

IN5170 Models of Concurrency

Lecture 1: Introduction

Andrea Pferscher

August 20th, 2025

University of Oslo

Motivation

Concurrency is everywhere and challenging

- Multiple computations at the same time
- Distributed systems, cloud computing, 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
 - ...
- What is the essence of these mechanisms and how should we use them?

Learning outcomes

- Knowledge about different mechanisms of parallelism, including shared variables and communication-based techniques
- Good insight into typical problems with parallel systems like deadlock, fairness, etc.
- Specification, design and analysis of a parallel system so that it meets the desired properties
- Consideration of the characteristics of a parallel system

What this course is not about ...

- Exploiting data parallelism
- Hardware-level concurrency

General Info

Structure

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

Literature

- First part: **G. R. Andrews. Foundations of Multi-threaded, Parallel, and Distributed Programming.** Addison Wesley, 2000 (Chapters 1 to 5, 7 and 8).
- Second and third part: Slides from the lectures, supplementary material

General Info

Exercises, obligs, exam

- (Almost) weekly exercise sessions (check schedule online)
- Two obligatory assignments (obligs):
 - Oblig 1: handout on 29.08.2025, submission deadline 15.09.2025
 - Oblig 2: handout on 19.09.2025, submission deadline 20.10.2025
- **Exam 11.12.2025**, check website for details

Rooms

- **Lectures:** 10:15 on Wednesdays in Store auditorium, Kristen Nygaards hus
(except 1st lecture which is in Seminarrom C, Ole-Johan Dahls hus)
- **Exercises:** 12:15 on Fridays in Prolog
- First exercise on Friday, 29.08.2025

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, additional slides about Java (see course page)

Shared Memory Concurrency

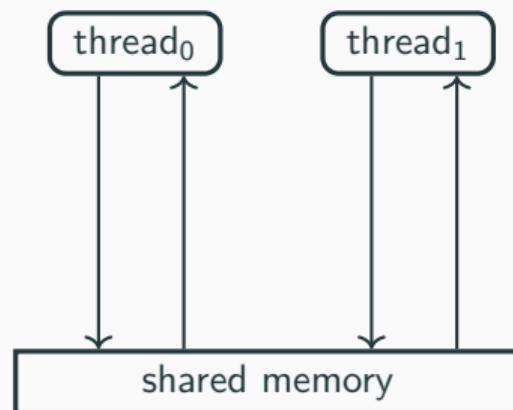
Parallel Processes

- **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

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

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)?
- 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 $\{\dots\}$ 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 Java Modeling language (JML) 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
```

- The execution of a parallel composition happens via the *concurrent* execution of the component processes S_1, \dots, 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

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
- *Conditional synchronization* enforces that processes must *wait* for a specific condition to be satisfied before execution can continue.

Atomic Expressions Java Virtual Machine (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
- 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.)
- Assignments $x := e$ are *not* atomic

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
- 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** {?}}

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

read x ; inc; write x ;

Atomic x -operations:

- P_1 reads value of x (R1)
- P_1 writes a value into x (W1)
- P_2 reads value of x (R2)
- P_2 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

- 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).

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
x	0	-1	1	-1	1
					0

Non-deterministic Behavior

- 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 $P_1 \parallel P_2 \parallel P_3$

nr. of atomic operations	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:

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

- $n=m=5$ gives 311680371562560 , i.e. $> 3 \cdot 10^{14}$
- 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

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, and 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}  
{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

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[n]
```

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 S1 || S2 || ... || Sn 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;
}
```

- 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.

Await

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

Starts two processes

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

Await

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

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 a synchronized method at any time and no other thread can execute any other synchronized method on the same object

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?

Next, a first helping concept: interference

Read- and Write-variables

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

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

$$\mathcal{V}(S_1; S_2) = \mathcal{V}(S_1) \cup \mathcal{V}(S_2)$$

$$\mathcal{V}(\text{if } (b) \text{ then } S) = \mathcal{V}(b) \cup \mathcal{V}(S)$$

$$\mathcal{V}(\text{while}(b)S) = \mathcal{V}(b) \cup \mathcal{V}(S)$$

Remaining cases analogous

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

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

Example

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

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
-
- 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

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.

- *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?

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 : $\langle S \rangle$ 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

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

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: $0 \leq \text{counter} \leq 1$ (= 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

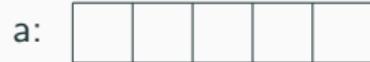
```
process Producer {  
    int a[N]; ...  
    while(p < N){  
        <await (p = c) >;  
        buf := a[p];  
        p := p+1;  
    }  
}
```

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)



buf: A single empty square box.
p: A single empty square box.
c: A single empty square box.
N: A single empty square box.



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

Wrap-Up

Summary

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

IN5170 Models of Concurrency

Self-study material: Basics about Concurrency in Java

Andrea Pferscher

August 27, 2025

University of Oslo

Threads Basics

Threads in Java (I/IV)

- Threads are units of execution that allow programs to perform multiple tasks simultaneously
- An application begins with a single thread, called the main thread. It is possible to create threads from this main thread.

Processes vs. threads (in Java)

- A process is an independent instance running on its own memory space.
 - A thread runs inside a process and shares its resources with other threads
-
- Here we focus on threads, multi-process applications in Java are possible but come with OS/JVM specific issues
 - Both are handled by OS scheduler
 - Both have costly context switches but threads are more light-weight
 - Only processes require full cache flushing as they change virtual memory
 - Thread-switch can retain caches, only changes processor state (registers etc.)

Threads in Java (II/IV)

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

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();  
    }}
```

Threads in Java (III/IV)

Start vs run

- `Thread.start()` starts a new *concurrent* thread
- `Runnable.run()` just executes the code *sequentially*
- `Thread.start()` calls `Runnable.run()` internally
- Calling `Runnable.run()` directly rarely makes sense

Threads in Java (IV/IV)

Common pattern for anonymous runnables to start a thread

Java

```
// option 1 with Runnable interface  
new Thread(new Runnable(){  
    public void run() { /* do things */ }}).start();  
// option 2 with lambdas  
new Thread( () -> { /* do things */ } ).start();
```

- Rather than declare a class that implements Runnable, it is possible to create an instance of that class, and pass that instance to the thread constructor.
- Lambdas offer a convenient short-cut to define an anonymous runnables more concisely and directly.

Shared State (I/III)

A shared state between threads is introduced through multiple means

- **Static state**
- Shared references to objects
- Resources, e.g., files

Java

```
class C { public static int i = 0; } ...
new Thread ( () -> { C.i++; }).start();
new Thread ( () -> { C.i++; }).start();
```

A variable declared as static means there is only one copy of it for the entire class becoming a shared variable.

Shared State (II/III)

A shared state between threads is introduced through multiple means

- Static state
- **Shared references to objects**
- Resources, e.g., files

Java

```
public class C { public int i = 0; } ...
final C c = new C();
new Thread ( () -> { c.i++ } ).start();
new Thread ( () -> { c.i++ } ).start();
```

What is final (immutable) is the reference to the object c, however its field i acts as a shared variable between the threads

Shared State (III/III)

Shared state between threads is introduced through multiple means

- Static state
- Shared references to objects
- **Resources, e.g., files**

Java

```
new Thread ( () -> { new File("/path/").delete(); } ).start();
new Thread ( () -> { new File("/path/").delete(); } ).start();
```

In this case, one could have two different links pointing to the same file,
becoming a shared resource, which means that the program will try to delete the file twice.

Methods of Java Threads (I/III)

- Java offers some additional operations that are abstracted away in `Await`
- Methods of `Thread` object:
 - `join` **waits for the thread to finish**
 - `sleep` suspends the thread for at least n milliseconds
 - `yield` gives the scheduler the signal to schedule someone else first

Java

```
public static void main(String args[]) {  
    Thread t1 = new Thread(new Printer("Hello"));  
    Thread t2 = new Thread(new Printer("Concurrency"));  
    t1.start(); t1.join(); // waits for t1 to finish  
    t2.start();  
}
```

Methods of Java Threads (II/III)

- Java offers some additional operations that are abstracted away in `Await`
- Methods of `Thread` object:
 - `join` waits for the thread to finish
 - `sleep` **suspends the thread for at least *n* milliseconds**
 - `yield` gives the scheduler the signal to schedule someone else first

Java

```
int i = 0; //shared
...
return m(){
    while(i == 0) Thread.sleep(10); //some constant chosen
    return 10/i;
}
```

Methods of Java Threads (III/III)

- Java offers some additional operations that are abstracted away in `Await`
- Methods of `Thread` object:
 - `join` waits for the thread to finish
 - `sleep` suspends the thread for at least n milliseconds
 - `yield` gives the scheduler the signal that it is possible to schedule someone else first

Await

```
int i = 0; //shared  
...  
return m(){  
    while(i == 0) Thread.yield();  
    return 10/i;  
}
```

Quiz: 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});  
??
```

Quiz: 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();
```

In the remainder of this presentation, we mostly show the code inside the threads and omit the boilerplate code for the Thread/Runnables

Atomic Blocks and synchronized

Synchronization

- Java does not have atomic blocks in the same form as described in the Await language
- Instead it is possible to use 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

Synchronization – Synchronized Methods

- A synchronized method locks the instance of the *object* that the method belongs to.
- All synchronized methods are synchronized with each other
- No two synchronized methods can be executed at the same time *in one instance*

Java

```
SynchronizedCounter c1 = new SynchronizedCounter();
SynchronizedCounter c2 = new SynchronizedCounter();
Runnable r1 = new Runnable(){
    public void run() { c1.increment(); } }
Runnable r2 = new Runnable(){
    public void run() { c1.increment(); } }
Runnable r3 = new Runnable(){
    public void run() { c1.decrement(); } }
Runnable r4 = new Runnable(){
    public void run() { c2.increment(); } }
```

Synchronization – Synchronized Methods (continuation)

The following will not interleave on c1:

Java

```
new Thread(r1).start();
new Thread(r2).start();
```

The following will also not interleave on c1:

Java

```
new Thread(r1).start();
new Thread(r3).start();
```

Synchronization – Synchronized Methods (continuation)

The following will interleave – the call is to two *different objects*

Java

```
new Thread(r1).start();
new Thread(r4).start();
```

Synchronization – Synchronized Static Methods

- Synchronized static methods are *per class*
- A synchronized static methods locks the Class object, meaning it affects all instances of the class rather than only one particular instance.

Java

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

Synchronization – Synchronized Static Methods (continuation)

Java

```
Runnable s1 = new Runnable(){
    public void run() { StaticSyncCounter.increment(); } }
Runnable s2 = new Runnable(){
    public void run() { StaticSyncCounter.decrement(); } }
```

The following code that starts two instances that will not interleave

Java

```
new Thread(s1).start();
new Thread(s2).start();
```

Synchronization – Synchronized Blocks

- Synchronized blocks allow to synchronize only *a part* of a code, rather than an entire method.
- **One need to give the object from which the lock should be obtained. Any object can be a lock.**
- Synchronized methods can have **this** as the lock

Java

```
public class C(){
    int l = 0;
    int r = 0;
    void method( Object lock , boolean left ){
        synchronized( lock ){
            if( left ) l++ else r++;
        }
    }
}
```

Synchronization – Synchronized Blocks (continuation)

- Synchronized blocks allows to synchronize only a *part* of a code, rather than an entire method.
- **One need to give the object from which the lock should be obtained. Any object can be a lock.**
- Synchronized methods can have **this** as the lock

The following will interleave

Java

```
C c = new C();
Object o1 = new Object();
Object o2 = new Object();
//thread 1 :
c.method(o1, true);
//thread 2 :
c.method(o2, false);
```

Synchronization – Synchronized Blocks (continuation)

- Synchronized blocks allows to synchronize only *a part* of a code, rather than an entire method.
- One need to give the object from which the lock should be obtained. Any object can be a lock.
- **Synchronized methods can have *this* as the lock**

Both code snippets are equivalent

Java

```
public class C(){ synchronized void method(){ ... } }
```

Java

```
public class C(){ void method(){ synchronized(this) { ... } }}
```

Atomic Expressions

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**

JVM

The JVM has no registers, but loads from variables/heap onto a stack. All computations target the top values on the stack.

Atomic Expressions

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

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
- Accessing variables modified by volatile is always atomic

Quiz: Java and Await

Await

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

Java

```
public class C() {  
??  
s1  
??  
s2  
??  
}
```

Quiz: 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();  
}
```

Weak Memory and volatile

Weak Memory

Java memory model

The JVM defines a *weak* memory model.

A weak memory model allows certain reordering in read and write operations.

- Mainly targeting performance, e.g., for cache optimization
- Can be done statically (by compiler) or dynamically (by processor)
- Which reorderings are allowed exactly, is defined by the memory model
- Strong memory model = no reorderings are allowed

Weak Memory

Independence

Reordering must take into account whether the operations are independent

- $x := 1; r := x;$ cannot be reordered
- $r := x; x := 1;$ cannot be reordered

because the result can change if the order of execution is changed

- Read-read reordering can reorder reads
- Write-read reordering can move a read before a write
- Read-write reorderings can move a write before a read
- Write-write reorderings change order of stores
- Some architectures also reorder other atomic operations

Reordering can happen if operations refer to different memory locations and therefore the order of execution does not affect the result of the execution.

Weak memory

Weak memory models can lead to very unintuitive results in concurrent settings

Assume P1 and P2 execute concurrently.

```
int x,y; //default 0
```

P1:

```
x := 1; //shared variable  
r1 := y; //register  
print r1;
```

P2:

```
y := 1; //shared variable  
r2 := x;  
print r2;
```

What are possible outputs? Is 0,0 possible?

- If the read of x in P2 and read of y in P1 is reordered, then 0,0 is possible, e.g., the compiler may decide to do such reordering in P1 and P2 since locally the result will not change
- This output cannot be explained by reasoning about interleavings
- If the language does not require variables to be initialized, we get *out-of-thin-air* values.

Weak Memory

Sequential consistency

Most weak memory models guarantee *sequential consistency*: the result of execution is the same as if all the operations were executed in some sequential order. *If there is no data race, then the observable behavior of the program is as if under a strong memory model.*

`volatile` can help to mitigate some of the challenges posed by weak memory models by providing specific guarantees about the visibility and ordering of variables declared as `volatile`.

Weak Memory

Java

```
public class C {  
    private volatile long l = 5;  
    long incRet() { return l++; } //called from two threads  
}
```

- All read and write accesses to `l` are atomic
- All write accesses to `l` are *immediately* visible to all threads
- In terms of memory model: no reads and writes to `l` are reordered before any write
- In terms of memory: `l` is read and written from global memory, not thread caches
- Does *not* introduce synchronization, but removes opportunities for optimization and makes access more expensive

Weak Memory

Weak memory is a complex topic, mostly relevant to low level architectures and compilers

- Further operations: read-own-write-early, read-others-write-early
- Leaks to programmer in concurrent settings
- Hard to debug, most languages have no clearly defined memory model
- Often hardware-dependent solutions

Rough guideline on when to use volatile

- If a field is not supposed to have data races, do not use volatile
- If a field will have data races, and you do not want to remove them, consider using volatile to avoid unintuitive behavior

Further Concepts

Java's Standard Library

- Java's standard library offers further data structures for common patterns or to encapsulate complex but efficient solutions
- Thread-safe collections (e.g., lists, maps, etc.) are less efficient, but internally race-free versions of collections
- Atomic classes encapsulate data with efficient, atomic access

Standard Library – Atomic Classes

- Atomic classes are available for data and references
- Operations that are atomic without synchronized blocks are more efficient (less blocking) but they can affect control flow and requires a deep understanding of how the operations work, and how they are used in the program e.g., the method `compareAndSet(a,b)` of `AtomicInteger` atomically updates the value to `b` if the current value matches `a`. The programmer will need to understand in the current control flow if the value will or will not be updated.
- `int` vs. `AtomicInteger`
 - `int` is a primitive data type. It is not thread-safe for operations that require atomicity. Multiple threads cannot safely perform operations that modify its value (e.g., incrementing the value) without additional synchronization.
 - `AtomicInteger` is thread-safe. Multiple threads can update a shared integer value without the risk of race conditions.

Standard Library – Atomic Classes (continuation)

Java

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

vs.

Java

```
public class SynchronizedCounter {
    private AtomicInteger c = new AtomicInteger(0);
    public void increment() {c.incrementAndGet();}
    public void decrement() {c.decrementAndGet();}
    public int value() {return c.get();} }
```

Standard Library – Atomic Classes (continuation)

AtomicReference does *not* make atomic the content of the called methods.

Java

```
AtomicReference<C> cache = new AtomicReference<C>();  
cache.get();      // atomic  
cache.get().m(); // the content of m is not executed atomically
```

Standard Library – Concurrent Collections

- Thread-safe collections provide atomic methods to access collections e.g., lists etc.

Examples of concurrent collections

Concurrent collections provide concurrent implementations that enable concurrent access

- Map vs. ConcurrentHashMap
 - Most implementations of Map are not thread-safe, e.g., HashMap is not thread-safe, meaning they can lead to race conditions and unpredictable behavior if accessed by multiple threads concurrently.
 - ConcurrentHashMap is thread-safe. All of its methods are designed to handle concurrent access from multiple threads without requiring explicit synchronization.
- ConcurrentLinkedQueue and variants for lists without random access
- CopyOnWriteArrayList makes an ArrayList concurrent by making a copy on every write, this is not efficient

Standard Library – Concurrent Collections (continuation)

Synchronized collections

Normal implementations, but add **synchronized** at the right places.

Java

```
<T> Collection<T> synchronizedCollection(Collection<T> c)
```

Java

```
ArrayList<Object> a = new ArrayList<>();
Collection<Object> b = Collections.synchronizedCollection(a);
// access through the object a is still unsafe
```

Non-thread-safe implementations of collections e.g., Map, tend to be faster in single-threaded environments or when explicit synchronization is not needed.

Thread Management

- In bigger applications, you may need to manage sets of threads
- We consider three concepts
 - Lifecycle of a thread object
 - Interrupts
 - Thread pools

Lifecycle

- A created thread object is **new**
- After calling `start` the thread is either
 - **Running**, i.e., executes right now
 - **Runnable**, i.e., waits to be scheduled (yields of other threads get you here)
 - **Waiting/Sleeping/Blocked**, i.e., waits for time to pass or some notification or lock
- Once the internal `run` method terminates, the object is **dead**
- Once a thread is dead it cannot be restarted

Interrupts

An interrupt is an *indication* to a thread that it should stop what it is doing and reconsider.

- Can be invoked using `t.interrupt();`
- This sets the `Thread.interrupted` flag, however the programmer must take care of reacting to this flag.
- `Thread.interrupted()` checks if the interrupt flag of the current thread has been set. This method also clears the interrupt flag when it is checked.
- Some methods will throw a `InterruptedException` if active (e.g., `Thread.sleep()` Throws `InterruptedException` if the thread is interrupted while it is sleeping)

Java

```
//long computation 1
if(Thread.interrupted){ /* handler */ }
//long computation 2
```

Thread Pools

- Creating and starting threads is costly
- Dead threads cannot be reused
- Solution: create a set of threads that do not terminate, but wait for new runnables to execute
- Automatic scaling

Thread Pools (continuation)

An ExecutorService manages a set of threads, and accepts Runnable instance submissions

Java

```
//has exactly 2 threads
ExecutorService service = Executors.newFixedThreadPool(2);
service.submit(() -> { /* do things */ });
service.submit(() -> { /* do things */ });
service.submit(() -> { /* do things */ });
//last runnable put in query, will be executed later
```

Thread Pools (continuation)

An ExecutorService manages a set of threads, and accepts Runnable instance submissions

Java

```
ExecutorService service = Executors.newCachedThreadPool(0,3);
//starts with 0 threads
service.submit(() -> { /* do things */ });
service.submit(() -> { /* do things */ });
service.submit(() -> { /* do things */ });
//up to 3 threads running
```

Thread Pools (continuation)

- To join on such a task, we get a *Future*
- We will investigate futures in more detail in Part 2 of the course

Java

```
//has exactly 2 threads
ExecutorService service = Executors.newFixedThreadPool(2);
Future<Int> f = service.submit(() -> { /* do */ return 1;});
...
Int = f.get(); //essentially a join
```

- Thread pools have further capabilities (shutdown)
- Details very java-specific, omitted here

Outlook

- We use the material in this slides as the basis for obligs and exercises
- Next lectures connect concepts with corresponding Java concept

IN5170 Models of Concurrency

Lecture 3: Locks and Barriers

Andrea Pferscher

September 3, 2025

University of Oslo

Repetition

Repetition

- Issues related to concurrency
- *await language* and a simple version of the *producer/consumer* example
- Basic concurrency in Java (see self-study material on the web page)

Today

- Entry and exit protocols to *critical sections*
 - Protect reading and writing to shared variables
- *Barriers*
 - Iterative algorithms:
Processes must *synchronize* between each iteration
 - Coordination using *flags*

Mutual Exclusion

Critical Section

Critical section

A *critical section* is part of the program that needs to be protected against interference by other processes

- Fundamental concept for concurrency - many solutions with different properties
- Execution under *mutual exclusion*
- Related to “atomicity”
- Using *locks* and low-level operations with software or hardware support

How can we implement (conditional) critical sections?

Access to Critical Section (CS)

- Basic scenario: Several processes compete for access to a shared resource
- Usage of the resource needs to be protected in a critical section
- Only one process can have access at a time (i.e., mutual exclusion)
 - Execution of bank transactions
 - Access to a printer or other resources
 - ...
- How do we control the access of processes to the CS?

Await

If we can implement critical sections, then we can also implement **await** statements!

- Must have exclusive control over state to atomically evaluate guard
- Must be able to block other processes if guard holds

General Patterns for Critical Sections (CSs)

- Inside the CS we have operations on shared variables.
- Access to the CS must then be protected to prevent interference.
- Coarse-grained pattern for n uniform processes repeatably executing some CS

Await

```
process p[i=1 to n] {
    while (true) {
        CSentry           # entry protocol to CS
        CS
        CSexit           # exit protocol from CS
        non-CS
    }
}
```

- *Assumption:* A process which enters the CS will eventually leave it.
⇒ *Programming advice:* be aware of exceptions inside CS!

Initial Solution

Await

```
int in = 1 # always 1 or 2
```

Await

```
process p1 {  
    while (true) {  
        while( in = 2 ) {skip};  
        CS  
        in := 2;  
    } }
```

Await

```
process p2 {  
    while (true) {  
        while( in = 1 ) {skip};  
        CS  
        in := 1;  
    } }
```

- *Entry protocol:* Busy waiting
- *Exit protocol:* Atomic assignment

Question:

- What will happen if we have more than two processes?

Desired Properties

1. **Mutual exclusion:** At any time, at most one process is inside CS.
2. **Absence of deadlock:** If all processes are trying to enter CS, at least one will succeed.
3. **Absence of unnecessary delay:** If some processes are trying to enter CS, while the other processes are in their non-critical sections, at least one will succeed.
4. **Eventual entry:** A process attempting to enter CS will eventually succeed.

Liveness and safety in critical sections

The first three are *safety* properties. The last is a *liveness* property.

Reminder: Invariants and Atomic Sections

Global invariants

A *global* invariant is a property over the shared variables that holds at every point during program execution (strong) or at every point outside an atomic section (weak)

- Safety property (something bad does not happen)
- Proof by induction: Holds initially and is preserved by every step

Atomic Sections

Statements grouped into a section that is always executed atomically.

- Conditional: <**await**(B) S>
- **await**(B) is known as condition synchronization, where B is evaluated atomically
- The whole block is executed atomically when B is true
- Unconditional: we write just < S >

Critical Sections using “Locks”

Await

```
bool lock := false;  
process [i=1 to n] {  
    while (true) {  
        < await (!lock) lock := true >;  
        CS;  
        lock := false;  
        non-CS  
    }  
}
```

Safety properties

- Mutex
- Absence of deadlock and absence of unnecessary waiting

Can we remove the angle brackets < ... >?

Critical Section – Test & Set (TAS)

Test & Set is a pattern for implementing a *conditional atomic action*:

Await

```
TAS(lock) {  
    < bool initial := lock;  
    lock := true >;  
    return initial;  
}
```

Effects of TAS(lock)

- *Side effect*: The variable lock will always have value true after TAS(lock),
- *Returned value*: true or false , depending on the original state of lock
- Exists as an atomic HW instruction on many machines.

Note: here it is assumed that the variable lock is passed as a reference

Critical section with TAS and spin-lock

Await

```
bool lock := false;

process p [i=1 to n] {
    while (true) {
        while (TAS(lock)) {skip};           # entry protocol
        CS
        lock := false;                      # exit protocol
    }
}
```

- Safety: Mutex, absence of deadlock and of unnecessary delay
- Strong fairness is needed to guarantee eventual entry for a process
- Problematic memory access pattern: lock as a hotspot

Reducing Writes

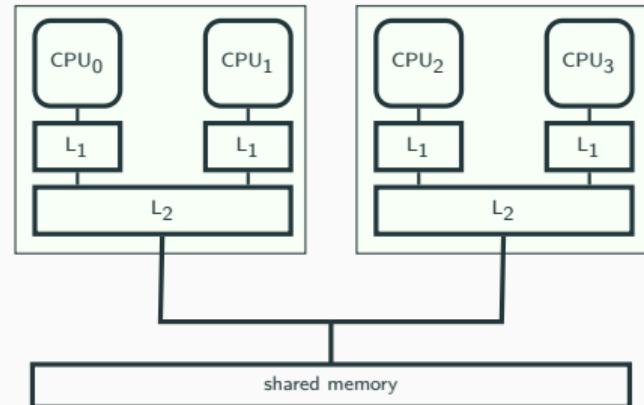
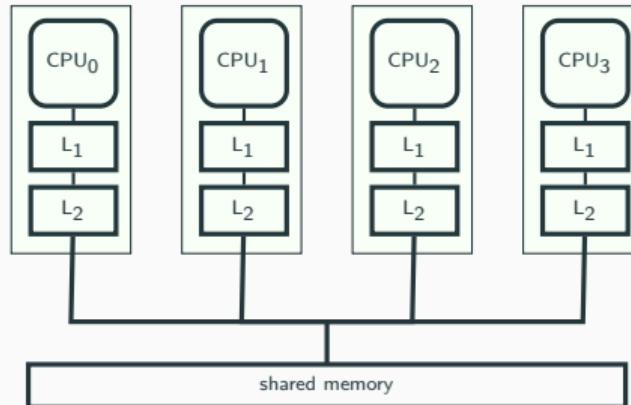
Test, Test and Set

Test, Test & Set (TTAS) reduces the number of writes by introducing more reads in the entry protocol.

Await

```
bool lock = false;  
process p[i = 1 to n] {  
    while (true) {  
        while (lock) {skip};      # two additional spin lock checks  
        while (TAS(lock)) { while (lock) {skip} };  
        CS;  
        lock := false;  
    }  
}
```

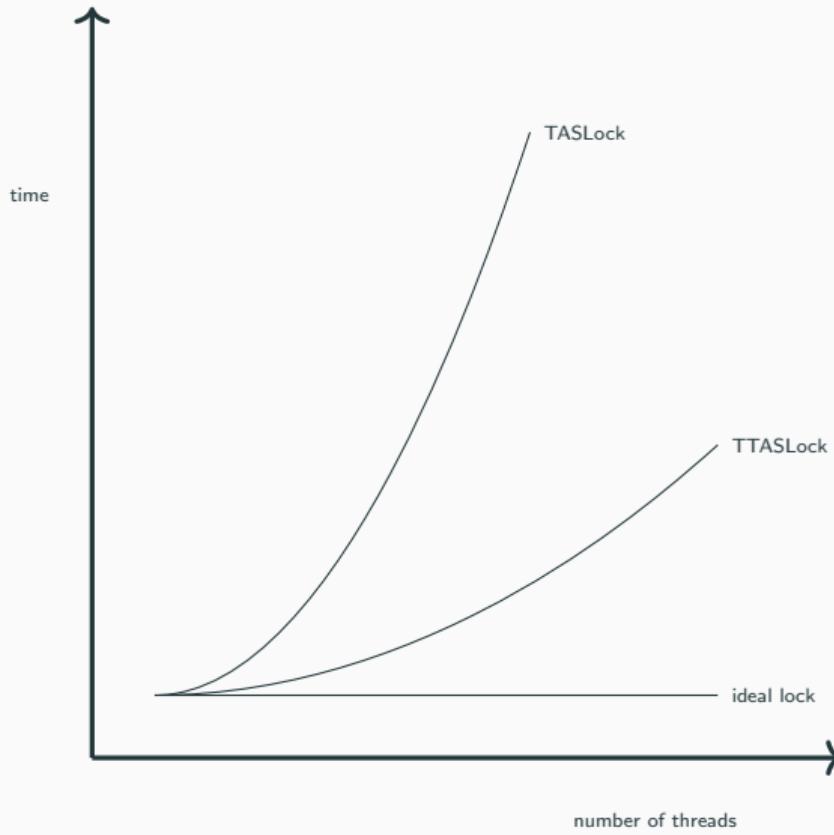
A Glance at HW for Shared Memory



Competition

- TAS accesses main memory and synchronizes the caches through writing
- Reading can just access cache

Multiprocessor Performance under Load (Competition)



Scheduling and Atomic Sections

Implementing Await-Statements

Let CSentry and CSexit implement entry- and exit-protocols to the critical section.

Unconditional atomic sections

The statement <S> can be implemented by

```
CSentry; S; CSexit;
```

Conditional atomic sections

Implementation of < **await** (B) S;> :

Await

```
CSentry;  
while (!B) { CSexit ; CSentry };  
S;  
CSexit;
```

Implementation can be optimized with some delay between the exit and entry in the while body. 14/47

Scheduling and Fairness - Processes share a Processor

- We want liveness properties as well, in particular *eventual entry*
- Eventual entry relies on scheduling and fairness

Enabledness

A statement is *enabled* in a state if the statement can in principle be executed next.

Await

```
bool x := true;  
co while (x){ skip }; || x := false co
```

Scheduling

A strategy that for all points in an execution decides which enabled statement to execute.

Fairness (informally)

Enabled statements should not “systematically be neglected” by the scheduling strategy.

Fairness Notions

Possible state changes

- Disabled → enabled
- Enabled → disabled

In our language, only conditional atomic segments can have state changes

Different forms of fairness for different forms of statements

1. For statements that are always enabled
2. For those that once they become enable, they *stay enabled*
3. For those whose enabledness shows “on-off” behavior

Unconditional Fairness

Definition (Unconditional fairness)

A scheduling strategy is *unconditionally fair* if each enabled unconditional atomic action, will eventually be chosen.

Await

```
bool x := true;  
co while (x){ skip }; || x := false co
```

- $x := \text{false}$ is unconditional
⇒ The action will eventually be chosen
- Guarantees termination (in this example)
- A round robin scheduling strategy execution is unconditionally fair

Weak Fairness

Definition (Weak fairness)

A scheduling strategy is *weakly fair* if

- it is unconditionally fair, and
- every conditional atomic action will eventually be chosen, assuming that the condition becomes true and *remains true* until the action is executed.

Await

```
bool x = true; int y = 0;  
co while (x) y := y + 1; || < await y ≥ 10; > x := false; oc
```

- When $y \geq 10$ becomes true, this condition remains true
- This ensures termination of the program
- Here: again round robin scheduling

Strong Fairness

Definition (Strongly fair scheduling strategy)

- Unconditionally fair, and
- Each conditional atomic action will eventually be chosen, if the condition is true infinitely often.

Await

```
bool x := true; y := false;  
co  
    while (x) {y:=true; y:=false}  
||  
    < await(y) x:=false >  
oc
```

- under strong fairness: y true ∞ -often \Rightarrow termination
- under weak fairness: non-termination possible

Fairness for Critical Sections using Locks

The CS solutions shown need strong fairness to guarantee liveness, i.e., access for a given process (i):

- Steady inflow of processes which want the lock
- value of lock **alternates**
(infinitely often) between true and false

Challenges

- How to design a scheduling strategy that is both practical and strongly fair?
- Next part: How to design critical sections where eventual access is guaranteed for weakly fair strategies?

Weakly fair solutions for critical sections

- *Tie-Breaker Algorithm*
- *Ticket Algorithm*
- Others described in the literature

Tie-Breaker Algorithm

Idea

- Requires no special machine instruction (like TAS)
- We will look at the solution for two processes
- Each process has a private lock
- Each process sets its lock in the entry protocol
- The private lock is read, but is not changed by the other process

Initial Solution

Await

```
int in = 1 # always 1 or 2
```

Await

```
process p1 {  
    while (true) {  
        while( in = 2 ) {skip};  
        CS  
        in := 2;  
    } }
```

Await

```
process p2 {  
    while (true) {  
        while( in = 1 ) {skip};  
        CS  
        in := 1;  
    } }
```

- *Entry protocol:* Busy waiting
- *Exit protocol:* Atomic assignment

Question:

- What will happen if we have more than two processes?

Initial Solutions: Characteristics and Restrictions

- Strict alternation
- Entry protocol: busy waiting
- Exit protocol: atomic assignment
- What about more than two processes?
- What about different execution times?

Tie-Breaker Algorithm: Attempt 1

Await

```
Boolean in1 = false, in2 = false;
```

Await

```
process p1 {
    while(true){
        while(in2) {skip};
        in1 := true;
        CS
        in1 := false;
    }
}
```

Await

```
process p2 {
    while(true){
        while(in1) {skip};
        in2 := true;
        CS
        in2 := false;
    }
}
```

Problem

Mutex not established, because both processes may be able to pass the entry protocol

What do we want as a global invariant?

Tie-Breaker Algorithm: Attempt 2 (reordering)

Await

```
Boolean in1 = false, in2 = false;
```

Await

```
process p1 {
    while(true){
        in1 := true;
        while(in2) {skip};
        CS
        in1 := false;
    }
}
```

Await

```
process p2 {
    while(true){
        in2 := true;
        while(in1) {skip};
        CS
        in2 := false;
    }
}
```

Problem

Can deadlock if both variables are written before read.

Tie-Breaker Algorithm: Attempt 3 (with await)

- Avoid deadlock through *tie-break* and decide for one process
- For fairness: do not always give priority to same specific process
- Add new variable: last to know which process last started the entry protocol

Await

```
Boolean in1 = false, in2 = false; Int last = 1;
```

Await

```
process p1 {
  while(true){
    in1 := true; last := 1;
    < await (!in2 or last = 2) >
    CS
    in1 := false;
  }}
```

Await

