2
{x = 0} co x := x+1 || x := x−1 oc {?}
```

Example

Await

```
                    P1              P2
{x = 0} co x := x+1 || x := x−1 oc {?}
```

Each statement actually consists of *3* operations, e.g., P_1 is

```
read x; inc; write x;
```

Example

Await

```
                    P1                P2
{x = 0} co x := x+1 || x := x−1 oc {?}
```

Each statement actually consists of *3* operations, e.g., P_1 is

```
read x; inc; write x;
```

Atomic x-operations:

Example

Await

```
                    P1                P2
{x = 0} co x := x+1 || x := x−1 oc {?}
```

Each statement actually consists of *3* operations, e.g., P_1 is

```
read x; inc; write x;
```

Atomic x-operations:

- P1 reads value of x (R1)
- P1 writes a value into x (W1)
- P2 reads value of x (R2)
- P2 writes a value into x (W2)

Example

Await

```
                    P1                P2
{x = 0} co x := x+1 || x := x−1 oc {?}
```

Each statement actually consists of *3* operations, e.g., P_1 is

```
read x; inc; write x;
```

Atomic x-operations:

- P1 reads value of x (R1)
- P1 writes a value into x (W1)
- P2 reads value of x (R2)
- P2 writes a value into x (W2)

What is the final state of our program?

Interleaving & Possible Execution Sequences (I)

The four operations cannot be executed in any order, the *program order* gives two constraints

Interleaving & Possible Execution Sequences (I)

The four operations cannot be executed in any order, the *program order* gives two constraints

- R1 must happen before W1
- R2 must happen before W2

Interleaving & Possible Execution Sequences (I)

The four operations cannot be executed in any order, the *program order* gives two constraints

- R1 must happen before W1
- R2 must happen before W2

Definition (Program Order)

- Two statements S_1, S_2 are program-ordered if they are in the same thread of the program, and S_1 occurs before S_2.
- Two operations O_1, O_2 from the same statement are program-ordered if O_1 occurs before O_2 in the translation of the statement.

In the example, `inc` and `dec` ("-1") are process-local, so we can ignore them

Interleaving & Possible Execution Sequences (II)

Definition (Interleaving)

An interleaving of two sequences A, B is a sequence C, such that

- exactly all elements of A and B are elements of C, and
- the order of elements in A (resp. B) is respected in C

An interleaving may have additional constraints (for us, e.g, program order).

Interleaving & Possible Execution Sequences (II)

Definition (Interleaving)

An interleaving of two sequences A, B is a sequence C, such that

- exactly all elements of A and B are elements of C, and
- the order of elements in A (resp. B) is respected in C

An interleaving may have additional constraints (for us, e.g, program order).

Interleavings for our example:

R1	R1	R1	R2	R2	R2
W1	R2	R2	R1	R1	W2
R2	W1	W2	W1	W2	R1
W2	W2	W1	W2	W1	W1
0	-1	1	-1	1	0

Non-determinism

- Final values for x: $\{0, 1, -1\}$
- As (post)-condition: $-1 \leq x \leq 1$
- Which one is chosen during an execution?

Await

```
{x = 0} co x := x+1 || x := x−1 oc {−1 <= x <= 1}
```

Non-determinism

- Final values for x: $\{0, 1, -1\}$
- As (post)-condition: $-1 \leq x \leq 1$
- Which one is chosen during an execution?

Await