```
process p2 {
  while(true){
    in2 := true; last := 2;
    < await (!in1 or last = 1) >
    CS
    in2 := false;
  }}
```

Tie-Breaker Algorithm

Even if the variables `in1`, `in2` and `last` can change the value while a wait-condition evaluates to true, the wait condition will *remain true*.

`p1` sees that the wait-condition is true:

- `in2 = false`
 - `in2` can eventually become true,
but then `p2` must also set `last` to 2
 - Then the wait-condition to `p1` still holds
- `last = 2`
 - Then `last = 2` will hold until `p1` has executed

Thus we can replace the **await**-statement with a **while**-loop.

Tie-Breaker Algorithm: Attempt 4

Await

```
Boolean in1 = false, in2 = false; Int last = 1;
```

Await

```
process p1 {
  while(true){
    in1 := true;
    last := 1;
    while (in2 and last != 2)
      skip;
    CS
    in1 := false;
  }}
```

Await

```
process p2 {
  while(true){
    in2 := true;
    last := 2;
    while (in1 and last != 1)
      skip;
    CS
    in2 := false;
  }}
```

Ticket Algorithm

Multi-Tie-Breaker

- Generalizable to many processes (see Andrews; Sect. 3.3.1)
- But does not scale: If the Tie-Breaker algorithm is scaled up to n processes, we get a loop with a complex condition

The *ticket algorithm* provides a simpler solution for critical sections for n processes.

- Intuition: ticket queue at old-fashioned government agencies
- A customer/process which comes in takes a number which is higher than the number of all others who are waiting
- The customer is served when a ticket window is available and the customer has the lowest ticket number.

Ticket Algorithm: Sketch (n processes)

Await

```
int number := 1; next := 1; turn[1:n] := ([n] 0);
process [i = 1 to n] {
    while (true) {
        < turn[i] := number; number := number + 1 >;
        < await (turn[i] = next)>;
        CS
        <next := next + 1>;
    }
}
```

- **await**-statement: can be implemented as while-loop

Ticket Algorithm: Implementation

Await

```
int number := 1; next := 1; turn[1:n] := ([n] 0);
process [i = 1 to n] {
    while (true) {
        turn[i] := FA(number, 1);
        while (turn[i] != next) {skip};
        CS
        next := next + 1;
    }
}
```

- $\text{turn}[i]$ can be a local variable of $\text{process}[i]$
- Some machines have an *instruction* fetch-and-add (FA):
 $\text{FA}(\text{var}, \text{incr}) = <\text{int } \text{tmp} := \text{var}; \text{var} := \text{var} + \text{incr}; \text{return } \text{tmp};>$
- Without FA, we use an extra CS:

CSentry; $\text{turn}[i]:=\text{number}$; $\text{number}:= \text{number} + 1$; CSexit;

Ticket Algorithm: Global Invariant

Await

```
int number := 1; next := 1; turn[1:n] := ([n] 0);
process [i = 1 to n] {
    while (true) {
        turn[i] := FA(number, 1);
        while (turn [i] != next) {skip};
        CS
        next := next + 1;
    }
}
```

- What is a *global* invariant for the ticket algorithm?

$$0 < \text{next} \leq \text{number}$$

Locks and Barriers

Barrier Synchronization

- Computation of disjoint parts in parallel (e.g. array elements).
- Processes go into a loop where each iteration is dependent on the results of the previous.

Await

```
process Worker[ i=1 to n] {  
    while (true) {  
        # perform task i;  
        # barrier:  
    }  
}
```

All processes must reach the barrier before any can continue.

Shared counter

A number of processes can synchronize the end of their tasks using a *shared counter*:

Await

```
int count := 0;  
process Worker[i=1 to n] {  
    while (true) {  
        # perform task i  
        < count := count+1>; < await(count = n)>;  
    }  
}
```

- Can be implemented using the FA instruction.

Disadvantages

- count must be reset between each iteration and is updated using atomic operations.
- Inefficient: Many processes read and write count concurrently.

Coordination using Flags

- **Goal:** Avoid competition, i.e., too many read- and write-operations on one variable!
- Divides shared counter into *several* variables, with one global coordinator process

Await

```
Worker[i]:  
    # task i;  
    arrive[i] := 1;  
    < await (continue[i] = 1);>  
  
Coordinator:  
    for [i=1 to n] < await (arrive[i]=1);>  
    for [i=1 to n] continue[i] := 1;
```

Flag synchronization principles:

1. The process waiting for a flag is the one to reset that flag
2. A flag will not be set before it is reset

Synchronization using Flags

Both arrays `continue` and `arrived` are initialized to 0.

Await

```
process Worker [ i = 1 to n] {
    while (true) {
        # code to implement task i
        arrive[i] := 1;
        <await (continue[i] := 1)>;
        continue[i] := 0;
    }
}
```

Await

```
process Coordinator {
    while (true) {
        for [ i = 1 to n] {
            <await (arrive[i] = 1)>;
            arrive[i] := 0;
        };
        for [ i = 1 to n] {
            continue[i] := 1;
        }
    }
}
```

Summary: Implementation of Critical Sections

Await

```
bool lock = false;  
<await (!lock) lock := true>; # entry protocol  
# CS  
<lock := false> # exit protocol
```

- Spin lock implementation of entry: while (TAS(lock)) skip
- Exit without critical region.

Drawbacks:

- Busy waiting protocols are often complicated
- Inefficient if there are fewer processors than processes:
wastes time executing an empty loop
- No clear distinction between variables used for synchronization and computation

Desirable to have special tools for synchronization protocols: semaphores (next lecture)

Locks and Barriers in Java

Locks in Java: Introduction

- How to ensure mutual exclusion in Java?
- The `java.util.concurrent.locks` package contains interfaces and classes for locking and waiting for conditions (distinct from built-in synchronization/monitors)
- Manual lock management: flexible, but must be used cautiously

Lock interface

- Supports different semantics of locking
- Main implementation: *ReentrantLock*

ReadWriteLock Interface

- Locks that may be shared among readers but are exclusive to writers
- Implementation: *ReentrantReadWriteLock*

Condition Interface

Condition (variables) associated with locks

Locks in Java: The Lock Interface (I/II)

Flexibility of locks \Rightarrow responsibility to use locks correctly

- Ensure that the lock is acquired before executing the code in the critical section
- Ensure that the lock is released in the end, even if something went wrong

Generic pattern for a method using a lock

Java

```
Lock mutex = new Lock(); // shared between processes
...
mutex.lock();
try {
    ...
} finally { mutex.unlock(); }
```

Locks in Java: The Lock Interface (II/II)

Includes methods:

- lock(): For acquiring the lock
- unlock(): For releasing the lock
- newCondition(): Returns a new Condition instance that is bound to this Lock instance

Example: A Counter

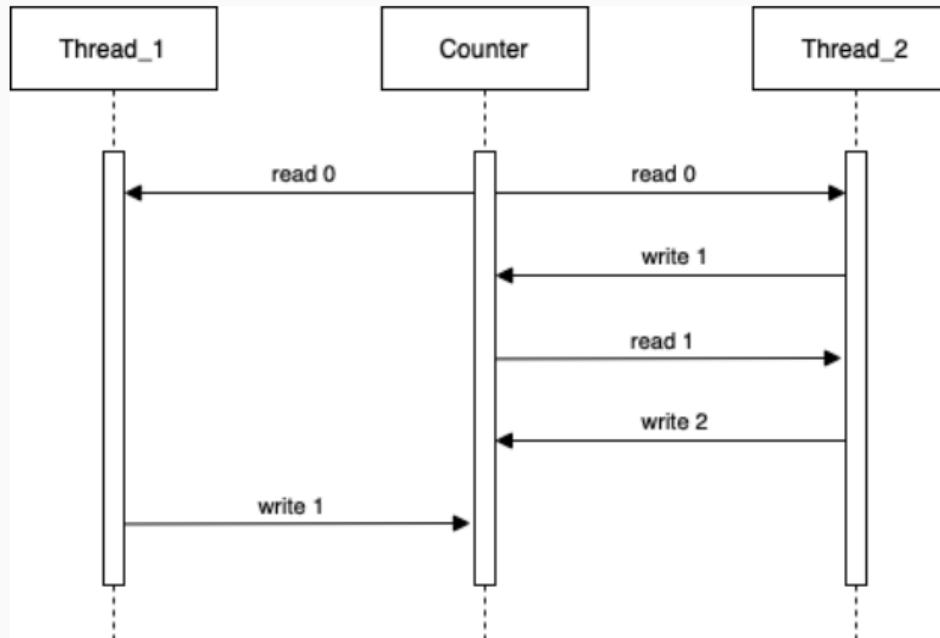
- Task: Add numbers from 0 to x using several threads
- Idea: Have a shared variable $value$ that the threads increase

Java

```
public class Counter {  
    private int value;  
    public Counter(int c) { value = c; }  
    public int getAndIncrement() {  
        int temp = value;  
        value = temp + 1;  
        return temp;  
    }  
}
```

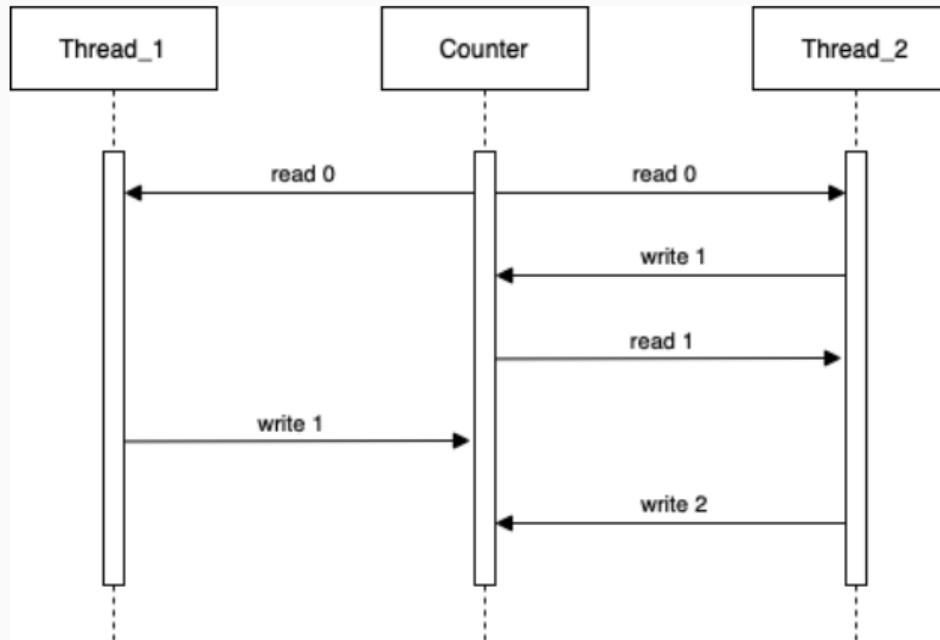
Example: A Counter (cont.)

Which values can *value* take if two threads call `getAndIncrement` three times in total?



Example: A Counter (cont.)

Which values can *value* take if two threads call `getAndIncrement` three times in total?



```
public class CounterLock {  
    private int value;  
    private Lock mutex = new ReentrantLock();  
    public Counter(int c) { value = c; }  
    public int getAndIncrement() {  
        mutex.lock();           // entry  
        int temp = 0;  
        try {  
            temp = value;      // critical section  
            value = temp + 1;  // critical section  
        } finally { mutex.unlock(); } //exit  
        return temp;  
    }  
}
```

Repetition: Barrier Synchronization

— *Await* —

```
process Worker[ i=1 to n] {
    while (true) {
        task i;
        wait until all n tasks are done      # barrier
    }
}
```

Barrier Synchronization

Barrier Synchronization in Java: The CyclicBarrier Class

- To make multiple threads to wait for each other until they all reach a certain point in their execution. Once all threads have reached the barrier, they are released simultaneously.
- It provides an optional Runnable barrier action that will be executed once all threads have reached the barrier, but before they are released.
- The barrier is called *cyclic* because it can be reused after the waiting threads are released.

Barrier Synchronization in Java: The CyclicBarrier Class

Example can be found at: [Example link](#)

which is based on the webpage:

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html>

Semaphores

Silvia Lizeth Tapia Tarifa

September 10, 2025

University of Oslo

Overview

Last lecture: Locks and Barriers

- Complex techniques
- No clear separation between variables for synchronization and variables for computation
- Busy waiting

This lecture: Semaphores

- Synchronization tool
- Used easily for mutual exclusion and condition synchronization
- A way to implement signaling and scheduling
- Implementable in many ways on hardware (CMPXCHG)
- Available in programming language libraries and OS

Outline

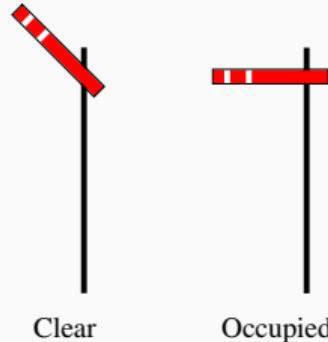
- Semaphores: Syntax and semantics
- Synchronization examples:
 - Mutual exclusion (critical sections)
 - Barriers (signaling events)
 - Producers and consumers (split binary semaphores)
 - Bounded buffer: resource counting
 - Dining philosophers: mutual exclusion – deadlock
 - Readers and writers:
 - condition synchronization
 - passing the baton

Semaphores

Semaphores

Origins of Term

- Introduced by Dijkstra in 1968
- Inspired by railroad traffic synchronization
- Railroad semaphore indicates whether the track ahead is clear or occupied by another train



Properties

- Semaphores in concurrent programs: work similarly
- Used to implement
 - *mutex* and
 - *condition synchronization*
- Included in most standard libraries for concurrent programming
- Also *system calls* in, e.g., Linux kernel, Windows etc.

Concept

Concept of a Semaphore

- *Semaphore*: special kind of shared program variable (with built-in sync. power)
- value of a semaphore: a *non-negative* integer
- can *only* be manipulated by two *atomic* operations:

The Semaphore Operations: *P* and *V*

- **P**: (Passeren) Wait for signal – want to *pass*
Wait until value is greater than zero, and *decrease* value by one
- **V**: (Vrijgeven) Signal an event – *release*
Increase the value by one

- Today, libraries and sys-calls prefer other names: up/down, wait/signal, acquire/release
- Different flavors of semaphores: binary vs. counting
- Most common: mutex as a synonym for binary semaphores

Syntax and Semantics

Declaration

- sem s; default initial value is zero
- sem s := 1;
- sem s[4] := ([4] 1);

Operations and Semantics

P-operation P(s)

$\langle \text{await } (s > 0) s := s - 1 \rangle$

V-operation V(s)

$\langle s := s + 1 \rangle$

Processes waiting on a semaphore are woken up by the op. system.

Remarks on Semaphores

Remark 1

Important: No *direct* access to the value of a semaphore.

For example, a test like if $(s = 1)$ then ... else is *forbidden*!

Kinds of semaphores

General semaphore: Possible values: *all non-negative integers*

Binary semaphore: Possible values: 0 and 1

Fairness

- As for await-statements.
- In most languages: *FIFO* (“waiting queue”): processes delayed while executing P-operations are *awoken* in the *order* they were delayed

Example: Mutual Exclusion (critical section)

Mutex implemented by a *binary semaphore*

Await

```
sem mutex := 1;
process CS[ i = 1 to n] {
    while (true) {
        P(mutex);
        # critical section
        V(mutex);
        # noncritical section
    }
}
```

- The semaphore is *initially 1*
- Always P before V → (used as) binary semaphore

Example: Barrier Synchronization

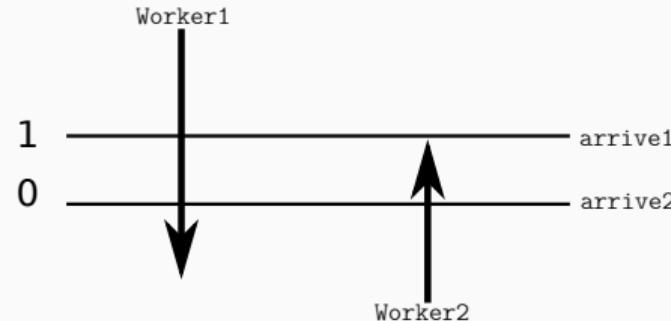
Semaphores may be used for *signaling events*

Await

```
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
    ...
    V(arrive1); # reach barrier
    P(arrive2); # wait for other
    ...
}
process Worker2 {
    ...
    V(arrive2); # reach barrier
    P(arrive1); # wait for other
    ...
}
```

Example: Barrier Synchronization

- *Signalling semaphores*: usually *initialized to 0* and
- *Signal* with a V and then *wait* with a P



Split Binary Semaphores

Split binary semaphore

A set of semaphores, whose *sum* ≤ 1

Mutex by split binary semaphores

- Initialization: *one* of the semaphores = 1, all others = 0
- Discipline: all processes call *P* on a semaphore, *before* calling *V* on (*another*) semaphore
⇒ Code between the *P* and the *V*
 - All semaphores = 0
 - Code executed *in mutex*

Example: Producer/Consumer with Split Binary Semaphores

Await

```
T buf; # one element buffer, some type T  
sem empty := 1;  
sem full := 0;
```

Await

```
process Producer {  
    while (true) {  
        P(empty);  
        buff := data;  
        V(full);  
    }  
}
```

Await

```
process Consumer {  
    while (true) {  
        P(full);  
        data_c := buff;  
        V(empty);  
    }  
}
```

- empty and full are both *binary semaphores*, together they form a split binary semaphore.
- Solution works with *several* producers/consumers

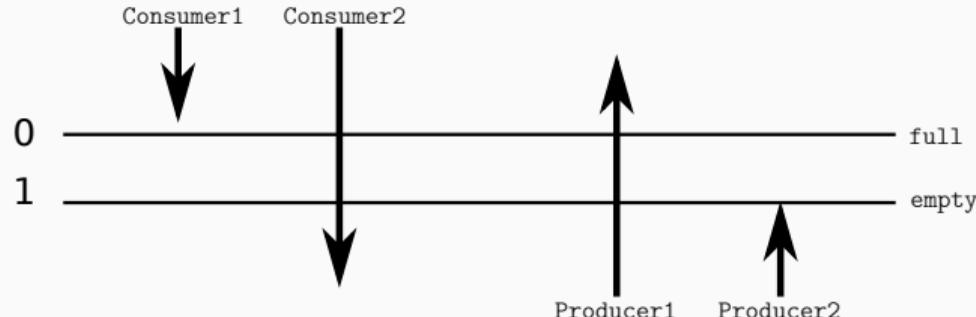
Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {  
    while (true) {  
        P(empty);  
        buff := data;  
        V(full);  
    }  
}
```

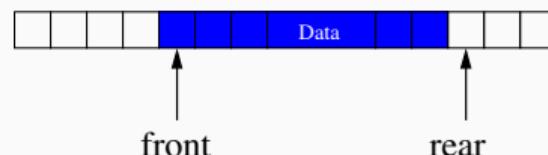
Await

```
process Consumer {  
    while (true) {  
        P(full);  
        data_c := buff;  
        V(empty);  
    }  
}
```



Producer/Consumer: Increasing Buffer Capacity

- Previously: tight coupling, the producer must wait for the consumer to empty the buffer before it can produce a new entry.
- Easy *generalization*: buffer of size n .
- Loose coupling/asynchronous communication \Rightarrow “buffering”
 - *Ring-buffer*, typically represented
 - by an array
 - + two integers **rear** and **front**.
 - Semaphores to *keep track* of the number of free/used slots \Rightarrow general semaphore



Producer/Consumer: Increased Buffer Capacity

Await

```
T buf[n]                      # array , elements of type T
int front := 0, rear := 0; # ``pointers''
sem empty := n;             # number of empty slots
sem full := 0;              # number of filled slots
```

Await

```
process Producer {
    while (true) {
        P(empty);
        buff[rear] := data;
        rear := (rear + 1);
        V(full);
    }
}
```

Await

```
process Consumer {
    while (true) {
        P(full);
        result := buff[front];
        front := (front + 1);
        V(empty);
    }
}
```

Producer/Consumer: Increased Buffer Capacity

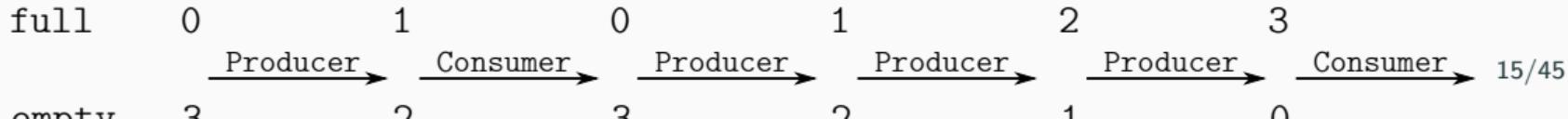
Await

```
process Producer {
    while (true) {
        P(empty);
        buff[rear] := data;
        rear := (rear + 1);
        V(full);
    }
}
```

Await

```
process Consumer {
    while (true) {
        P(full);
        result := buff[front];
        front := (front + 1);
        V(empty);
    }
}
```

- Important: there are no critical sections!
- How to enable several producers and consumers?



Increasing the Number of Processes

How to enable several producers and consumers?

New synchronization problems

- *Avoid* that two producers *deposit* to buf [rear] before rear is updated
- *Avoid* that two consumers *fetch* from buf [front] before front is updated.

Solution

Add 2 extra binary semaphores for protection:

- mutexDeposit to deny two producers to deposit to the buffer at the same time.
- mutexFetch to deny two consumers to fetch from the buffer at the same time.

Example: Producer/Consumer with Several Processes

Await

```
T buf[n]                      # array, elements of type T
int front := 0, rear := 0; # ``pointers''
sem empty := n;             # number of empty slots
sem full := 0;              # number of filled slots
sem mutexDeposit; mutexFetch := 1; # protect the data stuct.
```

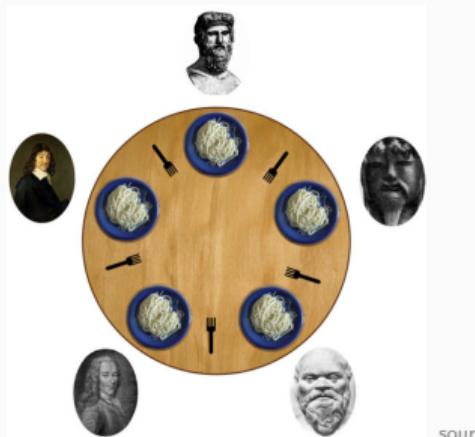
Await

```
process Producer {
    while (true) {
        P(empty);
        P(mutexDeposit);
        buff[rear] := data;
        rear := (rear + 1);
        V(mutexDeposit);
        V(full);
    }
}
```

Await

```
process Consumer {
    while (true) {
        P(full);
        P(mutexFetch);
        result := buff[front];
        front := (front + 1);
        V(mutexFetch);
        V(empty);
    }
}
```

Problem: Dining Philosophers



source:wikipedia.org

- Famous sync. problem (Dijkstra)
- Five philosophers around a circular table.
- One fork placed between each pair of philosophers
- Each philosopher alternates between thinking and eating
- A philosopher needs two forks to eat (and none for thinking)

Dining Philosophers: Sketch

Await

```
process Philosopher [i = 0 to 4] {
    while true {
        # think
        acquire forks;
        # eat
        release forks;
    }
}
```

Task:

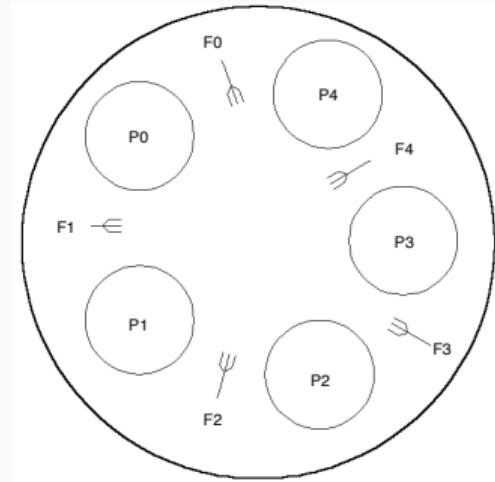
Program the actions `acquire forks` and `release forks`

Dining philosophers: 1st attempt

- Forks as *semaphores*
- Philosophers: pick up left fork first

Await

```
sem fork[5] := ([5] 1);
process Philosopher [ i = 0 to 4 ] {
    while true {
        # think
        P(fork[i]);
        P(fork[(i+1)%5]);
        # eat
        V(fork[i]);
        V(fork[(i+1)%5]);
    }
}
```



Dining philosophers: 2nd attempt

Breaking the symmetry

To avoid *deadlock*, let 1 philosopher (say 4) grab the *right* fork first

Await

```
process Philosopher [ i = 0 to 3] {
    while true {
        think;
        P(fork[i]);
        P(fork[(i+1)%5]);
        eat;
        V(fork[i]);
        V(fork[(i+1)%5]);
    }
}
```

Await

```
process Philosopher4 {
    while true {
        think;
        P(fork[4]); #!
        P(fork[0]); #!
        eat;
        V(fork[4]);
        V(fork[0]);
    }
}
```

Dining philosophers

- Important illustration of problems with concurrency:
 - Deadlocks,
 - Other aspects: liveness, fairness, etc.
- Resource access
- Connection to mutex/critical sections

Invariants and Condition Synchronization

Readers/Writers: Overview

- Classic synchronization problem
- *Reader* and *writer* processes, share access to a database/shared data structure
 - Readers only read from the database
 - Writers update (and read from) the database
- As soon as one writer is included, read and write accesses may cause interference
- Readers and writers have **asymmetric requirements:**
 - Every *writer* needs *mutually exclusive* access
 - When no writers have access, *many readers* may access the database

Readers/Writers: Approaches

- Dining philosophers: Pair of processes compete for access to “forks”
- Readers/writers: Different *classes* of processes compete for access to the database
 - Readers *compete* with writers
 - Writers *compete* both with readers and other writers
- **General synchronization problem:**
 - Readers: must wait until no writers are active in DB
 - Writers: must wait until no readers or writers are active in DB
- Here: two different approaches
 1. **Mutex:** easy to implement, but “*unfair*”
 2. **Condition synchronization:**
 - Using a *split binary semaphore*
 - Easy to adapt to different scheduling strategies

Readers/Writers with Mutex (1)

Await

```
sem rw := 1;
```

Await

```
process Reader [ i=1 to M] {
    while (true) {
        P(rw);
        # read
        V(rw);
    }
}
```

Await

```
process Writer [ i=1 to N] {
    while (true) {
        P(rw);
        # write
        V(rw);
    }
}
```

We want *more than one reader simultaneously*.

Readers/Writers with Mutex (2)

Await

```
int nr := 0;    # number of active readers
sem rw := 1      # lock for reader/writer mutex
```

Await

```
process Reader [ i=1 to M] {
    while (true) {
        < nr := nr + 1;
        if (nr=1) P(rw) >;
        # read
        < nr := nr - 1;
        if (nr=0) V(rw) >;
    }
}
```

Await

```
process Writer [ i=1 to N] {
    while (true) {
        P(rw);
        # write
        V(rw);
    }
}
```

How do semaphores work *inside* atomic sections?

Readers/Writers with Mutex (3)

Await

```
int      nr = 0; # number of      active readers
sem      rw = 1; # lock for reader/writer exclusion
sem mutexR = 1; # mutex for readers

process Reader [ i=1 to M] {
    while (true) {
        P(mutexR)
        nr := nr + 1;
        if (nr=1)  P(rw);
        V(mutexR)
        # read
        P(mutexR)
        nr := nr - 1;
        if (nr=0)  V(rw);
        V(mutexR)
    }
}
```

Readers/Writers with Condition Synchronization: Overview

Reader's preference

- With a constant stream of readers, the writer will never run
- Even under strong fairness
- Previous *mutex* solution solved *two* separate synchronization problems
 - ***rw*** : *Readers and writers* for access to the *database*
 - ***mutexR***: *Reader vs. reader* for access to the *counter*
- Now: a solution based on **condition synchronization**

Invariant

Reasonable invariant for the critical sections

1. When a *writer* accesses the DB, *no one else* can
2. When *no writers* access the DB, *one or more readers* may get access

Introducing state for the invariant

Introduce two counters:

- **nr**: number of active readers
- **nw**: number of active writers

Invariant

$RW: (nr = 0 \text{ or } nw = 0) \text{ and } nw \leq 1$
(same as:) $RW': nw=0 \text{ or } (nw = 1 \text{ and } nr = 0)$

Code for counting Readers and Writers

Await

```
< nr := nr + 1; >  
# read  
< nr := nr - 1; >
```

Await

```
< nw := nw + 1; >  
# write  
< nw := nw - 1; >
```

- Add synchronization code to maintain the invariant
- Decreasing counters is not dangerous
- Before increasing, we need to check some conditions for synchronization
 - before increasing nr: nw = 0
 - before increasing nw: nr = 0 and nw = 0

Condition Synchronization: Without Semaphores

Await

```
int nr := 0;    # number of active readers
int nw := 0;    # number of active writers
# Invariant RW: (nr = 0 or nw = 0) and nw <= 1
```

Await

```
process Reader [ i=1 to M]{
  while (true) {
    < await (nw=0)
      nr := nr+1>;
    # read
    < nr := nr - 1>
  }
}
```

Await

```
process Writer [ i=1 to N]{
  while (true) {
    < await (nr = 0 and nw = 0)
      nw := nw+1>;
    # write
    < nw := nw - 1>
  }
}
```

Condition Synchronization: Converting to Split Binary Semaphores

Convert awaits with different guards $B_1, B_2 \dots$ to Split Binary Semaphores

- *Entry:* semaphore e , initialized to 1
- For each guard B_i :
 1. associate one *counter* and
 2. one *delay-semaphore*

Both initialized to 0

- Semaphore *delays* the processes waiting for B_i
- Counters counts the number of processes *waiting* for B_i

For *readers/writers* problem we need 3 semaphores and 2 counters:

Await _____

```
sem e = 1;
sem r = 0; int dr = 0; # condition reader: nw == 0
sem w = 0; int dw = 0; # condition writer: nr == 0 and nw == 0
```

Condition Synchronization: Converting to Split Binary Semaphores (2)

- e , r and w form a *split binary semaphore*.
- All execution paths *start* with a *P-operation* and *end* with a *V-operation* → Mutex

Signaling

We need a signal mechanism *SIGNAL* to pick which semaphore to signal.

- SIGNAL: make sure the invariant holds
- B_i holds when a process enters *CS* because either:
 - the process checks itself,
 - or the process is only *signalled* if B_i holds
- **Another pitfall:**

Avoid *deadlock* by checking the counters before the delay semaphores are signalled.

 - r is not signalled ($V(r)$) *unless* there is a delayed reader
 - w is not signalled ($V(w)$) *unless* there is a delayed writer

Condition Synchronization: Reader

Await

```
int nr := 0, nw = 0;      # counter variables (as before)
sem e := 1;                # entry semaphore
int dr := 0; sem r := 0; # delay counter + sem for reader
int dw := 0; sem w := 0; # delay counter + sem for writer
# invariant RW: (nr = 0 or nw = 0 ) and nw <= 1
process Reader [i=1 to M]{ # entry condition: nw = 0
    while (true) {
        P(e);
        if (nw > 0) { dr := dr + 1;    # < await (nw=0)
                        V(e);          #     nr:=nr+1 >
                        P(r)};
        nr:=nr+1; SIGNAL;
        # read
        P(e); nr:=nr-1; SIGNAL;      # < nr:=nr-1 >
    }
}
```

With Condition Synchronization: Writer

Await

```
process Writer [i=1 to N]{ # entry condition: nw = 0 and nr = 0
    while (true) {
        P(e);                                # < await (nr=0 and nw=0)
        if (nr > 0 or nw > 0) {      #     nw:=nw+1 >
            dw := dw + 1;
            V(e);
            P(w) };
        nw:=nw+1; SIGNAL;
        # write
        P(e); nw:=nw-1; SIGNAL      # < nw:=nw-1>
    }
}
```

With Condition Synchronization: Signalling

Await

```
if (nw = 0 and dr > 0) {  
    dr := dr -1; V(r);           # awake reader  
}  
elseif (nr = 0 and nw = 0 and dw > 0) {  
    dw := dw -1; V(w);         # awake writer  
}  
else V(e);                      # release entry lock
```

- This passes the control (the “baton”) to an appropriate next process
- SIGNAL has no P operation, each path has exactly one V operation.
- Using the conditions to see who goes next.
- Called “passing the baton” technique (as in relay competition).
- Conditions for awakening must be disjoint

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
nw	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0
e	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0
dw	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
w	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0

Await

```
if (nw = 0 and dr > 0) {
    dr := dr -1; V(r);                      # awake reader
}
elseif (nr = 0 and nw = 0 and dw > 0) {
    dw := dw -1; V(w);                      # awake writer
}
else V(e);                                # release entry lock
```

Await

```
process Reader [i=1 to M]{  # entry condition: nw = 0
    while (true) {
        P(e);
        if (nw > 0) { dr := dr + 1;  # < await (nw=0)
```

Semaphores in Java

Basic Methods of Semaphores in Java

- `Semaphore(int n)`
 - constructor for semaphores
 - initializes semaphore value with integer *n set of permits*
- `acquire()`
 - corresponds to the P operation
 - tries to decrease the number of permits by 1
 - blocks, if that is not possible and waits, until semaphore gives permit
- `release()`
 - corresponds to the V operation
 - increases the number of permits by 1

Dining Philosophers: Naïve Solution in Java (I)

Philosophers in Java

- Philosopher has references to two binary Semaphores (leftFork and rightFork),
- and the functions eat(), sleep() and run()

Java

```
Semaphore[] forks = new Semaphore[numberOfPhilosophers];
for (int i=0; i < forks.length; i++)
    forks[i] = new Semaphore(1);

philosophers = new Philosopher[numberOfPhilosophers];
for (int i=0; i < philosophers.length; i++)
    philosophers[i] =
        new Philosopher(i, forks[i], forks[(i+1) % forks.length]);
```

Dining Philosophers: Naïve Solution in Java (II)

Java

```
while(true) {
    think();                                // think
    if(i == 0) {
        rightFork.acquire();                // acquire forks
        leftFork.acquire();
    } else {
        leftFork.acquire();                // acquire forks
        rightFork.acquire();
    }
    eat();                                    // eat
    leftFork.release();                     // release forks
    rightFork.release();
}
```

The Condition Interface

- A **condition** allows to transfer the ownership of the lock without lock/unlock
- Each condition is, thus, bound to a lock

The Condition interface includes the following methods:

- `cond.await()`
 - The lock associated with the Condition is atomically released (unlock) and the thread becomes disabled
 - After cond is signalled, the thread continues with its instructions.
- `cond.signal()`
 - Wakes up one thread that is waiting on this Condition
- Note: threads interacting with cond still need to acquire and release its lock!

```
Lock mutex = new ReentrantLock();
Condition condition = mutex.newCondition();

public void waitingThread() throws InterruptedException {
    mutex.lock();           // thread acquires the lock
    try {
        while(/*not finished*/) {
            condition.await();      // Release the lock and wait for signal
            /* thread does something (1) */
        }
    } finally {
        mutex.unlock();    // thread releases the lock
    }
}
```

Java

```
Lock mutex = new ReentrantLock();
Condition condition = mutex.newCondition();

public void signallingThread() throws InterruptedException {
    mutex.lock();                      // thread acquires the lock ;
    try {
        /* thread does something (2) */
        condition.signal();           // Signal (wake up) one waiting thread
    } finally {
        mutex.unlock();              // thread releases the lock
    }
}
```

Producer Consumer with Locks and Conditions

Example can be found at: [Example link](#)

Conclusion

Condition synchronization

- One semaphore to protect shared variables (the counters)
- For each condition: a semaphore + a “delay” counter
- On entry: increase delay counter if your condition is not true
- Wait on your condition semaphore
- Decide who is next (SIGNAL) using
 - the conditions, and
 - the delay counters to see who is waiting to enter
- SIGNAL whenever someone should get a chance to enter.

Semaphores

Silvia Lizeth Tapia Tarifa

September 10, 2025

University of Oslo

Overview

Last lecture: Locks and Barriers

- Complex techniques
- No clear separation between variables for synchronization and variables for computation
- Busy waiting

Overview

Last lecture: Locks and Barriers

- Complex techniques
- No clear separation between variables for synchronization and variables for computation
- Busy waiting

This lecture: Semaphores

- Synchronization tool
- Used easily for mutual exclusion and condition synchronization
- A way to implement signaling and scheduling
- Implementable in many ways on hardware (CMPXCHG)
- Available in programming language libraries and OS

Outline

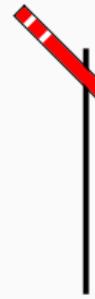
- Semaphores: Syntax and semantics
- Synchronization examples:
 - Mutual exclusion (critical sections)
 - Barriers (signaling events)
 - Producers and consumers (split binary semaphores)
 - Bounded buffer: resource counting
 - Dining philosophers: mutual exclusion – deadlock
 - Readers and writers:
 - condition synchronization
 - passing the baton

Semaphores

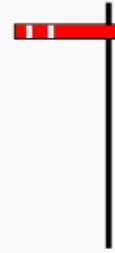
Semaphores

Origins of Term

- Introduced by Dijkstra in 1968
- Inspired by railroad traffic synchronization
- Railroad semaphore indicates whether the track ahead is clear or occupied by another train



Clear

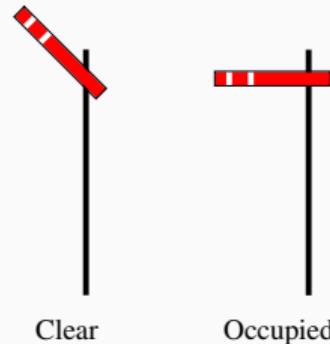


Occupied

Semaphores

Origins of Term

- Introduced by Dijkstra in 1968
- Inspired by railroad traffic synchronization
- Railroad semaphore indicates whether the track ahead is clear or occupied by another train



Properties

- Semaphores in concurrent programs: work similarly
- Used to implement
 - *mutex* and
 - *condition synchronization*
- Included in most standard libraries for concurrent programming
- Also *system calls* in, e.g., Linux kernel, Windows etc.

Concept

Concept of a Semaphore

- *Semaphore*: special kind of shared program variable (with built-in sync. power)
- value of a semaphore: a *non-negative* integer
- can *only* be manipulated by two *atomic* operations:

Concept

Concept of a Semaphore

- *Semaphore*: special kind of shared program variable (with built-in sync. power)
- value of a semaphore: a *non-negative* integer
- can *only* be manipulated by two *atomic* operations:

The Semaphore Operations: *P* and *V*

- **P**: (Passeren) Wait for signal – want to *pass*
Wait until value is greater than zero, and *decrease* value by one
- **V**: (Vrijgeven) Signal an event – *release*
Increase the value by one

Concept

Concept of a Semaphore

- *Semaphore*: special kind of shared program variable (with built-in sync. power)
- value of a semaphore: a *non-negative* integer
- can *only* be manipulated by two *atomic* operations:

The Semaphore Operations: *P* and *V*

- **P**: (Passeren) Wait for signal – want to *pass*
Wait until value is greater than zero, and *decrease* value by one
- **V**: (Vrijgeven) Signal an event – *release*
Increase the value by one

- Today, libraries and sys-calls prefer other names: up/down, wait/signal, acquire/release
- Different flavors of semaphores: binary vs. counting
- Most common: mutex as a synonym for binary semaphores

Declaration

- sem s; default initial value is zero
- sem s := 1;
- sem s[4] := ([4] 1);

Syntax and Semantics

Declaration

- sem s; default initial value is zero
- sem s := 1;
- sem s[4] := ([4] 1);

Operations and Semantics

P-operation P(s)

$\langle \text{await } (s > 0) s := s - 1 \rangle$

V-operation V(s)

$\langle s := s + 1 \rangle$

Processes waiting on a semaphore are woken up by the op. system.

Remarks on Semaphores

Remark 1

Important: No *direct* access to the value of a semaphore.

For example, a test like if $(s = 1)$ then ... else is *forbidden*!

Remarks on Semaphores

Remark 1

Important: No *direct* access to the value of a semaphore.

For example, a test like if $(s = 1)$ then ... else is *forbidden*!

Kinds of semaphores

General semaphore: Possible values: *all non-negative integers*

Binary semaphore: Possible values: 0 and 1

Remarks on Semaphores

Remark 1

Important: No *direct* access to the value of a semaphore.

For example, a test like if $(s = 1)$ then ... else is *forbidden*!

Kinds of semaphores

General semaphore: Possible values: *all non-negative integers*

Binary semaphore: Possible values: 0 and 1

Fairness

- As for await-statements.
- In most languages: *FIFO* (“waiting queue”): processes delayed while executing P-operations are *awoken* in the *order* they were delayed

Example: Mutual Exclusion (critical section)

Mutex implemented by a *binary semaphore*

Await

```
sem mutex := 1;
process CS[ i = 1 to n] {
    while (true) {
        P(mutex);
        # critical section
        V(mutex);
        # noncritical section
    }
}
```

- The semaphore is *initially 1*
- Always P before V → (used as) binary semaphore

Example: Barrier Synchronization

Semaphores may be used for *signaling events*

Await

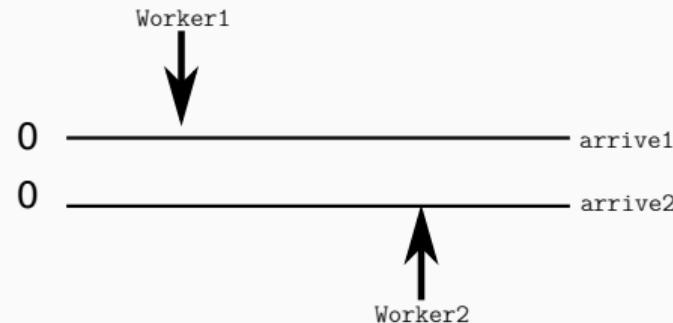
```
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
    ...
    V(arrive1); # reach barrier
    P(arrive2); # wait for other
    ...
}
process Worker2 {
    ...
    V(arrive2); # reach barrier
    P(arrive1); # wait for other
    ...
}
```

Example: Barrier Synchronization

- *Signalling* semaphores: usually *initialized* to 0 and
- *Signal* with a V and then *wait* with a P

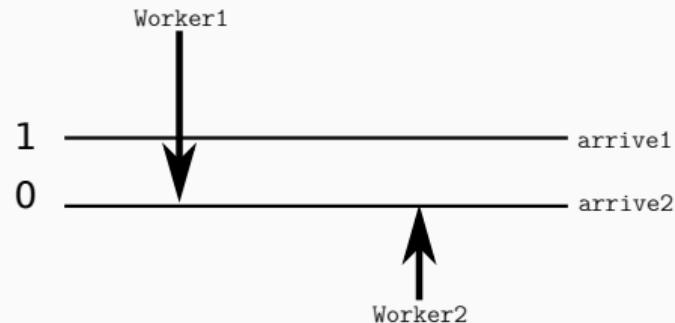
Example: Barrier Synchronization

- *Signalling* semaphores: usually *initialized* to 0 and
- *Signal* with a V and then *wait* with a P



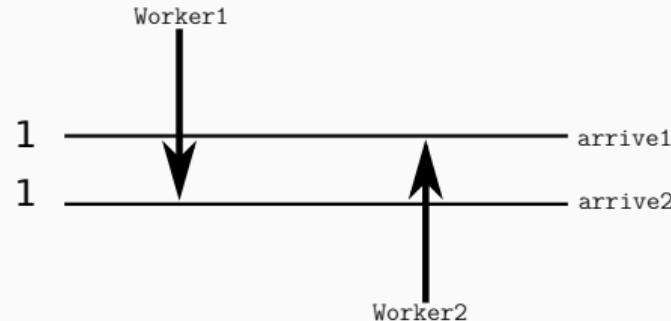
Example: Barrier Synchronization

- *Signalling* semaphores: usually *initialized* to 0 and
- *Signal* with a V and then *wait* with a P



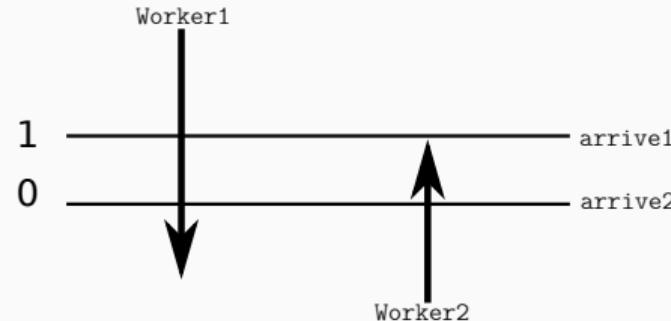
Example: Barrier Synchronization

- *Signalling semaphores*: usually *initialized to 0* and
- *Signal* with a V and then *wait* with a P



Example: Barrier Synchronization

- *Signalling semaphores*: usually *initialized to 0* and
- *Signal* with a V and then *wait* with a P



Split Binary Semaphores

Split binary semaphore

A set of semaphores, whose *sum* ≤ 1

Mutex by split binary semaphores

- Initialization: *one* of the semaphores = 1, all others = 0
- Discipline: all processes call *P* on a semaphore, *before* calling *V* on (*another*) semaphore
⇒ Code between the *P* and the *V*
 - All semaphores = 0
 - Code executed *in mutex*

Example: Producer/Consumer with Split Binary Semaphores

Await

```
T buf; # one element buffer, some type T  
sem empty := 1;  
sem full := 0;
```

Await

```
process Producer {  
    while (true) {  
        P(empty);  
        buff := data;  
        V(full);  
    }  
}
```

Await

```
process Consumer {  
    while (true) {  
        P(full);  
        data_c := buff;  
        V(empty);  
    }  
}
```

Example: Producer/Consumer with Split Binary Semaphores

Await

```
T buf; # one element buffer, some type T  
sem empty := 1;  
sem full := 0;
```

Await

```
process Producer {  
    while (true) {  
        P(empty);  
        buff := data;  
        V(full);  
    }  
}
```

Await

```
process Consumer {  
    while (true) {  
        P(full);  
        data_c := buff;  
        V(empty);  
    }  
}
```

- empty and full are both *binary semaphores*, together they form a split binary semaphore.
- Solution works with *several* producers/consumers

Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {  
    while (true) {  
        P(empty);  
        buff := data;  
        V(full);  
    }  
}
```

Await

```
process Consumer {  
    while (true) {  
        P(full);  
        data_c := buff;  
        V(empty);  
    }  
}
```

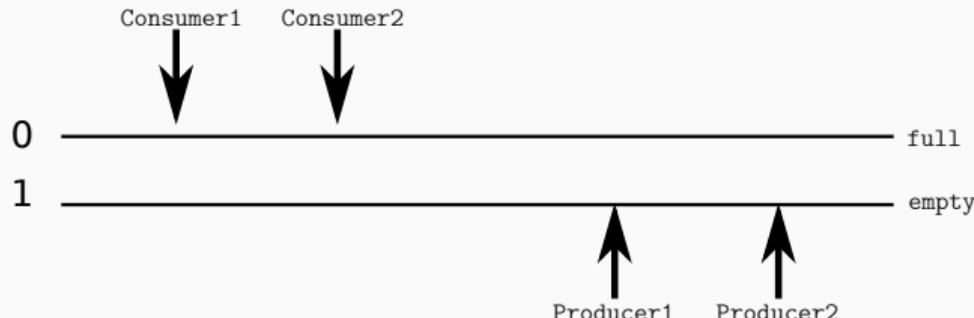
Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {  
    while (true) {  
        P(empty);  
        buff := data;  
        V(full);  
    }  
}
```

Await

```
process Consumer {  
    while (true) {  
        P(full);  
        data_c := buff;  
        V(empty);  
    }  
}
```



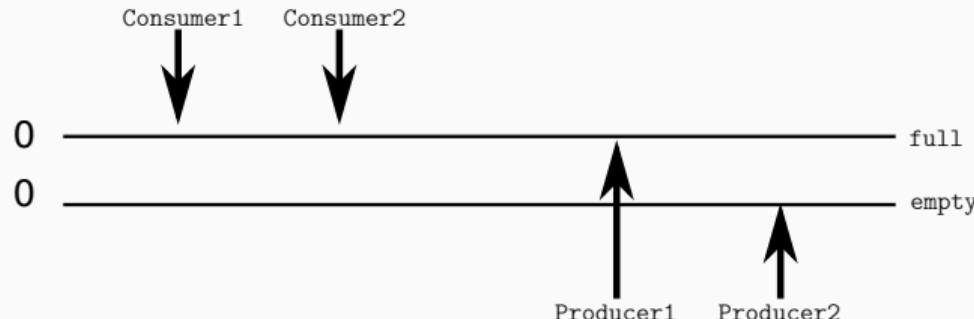
Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {  
    while (true) {  
        P(empty);  
        buff := data;  
        V(full);  
    }  
}
```

Await

```
process Consumer {  
    while (true) {  
        P(full);  
        data_c := buff;  
        V(empty);  
    }  
}
```



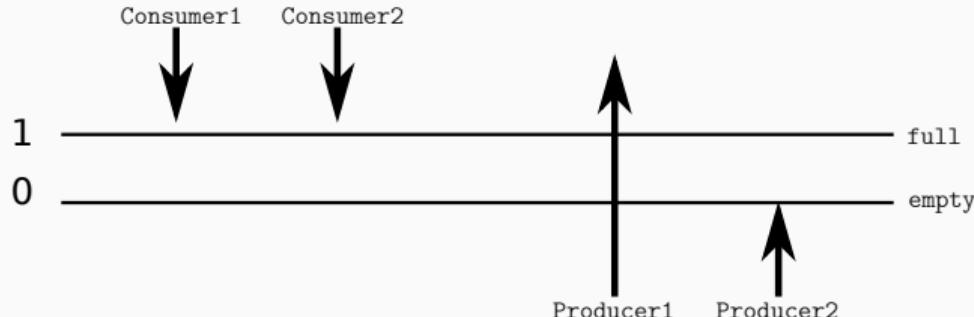
Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {  
    while (true) {  
        P(empty);  
        buff := data;  
        V(full);  
    }  
}
```

Await

```
process Consumer {  
    while (true) {  
        P(full);  
        data_c := buff;  
        V(empty);  
    }  
}
```



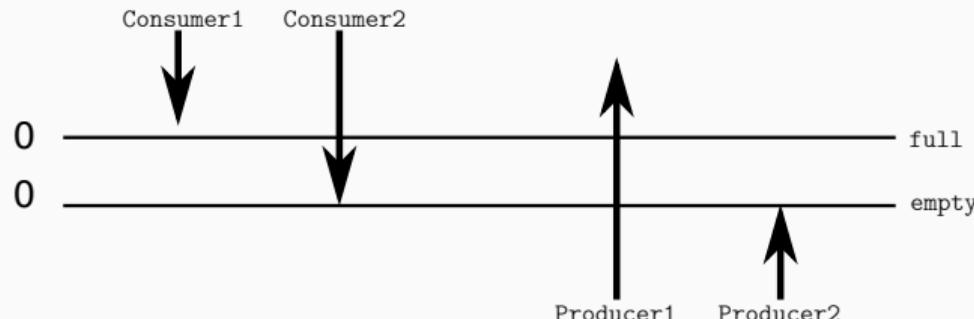
Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {  
    while (true) {  
        P(empty);  
        buff := data;  
        V(full);  
    }  
}
```

Await

```
process Consumer {  
    while (true) {  
        P(full);  
        data_c := buff;  
        V(empty);  
    }  
}
```



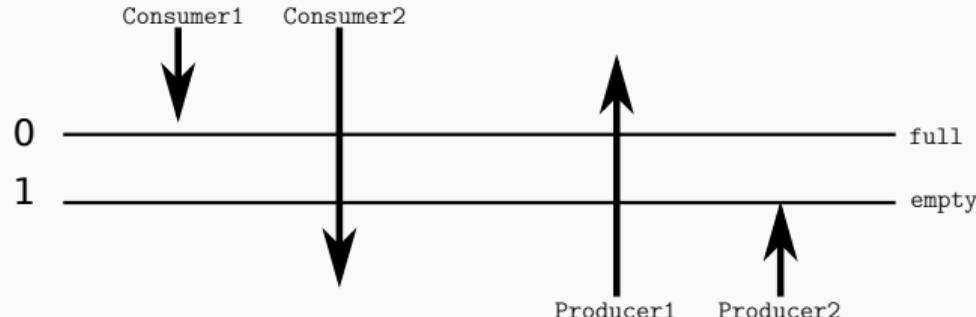
Example: Producer/Consumer with Split Binary Semaphores

Await

```
process Producer {  
    while (true) {  
        P(empty);  
        buff := data;  
        V(full);  
    }  
}
```

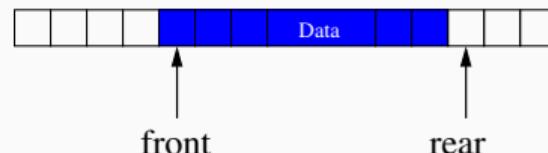
Await

```
process Consumer {  
    while (true) {  
        P(full);  
        data_c := buff;  
        V(empty);  
    }  
}
```



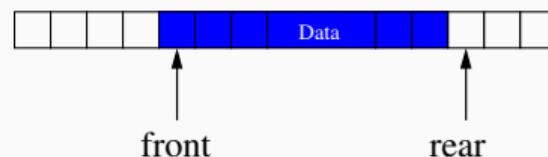
Producer/Consumer: Increasing Buffer Capacity

- Previously: tight coupling, the producer must wait for the consumer to empty the buffer before it can produce a new entry.
- Easy *generalization*: buffer of size n .
- Loose coupling/asynchronous communication \Rightarrow “buffering”
 - *Ring-buffer*, typically represented
 - by an array
 - + two integers `rear` and `front`.
 - Semaphores to *keep track* of the number of free/used slots



Producer/Consumer: Increasing Buffer Capacity

- Previously: tight coupling, the producer must wait for the consumer to empty the buffer before it can produce a new entry.
- Easy *generalization*: buffer of size n .
- Loose coupling/asynchronous communication \Rightarrow “buffering”
 - *Ring-buffer*, typically represented
 - by an array
 - + two integers **rear** and **front**.
 - Semaphores to *keep track* of the number of free/used slots \Rightarrow general semaphore



Producer/Consumer: Increased Buffer Capacity

Await

```
T buf[n]                      # array , elements of type T
int front := 0, rear := 0; # ``pointers''
sem empty := n;             # number of empty slots
sem full := 0;              # number of filled slots
```

Await

```
process Producer {
    while (true) {
        P(empty);
        buff[rear] := data;
        rear := (rear + 1);
        V(full);
    }
}
```

Await

```
process Consumer {
    while (true) {
        P(full);
        result := buff[front];
        front := (front + 1);
        V(empty);
    }
}
```

Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {
    while (true) {
        P(empty);
        buff[rear] := data;
        rear := (rear + 1);
        V(full);
    }
}
```

full 0

empty 3

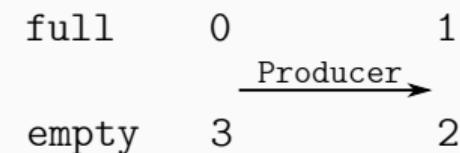
Await

```
process Consumer {
    while (true) {
        P(full);
        result := buff[front];
        front := (front + 1);
        V(empty);
    }
}
```

Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {
    while (true) {
        P(empty);
        buff[rear] := data;
        rear := (rear + 1);
        V(full);
    }
}
```



Await

```
process Consumer {
    while (true) {
        P(full);
        result := buff[front];
        front := (front + 1);
        V(empty);
    }
}
```

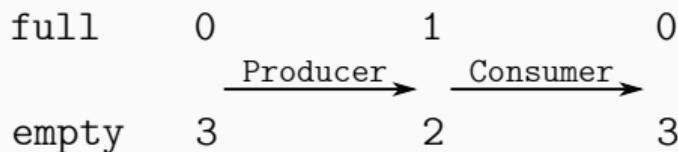
Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {
    while (true) {
        P(empty);
        buff[rear] := data;
        rear := (rear + 1);
        V(full);
    }
}
```

Await

```
process Consumer {
    while (true) {
        P(full);
        result := buff[front];
        front := (front + 1);
        V(empty);
    }
}
```



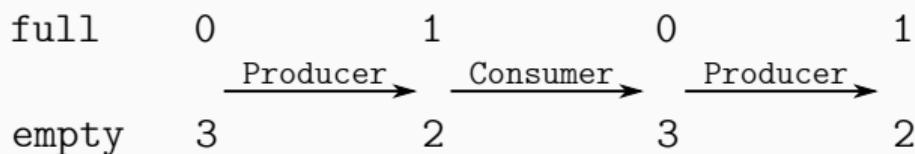
Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {  
    while (true) {  
        P(empty);  
        buff[rear] := data;  
        rear := (rear + 1);  
        V(full);  
    }  
}
```

Await

```
process Consumer {  
    while (true) {  
        P(full);  
        result := buff[front];  
        front := (front + 1);  
        V(empty);  
    }  
}
```



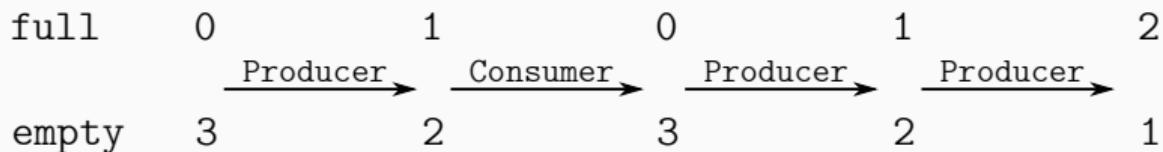
Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {
    while (true) {
        P(empty);
        buff[rear] := data;
        rear := (rear + 1);
        V(full);
    }
}
```

Await

```
process Consumer {
    while (true) {
        P(full);
        result := buff[front];
        front := (front + 1);
        V(empty);
    }
}
```



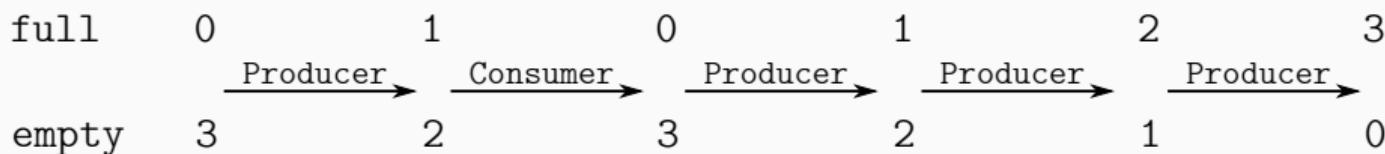
Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {  
    while (true) {  
        P(empty);  
        buff[rear] := data;  
        rear := (rear + 1);  
        V(full);  
    }  
}
```

Await

```
process Consumer {  
    while (true) {  
        P(full);  
        result := buff[front];  
        front := (front + 1);  
        V(empty);  
    }  
}
```



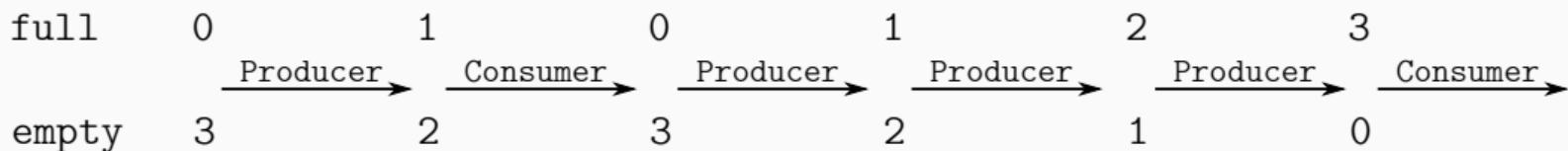
Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {
    while (true) {
        P(empty);
        buff[rear] := data;
        rear := (rear + 1);
        V(full);
    }
}
```

Await

```
process Consumer {
    while (true) {
        P(full);
        result := buff[front];
        front := (front + 1);
        V(empty);
    }
}
```



Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {
    while (true) {
        P(empty);
        buff[rear] := data;
        rear := (rear + 1);
        V(full);
    }
}
```

Await

```
process Consumer {
    while (true) {
        P(full);
        result := buff[front];
        front := (front + 1);
        V(empty);
    }
}
```

- Important: there are no critical sections!

Producer/Consumer: Increased Buffer Capacity

Await

```
process Producer {
    while (true) {
        P(empty);
        buff[rear] := data;
        rear := (rear + 1);
        V(full);
    }
}
```

Await

```
process Consumer {
    while (true) {
        P(full);
        result := buff[front];
        front := (front + 1);
        V(empty);
    }
}
```

- Important: there are no critical sections!
- How to enable several producers and consumers?

Increasing the Number of Processes

How to enable several producers and consumers?

New synchronization problems

- *Avoid that two producers deposit to buf [rear] before rear is updated*
- *Avoid that two consumers fetch from buf [front] before front is updated.*

Increasing the Number of Processes

How to enable several producers and consumers?

New synchronization problems

- *Avoid* that two producers *deposit* to buf [rear] before rear is updated
- *Avoid* that two consumers *fetch* from buf [front] before front is updated.

Solution

Add 2 extra binary semaphores for protection:

- mutexDeposit to deny two producers to deposit to the buffer at the same time.
- mutexFetch to deny two consumers to fetch from the buffer at the same time.

Example: Producer/Consumer with Several Processes

Await

```
T buf[n]                      # array, elements of type T
int front := 0, rear := 0; # ``pointers''
sem empty := n;             # number of empty slots
sem full := 0;              # number of filled slots
sem mutexDeposit; mutexFetch := 1; # protect the data stuct.
```

Await

```
process Producer {
    while (true) {
        P(empty);
        P(mutexDeposit);
        buff[rear] := data;
        rear := (rear + 1);
        V(mutexDeposit);
        V(full);
    }
}
```

Await

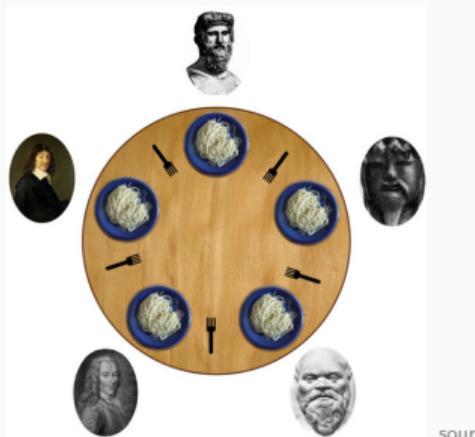
```
process Consumer {
    while (true) {
        P(full);
        P(mutexFetch);
        result := buff[front];
        front := (front + 1);
        V(mutexFetch);
        V(empty);
    }
}
```

Problem: Dining Philosophers



source:wikipedia.org

Problem: Dining Philosophers



source:wikipedia.org

- Famous sync. problem (Dijkstra)
- Five philosophers around a circular table.
- One fork placed between each pair of philosophers
- Each philosopher alternates between thinking and eating
- A philosopher needs two forks to eat (and none for thinking)

Dining Philosophers: Sketch

Await

```
process Philosopher [i = 0 to 4] {
    while true {
        # think
        acquire forks;
        # eat
        release forks;
    }
}
```

Task:

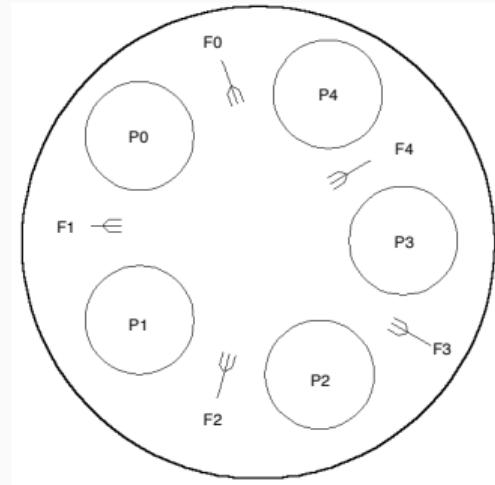
Program the actions `acquire forks` and `release forks`

Dining philosophers: 1st attempt

- Forks as *semaphores*
- Philosophers: pick up left fork first

Await

```
sem fork[5] := ([5] 1);
process Philosopher [ i = 0 to 4 ] {
    while true {
        # think
        P(fork[i]);
        P(fork[(i+1)%5]);
        # eat
        V(fork[i]);
        V(fork[(i+1)%5]);
    }
}
```



Dining philosophers: 2nd attempt

Breaking the symmetry

To avoid *deadlock*, let 1 philosopher (say 4) grab the *right* fork first

Await

```
process Philosopher [ i = 0 to 3] {
    while true {
        think;
        P(fork[i]);
        P(fork[(i+1)%5]);
        eat;
        V(fork[i]);
        V(fork[(i+1)%5]);
    }
}
```

Await

```
process Philosopher4 {
    while true {
        think;
        P(fork[4]); #!
        P(fork[0]); #!
        eat;
        V(fork[4]);
        V(fork[0]);
    }
}
```

Dining philosophers

- Important illustration of problems with concurrency:
 - Deadlocks,
 - Other aspects: liveness, fairness, etc.
- Resource access
- Connection to mutex/critical sections

Invariants and Condition Synchronization

Readers/Writers: Overview

- Classic synchronization problem
- *Reader* and *writer* processes, share access to a database/shared data structure
 - Readers only read from the database
 - Writers update (and read from) the database

Readers/Writers: Overview

- Classic synchronization problem
- *Reader* and *writer* processes, share access to a database/shared data structure
 - Readers only read from the database
 - Writers update (and read from) the database
- As soon as one writer is included, read and write accesses may cause interference
- Readers and writers have **asymmetric requirements:**
 - Every *writer* needs *mutually exclusive* access
 - When no writers have access, *many readers* may access the database

Readers/Writers: Approaches

- Dining philosophers: Pair of processes compete for access to “forks”
- Readers/writers: Different *classes* of processes compete for access to the database
 - Readers *compete* with writers
 - Writers *compete* both with readers and other writers
- **General synchronization problem:**
 - Readers: must wait until no writers are active in DB
 - Writers: must wait until no readers or writers are active in DB
- Here: two different approaches
 1. **Mutex:** easy to implement, but “*unfair*”
 2. **Condition synchronization:**
 - Using a *split binary semaphore*
 - Easy to adapt to different scheduling strategies

Readers/Writers with Mutex (1)

Await

```
sem rw := 1;
```

Await

```
process Reader [ i=1 to M] {
    while (true) {
        P(rw);
        # read
        V(rw);
    }
}
```

Await

```
process Writer [ i=1 to N] {
    while (true) {
        P(rw);
        # write
        V(rw);
    }
}
```

Readers/Writers with Mutex (1)

Await

```
sem rw := 1;
```

Await

```
process Reader [ i=1 to M] {
    while (true) {
        P(rw);
        # read
        V(rw);
    }
}
```

Await

```
process Writer [ i=1 to N] {
    while (true) {
        P(rw);
        # write
        V(rw);
    }
}
```

We want *more than one reader simultaneously*.

Readers/Writers with Mutex (2)

Await

```
int nr := 0;    # number of active readers
sem rw := 1      # lock for reader/writer mutex
```

Await

```
process Reader [ i=1 to M] {
    while (true) {
        < nr := nr + 1;
        if (nr=1) P(rw) >;
        # read
        < nr := nr - 1;
        if (nr=0) V(rw) > ;
    }
}
```

Await

```
process Writer [ i=1 to N] {
    while (true) {
        P(rw);
        # write
        V(rw);
    }
}
```

Readers/Writers with Mutex (2)

Await

```
int nr := 0;    # number of active readers
sem rw := 1      # lock for reader/writer mutex
```

Await

```
process Reader [ i=1 to M] {
    while (true) {
        < nr := nr + 1;
        if (nr=1) P(rw) >;
        # read
        < nr := nr - 1;
        if (nr=0) V(rw) >;
    }
}
```

Await

```
process Writer [ i=1 to N] {
    while (true) {
        P(rw);
        # write
        V(rw);
    }
}
```

How do semaphores work *inside* atomic sections?

Readers/Writers with Mutex (3)

Await