```
{x = 0} co x := x+1 || x := x−1 oc {−1 <= x <= 1}
```

- **Non-determinism:** some choices for the program are decided during execution
- For us: the exact interleaving of instructions
- In practice, choices are not "random", but depend on factors *outside* the program code:
  - Timing of the threads
  - Scheduler of the operating system
  - ...

Execution-space Explosion

How many interleavings of statements are possible for one given input?

- Assume that we have 3 processes, each with the same number of atomic operations, and the same starting state
- Consider executions of $P_1 \parallel P_2 \parallel P_3$

Execution-space Explosion

How many interleavings of statements are possible for one given input?

- Assume that we have 3 processes, each with the same number of atomic operations, and the same starting state
- Consider executions of $P_1 \parallel P_2 \parallel P_3$

nr. of atomic op's	nr. of executions
2	90
3	1680
4	34 650
5	756 756

- Factorial growth!
- Different executions can lead to different final states.
- Even for simple systems: *impossible* to consider every possible execution in isolation

Factorial Explosion

The number of executions grows exponentially!

For n processes with m atomic statements each:

$$\text{number of executions} = \frac{(n * m)!}{m!^n}$$

- *n=m=5 gives 311680371562560, i.e.* $> 3 * 10^{14}$
- It would take ten million years to check, checking one execution each second!
- ... for each choice of input
- Testing hopeless as a validation technique!

How can we reduce the complexity?

The “at-most-once” Property

Fine-grained atomicity

Only the most basic operations (R/W) are atomic

- However, some non-atomic interactions appear to be atomic
- Note expressions only perform read-accesses (unlike statements)
- A *critical reference* in an expression e is a variable that is changed by another process
- An expression without critical references is evaluated as if atomic

The "at-most-once" Property

Fine-grained atomicity

Only the most basic operations (R/W) are atomic

- However, some non-atomic interactions appear to be atomic
- Note expressions only perform read-accesses (unlike statements)
- A *critical reference* in an expression e is a variable that is changed by another process
- An expression without critical references is evaluated as if atomic

Definition (At-most-once property)

$x := e$ satisfies the *amo*-property if either

1. e contains *no* critical reference, or
2. e contains *at most one* critical reference and x is not *referenced* by other processes

Assignments with the at-most-once property can be considered atomic!

At-most-once examples

`x`, `y` shared variables, `r`, `s` local variables

At-most-once examples

x, y shared variables, r, s local variables

Await

```
{x=y=0} co x := x+1 || y := x+1 oc {                    }
```

At-most-once examples

x, y shared variables, r, s local variables

Await

```
{x=y=0} co x := x+1 || y := x+1 oc {x = 1 & (y = 1 | y = 2)}
```

At-most-once examples

x, y shared variables, r, s local variables

Await

```
{x=y=0} co x := x+1 || y := x+1 oc {x = 1 & (y = 1 | y = 2)}
{x=y=0} co x := y+1 || y := x+1 oc {                        }
```

At-most-once examples

x, y shared variables, r, s local variables

Await

```
{x=y=0} co x := x+1 || y := x+1 oc {x = 1 & (y = 1 | y = 2)}
{x=y=0} co x := y+1 || y := x+1 oc {(x,y) ∈ {(1,1),(1,2),(2,1)}}
```

At-most-once examples

x, y shared variables, r, s local variables

Await

```
{x=y=0} co x := x+1 || y := x+1 oc {x = 1 & (y = 1 | y = 2)}
{x=y=0} co x := y+1 || y := x+1 oc {(x,y) ∈ {(1,1),(1,2),(2,1)}}
{x=y=0} co x := y+1 || x := y+3 || y := 1 oc {                }
```

At-most-once examples

x, y shared variables, r, s local variables

Await

```
{x=y=0} co x := x+1 || y := x+1 oc {x = 1 & (y = 1 | y = 2)}
{x=y=0} co x := y+1 || y := x+1 oc {(x,y) ∈ {(1,1),(1,2),(2,1)}}
{x=y=0} co x := y+1 || x := y+3 || y := 1 oc {y = 1 & 1<=x<=4}
```

At-most-once examples

x, y shared variables, r, s local variables

Await

```
{x=y=0} co x := x+1 || y := x+1 oc {x = 1 & (y = 1 | y = 2)}
{x=y=0} co x := y+1 || y := x+1 oc {(x,y) ∈ {(1,1),(1,2),(2,1)}}
{x=y=0} co x := y+1 || x := y+3 || y := 1 oc {y = 1 & 1<=x<=4}
{x=y=0} co r := y+1 || s := y−1 || y := 5 oc {?}
```

At-most-once examples

x, y shared variables, r, s local variables

Await

```
{x=y=0} co x := x+1 || y := x+1 oc {x = 1 & (y = 1 | y = 2)}
{x=y=0} co x := y+1 || y := x+1 oc {(x,y) ∈ {(1,1),(1,2),(2,1)}}
{x=y=0} co x := y+1 || x := y+3 || y := 1 oc {y = 1 & 1<=x<=4}
{x=y=0} co r := y+1 || s := y−1 || y := 5 oc {?}
```

Beware of unintuitive behavior:

Await