```
int      nr = 0; # number of      active readers
sem      rw = 1; # lock for reader/writer exclusion
sem mutexR = 1; # mutex for readers

process Reader [ i=1 to M] {
    while (true) {
        P(mutexR)
        nr := nr + 1;
        if (nr=1)  P(rw);
        V(mutexR)
        # read
        P(mutexR)
        nr := nr - 1;
        if (nr=0)  V(rw);
        V(mutexR)
    }
}
```

Readers/Writers with Condition Synchronization: Overview

Reader's preference

- With a constant stream of readers, the writer will never run
 - Even under strong fairness
-
- Previous *mutex* solution solved *two* separate synchronization problems
 - ***rw*** : *Readers and writers* for access to the *database*
 - ***mutexR***: *Reader vs. reader* for access to the *counter*
 - Now: a solution based on **condition synchronization**

Invariant

Reasonable invariant for the critical sections

1. When a *writer* accesses the DB, *no one else* can
2. When *no writers* access the DB, *one or more readers* may get access

Invariant

Reasonable invariant for the critical sections

1. When a *writer* accesses the DB, *no one else* can
2. When *no writers* access the DB, *one or more readers* may get access

Introducing state for the invariant

Introduce two counters:

- **nr**: number of active readers
- **nw**: number of active writers

Invariant

Reasonable invariant for the critical sections

1. When a *writer* accesses the DB, *no one else* can
2. When *no writers* access the DB, *one or more readers* may get access

Introducing state for the invariant

Introduce two counters:

- **nr**: number of active readers
- **nw**: number of active writers

Invariant

$RW: (nr = 0 \text{ or } nw = 0) \text{ and } nw \leq 1$
(same as:) $RW': nw=0 \text{ or } (nw = 1 \text{ and } nr = 0)$

Code for counting Readers and Writers

Await

```
< nr := nr + 1; >  
# read  
< nr := nr - 1; >
```

Await

```
< nw := nw + 1; >  
# write  
< nw := nw - 1; >
```

- Add synchronization code to maintain the invariant
- Decreasing counters is not dangerous
- Before increasing, we need to check some conditions for synchronization
 - before increasing nr: nw = 0
 - before increasing nw: nr = 0 and nw = 0

Condition Synchronization: Without Semaphores

Await

```
int nr := 0;    # number of active readers
int nw := 0;    # number of active writers
# Invariant RW: (nr = 0 or nw = 0) and nw <= 1
```

Await

```
process Reader [ i=1 to M]{
  while (true) {
    < await (nw=0)
      nr := nr+1>;
    # read
    < nr := nr - 1>
  }
}
```

Await

```
process Writer [ i=1 to N]{
  while (true) {
    < await (nr = 0 and nw = 0)
      nw := nw+1>;
    # write
    < nw := nw - 1>
  }
}
```

Condition Synchronization: Converting to Split Binary Semaphores

Convert awaits with different guards $B_1, B_2 \dots$ to Split Binary Semaphores

- *Entry:* semaphore e , initialized to 1
- For each guard B_i :
 1. associate one *counter* and
 2. one *delay-semaphore*

Both initialized to 0

- Semaphore *delays* the processes waiting for B_i
- Counters counts the number of *processes waiting* for B_i

Condition Synchronization: Converting to Split Binary Semaphores

Convert awaits with different guards $B_1, B_2 \dots$ to Split Binary Semaphores

- *Entry:* semaphore e , initialized to 1
- For each guard B_i :
 1. associate one *counter* and
 2. one *delay-semaphore*

Both initialized to 0

- Semaphore *delays* the processes waiting for B_i
- Counters counts the number of processes *waiting* for B_i

For *readers/writers* problem we need 3 semaphores and 2 counters:

Await _____

```
sem e = 1;
sem r = 0; int dr = 0; # condition reader: nw == 0
sem w = 0; int dw = 0; # condition writer: nr == 0 and nw == 0
```

Condition Synchronization: Converting to Split Binary Semaphores (2)

- e , r and w form a *split binary semaphore*.
- All execution paths *start* with a *P-operation* and *end* with a *V-operation* → Mutex

Condition Synchronization: Converting to Split Binary Semaphores (2)

- e , r and w form a *split binary semaphore*.
- All execution paths *start* with a *P-operation* and *end* with a *V-operation* → Mutex

Signaling

We need a signal mechanism *SIGNAL* to pick which semaphore to signal.

- SIGNAL: make sure the invariant holds
- B_i holds when a process enters *CS* because either:
 - the process checks itself,

Condition Synchronization: Converting to Split Binary Semaphores (2)

- e , r and w form a *split binary semaphore*.
- All execution paths *start* with a *P-operation* and *end* with a *V-operation* → Mutex

Signaling

We need a signal mechanism *SIGNAL* to pick which semaphore to signal.

- SIGNAL: make sure the invariant holds
- B_i holds when a process enters *CS* because either:
 - the process checks itself,
 - or the process is only *signalled* if B_i holds

Condition Synchronization: Converting to Split Binary Semaphores (2)

- e , r and w form a *split binary semaphore*.
- All execution paths *start* with a *P-operation* and *end* with a *V-operation* → Mutex

Signaling

We need a signal mechanism *SIGNAL* to pick which semaphore to signal.

- SIGNAL: make sure the invariant holds
- B_i holds when a process enters *CS* because either:
 - the process checks itself,
 - or the process is only *signalled* if B_i holds
- **Another pitfall:**

Avoid *deadlock* by checking the counters before the delay semaphores are signalled.

 - r is not signalled ($V(r)$) *unless* there is a delayed reader
 - w is not signalled ($V(w)$) *unless* there is a delayed writer

Condition Synchronization: Reader

Await

```
int nr := 0, nw = 0;      # counter variables (as before)
sem e := 1;                # entry semaphore
int dr := 0; sem r := 0; # delay counter + sem for reader
int dw := 0; sem w := 0; # delay counter + sem for writer
# invariant RW: (nr = 0 or nw = 0 ) and nw <= 1
process Reader [i=1 to M]{ # entry condition: nw = 0
    while (true) {
        P(e);
        if (nw > 0) { dr := dr + 1;    # < await (nw=0)
                        V(e);          #     nr:=nr+1 >
                        P(r)};
        nr:=nr+1; SIGNAL;
        # read
        P(e); nr:=nr-1; SIGNAL;      # < nr:=nr-1 >
    }
}
```

With Condition Synchronization: Writer

Await

```
process Writer [i=1 to N]{ # entry condition: nw = 0 and nr = 0
    while (true) {
        P(e);                                # < await (nr=0 and nw=0)
        if (nr > 0 or nw > 0) {      #     nw:=nw+1 >
            dw := dw + 1;
            V(e);
            P(w) };
        nw:=nw+1; SIGNAL;
        # write
        P(e); nw:=nw-1; SIGNAL      # < nw:=nw-1>
    }
}
```

With Condition Synchronization: Signalling

Await

```
if (nw = 0 and dr > 0) {  
    dr := dr -1; V(r);           # awake reader  
}  
elseif (nr = 0 and nw = 0 and dw > 0) {  
    dw := dw -1; V(w);         # awake writer  
}  
else V(e);                      # release entry lock
```

- This passes the control (the “baton”) to an appropriate next process
- SIGNAL has no P operation, each path has exactly one V operation.
- Using the conditions to see who goes next.
- Called “passing the baton” technique (as in relay competition).
- Conditions for awakening must be disjoint

Example: 1 Reader, 1 Writer, Reader starts

nr	0
nw	0
e	1
dw	0
w	0

Await

```
process Reader [i=1 to M]{ # entry condition: nw = 0
    while (true) {
        P(e);
        if (nw > 0) { dr := dr + 1;   # < await (nw=0)
                        V(e);          #      nr:=nr+1 >
                        P(r)};
        nr:=nr+1; SIGNAL;
        # read
        P(e); nr:=nr-1; SIGNAL;      # < nr:=nr-1 >
    }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0
nw	0	0
e	1	0
dw	0	0
w	0	0

Await

```
process Reader [i=1 to M]{ # entry condition: nw = 0
    while (true) {
        P(e);
        if (nw > 0) { dr := dr + 1;   # < await (nw=0)
                        V(e);          #     nr:=nr+1 >
                        P(r)};
        nr:=nr+1; SIGNAL;
        # read
        P(e); nr:=nr-1; SIGNAL;      # < nr:=nr-1 >
    }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1
nw	0	0	0
e	1	0	0
dw	0	0	0
w	0	0	0

Await

```
if (nw = 0 and dr > 0) {
    dr := dr -1; V(r);                      # awake reader
}
elseif (nr = 0 and nw = 0 and dw > 0) {
    dw := dw -1; V(w);                      # awake writer
}
else V(e);                                # release entry lock
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1
nw	0	0	0	0
e	1	0	0	1
dw	0	0	0	0
w	0	0	0	0

Await

```
process Writer [i=1 to N]{ # entry condition: nw = 0 and nr = 0
    while (true) {
        P(e);                      # < await (nr=0 && nw=0)
        if (nr > 0 or nw > 0) {    #      nw:=nw+1 >
            dw := dw + 1;
            V(e);
            P(w) };
        nw:=nw+1; SIGNAL;
        # write
        P(e); nw:=nw-1; SIGNAL    # < nw:=nw-1>
    }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1
nw	0	0	0	0	0
e	1	0	0	1	0
dw	0	0	0	0	0
w	0	0	0	0	0

Await

```
process Writer [i=1 to N]{ # entry condition: nw = 0 and nr = 0
    while (true) {
        P(e);                      # < await (nr=0 && nw=0)
        if (nr > 0 or nw > 0) {    #      nw:=nw+1 >
            dw := dw + 1;
            V(e);
            P(w) };
        nw:=nw+1; SIGNAL;
        # write
        P(e); nw:=nw-1; SIGNAL   # < nw:=nw-1>
    }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1
nw	0	0	0	0	0	0
e	1	0	0	1	0	0
dw	0	0	0	0	0	1
w	0	0	0	0	0	0

Await

```
process Writer [i=1 to N]{ # entry condition: nw = 0 and nr = 0
    while (true) {
        P(e);                      # < await (nr=0 && nw=0)
        if (nr > 0 or nw > 0) {    #      nw:=nw+1 >
            dw := dw + 1;
            V(e);
            P(w) };
        nw:=nw+1; SIGNAL;
        # write
        P(e); nw:=nw-1; SIGNAL   # < nw:=nw-1>
    }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1
nw	0	0	0	0	0	0	0
e	1	0	0	1	0	0	1
dw	0	0	0	0	0	1	1
w	0	0	0	0	0	0	0

Await

```
process Reader [i=1 to M]{ # entry condition: nw = 0
    while (true) {
        P(e);
        if (nw > 0) { dr := dr + 1;   # < await (nw=0)
                        V(e);          #      nr:=nr+1 >
                        P(r)};
        nr:=nr+1; SIGNAL;
        # read
        P(e); nr:=nr-1; SIGNAL;      # < nr:=nr-1 >
    }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	1
nw	0	0	0	0	0	0	0	0
e	1	0	0	1	0	0	1	0
dw	0	0	0	0	0	1	1	1
w	0	0	0	0	0	0	0	0

Await

```
process Reader [i=1 to M]{ # entry condition: nw = 0
    while (true) {
        P(e);
        if (nw > 0) { dr := dr + 1;   # < await (nw=0)
                      V(e);          #     nr:=nr+1 >
                      P(r)};
        nr:=nr+1; SIGNAL;
        # read
        P(e); nr:=nr-1; SIGNAL;      # < nr:=nr-1 >
    }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	1	0
nw	0	0	0	0	0	0	0	0	0
e	1	0	0	1	0	0	1	0	0
dw	0	0	0	0	0	1	1	1	1
w	0	0	0	0	0	0	0	0	0

Await

```
if (nw = 0 and dr > 0) {
    dr := dr -1; V(r);                      # awake reader
}
elseif (nr = 0 and nw = 0 and dw > 0) {
    dw := dw -1; V(w);                      # awake writer
}
else V(e);                                # release entry lock
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	0	0
nw	0	0	0	0	0	0	0	0	0
e	1	0	0	1	0	0	1	0	0
dw	0	0	0	0	0	1	1	1	0
w	0	0	0	0	0	0	0	0	1

Await

```
if (nw = 0 and dr > 0) {
    dr := dr -1; V(r);                      # awake reader
}
elseif (nr = 0 and nw = 0 and dw > 0) {
    dw := dw -1; V(w);                      # awake writer
}
else V(e);                                # release entry lock
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	0	0	0
nw	0	0	0	0	0	0	0	0	0	0
e	1	0	0	1	0	0	1	0	0	0
dw	0	0	0	0	0	1	1	1	0	0
w	0	0	0	0	0	0	0	0	1	1

Await

```
process Writer [i=1 to N]{ # entry condition: nw = 0 and nr = 0
    while (true) {
        P(e);                      # < await (nr=0 && nw=0)
        if (nr > 0 or nw > 0) {    #      nw:=nw+1 >
            dw := dw + 1;
            V(e);
            P(w) };
        nw:=nw+1; SIGNAL;
        # write
        P(e); nw:=nw-1; SIGNAL   # < nw:=nw-1>
    }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	0	0	0	0
nw	0	0	0	0	0	0	0	0	0	0	1
e	1	0	0	1	0	0	1	0	0	0	0
dw	0	0	0	0	0	1	1	1	0	0	0
w	0	0	0	0	0	0	0	0	1	1	0

Await

```
process Writer [i=1 to N]{ # entry condition: nw = 0 and nr = 0
    while (true) {
        P(e);                      # < await (nr=0 && nw=0)
        if (nr > 0 or nw > 0) {    #      nw:=nw+1 >
            dw := dw + 1;
            V(e);
            P(w) };
        nw:=nw+1; SIGNAL;
        # write
        P(e); nw:=nw-1; SIGNAL   # < nw:=nw-1>
    }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	0	0	0	0	0
nw	0	0	0	0	0	0	0	0	0	0	1	1
e	1	0	0	1	0	0	1	0	0	0	0	1
dw	0	0	0	0	0	1	1	1	1	0	0	0
w	0	0	0	0	0	0	0	0	1	1	0	0

Await

```
process Writer [i=1 to N]{ # entry condition: nw = 0 and nr = 0
    while (true) {
        P(e);                      # < await (nr=0 && nw=0)
        if (nr > 0 or nw > 0) {    #      nw:=nw+1 >
            dw := dw + 1;
            V(e);
            P(w) };
        nw:=nw+1; SIGNAL;
        # write
        P(e); nw:=nw-1; SIGNAL   # < nw:=nw-1>
    }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	0	0	0	0	0	0
nw	0	0	0	0	0	0	0	0	0	0	1	1	1
e	1	0	0	1	0	0	1	0	0	0	0	1	0
dw	0	0	0	0	0	1	1	1	1	0	0	0	0
w	0	0	0	0	0	0	0	0	1	1	0	0	0

Await

```
process Writer [i=1 to N]{ # entry condition: nw = 0 and nr = 0
    while (true) {
        P(e);                      # < await (nr=0 && nw=0)
        if (nr > 0 or nw > 0) {    #      nw:=nw+1 >
            dw := dw + 1;
            V(e);
            P(w) };
        nw:=nw+1; SIGNAL;
        # write
        P(e); nw:=nw-1; SIGNAL   # < nw:=nw-1>
    }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0
nw	0	0	0	0	0	0	0	0	0	0	1	1	1	0	
e	1	0	0	1	0	0	1	0	0	0	0	1	0	0	
dw	0	0	0	0	0	1	1	1	1	0	0	0	0	0	
w	0	0	0	0	0	0	0	0	1	1	0	0	0	0	

Await

```
process Writer [i=1 to N]{ # entry condition: nw = 0 and nr = 0
    while (true) {
        P(e);                      # < await (nr=0 && nw=0)
        if (nr > 0 or nw > 0) {    #      nw:=nw+1 >
            dw := dw + 1;
            V(e);
            P(w) };
        nw:=nw+1; SIGNAL;
        # write
        P(e); nw:=nw-1; SIGNAL   # < nw:=nw-1>
    }
}
```

Example: 1 Reader, 1 Writer, Reader starts

nr	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
nw	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0
e	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0
dw	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
w	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0

Await

```
if (nw = 0 and dr > 0) {
    dr := dr -1; V(r);                      # awake reader
}
elseif (nr = 0 and nw = 0 and dw > 0) {
    dw := dw -1; V(w);                      # awake writer
}
else V(e);                                # release entry lock
```

Semaphores in Java

Basic Methods of Semaphores in Java

- `Semaphore(int n)`
 - constructor for semaphores
 - initializes semaphore value with integer *n set of permits*
- `acquire()`
 - corresponds to the P operation
 - tries to decrease the number of permits by 1
 - blocks, if that is not possible and waits, until semaphore gives permit
- `release()`
 - corresponds to the V operation
 - increases the number of permits by 1

Dining Philosophers: Naïve Solution in Java (I)

Philosophers in Java

- Philosopher has references to two binary Semaphores (leftFork and rightFork),
- and the functions eat(), sleep() and run()

Java

```
Semaphore[] forks = new Semaphore[numberOfPhilosophers];
for (int i=0; i < forks.length; i++)
    forks[i] = new Semaphore(1);

philosophers = new Philosopher[numberOfPhilosophers];
for (int i=0; i < philosophers.length; i++)
    philosophers[i] =
        new Philosopher(i, forks[i], forks[(i+1) % forks.length]);
```

Dining Philosophers: Naïve Solution in Java (II)

Java

```
while(true) {
    think();                                // think
    if(i == 0) {
        rightFork.acquire();                // acquire forks
        leftFork.acquire();
    } else {
        leftFork.acquire();                // acquire forks
        rightFork.acquire();
    }
    eat();                                    // eat
    leftFork.release();                     // release forks
    rightFork.release();
}
```

The Condition Interface

- A **condition** allows to transfer the ownership of the lock without lock/unlock
- Each condition is, thus, bound to a lock

The Condition Interface

- A **condition** allows to transfer the ownership of the lock without lock/unlock
- Each condition is, thus, bound to a lock

The Condition interface includes the following methods:

- `cond.await()`
 - The lock associated with the Condition is atomically released (unlock) and the thread becomes disabled
 - After cond is signalled, the thread continues with its instructions.
- `cond.signal()`
 - Wakes up one thread that is waiting on this Condition

The Condition Interface

- A **condition** allows to transfer the ownership of the lock without lock/unlock
- Each condition is, thus, bound to a lock

The Condition interface includes the following methods:

- `cond.await()`
 - The lock associated with the Condition is atomically released (unlock) and the thread becomes disabled
 - After cond is signalled, the thread continues with its instructions.
- `cond.signal()`
 - Wakes up one thread that is waiting on this Condition
- Note: threads interacting with cond still need to acquire and release its lock!

```
Lock mutex = new ReentrantLock();
Condition condition = mutex.newCondition();

public void waitingThread() throws InterruptedException {
    mutex.lock();           // thread acquires the lock
    try {
        while(/*not finished*/) {
            condition.await();      // Release the lock and wait for signal
            /* thread does something (1) */
        }
    } finally {
        mutex.unlock();    // thread releases the lock
    }
}
```

Java

```
Lock mutex = new ReentrantLock();
Condition condition = mutex.newCondition();

public void signallingThread() throws InterruptedException {
    mutex.lock();                      // thread acquires the lock ;
    try {
        /* thread does something (2) */
        condition.signal();           // Signal (wake up) one waiting thread
    } finally {
        mutex.unlock();              // thread releases the lock
    }
}
```

Producer Consumer with Locks and Conditions

Example can be found at: [Example link](#)

Conclusion

Condition synchronization

- One semaphore to protect shared variables (the counters)
- For each condition: a semaphore + a “delay” counter
- On entry: increase delay counter if your condition is not true
- Wait on your condition semaphore
- Decide who is next (SIGNAL) using
 - the conditions, and
 - the delay counters to see who is waiting to enter
- SIGNAL whenever someone should get a chance to enter.

Message Passing and Channels

Andrea Pferscher

September 24, 2025

University of Oslo

Message Passing

Structure

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

Content of next part:

- Synchronous and asynchronous message passing
- Channels, actors, go-routines, asynchronous programming

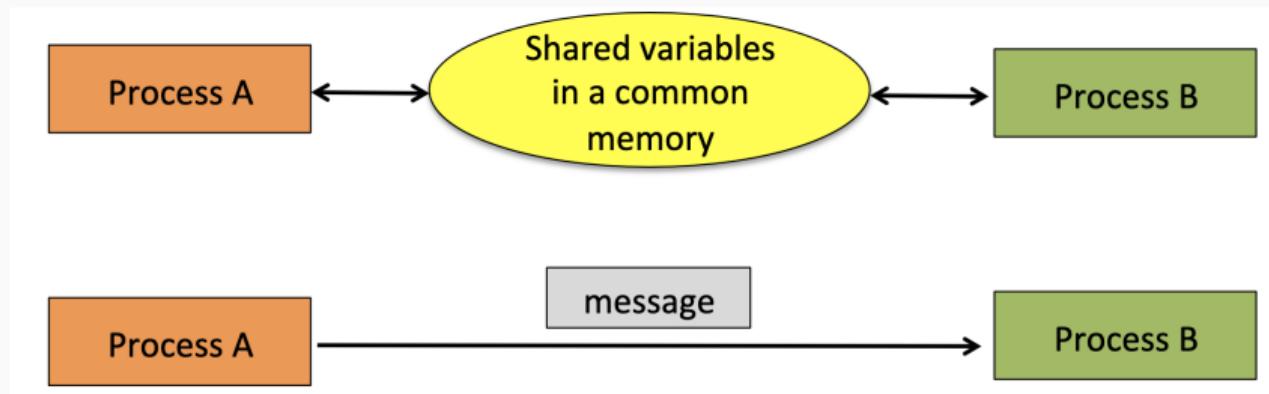
Outline today

- *Asynchronous message passing*: channels, messages, primitives
- Example: filters and sorting networks
- Comparison of message passing and monitors
- *Basics synchronous message passing*

Concurrent Programming: Shared State vs. Messages

Concurrent programming

- *Concurrent program*: two or more processes that work together to perform a task.
- The processes work together by communicating with each other using:
 - *Shared variables*: One process writes into a variable that is read by another.
 - *Message passing*: One process sends a message that is received by another



Program Synchronization (Recap)

Two kinds of synchronization approaches (regardless of the form of communication)

- Mutual exclusion (mutex)
 - A program mechanism that prevents processes from accessing a shared resource at the same time.
 - Only one process or thread owns the mutex at a time.
- Condition synchronization
 - Delay a process until a given condition is true.
- To prevent race condition: when concurrent processes access and change a shared resource.
- Used for critical section.

Recap

- So far: shared variable programming
- **Now:** Distributed programming

Distributed Systems

Shared Memory vs. Distributed Memory

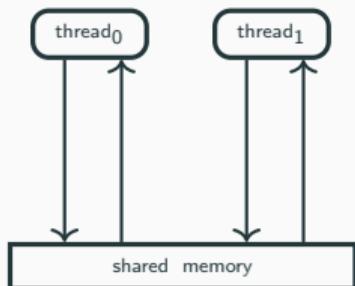
System architectures with shared memory:

- Many processors access the same physical memory
- Examples: laptops, fileservers with many processors on one motherboard

Distributed memory architectures:

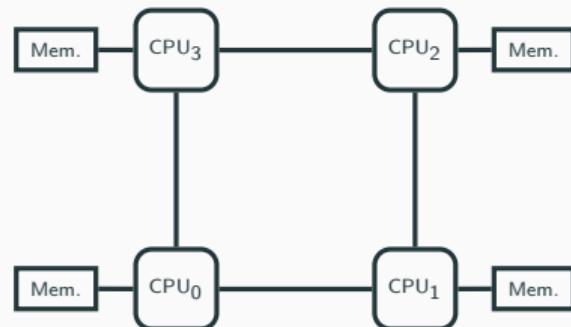
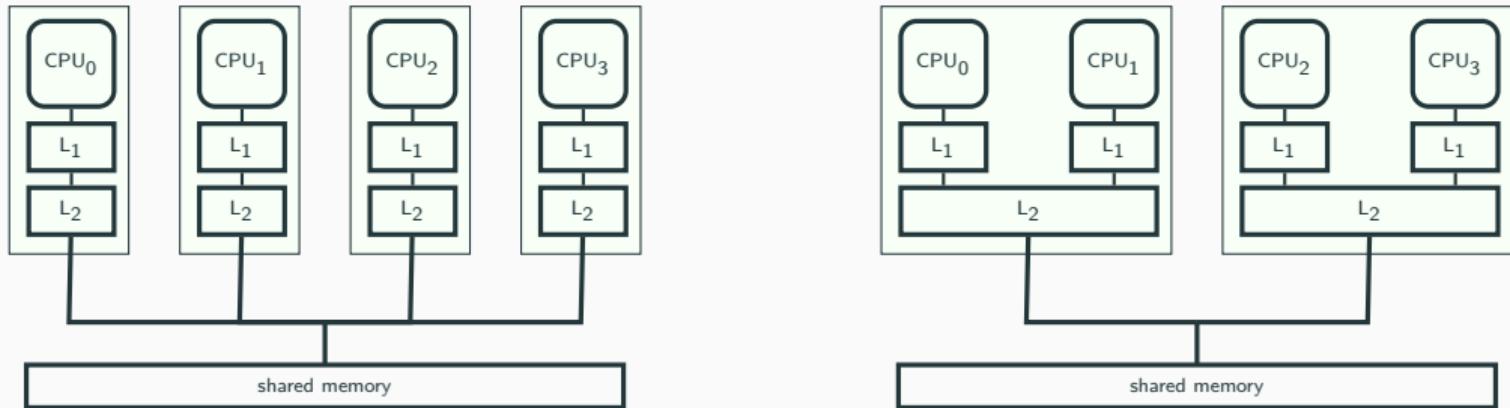
- Each processor has private memory, communication over connections in a “network”
- Examples:
 - Multicomputer: asynchronous multi-processor with distributed memory
 - Workstation clusters: PC's in a local network, NFS (Network File System)
 - Grid system: machines on the Internet, resource sharing
 - Cloud computing: cloud storage service
 - NUMA-architectures
 - Cluster computing ...

Shared Memory Concurrency in the Real World



- Shared memory architecture is a simplification
- Out-of-order executions:
 - Due to complex memory hierarchies, caches, buffers,...
 - Due to weak memory, micro-ops, compiler optimizations,...

SMP (Symmetric Multiprocessing), Multi-Core Architecture, and NUMA



Concurrent vs. Distributed Programming

Shared-Memory Systems

- Processors share one memory
- Processors communicate via reading and writing of shared variables

Concurrent programming provides primitives to synchronize over memory

Distributed Systems

- Memory is distributed: processes cannot share variables/memory locations
- Processes communicate by sending and receiving *messages* via e.g., shared *channels*,
- or (in future lectures): communication via *RPC* and *rendezvous*

Distributed programming provides primitives to communicate

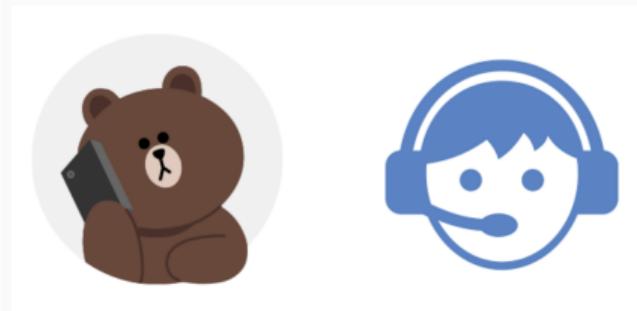
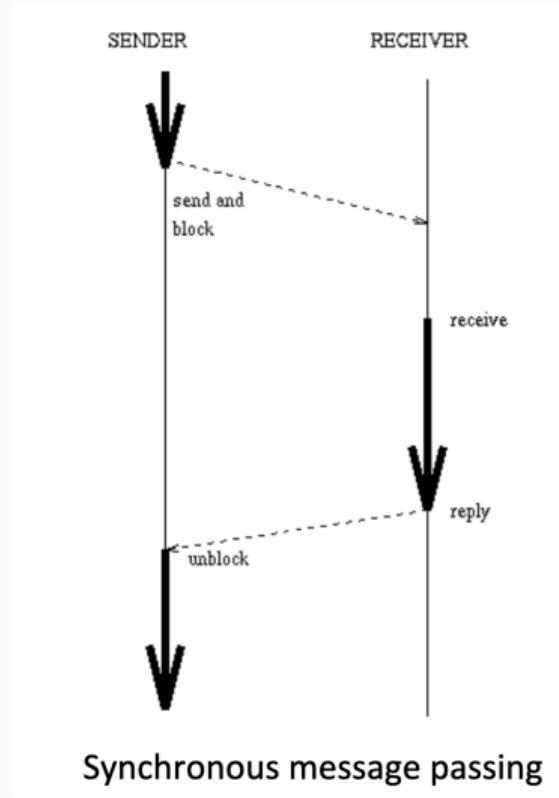
- Some concepts from distributed systems are also useful abstractions for shared memory
- Abstractions can be decoded to different primitives, e.g., channels as shared-memory
- Also: mixed shared-distributed systems

Synchronous and Asynchronous Message Passing

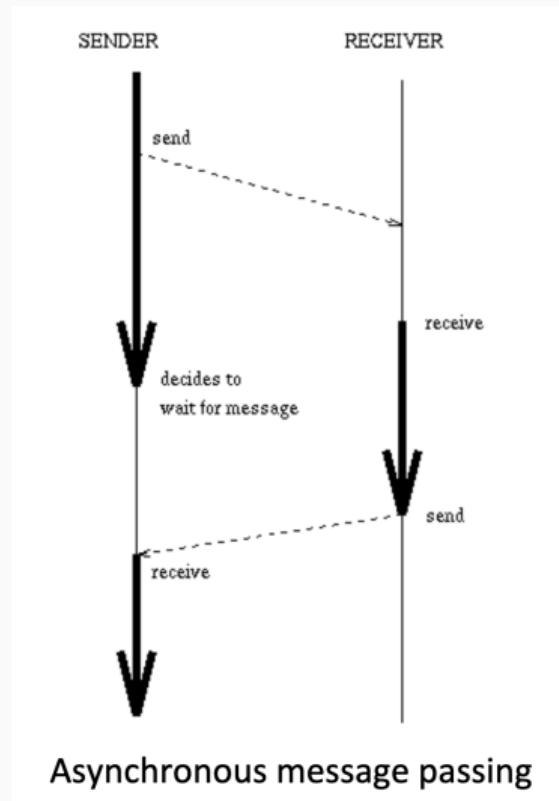
Message Passing

- *Message passing* refers to the sending of a message to a process.
- This message can be used to invoke a process
- Two types of message passing:
 - *Synchronous* message passing
 - *Asynchronous* message passing

Synchronous Message Passing: High Level Concept



Asynchronous Message Passing: High Level Concept



Synchronous vs. Asynchronous Message Passing: Trade Off

Synchronous message passing

- No memory buffer is required
- Concurrency is reduced
- Programs are more prone to deadlock

Asynchronous message passing

- Memory buffer is required (memory is cheap)
- Have more concurrency
- Programs are less prone to deadlock

We will comeback to this comparison later in the lecture.

Channels

Asynchronous Message Passing: Channel Abstraction

Channel

Abstraction, e.g., of a physical communication network, for one-way communication between two entities (similar to producer-consumer). For us:

- Unbounded FIFO (queue) of waiting messages
- Preserves message order
- Atomic access
- Error-free
- Typed

Numerous variants exists in different language: untyped, lossy, unnamed, bounded . . .
We will look at more complex types later

Asynchronous Message Passing: Primitives

Channel declaration

Await

```
chan c(type1 id1, ..., typeN idN);
```

Messages are n -tuples of respective types.

Communication primitives

- `send c(expr1, ..., exprN);`

Non-blocking, i.e. asynchronous: message is sent and process continues its execution

- `receive c(v1, ..., vN);`

Blocking: receiver process waits until message is sent on the channel

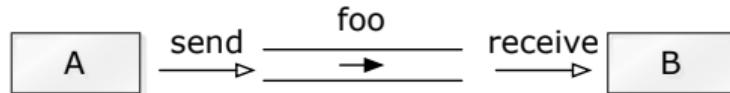
Message stored in variables $v1, \dots, vN$.

- `empty(c);`

True if channel is empty

Example: Message Passing

$(x,y) = (1,2)$



Await

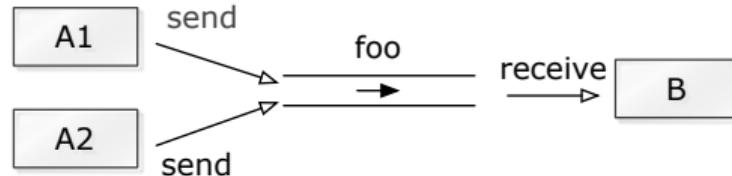
```
chan foo(int);

process A {
    send foo(1);
    send foo(2);}

process B {
    int x; int y;
    receive foo(x);
    receive foo(y);}
```

Example: Shared Channel

$(x,y) = (1,2)$ or $(2,1)$



Await