```
{ x = 0 } co r := x−x || x := 5 oc { r = 0? }
{ x = 0 } co x := x || ... oc { ? }
```

A Minimal Language for Concurrency

The Await Language

- Await is used to illustrate basic ideas about concurrency without boilerplate from mainstream languages
- Their implementation in mainstream languages is shown afterwards

The Await Language

- Await is used to illustrate basic ideas about concurrency without boilerplate from mainstream languages
- Their implementation in mainstream languages is shown afterwards

Features of Await

- Standard imperative constructs: sequence (;), assignment, branching, loops
- **co** .. || .. **oc** for parallel execution
- $<$ .. $>$ for atomic sections
- **await** for synchronization

Syntax: The Sequential Part

We use the following syntax for non-parallel control-flow

Declarations

```
int i = 3
int a[1:n]
int a[n] (= int a[0:n-1])
```

Assignments

```
x := e
a[i] := e
x++
sum +:= i
```

- **Sequential composition**
  statement; statement
- **Compound statement (block)**
  {statement}
- **Conditional**
  if (condition) statement
- **While-loop**
  while (condition) statement
- **For-loop**
  for [i=0 to n−1] statement

Parallel Statements

Await

co S_1 || S_2 || ... || S_n **oc**

- The statements S_i are executed *in parallel* with each other
- The parallel statement terminates when all S_i have terminated ("join" synchronization)

Parallel Statements

Await

co S_1 || S_2 || ... || S_n **oc**

- The statements S_i are executed *in parallel* with each other
- The parallel statement terminates when all S_i have terminated ("join" synchronization)

Await

{x = 0 & y = 0} **co** x := 1 || y := 1 **oc**; z := x+y { z = 2 }

Parallel Processes

For modularity, we also allow processes

Await

```
process foo {
    int sum := 0;
    for [i=1 to 10]
      sum +:= 1;
    x := sum;
  }
```

Parallel Processes

For modularity, we also allow processes

Await

```
process foo {
    int sum := 0;
    for [i=1 to 10]
      sum +:= 1;
    x := sum;
  }
```

- Processes are declared globally
- All declared processes are started in the beginning and evaluated in an arbitrary order

Example

Await

```
process bar1{
  for [i = 1 to n]
    write(i);
}
```

Starts one process

The numbers are printed in increasing order.

Example

Await

```
process bar1{
  for [i = 1 to n]
    write(i);
}
```

Await

```
process bar2a{ write(1); }
process bar2b{ write(2); }
```

Starts one process

The numbers are printed in increasing order.

Starts two processes

The numbers are printed in arbitrary order because the execution order of the processes is *non-deterministic*.

Example

Await

```
process bar1{
  for [i = 1 to n]
    write(i);
}
```

Await

```
process bar2a{ write(1); }
process bar2b{ write(2); }
```

Await

```
process barn [i=1 to n]{
    write(i);
}
```

Starts one process

The numbers are printed in increasing order.

Starts two processes

The numbers are printed in arbitrary order because the execution order of the processes is *non-deterministic*.

Starts n processes

The numbers are printed in arbitrary order.

Threads in Java

Java

```
class Printer implements Runnable {
  private String text;
  public Printer(String text) { this.text = text; }
  public void run() { System.out.println(text); }
  public static void main(String args[]) {
    Thread t1 = new Thread(new Printer("Hello"));
    Thread t2 = new Thread(new Printer("Concurrency"));
    t1.start(); t2.start();
  }}
```

- The `Thread` class encapsulates a system thread
- The `Runnable` interface is used to define thread behavior

Synchronization in Java

- Java does not have atomic blocks in the same form
- Instead: synchronized methods and blocks

Java

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {c++;}
    public synchronized void decrement() {c--;}
    public synchronized int value() {return c;}
}
```

- Synchronized methods are atomic *per object*
- Only one thread can execute such a method at any time

Java and Await

Await

```
co s1 || s2 oc; s3
```

Java

```
Thread t1 = new Thread(() -> {s1});
Thread t2 = new Thread(() -> {s2});
Thread t3 = new Thread(() -> {s3});
t1.start(); t2.start();
t1.join(); t2.join();
t3.start();
```

Java and Await

Await

```
co <s1> || <s2> oc
```

Java

```
public class C() {
public static synchronized void m1(){s1}
public static synchronized void m2(){s2}
...

new Thread(() -> {C.m1();}).start();
new Thread(() -> {C.m2();}).start();
}
```

Semantic Concepts ("Interleaving Semantics")

- A *state* in a parallel program consists of the values of the variables at a given moment in the execution.
- Each process executes independently of the others by *modifying* global variables using atomic operations.
- Do we really need to consider all interleavings to reason about possible states?
  - How to exclude some interleavings?
  - How to make reasoning modular and compositional?