```
chan foo(int);
process A1 {
    send foo(1); }

process A2 {
    send foo(2); }

process B {
    int x; int y;
    receive foo(x);
    receive foo(y); }
```

Asynchronous Message Passing and Semaphores

A channel acts as a semaphore, where sending and receiving have the same asymmetry as **V** (increase the value of the semaphore by one) and **P** (wait until value of the semaphore is greater than zero, and then decrease the value by one).

Comparison with general semaphores

channel	\approx	semaphore
send	\approx	V
receive	\approx	P

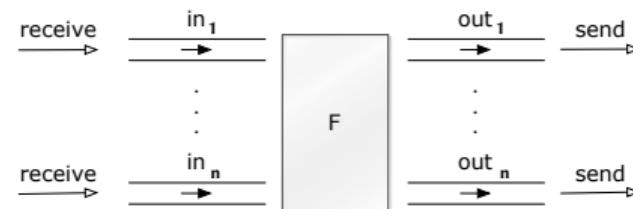
The value of the message plays no role for the semaphore-interpretation.

Filters: One-Way Interaction

Filters **F**

A filter **F** is a process which:

- Receives messages on input channels,
- Sends messages on output channels, such that
- the output is a function of the input (and the initial state).



- Some computations are naturally seen as a composition of filters:
- stream processing, feedback loops and *dataflow programming*

Example: A Single Filter Process

Task: Sort a list of n numbers into ascending order.

Filter

Process **Sort** with input channel `input` and output channel `output`.

Example implementation: get n over `input`, then read n times from `input` and send the sorted list at once over `output`.

Sort predicate

- n : number of values sent to output.

$sent[i]$: i 'th value sent to output, $received[j]$: j 'th value received in input,

$$\forall i : 1 \leq i < n. (sent[i] \leq sent[i + 1]) \wedge$$

$$\forall i : 1 \leq i \leq n. \exists j : 1 \leq j \leq n. sent[i] = received[j] \wedge$$

$$\forall i : 1 \leq i \leq n. \exists j : 1 \leq j \leq n. received[i] = sent[j]$$

Filter for Merging of Streams

Task: Merge two sorted input streams into one sorted stream.

Process Merge with input channels in_1 and in_2 and output channel out :

$$\text{in}_1 : \langle 1 \ 4 \ 9 \dots \rangle \quad \text{in}_2 : \langle 2 \ 5 \ 8 \dots \rangle \quad \text{out} : \langle 1 \ 2 \ 4 \ 5 \ 8 \ 9 \dots \rangle$$

Special value **EOS** marks the end of an input, but result should be output online.

Merge predicate

n : number of values sent to out so far, $\text{sent}[n]$: i 'th value sent to out so far.

The following shall hold when **Merge** terminates:

$$\text{empty}(\text{in}_1) \wedge \text{empty}(\text{in}_2) \wedge \text{sent}[n + 1] = \text{EOS}$$

$$\wedge \quad \forall i : 1 \leq i < n \cdot \text{sent}[i] \leq \text{sent}[i + 1]$$

\wedge values sent to out are an *interleave* of values from in_1 and in_2

Await

```
chan in1(int), in2(int), out(int);

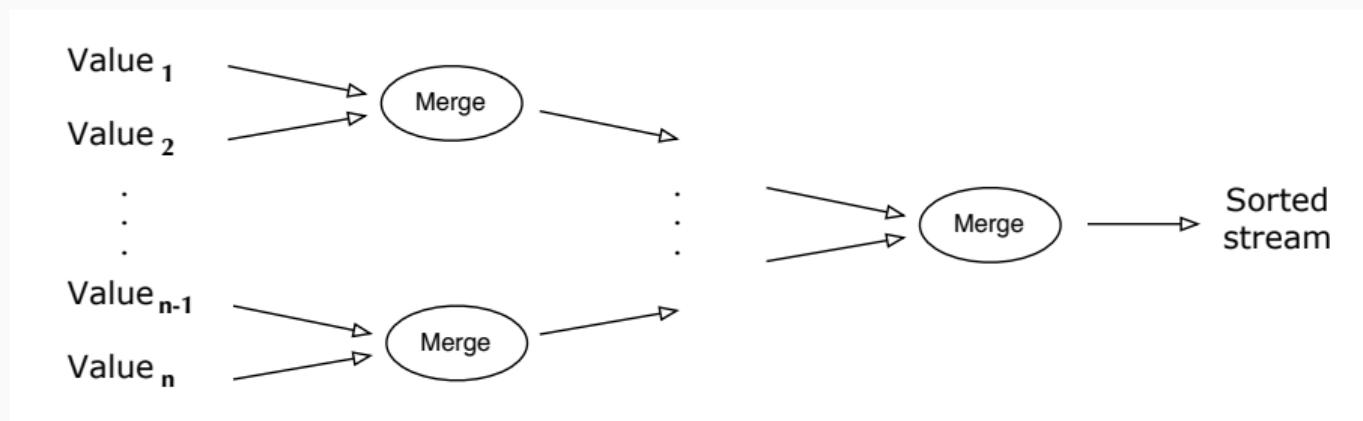
process Merge {
    int v1, v2;
    receive in1(v1);           # read the first two
    receive in2(v2);           # input values

    while (v1 != EOS and v2 != EOS) {
        if (v1 <= v2) { send out(v1); receive in1(v1); }
        else           { send out(v2); receive in2(v2); }
    }

    while (v1 != EOS) { send out(v1); receive in1(v1); }
    while (v2 != EOS) { send out(v2); receive in2(v2); }
    send out(EOS);
}
```

Sorting Network

To scale, we can now build a network that sorts n numbers, using a **collection** of Merge processes with tables of shared input and output channels.



Call-Backs to a Channel

- How to communicate a result back via channels?
- For example: Assume a process that adds two numbers it receives via a channel and then returns the result to the same channel.

Bi-directional channel

Await _____

```
chan c(int);  
process P { int a, b; receive c(a); receive c(b); send c(a+b); }
```

Requires same channel type for input and result.

Call-Backs to a Channel

- How to communicate a result back via channels?
- For example: Assume a process that adds two numbers it receives via a channel and then returns the result to a channel.

Answer channel per sender

Await _____

```
chan c(int), chan d[n](int);
process P { int a, b; int id;
    receive c(a); receive c(b); receive c(id); send d[id](a+b); }
```

Requires pre-sharing of channels, rather static.

Call-Backs to a Channel

- How to communicate a result back via channels?
- For example: Assume a process that adds two numbers it receives via a channel and then returns the result to the a channel.

Call-back channel

Await

```
chan c(...);
process P {
    int a, b;
    chan res(int);
    receive c(a); receive c(b); receive c(res);
    send res(a+b);
}
```

Requires (a) sending channels over channels and (b) more complex type for c.

Message Passing

Client-Server Applications using Messages

Roles

- Server process: repeatedly handling requests from clients
- Client processes: send requests to server, retrieve results later

Await

```
chan request(int, T1); # client ID, arguments of the operation  
chan reply[n](T2); # result of the operation
```

Await

```
process Client[i = 1 to n]{  
    ...  
    send request(i, args);  
    receive reply[i](var);  
    ...  
}
```

Await

```
process Server{  
    while(true){ int id; ...  
        receive request(id, args);  
        ... # code of the operation  
        send reply[id](result);  
    } }
```

Monitor Implementation using Message Passing

Monitors are very useful in a shared-memory setting, can we implement it in a channel-based concurrency model?

Classical monitor

- Controlled access to shared resource
- Global variables safeguard the resource state
- Access to a resource via procedures
- Procedures are executed under mutual exclusion
- Condition variables for synchronization

Active Monitors

- One server process that actively runs a loop listens on a channel for requests
- Procedure calls correspond to values send over request channel
- Resource and variables are local to the server process

Allocator for Multi-Unit Resources

Task

Multi-unit resource: a resource consisting of multiple units, which can be allocated separately, e.g., memory blocks, file blocks, etc.

- Client can request resources, use them, and return/free them
 - All the access to resources is managed for safety by the allocator
 - Unit usage itself is not managed
-
- Safety and efficient allocation is hard
 - Several simplifications here, e.g., only one unit of resource requested at a time
 - No focus on efficiency, resource is modeled as a set

Next slides: two versions

1. Allocator as (passive) monitor
2. Allocator as active monitor

Recap: Semaphore Monitor Passing the Condition

Await

```
monitor Semaphore { # monitor invariant: s >= 0
    int s := 0;           # value of the semaphore
    cond pos;            # wait condition

    procedure Psem() {
        if (s=0) wait(pos);
        else      s := s - 1; }

    procedure Vsem() {
        if (empty(pos)) s := s + 1;
        else              signal(pos); }

}
```

Allocator as a (Passive) Monitor

Await

```
monitor Resource_Allocator {
    int avail := MAXUNITS;
    set units;
    cond free;           // signalled when process wants a unit

    procedure acquire(int &id) {
        if (avail = 0) wait(free);
        else           avail := avail -1;
        remove(units, id); } // exact management abstracted here

    procedure release(int id) {
        insert(units, id);
        if (empty(free)) avail := avail +1;
        else             signal(free); }

}
```

Allocator as a Server Process: Code-Design Process for Monitors

1. Interface and internal variables
 - 1.1 Two types of operations: get unit, free unit
 - 1.2 One request channel *encoded* in the arguments to a request.
2. Control structure
 - 2.1 First check the kind of requested operation,
 - 2.2 Then, perform resource management for that operation
3. Synchronization, scheduling, and mutex
 - 3.1 Cannot wait (ie. `wait(free)`) when no unit is free.
 - 3.2 Must save the request and return to it later
 - ⇒ queue of pending requests (**queue; insert, remove**).
 - 3.3 Upon request: synchronous/blocking call ⇒ “ack”-message back
 - 3.4 No internal parallelism due to mutex

Channel Declarations

Await

```
type op_kind = enum(ACQUIRE, RELEASE);
chan request(int clientID, op_kind kind, int unitID);
chan reply[n](int unitID);

process Client[i = 0 to n-1] {
    int unitID;
    send request(i, ACQUIRE, 0); // make request
    receive reply[i](unitID);           // works as "if synchronous"
    ...                                // use resource unitID
    send request(i, RELEASE, unitID); // free resource
    ...
}
```

Note the problems with type-uniform channels: ACQUIRE request does not use its last parameter, RELEASE does not use the first one.

Await

```
process Resource_Allocator {
    int avail := MAXUNITS;
    set units := ...;           // initial value
    queue pending;              // initially empty
    int clientID, unitID; op_kind kind; ...
    while (true) {
        receive request(clientID, kind, unitID);
        if (kind = ACQUIRE) {
            if (avail = 0) insert(pending, clientID); // save request
            else { // perform request now
                avail:= avail-1;
                remove(units, unitID);
                send reply[clientID](unitID); } }
        else { // kind = RELEASE
            if empty(pending) avail := avail+1; insert(units, unitID);
            else { // allocates to waiting client
                remove(pending, clientID);
                send reply[clientID](unitID); } } } }
```

Duality: Mainonitors & Message Passing

monitor-based programs	message-based programs
monitor variables	local server variables
process-IDs	request channel, operation types
procedure call	send request(), receive reply[i]()
go into a monitor	receive request()
procedure return	send reply[i]()
wait statement	save pending requests in a queue
signal statement	get and process pending request (reply)
procedure body	branches in branching over op. type

Synchronous Message Passing

Synchronous Channels

- Asynchronous channels pass messages, but do not synchronize two processes
- Next: Synchronous channels
- Natural connection to barriers

Primitives

```
synch_send c(expr1, ..., exprN);
```

- Sender waits until message is received via the channel,
- Sender and receiver synchronize by the sending and receiving of message
- Same receiving primitive

Synchronous Message Passing: Discussion

Advantages

- Gives maximum *size* of channel (for fixed number of processes), as sender synchronizes with receiver
 - Receiver has at most 1 pending message per channel per sender
 - Each sender has at most 1 unsent message

Disadvantages

- Reduced parallelism: when 2 processes communicate, 1 is always blocked
- Higher risk of *deadlock*

Example: Blocking with Synchronous Message Passing

Await

```
chan values(int);  
  
process Producer {  
    int data[n];  
    for (i = 0 to n-1) {  
        ... //computation  
        synch_send values(data[i]); }  
}  
  
process Consumer {  
    int results[n];  
    for (i = 0 to n-1) {  
        receive values(results[i]);  
        ... //computation  
    } }
```

- Assume both producer and consumer vary in time complexity.
- Communication using `synch_send/receive` will **block**.
- With *asynchronous* message passing, the waiting is reduced.

Example: Deadlock using Synchronous Message Passing

Await

```
chan in1(int), in2(int);

process P1 {
    int v1 = 1, v2;
    synch_send in2(v1);
    receive in1(v2);}

process P2 {
    int v1, v2 = 2;
    synch_send in1(v2);
    receive in2(v1);}
```

- P1 and P2 both block on `synch_send` – program *deadlocks*
- One process must be modified to do `receive` first ⇒ asymmetric solution.
- With asynchronous channels, all goes well

Encoding

- Despite all, many implementations (e.g., Go) and theories (e.g., π -calculus have *synchronous channels*)
- Main reason: It is easier to encode asynchronous message passing with synchronous channels than vice versa
- Requires way to spawn new thread/process

Await

```
chan v(int);  
  
process Send{  
    spawn { synch_send v(1); } //spawns new thread and continues  
}  
process Receive {  
    int res;  
    receive v(res);  
}
```

Summary

Today's lecture

- Shared memory vs. distributed memory
- Synchronous and asynchronous message passing, the high level picture
- *Asynchronous message passing*: channels, messages, primitives
- Example: filters and sorting networks
- Comparison of message passing and monitors
- Basics *synchronous message passing*

Next lectures in this module

- Concurrency in Go
- Actors with asynchronous communication / Await primitive

Concurrency in Go

Andrea Pferscher

October 01, 2025

University of Oslo

Repetition & Outlook

- Distributed systems and synchronous channels
- Asynchronous channels

Concurrent Programming Languages

- Concurrency model part of the language
- Provides abstraction and first-class primitives (in addition to libraries)
- How to fit concurrency nicely into language design?

Go Basics

Background

Growing dissatisfaction with C and C++ as system programming languages in 2000s, when multi-core programs became more important

Common criticisms (back then)

- Concurrency hard to do, even harder to get right — no built-in language support
- Type system overly complex
- Long compilation times, complex build systems
- Memory safety

Emergent Solutions

- Solution 1: Make C++ better (e.g., coroutines in C++20, compositional futures in C++23)
- Solution 2: A new language with simplicity and asynchronous communication first: **Go**
- Solution 3: A new language with type and memory safety first: **Rust**

History of Go

- First plans around 2007 at Google, due to above dissatisfaction
- Public announcement 2009, first release 2012, widely adapted by now
- Inspired by:
 - C (systems programming language)
 - Communicating Sequential Processes (research model: process calculi for message passing via channels)
 - Newspeak (research language)
 - Erlang (concurrent, functional, systems programming language)
 - Concurrent ML (systems programming language)
 - Python (scripting language)
- Very much a consolidation language along the idea of “less is more”

Go's Non-revolutionary Feature Mix

- Imperative
- Compiled, no VM
- Garbage collected
- Concurrency with light-weight processes (goroutines) and channels
- Strongly typed
- Portable
- Higher-order functions and closures
- No orthodox OO, but common patterns for emulation

Agenda

1. Objects in Go
2. Types in Go
3. Concurrency in Go

Go code on the slides is sometime abbreviated to fit the format

Go Object Model

Go's heterodox take on OO

- No classes, but there are only structs
- No class inheritance, also no inheritance on records
- Interfaces as types

Code reuse

Code reuse encouraged by

- Embeddings

A First Glimpse at Go

Go

```
type Pair struct { X, Y float64 }

func main() {
    var pair1 Pair
    pair1 = Pair{ 3,4 }
    pair2 := Pair{ 1,2 } //no type needed if initialized
    var res float64 = pair1.Abs() + pair2.Abs()
    fmt.Println(res)
}

func (x *Pair) Abs() float64 { ... }
```

What is a Type?

Views on types

- Compiler & run-time system
 - A hint for the compiler of memory usage & representation layout
 - Piece of meta-data about a chunk of memory
- Programmer
 - Partial specification for safety
 - Whatever I must do to make the compiler happy
- Orthodox OO
 - A type is essentially a class (at least the interesting ones/custom types)

Milner's dictum on static type systems

"Well-typed programs cannot go wrong"

For some notion of going wrong.

How to Implement an Interface with an Object?

- Interfaces contain *methods* (but no fields)
- Records contain *fields* (but no methods)

What is an object?

data + control + identity

And how to get one, implementing an interface?

Java ...

1. Interface: given
2. name a class which implements I
3. fill in data (fields)
4. fill in code (methods)
5. instantiate the class

Go

1. Interface: given
2. —
3. choose data (state)
4. bind methods
5. get yourself a data value

Interfaces

Go

```
type AbsI interface { Abs() float64 }
type Triple struct { X, Y, Z float64 }
func (x *Triple) Abs() float64 { ... }
func main() {
    var a AbsI //must contain something that implements AbsI
    pair := Pair{1,2}
    triple := Triple{3,4,5}

    a = &pair // a *Pair is ok
    a = &triple // a *Triple is ok
    a = pair // a Pair is not ok, except pair := &Pair{1,2}
}
```

Duck Typing

Duck Typing

"If it walks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."

- If interface and record are detached, a method can be called if the record fits its signature
- Dynamic duck typing: check at runtime whether the record fits
- Static duck typing: check at compile time whether the type of the value/variable fits

Beware: Go does *static* duck typing: smaller runtime, no need for type tagging

Code Reuse with Embeddings

Go

```
type Pair struct { X, Y float64 }
type Triple struct {
    Pair // no variable name -> implicit field
    Z float64
}

func main(){
    triple := Triple{ Pair { 1,2 } , 3}
    fmt.Println(triple.X)
}
```

Go Concurrency

Go Concurrency

Go's concurrency mantra

"Don't communicate by sharing memory, share memory by communicating!"

- Go does have shared memory via global variables, heap memory etc.
- But you are supposed to only send references – getting a reference transfers ownership, i.e., the permission to write/read it

Go's primitives

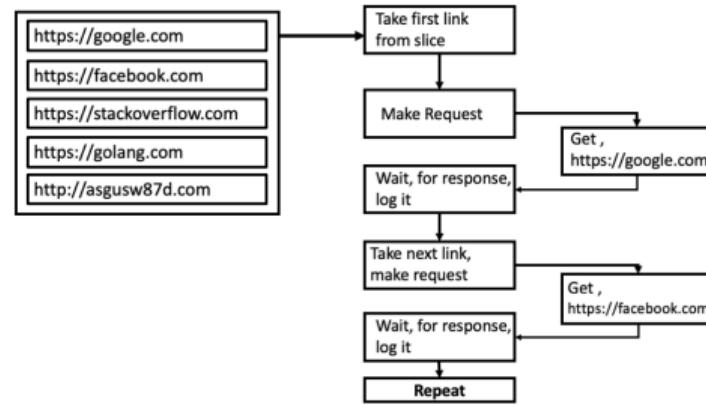
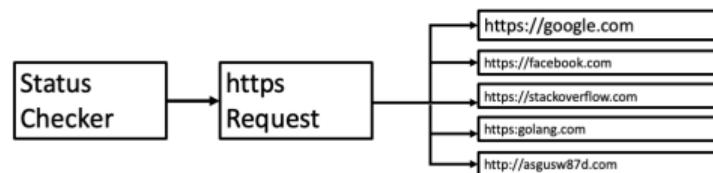
- Goroutines: lightweight threads
 - Own call stack, small stack memory (2KB initially), handled by go runtime
 - Very cheap context switch
- Channels
 - Synchronous and typed
 - Communication between (lightweight) threads
 - Main means of synchronization

Goroutines

3 ways to call a function

- `f(x)` – ordinary (synchronous) function call, where `f` is a defined function or a functional definition
- `go f(x)` – called as an asynchronous process, i.e., goroutine
- `defer f(x)` – the call is delayed until the surrounding function returns

Example 1: Status Checker



Channels in Go

- Channels provide a way to send messages from one go routine to another.
- Channels are created with **make**
- The arrow operator ($<-$) is used both to signify the direction of a channel and to *send* or *receive* data over a channel

Go

```
func main(){
    chl := make(chan float64)
    go sendf(chl); go receivef(chl)
}
func sendf(ch chan<- float64) {
    ch <- 0.5 }
func receivef(ch <-chan float64){
    v := <-ch }
```

Go Routines - Example 1

Back to our server status checker

Go

```
func main() {
    links := []string{
        "https://google.com",
        "https://facebook.com",
        "https://golang.org",
        "https://stackoverflow.com",
    }
    c := make(chan string)
    for _, link := range links { go checklink(link, c) }
    for i := 0; i < len(links); i++ { fmt.Println(<-c) }
}
```

Goroutines - Example 1

Go

```
func checklink(link string, c chan string) {
    _, err := http.Get(link) //pairs as language builtins
    if err != nil {
        c <- link + " is down!"
        return}
    c <- link + " is up!"}
```

- The four checklink goroutines starts up concurrently and four calls to `http.Get` are made concurrently as well.
- The main process does not wait until one response comes back before sending out the next request.
- Go scheduler picks up other goroutine in case a goroutine makes a blocking call (e.g. file-based system calls)

Waiting for Goroutines to Finish

Go offers several synchronization primitives in the sync package to avoid using channels in certain situations.

WaitGroup

A WaitGroup is a semaphore, used to join over several activities

- The Add method is used to add a counter to the WaitGroup.
- The Done method of WaitGroup is scheduled using a defer statement to decrement the WaitGroup counter.
- The Wait method of the WaitGroup type waits for the program to finish all goroutines: The Wait method is called inside the main function, which blocks execution until the WaitGroup counter reaches the value of zero and ensures that all goroutines are executed.

Waiting for Goroutines to Finish (Example: Part I/II)

Go

```
func main() {
    var wg sync.WaitGroup
    var i int = -1
    var file string
    for i, file = range os.Args[1:] {
        wg.Add(1) // add before asynchronous call!
        go func(file string) { // anonymous function
            compress(file)
            wg.Done()
        }(file)
    }
    wg.Wait()
    fmt.Printf("compressed %d files \n", i+1)
}
```

Waiting for goroutines to Finish (Example: Part II/II)

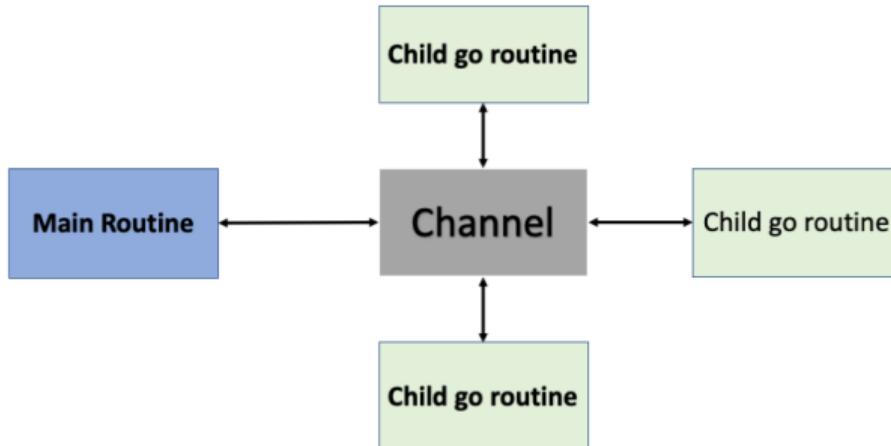
Go

```
func compress(filename string) error { //errors as builtin type
    in, err := os.Open(filename)
    if err != nil {
        return err}
    defer in.Close()
    out, err := os.Create(filename + ".gz")
    if err != nil {
        return err}
    defer out.Close()
    gzipout := gzip.NewWriter(out)
    ... // read file and write compressed content
}
```

Channels

Channels in Go

- Channels are bidirectional, synchronous and typed
- Careful which routine is receiving and which is sending
- Type support to enforce that



Channel Operations: Buffer (I/II)

- Send and receive
- Create channels (with buffer)

Go

```
func main() {  
    ch := make(chan int, 2)  
    ch <- 1 //does not block!  
    ch <- 2  
    fmt.Println(<-ch)  
    fmt.Println(<-ch)  
}
```

Channel Operations: Buffer (II/II)

- Send and receive
- Create channels (with buffer)

Go

```
func main() {  
    ch := make(chan int, 2)  
    ch <- 1  
    ch <- 2  
    ch <- 3 //deadlock  
    fmt.Println(<-ch)  
    fmt.Println(<-ch)  
}
```

Channel Operations: Close (I/III)

- Send and receive
- Create channels (with buffer)
- Close a channel

Go

```
func main() {
    ch := make(chan int)
    go send(ch)
    for {
        i, ok := <-ch
        if !ok {break}
        fmt.Println(i) } }

func send(ch chan<- int) {
    ch <- 1; ch <- 2; close(ch) }
```

Channel Operations: Close (II/III)

- Send and receive
- Create channels (with buffer)
- Close a channel

Go

```
func main() {
    ch := make(chan int)
    go send(ch)
    for i := range ch {
        fmt.Println(i)
    }
}

func send(ch chan<- int) {
    ch <- 1; ch <- 2; close(ch) }
```

Channel Operations: Close (III/III)

- Send and receive
- Create channels (with buffer)
- Close a channel

Go

```
func main() {
    ch := make(chan int)
    go send(ch)
    fmt.Println(<-ch)
    fmt.Println(<-ch) } // ?

func send(ch chan<- int) {
    ch <- 1; close(ch) }
```

Channel operations: Selection (I/II)

Selection

- Waiting on several channels in parallel
- The select statement can watch multiple channels (zero or more). Until something happens, it will wait (or execute a default statement, if supplied).
- When a channel has an event, the select statement will execute that event.

Go

```
ch1 := make(chan int); ch2 := make(chan int)
go send(ch1); go send(ch2)
select {
    case i1 = <-ch1: fmt.Printf("first call %d \n",i1)
    case i2 = <-ch2: fmt.Printf("second call %d \n",i2) }
```

Channel operations: Selection (II/II)

Selection

- Waiting on several channels in parallel
- The select statement can watch multiple channels (zero or more). Until something happens, it will wait (or execute a default statement, if supplied).
- When a channel has an event, the select statement will execute that event.

Go

```
ch1 := make(chan int); ch2 := make(chan int)
go send(ch1); go send(ch2)
select {
    case i1 = <-ch1: fmt.Printf("first call %d \n",i1)
    case i2 = <-ch2: fmt.Printf("second call %d \n",i2)
    default: fmt.Println("I don't block") }
```

Select Operation on Channels in Go

Go

```
func main() {
    done := time.After(30 * time.Second)
    echo := make(chan []byte)
    go readStdin(echo)
    for {
        select {
            case buf := <-echo:
                os.Stdout.Write(buf)
            case <-done:
                fmt.Println("Timed out")
                os.Exit(0) } } }

func readStdin(out chan<- []byte) {
    for {
        data := make([]byte, 1024)
        l, _ := os.Stdin.Read(data)
        if (l > 0) {out <- data} } }
```

Lock Implementation with Channels in Go

Go

```
func main() {
    lock := make(chan bool, 1)
    for i := 0; i < 7; i++ {
        go worker(i, lock)
    }
    time.Sleep(10 * time.Second)
}

func worker(id int, lock chan bool) {
    fmt.Printf("%d wants the lock \n", id)
    lock <- true
    fmt.Printf(" %d has the lock \n", id)
    time.Sleep(500 * time.Millisecond)
    fmt.Printf(" %d is releasing the lock \n", id)
    <-lock
}
```

Producer-Consumer Implementation in Go

Go

```
const producerCount int = 4
const consumerCount int = 3

func produce(link chan<- string, id int, wg *sync.WaitGroup) {
    defer wg.Done()
    for _, msg := range messages[id] {
        link <- msg
    }
}

func consume(link <-chan string, id int, wg *sync.WaitGroup) {
    defer wg.Done()
    for msg := range link {
        fmt.Printf("Message %v is consumed by consumer %v\n", msg, id)
    }
}
```

Producer-Consumer Implementation in Go

Go

```
func main() {
    link := make(chan string)
    wp := &sync.WaitGroup{}
    wc := &sync.WaitGroup{}

    wp.Add(producerCount)
    wc.Add(consumerCount)

    for i := 0; i < producerCount; i++ {
        go produce(link, i, wp)
    }

    for i := 0; i < consumerCount; i++ {
        go consume(link, i, wc)
    }

    wp.Wait()
    close(link)
    wc.Wait()
```

Dining Philosophers

Go

```
type Fork struct{ sync.Mutex }

type Philosopher struct {
    id int
    leftFork, rightFork *Fork
}
// Goes from thinking to hungry to eating, done eating then starts over.
func (p Philosopher) eat() {
    defer eatWgroup.Done()
    for j := 0; j < 3; j++ {
        p.leftFork.Lock()
        p.rightFork.Lock()
        p.say("eating")
        time.Sleep(time.Second)
        p.rightFork.Unlock()
        p.leftFork.Unlock()
        p.say("finished eating")
        time.Sleep(time.Second)}}
func (p Philosopher) say(action string) {
    fmt.Printf("Philosopher #%d is %v\n", p.id, action) }
```

Dining Philosophers

Go

```
func main() {
    count := 5

    // Create forks
    forks := make([]*Fork, count)
    for i := 0; i < count; i++ {
        forks[i] = new(Fork) }

    // Create philosopher, assign them 2 forks and send them to the dining table
    philosophers := make([]*Philosopher, count)
    for i := 0; i < count; i++ {
        philosophers[i] = &Philosopher{
            id: i, leftFork: forks[i], rightFork: forks[(i+1)%count]}
        eatWgroup.Add(1)
        go philosophers[i].eat()
    }
    eatWgroup.Wait()}
```

Wrap-Up

Today's lecture

- Object-orientation in Go: interface types and embeddings
- Goroutines: lightweight threads with builtin support
- Channels in mainstream programming: selection, creation, typing

Next block: actors, active objects and asynchronous communication

Actors, Active Objects and Asynchronous Communication

Andrea Pferscher

14.10.2024

University of Oslo

Part 2: Message Passing

Structure

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

Content of this part:

- Synchronous and asynchronous message passing
- Channels, actors, go-routines, asynchronous programming

Outline Today

- Actors
- Futures and promises
- Active objects
- Asynchronous communication with await-statement

Message Passing and Channels

- Shared memory vs. distributed memory
- Synchronous and asynchronous message passing, the high level picture
- *Asynchronous message passing*: channels, messages, primitives
- Example: filters and sorting networks
- Comparison of message passing and monitors
- Basics *synchronous message passing*

Actors

Async. Communication without Channels

Channels

- Need additional primitives for concurrency; send and receive
- Channels are explicit while process/objects are implicit
- Complex typing disciplines

Can we do asynchronous communication without explicit channels?

- Actors: Messages between objects
- Active Objects: Messages between objects with cooperative scheduling
- Async/Await in mainstream languages: Using (lightweight) threads (with shared memory)

Actors

- Actors: a programming concept for distributed concurrency which combines a number of topics we have discussed in the course;
 - active monitors,
 - objects and encapsulation,
 - race-free (no race conditions on shared state)
- Examples of programming languages that implement actors:
Erlang, Scala's Akka library, Dart, Swift, etc.

Object-Oriented Programming and Language Design

What are objects

How do OO programs fit into the design of programming languages?

- **State space:** local or global?
- **Thread interaction and objects**
- **Communication:** shared variables, channels or messages?
- **Communication:** synchronous or asynchronous?
- **Dynamic state allocation:** object creation

What can we do to protect objects against races?

Can we combine objects with ideas from monitors?

- Passive monitors vs. active monitors
- A method is *active*, if a statement in the method is executed by some thread

Passive Monitors – Repetition

Await

```
monitor name {  
    monitor variables  
    ## monitor invariant  
    initialization code  
    procedures  
}
```

- Threads *communicate* by calling monitor methods
- Threads do not need to know all the implementation details: only the procedure names are visible from outside the monitor
- Statements *inside* a monitor: *no* access to variables *outside* the monitor
- Statements *outside* a monitor: *no* access to variables *inside* the monitor
- **Monitor variables:** *initialized* before the monitor is used
- **Monitor invariant:** describes a condition on the inner state
- The monitor invariant can be analyzed by sequential reasoning inside the monitor

Passive Monitors: Synchronization with condition variables – Repetition

- Monitors contain *special* type of variables: **cond** (condition)
- Used for synchronization/to delay processes
- Each such variable is associated with a *wait condition*
- The value of a condition variable: *queue* of delayed threads
- Not directly accessible by programmer, instead, manipulated by special operations

```
cond cv;          # declares a condition variable cv
empty(cv);       # asks if the queue on cv is empty
wait(cv);        # causes thread to wait in the cv queue
signal(cv);      # wakes up a thread in the queue to cv
signal_all(cv); # wakes up all threads in the cv queue
```

Passive Monitors – Repetition

Await

```
monitor Mon { // monitor invariant: r ≥ 0
    int r := 0 // number of resources
    cond res; // wait condition variable

    procedure Acquire() {
        while (r=0) { wait (res); }
        r := r - 1 }

    procedure Release() {
        r := r+1;
        signal (res); }}
```

- wait and signal: *FIFO signaling strategy*
- A thread in the monitor can execute signal(cv).
If there is a waiting thread, do we get *two active methods* in the monitor?

Objects as Passive Monitors in Java

Java

```
class Mon { // class invariant: this.r >= 0
    int r = 0; // number of resources
    Condition res; // wait condition
    public synchronized void Acquire() {
        while( r == 0 ) { res.await(); };
        r = r - 1;
    }
    public synchronized void Release() {
        r = r + 1;
        res.signal();
    }
}
```

- How do condition variables and synchronized methods relate?

Actors

Fundamental idea: Decouple communication and control.

Capabilities of Actors

An actor reacts to incoming messages to

- change its state,
- send a finite number of messages to other actors, and
- create a finite number of new actors.

Intuition

We can think of an actor as an object that can only communicate asynchronously.

Some actor models can also pattern match over its message queue of incoming messages.

Implementation of Actors in Programming Languages

- Supported by numerous languages and frameworks
 - Not always strictly OO: Erlang, ...
 - Sometimes as library, not part of language: Akka actors, ...
 - Numerous differences on how basic capabilities are implemented or extended
-
- Type safety: Can we guarantee statically whether messages can be processed?
 - Integration with OO: Are messages methods? Do actors have a class?
 - Integration with other primitives: Can actors share state?
 - Integration with error handling: What happens when an actor fails?
 - Here: foundations

Actors: Communication & Concurrency

Actors

- Recipients of messages are identified by name (no channels).
- An actor can only communicate with actors that it knows.
- An actor can obtain names from messages that it receives, or because it has created the actor

The actor model is characterized by

- inherent concurrency among actors
- dynamic creation of actors,
- inclusion of actor names in messages, and
- interaction only through direct asynchronous message passing with no restriction on message arrival order.
- *message servers* might be implemented by matching messages from the queue to procedures

Example: Erlang-style Actors - Matching Messages

Publish and Subscribe Server

```
runServer(Subs) ->
    receive
        {sub,from} -> runServer(Subs + from); % subscribe
        {publish,value} -> % publish
            for(id in Subs) id!{value}, % broadcast value
            runServer(Subs);
        _ -> runServer(Subs); % ignore other messages
    end.
```

```
Server { % publish and subscribe server
    start() -> spawn(fun() -> runServer([])).} % start the server
```

```
Client { % send requests to the server
    start() -> Server!{sub,self}, Server!{publish,10}.}
```

Example: Erlang-style Actors

- State as argument to recursive calls
- We can dynamically change the message server
- An actor can match different messages in different states
- ... but tricky to detect errors in message servers

```
runServer1(Subs) -> receive % subscribe when there is space
    {sub,from} -> if(size(Subs) >= 9) runServer2(Subs + from)
                    else runServer1(Subs + from);
    {unsub,from} -> runServer1(Subs - from);
    ...

```

Example: Erlang-style Actors – Handling Return Values between Actors

```
id1 = spawn(fun() -> func1([])); id2 = spawn(fun() -> func2([]))
id1!{step1, 42, id2};

...
func1(history) -> receive
    {step1, data, other} -> newData = doSomethingFirst(data),
                                other!{step2, newData, self},
                                func1(insert(history,data));
    {step3, data, other} -> newData = doSomethingThird(data),
                                other!{step4, newData, self},
                                func1(insert(history,data));

func2(history) -> receive
    {step2, data, other} -> newData = doSomethingSecond(data),
                                other!{step3, newData, self},
                                func2(insert(history,data)); ...
```

Futures

Futures – Handling Return Values between Actors

Welcome to “callback hell”!

- Problem: Logically related code is scattered in program
- We need a way to identify callback messages
- We also need a way to wait for a result
- Solution: futures, special mailboxes transmit return values

Reminder in Java:

Java

```
ExecutorService service = Executors.newFixedThreadPool(2);
Future<Int> f = service.submit(() -> { /* do */ return 1;});
...
Int x = f.get(); //essentially a join
```

Futures and Promises

Futures

- It is a handle for the caller of a process. It will contain the result value once computed
- It can be read multiple times
- It can be used by the caller to synchronize with the callee

Java

```
Future<Int> f = service.submit(() -> { return 1;});  
...  
Int x = f.get();
```

Promises

- What if the value will be computed somewhere else?
- A *promise* is a future for which it is not clear who computes the value

Promises

A promise:

- May be eventually completed (but maybe by somebody else)
- Must be completed (written) only once
- Deadlock/starvation occurs if it is never completed
- It can be seen as a handle for the callee and the callee does not synchronise with the caller

Java calls promises *CompletableFuture*:

Java

```
CompletableFuture<Integer> f = new CompletableFuture<>();
service.submit(() -> { f.complete(1); return null;});
...
Int x = f.get();
```

Promises – Example: Service Delegation

Java

```
/* the function casts a promise as a future */
/* from outside the future can only be retrieved */
Future<Integer> callAsync() ... {
    CompletableFuture<Integer> completableFuture = new CompletableFuture<>();
    service1.submit(() -> {
        if(/* service1 cannot process, then it delegates to service2 */)
            service2.submit(() ->
                { /* compute */ completableFuture.complete(1); return null })
        else { /* process the request */
            /* compute */ completableFuture.complete(1); }
        return null;
    });
    return completableFuture;
}
```

Composition Futures/Promises

Logically related Futures/Promises scattered in the code.

Java

```
CompletableFuture<Integer> f1 = CompletableFuture.supplyAsync(() -> 1);
...
CompletableFuture<Integer> f2
    = CompletableFuture.supplyAsync(() -> f1.get + 1);
```

Connecting Futures/Promises (composition)

Java

```
CompletableFuture<Integer> f
    = CompletableFuture.supplyAsync(() -> 1)
        .thenApply((res) -> res + 1);
```

Very similar patterns are common in web development with JavaScript

Interpreting Futures/Promises as Channels

Channel-view on single-read futures

- Create channel and send it via an asynchronous message
- For the caller, the channel behaves as a future:
caller waits on the channel for a return (caller side does not write on the channel).
- For the callee, the channel behaves as a promise:
it can be passed around, and eventually someone will write on it *exactly once*
(callee side does not read on the channel)

Limits of this view

- Futures may be read more than once
- “immediately creating and sharing a channel” may be more complex and its implementation is delegated to the programmer

Active Objects

Motivation

- How to combine monitors and actors?
- How to make signalling less error-prone?
- How to make conditions/invariants easier to use?
- How to connect futures/promises with actors?

Active Objects

An active object^a is an actor with an *implicit* message server, that only communicates asynchronously, but allows internal message handlers to use *cooperative scheduling*.

- One process/thread per object
- Messages identified with methods
- Implicit queue of tasks (procedures in the methods)
- Explicit synchronization

^aABS is a modelling language to run simulations of distributed systems.

The simulation tool is maintained by the PSY group at IFI: <https://abs-models.org/>

Cooperative concurrency

Active Monitors as Active Objects

- Cooperative concurrency:
constructs to suspend and resume execution (=task) of a local method
- External cooperation (operations on futures)
 - Send is **asynchronous**: `Fut < T > f = o!m(...); ... ;`
 - Retrieve value is **blocking**: `x = f.get;`
 - Check for value is **suspending**: `await f?`
- Interaction patterns between methods
 - `Fut < T > f = o!m(...);x = f.get;`
 - `Fut < T > f = o!m(...); ...; x = f.get;`
 - `Fut < T > f = o!m(...); ...; await f?; x = f.get;`

Cooperative Scheduling – Example: The Diner

- Each object runs one thread and each method call spawns a *task*
- Thread is responsible to schedule tasks in some order
- Waiting on future suspends the task, not the thread!
- Reading on future potentially blocks task and thread – no other task can run

ABS

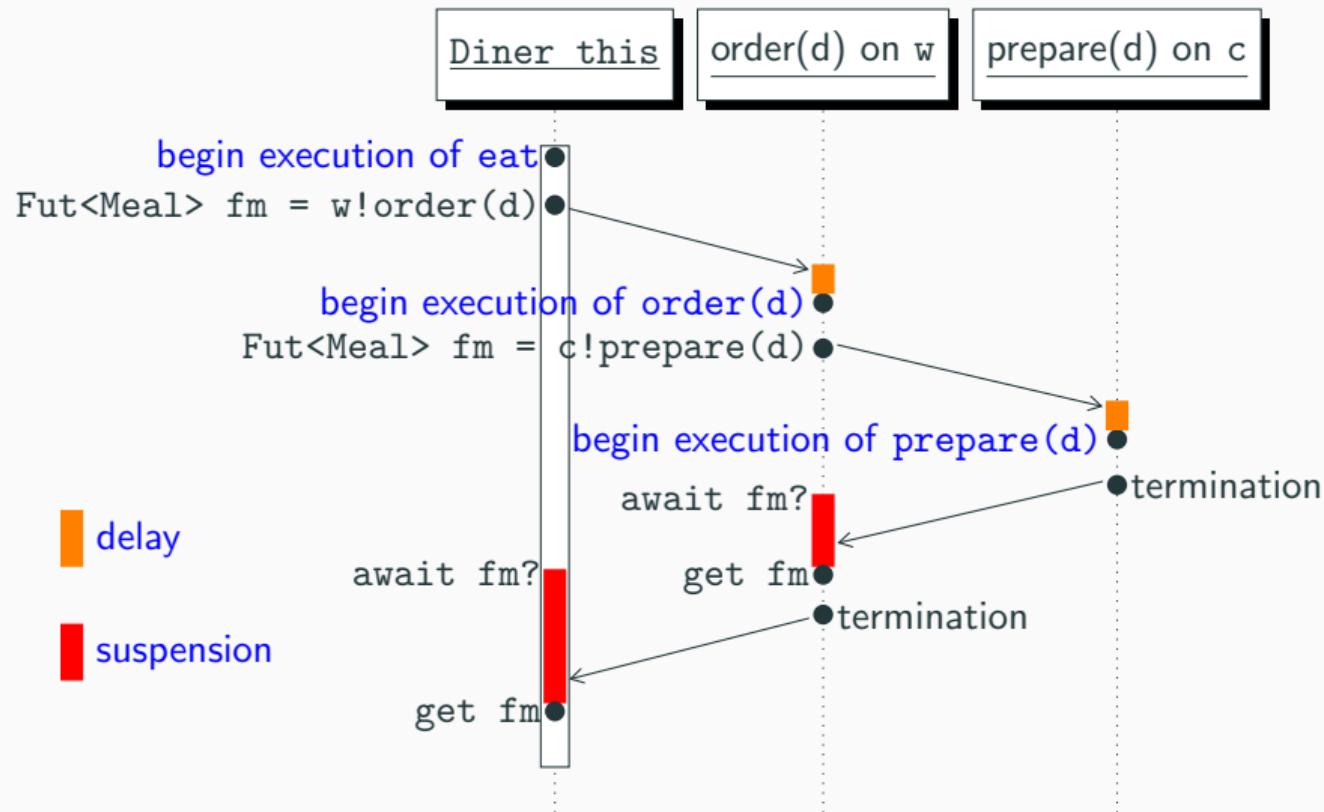
```
class Diner(IWaiter w) implements IDiner {  
    Unit eat(Dish d) {  
        Fut<Meal> fm = w!order(d); // place order with waiter  
        await fm?; // while waiting do something else, e.g., take a phone call  
        Meal m = fm.get; // receive meal  
        Fut<Unit> fc = this!consume(m);  
        Fut<Unit> fp = w!pay(this, d); // eating, paying in some order  
        await fc? & fp?; // eaten and paid – ready to leave!  
    }  
    Unit takeCall(){ ... }  
    Int takeMoney(Int a) { ... }  
    ...  
}
```

Example (Continuation): – The Waiter

ABS

```
class Waiter(ICook c, Int purse) implements IWaiter {
    Meal order(Dish d) {
        Fut<Meal> fm = c!prepare(d); // place order with cook
        await fm?; // waiter serves other guests while meal is cooked
        Meal m = fm.get(); // receive meal reuse names for local variables
        return m; // ready to serve the meal!
    }
    Unit pay(IDiner g, Dish d) {
        Int amount = price(d); // lookup price in the menu
        Int a = g.takeMoney(amount); // synchronous (blocking) call, no wait
        this.purse = this.purse + amount; // no data race possible
    }
}
```

Example (Continuation) – The Restaurant Experience



Condition Synchronization

- Condition variables can be derived from monitor invariant
- Or can be bound to some other condition
- Error-prone implementations
- Active Object approach: condition synchronization as primitive

ABS

```
class C () {  
    int i = 0;  
    Unit inc() { i = i+1; return; }  
    Int isGreaterThanTen(){ await i > 10; return i; }  
}
```

- Condition variables: explicit suspension instead of busy waiting
- Every time the object is idle, the object thread evaluates all conditions of suspended tasks, otherwise it waits for new messages to arrive

Objects as Passive Monitors (reminder) – Example

Java

```
class Mon {  
    int r = 0;  
    Condition res;  
  
    public synchronized void Acquire() {  
        while( r == 0 ) { res.await(); }  
        r = r - 1;  
    }  
  
    public synchronized void Release() {  
        r = r + 1;  
        res.signal();  
    }  
}
```

Monitors with active objects – Example

ABS

```
class Mon {  
    int r = 0  
  
    Unit Acquire() {  
        await (r!=0);  
        r = r - 1;  
    }  
  
    Unit Release() {  
        r = r+1;  
    }  
}
```

- With cooperative concurrency, we can avoid error-prone signaling in the monitor.
- The active object only has one queue, but reactivation of Acquire methods can only happen when the await-condition holds

Bounded Buffer Synchronization with Active Objects(1)

Let us now solve the bounded buffer problem with active objects

Bounded buffer synchronization

- buffer of size n (“channel”, “pipe”)
- producer: performs put operations on the buffer.
- consumer: performs getVal operations on the buffer.
- two access operations (“methods”)
 - put operations must wait if buffer full
 - getVal operations must wait if buffer empty

Bounded Buffer Synchronization with Active Objects (2)

ABS

```
class Bounded Buffer (Int n) {
    List<T> buf = [];
    Unit put(T data){
        await (length(buf) < n);
        buf = appendright(buf,data);
    }
    T getVal() {
        await (length(buf) > 0);
        T tmp = head(buf); buf = tail(buf); return tmp;
    }
}
```

What is a deadlock?

A system is deadlocked if it is *stuck*:
It cannot continue execution, and
it has not finished its execution.

A system is deadlocked if there is a circular dependency: There is a sequence of components C_1, \dots, C_n , such that C_i depends on C_{i+1} before it can continue and C_n depends on C_1 .

- Actors without futures/channels cannot deadlock – they can always continue execution
... but there can be messages that cannot be processed with the current message server!
- In some concurrency models, a system can only get stuck because of a circular dependency

Local Dependencies – Between the Object and its Tasks

ABS

```
class C (){

    Unit m(){
        Fut<T> f = this!n();
        f.get; // deadlock
    }

    T n(){ /* do some computation */ return value; }

}
```

Dependencies due to Synchronization Between Tasks

- A task depends on another task if it waits for its future

ABS

```
class C {  
    Fut<Unit> f1;  
    Unit store(Fut<Unit> fut) { f1 = fut; }  
    Unit m(){ await f1?; return; }} //depends on d.n  
  
class D(C c) {  
    Unit n(){ Fut<Unit> f2= c!m();  
              await f2?; //depends on c.m  
              return; } }  
{ // Main block  
    C c = new C(); D d = new D(c);  
    Fut<Unit> f;  
    await c!store(f); f= d!n(); // deadlock  
}
```

Dependencies Related to the State of an Object

- In a given state a task t_1 , that might be stuck on condition e_1 , depends on another task t_2 , that might be stuck on condition e_2 .
- Here e_1 and e_2 are conditions related to the state of an object, which create dependencies between the tasks.

ABS

```
class D { // here exclamation mark is negation
    Bool b1 = false; Bool b2 = false;
    Unit m(){ b1 = true; await b2; b1 = !b2;}
    Unit n(){ await !b1; b2 = !b1;}
}
```

- What happens if we call $n()$ and then $m()$ on a D -object?
- There is no procedure to decide whether an arbitrary program ever deadlocks because it depends on the scheduling of tasks

Outlook: Analysis and Modelling

Reasoning

Monitors and actors are well-suited for manual and automatic reasoning

- Built-in mutex ensures that between interaction points, code can be seen as sequential
- Sequential reasoning has to be extended only at these points
- Full concurrency requires non-local reasoning at every point

Programming is Modelling

A program can be used to model a part of the world.

- A program analysis then can be used to derive properties over the world
- For example, 5 philosophers programs are *executable* models
- Allows analysis for deadlock freedom.

Async/Await

Recap on Message Passing

Message Passing So Far

- Channels: Asynchronous shared entities
- Actors: Monitors that send asynchronous messages
- Active Objects: Monitors with their own thread that send asynchronous messages

Java and Async/Await

Reminder in Java:

Java

```
ExecutorService service = Executors.newFixedThreadPool(2);
Future<Int> f = service.submit(() -> { /* do */ return 1;});
...
Int x = f.get(); //essentially a join
```

- Executed function disconnected from classes
- Much boilerplate code, especially when call-backs are involved
- Asynchronous code (library) does not mirror synchronous code (language constructs)

C# and Async/Await

C#'s Asynchronous Concurrency

- Better abstraction to handle Futures/Tasks.
 - Concurrency as first-class construct of language
-
- Methods annotated with `async` can only be called asynchronously
 - Methods annotated with `async` return a `Task`
 - Only methods annotated with `async` can perform an `await`
 - Expression `await` suspends the thread until the task has finished.

C# and Async/Await – Example: Comp. Two Numbers

- Example: Reading two numbers from user and performing some long-lasting computation
- Synchronous version
- Await version: Note that Method must be `async` to use `await`
- Asynchronous version: Now both reads can be concurrent

C#

```
class C{
    void async Method() {
        Task<Int> t1 = GetFirstNumber(); Task<Int> t2 = GetSecondNumber();
        Int i1 = await t1; Int i2 = await t2;
        Int res = await Compute(i1,i2);
    }
    Task<Int> async GetFirstNumber() {...}
    ...
}
```

Pros and Cons of Async/Await

What Color is your Function

- Only async methods can access results of async methods
- Separates whole program into two sets of methods that can only interact at specific points
- Sometimes called colored-function problem, after a popular blog entry^a

^a<https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/>

- Forces programmer to think about concurrency
- Can still use threads and tasks directly to circumvent all this

Wrap-Up

Today's Lecture

- Actors – Monitors with message passing
- Futures/Promises – Handling asynchronous results
- Active Objects – Actors with cooperative concurrency and futures
- Async/Await – Language-integrated asynchronicity with threads and futures

Next Lectures

- Next Block: How to type channels?

Note: ABS example courtesy of Reiner Hähnle

Part 3: Type Systems and Concurrency

Andrea Pferscher

October 22, 2025

University of Oslo

Types: Foundations

Analyses

Next lectures

- Type systems
- Types for channels
- Ownership and Rust

Reading material

- *Types and Programming Languages*, Benjamin Pierce, 2000, MIT Press
- *Type Systems for Concurrent Programs*, Naoki Kobayashi, 2002, Springer LNCS
- *Uniqueness Typing Simplified*, de Vries et al., 2007, Springer LNCS

Why Types?

- Detecting errors
 - Compiler detects errors before execution (static)
 - Clearer error messages at runtime (dynamic)
 - Enforcing certain programming patterns
- Abstraction
 - Modularity by providing interfaces
 - Hides memory/implementation details
- Documentation/Specification
 - Expresses *intended* behavior
 - Communication with other developers
 - In contrast to comments/documents: enforced to be updated

Why type systems here?

- Insights into compiler construction
- Type systems are a formalization of how developers analyze: How to think about programs?

Foundations of Types

“Well-typed programs cannot go wrong”(Robin Milner, '78)

- What is a “type”?
- What means “well-typed”?
- What means “go wrong”?
- What kind of type systems exist?
- What does this mean especially for concurrent systems?

Foundations of Types: What is a Type?

"Well-typed programs cannot go wrong"

Types for Expressions

- Types classify expressions
- Expression e has a **type T** if e will (always) evaluate to a value of **type T**
 - $\{\dots, -1, 0, 1, \dots\}$ are values of type **int**
 - $22 + 2$ evaluates to 24 , which has type **int**

- Data types of variables are abstractions over memory layout
- What is the type of a function? The type of a channel?
- For us: A type is an abstraction over *data or behavior*
- Channel types are *behavioral types*

Foundations of Types: What is Well-typedness?

"Well-typed programs cannot go wrong"

Type systems

If we know our abstractions, we need to ensure that our program adheres to them.

A type system is a method to check whether a program adheres to its types.

- Dynamic vs. static
 - Static systems check *type annotations* at compile time
 - Dynamic systems check *type tags* at runtime
 - Gradual system check as much as possible statically, and refer the rest to a dynamic system
- Decidable vs. undecidable
 - Static systems should not take too much time, more precise types abstract less
 - Type system can become undecidable (e.g. Java Generics)
- Strong vs. weak typing
 - Strong type systems aim to cover as many possible error sources
 - Weak type systems give more freedom

Foundations of Types: What are Errors? (I/IV)

"Well-typed programs cannot go wrong"

Examples for errors

- General: Applying operators that are not defined on all inputs

1+'string' //ill-typed

1+1 //well-typed

...

public Integer f(Integer i) { return 2/i; }

...

f(true) // ill-typed

f(0) // ill-typed?

Foundations of Types: What are Errors? (II/IV)

"Well-typed programs cannot go wrong"

Examples for errors

- General: Applying operators that are not defined on all inputs
- Object Orientation (OO): Calling a method that is not supported

```
public class C {  
    public Integer f(Integer i) { return i*2; }  
}  
...  
C c = new C();  
c.g(1);
```

Foundations of Types: What are Errors? (III/IV)

"Well-typed programs cannot go wrong"

Examples for errors

- General: Applying operators that are not defined on all inputs
- OO: Calling a method that is not supported
- Concurrent: Deadlock

?? How to specify deadlocks? → channel types

Foundations of Types: What are Errors? (IV/IV)

"Well-typed programs cannot go wrong"

Examples for errors

Not every error is considered a type error. Sometimes the line is not clear, e.g., for null access.

```
public void method(C o){ o.m(); } //Java: Type C allows null  
...  
this.method(null);
```

```
fun method(o : C){ o.m(); } //Kotlin: Type C does not allow null  
...  
this.method(null);
```

Type Soundness

“Well-typed programs cannot go wrong”

Type soundness

If a program adheres to its types at compile time, then certain errors do not occur at runtime

- Formalized either as reachability or reduction.
- $e_1 \rightsquigarrow e_2$ is one execution/evaluation step from e_1 to e_2

Type soundness as reachability

- A bad operation results in an error state.
- Well-typed programs never reach an error state.

$$\begin{array}{ll} (1 + 1) + 1 \rightsquigarrow 2 + 1 \rightsquigarrow 1 & \checkmark \\ (1 + 1) + 'a' \rightsquigarrow 2 + 'a' \rightsquigarrow \text{error} & \times \end{array}$$

Type soundness as reduction

- A bad operation blocks the program.
- Well-typed programs never block.

$$\begin{array}{ll} (1 + 1) + 1 \rightsquigarrow 2 + 1 \rightsquigarrow 1 & \checkmark \\ (1 + 1) + 'a' \rightsquigarrow 2 + 'a' & \times \end{array}$$

Type Analysis

How would we analyze this? How would we formally reason about it?

Completeness of Type Systems

Types and logic

Type systems and logics share some properties

- Notions of soundness and completeness
- Judgment (later today)
- Dual use as documentation and specification

Static types

Static type systems are typically incomplete

- In many cases to keep them are decidable
- Their wide adaption hints that the incomplete part is not important in practice

Dynamic types

Dynamic type systems are “complete”, but detect the error too late.

A Simple Type System

A typing discipline consists of

- A type syntax
- A subtyping relation
- A typing environment
- A type judgment
- A set of type rules (the type system itself)
- A notion of type soundness

Next slides

A simple type system for a simple sequential language.

A Simple Type System

Typing Literal Expressions

Language syntax

Expression e with integer ($n \in \mathbb{Z}$) and Boolean literals:

$$e ::= n \mid \text{true} \mid \text{false} \mid e + e \mid e \wedge e \mid e \leq e$$

Type syntax

Booleans and integers:

$$T ::= \text{Bool} \mid \text{Int}$$

- 1
- $1 + 2 \leq 3$
- We allow parentheses if necessary $((1 + 2) \leq 3) \wedge \text{true}$

A Simple Type System: Typing Judgment

A judgment is a meta-statement over formal constructs.

Typing judgment

To express that an expression e is well-typed with type T . We write

$$\vdash e : T$$

- Judgment is *true*: $\vdash 1 + 1 : \text{Int}$
- Judgment is *false*: $\vdash 1 + 1 : \text{Bool}$

A Simple Type System: Typing Rules

Typing rules

- A typing rule contains one conclusion (conclusion) and a list of premises (premise_i).
- Each conclusion and premise is one judgment
- Its meaning is that if all premises are true, then the conclusion is also true (inference rule)
- A rule without premises is an *axiom* and expresses that something is always true

Notation:

$$\frac{\text{premise}_1 \quad \dots \quad \text{premise}_n}{\text{conclusion}} \text{rule name}$$

Our axioms:

$$\frac{}{\vdash \textit{false} : \text{Bool}} \text{bool-f}$$

$$\frac{}{\vdash \textit{true} : \text{Bool}} \text{bool-t}$$

$$\frac{}{\vdash n : \text{Int}} \text{int-literal}$$

A Simple Type System: Expression Rules

The following expresses that if e_1 and e_2 can be typed with boolean type, then so can $e_1 \wedge e_2$.

$$\frac{\vdash e_1 : \text{Bool} \quad \vdash e_2 : \text{Bool}}{\vdash e_1 \wedge e_2 : \text{Bool}} \text{ bool-and}$$

The following expresses that if e_1 and e_2 can be typed with integer type, then so can $e_1 + e_2$.

$$\frac{\vdash e_1 : \text{Int} \quad \vdash e_2 : \text{Int}}{\vdash e_1 + e_2 : \text{Int}} \text{ int-plus}$$

The following expresses that if e_1 and e_2 can be typed with integer type, then $e_1 \leq e_2$ can be typed with boolean type.

$$\frac{\vdash e_1 : \text{Int} \quad \vdash e_2 : \text{Int}}{\vdash e_1 \leq e_2 : \text{Bool}} \text{ bool-leq}$$

A Simple Type System: Typing Tree

A typing rule is a schema that can be applied to a concrete expression. If we do so repeatedly, then the result is a typing tree.

Typing tree

A typing tree is a tree, where each node is a type rule application on a concrete expression.
A tree is closed if all leaves are stemming from axioms.

Rule:

$$\frac{\vdash e_1 : \text{Int} \quad \vdash e_2 : \text{Int}}{\vdash e_1 + e_2 : \text{Int}} \text{ int-plus}$$

Rule application:

$$\frac{\vdash 12 : \text{Int} \quad \vdash 13 : \text{Int}}{\vdash 12 + 13 : \text{Int}} \text{ int-plus}$$

A Simple Type System: Example

$$\frac{\frac{\frac{\vdash 1 : \text{Int}}{\text{int-literal}} \quad \frac{\vdash 2 : \text{Int}}{\text{int-literal}}}{\vdash 1 + 2 : \text{Int}} \text{int-plus}}{\vdash (1 + 2) \leq 3 : \text{Bool}}$$
$$\frac{\vdash 3 : \text{Int}}{\text{int-literal}}$$
$$\text{bool-leq}$$

This means that $1 + 2 \leq 3$ indeed has type Bool.

$$\frac{\frac{\frac{\vdash \text{true} : \text{Int}}{\text{int-literal}} \quad \frac{\vdash 2 : \text{Int}}{\text{int-literal}}}{\vdash \text{true} + 2 : \text{Int}} \text{int-plus}}{\vdash \text{true} + 2 \leq 3 : \text{Bool}}$$
$$\frac{\vdash 3 : \text{Int}}{\text{int-literal}}$$
$$\text{bool-leq}$$

This means that $\text{true} + 2 \leq 3$ does not have type Bool.

A Simple Type System: Termination

We have types and typing rules, for type soundness we also need expression evaluation.

Evaluation

We do not define evaluation formally here, but assume that $e_1 \rightsquigarrow e_2$ is one execution/evaluation step from e_1 to e_2 .

- $1 + 2 \rightsquigarrow 3$
- $1 + 2 \leq 5 \rightsquigarrow 3 \leq 5$
- $3 \leq 3 \rightsquigarrow \text{true}$

Literals and termination

An evaluation of expression e_1 *successfully terminates*, if

$$e_1 \rightsquigarrow \dots \rightsquigarrow e_{\text{final}}$$

and e_{final} is either a literal, e.g. an integer literal n or a boolean literal ($\text{true}, \text{false}$), or if e_1 is one of these expressions itself.

A Simple Type System: Type Soundness

Type soundness

Typically, *soundness* (also called *safety*) requires three properties:

- All expressions that are successfully terminated are well-typed
- If a well-typed expression can evaluate, then the result is well-typed (preservation/reduction)
- If a well-typed expression is not successfully terminated, then it can evaluate (progress)

Together, these properties imply that if an expression is well-typed, and its evaluation terminates, then it terminates successfully.

A Simple Type System: Preservation & Progress

Preservation

If a well-typed expression can evaluate, then the result is well-typed

$$\forall e, e', T. ((e : T \wedge e \rightsquigarrow e') \rightarrow e' : T)$$

Progress

If a well-typed expression is not successfully terminated ($\text{term}(e)$), then it can evaluate

$$\forall e, T. ((e : T \wedge \neg\text{term}(e)) \rightarrow \exists e'. e \rightsquigarrow e')$$

- Preservation states that typeability is an invariant
- Progress is almost deadlock freedom, typically harder to proof
- More general formulations possible

Typing Environment and Subtyping

A Simple Type Environment

- We can now type simple expressions
- How do we move towards types for concurrency?
- Next two ingredients:
- Typing of variables
 - Typing variables requires to keep track of which variables are declared
 - We will record information in a *type environment*
- Subtyping
 - We will introduce a second judgment to express the relation between types
- Typing environments and subtyping relations are critical for channel types

A Simple Type Environment: Variables

Language syntax

Expressions with integer and boolean literals:

$$e ::= n \mid \text{true} \mid \text{false} \mid e + e \mid e \wedge e \mid e \leq e \mid v$$

Type syntax (unchanged)

Booleans and integers:

$$T ::= \text{Bool} \mid \text{Int}$$

- v
- $1 + v \leq 3$
- We allow parentheses $((1 + v) \leq 3) \wedge w$

A Simple Type Environment: Definition

Type environment

A type environment Γ is a map from variables to types.

- Notation to access the type of a variable v in environment Γ : $\Gamma(v)$
- Notation for an environment with two integer variables v, w :

$$\Gamma = \{v \mapsto \text{Int}, w \mapsto \text{Int}\}$$

An empty type environment is denoted $\Gamma = \emptyset$.

- Notation for updating the environment

$$\Gamma[x \mapsto T] = \Gamma'$$

, where $\Gamma'(x) = T$ and $\Gamma'(y) = \Gamma(y)$ for all other variables $y \neq x$.

- Notation if a variable has no assigned type

$$\Gamma(x) = \perp$$

A Simple Type Environment: Type Judgment

Type judgment

The type judgment includes the type environment:

$$\Gamma \vdash e : T$$

This reads as *expression e has type T if all variables are as described by Γ .*

New rule: The premise is a new judgment that holds iff the equality holds.

$$\frac{\Gamma(v) = T}{\Gamma \vdash v : T} \text{ var}$$

The type environment is added to all other rules and carried over from conclusion to premises.

For example:

$$\frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \text{Bool}}{\Gamma \vdash e_1 \wedge e_2 : \text{Bool}} \text{ bool-and}$$

A Simple Type Environment: Examples

Typing now depends on the type of the variables. Let $\Gamma_1 = \{v \mapsto \text{Int}\}$, $\Gamma_2 = \emptyset$

$$\frac{}{\Gamma_1 \vdash 1 : \text{Int}} \text{int-literal} \quad \frac{\Gamma_1(v) = \text{Int}}{\Gamma_1 \vdash v : \text{Int}} \text{var} \quad \frac{}{\Gamma_1 \vdash 3 : \text{Int}} \text{int-literal} \quad \frac{\Gamma_1 \vdash 1 + v : \text{Int} \quad \Gamma_1 \vdash v : \text{Int}}{\Gamma_1 \vdash 1 + v \leq 3 : \text{Bool}} \text{int-plus} \quad \frac{\Gamma_1 \vdash 1 + v \leq 3 : \text{Bool}}{\Gamma_1 \vdash (1 + v) \leq 3 : \text{Bool}} \text{bool-leq}$$

$$\frac{}{\Gamma_2 \vdash 1 : \text{Int}} \text{int-literal} \quad \frac{\Gamma_2(v) = \text{Int}}{\Gamma_2 \vdash v : \text{Int}} \text{var} \quad \frac{\Gamma_2 \vdash 3 : \text{Int}}{\Gamma_2 \vdash 1 + v \leq 3 : \text{Bool}} \text{int-literal} \quad \frac{\Gamma_2 \vdash 1 + v : \text{Int} \quad \Gamma_2 \vdash v : \text{Int}}{\Gamma_2 \vdash 1 + v \leq 3 : \text{Bool}} \text{int-plus} \quad \frac{\Gamma_2 \vdash 1 + v \leq 3 : \text{Bool}}{\Gamma_2 \vdash (1 + v) \leq 3 : \text{Bool}} \text{bool-leq}$$

A Simple Type Environment: Evaluation

Evaluation

Let σ be map from variables to literals. We do not define evaluation formally here, but assume that $e_1 \rightsquigarrow_{\sigma} e_2$ is one execution/evaluation step from e_1 to e_2 . In particular, $v \rightsquigarrow_{\sigma} \sigma(v)$.

A Simple Type Environment: Preservation & Progress

Preservation

If a well-typed expression can evaluate, then the result is well-typed

$$\forall \Gamma, e, e', T. ((\Gamma \vdash e : T \wedge e \rightsquigarrow e') \rightarrow \Gamma \vdash e' : T)$$

Progress

If a well-typed expression is not successfully terminated ($\text{term}(e)$), then it can evaluate

$$\forall \Gamma, e, T. ((\Gamma \vdash e : T \wedge \neg \text{term}(e)) \rightarrow \exists e'. e \rightsquigarrow e')$$

- Additionally, we must ensure that σ , adheres to Γ
- For every variable v we must have $\Gamma \vdash \sigma(v) : \Gamma(v)$

Simple Subtyping

- Let us introduce a simple subtype of rational numbers: integers
- We need to extend the type syntax, adjust the typing rules and formalize subtyping
- Subtyping is formalized as a special type

Type syntax

Booleans, integers and rational numbers:

$$T ::= \text{Bool} \mid \text{Int} \mid \text{Number}$$

$$\frac{n \in \mathbb{Z}}{\vdash n : \text{Int}} \text{ int-literal}$$

Simple Subtyping: Rules

We introduce a new judgment to express that T_1 is a subtype of T_2 : $T_1 <: T_2$

Reflexivity and transitivity

Every type is a subtype of itself (reflexive), subtyping is transitive

$$\frac{}{T <: T} \text{ T-refl}$$

$$\frac{T_1 <: S \quad S <: T_2}{T_1 <: T_2} \text{ T-trans}$$

Core rules

The actual subtyping rules are specific for the language, for us it is just this one

$$\frac{}{\text{Int} <: \text{Number}} \text{ T-int}$$

Application

At every point during type-checking, we can chose to use a subtype

$$\frac{S <: T \quad \Gamma \vdash e : S}{\Gamma \vdash e : T} \text{ T-sub}$$

Simple Subtyping: Example

Now we can type the literal 1 with Int using the new rules

$$\frac{\text{Int} <: \text{Number}}{\emptyset \vdash 1 : \text{Number}} \text{ T-int} \quad \frac{\overline{1 \in \mathbb{Z}}}{\emptyset \vdash 1 : \text{Int}} \text{ int-literal}$$
$$\frac{\text{Int} <: \text{Number} \quad \emptyset \vdash 1 : \text{Int}}{\emptyset \vdash 1 : \text{Number}} \text{ T-sub}$$

- Soundness is not affected by subtyping
- Rule T-sub is not *syntax-directed*
 - Can always be applied
 - Requires to chose a suitable S
 - Hard to implement in an algorithm
- This is orthogonal to concurrency, Pierce (Ch. 16) has details on algorithmic subtyping

Syntax-directed Subtyping

- Instead of T-sub, we can allow subtyping in other rules
- In the rest of the lecture, we do no use T-sub

$$\frac{\vdash e_1 : T_1 \quad T_1 <: \text{Number} \quad \vdash e_2 : T_2 \quad T_2 <: \text{Number}}{\vdash e_1 + e_2 : \text{Number}} \text{ number-plus}$$

Types for Statements

Syntax

Language

Expressions are as before, statements are a simple imperative language

$$\begin{aligned}s ::= & v = e; s \mid T v = e; s \\& \mid \text{skip} \mid \text{if}(e)\{s\} s\end{aligned}$$

Type syntax

Integers, rational number, booleans, unit type. Subtyping as before.

$$T ::= \text{Int} \mid \text{Number} \mid \text{Bool} \mid \text{Unit}$$

- Unit type is used to type statements
- A statement has unit type if it is typeable, and no type if it is not typeable
- Akin to **void** in Java
- No subtype relation to any other type

Type System (I/II)

Rules for expressions are as before.

Simple Statements

Skip is always well-typed

$$\frac{}{\Gamma \vdash \text{skip} : \text{Unit}} \text{skip}$$

Assignment

Assignment checks that the type of the expression is a subtype of the variable, and that the continuation is typeable. Note that this also checks that the variable is declared – otherwise $\Gamma(v) = \perp$ and the second premise fails.

$$\frac{\Gamma \vdash e : S \quad S <: \Gamma(v) \quad \Gamma \vdash s : \text{Unit}}{\Gamma \vdash v = e; s : \text{Unit}} \text{assign}$$

Type System (II/II)

Declaration

Declaration is as before, but additionally updates the environment for the continuation.

$$\frac{\Gamma \vdash e : S \quad S <: T \quad \Gamma[v \mapsto T] \vdash s : \text{Unit}}{\Gamma \vdash T v = e; s : \text{Unit}} \text{ decl}$$

Branching

Branching checks that the condition has boolean type, and both conditional statement and continuation. This implements scoping: if the environment get updated by s_1 , then these declarations are lost for s_2 .

$$\frac{\Gamma \vdash e : \text{Bool} \quad \Gamma \vdash s_1 : \text{Unit} \quad \Gamma \vdash s_2 : \text{Unit}}{\Gamma \vdash \mathbf{if}(e)\{s_1\}s_2 : \text{Unit}} \text{ branch}$$

Soundness (I/II)

- Important: terminated program must be well-typed!
- If one uses the error state, it *must not* be well-typed.

Type soundness

If statement s can be typed with `Unit`, and its execution terminates, then it terminates with s is fully reduced to **skip**.

`Int v = 1; v = v + 2`

$\rightsquigarrow v = v + 2$

$\sigma = \{v \mapsto 1\}$

$\rightsquigarrow \text{skip}$

$\sigma = \{v \mapsto 3\}$

`Int v = 1; v = v + true`

$\rightsquigarrow v = v + \text{true}$

$\sigma = \{v \mapsto 1\}$

Soundness (II/II)

Remarks

- Usual preservation and progress properties
- Initial typing starts with empty environment, i.e., no declared variables
- Each branch and programs ends in skip.

Environment Example

Environment

A variable v must be declared to be recorded in the environment, otherwise any rule that tries to evaluate $\Gamma(v)$ fails.

$$\frac{\vdots}{\begin{array}{c} \frac{\{v \mapsto \text{Int}\} \vdash v + 2 : \text{Int}}{\{v \mapsto \text{Int}\} \vdash v = v + 2; \text{skip} : \text{Unit}} \quad \frac{\text{Int} <: \text{Int}}{\{v \mapsto \text{Int}\} \vdash \text{skip} : \text{Unit}} \quad \frac{}{1 \in \mathbb{Z}} \\ \frac{\emptyset \vdash 1 : \text{Int}}{\emptyset \vdash \text{Int } v = 1; v = v + 2; \text{skip} : \text{Unit}} \quad \frac{}{\text{Int} <: \text{Int}} \end{array}}$$
$$\frac{\{v \mapsto \text{Int}\} \vdash w : \text{Int} \quad \text{Int} <: \text{Int}}{\{v \mapsto \text{Int}\} \vdash v = w; \text{skip} : \text{Unit}} \quad \frac{\{v \mapsto \text{Int}\} \vdash \text{skip} : \text{Unit}}{\emptyset \vdash 1 : \text{Int}} \quad \frac{1 \in \mathbb{Z}}{\emptyset \vdash \text{Int } v = 1; v = w; \text{skip} : \text{Unit}} \quad \frac{}{\text{Int} <: \text{Int}}$$

Example

Scoping

Scoping is implemented by not transferring the updated environment. In our rule for branching, we type the continuation with the type environment *before* the branching – all variables declared within are lost.

$$\frac{\text{---} \quad \vdots}{\frac{\emptyset \vdash \text{true} : \text{Bool} \quad \emptyset \vdash \text{Int } v = 1; \text{ skip} : \text{Unit} \quad \emptyset \vdash v = 2; \text{ skip} : \text{Unit}}{\emptyset \vdash \text{if}(\text{true})\{\text{Int } v = 1; \text{ skip}\}v = 2; \text{ skip} : \text{Unit}}}$$

Channel Types

Typing Channels

- From now on, we will not fully define a language and give all rules
- Syntax will be Go-like (goroutines, channel operations)
- Real Go-Code will be annotated with Go

Mismatched message types

The basic error is that the receiver expects the result to be of a different type than the value the sender sends. Implemented in Go.

Go