Semantic Concepts ("Interleaving Semantics")

- A *state* in a parallel program consists of the values of the variables at a given moment in the execution.
- Each process executes independently of the others by *modifying* global variables using atomic operations.
- Do we really need to consider all interleavings to reason about possible states?
  - How to exclude some interleavings?
  - How to make reasoning modular and compositional?

Next, a first helping concept: interference

Read- and Write-variables

- $\mathcal{V}$: statement $\cup$ expression $\to \mathcal{P}$(variable): set of global variables in a statement or expression
- $\mathcal{W}$: statement $\cup$ expression $\to \mathcal{P}$(variable): set of global *write*–variables

Read- and Write-variables

- $\mathcal{V}$: statement $\cup$ expression $\to \mathcal{P}$(variable): set of global variables in a statement or expression
- $\mathcal{W}$: statement $\cup$ expression $\to \mathcal{P}$(variable): set of global *write*–variables

Read-variables

$$\mathcal{V}(\mathsf{x} := \mathsf{e}) \quad = \quad \mathcal{V}(\mathsf{e}) \cup \{\mathsf{x}\}$$

Read- and Write-variables

- $\mathcal{V}$: statement $\cup$ expression $\to \mathcal{P}$(variable): set of global variables in a statement or expression
- $\mathcal{W}$: statement $\cup$ expression $\to \mathcal{P}$(variable): set of global *write*–variables

Read-variables

$$\begin{aligned} \mathcal{V}(x := e) &= \mathcal{V}(e) \cup \{x\} \\ \mathcal{V}(S_1; S_2) &= \mathcal{V}(S_1) \cup \mathcal{V}(S_2) \end{aligned}$$

Read- and Write-variables

- $\mathcal{V}$: statement $\cup$ expression $\to \mathcal{P}$(variable): set of global variables in a statement or expression
- $\mathcal{W}$: statement $\cup$ expression $\to \mathcal{P}$(variable): set of global *write*–variables

Read-variables

$$
\begin{aligned}
\mathcal{V}(\mathsf{x} := \mathsf{e}) &= \mathcal{V}(\mathsf{e}) \cup \{\mathsf{x}\} \\
\mathcal{V}(\mathsf{S}_1; \mathsf{S}_2) &= \mathcal{V}(\mathsf{S}_1) \cup \mathcal{V}(\mathsf{S}_2) \\
\mathcal{V}(\text{if (b) then S}) &= \mathcal{V}(b) \cup \mathcal{V}(\mathsf{S})
\end{aligned}
$$

Read- and Write-variables

- $\mathcal{V}$: statement $\cup$ expression $\rightarrow \mathcal{P}$(variable): set of global variables in a statement or expression
- $\mathcal{W}$: statement $\cup$ expression $\rightarrow \mathcal{P}$(variable): set of global *write*–variables

Read-variables

$$
\begin{aligned}
\mathcal{V}(\mathsf{x} := \mathsf{e}) &= \mathcal{V}(\mathsf{e}) \cup \{\mathsf{x}\} \\
\mathcal{V}(\mathsf{S}_1; \mathsf{S}_2) &= \mathcal{V}(\mathsf{S}_1) \cup \mathcal{V}(\mathsf{S}_2) \\
\mathcal{V}(\text{if (b) then S}) &= \mathcal{V}(b) \cup \mathcal{V}(\mathsf{S}) \\
\mathcal{V}(\text{while(b)S}) &= \mathcal{V}(\mathsf{b}) \cup \mathcal{V}(\mathsf{S})
\end{aligned}
$$

Read- and Write-variables

- $\mathcal{V}$: statement $\cup$ expression $\to \mathcal{P}$(variable): set of global variables in a statement or expression
- $\mathcal{W}$: statement $\cup$ expression $\to \mathcal{P}$(variable): set of global *write*–variables