```
c := make(chan int)
go func() { c <- "foo" }()
res := (<-c) + 1
```

cannot use "foo" (untyped string constant) as int value in send

A Simple Type System for Channels: Variance

Types

If T is type then $\text{chan } T$ is a type.

Variance

Let $T <: T'$, with $T \neq T'$. A type constructor C is

- *Covariant* if $C(T) <: C(T')$
- *Contravariant* if $C(T') <: C(T)$
- *Invariant* if $C(T') \not<: C(T) \wedge C(T) \not<: C(T')$

Subtyping

Channels types are *covariant*: If T is a subtype of T' then $\text{chan } T$ is a subtype of $\text{chan } T'$.

A Simple Type System for Channels: Rule for Writing

Typing writing

$$\frac{\Gamma \vdash e : \text{chan } T \quad \Gamma \vdash e' : T' \quad T' <: T}{\Gamma \vdash e \leftarrow e' : \text{Unit}}$$

- First premise types channel
- Second premise types sent value
- Third premise connects via subtyping

Go

```
type Cat struct{};type Car struct{}

type Animal interface{ name() string }

func main() {
    ch := make(chan Animal)
    go func(c chan Animal) { c <- Cat{} }(ch)
    func (Cat) name() string { return "Meowth" }
```

A Simple Type System for Channels: Rule for Reading

Typing reading

$$\frac{\Gamma \vdash e : \text{chan } T' \quad T' <: T}{\Gamma \vdash \leftarrow e : T}$$

- Essentially the same as calling a method and reading its result

A Glimpse of Input/Output Modes (I/III)

Beware! The next slides use modified Go-like syntax:

- `<-chan` becomes `chan?`
- `chan->` becomes `chan!`
- `chan` becomes `chan!?`

Modes

- The previous system makes sure the sent data has the right data, but does not consider the direction.
- Modes specify the direction of a channel in a given scope

Go

```
c := make(chan int)
go func() { c<-1 }()
res := (<-c) + 1
```

A Glimpse of Input/Output Modes (II/III)

Types

Channel types are now annotated with their *mode* or *capability*.

$$T ::= \dots \mid \text{chan}_M \ T \quad M ::= ! \mid ? \mid !?$$

- A channel that can only read/receive: ?
- A channel that can only write/send: !
- A channel that allows both: !?

Subtyping

We can pass a channel that allows both operation to a more constrained context

$$\text{chan}_! \ T <: \text{chan}_{!?} \ T$$

$$\text{chan}_? \ T <: \text{chan}_{!?} \ T$$

A Glimpse of Input/Output Modes (III/III)

- How to use channels with restricted mode !?
- Either use subtyping at every evaluation (like in Go)
- Or use *weakening* to enforce that subtyping relation is used only once
- This ensures that once a channel is used for reading (writing) once in a thread, then it is only used for reading (writing) afterwards

```
func main() {  
    chn := make(chan!? int) //!?  
    go read(chn) //!?  
    // weaken chn to chan! int  
    chn <- v //<- c would be illegal  
}  
func read(c chan? int) int { // removes ! mode  
    return <-c //c <- 1 would be illegal }
```

Input/Output Modes: Rules (I/II)

Weakening rule

Allows to make a type less specific. This is *not* just using the T-sub rule – we modify the stored type in the environment.

$$\frac{\Gamma[x \mapsto T''] \vdash s : T \quad T'' <: T'}{\Gamma[x \mapsto T'] \vdash s : T} \text{-weak}$$

Other rules: Receive and send with modes

$$\frac{\Gamma \vdash e : \text{chan}_! T \quad \Gamma \vdash v : T' \quad T' <: T}{\Gamma \vdash e \leftarrow v : \text{Unit}} \text{M-send}$$

$$\frac{\Gamma \vdash e : \text{chan}? T' \quad T' <: T}{\Gamma \vdash \leftarrow e : T} \text{M-receive}$$

Input/Ouput Modes: Rules (II/II)

Other rules: receive and send with modes

$$\frac{\Gamma \vdash e : \text{chan}_! T \quad \Gamma \vdash v : T' \quad T' <: T}{\Gamma \vdash e \leftarrow v : \text{Unit}} \text{M-send}$$

$$\frac{\Gamma \vdash e : \text{chan}? T' \quad T' <: T}{\Gamma \vdash \leftarrow e : T} \text{M-receive}$$

Important: No subtyping on $\text{chan}? T'$ and $\text{chan}_! T$. A channel must be weakened before it can be used!

Next Lecture: Linear types

Wrap-up

Today's lecture

- General structure of static type systems
- Simple type systems for channels
- Introduction: modes

Upcoming lectures

- More on modes
- More complex channel types
 - Linear types
- Uniqueness types, towards the ownership system of Rust

Part 3: Linearity

Andrea Pferscher

October 29, 2025

University of Oslo

Recap: Setting up a Type System

- A type syntax (T)
- A subtyping relation ($T <: T'$)
- A typing environment ($\Gamma : \text{Var} \mapsto T$)
- A type judgment ($\Gamma \vdash e : T$)
- A set of type rules and a notion of type soundness

Topic today: type systems for message-passing concurrency

Recap: Data vs. Behavioral Type, Syntax and Subtyping

Data and behavioral types

- A data type is an abstraction over the contents of memory
 - Can it be interpreted as a member of a set? E.g., integers
 - Are certain operations *defined* on it? E.g., + or method lookup
- A behavioral type is an abstraction over *allowed* operations

Goal:

- In channel types, the operations are channel operations
- Specify, document and ensure intended communication patterns
- In the very best case: ensure deadlock freedom

Recap: Environment and Judgment

Type environment

A type environment Γ is a partial map from variables to types.

- Notation to access the type of a variable v in environment Γ : $\Gamma(v)$
- Example notation for an environment with two integer variables v, w : $\{v \mapsto \text{Int}, w \mapsto \text{Int}\}$
- Notation for updating the environment: $\Gamma[x \mapsto T]$
- Notation if a variable has no assigned type: $\Gamma(x) = \perp$

Type judgment

To express that statement e is well-typed with type T in environment Γ .

$$\Gamma \vdash e : T$$

Recap: Type Soundness

Type soundness expresses that if the program is well-typed, then evaluation does not block.

- Three intermediate lemmas (error states are not well-typed, preservation, progress)
- Note that we do not ensure termination
- Main consideration for later: Are deadlocked states successfully terminated?

Preservation

If a well-typed expression can be evaluated, then the result is well-typed

$$\forall s, s', \Gamma. ((\Gamma \vdash s : \text{Unit} \wedge s \rightsquigarrow s') \rightarrow \exists \Gamma'. \Gamma' \vdash s' : \text{Unit})$$

Progress

If a statement is well-typed, but not successfully terminated (i.e., **skip**), then it can evaluate

$$\forall s. (\Gamma \vdash s : \text{Unit} \wedge \neg \text{term}(s) \rightarrow \exists s'. s \rightsquigarrow s')$$

Channel Types

Typing Channels

- From now on, we will not fully define a language and give all rules
- Syntax will be Go-like (goroutines, channel operations)
- Real Go-Code will be annotated with Go

Mismatched message types

The basic error is that the receiver expects the result to be of a different type than the value the sender sends. The type system of Go can detect such error.

Go

```
c := make(chan int)
go func() { c <- "foo" }()
res := (<-c) + 1
```

cannot use "foo" (untyped string constant) as int value in send

A Simple Type System for Channels (I/III)

Types

If T is type then $\text{chan } T$ is a type.

Variance

Let $T <: T'$, with $T \neq T'$. A type constructor C is

- *Covariant* if $C(T) <: C(T')$
- *Contravariant* if $C(T') <: C(T)$
- *Invariant* if $C(T') \not<: C(T) \wedge C(T) \not<: C(T')$

Subtyping

Channels types are *covariant*: If T is a subtype of T' then $\text{chan } T$ is a subtype of $\text{chan } T'$.

A Simple Type System for Channels (II/III)

Typing send

$$\frac{\Gamma \vdash e : \text{chan } T \quad \Gamma \vdash e' : T' \quad T' <: T}{\Gamma \vdash e \leftarrow e' : \text{Unit}}$$

- First premise types channel
- Second premise types sent value
- Third premise connects via subtyping

Go

```
type Cat struct{};type Car struct{}

type Animal interface{ name() string }

func main() {
    ch := make(chan Animal)
    go func(c chan Animal) { c <- Cat{} }(ch)
    func (Cat) name() string { return "Meowth" }
```

A Simple Type System for Channels (III/III)

Typing receiving

$$\frac{\Gamma \vdash e : \text{chan } T' \quad T' <: T}{\Gamma \vdash <- e : T}$$

- Essentially the same as calling a method and receiving its result

A Glimpse of Input/Output Modes (I/III)

Beware! The next slides use modified Go-like syntax:

- `<-chan` becomes `chan?`
- `chan->` becomes `chan!`
- `chan` becomes `chan!?`

Modes

- The previous system makes sure the sent data has the right data, but does not consider the direction.
- Modes specify the direction of a channel in a given scope

Go

```
c := make(chan int)
go func() { c<-1 }()
res := (<-c) + 1
```

A Glimpse of Input/Output Modes (II/III)

Types

Channel types are now annotated with their *mode* or *capability*.

$$T ::= \dots \mid \text{chan}_M \ T \quad M ::= ! \mid ? \mid !?$$

- A channel that can only receive: ?
- A channel that can only send: !
- A channel that allows both: !?

Subtyping

We can pass a channel that allows both operation to a more constrained context

$$\text{chan}_! \ T <: \text{chan}_{!?} \ T$$

$$\text{chan}_? \ T <: \text{chan}_{!?} \ T$$

A Glimpse of Input/Output Modes (III/III)

- How to use channels with restricted mode !?
- Either use subtyping at every evaluation (like in Go)
- Or use *weakening* to enforce that subtyping relation is used only once
- This ensures that once a channel is used for receiving (sending) once in a thread, then it is only used for receiving (sending) afterwards

```
func main() {  
    chn := make(chan!? int) //!?  
    go receive(chn) //!?  
    // weaken chn to chan! int  
    chn <- v //<- c would be illegal  
}  
func receive(c chan? int) int { // removes ! mode  
    return <-c //c <- 1 would be illegal }
```

Weakening

Weakening rule

Allows to make a type less specific. This is *not* just using the T-sub rule – we modify the stored type in the environment.

$$\frac{\Gamma[x \mapsto T''] \vdash s : T \quad T'' <: T'}{\Gamma \cup \{x \mapsto T'\} \vdash s : T} \text{-weak}$$

Input/Output Modes

Other rules: receive and send with modes

$$\frac{\Gamma \vdash e : \text{chan}_! T \quad \Gamma \vdash v : T' \quad T' <: T}{\Gamma \vdash e \leftarrow v : \text{Unit}} \text{M-send}$$

$$\frac{\Gamma \vdash e : \text{chan}? T' \quad T' <: T}{\Gamma \vdash \leftarrow e : T} \text{M-receive}$$

Important: No subtyping on $\text{chan}? T'$ and $\text{chan}_! T$. A channel must be weakened before it can be used!

More Channel Types

- Formalizing Γ -splitting and ensuring correct number of uses \rightarrow substructural/**linear types**
- Formalizing order \rightarrow **usage types**
- More expressive protocols and allows different types to be send \rightarrow session types

Learning goals of this lecture:

- How are order and capabilities used to structure concurrency?
- How are order and capabilities described in type systems?
- What parts of type systems must be modified?

Not in this lecture: Full formal treatment and most general cases.

- For this reason the language is a bit simplified.
- No arbitrary expressions, no nested channel types

Linear Types

Linear Types (I/II)

Linearity

The previous systems do not prevent the channels from being used *too little* or *too often*.

```
func main() {  
    chn := make(chan!? int)  
    <- chn //locks and waits forever  
}
```

Linear Types (II/II)

Linearity

The previous systems do not prevent the channels from being used *too little* or *too often*.

```
func main() {
    chn := make(chan!< int)
    go receive(chn)
    chn <- 1
    chn <- 1 //locks and waits forever
}

func receive(c chan? int) int {
    return <-c
}
```

Linearity

In types, logic and related fields, *linearity* refers to capabilities that are used *exactly once*.

- A linear channel can be used for exactly one send/receive operation
- A linear resource cannot be reused after being accessed, and must be accessed
- Simplifies reasoning about systems because one prohibits reuse in different context.
- In the following: no nested channel operations ($<- <- c$)

Linear Types: Syntax

Type syntax

Let T be a type, and $n, m \in \{0, 1\}$. $\text{chan}_{?n,!m} T$ is a channel type.

Multiplicity $!0$ denotes that the channel must not be used for a send operation, $!1$ that exactly one message must be sent. Analogously for $?$.

- $c \mapsto \text{chan}_{?1,!1} T$ is linear
- $c \mapsto \text{chan}_{?0,!0} T$ cannot be used anymore
- $c \mapsto \text{chan}_{?1,!0} T$ receiving possible but no sending anymore
- $c \mapsto \text{chan}_{?0,!1} T$ sending possible but no receiving anymore
- Subtyping possible, but not needed
- No weakening rule, syntax-driven subtyping

Linear Types: Example

The previous example can be written using linear types, and to forbid multiple accesses.

```
func main() {
    chn := make(chan<?1,!1> int)
    go receive(chn)
    chn <- 1 //chan<?0,!1> int
}

func receive(c chan<?1,!0> int) int {
    return <-c
}
```

Splitting the Environment

```
chn := make(chan<?1,!1> int)  
go receive(chn)
```

- Transfer capability to receive messages to new thread
- Limit capabilities for the current thread
- Ensure that no capabilities are remaining
- And catch violation, e.g, <-c + <-c

Linear Types: Definition of Splitting Environment

Typing environment

A typing environment Γ can be split into two environments Γ^1, Γ^2 by

- Having all variables with non-channel types in both Γ^1 and Γ^2 , and
- For each x with channel type we have $\Gamma(x) = \Gamma^1(x) + \Gamma^2(x)$, where

$$\text{chan}_{?n^1,!m^1} T + \text{chan}_{?n^2,!m^2} T = \text{chan}_{?n^1+n^2,!m^1+m^2} T$$

- $\text{chan}_{?1,!1} T = \text{chan}_{?0,!1} T + \text{chan}_{?1,!0} T$
- $\text{chan}_{?1,!1} T = \text{chan}_{?1,!1} T + \text{chan}_{?0,!0} T$

$$\{n \mapsto \text{Int}, c \mapsto \text{chan}_{?0,!1} \text{ Int}\} =$$

$$\{n \mapsto \text{Int}, c \mapsto \text{chan}_{?0,!0} \text{ Int}\} + \{n \mapsto \text{Int}, c \mapsto \text{chan}_{?0,!1} \text{ Int}\}$$

Linear Types: Definition of Complete Use

Literals and termination

- Γ is *unrestricted* if all contained channels have $n = 0$ and $m = 0$. We write $\text{un}(\Gamma)$.
- All literals only type check in an unrestricted environment
- First, sub-system only for expressions

$$\frac{\text{un}(\Gamma)}{\Gamma \vdash \text{true} : \text{Bool}} \text{-true}$$

$$\frac{\text{un}(\Gamma)}{\Gamma \vdash n : \text{Int}} \text{-int}$$

$$\frac{\text{un}(\Gamma) \quad \Gamma(v) = T}{\Gamma \vdash v : T} \text{-var}$$

$$\frac{\overline{\text{un}(\{c \mapsto \text{chan}_{?0,!0}\text{Int}\})}}{\{c \mapsto \text{chan}_{?0,!0}\text{Int}\} \vdash 1 : \text{Int}} \quad \frac{\overline{\text{un}(\{c \mapsto \text{chan}_{?1,!0}\text{Int}\})}}{\{c \mapsto \text{chan}_{?1,!0}\text{Int}\} \vdash 1 : \text{Int}}$$

Linear Types for Expressions: Typing Trees Examples

Splitting in arithmetic expressions

We split the environment at every point we descend into subexpressions. Number of splits depends on arity (number of arguments) of operator

$$\frac{\Gamma = \Gamma^1 + \Gamma^2 \quad \Gamma^1 \vdash e_1 : \text{Int} \quad \Gamma^2 \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 + e_2 : \text{Int}} \text{ L-add} \qquad \frac{\Gamma \vdash e : \text{Int}}{\Gamma \vdash -e : \text{Int}} \text{ L-minus}$$

Rules for Booleans operators are analogous

Linear receive

Rule for receiving requires that we are still allowed to receive

$$\frac{\Gamma(v) = \mathbf{chan}_{?1,!0} T \quad \mathbf{un}(\Gamma[v \mapsto \mathbf{chan}_{?0,!0} T])}{\Gamma \vdash \leftarrow v : T} \text{ L-receive}$$

Linear Types for Expressions: Examples

Type safe example:

$$\frac{\text{un}(\{\text{chan}_{?0,!0} \text{ int}\}) \quad \{x \mapsto \text{chan}_{?1,!0} \text{ int}\}(x) = \text{chan}_{?1,!0} \text{ int}}{\{x \mapsto \text{chan}_{?1,!0} \text{ int}\} \vdash (<-x) : \text{int}}$$
$$\frac{\text{un}(\{\text{chan}_{?0,!0} \text{ int}\})}{\{x \mapsto \text{chan}_{?0,!0} \text{ int}\} \vdash 1 : \text{int}}$$
$$\frac{\text{un}(\{\text{chan}_{?0,!0} \text{ int}\}) \quad \{x \mapsto \text{chan}_{?1,!0} \text{ int}\} + \{x \mapsto \text{chan}_{?0,!0} \text{ int}\} = \{x \mapsto \text{chan}_{?1,!0} \text{ int}\}}{\{x \mapsto \text{chan}_{?1,!0} \text{ int}\} \vdash (<-x) + 1 : \text{int}}$$

No-use prohibited:

$$\frac{\text{un}(\{\text{chan}_{?1,!0} \text{ int}\})}{\{x \mapsto \text{chan}_{?1,!0} \text{ int}\} \vdash 1 : \text{int}}$$
$$\frac{\text{un}(\{\text{chan}_{?0,!0} \text{ int}\})}{\{x \mapsto \text{chan}_{?0,!0} \text{ int}\} \vdash 2 : \text{int}}$$
$$\frac{\text{un}(\{\text{chan}_{?1,!0} \text{ int}\}) + \{x \mapsto \text{chan}_{?0,!0} \text{ int}\} = \{x \mapsto \text{chan}_{?1,!0} \text{ int}\}}{\{x \mapsto \text{chan}_{?1,!0} \text{ int}\} \vdash 1 + 2 : \text{int}}$$

Double-use prohibited:

$$\frac{\text{un}(\{\text{chan}_{?0,!0} \text{ int}\}) \quad \{x \mapsto \text{chan}_{?1,!0} \text{ int}\}(x) = \text{chan}_{?1,!0} \text{ int}}{\{x \mapsto \text{chan}_{?1,!0} \text{ int}\} \vdash (<-x) : \text{int}}$$
$$\frac{\text{un}(\{\text{chan}_{?0,!0} \text{ int}\}) \quad \{x \mapsto \text{chan}_{?0,!0} \text{ int}\} \vdash (<-x) : \text{int}}{\{x \mapsto \text{chan}_{?1,!0} \text{ int}\} \vdash (<-x) + (<-x) : \text{int}}$$
$$\frac{\text{un}(\{\text{chan}_{?0,!0} \text{ int}\}) + \{x \mapsto \text{chan}_{?0,!0} \text{ int}\} = \{x \mapsto \text{chan}_{?1,!0} \text{ int}\}}{\{x \mapsto \text{chan}_{?1,!0} \text{ int}\} \vdash (<-x) + (<-x) : \text{int}}$$

Linear Types for Statements

Termination

- All capabilities must be used up
- Either before termination (**skip**) or by our last expression (**return**)

$$\frac{\text{un}(\Gamma)}{\Gamma \vdash \mathbf{skip} : \text{Unit}} \text{ L-skip}$$

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \quad \Gamma_1 \vdash e : T \quad \text{un}(\Gamma_2)}{\Gamma \vdash \mathbf{return} \ e : \text{Unit}} \text{ L-return}$$

Linear Types for Statements: Examples

Example 1:

$$\frac{\{c \mapsto \text{chan}_{?1,!0} \text{Int}\} \vdash 0 : \text{Unit}}{\{c \mapsto \text{chan}_{?1,!0} \text{Int}\} \vdash \text{return } 0 : \text{Unit}}$$

Example 2: Let $\Gamma = \Gamma_1 = \{c \mapsto \text{chan}_{?1,!0} \text{ Int}\}$, $\Gamma_0 = \{c \mapsto \text{chan}_{?0,!0} \text{ Int}\}$

$$\frac{\begin{array}{c} \text{un}(\Gamma_1[c \mapsto \text{chan}_{?0,!0} \text{ Int}]) \\ \hline \Gamma_1(c) = \text{chan}_{?1,!0} \text{ Int} \end{array}}{\frac{\Gamma_1 \vdash c : \text{chan}_{?1,!0} \text{ Int}}{\frac{\Gamma_1 \vdash <-c : \text{Int}}{\Gamma \vdash \text{return } <-c : \text{Unit}}}} \quad \frac{\text{un}(\Gamma_0)}{\Gamma = \Gamma_1 + \Gamma_0}$$

Example 3: Let $\Gamma = \Gamma_2 = \{c \mapsto \text{chan}_{?1,!0} \text{Int}, d \mapsto \text{chan}_{?1,!0} \text{Int}\}$, $\Gamma_3 = \{c \mapsto \text{chan}_{?0,!0} \text{Int}, d \mapsto \text{chan}_{?0,!0} \text{Int}\}$

$$\frac{\begin{array}{c} \text{un}(\{c \mapsto \text{chan}_{?0,!0} \text{Int}, d \mapsto \text{chan}_{?1,!0} \text{Int}\}) \\ \hline \Gamma_2(c) = \text{chan}_{?1,!0} \text{Int} \end{array}}{\frac{\Gamma_2 \vdash c : \text{chan}_{?1,!0} \text{Int}}{\frac{\Gamma_2 \vdash <-c : \text{Int}}{\Gamma \vdash \text{return } <-c : \text{Unit}}}} \quad \frac{\text{un}(\Gamma_3)}{\Gamma = \Gamma_2 + \Gamma_3}$$

Linear Types for Statements: Sending Rule

Sending (version 1)

- Check that we can send now
- Remove send capability and split the environment into two parts
- One (Γ_1) records the send capability and the capabilities afterwards
- One (Γ_2) record the capabilities of the evaluated expression

$$\frac{\Gamma[c \mapsto \mathbf{chan}_{?n,!0} \ T] = \Gamma_1 + \Gamma_2 \quad \Gamma(c) = \mathbf{chan}_{?n,!1} \ T \quad \Gamma_1 \vdash s : \text{Unit} \quad \Gamma_2 \vdash e : T}{\Gamma \vdash c \leftarrow e; s : \text{Unit}} \text{ L-send}$$

Linear Types for Statements: Other Rules

- Remaining rules all have the same structure:
- Split environment for each subexpression/substatement
- Propagate split environment into each subexpression/substatement

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \quad \Gamma_1 \vdash e : T \quad \Gamma(v) = T \quad \Gamma_2 \vdash s : \text{Unit}}{\Gamma \vdash v = e; s : \text{Unit}} \text{ L-assign}$$

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 + \Gamma_3 \quad \Gamma_1 \vdash e : \text{Bool} \quad \Gamma_2 \vdash s_1 : \text{Unit} \quad \Gamma_2 \vdash s_2 : \text{Unit} \quad \Gamma_3 \vdash s_3 : \text{Unit}}{\Gamma \vdash \text{if}(e)\{s_1\} \text{ else}\{s_2\} s_3 : \text{Unit}} \text{ L-branch}$$

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \quad \Gamma_1 \vdash s_1 : \text{Unit} \quad \Gamma_2 \vdash s_2 : \text{Unit}}{\Gamma \vdash \text{go func()}\{s_1\}(); s_2 : \text{Unit}} \text{ L-parallel}$$

Example: Linear Types and Sequential Branching

Consider the following environments:

$$\Gamma = \{\text{chn} \mapsto \text{chan}_{?1,!1} \text{ Int}\}$$

$$\Gamma^? = \{\text{chn} \mapsto \text{chan}_{?1,!0} \text{ Int}\}$$

$$\Gamma^! = \{\text{chn} \mapsto \text{chan}_{?0,!1} \text{ Int}\}$$

$$\Gamma^0 = \{\text{chn} \mapsto \text{chan}_{?0,!0} \text{ Int}\}$$

Type-safe:

$$\frac{\vdots}{\Gamma^? \vdash (\text{chn} < -) \geq 0 : \text{Bool}} \quad \frac{\vdots}{\Gamma^! \vdash \text{chn} < -0 : \text{Unit}} \quad \frac{\vdots}{\Gamma^! \vdash \text{chn} < -1 : \text{Unit}} \quad \frac{\vdots}{\Gamma^0 \vdash \text{skip} : \text{Unit}} \quad \frac{}{\Gamma = \Gamma^? + \Gamma^! + \Gamma^0}$$
$$\Gamma \vdash \text{if}((\text{chn} < -) \geq 0)\{\text{chn} < -0\} \text{else}\{\text{chn} < -1\} \text{ skip} : \text{Unit}$$

Missed use in branch is detected:

$$\frac{\vdots}{\Gamma^? \vdash (\text{chn} < -) \geq 0 : \text{Bool}} \quad \frac{\vdots}{\Gamma^! \vdash \text{chn} < -0 : \text{Unit}} \quad \frac{\vdots}{\Gamma^! \vdash \text{skip} : \text{Unit}} \quad \frac{\vdots}{\Gamma^0 \vdash \text{skip} : \text{Unit}} \quad \frac{}{\Gamma = \Gamma^? + \Gamma^! + \Gamma^0}$$
$$\Gamma \vdash \text{if}((\text{chn} < -) \geq 0)\{\text{chn} < -0\} \text{else}\{\text{skip}\} \text{ skip} : \text{Unit}$$

Example: Linear Types and Parallelism

We can now, assuming a simple rule for function calls, prove the receiving example.

```
chn := make(chan<?1,!1> int)
go receive(chn)
chn <- 1
```

```
func receive(c chan<?1,!0> int) int {
    return <- c
}
```

$$\frac{\begin{array}{c} \vdots \\ \{chn \mapsto chan_{0,!1} int\} \vdash chn <-1; \mathbf{skip} : Unit \quad \{chn \mapsto chan_{1,!0} int\} \vdash go\ receive(chn) : Unit \end{array}}{\begin{array}{c} \{chn \mapsto chan_{0,!1} int\} + \{chn \mapsto chan_{1,!0} int\} \vdash go\ receive(chn); chn <-1; \mathbf{skip} : Unit \\ \{chn \mapsto chan_{1,!1} int\} \vdash go\ receive(chn); chn <-1 \mathbf{skip} : Unit \end{array}} \frac{}{\{ \} \vdash chn := make(chan <?1,!1 > int); go\ receive(chn); chn <-1; \mathbf{skip} : Unit}$$

Type Soundness

Is this enough?

To check that a channel is used exactly once, it is *not* enough to check that the multiplicity is 0 at the end – additionally one must ensure deadlock-freedom!

```
c1 := make(chan<!1,?1> bool)
if(<-c1){ c1 <- true}
```

```
c1 := make(chan<!1,?1> int)
c1 <- (<-c1)
```

Type Soundness: Enforce Parallelism

Sending

- Check that we can send to *but not receive from* c now
- Remove send capability and split the environment into two parts
- One (Γ_1) records the send capability and the capabilities afterwards
- One (Γ_2) records the capabilities of the evaluated expression
- Catches the two single-channel examples from the previous slides

$$\frac{\Gamma[c \mapsto \mathbf{chan}_{?0,!0} \ T] = \Gamma_1 + \Gamma_2 \quad \Gamma(c) = \mathbf{chan}_{?0,!1} \ T \quad \Gamma_1 \vdash s : \mathbf{Unit} \quad \Gamma_2 \vdash e : T}{\Gamma \vdash c \leftarrow e; s : \mathbf{Unit}} \text{L-send-DL}$$

Type Soundness

Is this enough?

To check that a channel is used exactly once, it is *not* enough to check that the multiplicity is 0 at the end – additionally one must ensure deadlock-freedom!

```
c1 := make(chan<!1,?1> int)
c2 := make(chan<!1,?1> int)
go func() {v := <-c1; c2 <- 1}()
w := <-c2; c1 <- 1
```

Type Soundness: Assignments

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \quad \Gamma_1 \vdash e : T \quad \Gamma'_2 \vdash s : \text{Unit} \quad \Gamma(v) = T}{\Gamma \vdash v = e; s : \text{Unit}} \text{ L-assign-DL}$$

- Where Γ'_2 sets all receive x in e to $\text{chan}_{?0,!0} T$ and is Γ_2 otherwise.

$$\forall x. \Gamma_1(x) = \text{chan}_{?1,!0} T \rightarrow \Gamma'_2(x) = \text{chan}_{?0,!0}$$

- Enforces that when one receives from or sends to a channel, the other capability has been passed to a different thread

Type Soundness

- One can apply the modification of L-assign-DL to all rules
- Guarantee: if system deadlocks more then one channel must be involved.
- Formalized: a state is successfully terminated if (1) all threads are terminated or (2) all threads are stuck or terminated and there are at least 2 stuck threads that waiting on 2 different channels.
- Deadlock analysis can be reduced to relations *between* channels.

```
c1 := make(chan<!1,?1> int)
c2 := make(chan<!1,?1> int)
go func() {v := <-c1; c2 <- 1}()
w := <-c2; c1 <- 1
```

- What else are linear type systems good for?
- Instead of delving into deadlock checkers: can we specify order more elegantly?

Dropping Unrestricted Environments

- What happens if we drop $\text{un}(\Gamma)$ everywhere?

```
c := make(chan<!1,?1> int)  
c <- 1
```

- We still have the restriction that we cannot use more than once

Affine types

A variable or channel is *affine* if it is used at most once. A variable or channel is *relevant* if it is used at least once.

- Not very useful for channels
- Useful for other types, e.g., to express that a declared variable may not be used, but if used then only once (for optimizations) or at least once (i.e., no dead declaration)

Other Uses for Linear Types

- Linearity must not be restricted to channel types
- Can be used to detect unused variables (with relevant types)
- Can be modified to be used for resource management
- In particular: every allocation (=declaration) must be paired with a deallocation (=use)

Normal Types and Linear Types in One Language: Syntax

- How to use linear and normal types for channels in one language?
- Idea: use a special symbol to distinguish arbitrary use
- Extend type syntax, environment split and notion of unrestricted environment

Type Syntax

Let T be a type, and $n, m \in \{0, 1, \omega\}$. $\text{chan}_{?n, !m} T$ is a type.

Multiplicity $!\omega$ denotes that the channel can be sent arbitrarily often, analogously for $?$.

Normal Types and Linear Types in One Language: Typing Environment

Typing environment

A typing environment Γ can be split into two environments $\Gamma^1 + \Gamma^2$ by

- Having all variables with non-channel types in both Γ^1 and Γ^2 .
- For each x with channel type we have $\Gamma(x) = \Gamma^1(x) + \Gamma^2(x)$, where

$$\text{chan}_{?n^1,!m^1} T + \text{chan}_{?n^2,!m^2} T = \text{chan}_{?n^1+n^2,!m^1+m^2} T$$

$$n + n' = n \text{ if } n' = 0$$

$$n + n' = n' \text{ if } n = 0$$

$$n + n' = \omega \text{ otherwise}$$

- $\text{chan}_{?\omega,!?\omega} T = \text{chan}_{?1,!1} T + \text{chan}_{?\omega,!?\omega} T$
- $\text{chan}_{?\omega,!?\omega} T = \text{chan}_{?0,!0} T + \text{chan}_{?\omega,!?\omega} T$
- $\text{chan}_{?\omega,!?\omega} T = \text{chan}_{?1,!1} T + \text{chan}_{?1,!1} T$

Normal Types and Linear Types in One Language: Adoptions

- Γ is unrestricted if all contained channels have $n = 0$ or $n = \omega$, and $m = 0$ or $m = \omega$.
- A channel is affine if we drop the restriction constraint, but it has been declared with

$$n = m = 1$$

- All rules stay the same except we must exchange every $n = 1$ for $n > 0$ (and same for m)

$$\frac{\Gamma \vdash e : \mathbf{chan}_{?n,!0} \ T \quad n > 0}{\Gamma \vdash \leftarrow e : T} \text{ L-receive}$$

Usage Types

Usage Types

- Linear types are not enough to describe protocols
- Consider a channel that is used as a lock
 - Channel is created, token is put it
 - Reading from channel is acquiring token
 - Sending to channel is releasing

Go

```
func main(){
    global = 0
    lock := make(chan int)
    finish := make(chan int)
    go dual(1, lock, finish)
    go dual(2, lock, finish)
    lock <- 0
    <-finish; <-finish
    <-lock
}
```

Go

```
func dual(i int, lock chan int,
          finish chan int) {
    <-lock
    //critical here
    lock <- 0
    //non-critical
    <-lock
    //critical here
    finish <- 0; lock <- 0
}
```

Usage Types

What is the type of lock? We need something that can express more than linear types!

Go

```
func dual(i int,
    lock chan<?ω,!ω> int,
    finish chan<?0,!1> int) {
    <-lock
    //critical here
    lock <- 0
    //non-critical
    lock <- 0 //bug!
    <-lock
    //critical here
    finish <- 0
    lock <- 0
}
```

Usage Types: Type Syntax

Type syntax

A usage describes the structure of all allowed actions on a channel.

$$T ::= \dots \dots \mid \text{chan}_U T$$

$U ::= 0$	cannot be used
$\mid ? . U$	receive
$\mid ! . U$	send
$\mid U + U$	parallel usage
$\mid U \& U$	alternative

- Not considering infinite/arbitrary channel usage ($*U$)

Usage Types: Examples

First receive, then send, then no usage:

?!.0

Receive or send, no other usage:

?0&!.0

Use for synchronization once:

?0+!.0

Synchronize twice:

?!.0+!.?.0

Usage Types: Splitting Type Environment

Splitting environment

Split is *explicit*.

$$\text{chan}_{U_1+U_2} T = \text{chan}_{U_1} T + \text{chan}_{U_2} T$$

- Also, $0 + 0 = 0$
- The operator $+$ is commutative, so

$$U_1 + U_2 = U_2 + U_1$$

An environment is unrestricted if all its channels are assigned 0

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \quad \Gamma_1 \vdash s_1 : \text{Unit} \quad \Gamma_2 \vdash s_2 : \text{Unit}}{\Gamma \vdash \text{go func}()\{ s_1 \}();\; s_2 : \text{Unit}} \text{ U-parallel}$$

Splitting Γ : Split Only at Start of New Thread!

Unsound: split at expressions

$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \quad \Gamma_1 \vdash e_1 : \text{Int} \quad \Gamma_2 \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 + e_2 : \text{Int}} \text{U-add-1}$$

Unsound: propagate

$$\frac{\Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 + e_2 : \text{Int}} \text{U-add-2}$$

Sound: match evaluation order on sequence

$$\frac{\Gamma = \Gamma_1.\Gamma_2 \quad \Gamma_1 \vdash e_1 : \text{Int} \quad \Gamma_2 \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 + e_2 : \text{Int}} \text{U-add-3}$$

- Here $\Gamma_1.\Gamma_2$ is the split along . for all channels used in e_1 and e_2

Usage Types: Rules for Send & Receive

Send

$$\frac{\Gamma + \{c \mapsto \text{chan}_U T\} \vdash s : \text{Unit} \quad \Gamma \vdash e : T' \quad T' <: T}{\Gamma + \{c \mapsto \text{chan}_{!U} T\} \vdash c \leftarrow e; s : \text{Unit}} \text{ U-send}$$

The rule for sending matches on *two* operators

- Sending (\leftarrow) is matched on !
- Sequence ($;$) is matched on .

Receive

This is the rule for receiving from a non-composed expression into a location, which can apply the same matching as for sending.

$$\frac{\Gamma + \{c \mapsto \text{chan}_U T\} \vdash s : \text{Unit} \quad \Gamma \vdash v : T' \quad T <: T'}{\Gamma + \{c \mapsto \text{chan}_{?U} T\} \vdash v = \leftarrow c; s : \text{Unit}} \text{ U-receive}$$

Splitting Γ : Example

$$\frac{\frac{\frac{\{c \mapsto \mathbf{chan}_{?.0} \text{ Int}\} \vdash (<- c) : \text{Int}}{\{c \mapsto \mathbf{chan}_{?.0} \text{ Int}\} \vdash (<- c) : \text{Int}} \quad \frac{\{c \mapsto \mathbf{chan}_0 \text{ Int}\} \vdash 1 : \text{Int}}{\{c \mapsto \mathbf{chan}_{?.0} \text{ Int}\} \vdash (<- c + 1) : \text{Int}}}{\{c \mapsto \mathbf{chan}_{?.0} \text{ Int}\} \vdash (<- c) + (<- c + 1) : \text{Int}} \quad \{c \mapsto \mathbf{chan}_{?.?.0} \text{ Int}\} = \{c \mapsto \mathbf{chan}_{?.0} \text{ Int}\}. \{c \mapsto \mathbf{chan}_{?.0} \text{ Int}\}}$$

Usage Types: Running Example (I/IV)

Go

```
func main(){
    global = 0
    lock := make(chan<!?.0 + ?.!.!.0 + ?!.?.!.0> int)
    finish := make(chan<?.?.0 + !.0 + !.0> int)

    go dual(1, lock, finish)
    go dual(2, lock, finish)
    lock <- 0
    <-finish
    <-finish
    <-lock
}
```

Usage Types: Running Example (II/IV)

- Let $\Gamma = \{\text{lock} \mapsto \text{chan}_{!.?.0+?!.?.0+?!.?.0} \text{ Int}, \text{ finish} \mapsto \text{chan}_{?.?.0+!.0+!.0} \text{ Int}, \text{ global} \mapsto \text{Int}\}$
- Let $\Gamma_1 = \{\text{lock} \mapsto \text{chan}_{!.?.0+?!.?.0+!.0} \text{ Int}, \text{ finish} \mapsto \text{chan}_{?.?.0+!.0} \text{ Int}, \text{ global} \mapsto \text{Int}\}$
- Let $\Gamma_2 = \{\text{lock} \mapsto \text{chan}_{?.!.?.!.0} \text{ Int}, \text{ finish} \mapsto \text{chan}_{!.0} \text{ Int}, \text{ global} \mapsto \text{Int}\}$

$$\frac{\vdots \quad \vdots}{\Gamma_1 \vdash s : \text{Unit} \quad \Gamma_2 \vdash \text{dual}(1, \text{ lock}, \text{ finish}) : \text{Unit}} \quad \Gamma = \mathbf{go} \text{ dual}(1, \text{ lock}, \text{ finish}); s : \text{Unit}$$

Usage Types: Running Example (III/IV)

- After another split at the two go's

$$\frac{\frac{\frac{\frac{\frac{\vdots}{\{lock \mapsto chan_0 \text{ int}, finish \mapsto chan_0 \text{ int}\} \vdash skip : Unit}}{\{lock \mapsto chan_{?.0} \text{ int}, finish \mapsto chan_0 \text{ int}\} \vdash \leftarrow lock : Unit}}{\{lock \mapsto chan_{?.0} \text{ int}, finish \mapsto chan_{?.0} \text{ int}\} \vdash \leftarrow finish; \leftarrow lock : Unit}}{\{lock \mapsto chan_{?.0} \text{ int}, finish \mapsto chan_{?.?.0} \text{ int}\} \vdash \leftarrow finish; \leftarrow finish; \leftarrow lock : Unit}}{\{lock \mapsto chan_{!.?.0} \text{ int}, finish \mapsto chan_{?.?.0} \text{ int}\} \vdash lock \leftarrow 0; \leftarrow finish; \leftarrow finish; \leftarrow lock : Unit}}$$

Usage Types: Running Example (IV/IV)

Go

```
func dual(i int,
    lock chan<?!.?!.0> int,
    finish chan<!0> int) {
    <-lock
    //critical here
    lock <- 0
    //non-critical
    lock <- 0 //bug!
    <-lock
    //critical here
    finish <- 0
    lock <- 0
}
```

- Found during typing: receive expected, but send found

$$\{lock \mapsto chan_{?!.0} \text{ int}, finish \mapsto chan_{!.0} \text{ int}\} \vdash lock <- 0; \dots : Unit$$

Limitations of Usages

Data types

Cannot express to first send one data type and then another one. e.g., first send a string and then an integer.

Split

Split must be done manually, programmer must ensure that both part match.

$!?.0 + !?.0 \quad x$

In particular with alternative.

$(!.0 \& ?.0) + (!.0 \& ?.0)$

Wrap-up

This lecture

- Linear types
 - Restrict and control how often operations are performed on value
 - General idea, used beyond channels
- Usage types
 - Explicitly specify order
 - Explicitly specify splits

Next lectures

Concurrency in Rust

Reading: *Type Systems for Concurrent Programs*, Naoki Kobayashi, 2002, Springer LNCS

Concurrency in Rust

Riccardo Sieve

November 5, 2025

University of Oslo

Outline

- Background on Types for Concurrent Systems
- Memory Management in Rust
- Threads in Rust
- Message Passing in Rust

Background

Types for Concurrent Systems

- So far: main ideas on theory of types
- Tool support external to language and standard library
 - Developers mixing libraries lose guarantees
 - Maintenance of libraries becomes a problem

Practical Types for Concurrent Systems

Rust, and to a smaller degree Go, have practical implementations

- Motivated by memory safety and concurrency
- Session types still external, but Rust library most sensible

Remark

Ownership types (introduced by David Clarke) are types for object oriented languages with similar basic ideas. Rust ownership system is not based on Clarke's ownership types.

Connecting Syntax and Semantics

There are several ideas and angles that connect syntax and semantics for memory and reference management.

- Linear types (deallocation after use)
- Aliasing (in OO)
- RAII (Resource Acquisition Is Initialization) and RBMM (Region-Based Memory Management)

Aliasing

Aliasing occurs if multiple references to one value/object exist

- Makes reasoning about the program more difficult
- Especially in concurrency: is there another *active* reference?
- Separates (semantic) value from (syntactic) variable

Connecting Syntax and Semantics

Region/Scope Based Memory Management (RBMM)

- RAII is mostly used to refer to OO, RBMM is more general
- Associate lexically-scoped part of the program with a *region* in the heap
- Region deallocated once scope exited
- Type checker ensures that no external pointers into region exist

Shared Themes

Linear types, uniqueness types, RBMM and alias analyses relate values, their lifetime at runtime and the syntactic structure of the program.

- Keep track of number of possible (types) references to reason about concurrent operations
- Prevent general errors from faulty memory management

Memory Management in Rust

Rust combines many ideas to guarantee memory and thread-safety, as well as static memory management without garbage collection

Ownership is how Rust manages memory

Ownership In Rust

- Each value (a String, i32, Vec, etc.) is owned by a single variable, called *owner*.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

Ownership in Rust

- Rust uses ownership to manage memory.
- This helps prevent data races at compile time.
- In other programming languages, memory is managed through garbage collection.
- In Rust, memory is managed through a set of ownership rules that enforce rules at compile time.
 - You can have only one owner for each value.
 - You can borrow a value through references, either mutably or immutably.
 - You can either have one mutable reference or many immutable references to a value at a time.
- An owner is usually a variable, but it can also be a data structure that contains other values.

Example in Rust

Does the following snippet of code compile?

Rust

```
fn calculate_length(s: String) -> usize {
    s.len()
}

fn main() {
    let s1 = String::from("Rust ownership");
    let s2 = s1;
    let len = calculate_length(s2);

    println!("The length of '{}' is {}", s1, len);
}
```

Ownership in Rust

- The previous code does not compile.
- We defined correctly the string s1
- However, when we assign s1 to s2, the ownership of the string is moved to s2
- This means that s1 is no longer valid and cannot be used
- If we try to print s1 after the assignment, we will get a compile-time error
- How can we fix it?

Solution 1: using references

Rust

```
fn calculate_length(s: &String) -> usize {
    s.len()
}

fn main() {
    let s1 = String::from("Rust ownership");
    let s2 = &s1; // we only assign the reference
    let len = calculate_length(s2);

    println!("The length of '{}' is {}", s1, len);
}
```

Solution 2: using clone

Rust

```
fn calculate_length(s: String) -> usize {
    s.len()
}

fn main() {
    let s1 = String::from("Rust ownership");
    let s2 = s1.clone(); // we clone s1
    let len = calculate_length(s2);

    println!("The length of '{}' is {}", s1, len);
}
```

Mutability and Immutability in Rust

- Even if we pass the reference to the string, it is still immutable.
- Trying to append a string, we will get a compile-time error.
- If we want to change the value of `s1`, we need to make it mutable.
- This must be done declaring the variable with `mut`.

Example in Rust

Does the following snippet of code compile?

Rust

```
fn append_crab(s: &String) {
    s.push_str("Ferris was here");
}

fn main() {
    let mut s = String::from("Rust is cool! ");
    append_crab(&s);

    println!("{}", s);
}
```

Mutability and Immutability in Rust

- By default, variables are immutable in Rust.
- When we defined the string `s1`, it is immutable by default.
- If we want to change the value of `s1`, we need to make it mutable.
- This must be done at the time of declaration.

Solution: use mut

Using `mut` we make the string mutable, so we can change its value.

Rust

```
fn append_crab(s: &mut String) {
    s.push_str("Ferris was here");
}

fn main() {
    let mut s = String::from("Rust is cool! ");
    append_crab(&mut s);

    println!("{}", s);
}
```

Lifetime

- A reference/variable has a lifetime from until it goes out-of-scope.
- One cannot have a reference with a longer lifetime than the value it refers to

Lifetime not respected due to scoping

Rust

```
fn main() {
    let ref_vec; //----+
    {           //  |
        let vec = vec![1, 2, 3]; //--+ |
        ref_vec = &vec;      // | |
    }           //--+ |
    println!("{}", (*ref_vec)[0]); //  |
}           //----+
```

Lifetime

- A reference/variable has a lifetime from until it goes out-of-scope.
- One cannot have a reference with a longer lifetime than the value it refers to

Lifetime respected

Rust

```
fn main() {  
    let vec = vec![1, 2, 3];           //----+  
    let refVec = &vec;                //--+ /  
    println!("{}", (*refVec)[0]);     // / /  
}                                     //---+
```

Referencing in Rust

A reference to a value cannot outlive the owner

Rust

```
let v = vec![1, 2];
let x=&v[0] ;
let v2=v ;
let y = *x +1 // ERROR - x refers to v, but v is dead
```

A value can have one mutable reference or many immutable references

Rust

```
let mut v = vec![1, 2];
let x=&v[0];    //immutable borrow here
Vec::push(&mut v, 3); // ERROR: mutable borrow here
let y = *x +1; // x still alive
```

Threads in Rust

- We can use multi-threading applications in Rust to improve performance.
- This can lead to some issues
 - Race conditions — multiple threads accessing shared data simultaneously
 - Deadlocks — two or more threads waiting for each other to release resources
 - Bugs that are hard to reproduce
- In Rust, we can create a new thread through the `std::thread::spawn` function

Example of thread in Rust

Rust

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("Reached {} from the spawned thread!");
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("Reached {} from the main thread!");
        thread::sleep(Duration::from_millis(1));
    }
}
```

Example of thread in Rust

Reached 1 from the main thread!

Reached 1 from the spawned thread!

Reached 2 from the main thread!

Reached 2 from the spawned thread!

Reached 3 from the main thread!

Reached 3 from the spawned thread!

Reached 4 from the spawned thread!

Reached 4 from the main thread!

How can we ensure that all threads are spawned

- So far only 5 threads are spawned.
- The main thread exits its execution after 5 iterations in the loop
- This cause the `main` function to end prematurely
- We can use `join` to ensure that all threads are spawned
- This is due to the fact that `thread::spawn` returns `JoinHandle<T>`
- We can call `join` on the `JoinHandle` to block the main thread until the thread completes

Ensure to spawn all threads

Rust

```
fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("Reached {} from the spawned thread!");
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("Reached {} from the main thread!");
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

Data Races and Race Conditions in Rust

Data Race

A data race occurs on a value in memory if

- two or more threads are concurrently accessing memory,
- one or more of them is a write, and
- one or more of them is unsynchronised.

Data races are prevented by the ownership system/borrow checker, since we are unable to alias a mutable reference.

However Rust does not prevent general race conditions

- Since, our hardware, OS and other programs might be **Racy**
- Still, a race condition cannot violate memory safety in a Rust program on its own
- Only in conjunction with some other unsafe code can a race condition actually violate memory safety

Race condition in Rust

Rust

```
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

let data = vec![1, 2, 3, 4];
let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

thread::spawn(move || {
    other_idx.fetch_add(10, Ordering::SeqCst);
}); // we can mutate idx since its atomic it cannot cause a Data Race.

if idx.load(Ordering::SeqCst) < data.len() {
    unsafe { // Incorrectly loading the idx after we did the bounds check.
        // This is an unsafe race condition
        println!("{}", data.get_unchecked(idx.load(Ordering::SeqCst)));
    }
}
```

Message Passing in Rust

- The main idea comes from Go, where it says “Do not communicate by sharing memory; instead, share memory by communicating”
- Channels are a way to communicate between threads in Rust
- Rust provides synchronous channels in the standard library through `std::sync::mpsc` module
- Rust uses `transmitter(tx)` and `receiver(rx)` for channel communication to send and receive over channel respectively.
- Channel can have multiple sending ends producing values but only one receiving end consuming values

Message passing to transfer data between Threads

- First, we create a new channel using the `mpsc::channel` function (`mpsc` stands for multiple producer, single consumer).
- The `mpsc::channel` function returns a tuple, the first element of which is the sending end — the transmitter — and the second element of which is the receiving end — the receiver
- Using `move` moves ownership of `tx` to new thread
- Thread must own transmitter to send messages on channel
- The `send` method returns a `Result<T, E>` type and `unwrap` to panic in case of an error (send has nowhere to send).
- The `recv` method is used to receive messages from the channel and will block the current thread until a message is available.

Example of Message Passing in Rust

Rust

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        let val = String::from("Sending data");
        tx.send(val).unwrap();
    });
    let received = rx.recv().unwrap();

    println!("Got: {}", received);
}
```

Channels and Ownership Transference

- Ownership rules play a vital role in message sending because they help you write safe, concurrent code
- Let's consider, for example, to try to print `val` after we sent it down the channel via `tx.send`
- This results in `error`, since once the value has been sent to another thread, that thread (i.e., function `recv`) takes the ownership
- This means we cannot use `val` in the original thread anymore

Creating Multiple Producers by Cloning

- Before creating the first spawned thread, we call clone on the transmitter.
- Gives us a new transmitter we can pass to the first spawned thread.
- We pass the original transmitter to a second spawned thread which gives us two threads, each sending different messages to the one receiver.

Creating Multiple Producers by Cloning

Rust

```
let tx1 = tx.clone();
thread::spawn(move || {
    let vals = vec![ ... ];
    for val in vals {
        tx1.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});
thread::spawn(move || {
    let vals = vec![ ... ];
    for val in vals {
        tx.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});
for received in rx {
    println!("Got: {}", received);
}
```

Mutex in Rust

- A `Mutex<T>` is a mutual exclusion primitive useful for protecting shared data.
- It is a type that provides interior mutability, meaning that it allows you to mutate data even when the `Mutex<T>` itself is immutable.
- A `Mutex<T>` has two main methods: `lock` and `unlock`.
- The `lock` method locks the mutex, preventing other threads from accessing the data until the mutex is unlocked.
- The `unlock` method unlocks the mutex, allowing other threads to access the data.

Summary

- Rust ownership system is a practical implementation of ideas from linear types, RAII/RBMM, and aliasing
- Ownership system guarantees memory safety and thread safety at compile time
- Message passing and concurrency supported in standard library
- Session types supported through external library
- We can use threads and message passing to build concurrent applications in Rust
- In the next lecture, we will look more in detail in the type system elements to support concurrency in Rust

Typed Concurrency in Rust

Riccardo Sieve

November 12, 2025

University of Oslo

Recap on Rust

- Rust is a systems programming language focused on safety and performance.
- It achieves memory safety without a garbage collector.
- It uses a unique ownership model to manage memory.
- Rust has a strong type system that helps catch errors at compile time.
- Rust has built-in support for concurrency.
 - Threads
 - Message passing
- Ownership, borrowing, and lifetimes are key concepts for typed concurrency in Rust.
- Rust's type system includes the `Send` and `Sync` traits to ensure thread safety.
- Smart pointers like `Arc<T>` are used for shared ownership and thread-safe reference counting.
- Asynchronous programming is supported through the `Future` trait and the `async/await` syntax.

Mutability and Immutability in Rust

- By default, variables in Rust are immutable.
- This means that once a value is assigned to a variable, it cannot be changed.
- To make a variable mutable, you need to use the `mut` keyword.
- This helps prevent accidental changes to values.

Ownership in Rust

- Rust uses ownership to manage memory.
- This helps prevent data races at compile time.
- In other programming languages, memory is managed through garbage collection.
- In Rust, memory is managed through a set of ownership rules that enforce rules at compile time.
 - Each value in Rust has an owner.
 - A value can only have one owner at a time.
 - When the owner goes out of scope, the value is dropped.

Borrowing in Rust

- Borrowing allows you to use a value without taking ownership of it.
- You can borrow a value by creating a reference to it.
- We can also use references to borrow values.
- References can be either mutable or immutable.
- We can have either one mutable reference or multiple immutable references to a value at a time.

Reference Counted in Rust

- `Rc<T>` is a reference-counted smart pointer that enables multiple ownership of the same data.
- It keeps track of the number of references to the data, and when the reference count reaches zero, the data is automatically deallocated.
- `Rc<T>` is not thread-safe, meaning that it cannot be shared between threads.
- For thread-safe reference counting, Rust provides the `Arc<T>` type, which stands for atomic reference counting.

Atomic Reference Counted in Rust

- `Arc<T>` is a thread-safe reference-counted smart pointer that enables multiple ownership of the same data across threads.
- It uses atomic operations to manage the reference count, ensuring that it is safe to share between threads.
- Like `Rc<T>`, when the reference count reaches zero, the data is automatically deallocated.
- `Arc<T>` is often used in conjunction with `Mutex<T>` to provide thread-safe shared mutable state.
- Note that, in terms of performance, `Arc<T>` is generally slower than `Rc<T>` due to the overhead of atomic operations.

Send and Sync Traits

- The `Send` and `Sync` traits are embedded in the language rather than the standard library.
- The `Send` marker trait indicates that ownership of values of the type implementing `Send` can be transferred between threads.
- The `Sync` marker trait indicates that it is safe to reference values of the type from multiple threads.
- Most primitive types in Rust are both `Send` and `Sync`.
- Types that are not `Send` or `Sync` include raw pointers, `Rc<T>`, and `RefCell<T>`.

Send and Sync Traits (Cont.)

- Types that are composed entirely of `Send` types are automatically `Send`.
- The same applies to `Sync`.
- This means that most types in Rust are `Send` and `Sync`.
- Manually implementing these traits involves implementing unsafe Rust code.

Example of Programming with Send and Sync Traits in Rust

Rust

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0)); let mut handles = vec![];
    for _ in 0..10 {
        let counter_clone = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter_clone.lock().unwrap();
            *num += 1; // Increment the counter
        });
        handles.push(handle);
    }
    for handle in handles {
        handle.join().unwrap();
    }
    println!("Final counter value: {}", *counter.lock().unwrap());
}
```

Send and Sync Traits (Cont.)

- `Mutex` is a synchronization primitive that is `Send` and `Sync` as long as the data it protects is also `Send` and `Sync`.
- `Arc` is an atomic reference-counted smart pointer that is `Send` and `Sync` as long as the data it points to is also `Send` and `Sync`.
- The closure passed to `thread::spawn` must be `Send` because it will be executed in a different thread.
- `Arc::clone` creates a new reference to the same `Mutex`-protected data, allowing multiple threads to share ownership of the data safely.
- The `lock` method on `Mutex` returns a guard that provides mutable access to the data, ensuring that only one thread can access the data at a time.
- While no `unlock` method is needed, the lock is automatically released when the guard goes out of scope.

Scope in std::thread

- Rust's standard library provides the `std::thread` module for creating and managing threads.
- The function `scope` creates a scope for spawning threads that can borrow data from the parent thread.
- The function passed to `scope` will be provided a `Scope` through which it can spawn threads.
- Threads spawned within the scope can borrow non-static data.
- This is due to the fact that scope guarantees all threads to be joined at the end of the scope.
- If some threads haven't been manually joined, they will be joined automatically when the function returns.

Example of Scopes in Rust (using std::thread::scope)

Rust

```
use std::thread;
fn main() {
    let mut a = vec![1, 2, 3];
    let mut x = 0;

    thread::scope(|s| {
        s.spawn(|| {
            a.push(4);
        });
        s.spawn(|| {
            x += 1;
        });
    }); // All threads are joined here

    println!("a: {:?}", a, x: x);
    assert_eq!(a, vec![1, 2, 3, 4]);
    assert_eq!(x, 1);
}
```

Futures and Async/Await in Rust

- A future is a value that may not be available yet, but will be at some point in the future.
- Rust provides a `Future` trait as a building block so that different async operations can be implemented with different data structures but with a common interface.
- You can apply the `async` keyword to blocks and functions to specify that they can be interrupted and resumed.
- Within an async block or async function, you can use the `await` keyword to await a future.
- The process of checking with a future to see if its value is available yet is called `polling`.

The tokio crate

- `tokio` is a crate that provides an asynchronous runtime for Rust.
- A crate^a is a package of Rust code. It can be a library or a binary.
- It provides a way to write asynchronous code that is efficient and scalable.
- `tokio` provides a number of features, including:
 - An event loop that can handle a large number of concurrent tasks.
 - A set of asynchronous I/O primitives, such as TCP and UDP sockets, file I/O, and timers.
 - A task scheduler that can manage the execution of asynchronous tasks.
- `tokio` is widely used in the Rust community for building high-performance network applications, such as web servers and databases.
- The `tpml` (short for *The Rust Programming Language*) crate from Rust also uses `tokio` under the hood.

^a<https://doc.rust-lang.org/book/ch07-01-packages-and-crates.html>

Example of an Asynchronous program in Rust (using Tokio)

Rust

```
#[tokio::main]
async fn main() {
    println!("Hello from the main async function!");
    say_world().await;
    // Spawn a new asynchronous task
    tokio::spawn(async {
        println!("Hello from a spawned task!");
    });
    // Introduce a delay using tokio::time::sleep
    tokio::time::sleep(tokio::time::Duration::from_secs(1)).await;
    println!("Main function finished.");
}

async fn say_world() {
    println!("World from another async function!");
}
```

The tokio crate (Cont.)

- `#[tokio::main]` is the macro that transforms the `async main` function into a regular main function that sets up a Tokio runtime and starts executing the asynchronous code within it.
- `say_world().await` calls the asynchronous function `say_world` and waits for it to complete before proceeding.
- `tokio::spawn(async ...)` spawns a new asynchronous task that runs concurrently with the main function.
- `tokio::time::sleep(...).await` introduces a delay of 1 second in the execution of the main function.

Writing the Dining Philosophers Problem in Rust

- Let's now write the Dining Philosophers Problem in Rust using the `tokio` crate for asynchronous programming.
- We will use `Arc<Mutex<T>>` to manage shared state and ensure thread safety.
- For thoughts and eating, we will use asynchronous functions to simulate the actions of the philosophers.
- While the chopsticks will be represented as shared resources that philosophers will need to acquire before eating.

Defining the Philosopher Struct in Rust

Rust

```
use std::sync::Arc;
use tokio::sync::{Mutex, mpsc};
use tokio::time;

struct Chopstick;

#[derive(Clone)]
struct Philosopher {
    name: String,
    // left_chopstick: ...
    // right_chopstick: ...
    // thoughts: ...
    left_chopstick: Arc<Mutex<Chopstick>>,
    right_chopstick: Arc<Mutex<Chopstick>>,
    thoughts: mpsc::Sender<String>,
}
```

Implementing thinking a in Philosopher Struct in Rust

Rust

```
impl Philosopher {
    async fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name))
            .await
            .unwrap();
    }

    async fn eat(&self) {
        ...
    }
}
```

Implementing eating in Philosopher Struct in Rust

Rust

```
impl Philosopher {
    async fn think(&self) {
        ...
    }

    async fn eat(&self) {
        println!("{} is hungry...", &self.name);
        // Keep trying until we have both chopsticks
        let _left = self.left_chopstick.lock().await;
        println!("{} picked up left chopstick.", &self.name);
        let _right = self.right_chopstick.lock().await;
        println!("{} picked up right chopstick.", &self.name);

        println!("{} is eating...", &self.name);
        time::sleep(time::Duration::from_millis(5)).await;
        println!("{} is done eating.", &self.name);
    }
}
```

Structuring the main Function

Rust

```
static PHILOSOPHERS: &[&str] = &["Socrates", "Hypatia",
    "Aristoteles", "Plato", "Hegel", "Kant"];

#[tokio::main]
async fn main() {
    // Create chopsticks
    let chopsticks: Vec<_> = (0..PHILOSOPHERS.len())
        .map(|_| Arc::new(Mutex::new(Chopstick)))
        .collect();

    // Create philosophers and their thought receivers
    ...
}
```

Creating the philosopher

Rust

```
#[tokio::main]
async fn main() {
    ...
    // Create philosophers and their thought receivers
    let mut receivers = Vec::new();
    let philosophers: Vec<_> = PHILOSOPHERS
        .iter().enumerate().map(|(i, &name)| {
            let left = chopsticks[i].clone();
            let right = chopsticks[(i + 1) % chopsticks.len()].clone();
            let (thoughts_tx, thoughts_rx) = mpsc::channel(PHILOSOPHERS.len());
            receivers.push((name.to_string(), thoughts_rx));
            Philosopher {
                name: name.to_string(),
                left_chopstick: left,
                right_chopstick: right,
                thoughts: thoughts_tx,
            }
        }).collect();
    ...
}
```

Making the philosophers think and eat

Rust

```
#[tokio::main]
async fn main() {
    ...

    // Make them think and eat
    let mut handles = Vec::new();
    for philosopher in philosophers {
        let handle = tokio::spawn(async move {
            philosopher.think().await;
            philosopher.eat().await;
        });
        handles.push(handle);
    }

    ...
}
```

Output the thoughts and wait for all tasks to finish

Rust

```
#[tokio::main]
async fn main() {
    ...
    // Output their thoughts
    for (name, mut thoughts_rx) in receivers {
        let handle = tokio::spawn(async move {
            while let Some(thought) = thoughts_rx.recv().await {
                println!("{} thinks: {}", name, thought);
            }
        });
        handles.push(handle);
    }

    // Wait for all tasks to finish
    for handle in handles {
        let _ = handle.await;
    }
}
```

Summary

- Rust provides a powerful type system that helps ensure memory safety and thread safety.
- `Arc<T>` and `Mutex<T>` are essential for managing shared state in a thread-safe manner.
- The `Send` and `Sync` traits are key to ensuring that types can be safely shared between threads.
- Asynchronous programming in Rust is facilitated by the `Future` trait and the `async/await` syntax.
- The `tokio` crate provides a robust runtime for building asynchronous applications in Rust.
- In the next lecture, we will do a recap of the entire course.