Read-variables

$$
\begin{aligned}
\mathcal{V}(\mathsf{x} := \mathsf{e}) &= \mathcal{V}(\mathsf{e}) \cup \{\mathsf{x}\} \\
\mathcal{V}(\mathsf{S}_1; \mathsf{S}_2) &= \mathcal{V}(\mathsf{S}_1) \cup \mathcal{V}(\mathsf{S}_2) \\
\mathcal{V}(\text{if (b) then S}) &= \mathcal{V}(b) \cup \mathcal{V}(\mathsf{S}) \\
\mathcal{V}(\text{while(b)S}) &= \mathcal{V}(\mathsf{b}) \cup \mathcal{V}(\mathsf{S})
\end{aligned}
$$

Remaining cases analogous

Read- and Write-variables

- $\mathcal{V}$: statement $\cup$ expression $\to \mathcal{P}$(variable): set of global variables in a statement or expression
- $\mathcal{W}$: statement $\cup$ expression $\to \mathcal{P}$(variable): set of global *write*–variables

Read-variables

$$
\begin{aligned}
\mathcal{V}(\mathsf{x} := \mathsf{e}) &= \mathcal{V}(\mathsf{e}) \cup \{\mathsf{x}\} \\
\mathcal{V}(\mathsf{S}_1; \mathsf{S}_2) &= \mathcal{V}(\mathsf{S}_1) \cup \mathcal{V}(\mathsf{S}_2) \\
\mathcal{V}(\text{if (b) then S}) &= \mathcal{V}(b) \cup \mathcal{V}(\mathsf{S}) \\
\mathcal{V}(\text{while(b)S}) &= \mathcal{V}(\mathsf{b}) \cup \mathcal{V}(\mathsf{S})
\end{aligned}
$$

Remaining cases analogous

$\mathcal{W}$ analogously, except the only difference for read-only expressions.

$$\mathcal{W}(\textit{expression}) = \emptyset$$

Read- and Write-variables

- $\mathcal{V}$: statement $\cup$ expression $\to \mathcal{P}$(variable): set of global variables in a statement or expression
- $\mathcal{W}$: statement $\cup$ expression $\to \mathcal{P}$(variable): set of global *write*–variables

Read-variables

$$
\begin{aligned}
\mathcal{V}(\mathsf{x} := \mathsf{e}) &= \mathcal{V}(\mathsf{e}) \cup \{\mathsf{x}\} \\
\mathcal{V}(\mathsf{S}_1; \mathsf{S}_2) &= \mathcal{V}(\mathsf{S}_1) \cup \mathcal{V}(\mathsf{S}_2) \\
\mathcal{V}(\text{if (b) then S}) &= \mathcal{V}(b) \cup \mathcal{V}(\mathsf{S}) \\
\mathcal{V}(\text{while(b)S}) &= \mathcal{V}(\mathsf{b}) \cup \mathcal{V}(\mathsf{S})
\end{aligned}
$$

Remaining cases analogous

$\mathcal{W}$ analogously, except the only difference for read-only expressions.

$$\mathcal{W}(\textit{expression}) = \emptyset$$

Example
$\mathcal{W}(\mathsf{x} := \mathsf{e}) = \{\mathsf{x}\}$

Disjoint Processes

Interference freedom

Processes without common global variables are *interference-free*

$$\mathcal{V}(S_1) \cap \mathcal{V}(S_2) = \emptyset$$

- Statements obviously cannot perform any action that influence each other
- As all interleavings are the same, one can just run $S_1; S_2$ for analysis
- Sequence $S_1; S_2$ is of course less performant

Disjoint Processes

Interference freedom

Processes without common global variables are *interference-free*

$$\mathcal{V}(S_1) \cap \mathcal{V}(S_2) = \emptyset$$

- Statements obviously cannot perform any action that influence each other
- As all interleavings are the same, one can just run $S_1; S_2$ for analysis
- Sequence $S_1; S_2$ is of course less performant

- If variables accessed by both processes are *read-only* variables, the same holds

Disjoint Processes

Interference freedom

Processes without common global variables are *interference-free*

$$\mathcal{V}(S_1) \cap \mathcal{V}(S_2) = \emptyset$$

- Statements obviously cannot perform any action that influence each other
- As all interleavings are the same, one can just run $S_1; S_2$ for analysis
- Sequence $S_1; S_2$ is of course less performant

- If variables accessed by both processes are *read-only* variables, the same holds
- Is the following *interference criterion* sufficient?

$$\mathcal{W}(S_1) \cap \mathcal{W}(S_2) = \emptyset$$

Disjoint Processes

Interference freedom

Processes without common global variables are *interference-free*

$$\mathcal{V}(S_1) \cap \mathcal{V}(S_2) = \emptyset$$

- Statements obviously cannot perform any action that influence each other
- As all interleavings are the same, one can just run $S_1; S_2$ for analysis
- Sequence $S_1; S_2$ is of course less performant

- If variables accessed by both processes are *read-only* variables, the same holds
- Is the following *interference criterion* sufficient?

$$\mathcal{W}(S_1) \cap \mathcal{W}(S_2) = \emptyset$$

- Write-variables are important for *race conditions*, *critical* references/amo-property, ...
- If only read-only variables are accessed, no races or critical references exist

Critical Sections and Invariants

Properties

Read-only variables are a very coarse way to think,
how to express more specific properties?

- A *property* is a predicate over a program, resp. its execution and reachable states
- A program has a property, if the property is true for all possible executions of the program

Properties

Read-only variables are a very coarse way to think,
how to express more specific properties?

- A *property* is a predicate over a program, resp. its execution and reachable states
- A program has a property, if the property is true for all possible executions of the program

Classification (I)

- Safety property: program will never reach an undesirable state
- Liveness property: program will eventually reach a desirable state

Properties

Read-only variables are a very coarse way to think,
how to express more specific properties?

- A *property* is a predicate over a program, resp. its execution and reachable states
- A program has a property, if the property is true for all possible executions of the program

Classification (I)

- Safety property: program will never reach an undesirable state
- Liveness property: program will eventually reach a desirable state

Classification (II)

- *Termination*: all executions are finite.
- *Partial correctness*: *If* the program terminates, it is in a desired final state (safety property).
- *Total correctness*: The program terminates and is partially correct.

How to Check Properties of Programs?

- *Testing* or *debugging* increases confidence in a program,
  but gives no guarantee of correctness.
- *Operational reasoning* considers *all* executions of a program explicitly
- *Formal analyses* reason about the properties of a program
  without considering the executions one by one.

Dijkstra's dictum:

A test can only show errors, but never prove that a program is correct!

Properties: Invariants (I)

Definition (Invariant)

An *invariant* is a property of program states, that holds for all reachable states of a program.

Properties: Invariants (I)

Definition (Invariant)

An *invariant* is a property of program states, that holds for all reachable states of a program.

- *Invariant* (adj): constant, unchanging
- Prototypical safety property
- Appropriate for non-terminating systems (does not require a final state)
- *All* reachable states often too strong

Properties: Invariants (I)

Definition (Invariant)

An *invariant* is a property of program states, that holds for all reachable states of a program.

- *Invariant* (adj): constant, unchanging
- Prototypical safety property
- Appropriate for non-terminating systems (does not require a final state)
- *All* reachable states often too strong

Kinds of Invariants

- Strong invariant: Holds for all reachable states
- Weak invariant: Holds for all states where an atomic block starts or ends
- Loop invariant: Holds at the start and end of a loop body
- Global invariant: Reasons about state of many processes
- Local invariant: Reasons about state of one process

Properties: Invariants (II)

- How to show that a program has a weak invariant?
- Without exploring all executions?

Properties: Invariants (II)

- How to show that a program has a weak invariant?
- Without exploring all executions?

Induction for Invariants

One can show that a program has a weak invariant by

1. Showing that the invariant property holds initially,
2. and that each atomic statement maintains the property

Critical Sections

To enforce atomicity, we have a special construct in the language : <S> performs S atomically

Critical Sections

To enforce atomicity, we have a special construct in the language : <S> performs S atomically

Use of Critical Sections

- When the processes interfere: *synchronization* to restrict the possible interleavings
- Synchronization gives coarser grained atomic operations ("atomic blocks")
- Combines operations into an *atomic block* where the process shall not be interrupted

Critical Sections

To enforce atomicity, we have a special construct in the language : <S> performs S atomically

Use of Critical Sections

- When the processes interfere: *synchronization* to restrict the possible interleavings
- Synchronization gives coarser grained atomic operations ("atomic blocks")
- Combines operations into an *atomic block* where the process shall not be interrupted

Characteristics of Atomic Operations

- Internal states are *not visible* to other processes.
- Variables *cannot* be changed underway by other processes.
- *S*: executed like a *transaction*

Critical Sections

To enforce atomicity, we have a special construct in the language : <S> performs S atomically

Use of Critical Sections

- When the processes interfere: *synchronization* to restrict the possible interleavings
- Synchronization gives coarser grained atomic operations ("atomic blocks")
- Combines operations into an *atomic block* where the process shall not be interrupted

Characteristics of Atomic Operations

- Internal states are *not visible* to other processes.
- Variables *cannot* be changed underway by other processes.
- *S*: executed like a *transaction*

Await

```
int x:=0; co <x:=x+1> || <x:=x−1> oc {x=0}
```

Conditional Critical Sections

Await statement

The <**await** (B) S > statement executes the statement S once the boolean condition B holds.

- Boolean condition B: *await condition*, evaluated atomically
- Body S: *critical section* executed atomically

Conditional Critical Sections

Await statement

The <**await** (B) S > statement executes the statement S once the boolean condition B holds.

- Boolean condition B: *await condition*, evaluated atomically
- Body S: *critical section* executed atomically

The following delays the decrement until $y > 0$ holds – or does not terminate if it never holds

Await

```
<await (y > 0) y := y−1> { y >= 0 }
```

Conditional Critical Sections

Await statement

The <**await** (B) S > statement executes the statement S once the boolean condition B holds.

- Boolean condition B: *await condition*, evaluated atomically
- Body S: *critical section* executed atomically

The following delays the decrement until $y > 0$ holds – or does not terminate if it never holds

Await

```
<await (y > 0) y := y-1> { y >= 0 }
```

- Important that B has no side-effects!

Typical Pattern for Critical Sections

- One wants to avoid using atomic blocks as much as possible
- Use them in certain places to enable correct, interleaved executions

Typical Pattern for Critical Sections

- One wants to avoid using atomic blocks as much as possible
- Use them in certain places to enable correct, interleaved executions

Await

```
int counter = 1;  // global variable

// start CS
< await (counter > 0) counter := counter-1; >
critical statements;
// end CS
counter := counter+1
```

Typical Pattern for Critical Sections

- One wants to avoid using atomic blocks as much as possible
- Use them in certain places to enable correct, interleaved executions

Await

```
int counter = 1;  // global variable

// start CS
< await (counter > 0) counter := counter-1; >
critical statements;
// end CS
counter := counter+1
```

- "Critical statements" *not* enclosed in atomic block
- Invariant: $0 \leq \mathit{counter} \leq 1$ (= counter acts as *binary lock*)
- Next lectures: patterns for correctness while minimizing atomic blocks

Example: Synchronization of Strongly Coupled Producer-Consumer System

Await

```
int buf, p := 0; c := 0;
```

Await

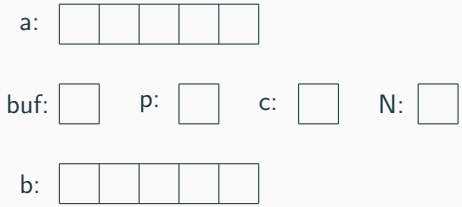
```
process Producer {
 int a[N];   ...
 while(p < N){
   <await (p = c) >;
   buf := a[p];
   p := p+1;
 }}
```

Await

```
process Consumer {
  int b[N];   ...
  while (c < N) {
    <await (p>c) >;
    b[c] := buf;
    c := c+1;
  }}
```

- `buf` as only shared variable, acting as a one element buffer
- Tasks of synchronization:
  - Coordinating the "speed" of the two processes
  - Avoiding to read data which is not yet produced

Example (Continued)



- A strong invariant holds in *all states* in *all* executions of the program.
- *Global invariant*: $\texttt{c} \leq \texttt{p} \leq \texttt{c+1}$
- *Local invariant (Producer)*: $\texttt{0} \leq \texttt{p} \leq \texttt{N}$

Wrap-Up

Summary

- Shared memory
- Synchronization
- Atomic operations,
- Interleavings
- `await`-language and critical sections
- A taste of concurrency in Java